UNIVERSITAT JAUME I
Departament de Llenguatges i Sistemes Informàtics

Towards a Multiresolution Model on GPU

Ph. D. dissertation
Oscar Enrique Ripollés Mateu
Advisor: Dr. Miguel Chover Sellés

Castellón, September 2009

UNIVERSITAT JAUME·I

Departament de Llenguatges i Sistemes Informàtics

Hacia un modelo multirresolución en la tarjeta gráfica

Tesis doctoral
Oscar Enrique Ripollés Mateu
Director: Dr. Miguel Chover Sellés

Castellón, Septiembre 2009

El objetivo principal de esta tesis es presentar un conjunto de técnicas que se han desarrollado para mejorar la visualización en tiempo real de mallas poligonales. Se describen varias soluciones a la gestión del nivel de detalle, haciendo énfasis en la explotación del hardware gráfico actual. En consecuencia, las contribuciones optimizan el coste de almacenamiento, reducen el tiempo de extracción y minimizan los datos transferidos a través del BUS. Por último, se presenta una solución que es capaz de gestionar el nivel de detalle completamente en la tarjeta gráfica, reduciendo al máximo el tráfico de datos y ofreciendo tanto resoluciones uniformes como variables. Además, este último método ofrece una fácil implementación en aplicaciones 3D que permiten la aplicación de *shaders* a mallas poligonales.

Objeto y objetivos de la investigación

Uno de los principales problemas de las aplicaciones gráficas interactivas, como los juegos por ordenador o los entornos de realidad virtual, es la complejidad geométrica de las escenas que representan. La necesidad de escenas muy realistas supone utilizar modelos formados por una gran cantidad de polígonos, lo que presenta dificultades para mantener un alto número de imágenes por segundo.

Una de las posibles soluciones a este problema es el uso de técnicas multirresolución, que representan un objeto a través de un conjunto de aproximaciones de diferente nivel de detalle y permite recuperar cualquiera de ellas bajo demanda. De esta manera, las técnicas de modelado multirresolución permiten adecuar el nivel de representación a las necesidades de visualización, teniendo en cuenta las características de la aplicación y del hardware que le da soporte.

Los primeros modelos multirresolución sólo almacenaban un conjunto pequeño de niveles de detalle (generalmente entre 5 y 10) y se les denominó modelos multirresolución discretos. Sin embargo, este conjunto de soluciones presenta problemas en la transición entre dos niveles, donde se produce un efecto perceptible de salto en la imagen. Por este motivo aparecieron los modelos multirresolución continuos, capaces de albergar un amplio rango de niveles de detalle, que habitualmente se diferencian en un vértice, una arista o un triángulo, y que permiten transiciones suaves o interpoladas entre niveles de detalle contiguos. Los modelos multirresolución continuos son capaces de resolver los problemas de la visualización interactiva, la transmisión progresiva, la compresión geométrica y la resolución variable.

El hardware gráfico ha mejorado considerablemente en los últimos años; su rendimiento se multiplica cada seis meses mientras que los microprocesadores solamente sufren un incremento del 40 % cada año. De esta forma, la posibilidad de aprovechar al máximo su poder computacional, su tremendo ancho de banda, la programación paralela y también la posibilidad de aligerar la carga de la CPU, han aumentado el interés en utilizar el hardware gráfico como una solución computacional. Así, es posible explotar el hardware gráfico no solo para tareas propias de los gráficos por ordenador sino también para la programación de propósito general. Además, el desarrollo del bus de datos PCIExpress ha mejorado notablemente el rendimiento del bus AGP, facilitando el tráfico de información entre la CPU y la GPU en una aplicación en tiempo real.

Así, siguiendo con la evolución de los modelos multirresolución, las últimas aportaciones se enfocan hacia el uso eficiente del hardware gráfico, dado que su increíble evolución augura grandes avances en el campo de los gráficos por ordenador. Existen modelos que se han desarrollado con la intención de trabajar completamente en la GPU. Estos modelos, pese a sus buenos resultados, presentan ciertas limitaciones y complejidades que los hacen difíciles de integrar en un motor de juegos o librerías gráficas para su posterior utilización en aplicaciones comerciales.

El trabajo presentado en esta tesis se ha orientado hacia el desarrollo de técnicas de nivel de detalle adaptadas a las necesidades de las aplicaciones gráficas con distintas configuraciones hardware. En este sentido, el objetivo final es ofrecer soluciones que:

- obtengan aproximaciones que sean visualmente satisfactorias al ojo humano
- optimicen las necesidades de memoria de las estructuras de datos, tanto en CPU como en GPU
- reduzcan el coste temporal de los algoritmos de extracción
- ofrezcan una fácil implementación en motores de juegos y aplicaciones finales

Planteamiento y metodología utilizados

Esta tesis tiene como objetivo fundamental el desarrollo de nuevas técnicas de modelado multirresolución que ofrezcan una solución definitiva para el manejo del nivel de detalle en motores de juegos y aplicaciones interactivas.

Para ello, se propone realizar las tareas siguientes:

Estado del arte

Como paso previo al trabajo de la tesis, es necesario desarrollar un estudio en profundidad de técnicas de modelado multirresolución. Se propone también el análisis de las últimas aportaciones en la búsqueda de tiras y los métodos de simplificación. Además, la problemática del modelado geométrico, la programación de las tarjetas gráficas y las nuevas técnicas de explotación de las capacidades del hardware también se estudiarán debido a su influencia sobre el trabajo a desarrollar.

Modelado multirresolución

Como ya se ha comentado previamente, el objetivo fundamental de esta tesis es la obtención de un modelo de nivel de detalle que cumpla con los requisitos mencionados anteriormente. Así, se pretende desarrollar un modelo que esté totalmente integrado en el hardware y que tenga en cuenta todas las características de los objetos de los motores de juegos, como pueden ser la iluminación o el uso de texturas.

Integración en motores de juegos

La necesidad de obtener unos resultados aplicables obliga a estudiar las posibilidades de integración de los modelos desarrollados en varios motores de juegos, para poder comprobar su calidad en aplicaciones reales y mejorar los aspectos que sean necesarios.

Evaluación y comparación del modelo desarrollado

La verificación y prueba de los resultados obtenidos se realizará sobre el software de soporte a los proyectos de investigación en los que se enmarca el trabajo de esta tesis. El rendimiento del modelo se podrá medir en las aplicaciones desarrolladas en dichos proyectos y con datos de los mismos. Por supuesto, se pretende demostrar la calidad de la solución propuesta frente a otras soluciones continuas y discretas, tanto en pruebas de tiempo o de coste espacial como en aplicaciones reales.

Aportaciones originales

Teniendo estos objetivos en mente, se han propuesto diferentes soluciones adecuadas para motores de juegos y librerías gráficas que habitualmente recurren a modelos discretos a la hora de elegir la técnica multirresolución más adecuada.

Inicialmente, el Capítulo 2 presenta el estado del arte en modelado por nivel de detalle, así como un breve estudio de las técnicas de simplificación y búsqueda de tiras. Dado que la tesis presentada está enfocada hacia el modelado multirresolución, también se estudian las técnicas orientadas hacia la elección del nivel de detalle más adecuado a las características concretas de la aplicación, de manera que se adapte la cantidad de geometría visualizada a la capacidad del hardware o del software que se utilice.

Para mejorar soluciones anteriores, el Capítulo 3 describe *Speed Strips*, un nuevo modelo multirresolución que realiza todas las operaciones necesarias para modificar el nivel de detalle y eliminar triángulos degenerados al mismo tiempo, reduciendo el coste de extracción de soluciones previas. Además, tras un estudio de las características del bus PCI Express, se ha orientado el proceso de actualización de datos en la tarjeta gráfica para maximizar la transmisión de datos a través del BUS.

Continuando con la explotación del hardware gráfico, el Capítulo 4 introduce una solución diferente: *Masking Strips*. El principal objetivo de este método es la codificación de las operaciones para modificar el nivel de detalle en máscaras de bits, dado que con el último hardware es posible ejecutar operaciones a nivel de bit en la propia tarjeta gráfica. Además, el uso de máscaras permite eliminar completamente todos los triángulos degenerados innecesarios,

mejorando soluciones anteriores que solamente eran capaces de eliminar ciertos patrones de triángulos degenerados. *Masking Strips* presenta también una interesante solución para la transmisión progresiva de modelos 3D, ya que es posible enviar inicialmente una aproximación con poco detalle e ir refinándola mediante sucesivos paquetes de datos.

Por último, el Capítulo 5 describe *Interactive Meshes*, cuyo objetivo es ofrecer un modelo multirresolución que funcione completamente en la tarjeta gráfica. Este método se basa en el uso de triángulos como primitiva de dibujado y modifica la información de los vértices directamente en la tarjeta gráfica mediante la programación de *shaders*. Además, esta solución es capaz de mostrar tanto resoluciones uniformes como resoluciones variables, pudiendo añadir más detalle a aquellas zonas que lo necesiten como, por ejemplo, las siluetas. Por último, este modelo permite mejorar la calidad visual final mediante procesos como el *geo-morphing*.

Conclusiones obtenidas y futuras líneas de investigación

Del estudio inicial planteado en el capítulo del estado del arte podemos concluir que, aunque existe una amplia investigación en este campo, todavía existe la necesidad de desarrollar técnicas eficientes. La mayoría de las soluciones basadas en la CPU suponen el uso de complejos procesos que no las hacen adecuadas para aplicaciones gráficas, mientras que las técnicas más recientes, basadas en la GPU, están generalmente enfocadas hacia la teselación de una malla inicial.

De los experimentos realizados durante el desarrollo del modelo *Speed Strips* podemos concluir que es mejor realizar varias operaciones que suponen poco tráfico de datos en lugar de actualizar toda la información sobre las tiras en una única vez. Además, el estudio de primitivas desarrollado en este capítulo con el hardware disponible demostró que las tiras obtenidas con el algoritmo *Stripe* ofrecen el mejor rendimiento.

El modelo *Masking Strips* ofrece un tratamiento más avanzado de los triángulos degenerados, lo que permite reducir la cantidad de índices procesados en las aproximaciones más burdas en un 40 %. Por otra parte, el coste espacial se reduce considerablemente si comparamos esta técnica con otros modelos desarrollados previamente. Por último, es importante mencionar que este modelo se ha integrado en el motor gráfico Ogre, ofreciendo interesantes resultados al compararlo con la solución discreta que este motor implementa.

Interactive Meshes es la solución más interesante de las introducidas en esta tesis, ya que ofrece un modelo multirresolución simple y eficiente que además está completamente integrado en la GPU utilizando *shaders*. Este modelo reduce el coste espacial de las estructuras de datos y el coste temporal de los algoritmos de extracción respecto a las soluciones presentadas con anterioridad.

Además, es importante comentar que una de las ventajas más importantes de una solución como esta es la posibilidad de incorporar un modelo multirresolución en cualquier aplicación que soporte el Shader Model 4.0, ya que dos *shaders* y muy pocas líneas de código son suficientes para poder recuperar las estructuras de datos y realizar la selección y visualización de los distintos niveles de detalle.

Trabajo futuro

En esta tesis hemos presentado diferentes técnicas multirresolución que son adecuadas para distintas configuraciones de software y hardware. Sin embargo, existen varios aspectos de estas técnicas que se pueden mejorar y también varios campos de investigación que se pueden beneficiar de las soluciones propuestas.

En este sentido, sería interesante aplicar las técnicas desarrolladas a la representación 3D de otros elementos como árboles o sistemas de partículas, que pueden llegar a ser útiles a la hora de representar fenómenos como el fuego o la lluvia.

Por otra parte, hay que tener en cuenta que para incluir un modelo multirresolución en una aplicación gráfica no es suficiente con ofrecer los algoritmos de manejo. Así, también sería interesante considerar la necesidad de desarrollar un conjunto de técnicas de gestión de los múltiples modelos 3D que puedan estar incluidos en una escena. De esta manera, en este tipo de situaciones creemos que es necesario mantener un equilibrio entre la cantidad de geometría a visualizar y el tiempo de proceso.

Tras analizar el desarrollo del hardware actual, se considera que la potencia ofrecida por el hardware gráfico presenta nuevas posibilidades para mejorar los modelos dependientes de la vista, como es el caso del último modelo presentado en esta tesis (*Interactive Meshes*). Asimismo, creemos que nuestras propuestas se podrían aplicar para modelos masivos, cuyas necesidades de memoria requieren rutinas específicas para su gestión tanto en la CPU como en la GPU. Como consecuencia de ello, las técnicas propuestas se pueden combinar para ofrecer una nueva solución al problema de la visualización de modelos masivos.

Desde una perspectiva diferente, es importante mencionar que uno de los principales objetivos del trabajo presentado es el desarrollo de un modelo multirresolución que funcione completamente en la tarjeta gráfica. Aunque las soluciones propuestas han demostrado ser satisfactorias, es interesante plantearse la traducción de estas aportaciones a CUDA, esperando un aumento de rendimiento. CUDA (Compute Unified Device Architecture) es una tecnología reciente elaborada por nVidia con el objetivo de aprovechar la enorme capacidad de procesamiento de las actuales tarjetas gráficas. De esta manera, en lugar de emplear un gran número de ordenadores, es posible recurrir a los procesadores gráficos para hacer cálculos matemáticos o resolver problemas con una alta carga de trabajo computacional.

Siguiendo con esta línea de trabajo sobre la GPU, cabe mencionar que la

inminente aparición de DirectX 11 implicará nuevos avances en los gráficos. Entre las nuevas etapas de la *pipeline* gráfica destaca la unidad de teselación, que será capaz de producir teselaciones semiregulares. Esta característica puede ser utilizada directamente como técnica multirresolución y visualizar de una manera sencilla aproximaciones dependientes de la vista. Por lo tanto, teniendo en cuenta las características de la futura versión de DirectX, creemos que esta unidad será un elemento clave en la próxima generación de modelos multirresolución.

The excitement is the feeling that we experiment immediately after having a great idea and right before we realize its drawbacks

Anonymous

x

Preface

Abstract

The main aim of this dissertation is to present a set of techniques which have been developed to improve the real-time visualization of polygonal meshes. Several solutions to the management of the level of detail are described, emphasizing on the exploitation of current graphics hardware. As a consequence, the contributions optimize the storing cost, reduce the extraction time and minimize the data transferred through the BUS. Finally, a solution is presented which is capable of managing the level-of-detail completely on GPU, reducing to the maximum the traffic of data between the CPU and the GPU, and offering continuous and view-dependent resolutions. Moreover, this latest approach offers a very easy implementation in applications like FX Composer, which enables the user to easily create, edit and visualize shaders applied to polygonal meshes.

Keywords: multiresolution modeling, level-of-detail, graphics hardware

Funding

This research has been partially supported by the *GameTools* project from the VIth Framework Program from the European Union (2001/SGR/00296), by the project *Mejora y Aplicación de la Tecnología de Juegos en Realidad Virtual y Contenidos Web* from the Comisión Interministerial de Ciencia y Tecnología from the Spanish government (TIC2004-7451-C03-03), by the project *Geometría Inteligente* form Fundació Bancaixa (P1 1B2007-56) and by the project *Contenido Inteligente para Aplicaciones de Realidad Virtual: una Aproximación Basada en Geometría* funded by the Ministerio de Educación y Ciencia from the Spanish government (TIN2007-68066-C04-02).

Acknowledgements

This thesis is the result of many years of work in the field of Computer Graphics which would not have been possible without the help of many people.

First of all, I would like to express my sincere gratitude to Miguel Chover, who has been my supervisor since the beginning of my Ph.D. studies. He provided me with many helpful suggestions and constructive advices during the course of this work.

I have been very lucky to make good friends at the Computer Graphics group. The many discussions we had (most of them not research-related), were often the occasion for new discoveries and always truly agreeable moments.

My experience at Limoges would not have been such a pleasurable one without the presence of all the people working there. I would like to specially thank Dimitri and Benoît, who were incredibly helpful and kind throughout the whole stay.

I would like to express my heartiest thanks to Anna, who helped me to make the most of my life (and still does).

My special appreciation goes to my parents for their support and encouragement to do my best. I would like to finish by thanking my family and friends for their help and for understanding my little availability.

Thank you very much to all of you.

Index

1. Introduction **1**
1.1. Motivation . . . 3
1.2. Contributions . . . 5
1.3. Document organization . . . 6

2. Previous Work **9**
2.1. Introduction . . . 9
2.2. Mesh simplification . . . 11
2.3. Rendering primitive optimization . . . 13
2.3.1. Triangle strips . . . 14
2.3.2. Cache-optimized primitives . . . 15
2.4. Multiresolution modeling . . . 16
2.4.1. Discrete models . . . 16
2.4.2. Continuous models . . . 18
2.4.3. View-dependent models . . . 20
2.4.4. Other techniques . . . 21
2.4.5. Characterization . . . 22
2.5. LOD selection criteria . . . 23
2.6. Conclusions . . . 26

3. Optimizing the Management of Level-of-Detail Models **29**
3.1. Introduction . . . 29
3.2. General Framework . . . 31
3.2.1. Simplification of the original mesh . . . 32
3.2.2. Stripification of the original mesh . . . 34
3.2.3. Construction process . . . 36
3.3. The Speed Strips model . . . 37
3.3.1. Data structures . . . 37
3.3.2. Extraction and visualization algorithms . . . 41
3.4. A memory version . . . 43
3.5. Results . . . 45

3.5.1. Memory cost 45
3.5.2. Rendering cost 45
3.5.3. Rendering primitive 47
3.6. Conclusions 48

4. Rendering Continuous Level-of-Detail Meshes by Masking Strips 51
4.1. Introduction 51
4.2. General Framework 53
4.2.1. Simplification of the original mesh 54
4.2.2. Masking for LOD management 57
4.3. The Masking Strips model 59
4.3.1. Data structures 59
4.3.2. Extraction and visualization algorithms 61
4.3.3. Progressive transmission 61
4.4. Results 64
4.4.1. Memory cost 65
4.4.2. Rendering cost 65
4.5. Conclusions 70

5. Interactive Visualization of Meshes on the GPU 71
5.1. Introduction 71
5.2. Continuous resolution framework 73
5.2.1. Pre-processing the original mesh 74
5.2.2. Data structures 77
5.2.3. Extraction algorithms 79
5.3. View-dependent resolution framework 83
5.3.1. Implementation details 84
5.4. Integration into a real application 87
5.5. Results 89
5.5.1. Storage and memory cost 89
5.5.2. Collapse list size study 90
5.5.3. Primitive study 90
5.5.4. Rendering time 91
5.6. Conclusions 92

6. Conclusions and Future Work 95
6.1. Conclusions 95
6.2. Future work 97
6.3. Publications 98

Bibliography 103

List of Figures

1.1. Animated man model. From left to right: original (138,517 triangles), 75 % (103,889 triangles), 50 % (69,261 triangles) and 25 % (34,629 triangles) simplified versions. 2
1.2. *Popping* artifacts in Far Cry game by Crytek Labs (2004). . . . 3

2.1. Two levels of detail of an ogre model. 10
2.2. Example of edge collapses. From top to bottom: a full-edge collapse and a half-edge collapse. 12
2.3. Stripified version of the phlegmatic dragon model. 14
2.4. Different multiresolution approaches for the Isis model: on the left the original geometry, in the middle a continuous approximation and on the right a view-dependent approximation. . . . 17
2.5. Collapse of a strip. 19
2.6. Tessellation of a sample polygon. 22
2.7. Army of multiresolution ninjas in a game engine. 23

3.1. Color-coded LOD scene with many Speed Strips models inside the Ogre rendering engine. 30
3.2. Pre-process diagram. 31
3.3. Example of simplification. Figure on the left is the original model. Figure in the middle represents the model simplified to 50 %. Figure on the right is the model simplified to 25 %. 32
3.4. Example of stripification of a Toyota model: on the left the output of the Stripe algorithm and on the right the result of the cache-aware nVidia approach. 33
3.5. Degenerate triangles after several edge collapse operations. The red-coloured numbers indicate positions that might be eliminated without altering the geometry. 35
3.6. *Speed Strips* data structures. 38
3.7. Data structure for the example given. 39
3.8. Example of change to LOD 2 (collapse $v_2 \rightarrow v_{10}$). 42

3.9. Frame rate obtained with triangles and strips when rendering the bunny model at different LODs. 48

4.1. Space woman model. From left to right: original (4,130 triangles), 60 % (2,478 triangles) and 20 % (826 triangles) approximations. 52
4.2. Basic framework of Masking Strips. 54
4.3. The edge collapse $v_a \rightarrow v_b$ (true edge) forces the collapses $v_c \rightarrow v_d$ (twin edge) and $v_e \rightarrow v_f$ (fake edge). 55
4.4. Three levels of detail of different 3D models. 56
4.5. Masking example. 57
4.6. Patterns used for constructing the filter masks. 59
4.7. Masking Strips data structures. 60
4.8. Progressive transmission data structures. 63
4.9. Indices rendered for the bunny model. 66
4.10. Comparison of the extraction and visualization times of the phlegmatic dragon model. 66
4.11. Comparison of the triangles rendered for the space woman model at different distances. 67
4.12. Space woman at different distances and levels of detail. On the left we present the models rendered with a discrete solution, while on the right we show the results of the Masking Strips approach. 68
4.13. Scene rendering a crowd of models developed inside the Ogre graphics engine. 68
4.14. Performance obtained in the crowded scenario using our multiresolution model, the discrete solution included in Ogre and disabling any level-of-detail solution. 69

5.1. Approximations of a man model (136,410 triangles). From left to right: original model and simplifications to 50 %, 25 % and 10 % respectively. 72
5.2. Construction of an Interactive Meshes model. 74
5.3. Example of the collapse hierarchy of a sample model. 75
5.4. Example of simplification and ordering of vertices following the collapse order. 76
5.5. Data structures preparation and GPU storage. 78
5.6. Rendering pipeline for the continuous Interactive Meshes approach. 80
5.7. Example of the extraction process of four levels-of-detail. . . . 82
5.8. Bunny model with its right half simplified to 80 %. 84
5.9. Rendering pipeline for the view-dependent Interactive Meshes approach. 85

5.10. Construction and rendering pipeline for the integration of our proposal. 88

List of Tables

2.1. Characterization of GPU-based multiresolution models. 24

3.1. Models used in the experiments, with their storage cost (in bits/vertex.). 46
3.2. Average rendering time (extraction+visualization) (in ms.). . . 46
3.3. Average extraction time of hardware models (in ms.). 47
3.4. Average data traffic (in MB.). 47

4.1. Detailed information of the models used in the experiments. . . 64
4.2. Storage cost study (in bits/vertex). 65
4.3. Storage cost study of the progressive solutions (in bits/vertex). 65

5.1. Details of the models and storage cost study (in bits/vertex). . 90
5.2. Collapse list size information. 90
5.3. Performance comparison (in fps) among different primitives and models at two levels of detail. 91
5.4. Comparison of average rendering time (extraction+visualization) (in ms.). 92
5.5. Comparison of average extraction time (in ms.). 92

List of Algorithms

1. Pseudocode of the Speed Strips algorithms. 43
2. Pseudocode of the memory version of Speed Strips. 44
3. Pseudocode of the memory version with hardware support of Speed Strips. 44
4. Pseudocode of the Masking Strips algorithms. 62
5. Pseudocode of the extraction shader of the continuous version of Interactive Meshes 81
6. Pseudocode of the extraction shader of the view-dependent version of Interactive Meshes. 86

CHAPTER 1

Introduction

Nowadays, applications such as computer games, virtual reality environments or scientific simulations are increasing the detail of their scenarios with the aim of offering more realism. This objective usually involves dealing with larger scenes containing lots of objects which are more and more complex and generally include submeshes, materials, textures, normal maps or skeletal animations.

The need for very detailed 3D scenarios is growing faster than the capabilities offered by graphics hardware. Despite the constant improvements in performance and capabilities of GPUs, it is still difficult to render such complex datasets and scenes as vertex throughput and memory bandwidth become considerable bottlenecks when dealing with them. As a result, these environments cannot be interactively rendered by brute force methods, presenting a set of problems for visualization, storage and compression.

It is possible to find plenty of works with the aim of managing these detailed environments, which has led to the following specific research areas:

- *Mesh simplification.* Simplification methods reduce the geometry of the meshes to obtain a lighter version which preserves the original appearance as possible [1]. There are many simplification methods available which apply different simplification operations, like vertex clustering, vertex removal, edge collapse, etc. Moreover, depending on the metric they apply, they can be classified into geometry driven and viewpoint driven algorithms.

- *Level-of-Detail or Multiresolution modeling.* These techniques scale the detail of the objects according to their importance within the scene [2].

Figure 1.1: Animated man model. From left to right: original (138,517 triangles), 75 % (103,889 triangles), 50 % (69,261 triangles) and 25 % (34,629 triangles) simplified versions.

Multiresolution models allow us to reduce the amount of geometry to process, transfer and visualize, which results in an improvement in performance and offers a perfect framework to adapt the complexity of the 3D models to the characteristics of the underlying hardware or to the limitations of the application.

- *Mesh optimization.* These techniques organize the geometry data in order to send it to the graphics hardware in the most appropriate way. This can be done by means of rendering primitives with implicit connectivity or by re-organizing the polygons of the mesh to exploit the vertex cache [3]. Maintaining the data on the GPU allows for exploiting hardware while avoiding bus traffic.

- *Mesh compression.* Research in this field has produced a wealth of works. In a general classification, they could be grouped into progressive compression and single-rate techniques, depending on whether the model is decompressed during the transmission or once the model has been completely received [4, 5, 6].

Among the solutions presented above, one of the most widely used is level-of-detail (LOD) modeling. A level-of-detail or multiresolution model is a compact description of multiple representations of a single object [7] that must be capable of extracting the appropriate representation in different contexts. Figure 1.1 presents an animated man model at four different levels of detail, each of them reducing the geometry of the previous approximation in 25 %.

Multiresolution modeling has been successfully applied to solve problems in many areas [8] and there is an important body of literature on the subject [2]. A comprehensive description of multiresolution models can be found in [9].

The first multiresolution models that were developed were based on a relatively small number of approximations (usually between 5 and 10) [9], and were known as discrete multiresolution models. These discrete models suffer

Figure 1.2: *Popping* artifacts in Far Cry game by Crytek Labs (2004).

from *popping* artifacts that appear when switching between the different levels of detail, causing noticeable and visually disturbing effects. Figure 1.2 presents two images captured from the Far Cry videogame by Crytek Labs. These snapshots show two instants of the game which are very close in time, as the user has made a very small change in its position. Nevertheless, the position change has forced the rocks located in the middle of the scenario to swap to a highest level of detail, causing a clearly noticeable *popping* effect.

Continuous multiresolution models appeared later with the aim of improving discrete models, offering a wide range of different approximations to represent the original object. These models are capable of solving the problems of interactive visualization, progressive transmission and geometric compression.

A further improvement on continuous multiresolution models are those that present view-dependent capabilities, which enable an object to include different resolutions in different areas at the same time. These models, although they offer better granularity, present important time limitations as their extraction process is usually more complex and they need to obtain some extra information of the scene conditions.

Nowadays, multiresolution modeling can be considered as a compulsory feature of libraries and game engines. In this sense, graphics libraries like OpenInventor or OSG, and game engines such as Torque or Ogre, introduce multiresolution models to easily alleviate the amount of geometry that must be rendered in a scene, thus resulting in an improvement in performance.

1.1. Motivation

For more than a decade, researchers working on level-of-detail techniques have oriented their efforts towards developing better frameworks. This research field has been exploited for many years, and it is possible to find a wealth of

papers which present very different solutions.

The use of a multiresolution model in a graphics application entails increasing the total time required to render the scene. This is due to the fact that, for obtaining the desired approximation, the algorithms related to the multiresolution scheme must perform an *extraction process*, composed of different tasks which are necessary to update the geometry according to the desired level-of-detail. Usually, the overcost related to the extraction process is amortized by the increase in performance obtained when reducing the geometry to visualize. Thus, reducing the extraction time is key for developing efficient multiresolution schemes.

In this sense, it is important to say that, despite the better features shown by continuous models, traditional solutions usually involve discrete multiresolution models. The reasons behind this decision are quite simple: discrete models are more easily integrated and they also offer an easier and more straightforward level of detail update. Many authors consider that using continuous models is not worth the effort, as in an interactive application the viewer keeps moving all the time and this would involve updating the whole scene continuously, which would lower overall performance. As a consequence, it is easier to discard one model and use another one (which happens with discrete models) and accept the *popping* artifacts.

From a different perspective, as we all know graphics hardware has improved outstandingly over recent years. Performance is doubling every six months [10], in contrast to microprocessors which grow by approximately 40 % every year [11]. Thus, the possibility of taking maximum benefit from its computational power, its tremendous memory bandwidth, the possibility of parallel programming, as well as the alleviation of CPU load, has increased the interest in using GPUs as a computational solution, not only for computer graphics, but also for general-purpose routines. The development of Shader Model 4.0 was a breakthrough in computer graphics as it offers a new range of functionalities, like a *Geometry Shader* that enables the dynamic creation and elimination of geometry and the *Stream Output* that stores the outputted geometry. Furthermore, the development of the PCI Express bus has boosted the performance if compared with previous AGP buses, making the traffic of information between the CPU and the GPU much more efficient.

Many of the works available in the literature were written in the early days of the GPUs (or even in earlier times [12]) when it was advisable to spend some CPU processing time to optimize the GPU rendering process. Nowadays, due to the great scalability of the graphics cards, we must revise all that previous work to provide an updated and practical viewpoint of that situation: overloading the CPU is a delicate task that in most cases will cause it to be a bottleneck for the graphics hardware. As a consequence, the main problem with existing multiresolution models is that, even though these solutions provide interactive rates, it proves very difficult to adapt them to the new GPU architectures due to the complex data structures and algorithms they require. In this sense, it is

possible to find in the literature very few models which have been developed to work on the GPU. Thus, we believe there is still a gap for efficient yet simple multiresolution models that fully exploit the potential of current GPUs.

1.2. Contributions

For addressing these problems and also for offering a complete, simple, and efficient solution for applying continuous LOD modeling in real-time applications, we have developed different multiresolution approaches. The works proposed in this dissertation are oriented towards the development of level-of-detail techniques that exploit graphics hardware, but always from the perspective of the graphics hardware that was available at the moment. Thus, the final aim of the work of this Ph.D. thesis is to develop a multiresolution model which works completely on GPU.

The different solutions presented offer continuous updates, so that updating the level of detail is performed in a smooth manner, avoiding popping artifacts which are often visually noticeable and disturbing. Moreover, it is our aim is to reduce storage cost and to overcome the difficulties of integrating previous solutions in final applications due to their complexity. Our intention is to provide competitive solutions in comparison with discrete models, describing the integration of the proposed solutions into real applications and game engines.

With these objectives in mind, this dissertation presents three multiresolution frameworks. The first contribution of this thesis introduces a new level-of-detail framework that improves on previously presented solutions by offering an extraction process that maximizes the traffic through the BUS and applies all the changes in one single pass. These changes include those operations that are necessary to modify the level of detail and also the steps to eliminate degenerate triangles. This model is based on triangle strips and its description includes a primitive study to address the performance obtained with different rendering primitives.

An important application for multiresolution models is progressive transmission, as the transmission of a complex 3D model might take very long when using low-bandwidth networks. Therefore, level-of-detail modeling can be successful at this task by transmitting an initial coarse model and successively refining it by sending and processing small data packages. Thus, the subsequent proposal is based on the codification of all the operations to modify the triangle throughout the different levels of detail in masks of bits. This approach reduces storage cost and offers a perfect solution for the progressive transmission of the model. Moreover, all unnecessary information that appears while simplifying the original mesh is eliminated, in contrast to previous approaches that were just capable of eliminating certain patterns of degenerate triangles. This multiresolution model has been integrated into the Ogre game engine, and results on the performance obtained are also offered.

Finally, our last approach offers a fully-GPU level-of-detail solution. This triangle-based framework performs LOD calculations by modifying vertices information directly on the GPU using *shaders*. The use of the *Vertex Shader* to perform the vertex update entails the necessity of developing an extraction algorithm which is capable of applying changes on a vertex basis. The proposed solution offered a perfect framework for extending the work and developing an efficient variable resolution model which also considers geo-morphing for maintaining visually satisfactory renders. This GPU model offers a very easy implementation and, thus, could be integrated into applications like nVidia's FX Composer, which is a powerful tool that enables an easy creation and testing of *shaders*. Thus, with this level-of-detail model we are capable of offering the final user a complete solution with just two *shaders* and very little scripting.

1.3. Document organization

The Ph.D. thesis that is presented in this dissertation is organized as follows:

- Chapter 2: Previous Work
  We present the state-of-the-art on multiresolution modeling, stressing those approaches which exploit graphics hardware. Moreover, we also describe current and past techniques on mesh simplification, mesh stripification and also primitive optimization.

- Chapter 3: Optimizing the Management and Rendering of Level-of-Detail Models
  The multiresolution model *Speed Strips* is introduced in this chapter. This technique is presented as an improvement over previous solutions, reducing the extraction cost and enhancing BUS traffic. In addition, this proposal is capable of applying the level-of-detail changes and eliminating unnecessary information in one single pass.

- Chapter 4: Rendering Continuous Level-of-Detail Meshes by Masking Strips
  We describe the characteristics of *Masking Strips*, a different level-of-detail approach which codes the information in efficient bit-masks. This feature enables us to improve on previous solutions by eliminating all unnecessary information. *Masking Strips* is also capable of offering progressive transmission capabilities.

- Chapter 5: Interactive Visualization of Meshes on the GPU
  We propose *Interactive Meshes*, a new framework which uses triangles as the rendering primitive and modifies vertices instead of indices. This solution has been completely integrated on GPU to offer both continuous and view-dependent resolutions. Moreover, its integration into a final application is described, showing that it is possible to manage this level-of-detail solution by means of *shaders* and some scripting.

- Chapter 6: Conclusions and Future Work
  Finally, this chapter summarizes the contributions and concludes the work presented in this dissertation. Moreover, a list of the different publications obtained while developing this thesis is presented, as well as other publications not directly related and a list of research projects that have funded the work.

CHAPTER 2

Previous Work

Multiresolution modeling has been successfully applied to solve problems in many areas, and there is an important amount of bibliography available. This section includes a review of the most important techniques around this field of research and also of the last tendencies in using multiresolution in real-time rendering.

2.1. Introduction

The need for very detailed 3D models is growing faster than the capabilities offered by graphics hardware. Meshes created with traditional modeling solutions are usually composed of a high amount of geometry that needs specific attributes. Applications like computer games or virtual reality environments are including in their scenes more and more complex meshes, which generally include submeshes, materials, textures, normal maps or skeletal animations. As a consequence, they present a set of problems for visualization, storage and compression.

In the previous chapter we introduced several techniques that can be applied to solve these problems, like mesh simplification, level-of-detail modeling, mesh optimization or mesh compression. Among these techniques, the work presented in this Ph.D. dissertation has been concentrated on multiresolution modeling, as they offer acceleration capabilities that can be useful under many circumstances. Multiresolution models allow us to reduce the amount of geometry to process, transfer and visualize, which results in an improvement in performance and offer a perfect framework to adapt the complexity of the 3D models to the characteristics of the underlying hardware or to the limitations

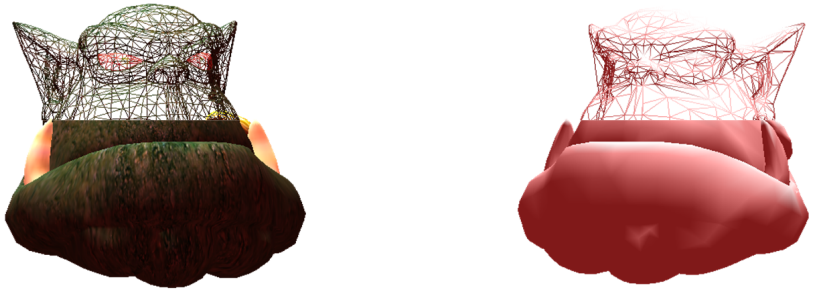


Figure 2.1: Two levels of detail of an ogre model.

of the application while respecting visual fidelity. Figure 4.14 presents two approximations of an ogre model at different levels of detail.

This thesis is aimed at improving the multiresolution models that can be found in the literature. In this chapter we offer a description of the works previously carried out on this issue, making a special effort to describe the GPU-oriented solutions. In this sense, tessellation techniques on the GPU will also be considered, as they offer a solution to the management of the geometry complexity which has received a lot of attention from researchers lately.

Even though the work presented in this dissertation is concentrated on level-of-detail modeling, there are several polygonal techniques that are also fundamental. When working with 3D meshes it is important to consider that the input meshes are usually not available in the format we need or with the most adequate amount of detail. Some techniques are necessary to prepare the original meshes, like:

- tessellation or triangulation, which is often necessary in order to split the input geometry into more adequate primitives, like triangles or quadrilaterals.
- consolidation, which is useful to merge and link the polygonal data, as well as inferring new information like normals.
- optimization, which groups and orders the data so that the performance is improved.
- simplification, which attempts to reduce the polygonal complexity by eliminating unnecessary geometry or imperceptible details.

In the specific case of the work presented in this thesis, we must assure that some requirements are fulfilled so that the multiresolution models can work properly. In this sense, two of these techniques are very important: simplification and optimization.

In this sense, in order to give a complete overview of the solutions we are presenting throughout this Ph.D. thesis, we will present the state-of-the-art on simplification and also on improving rendering primitives. Both techniques are basic for the multiresolution proposal of the thesis. Moreover, we consider also interesting to outline the main ideas on level-of-detail management, understood as the criterions used to decide which level-of-detail is more suitable for a specific mesh depending on characteristics of the whole scene.

This chapter starts by describing general techniques on simplification, covering the different elements that define this set of techniques. After that, we offer a study on primitive optimization, including stripification techniques and cache-optimized primitives. Then, a comprehensive description of GPU-based multiresolution model is offered, including a characterization of the main solutions presented in recent years. Lastly, level-of-detail selection criteria are analyzed as they are key when applying multiresolution techniques to graphics applications. To finish this chapter, some brief conclusions are given to summarize the previous work study.

2.2. Mesh simplification

The objective of the simplification is to reduce the complexity of the input mesh to obtain a coarser approximation which meets some restriction established as a fidelity value or a triangle-count limit [1]. This process usually modifies the geometry and the connectivity of the original mesh; the geometry refers to the vertices of the input mesh, while the connectivity is represented by the edges or faces that connect the vertices. In the specific case of multiresolution modeling, the simplification process is a key aspect as it gathers the information that will be later used to retrieve the different levels of detail.

With this information, the multiresolution model is capable of offering a hierarchy of meshes with varying number of polygons. Depending on the multiresolution framework, it will be possible to obtain several approximations of the original model or a sequence of operations which enables the retrieval of a continuous spectrum of approximations.

To characterize the simplification techniques, we must consider that the simplification process depends on:

- the operator applied, which will reduce the complexity of the mesh by some small amount.
- the error metric selected, used to guide the simplification process and to measure the simplification quality.

By means of these elements, the simplification process will decide which simplification operations to apply and the order in which they should be applied.

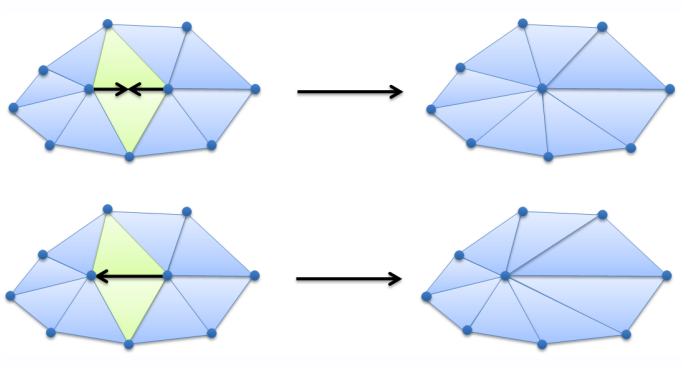


Figure 2.2: Example of edge collapses. From top to bottom: a full-edge collapse and a half-edge collapse.

Simplification operators

There are many different possible operators that a simplification method can use. Among them, we highlight:

- Vertex Removal [13]. The techniques that apply this operator basically use an iterative vertex selection for removal. Once the vertex is removed, all faces that share that vertex are also removed and the resulting hole is triangulated. This type of algorithms is limited to manifold meshes due to its retriangularization schemes.

- Vertex Clustering [14, 15]. The basic idea is to group vertices which are close. Basically, they partition the space in a grid so that all the vertices contained in a single cell are mapped into a single vertex, modifying the faces to reflect the changes. These methods tend to be really fast, but the quality of the resulting simplified mesh is not visually satisfying.

- Edge Collapse [16, 17, 18]. This operator was firstly introduced in [19] and later widely used in multiresolution. At each step, the operator selects a new edge (or vertex pair) and collapses its two vertices, which forces the elimination of the unnecessary geometry. This operator enables us to use geo-morphing, which offers soft transitions among the simplification steps. There are two approaches when applying the edge collapse operator: a full-edge collapse, where the two vertices are collapsed to a newly computed vertex, and the half-edge collapse, where one of the two vertices collapses to the other one. This latter variant avoids adding new vertices, which can be a compulsory restriction for some level-of-detail schemes. Figure 2.2 depicts these two versions applied to the simplification of a sample geometry.

- Vertex-Pair Collapse [20, 21]. The objective of this operator is to collapse two unconnected vertices. This operator is more flexible than the edge collapse and can be useful for closing holes and tunnels.

Simplification error metrics

When simplifying a mesh by using any of the operators presented above, it is necessary to establish some criterion to decide the order in which the elements of the geometry should be collapsed or eliminated.

Most common simplification methods use some technique based on a geometric distance as a quality measure between the original mesh and the one obtained from simplification. These geometric criterions usually take into account the position of vertices, edges and faces. Lately, some methods incorporate surface properties such as colour, normals or texture coordinates [22, 23, 24]. Some authors also use mesh saliency as the metric to guide their simplification [25]. The most common way of incorporating such properties is to add some weighted sum of deviations to the geometric distance. However, these weights are arbitrarily chosen by the user.

On the other hand, one of the objectives of the image-based methods is to manipulate in a natural way the different interactions between the properties of a mesh in only one metric. The image-based simplifications try to carry out a simplification guided by differences between images more than by geometric distances [26, 27, 28, 29, 30]. The goal is to create simplified meshes that appear similar to a human observer. An important improvement of image-based methods is the possibility of obtaining meshes with a maximum simplification in hidden zones.

A reduced number of applications require exact geometric tolerances with regard to the original model. For this type of applications it would be better to consider some simplification method based on a purely geometric measure. Examples of such applications include collision detection and path planning. From a different perspective, the applications that can be benefited by using image-based simplification are those in which the main requirement is visual similarity. Examples of such applications are video games, vehicle simulations, building walk-throughs, etc.

2.3. Rendering primitive optimization

In computer graphics, 3D meshes are usually represented as polygonal meshes, which are considered as a collection of vertices and polygons that connect those vertices. In this sense, we can consider that vertices represent the geometry of the mesh, while the polygons represent its connectivity.

The representation of a polygonal model is usually based on a list of indexed triangles. This representation is based on two sets:

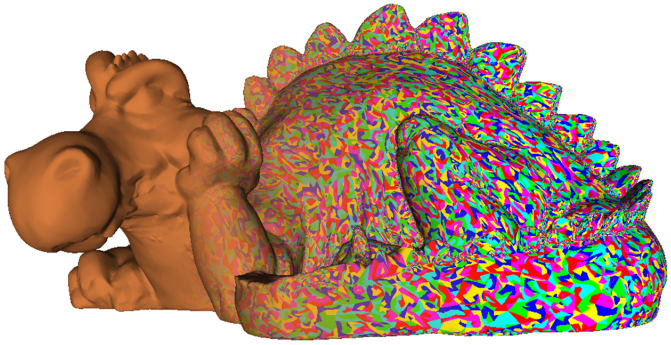


Figure 2.3: Stripified version of the phlegmatic dragon model.

- a list of vertices, describing the coordinates of the vertices.
- a list of indices to these vertices, describing the connectivity of the mesh.

Traditionally, triangles have been the selected primitive for rendering 3D models. Nevertheless, advances have been made in the use of new graphics primitives which minimize the data transfer between the CPU and the GPU. A standard way to increase graphics performance is to send groups of triangles that share vertices to the graphics pipeline. This codification usually offers a wise use of the connectivity information provided by a polygonal mesh. For this purpose, graphics primitives with implicit connectivity, such as triangle strips and triangle fans, have been developed. Figure 2.3 presents an image of the dragon model using triangle strips as the rendering primitive. In this image, each color represents a different triangle strip.

2.3.1. Triangle strips

The codification of a set of n triangles usually requires a list of indices composed of $3 \cdot n$ elements, as it is necessary to indicate the three vertices of each triangle.

A triangle strip is a more compact representation which consists of a series of $n+2$ vertices representing n triangles. In Figure 2.5, the sequence {6,5,4,7,0, 8,1,9,10} corresponds with triangles {6,5,4}, {5,4,7}, {4,7,0}, {7,0,8}, {0,8,1}, {8,1,9}, and {1,9,10}. With this codificatoin, the transmission cost of n triangles is reduced in a factor of three, from $3 \cdot n$ to $n + 2$ vertices.

The use of triangle strips in 3D models offers an important improvement, since this primitive offers an implicit codification of the connectivity which has many advantages, like lower storage necessities and faster rendering.

Stripifying techniques

Although converting a triangularized mesh into an optimum set of strips is an NP-complete problem [31], many papers have presented different solutions to maximize the performance of the output strips [32, 33, 34, 35]. A complete overview of stripification methods as well as a deep description and comparison of the most used models can be found in [36].

A different approach is offered by those works that represent a mesh using a single triangle strip [37, 38]. These works create an efficient triangle strip with the addition of a small amount of faces that do not alter the original geometry. Thus, they are capable of offering a faster rendering. Nevertheless, from the perspective of a multiresolution scheme, these set of techniques are difficult to apply as the update of the level of detail and also the management of the degenerate triangles in a single strip is usually too expensive. It would not be possible to exploit coherence and it would involve rearranging and moving large sections of the strip. As a consequence, the initial increase of performance would be easily lost.

Finally, mention should be made of algorithms like the one proposed by Belmonte et al. [39], which considers the generation of strips following a simplification criterion. In paper [40], the authors propose a different method for strips generation which starts the stripification process from the mesh simplified to the minimum level of detail, and constructs the strips by following the simplification information until the original geometry is obtained.

The suggested algorithms show differences in generation and rendering speed, in the use of memory or in the number of strips generated, which make them more suitable for a specific use.

2.3.2. Cache-optimized primitives

Following with the improvement of the rendering primitive, it is also important to comment on the studies which make optimum use of the vertex cache and from the spatial locality of vertex buffers. The vertex cache is a special GPU memory where vertex data is stored. If a vertex is in the cache, then it does not have to be transformed and lit again if we need to use it again. Then, the objective of cache-aware optimizators is to reuse calculated vertices as much as possible, considering that the size of the vertex cache is very limited.

Primitive optimization has been addressed by several authors in order to increase the performance of their models [41, 42], ordering the indices in an optimized way and obtaining a much faster rendering.

In the specific case of triangle strips, Hoppe proved that an efficient management of the vertex cache can reduce vertex processing time by a factor of approximately 1.6 to 1.9 [3]. These stripification techniques often produce shorter triangle strips, which frequently offer better performance as they lead to a wiser use of coherence [35, 38, 43].

Most of the cache-aware methods require knowing in advanced the cache size of the targeted hardware, as input meshes ordered for a specific size may offer a bad performance when rendered with different cache sizes. For those cases when the cache size is unknown, it is possible to use *cache-oblivious* algorithms which output triangle orderings that work well with different sizes. This orderings are also known as *universal* index sequences [44, 45, 46].

2.4. Multiresolution modeling

Extensive research has been carried out in multiresolution models for more than ten years. Evolution of graphics hardware has given rise to new techniques that allow us to accelerate multiresolution models. A comprehensive characterization of multiresolution models can be found in [9].

Following the taxonomy presented in [2], we can basically distinguish between three approaches when referring to multiresolution models:

- discrete models, which contain various representations of the same object with different levels of detail (typically between five and ten).
- continuous models, which represent a vast range of levels of detail where two consecutive approximations only involve altering a few polygons.
- view-dependent models, which are *anisotropic* models, capable of offering different levels of detail for different areas of the same object at the same time.

Figure 2.4 offers the Isis model under different circumstances. The left image depicts the mesh at the highest level of detail, while the image on the middle presents a simplified version. Finally, on the right it is shown a view-dependent situation, where half the model is simplified.

Many of the works available in the literature were written in the early days of the GPUs (or even in earlier times [12]) when it was advisable to spend some CPU processing time to optimize the GPU rendering process. Nevertheless, in recent years the authors have re-oriented their efforts towards the development of new models which consider the possibilities offered by new graphics hardware. Recent GPUs include different processors which have evolved from being configurable to being programmable, allowing us to execute shader programs in parallel. Thus, in this section we will mainly focus on the lines of work that are currently active in the level-of-detail field which are oriented towards the exploitation of GPUs.

2.4.1. Discrete models

For creating a discrete model, 3D designers create a set of accurate representations of the same object with different geometrical complexities. Then, the

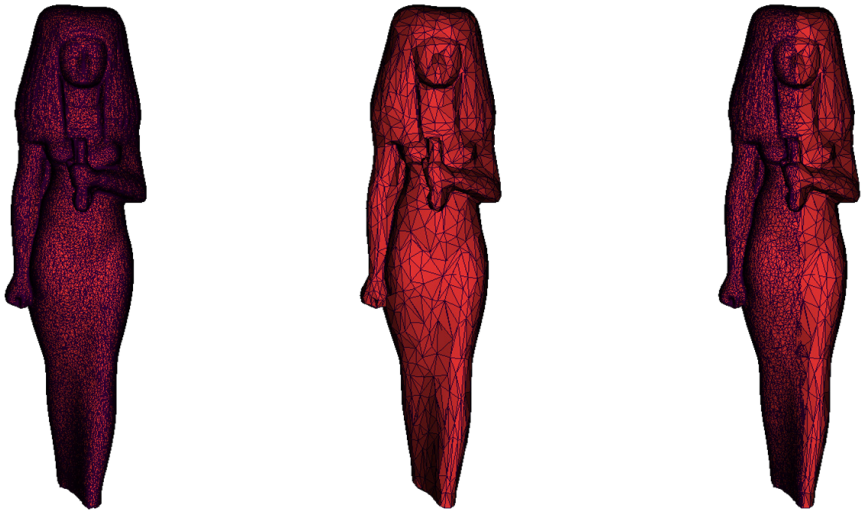


Figure 2.4: Different multiresolution approaches for the Isis model: on the left the original geometry, in the middle a continuous approximation and on the right a view-dependent approximation.

level-of-detail algorithms decide in real-time which representation of the mesh is the most suitable one, depending on the scene requisites and the criterion used. These multiresolution models are widely included in graphics libraries such as OSG, in standards like VRML and in the majority of game engines due to their simplicity.

The main problem of discrete models is the *popping* produced when changing among the pre-calculated levels of detail, as swapping between two approximations of the original model with different complexity can be visually perceptible. These artifacts were shown in Figure 1.2, presented in the previous chapter. Moreover, discrete models have a high storage cost as they have to store different copies of the connectivity data. Despite these problems, in practice this is a very used technique because switching between static geometry is cheap and efficient.

The evolution of graphics hardware has encouraged the development of new discrete models which upload the different approximations to the GPU and perform smooth transitions between them by means of geo-morphing or blending [47, 48], reducing the noticeable *popping* artifacts. An important drawback is the fact that they do not work properly with meshes that include different sets of attributes per-vertex [47].

2.4.2. Continuous models

In the beginning, discrete models were employed in graphics applications, mainly due to the low implementation complexity they showed, which is the reason why they are still used in applications with no great graphics requirements. Nevertheless, the increase in realism in graphics applications compels the use of multiresolution models which are more exact in their approximations, which do not require high storage costs and which offer faster visualization. This has given way to continuous models where two consecutive levels of detail only differ in a few polygons, and where duplicated information is avoided, which considerably improves memory cost. By using these models it is possible to obtain a continuous spectrum of approximations. In addition, continuous models are able to avoid the visually disturbing *popping* artifacts.

It is possible to find in the literature continuous algorithms aimed at rendering common meshes by exploiting GPUs. An approach to improve traditional level-of-detail techniques was the use of primitives with implicit connectivity, like triangle strips, which offer a faster processing on the GPU and also reduce BUS traffic. Using this idea, one of the most used continuous models is Progressive Meshes [16, 49]. It is included in Microsoft Corporation's graphics library DirectX since the 5.0 version. Afterwards, many extensions were developed to improve this model from different perspectives [50, 51]. Following with the ideas introduced in Progressive Meshes, El-Sana et al. [52] presented later the Skip Strips model, which was the first model to maintain a data structure to store the strips that avoided the need to calculate them in real time. The MTS model [53] used triangle strips both as the storage and the visualization primitive. It consisted of a set of multiresolution strips, each of which represents a triangle strip and all its levels of detail; only the ones that are modified when changing the level of detail are updated before being rendered. More recently, the LodStrips model was developed [54]. This continuous model was entirely based on optimized hardware primitives, triangle strips, and dealt with the apparition of degenerate triangles by applying pre-calculated filters.

A simple way to harness the power of GPUs is to exploit the complex memory hierarchy of modern graphics platforms. In this sense, many authors develop "GPU-friendly"vertex and index buffers. The idea of these buffers (known as *Vertex Buffer Objects* in OpenGl and simply *Vertex Buffers* in DirectX) is to store the model data in contiguous chunks of memory on the graphics card, which offers a faster rendering. Moreover, this approach allows us to minimize data transfers between CPU and GPU. The LodStrips model was reconsidered in [18] to offer a GPU-oriented solution, by storing the mesh information in Vertex Buffer Objects, which offer a faster rendering. Their algorithms were adapted in order to update the information in the memory of the graphics hardware, although their extraction process was still costly. The work presented by Turchyn [55] combines the Progressive Meshes works [16, 49] with a sliding window algorithm. This proposal minimizes data transfers between CPU and

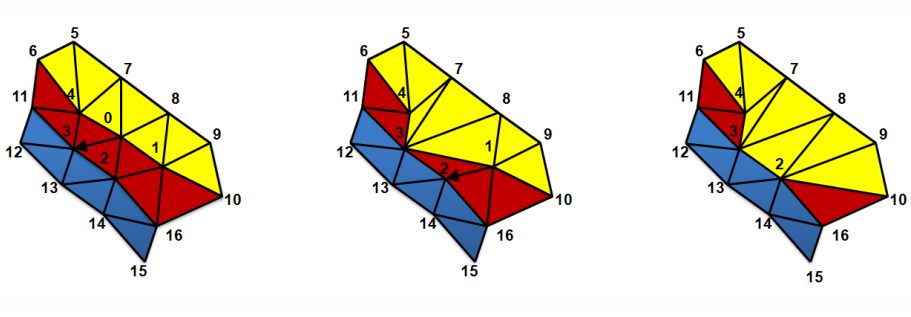


Figure 2.5: Collapse of a strip.

GPU by storing mesh data in static buffers on the GPU. The problem is that it builds a complex hierarchical data structure that derives in great memory requirements and, moreover, it changes the mesh connectivity trying to reduce memory costs.

The tendency in recent years has been to harness the potential of GPUs to perform the level-of-detail extraction completely on graphics hardware. Ji et al. [56] 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. The main problem of this method is the costly process that the CPU must execute in every change of level of detail. 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. Besides, the authors point out that this solution has problems with the data-transmitting rate of graphics bus.

Degenerate triangles

Multiresolution models based on strips present a limitation that arises when, starting from a set of strips representing the initial mesh at maximum detail and applying the successive simplifications, the strips start to include a large quantity of degenerated triangles, repeated vertices and unnecessary edges. An example of these strips can be observed in Figure 2.5, where the strip in the middle is collapsed after two simplification steps. This strip, coloured in red, is initially formed by indices $\{6, 11, 4, 3, 0, 2, 1, 16, 10\}$. After collapsing edges $\{0, 3\}$ and $\{1, 2\}$, this strip is represented with indices $\{6, 11, 4, 3, 3, 2, 2, 16, 10\}$, where it is possible to find four degenerate triangles ($\{4, 3, 3\}$, $\{3, 3, 2\}$, $\{3, 2, 2\}$ and $\{2, 2, 16\}$) that do not add any geometric information but involve a higher processing time.

One possible way to overcome this problem is to use strips which are dynamically generated for each level of detail. Research has been conducted on

this approach, and it is possible to find methods of building and maintaining a good set of triangle strips [57], and also multiresolution models based on dynamic strips [58].

Static stripification involves filtering the degenerate information every time there is a change in the level of detail. Some authors apply a filtering process in visualization time to avoid sending those vertices at the moment of rendering [52]. Others detect the degenerate triangles in a pre-process step in order to collect the information that will be used for their elimination at the moment of rendering [18, 54].

Depending on the targeted multiresolution scheme, both solutions can be more or less adequate. The additional cost involved in generating the strips for every level of detail is high, and can be inefficient for interactive visualization. Therefore, the use of static strips can turn out to be more suitable. Nevertheless, the application of filters is only able to eliminate degenerate triangles to a limited extent, as these models still present a noticeable amount of degenerate triangles in the final rendered geometry.

2.4.3. View-dependent models

A different improvement in multiresolution models was the introduction of view-dependent techniques. These methods introduce more accurate representations, presenting a higher detail in those areas of the object where it is necessary. Among the wide range of existing algorithms, it is important to mention the works presented in [17, 58, 52, 59, 60, 61].

Despite their better granularity, these methods require a higher extraction time as they have to calculate the viewing conditions and extract the detail according to these conditions. Therefore, the data structures are usually more complex and the overall process is costly for a final application. Some years ago, when using pre-3D-hardware PCs, it was encouraged the use of significant CPU time to alleviate some geometry rendering. Nowadays, graphics hardware is very powerful and therefore we will only spend CPU time if we can discard a lot of triangles or if it can save other costly processes. As a consequence, view-dependent techniques are aimed at specific applications, like rendering terrain or massive models.

The appearance of massive models, which can be considered as gigabyte-sized polygon models that cannot be completely loaded into the main memory, led to the development of new view-dependent techniques. In order to process this enormous amount of information, authors resort to out-of-core algorithms. The basic idea is to arrange the mesh so that it does not need to be kept in memory in its entirety, and adapt its computations to operate mainly on the loaded parts. Many works appeared as the use of massive models became more and more common. Among them, the reader is referred to [62, 63, 64] or more recently [65]. However, these models are more suitable for CAD or cultural heritage applications where highly-detailed models are needed [66].

Many of the GPU-based continuous models are aimed at view-dependent rendering of massive models. Many researchers have recently proposed methods for moving the granularity of the representation from triangles to triangle patches in order to offer view-dependent capabilities for rendering out-of-core models [67, 68, 69, 70]. These works have adapted their data structures so that CPU/GPU communication can be optimized to fully exploit the complex memory hierarchy of modern graphics platforms. The initial Multi Triangulation framework (MT) was improved in [69] to offer a new out-of-core multiresolution model which was redesigned in a GPU friendly fashion. Following the same idea, it is also possible to find the Progressive Buffers [70] model, which was developed as a view-dependent algorithm for massive models that do not fit inside the GPU.

With a similar objective but with a further GPU exploitation, the GoLD method [48] introduced a hierarchy of geometric patches for very detailed meshes with high resolution textures. The maintenance of boundaries was assured by means of geo-morphing performed on the GPU. Niski et al. [71] offered a multi-grained hierarchical solution which avoids the appearance of cracks in the borders of nodes at different levels of detail by applying a border-stitching technique directly on the GPU. Later, the work presented in [72] introduced a GPU-based adaptive model for non-photorealistic rendering. They proposed a hierarchical multiresolution model and used the GPU to refine the areas around the silhouettes. More recently, Hu et al. [73] presented a fully-GPU implementation of Progressive Meshes [59]. They offered view-dependent capabilities at the expense of a high memory cost and an extraction process which consists in three rendering passes.

2.4.4. Other techniques

It is possible to find in the literature many approaches which aim at reducing the mesh complexity to offer real-time adaptive techniques. This set of methods provides a solution to the geometry management on the GPU and suppose a line of investigation which is currently very active.

In the field of computer graphics, tessellation techniques are often used to divide a surface in a set of polygons. Thus, we can tessellate a polygon and convert it into a set of triangles (see Figure 2.6), or we can tessellate a curved surface. These approaches are typically used to amplify coarse geometry, while multiresolution frameworks are usually designed to exactly reproduce an originally detailed mesh.

Following this idea, it is possible to find solutions which propose sending to the GPU a mesh at minimum level of detail and applying later a refining pattern to every face of the model [74, 75, 76]. Other researchers have proposed frameworks for calculating silhouettes on the GPU and tessellating afterwards those areas that need further detail [72, 77], although the process for calculating the silhouettes is complicated. Programmable graphics hardware has

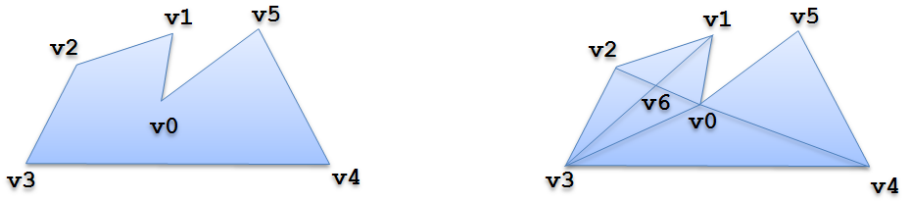


Figure 2.6: Tessellation of a sample polygon.

allowed many surface tessellation approaches to migrate to the GPU, including isosurface extraction [78], subdivison surfaces [79], NURBS patches [80], and procedural detail [81, 82]. These models offer very interesting results although they are not completely aimed at rendering meshes in real-time applications due to the load suffered by the GPU when a model maintains its level of detail, as a pass must be made for each face that the coarser model has. Moreover, these solutions are not capable of retrieving the original mesh geometry, and some of them still suffer from *popping* artifacts.

From a different perspective, DeCoro et al. [83] present a scheme for simplifying arbitrary meshes using octree-based vertex clustering. This clustering strategy avoids precomputation and storage of a vertex hierarchy, but the resulting approximating meshes are less accurate.

Some models introduce surfaces with a completely regular structure called *geometry image* [84]. They capture geometry as a simple 2D array of quantized points. Surface signals like normals and colors are stored in similar 2D arrays using the same implicit surface parameterization. The problem with this representation is aliasing [85] and the complex construction process of these representations [86].

2.4.5. Characterization

In Table 2.1 we can observe a description of the multiresolution models that take advantage of current GPUs that have been considered in this section. The description takes into account the following aspects:

- Model: this information identifies the research article, indicating the authors, the year of publication and the reference.
- Type: it indicates whether the model is discrete, continuous or view-dependent.
- Primitive: this field includes the type of primitive that the model uses in visualization.

Figure 2.7: Army of multiresolution ninjas in a game engine.

- GPU usage: whether the scheme uses vertex buffers or shaders, indicating, if it is applicable, which type or shaders are used.
- Others: this aspect offers further features that characterize each of the solutions.

2.5. LOD selection criteria

The necessity of highly realistic scenarios often involves including many polygonal meshes made up of a high number of triangles, which poses a problem for maintaining interactivity. Figure 2.7 presents a scene containing hundreds of animated ninjas where each of them adapts its geometry to the distance to the camera at the same time. In these applications, it is important to guarantee stable frame rates while reducing perceived lag [87]. The lag, which is the delay between performing an action and seeing the result of that action, is as important as the frame rate to perceive interactivity in an application.

Multiresolution modeling offers a perfect framework to assure a minimum frame-rate that offers interactivity to the final user. A problem that must be considered is the criterion used to select the proper level of detail of the meshes included in the rendered scenario. Thus, the multiresolution scheme must increase or decrease the complexity of each model to maintain the frame-rate and, as a consequence, needs a criterion to decide which is the most adequate level-of-detail for each mesh.

Many authors have addressed the need of investigating how the human perception system works. In [88] the author considers the necessity of including an analysis of the human visual system to understand how it works and to offer more adequate results, extending his results in his subsequent publications.

Model	Type	Primitive	GPU usage	Others
Southern et al., 2003 [47]	Discrete	Triangles	Vertex Shaders	Geo-Morphing
Losasso et al., 2003 [86]	Discrete	Triangles	Vertex and Pixel Shaders	Geometry Images
Cignoni et al., 2004 [67]	View-dependent	Strips	Vertex Buffers	Massive Models
Yoon et al., 2004 [68]	View-dependent	Triangles	Vertex and Pixel Shaders	Massive Models
Sander et al, 2005 [70]	View-dependent	Triangles	Vertex and Pixel Shaders	Massive Models
Cignoni et al., 2005 [69]	View-dependent	Strips	Vertex Buffers	Massive Models
Borgeat et al., 2005 [48]	View-dependent	Triangles	Vertex and Pixel Shaders	
Boubekeur et al., 2005 [81]	View-dependent	Triangles	Vertex and Pixel Shaders	Tessellation
Ji et al., 2006 [56]	View-dependent	Fans	Vertex and Pixel Shaders	Fully-GPU
Ramos et al., 2006 [18]	Continuous	Strips	Vertex Buffers	
Turchyn, 2007 [55]	Continuous	Triangles	Vertex Buffers	Cache-Optimization
Niski et al., 2007 [71]	View-dependent	Strips	Vertex and Pixel Shaders	Massive Models
DeCoro et al., 2007 [83]	View-dependent	Triangles	Vertex, Pixel and Geometry Shaders	Real-time Simplification
Livny et al., 2008 [72]	View-dependent	Triangles	Vertex and Pixel Shaders	Silhouette Preserving
Boubekeur et al., 2008 [74]	View-dependent	Triangles	Vertex, Pixel and Geometry Shaders	
Dyken et al., 2008 [77]	View-dependent	Triangles	Vertex and Pixel Shaders	Silhouette Preserving
Lorenz et al., 2008 [75]	View-dependent	Triangles	Vertex, Pixel and Geometry Shaders	Refinement Patterns
Schwarz et al., 2009 [76]	View-dependent	Triangles	Vertex, Pixel and Geometry Shaders	Tessellation
Hu et al., 2009 [73]	View-dependent	Triangles	Vertex, Pixel and Geometry Shaders	Fully-GPU

Table 2.1: Characterization of GPU-based multiresolution models.

In this sense, several authors have included biometrics into their heuristics, considering spatiotemporal sensitivity [89] or developing frameworks with eye tracking as the basis [90].

Most multiresolution models use static heuristics, like the distance, the speed or the position in the screen, as the metric to select the suitable level of detail. The approach presented in [91] uses a multiresolution hierarchy based on bounding spheres and perform the LOD selection based on the projected size in the screen. They also gradually refine the model when the viewpoint is not moved for a period of time. Other works like [92] add a static heuristic based on the occlusion information to obtain a tighter estimation of the contribution of each object to the scene. These set of heuristics, despite improving frame rates, are usually not enough. They cannot guarantee a stable performance and often present jerky frame rates, as they are not adaptive and cannot work correctly in scenarios where objects are moving in and out of the scene or where the objects become bigger or smaller quickly.

In order to improve the results of the static heuristics, some authors have introduced the use of feedback algorithms, which take into account the past rendering times. These algorithms, even though are more adapted to the rendering conditions, also suffer from oscillation and unavoidable overshoot when rendering discontinuous environments. They present a good alternative for scenarios where there is a large amount of coherence between frames, as it happens with flight simulators. This is the case of the solution presented in [93], which provides temporal coherence through the runtime creation of geomorphs to control de level of detail.

Funkhouser and Séquin [12] demonstrated that it is necessary to use a predictive selection scheme, based mainly on the complexity of the current frame, rather than a reactive framework, based on the feedback obtained. They formulated this problem as an optimization task which is equivalent to a constrained version of the Multiple Choice Knapsack Problem. Even though this problem is NP-complete, some authors like [12] or [94] obtained several techniques that could only guarantee a solution that is at least half as good as the optimum one. Wimmer et al. [95] reconsidered this problem for the special case of continuous multiresolution models, obtaining a non-iterative closed form solution which was cheap to evaluate for every frame.

This way, the problem of the time-critical multiresolution rendering can be presented as an optimization problem for finding the LOD that maximizes the scene quality under timing constraints. The work in [96] extended the use of predictive techniques with more precise heuristics for the cost and the benefit of the resolution of the objects. It also considers temporal coherence to minimize sudden changes, although the authors did not include it in their tests. These optimizations are very accurate but costly, and as they assign one variable for each object, rendering scenes with a large number of objects tends to be a slow solution.

Different researchers have presented architectures to solve this problem, like

[97], which uses a distributed rendering architecture to obtain a stable frame-rate, or [98], which proposes a parallel architecture combined with levels-of-detail and occlusion culling techniques. The most novel aspect of [99] is the concept of interruptible rendering, which finds a rational compromise between spatial and temporal detail. They produce a complete image in the back buffer almost immediately and then incrementally refine it so that the refinement can be interrupted at any time. Zach [100] presents a solution based on geomorphing where the LOD management is achieved by distributing the LOD selection and calculation between several frames, reusing the old resolution until the new one is ready. As the new LODs will appear in future frames, they need a path prediction process to obtain future viewpoints and directions. They also use cost and benefits computation, but include some feedback strategy to compensate for some assumptions they make. These authors extended their work in [101], presenting an approach for discrete and continuous models where the time spent for LOD selection is amortized over several frames.

2.6. Conclusions

This chapter has presented the state-of-the-art on multiresolution modeling, mainly focusing on the exploitation of GPUs.

In general, we can say that discrete models are easier to be implemented in GPUs. In this sense, Southern and Gain [47] implemented a discrete model manager which puts blending into practice by using vertex shaders. Nevertheless, these models still present noticeable *popping* artifacts.

By contrast, continuous models offers a better granularity and avoid that problem, but their memory requirements are high and some of them even need several rendering passes to change the level of detail. It is worth mentioning that the latest research on multiresolution modeling is aimed at developing view-dependent techniques. Thus, the computational power of current GPUs can be exploited to offer more accurate approximations and manage massive models. GPU programming has also been applied to tessellate an initial coarse mesh, by means of the recent *Geometry Shader* which is capable of creating and eliminating geometry directly on the GPU.

Moreover, in this study of the previous work we have also analyzed simplification techniques, primitive optimization and LOD selection criteria. These techniques are fundamental for offering a successful level-of-detail model. Researchers must select the most proper techniques. The simplification algorithm affects not only the visual quality of the 3D model throughout the different levels of detail, but also the related extraction process and data structures as the selected simplification operation may require more complex implementations. Regarding the primitive selection, we have shown that it is possible to orient the multiresolution models towards using cache-aware orderings and also primitives with implicit connectivity. Nevertheless, when this type of rendering

primitive is selected the multiresolution model must perform an extra process to assure that the degenerate information is treated correctly. Finally, it is worth mentioning that selecting the most adequate level-of-detail for a specific scenario is not an arbitrary task. Although in most cases the distance to the camera is the selected criterion, it is possible to apply more accurate heuristics that balance the perceived visual quality and the extraction time that the multiresolution model needs. Moreover, it is also interesting to consider that those scenarios in which many multiresolution models are managed at the same time need a more complex level-of-detail management as the total time required by the different models may become a bottleneck for the graphics application.

CHAPTER 3

Optimizing the Management of Level-of-Detail Models

In this chapter we present *Speed Strips*, a new continuous multiresolution framework which has been developed in view of the outstanding evolution of hardware. Our interest not only focuses on exploiting GPUs possibilities, but also on making the best possible use of the capabilities offered by new bus technologies. The results section shows that our model improves the efficiency of previously existing solutions.

3.1. Introduction

Graphics hardware has improved outstandingly over recent years, although the need for very detailed 3D models is growing faster than the capabilities offered by graphics hardware. Thus, since hardware was improving and new technologies were presented, we decided to develop a new continuous multiresolution framework to improve the performance of previous solutions by taking advantage of the features of the latest hardware.

The solution presented in this chapter has been devised from the experience obtained after analyzing different multiresolution approaches, and in a more precise way, after the implementation of LodStrips [18, 102]. This continuous model was entirely based on optimized hardware primitives, triangle strips, and dealt with the creation of degenerate triangles applying pre-calculated filters. Nevertheless, this multiresolution model presented important limitations. The main drawback was the extraction process, which entailed updating the strips by performing costly random insertions and resizes.

Figure 3.1: Color-coded LOD scene with many Speed Strips models inside the Ogre rendering engine.

Our main objective was to develop a level-of-detail update routine which maximized the efficiency of the extraction process and minimized the data traffic through the bus. The approach presented, Speed Strips, exploits the new hardware capabilities in two ways. On the one hand, the possibility of storing geometry information in the graphics hardware by means of vertex buffer objects vastly improves the visualization time. On the other hand, the PCI Express bus supports isochronous data transfers and different QoS levels, which guarantees that the data arrive at their destination in a given time. The use of multiple isochronous virtual channels per lane presents a perfect solution for applications which require real-time data transfer [103]. With these two features in mind, we have developed a triangle strips updating routine which modifies the strips in one single step by working directly with the information stored in the GPU and sending the minimum amount of information in the most appropriate way.

Speed Strips presents the following features:

- low memory cost, if compared with discrete models which involve having to store different approximations of the same model.
- short extraction time, as it is capable of applying the level-of-detail changes and eliminating unnecessary information in one single pass.
- based on triangle strips, which not only offer a more compact representation of the connectivity existing in a triangle mesh but also a faster rendering.

Figure 3.1 presents a scene of the Ogre rendering engine where our multiresolution model has been integrated. To adjust the detail, we have considered the distance to the viewer criterion. We have also color-coded the different models to represent the level of detail at which they are rendered.

This chapter starts by proposing the general framework in which this model has been developed. After that, thorough details of the proposed method are

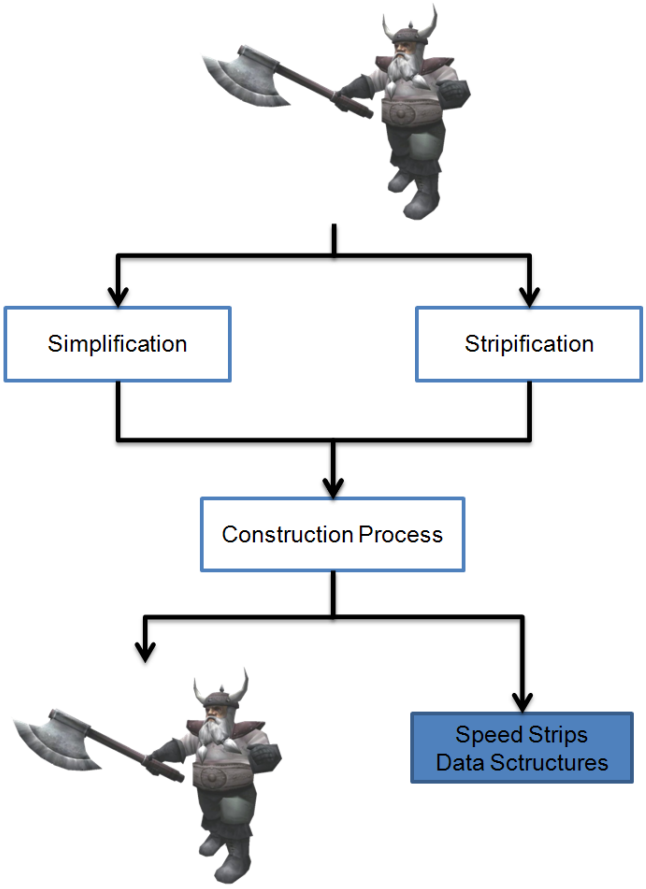


Figure 3.2: Pre-process diagram.

presented. Then, we describe different versions of the original method which have been developed for testing purposes. Afterwards, a comparative study of our algorithm against previous discrete and continuous solutions is introduced. Lastly, we conclude this chapter by commenting on the results obtained and outlining briefly future lines of work.

3.2. General Framework

Multiresolution models usually perform a pre-process step where the necessary data is gathered. Figure 3.2 describes the different tasks that must be carried out to prepare a Speed Strips model.

Figure 3.3: Example of simplification. Figure on the left is the original model. Figure in the middle represents the model simplified to 50 %. Figure on the right is the model simplified to 25 %.

In order to construct a Speed Strips model we need to meet two initial requirements:

- the simplification of the original mesh, which will allow us to iteratively reduce the geometrical complexity of the mesh.
- the stripification of the initial mesh, which will offer a more compact representation of the geometry.

With both sets of information, the construction process will output the necessary data structures and, in addition, a re-ordered mesh which is directly renderable, as its information is not altered.

In the rest of this section we will present the algorithms chosen for simplifying and stripifying the original meshes, addressing the reasons behind the selection of these techniques. We will also describe the tasks that are included in the construction process. It is worth mentioning that the selected simplification and stripification techniques will also be used in the other multiresolution approaches presented in the subsequent chapters.

3.2.1. Simplification of the original mesh

The objective of the simplification is to reduce the complexity of the input mesh to obtain a coarser approximation. In the specific case of level-of-detail modeling, the simplification process is a key aspect as it gathers the information that will be later used to retrieve the different levels of detail.

Between the possible simplification operators, throughout the different multiresolution models that we have developed we will apply the edge collapse, which was firstly introduced in [19] and later widely used in multiresolution. More precisely, for our framework we will choose a simplifications process based on a half-edge collapse [2] (see Figure 2.2). With this approach, the vertex to which the edge collapses to is one of its end points and, as a consequence, it will not be necessary to add any new vertex to the original mesh. The selection

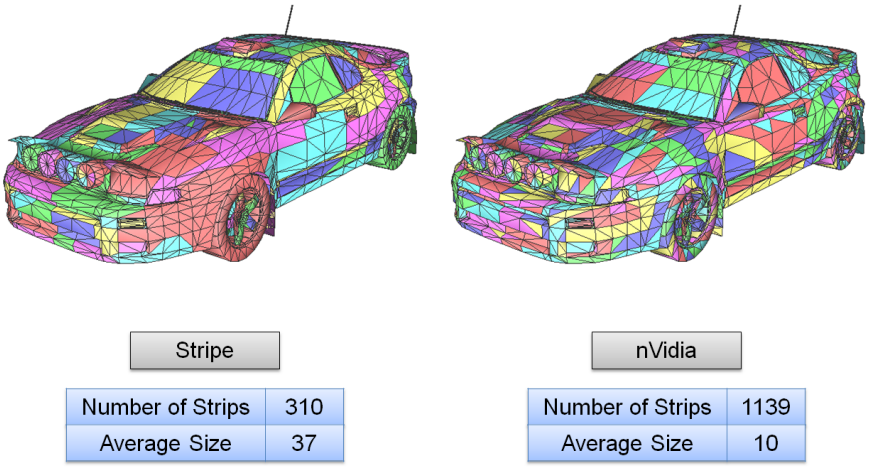


Figure 3.4: Example of stripification of a Toyota model: on the left the output of the Stripe algorithm and on the right the result of the cache-aware nVidia approach.

of this type of edge-collapse simplifies the data structures of our model and still offers very accurate simplifications. Nevertheless, our approaches are not restricted to use half-edge collapses where we do not add new vertices. The only restriction is to pre-calculate the vertices so that we can store them properly before starting to use our multiresolution model.

There are several mesh simplification methods [1, 20] which could be used for our purposes. For constructing a Speed Strips model we decided to use the method based on viewpoint entropy presented in [29]. After the simplification process, it will be necessary that the algorithm outputs an ordered sequence of the collapses that have been applied. It is possible to collect all this information during the simplification of the original model with the algorithm we are presenting. We will consider that each collapse operation entails a change in the level of detail. Thus, we will have as many levels of detail as edge collapses have been recorded during the simplification process. Figure 3.3 offers some results of this algorithm. The original model is shown on the left, which is composed of several submeshes. In the center, we present the simplified model to 50 %. Finally, we can observe the same object simplified to 25 % on the right. This simplification preserves the original shape to a large extent, avoiding the appearance of artifacts and holes and offering a high visual quality.

3.2.2. Stripification of the original mesh

The use of triangle strips in 3D models offers an important improvement, since this primitive offers an implicit codification of the connectivity which has many advantages, like lower storage necessities and faster rendering. Moreover, the data structures of a strip-based multiresolution model also present a lower memory cost.

In the work we are presenting we have decided to use the Stripe algorithm [32], which is one of the mostly used stripification techniques. Despite the better performance of the cache-aware techniques [35], their triangle strips are less adequate for our objectives. Figure 3.4 presents the stripification of a Toyota model using both algorithms and shows how Stripe outputs fewer strips and, in addition, these strips tend to be longer. These characteristics reduce the storage cost of the strips and make it easier for our filtering approach to find and erase degenerate triangles. Nevertheless, in the results section we will address more thoroughly the reasons behind the selection of this algorithm and also behind choosing triangle strips as rendering primitive.

Dealing with degenerate triangles

Following the information obtained during the simplification process, the multiresolution model will be able to perform the collapses in order to obtain the desired level of detail. As previously commented, the simplification method we are using is based on iterative edge contractions, and therefore the basic operation of our model will be the *edge collapse*. These collapses are reflected in the geometry by updating the indices of the triangle strips. An example of the simplification process can be seen in Figure 3.5, where two triangle strips are depicted throughout three simplification steps. The first step entails vertex v_0 collapsing to vertex v_2. For our extraction process, this collapse involves locating all the indices referencing vertex v_0 in all the strips of the mesh and replacing them by vertex v_2.

Subsequent collapses of a triangle strip poses an important difficulty: the appearance of degenerate triangles. A small percentage of these degenerate triangles will be necessary to avoid discontinuities inside the strips, although most of them are unnecessary and involve processing zero-area triangles that will not influence the final rendered geometry but involve a higher processing time. As a consequence, when applying collapses to triangle strips it will be possible to find a wide range of degenerate triangles. In the example provided in Figure 3.5 we can observe two different kinds of repetitions creating degenerate triangles. After three simplification steps, strip number one presents an edge repetition and strip number two a vertex repetition. These repetitions follow the patterns $ab(ab)^+$ and $aa(a)^+$ respectively, where a, b are indices of the triangle strips. The pattern $aa(a)^+$ can be replaced by aa and $ab(ab)^+$ by ab without altering the final geometry.

To avoid this unnecessary geometry, some authors re-stripify the mesh after

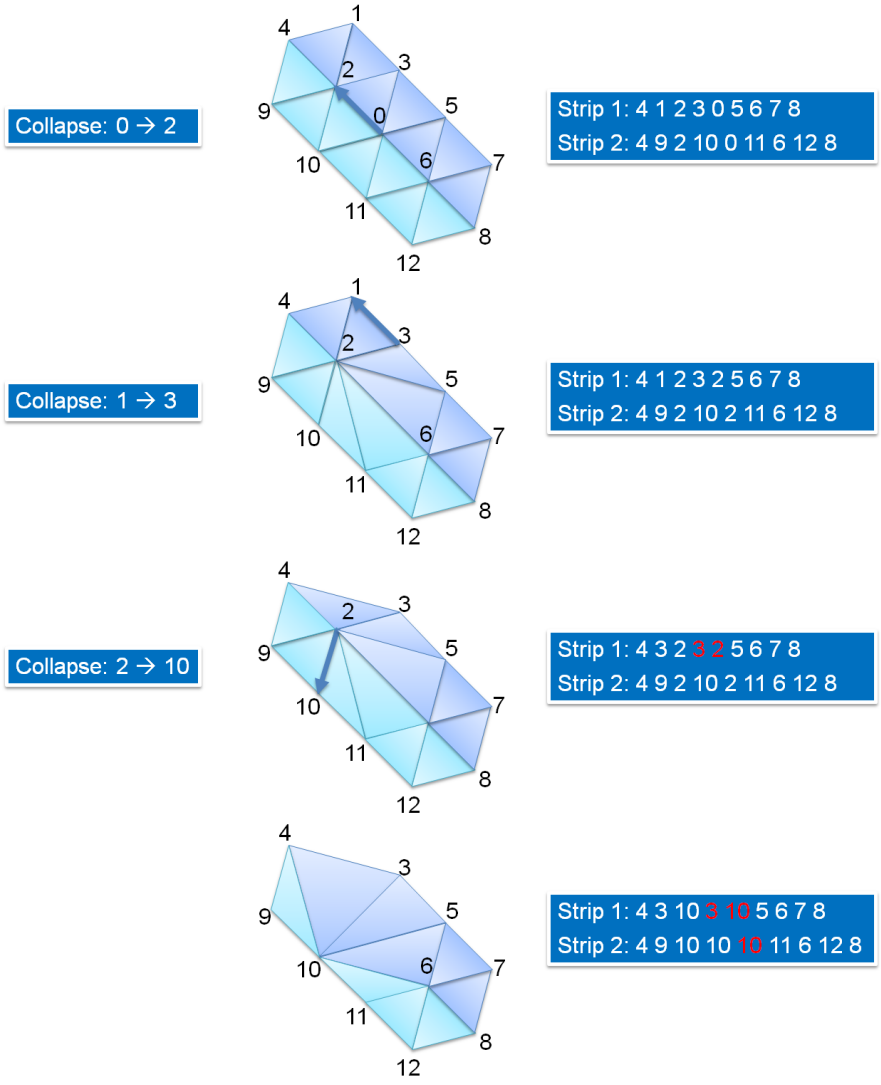


Figure 3.5: Degenerate triangles after several edge collapse operations. The red-coloured numbers indicate positions that might be eliminated without altering the geometry.

the LOD update. This dynamic strategy would force us to create and destroy strips on the GPU, which would make our model considerably less competitive. In our case we decided to maintain a static set of triangle strips and solve the problem by applying filters to eliminate degenerate triangles. It could also be possible to adopt an intermediate solution, maintaining an original stripification but merging strips when possible. Nevertheless, this solution would entail a more complex extraction process which would also include copying and rearranging strips in GPU memory.

Regarding the filtering process, some authors apply the filters in visualization time to avoid sending those vertices at the moment of rendering [52]. Others detect the degenerate triangles in a pre-process step in order to collect the information that will be used for their elimination at the moment of rendering [18, 54]. Since we are developing our multiresolution model with modern GPU architectures in mind, this second option will be the one we will use for our model, as our main objective is to reduce the extraction time as much as possible.

Our tests have proven that most degenerate triangles follow the two patterns presented above. Applying those filters allows us to reduce the amount of degenerate triangles by around 75 %. It would be possible to eliminate more degenerate triangles adding more patterns, but the amount of information that we would have to store, the wide range of combinations that should be considered and the time required to apply those filters have led us to accept these degenerate triangles. Moreover, if we applied all the filters to eliminate degenerate triangles as much as possible, the performance when rendering would slightly increase, which we considered is not worth the effort as it would not compensate for the extra time that the extraction process would needed to prepare the triangle strips.

3.2.3. Construction process

Before describing the construction process, it is worth reminding that we will use a simplification algorithm which uses half-edge collapses. In this operation two vertices are necessary: the vertex that disappears and the vertex that is maintained. In order to reduce storage needs, we will initially reorder the vertices according to their simplification order. As, by definition, each edge collapse supposes a change in the level of detail, with this order we can assume that vertex v_i will disappear when changing from lod_i to lod_{i+1}. In the example given in Figure 3.5 we presented three simplification steps. The vertices had been previously ordered, and as a consequence, in LOD 1 vertex v_0 disappears, in LOD 2 vertex v_1 is eliminated, and so on. Thus, we will only need to store the vertex that is maintained after the collapse happens. Finally, it is important to comment that through all the examples we have considered that vertices and indexes arrays start at position 0.

Preparing the necessary data for the Speed Strips model entails consider-

ing which indices change when applying a collapse and where can be found degenerate triangles that might be removed. Thus, for each collapse of the simplification sequence we will:

- locate which strips contain the vertex that will disappear and in which position. This information must be stored and the strips must be updated to reflect the simplification step.
- locate repetitions that might be removed in the modified strips. Once again, this information must be stored in the data structures and the strips must be consequently updated.

As we have already mentioned, one of the most important advantages of this multiresolution model is the possibility of performing all changes in one single pass. These changes include, on the one hand, updating the triangle strips to reflect the level-of-detail change and, on the other, eliminating the unnecessary geometry. Performing both types of changes at the same time requires all the operations to be in the correct order. Thus, after locating the elements to change and the degenerate triangles to remove in a level of detail, we will order the operations before storing the information in the data structures. This process is repeated until we obtain the coarsest approximation.

3.3. The Speed Strips model

The solution we are presenting has been developed taking LodStrips [18, 102] as a basis. LodStrips presented a level-of-detail extraction process which entailed rearrangements of memory when making insertions or deletions in the triangle strips which were too costly.

In the previous section we have commented on the selected algorithms for simplifying and stripifying the meshes, which are more adequate than the ones LodStrips used. On top of that, we have also considered more thoroughly the treatment of the degenerate triangles. But the main improvements of Speed Strips appear in the extraction process, as it applies all changes in one step and updates the level-of-detail directly in the GPU, reducing bus traffic to a high extent as the CPU is only needed during the filtering process. As a consequence, new algorithms and data structures have been developed for the correct performance of the model.

3.3.1. Data structures

The final data structure will be composed of three major elements. Figure 3.6 contains these data structures in a *c-like* notation. The information about the vertex that is preserved after the collapse and how many strips should be updated is stored in an array called *LODInfo*. The strips that must be updated and the information about the number of changes that must be applied

will be stored in a data structure called *StripInfo*. This number of changes includes both operations: edge collapses and repetitions removal. Finally, we have an *UpdateInfo* array which includes all the information about collapses and unnecessary degenerate triangles in an ordered and interleaved manner. This array contains the position where the collapses and the repetitions are found and, with respect to the repetitions, it also includes the type of repetition and the number of elements to eliminate.

```
struct LodInfo
{
    unsigned int vertex
    char strips_affected
}

struct StripInfo
{
    unsigned int strip_number
    char num_ops
}

int* UpdateInfo
```

Figure 3.6: *Speed Strips* data structures.

Following the example shown in Figure 3.5, where we presented three levels of detail, the stored information would be that shown in Figure 3.7. As an example, the first change in the level of detail (LOD 1) involves vertex v_0 collapsing to vertex v_2. Thus, the first register of *LODInfo* array will indicate that we are collapsing to vertex v_2 and that this collapse affects two strips. The first two registers of *StripInfo* indicate that we must update strip number one with one change and strips number two with another change. Finally, the first position of the *UpdateInfo* array shows that we must update position 4 in strip number one, while the second position indicates that we must also update position 4 of strip number two.

The *UpdateInfo* array needs a more detailed explanation, as it will store in a condensed and interleaved manner all the information about the changes. The positive values in the *UpdateInfo* array indicate positions where an edge collapse occurs. A negative value indicates a position where some degenerates are found, and the next value indicates how many elements should be removed. If this value is positive, we are facing a vertex repetition ($aa(a)^+$); if it is negative, it is an edge repetition ($ab(ab)^+$). This distinction is necessary for being able to return to higher levels of detail and retrieve the original geometry. We will explain in more detail this process in the following section, where the

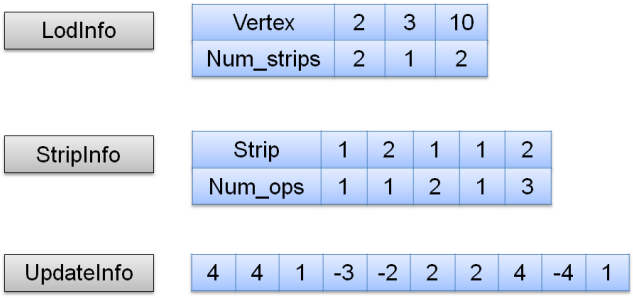


Figure 3.7: Data structure for the example given.

algorithm and a thorough example will be presented.

The objective of maintaining all the information about the changes in only one structure is to avoid having to calculate the order of the different operations in real time, which results in a better performance. Even though we need some operations to distinguish between the different operations that may be applied, this approach increases performance, as the results of the calculations related to what should be done first are obtained in a pre-process.

These data structures are the only information stored in the CPU. Information about the model (vertices, normals, texture coordinates, triangle strips, skeletal animations, etc.) is stored directly on the GPU and only the triangle strips will be modified during the LOD update.

Storage cost

For analyzing the memory requirements of *Speed Strips*, it would be possible to make an estimation of its storage cost. For this estimations we assume that the cost of an *integer* and a *float* is a word (4 bytes), while a *char* can be coded in a single byte. Nevertheless, before addressing the storage needs of the data structures we start by calculating the cost of the original mesh both in triangles and triangle strips.

For storing a triangulated mesh, we need 3 words for the spatial coordinates of the vertices and 3 more to code the indices of each face:

$$3 \cdot v + 3 \cdot f \tag{3.1}$$

where v and f are, respectively, the numbers of vertices and faces in the triangulated mesh. The Euler's formula assures that, for any convex polyhedron, the number of vertices and faces together is exactly two more than the number of edges:

$$v - e + f = 2 \tag{3.2}$$

where e is the numbers of edges in the given polyhedron [104]. In a triangulated mesh displaying manifold topology, every edge is shared by exactly two triangles, and every triangle shares an edge with exactly three neighbouring triangles [2]. Thus, we can assume that the number of edges is:

$$e = \frac{3 \cdot f}{2} \tag{3.3}$$

By using Equations 3.2 and 3.3, we can say that:

$$v - \frac{3 \cdot f}{2} + f = 2 \tag{3.4}$$

$$v - \frac{f}{2} = 2 \tag{3.5}$$

$$f = 2 \cdot (v - 2) \cong 2 \cdot v \tag{3.6}$$

As a consequence, we can conclude that in this kind of meshes the number of faces is approximately twice the number of vertices.

Following with the cost of a triangulated mesh, from equation 3.7 we can conclude that the storage cost in words of a triangularized mesh is $9 \cdot v$, which represents 288 $bits/vertex$.

$$3 \cdot v + 3 \cdot f \cong 3 \cdot v + 3(2v) = 9 \cdot v \tag{3.7}$$

Triangle strips are capable of coding n triangles with $n + 2$ indices. Thus, simplifying the stripification process as if only one strip was output to code the whole geometry of a mesh, we could assume that the storage cost in triangle strips is that presented in Equation 3.8, amounting to $5 \cdot v$ (160 $bits/vertex$).

$$3 \cdot v + (f + 2) \cong 3 \cdot v + (2 \cdot v + 2) \cong 3 \cdot v + 2 \cdot v = 5 \cdot v \tag{3.8}$$

Regarding the data structures of the *Speed Strips* model, each component of the *LODInfo* and *StripInfo* array involves a cost of 1,25 words, while each element of the *UpdateInfo* array only requires 1 word. We will need as many *LODInfo* elements as levels of detail are available. Our tests have proven that, on average, two strips are modified for each level-of-detail update. As a consequence, we will need two times more *StripInfo* elements than LODs available. Finally, each strip that needs update involves, on average, an edge-collapse operation and a repetition elimination, which suppose 3 components of the *UpdateInfo* array for each operation. Summing up, Equations 3.9 to 3.11 present

the cost in words of each of the data structures, considering that a *char* only needs 0.25 words.

$$LODInfo \rightarrow 1{,}25 \cdot v \tag{3.9}$$

$$StripInfo \rightarrow 2 \cdot (1{,}25 \cdot v) \tag{3.10}$$

$$UpdateInfo \rightarrow 3 \cdot 2 \cdot v \tag{3.11}$$

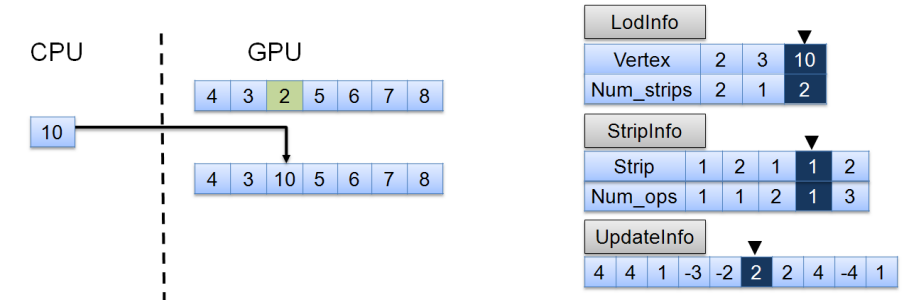
Thus, the total storing cost of a *Speed Strips* mesh is $14{,}5 \cdot v$, which represents 472 $bits/vertex$.

$$3 \cdot v + 2 \cdot v + 1{,}25 \cdot v + 2{,}5 \cdot v + 6 \cdot v = 14{,}75 \cdot v \tag{3.12}$$

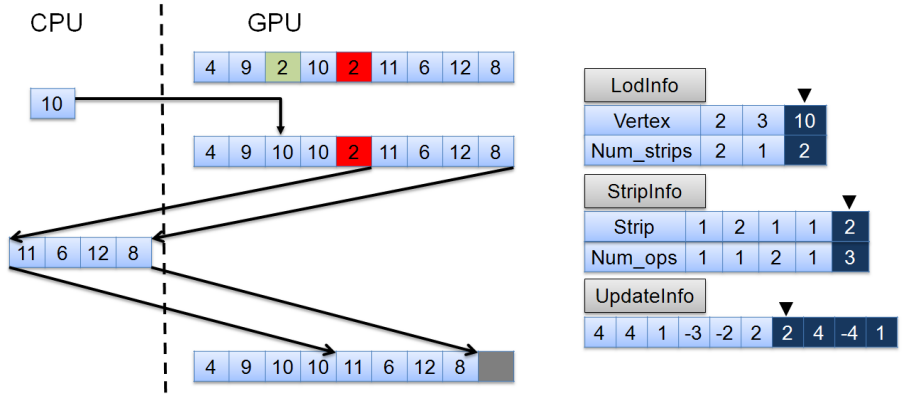
3.3.2. Extraction and visualization algorithms

An important advantage of this multiresolution model is the fact that we do not need to store the strips information in the CPU, as they are only stored on the GPU. Furthermore, the indices of the strips will be the only information modified, as the vertices information remains unaltered throughout the different levels of detail. As a consequence, when a change in the level of detail is needed, most changes will be applied directly and sequentially to the triangle strips stored in the GPU. If we have to deal with the problem of repetitions while moving to either higher or lower levels of detail, we will have to modify the triangle strip length. In such cases, we download the minimum information from the GPU, apply the modifications, and then upload it again to the GPU. This LOD traversing approach enables us to minimize the use of CPU memory and to make an optimum use of the PCI Express bus. Nevertheless, our experiments have proven that, on average, just 40 % of the level-of-detail updates will entail dealing with degenerate triangles.

In Figure 3.8 we offer a diagram of how the last change in the level of detail presented in Figure 3.5 is performed. This figure presents the CPU and GPU memory on the left and the data structures on the right. The cells of the strips shaded in green reflect a strip position that must be modified, while those shaded in red refer to positions that must be eliminated as they contain vertices and edges repetitions. In this example, the *LODInfo* structure shows that we will be collapsing to vertex v_10. As we are changing to LOD 2, we can say that we are changing vertex v_2 to vertex v_10. In addition, the *LODInfo* structure also indicates that this change will affect two triangle strips. In the *StripInfo* structure we can observe that we have to apply one change to strip number one. As the value associated in the *UpdateInfo* array is positive, this strip will only need an index change, and therefore we will not download any information to the CPU. We will simply upload the new vertex to the correct position, minimizing data traffic through the bus. Figure 3.8(b) presents the update of strip number two. The next *StripInfo* element shows that we must



(a) Updating Strip 1.



(b) Updating Strip 2.

Figure 3.8: Example of change to LOD 2 (collapse $v_2 \rightarrow v_{10}$).

apply three changes to this strip. We can apply the first collapse presented in the *UpdateInfo* array. Before we apply the second one, the algorithm realizes that this position is to be eliminated and avoids the unnecessary change. At that moment, we will start downloading the rest of the triangle strip to the CPU while avoiding the elements that should be eliminated. Therefore, once we have the triangle strip completely downloaded to the CPU, we have also processed all the changes and the strip is ready to be uploaded to the GPU again, taking into account the position where the first vertex or edge repetition was found. Thus, we benefit from the coherence as much as possible, as the unmodified triangle strips remain unaltered on the GPU, and we also maintain the maximum information of the strips needed to be updated.

Following the idea presented before, the algorithm for LOD extraction and visualization of the geometry would be similar to the pseudocode presented in Algorithm 1.

Algorithm 1 Pseudocode of the Speed Strips algorithms.

```
// LOD Extraction algorithm.
for LOD = currentLOD to demandedLOD do
  for all Strip in StripsAffected(LOD) do
    // Updating indices directly on the GPU.
    while changes_left && index_change do
      UploadToGPU(vertex);
    end while
    // If there is at least one repetition removal.
    if changes_left then
      auxStrip=0; // Variable for storing the strip temporally in CPU.
      // Updating indices and eliminating degenerates.
      while changes_left && index_change do
        auxStrip+=DownloadAndChangeFromGPU();
      end while
      auxStrip+=DownloadRestFromGPU();
      UploadToGPU(auxStrip); // Copy the updated part of the strip.
    end if
  end for
end for

// Visualization algorithm.
for all Strip in Strips() do
  glDrawElements (Strip);
end for
```

3.4. A memory version

With the intention of testing our approach, we have developed two revisions of the original Speed Strips model presented above. These revisions affect only a minor part of the model, but are interesting for our research purposes and also for the final GPU integration.

As many existing models do not include support for hardware extensions, we have developed a limited memory version which functions on an immediate mode and does not exploit hardware characteristics, but which still presents better results. This way, before resorting to vertex buffer objects, we will be able to prove that our approach is faster than previously existing models. The data structure remains the same and only the algorithms are slightly modified. Instead of directly using the GPU buffers, we will use a fast CPU array to update the strips and also to visualize them. We show the algorithms of this memory version in Algorithm 2.

In order to prove in more detail the performance of our method, we have also extended the memory version to work with vertex buffer objects. We will

Algorithm 2 Pseudocode of the memory version of Speed Strips.

```
// LOD Extraction algorithm.
for LOD = currentLOD to demandedLOD do
  for all Strip in StripsAffected(LOD) do
    // Updating indices directly in the CPU.
    while changes_left && index_change do
      CopyToCPU(vertex);
    end while
    // If there is at least one repetition removal.
    if changes_left then
      auxStrip=0; // Variable for storing the strip temporally in CPU.
      // Updating indices and eliminating degenerates.
      while changes_left && index_change do
        auxStrip+=CopyAndChangeFromCPU();
      end while
      auxStrip+=CopyRestFromCPU();
      CopyToCPU(auxStrip); // Copy the updated part of the strip.
    end if
  end for
end for

// Visualization algorithm.
for all Strip in Strips() do
  if Strip_changed(Strip) then
    for i = 0 to SubMesh sizeofstrip(Strip) do
      glVertex3f ();
    end for
  end if
end for
```

Algorithm 3 Pseudocode of the memory version with hardware support of Speed Strips.

```
// Visualization algorithm.
for all Strip in Strips() do
  if Strip_changed(Strip) then
    UploadToGPU(Strip);
  end if
  glDrawElements (Strip);
end for
```

still apply the changes to the data located in the CPU, but we will upload them to the GPU to take advantage of its higher rendering speed. In Algorithm 3, we also present the visualization algorithm of this new version. The intention of developing this second version is to prove that it is faster to download and upload less information, but more frequently, rather than simply uploading the strips at the end. This is owing to the PCI Express bus characteristics.

3.5. Results

To test the performance of our multiresolution model we will compare its memory cost and rendering time against two models: the Progressive Meshes first introduced in [16] and the memory efficient version of LodStrips with hardware orientation described in [18]. We will also present a small study of the degenerate triangles and stripifying techniques, with the intention of explaining why we selected the previously presented approaches.

The experiments were carried out using Windows XP on a Dell PC with a processor at 2.8 Ghz, 2 GB RAM and an nVidia GeForce 7800 graphics card with 256MB RAM.

3.5.1. Memory cost

Table 3.1 summarizes the characteristics of the polygonal models used in the experiments. In these results the storing cost includes the geometry, but it does not include textures or normals. We have assumed that integers and floats are represented with 4 bytes. This table also presents the storing cost of the different multiresolution approaches studied. The depicted discrete model is based on triangles and will offer 5 different levels of detail, each one reducing the geometry in 25 %. As shown, discrete models offer a very high storage cost as they are obliged to store the geometry of different approximations, increasing the original memory cost in more than 3 times. Regarding the strip-based solutions that are included in this comparison, our model offers more than 40 % improvement over Progressive Meshes and even 5 % over LodStrips with its efficient version, which offered the best memory cost.

It is worth mentioning that the calculations we made for the theoretical memory cost were quite accurate, although the size in triangle strips is slightly higher due to the fact that we are not working with a single-strip stripifier.

3.5.2. Rendering cost

To evaluate the presented model and its variations, we have conducted several linear tests to measure the model performance when a linear sequence of LODs is required [105]. The scene containing the models will lack any kind of illumination or texturing and will just render the mesh. We must consider two important aspects in these tests: the increasing and decreasing order and

Model	Cow	Bunny	Phone	Isis	Buddha
Vertices	2,904	34,834	83,044	187,644	543,644
Faces	5,804	69,451	165,963	375,283	1,085,634
Original (triangles)	289	282	287	288	288
Original (strips)	177	172	175	178	180
Discrete Model	882	860	875	878	878
Progressive Meshes	780	766	794	770	791
LodStrips	496	531	513	522	547
Speed Strips	462	483	475	496	546

Table 3.1: Models used in the experiments, with their storage cost (in bits/vertex.).

Model	Cow	Bunny	Phone	Isis	Buddha
Immediate mode					
Progressive Meshes	0.18	8.44	13.29	29.72	85.56
LodStrips	0.22	3.45	8.66	23.71	77.56
Speed Strips (memory)	0.17	2.13	5.38	15.28	53.24
Hardware acceleration					
LodStrips	0.046	0.31	0.42	1.55	10.04
Speed Strips (memory)	0.037	0.23	0.36	1.45	9.89
Speed Strips	0.036	0.21	0.29	1.21	7.22

Table 3.2: Average rendering time (extraction+visualization) (in ms.).

the difference between the LODs that are consecutive in the sequence (the *step*). The order of the sequence could lead to different test results, due to the fact that some models do not have the same behaviour when refining than when decimating. Therefore, we will consider both approaches in our tests, coarsening the model to the maximum and refining it to retrieve the original geometry. With respect to the step used, we decided to use 0.1 % of the number of available LODs, as this percentage is sufficient to test the performance of the level-of-detail models.

The results of these tests are shown in Table 3.2, where we present results for both with and without applying hardware acceleration techniques. When analyzing the table we can observe how, in its memory version, the Speed Strips model offers an increase of performance of nearly 200 % in relation to the best performance of the previous models. Note that the Progressive Meshes model was not tested with hardware acceleration, as its immediate mode already showed the lowest performance. Among the models presented in the hardware section of the table, both versions of the Speed Strips model improve the rendering time of other solutions. Nevertheless, the memory version only offers

Model	Cow	Bunny	Phone	Isis	Buddha
LodStrips	0.024	0.122	0.146	0.145	0.940
Speed Strips (memory)	0.017	0.072	0.092	0.087	0.530
Speed Strips	0.015	0.041	0.025	0.026	0.132

Table 3.3: Average extraction time of hardware models (in ms.).

Model	Cow	Bunny	Phone	Isis	Buddha
Speed Strips (memory)	2.53	66.64	284.69	317.42	389.57
Speed Strips	0.90	30.04	113.05	145.19	187.61

Table 3.4: Average data traffic (in MB.).

an improvement of 5 % while the version that works directly with the GPU reduces the total time in 30 %.

Due to the fact that the hardware implementations of Speed Strips are based on similar rendering algorithms, Table 3.3 offers the average extraction time only, to show which is the most appropriate way of updating the geometry. To calculate the extraction time we have also included the time spent on updating the data to the GPU. In this table we are also presenting the extraction time of the efficient version of LodStrips, which is much higher than the time needed by both versions of Speed Strips. The results prove that the traversal algorithm which works directly with the GPU is much better than maintaining the strips in the CPU and uploading the affected strips after all changes have been performed. Consequently, on average, we can obtain a 70 % decrease of the extraction time.

With the intention of understanding why the extraction time is reduced, Table 3.4 presents the average data traffic produced when updating the detail in our tests. These data are related to the strip information that is sent to and from the GPU. As the table shows, even though our selected extraction algorithm implies downloading and uploading data to the GPU, the total bus traffic is reduced by nearly 40 % as the coherence is more adequately considered.

3.5.3. Rendering primitive

As we stated before, it is very important for a level-of-detail model to correctly select which will be the final drawing primitive. Different works have considered the rendering speed of different triangle sequences. Bogomjakov and Gotsman [44] proved that cache-optimized triangles considerably improve triangle strips. Later, the work presented in [106] conducted similar tests using different PCs and graphics cards to obtain a more comprehensive analysis. These tests proved that using strips instead of optimized triangles with new

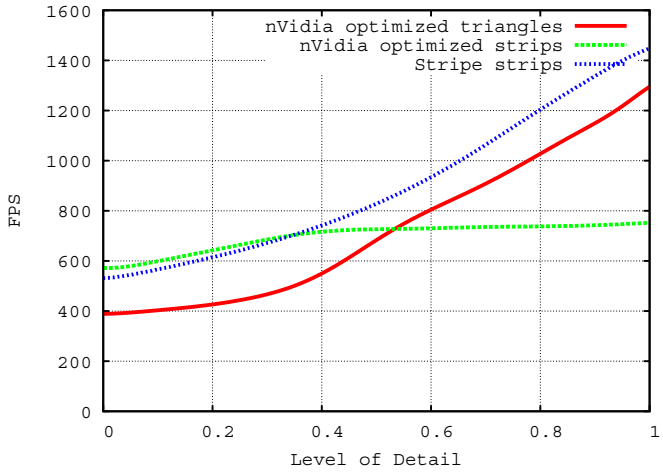


Figure 3.9: Frame rate obtained with triangles and strips when rendering the bunny model at different LODs.

architectures improves performance by at least 100 %. In addition, the memory requirements of strips are also around 30 % lower than in the case of triangles. For these reasons, the model has been oriented towards a strip-based multiresolution model. Nevertheless, in these analyses the authors considered a full-detailed geometry. Thus, it is also important to test the performance of these primitives in a level-of-detail scheme.

Figure 3.9 presents a comparison of the frame rate obtained through all the levels of detail using different approaches. We considered triangles and triangle strips optimized for the cache with the nVidia technique [35], and the strips outputted by Stripe [32].

This graph shows that the resulting performance is always higher if we use Stripe, even though more degenerate triangles are sent. The stripification technique offered by nVidia initially performs better than Stripe, although this optimization is lost after applying a few LOD changes. Moreover, the optimized triangles outperform optimized nVidia strips in nearly half of the levels of detail.

3.6. Conclusions

In this chapter we have presented a new multiresolution model called Speed Strips. It is our intention to use this model as a sound option against traditional

discrete models. Results obtained have shown that our model improves to a great extent previously existing continuous models and presents the desirable characteristics of a multiresolution model: low memory cost, short extraction time and a high frame rate. We have also demonstrated that the approach for deleting degenerate triangles is efficient and sufficient for fast rendering of 3D objects.

The model has been devised with hardware in mind, but always considering that this model must work with consumer hardware. In addition, we have developed different approaches in order to find the best way to work with the current graphics hardware in order to create the most GPU-friendly multiresolution model possible.

The main line for future work is oriented towards being able to maintain our multiresolution model completely on the GPU. Nevertheless, we must reconsider our framework to contemplate the emergence of the Shader Model 4.0, which offers new opportunities for multiresolution modeling. Other lines of future research include data compression and instancing, which we consider as very important features.

CHAPTER 4

Rendering Continuous Level-of-Detail Meshes by Masking Strips

This chapter presents *Masking Strips*, a different multiresolution approach which has been developed for the exploitation of the bit-wise operations introduced in the Shader Model 4.0. The use of masks of bits enables us to code the extraction information in an efficient and compact manner. Moreover, this implementation supposes as step forward in the implementation of a level-of-detail model on the GPU, as its extraction process fits more adequately the way graphics hardware works.

4.1. Introduction

The aim of the different solutions presented in this thesis is to develop a level-of-detail model which exploits graphics hardware as much as possible. The model introduced in the previous chapter presented some interesting improvements over previous solutions, as it included an study of the most suitable way of dealing with data traffic through the PCI bus. Nevertheless, there is still room for improvement as the final objective is the development of a level-of-detail model which works completely on the GPU and totally avoids CPU-GPU traffic, which is one of the bottlenecks of current multiresolution models. This traffic is part of the extraction process, which must be minimized in order to develop a competitive multiresolution model.

Figure 4.1: Space woman model. From left to right: original (4,130 triangles), 60 % (2,478 triangles) and 20 % (826 triangles) approximations.

The main problem with existing multiresolution models is that, although they are well designed and some of them make use of hardware buffers, it proves very difficult to adapt them to the new GPU architectures due to their complex processes. The model we present in this chapter has been developed as an improvement over previous strip-based models. The main motivation behind its development is the possibility of performing bit-wise operations on the GPU, which has been possible with the appearance of the Shader Model 4.0. The use of masks of bits allows us to code the extraction information in an efficient and compact manner. In addition, the limitation of previous solutions related to the treatment of degenerate triangles is overcome, and our proposal is capable of eliminating all unnecessary degenerate information. These solutions were not capable of eliminating all the unnecessary information and, in addition, they were obliged to use complex data structures and extraction algorithms to manage them, involving higher spatial and temporal costs.

This multiresolution model has been designed to consider meshes produced by CAD applications with several submeshes and different vertex attributes like normals, texture coordinates and bones for skeletal animations. Figure 4.1 presents an example of a space woman model rendered at three levels of detail using our algorithm.

It is our objective to improve on our previous solution by developing a new multiresolution model which presents the following features:

- *Hardware optimization.* The model is rendered using primitives with implicit connectivity, triangle strips, which, in addition, are optimized for the vertex cache. Mesh data is stored statically in the GPU, and only when it is necessary to update the LOD these data are modified. Moreover, only the affected strips are updated and sent to the GPU, maintaining coherence.

- *Masking for LOD management.* The use of masks of bits allows for a reduction of the memory cost and for a more adequate hardware orientation.
- *Complete degenerate removal.* A new strategy for eliminating all the unnecessary degenerate triangles of the triangle strips is introduced.
- *Progressive transmission.* Even though the model has been developed for its interactive visualization, it includes a progressive transmission scheme that allows us to reduce the storage cost and enables a progressive loading of the mesh.

This chapter is organized as follows. First, we introduce the main characteristics of the model and we provide its general framework. At this point, further details of the proposed method are presented, including data structures, algorithms and the progressive transmission details. Then, a study of storing and rendering costs is detailed, including a comparison of our solution against previous level-of-detail models. To conclude, we briefly comment on the results obtained.

4.2. General Framework

When introducing the model presented in the previous chapter, we described a set of tasks that were necessary to prepare the multiresolution model. Thus, as it happened with Speed Strips, for constructing a Masking Strips model it is necessary to apply a simplification process to the original mesh and also to obtain an adequate set of triangle strips. Both sets of data will be combined in the construction process to prepare the necessary data structures. These three different processes are executed in a pre-process step.

On the one hand, as the simplification algorithm we use the one that will be described in the following sub-section, which deals correctly with the existence of submeshes in the original 3D model and preserves the original shape to a large extent, avoiding the appearance of artifacts and holes and offering a high visual quality.

On the other hand, among the existing stripifiers, the short triangle strips that can be obtained with the cache-aware approaches are more suitable for our masking model. Furthermore, we use a static stripification combined with a correct degenerates treatment, avoiding the need to re-stripify the mesh each time we change the level of detail. This approach allows us to obtain a higher performance, as we will show in the following sections.

In Figure 4.2 we present a diagram of the basic operation of our model. The information for managing the level-of-detail is stored in CPU memory. The CPU maintains also a copy of the strips at the level of detail that we are rendering. On the GPU side we store the vertices, normals, textures, and any other attribute. The GPU memory stores also a filtered version of the strips,

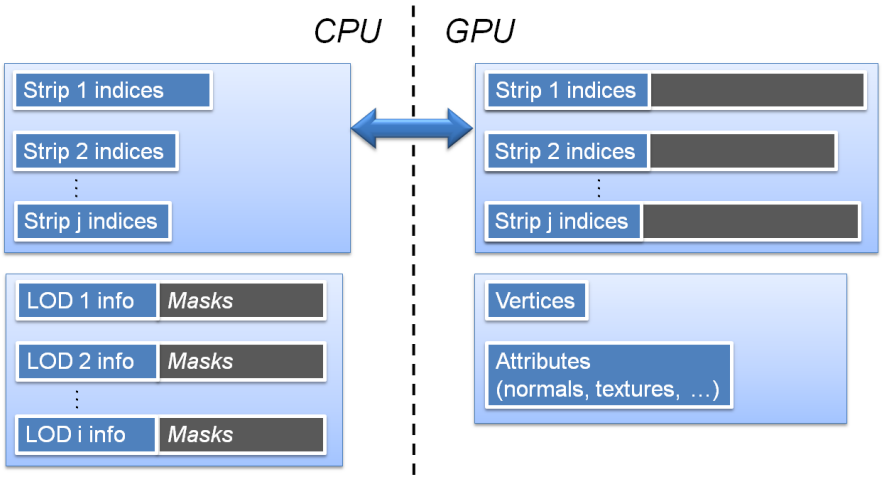


Figure 4.2: Basic framework of Masking Strips.

obtained after the removal of all unnecessary degenerate triangles. Thus, the CPU keeps an updated and full-sized version of the strips, while the GPU stores the filtered strips for fast rendering.

4.2.1. Simplification of the original mesh

The simplification algorithms presented in the state of the art in Section 2 offer very accurate simplifications. The main issue is the fact that they are not completely aimed at working with the meshes that final applications actually use. As it has been mentioned in the introduction, meshes are often composed of multiple attributes per vertex, as for example the different normals that are necessary for modeling surface discontinuities (sharp edges). This happens also with meshes composed of different submeshes, where border vertices share the same location in space but need different attributes.

A convenient way to manage these multiple per-vertex attributes is to assign them to a wedge [49]. A wedge is a set of vertex-adjacent corners whose attributes are the same. However, this representation is not adequate to be optimally implemented on the graphics hardware. The necessity of offering a simple and hardware-oriented vertex structure forces us to repeat each vertex for each different combination of attributes. This technique is commonly used in the area of computer games, because it is the most optimal for the underlying hardware. Nevertheless, this kind of representation introduces invisible holes in the mesh due to the vertex separation, posing a problem for many simplification methods.

The simplification algorithm selected for our proposed multiresolution model is the one presented in [24]. We could use any kind of metric to determine

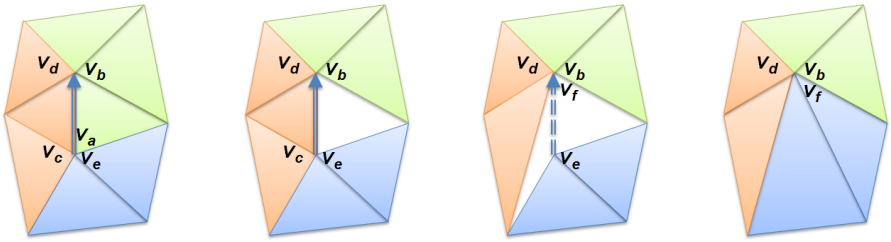


Figure 4.3: The edge collapse $v_a \rightarrow v_b$ (true edge) forces the collapses $v_c \rightarrow v_d$ (twin edge) and $v_e \rightarrow v_f$ (fake edge).

which edges should be collapsed, although we decided to apply the image-based metric presented in [29]. For explaining the basic ideas behind this simplification strategy, Figure 4.3 represents a simplification step in a border among three different submeshes. Each of these submeshs has a different material (depicted as a different color) and, as a consequence, some vertices have been originally duplicated to allow a correct rendering of the geometry. For example, vertices v_a, v_c, v_e represent the same spatial coordinates but different attributes.

This proposed simplification strategy uses the concept of *true edge*, *twin edge* and *fake edge*. The *true edge* refers to the collapsed edge, while the *twin edge* is the one composed of vertices with the same spatial location that the vertices of the *true edge*. Thus, in the example we are presenting, the edge v_a, v_b is chosen for a collapse (it is the *true edge*). The simplifier decides that edge v_c, v_d is a *twin edge* of a different submesh, and that it must be collapsed in order to avoid a hole.

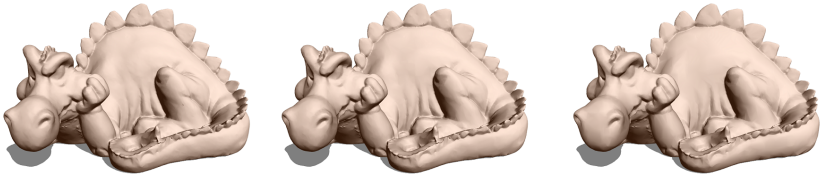
Contracting a *true edge* and its *twin edge* is not always sufficient to avoid holes in these meshes. The second image presented in Figure 4.3 shows how, after contracting the edges, there is still a visible hole. The simplification algorithm may decide to collapse the isolate vertex v_e with any of the vertices used in the other collapses (v_b or v_d). Nevertheless, although this collapse would fill the hole, it would not allow vertex v_e to maintain its appropriate attributes. This is the reason why the isolate vertex (v_e) must be connected to a new vertex (*fake vertex*) creating a *fake edge*. This *fake vertex* will be used as a collapse destination to avoid cracks in the mesh while maintaining the proper set of attributes. When adding the new *fake vertex*, its location coordinates and its bone assignments are cloned from the proper vertex of the *true edge*. Other vertex attributes, like texture coordinates and normals, can be calculated as desired to improve the quality of the simplified model.

Once the simplification process is finished, the output will be composed of a list of edge collapses that can be used by any multiresolution model based on edge-collapses to obtain the desired level of detail. It is important to note that in order to avoid holes in the run time of the LOD algorithm, it is necessary to apply all the simplification steps related to an edge collapse at the same time,

(a) Ogre model. From left to right: original (1,960 triangles), 66 % (1,292 triangles) and 33 % (646 triangles) version.

(b) Buggy model. From left to right: original (91,949 triangles), 66 % (60,686 triangles) and 33 % (30,356 triangles) version.

(c) Racing car model. From left to right: original (8,345 triangles), 66 % (5506 triangles) and 33 % (2,753 triangles) version.



(d) Phlegmatic Dragon. From left to right: original (480,044 triangles), 66 % (284,822 triangles) and 33 % (146,386 triangles) version.

Figure 4.4: Three levels of detail of different 3D models.

which include the *true edge*, its *twin edge* and any *fake edge* that might appear.

Figure 4.4 presents several levels of detail of different meshes. These images show the visual quality obtained with the simplification method described. Thus, when applied to our multiresolution model, this simplification technique respects textures to a high extent avoiding cracks and holes.

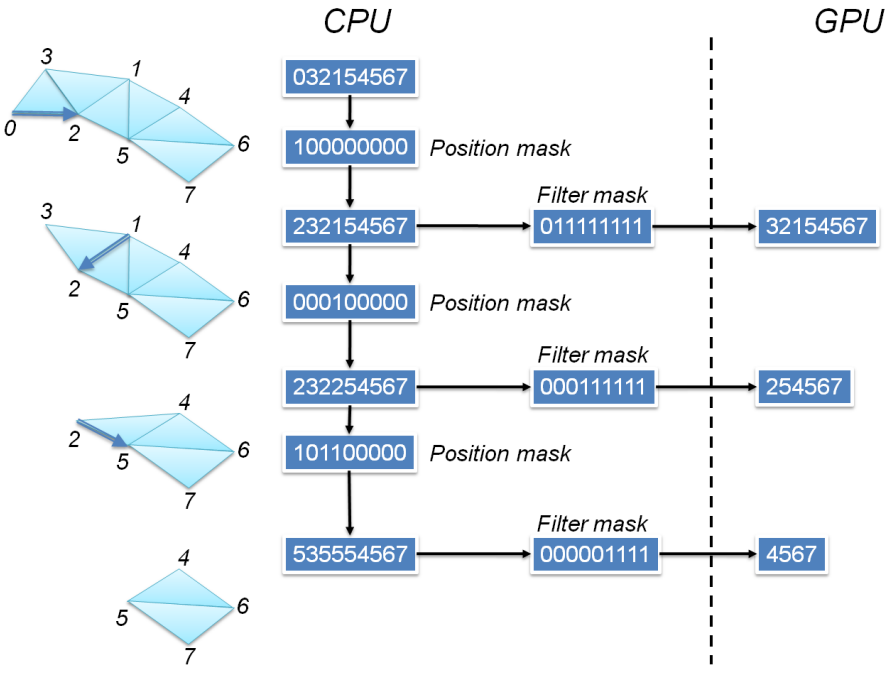


Figure 4.5: Masking example.

4.2.2. Masking for LOD management

The third basic aspect of this multiresolution scheme is the use of masks for codifying the positions where edge collapses happens and also where degenerate triangles are located. The fact that the selected stripifier usually outputs small triangle strips makes this masking approach efficient for storage and processing. Our model calculates these masks during the construction time and stores them in an efficient data structure.

The method we propose is based on the use of two types of masks: position and filter masks. An example of these masks is introduced in Figure 4.5, presenting an original triangle strip on top and its evolution after three edge collapses. For each simplification of the initial strip, the figure includes the resulting strip, the masks and the indices of the strip that are stored both in the CPU and in the GPU. The position masks indicate which indices of the strip must be updated to reflect the changes in geometry. These changes are applied to the indices stored in the CPU. Once the indices have been correctly modified, the filter mask indicates which positions must be rendered. These filters are considered when uploading the modified strip to the GPU. Thus, the copy in the CPU maintains the full updated geometry without eliminating degenerate information. This approach allows us to return to higher levels of

detail with the same position and filters masks. A further benefit of our masking approach is the fact that there is no dependency between filter masks. As a consequence, we can apply several position masks but it is only necessary to apply the last filter mask to eliminate the corresponding degenerate triangles. Following with the given example, if we wanted to traverse the three levels-of-detail at a time, we would need to apply the three position masks but only the last filter mask, which would be sufficient for obtaining the correct and filtered geometry.

Eliminating degenerate triangles

A complete level-of-detail method should include an efficient treatment of degenerate triangles. Among the different possibilities for solving this problem, authors usually resort to the application of filters following different simple patterns. As we commented in the previous chapter, some authors prefer to calculate the filters *on-the-fly* [52] while other works calculate the filters in a preprocess [18].

Nevertheless, the above-mentioned models still present some unnecessary information in the final rendered geometry as they apply simple filters. Our experiments have proven that these approaches can eliminate up to a 75 % of the total number of degenerate triangles. This is due to the fact that some degenerate triangles are difficult to eliminate, and processing them would entail storing a high amount of information to be able to return to higher levels of detail.

In the approach we are presenting we are able to remove all the unneeded degenerate triangles. The main difference is that we do not look for specific patterns of degenerate triangles. By contrast, we try to connect two meaningful triangles while removing all the unnecessary information between them. More precisely, we look for a significant triangle, locate the next significant one, and connect them in the most appropriate way.

Figure 4.6 presents the main patterns that we follow. Each pattern presents two correct triangles which are separated by degenerate ones. For example, the first pattern is related to a triangle strip where two triangles that share an edge (*abc* and *bcd*) are separated by several degenerate triangles. In this case, we would be able to eliminate all those degenerate triangles to obtain the correct geometry (*abcd*). Obviously, this is the best case, as we are able to render 2 triangles with just 4 indices. The patterns have been ordered according to their importance, as the first ones involve rendering fewer indices than the last ones. Moreover, all degenerate triangles located before the first meaningful triangle and after the last one are also eliminated.

It is important to note that the feature that allows us to return to higher levels of detail is the fact that on the CPU we store the triangle strips without eliminating degenerate triangles. This way, the filtering process is only applied when copying the updated strips to the GPU. Thus, the data structures and

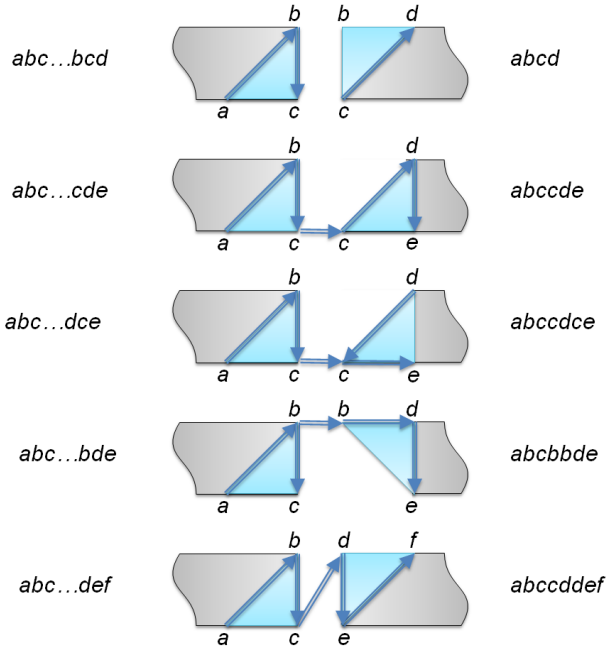


Figure 4.6: Patterns used for constructing the filter masks.

the extraction process can be consequently simplified.

4.3. The Masking Strips model

An important advantage of our Masking Strips is the way the information for updating the level of detail is stored. As we have coded all the information in masks of bits, every strip that must be updated only needs a mask with the position changes and another one with the degenerate triangles to eliminate. The vertices that are not modified and can therefore be uploaded to the most optimal data location.

4.3.1. Data structures

The information for modifying the detail of the polygonal model is stored in two data structures: *InfoUpdate* and *InfoStrip*. We consider that every edge collapse operation supposes a change in the level of detail. Thus, we offer as many levels of detail as edge collapses have been recorded in the simplification process. Figure 4.7 contains these data structures in a *c-like* notation.

The structure *InfoUpdate* indicates how an edge collapse operation is reflected in the triangle strips. This data structure includes which vertex is involved in the collapse and also how many strips are affected by this update. For optimizing this data structure, we re-order the vertices so that the vertex that disappears coincides with the number of LOD we are changing to. With this ordering, only the vertex that survives is stored. Moreover, *InfoUpdate* also contains a field called *next_update*, which is necessary for preserving the correctness of the mesh. As we stated in a previous section, *true edges*, *twin edges* and *fake edges* must be collapsed together to assure that all the vertices related to the same spatial position are simplified in the same way and at the same time. The value stored in *next_update* indicates whether we must apply the operations of the next *InfoUpdate* register.

Once *InfoUpdate* has exactly indicated which strips are affected, we must know how to modify each of them. This information includes which strip is affected and the two masks for updating the strip. The field for the submesh is not necessary, as all the strips of the mesh have been numerated in a correlative manner. For storing the arrays of bits, we use chars, and adjust the number of chars to the size of the triangle strip. The data structure *InfoStrip* contains all these necessary data.

```
struct InfoUpdate
{
    unsigned int vertex_number
    char strips_affected
    bool next_update
}

struct InfoStrip
{
    unsigned int strip_number
    char* position_mask
    char* filter_mask
}
```

Figure 4.7: Masking Strips data structures.

It would be interesting to make an estimation of the storing cost of these data structures. Let us suppose that the cost of an integer and a float is a word (4 bytes). In these estimations we just consider spatial positions for each vertex. For storing a mesh in triangles we would need 3 words for each vertex and 3 more for each face. As we showed in the previous chapter, we can assume that the number of faces is approximately twice the number of vertices and, as a consequence, we can conclude that the cost in words of a triangularized mesh

is $9 \cdot v$. In addition, we also studied the cost of a mesh in triangle strips, which suppose $5 \cdot v$ words, which is 55 % of the cost in triangles.

For approximating the cost of storing the different data structures presented before, we consider that the number of edge collapses is of the same order of the number of vertices. Furthermore, our tests have proven that one edge collapse affects an average number of three strips. With all this information we can assume that we need as many *InfoUpdate* registers as collapses, and three times more *InfoStrip* registers. Equations 4.1 and 4.2 present the memory cost of the data structures, considering that a word is enough for storing the masks and that we only need 1 byte (0.25 words) for a char.

$$InfoUpdate \rightarrow 1{,}25 \cdot v \tag{4.1}$$

$$InfoStrip \rightarrow 3 \cdot (2 \cdot v) = 2 \cdot v \tag{4.2}$$

The total size in words of our model, both the stripified mesh and the data structures, is $12{,}25 \cdot v$, which is 392 $bits/vertex$ (Eq. 4.3).

$$5 \cdot v + 1{,}25 \cdot v + 6 \cdot v = 12{,}25 \cdot v \tag{4.3}$$

This value is just 35 % higher than the size in triangles, which was $9 \cdot v$ (288 $bits/vertex$). If we consider normals in this estimation, the storage cost of the mesh in triangles and strips would be, respectively, $12 \cdot v$ (384 $bits/vertex$) and $8 \cdot v$ (256 $bits/vertex$). In this case, the cost of the Masking Strips model would be $15{,}25 \cdot v$ (488 $bits/vertex$), which suppose only a 20 % increase if compared with the triangularized mesh with normals.

4.3.2. Extraction and visualization algorithms

The algorithms presented in Algorithm 4 provide a simplified but complete version of the extraction and visualization process. It is important to note the existence of *aux_strip*, which is useful to construct the filtered version of the strips in the CPU. This version is the one uploaded to the GPU and also the one that is finally rendered. Besides, the *UseStrip* method is introduced for indicating when a strip is no longer necessary. As we reduce the level of detail, it is possible that all the edges of a strip collapse. For that reason, *UseStrip* is necessary to indicate if a strip is useful in a precise LOD. Finally, for obtaining the correct results, we must consider the existence of submeshes when uploading and rendering the triangle strips.

4.3.3. Progressive transmission

The design of our multiresolution model allows us to perform a progressive representation of the meshes. This feature can be exploited for transmitting

Algorithm 4 Pseudocode of the Masking Strips algorithms.

```
// LOD Extraction algorithm.
for LOD = currentLOD to demandedLOD do
  for all Strip in StripsAffected(LOD) do
    if UseStrip(Strip, LOD) then
      Strip=apply_mask(position_mask,Strip);
    end if
  end for
end for
for all Strip in ModifiedStrip() do
  if UseStrip(Strip, LOD) then
    aux_strip=apply_mask(filter_mask,Strip);
    UploadToGPU(aux_strip);
  end if
end for

// Visualization algorithm.
for all SubMesh in SubMeshes() do
  for all Strip in SubMesh do
    if UseStrip(Strip, LOD) then
      glDrawElements (Strip);
    end if
  end for
end for
```

geometry, but also for a smart storing of the model in the hard drive. Our progressive transmission approach is composed of only three types of packages, presented in Figure 4.8: the *InitialPackage*, the *UpdatePackage* and the *StripPackage*.

The *InitialPackage* stores all the information that is necessary for allocating memory and for the further refinement of the model. It includes the number of strips, the number of vertices, an array with the sizes of the strips, and also the initial vertex. This package allows us to construct the data structures and offers the coarsest approximation.

Every *UpdatePackage* refines the previous approximation. This package includes the new vertex, the number of strips that are affected by this geometry change and the *next_update* field which is necessary to indicate whether it should be applied the next simplification step.

Each of the strips that must be updated for a level of detail change needs a *StripPackage*. This package includes the arrays of bits for positions. The triangle strip to modify can be found out with the contents of the masks. Furthermore, due to the small size of the strips we work with, we can consider that the filter calculation can be performed when receiving the mesh, saving storage needs.

```
struct InitialPackage
{
    unsigned int vertex_number
    unsigned int strips_number
    unsigned int* strips_size
    vertex initial_vertex
}

struct UpdatePackage
{
    vertex new_vertex
    char strips_affected
    bool next_update
}

struct StripPackage
{
  char* position_mask
}
```

Figure 4.8: Progressive transmission data structures.

With these three data structures, the process of reconstructing the original mesh involves:

- processing the *InitialPackage* to create the arrays that will contain the information on vertices and indices for the triangle strips. The information about the initial vertex will be replicated throughout the arrays as if the mesh was composed of a single vertex.

- processing each *UpdatePackage* adds a new vertex and indicates the number of *StripPackages* to use. These latter packages contain the position masks that are needed to indicate which position should be updated with the new vertex.

By repeating the last step the process will expand in memory the geometry of the original mesh. During this process, it will be necessary to calculate the filter masks that will be necessary for the correct performance of the multiresolution model.

These data structures allow us to obtain a reduction in the memory cost. As we did in the previous section, Equations 4.4 to 4.6 present an approximation to the total storing needs of the different packages, considering again only spatial coordinates for the vertices. The cost of the *InitialPackage* is simplified to the

cost of the array of strip sizes. Estimating the number of triangle strips is not simple and, as a consequence, we have simply considered that we will have v strips. We will have as many *UpdatePackages* as levels-of-detail available. Moreover, regarding the cost of all the *StripPackages*, we have considered again that three strips are affected on average in a simplification step.

$$InitialPackage \rightarrow 5 + v \cong v \tag{4.4}$$

$$UpdatePackage \rightarrow 3{,}25 \cdot v \tag{4.5}$$

$$StripPackage \rightarrow 3 \cdot v \tag{4.6}$$

$$v + 3{,}25 \cdot v + 3 \cdot v = 7{,}25 \cdot v \tag{4.7}$$

Thus, Equation 4.7 presents the total cost in words of a progressive version of our Masking Strips, which is $7{,}25v$ ($232\ bits/vertex$). This value reduces the storage cost of Masking Strips without progressive transmission to 70 %. In addition, this progressive version has a storage cost that is lower than the original cost in triangles and supposes only 45 % increase with respect to the cost of the original model in triangle strips.

If we wanted to analyze the cost with normals, the progressive version of Masking Strips would have a cost in words of $10{,}25 \cdot v$ (328 bits/vertex), supposing only two times the cost of the original model in strips.

4.4. Results

To test the performance of our multiresolution model, we have studied the spatial cost and also the rendering time. The experiments were carried out using Windows XP on a Dell PC with a processor at 2.8 Ghz, 2 GB RAM and an nVidia GeForce 7800 graphics card with 256MB RAM. Table 4.1 offers a description of the models used in the different experiments performed.

It is important to mention that, in these examples, we use the triangle strips obtained with the method included in the NVTriStrip library, devised by nVidia [35]. These short triangle strips are more adequate for our proposal and offer a fast rendering.

Model	Ogre	Racing	Cessna	Bunny	Dragon	Buddha
Vertices	1,645	5,195	6,795	34,834	240,028	543,644
Faces	1,960	8,345	13,546	69,451	480,044	1,085,634
Submeshes	3	2	1	1	1	1
Textures	yes	yes	no	no	no	no
Bones	yes	no	no	no	no	no

Table 4.1: Detailed information of the models used in the experiments.

Model	Ogre	Racing	Cessna	Bunny	Dragon	Buddha
Original (triangles)	306	364	384	384	384	383
Original (strips)	251	264	289	268	284	276
Progressive Meshes	1,106	1,060	1,124	959	955	961
LodStrips	567	643	788	611	618	643
Speed Strips	497	506	608	594	613	613
Masking Strips	465	460	486	502	500	497

Table 4.2: Storage cost study (in bits/vertex).

Model	Ogre	Racing	Cessna	Bunny	Dragon	Buddha
Progressive Meshes	543	510	517	448	446	449
Masking Strips	299	301	324	328	321	331

Table 4.3: Storage cost study of the progressive solutions (in bits/vertex).

4.4.1. Memory cost

For analyzing the memory cost of Masking Strips, Table 4.2 summarizes the costs of the different models. We have also calculated the cost of the meshes in triangles and strips. In addition, we compare our solution against three models: Progressive Meshes (PM), firstly introduced in [16], the LodStrips model presented in [18] and the Speed Strips model presented in the previous chapter. It is important to note that, in these results, all the costs have been calculated without textures or bones information, in order to assure that the different solutions are compared on equal terms.

In this table it can be seen how LodStrips improved on Progressive Meshes, and also how our model reduces the cost of Speed Strips to less than 85 %. With the estimations of the storage needs we made in previous sections, we obtained that the cost was 488 $bits/vertex$ for Masking Strips and 328 $bits/vertex$ for its progressive structures. The results in Table 4.2 show how the experimental costs are very similar to the costs calculated in previous sections. The deviations in the first two models are due to the fact that our simplification scheme needed to add some extra vertices.

From a different perspective, in Table 4.3 it is presented the cost for the progressive transmission version. Once again, our model offers an important improvement, reducing the needs of Progressive Meshes in nearly a 30 %.

4.4.2. Rendering cost

The performance of our model has been analyzed from two different perspectives. On the one hand, Figure 4.9 presents a study of the indices rendered throughout the different levels of detail. The strips we render always involve

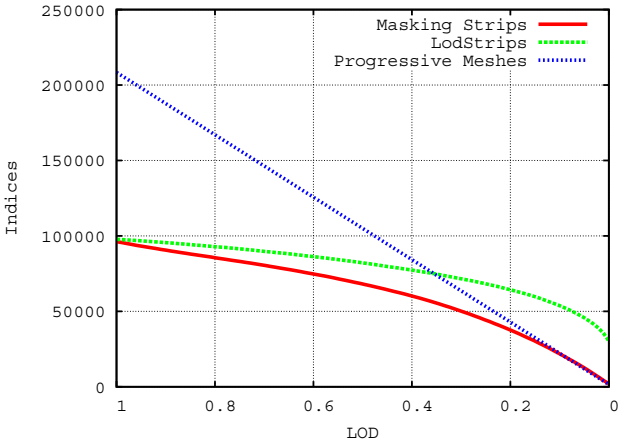


Figure 4.9: Indices rendered for the bunny model.

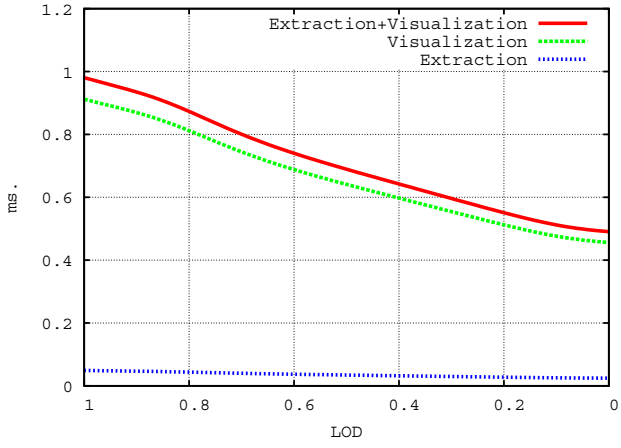


Figure 4.10: Comparison of the extraction and visualization times of the phlegmatic dragon model.

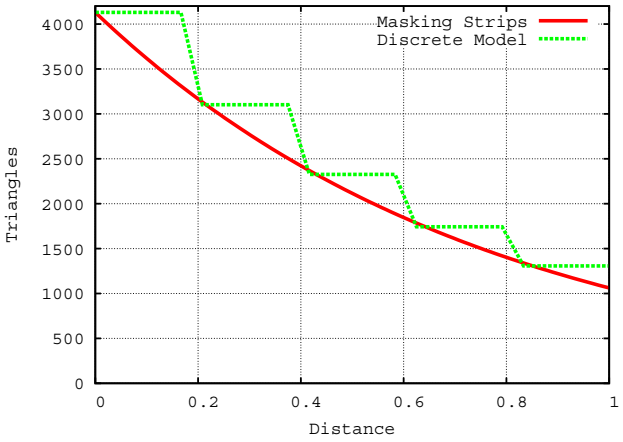


Figure 4.11: Comparison of the triangles rendered for the space woman model at different distances.

less indices than a triangle-based approach like Progressive Meshes. This is due to the fact that we eliminate all the unnecessary information, improving on previous multiresolution models like LodStrips which offered bad results in low levels of detail. Thus, in the coarsest approximation we can reduce the indices processed by LodStrips or Speed Strips in more than 40 %. Furthermore, we want to underline that the indices we send when updating the level of detail supposes always a very small percentage of the total rendered, as we only upload the modified strips and the rest are kept unaltered in the GPU. This is an important advantage against models which have to upload all the geometry when performing a level-of-detail update, like those models which re-stripify the mesh.

On the other hand, Figure 4.10 includes the extraction and visualization time of the phlegmatic dragon model. This figure shows how the extraction process of our model is very fast, supposing less than a 5 % of the total time. This is a compulsory feature for developing a competitive multiresolution model, and is due to the way we update and upload the strips, sending the minimum information through the bus.

In order to test more thoroughly the performance of our multiresolution model, we have prepared a real scenario with a crowd of animated models. This scenario has been built using Ogre [107], an open source graphics engine. Consequently, our level-of-detail approach has been implemented in this engine to test it against the discrete model offered by the engine. The discrete models created for this test are rendered using triangle strips and offer 5 different

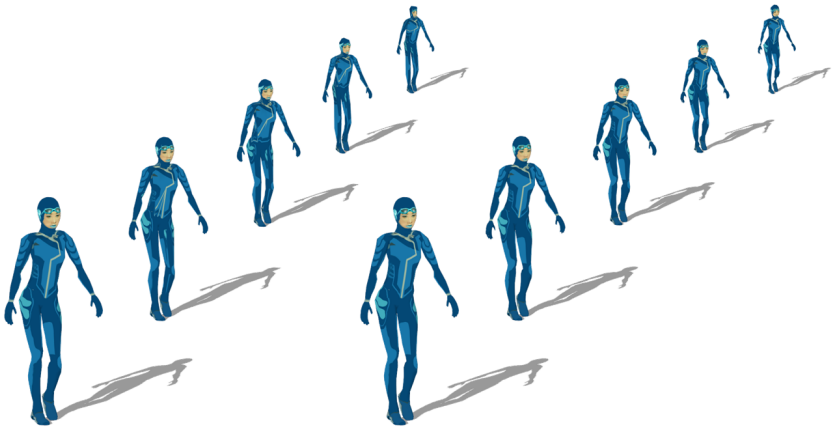


Figure 4.12: Space woman at different distances and levels of detail. On the left we present the models rendered with a discrete solution, while on the right we show the results of the Masking Strips approach.

Figure 4.13: Scene rendering a crowd of models developed inside the Ogre graphics engine.

approximations, each of them reducing the geometry in 25 %. The criterion used to change the level of detail has been the distance to the viewer, and the values used for both models have been the same ones. Figure 4.11 shows the number of polygons rendered at different distances, considering 0 as the closest point and 1 the farthest one. The distance values used to select the level of detail have been adjusted so that at those distances where the discrete model changes to a different level-of-detail, our multiresolution model extracts a level-of-detail with a similar polygonal complexity. Figure 4.12 visually compares both multiresolution models at similar distances to show how, throughout the different levels of detail, our model is able to reduce the rendered geometry while maintaining the visual quality.

The scene rendered is depicted in Figure 4.13. This scene is composed of

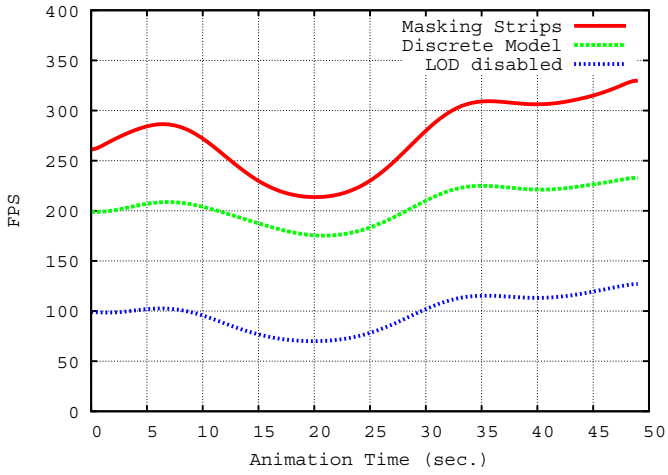


Figure 4.14: Performance obtained in the crowded scenario using our multiresolution model, the discrete solution included in Ogre and disabling any level-of-detail solution.

250 animated space woman models, amounting to a total number of 1,032,500 triangles. The test consisted in traversing the same random path during 50 seconds. Along this path, the number of models that are visualized and the level of detail at which they are rendered varies. As a consequence, the frame rate fluctuates in accordance to the final amount of geometry that is visualized. The results of this test are shown in Figure 4.14, where we include the frame-rate obtained for our model, for the discrete solution and also the performance that we could obtain when no level-of-detail method is applied. Both multiresolution models considerably increase the performance obtained when the whole geometry is rendered. Moreover, our model improves the results obtained with the discrete model, reducing the rendering time in 30 %. These results are possible due to the fact that the time required for the extraction process is very low. Moreover, our multiresolution model extracts less geometry, which also diminishes the skinning and the shadowing calculations. In addition to a better performance, our continuous model is capable of offering a higher visual quality as the model decreases and increases the detail of the models smoothly.

4.5. Conclusions

This chapter introduces Masking Strips, a multiresolution framework for interactive visualization which presents the use of masks for storing the data reducing storing needs and, combined with our filtering approach, enabling a fast and efficient extraction process. In addition, the data structures presented allow for a progressive representation of the model.

Our experiments have proven that the model offers interesting results in storing cost and also in extraction and rendering times. It is important to note that it is possible to find more advanced techniques, like the patch-based ones [67]. Nevertheless, our main aim is to offer an easy-to-integrate solution which is suitable to any kind of architecture where it is necessary to minimize data traffic. In addition, we also offer an interactive visualization, better costs, coherence exploitation and a very short extraction time, which are key for the use of continuous multiresolution models in computer games and other real-time applications.

As commented in the introduction, it is our interest to develop a model which is fully integrated in the GPU. Thus, the data structure based in mask of bits fits perfectly inside the possibilities offered with the recent Shader Model 4.0, and encourages us to adapt this level-of-detail framework to work directly in the graphics hardware.

CHAPTER 5

Interactive Visualization of Meshes on the GPU

The multiresolution approaches presented in previous sections have been oriented towards the exploitation of graphics hardware. Nevertheless, although they present very good results, they are still based on complex data structures and algorithms which are difficult to adapt to the graphics pipeline to perform the level-of-detail extraction process completely on the GPU.

In this chapter we present a new level-of-detail scheme based on triangles which is simple and efficient. In this approach the extraction process updates indices instead of vertices. This feature provides a perfect framework for adapting the algorithms to work on GPU *shaders*. One of the key aspects of our proposal is the need for only a single rendering pass to obtain the desired geometry. Moreover, coherence among the different approximations is maximized by means of a symmetric extraction algorithm, which performs the same process both when refining and coarsening the mesh. We also introduce different uses of the scheme to offer both continuous and view-dependent resolution. Lastly, we propose the integration of our scheme into a commercial application to show how our solution is easy to integrate and how it minimizes the tasks that the user needs to perform in order to prepare and use the multiresolution model.

5.1. Introduction

The development of Shader Model 4.0 was a breakthrough in computer graphics as it offers a new range of functionalities [108]. The main contribu-

Figure 5.1: Approximations of a man model (136,410 triangles). From left to right: original model and simplifications to 50 %, 25 % and 10 % respectively.

tion is the *geometry shader*, which establishes a new stage inside the graphics pipeline enabling the dynamic creation and elimination of geometry in the GPU. Furthermore, it also offers the possibility of modifying the flow of information by means of the technique known as *Transform Feedback* in OpenGL and *Stream Output* in DirectX.

We decided to exploit these features in order to offer a fully-GPU implementation of the multiresolution model we are presenting. The main objective of this chapter is to describe a new multiresolution framework for real-time rendering of arbitrary meshes which contributes to diminish the existing distance between a multiresolution GPU-based solution and its implementation in any 3D application.

In this chapter we present *Interactive Meshes*, a multiresolution scheme whose simplicity enables us to perform the level-of-detail management completely on the GPU. Moreover, coherence among extracted levels of detail is maintained, reusing the information and improving the final performance. The memory cost is optimized as we need very little information to perform the LOD extraction, which is based on half-edge collapses.

The proposed model offers a wide range of new possibilities as the model is completely integrated in the GPU and no CPU/GPU communication is needed once all the information is correctly loaded in hardware memory. The framework we are presenting offers a very promising extraction process which can be combined with different solutions to enhance the results. Thus, given the possibilities offered by the aforementioned framework, we present two approaches. On the one hand, we describe a continuous resolution implementation to introduce the basics of this framework. On the other, we improve this solution to offer view-dependent resolution, being the GPU on charge of selecting the optimal level-of-detail to be extracted for the particular viewing conditions. Figure 5.1 presents different visualizations of a man model at different levels of

detail using a silhouette-based extraction algorithm.

More precisely, our new approach includes the following main features:

- a simple data structure based on vertex hierarchies adapted to the GPU architecture. The vertex hierarchy is given through the edge contraction operations of the simplification process [20]. This data structure permits offering a storage cost with very low memory requirements.
- an extraction process based on the update of vertex coordinates to reflect LOD changes, in contrast to traditional models which have always updated the information related to the indices. This approach integrates well with other pixel-based methods like sub-surface scattering or parallax occlusion mapping. In addition, this extraction algorithm is symmetric, being capable of increasing or decreasing the detail of the mesh with the same process.
- representations are stored and processed entirely in the GPU avoiding the typical bottleneck in the CPU-GPU bound and thus obtaining a great performance by exploiting the implicit parallelism existing in current GPUs. The extraction process exploits the features offered by Shader Model 4.0, as we will see throughout the different sections.
- view-dependent capabilities, which are possible by means of a slight modification of the continuous resolution algorithms but with the same data structures.
- only one pipeline pass is required to adapt the geometry to the needs of the application. Moreover, this pass is able to maintain coherence among the extracted approximations when refining and coarsening the mesh.
- ease of integration of the whole solution, which can be described with two shaders and very little scripting.

This chapter has the following structure. Section 2 presents the approach for continuous resolutions, describing the basics of our extraction process that will also be applied for the view-dependent model. Section 3 provides thorough details of the implementation of the view-dependent model, making a special effort to describe the differences between them. In Section 4 we outline the integration of Interactive Meshes in a real application. Section 5 includes a comparative study of spatial costs and rendering times. Lastly, Section 6 concludes this chapter by commenting on the results obtained and outlining future lines of work.

5.2. Continuous resolution framework

As we have already commented, the main idea of our framework is to offer a fast and efficient extraction process, modifying the level-of-detail directly in the

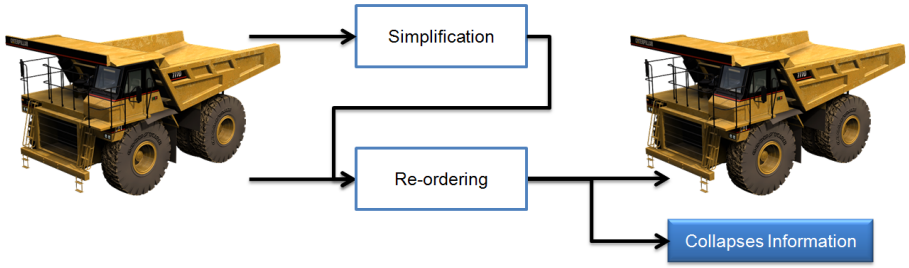


Figure 5.2: Construction of an Interactive Meshes model.

GPU. In our case, we decided to update the contents of the vertices list instead of the indices one. The reason behind this decision is the fact that applying LOD changes in a vertex basis suits perfectly the graphics pipeline, as we will be able to update the vertices information in a vertex shader. Nevertheless, the extraction algorithm that we are presenting would be equally suitable for modifying a list of indices.

The use of the vertex shader to perform the vertex update entails the necessity of developing an extraction algorithm which is capable of applying changes on a vertex basis. Thus, each vertex will be processed individually and there will be no shared memory once the process has started; all the information must be arranged before the extraction process starts. Moreover, as we aimed at keeping the framework simple, we wanted to avoid multiple passes for updating the level of detail. As a consequence, we must develop a framework which can update a vertex in a single pass no matter which is the difference between the old and the new level of detail.

The update of vertices offers an efficient extraction of the different levels of detail of an original mesh. Nevertheless, as it happened with the models presented in previous sections, we must consider the appearance of degenerate triangles. As we will see, a possible solution is to order indices so that we can apply a sliding-window approach to the level-of-detail extraction process.

5.2.1. Pre-processing the original mesh

In order to use Interactive Meshes, the user must perform an initial preparation task so that the input mesh meets certain requirements and the simplification information is made available. Figure 5.2 presents the construction process of our model, showing that the pre-process is based on two aspects:

- a simplification process, which also involves the storage of the collapse information.
- a re-ordering step of vertices and indices.

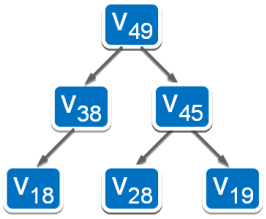


Figure 5.3: Example of the collapse hierarchy of a sample model.

Storing the collapses information

The construction process of *Interactive Meshes* includes outputting, for each vertex, the different vertices it collapses to throughout the levels of detail. In our proposed scheme, collapsing vertex v_i to vertex v_j would mean that the coordinates values of vertex v_i will be replaced with the values of vertex v_j. The simplification step can be used to collect the collapse information involved in the whole simplification process.

The collapse information of a vertex will be composed of a list of references to the vertices that it collapses to. Figure 5.3 presents the collapse hierarchy of a section of a mesh. The collapse list of each vertex can be understood as the branch of this tree that links it with the root node. Thus, for example, vertex v_{18} would have a list of indices to vertices composed of values (v_{38}, v_{49}).

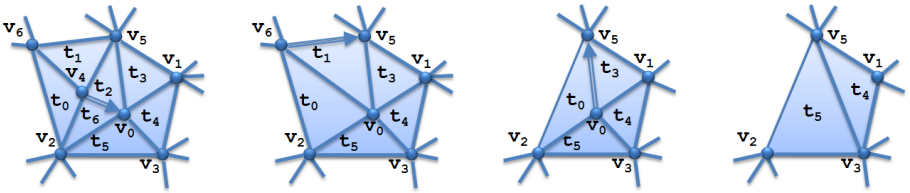
Ordering vertices and indices

On the other hand, the collapse information is also used to order the geometry of the original mesh. As we did in the models we previously presented, vertices are ordered following the collapse order. The vertex that first collapses will become vertex v_0, the second one will be vertex v_1, and so on.

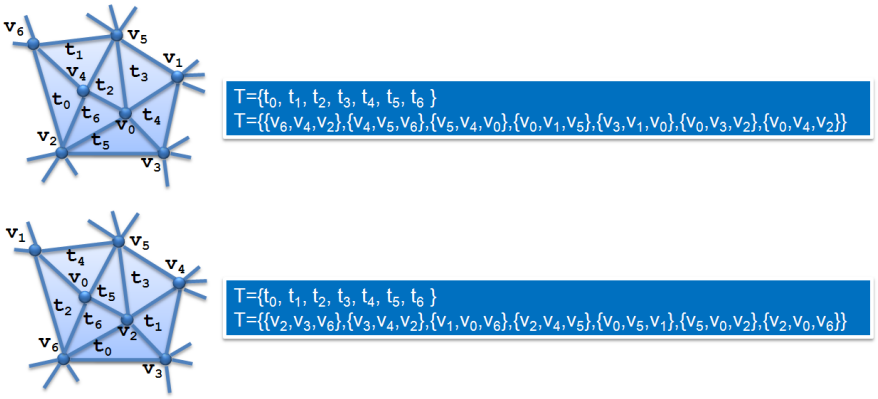
As the vertices have been ordered following the collapse order, the collapse list will let us know in which LOD a particular vertex must change and, in addition, which change should be performed. More precisely, we can assure that the collapse list of vertex v_i satisfies that we must use the contents of its j-th element while:

$$e_{i,j} \leq demandedLOD \tag{5.1}$$

where e represent the elements of the collapse list. More precisely, we can assume that vertex v_i will change when swapping from lod_i to lod_{i+1}. Thus, following on with the example offered in Figure 5.3, we can say that we must perform the collapse $v_{18} \rightarrow v_{38}$ when changing to LOD 19, and that we must apply collapse $v_{38} \rightarrow v_{49}$ when swapping to LOD 39. It is worth remembering



(a) Simplification of a section of a polygonal model.



(b) Initial ordering (top) and re-order of triangles and vertices (bottom).

Figure 5.4: Example of simplification and ordering of vertices following the collapse order.

that, in our proposed scheme, collapsing vertex v_i to vertex v_j would mean that the contents of vertex v_i will be replaced with the values of vertex v_j. As a consequence, the contents of vertex 18 will be replaced by the contents of vertex 38 when changing to LOD 19, and will be modified to contain the information of vertex 49 when changing to LOD 39.

From a different perspective, the update of vertices we are proposing offers an efficient extraction of the different levels of detail of an original mesh, but we must consider the appearance of degenerate information. A possible solution is to process the triangles in a geometry shader in order to eliminate the unnecessary geometry. However, in our continuous framework we decided to order the triangles by their elimination order, so that the last triangle in the triangle list will be the first one to disappear. Thus, as each simplification step involves eliminating two triangles, when modifying the level of detail we will easily be able to discard unnecessary degenerate triangles by modifying the number of indices sent to render.

To clarify this process, an example of the ordering step can be observed in Figure 5.4. Figure 5.4(a) depicts a section of a polygonal mesh which is

simplified with two edge-collapse operations. This image offers the resulting geometry after vertex v_4 collapsed to vertex v_0 and vertex v_0 collapsed to v_5. Figure 5.4(b) depicts the changes that are necessary to meet the ordering requirement. On top we present the original contents of the triangle lists. At the bottom, we offer the list obtained after the re-ordering. Thus, we can see how vertex v_4 is now vertex v_0 and vertex v_0 is now vertex v_1, following the order of vertex collapses. Moreover, it can be seen how triangles t_5 and t_4 become t_0 and t_1, as they will be the last ones to disappear.

5.2.2. Data structures

Once this pre-process step is finished, the collapse information and the re-ordered mesh are obtained. Interactive Meshes has very low memory requirements. The only extra information that we will need to store is the information about the collapse list of each vertex. Nevertheless, it will not be necessary to store the whole list for each vertex. Following with the example shown in Figure 5.3, the list of vertex v_{18} will be (v_{38}, v_{49}), while the list of vertex v_{38} will be (v_{49}). As a consequence, we will only need to store, for each vertex, the vertex it collapses to, as we will be able to recover the whole hierarchy of collapses afterwards.

For offering an estimation of the storing cost of the model we are presenting, let us suppose that the cost of an integer and a float is a word (4 bytes). In these estimations we will just consider spatial coordinates for each vertex. As we showed in previous chapters, a mesh in triangles requires $6 \cdot v$ for indices and $3 \cdot v$ words for vertices, supposing $9 \cdot v$ words. The only extra information that we need is the value of the vertex to collapse to, as the complete list of collapses will be obtained every time we decide to use this multiresolution model. As a consequence, the total size in words of our model will be $10v$ (Eq. 5.2), which is 320 $bits/vertex$ and just 10 % higher than the size in triangles.

$$9 \cdot v + v = 10 \cdot v \tag{5.2}$$

Storing the data structures on GPU

A key aspect of our proposed framework is the adequate storage of the necessary information in the GPU. Thus, starting from the information stored in main memory, it will be necessary to create the data structures on GPU memory.

Figure 5.5 depicts the information that is initially stored in CPU and afterwards uploaded to GPU memory. It is necessary to distinguish between two kinds of data: static data, which will not be altered, and dynamic data, which is updated in each extraction step.

On the one hand, the static data contains the index buffer and two textures:

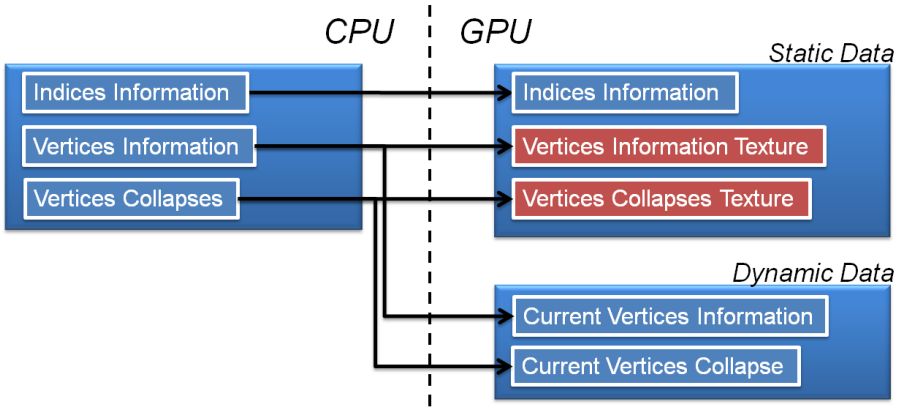


Figure 5.5: Data structures preparation and GPU storage.

- the information of the original mesh (vertex coordinates, texture information, normals and so on) will be stored in floating point textures. This information will be accessed in the vertex shader if the contents of a vertex must be modified when changing the level of detail.
- the collapse information will be stored in another texture. In the preprocess step we stored the collapse information, indicating for each vertex the vertex it collapses to. Nevertheless, in the run-time of our extraction algorithm we will need the complete collapse list of each vertex. The collapse list reflects the different vertices an original one collapses to throughout the levels of detail, and is key information for our proposal.

On the other hand, the dynamic data is composed of two vertex buffers containing the information of the latest extracted approximation that is currently used for rendering. These two buffers store:

- the current information of the vertices, including spatial coordinates, normals and so on.
- the current element of the collapse list that the vertex is using, which can be seen as a pointer to traverse this list.

Once these data has been properly stored, the only information that the CPU must send to the GPU is the new LOD value. It is important to mention that the different vertex and index buffers can be easily created, while the two textures require a small process. It is worth mentioning that Shader Model 4.0 enables us to define and use non-squared textures, which do not have the restriction of having a size power of two and thus, offer more cost-effective information storage.

Storage cost

For analyzing the multiresolution framework we present, it is necessary to address its memory cost in the GPU. The memory cost of the three elements of the static information is presented in Equations 5.3 to 5.5. Equations 5.3 to 5.4 present the storage of a mesh in triangles, which require $6 \cdot v$ words for indices and $3 \cdot v$ words for vertices. Regarding the storage cost of the collapse list, our tests have proven that, on average, the collapse list of each vertex has a maximum size of 13 elements, although the average is 3 elements. If we see the collapse hierarchy as a tree, in the levels close the root nearly all vertices are collapsed, which means that when retrieving very coarse levels of detail we must change a lot of vertices. Nevertheless, by simply limiting the maximum LOD extraction to 95 % of the total possible levels of detail, the maximum collapse list size is reduced to 5 and its cost is $5 \cdot v$ (see Equation 5.5).

$$IndicesInformation \rightarrow 6 \cdot v \tag{5.3}$$
$$VerticesInformation \rightarrow 3 \cdot v \tag{5.4}$$
$$VerticesCollapses \rightarrow 5 \cdot v \tag{5.5}$$

The dynamic information consists of two elements: the current vertices buffer and also the current collapse for each vertex.

$$CurrentVerticesInformation \rightarrow 3 \cdot v \tag{5.6}$$
$$CurrentVerticesCollapse \rightarrow v \tag{5.7}$$

Both dynamic structures must be duplicated for *ping pong*, as we cannot read and write to the same buffer with the current 3D API. Thus, with the presented equations we can conclude that the total memory cost is $22 \cdot v$ (see Equation 5.8) or 704 $bits/vertex$, which represent 2.4 times the original model. If we wanted to apply normal mapping and textures, the original size would increase to $13 \cdot v$ (416 $bits/vertex$), and our approach would have a memory cost of $34 \cdot v$ (1088 $bits/vertex$), which represents 2.6 times the original cost.

$$3 \cdot v + 6 \cdot v + 5 \cdot v + (3 \cdot v + v) \cdot 2 = 22 \cdot v \tag{5.8}$$

5.2.3. Extraction algorithms

In the previous sub-section we introduced the information that our multiresolution model will use to extract and render the different approximations. Once the data is correctly stored, the only information that the CPU must send to the GPU is the new LOD value. The extraction process has been carefully adapted to work using shaders. We will consider that each collapse operation

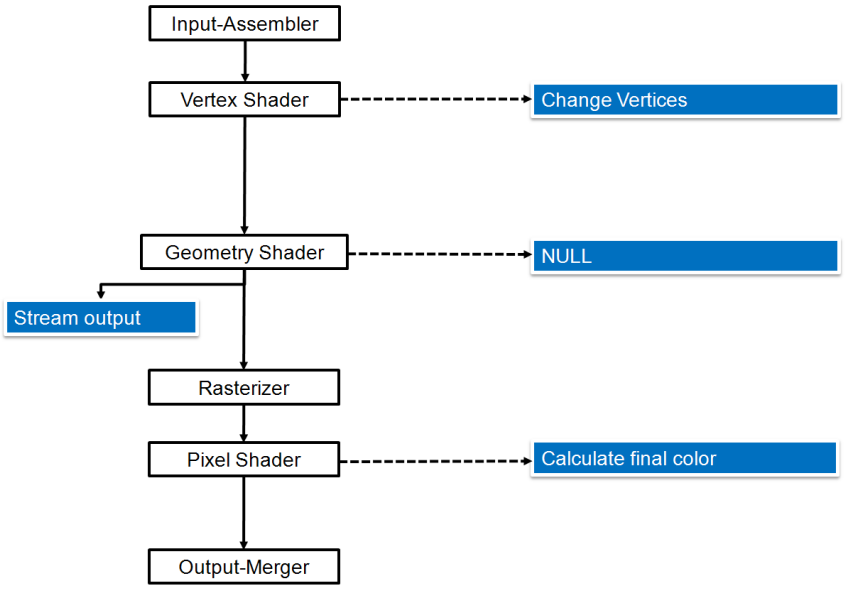


Figure 5.6: Rendering pipeline for the continuous Interactive Meshes approach.

entails a change in the level of detail. Thus, we will have as many levels of detail as edge collapses recorded during the simplification process.

Figure 5.6 depicts how the rendering pipeline is used in this approach. The Stream Output possibilities of the current Shader Model enables storing the updated vertices information to be used in subsequent renders. This feature is very interesting when the level-of-detail remains stable. Moreover, it also assures that coherence among extracted levels of detail is maintained, both when refining and coarsening the mesh.

We will describe separately the vertices and indices update. The vertices updating process will be similarly used in the following section when describing the features of the view-dependent model, as both multiresolution implementations are based on the same extraction process.

Vertices update

Each time we change to a different LOD, the pipeline shown in Figure 5.6 is activated so that each vertex is processed in a specifically designed vertex shader. Nevertheless, due to the order we have chosen for the vertices, it will only be necessary to check vertices from 0 to $demandedLOD-1$. It is important to say that, among the vertices to check when changing to a new level of detail,

Algorithm 5 Pseudocode of the extraction shader of the continuous version of Interactive Meshes

```
// Vertex shader.
float CurrentID;
float3 NewVertexInfo,OldVertexInfo;

CurrentID = getCollapseInfoFromTex(VertexID,CurrentCollapse);
if CurrentID < DemandedLOD then
  CurrentCollapse+=1;
  NewID=getCollapseInfoFromTex(VertexID,CurrentCollapse);
  NewVertexInfo = getVertexInfoFromTex(newID);
else
  NewVertexInfo = OldVertexInfo;
end if
Output(NewVertexInfo,CurrentCollapse);
```

we will not have to update all of them, as just a small percentage must be modified. The size of the collapse list of each vertex is usually small. Our experiments have shown that the collapse list of the vertices of most models have an average size of 2 (see Table 5.2) and, thus, each vertex will be updated on average two times. Obviously, the bigger the difference between the current LOD and the demanded LOD, the greater the number of vertices that will be updated.

Algorithm 5 presents a pseudocode that describes the extraction shader for the vertices. The proposed shader receives as input the $CurrentCollapse$ of the current vertex ($VertexID$). This value can be understood as a pointer to the collapse list that the algorithm moves to increase or decrease the level of detail. With this value, we access the texture containing the collapse list of the vertex to find out which vertex we are currently using. If, according to the $CurrentID$ of the vertex its contents must change, we increase the $CurrentCollapse$ value and we retrieve the following value of the collapse list. Then, we fetch the adequate vertices coordinates to update the contents of the vertex buffer. The vertices coordinates and $CurrentCollapse$ value are outputted by means of the Stream Output so that they can be input to the following execution of the extraction shader. This information represents the dynamic data of our approach (see Figure 5.5).

This extraction process assures that we will update only those vertices that need to be modified from the latest extracted approximation. Thus, the process is capable of re-using the latest calculated approximation. The proposed algorithm can be easily modified to return to more detailed approximations, by changing the comparison used and reducing the $CurrentCollapse$ value when necessary.

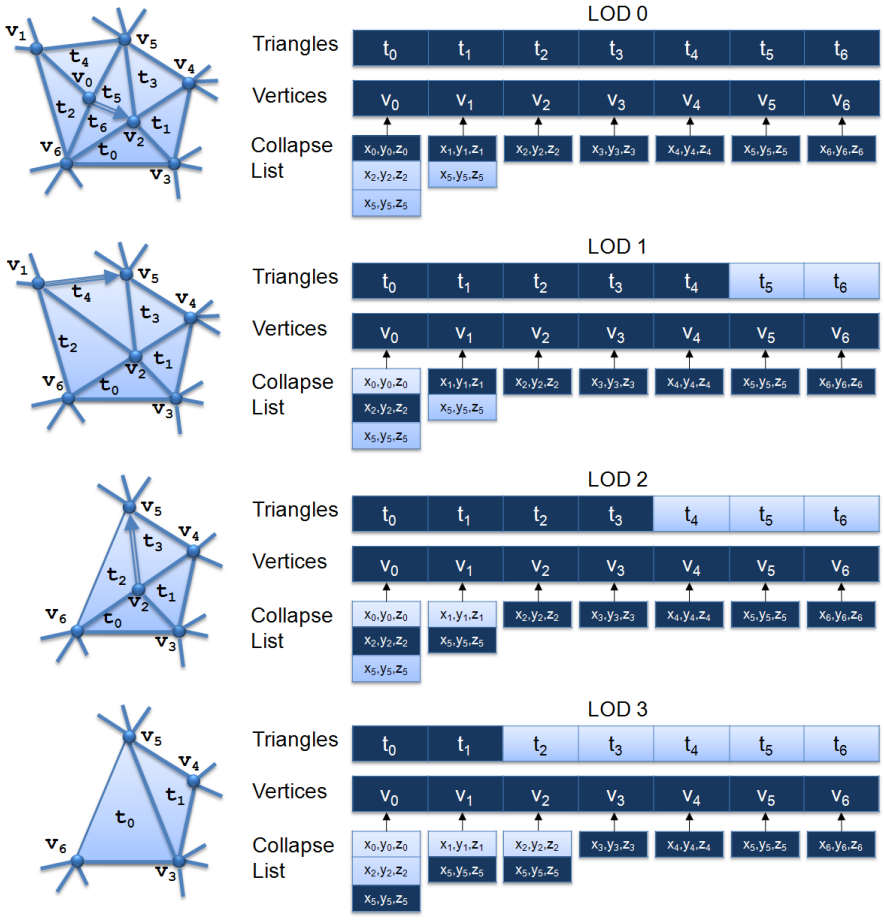


Figure 5.7: Example of the extraction process of four levels-of-detail.

Indices update

As we commented above, the triangle list has been ordered in an elimination fashion. Thus, once we have updated the necessary vertices, the *triangle_count* must be increased or reduced appropriately in order to render the proper number of indices for the level-of-detail extracted.

As an example of how the whole algorithm works, Figure 5.7 presents a sample mesh during three edge collapses using our model. This figure includes, for each level of detail, the array of triangles and vertices and, for each vertex, its collapse list. The array of triangles is shaded in accordance to the geometry that is currently being rendered. With respect to the collapse lists, the shaded cell

reflects the current contents of the vertices. Following the algorithm introduced in Algorithm 5, let's suppose we change from LOD 0 to LOD 1. We would decrease the triangle count in two and, in this case, we would only modify vertex v_0 so that its coordinates are updated with the contents of vertex v_2. For the second LOD change, vertex v_1 must be updated, and according to the contents of its collapse list, it must change its values for vertex v_5. The next level-of-detail change implies updating vertices v_0 and v_2, as both must collapse to vertex v_5.

5.3. View-dependent resolution framework

The modification of the vertices to perform the level-of-detail updates introduced above offers a perfect framework for enhancing the extraction process. What we propose now is a revision of this continuous model. The main contribution is to develop a view-dependent model that preserves appearance and avoids popping artifacts while offering high performance:

- the use of triangles as rendering primitive limits performance, compared to triangle strips. This limitation can be overcome with the use of primitives optimized for the vertices cache [41, 46], which orders the indices in an optimized way which renders much faster than the triangles ordered in an elimination fashion. In the results section we will present a small study which shows how indexed primitives offer the fastest rendering. In our case we apply the method presented in [109] which is based on one of the latest methods [42].

- the pre-ordered list of triangles assured that no degenerate triangle was rendered. In this new solution, we control the appearance of degenerate information directly in the geometry shader.

- we have extended the original algorithms to offer view-dependent resolutions. Among the criterions that can be applied to select the refinement level, we have chosen a simple yet efficient silhouette method. Figure 5.1 presents several visualizations of a model of a man at different levels of detail using a silhouette-preserving extraction algorithm.

- to enhance the visual quality, we will perform geo-morphing between the collapsed vertices in order to avoid disturbing effects like discontinuities or popping artifacts.

It is important to comment that the extraction approach based on collapse lists offers a truly selective refinement, where we can apply any collapse without applying further collapses or other requisites. As the simplifications are applied in a vertex-basis, all the triangles sharing that vertex will be modified in the same way. Figure 5.8 shows an image of the Stanford bunny model simplified

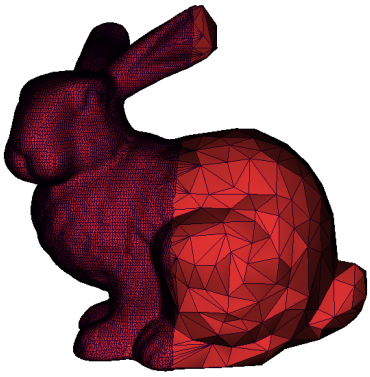


Figure 5.8: Bunny model with its right half simplified to 80 %.

with this method, where the detail of half of the model is reduced 80 %. It can be seen how no crack or other disturbing effect is produced, even though there is a severe change of resolution between the two halves of the model.

5.3.1. Implementation details

Our aim is to extend the previously presented approach to offer a fully-GPU view-dependent model. As a consequence, in this section we will address the modifications of the previous continuous resolution model that are necessaries to develop the view-dependent approximation.

The pre-process step and the data structures presented before can be equally used for this view-dependent resolution approach. The only difference is that the indices information becomes part of the dynamic data, as the degenerate elimination takes also place on the GPU.

Figure 5.9 shows a diagram of the different processes that will take place at each rendering stage. The extraction process is similar to that presented in the previous section. Nevertheless, we have modified the criterion to select the appropriate level of detail, which will depend on a silhouette criterion. In the geometry shader we will mainly perform the degenerate triangles elimination. The different shaders will be commented separately in the following sub-sections.

Vertex shader

The vertex shader will be responsible of calculating the appropriate LOD, updating vertices information and performing geo-morphing. In Algorithm 6 we present a detailed description of the implemented shader using pseudocode.

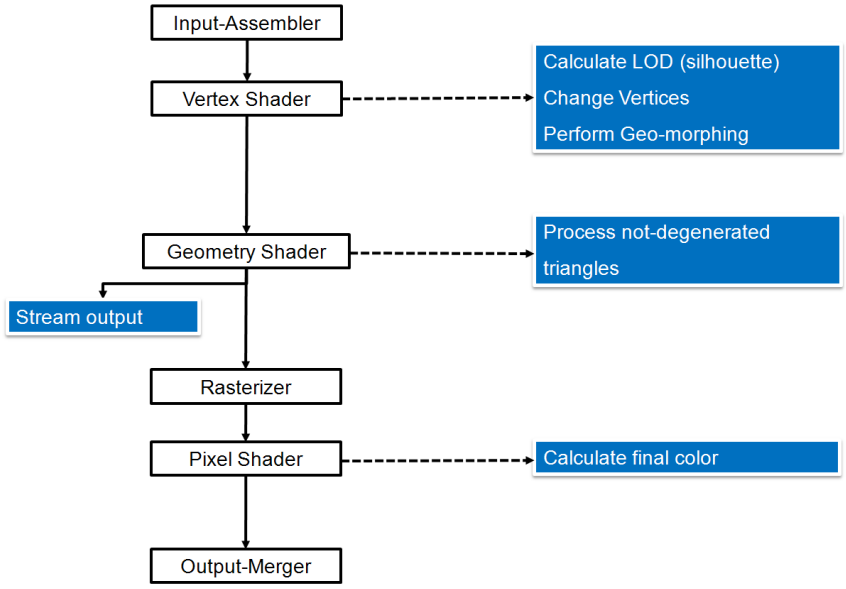


Figure 5.9: Rendering pipeline for the view-dependent Interactive Meshes approach.

We want to develop a model that will not need any information from the CPU once the model is correctly loaded into GPU memory. With that aim, we want to calculate the appropriate LOD according to the scene conditions inside the GPU. Knowing the angle between the vector that points towards the camera and the normal of the vertex will allow us to easily perform a silhouette-based extraction process. In those cases where the vectors are nearly perpendicular, we will need to render highly-detailed geometry to obtain the visual perception of the silhouette. In those cases where the vectors are nearly parallel, we will simplify the vertices as they do not contribute to the silhouette. In the rest of cases, we will perform a linear interpolation so that the geometry refines progressively towards the areas of the silhouette.

The first instructions of the vertex shader will calculate the angle between the view vector and the normal of the vertex. The vertex shader has full access to the ModelView matrix. As a consequence, we can easily calculate the dot product between the vertex normal and the view direction.

After this step, we are able to extract the correct geometry for the *CalculatedLod*. As we explained in the previous section, we will consult the collapse information to know which vertex information must be used, recovering all the information from the previously-defined textures. This process is very similar

Algorithm 6 Pseudocode of the extraction shader of the view-dependent version of Interactive Meshes.

```
// Vertex shader.
float CalculatedLod;
float Angle;
float NewID,NextID;
float3 NewVertexInfo,NextVertexInfo,FinalVertexInfo;

Angle=calculateAngle(view,normal);
CalculatedLod=interpolate(DemandedLOD,Angle);
NewVertexInfo= getNewCoordinates(VertexID,CalculatedLod);
NextVertexInfo = getNextCoordinates(VertexID,CalculatedLod);
NewID=getNewID(VertexID,CalculatedLod);
NextID=getNextID(VertexID,CalculatedLod);
FinalVertexInfo = geomorph(NewVertexInfo,NextVertexInfo,newID);
```

to the extraction process presented in the first sections, as this model is also based in the correct ordering of vertices and in the search of the appropriate value through the collapse list. As a consequence, the extraction process has been summarized as *getNewCoordinates* so that the reader can focus on the new parts of the algorithm.

It is important to note that we will recover the information of the vertex that we currently need and the following one. With the two extracted vertices we can make some simple calculations to assure a progressive transition among LODs. The way that the collapse information is stored will assure that a vertex will collapse to its j-th element of the collapse list once we reach LOD j. Our proposal is to geo-morph between vertices stored in the positions j and $j+1$ while the LOD value is contained between the *ids* of vertices stored in positions j and $j+1$ of the collapse list. Thus, once we reach LOD j the vertex will be completely changed to vertex j, ensuring that the collapse information is correctly applied. With this approach the continuity of the mesh is ensured, as all the vertices will be collapsed in the same way.

Geometry shader

The development of the geometry shader has made it possible to work directly with triangles in a new stage. This feature is very powerful but the geometry shader is not a stage that must be activated. Consequently, the use of geometry shaders involves slowing down the whole rendering process. Nevertheless, it is worth activating this rendering stage when we are able to discard a considerable amount of geometry or when we need to create geometry on-the-fly. In our case, we can expect that the coarser the approximation that we want to render, the greater the number of degenerate triangles that will be obtained.

Thus, we decided to use the geometry shader to filter the degenerate triangles in real time. As a consequence, we will perform a simple test to discard those triangles which have repeated vertices.

5.4. Integration into a real application

The simplicity of our proposal enables the user to integrate our model into any application, being able to code the whole process in shaders only. A further improvement is the fact that the extraction process has been designed so that it requires a single rendering pass, in contrast to previous solutions that require several. Thus, our algorithm only requires the user to perform a very small number of tasks to prepare the multiresolution model and to use it afterwards.

Figure 5.10 depicts the different steps that the user should follow to use our proposal. Firstly, in a pre-process step, the original mesh is re-ordered and the simplification information is stored, based on edge-collapses. The output mesh is directly renderable, as its information is not altered. This task is only performed once.

Then, with this information and the appropriate shaders, the user will be able to prepare the data structures and extract the desired approximation. Thus, each time the user wants to apply our multiresolution technique it will be necessary to execute two shaders:

- The construction shader only need to be executed once to process the original information and output, to GPU memory, all the information needed to correctly run the extraction algorithm. As we had seen in previous sections, this information is composed of a set of static data and a set of dynamic data which is altered continuously.

- The LOD update shader is used in the run-time of the algorithm, processing the level-of-detail extraction in one single rendering pass and updating the contents of the dynamic data structures.

To show how we could implement our solution in a real application, we have decided to propose an adequate framework for its use with nVidia's FX Composer. FX Composer is a powerful tool that enables easy creation and testing of shaders. It includes advanced features such as render to texture, Direct3D FX file support and a pipeline that is configurable via scripting.

Our objective is to define a single scripting file which is capable of preparing and managing the level of detail. Thus, the FX Composer only requires the original mesh, the basic collapse information and the script containing the shaders.

In the construction pass, the aim is to generate a texture out of the vertex information located in vertex buffers, so that the LOD shader can access it afterwards. Moreover, in the same pass we generate the collapse lists texture,

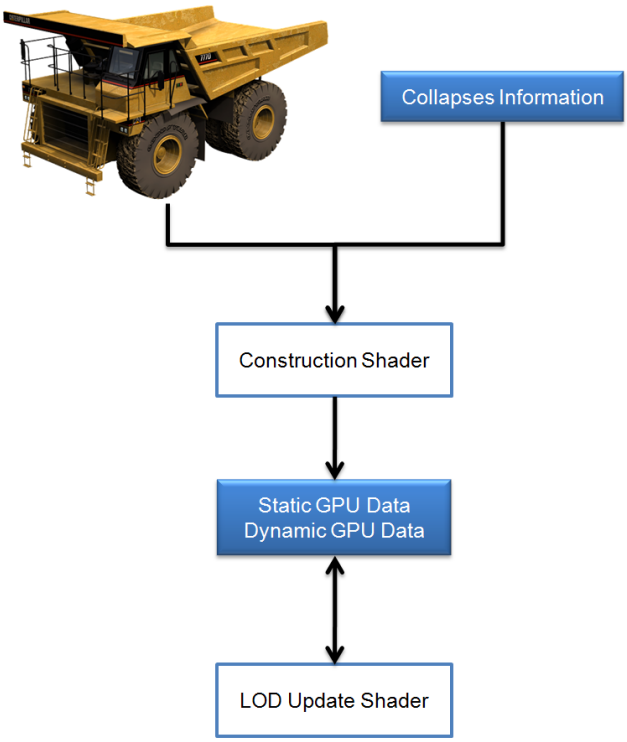


Figure 5.10: Construction and rendering pipeline for the integration of our proposal.

which will be obtained with the information on collapses. Thus, for each vertex, we would consult the basic collapse information as many times as necessary to build to the complete collapse list.

To create these textures, we configure different color buffers to store the required information, such as vertex positions, normals and texture coordinates. In order to generate these textures efficiently, we use Multiple Render Targets (MRT), which are supported on modern graphics hardware and enable us to write to different color buffers at a time in the pixel shader.

Once the necessary data has been made available, the LOD update shader would need two sets of vertex buffers to be used as *ping-pong* buffers in order to store data for each frame. FX Composer enables us to define these *ping-pong* buffers by setting two global variables inside our FX file to define which of the buffers are going to be used as input and as output for the render. At each frame, one of these buffers is configured as data source input, whereas the other one is assigned as the location for the Stream Output to store the output data. Then, after the execution of this shader, the geometry would be ready to

be rendered.

5.5. Results

In this section we present some tests that analyze the rendering performance of Interactive Meshes in its both implementations: continuous and view-dependent. Moreover, we also describe the results obtained in some tests that have helped us to select the most appropriate techniques when developing the current multiresolution framework. Table 5.1 characterizes the polygonal models that have been used throughout the different experiments. We show their vertices and faces numbers, which are useful to understand their polygonal complexity.

The experiments were carried out using Windows XP on a PC with a 2.8 GHz processor, 2 GB RAM and an nVidia GeForce 9800 GT graphics card with 512MB RAM. The different implementations have been done in C++, OpenGL and GLSL. Finally, it is important to note that we have used the GL_TIME_ELAPSED_EXT extension, which provides a query mechanism to determine the amount of time used for completing a set of GL tasks without stalling the rendering pipeline. This approach offers more accurate timing calculations and a better control over the time required by each of the stages that compose the multiresolution techniques.

5.5.1. Storage and memory cost

The memory needed by a multiresolution model is a key feature, considering both the storage cost and the memory cost once the model is loaded into main or GPU memory. In previous sections we performed some small studies to estimate these costs, concluding that we need 320 $bits/vertex$ for storing the model and 704 $bits/vertex$ of GPU memory.

Table 5.1 shows a comparison of spatial costs among previous continuous resolution models: Progressive Meshes [16], a triangle-based approach, and Speed Strips and Masking Strips, which are based in triangle strips and have been presented in previous chapters. The storing costs of both versions of Interactive Meshes are the same. As it can be observed, the presented model offers the best spatial cost, improving on the results offered by Masking Strips in more than 20 %. On average, our approach fits in 1.1 times the original mesh in triangles, in contrast to Speed Strips and Masking Strips which fit in 1.7 and 1.4 times respectively. This is due to the fact that the only extra information that we store is the collapse information of the vertices.

On the other hand, the GPU memory that is necessary for the proper performance of our model supposes 2.6 times the original model size. This cost is low when compared with the GPU solution introduced in [73], whose memory needs can be more than 3 times the original mesh. It is worth mentioning that

Model	Cow	Bunny	Phone	Isis
Vertices	2,904	34,834	83,044	187,644
Faces	5,804	69,451	165,963	375,283
Original (triangles)	289	282	287	288
Progressive Meshes	780	766	794	770
Speed Strips	462	483	475	496
Masking Strips	391	416	402	422
Interactive Meshes	318	320	320	320

Table 5.1: Details of the models and storage cost study (in bits/vertex).

Model	Cow	Bunny	Phone	Isis
Average size	5	2	2	2
Maximum size	11	6	13	13

Table 5.2: Collapse list size information.

these results are helpful to prove that the estimations performed in previous sections for storage and memory costs are correct.

5.5.2. Collapse list size study

One of the main aims of our proposal is reducing the extraction time. In this sense, it would me important to know how many vertices are processed each time we change to a different level of detail. Table 5.2 presents a small study on the size of the collapse list of the different meshes used in our experiments. It is important to note that the size of each collapse list is usually small. Our experiments have shown that the collapse list of the vertices of most models have a maximum number of 12 elements and an average of 2, despite being meshes composed of thousands of vertices. Thus, most vertices will just be updated twice when traversing all the levels of detail.

5.5.3. Primitive study

The decision of using primitives optimized for the cache in the view-dependent version has been taken after testing the performance of different rendering primitives. In Table 5.3 we present the frame rate obtained with different models at two levels of detail. It is important to say that for the low levels of detail we have eliminated all the degenerate information possible. It can be seen how optimized triangles are the fastest primitive (40 % faster if compared with regular triangles) even in coarse approximations where the optimization has not been re-calculated. Triangle strips have a similar performance, while the optimized strips are not very efficient.

Model	Cow		Bunny		Isis	
Approximation	100 %	20 %	100 %	20 %	100 %	20 %
Triangles	4,850	5,320	1,212	3,210	289	2,035
Optimized Triangles	5,210	5,048	2,419	4,750	758	3,520
Strips	5180	5,450	2,022	3,160	650	1,678
Optimized Strips	3,620	4,250	1,015	1,427	151	192

Table 5.3: Performance comparison (in fps) among different primitives and models at two levels of detail.

These tests prove that optimized triangles offer the fastest rendering. Thus, these results have encouraged us to develop the view-dependent model with cache-optimized triangles.

5.5.4. Rendering time

In this chapter we have presented a new multiresolution framework and two different implementations, one that offers continuous resolutions and one which is capable of offering view-dependent resolutions.

To evaluate the continuous model, we have conducted several linear tests to measure the model performance when a linear sequence of LODs is required [105]. The scene containing the models is illuminated but it lacks any kind of texturing. For the difference between LODs that are consecutive in the sequence (the *step*), we decided again to use 0.1 % of the number of available LODs. In these cases we analyze the average performance obtained when reducing the level-of-detail to the minimum and recovering the highest one.

We have also tested our proposed solution against Speed Strips, the continuous solution introduced in Chapter 3. Table 5.4 presents the average rendering time, including both the extraction and visualization times. It can be seen how, on average, the continuous approach we are presenting offers a rendering time which is about 15 % higher in comparison. Nevertheless, it is important to note that the performance comparison among different primitives presented in Table 5.3 shows that triangles strips are much faster than regular triangles. Thus, it could be expected that Speed Strips, which is based on triangle strips, rendered faster that our proposed triangle-based framework. Regarding the view-dependent model, the results prove that the model entails a performance which is about 40 % slower if compared with its continuous approximation.

Seeing the differences in performance of the different rendering primitives, we have considered that it would be interesting to analyze the extraction process individually. As a consequence, Table 5.5 presents the temporal costs obtained when the visualization part is omitted. In this case, the Interactive Meshes model involves an extraction time which is, on average, 25 % lower. This is due to the fact that processing LOD information on shaders offers a very effi-

Model	Cow	Bunny	Phone	Isis
Speed Strips	0.62	0.98	1.46	3.11
Continuous Interactive Meshes	0.64	1.15	1.77	3.12
View-dependent Interactive Meshes	0.98	1.76	2.60	4.07

Table 5.4: Comparison of average rendering time (extraction+visualization) (in ms.).

Model	Cow	Bunny	Phone	Isis
Speed Strips	0.59	0.85	1.10	1.96
Continuous Int. Meshes	0.62	0.72	0.96	1.35
View-dep. Int. Meshes (no Degenerate Filter)	0.62	0.74	1.02	1.39
View-dep. Int. Meshes	0.94	1.51	2.18	2.84

Table 5.5: Comparison of average extraction time (in ms.).

cient framework. Regarding the view-dependent model, the table presents two different rows, as we are also including the extraction time when no degenerate elimination is performed. This row is interesting as it shows that, despite including the silhouette extraction mechanism, the view-dependent solution without degenerate treatment is capable of offering similar extraction times, proving that the view-dependent algorithm does not affect much the final performance. It is the elimination of the degenerate information which considerably increases the final rendering time. As we commented before, enabling the geometry shader diminishes the overall performance, although in our case it is necessary to eliminate the unnecessary information.

5.6. Conclusions

In this chapter we have presented a new multiresolution framework which combines the power of current GPUs with traditional techniques to offer a fully-GPU solution, benefiting from the parallelism of graphics hardware. This scheme offers low storing cost, easy implementation and a fast extraction process which make it suitable for any rendering engine. Moreover, it has the interesting feature of, once the LOD approximation is created, being able to render it as an ordinary indexed triangle list.

Updating vertices instead of indices allows us to perform geo-morphing among the different levels of detail to offer smooth transitions, improving on the final visual quality. The framework also allows for view-dependent resolutions, which can be oriented towards applying silhouette-based visualizations that better preserve the appearance of the model. This method is suitable for combining with other techniques, such as normal mapping, hardware skinning,

and other pixel-based approaches.

From the results obtained we can conclude that performing the LOD extractions directly on GPU reduces the temporal cost. As the results section shown, the view-dependent model entails a considerably increase in the extraction time, as the geometry shader is not capable of performing the geometry elimination very fast, increasing the final rendering time by 25 %. Nonetheless, the level-of-detail extraction would be much more costly if it was applied in a CPU-based way. Furthermore, the extra cost that our model introduces is compensated by the number of calculations performed and the final visual quality.

As we have described, Interactive Meshes offers a perfect solution for incorporating a LOD approach into any application, as it only requires the input information and two shaders, one for constructing the GPU information and the other one for processing the level-of-detail. Moreover, the tasks that need to be performed to enable the extraction, such as indicating the input vertex buffers or performing *ping-pong*, can be incorporated into the effects file of applications like nVidia's FX Composer.

CHAPTER 6

Conclusions and Future Work

The work presented in this Ph.D. thesis has been oriented towards the development of level-of-detail techniques which suit the necessities of graphics applications with different hardware configurations. In this sense, the final aim is to offer level-of-detail solutions which:

- obtain approximations which are visually satisfying
- optimize the memory needs of the data structures
- reduce the temporal cost of the extraction algorithms
- offer an easy-to-implement solution

With these objectives in mind, we have proposed different models to develop a solution which is suitable for game engines and graphics libraries which often resort to discrete models when it comes to selecting a multiresolution technique.

This chapter is organized as follows. First, we conclude on the contributions offered by the different proposals. Then, we outline ideas for future work. Finally, we list the publications related to the presented dissertation and the research projects that have enabled the development of this thesis work.

6.1. Conclusions

Before analyzing the contributions of each proposal, it is important to comment that each of them has been studied from the perspective of the graphics hardware that was available at the moment. Thus, it is possible that some of the techniques have become old-fashioned due to the graphics hardware evolution.

Nevertheless, these techniques are still interesting as there are platforms like gaming consols, PDAs or mobile phones which include hardware with similar architectures to those analyzed in the different chapters.

Chapter 2 presented the state-of-the-art on multiresolution modeling, as well as a brief study on simplification, primitive optimization and LOD selection criteria. From this initial study we concluded that, although there is a wealth of research on this issue, there is still a gap for efficient level-of-detail techniques. Most CPU-based solutions entail complex processes that make them not adequate for graphics applications, while recent GPU-based approaches are more aimed at *tesselating* an initial coarse mesh than at retrieving the original geometry.

For improving on previous solutions, Chapter 3 described Speed Strips, a multiresolution model which simplifies the extraction process by performing all changes in only one pass, reducing the temporal cost of previous solutions. The extraction process has been devised to optimize the use of the PCI Express bus, studying its features. From this study we concluded that it is better to perform several operations that involve small traffic instead of uploading all the information at the same time once the approximation is extracted. In addition, in this chapter a primitive study to address the performance obtained with different rendering primitives showed that, with the hardware available, the strips obtained with *Stripe* offered the best performance. With all these considerations, Speed Strips offered in the results section a 5 % reduction in storage cost in comparison with LodStrips and reduced its total temporal cost in 30 %, while the extraction time was reduced to just 70 %.

Following with the exploitation of graphics hardware, in Chapter 4 a different level-of-detail scheme was presented: Masking Strips. The main objective of this solution was to code the operations to modify the level-of-detail in masks of bits, as an extraction process based on bit-wise operations is feasible to be ported and executed on the GPU with the Shader Model 4.0. Moreover, the use of masks of bits offers a perfect framework for eliminating all the unnecessary degenerate triangles, in contrast to previous approaches that were just capable of eliminating degenerate triangles of certain types. Consequently, in the coarsest approximations we can reduce the indices processed in 40 %. Moreover, this approach reduces considerably storage cost, this reduction being up to 15 % if compared with LodStrips. Masking Strips offers a perfect solution for the progressive transmission of 3D models, as it is possible to refine an initially sent coarse model with successive data packages. This multiresolution model has been integrated into the Ogre game engine, offering very interesting results if compared with the discrete solution that this game engine includes.

Finally, Chapter 5 thoroughly describes Interactive Meshes, an approach to offer a multiresolution model which works completely on graphics hardware, as very few models have been presented to offer level-of-detail modeling on GPU. This triangle-based framework performs LOD calculations by modifying vertices information using *shaders*. Moreover, the proposed solution is capable

of displaying continuous and view-dependent approximations while enhancing the final visual quality with further processes like geo-morphing. This scheme can be easily incorporated in any application supporting Shader Model 4.0, as two shaders and very little scripting are sufficient for expanding the data structures on GPU and performing the LOD selection and extraction. From the results obtained we can highlight that the storage cost is reduced in more than 20 % if compared with Masking Strips and that, when extracting uniform approximations, the extraction process is capable of outperforming previous solutions in 35 %.

6.2. Future work

In this dissertation we have presented different multiresolution techniques that are suitable for different software and hardware characteristics. Nevertheless, there are different aspects of these techniques that can be improved and also several research areas where these techniques can be of benefit.

In this sense, it would be interesting to apply the developed techniques to the representation of other 3D elements like trees or particle systems that can become useful when representing fuzzy phenomena like fire or rain. Moreover, for including a multiresolution model in a graphics application, it is not sufficient to offer the management algorithms. It is also necessary to develop a set of techniques that manage the multiple 3D models that are included in a scene, so that a balance in the rendered geometry is obtained and sharing of geometry is considered.

After analyzing graphics hardware while developing the different multiresolution models, we concluded that the power offered by present graphics hardware opens new possibilities to improve on view-dependent models. The last model presented (Interactive Meshes) was able to offer view-dependent approximations. Thus, we believe that our proposal could be applied to render out-of-core models, whose memory needs require specific routines for their management both in CPU and GPU. As a consequence, the proposed techniques can be combined to offer a new solution to the problem of rendering massive models.

From a different perspective, it is worth mentioning that one of the main objectives of the presented work was the development of a multiresolution model on GPU. Although the proposed solutions have proven to be satisfactory, it is our interest to port our code to CUDA, expecting an increase of performance. CUDA (Compute Unified Device Architecture) is a recent technology devised by nVidia with the final aim of making the most of the huge processing capabilities of current graphics cards [110]. This way, instead of employing a large number of computers, it is possible to resort to graphics processors to make mathematical calculations. CUDA has been designed with the objective of making the most of the great processing capacity of the current graphics

cards to solve problems with a high computational load [110].

The problems presented by graphic applications do not generally require a very high processing capacity, contrary to the problems that have been traditionally solved through GPGPU techniques. Even though, the use of CUDA in a graphics application is very interesting because of the way in which information can be accessed and shared between the different processes running in parallel. In CUDA, a thread has its own processor, variables (registers), processor states, etc. A block of threads is represented as a virtual multiprocessor. The blocks can be run in any order in a concurrent or sequential way. The memory is shared between the threads; in a similar way, there is shared memory between the blocks of a kernel. This means that it is possible to work on the same data in different threads and in different blocks, besides being able to do it in an asynchronous way with the CPU. This supposes a great advantage with respect to the previous architecture. Moreover, it also offers promising possibilities for the development of new solutions or the improvement of previous level-of-detail methods.

Following with this line of work on the GPU, it is worth mentioning that the appearance of Directx 11 will involve further advances in graphics. Among the new stages that the rendering pipeline will include, we highlight the tessellation unit, which will be able to produce semi-regular tessellations [111]. This feature can be directly used as a multiresolution technique to offer view-dependent levels of detail very easily. Thus, we believe that this unit will be key in the next generation of multiresolution techniques.

6.3. Publications

For assessing the work presented throughout this Ph.D. dissertation, this section lists the different publications obtained while developing the thesis, as well as other publications not directly related and a list of research projects that have funded the development.

Regarding the publications related to this dissertation, we highlight:

- Journal Publications:

  - **Interactive Visualization of Meshes on the GPU**
    O. Ripollés, M. Chover, F. Ramos
    The Visual Computer, Under Review. Impact factor (JCR 2008): 1.061.

  - **Rendering continuous level-of-detail meshes by Masking Strips**
    O. Ripollés, M. Chover, J. Gumbau, F. Ramos, A. Puig-Centelles
    Graphical Models, vol. 71 (5), pp. 169-196, 2009. Impact factor (JCR

2008): 0.913.

- **Optimizing the management of continuous level-of-detail models on GPU**
  O. Ripollés, M. Chover
  Computers & Graphics-UK, vol. 32 (3), pp. 307-319, 2008. Impact factor (JCR 2008): 0.731.

- Book Chapters:

  - **Sliding-Tris: A Sliding Window Level-of-Detail Scheme**
    O. Ripollés, M. Chover, F. Ramos
    ICCS 2008. Lecture Notes in Computer Science 5102, pp. 5-14, 2008.

  - **Vertex Buffer Objects: almacenamiento de geometría en la memoria de la tarjeta gráfica**
    O. Ripollés
    OpenGL en fichas II: Aspectos Avanzados, 2008.

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

  - **Implementación eficiente de LodStrips**
    F. Ramos, M. Chover, O. Ripollés, C. Granell
    Métodos Informáticos Avanzados. Publicaciones Universitat Jaume I, p. 213, 2007.

- Conferences:

  - **View-Dependent Multiresolution Modeling on the GPU**
    O. Ripollés, J. Gumbau, M. Chover, F. Ramos, A. Puig-Centelles
    17^{th} Winter School of Computer Graphics (WSCG), pp. 121-126, 2009.

  - **Multiresolution Modeling: A Technique for Efficient Geometry Processing in Real-Time Applications**
    F. Ramos, M. Chover, O. Ripollés
    Visualization, Imaging and Image Processing (VIIP), pp. 125-130, 2008.

- **LODManager: a framework for rendering multiresolution models in real-time applications**
  J. Gumbau, O. Ripollés, M. Chover
  15^{th} Winter School of Computer Graphics (WSCG), pp. 39-46, 2007.

- **Efficient Implementation of LodStrips**
  F. Ramos, M. Chover, O. Ripollés, C. Granell
  Visualization, Imaging and Image Processing (VIIP), pp. 365-370, 2006.

- **Búsqueda de tiras para modelos multirresolución estáticos**
  O. Ripollés, M. Chover
  Congreso Español de Informática Gráfica (CEIG), pp. 117-123, 2005.

- **Quality Strips for Models with Level of Detail**
  O. Ripollés, M. Chover, F. Ramos
  Visualization, Imaging and Image Processing (VIIP), pp. 268-273, 2005.

Other publications on multiresolution which are not directly related have been:

- Journal Publications:

  - **Creation and Control of Rain in Virtual Environments**
    A. Puig-Centelles, O. Ripollés, M. Chover
    The Visual Computer. Accepted Manuscript. Impact factor (JCR 2008): 1.061.

  - **A Tool for the Creation and management of level-of-detail models 3D applications**
    O. Ripollés, F. Ramos, M. Chover, J. Gumbau, R. Quiros
    WSEAS Transactions on Computers, vol. 7 (7), pp. 1020-1029, 2008.

  - **A Clustering Framework for Real-Time Rendering of Tree Foliage**
    C. Rebollo, I. Remolar, M. Chover, J. Gumbau, O. Ripollés
    Journal of Computers, vol. 2 (4), pp. 57-67, 2007.

- Conferences:

  - **Simulación de Lluvia sobre Escenas Dinámicas**
    N. Sunyer, A. Puig-Centelles, O. Ripollés, M. Chover, M. Sbert

Congreso Español de Informática Gráfica (CEIG), 2009.

- **Optimizing the Management and Rendering of Rain**
  A. Puig-Centelles, O. Ripollés, M. Chover
  Int. Conf. on Computer Graphics Theory and Applications (GRAPP), pp. 373-378, 2009.

- **Automatic Terrain Generation with a Sketching Interface**
  A. Puig-Centelles, P. A. C. Varely, O. Ripollés, M. Chover
  15^{th} Winter School of Computer Graphics (WSCG), pp. 39-46, 2009.

- **Multiresolution Techniques for Rain Rendering in virtual Environments**
  A. Puig-Centelles, O. Ripollés, M. Chover
  Int. Symp. on Computer and Information Sciences (ISCIS), pp. 1-4, 2008.

- **Técnicas para visualización de lluvia en entornos virtuales**
  A. Puig-Centelles, O. Ripollés, M. Chover
  Congreso Español de Informática Gráfica (CEIG), pp. 159-167, 2008.

- **Educational instant messaging in a 3D environment**
  A. Puig-Centelles, O. Ripollés, M. Chover, P. Prades
  Int. Conf. on Cognition and Exploratory Learning in Digital Age (CELDA), pp. 49-56, 2007.

- **Fast Rendering of Leaves**
  C. Rebollo, J. Gumbau, O. Ripollés, M. Chover, I. Remolar
  Computer Graphics and Imaging (CGIM), pp. 46-53, 2007.

- **An efficient continuous level of detail model for foliage**
  C. Rebollo, I. Remolar, M. Chover, O. Ripollés
  14^{th} Winter School of Computer Graphics (WSCG), pp. 335-342, 2006.

Finally, it is worth mentioning that the work presented in this dissertation is embedded within several research projects:

- **Contenido Inteligente para Aplicaciones de Realidad Virtual: una Aproximación Basada en Geometría**
  Ministerio de Educación y Ciencia, (TIN2007-68066-C04-02), 2007 - 2010.

- **Geometría Inteligente**
  Fundació Caixa Castelló-Bancaixa (P1 1B2007-56), 2007 - 2010.
- **GAMETOOLS - Advanced Tools for Developing Highly Realistic Computer Games**
  Unión Europea (IST-2-004363), 2004 - 2007.
- **Técnicas de aceleración en gráficos por ordenador y su aplicación a la visualización de especies vegetales**
  Fundació Caixa Castelló-Bancaixa (P1 1B2002-12), 2002 - 2004.

Bibliography

[1] D. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics Application*, 21(3):24–35, 2001.

[2] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, Inc., 2003.

[3] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 269–276, 1999.

[4] P. Alliez and C. Gotsman. Recent advances in compression of 3d meshes. In *Proceedings of the Symposium on Multiresolution in Geometric Modeling*, pages 3–26. Springer-Verlag, 2003.

[5] Z. Karni, A. Bogomjakov, and C. Gotsman. Efficient compression and rendering of multi-resolution meshes. In *Proceedings of the conference on Visualization '02*, pages 347–354, 2002.

[6] J. Kim, S. Choe, and S. Lee. Multiresolution random accessible mesh compression. *Computer Graphics Forum (Eurographics 2006)*, 25(3):323–332, 2006.

[7] J. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 10(19):547–554, 1976.

[8] E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications. In *Eurographics Tutorial Notes*, volume 16(3). Eurographics, 1997.

[9] J. Ribelles, A. López, O. Belmonte, I. Remolar, and M. Chover. Multiresolution modeling of arbitrary polygonal surfaces: a characterization. *Computers & Graphics*, 26(3):449–462, 2002.

[10] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, 2005.

[11] M. Ekman, F. Warg, and J. Nilsson. An in-depth look at computer performance growth. Technical Report 04-9, Department of Computer Science and Engineering, Chalmers University of Technology, 2004.

[12] T. Funkhouser and C. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computers & Graphics*, 27(Annual Conference Series):247–254, 1993.

[13] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. Technical report, Centre National de la Recherche Scientifique, Paris, France, 1996.

[14] K. Low and T. Tan. Model simplification using vertex-clustering. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 75–81, 1997.

[15] J. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 455–465. Springer-Verlag, 1993.

[16] H. Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, 1996.

[17] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, 1997.

[18] F. Ramos, M. Chover, O. Ripollés, and C. Granell. Continuous level of detail on graphics hardware. In *Discrete Geometry for Computer Imagery*, volume 4245, pages 460–469, 2006.

[19] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. *Computers & Graphics*, 27(Annual Conference Series):19–26, 1993.

[20] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, 1997.

[21] W. J. Schroeder. A topology modifying progressive decimation algorithm. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 205–212, 1997.

[22] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *SIGGRAPH '98*, pages 115–122, 1998.

[23] M. Garland and P. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization'98: Proceedings of the conference on Visualization*, pages 263–269, 1998.

[24] C. Gonzalez, J. Gumbau, M. Chover, and P. Castelló. Mesh simplification for interactive applications. In *Proc. of 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2008)*, pages 87–91, 2008.

[25] C. Ha Lee, A. Varshney, and D. Jacobs. Mesh saliency. *ACM Transactions on Graphics*, 24(3):659–666, 2005.

[26] P. Lindstrom and G. Turk. Image-driven simplification. *ACM Transactions on Graphics*, 19(3):204–241, 2000.

[27] D. Luebke and B. Hallen. Perceptually-driven simplification for interactive rendering. In *12th Eurographics Workshop on Rendering*, pages 223–234, 2001.

[28] N. Williams, D. Luebke, J. D. Cohen, M. Kelley, and B. Schubert. Perceptually guided simplification of lit, textured meshes. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 113–121, 2003.

[29] P. Castelló, M. Sbert, M. Chover, and M. Feixas. Viewpoint entropy-driven simplification. In *Proc. of 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2007)*, pages 249–256, 2007.

[30] P. Castelló, M. Chover, M. Sbert, and M. Feixas. Applications of information theory to computer graphics (part 7). In *Eurographics Tutorial Notes*, volume 2, pages 891–902. Eurographics, 2007.

[31] M. B. Dillencourt. Finding hamiltonian cycles in delaunay triangulations is np-complete. *Discrete Applied Mathematics*, 64(3):207–217, 1996.

[32] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization*, pages 319–326, 1996.

[33] K. Akeley, P. Haeberli, and D. Burns. tomesh.c: C program on sgi developer's toolbox cd, 1990.

[34] X. Xiang, M. Held, and J. Mitchell. Fast and effective stripification of polygonal surface models. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 71–78, 1999.

[35] C. Beeson and J. Demer. Nvtristrip, library version 1.1. `http://developer.nvidia.com`, 2003.

[36] P. Vanecek and I. Kolingerova. Technical section: Comparison of triangle strips algorithms. *Computers & Graphics*, 31(1):100–118, 2007.

[37] D. Eppstein and M. Gopi. Single-strip triangulation of manifolds with arbitrary topology. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 455–456. ACM, 2004.

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

[39] O. Belmonte, J. Ribelles, I. Remolar, and M. Chover. Búsqueda de tiras de triángulos guiadas por un criterio de simplificación. In *Actas del X Congreso Español de Informática Gráfica (CEIG 2000)*, pages 51–64, 2000.

[40] O. Ripollés, M. Chover, and F. Ramos. Quality strips for models with level of detail. In *Proceedings of Visualization, Imaging and Image Processing (VIIP)*, pages 268–273, 2005.

[41] J. Chhugani and S. Kumar. Geometry engine optimization: cache friendly compressed representation of geometry. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 9–16, 2007.

[42] P. Sander, D. Nehab, and J. Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3):89, 2007.

[43] T. Fautré. Tri stripper algorithm. `http://users.pandora.be/tfautre/softdev/tristripper/`, 2002.

[44] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. In *GRIN'01: No description on Graphics interface 2001*, pages 81–90, 2001.

[45] S. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 886–893, 2005.

[46] 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.

[47] 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.

[48] L. Borgeat, G. Godin, F. Blais, P. Massicotte, and C. Lahanier. Gold: interactive display of huge colored and textured models. *ACM Transactions on Graphics*, 24(3):869–877, 2005.

[49] H. Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22(1):27–36, 1998.

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

[51] C. C. Chen and J. H. Chuang. Texture adaptation for progressive meshes. *Computer Graphics Forum (Eurographics 2006)*, 25(3):343–350, 2006.

[52] J. El-Sana, E. Azanli, and A. Varshney. Skip strips: maintaining triangle strips for view-dependent rendering. In *VIS'99: Proceedings of the conference on Visualization'99*, pages 131–138, 1999.

[53] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, and M. Fernández. Efficiently using connectivity information between triangles in a mesh for real-time rendering. *Future Generation Computer Systems, Special issue on Computer Graphics and Geometric Modeling*, 20(8):1263–1273, 2004.

[54] F. Ramos and M. Chover. Lodstrips: Level of detail strips. In *International Conference on Computational Science*, pages 107–114, 2004.

[55] P. Turchyn. Memory-efficient sliding window progressive meshes. In *Proc. of 15-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2007)*, pages 33–40, 2007.

[56] J. Ji, E. Wu, S. Li, and X. Liu. Dynamic lod on GPU. In *Computer Graphics International*, pages 108–114, 2005.

[57] A. J. Stewart. Tunneling for triangle strips in continuous level-of-detail meshes. In *GRIN'01: Graphics interface 2001*, pages 91–100, 2001.

[58] M. Shafae and R. Pajarola. Dstrips: Dynamic triangle strips for real-time mesh simplification and rendering. In *Proceedings Pacific Graphics 2003*, pages 271–280, 2003.

[59] H. Hoppe. View-dependent refinement of progressive meshes. *Computers & Graphics*, 31(Annual Conference Series):189–198, 1997.

[60] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *VIS'98: Proceedings of the conference on Visualization'98*, pages 43–50, 1998.

[61] R. Pajarola. Fastmesh: Efficient view-dependent meshing. In *PG '01: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, page 22. IEEE Computer Society, 2001.

[62] J. El-Sana and Y. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):139–150, 2000.

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

[64] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *SI3D'03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 93–102, 2003.

[65] H. Birkholz. Out of core continuous lod-hierarchies for large triangle meshes. In *Proc. of 14-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2006)*, pages 95–100, 2006.

[66] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3D scanning of large statues. In *Siggraph 2000, Computer Graphics Proceedings*, pages 131–144, 2000.

[67] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In *SIGGRAPH*, pages 796–803, 2004.

[68] S. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-vdr: Interactive view-dependent rendering of massive models. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 131–138. IEEE Computer Society, 2004.

[69] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. In *IEEE Visualization*, pages 207–214, 2005.

[70] P. Sander and J. Mitchell. Progressive buffers: view-dependent geometry and texture lod rendering. In *SGP '05: Proceedings of the third Eurographics symposium on Geometry processing*, page 129, 2005.

[71] K. Niski, B. Purnomo, and J. Cohen. Multi-grained level of detail using a hierarchical seamless texture atlas. In *Proceedings of I3D'07:*, pages 153–160, 2007.

[72] Y. Livny, M. Press, and J. El-Sana. Interactive GPU-based adaptive cartoon-style rendering. *The Visual Computer*, 24(4):239–247, 2008.

[73] L. Hu, P. Sander, and H. Hoppe. Parallel view-dependent refinement of progressive meshes. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 169–176, New York, NY, USA, 2009.

[74] T. Boubekeur and C. Schlick. A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum*, 27(1):102–114, 2008.

[75] H. Lorenz and J. Döllner. Dynamic mesh refinement on GPU using geometry shaders. In *Proc. of 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2008)*, pages 97–104, 2008.

[76] M. Schwarz and M. Stamminger. Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum*, 28(2):365–374, 2009.

[77] C. Dyken, M. Reimers, and J. Seland. Real-time GPU silhouette refinement using adaptively blended bézier patches. *Computer Graphics Forum*, 27(1):1–12, 2008.

[78] L. Buatois, G. Caumon, and B. Lévy. GPU accelerated isosurface extraction on tetrahedral grids. In *International Symposium on Visual Computing*, pages 383–392, 2006.

[79] L. Shiue, I. Jones, and J. Peters. A real-time GPU subdivision kernel. *ACM Transactions on Graphics*, 24(3):1010–1015, 2005.

[80] M. Guthe, A. Balázs, and R. Klein. GPU-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, 2005.

[81] T. Boubekeur and C. Schlick. Generic mesh refinement on GPU. In *Graphics Hardware*, pages 99–104, 2005.

[82] M. Bokeloh and M. Wand. Hardware accelerated multi-resolution geometry synthesis. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 191–198, New York, NY, USA, 2006. ACM.

[83] C. DeCoro and N. Tatarchuk. Real-time mesh simplification using the GPU. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 161–166, 2007.

[84] X. Gu, S. Gortler, and H. Hoppe. Geometry images. In *Proceedings of SIGGRAPH'02*, pages 355–361, 2002.

[85] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155, 2003.

[86] F. Losasso, H. Hoppe, S. Schaefer, and J. Warren. Smooth geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 138–145, 2003.

[87] M. Wloka. Lag in multiprocessor virtual reality. *Presence*, 4(1):50–63, 1995.

[88] M. Reddy. Reducing lags in virtual reality systems using motion-sensitive level of detail. In *Proceedings of the second UK VR-SIG Conference*, pages 25–31, 1994.

[89] H. Yee, S. Pattanaik, and D. Greenberg. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. In *ACM Transactions on Graphics*, pages 39–65. 2001.

[90] R. Danforth, A. Duchowski, R. Geist, and E. McAliley. A platform for gaze-contingent virtual environments. In *Smart Graphics (Papers from the 2000 AAAI Spring Symposium, Technical Report SS-00-04)*, pages 66–70, 2000.

[91] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352, 2000.

[92] C. Andújar, C. Saona-Vázquez, I. Navazo, and P. Brunet. Integrating occlusion culling with levels of detail through hardly-visible sets. *Computer Graphics Forum (Proceedings of Eurographics'00)*, 3:499–506, 2000.

[93] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, 1998.

[94] A. Mason and E. Blake. A graphical representation of the state spaces of hierarchical level-of-detail scene descriptions. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):70–75, 2001.

[95] M. Wimmer and D. Schmalstieg. Load balancing for smooth lods. Technical Report TR-186-2-98-31, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1998.

[96] E. Gobbetti and E. Bouvier. Time-critical multiresolution scene rendering. In *Proceedings IEEE Visualization*, pages 123–130, 1999.

[97] J. Swan II, J. Arango, and B. Nakshatrala. Interactive distributed hardware-accelerated lod-sprite terrain rendering with stable frame rates. In *Proceedings SPIE, Visualization and Data Analysis 2002*, volume 4665, pages 177–188, 2002.

[98] W. Baxter III, A. Sud, N. Govindaraju, and D. Manocha. Gigawalk: interactive walkthrough of complex environments. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 203–214, 2002.

[99] C. Woolley, D. Luebke, B. Watson, and A. Dayal. Interruptible rendering. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 143–151, 2003.

[100] C. Zach. Integration of geomorphing into level of detail management for realtime rendering. In *SCCG '02: Proceedings of the 18th Spring Cdukonference on Computer graphics*, pages 115–122, 2002.

[101] C. Zach, S. Mantler, and K. Karner. Time-critical rendering of discrete and continuous levels of detail. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 1–8, 2002.

[102] F. Ramos, M. Chover, O. Ripollés, and C. Granell. Efficient implementation of lodstrips. In *Visualization, Imaging, and Image Processing*, pages 365–370, 2006.

[103] J. Brewer and J. Sekel. PCI express technology. DELL Technology White Paper, `http://www.dell.com/downloads/global/vectors/2004_pciexpress.pdf`, 2004.

[104] R. Wilson. *Introduction to graph theory.* Academic Press, New York, 1972.

[105] J. Ribelles, M. Chover, A. López, and J. Huerta. A first step to evaluate and compare multiresolution models. In *Short Papers and Demos of Eurographics'99*, pages 230–232, 1999.

[106] P. Castelló, F. Ramos, and M. Chover. A comparative study of acceleration techniques for geometric visualization. In *International Conference on Computational Science (2)*, pages 240–247, 2005.

[107] G. Junker. *Pro OGRE 3D Programming (Pro).* Apress, Berkely, CA, USA, 2006.

[108] D. Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, 25(3):724–734, 2006.

[109] B. Purnomo. Amd tootle ver 2.0. `http://ati.amd.com/developer/tootle.html`, 2008.

[110] nVidia CUDA compute unified device architecture - programming guide version 2.0. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`, 2007.

[111] S. Tariq. D3D11 tesselation. Game Developers Conference. Session: Advanced Visual Effects with Direct3D for PC, http://developer.download.nvidia.com/presentations/2009/GDC/GDC09_D3D11Tessellation.pdf, 2009.