

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK
Kasteelpark Arenberg 10, B-3001 Leuven

PLATFORM INDEPENDENT PERFORMANCE METRICS FOR 3D TERMINAL QUALITY OF SERVICE

Jury:
Prof. dr. ir. P. Van Houtte, voorzitter
Prof. dr. ir. R. Lauwereins, promotor
Prof. dr. ir. F. Catthoor, promotor
Prof. dr. ir. J. Cornelis
Prof. dr. ir. L. Van Eyken
Prof. dr. ir. L. Van Gool
Prof. dr. ir. P. Dutré
Prof. Asoc. F. Morán Burgos (UPM Madrid)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Nicolaas Tack

UDC 681.3*H51

April 2006

©Katholieke Universiteit Leuven - Faculteit Ingenieurswetenschappen
Arenbergpark 10, B-3001 Leuven - Heverlee

Alle rechten voorbehouden. Niets uit deze uitgave mag vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2006/7515/24
ISBN 90-5682-690-5

Acknowledgements

I received help, support and advice from many people during my Ph.D. program. Without them, my work would have been much harder and I wish to thank them.

First of all, my gratitude goes to my advisor, Dr. Gauthier Lafruit who introduced me in the exiting world of 3D graphics. His great help and dedication have helped me finishing my thesis. I also want to thank my supervisors, Prof. Francky Catthoor for his guidance and suggestions and Prof. Rudy Lauwereins for giving me the opportunity of doing this thesis.

My appreciation also goes to Prof. Jan Cornelis, Prof. Luc Van Eyken and Prof. Luc Van Gool for proofreading my thesis and for the useful comments and suggestions. I also want to thank Prof. Philip Dutré to be part of the jury and Prof. Paul Van Houtte to be the chairman of the jury. Special thanks go to Prof. Francisco Morán Burgos, who flew all the way from Madrid for my presentation. I really appreciate this!

I also want to express the pleasure that I had in working with my former and current colleagues of the Multimedia group at IMEC. Thank you (alphabetically) Adrian, Adrian, Aris, Bart, Bart, Bert, Carolina, Eric, Guan, Iole, Jan, Jiangbo, Johan, Kristof, Wilfried, Wolfgang, Yann and Xin.

Tevens ben ik het IWT Vlaanderen erkentelijk voor de financiële steun.

Eén van de belangrijkste componenten van een geslaagde thesis is om die af te werken. Dit zou nooit mogelijk geweest zijn zonder de vriendschap en liefde van mijn vrouw Elke. Dank je!

Leuven 2006,

Klaas Tack

Abstract

In the era of internet and multimedia communications, multimedia content will be accessible from anywhere at anytime. Consequently, a flurry of terminals with large differences in terminal capabilities will be used to access the multimedia content. As a result, content creators and providers do not longer know in advance the terminal that will consume their multimedia content. This makes it challenging for them to guarantee a satisfying quality for this content on all these very different terminals. In addition, the processing time for 3D graphics applications can easily vary for the same content over one order of magnitude. Fixed quality content satisfying the worst-case performance conditions of the terminal could therefore reveal being qualitatively inappropriate in the terminals average working conditions. One of the key technologies to solve this problem is scalable content, which allows a gradual trade-off between quality and complexity by encoding the 3D data with different resolution scales in an embedded bit stream. The decision of the resolution of the different objects is however left to an optimisation algorithm, which is part of the terminal QoS and which task it is to maximise the visual quality for a given target frame time. Unfortunately, prior art techniques typically use very simple heuristics to assess the quality of the content and to estimate the frame time. The realism/accuracy of the models is sacrificed for simplicity in the online problem solving. On the other hand, accurate error metrics for 3D graphics content exist. These metrics are typically used to compare different 3D simplification algorithms, but are too complex for interactive applications; therefore, this manuscript presents a new optimisation framework that combines the better accuracy of 3D simplification algorithms with the simplicity of previous optimisation algorithms. The optimisation is split into off-line preprocessing steps, which execute the complex computations and simple online steps that combine the off-line results of the visible objects into a global trade-off between quality and cost. The result is a better visual quality for the same processing power compared with previous art.

Samenvatting

In het tijdperk van de internet communicatie zal multimedia informatie toegankelijk zijn van op om het even welke plaats en tijdstip. De gebruiker zal daarvoor een brede waaier van toestellen gebruiken met grote verschillen in mogelijkheden. Het is voor de aanbieders van de multimedia informatie daarom onmogelijk geworden om op voorhand het toestel te kennen dat gebruikt zal worden om hun multimedia informatie op te vragen. Dit maakt het voor hen zeer uitdagend om een geschikte kwaliteit te garanderen voor deze brede waaier van toestellen. Bovendien kan de nodige rekentijd voor '3D graphics' toepassingen zeer sterk variëren. '3D graphics' data waarvan de kwaliteit vast is ingesteld voor de slechts mogelijke omstandigheden op het toestel, zal daarom een veel te lage kwaliteit hebben in de gemiddelde omstandigheden. Eén belangrijke pijler voor de oplossing van dit probleem is schaalbare data. Schaalbaarheid laat een graduele uitwisseling van kwaliteit en rekentijd toe door de 3D objecten met verschillende kwaliteitsniveaus te coderen in één enkele stroom van bits. De eigenlijke beslissing van het juiste kwaliteitsniveau voor elk object in de 3D wereld wordt overgelaten aan een optimalisatiealgoritme dat deel is van de toestel QoS en dat tot taak heeft de visuele kwaliteit te maximaliseren voor een gegeven maximale rekentijd. Helaas werden voor dit algoritme in het verleden zeer eenvoudige heuristieken gebruikt voor het modelleren van de visuele kwaliteit en rekentijd. Het realisme en de nauwkeurigheid van de modellen werd opgeofferd voor de eenvoud van het algoritme. Nauwkeurige kwaliteitsmaten voor het vergelijken van verschillende algoritmen voor de vereenvoudiging van 3D objecten bestaan nochtans. Deze maten zijn echter te complex om rechtstreeks gebruikt te worden in interactieve toepassingen. Deze thesis stelt daarom een nieuw optimalisatiealgoritme voor die de nauwkeurigheid van de complexe kwaliteitsmaten combineert met de eenvoud van de heuristieken. De optimalisatie is daarom gesplitst in offline voorbereidende stappen die de complexe taken uitvoeren en eenvoudige online stappen die de resultaten van de voorbereidende stappen combineren. Het resultaat is een betere visuele kwaliteit voor dezelfde rekentijd.

Contents

Abstract **iii**

Samenvatting **v**

Contents **vii**

List of figures **xi**

List of Tables **xvii**

List of Symbols **xix**

1 Introduction **1**

1.1 Motivation and problem formulation 1

1.2 Related work 3

1.3 An introduction to 3D computer graphics 6

1.4 Overview of approach and outline of the text 9

1.5 Summary of Contributions 13

2 Optimisation Framework **15**

2.1 Related work 16

2.2 The framework in a nutshell 17

2.3 Off-line Preprocessing steps 20

2.3.1 Step 1: Decomposing the object 20

2.3.2 Steps 2 and 3: Derive a view-independent distortion plot . . . 21

2.3.3 Steps 4 and 5: Derive a view-dependent distortion plot 23

2.4 Online work: Derive a global scene distortion plot 24

2.4.1 Gradient descent combination 24
2.4.2 Complexity analysis 26
2.5 Example plots 28
2.6 Conclusion 30

3 Modelling performance **33**
3.1 Related work 34
3.2 Method for Deriving the Software Rendering Performance Model . . . 35
3.2.1 Profiling the Code 36
3.2.2 Extract Data Dependencies 37
3.2.3 Platform Dependencies - the reason for an adaptive model . . 43
3.2.4 Deriving the performance model coefficients T_i 46
3.2.5 Results 50
3.3 Method for Deriving the Hardware Rendering Performance Model . . 52
3.3.1 Measuring time on the GPU 53
3.3.2 Modelling the vertex processor 54
3.3.3 Modelling the rendering 58
3.3.4 Performance model 60
3.3.5 Results 61
3.4 Conclusions 64

4 Modelling Visual Quality **67**
4.1 Overview 67
4.1.1 The influence of the geometric model 68
4.1.2 View-dependent quality parameters 68
4.1.3 Influence of the shading and illumination models 70
4.1.4 The influence of the texture 70
4.2 Available quality metrics 72
4.2.1 Analytic benefit models 72
4.2.2 Benefit models based on the output image 75
4.3 The proposed quality metric 76
4.3.1 The quality of the geometric approximation 76
4.3.2 Extra scaling parameters 81

4.4 Results 85
4.5 Conclusion 87

5 Multi-resolution mesh representation 93
5.1 Related work 93
5.1.1 Discrete Level Of Detail 94
5.1.2 Continuous Level Of Detail 94
5.1.3 View-dependent Level Of Detail 95
5.1.4 Wavelets and Subdivision Surfaces 96
5.2 Continuous Level Of Detail with Subdivision Surfaces 98
5.2.1 The Selector - Importance criteria 99
5.2.2 The selector - Subdivision code 102
5.2.3 Summary of the selector parameters 106
5.2.4 The Convertor 106
5.2.5 Non-uniform decoder 107
5.3 Performance analysis and modelling 108
5.3.1 Decoder activation 108
5.3.2 Selector and convertor complexity 111
5.3.3 Decoding complexity 113
5.4 Cancelling the adaptation overhead 114
5.5 Conclusions 116

6 Optimisation over Successive Frames 119
6.1 Putting all the pieces together 120
6.2 Continuous LOD with non-uniform subdivision surfaces 124
6.3 Discrete LOD with Uniform Subdivision 126
6.4 Stability and sensitivity of the system 131
6.5 Conclusions 131

7 Control energy consumption **133**
7.1 Related work . . . 134
7.2 The simulated architecture . . . 135
7.2.1 The architecture . . . 136
7.2.2 Texture memory . . . 137
7.2.3 Energy consumption . . . 139
7.3 The framework extension . . . 139
7.4 Conclusions . . . 144

8 Conclusions and suggestions for future work **145**
8.1 Conclusions . . . 145
8.2 Future work . . . 147

A Presentation of the content **149**

B Nederlandstalige samenvatting **155**

List of Figures

1.1 Universal Media Access 2
1.2 MPEG-4 coding 4
1.3 3D model as a combination of geometry and appearance 7
1.4 The 3D graphics pipeline 8
1.5 Mesh simplification illustrated on the Stanford Bunny 9
1.6 Complex error metric versus simple accuracy heuristic 10
1.7 Comparison of the optimisation 11
1.8 High-level overview of the system 11
1.9 Flow chart for reading the text. 13

2.1 Overview of the preprocessing work 18
2.2 Overview of the run-time work. 19
2.3 The decomposition of an object 20
2.4 A complete distortion plot 22
2.5 A pruned distortion plot 23
2.6 Visibility sphere 24
2.7 Convex relation of execution time with number of triangles 26
2.8 Execution time of the optimisation 27
2.9 Distortion plots for single object 29
2.10 Distortion plots for a complete scene. 29
2.11 Online distortion plots for a city scene. 30

3.1 Complexity distribution of the software 3D pipeline 36
3.2 Legend for the control flowcharts. 38
3.3 Flowchart of the code of the vertex processor. 39

3.4 Flowcharts for rasterisation . . . 41
3.5 Deriving performance model parameters . . . 42
3.6 Accuracy of pixel estimation . . . 43
3.7 Locality as a function of texture size . . . 44
3.8 Locality as a function of triangle orientation. . . . 45
3.9 Number of cache misses as a function of texture size . . . 45
3.10 Extra pixel processing because of texture size and orientation . . . 46
3.11 Calibration of the vertex processing model coefficients. . . . 47
3.12 Calibration of the triangle rasterisation model coefficients. . . . 48
3.13 Non-linearities because of the platform architecture. . . . 49
3.14 Introduction of timers in the code . . . 49
3.15 Online correction of the performance model coefficients . . . 50
3.16 Histogram of errors on the embedded device . . . 51
3.17 The measurements versus prediction for the city scene . . . 51
3.18 High-level overview of a hardware accelerated rendering system. . . . 52
3.19 Bottleneck identification in a GPU. . . . 54
3.20 Vertex time for different types of light sources. . . . 55
3.21 Vertex time as a function of the number of light sources . . . 56
3.22 Calibration of the time per vertex and per light. . . . 56
3.23 6 triangles using the same vertex . . . 57
3.24 FIFO cache simulator for the cache-miss rate m . . . 57
3.25 Test with two meshes: 1 mesh is completely invisible . . . 58
3.26 Pixel and rasterisation time for a small texture . . . 59
3.27 Pixel and rasterisation time for a large texture . . . 59
3.28 Histogram of uncorrected errors on the PC (max model) . . . 62
3.29 The result for the model on the city scene (max) . . . 63
3.30 Histogram of errors of the performance model on the PC (sum) . . . 64

4.1 Visual quality as a function of the number of triangles . . . 68
4.2 Influence of the distance to the viewpoint on the visual quality . . . 69
4.3 Influence of silhouette preservation on the visual quality . . . 69
4.4 Quality as a function of the material for Gouraud shading . . . 71

4.5 Quality as a function of the material for flat shading 71
4.6 texture mapping illustrated 71
4.7 Mesh simplification and texturing 73
4.8 The effect of texture masking 73
4.9 The effect of the texture resolution 73
4.10 The surface to surface error on the Stanford Bunny 78
4.11 The error metric without the invisible triangles 79
4.12 The view-dependent error metric on the Stanford Bunny 82
4.13 The variation of the quality because of different materials 84
4.14 The scaling factor f_c of the shoe is the average value of the ratio (b) of the curves in (a). 85
4.15 The method with scaling factor fails for textured objects 86
4.16 Comparison of proposed quality metric with existing heuristic 86
4.17 Visual comparison: 12000 triangles 88
4.18 Visual comparison: 18000 triangles 89
4.19 Visual comparison: 30000 triangles 90

5.1 Simplification operators. 95
5.2 Subdivision surfaces applied on a sphere 96
5.3 Subdivision of a single triangle. 97
5.4 Displace Subdivision Surfaces 98
5.5 The proposed resolution selector. 99
5.6 Smoothness as a parameter for importance 100
5.7 Sweeping the parameter h_t changes the number of triangles 101
5.8 First dynamic criterion:silhouette 101
5.9 The second dynamic criterion: distance 102
5.10 The seven possible ways for subdividing a triangle 104
5.11 Non-uniform subdivision around the bold line. 104
5.12 Inheritance of the subdivision code 105
5.13 Pseudo code for the normal and simplified conversion of a triangle . . 107
5.14 Pseudo code for drawing a triangle 107
5.15 Data dependencies for the Butterfly subdivision scheme 108
5.16 Comparison of CPU and GPU time for silhouette preservation 109

5.17 GPU most of the time idle for view-dependent rendering 110
5.18 The execution time of the selector 112
5.19 The execution time of the convertor 112
5.20 Block diagram and complexity of the MPEG-4 WSS decoder 113
5.21 Execution time of the WSS synthesis filters 114
5.22 Eliminating CPU overhead 115
5.23 Cancelling the adaptation overhead: results 117

6.1 Legend for the figures of this and the next subsections 120
6.2 The decoding and optimisation of MPEG-4 WSS geometry 120
6.3 Instantaneous control with non-uniform subdivision surfaces. 122
6.4 Execution time for instantaneous adaptation of the city scene. 122
6.5 Execution time for instantaneous adaptation of the Potato scene. . . 123
6.6 Execution time for instantaneous adaptation of the chess scene. . . . 123
6.7 The change of working point A to B is gradually done 125
6.8 The execution time varies because of the gradual control 125
6.9 Execution time for gradual adaptation of the City scene. 127
6.10 Execution time for gradual adaptation of the Mr. Potato scene. . . . 127
6.11 Execution time for gradual adaptation of the Chess scene. 128
6.12 Distortion plot for uniform subdivision in the city scene. 129
6.13 Execution time for uniform adaptation of the City scene. 129
6.14 Execution time for uniform adaptation of the Mr. Potato scene. . . . 130
6.15 Execution time for uniform adaptation of the Chess scene. 130

7.1 Comparison of our proposal with existing work 134
7.2 The ARM performance figures 135
7.3 The simulated GPU architecture: vertex and triangle processing on the left and pixel processing on the right. 136
7.4 Estimated versus simulated rendering time 137
7.5 Texture cache miss rate comparison 138
7.6 Energy consumption as a function of throughput 139
7.7 Global distortion plot for 5 different viewpoints 140
7.8 The variation of the triangle cost for a constant quality 141

7.9 Energy, execution time and quality for a walk through the city scene. 142
7.10 Example frames for different viewpoints 143
7.11 The character from the city scene shown with different error E 144

A.1 The scenes used in this manuscript 150
A.2 Uncontrolled execution time for the city scene 152
A.3 Uncontrolled execution time for the Chess scene 152
A.4 Uncontrolled execution time for the Mr. Potato scene 153

B.1 Een 3D model waarbij geometrie en kleur gecombineerd worden. . . . 156
B.2 De vereenvoudiging van een set driehoeken toegepast op het bekende Stanford konijn. 156
B.3 Vergelijking van de voorgesteld kwaliteitsmetriek 159
B.4 Het verschil tussen de voorgestelde en traditionele algoritmen 163

List of Tables

3.1 Model error for the embedded system 50
3.2 Error of max model for the hardware accelerated rendering 62
3.3 Error of sum model for the hardware accelerated rendering 63

5.1 Effect of silhouette preservation on rendering (software) time 110

Glossary

Adaptive (non-uniform) subdivision: Adaptive subdivision only subdivides the important regions of the mesh.

Compressed progressive mesh: The compressed form of a progressive mesh.

Culling: Culling is the process of removing everything from a scene that will not contribute to the final image, including things that are behind the observer, off-screen, or, in more advanced systems, hidden behind other objects.

Fragment: A fragment is a pixel-sized element that contains extra information needed for deriving the colour of the pixel next to the pixel itself.

GPGPU: General-Purpose Computation Using Graphics Hardware.

Level of detail (LOD): One of a series of increasingly simplified models that approximate an original shape by trading fidelity for a decrease in cost.

Mesh: A linear approximation of the surface of a 3D object with polygons.

Mesh simplification: the process of generating a 3D object M_i, which is an approximation of the original object M, but with fewer elements.

Mipmapping: Mipmapping features multiple images of a texture at different resolutions to represent surface textures at varying distances from the viewer's perspective: the largest scaled image is placed in the foreground and progressively smaller ones recede toward the background area. Mipmapping helps avoid unwanted jagged edges in an image that can result from using bit map images at different resolutions. MIP comes from the Latin multum in parvo, meaning a multitude in a small space. The term mipmapping is typically used in a rendering context.

Multi-resolution model: A multi-resolution model captures a wide range of LOD's, which can be used to reconstruct any one of those levels on demand. The term multi-resolution model is typically used in a coding context (e.g. coding of geometry).

Pareto curve: A Pareto curve is a collection of Pareto optimal points.

Pareto optimal point: A point is called Pareto optimal in an N-dimensional space of different objectives placed on the axis, when it is impossible to improve one objective without worsening at least one of the other objectives.

Pixel: The term pixel is used for picture element and represents the content of the frame buffer at a specific location.

Progressive mesh: A progressive mesh is a special type of a multi-resolution model, which first stores a low resolution mesh M_0, followed by detail information which is stored by importance and which gradually increases the number of elements.

Rasterization: Rasterization is the process of determining the set of pixels covered by a geometric primitive such as a triangle or a line.

Regular (uniform) subdivision: A sequence of mesh refinement steps, each of which insert new vertices in the middle of the edges and split all triangles into four.

Terminal Quality of Service: A terminal resource management that allows for (continuous) tradeoffs between the available resource budget and the end user's perception.

3D Terminal Quality of Service: Terminal QoS for 3D graphics applications.

Texel: The term texel is used for texture element and represents the content of the texture at a specific location.

Texture: A 2D image that is used as a kind of wallpaper on the surface of the 3D object.

View-dependent LOD: A simplified model created by adjusting the tolerance to a particular set of view parameters, e.g., increasing the accuracy of the approximation close to the viewer and to the visible silhouettes of the model.

X3D: X3D is an Open Standards XML-enabled 3D file format to enable real-time communication of 3D data across all applications and network applications. It has a rich set of features for use in engineering and scientific visualisation, CAD and Architecture, Medical visualisation, Training and simulation, multimedia, entertainment, educational, and more.

Acronyms and Abbreviations

2D	Two dimensional
3D	Three dimensional
3DMC	3D Model Coding
AGP	Accelerated Graphics Port
API	Application Programming Interface
Cg	C for graphics
CPU	Central Processing Unit
CSF	Contrast Sensitivity Function
DIA	Digital Item Adaptation
fps	frames per second
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
JPEG	Joint Photographic Experts Group
LOD	Level Of Detail
MCKP	Multiple Choice Knapsack Problem
MPEG	Moving Picture Experts Group
MSE	Mean Squared Error
PDA	Personal Digital Assistant
PVS	Potential Visible Set
PSNR	Peak Signal to Noise Ratio
QoS	Quality of Service
ROI	Region Of Interest
RMSE	Root Mean Squared Error
SSIM	Structural SIMilarity
sub-ROI	sub-Region Of Interest
STB	Set-Top Box
UMA	Universal Media Access
VRML	Virtual Reality Modelling Language
Vid Mem	Video Memory
WSS	Wavelet Subdivision Surfaces

Chapter 1

Introduction

The purpose of Chapter 1 is to provide the necessary motivation and background information for this PhD manuscript. Chapter 1 also includes a brief overview of the contributions and structure of the text. Section 1.1 presents our motivation and formulates the problem. Section 1.2 discusses previous work that is complementary to the presented work. In addition, each chapter of the rest of this text describes in detail the work that is directly related to the chapter's topic. Section 1.3 introduces the necessary terminology. Section 1.4 presents a high level overview of the proposed solution. Section 1.5 lists the main contributions of the proposed work and the last section describes the organisation of the text.

1.1 Motivation and problem formulation

In the era of internet and multimedia communications, information and multimedia content will be accessible from anywhere at anytime. Consequently, a flurry of networks and terminals with large differences in network and terminal capabilities will be used to access the multimedia content. This principle is called Universal Media Access (UMA) and is illustrated in Figure 1.1, which shows that the networks and terminals that are used for the content consumption will actually depend on the preferences and location of the user.

Because of UMA, content creators and providers do not longer know in advance which networks and terminals will be used to consume their multimedia content. This makes it challenging for them to guarantee that the content can be shown with a satisfying quality in all these very different environments. The response to this problem is an advanced resource management technique that makes on-the-fly trade-offs to match the content bandwidth, the media coding and rendering complexity to the available network and end-system resources, while maximising the overall perceived quality. This process is often called Quality of Service (QoS) management.

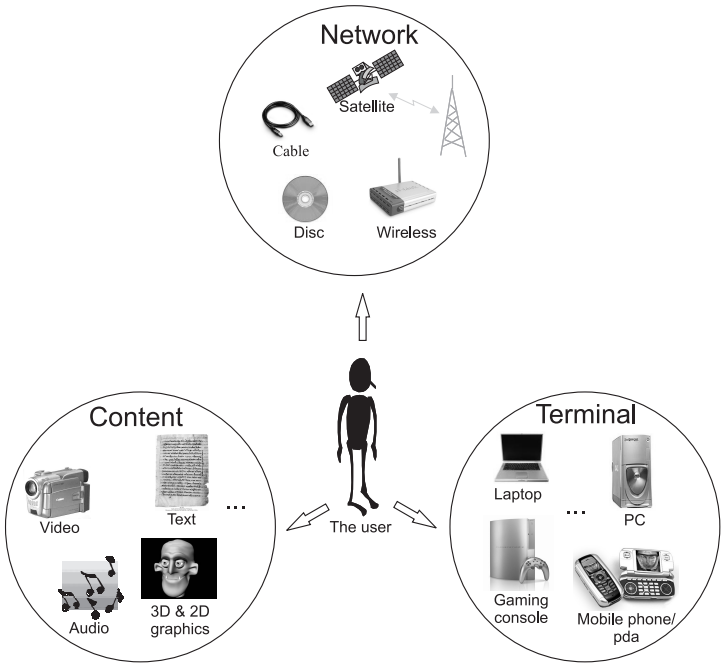


Figure 1.1: Universal Media Access (UMA) enables the user to access rich media (including 3D graphics) over many different networks on many different terminals.

The most well known aspect of QoS management is network QoS, which qualitatively or quantitatively defines a performance contract for the communication network between the service provider and the user application [31]. The network QoS is however not the only issue to be solved. Terminals vary widely in their features such as screen size, resolution, computing power, etc. An advanced terminal resource management system, known as terminal QoS, is therefore also needed to maximise the user's perception with the limited terminal resources [19]. To support network and terminal QoS, the MPEG-21 multimedia framework [49] includes a part on Digital Item Adaptation (DIA), which specifies the adaptation of bit streams of multimedia content according to constraints on the storage, transmission and consumption [154].

However, when dealing with rich media, not all issues can be solved with bit stream adaptation. In 3D graphics applications for instance, the needed performance for the same content can easily vary over one order of magnitude. Brandt et al. [22] for instance show how a 3D graphics application (a simple moving object) can easily change its required processing time by varying only 3 parameters. In benchmarking 3D graphics applications, the OpenGL Performance Characterisation Organisation [30] shows applications with requirements varying by as much as a factor of 10. Fixed quality content, which results from the adapted bit streams and satisfies the worst-case performance conditions on the terminal could therefore reveal being qualitatively inappropriate for the average performance conditions on the terminal. A

Quality of Service (QoS) framework where the content's quality continuously adapts to the terminal's instantaneous workload is thus more appropriate to conciliate performance with the end user's experience.

Because the area of research on QoS is very broad, we have focussed on one major issue, namely the control of the terminal's workload for interactive 3D graphics applications through the control of the 3D content complexity. To solve this issue, we have developed a fast content optimisation framework that maximises the visual quality for a given execution time deadline.

Typical applications that could use this framework are collaborative games [131] or online games with access to the game content through a network. For instance, at home, a user will typically use a high performance PC or gaming console with a broadband connection, but when travelling, the user will typically play the game on a mobile terminal with a wireless connection.

1.2 Related work

Several Quality of Service architectures have already been proposed in the past. Xu et al. [166] have introduced the $2K^Q$ architecture to enable the universal access of multimedia services. It contains a resource monitor which reports the resource availability to the server and the server then selects the correct quality level for the multimedia content. Sabata et al. [126] have presented a QoS driven resource management as an end-to-end allocation and scheduling of resources to applications, based on the QoS requirements of the applications. When their system is faced with inadequate resources, it is able to make trade-offs between the various aspects that are important for the application's quality, such as the frame rate and visual quality in 3D graphics applications and the frame delay and frame jitter in video on demand applications. Both systems however lack on-the-fly adaptation, which is needed for dynamic 3D graphics applications. On-the-fly adaptation of the content must be fast, while negotiations with the server take too much time. A content adaptation at the client side, which refines the decisions taken at the server side, is thus needed. Khan [74] has introduced the Padma architecture, which supports the graceful adaptation of multimedia content (video). It is however unclear how dynamic content such as 3D graphics is handled in this system. The quality to resource mapping is for instance done with platform dependent tables with measurements of the resource availability. This is not practical in a UMA context because the terminal is actually unknown in advance. The optimisation of the quality in this system is also handled with a knapsack problem. However, because of the fine grained scalability of 3D graphics content, other solutions, e.g. convex optimisation, are possible that provide a higher accuracy for the optimisation [56]. Pham Ngoc [100] has described a QoS framework, which implements an optimisation problem for 3D graphics to find the best possible quality level given the limited resources. The present work further

elaborates on and improves the optimisation algorithm of Pham Ngoc [100] for 3D graphics applications. Section 1.5 lists the main contributions of our work, which are also the main differences with the work of Pham Ngoc.

One of the key technologies to enable the gradual trade-off between quality and complexity is scalable coding, which encodes multimedia objects with different resolution scales in an embedded bit stream. An important standard for advanced multimedia applications that supports scalable coding is MPEG-4 [78]. Figure 1.2 shows a typical MPEG-4 application, which encodes multimedia objects as separate objects and describes their relations in space and time in a scene description. MPEG-4 also provides generic object descriptors for the different MPEG-4 media, but the exact translations from the object parameter set for each medium to the network and terminal QoS are beyond the scope of MPEG-4. The MPEG-4 simple (version 1) and fine-grained (version 2) scalable profiles [50] and MPEG's current work on Scalable Video Coding (SVC) [48] implement scalability for video. MPEG's Animation Extension Framework [47] supports scalability for 3D graphics content. Other standards that support scalable content are the VRML/X3D [158] standard for 3D graphics content and the JPEG2000 [51] standard for still images. Mohan et al. [97] for instance use this scalable content to adapt multimedia web content to optimally match the resource and capabilities of diverse client devices.

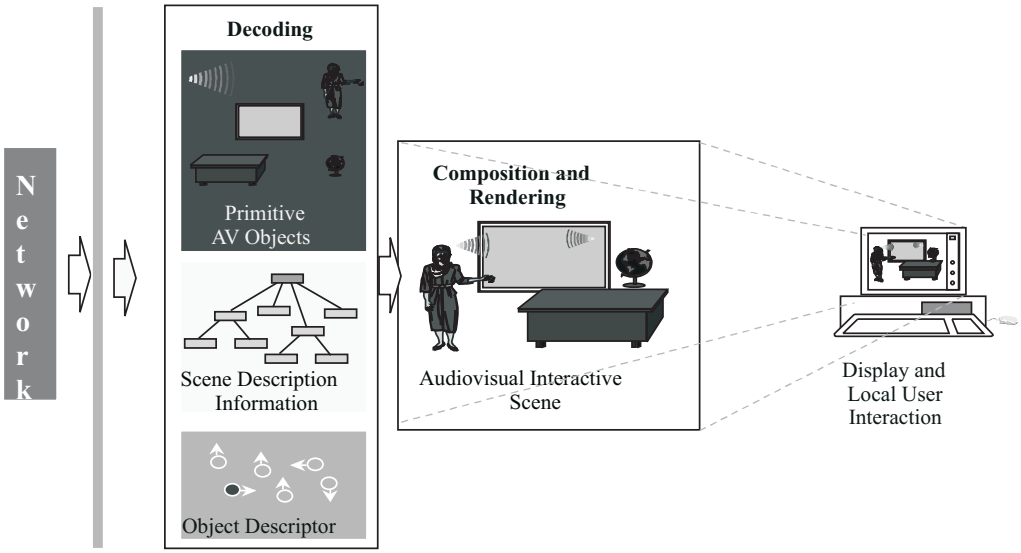


Figure 1.2: MPEG-4 encodes multimedia objects as separate objects together with generic object descriptors and describes their relations in space and time in a scene description.

Another important issue to enable the universal access of 3D graphics content is the development of 3D rendering chips for mobile devices. The issue there is to keep good performance at low cost (i.e. the price of the end-product) and low power

consumption. Academic work in the design of mobile graphics chips is done by the RAMP research group at KAIST: Woo et al. [164] use the conventional triangle based algorithms for a 210 mW graphics chip, which was made programmable by Sohn et al. [137]. Since external memory accesses are typically one of the most energy consuming operations, Akenine and Ström [1] have proposed a new hardware architecture that limits the number of accesses into the texture memory, but is not decreasing the visual quality. A second approach to limit the number of memory accesses is tile-based rendering, which is implemented in the pipelines of PowerVR [118] and which is used in the ARM MBX core [139]. Tile-based rendering divides the screen into several smaller parts. This allows using more on-chip memory and it eliminates a lot of costly off-chip memory accesses. Although this research into hardware accelerated mobile devices is evolving rapidly, few mobile devices support 3D hardware acceleration. For these mobiles, software-rendering engines such as Swerve3D [140], PocketGL [83] or the Vincent Mobile 3D Rendering Library [159] are used.

3D graphics hardware accelerators or Graphical Processing Units (GPUs) are commonplace in today's PCs and gaming consoles. A special characteristic of the simple parallel architecture of the GPU is that it allows for its performance to grow at a rate faster than the well known Moore's law (at a rate of 2.5 to 3.0 a year) [150]. The main reason for this is their stream architecture which enables all additional transistors to be devoted to increasing computational power instead of being devoted to cache memory as on a CPU [153]. The rapid growth of the GPU's performance has enabled the development of programmable GPUs and special programming languages, such as C for Graphics (Cg) [43] and brookGPU [23] for general purpose programming on GPUs.

The extra flexibility of the GPUs has opened the power of the GPU for other non-graphics applications. This has lead to a new research field known as general purpose computation on graphics processing units or GPGPU (general purpose GPU) [58, 109]. Interesting examples for multimedia are the computation of the wavelet [155] and the Discrete Cosine [105] transforms and the implementation of a part of an MPEG-2 video decoder on the GPU [136].

Along with the development of graphics hardware, graphical APIs are also continuously defined that allow the easy and interoperable programmability of the graphics hardware. The important 3D graphics APIs are OpenGL [134] and Direct3D [95] and their embedded versions are OpenGL ES [13] and Mobile Direct3D [94]. These APIs are however only a very thin layer above the graphics hardware. They only execute the commands that are issued by the application, whose task it is to manage the 3D scene. This involves the removal of objects invisible from the changing user viewpoint, the correct and optimal initialisation of the visualisation algorithms (state selection), the selection of the correct Level Of Detail (our work), etc. Examples of libraries that have been developed for these purposes are the Openscenegraph API [108] for C++ development and the Java3D API [138] for Java development.

Because polygonal (triangular) rendering was until now the most popular way to render a 3D scene, we have chosen to develop our framework for this type of rendering. Alternative rendering methods have however also been presented to avoid bottlenecks due to 3D polygonal rendering on mobile devices. A very interesting path for future work is to integrate the choice between the alternative rendering methods into the optimisation framework that we propose.

A first alternative rendering method has been proposed by Lamberti et al. [79]. They have proposed a remote rendering system that executes the rendering on a remote machine and sends the resulting images back to the mobile device. They use the Chromium system [67] for the remote rendering on a cluster of graphics workstations. Another alternative is the approach of Chang et al. [29] who use image-based rendering. This approach models the 3D objects with images with the advantage that the complexity is independent of the number of triangles in the scene.

Diepstraten et al. [38] have proposed an alternative method for the image-based approach by splitting the rendering process between the client and the server. 2D lines are extracted from feature lines of 3D objects on the server, which are rendered locally by the mobile device.

Duguet et al. [40] have proposed to use point-based rendering to display complex geometry on a mobile device. With complex geometry, the triangles typically project onto a single pixel on a mobile device with a small screen. Therefore, the gain of using triangular rendering is completely lost, and as a result, they use points instead of triangles to fill the object. As also shown by Rusinkiewicz and Levoy [124], point-based rendering offers a flexible and simple-to-use level of detail mechanism in which the number of rendered points is easily adapted to the underlying object's screen size. However, if an object's triangles project onto more than one pixel, the visual quality of point-based rendering is inferior to the quality of polygonal rendering.

1.3 An introduction to 3D computer graphics

The goal of this subsection is to give a short introduction to 3D graphics and the terminology necessary for this text. More in depth information on the basics of 3D graphics can be found in the books of Foley et al. [46] and Hill [63].

3D computer graphics is the field of study of the storage, manipulation and visualisation of three dimensional numerical models of objects with a digital system. Typical applications that use 3D computer graphics are games, film and television productions, e-commerce, engineering, scientific visualisation, etc. These applications can be divided in two categories with totally different demands: photo realistic applications demand a very high visual quality but allow a larger cost and/or computing time and real-time or interactive applications, which target the consumer market, must be low cost, but allow a lower quality. Our work focuses on the latter type of applications.

A popular way to construct the three dimensional numerical models is surface modelling, which only takes the surface of the 3D object into account and neglects the inner volume. The area of surface modelling is very broad, but the most popular method for real-time rendering is to use a piecewise linear approximation with triangles (triangular mesh). The main reason for this choice is simplicity: triangular rendering is easily implemented in hardware.

Triangular meshes are very commonly represented with a pair $\{V, K\}$ [66, 25], where V is a set of vertices (3D points) and K is a set of triangles that represent the connectivity of the vertices and determine the topology of the mesh. As shown in Figure 1.3, a triangular mesh defines the geometric structure of the object, but does not reveal anything about its colour or appearance. The appearance is for real-time applications typically defined with one or multiple 2D images (textures) in combination with a mathematical model of light reflection.

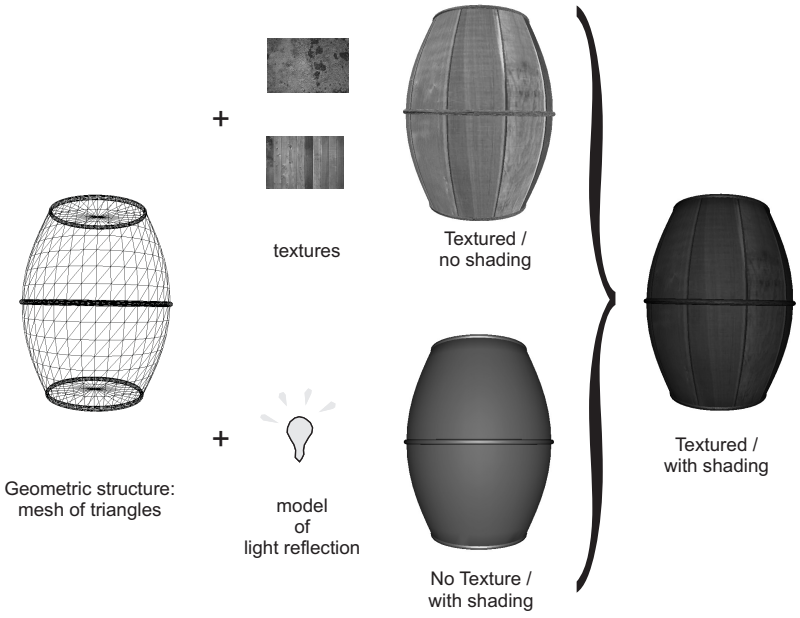


Figure 1.3: A 3D object is typically modelled with a mesh of triangles that describes the structure of the surface and an appearance model that defines the colours of the surface.

The definition of a triangular mesh is often extended to provide enough information to the rendering pipeline for mapping the appearance on the geometry. Hoppe [64] has defined two categories of appearance attributes: discrete attributes that are related to the triangles of K (e.g. selection of the texture mapped on the triangle) and scalar attributes that are related to the vertices of V (e.g. texture coordinates that define which texture element must be used on the vertex). A triangular mesh is then typically expressed as a tuple $M = \{V, K, D, S\}$ where D is the set of discrete attributes and S is the set of scalar attributes.

As shown in Figure 1.4, the rendering of an object starts with a transformation of the coordinates from V and the attributes from S. The vertices are then assembled into triangles using the information in K and the triangles are turned into fragments (the squares in Figure 1.4) by interpolating the vertex positions and attributes in the rasterisation. These fragments then pass through the fragment processor, which uses their information for calculating the fragment's final colour. The fragment processor for instance uses the texture coordinates that are interpolated from the triangle's vertices to sample a colour from the texture. If the fragment survives the different tests (e.g. depth test, stencil test, etc. [134]) in this unit, it updates the colour of the pixel in the frame buffer. The distinction between fragments and pixels is made because fragments are only "potential pixels" that contain much more information than needed in the frame buffer.

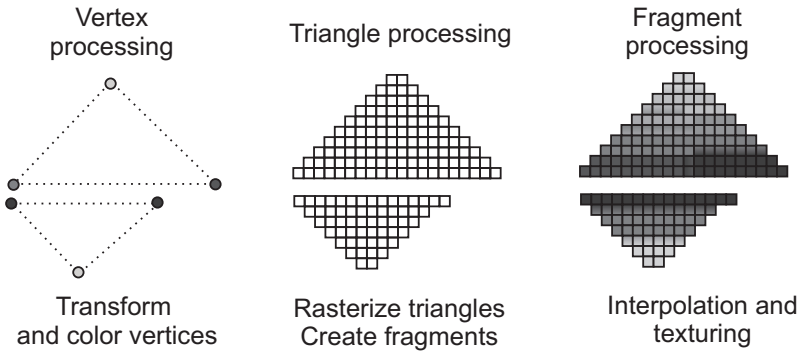


Figure 1.4: The 3D graphics pipeline: the vertex processing projects the vertices onto the screen, the triangle processing calculates the screen pixels that are covered by the triangle and the fragment processing interpolates the vertex attributes and samples the textures to calculate the pixel's colours.

Two important and independent data parameters for the execution time are the number of vertices/triangles and fragments that are processed by the rendering pipeline. A third parameter that influences the execution time is the state of the rendering pipeline, which selects the algorithms and their settings for rendering the object. Both the vertex and pixel units can use a configurable but standard algorithm, but they can also use a user-defined vertex or fragment program [43].

The simplest way to control the execution time of the rendering is by changing the number of triangles/vertices in the geometry. As illustrated in Figure 1.5, mesh simplification techniques generate a mesh M_i, which is an approximation of the original mesh M, but with fewer elements and a lower quality. One such simplified mesh, which is part of a series of increasingly simplified models, is called a Level Of Detail (LOD).

Mesh simplification techniques can be used to derive a multi-resolution model which captures a continuous range of LODs of an object and which can be used to reconstruct any one of those levels on demand. A special type of a multi-resolution model



Figure 1.5: The famous Stanford bunny shown at different Levels of Detail (LOD): LOD 0 = low quality/low complexity ... LOD 4 = high quality/high complexity.

is a progressive mesh that eliminates a lot of computation by encoding the mesh in a special way. Progressive meshes first store a low resolution mesh M_0, which is then followed by detail information ordered by importance. This detail information gradually increases the quality of the object and the number of elements in the mesh representation M.

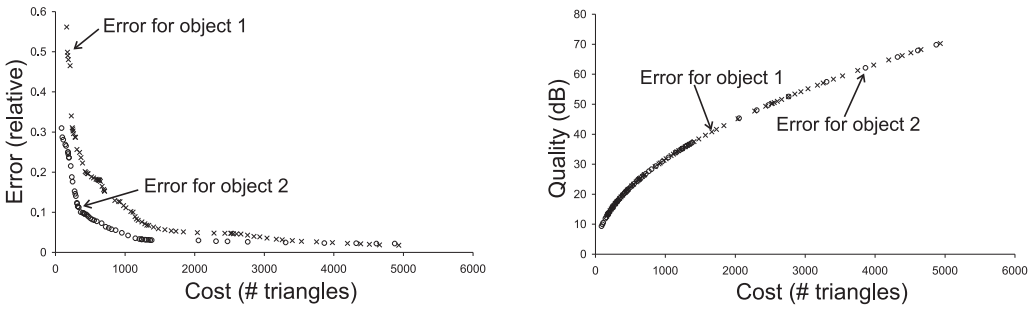
Mesh simplification methods alter the geometric model of a triangular mesh and typically use geometric criteria for determining the order for removing triangles. However, as explained in Chapter 4 on quality modelling, the vertex transformations of the model in the rendering pipeline, which depend on the user viewpoint and object position, heavily influence the visual quality. The choice of the number of elements in M thus also depends on these transformations; therefore, view-dependent mesh simplification techniques include the impact of these transformations in the simplification process.

1.4 Overview of approach and outline of the text

To allow the universal access of 3D content, we have developed a framework that controls the execution time of an arbitrary 3D rendering pipeline by the control of the quality and complexity of the input data. The trade-off between visual quality and execution time is in our work done with multi-resolution content, i.e. content that starts with a coarse representation of an object with a low number of triangles and gradually improves the visual quality by adding more and more triangles [64]. Because our target applications (e.g. an online game) need networked access to 3D content, we have implemented a standardised technique with a very good compression: MPEG-4 Wavelet Subdivision Surfaces (WSS) [47].

An optimisation algorithm distributes the available triangle budget among the visible objects in the scene based on their impact on the visual quality. Because this optimisation problem is used in real-time rendering, the assessment of the visual quality must be very fast. Consequently, previous work typically uses very simple heuristics to assess the accuracy of the content. Gobbetti et al. [56] for instance use

the square root of the number of vertices. Figure 1.6.b shows the resulting evaluation of the quality for two different objects. Although the real accuracy differs for both objects, as illustrated in Figure 1.6.a, the heuristic is unable to detect these differences.



(a) Real error (b) Simple heuristic

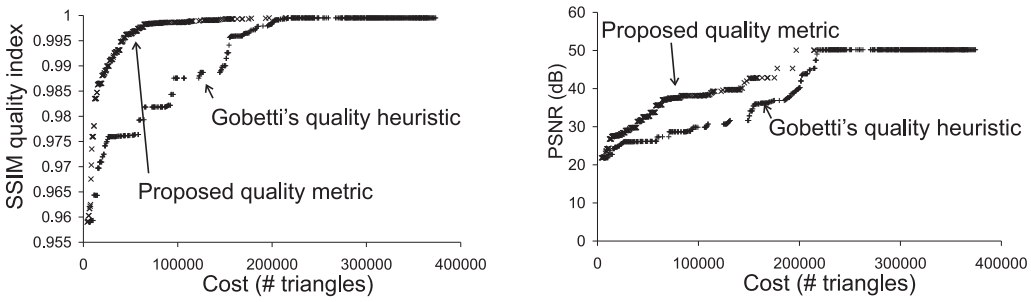
Figure 1.6: The geometric error for two different objects: a) the real error metric and b) a simple heuristic [56] which is unable to detect the differences between both objects.

On the other hand, Hoppe [65] has proposed a view-dependent mesh simplification algorithm that uses a complex error metric (similar to the one of Figure 1.6.a) to steer the simplification. However, because these metrics are measured between the approximating and the original content, which is not available at the client side, the measurements must be encoded with the content. This results in a vast amount of extra data. In addition, Chapter 5 illustrates that this type of view-dependent content adaptation has a significant complexity on current 3D graphics platforms.

We propose a hybrid approach that combines the higher accuracy of the complex error metrics with the simplicity of the simple heuristics. The quality and cost of the different input settings of the WSS decoder are measured in off-line preprocessing steps for the separate objects in the scene. The optimal trade-offs between quality and cost are summarised in a distortion plot (also called Pareto curve [113]). The online steps of the optimisation then combine the distortion plots of the visible objects in the scene with a minimum of online parameters (e.g. user viewpoint) to find the overall trade-off between the quality and cost.

The result is shown in Figure 1.7 (taken from Chapter 4), which shows the quality of the rendered images with two different error metrics: the PSNR and SSIM index (Structural SIMilarity index [156], discussed in Chapter 4). From Figure 1.7, it is clear that the proposed framework increases the visual quality for the same rendering cost, while the execution time of the optimisation itself is comparable with previous work (Chapter 2, Section 2.4).

The complete framework is presented in Figure 1.8. The optimisation that maximises



(a) SSIM index (b) PSNR

Figure 1.7: Comparison of the image quality of an optimisation using the proposed quality metric compared with an optimisation that uses a simple Benefit heuristic: a) measured with the SSIM index and b) measured with the PSNR

the visual quality for a given triangle budget is the central part of this framework and is further discussed in Chapter 2.

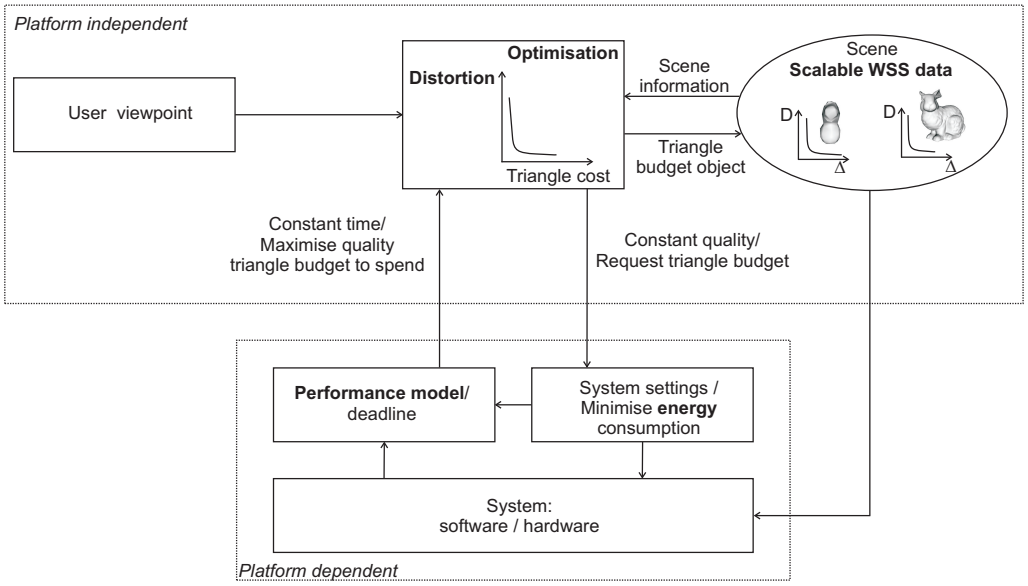


Figure 1.8: High-level overview of the system

Several control strategies are available to decide the triangle budget, from which the feedback and model based control techniques are the most important ones [71]. A typical feedback algorithm that is used in 3D graphics is to increase or decrease the threshold for the distance between the object and the viewpoint for which the LOD is increased or decreased [53]. As stated by Funkhouser et al. [52], this method is not very suitable for abruptly varying 3D graphics content; therefore, a model that translates the execution time deadline into a triangle budget is needed for

abruptly varying content. However, since no mathematical model is perfect and cannot completely predict future values, feedback is still used to correct for model inaccuracies. Because our research targets the universal access of 3D content on different devices, two completely different terminals were considered in this work: a low performance terminal lacking a hardware accelerator for the 3D rendering and a PC with a Geforce FX 5900 graphics chip. Chapter 3 discusses the methods for deriving the performance models.

A good, but fast assessment of the visual quality is very important for our framework. Therefore, chapter 4 presents our quality model, which is derived from existing geometric error metrics. These metrics are typically used to compare different mesh simplification techniques, but neglect the user's viewpoint and the object's appearance model. Chapter 4 also discusses the changes to this model to make it view-dependent and to include a minimum dependency on the object's appearance model. Chapter 4 starts with a short overview of the parameters that influence the quality in 3D graphics applications. Several heuristics from previous work are discussed and a view-dependent quality heuristic is proposed.

The optimisation block directly controls the WSS decoder so that it presents the content with the correct complexity to the rendering. Chapter 5 discusses several techniques that can be used for changing the geometric structure of the object in order to meet the budgets that are decided by the optimisation. The current MPEG-4 specification is introduced and improved to support continuous LOD. An important part of this chapter discusses the complexity of the different adaptations and their usability for the optimisation. A method that eliminates this complexity, but (unfortunately) decreases the cache efficiency is also proposed.

Chapter 6 combines all the building blocks that are presented in the previous chapters into the complete framework and illustrates how the behaviour of the optimisation over successive frames determines the method that is used for the content adaptation.

The last block (system settings) in Figure 1.8, which is discussed in Chapter 7 is needed to extend the optimisation framework, so that it tries to provide a constant and user requested quality and exploits the resulting dynamic workload to minimise the energy consumption for a given execution time deadline. The optimisation framework then requests a triangle budget to the platform and if this requested triangle budget is lower than the available one, the platform can adapt itself to a lower performance/energy mode. If however the requested triangle budget is larger than the maximum budget at the highest energy consumption, the traditional approach is followed that adapts the content to the available triangle budget.

Figure 1.9 visualises the structure of the text and illustrates how the chapters are best read.

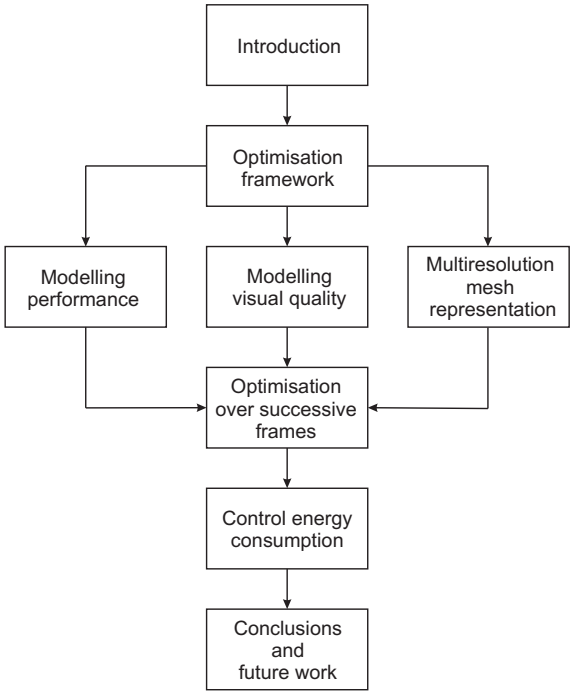


Figure 1.9: Flow chart for reading the text.

1.5 Summary of Contributions

The main contributions of the proposed framework are:

1. The development of an optimisation algorithm that is able to use a complex, but realistic error metric by a partitioning of the work in off- line preprocessing steps and a simple online optimisation. The overhead of this algorithm is comparable to previous work (see Chapter 2), while a better quality is achieved for the same execution time deadline (see Chapter 4)

2. The development of a realistic error metric that improves on the simple quality heuristics that is used in previous work.

3. An improvement of the standard MPEG-4 WSS decoder and a first selection of usable adaptation methods for the optimisation based on a study of complexity.

4. An improvement of existing performance models of 3D graphics rendering pipelines for more recent graphics architectures.

5. An extension of the optimisation framework providing a constant and user requested quality, which exploits the resulting dynamic workload to minimise the energy consumption for a given execution time deadline. The example of this text consumes 76% and 30% of the energy consumption that is obtained

with a traditional optimisation algorithm (that uses a fixed triangle budget) for a high respectively low visual quality.

Chapter 2

Optimisation Framework

Chapter 2 introduces the optimisation framework that automatically adapts the Level Of Detail (LOD) of the 3D content to the terminal's available performance at a maximum visual quality.

Prior art techniques sacrifice the accuracy of the quality and sometimes also the performance models for simplicity in the online problem solving. Though this looks reasonable, one may question whether it would not be possible to reconciliate accuracy to fast online decision taking. Therefore, the basic strategy underlying our work is to shift the complex parts of the work to off-line preprocessing steps and to reduce the online work to simple combination steps. This enables the use of complex but realistic quality models, which increase the 'quality' of the final optimisation. The execution time of the optimisation is however competitive with prior art techniques.

The link between the preprocessing steps and the online work is a distortion plot that stores the optimal trade-offs between the quality and the cost. The distortion plots may also be called Pareto plots because they only contain optimal trade-offs for which it is impossible to improve the quality without worsening at least one of the costs [113].

The present chapter elaborates on these issues. Section 2.1 first describes prior art techniques. Section 2.2 then gives an overview of the framework. The preprocessing is discussed in Section 2.3 and the online work is discussed in Section 2.4. Some examples are presented in Section 2.5 and finally this chapter ends with the conclusions in Section 2.6. Results for the complete framework, including the scalable decoder and the performance and quality models are discussed in Chapter 6. A comparison of the results of our proposal with prior art is given in Chapter 4.

2.1 Related work

The first techniques that were used for choosing an object's Level Of Detail (LOD) used static heuristics based on the screen size of an average triangle of the object [53] and/or the distance of the object to the user [12, 122]. These simple heuristics improve the frame rate in many cases, but cannot guarantee a regulated or bounded execution time.

Feedback algorithms [121, 132] continuously track the execution time and adapt the LOD (e.g. with size thresholds) of the objects based on the measurements. This method provides nice results for systems with small changes of the content in the successively rendered frames, but suffers of unavoidable overshoot and oscillation problems during the visualisation of dynamic scenes [52].

To guarantee a bounded frame rate, Funkhouser and Séquin have [52] presented a predictive technique that uses an estimate of the execution time for the correct choice of the LOD. The core of this work is a multiple choice knapsack problem [68] that maximises the visual quality for a given maximum execution time. The greedy solution of Funkhouser and Séquin however only guarantees to be half as good as the optimal solution [56]. Therefore, Gobbetti and Bouvier have [56, 57] proposed to use convex optimisation (interior point algorithm) [11] with a guaranteed specified accuracy ϵ. However, because a convex optimisation needs convex functions for the cost and quality, Gobbetti and Bouvier had to define a convex quality heuristic. Convex optimisation guarantees a specified accuracy for the given quality function, but their quality function cannot guarantee the same accuracy.

The optimisation algorithm of Funkhouser and Séquin selects the objects with the highest benefit/cost value until the target frame time is met. The result is that low-valued visible objects may not be displayed. Therefore, Maciel and Shirley [90] have extended this work with a hierarchical description of the scene and extra impostors, such as texture mapped bounding boxes. Their algorithm successively replaces the selected representation of the most important object in the current scene representation with the selected representations of its children, until the available rendering time is completely consumed. Mason and Blake [12] have correctly mentioned that this problem is a constrained multiple choice knapsack problem and further refined this approach.

A different approach is followed by Kim et al. [76], who have used the principles of the MPEG-21 multimedia framework (bit stream adaptation) [19] for adapting 3D content to various network and terminal capabilities. The adaptation is however limited to adapting the content for meeting the average performance constraints of the terminals, neglecting the variations of the frame rate.

Zach et al. [167] have proposed a specific optimisation algorithm for a terrain flyover application. Interesting in their approach is that they not only use discrete and continuous Level Of Detail for polygonal rendering but also point-based rendering

for the trees on the terrain. Aliaga et al. [2] have proposed a different kind of trade-off for guaranteeing a frame rate. This approach trades off space instead of quality for frame rate by using images to replace distant content. The extension to use different rendering techniques is not implemented in our work, but is an interesting path for future work.

The work of Gobbetti et al. uses a simple, but inaccurate model for the visual quality; therefore, the presented work introduces an optimisation algorithm using a more realistic, but complex model for the visual quality. To resolve the complexity issue, the complex parts of the algorithm are shifted to off-line preprocessing steps. The online work is then limited to simple combination steps. Chapter 4 illustrates the gain in quality (10 dB PSNR for the used example) of our approach with a performance that is similar to the performance reported in previous work (Section 2.4.2).

The trade-off between execution time or resource consumption in general and visual quality is also used in other media than 3D graphics. Hentschel et al. [62] have for instance introduced a quality of service system that adapts the visual quality of MPEG-2 video to the available resources on the used terminal. The work of Mattavelli et al. [91], on computational graceful degradation, was used as input for the MPEG-4 standard. They have added metadata to MPEG-4 streams to be able to use a predictive approach to control the decoding time of an MPEG-4 stream. The needed scalability of the video streams was investigated by Peng [116] and Chen et al. [32].

The approach used in our work to shift complex work into preprocessing and to use summarising plots at run-time is also used in different domains. Bougard et al. [20] have developed a methodology for wireless systems that summarises the optimal trade-offs between energy and performance into a Pareto curve and that selects the optimal trade-off at run-time based on the state of the wireless channel. Another example is the work of Wong et al. [163] who use the same approach to schedule different tasks on a platform.

2.2 The framework in a nutshell

The goal of the optimisation framework is to provide the best possible image quality within the performance constraints of the used terminal. The present framework derives for every separate object in the scene the optimal trade-offs between the quality and the cost in off-line preprocessing steps and summarises the results in a distortion plot for a single object. These steps are illustrated in Figure 2.1 and are further discussed in Section 2.3. The online optimisation steps combine the distortion plots of the visible objects to find the optimal trade-off between error/quality and cost for all the visible objects.

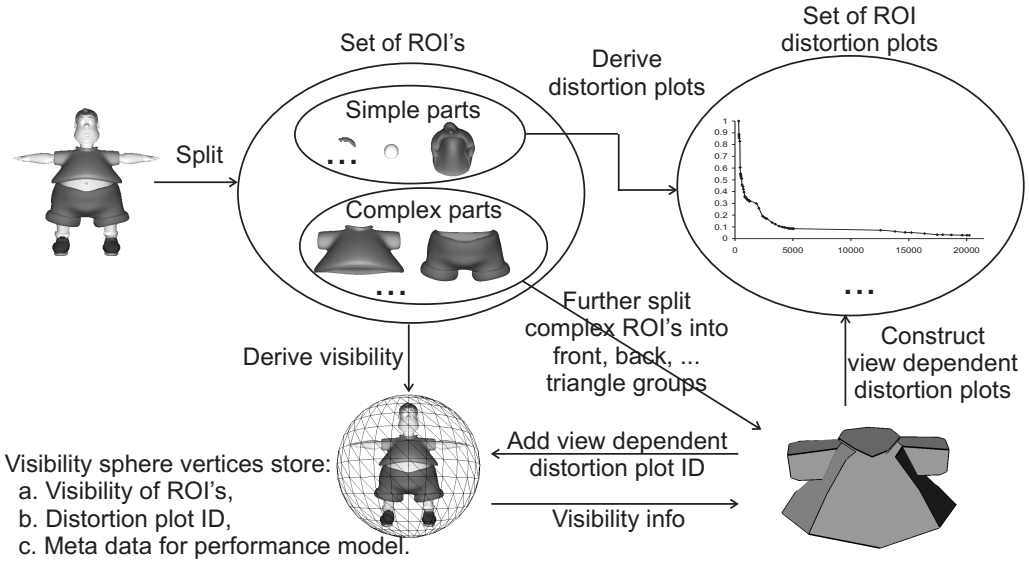


Figure 2.1: Overview of the preprocessing work to derive the view-dependent distortion plots of the object.

Objects contain both visible and invisible regions, for which the visibility depends on the viewpoint of the user. The optimisation and rendering thus contain object regions that do contribute to the cost but not to the quality. Therefore, a first issue is to limit the online decision taking and rendering to visible object regions. Ideally, the visibility of every triangle is derived online. However, this results in a large overhead and a better approach is to group triangles with more or less the same visibility. Consequently, the first step in the preprocessing (Subsection 2.3.1) is to split the objects in the scene into groups of triangles for which it is easy to derive the online visibility. If possible, an object is first split into groups (called Region Of Interest or ROI) that do not share any vertices, edges and triangles. If the complexity of these groups is still significant, they are further split into smaller subgroups (called sub-ROI).

The optimal trade-offs between the quality and the cost are summarised for a single ROI in a distortion plot (Subsection 2.3.2). The visibility of the different ROIs is checked for viewpoints on the vertices of a visibility sphere around the object and stored as meta-data on these vertices. For a specific viewpoint on the visibility sphere, the distortions and costs of the sub-ROIs invisible from that viewpoint are subtracted from the ROI's distortion plot (Subsection 2.3.3). The result is a view-dependent distortion plot for every viewpoint on the visibility sphere. Luckily, a lot of viewpoints have more or less the same visibility of the sub-ROIs, so their corresponding view-dependent distortion plots are more or less the same. These plots are then represented with a single view-dependent plot for which the ID is attached to the corresponding viewpoints on the visibility sphere.

Figure 2.2 illustrates the online steps of the optimisation, which are further elabo-

rated in Section 2.4. Traditional culling techniques derive the objects of the scene that are (potentially) visible from the current viewpoint. The user viewpoint is tracked and projected on the visibility sphere of the visible objects. The visibility and, if needed, the distortion plot ID of the different parts is then loaded. The distortion plots were derived for a unit viewpoint distance during preprocessing and are therefore scaled in a next step with the actual viewport size and distance d to the user viewpoint. The distortion plots use the number of triangles as a platform independent cost metric. A performance model uses the visibility information (e.g. number of pixels) on the visibility spheres of the visible objects to calculate the triangle budget for the optimisation. Because the view-dependent distortion plots are made convex in preprocessing (Subsection 2.3.2), the selection of the best configuration point can be established by a simple gradient-descent method [102, 21]: all objects parts start at the highest distortion (lowest execution time) and the next configuration point on any object part that yields the highest distortion reduction for the smallest cost increase is selected. This process is repeated until the target triangle budget is reached. All object parts are then adapted to their respective configuration point and rendered.

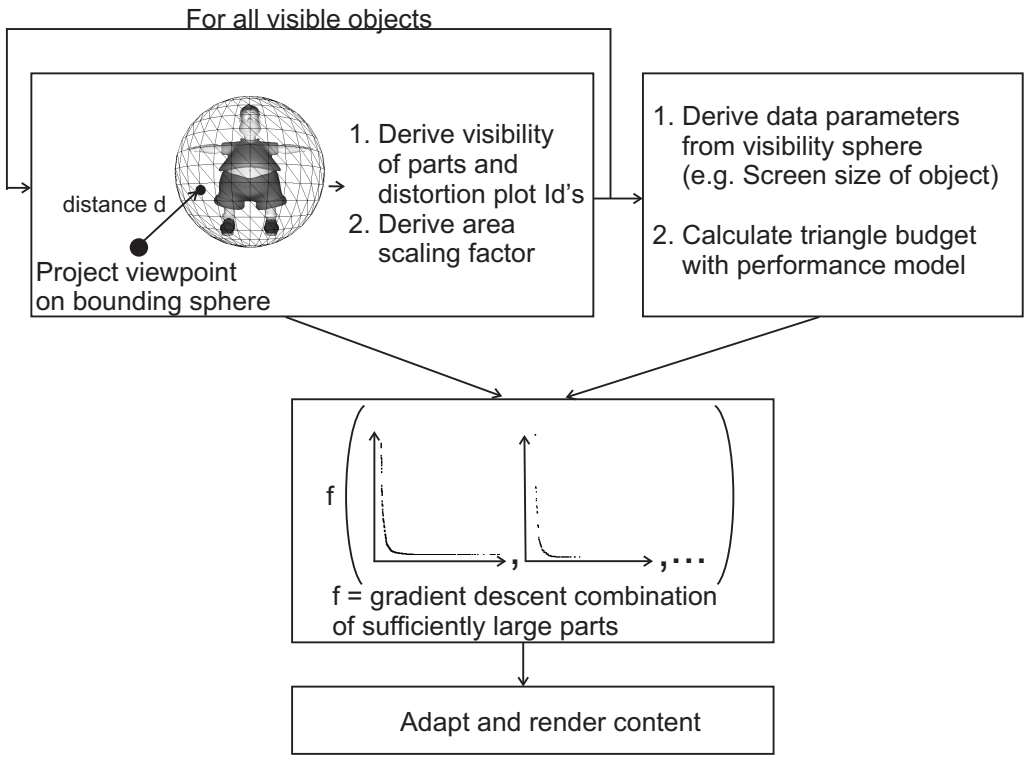


Figure 2.2: Overview of the run-time work.

2.3 Off-line Preprocessing steps

Our method for optimising the visual quality uses five preprocessing steps (see also Figure 2.1) that do the expensive work in terms of computation. These steps are:

1. Decompose the object in unconnected regions and if needed further decompose these regions in subregions (Subsection 2.3.1),
2. Compute the distortion plots for all regions (Subsection 2.3.2),
3. Remove redundant and suboptimal points from the distortion plots (Subsection 2.3.2),
4. Derive the visibility of the different regions and subregions (Subsection 2.3.3).
5. Create view-dependent distortion plots for the object regions by eliminating the influence of invisible subregions (Subsection 2.3.3),

2.3.1 Step 1: Decomposing the object

It is preferable that the model for the visual quality only contains information from visible parts and discards invisible ones. Traditional view frustum and occlusion culling techniques [99] eliminate complete objects from the rendering pipeline, but objects that pass these tests can still contain a lot of invisible parts. Therefore, the first step is to split an object in a number of parts for which the visibility is easily derivable and for which the resulting distortion plots can be easily combined online.

The accuracy of the visibility test depends on the object decomposition in the first step. The more object parts that are used to decompose the object, the better the elimination of invisible triangles, but the larger the extra cost for the online steps. Therefore, the decomposition of objects must be carefully done.

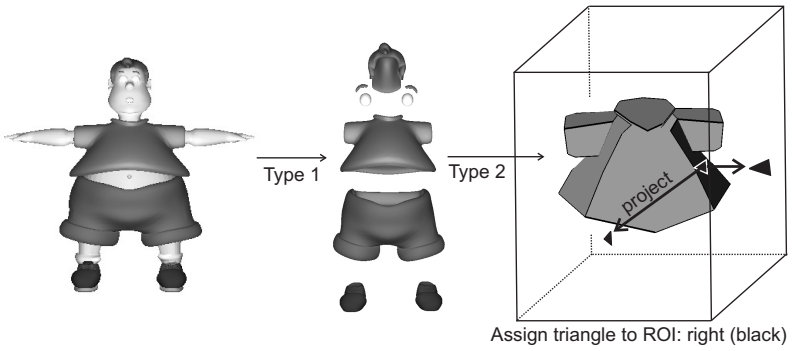


Figure 2.3: The decomposition of an object: type 1 splits an object into ROIs and type 2 further splits an ROI into sub-ROIs.

Figure 2.3 illustrates the first type of object decomposition, which is to split the object into parts, called Regions Of Interest (ROI), that do not share any vertices, edges or triangles. This split is only possible for objects that contain unconnected parts, such as the character of Figure 2.1, which is decomposed into the body, shirt, short, etc. Because these unconnected parts are independent of each other, their resolution may be varied independently without the introduction of artifacts [148] (called cracks in the specialised literature). An extra motivation for this decomposition is that the character can be rendered with or without clothes, etc. Consequently, the rendered parts of the object and their distortion plots are only known in the online steps and must be combined online and not in preprocessing. The famous Stanford Bunny (see Figure 1.5) is an example for which this type of object decomposition is impossible.

Figure 2.3 also illustrates the second type of object decomposition that follows the first type and which is only done on complex ROIs. This object decomposition splits each independent ROI into sub-ROIs. As shown in Figure 2.3, sub-ROIs are formed with an orthographic projection of the triangles on the rectangles of its bounding box. The triangles are then assigned to the rectangle of the bounding box (the sub-ROI) for which they have the largest visible projected area.

The only intention of a sub-ROI is to tag a group of triangles as visible or not, and to eliminate them easily from the rendering and the optimisation process. Therefore, the visibility tags of the different sub-ROIs are stored on the visibility sphere around the complete object of Figure 2.1. Every vertex (viewpoint) of the visibility sphere contains a visibility vector for which every element determines whether the corresponding sub-ROI is visible from that vertex. A sub-ROI is visible if one of its triangles is visible from the viewpoint.

Different resolutions between adjacent sub-ROIs cause artifacts (cracks) at the boundaries of the adjacent sub-ROIs. Because the resolution is decided online, these artifacts must be handled with an extra module (increasing the complexity) in the content adaptation that fills the cracks between the boundaries of the adjacent sub-ROIs. To limit the online complexity, we do not allow choosing a different resolution level for the different sub-ROIs of the same ROI. Therefore, the content adaptation only changes the geometry of the complete ROI, as if sub-ROIs do not exist.

To limit the additional online overhead, the decomposition of ROIs into sub-ROIs is only done on complex objects (e.g. meshes with more than 10000 triangles), for which a lot of triangles can be eliminated by using sub-ROIs.

2.3.2 Steps 2 and 3: Derive a view-independent distortion plot

The second step for deriving the distortion plots is to calculate the error metric of Chapter 4 for the ROIs and sub-ROIs. This metric is calculated for all the different

working points of the decoder proposed in Chapter 5 and a unit distance to the visibility sphere around the object. This typically results in the plot of Figure 2.4, which uses different markers to indicate the difference in complexity to obtain these different working points of the used decoder.

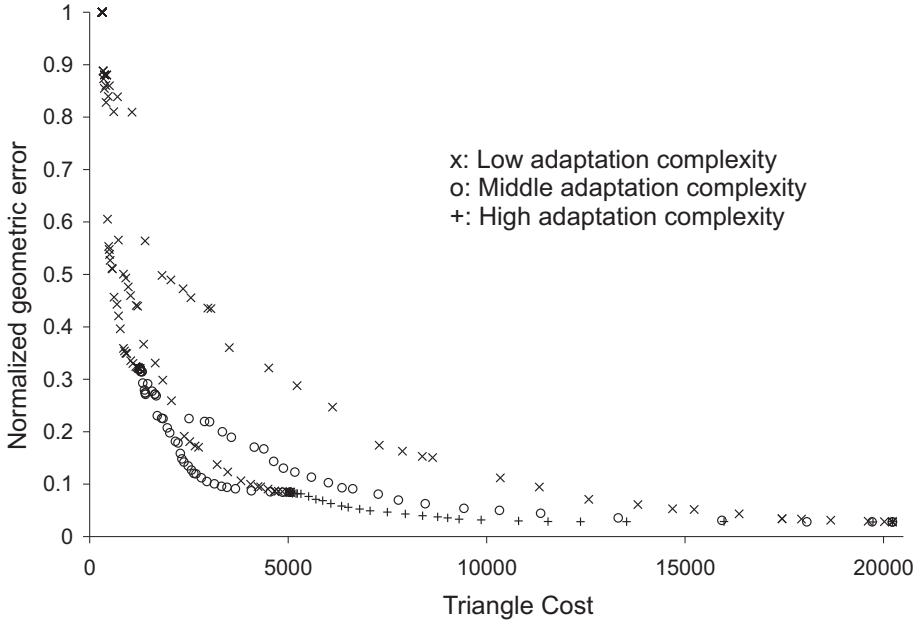


Figure 2.4: A distortion plot that contains the error and cost values for all working points. The different markers indicate a difference in adaptation complexity to obtain these working points.

The third step is then to remove all configurations with a suboptimal combination of error and cost. The result of this operation is the pruned distortion plot or Pareto curve [113]) shown in Figure 2.5. The cost that is used in Figure 2.5 depends both on the rendering and on the adaptation cost, and their actual execution time depends on which system part (e.g. CPU or GPU) they are mapped on. Typically, the adaptation is mapped on a CPU and the rendering on a GPU, in software-only platforms both are mapped on a CPU, but on future platforms one could imagine performing the full mapping on the GPU. To support all possible triangle costs to execution time mappings, adaptation and rendering are separated into two different axes. Both costs are expressed as platform independent parameters (triangle cost), because the actual used platform is not yet known in the off-line preprocessing process (could be CPU, GPU, etc.).

It is the purpose of a multi-resolution decoder to maximise the quality for a minimum number of triangles for a single object; therefore, the progressive decoder adds the triangles to the geometry according to their impact on the error. The first triangles thus decrease the error much more than the last triangles. Adding triangles decreases the error and the first derivative of the function that describes the error as a function

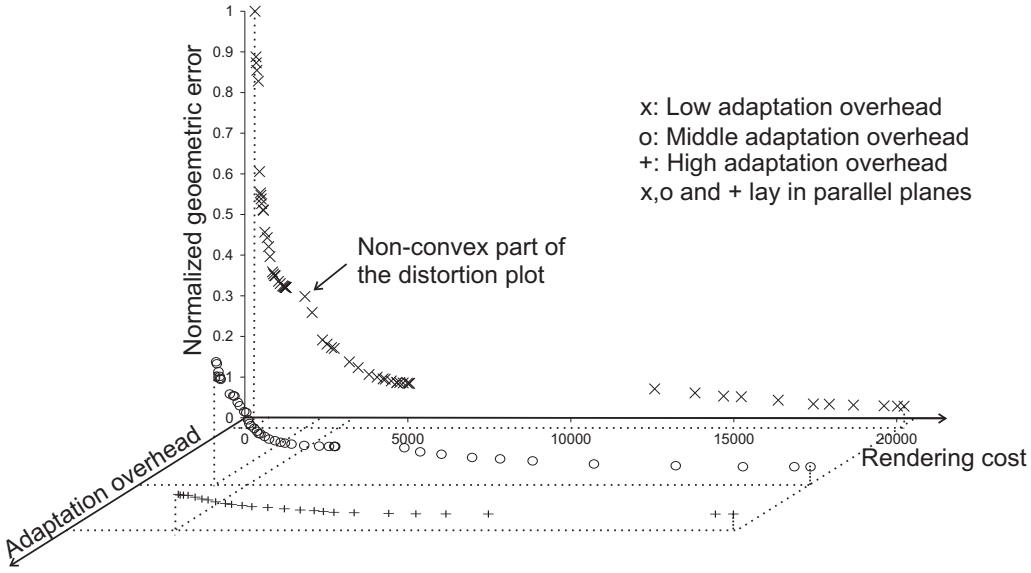


Figure 2.5: A pruned distortion plot or Pareto curve, which only contains the working points with the optimal errors as function of a two dimensional cost (rendering and adaptation overhead).

of the cost is thus negative. The impact of the next triangles on the error or the slope of the function decreases, which means that the first (negative) derivative increases; therefore, the second derivative is positive. This is a sufficient condition for the ideal quality-cost trade-off to be a convex function [11].

However, practical progressive decoders do not necessarily provide convex (view-dependent) distortion plots (as shown in Figure 2.5). As explained in the previous paragraph, the ideal cost-quality trade-off is convex; therefore, we use a statistically average convex curve fitted onto the real distortion plots for the optimisation algorithm described in Section 2.4. This curve, which has empirically been chosen of the form $a \cdot x^b$ (with $a > 0$ and $b < 0$) is a convex curve and enables a simple gradient descent method for convex optimisation [21]. Note that this function is actually a generalisation of the quality heuristic used by Gobbetti et al. [57] with $a = 1$ and $b = 0.5$ ($b > 0$ because they model quality instead of error).

2.3.3 Steps 4 and 5: Derive a view-dependent distortion plot

The fourth step is computing the visibility of the ROIs of the object. This is done by measuring the number of pixels using occlusion querying [42] for a selected number of viewpoints on the vertices of a visibility sphere around the object (see Figure 2.6). A vector with the visibility states of the different ROIs is then attached to that vertex of the visibility sphere. At run-time, the viewpoint is projected on this visibility sphere and the visibility of the different ROIs is then estimated from the

nearby vertices of the visibility sphere. The number of pixels that was measured with the occlusion querying is also attached to the vertex, so that it can be used later on for distance scaling of the distortion plot (see Subsection 2.4) and for estimating the execution time (see Chapter 3).

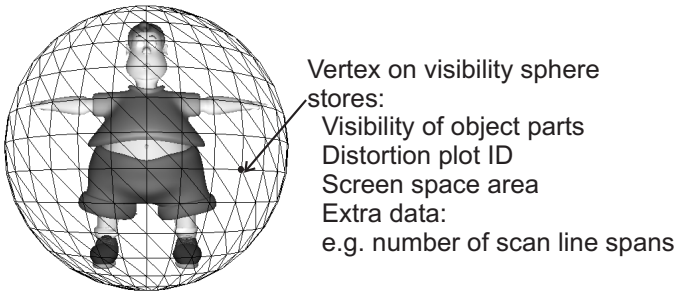


Figure 2.6: A visibility sphere around the object stores on its vertices the necessary data for all the possible viewpoints around the object.

The ROI distortion plots are made view-dependent for different viewing angles, but a single unit distance, to the object in a fifth and last step. The viewpoints on the visibility sphere for which the same sub-ROIs are visible are grouped. The distortion plots for the invisible sub-ROIs for those viewpoints are then subtracted from the ROIs distortion plot. The ID of the view-dependent distortion plot is then attached as meta-data to the vertices of the visibility sphere under consideration.

2.4 Online work: Derive a global scene distortion plot

The trade-off between the quality and the cost for all objects in the scene is in the online steps found by a simple combination of the view-dependent distortion plots derived in the preprocessing steps. The combination is done with a gradient descent method elaborated in Subsection 2.4.1. The complexity of the online steps is discussed in Subsection 2.4.2.

2.4.1 Gradient descent combination

Figure 2.2 illustrates the online combination of the different distortion plots. Only distortion plots of visible objects are combined; therefore, the first step in the online processing is the selection of the visible objects using traditional view frustum and occlusion culling techniques. The second step is the selection of the visible object parts and the ID of their (view-dependent) distortion plot from the visibility sphere around the object.

The starting point or first selected point of the optimisation is the combination of the object distortion plot working points having the smallest cost and largest error (smallest quality contribution). Equation 4.14 (presented in Chapter 4) is used for calculating the global error metric and the costs are summed together. The next point is then the point for which the error (E) of Equation 4.14 decreases maximally for a minimum triangle cost (C) increase, i.e. the absolute value of the derivative $\frac{dE}{dC}$ (calculated on the approximating curve ax^b) should be maximised. Because of the convexity of the curve ax^b derived in preprocessing, one knows that the absolute value of the derivatives in the selected working points on the object distortion plots are higher than in their following working points; therefore, the working point is changed for the object with the highest absolute value of the derivative $\frac{dE}{dC}$ in the selected working point of its view-dependent distortion plot. The optimisation is stopped when the minimum error is found or the maximum triangle budget is reached. The error and cost pairs that result from the optimisation can also be summarised in a distortion plot, which is called the global distortion plot in the rest of this text.

The optimisation process described in this subsection is also known as a classical gradient descent method or as a steepest descent method with Euclidean norms [21]. Equation 2.1 formalises the method with x^k and x^{k+1} the number of triangles for respectively the selected and next working point on the global distortion plot, $\nabla f(x)$ the (negative) derivative and t^k the step size on the global distortion plot. In our case, $\nabla f(x)$ is equal to the derivative $\frac{dE}{dC}$ of the object distortion plots and the step size t is determined with the distance between subsequent working points on the object distortion plots.

$$x^{k+1} = x^k - t^k \cdot \nabla f(x) \tag{2.1}$$

Gradient descent methods are typically used to solve unconstrained minimisation problems and have a linear convergence [21]. The stopping criterion is usually of the form $\|\nabla f(x)\| < \varepsilon$ with ε small and positive. This stopping criterion is however easily extended with a check on the triangle budget.

Note that the interior point optimisation of Gobbetti et al. [56] guarantees an accuracy ε and the greedy solution of Funkhouser et al. [52] to solve the MCKP optimisation can only guarantee a solution which is half as good as the optimal one.

The platform independent triangle cost that is used by the optimisation is translated into the real execution time T by the performance model, which is discussed in Chapter 3. Figure 2.7 shows this execution time as a function of the number of triangles for working points on a global distortion plot, measured on two different platforms.

The optimisation changes the resolution of an object if its change of error is maximal for a minimal increase of the triangle cost. This approach only guarantees a maximal

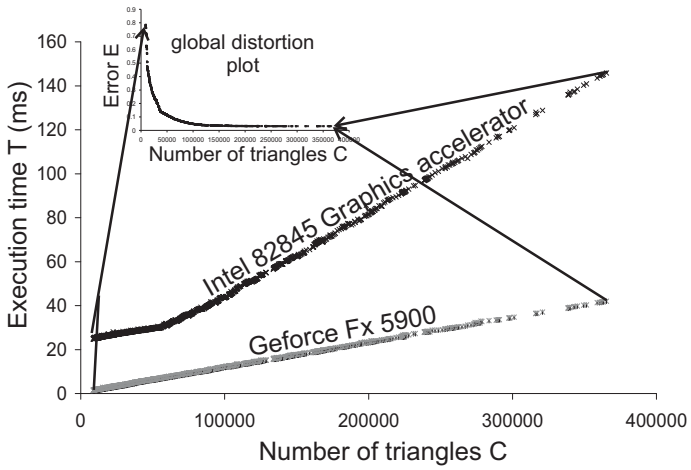


Figure 2.7: Measurement of the execution time T as a function of the number of triangles C for the working points on a global distortion plot: the relation is convex and even linear.

quality for minimum *execution time* if there is no other point further on the distortion plot for which the derivative $\frac{dE}{dC}$ is smaller, but for which the derivative $\frac{dE}{dT}$ is larger.

The relation between the execution time T and the number of triangles C is however for most platforms convex and even linear, i.e. the slope of the execution time (dT) in Figure 2.7) increases or remains equal with an increasing number of triangles. Thus, if the derivative $\frac{dE}{dC}$ is maximal for a point on the global distortion plot, the derivative $\frac{dE}{dT}$ is also maximal for that point. This is easily illustrated with Equation 2.2 in the linear case.

$$T = a \cdot C \Rightarrow dT = a \cdot dC \Rightarrow \frac{dE}{dT} = \frac{1}{a} \cdot \frac{dE}{dC} \tag{2.2}$$

2.4.2 Complexity analysis

The complexity of the online combination of the distortion plots is linear with the total number of points that are visited by the gradient descent method, which is equal to $\sum_{i=0}^{M} P_i$ with M the number of distortion plots and P_i the number of points on distortion plot i that are visited by the gradient descent method. It is thus important that the pruning step in the off-line preprocessing only selects points with a significant change of cost and quality. Additionally, the following online optimisations further reduce the complexity of the optimisation:

- Meshes with a small screen space area are eliminated from the optimisation during the visibility test. The screen space area is read from the visibility sphere and is then scaled to the viewport and the current distance from the

viewpoint to the mesh. If this area is smaller than a threshold value (e.g. 400 pixels), the mesh is rejected from the optimisation, but is still rendered at a fixed resolution (e.g. the base mesh).

- The quality that is read from the distortion plots is scaled online with the screen space area, through which it is possible that the difference in quality and cost becomes negligible in the online combination phase. Therefore, an extra check is done to prevent the work for points with a high derivative but a low increase in absolute quality (and cost).

These extra online pruning steps can be used to control the number of visited points and thus the execution time of the optimisation process. Figure 2.8 shows the resulting execution time for the visibility determination and the gradient descent method as a function of the total number of visited points. The execution time in Figure 2.8 is measured on a Pentium IV processor and is the time needed to derive the minimum error. Figure 2.8 also shows some distortion plots for a different execution time of the optimisation. These plots illustrate how the accuracy of the optimisation decreases with a lower number of visited points. This is because the real ratio between the decrease of error and the increase of cost $\frac{\Delta E}{\Delta C}$ deviates too much from the derivative $\frac{dE}{dC}$ of the function ax^b when the number of points is decreased.

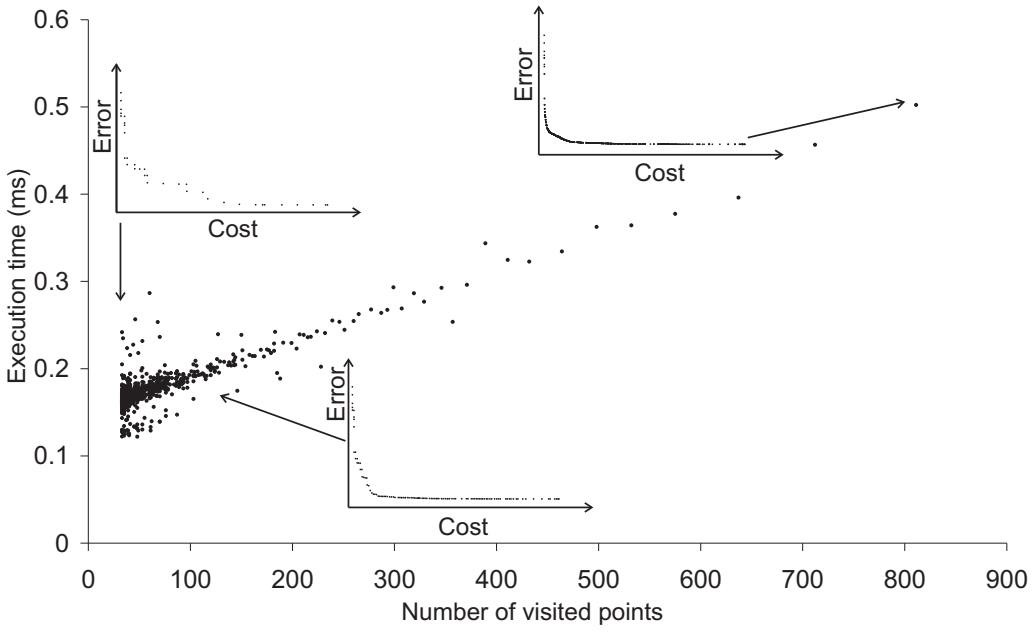


Figure 2.8: The execution time (including visibility determination) and resulting online distortion plots of the optimisation as function of the number of visited points to find the minimum error.

The highest execution time of the optimisation was equal to 6% of the rendering time for the used city scene. This is similar to the optimisation execution times

reported by Funkhouser et al. [52] (average 10% and maximum 30%) and Gobbetti el al. [56] (between 5% and 20%), which use very simple heuristics for calculating the visual quality.

Measuring the visual quality presented in Chapter 4 takes 170 ms for a simple object with 200 triangles in the approximating mesh and 1500 triangles in the original mesh. Evaluating the visual quality for the city scene of Figure 2.8 completely online would take at least 20 s, against which the 0.6 ms from Figure 2.8 is negligible.

2.5 Example plots

The next chapters will frequently use global online distortion plots which actually show all the intermediate solutions of the optimisation to find the actual minimum value. This section illustrates some example online distortion plots for three different scenes.

Figure 2.9 shows a view-dependent distortion plot of an object (Mr. Potato from the Toy Story movie) for three viewpoints: front (left), back (middle) and top (right) view. All quality numbers are given relatively against the worst quality measure for the back viewpoint. The quality of the approximation for the front view object is, for the lowest quality settings, a little better than for the back view scenario. This improvement is because of the influence of the extra meshes (eyes, nose, ...) on the (area) averaged quality. Because the front view scenario is a scenario with all meshes visible, the cost is always the highest for this scenario. For the back view, meshes are culled and the cost decreases for similar qualities. The quality increase is for the front and back view scenario mainly visible through a smoother visualisation of the object, especially at the silhouette. For the up view, the base mesh representation of the hat is already a good approximation and a very low approximation error is obtained at low cost. As can be seen in the distortion plot, adding more triangles improves the quality, but not so drastically as for the back and front views. The quality increase is here mainly visible by the better spot on the hat.

Figure 2.10 shows global distortion plots for a chess scene. This scene is built from scalable (the chess pieces and checkerboard frame) and non-scalable (checkerboard and table) objects. In the picture on the right, the chess scene is seen from below and all scalable objects are culled; therefore, the online distortion plot collapses to a single point on the x-axis giving the cost of the non-scalable geometry. For the middle picture, the scalable objects are closer to the screen than for the left picture. Therefore, artifacts on the objects are more visible and the errors on the online distortion plot are larger than for the left picture.

Figure 2.11 shows different online distortion plots for different viewpoints in the city scene that contains scalable (high complexity lampposts and character) and non-scalable (the low complexity buildings) content. The distortion plots that are shown

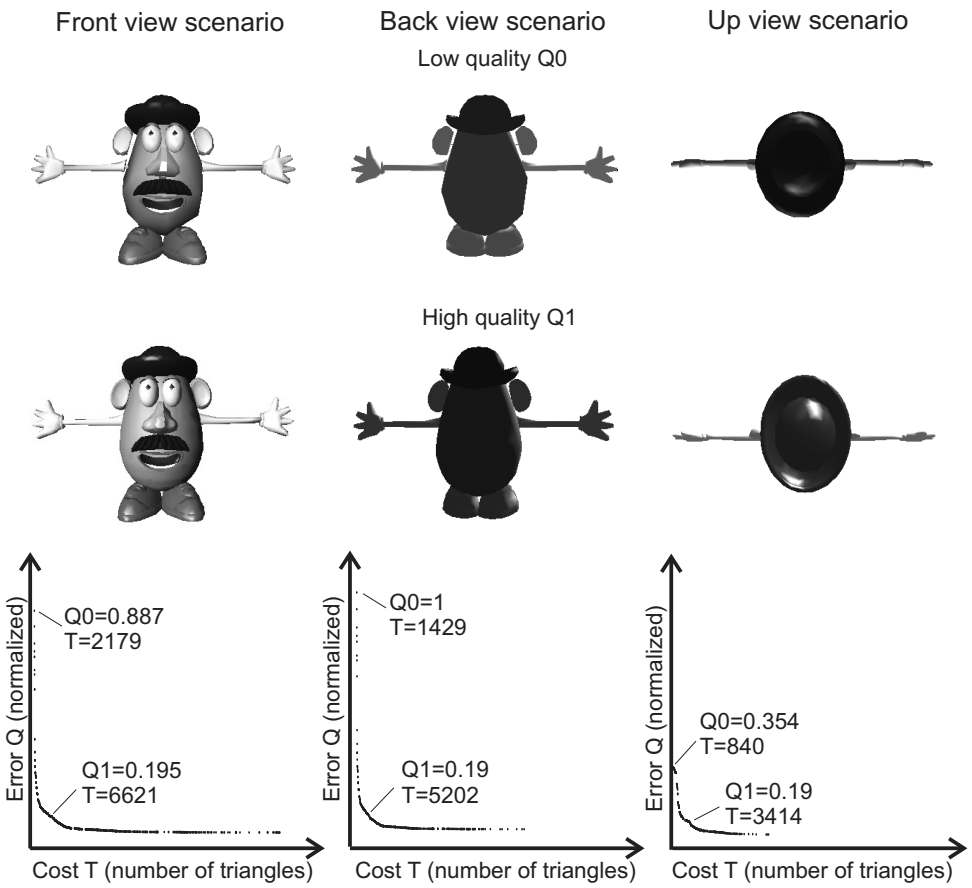


Figure 2.9: Several examples of distortion plots for a single object seen from different viewpoints.

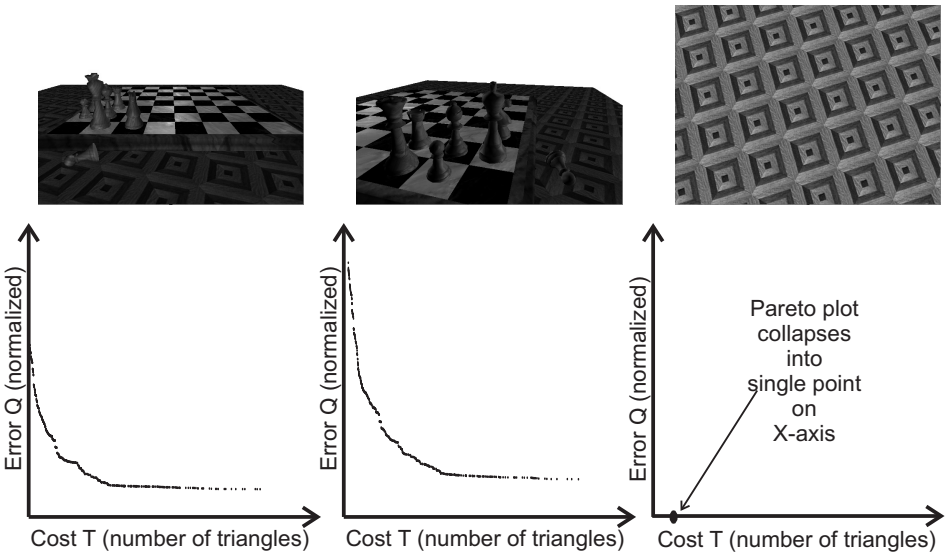


Figure 2.10: Several examples of online distortion plots for a simple scene seen from different viewpoints.

for these viewpoints are snapshots of the global distortion plot for the walk through from point a to point d. Note that although the distortion plots of point c and point d look very similar, they contain the working points of totally different objects.

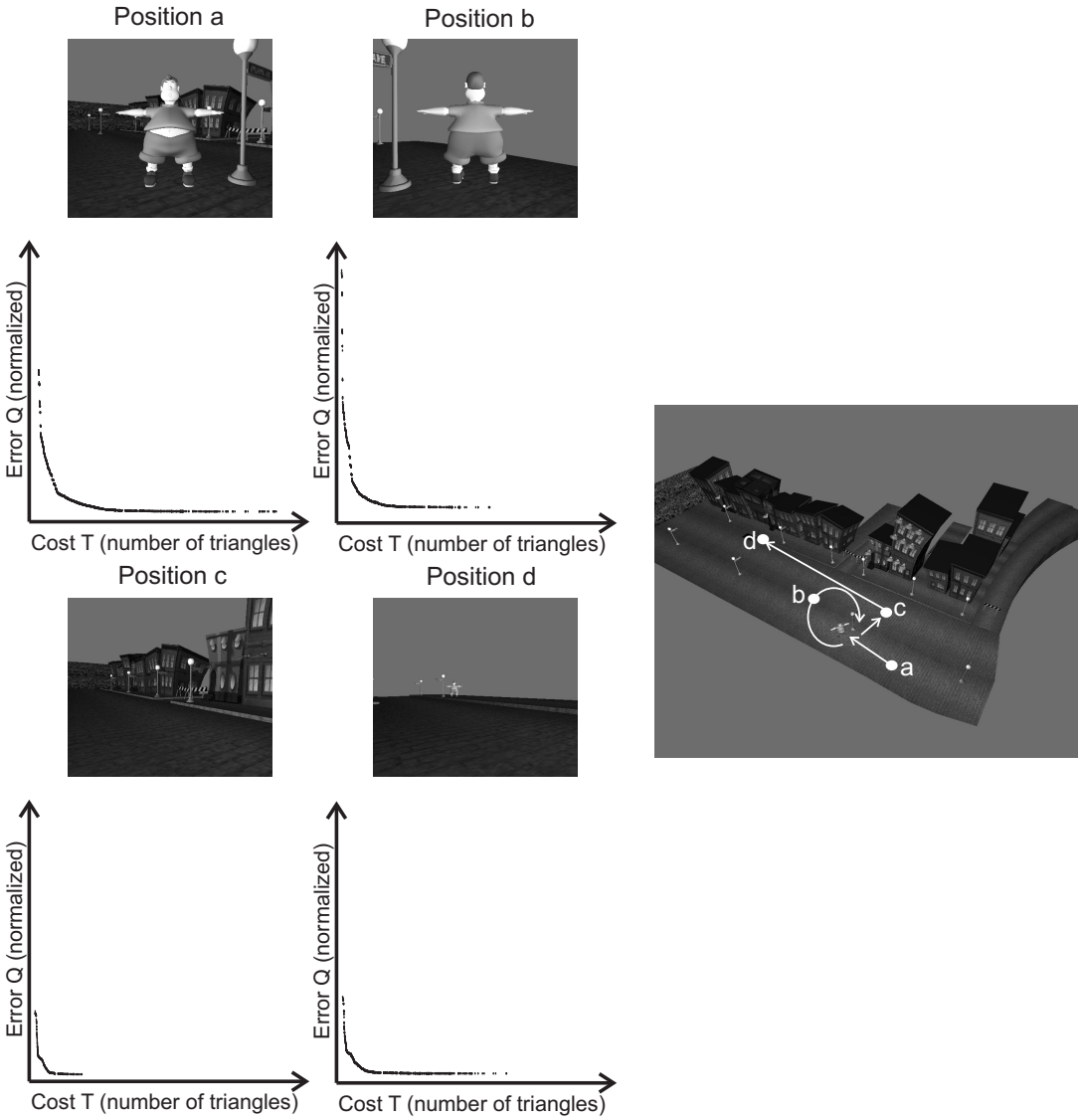


Figure 2.11: Several examples of online distortion plots for a more complex scene seen from different viewpoints.

2.6 Conclusion

This chapter introduces the optimisation framework. This framework automatically adapts the Level Of Detail (LOD) of the 3D content to the available resources on the terminal with a maximum visual quality.

The basic strategy underlying our work is to shift the complex parts of the work to off-line preprocessing steps and to reduce the online work to simple combination steps. This enables the use of complex but realistic quality models. The preprocessing work is summarised in distortion plots that store the optimal trade-offs between the quality and the platform independent cost for a single object. A simple gradient descent method combines these distortion plots to find the optimal settings of the decoder that yield a maximum visual quality for a given maximum triangle budget.

Chapter 3

Modelling performance

The optimisation presented in Chapter 2 uses the number of triangles as a cost metric. This chapter relates this platform independent cost metric to the real execution time for two different types of platforms: an embedded device using software rendering and a PC using hardware accelerated rendering. This issue is usually addressed along two main lines of thought. Either the execution time is continuously measured and the triangle budget is slightly adapted from frame to frame with an execution time control loop, or the triangle budget is calculated from an explicit performance model.

Because of the large consistency in subsequent frames, flight simulators frequently use measurements for controlling the execution time. This method is however unable to control the execution time well for very abruptly changing content [52]. For instance, if the user steps through a door between a room and a corridor in a building walk through, the number and complexity of the objects visible to the user suddenly changes and measurements of the execution time in the room cannot be used for controlling the execution time in the corridor. For this type of scenes, triangle budgets are better estimated with a model of the execution time, which is further discussed in this chapter.

Section 3.1 starts the chapter with previous work on performance models for 3D graphics. Because hardware accelerators were not yet available on mobile devices at the time of the research, a performance model for a software renderer was needed. Therefore, Section 3.2 discusses the method for deriving this performance model. This method can however also be used to model the execution time of other code (e.g. the multi-resolution decoder of Chapter 5). Section 3.3 then describes the performance models for hardware accelerated systems and Section 3.4 ends with the conclusions.

3.1 Related work

Funkhouser and Séquin [52] were among the first to propose a technique for optimising discrete Levels Of Detail. They have identified the need for an execution time model for a good performance of the rendering time control system of dynamic scenes, while preventing overshoot and oscillations. Their cost heuristic was derived from the Graphics Library Programming Tools and Techniques document [70] that represent the rendering system as a pipeline with two functional stages: the geometry and fragment stages. Since the separate stages of the pipeline run in parallel, the throughput of the pipeline is determined by the speed of the slowest stage. Consequently, they have modelled the time required to render an object as the maximum of the times taken by any of the stages (Equation 3.1).

$$T = max(V \cdot T_v + F \cdot T_f, P \cdot T_p) \tag{3.1}$$

The parameter V in Equation 3.1 is the number of vertices, F is the number of triangles, P is the number of fragments and T_v, T_f and T_p are respectively the time needed to process a vertex, triangle or pixel. Most authors still use Equation 3.1 in more or less the same form. Aliaga et al. [2] for instance simplified Equation 3.1 by neglecting the influence of the fragment stages.

A more recent study of Wimmer and Wonka [162] re-evaluated Equation 3.1 and some of its derivatives for a more recent graphics chip architecture: the Geforce 3 [77]. They have correctly mentioned that the speed of an application is not defined by a single bottleneck. While this is true for each particular point in time, the bottleneck can change several times even during the rendering of a single object. Indeed, the fragment and geometry stages can work in parallel only for one particular triangle size, which depends on the complexity of the used vertex (T_v/T_f) and fragment algorithms (T_P). Larger triangles usually stall the geometry unit, whereas smaller triangles will cause the fragment stage to sit idle. As a result, Equation 3.1 is an underestimate of the real processing time; therefore, Wimmer and Wonka have proposed to use a sum instead of a maximum function as an upper bound of the execution time. Their experiments on the Geforce 3 have suggested that the actual rendering time often tends towards the sum.

$$T = max(V \cdot T_v + P \cdot T_p) \tag{3.2}$$

The two main parameters of the performance of a graphics system are the fragment (T_p) and triangle (T_v/T_f) throughput of the graphics pipeline. These parameters can be measured for a *graphics chip* with several benchmarks, such as 3D mark [35] and SPECglperf [30]. These benchmarks use many different applications for measuring the performance and typically summarise the results in simple average numbers.

The book GPU Gems [42] reveals more information on the detection of performance bottlenecks for one specific *3D application*, which is supported by several tools (e.g. NVPerfHUD [106], NVPerfKit [107] and PIX [93]). The book GPU Gems 2 [117] reveals more information on the high-level architecture of one of the latest graphics processors (the Geforce 6800).

Sheaffer et al. [135] have developed a simulator of the architecture of the Geforce 4 chip for thermal modelling and the detection of thermal hot-spots in the chip. This simulator also includes a rough estimate of the power/energy consumption of the hardware. Crisu et al. [36] have developed a high-level simulator of 3D graphics chips for mobile devices. This simulator allows exploring the different design choices of the chip at a high abstraction level. It also includes estimates of the energy consumption because it is oriented towards mobile devices,

A first approach for modelling the performance is to use black box modelling (e.g. Bontempi et al. [18]) that selects the parameters to which the execution time is sensitive and then derives the relation f between the parameters and the execution time by measurements with known parameters. A second approach for modelling the performance is to use white box modelling (e.g. Reichel [120] and Saavedra [125]) that selects the parameters to which the resource consumption is sensitive and that derives the relation f between the parameters and the resource consumption with an analysis of the source code.

Low cost mobile devices typically do not have a graphical processing unit and thus execute the visualisation task on the CPU in software. Therefore, the present work uses a white box approach for modelling the software rendering. Because hardware rendering pipelines are typically a black box, the black box approach has been used for modelling their performance. The performance models discussed by Wimmer and Wonka [162] are tested and improved for a recent graphics architecture (Geforce Fx 5900). The software performance models have been tested on a Pentium IV and on the Intel 80200 (XScale) processor. The performance model was not yet tested on VLIW processors, which can possibly introduce some non-linearities.

3.2 Method for Deriving the Software Rendering Performance Model

The first step to model the execution time, discussed in Subsection 3.2.1, is to profile the source code to extract its important parts for the execution time. A second step, described in Subsection 3.2.2, analyses the source code and extracts the important parameters of the model. The result is an analytical function describing the execution time as a function of the processed data. Subsection 3.2.2 also discusses how these data parameters are estimated for our software rendering pipeline.

Although the initial analysis of the source code may reveal a simple performance model, the coefficients of the model typically vary because of the architecture of the used hardware, performance model simplifications, etc. An important example for the architectural dependencies, discussed in Subsection 3.2.3, are the cache misses into the texture memory that influence the pixel processing time of our software rendering pipeline.

A first approach for taking these deviations into account is to model this cache behaviour. The cache behaviour however depends on many a priori unknown parameters and the complexity of the performance model would increase a lot when they are included. Therefore, subsection 3.2.4 describes how the performance model is made adaptive and how the coefficients are derived with a combination of off-line calibration and online correction (based on for instance a recursive least-squares algorithm [16]). Results are presented in the last subsection.

3.2.1 Profiling the Code

Some of the performance model parameters are only known when the modelled task has finished its execution. These parameters must thus be estimated before the task has started for a correct prediction of the execution time. However, the performance model is used in an interactive application and a fast evaluation is thus needed. The addition of a priori unknown parameters to the performance model must thus be carefully balanced between their contribution to the accuracy and the extra complexity because of their inclusion in the model.

A first step is therefore to determine the importance of the building blocks of the source code for the execution time. This gives a first indication of the needed accuracy of the performance model for that specific part. Figure 3.1 for instance illustrates how much time is spent in the different building blocks of the software 3D rendering pipeline. Almost all processing time is spent in the vertex processing and rendering units. Other initialisation code is (as expected) negligible and must be excluded from the performance model. On average 80% of the time is spent in the rendering, which is thus the most important unit for the performance model.

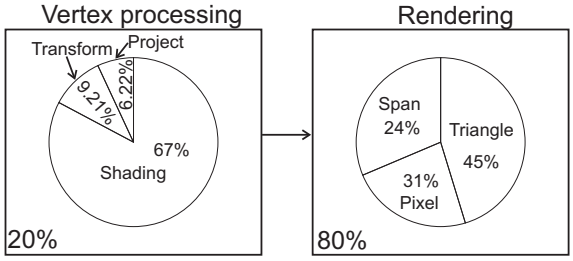


Figure 3.1: High-level overview of the complexity of the software 3D rendering pipeline.

Note that this step is only an initial step in the performance modelling and because it is executed on a different machine than the one used for the rendering, it reveals little information on the performance model coefficients. An advantage of this step is however that the information obtained with the profiling tool can be used to determine and optimise the bottlenecks in the code. In addition, some profiling tools reveal information to detect and remove memory leakage (e.g. Rational PurifyPlus [69]).

3.2.2 Extract Data Dependencies

A careful analysis of the control flow of the important units derives the data parameters of the performance model. This analysis extracts the loop bounds and if-conditions with a significant impact on the execution time.

With loops, the code of the loop body is repeated n times, multiplying its execution time accordingly by n. Loops may be eliminated from the performance model if the variation on the loop bounds is limited. Our experiments with the software renderer on the XScale and Pentium IV processor typically derive a linear relationship between the loop bounds and the execution time. This linear relationship is however not always guaranteed, e.g. on a VLIW processor. In this case, a modular technique for local modelling [17] can be used to derive a piecewise linear performance model. This technique then derives scenarios (or operating regimes) with different ranges for the loop bounds and an accompanying linear model. Because these scenarios depend on the platform architecture, the off-line calibration of the model, discussed in Section 3.2.4, is responsible for deriving the different scenarios.

Branching includes and excludes code, and the execution time will vary according to the selected branch complexity. A straightforward approach for modelling the execution time of the branch is to use the mean value $E[T]$ and the variance $Var[T]$ of the execution time T_i of each of the branches (see Equation 3.3).

$$\begin{array}{lcl} E[T] & = & p \cdot T_1 + (1-p) \cdot T_2 \\ Var[T] & = & p \cdot (1-p) \cdot \Delta T^2 \end{array} \tag{3.3}$$

with p the probability that the branch is taken. The variance of Equation 3.3 is a guideline for the selection of the branches that may be eliminated from the performance model. The following cases can be the reason for the model simplification:

1. Simplification 1: the complexity of the branches is more or less equal ($\Delta T \approx 0$ and $Var[T] \approx 0$). The model is then simplified to $T \approx T_1 \approx T_2$.

2. Simplification 2: the probability of one of the two branches is below a specified threshold. The model is then for instance simplified to $T \approx T_2$. This threshold depends on ΔT. With $\Delta T \approx 0$ (simplification 1), the parameter p may be

anything between 0 and 1. For a large ΔT, the probability that the event will take place while rendering a frame must be more or less equal to 0. In the latter case, the error because of this simplification can be large when the unlikely branch is selected, but the frequency of occurrence of this error must be low.

3. Simplification 3: the probability p that the branch is taken is more or less a constant value over time ($\frac{dp}{dt} \approx 0$ and $Var[T] > 0$). In this case the branch may be replaced by its expected value.

The parameter p must be included in the performance model in all other cases. The parameter selection is illustrated with the software renderer that was used on the embedded device. Figure 3.2 gives an overview of all the symbols that are used for illustrating the modelled source code in this text. The state condition of Figure 3.2 is a special if-condition that selects the same branch (state) for a complete group of vertices or triangles. Only one path of the if-condition is followed for the complete group of triangles and simplification 1 can be used that eliminates all other branches of the if-condition from the performance model. A state condition however incurs that for every possible state, a different performance model must be derived.

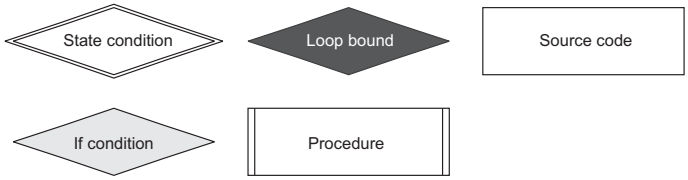


Figure 3.2: Legend for the control flowcharts.

The first part of the rendering pipeline [63] [46] is the vertex processing unit illustrated in Figure 3.3. Its first task is to execute the modelview transformation (matrix multiplication) to transform the object into the camera's coordinate space. In the rendering pipeline, vertices are only further processed when their depth falls into the covered range. The difference in the vertex processing time of the branches is high (90% of the execution time is spent after the modelview transformation), but the number of vertices that are eliminated from the rendering pipeline with this branch is relatively small. Because objects with all vertices clipped are easily detected before the rendering, this branch is only taken if an object is only partially in the depth range of the visualisation. In addition, objects that are far away and which are thus candidate to be partially clipped are not well visible. These objects are typically represented with a low number of triangles. Simplification 2 can thus be used for eliminating this branch from the performance model.

The next stages in Figure 3.3 project the vertex from the 3D space onto the 2D screen (matrix multiplication) and calculate the vertex colours with a standard illumination model (Equation 3.4).

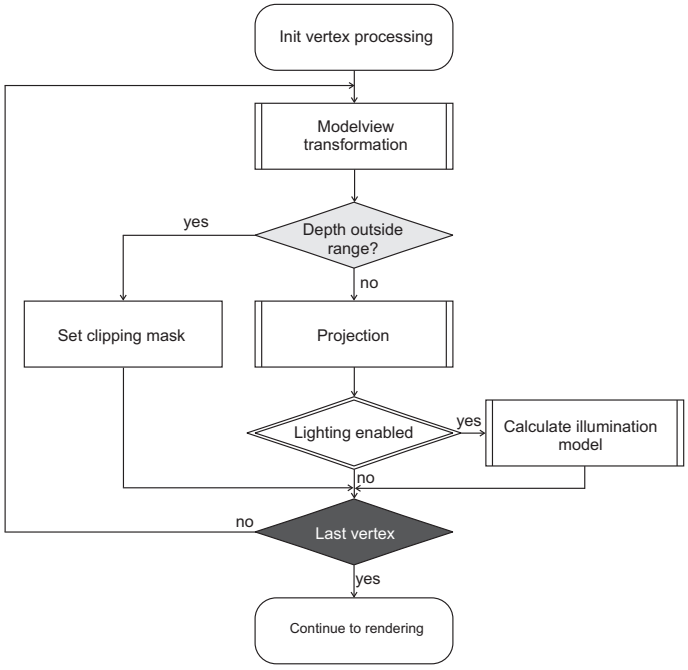


Figure 3.3: Flowchart of the code of the vertex processor.

$$C = \left\{ \begin{array}{l} e_r + I_{mr}\rho_{ar} + \\ \sum_{i=0}^{i=L} A^i \left(I_a^i \rho_a + I_d^i \rho_d M\left(0, \frac{\overrightarrow{s^i}\overrightarrow{m}}{|\overrightarrow{s^i}||\overrightarrow{m}|}\right) + I_s^i \rho_s M\left(0, \frac{\overrightarrow{h^i}\overrightarrow{m}}{|\overrightarrow{h^i}||\overrightarrow{m}|}\right)^f \right) \end{array} \right. \tag{3.4}$$

with $\overrightarrow{m}$ the normal vector on the vertex, $\overrightarrow{s}$ the vector from the light source to the vertex, $\overrightarrow{h}$ the halfway vector ($\overrightarrow{v} + \overrightarrow{s}$), $\overrightarrow{v}$ the vector from the viewpoint to the vertex, M the maximum function, I the intensity of the light source for the specific type of light (a=ambient, d = diffuse and s = specular), ρ the reflectivity of the material for the specific type of light, L the number of light sources, f the specular power term (a large f will give small intense spots, a small f gives a smeared out spot), e_r the light emitted by the object, and A^i the attenuation factor of the light because of the distance and spot light factor (the spot light factor is not implemented in the software rendering pipeline, but is needed for the next subsection).

The two maximum functions in Equation 3.4 incur two if-conditions that only allow the computation of the diffuse and specular terms of Equation 3.4 if the dot product is larger than zero, i.e. the vertex is seen by the light source. Because this if-condition uses the vertex coordinates after the modelview transformation, the number of taken branches is a priori unknown and must be estimated before the rendering has started. However, step 1 has indicated that only 20% of the execution time is spent in the vertex processing unit. In addition, one can assume that for most objects the triangles are more or less equally distributed over the object. For these objects more

or less 50% of the vertices are seen by the light source and simplification 3 may be used to eliminate this branch from the performance model. Equation 3.5 defines the finally obtained performance model of the vertex processing unit.

$$\begin{aligned} T_v &= V \cdot (T_{mp} + L \cdot T_l) \\ T_l &= T_{iL} + 0.5 \cdot (T_d + T_s) \end{aligned} \tag{3.5}$$

with V the number of vertices; T_{mp} the time needed for the modelview transformation and projection; L the number of light sources; T_{iL} the time needed for initialising the lighting of the vertex; T_d the time needed to compute the diffuse component and T_s the time needed to compute the specular component.

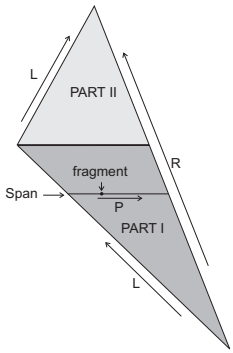
The second part of the pipeline is the rendering or rasterisation stage, which actually draws the projected triangles. The procedure is illustrated with Figure 3.4(a) and the flowcharts of the source code are given in Figure 3.4(b) and Figure 3.4(c).

A performance model must be derived for every possible rendering state, i.e. for wireframe, smooth shaded or flat shaded rendering in combination with the texture filter and the modulation function [63]. The first steps of the rendering are some visibility checks to eliminate back facing triangles and triangles outside the screen. Visible triangles are split in two parts because the slope of the edges on the left of the triangle in Figure 3.4(a) changes and must be recalculated. Normally, both parts are visible and simplification 2 is used to eliminate the branches that include or exclude the part drawing from the performance model. If the position of one or two of the vertices is outside the viewing frustum, the part initialisation corrects them so that rendering 3.4(a) starts and/or ends at the borders of the viewing frustum. Because this correction is executed on few triangles and its additional complexity is low, simplifications 1 and 2 can be used to eliminate this branch from the model. A first loop then interpolates the attribute values of the vertices over the edges L and R to compute the necessary information on the endpoints of the span. A second loop interpolates the information on the endpoints on the span to derive the information of a fragment (a pixel with extra information attached to it). If the depth of this fragment is lower than the value in the depth buffer, the texture filter and modulation function are calculated and the resulting pixel is drawn. Equation 3.6 defines the performance model of the rendering.

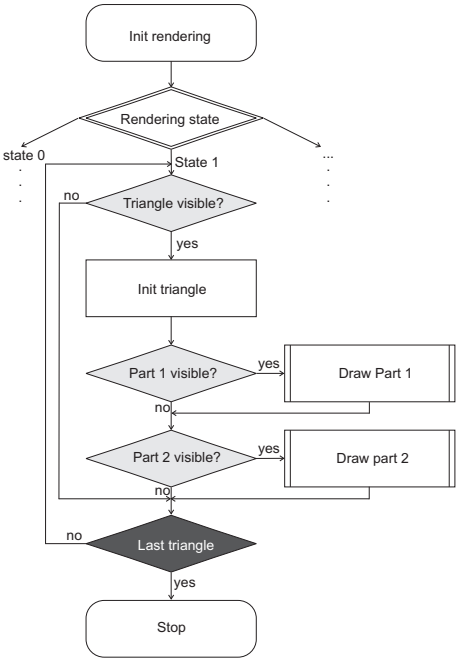
$$T_r = F_v \cdot T_f + S \cdot T_s + P \cdot T_p \tag{3.6}$$

with F_v the number of *visible* triangles and T_f the time needed to process the triangle; S the number of *visible* spans and T_s the time needed to process a span; P the number of *visible* pixels and T_p the time needed to process the pixel.

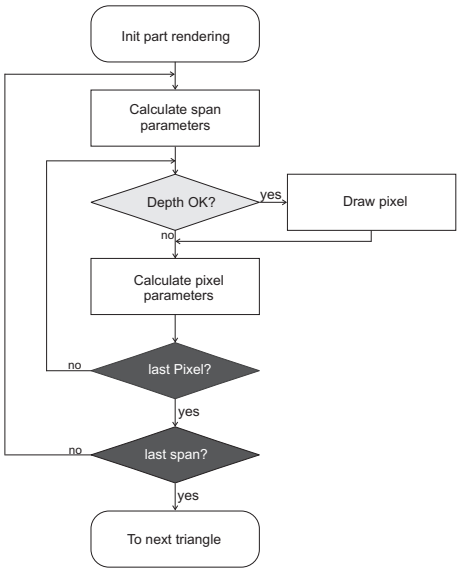
The data parameters of the performance model such as the number of light sources and the number of vertices are easily derived from the scene description, but other parameters such as the number of visible triangles, fragments and spans are only



(a) Procedure



(b) Flowchart



(c) Flowchart Draw part

Figure 3.4: The rasterisation: (a) procedure, (b) flowchart for triangle processing and (c) flowchart for span and pixel processing.

known after the actual rendering has finished. These parameters can be estimated before the rendering has started with the visibility sphere around the object (Figure 3.5). This visibility sphere is derived in the off-line preprocessing steps described in Chapter 2 when the visibility is checked of the object parts. The visibility sphere stores the values of the data parameters for viewpoints that are located on its vertices at a unit distance.

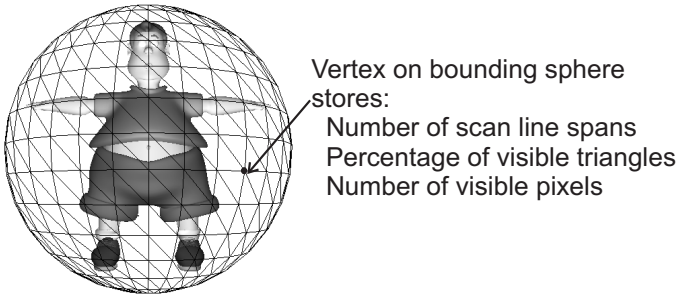


Figure 3.5: The visibility sphere around the object that stores performance model parameters.

Figure 3.6 shows the accuracy of the estimation of the number of pixels with the visibility sphere. The average error is for this experiment 2.3% and the maximum error is 13%. Figure 3.6 was derived for a single object to exclude errors because of overdraw. Indeed, if a hidden object is rendered first in a system with z-buffering, then the rendering system cannot detect in the depth buffer that this object is actually hidden and it needs to draw the object. A good occlusion culling that eliminates these hidden objects before the rendering is thus beneficial for both the rendering performance and the predictability.

At run-time, the viewpoint is projected onto the visibility sphere and the number of fragments, scan line spans and the ratio of the number of visible to the total number of triangles are then derived from the nearby vertices on the visibility sphere. The next step after reading these parameters from the visibility sphere is to scale the number of fragments and spans to the current viewport size and the actual distance between the viewpoint and the object. As illustrated in [148], the number of scan line spans not only depends on the position of the object in the scene, but also on its resolution. Therefore, the number of scan line spans of the lowest resolution mesh is stored on the visibility sphere and the ratios between the actual number of scan line spans for a specific resolution and the number of spans for the lowest resolution is then stored with the distortion plots used in the optimisation. This enables to correct the initial triangle budget while visiting the points on the distortion plots with the gradient descent optimisation.

The actual number of fragments, spans and triangles further depends on the visibility and rendering order of the individual objects. Early z-rejection prevents the

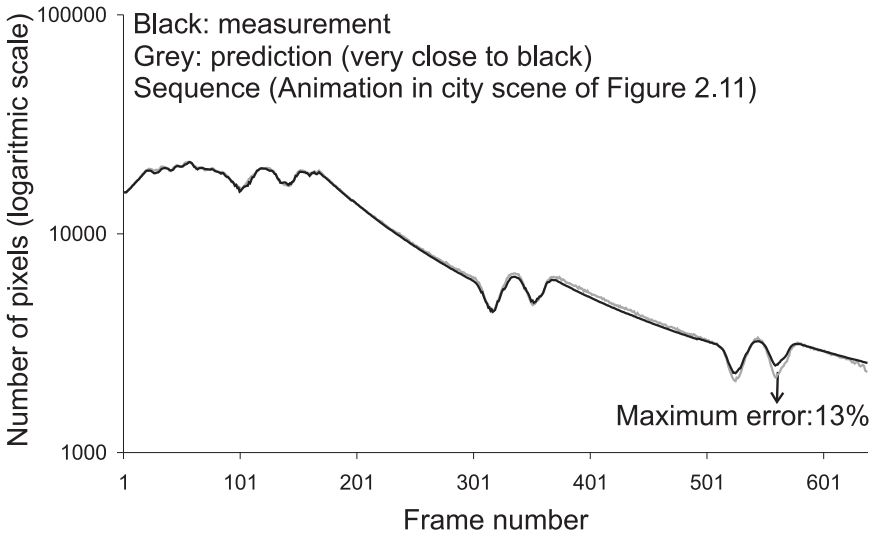


Figure 3.6: Estimation (Grey) versus measurement (Black) of the number of pixels with a visibility sphere for the animation in the city scene of Figure 2.11.

texturing and shading of fragments that are hidden by other already rendered fragments. Estimating the number of fragments that are hidden by early z-rejection is not a simple task and because it actually depends on the changing position of the objects and the user in the scene, it cannot be estimated in a preprocessing step. It is therefore necessary to minimise the number of processed but invisible fragments by a good occlusion culling. The remaining difference in the actual number of processed fragments and the estimated number of fragments can be corrected with an online correction, as will be explained in Subsection 3.2.4.

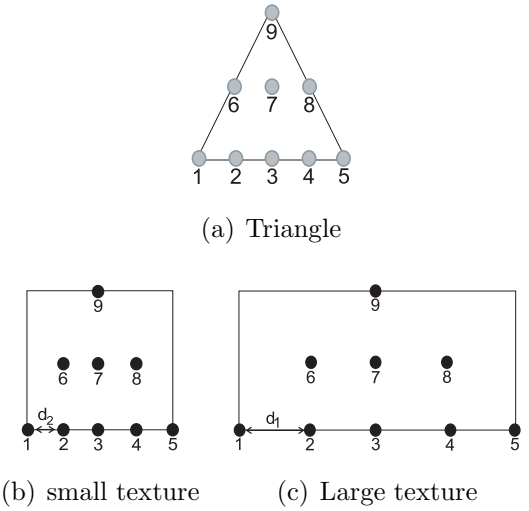
A second method that can be used for estimating the parameters is to measure them. With slow varying viewpoints, there is typically a large amount of coherence between successive frames and a lot of parameters (e.g. pixels, spans, ...) of the current frame are more or less equal to the parameters of the next frame. If the source code is accessible, counters can be added that measure the number of pixels, spans, ... These measurements are then used to estimate the next frame's parameters.

3.2.3 Platform Dependencies - the reason for an adaptive model

The performance model of Subsection 3.2.2 assumes constant performance model coefficients, i.e. a constant time per unit of data. This neglects the impact of the platform architecture and the performance model simplifications. An important issue for instance in the design of a system is the design of the memory hierarchy based on the principle of locality: i.e. programs tend to (re)use data and instructions that

they have recently used and programs tend to use data and instructions that are located near to recently used data and instructions.

The locality of the data and thus the performance of the memory system however vary for the different 3D objects in the scene. Figure 3.7 for instance illustrates how the memory distance between two accessed texture pixels (texels) depends on the texture size. The grey pixels in the triangle of Figure 3.7 sample a texture colour from the large or small sized texture. Texture coordinates are represented as normalised values between 0 and 1 and are multiplied with the texture size in the renderer. The same triangle will thus cover the whole range of both textures with the same number of texture samples. The distance d_2 between the samples on the small texture is thus smaller than the distance d_1 on the large texture, which improves its cache behaviour and thus the execution time.



(a) Triangle

(b) small texture (c) Large texture

Figure 3.7: Locality as a function of texture size: a) triangle that samples the texture, b) large texture with bad locality and c) small texture with good locality.

An additional parameter that influences the texture cache locality is the orientation of the triangle after the modelview transformation and projection. Figure 3.8 illustrates this with the triangle of Figure 3.7 that is now rotated. Its samples 1 to 5 are now located on a sloping line of the triangle in Figure 3.8. The spans of the triangle in Figure 3.8 (e.g. 6 to 5) however remain horizontal and its pixels sample the texture on a sloping line. Because our software renderer orders the textures line per line in memory, the orientation of the triangle influences locality and cache behaviour. Hakura et al. [60] have proposed to eliminate the dependencies of the triangle orientation with a block based representation of the texture in memory.

Figure 3.9 illustrates the number of cache misses for the used software renderer as a function of texture size and orientation. The texture filter that was used for Figure 3.9 is a simple filter (nearest sampling, without mipmapping) that samples

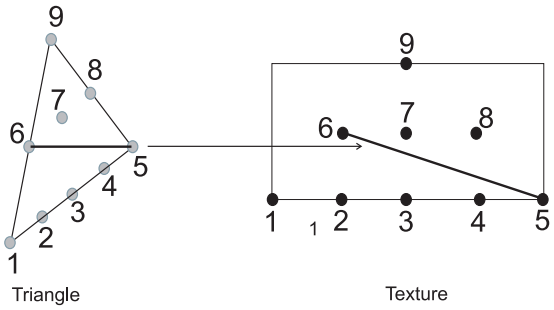
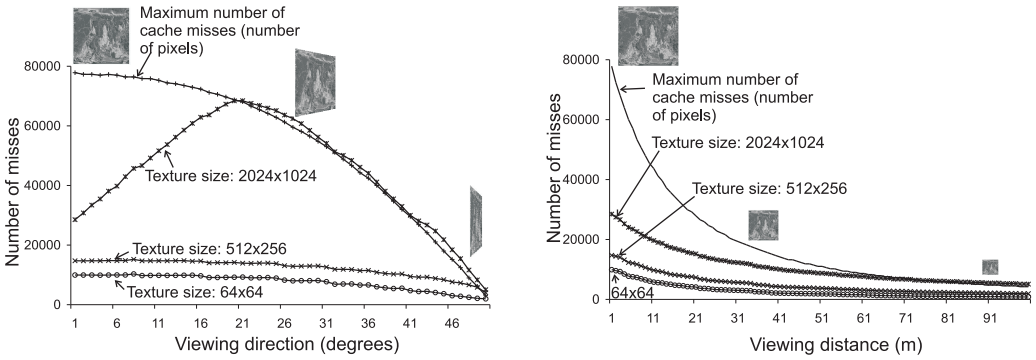


Figure 3.8: The number of cache misses increases because a horizontal scan line span of the triangle samples the texture on a sloping line.

one texel per triangle pixel. Figure 3.9.a shows the dependency of the number of cache misses on the size of the texture and the rotation of the triangle around its vertical axis. The number of misses first increases with the orientation because more pixels generate a cache miss, but then decreases because the number of pixels decreases with the rotation of the triangle around its vertical axis. The number of cache misses decreases for Figure 3.9.b because the number of pixels decreases very fast with an increasing distance to the viewpoint. However, relatively seen, the percentage of pixels that generate a cache miss increases with an increasing distance to the viewpoint. Indeed, the smaller triangles still sample the same range of the texture, which results in a larger distance between subsequent texture samples. The result for the execution time is shown in Figure 3.10, which plots the extra time needed for rendering a pixel as a function of texture size and orientation.



(a) Viewing direction (b) Viewing distance

Figure 3.9: The number of cache misses as a function of texture size: (a) for different triangle orientations, (b) for different distances to the viewpoint.

The experiments described in this subsection are specifically designed for showing the effect of the texture cache misses on the execution time. A real 3D application renders a mixture of texture and triangle sizes, with different texture filters (e.g.

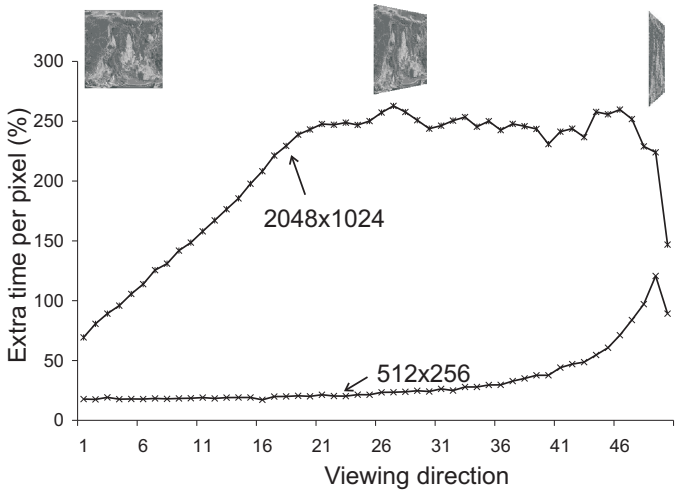


Figure 3.10: The extra time needed per pixel as a function of texture size and orientation.

mipmapping increases the texture cache efficiency [142]) and possibly different texture representations (e.g. block based). Taking all these effects into account in the performance model is not a straightforward task and would increase the performance model's complexity significantly. Therefore, Subsection 3.2.4 proposes an online correction system that uses measurements of the execution time to correct (calibrate) the performance model at run-time.

3.2.4 Deriving the performance model coefficients T_i

Average values for the performance model coefficients T_i are derived with an off-line calibration, discussed in the first subsection, which measures the execution time of the rendering pipeline for known and varying data-parameters. A model is then fit onto the measurements using regression techniques [98]. The real online model coefficients however deviate from these average values because of the model simplifications, rough estimations of the data-parameters and architectural dependencies. An online correction algorithm (e.g. Recursive Least Squares [17]) makes the model adaptive and updates the linear performance approximation to correct the average to the instantaneous values.

Off-line calibration

The calibration of the vertex processing time T_v (Equation 3.5) is done on several objects with a varying number of vertices. To derive the average value for the modelview time T_{mp}, calculating the illumination model is disabled so that its code

is not executed. For deriving the average value of the lighting time T_L, the illumination model is enabled with 7 light sources. To prevent non-linearities in the measurements, the rendering and the objects are initialised (e.g. prevent clipping, all vertices are seen by a light source, etc.) so that all vertices of the objects are equally treated, i.e. follow more or less the same program path. Figure 3.11 shows the resulting execution time as a function of the number of vertices. Two linear regressions are then used to initialise the coefficients T_{mp} and T_L.

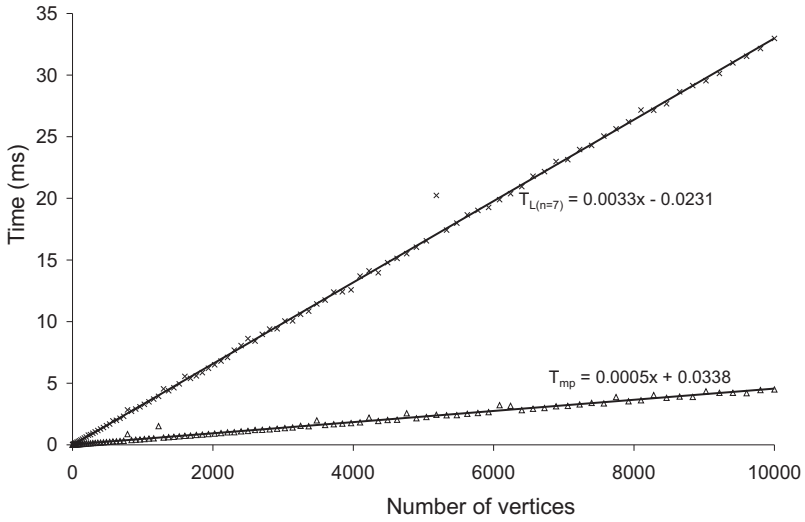


Figure 3.11: Calibration of the vertex processing model coefficients.

The rasterisation time (Equation 3.6) depends on three variables: the number of visible triangles (coefficient T_f), the number of spans (coefficient T_s) and the number of visible pixels (coefficient T_p). The calibration of T_p is done on a mesh that is zoomed out so that the number of pixels varies without varying the number of triangles. The content in this experiment is also initialised in such a way that non-linearities because of a different treatment of the pixels is prevented as much as possible (e.g. prevent overdraw, clipping, etc.). If the number of pixels varies, then typically also the number of scan line spans varies and a part of the execution time variation should be fit onto T_s. However, because the total number of scan line spans is equal to the product of the number of triangles multiplied with their height, the number of scan line spans is low if the total number of triangles is also low. Indeed, for a single triangle, the number of spans is equal to its *height*, while the number of pixels is equal to $width \cdot height/2$ with $width/2$ typically much larger than 1. In this case, the span processing time may be neglected and the execution time of the rendering shows a linear relationship with the number of pixels. This animation is repeated for every rendering state that is used in the application.

Because the number of scan line spans depends on the number of triangles, a variation on the number of triangles will also vary the number of scan line spans. Therefore, a second order linear model is fit onto the measured values of T_f and T_s as

a function of the number of triangles and the number of spans. This is illustrated in Figure 3.12 that shows the plane fitted onto the measurements of the execution time. This test is also repeated for every possible rendering state that is used in the application.

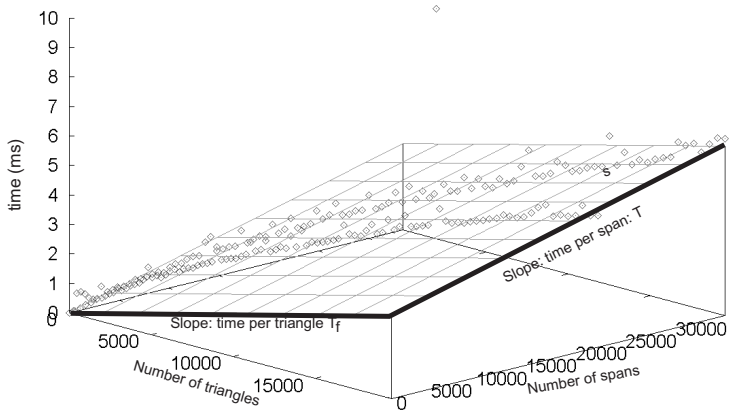


Figure 3.12: Calibration of the triangle rasterisation model coefficients.

The software performance model was tested on two different platforms: an embedded device with an XScale processor and a PC with a Pentium IV processor. For these platforms, the off-line calibrations typically yield a linear model. This linearity is however not guaranteed for all platforms. The performance of a VLIW processor for instance heavily depends on the compiler technology; therefore, It may be expected that this causes some non-linearities in the model. Figure 3.13 for instance shows a varying performance of an non-optimised [1] software renderer on a TriMedia 1300 processor as a function of the number of triangles. A divide and conquer approach with a modular technique [17] can then be used to divide this curve into two regions (called scenarios) and to derive a linear model for each one of the scenarios.

Online correction

Deviations on the calibrated average values for the coefficients of the performance model are corrected online with a feedback mechanism that corrects the coefficients with measurements of the execution time. Figure 3.14 illustrates where the measurements of the execution time could be added to the code of the rendering pipeline. To derive detailed information about all performance coefficients, a very detailed profiling is needed. This increases the accuracy of the online correction, but it also increases the execution time overhead. For instance, the overhead (measured on a Pentium IV processor) that is introduced by adding these measurements is 10% for adding timers 1 and 2, 60% for adding timer 3 and 200% for adding timer 4. Adding

[1] it may be expected that the non-linearity will be stronger with an optimised code.

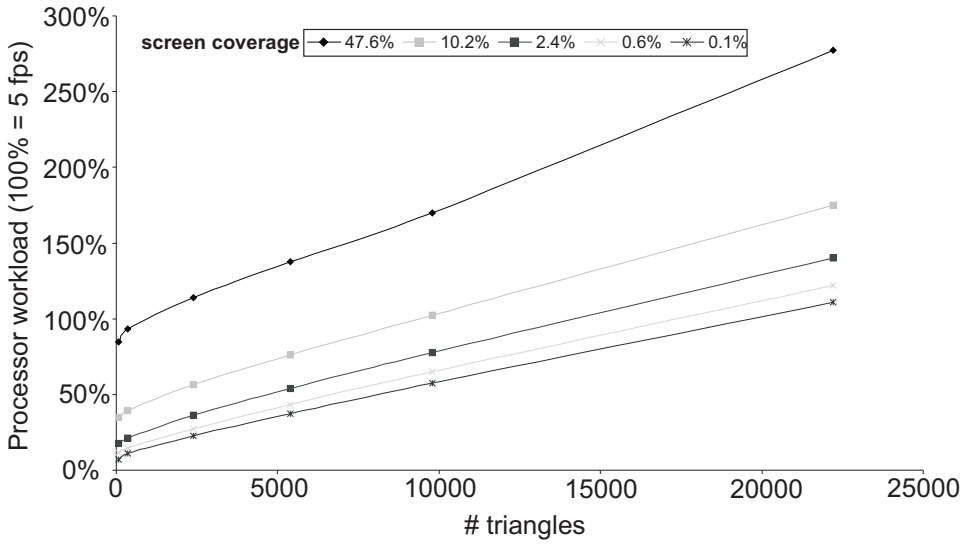


Figure 3.13: Non-linearities because of the platform architecture.

timers 3 and 4 gives very detailed information on the pixel and span processing times, but introduces too much overhead. Therefore, we have chosen to only use timers 1 and 2 that measure the vertex processing and rasterisation time. The disadvantage of this choice is that the measurements of the rasterisation must be fit onto several coefficients using a recursive least squares method, which is not always able to correctly estimate the instantaneous coefficients.

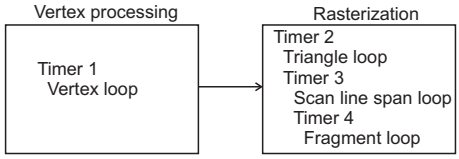


Figure 3.14: The correct introduction of timers in the code is important for the complexity.

A typical technique that is used for real-time parameter estimation is recursive least square estimation [16]. A proof for the convergence of this technique can be found in the book of Johnson [73]. The scheme can be viewed as a filter that averages the data to obtain the optimal estimates. Averaging is a good strategy if parameters of the model are constant in nature. However, the parameters of the performance model vary over time and a better technique is the recursive least square with forgetting [152]. As illustrated in Figure 3.15, this technique gradually discards old data and is only using the recent data for estimating the parameters. Campi [27] has provided a rigourous mathematical analysis of this technique including a proof of convergence.

The error on the rendering and vertex processing times (Equations 3.6 and 3.5) are measured and fit on the performance model coefficients using the values for the data-

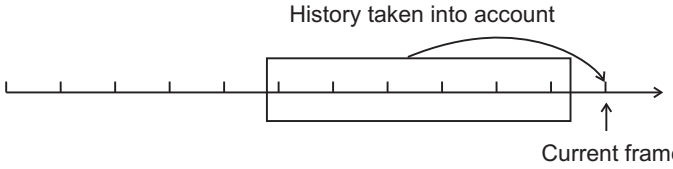


Figure 3.15: Online correction of the performance model coefficients using a recursive least square with forgetting technique.

parameters that were also used in the performance model. This assures that errors, because of wrong estimations of the data-parameters, are also taken into account in the correction.

3.2.5 Results

The software rendering pipeline is typically used on low cost mobile (and embedded) systems. This subsection discusses the results of the performance model on an embedded system, but similar results were also obtained on a PC with a superscalar processor. The validation of the software rendering performance model is given in Table 3.1 and a histogram of the error is given in Figure 3.16. Four different scenes, described in Chapter A, were used, ordered from low to high complexity. There were 3 light sources in the chess and fruit scenes and 1 light source in the Mr. Potato and city scenes. The average value of the absolute values of the errors for all the scenes is between 3.33% and 5% with a confidence level of 95%. Figure 3.17 illustrates the measurements and predictions of the execution time for different subsequent frames for the city scene of Table 3.1. Bontempi el al. [18] who have used a black box approach to derive a performance model for a texture decoder report errors between 15% and 40% for a linear model and 3.8% and 6% for a (more complex) non-linear model. Future work is still needed to validate the approach for VLIW processors, which heavily depend on the compiler for their parallelization (performance).

Scene	Mr. Potato	Chess	Fruit	City
# Objects	1	9	22	46
# Meshes	15	10	22	111
Mean error (%)	1.25	-2.88	3.12	1.4
Standard deviation (%)	9.24	3.12	5.43	11.7
Histogram	3.16(a)	3.16(b)	3.16(c)	3.16(d)

Table 3.1: The performance model error for the embedded system

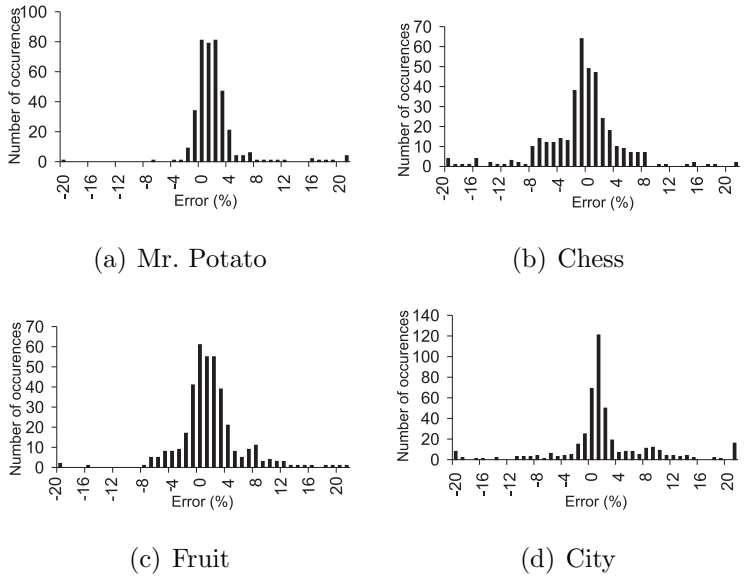


Figure 3.16: Histogram of errors of the performance model on the embedded device

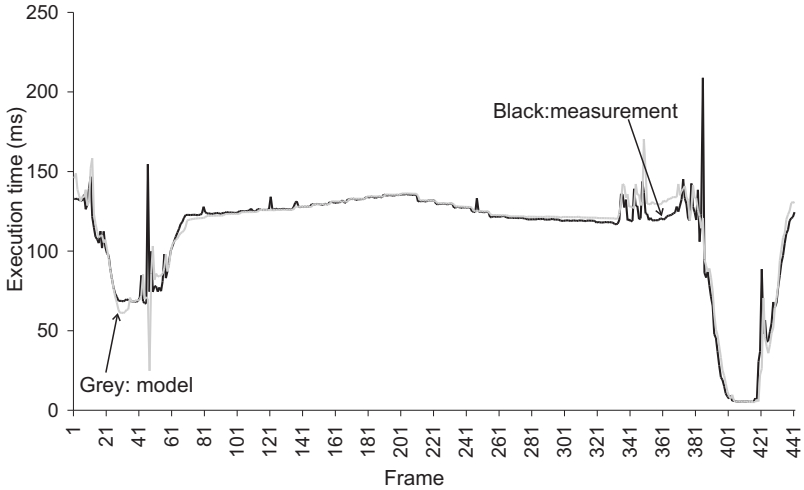


Figure 3.17: The measurements versus prediction for the animation of Figure 2.11 through the city scene.

3.3 Method for Deriving the Hardware Rendering Performance Model

The graphics pipeline, at the very highest level, can be broken into two parts: the CPU and the GPU (Graphics Processing Unit). The CPU is responsible for executing the application and sends commands and data (geometry and textures) to the GPU, which actually calculates the image. A high-level view of this architecture is given in Figure 3.18, which also shows the GPU as a pipeline of units working in parallel.

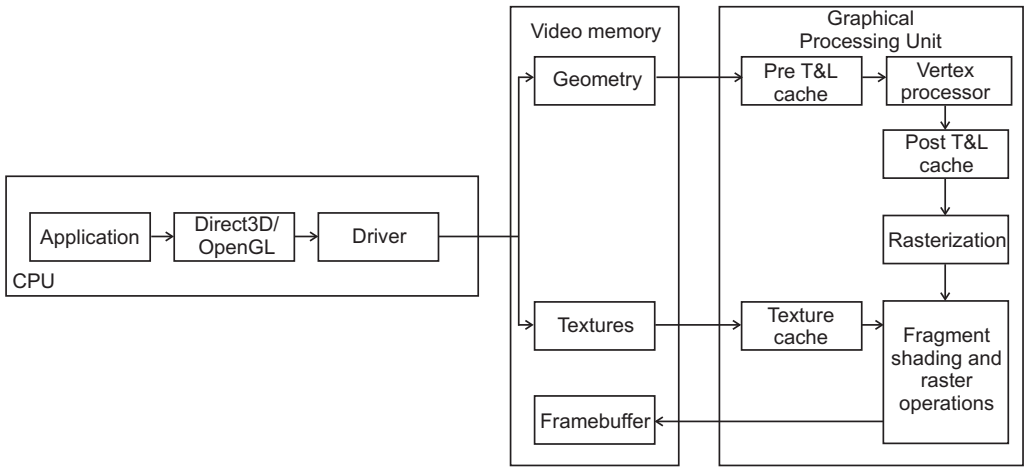


Figure 3.18: High-level overview of a hardware accelerated rendering system.

Applications typically use APIs, such as Direct3D [95] or OpenGL [134] for rendering the content. These APIs translate the high-level commands into a device-independent format that is again translated by the software driver into commands for the graphics hardware. Both the high-level API and the software driver may optimise [96] the sequence of commands that are sent to the GPU. This makes the complete system of high-level API, software driver and GPU a complicated black box, with only some high-level parameters known.

The performance modelling uses a black box [18] approach that aims to discover the potential correlations between sets of variables, on the basis of observations only. We use the divide and conquer paradigm [17], which attacks the problem by dividing the modelling problem into simpler problems and then combines the separate solutions into a global performance model. Therefore, our approach first isolates the different units in the pipeline of Figure 3.18 by deliberately creating a bottleneck in the tested unit. Subsection 3.3.1 discusses the measurement of the execution time on a GPU and the experiment that is used to determine the bottlenecks of the GPU. With the knowledge of Subsection 3.3.1, the GPU can be set in a state that the slowest chain is fixed and known.

The second step is then to derive the performance model of the different units in

the pipeline. The available high-level information is used to select the parameters to which the execution time is sensitive for the tested unit. A measurement of the execution time with different inputs is then used to derive the relationship between the parameter set and the execution time. These tests are illustrated for the Geforce FX 5900 in Subsection 3.3.2 that discusses the performance model of the vertex processor and in Subsection 3.3.3 that discusses the performance model of the rendering task. The latter includes the fragment processing and rasterisation units.

Finally, the performance models of the different units are combined into a global performance model for the complete GPU. This is discussed for the Geforce Fx 5900 in Subsection 3.3.4.

The last subsection presents the results.

3.3.1 Measuring time on the GPU

DirectX 9c (update of summer 2004) enables very specific measurements of the performance of the graphics card. However, the graphics card does not always support this type of measurements. For these cards, the rendering time can only be indirectly profiled by measuring CPU time. Because the CPU only sends commands to the GPU and then continues its execution, one must carefully synchronise the measurements on the CPU and the GPU execution. This synchronisation is done with $glFinish()$ in OpenGL and an event query in DirectX.

The system of Figure 3.18 is a pipeline of different units working in parallel. The slowest chain in this pipeline determines the execution time. Once the execution time is measured, it must thus be correlated to the correct unit in the pipeline. Figure 3.19 shows the experiment that we have used for the identification of the bottleneck. A tool (e.g. NVPerfHUD [106]) was used to validate whether the bottlenecks of Figure 3.19 were indeed located in the correct place. Three parameters were varied in this experiment:

1. The visibility of the geometry. The geometry was once put outside of the viewing frustum. This prevents the rasterisation and isolates the vertex unit from Figure 3.18;

2. The rendering state (textured / texture size) was varied.

3. The number of vertices was changed for all these different settings.

As shown in Figure 3.19, rendering a set of completely clipped triangles isolates the vertex unit and the execution time is completely linear with the number of processed vertices. With visible triangles, the execution time is linear with the number of vertices from the moment that the vertex unit becomes the bottleneck. For a lower number of vertices, the rasterisation or pixel processing unit form the bottleneck.

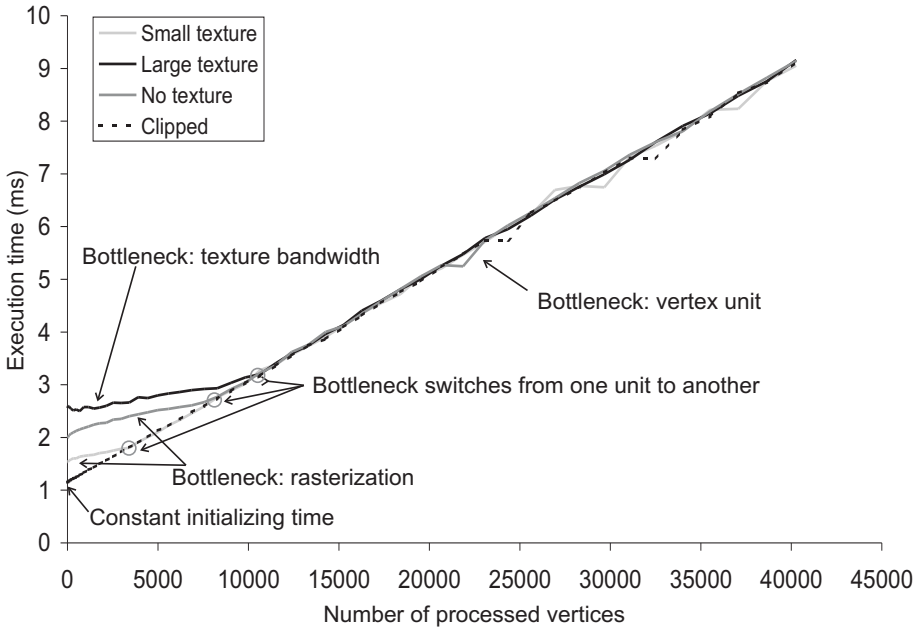


Figure 3.19: Bottleneck identification in a GPU.

The execution time of the latter units depends on the number of triangles, the number of pixels and the texture size.

3.3.2 Modelling the vertex processor

This subsection explains the performance modelling of the vertex processor, which is the first unit in the pipeline of the GPU. The performance model is derived with a black box approach that measures the execution time for varying parameters. The known high-level parameters that influence the execution time of the vertex processor are: (i) the number of light sources, (ii) the number of processed vertices and (iii) the type of the light source. The three possible light sources are:

1. Directional lights: light comes from 1 direction. The vector $\overrightarrow{s}$ from the light source to the vertex (see Equation 3.4) is then the same for all vertices and is thus computed once per drawing command.

2. Point lights: light is emitted from a point equally in all directions. The vector $\overrightarrow{s}$ then needs to be calculated for every vertex as the vector from the light source to the vertex.

3. Spot lights: light is emitted from a point in the direction given by a cone. The vector $\overrightarrow{s}$ thus needs to be calculated for every vertex as the vector from the light source to the vertex. The light intensity also depends on the direction of $\overrightarrow{s}$. If $\overrightarrow{s}$ is in the cone, no attenuation is applied, but if $\overrightarrow{s}$ is in a gray

zone between on and off, the light intensity is attenuated, which yields the attenuation factor A_i of Equation 3.4. If $\overrightarrow{s}$ is outside the cone, the intensity is easily set to zero.

Measurements were done for all types of light with all parameters of Equation 3.4 taken into account. Figure 3.20 shows the influence of different light types on the rendering time. Directional and point lights do not show different vertex processing times, but for spotlights the vertex processing significantly differs.

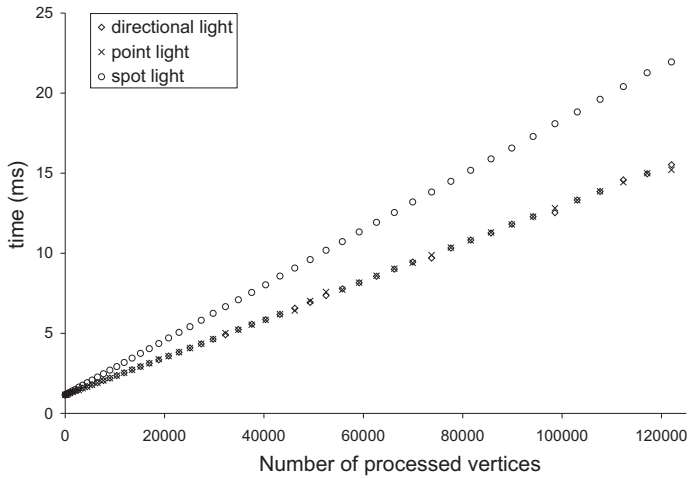


Figure 3.20: Vertex processing time for different types of light sources.

Figure 3.21 shows the influence of the number of light sources on the rendering time. For the experiment of Figure 3.21, a spot light is used, and the number of vertices was varied for different measurements with a different number L of light sources. Through a straightforward linear regression, the time per vertex for every number of light sources can be derived.

The time per vertex for the different number of light sources, i.e. the slope of the lines in Figure 3.21, is summarised in Figure 3.22 for the three different types of light. This figure shows a linear relation for a specific light type between the number of lights and the execution time for a single vertex (Equation 3.7).

$$T_{vertex} = T_v + L \cdot T_l \tag{3.7}$$

The parameter L is the number of light sources, T_l is a time per light and per vertex derived from the slope of the lines in Figure 3.22 and T_v is a time per vertex derived from the crossing of the lines with the Y-axis in Figure 3.22.

The total time needed by the vertex processor is linear (see Figure 3.19) with the number of processed vertices and is given in Equation 3.8.

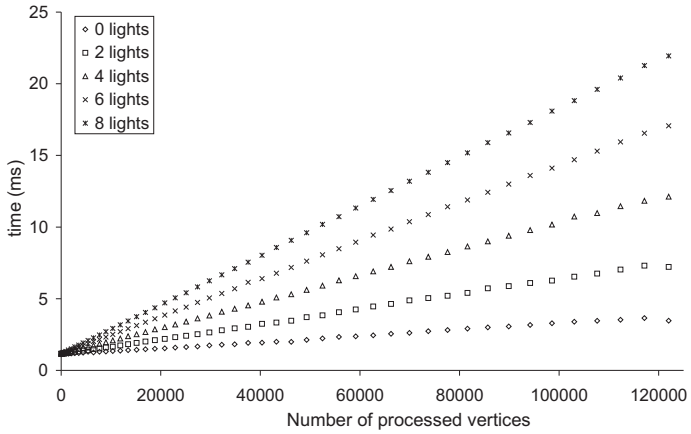


Figure 3.21: The different vertex processing time for different number of light sources.

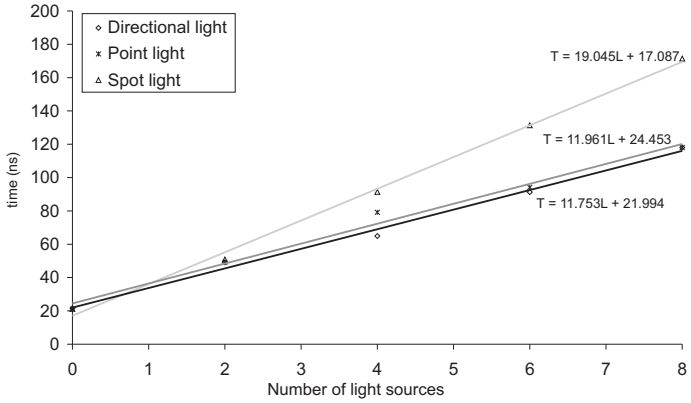


Figure 3.22: Calibration of the time per vertex and per light.

$$T = V \cdot T_{vertex} \tag{3.8}$$

The parameter V is the number of processed vertices. This number depends on the post T&L-cache that stores a small amount of transformed vertices. If for instance, in Figure 3.23, the rendering order is $T_1, T_2, \ldots, T_6$ then vertex v will be transformed when T_1 is rendered and for all other triangles $T_2, \ldots, T_6$ the result in the post T&L cache is reused. However, if first other triangles are rendered before $T_2, \ldots, T_6$, the vertices of these other triangles will overwrite the vertex v in the post T&L-cache and the vertex will be transformed a second time when triangle T_2 is rendered.

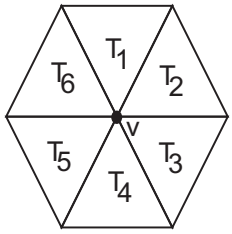


Figure 3.23: The vertex v is shared by 6 triangles and will be reused from the post T&L cache if the 6 triangles are rendered subsequently

The number of vertices that is processed by the vertex unit is given by Equation 3.9.

$$V = m \cdot 3 \cdot T \tag{3.9}$$

The parameter m is the miss rate and T the number of rendered triangles. The post-T&L cache is a cache with a small and unknown size (around 40 vertices) for which the miss rate m can be derived with the procedure shown in Figure 3.24.

```
For every triangle in the mesh
    Get vertex positions in vertex array
    For all vertices of triangle
        if vertex is in cache
            increment hits
        else
            increment misses
            Add vertex to FIFO
```

Figure 3.24: FIFO cache simulator for the cache-miss rate m

3.3.3 Modelling the rendering

This subsection discusses the black box approach for deriving the performance model of the last two stages of the pipeline shown in Figure 3.18, i.e. the rasterisation and the fragment shading and raster operations unit. The rasterisation stage generates fragments, i.e. pixel locations combined with extra data, from the triangles. The fragment-processing unit then calculates the pixel's colour. The known high-level parameters that influence the execution time are the number of triangles, the number of pixels visible at the moment that the object is rendered and the size of the texture.

Geforce products later than the Geforce 3 implement special techniques that cull invisible fragments before the rasterisation (e.g. fragments hidden by other already rendered objects). This culling does not guarantee to reject all invisible fragments before the rasterisation and does not replace the depth test at the end of the 3D graphics pipeline, but, it possibly reduces the fragment load with 30% [104]. In Figure 3.25, the execution time for rendering two objects, with one object completely occluded by the second, is shown. The texture sizes were varied so that the bottleneck was once located in the rasterisation and once in the fragment processing unit. For these two situations, the rendering order for the objects was varied: the occluded object was once rendered first and once rendered second. If the occluded object is rendered first, then the GPU does not yet know about the occluding object and draws both the occluded and occluding object. If the occluding object is drawn first, then the GPU knows about the occluding object when the occluded object is rendered and it is able to eliminate a lot of work for the occluded object. This is illustrated in Figure 3.25, which shows a clear difference in rendering time for both the rasterisation and the pixel-processing unit because of the rendering order. The rendering time thus depends on the number of pixels that are visible at the time they are rendered and not on the number of visible pixels in the final image.

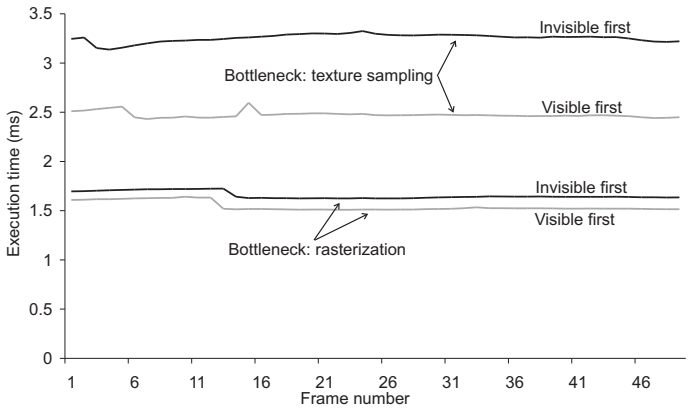


Figure 3.25: Test with two meshes: the rendering time increases when the completely invisible mesh is rendered first.

Figures 3.26 and 3.27 show an experiment that is used for deriving the performance

model of the rendering. Four different test meshes were rendered (8, 968, 3528 and 7688 triangles). The triangle numbers were chosen using Figure 3.18, so that the bottleneck was located in the rendering. For these 4 test meshes, the texture size was varied from small (Figure 3.26:64x64) to large (Figure 3.27:2048x1024) and for all textures sizes, the texture filter was varied.

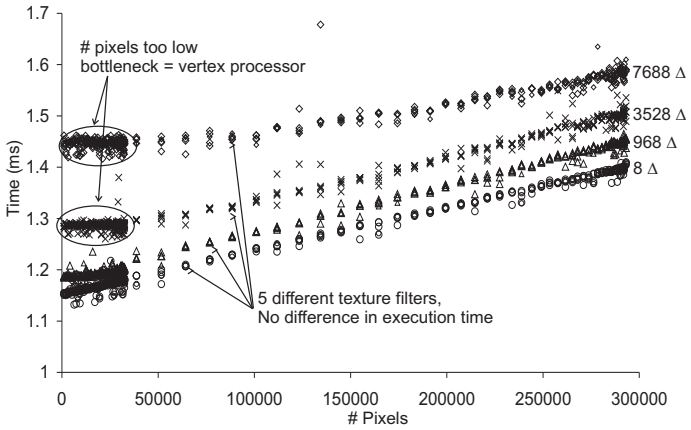


Figure 3.26: Pixel processing and rasterisation time for a small texture (64x64) and 8, 698, 3528 and 7688 triangles.

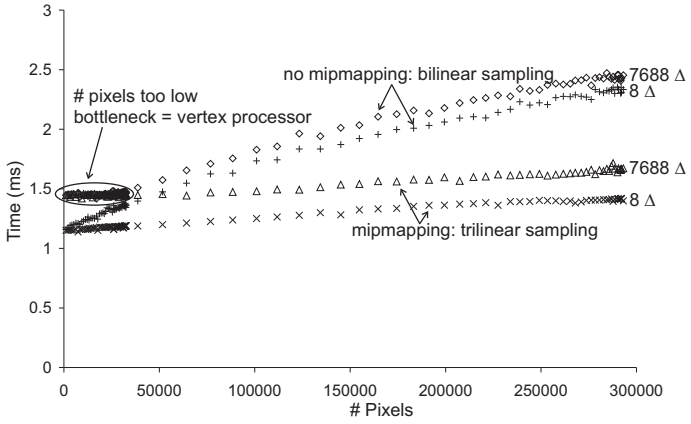


Figure 3.27: Pixel processing and rasterisation time for a large texture (2048x1024) and 8 and 7688 triangles..

For a small texture (Figure 3.26), the execution time depends on the number of triangles in the test mesh and on the number of pixels. For a small number of pixels in Figure 3.26, the execution time becomes independent of the number of pixels because the execution time of the pixel and rasterisation units is too small and the vertex unit becomes the bottleneck. For a large texture size (Figure 3.27), the situation is different. Without mipmapping, the bottleneck completely shifts to the texture unit because of the texture sampling. The execution time still depends on the number of triangles, but its dependency on the number of pixels is much larger.

With mipmapping enabled, the system chooses the mipmap level that closely fits the screen size of the mesh, the rendered texture size decreases and the texture cache efficiency increases. Because mipmapping gives a better quality and it saves time, mipmapping must always be enabled. The performance model of the rendering can be derived with the Figures 3.26 and 3.27.

$$T = F \cdot T_f + P_{visible} \cdot T_p \tag{3.10}$$

With F the number of triangles; $P_{visible}$ the number of visible fragments at the time of rendering and T_{Face} and T_{pixel} respectively the time per face and the time per pixel. The model does not have a texture size parameter because we assume that mipmapping is enabled.

3.3.4 Performance model

The previous subsections have derived a performance model for the individual units in the graphics pipeline. This subsection presents the last step in the modelling: combining the separate models into a performance model of the complete GPU.

The vertex and rendering units only work completely in parallel for one specific triangle size, which depends on the complexity of the vertex processing and triangle/pixel processing. This optimal size p_0 is given with Equation 3.11.

$$\begin{array}{lcl} \text{Vertex processing time} & = & \text{rendering time} \\ & \Updownarrow & \\ 3 \cdot m \cdot (T_v + L \cdot T_l) & = & T_f + p_0 \cdot T_p \\ & \Updownarrow & \\ p_0 & = & \frac{3 \cdot m \cdot (T_v + L \cdot T_l) - T_f}{T_p} \end{array} \tag{3.11}$$

For other triangle sizes, the overall performance of the pipeline is determined by the slowest chain in the pipeline: i.e. the rendering for triangles with size larger than p_0 and the geometry unit for triangles with a size smaller than p_0. This is summarised in Equation 3.12 for a single triangle.

$$T = \max(3 \cdot m \cdot (T_v + L \cdot T_l), T_f + p \cdot T_p) \tag{3.12}$$

The execution time for rendering a collection of triangles is given with Equation 3.13, which sums the execution time of the triangles with the bottleneck in the vertex processing with the execution time of the triangles with the bottleneck in the rendering.

$$T = F_{(p<p_0)} \cdot 3 \cdot m \cdot (T_v + L \cdot T_l) + F_{(p>p_0)} \cdot T_f + p_{(p>p_0)} \cdot T_p \tag{3.13}$$

Unfortunately, the parameters $F_{(p<p_0)}$ and $p_{(p>p_0)}$ depend on many factors and are not easily derived. Therefore, we can rewrite Equation 3.13 into the model of Wimmer and Wonka [162], presented in Equation 3.14.

$$T = T_{init} + V \cdot T_{vf} + p \cdot T_{fp} \tag{3.14}$$

The parameter T_{init} is the initialisation time that is independent of the data-parameters (e.g. time needed to change the rendering state). Wimmer and Wonka have measured the execution time for many objects and then fitted this equation onto the results. The real numbers of vertices and pixels, with their bottleneck respectively in the vertex processor and rendering, however deviate a lot from this average value. Therefore, we propose to make this model adaptive. A recursive least square with forgetting technique is used that measures the execution time at run-time and corrects the model to the instantaneous conditions.

A second possibility to simplify Equation 3.13 is to ignore the shifting bottleneck and to use the model proposed by Funkhouser et al. (Equation 3.15). This model is also corrected with an online correction system.

$$T = T_{init} + \max(F \cdot T_{vf}, p \cdot T_{fp}) \tag{3.15}$$

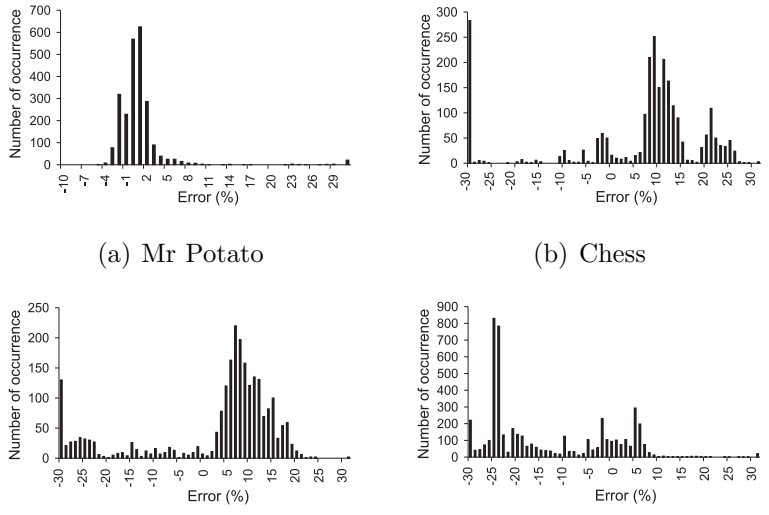
Note that this subsection proposes a rough performance model that is once calculated for the complete scene. The coefficients of the models are average execution times, while the actual vertex and pixel processing times depend on the rendering states of the objects. An alternative is to calculate the models per group of triangles with the same rendering state and model coefficients and to sum the results for modelling the execution time of the complete scene. These models are still rough performance models with errors between 10% and 20% [141] and also need an online correction system to correct them. The correction system however needs to measure the execution time for every group of triangles with the same rendering state. Because of the synchronisation that is needed for these measurements, such a fine grained measurement of the execution time introduces a large overhead (50% for the city scene of Subsection 3.3.5). We prefer to measure the execution time once per frame and to use the performance models proposed in this subsection.

3.3.5 Results

The validation for the PC (GPU) performance model of Equation 3.15 is given in Table 3.2 and the histograms of the error (without applying a feedback model) of the different scenes are given in Figure 3.28. The same scenes as for the embedded system (described in Chapter A), but at a higher quality, were used for testing the performance model.

Scene	Mr. Potato	Chess	Fruit	City
# Objects	1	9	22	46
# Meshes	15	10	22	111
Mean error (%)	0.58	3.64	2.28	-14.44
Standard deviation (%)	5.87	18.39	14.33	13.49

Table 3.2: Result for performance modelling on the PC system with Equation 3.15 (max)

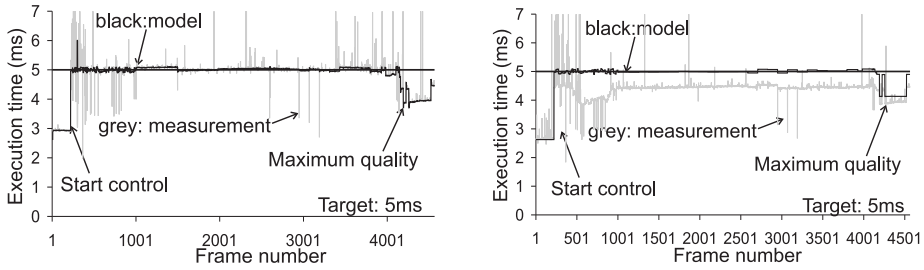


(a) Mr Potato (b) Chess

(c) Fruit (d) City

Figure 3.28: Histogram of uncorrected errors for the performance model of Equation 3.15 (max) on the PC

The difference for the performance model with and without the online correction is shown in Figure 3.29 that shows the controlled execution time (target 5 ms) for a navigation through the city scene. The online correction indeed improves the results of the performance model significantly with an average of the absolute values of the error between 1.01% and 1.35% with a confidence level of 95%.



(a) With online correction (b) Without online correction

Figure 3.29: The result for the performance model with Equation 3.15 (max) on the city scene: a) with online correction and b) without online correction.

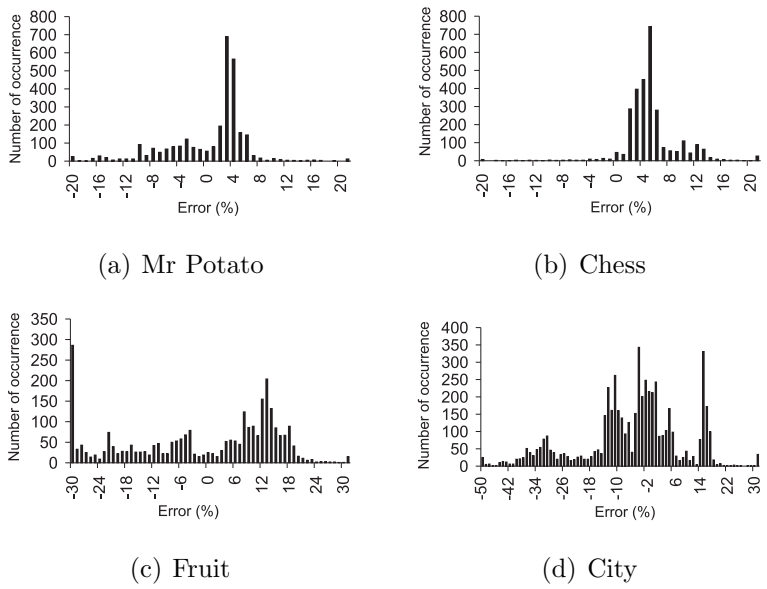
The validation for the PC (GPU) performance model (including the online correction) of Equation 3.14 is given in Table 3.3 and the histograms for the error of the different scenes are given in Figure 3.30. The average for the absolute values of the errors is for all scenes between 9.8% and 10.2% with a confidence level of 95% (including the online correction).

Scene	Mr. Potato	Chess	Fruit	City
# Objects	1	9	22	46
# Meshes	15	10	22	111
Mean error (%)	0.053	4.5	-2.88	-6.27
Standard deviation (%)	9.09	10.15	19.15	15.24

Table 3.3: Result for the performance modelling with Equation 3.14 on the PC system with the online correction

Although both proposed models give more or less the same results when the online correction is not applied, the model of Equation 3.15 (maximum function) matches more closely the reality than the model of Equation 3.14 (summation). Therefore, the online correction is much more accurate on the performance model of Equation 3.15 than for the model of Equation 3.14.

Wimmer and Wonka [162] report average errors for a complete scene rendered on a Geforce 3 product up to 30% for Equation 3.15 (maximum) and 20% for Equation 3.14 (sum), which is more or less the same as the errors on our uncorrected performance models. The online correction system however heavily improves the results of the performance models. Both presented models are however more vulnerable to



(a) Mr Potato (b) Chess

(c) Fruit (d) City

Figure 3.30: Histogram of errors of the performance model with Equation 3.14 on the PC (sum)

larger errors than the software model for the embedded system. This is mainly due to the fact that the graphics system (DirectX runtime, driver and chip) is a black box for which a lot of parameters are unsure: run-time and driver optimisations, exact Post T&L cache size, etc. [96]. The uncertainty on the coefficients of the performance model and even on the performance model itself is thus much larger as for the application on the embedded system. This situation can be improved a lot if query mechanisms as provided in DirectX 9.0c are supported by the hardware. This would enable a correct querying and update of the performance model coefficients.

3.4 Conclusions

This chapter relates the platform independent cost metric used in the optimisation to the real execution time for two different types of platforms: an embedded device using software rendering on an XScale processor and a PC using a Geforce chip for hardware accelerated rendering.

First, a method was proposed to derive the performance model of the software renderer running on the embedded device. The main issue was to find the balance between the extra accuracy obtained by adding a priori unknown parameters and the extra complexity because of their inclusion in the model.

Second, a black box approach was used to derive the performance model of the hardware renderer (GPU) used on the PC. The main issue for this model was to isolate

a unit of the GPU to be able to measure its performance. Two different approaches for combining the execution times of the different units into a global execution time model were evaluated. Initially, both approaches gave a similar accuracy. However, the accuracy of the performance model using the maximum function is significantly better after using a feedback system than for the summation function.

Chapter 4

Modelling Visual Quality

The goal of a 3D graphics application is to provide the best possible image quality within the performance constraints of the used terminal. The assessment of "best possible" is in an interactive application typically done with a heuristic that estimates the visual quality of the *rendered image*. Because of the interactivity, a prerequisite for the heuristic to be usable is a *fast* online evaluation. The challenge is thus to find the best possible quality metric for the lowest online complexity.

Unfortunately, many prior art techniques sacrifice the accuracy and realism of the model for simplicity in the online problem solving. On the other hand, realistic error metrics exist to measure the fidelity of the geometric model. These metrics are typically used to compare different mesh simplification techniques, but they neglect the user's viewpoint and are typically too complex to be used directly in an interactive application. Therefore, we propose to use a mixture of off-line preprocessing steps for computing the complex parts of the quality model and a simple online combination step for evaluating the visual quality with respect to the user's viewpoint.

A first section gives a short overview of the parameters that influence the quality in 3D graphics applications. Section 4.2 discusses several heuristics from previous work. Section 4.3 introduces the proposed view-dependent quality heuristic and finally the chapter ends with the conclusions in Section 4.5.

4.1 Overview

The visual quality of 3D rendered images depends on many factors, but this text focuses on the parameters for interactive polygonal rendering. Other quality aspects that are related to other rendering techniques or application stages are beyond the scope of this text.

The 3D rendering pipeline computes the image from an input model of the 3D scene, which is normally defined as a combination of geometry and appearance. The goal

of this section is to give a short overview of the influence of this input model on the visual quality. Subsections 4.1.1 and 4.1.2 respectively describe the influence of the geometric structure and its position in the scene. Subsection 4.1.3 discusses the influence of the illumination and shading models and Subsection 4.1.4 ends with the influence of the texture on the visual quality.

4.1.1 The influence of the geometric model

A popular way for modelling the geometric structure is to tile the surface of the 3D object with a mesh of triangles. Triangular meshes are only linear approximations of arbitrarily complex surfaces, which leads to approximation errors in the geometric model (and a low visual quality) if the number of triangles is too low. Figure 4.1 for instance illustrates how a sphere becomes unrecognisable if its number of triangles is decreased too much. Section 4.3.1 discusses how the quality model includes the geometric accuracy.

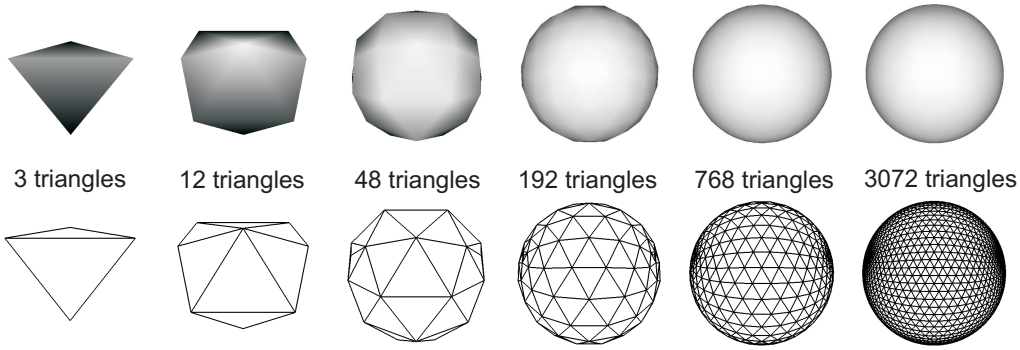


Figure 4.1: Decreasing the number of triangles decreases the visual quality of the sphere.

In addition to the accuracy of the geometric model, the visual quality also depends on the change of the geometric model over time. Rendering an object with different resolutions in subsequent frames may be bothersome to the user and reduces the quality of the image sequence. The model of the visual quality includes this parameter with a scaling factor that has been derived from the work of Funkhouser and Séquin [52] (Section 4.3.2).

4.1.2 View-dependent quality parameters

The position of the object in the virtual world as seen from the user viewpoint is also an important parameter for the visual quality. Errors on objects far away are much less visible than errors on nearby objects. Figure 4.2 for instance shows that it is harder to see the differences on the small Stanford Bunnies [151] than on the

large ones. Section 4.3.1 discusses how an existing geometric error metric is adapted to take this parameter into account.

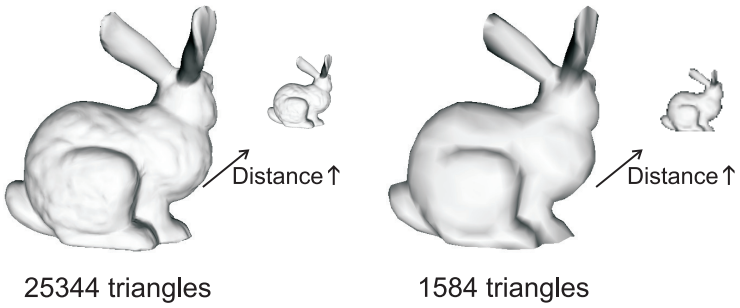


Figure 4.2: Influence of the distance to the viewpoint on the visual quality: the differences in the left and right bunnies are better visible for the large than for the small bunnies.

The silhouette of a 3D object, which also depends on the viewpoint, is also recognised as being important for the visual quality; therefore, special techniques have been developed that locally increase the number of triangles on the silhouette (Figure 4.3).

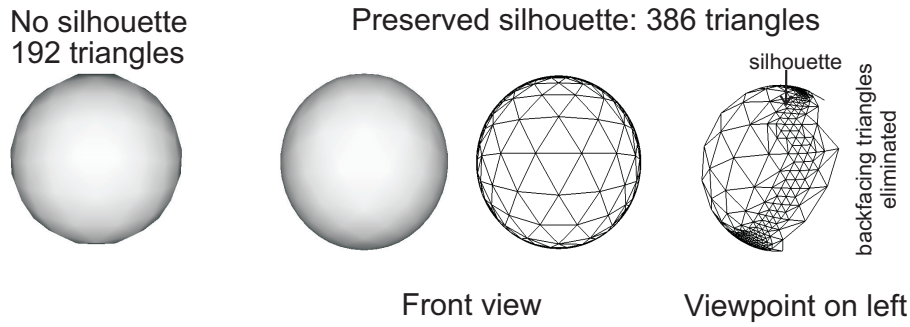


Figure 4.3: A local increase of number of triangles to preserve the silhouette smoothes the sphere and increases the visual quality.

The silhouette changes with the viewpoint and an update of the geometric structure is needed for these techniques every time the viewpoint changes. The gain of this extra overhead is the increased visual quality for a lower number of triangles. This gain is however on many systems insufficient for compensating the extra CPU overhead and communication between the CPU and the graphics accelerator. More information on this issue is given in Chapter 5. In addition to the extra overhead, silhouette preservation increases the number of geometry changes over successive frames. As discussed in Section 4.1.1, these changes can be bothersome to the user. Because of these two issues, silhouette preservation will not be considered in our model of the visual quality.

The last two factors that depend on the user's viewpoint are the motion of the objects and the user's focus. Objects that move around in the scene or objects that appear in the portion of the screen that is not in the focus of the user are not clearly seen by the user and are thus less important for the visual quality than other objects. The model implements these parameters with scaling factors that are derived from the work of Gobbetti and Bouvier [56] (see Subsection 4.3.2).

4.1.3 Influence of the shading and illumination models

Following the definitions of Foley et al. [46], an illumination model calculates a colour at a given pixel. The illumination model is ideally applied at each visible pixel, but unfortunately this technique is often too expensive. Therefore, a shading model determines when the illumination model is applied: per pixel (Phong), per vertex (Gouraud) or per triangle (flat). It should be clear that the choice of the shading model heavily influences the visual quality.

A popular shading model that provides good visual results is the Gouraud shading model. It uses a standard or user defined illumination model [46] for calculating the colours on the screen positions of the vertices. These colours are then interpolated to provide the colour of the remaining triangle pixels. Because the colour is only calculated on the vertices, the correctness of this shading model heavily depends on the number of triangles of the geometry. Figure 4.4 for instance illustrates how the spot in the middle of the sphere gradually improves by increasing the number of triangles. Therefore, Xia et al [165] have developed a mesh simplification technique that locally improves the geometric accuracy to improve the quality of the light spots. However, as with silhouette preservation, this technique is also too expensive and will not be considered in this work. Other techniques (e.g. light maps) also exist for getting the spots in Figure 4.4 right. These techniques are however beyond the scope of this text.

A second shading model is the flat or constant shading model. This model calculates one colour for every triangle using a standard or user defined illumination model. The price for the reduced complexity is a "faceted" appearance (Figure 4.5).

Next to the shading model, also the illumination model influences the visual quality. A bright surface for instance typically reveals much more artifacts than a dark surface [160]. Subsection 4.3.2 describes how this effect is included in our quality model.

4.1.4 The influence of the texture

Realism is in interactive 3D graphics often obtained by using textures as a kind of wall paper on the geometric structure. Figure 4.6 for instance illustrates how a sphere is turned into a model of the planet Jupiter by using texture mapping.

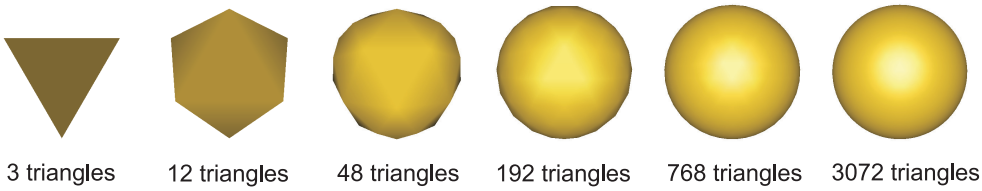


Figure 4.4: Influence of the material on the visual quality with Gouraud shading.



Figure 4.5: Influence of the material on the visual quality with flat shaded rendering.

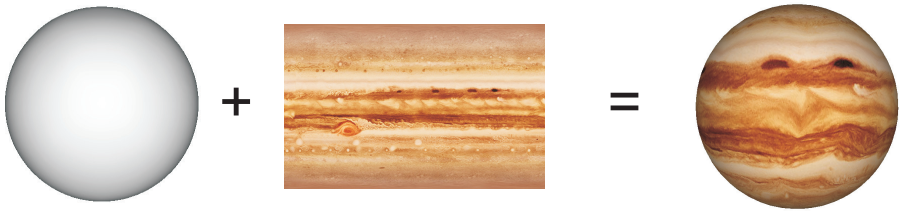


Figure 4.6: Turning a simple sphere with texture mapping into an image of the planet Jupiter.

Every pixel that is occupied by the object reads a colour value at a location in the texture that is specified by the texture coordinates attached to the object. The simplification of the geometric structure thus affects the texture mapping by the removal of texture coordinates. This results in a distortion of the rendered image because of incorrect texture mapping (Figure 4.7). To minimise this effect, several techniques (e.g. Sander et al. [130]) have been developed to take the textures coordinates into account in the mesh simplification process.

For years, graphics researchers have observed that textures can hide artifacts in the images that are caused by e.g. an inaccurate geometric structure. This effect is called texture masking and is in the most general form defined as the situation in which the presence of one visual pattern changes the visibility of another [44]. Figure 4.8 for instance illustrates the texture masking on an identical sphere with different textures. The presence of the earth texture hides the faceting artifacts of the sphere (introduced by a flat shading model [46]) that still appear on the sphere with the Jupiter texture. Ferwerda et al. [45] have presented a model for texture masking.

Another issue with textured rendering is the choice of the correct texture filter and resolution. Figure 4.7 shows the influence of these parameters on the visual quality. Decreasing the texture resolution (which also occurs by zooming in on the 3D object) introduces this blocky image. Applying a bilinear texture filter cannot restore the visual details but it removes the sharp edges and improves the visual quality. Similarly, mipmapping removes aliasing effects because of mismatches in the triangle size on the screen and in the sampled texture [80].

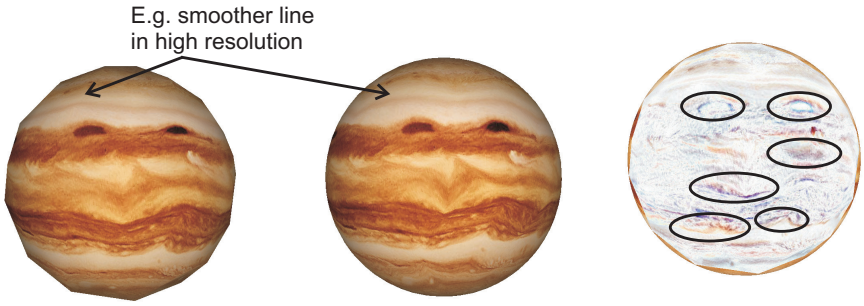
Section 4.3.2 discusses some experiments for deriving the parameters that include the texture in the model of the visual quality. This is however still an unsolved problem and is topic of future research.

4.2 Available quality metrics

The goal of the quality metric is to estimate the impact of changing the geometric structure on the visual quality with a minimum complexity. A first class of quality metrics, described in Subsection 4.2.1, predicts visual degradation via simple analytic functions. A second class of quality metrics, described in Subsection 4.2.2, tries to summarise all the contributing factors in one number by measuring the quality on the output image in off-line preprocessing steps.

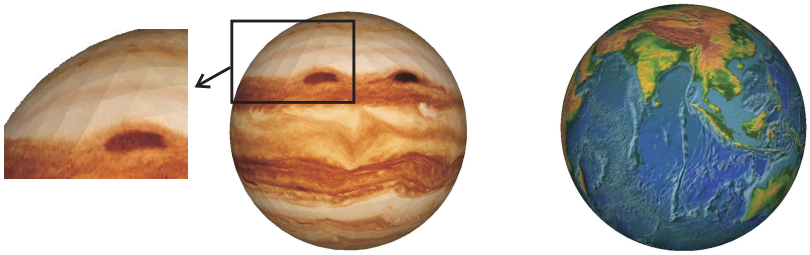
4.2.1 Analytic benefit models

Funkhouser and Séquin [52] were among the first to propose a technique and a quality heuristic (called benefit heuristic) for optimising discrete levels of detail. Their benefit heuristic depends primarily on the size of an object in the final image,



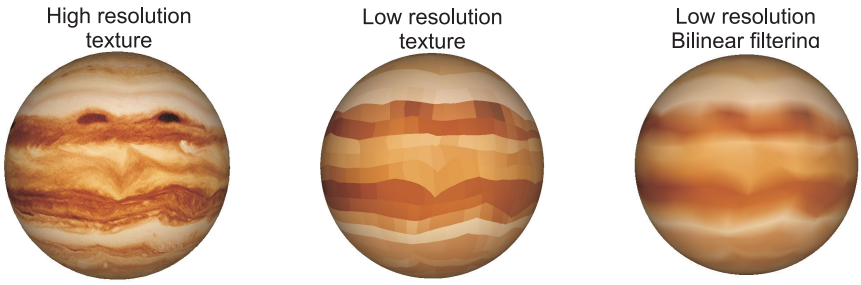
(a) Low versus High resolution (b) Difference

Figure 4.7: Mesh simplification causes artifacts in the texture mapping by the removal of texture coordinates, a) A low resolution mesh compared with a high resolution mesh; b) differences in texture patterns between the high and low resolution, examples marked with ovals.



(a) Faceting visible (b) Faceting invisible

Figure 4.8: The capability of texture masking to hide artifacts (introduced by a flat shading model) depends on the texture: a) The artifacts are visible for the Jupiter texture; b) The artifacts are hidden for the Earth texture.



(a) Reference (b) Blocky image (c) No sharp edges

Figure 4.9: Decreasing the texture resolution distorts the images (a and b) and using the appropriate texture filter (c) improves the quality.

but it also depends on the accuracy of the object O at Level Of Detail L and rendering algorithm (shading model) R. The $Accuracy(O, L, R)$ is defined with:

$$Accuracy(O, L, R) = 1 - \frac{BaseError}{Samples(L, R)^m} \tag{4.1}$$

where Samples(L,R) is the number of vertices of the object at Level Of Detail L for Gouraud shading or the number of polygons at Level Of Detail L for flat shading; m is an exponent depending on the shading model (flat = 1, Gouraud = 2). The BaseError is arbitrarily set to 0.5 to give a strong error for a curved surface represented by a single flat polygon.

In addition to the accuracy and the size of an object, their benefit heuristic also depends on other factors:

- $Importance(O)$: Some types of objects may have inherent importance and must be assigned larger rendering time budgets. E.g. walls may be more important than pencils in a building walk through.
- $Focus(O)$: Objects that appear in the portion of the scene to which the user is looking are more important (and thus deserve larger time budgets) than objects in the periphery of the user's view.
- $Motion(O)$: The user may not be able to clearly see moving objects (and more important their artifacts). The benefit must thus decrease relative to the speed of the object.
- $Hysteresis(O)$: Rendering an object at different levels of detail over successive frames may be bothersome and may reduce the quality of an image sequence.

These factors are represented by a number between 0 and 1 reflecting a possible reduction in the object's benefit. The overall benefit heuristic is the product of all aforementioned factors:

$$Benefit(O, L, R) = \begin{cases} Size(O) \cdot Accuracy(O, L, R) \cdot \\ Importance(O) \cdot Focus(O) \cdot Motion(O) \cdot \\ Hysteresis(O, L, R) \end{cases} \tag{4.2}$$

Equation 4.2 uses a product to combine the factors into a single metric because one parameter can completely undo the effect of all the other parameters. E.g., a very small object shown at full resolution has a very low benefit for the scene. Gobbetti and Bouvier [56] use a similar benefit model that multiplies different factors. The size and the accuracy of the object are also the core parameters in their heuristic. However, they have defined the accuracy model with the convex function given in

Equation 4.3, which has enabled to use an interior point algorithm for solving the (convex) optimisation.

$$Accuracy(M, r) = \sqrt{N_v^{max} \cdot r} \tag{4.3}$$

where $Accuracy(M, r)$ is the accuracy of the geometry M at resolution (i.e. percentage of shown vertices) r with N_v^{max} vertices at the highest Level Of Detail. Gobbetti and Bouvier also use scaling factors to take the focus and semantics of the objects into account.

A caveat in these benefit models is the equal approximation of the quality of an object that respectively needs many or few triangles. These benefit models will for instance estimate an equal visual quality for a tetrahedron approximating a sphere and a tetrahedron for which it is the intention to show a tetrahedron. This is obviously a wrong estimate. Therefore, Wimmer and Schmallstieg [161] have proposed to use an accuracy heuristic (Equation 4.4) that depends on the relative triangle count.

$$Accuracy(M, r) = \frac{{p_i}^{\beta}}{x_i} \tag{4.4}$$

where p_i is the number of available polygons, x_i is the number of rendered polygons and β is a constant value shaping this function. Although this proposal is a nice improvement, it still estimates the same visual quality for objects with the same relative triangle count. The visual quality is however very dependent on the used content and the viewing conditions, as mentioned in the previous section 4.1. A group of triangles can for instance hide a large amount of triangles of the same object.

4.2.2 Benefit models based on the output image

Although the goal of a 3D application is to provide the best possible image, most methods measure quality indirectly through geometry fidelity, scaled with some extra context parameters. Maciel and Shirley [90] have proposed to measure the quality of the output image in a preprocessing step for all possible object resolutions, representations and positions. This approach is also followed by Pham Ngoc et al. [101]. These proposals however use simple image quality metrics (e.g. MSE or PSNR [90, 101, 85]), which measure the real difference between the reference and rendered image. This difference does not necessarily correspond to the real quality that is perceived by the user. Texture shifting for instance results in large differences in MSE or PSNR, but this shifting is not always important for the user if he/she does not know the reference image.

Objective image quality metrics that perform better than the MSE or PSNR metrics are typically based on models of human vision, for which Ferwerda [44] describes the fundamental findings. Two examples are the Sarnoff [87] and Daly models [37] that were compared in a study of Li [84]. Watson et al. [157] has followed a different approach. They rate images by the results of some task given to a user. Many issues however remain. It is for instance still unclear how the image quality of two different objects should be combined into a quality metric for the objects together.

A disadvantage of this approach is the loss of flexibility. The same geometric structure can be used multiple times in the same scene with different appearance models. This geometric structure can also be animated so that the position of the object parts changes over time. This approach then needs to sample the visual quality for this object for all different appearance models and positions. This results in very large databases containing all the quality information of all the objects in the scene for all possible conditions. This motivated our choice to use the error for the input data that is changed by our framework: the geometry. The geometry is however not the only input parameter; therefore, we have followed the approach of previous work to use scaling factors that adapt the core error metric to the changing conditions.

4.3 The proposed quality metric

Our quality model tries to combine the advantages of both approaches proposed in Section 4.2, while avoiding their disadvantages as much as possible: it combines the flexibility and simplicity of the analytical function with the realism of measuring the quality in preprocessing steps. A first subsection introduces the existing geometric error metric and our extensions to be closer to the user-centric subjective experience of quality (in accordance to the observations of Section 4.1). A second subsection discusses the extra scaling factors that are introduced in the error metric for taking a limited effect of the material into account.

4.3.1 The quality of the geometric approximation

A popular error metric for measuring geometry preservation of simplified meshes is the surface to surface error metric based on the L^p norms. Equation 4.5 defines this error metric:

$$\begin{aligned} L^p(S_1, S_2) &= max(l^p(S_1, S_2), l^p(S_2, S_1)) \\ l^p(S_1, S_2) &= \sqrt[p]{\frac{1}{A(S_1)} \cdot \int\limits_{a \epsilon S_1} d(a, S_2)^p \, dS_1} \end{aligned} \tag{4.5}$$

where S_1 and S_2 are the surfaces to compare, $d(a, S_2)$ is the Euclidean distance between point a of surface S_1 and the closest point q on surface S_2. $A(S_1)$ is a

function returning the area of the surface S_1. The lower this approximation error is, the better the visual quality. This is illustrated in Figure 4.10 that shows an increasing image quality, which corresponds very well with the decreasing root mean square ($p = 2$) and mean ($p = 1$) surface to surface error metrics. The image quality has been measured with two independent image quality metrics: the PSNR and SSIM index (Structural SIMilarity index) [156] [1]. Note that in Figure 4.10 the Hausdorff distance ($p = \infty$) saturates very fast, although the visual quality further improves. This is due to the fact that the Haussdorf distance measures the maximum error, which is a very local phenomenon. The Hausdorff distance is thus not suited very well for our purposes; therefore, we currently use the root mean square error metric in our system.

The METRO [33] and M.E.S.H. [5] tools implement the surface to surface errors by means of uniform sampling and numerical integration. The MeshDev [123] tool also takes deviations into account of attributes such as texture coordinates. M.E.S.H. [5] implements these error metrics by calculating the error per triangle and then, as shown in Equation 4.6, averages the results for all triangles weighted by their respective area.

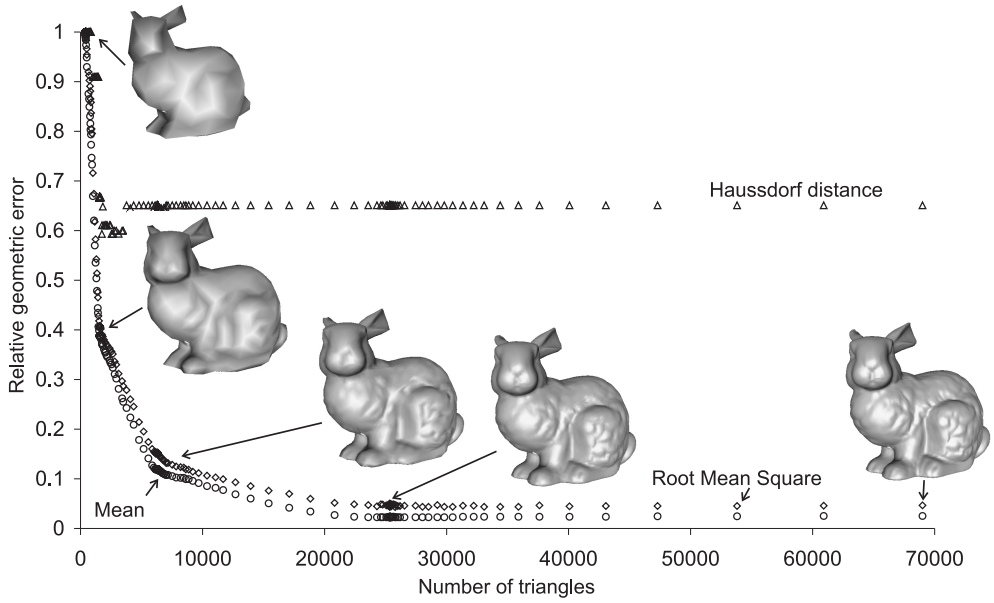
$$l^p(S_1, S_2) = \sqrt[p]{\frac{\sum_{i=0}^{F} A(T_i) \cdot (l^p(T_i, S_2))}{\sum_{i=0}^{F} A(T_i)}} \tag{4.6}$$

where F is the number of triangles in the surface and T_i is triangle i of the surface. The error metric of Equation 4.6 is typically calculated over the complete mesh in the object's coordinate space. This coordinate space is however transformed into the coordinate space of the camera with the modelview transformation that defines the position and orientation of the object in the scene. The surface parts (and errors) that are hidden for the user are thus known after the modelview transformation. Invisible errors are eliminated from Equations 4.5 and 4.6 by doing the integration/summation in the camera coordinate space over the visible surface parts instead of the complete surface (Equation 4.7).

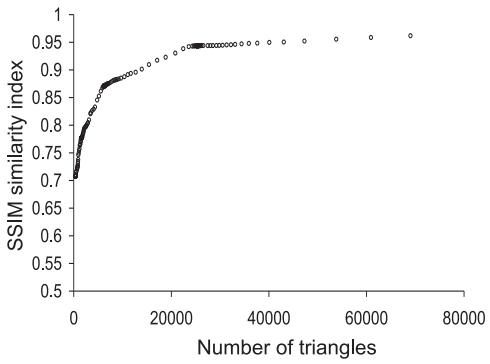
$$l^p(S_{v1}, S_{v2}) = \sqrt[p]{\frac{1}{A_v(S_{v1})} \cdot \int\limits_{a \epsilon S_{v1}} d(a, S_{v2})^p \, dS_{v1}} \tag{4.7}$$

where S_{v1} and S_{v2} are the visible surface parts of S_1 and S_2 after the modelview transformation and A_v is the visible area of S_{v1} in the camera coordinate space. Figure 4.11 shows the difference in the quality metric between a full object and an

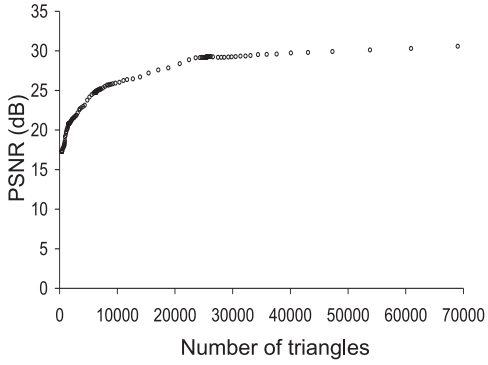
[1] Under the assumption that human visual perception is highly adapted for extracting structural information from a scene, the SSIM index is an alternative quality assessment for the PSNR that measures the change of structural information in an image. A mathematical definition of the SSIM index can be found in the work of Wang et al. [156].



(a) geometric error



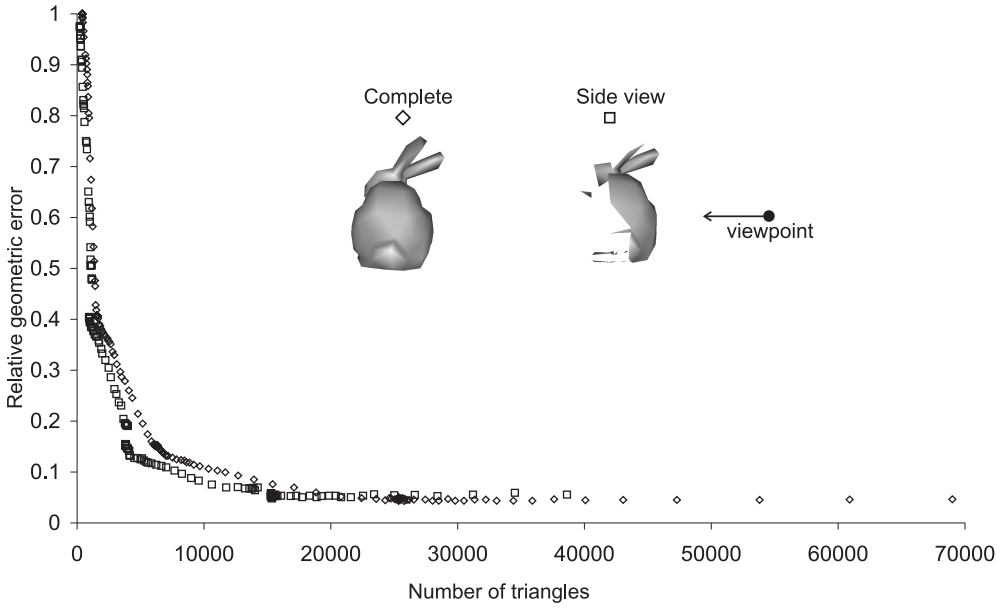
(b) SSIM index



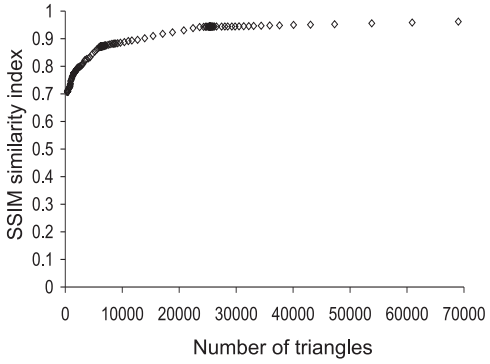
(c) PSNR

Figure 4.10: The decreasing mean and root mean square surface to surface *error* metrics (a) correspond very well with the increasing SSIM (b) and PSNR (c) image *quality* metrics for the Stanford bunny. The Haussdorf distance is not suited for our needs.

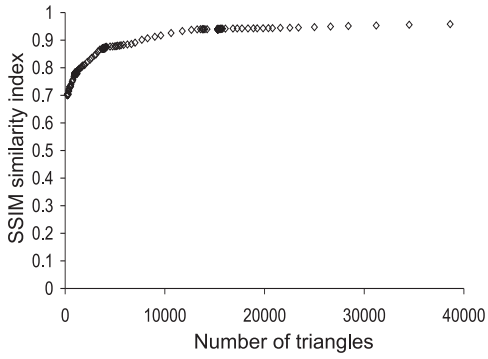
object with the invisible triangles removed. The viewpoint for removing triangles is set in the front of the object, but for showing the removed triangles, the object is in Figure 4.11 shown from its side. More or less 40% of the triangles are removed and the geometric error metric is slightly different for both setups. The SSIM similarity index that measures the quality of the output image however shows the same quality values for both setups. The PSNR metric is not shown in Figure 4.11, but shows the same results as the SSIM quality metric.



(a) Geometric error scaled with invisible triangles eliminated



(b) View-independent



(c) Invisible triangles removed

Figure 4.11: An illustration of the error metric in the camera coordinate space: the error metric (a) shows some differences, while the SSIM index is the same for both the view-independent (b) and view-dependent (c) setups.

The projection of the object on the screen follows the modelview transformation and

determines the size of the object on the screen. Mathematically, a point a projects onto the screen according to Equation 4.8

$$\begin{cases} x_a^* = \frac{w_v}{w_f} \cdot a_x \cdot \frac{N}{a_z} \\ \\ y_a^* = \frac{h_v}{h_f} \cdot a_y \cdot \frac{N}{a_z} \end{cases} \tag{4.8}$$

where N is the distance from the screen to the viewpoint, w_v and w_f are the width of respectively the viewport and frustum and h_v and h_f are the height of respectively the viewport and frustum. The factor $d(a, S_{v2})$ of Equation 4.7 that measures the Euclidean distance (projected on the 2D screen of the camera) between a point a on surface S_1 and the closest point q on surface S_2 then becomes:

$$\begin{array}{l} d(a, S_{v2})^* = \frac{w_v}{w_f} \cdot N \cdot \sqrt{\left(\frac{q_x}{q_z} - \frac{a_x}{a_z}\right)^2 + \left(\frac{q_y}{q_z} - \frac{a_y}{a_z}\right)^2} \\ \text{if } \frac{w_v}{w_f} = \frac{h_v}{h_f} \end{array} \tag{4.9}$$

The length the vector $d(a, S_{v2})^*$ is normally small with z-values that are almost equal ($q_z \approx a_z$). Therefore, Equation 4.9 can be simplified to Equation 4.10.

$$\begin{array}{lcl} d(a, S_{v2})^* & \approx & \frac{w_v}{w_f} \cdot \frac{N}{z} \cdot \sqrt{(q_x - a_x)^2 + (q_y - a_y)^2} \\ d(a, S_{v2})^* & \approx & \frac{w_v}{w_f} \cdot \frac{N}{z} \cdot d(a, S_{v2}) \end{array} \tag{4.10}$$

If the error metric is calculated per triangle and the results averaged (as in MESH), the z-values for all samples on the triangle are more or less equal over the complete triangle and Equation 4.7 may be simplified to Equation 4.11:

$$l^p(S_{v1}, S_{v2})^* \approx \frac{w_v}{w_f} \cdot \frac{N}{z} \cdot l^p(S_{v1}, S_{v2}) \tag{4.11}$$

The modelview transformation translates, rotates and scales the object from the object coordinate space into the camera coordinate space. The Euclidian distance $l^p(S_1, S_2)$ in the object coordinate space is, because of the scaling, scaled with a factor m into the Euclidian distance $l^p(S_{v1}, S_{v2})$. Therefore, Equation 4.11 allows to calculate the (computational expensive) error on the object coordinates $l^p(S_1, S_2)$ in an off-line preprocessing step and to use simple scaling parameters $\frac{w_v}{w_f} \cdot \frac{N}{z} \cdot m$ to adapt the error to the viewing conditions. This is already a huge simplification, but Equation 4.11 must still be implemented very carefully. Calculating the error in an off-line preprocessing step for every triangle generates too much data and even if this could be done, evaluating Equation 4.11 on a per triangle basis needs a visibility determination for every triangle (which is also much too costly). To limit this complexity, triangles are grouped and the error metric for the different groups is then computed in an off-line preprocessing step (see Chapter 2). The correct values

for the visible groups of triangles are combined online with Equation 4.12 to a global error metric (RMS value) for the complete scene.

$$l^2(S_{v1}, S_{v2})^* \approx \sqrt{\frac{\sum_{i=0}^{nG} A_v(G_i) \cdot l^2(G_i, S_2)}{\sum_{i=0}^{G} A(G_i)}} \tag{4.12}$$

where nG stands for the number of different groups, G_i stands for one specific group and $A_v(G_i)$ is the visible area of the group G_i:

$$A_v(G_i) = \left(\frac{w_v}{w_f} \cdot \frac{N}{z} \cdot m\right)^2 \cdot A(G_i) \tag{4.13}$$

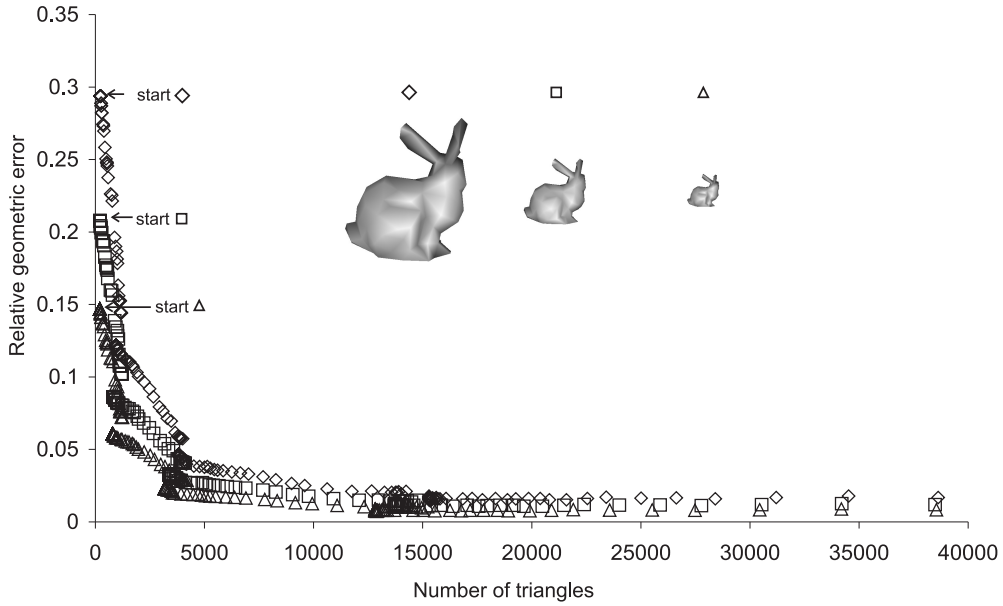
Note that the more triangles the group contains the more the z-value will vary over the surface and the more the real value of the surface integral will differ from Equation 4.12. The view-dependent quality metric is illustrated in Figure 4.12 with the Stanford bunny shown at different distances from the viewpoint. The geometric error metric shows similar trends as the SSIM and PSNR quality metrics. This confirms that the proposed quality metric approximates the real visual quality.

4.3.2 Extra scaling parameters

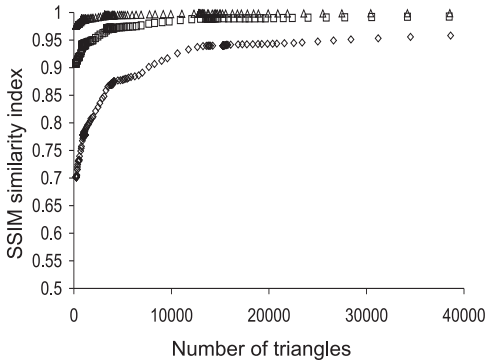
If the surface to surface error becomes smaller, the visual quality normally improves. The visual quality however also depends on other factors (see Section 4.1), which makes this relation sometimes more complicated. Funkhouser and Séquin [52] and Gobbetti and Bouvier [56] have used extra factors for scaling their benefit heuristic with the influence of motion, quality hysteresis, object importance and focus of the user. The same scaling factors correct our quality model, which was proposed in Subsection 4.3.1. The result is shown in Equation 4.14 with f the scaling factor.

$$l^2(S_{v1}, S_{v2})^* \approx \sqrt{\frac{\sum_{i=0}^{nG} f^2 \cdot A_v(G_i) \cdot l^2(G_i, S_2)}{\sum_{i=0}^{G} A(G_i)}} \tag{4.14}$$

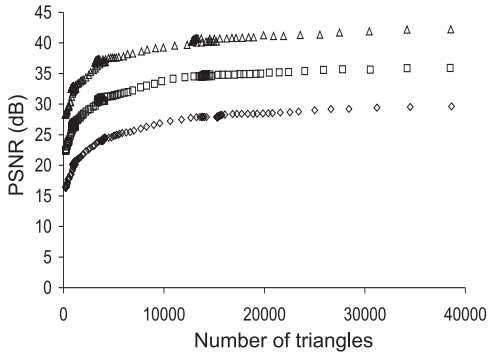
As shown in Section 4.1, the quality depends on the used illumination and shading models. If an object is for instance painted black, little contrast changes appear in the shading of the object and artifacts because of the geometry approximation are only visible for very bad approximations. When the object is painted white, it is more likely that contrast changes introduced by imperfect geometry will be visible. Therefore, an extra scaling factor f_c is multiplied with the scaling factors f from Equation 4.14. The scaling factor f_c is derived as the ratio between the quality of the image with colours that vary because of the geometry combined with the material model and a reference image with colours that only vary because of the geometry.



(a) Geometric error scaled with distance and invisible triangles eliminated



(b) SSIM index



(c) PSNR

Figure 4.12: The view-dependent error metric applied on the Stanford Bunny: the proposed error metric (a) compared with the SSIM (b) and PSNR (c) quality metrics.

For generating the reference, it should be understood how the colour of a pixel is calculated. This colour is the result of the sampled and filtered texture modulated with the colour that is computed from the shading model. An example of a modulation function is given in Equation 4.15.

$$C_p = C_t \cdot C_s \tag{4.15}$$

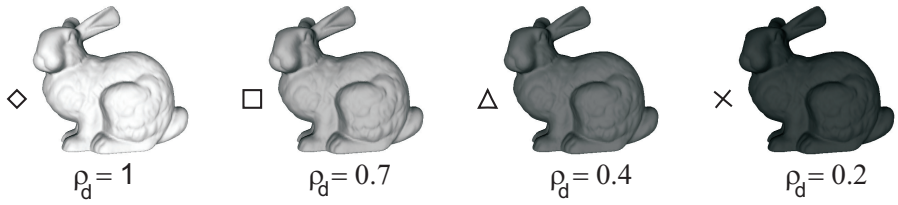
where C_p is the colour of the pixel, C_t is the sampled and filtered colour of the texture and C_s is the colour that is computed by the shading model. With Gouraud shading, C_s is a linear interpolation of the vertex colours that are calculated with an illumination model. A standard illumination model is given in Equation 4.16 [46].

$$C = \begin{cases} e_r + I_{mr}\rho_{ar} + \\ \sum_{i=0}^{i=L} A^i \left(I_a^i \rho_a + I_d^i \rho_d X \left(0, \frac{\overrightarrow{s^i}\overrightarrow{m}}{|\overrightarrow{s^i}||\overrightarrow{m}|} \right) + I_s^i \rho_s X \left(0, \frac{\overrightarrow{h^i}\overrightarrow{m}}{|\overrightarrow{h^i}||\overrightarrow{m}|} \right)^p \right) \end{cases} \tag{4.16}$$

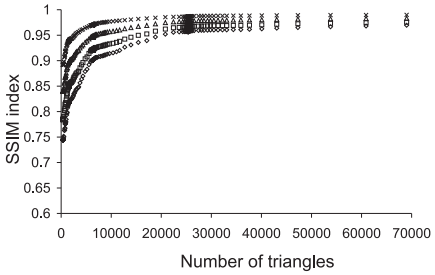
With $\overrightarrow{m}$ the normal vector on the vertex, $\overrightarrow{s}$ the vector from the light source to the vertex, $\overrightarrow{h}$ the halfway vector ($\overrightarrow{v} + \overrightarrow{s}$), $\overrightarrow{v}$ the vector from the viewpoint to the vertex, X the maximum function, I the intensity of the light source for the specific type of light (a=ambient, d = diffuse and s = specular), ρ the reflectivity of the material for the specific type of light, L the number of light sources, p the specular power term (a large p will give small intense spots, a small p gives a smeared out spot), e_r the light that is emitted by the object, and A^i the attenuation factor of the light because of the distance and spotlight factor.

To generate the reference image with colours that only vary because of the geometry, texture mapping is disabled ($C_t = 1$) and the illumination model is initialised so that it only depends on the normals of the vertices (diffuse component). To prevent that the quality is influenced by saturation, the number n of light sources is limited to 1. The position of the light source is the viewpoint of the user.

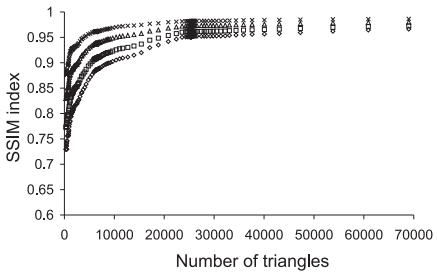
Figure 4.13 illustrates how the quality varies for different materials used with the Stanford bunny. The reference image in Figure 4.13 is the Bunny with a material reflectivity ρ_d equal to 1. The material reflectivity is then decreased for the other images generating darker materials that then mask the geometry artifacts. Four tests are shown in Figure 4.13. The first two tests show the SSIM similarity index and the other tests show the PSNR value for a dark and white background. The PSNR values differ for the two types of background because the screen space area occupied by the geometry typically increases with an increasing resolution. This results in background pixels that gradually become pixels shaded by the geometry. A large colour difference is therefore introduced at the pixels on the silhouette of the object shown with a white background and a dark material. These colour differences completely undo the improving PSNR in the middle of the object for a white background. The PSNR however does increase with a darker material if a dark background is used.



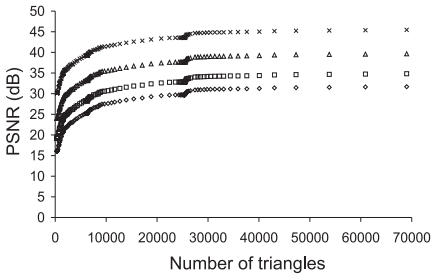
(a) Stanford bunny with different diffuse reflectivity ρ_d



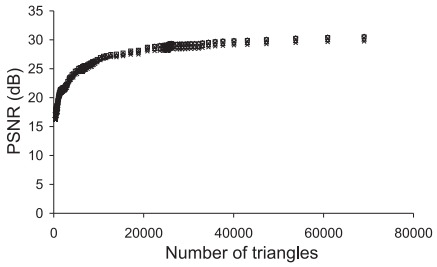
(b) SSIM: black background



(c) SSIM: white background



(d) PSNR: black background

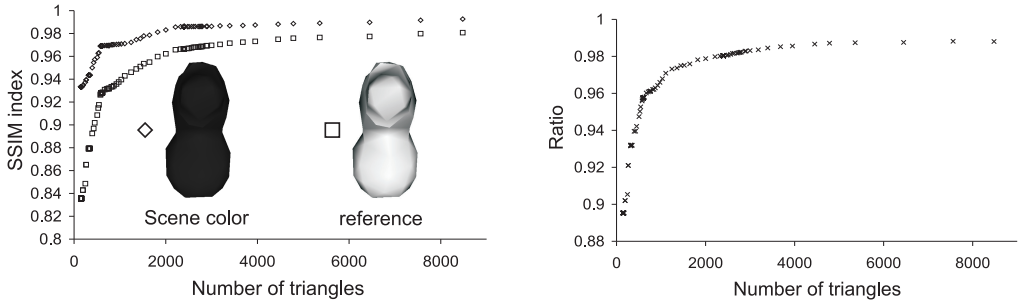


(e) PSNR: white background

Figure 4.13: The variation of the quality on the Stanford bunny because of different materials (a) measured with the SSIM index (b and c) and PSNR (d and e).

Because the SSIM structural similarity index measures the change of the structure in the image, it does not show this dependency on the background colour and it is therefore preferred for deriving the scaling factor f_c.

Figure 4.14.a illustrates the SSIM index as a function of the number of triangles for the reference image and the image with correct materials and lights of a shoe. Figure 4.14.b is the ratio between these two curves. The average of the values on this curve is used as the scaling factor f_c.



(a) SSIM index (b) Ratio

Figure 4.14: The scaling factor f_c of the shoe is the average value of the ratio (b) of the curves in (a).

Figure 4.15 shows the SSIM index as a function of the number of triangles for a reference image and a textured object. The SSIM index for the textured object is much lower than for the reference image because of texture shifting, i.e. the texture moves over the object because of the simplification. This texture shift is visible and is correctly measured by the SSIM index, but it is only noticed by the user when he/she has knowledge of how the image should look or when he/she notices the texture moving over the object when the number of triangles in the object is changed. The ratio of SSIM indices is thus not capable of correctly predicting the quality that is perceived by the user. Ferwerda et al [45] have derived correct models for the masking effect of textures, but these models are so complex that they can only be used to accelerate complex global illumination algorithms. The challenge for future work is thus to find correct simplifications of such a model that can also be used in interactive applications.

4.4 Results

Figure 4.16 illustrates the quality difference for an optimisation that uses our proposed quality model with an optimisation that uses the simple benefit heuristic of Gobbetti et al. [56]. The latter heuristic is a good representative of the quality heuristics that are used in previous work.

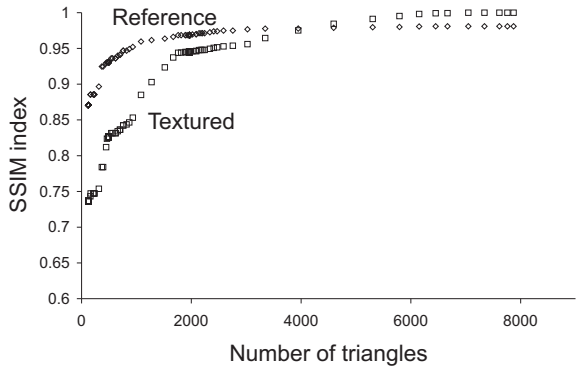
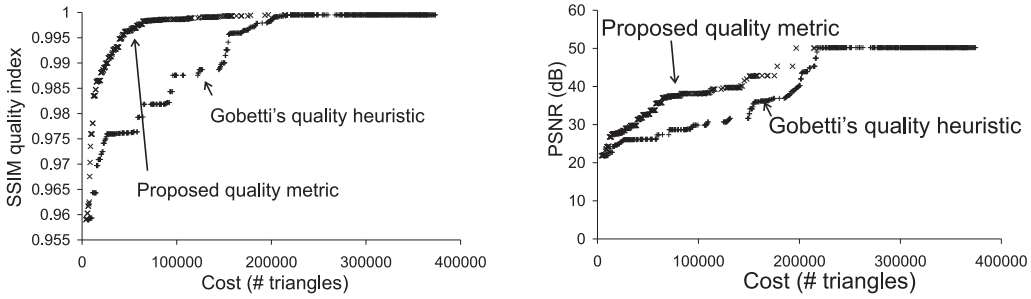


Figure 4.15: The method fails for textured objects: the textured object has a lower SSIM index than the reference model



(a) SSIM index (b) PSNR

Figure 4.16: Comparison of an optimisation using the proposed quality metric compared with an optimisation that uses a simple Benefit heuristic: a) SSIM index and b) PSNR.

To validate our proposed quality metric, we measure the similarity between the rendered images of the optimised and original content. Two different image quality metrics are used to compare these images: the SSIM similarity [156] and the PSNR. The SSIM index rates the quality of images with values between 0 and 1. Wang et al. [156] report SSIM values of 0.75 for very disturbing artifacts and 0.94 for visible artifacts. An SSIM value of 1 means perfect quality. The PSNR is used a lot in image processing to evaluate the visual quality. The PSNR is based on the mean squared error and is expressed in dB.

Because both heuristics start with the base meshes and end with the full resolution meshes, the visual quality is the same for both heuristics on the start and end points. Figures 4.17, 4.18 and 4.19 show a visual comparison of the rendered images for a respective cost of 12000, 18000 and 30000 triangles that are distributed over the objects with the two different heuristics. These images illustrate the numbers of Figure 4.16. The third picture (difference) in the images 4.17, 4.18 and 4.19 show the difference between the two metrics and illustrate where the reader should look in the first two pictures to see the differences.

The simple heuristic selects the lamppost in the city scene for the addition of triangles, because it has a large area and few triangles. The proposed quality metric however detects a large error on the shirt of the character and starts the optimisation on the shirt. This results in image quality values that are higher for the proposed heuristic than for the simple heuristic for the same cost. The PSNR gives a difference of 10 dB for a total triangle cost of 100000 triangles. The SSIM values show that the proposed quality metric results in an optimisation that reaches the almost perfect SSIM values much faster than the simple heuristic.

4.5 Conclusion

The assessment of quality is in an interactive application typically done with a heuristic that estimates the visual quality of the *rendered image*. Because, many prior art techniques sacrifice the realism/accuracy of the quality model for simplicity, we propose a new approach that uses realistic error metrics measuring the fidelity of the geometric model. These metrics however neglect the user's viewpoint and are typically too complex to be used directly in an interactive application. Therefore, we propose to use a mixture of off-line preprocessing steps for computing the complex parts of the quality model and a simple online combination step for evaluating the visual quality with respect to the user's viewpoint.

Besides the geometric accuracy, we have included extra scaling factors that take other important quality parameters (e.g. material) into account. The proposed quality metric performs better than the previous proposed error metrics, but many issues remain unsolved. One of the most important remaining issues is the influence of the

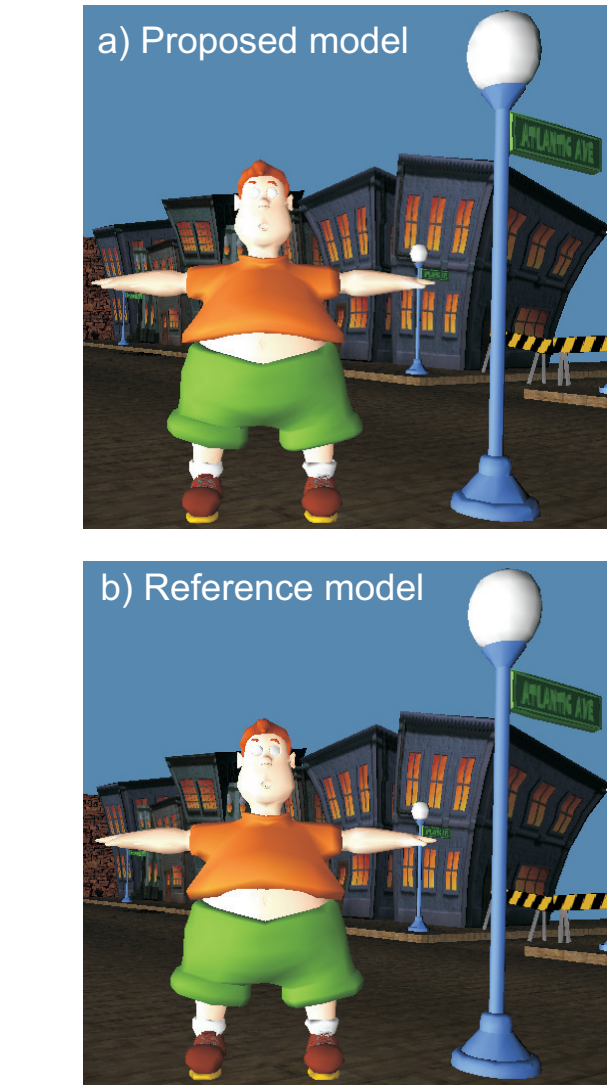


Figure 4.17: Visual comparison at 12000 triangles: a) Proposed quality model, b) Simple benefit heuristic and c) difference between the two images.

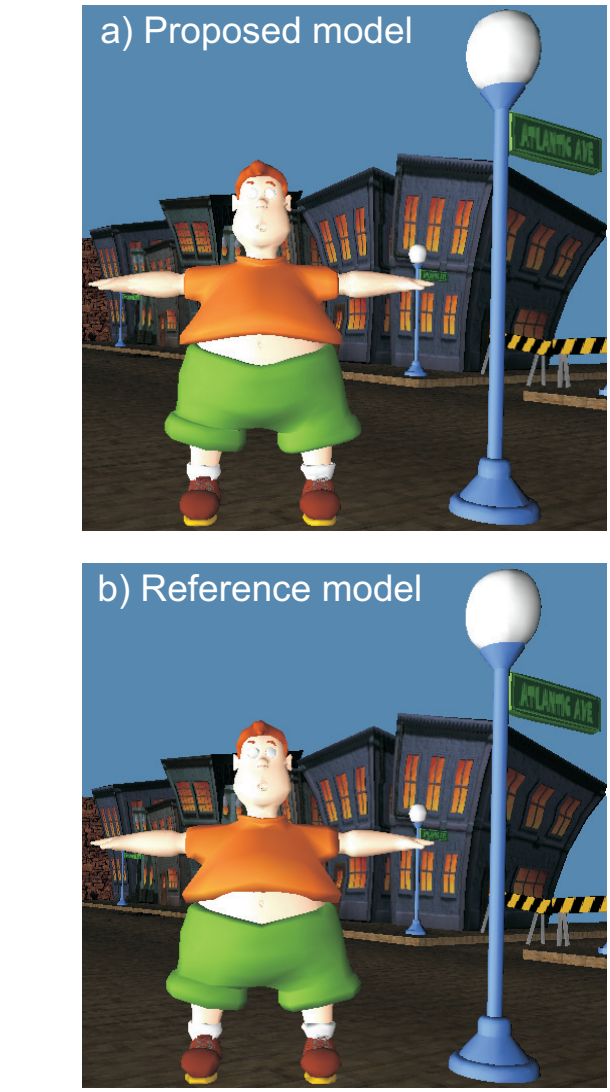


Figure 4.18: Visual Comparison at 18000 triangles: a) Proposed quality model, b) Simple benefit heuristic and c) difference between the two images



Figure 4.19: Visual Comparison at 30000 triangles: a) Proposed quality model, b) Simple benefit heuristic and c) difference between the two images

texture on the visual quality: texture shifting because of the simplification process, texture masking and the influence of the texture filters on the visual quality.

Chapter 5

Multi-resolution mesh representation

The optimisation framework of Chapter 2 decides for every subsequent frame a triangle budget for a 3D object according to its benefit in the scene. This chapter discusses several techniques that can be used for changing the geometric structure of the object in order to meet this triangle budget with a maximum quality. Because networked access of 3D content is important in our target scenario (e.g. the online game presented in the introduction), we have chosen to implement a multi-resolution technique from previous work with a very good compression: MPEG-4 Wavelet Subdivision Surfaces (WSS). Wavelet Subdivision Surfaces however change the mesh size in steps of a factor four, which only allows a coarse regulation of the rendering time. Therefore, we have extended the standard MPEG-4 WSS decoder to support adaptive subdivision that enables the important regions of the mesh to be rendered at a higher resolution than others.

Section 5.1 describes previous work in the field of multi-resolution mesh representation. Section 5.2 adds the necessary blocks to the standard decoder for making MPEG-4 WSS continuously scalable. Section 5.3 discusses the complexity of our proposed decoder and investigates the usability of the proposed methods for the optimisation. Section 5.4 proposes a new method that eliminates an important part of the CPU overhead and the last section ends with the conclusions.

5.1 Related work

The field of multi-resolution object representation has been very fertile so far. The goal of this section is to give an overview of the most important techniques for our work, but a complete discussion of the available techniques can be found in the book of Luebke et al. [89] and in the survey of Peng et al. [115]. A first subsection discusses the traditional approach of discrete Level Of Detail. Subsections 5.1.2 and

5.1.3 then discuss respectively continuous LOD and the needed extensions to make it view-dependent. The last subsection focuses on the technique that is used in this work: subdivision surfaces and extensions of it, known in MPEG-4 as Wavelet Subdivision Surfaces (WSS).

5.1.1 Discrete Level Of Detail

The first technique for multi-resolution object representation that is nowadays still famous is discrete Level Of Detail (LOD) proposed by Clark in 1976 [34]. This approach creates multiple versions of every object in an off-line preprocessing step and the online decision taking selects the most appropriate version. Deriving the LODs was first done manually, but with the advent of the mesh simplification techniques of Subsection 5.1.2, the process is now automated.

The main advantage of this technique is its low online complexity. The online step simply selects the appropriate Level Of Detail without extra computations. An additional advantage is the possibility to optimise the content in the preprocessing step for a maximum performance on the used hardware.

The main disadvantage of discrete LOD is the disturbing transition between different Levels Of Detail [52], which can be smoothed out at the cost of extra complexity. Alpha blending for instance combines the colour value of the pixel being processed with that of the pixel already stored in the frame buffer. During the transition between two LODs, both LODs are rendered with alpha values that sum up to a value of 1. In the beginning of the transition, the first LOD is opaque (alpha value 1) and the next LOD is invisible (alpha value 0). The alpha value of the first LOD is then gradually decreased until the first LOD is invisible and the alpha value of the target LOD is gradually increased until it is completely visible. Although this seems a nice solution, rendering two versions of the content increases the complexity, which diminishes the main advantage of discrete LOD.

5.1.2 Continuous Level Of Detail

Instead of creating different versions in the preprocessing step, continuous LOD (multi-resolution/progressive meshes) constructs a data-structure that records the simplification process [64]. This data structure is then used online for creating the geometry at the desired Level Of Detail. The major advantage of this approach is the fine-grained control of the geometries complexity. Progressive meshes also allow streaming the geometry: a low complexity base mesh is followed by a stream of detail information sorted from high to low importance. A scheme for compressing progressive meshes was proposed by Pajarola et al. [111, 112].

Figure 5.1 shows a selection of operators that can be used in off- and online steps for simplifying the geometry. Refining the meshes is done by inverting the arrows

(operators) of Figure 5.1. Hoppe et al. [66] have introduced the edge collapse that removes the triangles t_0 and t_1 by replacing the edge e with the vertex v_a. Garland and Heckbert [54] have extended this principle to a virtual edge collapse that allows collapsing two unconnected vertices. Schroeder et al. [133] have introduced the vertex removal operator that first removes the vertex v_0 and then triangulates the resulting hole (triangles t_a to t_d). The vertex removal operator is actually a special case of polygon merging that merges several triangles into one polygon, removes all the vertices in the interior of the polygon and then triangulates the polygon. Hamman [61] has introduced the triangle collapse operator that collapses the triangle t_0 into the single vertex v_b. The triangle collapse is equivalent to two edge collapses (edge e into vertex v_a and edge e_a into vertex v_b).

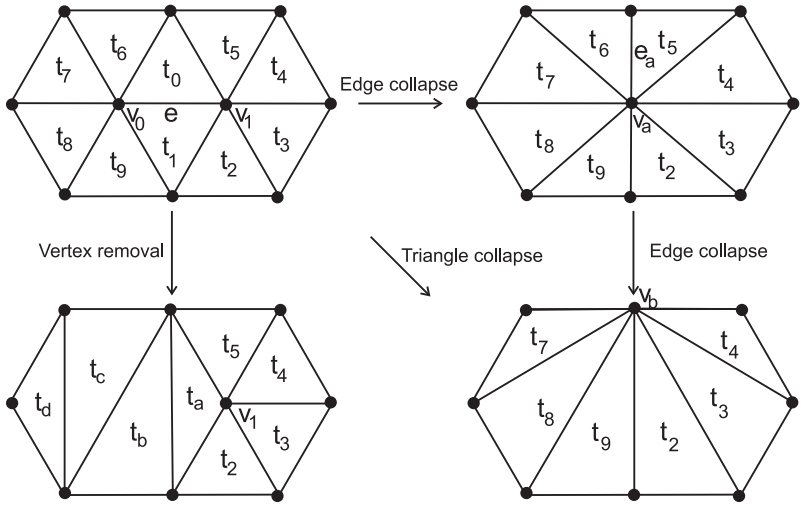


Figure 5.1: Simplification operators.

The measure of the impact of the operator on the quality of the geometric approximation determines the priority of the simplification operations, e.g. the next edge collapse is always the one with the lowest impact on the quality. Garland and Heckbert [54] evaluate the geometric approximation with a measure of the distance between the original and the simplified mesh. Lindstrom and Turk [85] use the Mean Squared Error between the images of the original and simplified meshes and Williams et al [160] and Reddy [119] have derived a metric from the Contrast Sensitivity Function (CSF), which is a simple model of human vision.

5.1.3 View-dependent Level Of Detail

View-dependent LOD extends continuous LOD by taking the viewpoint of the user into account. Typical criteria for view-dependent LOD are distance and silhouette preservation: regions of the object that are close to the viewer or regions that are on the silhouette of the object are shown at a higher LOD than others. Examples of

view-dependent LOD are presented by Hoppe et al. [65], Luebke et al. [88] and Xia et al. [165]. The latter has also included extra scene information, such as the light sources.

The advantage of view-dependent LOD is the possibility to assign the triangles to the object regions that are really important for the visual quality. The major disadvantage however is the larger complexity in terms of computation and memory consumption on the current platforms. On the current platforms, the view-dependent LOD is computed on the Central Processing Unit (CPU) and further communicated to and rendered on the Graphical Processing Unit (GPU). The gain of view-dependent LOD is found in the GPU that needs to render less triangles for an equal quality, but the pain is found on the CPU that needs to process and communicate the geometry every time the viewpoint changes (see e.g. Subsection 5.3.1). An interesting path for future work could be to investigate how the current platforms could be changed to decrease the overhead for view-dependent LOD. This work could for instance be important for mobile devices where the energy consumption is decreased and the performance increased by limiting the work to what is really important for the visual quality.

5.1.4 Wavelets and Subdivision Surfaces

Subdivision surfaces, introduced in computer graphics by Catmull and Clark [28] and Doo and Sabin [39] in 1978, are a simple and efficient way for deriving a smooth surface from an initial control mesh. Figure 5.2 for instance illustrates how subdivision surfaces turn an icosahedron with 20 triangles into a smooth sphere with 1280 triangles.

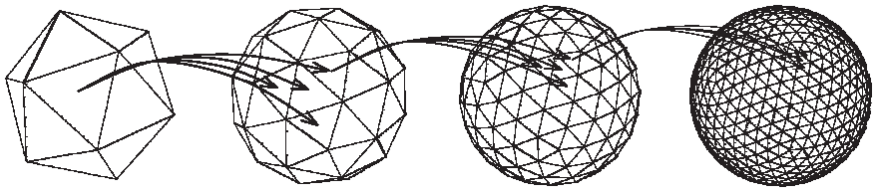


Figure 5.2: Smoothing an icosahedron with subdivision surfaces into a sphere.

The basic operations of subdivision, illustrated in Figure 5.3, are:

1. Splitting: Each old triangle (the parent) is split into four new triangles (the children) by inserting the new vertices d, e and f on the midpoints of the edges and connecting them with triangles;

2. Averaging: Each vertex d, e and f its position is replaced with a weighted average of its neighbours. The filters used for this operation are called subdivision

masks. Approximating masks average both the old and new vertices, while interpolating masks only average the new vertices. An important approximating mask for our work is the mask of Loop [86] which was refined by Biermann et al. [9] to support piecewise smooth subdivision surfaces. An important interpolating mask for our work is the modified Butterfly mask introduced by Zorin et al. [168].

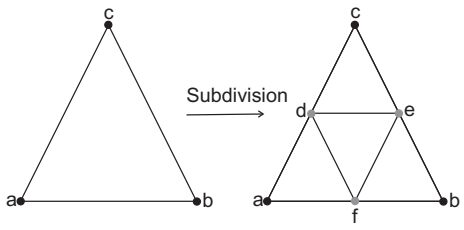


Figure 5.3: Subdivision of a single triangle.

The simplification methods, discussed in Subsection 5.1.2, typically derive the coarse control mesh M_0. After applying a subdivision operator on M_0, a new set of vertices, positioned on a new approximating mesh M_1, is generated. One can not expect these vertices to lie on the original target mesh T because subdivision surfaces only smooth the mesh M_0. However, nothing prevents one to correct their position so that their final position is on the target mesh T. After applying a number of subdivision and correction steps, a mesh M_i is derived that is the final approximation of the target mesh T. This process is illustrated in Figure 5.4 and was introduced by Lee et al. (MAPS) [82] and Morán [24]. Lee et al. [81] and Guskov et al. [59] have further refined this method to represent the correction vectors as simple scalar values.

Khodakovsky et al. [75] have used MAPS [82] to generate a mesh with subdivision connectivity and they have used wavelets combined with a traditional zero tree coder (e.g. SPIHT of Said and Pearlman [127, 7]) to compress the geometry. A very good compression is obtained that heavily improves on other schemes such as the compressed progressive meshes of Pajarola et al. [75]. Because of the nice compression obtained with wavelet meshes, the work of Khodakovsky et al. [75] and Morán et al. [24] has been the basis for the MPEG-4 Wavelet Subdivision Surfaces (WSS) specification [26].

Subdivision surfaces and wavelet meshes typically multiply the number of triangles with 4 in every subdivision step. Subdivision surfaces are thus a form of discrete LOD for which the different LODs are not computed in a preprocessing step, but calculated online. This obviously results in a much better compression of the 3D geometry. Continuous LOD can be achieved with adaptive or non-uniform subdivision surfaces or wavelets that only subdivide the important regions of the geometry. Khodakovsky et al. [75] have used adaptive subdivision in their mesh codec. Zorin et al. [169] have used adaptive subdivision surfaces in their framework for multi-resolution mesh editing.

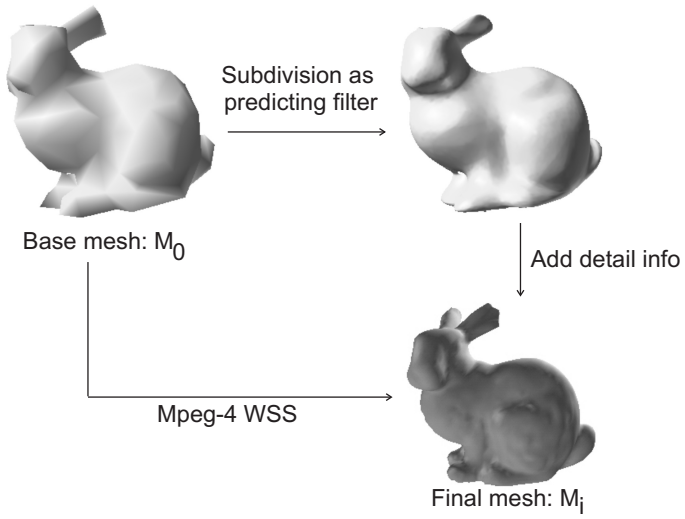


Figure 5.4: Displaced Subdivision Surfaces: subdividing and adding detail decodes the base mesh.

The MPEG-4 WSS specification supports the adaptive decoding by partial transmission [47], but a non-uniform decoder did not yet exist at the time of the research. Therefore, we have implemented an adaptive MPEG-4 WSS mesh decoder for real-time rendering. Several possibilities for the adaptive decoding of the content were implemented and tested for their usability in the optimisation framework.

Note that wavelet subdivision surfaces are not the only tools in MPEG-4 for compressing 3D geometry. 3DMC for instance is a method developed by Taubin and Rossignac [149]. 3DMC was however not suited for our framework because of the worse compression [75]. A third method that can be used for compressing 3D geometry in MPEG-4 is Meshgrid [128, 129]. The latter supports scalable transmission of 3D objects and also has a very nice compression, but it was still under development at the time of the research. Therefore, we have chosen to use MPEG-4 WSS geometry. Adding Meshgrid to the optimisation framework is an interesting path of future work.

5.2 Continuous Level Of Detail with Subdivision Surfaces

Subdivision surfaces and wavelets are typically forms of discrete LOD. Adaptive subdivision surfaces minimise the number of triangles by only subdividing the regions in the mesh that are important for the visual quality. We have extended this principle of adaptive subdivision to support continuous LOD. Therefore, we have followed the reasoning that is typical for continuous LOD to decode the geometry into a data-

structure that is then used online for easily adapting the content. Figure 5.5 shows a high-level overview of the proposed decoder for the geometry.

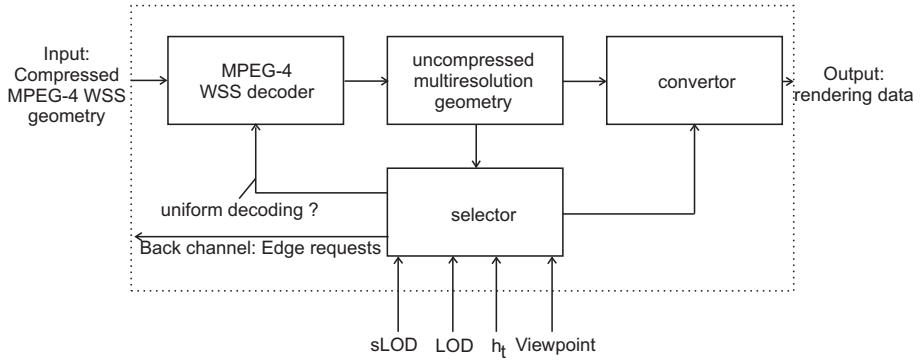


Figure 5.5: The proposed resolution selector.

The selector, which is further discussed in Subsections 5.2.1 and 5.2.2, selects the regions in the mesh that are the most important for the visual quality. The converter, which is described in Subsection 5.2.4, transforms the selected triangles of the multi-resolution structure into rendering data. The converter uses the input from the selector and is responsible for the smooth transitions between different resolutions in the same mesh. A standard decoder for WSS geometry [47] decodes the compressed data into a quad-tree data-structure containing the multi-resolution geometry. This decoder typically fills the multi-resolution data-structure in a uniform way, but it can also be extended with a non-uniform mode that only decodes the data that is needed by the selector and convertor. This extension is discussed in Subsection 5.2.5 and can be useful to limit the decoding work to what is really rendered. This non-uniform decoder can be useful to reduce the load time, the memory consumption and the energy consumption (important for mobile devices).

5.2.1 The Selector - Importance criteria

The selector of Figure 5.5 analyses the decoded data in the multi-resolution structure and selects the important regions of the mesh that need to be shown at a higher resolution than others. The assessment of importance can be done with the following criteria that are mainly collected from the literature and that are further explained in this subsection:

1. Geometric criteria intrinsic to the target surface;
2. View-dependent - silhouette preservation;
3. View-dependent - distance.

To check the geometric criteria, one could use a direct measurement of the error between the triangles of the approximating mesh and the target mesh T for the qualification of importance (e.g. the screen space error of Hoppe et al. [65]). This however requests a lot of extra information for all triangles at all subdivision levels that must be encoded and transmitted with the geometry. To limit this extra information, we have chosen to use a heuristic (Equation 5.1). Several heuristics already exist that are mainly used in mesh simplification algorithms to decide which triangles may be eliminated from the mesh for a minimum quality decrease. Hoppe et al. [66] have used an energy function; Garland et al. [54] have used quadrics and Lee et al. [82] have used a combination of curvature and area.

Figure 5.6 illustrates the choice of the heuristic in 2D. Figure 5.6 shows a smooth curve and two different linear approximations. The arrows on the linear approximations in Figure 5.6 represent the normals on the lines. The better the linear approximation, the smoother and the smaller the angle θ is between two neighbouring normals. Therefore, the angle θ between normals on neighbouring triangles is a good criterion to decide whether subdividing a triangle will or will not further smooth the approximating surface.

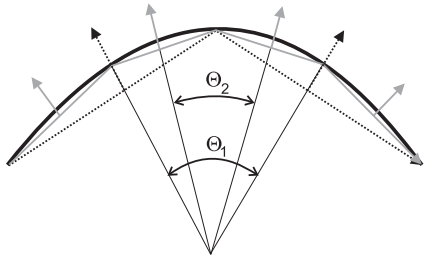


Figure 5.6: the quality of two linear approximations of a smooth curve: the smaller the angle θ is between two normals on neighbouring lines, the better the lines approximate the real smooth curve.

If a mesh however contains a small area with a high curvature, a lot of triangles will be added to this small area without a real increase in visual quality. Therefore, the quality heuristic is extended with an area parameter to the heuristic of Equation 5.1, which is similar to the heuristic used by Lee et al. [82].

$$h(\lambda, e) = \lambda * \left(1 - \frac{A(T_{i,j})}{max_{T_{i,j}} A(T_{i,j})}\right) + (1 - \lambda) * \left(\frac{\overrightarrow{n_1} \cdot \overrightarrow{n_2}}{|\overrightarrow{n_1}| \cdot |\overrightarrow{n_2}|}\right) \tag{5.1}$$

where $A(T_{i,j})$ is the area of the triangles i and j that contain the edge e and $\overrightarrow{n_1}$ and $\overrightarrow{n}_2$ are the normals on those triangles. The dot product between the two normals is an estimate of the curvature. λ is used to select which one of the two parameters (area or curvature) is considered the most valuable one for the used content. E.g. content with a lot of large and flat triangles which closely match the target surface

may be better off with small λ values. The heuristic $h(\lambda, e)$ of Equation 5.1 returns a value between 0 and 1, giving the importance of edge e. The parameter h_t from Figure 5.5 is a threshold which selects all edges with a value for $h(\lambda, e)$ smaller than h_t to be split with the rules shown in Figure 5.3. Figure 5.7 illustrates how triangles are added to the geometry by sweeping the parameter h_t from 0 to 1.

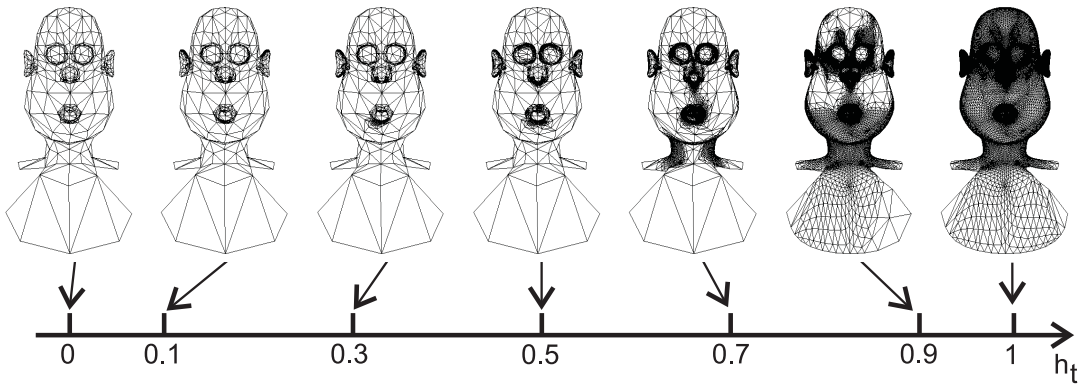
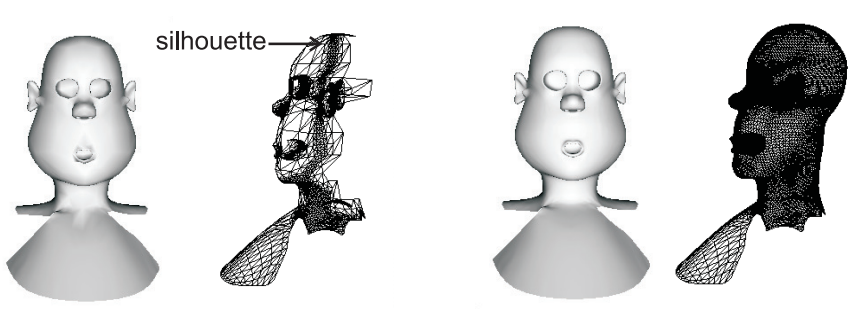


Figure 5.7: Sweeping the parameter h_t from 0 to 1 changes the number of triangles.

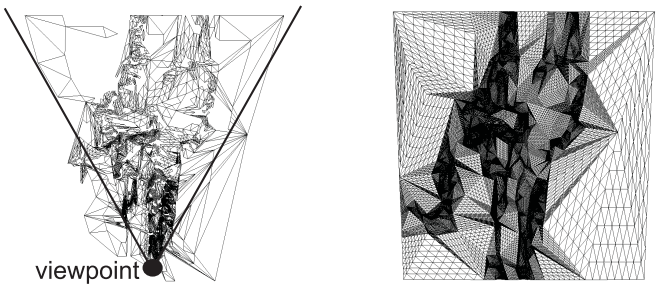
The first view-dependent criterion, illustrated in Figure 5.8, detects edges on the silhouette of the object as important because the visual quality is known to be sensitive to a smooth representation of the silhouette [165]. An edge of the mesh is on the silhouette and must be split if one of two neighbouring triangles is visible and the other is not. The importance of an edge thus depends on the user viewpoint. An example of a very efficient silhouette detection algorithm has been proposed by Benichou and Elber [8]. The *viewpoint* parameter in Figure 5.5 is used for detecting the silhouette.



(a) Silhouette preserved (b) Uniform

Figure 5.8: The first dynamic criterion improves the visual quality for a lower triangle cost by silhouette preservation: a) object subdivided at the silhouette; b) uniformly subdivided object.

The second view-dependent criterion marks edges close to the user viewpoint as more important as edges far away. Figure 5.9.a for instance shows how the number of triangles of the terrain decreases with an increasing distance to the viewpoint. Figure 5.9 illustrates the uniformly rendered mesh. The optimisation described in Chapter 2 typically deals with this issue for small objects, but for larger objects such as terrains, the distance varies a lot within an object. The distance criterion is then very useful for eliminating distant detail. The up-view in Figure 5.9 also shows how invisible triangles are eliminated by the selector.



(a) View-dependent (b) Uniform

Figure 5.9: The second dynamic criterion improves the visual quality for a lower cost by decreasing the resolution with an increasing distance to the viewpoint: a) resolution dependent on distance and viewing angle ; b) uniform subdivision

5.2.2 The selector - Subdivision code

The criteria explained in Subsection 5.2.1 are used to decide what edges need to be split. To prevent holes in the geometry (referred as cracks in the specialised literature), all child triangles that share a vertex with a selected edge must be created by the convertor. We have introduced a subdivision code, which is derived by the selector and which exactly tells to the convertor what child triangles need to be created. The subdivision code allows to check the above mentioned criteria for a triangle on an arbitrary subdivision level and to use the results on any of its children or grandchildren. This limits the number of triangles for which the criterion needs to be checked and thus also the complexity. The same subdivision code is also used by the MPEG-4 WSS decoder to control the non-uniform decoding discussed in Subsection 5.2.5.

The 7 possibilities for creating child triangles without the introduction of cracks in the geometry are shown in Figure 5.10. As illustrated in Figure 5.10.1, which shows the numbering of vertices and child triangles, the subdivision code is a 3-bit code for which every bit corresponds with a vertex of the triangle. Bit i of the code is set to 1 if vertex v_i is part of an edge selected by the above-mentioned criteria, and

to 0 otherwise. If vertex v_i is selected, then the child triangle c_i that contains the vertex is selected as important. The fourth child c_3, that does not correspond with a specific vertex, is selected when two or more of the triangle's vertices are part of a selected edge (Figures 5.10.4, 5.10.6, 5.10.7 and 5.10.8). Four cases now occur:

1. A triangle shares no vertices with the selected edges. The subdivision code is 0 (not represented in Figure 5.10).

2. A triangle shares a single vertex with the selected edges (black dots in Figures 5.10.2, 5.10.3 and 5.10.5). In Figure 5.10.2 for instance, vertex v_0 is part of a selected edge (from another triangle) and the child triangle c_0 is selected. Because only one of the four children is selected, a crack is created and two extra triangles are needed to fill it. These extra triangles must be removed when the other child triangles are also selected.

3. A triangle contains a selected edge (edges marked with e in Figures 5.10.4, 5.10.6 and 5.10.7). In Figure 5.10.4 for instance, the children next to the edge e are selected. To avoid cracks, child c_2 (marked with X) must also be selected for rendering, but it cannot be further subdivided as explained in the next paragraphs.

4. Two or more edges are selected and all child triangles are selected (Figure 5.10.8).

Figure 5.11 shows how these rules are used to locally increase the resolution around the bold edges of the mesh without introducing cracks. The important edges of the triangles a, d, e and f are subdivided (case 3) and to prevent a crack in the triangles b and c, the child triangle that shares a vertex with the important edges must be created.

As illustrated in Figure 5.2, subdivision surfaces iteratively smoothen a base mesh until a subdivision level LOD (one of the parameters of the selector in Figure 5.5), which means that once a child triangle is selected, it becomes a candidate to be further subdivided. The subdivision is ideally stopped on an arbitrary level, which can vary for the different triangles in the mesh. The only constraint, needed to prevent cracks, is that the level of neighbouring triangles may not differ by more than 1.

We assume that once a triangle, edge or vertex is eliminated, none of its child triangles, edges or vertices will be selected on a higher subdivision level. Therefore, a child triangle first inherits the subdivision code of its parent. In a next step, the importance of the edges that are selected by the inherited subdivision code is re-evaluated using the above mentioned criteria and in a third step, the subdivision code is adapted by removing the *ones* (vertices) of the code that are no longer important for this and higher subdivision levels.

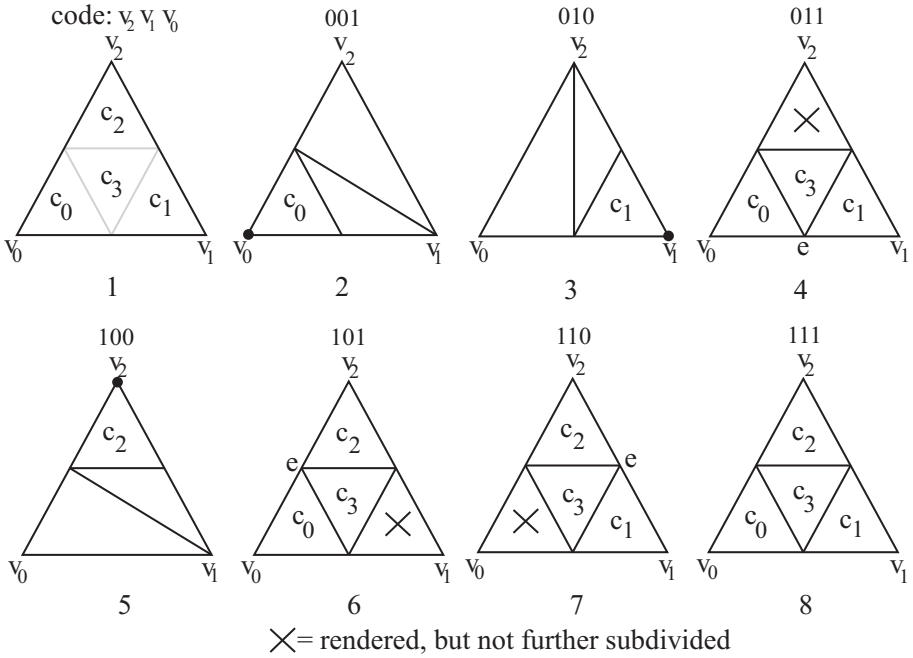


Figure 5.10: Non-uniform subdivision: 1) a triangle and its subdivision code derived from its vertices, 2 to 8) the seven possible ways for subdividing a triangle and their subdivision codes.

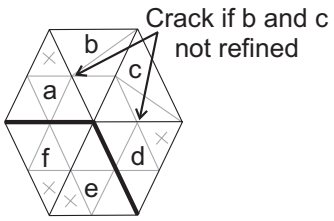


Figure 5.11: Non-uniform subdivision around the bold line.

The inheritance of the subdivision code is illustrated with Figure 5.12, which is the result of further subdividing Figure 5.10.4 next to the edge e. The resulting child edges e_a and e_b of the children c_0 and c_1 in Figure 5.12 are initially marked as important. Figure 5.12 shows the same subdivision pattern for the children c_0 and c_1 as for the parent triangle, which results in the same subdivision code 011 for the children c_0 and c_1. The child c_3 only has one vertex in common with the edge e and it therefore derives the code 001.

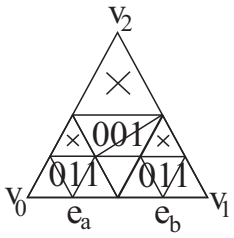


Figure 5.12: Inheritance of the subdivision code for higher Levels Of Detail: the edge e from Figure 5.10.4 is further split into the edges e_a and e_b

The subdivision code is then re-evaluated for the edges e_a and e_b and if they are marked as unimportant the subdivision codes adapted. If for instance the edge e_a is marked as unimportant, then vertex v_0 is no longer part of an important edge and its bit in the code of child c_0 is reset to 0, which yields the code 010 for child c_0. To prevent a crack, also the code of child c_3 is reset to 000.

This procedure can be simplified for the silhouette preservation and the static criterion. With silhouette preservation, one can assume that the silhouette is correctly detected on the base mesh and the child triangles may inherit the importance calculations from the parent triangle, without re-evaluating the importance of the edges. This significantly reduces the selector's complexity.

A second simplification is possible for the static criterion. Because subdivision surfaces typically smooth the mesh, one can assume (simplify) that the curvature of the edges and the area of the triangles on subdivision level $l+1$ is lower than on level l. Therefore, one can assume that all edges of level l are more important than edges on level $l+1$ and thus all triangles of level l must be added before the triangles of level $l+1$. Therefore, the proposed decoder subdivides the geometry uniformly up to the Level of Detail for selection ($sLOD$) and the selector then evaluates the static criterion for that subdivision level. The shown subdivision level (LOD) of the selected triangles is then ideally equal to $sLOD+1$. However, because the complexity of the selection is linear with the number of triangles and the number of triangles is multiplied by 4 every subdivision level, low $sLOD$ values are preferable with respect to the selection complexity. It may therefore be necessary that for complexity reasons only low $sLOD$ values are allowed. The subdivision code is then inherited without re-evaluation for higher LOD values.

5.2.3 Summary of the selector parameters

The selector of Figure 5.5 has 4 input parameters that control the selection discussed in the previous subsection.

1. The parameter h_t is a threshold which selects all edges for subdivision with a value for $h(\lambda, e)$ smaller than h_t (Equation 5.1);
2. The parameter $sLOD$ is the parameter that controls the Level Of Detail for the selection. The mesh is uniformly subdivided until this level and the criteria are calculated for the triangles on this level;
3. The parameter LOD is the maximum Level Of Detail that is generated and is the only parameter that can be used for changing the resolution with uniform subdivision surfaces. The LOD can be decreased by the selector if it decides that the distance to the viewpoint is too large.
4. The *viewpoint* parameter is needed by the selector to calculate the silhouette and distance of the object to the viewpoint.

5.2.4 The Convertor

The converter transforms the multi-resolution structure into rendering data and is responsible for the smooth transitions between different resolutions in the mesh. The convertor iterates over all triangles on the subdivision level for selection ($sLOD$) and uses their subdivision code for generating the triangles for the rendering. Figure 5.13 shows the pseudo code for the normal and simplified conversion of a triangle with subdivision code 011. The conversion first checks whether it is needed to further subdivide the triangle (stop condition for the recursion) and draws the triangle if not. In the other case, the subdivision code of the triangle is read and the according procedures, similar to the one of Figure 5.13, for converting the child triangles are called.

The simplified procedure of Figure 5.13 is used for the geometric and silhouette criteria (see subsection 5.2.1). These criteria only use the inherited subdivision code and the check for the two other possible subdivision codes is thus not needed in the simplified conversion.

The convertor should also generate rendering data structures that are the most efficient for the used renderer. One of the fastest ways for rendering the geometry is to group its vertices and triangles into arrays and to issue a single drawing command for the complete array. The convertor must at least create the triangle array, but must not necessarily create the vertex array. Some renderers only process the vertices that are actually accessed by the triangles. For these renderers, the MPEG-4 WSS decoder can create the vertex array with all possible vertices and forward it to the

```
void convertCode011( triangle T ) {
  if ( (level(T) == LOD) || (code == 0) )
    drawTriangle(T);
  else {
    if (code == 011)  {        // Inherited code
      convertCode011(T→child[0]);
      convertCode011(T→child[1]);
      convertCode001(T→child[3]);
      drawTriangle(T→child[2]);
    }
    else if (code == 001)    // Inherited code - 2
      convertCode001(T);
    else if (code == 010)    // Inherited code - 1
      convertCode010(T);
  }
}
```

Normal conversion

```
void convertCode011( triangle T ) {
  if ( level(T) == LOD )
    drawTriangle(T)
  else {         // Only inherited code
    convertCode011(T→child[0]);
    convertCode011(T→child[1]);
    convertCode001(T→child[3]);
    drawTriangle(T→child[2]);
  }
```

Simplified conversion

Figure 5.13: Pseudo code for the normal and simplified conversion of a triangle

renderer without an intervention of the convertor. Other renderers however (e.g. Mesa3D [114]), first manipulate all the vertices in the vertex array before actually touching the triangles. Consequently, it is important for these renderers that the convertor also minimises the number of vertices in the vertex array. Therefore, the procedure for drawing the triangles, illustrated in Figure 5.14, only changes the vertex array if it is needed for the renderer.

```
void drawTriangle ( triangle T) {
  for ( i = 0 ; i < 3 ; i++ ) {
    if (vertex array needed)
      if (vertex[i] not in vertex array) {
        Add vertex to vertex array
        Add attributes to attribute array
      }
    Add reference to vertex[i] to triangle array
  }
}
```

Figure 5.14: Pseudo code for drawing a triangle

5.2.5 Non-uniform decoder

The output of the selector can also be used by the MPEG-4 WSS decoder to only fill the multi-resolution structure with the triangles that are needed by the convertor. The procedure for non-uniform decoding is similar to the convertor's render procedure, but calculating the mathematical equations for the subdivision operation needs more data. Figure 5.15 for instance illustrates the data dependencies (grey vertices) to calculate the black vertex with a standard subdivision scheme. It is possible that the grey vertices a and b are not part of a rendered triangle, but that they are needed for calculating the black vertex. Because of the extra data dependencies,

the decoder calculates all children of a triangle if its subdivision code is different from 0, i.e. all subdivision codes larger than 0 are interpreted as subdivision code 7 for the decoding. The data dependencies also require that the decoded *LOD* of neighbouring triangles does not differ by more than 1.

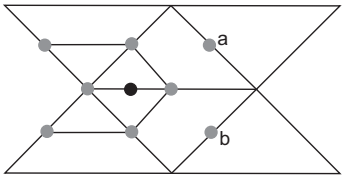


Figure 5.15: Data dependencies for the Butterfly subdivision scheme: the grey points are needed to calculate the black point.

Non-uniform decoding decreases the initial decoding time because it only decodes the actually needed data. When the viewpoint is changed, it is possible that the optimisation needs to adapt the content and that different triangles are selected for rendering. With non-uniform decoding, triangles are only created as needed and the decoder needs to update the uncompressed multi-resolution geometry from Figure 5.5. This increases the complexity of the content adaptation. The decoding time is thus spread over several frames, but because typically not all triangles are needed (decoded) time (and energy) is saved.

5.3 Performance analysis and modelling

Equation 5.2 shows the execution time of the decoder for constructing the rendering data of the 3D objects at the correct resolution.

$$T = \sum_{i=0}^{N} T^i_{select} + T^i_{convert} + T^i_{decode} \tag{5.2}$$

A first important issue for the execution time of the decoder, discussed in Subsection 5.3.1, is the number N of objects for which the optimisation framework decides to change the resolution. The second issue, discussed in Subsection 5.3.2, is the time T^i_{select} and $T^i_{convert}$ needed by the selector and the convertor. The last issue, discussed in Subsection 5.3.3, is the execution time T^i_{decode} needed for actually decoding the 3D content.

5.3.1 Decoder activation

A first important issue for the performance of the decoder is the number N of objects for which the optimisation framework decides to change the geometry.

With the simple geometric criterion, the decoder only needs to adapt the content if the triangle budget changes for a given execution time. The number N of objects depends on the behaviour of the optimisation framework, but is typically low. This issue is discussed in Chapter 6, which presents the complete framework at work.

With view-dependent adaptation, the decoder is activated for every scalable object every time the viewpoint changes. The advantage of view-dependent adaptation is that typically less triangles are needed for a better visual quality, but it also results in a larger number N of objects that need to be adapted every frame. Figure 5.16 shows a measurement of the rendering time on a hardware accelerated platform with a scene once shown at full resolution and once shown with the silhouette at full resolution, back facing triangles removed and triangles in the middle of the object at the lowest resolution. The rendering time for the latter is indeed lower than for the full resolution, but the time that is needed for updating the rendering data structures of all the objects is for the frames with a change of the viewpoint larger than the maximum rendering time. Figure 5.17 is a screen shot from the NVPerfHUD tool [106], which shows measurements of the GPU and CPU time, the number of Drawing calls (DP) and the memory consumption (VID and AGP) on the GPU over the rendered scene. Figure 5.17 confirms that the GPU is actually most of its time waiting (GPU idle curve compared to the rendering time curve) for the CPU every time the viewpoint has changed.

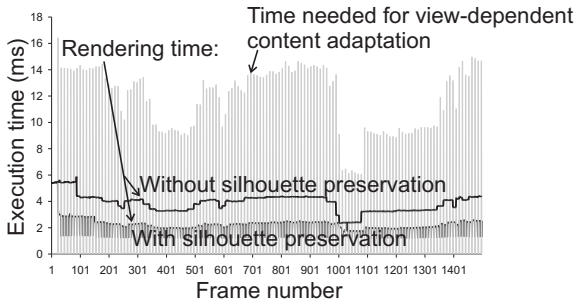


Figure 5.16: Comparison of CPU and GPU time for silhouette preservation: The GPU time decreases with silhouette preservation, but the CPU time is larger than the GPU time and becomes the bottleneck.

With current hardware, as illustrated in the Figures 5.16 and 5.17, rendering triangles is much faster than selecting, converting and communicating them. Therefore, view dependent rendering is only valuable for hardware accelerated systems on very large geometry (e.g. terrains), for which a lot of triangles are eliminated with a minimum effort (e.g. by only checking base mesh triangles). For accelerating the view-dependent adaptation, the work of Bolz et al. [14, 15] and Bischoff et al. [10] can be very important because they investigate the direct evaluation of subdivision surfaces on the GPU instead of first converting them to triangles and then sending them to the GPU.

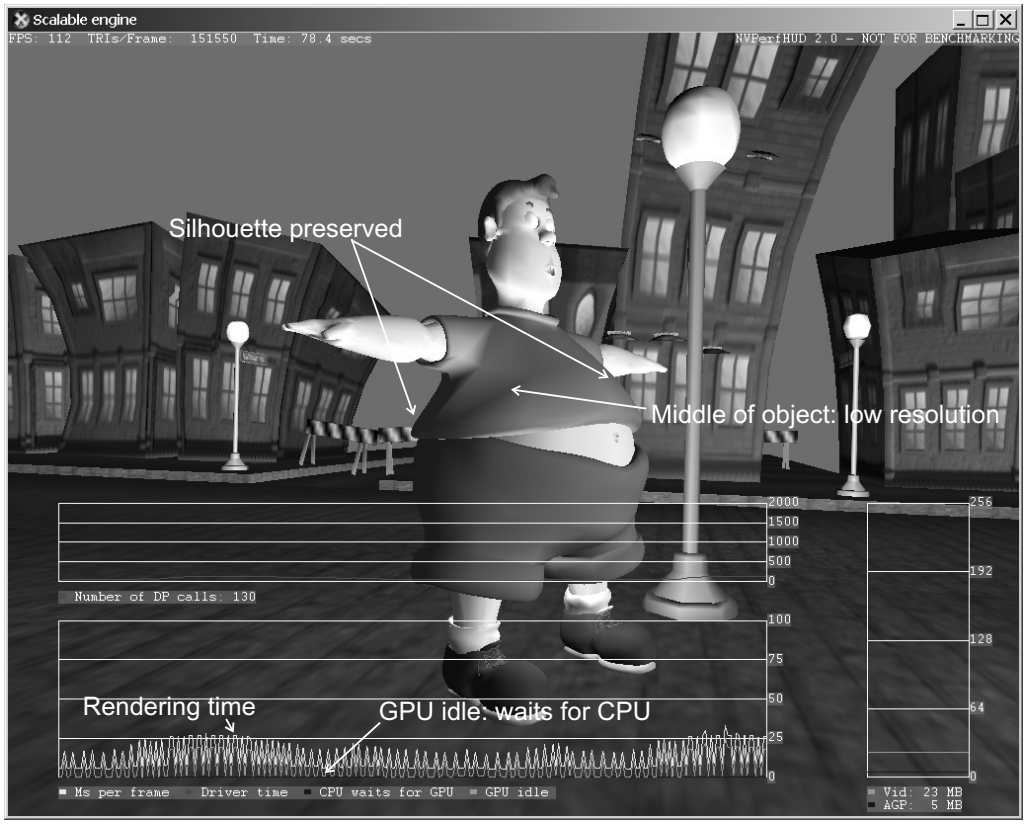


Figure 5.17: GPU most of the time idle for view-dependent rendering: screenshot from NVidia's NVPerfHUD tool.

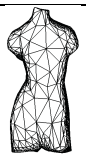
LOD		
1	87%	135%
2	75%	172%
3	42%	142%

Table 5.1: The effect of silhouette preservation on the rendering time of a software renderer, given as a percentage of the rendering time with uniform adaptation.

With software rendering, triangles are rendered much slower than with hardware rendering. In addition, no communication of content is needed because the rendering is executed on the same processor as the content adaptation. Table 5.1 illustrates how silhouette preservation with elimination of back facing triangles increases the rendering performance of a software renderer on a low cost mobile device if enough triangles can be eliminated from the rendering. In the first and second columns, the silhouette is only a small respectively large portion of the mesh and the total rendering time (including adaptation cost) is smaller respectively larger than for rendering the complete mesh at same LOD as the silhouette.

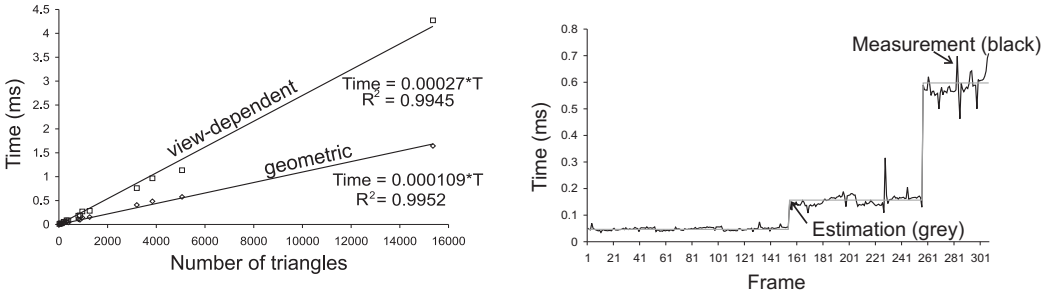
View-dependent adaptation clearly complicates the content adaptation and because hardware accelerators are also developed for mobile devices, we have chosen not to use view dependent adaptation of objects. View-dependent adaptation is however mainly limited by its need to do the adaptation on the CPU, to communicate it to the GPU and to render it on the GPU. As mentioned earlier, an interesting path for future work is to investigate how the platform can be changed to eliminate this bottleneck.

5.3.2 Selector and convertor complexity

The selector and convertor complexity are very important for the frame time because these blocks are invoked every time a content adaptation is needed. This occurs every time the viewpoint changes with view-dependent adaptation or every time the optimisation framework decides to change the number of triangles of the object.

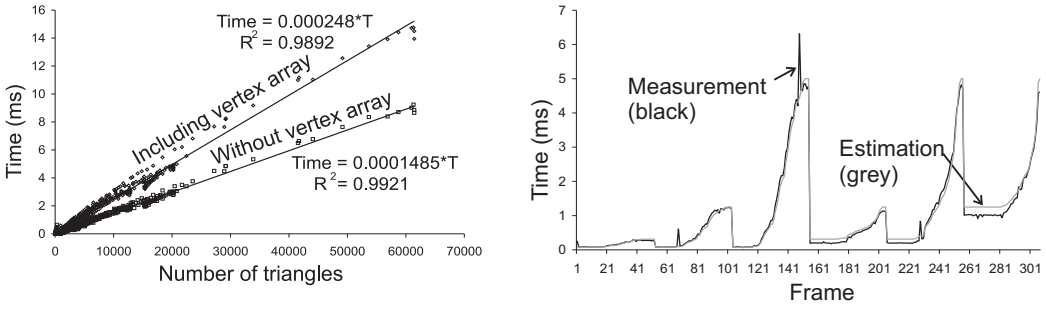
The selector calculates the subdivision code for all triangles on the level of detail for selection ($sLOD$) and its execution time T^i_{select} is linearly dependent on the number of triangles on the level $sLOD$. Figure 5.18 shows the calibration of the selection block and a test of the linear performance model. The work per triangle is equal for all triangles, but is larger for view dependent adaptation than for geometric adaptation. Moreover, for the view-dependent criterion based on distance, much more triangles must be checked by the selector and the total complexity is significantly larger.

The convertor iterates over the triangles of the selection level of detail and uses their subdivision code for building the rendering data structures. The number of triangles in these rendering data structures determines the execution time $T^i_{convert}$ of the convertor. The work per triangle depends on the subdivision code, but is more or less equal for all triangles. Figure 5.19 illustrates the calibration of the convertor block with and without the construction of the vertex array (see Subsection 5.2.4). Figure 5.19 also shows a test of the linear performance model.



(a) Model calibration (b) Measurement versus Model

Figure 5.18: The execution time of the selector: a) calibration and b) model test



(a) Model calibration (b) Measurement versus Model

Figure 5.19: The execution time of the convertor: a) calibration and b) model test

5.3.3 Decoding complexity

The decoding complexity has an impact on the time that the user needs to wait to start the session, but it does not influence the frame rate for applications that decode all content before the interactive session is started. The time T^i_{decode} in Equation 5.2 is then equal to 0. Applications that stream 3D content invoke the decoder every time an object arrives at the terminal. For these applications, the decoding time depends on the number N of objects (Equation 5.2) that need to be decoded for the processed frame. The decoding time for a single object is further elaborated in this subsection.

The block diagram of the standard MPEG-4 WSS decoder is given in Figure 5.20 [47], which also shows how much time the decoder parts need. The execution time distribution in Figure 5.20 is the average distribution of the decoding time for 20 different objects measured on a PC with a Pentium IV processor.

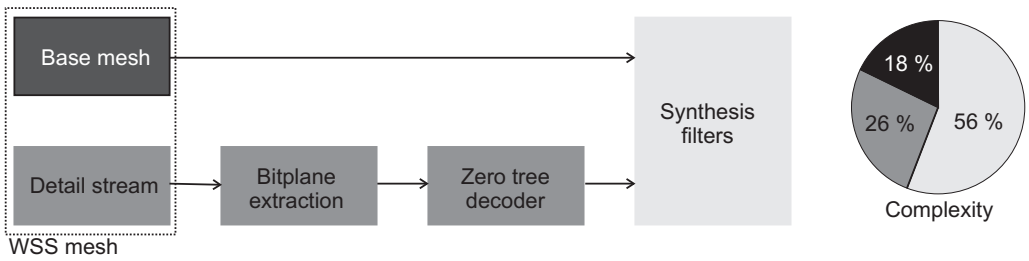


Figure 5.20: Block diagram of the MPEG-4 WSS decoder with associated complexity.

Reading the base mesh takes more or less 18% of the decoding time, including disk accesses for reading the content. The time needed for reading the base mesh is uncontrollable since the base mesh is the minimum set of data that is needed for showing the object. It is thus important to minimise its number of triangles while encoding the 3D content.

Decoding the detail information takes more or less 26%, including disk accesses for reading the detail information. This number was measured for a maximum amount of detail information and drops rapidly if detail information is skipped. This enables the control of the execution time. However, the execution time heavily depends on the processed bits and is hard to predict. The control is best done with a measurement of the execution time instead of a performance model.

Calculating the subdivision schemes takes the major part of the decoding time (more or less 56%). Their execution time depends on the number of decoded triangles and on the used filter. Figure 5.21 shows a linear relationship between the number of triangles and the execution time. The execution time of the synthesis filters is easily predicted and is easily controlled by limiting the number of decoded subdivision levels.

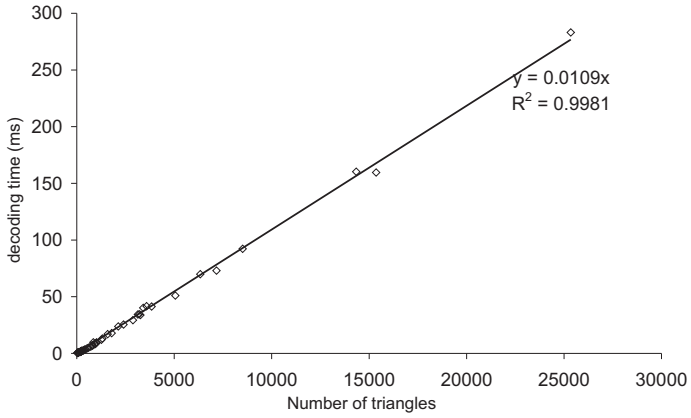


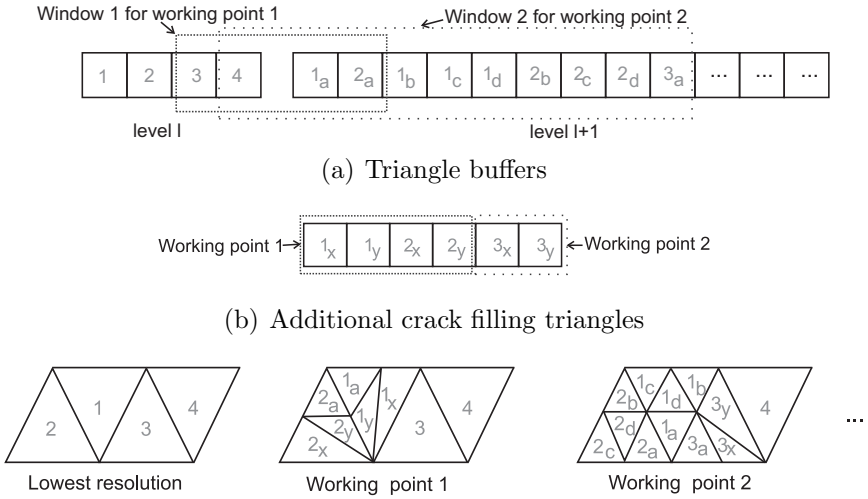
Figure 5.21: The execution time of the midpoint subdivision mask as a function of the number of triangles (measured on Pentium IV)

5.4 Cancelling the adaptation overhead

The selector and convertor need CPU processing power for extracting the geometry from the multi-resolution structure and must then communicate the new geometry over a specialised bus to the graphics processor. This section discusses how the CPU overhead can be traded for memory space and a worse Post T&L cache efficiency. This method relies on a fixed order for the refinement of triangles (decided by the optimisation framework) and can therefore only be applied on the selection with the geometric criteria for which the order of replacement is not determined by the viewpoint.

One array for the triangles, as shown in Figure 5.22, is allocated: it contains the triangles of two subsequent subdivision levels l and $l+1$, with the triangles of level l first. The triangles of level l are all refined to triangles of level $l+1$ before one of the triangles of level $l+1$ is further refined with a similar couple of buffers. If the resolution is now increased by increasing the threshold parameter h_t of the selector, then some of the triangles on level l disappear, being replaced by the corresponding triangles of the level $l+1$. The triangles that disappear from level l and appear on level $l+1$ are added to the first free position of respectively the array of level l and level $l+1$.

In Figure 5.22, the lowest resolution is rendered by starting the rendering at triangle 1 and ending it after triangle 4. The triangles 1 and 2 of Figure 5.22 that come first in the array l are for the first value of the threshold h_t replaced by their corner triangles 1_a and 2_a that then also come first in the level $l+1$. Uniform adaptation normally splits the triangles into 4 child triangles. This is however not the case with non-uniform adaptation and extra triangles are needed to fill the cracks. These triangles (the triangles x and y in figure 5.22) are created as needed for a particular resolution and are not shared by the different resolutions. Therefore, they are stored



(a) Triangle buffers

(b) Additional crack filling triangles

(c) Meshes for subsequent working points on distortion plot

Figure 5.22: Two subsequent subdivision levels l and $l+1$ in the same array enable to eliminate the CPU adaptation overhead: a) the array containing the subdivision levels, b) the array for the additional crack filling triangles and c) the rendered geometry

in a separate array, as shown in Figure 5.22. The first working point is then rendered by starting the rendering at the position of triangle 3 and to stop after triangle 2_a. The crack filling triangles are rendered by starting the rendering at the triangle 1_x and ending it after triangle 2_y.

For the second value of the threshold parameter h_t, also triangle 3 is replaced by one of its corner triangles 3_a and the rest of the child triangles of triangle 1 and 2 are also added. The crack filling triangles for the triangles 1 and 2 are no longer needed. The crack in triangle 3 is now filled with the triangles 3_x and 3_y. The rendering is started at triangle 4 from level l and ended after triangle 3_a from level $l+1$ for the regular triangles and is started at triangle 3_x and ended after triangle 3_y for the crack filling triangles.

The vertex array is constructed with a similar procedure. We currently only apply the method on interpolating subdivision surfaces because they only create new vertices without repositioning the old ones. Therefore, the vertex array starts with all the vertices of the base mesh and vertices are added to this vertex array in the order that they are needed by the triangles.

The main advantage of this new approach is the elimination of the computational overhead because the control is limited to the selection of the array window: i.e. the start and stop positions for the rendering. This is illustrated in Figure 5.23, which shows the total frame time - CPU adaptation and GPU rendering - for a PC with a Pentium IV CPU and Geforce FX 5900 GPU. The proposed method reduces this

total frame time compared to the traditional methods when the complexity of the geometry is changed (region a in Figure 5.23). The noise on the total execution time of the proposed method is also reduced a lot compared with the noise on the execution time of the traditional methods.

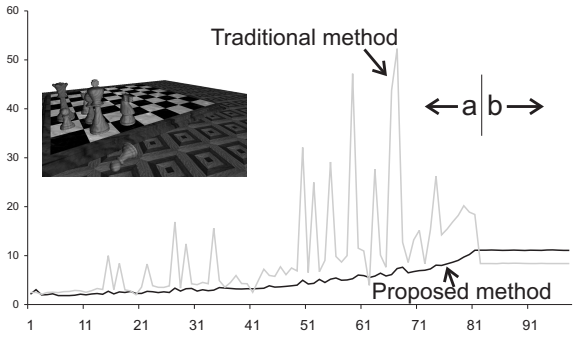
The major disadvantage of this method is the fixed order of the triangles, which can be a problem for the Post T&L cache efficiency on a graphics chip (see Subsection 3.3.2 for more information in Post T&L caches). Simulations of a post T&L FIFO cache gave indications that the efficiency of the Post T&L cache could decrease with a factor up to 33%. The higher post T&L cache miss rate increases the number of vertices that need to be processed and thus also the GPU time if the vertex processing units are the bottleneck in the rendering pipeline (Other parts in the pipeline are the triangle and pixel processing units). The effect of the decreasing post T&L-cache efficiency is shown in the region b of Figure 5.23, where the geometry is rendered at full resolution to make the vertex units the bottleneck. The rendering time is slightly higher (10% on average) for the proposed method compared to the traditional methods. It is thus beneficial to use the traditional methods in applications that do not need on-the-fly content adaptation.

A small disadvantage of this approach is the extra memory consumption. If all levels are decoded and stored in the arrays of Figure 5.22, then the memory that is needed by the system is more or less twice the memory needed for the highest resolution mesh.

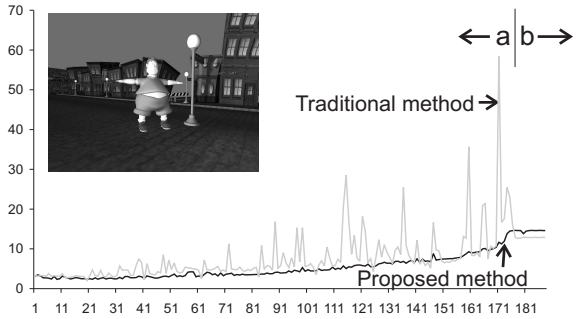
5.5 Conclusions

This chapter presented several methods that can be used for changing the number of triangles of a triangular mesh. The last few years were very fruitful for this field, which lead to many different techniques that try to generate the best possible 3D geometry with the lowest number of triangles. One of the techniques that deliver a very good compression of scalable content was actually standardised in MPEG-4, which lead to the MPEG-4 Wavelet Subdivision Surfaces. Because of its nice compression, we have selected this technique to be used in our framework.

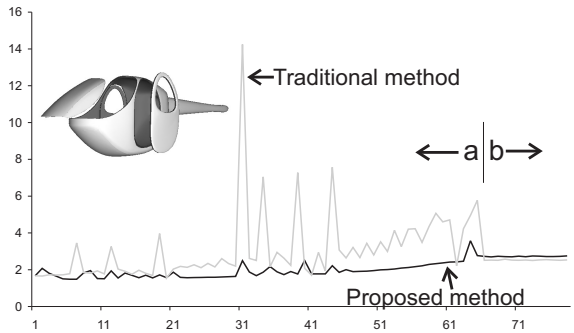
MPEG-4 Wavelet Subdivision Surfaces is typically a form of discrete LOD, which only allows a coarse regulation of the number of triangles. Therefore, we have extended the standard MPEG-4 WSS decoder to support continuous LOD. We have followed the reasoning that is typical for continuous LOD to decode the geometry into a data-structure that is then used online for easily adapting the content. Important regions in the mesh can be determined according to geometric criteria intrinsic to the target surface or according to view-dependent criteria that take the user navigation into account (distance and silhouette). Both types of content adaptation have been added to the standard MPEG-4 decoder.



(a) Chess scene



(b) City scene



(c) Helicopter scene

Figure 5.23: Comparison of the total frame time for the proposed versus traditional method. Region a on the left: gain because of overhead elimination. Region b on the right: pain because of worse Post T&L cache efficiency

The selector and convertor complexity are very important for the frame time because these blocks are invoked every time a content adaptation is needed. This is every time the viewpoint changes with view-dependent adaptation or every time the optimisation framework decides that the number of triangles for the object must be changed for the geometric criteria. Because of the high complexity, view-dependent adaptation is only useful with very large meshes for which many triangles can be easily eliminated from the rendering.

The selector and convertor need CPU processing power for extracting the geometry from the multi-resolution structure and must then communicate the new geometry over a specialised bus to the graphics processor. This overhead can be traded for memory space with a method that relies on a fixed order of the triangles. The major disadvantage of this method is the fixed order of the triangles which makes it impossible to optimise the mesh for a maximum Post T&L cache efficiency.

Chapter 6

Optimisation over Successive Frames

This chapter combines all building blocks presented in the previous chapters into the complete framework. Chapter 5 (on multi-resolution mesh representation) has selected a number of methods for the content adaptation. This chapter illustrates the use of these methods for different behaviours of the gradient descent combination (Chapter 2) over successive frames.

First, Section 6.1 presents the complete framework. Sections 6.2 and 6.3 discuss the control of the execution time with non-uniform (continuous LOD) and uniform (discrete LOD) subdivision surfaces. These sections also present the results for the controlled execution time. Section 6.4 discusses the stability of this system and the last section presents the conclusions.

The content that is used for showing the results of the framework and the uncontrolled execution time is introduced in Appendix A. Figure 6.1 presents a legend for the figures in this chapter that illustrates the control of the execution time:

1. The target execution time is a horizontal black line near the centre of the figure;

2. The measurement of the rendering time is a gray continuous line near the target execution time;

3. The model of the rendering time is a black continuous line close to the measurement of the rendering time;

4. The measurement of the execution time of the optimisation process of a single frame is the black line closest to the X-axis. This line is not continuous and has a lot of peaks because the optimisation is only done when the viewpoint actually changes;

5. The measurement of the execution time needed for the content adaptation (selection and conversion) of a single frame is the gray line closest to the X-axis. This line is not continuous because the content adaptation is only executed as needed.

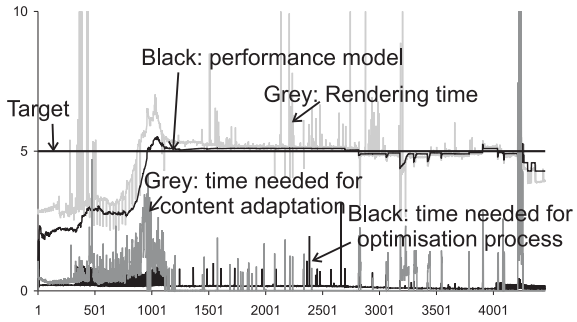


Figure 6.1: Legend for the figures of this and the next subsections

6.1 Putting all the pieces together

Figure 6.2 illustrates how the several pieces, presented in the previous chapters, fit together.

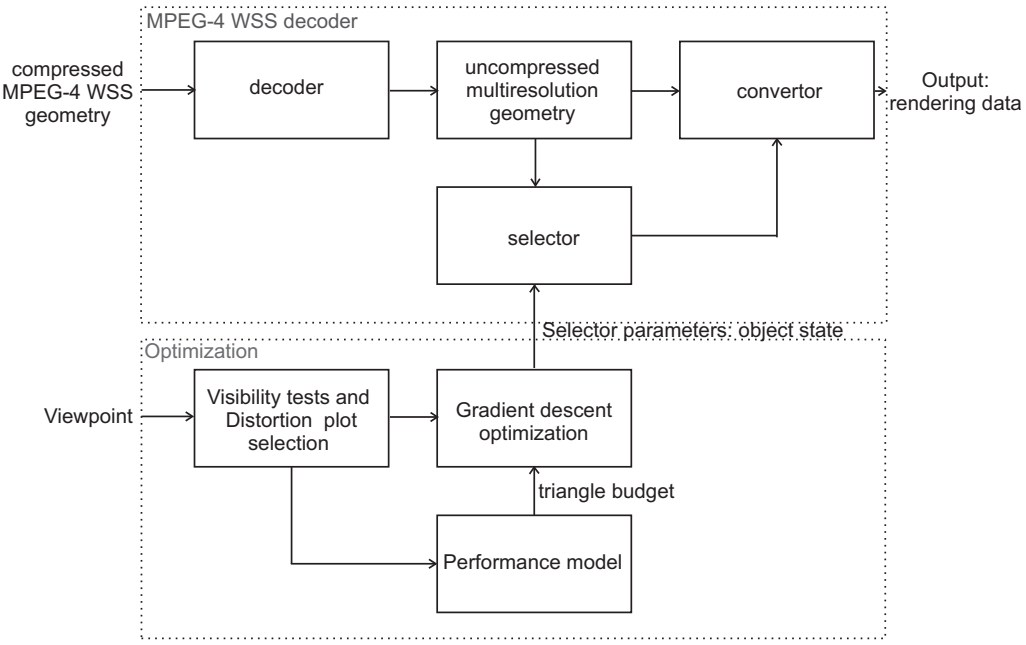


Figure 6.2: The decoding and optimisation of MPEG-4 WSS geometry

The optimisation starts with a traditional view frustum and occlusion culling visibility test, which extract the objects from the scene description that are potentially visible from the user's viewpoint. The screen space area of the selected objects is then derived from their visibility spheres and is used by the performance model to calculate the triangle budget.

The parameter settings of the MPEG-4 WSS decoder that match the triangle budget are for every object in the scene calculated with the gradient descent combination of the distortion plots. Every working point on an object distortion plot, derived in preprocessing, corresponds with a decoder setting, called object state (Section 5.2.3): $(sLOD, LOD, h_t)$ for non-uniform adaptation and LOD for uniform adaptation. The object state controls the number of triangles and is thus needed for every object involved in the gradient descent combination. The gradient descent combination starts at the distortion plot working points having the smallest cost and largest error and it initialises the objects with the object states attached to these working points. The gradient descent combination then calculates the best trade-off between quality and cost by changing the working point for the object with the highest decrease of error for the lowest increase of cost. The object state attached to this new working point on the object distortion plot then replaces the old object state of the selected object.

For performance reasons, we have chosen to fully decode the MPEG-4 content into the multi-resolution structure before the actual rendering loop is started. Because view-dependent adaptation has a large overhead (as explained in Chapter 5), we have also chosen to only use uniform or non-uniform adaptation based on the static geometry criterion. The back-channel and non-uniform decoding arrows and the viewpoint input parameter are therefore removed from the decoder in Figure 5.5, resulting in the MPEG-4 WSS decoder of Figure 6.2.

The results of the optimisation can be summarised in a global online distortion plot as presented in Figure 6.3, which shows two distortion plots for two subsequent frames. The points a and b on these curves are the working points for which the triangle cost meets the budget decided by the performance model [1]. A straightforward approach for the content adaptation is then to change the resolution of all involved objects directly to the settings attached to the new working point.

The results for this instantaneous adaptation are shown in Figures 6.4, 6.5 and 6.6. Although the control of the rendering time is good, the major disadvantage of this approach is the larger overhead compared to other approaches discussed further on. This is especially a problem on the hardware accelerated platform because the adaptation is executed on the CPU and the results must be communicated to a very fast GPU. In addition, the error is also changed abruptly, which can reduce the quality experienced by the user.

[1] This triangle budget differs because the number of triangles is not the only parameter for the execution time

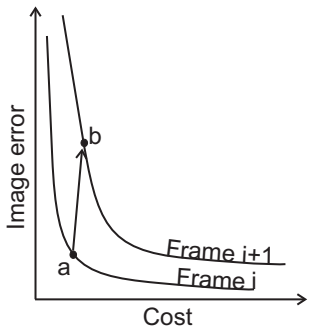
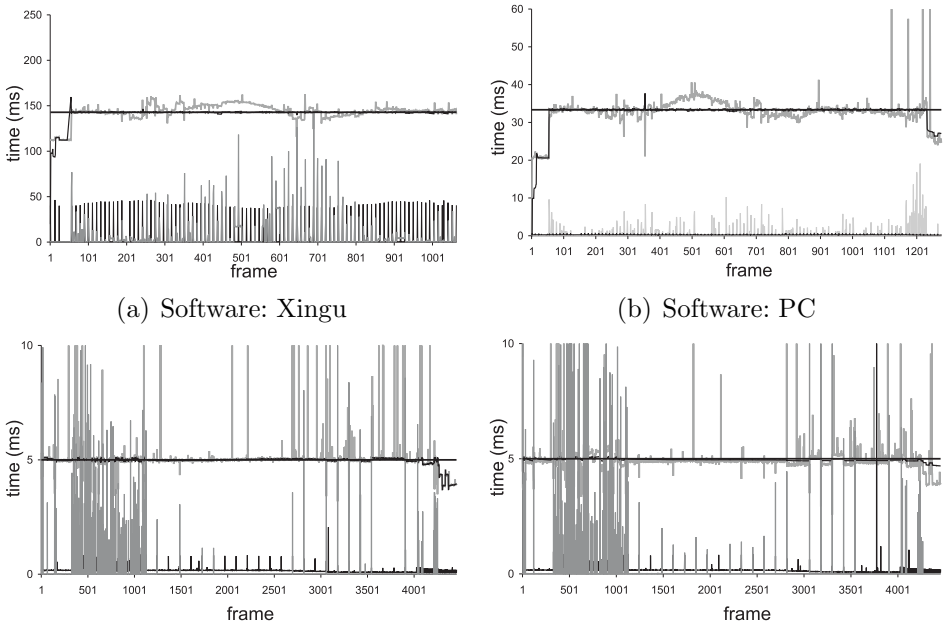


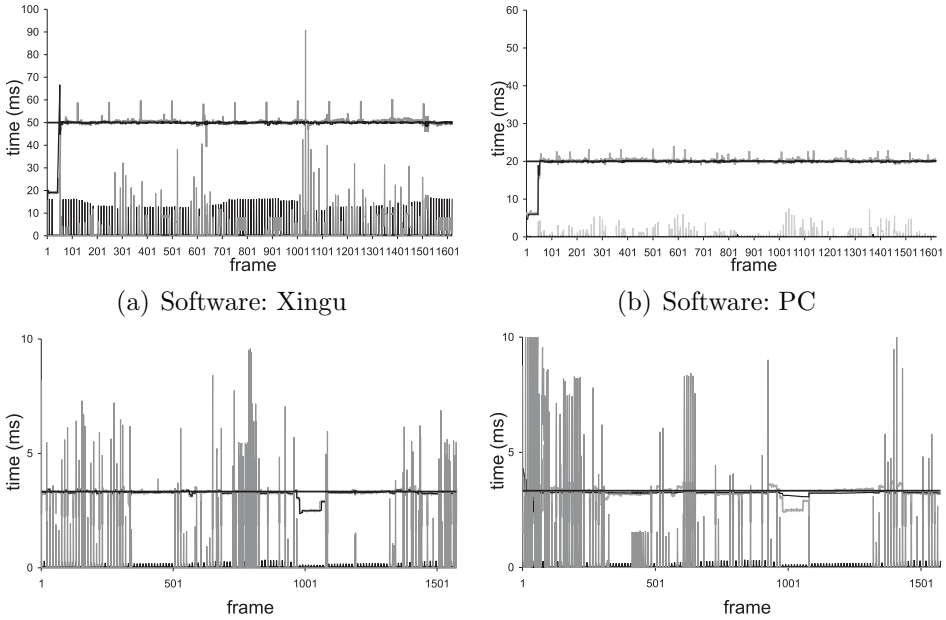
Figure 6.3: Instantaneous control with non-uniform subdivision surfaces.



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

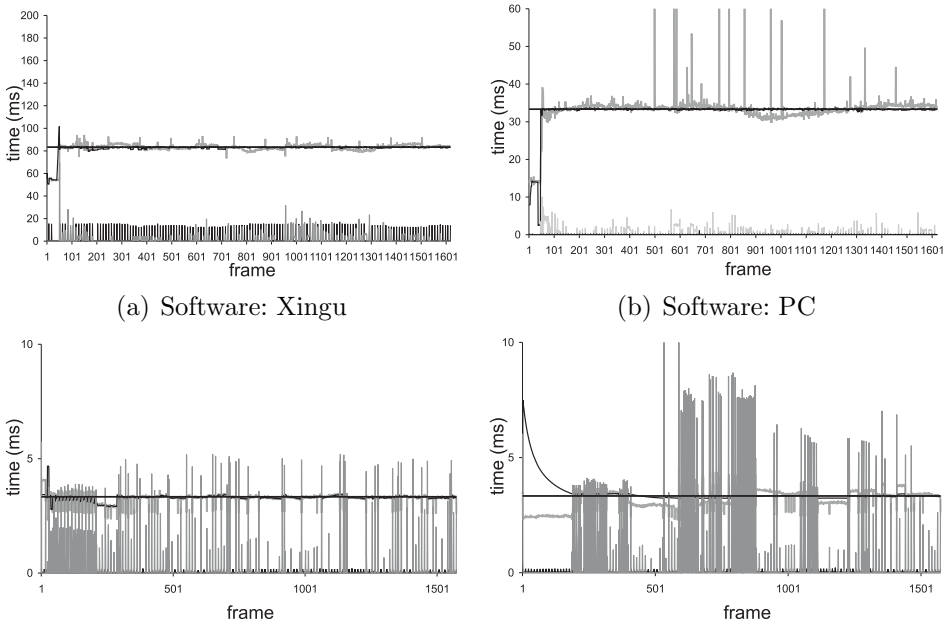
Figure 6.4: Execution time for instantaneous adaptation of the city scene.



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

Figure 6.5: Execution time for instantaneous adaptation of the Potato scene.



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

Figure 6.6: Execution time for instantaneous adaptation of the chess scene.

The execution time needed for the optimisation is for the PC systems running software and hardware accelerated rendering negligible with respect to the rendering time. However, the optimisation time for Xingu is much larger compared to the PC systems. This illustrates the effect of executing floating point operations on a device without floating point unit. The optimisation time on Xingu is relatively seen comparable with the optimisation time of the other systems when the optimisation is executed in fixed point arithmetic. Therefore, we have also developed a fixed point software renderer for the embedded system.

The limited accuracy of the summation performance model for the hardware accelerated platform results in a higher content adaptation overhead (see Figure 6.6). Indeed, a performance model with a limited accuracy relies on the online correction of the errors. This online correction however continuously changes the triangle processing times, which results in changing triangle budgets for identical content and viewing conditions. The overshoot during the initialisation of the performance model of Figure 6.6 is an error that is first introduced (because of the small amount of online data) and then slowly corrected by the online recursive least squares algorithm that is used for the online calibration of the performance model.

6.2 Scenario 1: Continuous LOD with non-uniform subdivision surfaces

The instantaneous adaptation, which was proposed in the previous section, chooses the working point on the distortion plot for frame $i+1$ independent of frame i. There is however typically a large amount of coherence between successive frames, which can be exploited to reduce the adaptation overhead and the abrupt content changes. As illustrated in Figure 6.7, the framework first selects the nearest [2] point A' for frame $i+1$ and then spreads the adaptation over several frames until point B is reached (at frame $i+x$).

The best choice to minimise the adaptation overhead is to choose very small steps, e.g. jump every frame to an adjacent working point on the distortion plot. However, the smaller the steps are chosen, the longer it takes to reach the final working point and the larger the probability is that the content and its distortion plot completely change before the actual working point B is reached. Figure 6.8 shows a possible result: an execution time that varies around the target execution time. The peak to peak variation of the execution time in Figure 6.8 is 18 ms, which is more or less 50% of the target execution time.

Currently, the step size is first initialised with the minimum step size and the mathematical variance of the execution time is measured. If the variance becomes too

[2] Nearest is here defined as the point with the least content changes.

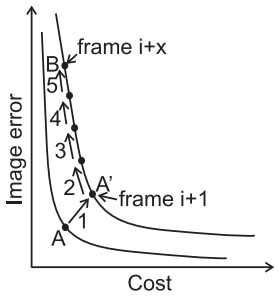


Figure 6.7: The change of working point A to working point B is gradually done in 5 steps (frames).

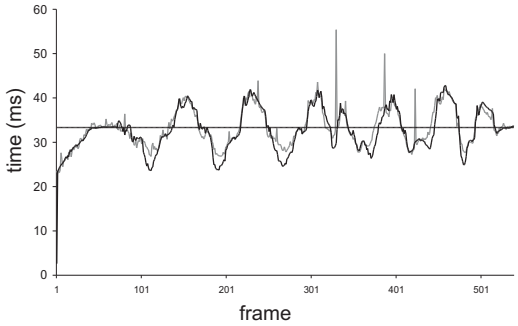


Figure 6.8: The execution time varies around the target because gradual control does not react immediately to content changes.

large (e.g. larger than 20%), the step size is increased so that the control of the execution time improves.

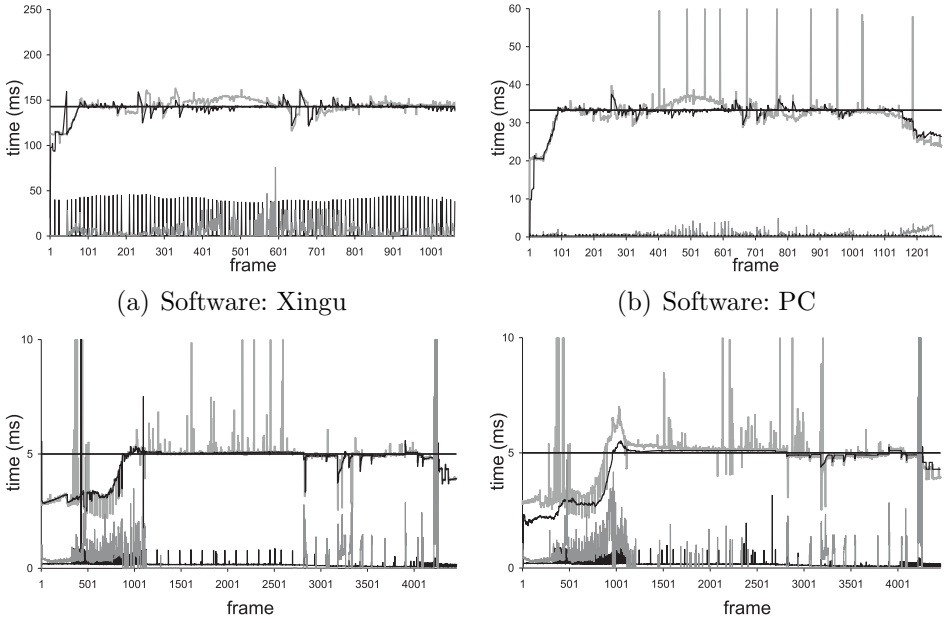
The control of the execution time with non-uniform subdivision surfaces is illustrated in the Figures 6.9, 6.10 and 6.11. These plots show a well controlled execution time for the three different scenes with slowly varying online distortion plots. For comparison reasons, the same animation sequences are used for these figures and the figures that illustrate the instantaneous control in the previous section. For this slowly varying content, the adaptation overhead in Figures 6.4, 6.5 and 6.6 is already significantly larger than the adaptation overhead in Figures 6.9, 6.10 and 6.11. The error on the control of the execution time because of the gradual non-uniform subdivision is visualised with the small peaks in the black curve of the performance model (especially visible in the plots for the software rendered city scene). The real execution time (grey curve) however also deviates from the target execution time because of the performance model error. In a lot of cases, the error made because the content adaptation was gradual is actually indistinguishable of the error made because of the performance model. For the city rendered on Xingu for instance, the average error on the performance model is 3.31 % and the maximum error is 23 %. The average difference between the measured and the target execution time is equal to 3.67 % and the maximum difference is equal to 19 %. For the initialisation, the gradual control starts from the working point with the lowest quality and gradually increases the quality over several frames. Because no history is yet taken into account, it can take some time before the target execution time is reached.

Note that this type of progressive Level Of Detail can also be done with the progressive meshes of Hoppe et al. [64], [56].

6.3 Scenario 2: Discrete LOD with Uniform Subdivision

Our first scenario uses non-uniform subdivision surfaces to change the resolution of the content. A criterion based on geometric information is then used for a local and small instead of complete refinement of the geometry (Chapter 5). Non-uniform subdivision surfaces allow a fine-grained control of the resolution of the geometry, resulting in small variations of the visual quality. The disadvantage of non-uniform subdivision surfaces is the extra overhead because of the computation of the adapted content, i.e. the criterion calculations and the local refinements that need to be extracted from the multi-resolution data structure for the rendering data structure (Chapter 5).

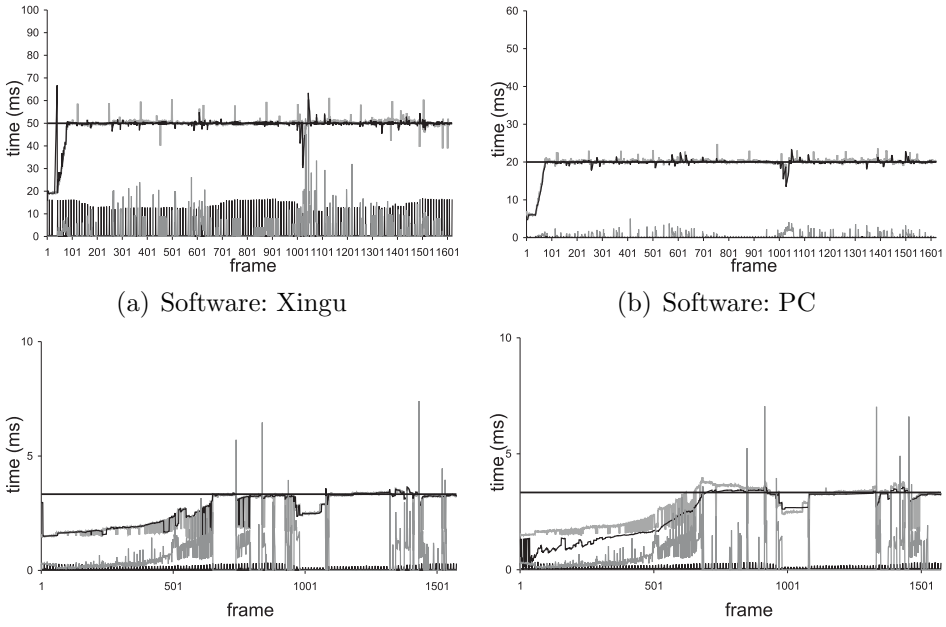
Non-uniform subdivision surfaces are suited very well for the scenario with small content adaptations in slowly varying scenes. In this scenario, the adaptation of the



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

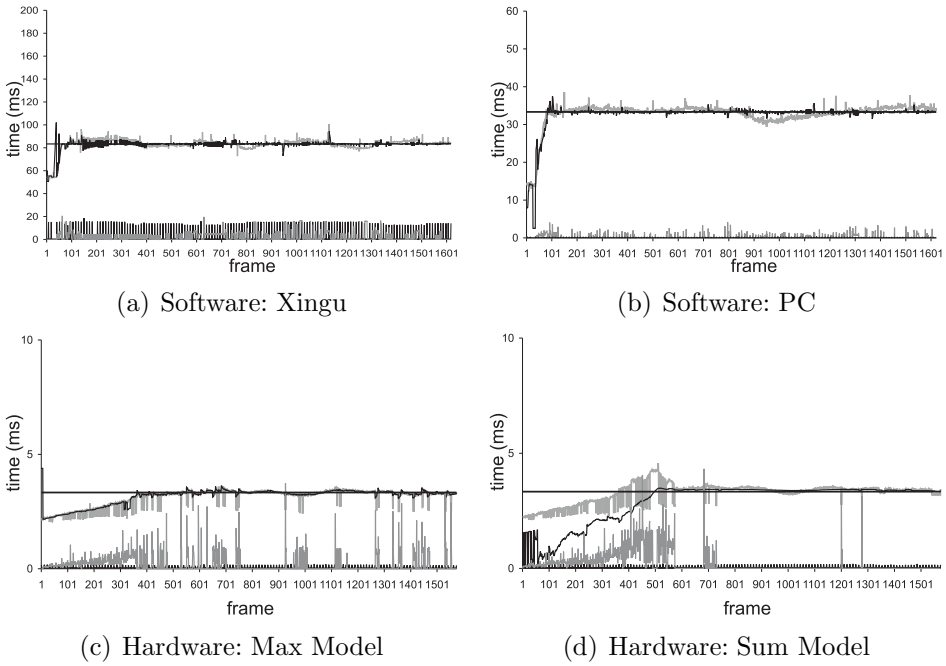
Figure 6.9: Execution time for gradual adaptation of the City scene.



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

Figure 6.10: Execution time for gradual adaptation of the Mr. Potato scene.



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

Figure 6.11: Execution time for gradual adaptation of the Chess scene.

content is so limited that its additional overhead can be neglected with respect to the rendering time. However, if the content starts changing more abruptly, the step sizes and adaptation overhead (modelled with Equation 5.2) increase.

A second scenario uses uniform subdivision, which changes the resolution of the geometry by selecting a complete and precomputed subdivision level. The advantage of uniform subdivision is its simplicity. Switching from one subdivision level to another is only a matter of selecting the correct buffers that contain the triangle info once the geometry is decoded. The cost of this approach is the extra memory size of 33% with respect to the geometry at the highest resolution, which is the same overhead as with mipmapped textures [80].

Because a subdivision operation multiplies the number of triangles by four for every subdivision level, uniform subdivision results in very few configurations on the distortion plots of the single objects (typically between 4 and 6). The slow reaction of scenario 1 however occurs when the content adaptation visits too many working points and the step size needs to be increased to skip working points. For this situation, the online distortion plot for uniform subdivision surfaces will contain a sufficient amount of working points. Figure 6.12 shows an example of such an online distortion plot for the initial viewpoint in the Chess scene, presented in Appendix A.

The reactive loop that controls the step size for the first scenario also measures or predicts the time needed for the content adaptation and changes to the discrete

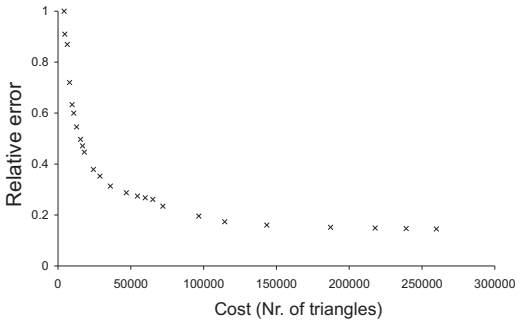
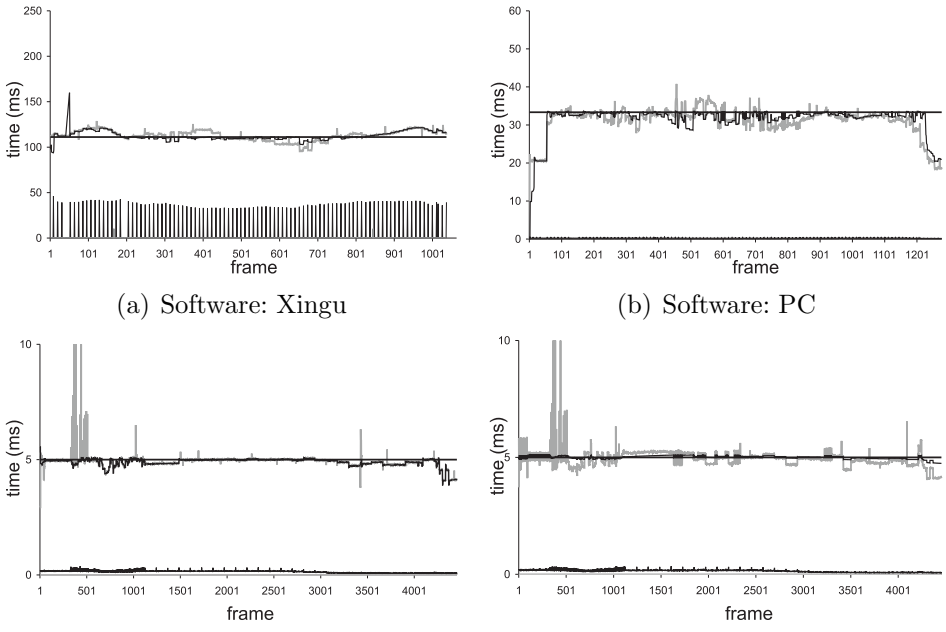


Figure 6.12: Distortion plot for uniform subdivision in the city scene.

LOD scenario if this adaptation time becomes too large.

Figures 6.13, 6.14 and 6.15 show the results of the execution time for the different scenes controlled with uniform adaptation. Although it is not always possible to guarantee a constant target execution time, the accuracy of the control is for the presented scenes satisfying. The Mr. Potato scene has the lowest number of working points on its online distortion plot and has the lowest accuracy for the control with uniform adaptation. Because the content adaptation is limited to selecting buffers, the content adaptation time does not exist and is not presented.



(a) Software: Xingu

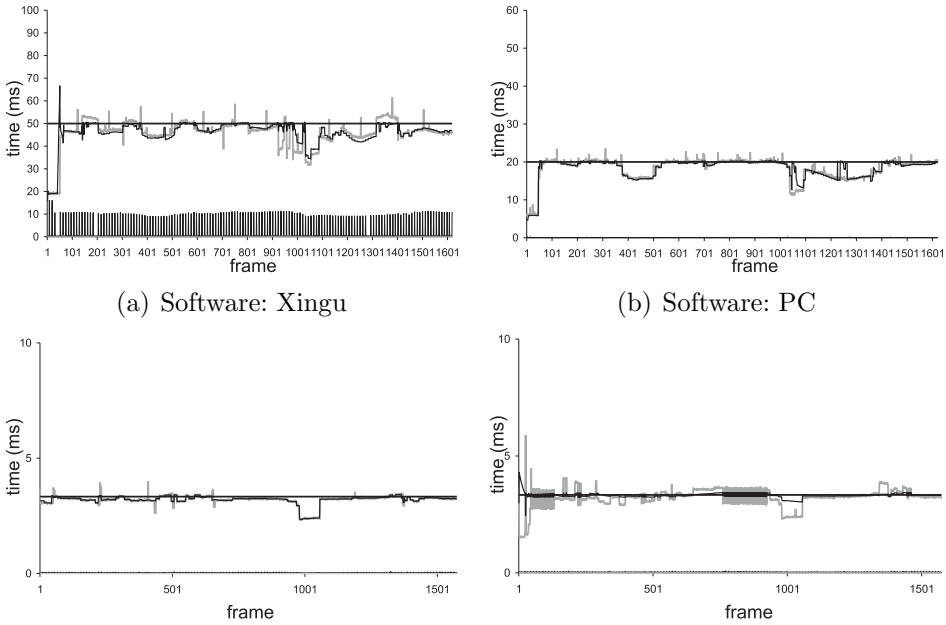
(b) Software: PC

(c) Hardware: Max Model

(d) Hardware: Sum Model

Figure 6.13: Execution time for uniform adaptation of the City scene.

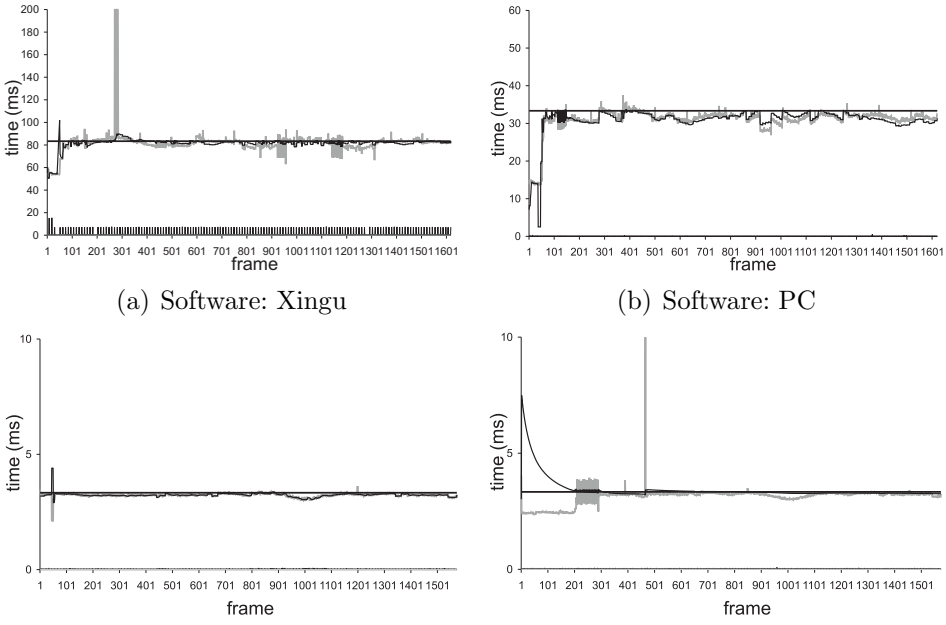
Note that instead of uniform subdivision surfaces also the discrete LOD techniques of Clark [34] can be used instead of uniform subdivision surfaces for this type of



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

Figure 6.14: Execution time for uniform adaptation of the Mr. Potato scene.



(a) Software: Xingu (b) Software: PC

(c) Hardware: Max Model (d) Hardware: Sum Model

Figure 6.15: Execution time for uniform adaptation of the Chess scene.

content adaptation.

6.4 Stability and sensitivity of the system

We use the definition of stability from the book of Ioannou and Sun [71], who calls a system stable if it has a bounded output for a bounded input. Although stability guarantees that the output execution time is bounded for any bounded input, quality requirements put restrictions on the variation of the execution time. The latter is called the sensitivity of the system.

A first block in the framework (see Figure 6.2) that is important for the stability is the adaptive performance model that feeds the measured execution time back into the model to correct its coefficients. The stability of the system is guaranteed by the convergence of the recursive least square with forgetting algorithm discussed in Chapter 3, Section 3.2.4. The recursive least square algorithm uses an average value for the correction and it therefore also has a low sensitivity to measurement noise.

A second block in the framework that is important for the stability is the gradient descent optimisation. Gradient descent methods have a linear convergence (proven in the book of Boyd et al. [21]) to the minimum value, which is in our case actually known in advance as the working point with all objects shown at full resolution. The issue for our framework is to guarantee that the subsequent iterations of the gradient descent method are optimal solutions with a maximum quality and minimum cost. This is guaranteed by the construction of the algorithm that changes the working points with a maximum decrease of the error for a minimum increase of the cost (see Chapter 2, Section 2.4).

The choice between gradual or instantaneous adaptation is very important for the sensitivity of the system to changing inputs. Gradual adaptation of the content introduces a delay in the decisions of the optimisation algorithm and therefore makes the system sensitive to input changes (e.g. change of viewpoint). This sensitivity can for instance lead to overshoot if objects are added/removed to the already visualised content and a constantly varying execution time if the content changes so fast that the optimisation never reaches its final destination (e.g. Figure 6.8). Changing the step size for the gradual adaptation (see Section 6.2) allows to switch between gradual and instantaneous adaptation and therefore to control the sensitivity of the framework to changing inputs. Sensitivity to measurement noise is also limited here by using average values to calculate the step size.

6.5 Conclusions

This chapter combined all the building blocks into the complete framework and illustrated the usability of the different content adaptation methods for different

behaviours of the gradient descent combination. Different methods were proposed for specific behaviours of the optimisation algorithm.

In a first scenario, non-uniform subdivision surfaces are suited very well for small content adaptations in slowly varying scenes and for gradual content adaptations in scenes with a few abrupt content changes. The adaptation of the content is then so limited that the additional overhead can be neglected with respect to the rendering time. An extra advantage of the gradual control is the gradual change of the visual quality, which improves the experience of the user.

If the number of abrupt content changes is too high, the second scenario must be used to control the resolution of the content. The major disadvantage of instantaneous control with non-uniform subdivision surfaces is the larger overhead, which is especially a problem on the hardware accelerated platform. Because the content must change completely in one frame, quality changes are also significant for this type of content adaptation. The quality change was however the largest drawback of uniform adaptation. Because of its simplicity, it is better to use uniform subdivision if the content changes so drastically that gradual content adaptation is not satisfactory instead of using instantaneous non-uniform subdivision surfaces.

Chapter 7

Extension of the framework: controlling energy consumption

Today, mobile terminals such as PDAs and mobile phones have reached a sufficiently high-level of performance as to support simple 3D Graphics applications. Currently, most research in the field of 3D rendering on mobile devices focuses on developing embedded 3D graphics architectures, which deliver maximum performance for minimum power consumption. However, a lot of work has still to be done to enable the rendering of high quality content on mobile devices. The challenge from the application point of view is to generate the best possible image within the performance constraints of the used terminal. The algorithms that solve this optimisation problem were however developed for terminals connected to a power supply. Therefore, these algorithms only trade image quality for execution time and totally ignore the energy consumption of the terminal.

Because of the observations that the change of image quality can be as disturbing to the user as a low image quality and that energy is a scarce resource on mobile terminals, we propose to extend our optimisation algorithm to provide a constant and user requested quality and to exploit the resulting dynamic workload to minimise the energy consumption for a given execution time deadline. Figure 7.1 illustrates the difference between our proposal and existing optimisation techniques. Traditional optimisation algorithms increase the visual quality (decrease error in Figure 7.1) until the complete execution time budget (represented as a constant triangle cost in Figure 7.1) is consumed with a maximum energy consumption. This sometimes results in a very small increase of the visual quality for a large increase of the execution time. Therefore, it is better to use an optimisation algorithm that tries to provide a constant and user requested error (quality) and to exploit the lower workload to minimise the energy consumption for a given execution time deadline.

The minimisation of the energy consumption is in this chapter done by changing the resolution of the geometry, i.e. controlling the number of triangles. A path for

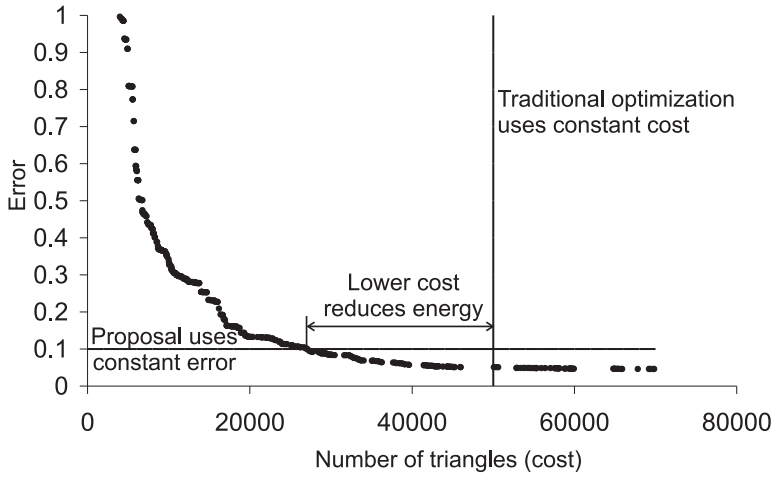


Figure 7.1: The difference between our proposal and existing optimisation techniques.

future work is to explore the minimisation of the energy consumption by changing the other input parameters, e.g. the appearance model (texture) or rendering state.

The rest of this chapter is organised as follows. Section 7.1 discusses related work. The architecture of the simulated platform is presented in Section 7.2. The extension of our optimisation algorithm and the results are then presented in Section 7.3 and finally, the conclusions are presented in Section 7.4.

7.1 Related work

3D rendering chips for mobile devices have gained the interest of most of the 3D Graphics chip manufactures (ATI [6], NVidia [103], ARM [4], Bitboys [110], etc.). The ATI IMAGEON 2300 is a completely fixed rendering pipeline, i.e. a fixed hardware implementation for the vertex, triangle and pixel processors. The NVidia GoForce 3D core also implements a full 3D rendering pipeline, but instead of a complete fixed implementation, the pixel processor is programmable. Both ATI and NVidia did not yet publish any power and performance numbers. The ARM MBX 3D Graphics accelerator uses an SIMD floating point coprocessor for the vertex geometry processor and implements a fixed triangle and pixel processor. ARM uses the tile based PowerVR technology which allows having the depth buffer memory on-chip. Performance figures for the ARM MBX 3D core are given in Figure 7.1 [4]. The HR-S 0.13 μm delivers 2.5 Mtriangles/s and 480 Mpixels/s and consumes 156 mW.

Excellent academic work on graphics hardware for mobiles is published by Woo et al. [164]. This state of the art mobile 3D accelerator delivers 70k vertices/s and 66M

MBX	mW/MHz	MHz	Mtriangles/s	Mpixels/s
HR-S 0.18 μm	2.6	80	2	320
HR-S 0.13 μm	1.3	120	2.5	480
R-S 0.18 μm	1.5	80	1	160
R-S 0.13 μm	0.7	120	1.5	240

Figure 7.2: The ARM performance figures

pixels/s and consumes 210 mW (0.16 μm DRAM process). This chip also contains a programmable power optimiser to change the performance and energy consumption at run-time. The vertex processor in this chip consumes 26% and 37% of the energy for respectively textured and non-textured mode. The pixel processor consumes more or less 35% of the energy in textured mode (including the texture memory) and less than 20% in non-textured mode. The on-chip L2 memories (depth buffer, texture memory and frame buffer) consume 33% of the energy. This chip was further optimised by Sohn et al [137] who replaced the software processing of the vertices on an ARM 9 with a programmable vertex processor (fixed point) that consumes 115 mW for a performance of 7.2 Mvertices/s. Simulators of 3D rendering architectures are provided by Sheaffer et al. [135] and Crisu et al. [36].

Because memory accesses are very important for the energy consumption, a hardware architecture for minimising the costly memory accesses was developed by Akenine et al. [1]. Antochi et al. [3] have explored different texture caches for embedded 3D Graphics. A good exploitation of caches reduces energy by shifting the costly off-chip to cheap on-chip memory accesses.

Euh et al. [41] have proposed an adaptive texture mapping that changes the texture filters according to their impact on the visual quality. This work is complementary to our work because it also takes the visual quality into account, but it works on the appearance in stead of the geometric model.

7.2 The simulated architecture

Subsection 7.2.1 describes the architecture of the simulated 3D rendering pipeline and its performance. The simulator of the 3D rendering pipeline, used in this chapter for obtaining the results, uses a statistical and fixed value for the cache miss rate of the texture memory. Because this miss rate heavily depends on the used content and on the viewing conditions, Subsection 7.2.2 first evaluates the correctness of this assumption. Subsection 7.2.3 then discusses the energy-performance trade-off for the simulated architecture.

7.2.1 The architecture

Real-time 3D graphics are nowadays typically rendered on a hardware accelerator called the Graphical Processing Unit (GPU). The energy and performance numbers of this GPU are in this PhD manuscript obtained with a GPU simulator developed by Sheaffer et al. [135]. This simulator uses the architecture of the GPU shown in Figure 7.3. Triangles are in this GPU architecture converted into a stream of vertices which are transformed into coloured points on the 2D screen [63] by the vertex processor. The rasteriser creates the fragments between these vertices and sends them to the fragment processor. The fragment processor then samples and filters the texture and modulates the resulting colour with the interpolated colour from the rasteriser. The processed fragments are then transferred to the frame buffer operations unit where they are blended, dithered, etc. The different units of Figure 7.3 are pipelined and drain a queue that is filled with data emanating from the previous unit.

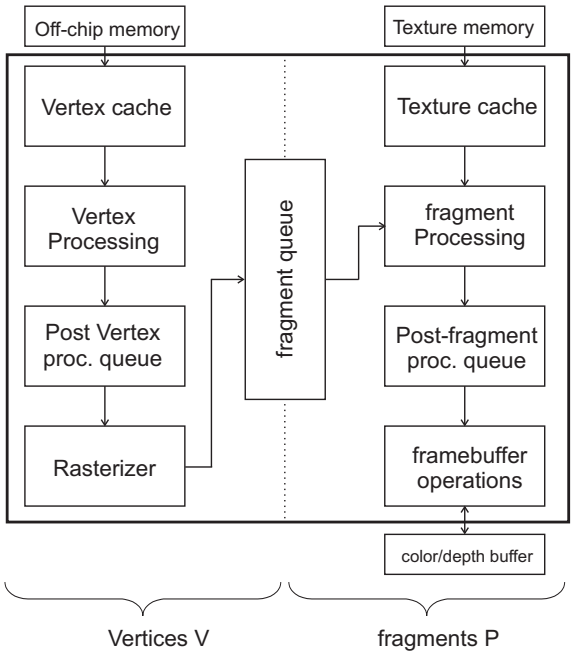


Figure 7.3: The simulated GPU architecture: vertex and triangle processing on the left and pixel processing on the right.

As discussed in Chapter 3, the time needed for executing the rendering commands is determined by the bottleneck in the rendering pipeline. A performance model, which is often used for modelling the execution time of the rendering pipeline is shown in Equation 7.1 (Chapter 3, [162]).

$$T(x) = max(T_f \cdot F(x), T_p \cdot P(x)) \tag{7.1}$$

The parameter x is the object under consideration, F is the number of triangles of the object and P is the number of fragments in the projection of the object. A post T&L cache is not yet implemented in this simulator and the cache miss rate m of Subsection 3.3.2 is thus equal to 1. The coefficients T_f and T_p are the reciprocal of respectively the vertex and fragment throughput of the GPU. The coefficients T_f and T_p are determined by the performance of the processing units that are respectively on the left and on the right of the fragment queue of Figure 7.3.

In practice, the real execution time will be a little higher than the estimated one (Figure 7.4) because the bottleneck regularly moves from one unit to the other. For instance, large triangles have their bottleneck typically in the fragment processing units, while small triangles typically have their bottleneck in the vertex processing units. The shifting bottleneck results in a vertex and pixel unit that are not constantly computing and hence are sometimes waiting for each other to process the data [162]. Figure 7.4 shows the resulting small difference for the simulator between the estimated and the real execution time.

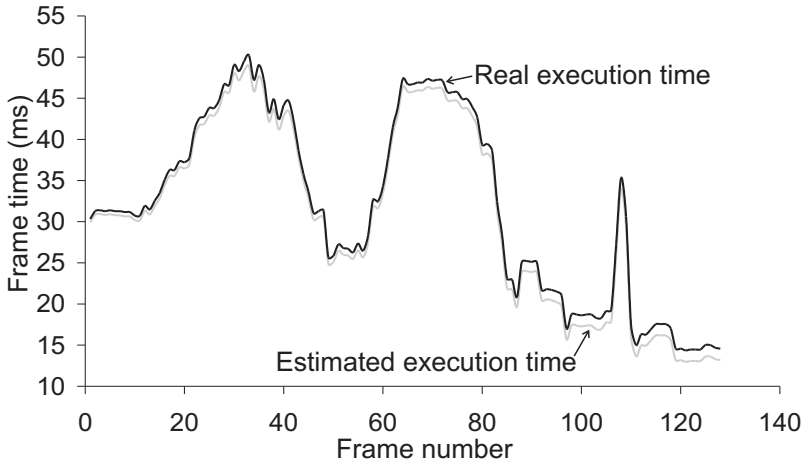


Figure 7.4: Estimated versus simulated rendering time (simulator of Sheaffer et al. [135])

7.2.2 Texture memory

An important issue for the execution time and energy consumption are the memory accesses of the pixel processing unit in the texture memory [1]. The simulator of Sheaffer et al. [135] takes the accesses into the texture cache into account, but uses a statistical and fixed value for the texture cache miss rate. The simulator does not take the off-chip texture memory accesses into account. The number of texture cache misses however heavily depends on the used content and on the viewing conditions.

Chapter 3, Subsection 3.2.3 illustrates how the texture cache misses depend on both the size and the orientation of the texture for a software renderer with a line based

representation of the texture memory. Block based representations of the texture memory (e.g. the leading texture compression method $S3TC^{TM}$ based on the work of Iourcha et al. [72]) eliminate the dependencies on the orientation of the texture cache misses [60], but the texture cache miss rate still depends on the size of the texture. Chapter 3, Subsection 3.3.3, also illustrates this dependency of the execution time of the fragment units on the size of the textures.

The solution for this effect, which is available on current graphics hardware, is to use mipmapped textures [80] that use multiple images of a single texture map at different resolutions. The resolution that is sampled is then chosen depending on the actual size of the triangle. This heavily reduces the texture cache miss rates, which is illustrated in Figure 7.5-mipmapped. Therefore, only mipmapped textures should be used because they increase the visual quality [80] and decrease the number of texture cache misses.

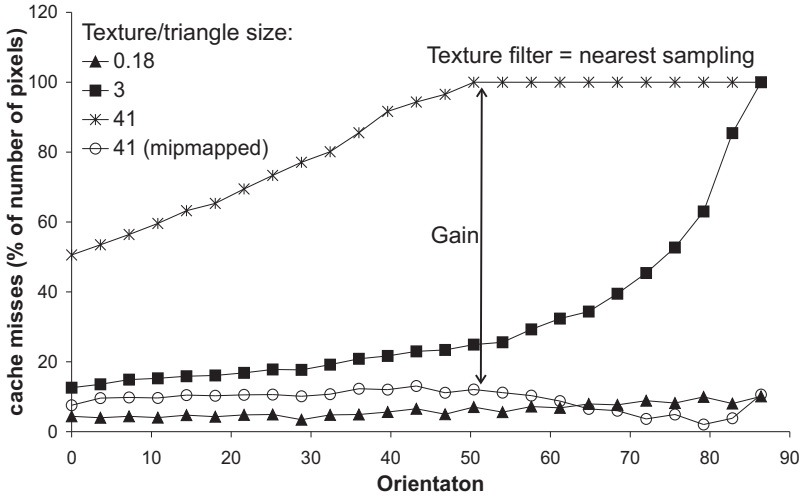


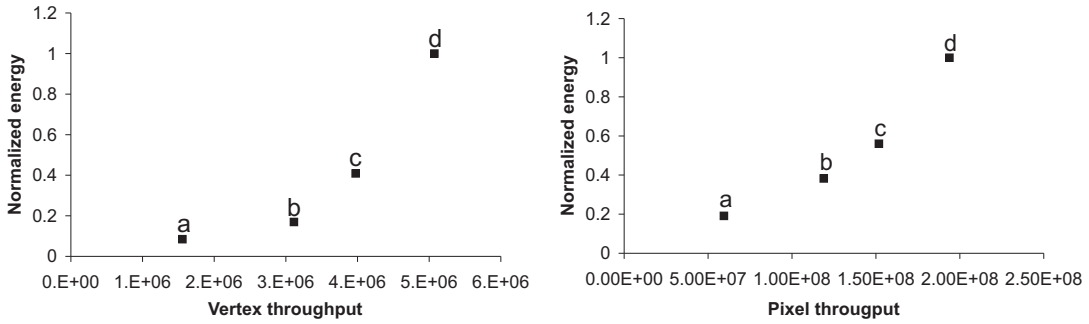
Figure 7.5: Comparison of texture cache miss rate for textured rendering with and without mipmapping.

Because the texture cache miss rate is according to Figure 7.5-mipmapped made independent of the orientation and size of the texture, the variation on the execution time of the pixel processing unit decreases. The use of mipmapped textures thus increases the predictability of the pixel throughput and the rendering time model. It also validates the choice of Sheaffer et al. [135] to use a statistical fixed texture cache miss rate in the simulator. This statistical fixed texture cache miss rate may however not be used with non-mipmapped textures. The penalty for using mipmapped textures is an extra memory size of 33%. If this larger size directly translates into a larger on-chip memory, then the energy consumption increases logarithmical with the increasing size. However, the energy consumption linearly increases with the number of off-chip accesses and the total energy consumption thus decreases.

7.2.3 Energy consumption

We have estimated the energy consumption of the vertex and pixel processing units for different configurations, by changing the clock frequencies and voltages of the units of Figure 7.3. The resulting trade-off is shown in Figure 7.6, with point d the reference. This trade-off depends on the initialisation of the rendering pipeline (e.g. using textured rendering or not) and must thus be derived for every possible rendering state that is used in the application.

The values for the energy consumption were derived with a high level simulator and the errors on the absolute values for the energy consumption are for such high-level estimators typically between 25% and 50% compared with low-level circuit simulators [36]. We however show that is possible to gain a lot of energy by exploiting the dynamic workload of a 3D graphics application for a constant quality level. We are thus more interested in the relative gains in energy consumption (following the CV^2f rule), for which it is reasonable to expect that they are more accurate.



(a) Vertex processing (b) Pixel processing

Figure 7.6: The energy consumption as a function of the vertex (a) and pixel (b) throughput.

The issue is now to choose a configuration that gives a vertex and pixel throughput that is just high enough for reaching the quality and performance requests of the user, but that consumes as little energy as possible. Note that the configurations for the vertex and pixel units in Figure 7.6 are independent of each other and only yield this performance with ideal conditions.

7.3 The framework extension

Figure 7.7 shows different global scene distortion plots for 5 frames of a walk through the city scene. The distortion plots are derived with the gradient descent method of Chapter 2. The trade-off between the error and the number of triangles changes with the size (frame 0 to 30) and visibility (frame 30 to 90) of the character object.

Note that the discrete distortion plots of Figure 7.7 are represented as continuous plots for an easier visualisation.

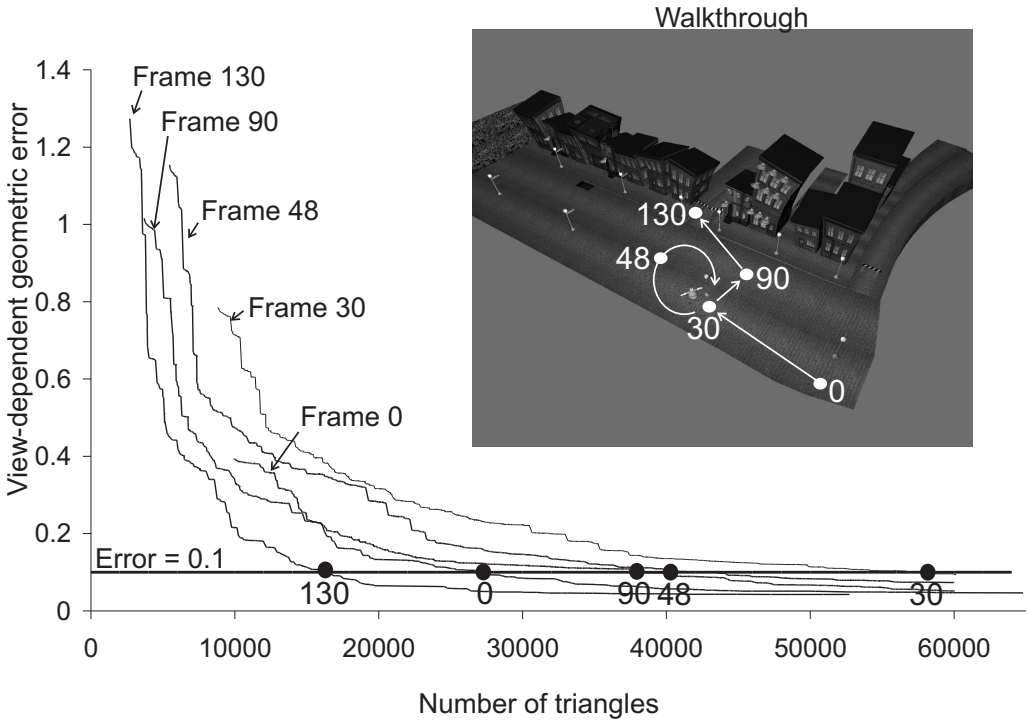


Figure 7.7: Global distortion plots for 5 different viewpoints on a walk through in the city scene.

Traditional optimisation algorithms, which do not take the energy consumption of the terminal into account, calculate a fixed triangle budget (using the performance model from Chapter 3) and adapt the visual quality to meet this budget. These algorithms typically increase the cost until the triangle budget is completely consumed (at a maximum energy consumption of 1 in Figure 7.9(a)) although the visual quality could already be satisfying. Additionally, changing the visual quality can be very disturbing for the user [52] and a better approach is thus to try to provide a constant visual quality that was requested by the user and to exploit the resulting changes of the triangle budget to lower the energy consumption. The horizontal line in Figure 7.7 shows this constant view-dependent error and its intersection with the view-dependent distortion plots yields the triangle budget that is needed for showing the object with this error (quality). Figure 7.8 illustrates how this triangle budget varies over the course of the walk through. This requested triangle budget is used together with the execution time deadline to derive the platform configuration (from Figure 7.6) that consumes a minimum of energy. if the quality request of the user cannot be met without exceeding the execution time deadlines for a maximum energy consumption, the optimisation algorithm decreases the visual quality (increase of geometric error).

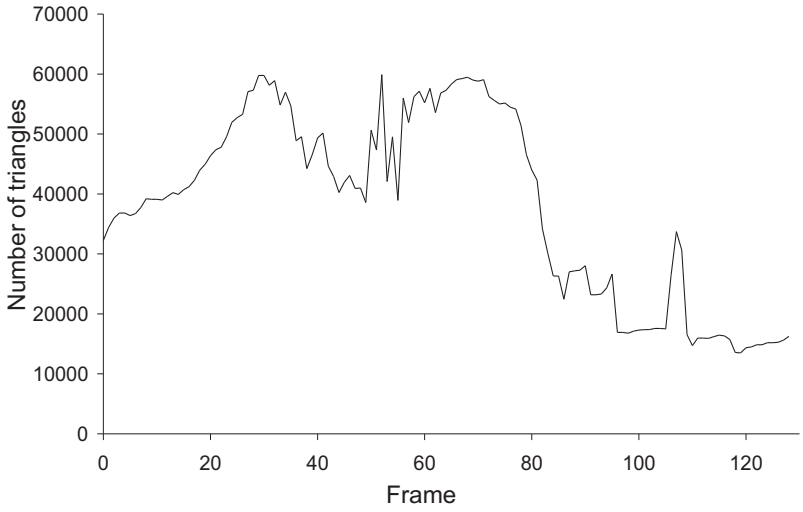


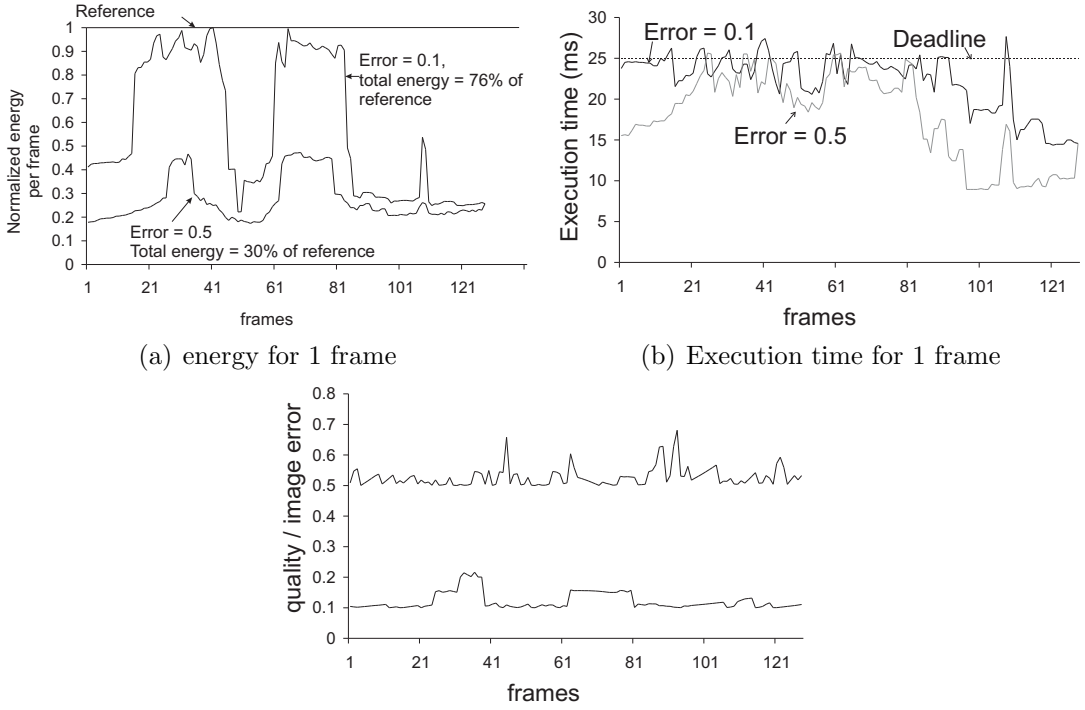
Figure 7.8: The variation of the triangle cost for a constant view-dependent geometric error for the walk through of Figure 7.7.

Figure 7.9 shows the energy consumption, execution time and image error for a walk through the city scene of Figure 7.7 for two different user selected quality levels. The view-dependent geometric errors (Figure 7.9(c)) of 0.1 and 0.5 select a high respectively mediocre visual quality, but also a high respectively low energy consumption per rendered frame (Figure 7.9(a)). The variation of the requested triangle budget is for the quality level 0.1 shown in Figure 7.8.

The real value of the execution time per frame (Figure 7.9(b)) depends on the shifting bottleneck and the deadline in Figure 7.9(b) is therefore sometimes a little exceeded.

The total energy consumption for the complete animation (all frames) is for an error of 0.1 and 0.5 respectively 76% and 30% of the energy consumption that is obtained with a traditional optimisation that uses a fixed triangle budget with the highest energy consumption (1 on the y-axis of Figure 7.9.a)

The sequence of frames of the walk through is illustrated in Figure 7.10 that also shows the triangle budget that is needed for guaranteeing a constant view-dependent geometric error of 0.1 (see also Figure 7.8). For the first frame, the execution time deadline of 25 ms (40 frames per second) is just met for an energy consumption for the frame of 40% compared to traditional approaches. The user then walks towards the character object. The visibility of artifacts on this object increases and its complexity must increase. To be able to meet the timing deadline, the configuration of the architecture must be changed at frame 17 so that not more than a pre-established maximum energy is consumed. From frame 25 to 38 and 63 to 80, the geometric error cannot be delivered for a maximum energy consumption and a deadline of 25 ms. Therefore, the geometric error is increased a little (see also Figure 7.9(c)) for these frames so that the deadline is met. The character is invisible



(a) energy for 1 frame

(b) Execution time for 1 frame

(c) View-dependent error for 1 frame

Figure 7.9: Energy (a), execution time (b) and quality (c) for a walk through the city scene.

from frame 83 on, which heavily decreases the needed triangle budget. Therefore, the deadline can easily be met with a platform configuration that consumes 30% of the maximum energy consumption.

The requested deadline is easily met for a requested view-dependent geometric error of 0.5 and the energy consumption per frame is significantly lower than for a requested image error of 0.1. The geometric error in Figure 7.9(c) is however harder to control. This is because of the large slope of the distortion plots in this region. Therefore, the geometric error varies a lot for the smallest change of the number of triangles in this region.

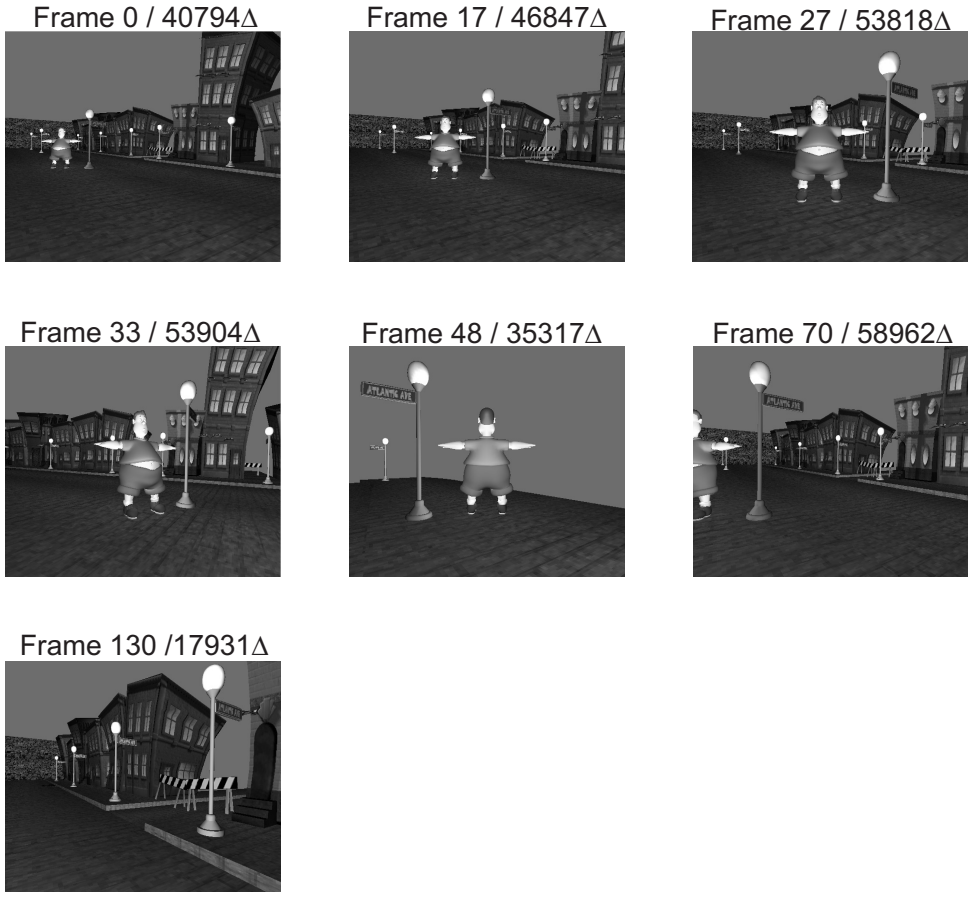
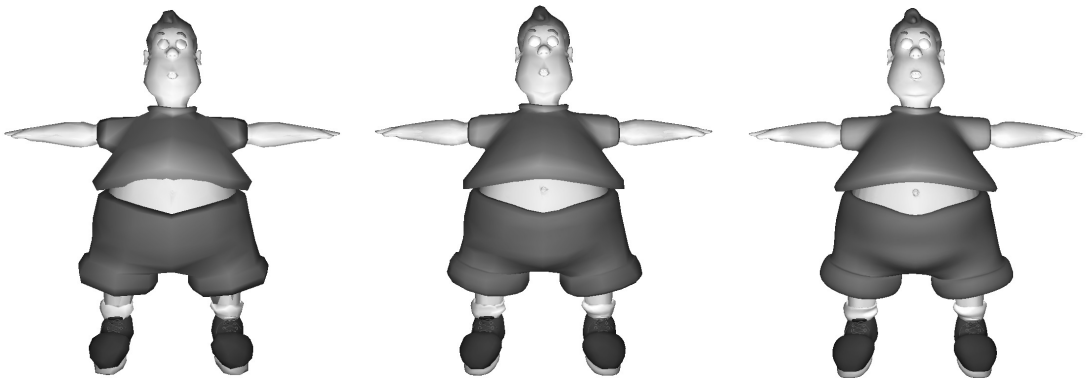


Figure 7.10: Visualised frames for the different viewpoints in Figure 7.9 and a requested image error of 0.1.

Figure 7.11 shows the impact of the lower requested quality on the character of the city scene. This character is in Figure 7.11 shown at three different quality levels: The two requested quality levels of 0.5 and 0.1 and the lowest possible quality level for the used optimisation method. Note that in Figure 7.11.a the quality could be easily improved with a local refinement of the silhouette of the object. But as explained in Chapter 5, this increases the online complexity so much that it results



(a) E = 1.0 (b) E = 0.5 (c) E = 0.1

Figure 7.11: The character from the city scene shown with different error E: a) Low quality, b) normal quality and c) high quality

in a non-optimal trade-off between quality and cost.

7.4 Conclusions

Because mobile devices run on batteries, the energy consumption of 3D graphics hardware must be limited. Work has already started to optimise the architecture for minimising the number of memory accesses and for building real low-power chips. Low-power design does not end here. Applications must exploit the dynamic behaviour of 3D graphics to adapt the performance and energy consumption of the system to the instantaneous complexity of the content. This results in a powerful energy-management when it is combined with existing optimisation algorithms, which adapt the visual quality to the available performance. Because of the observations that the change of image quality can be as disturbing to the user as a low image quality and that energy is a scarce resource on mobile terminals, we propose an optimisation algorithm that tries to provide a constant and user requested quality and that exploits the resulting dynamic workload to minimise the energy consumption for a given execution time deadline. The example of this text consumes 76% and 30% of the energy consumption that is obtained with a traditional optimisation algorithm (that uses a fixed triangle budget) for a high respectively low visual quality.

Chapter 8

Conclusions and suggestions for future work

This chapter presents the conclusions of the PhD manuscript in a first section. Section 8.2 then presents some suggestions for future work.

8.1 Conclusions

In the era of internet and multimedia communications, information and multimedia content will be accessible from anywhere at anytime. Consequently, a flurry of terminals with large differences in terminal capabilities will be used to access the multimedia content. As a result, content creators and providers do not longer know in advance which terminals will be used to consume their multimedia content. This makes it challenging for them to guarantee that the content can be shown with a satisfying quality on all these very different terminals. In addition, the processing time for 3D graphics applications can easily vary for the same content over one order of magnitude. Therefore, we present a framework that controls the execution time of an arbitrary 3D rendering pipeline by the control of the quality and complexity of the input data (submitted paper [145]). Lafruit et al. [55] have presented an overall overview of this work.

One of the key technologies to enable the gradual trade-off between quality and complexity is scalable or multi-resolution coding. Because networked access of 3D content is important in our target scenario, we have chosen to implement a multi-resolution technique with a very good compression: MPEG-4 Wavelet Subdivision Surfaces (WSS). The MPEG-4 WSS specification supports the adaptive decoding by partial transmission, but a non-uniform decoder did not yet exist at the time of the research. We have implemented an adaptive MPEG-4 WSS mesh decoder, which can be used for real-time rendering. Several possibilities for the adaptive decoding of the content were implemented and tested for their usability in the optimisation

framework. View-dependent adaptation is only useful with very large meshes for which many triangles can be easily eliminated from the rendering. Therefore, we have chosen to only use uniform or view-independent non-uniform adaptation in the optimisation framework. Tack et al. have presented this work in [146, 147, 148] (submitted paper [143]).

An optimisation algorithm distributes the available triangle budget among the visible objects in the scene based on their impact on the visual quality. Because this optimisation problem is used in real-time rendering, the assessment of the visual quality must be very fast. Therefore, previous work typically uses very simple heuristics to assess the accuracy of the content. On the other hand, complex error metrics exist that control the simplification of 3D content. We propose a hybrid approach that combines the higher accuracy of the complex error metrics with the simplicity of the simple heuristics. The quality and cost of the different input settings of the WSS decoder are for the separate objects in the scene measured in off-line preprocessing steps and the optimal trade-offs between quality and cost are summarised in a distortion plot. The online steps of the optimisation then combine the distortion plots of the visible objects in the scene with a minimum of online parameters (e.g. user viewpoint) to find the overall trade-off between the quality and cost. The proposed framework increases the visual quality for the same rendering cost, while the execution time of the optimisation itself is comparable with previous work. The highest execution time of the optimisation was equal to 6 % of the rendering time for the tested scenes. This is similar to the optimisation execution times reported by Funkhouser et al. [52] (average 10% and maximum 30%) and Gobbetti el al. [56] (between 5% and 20%), which use very simple heuristics for calculating the visual quality. Tack et al. have presented this work in [144].

Because the geometry is the only parameter that is changed by our optimisation framework, the view-dependent accuracy of the geometry is the core parameter of our quality model. The geometry is however not the only input parameter and therefore, we have followed the approach of previous work to use scaling factors that adapt the core parameter to the changing conditions. The proposed quality metric results in image quality values that are higher for the proposed heuristic compared with the simple heuristics for the same cost. The presented example gives a difference of 10 dB (PSNR) and the SSIM values show that the proposed quality metric results in an optimisation that reaches the almost perfect SSIM values much faster than the simple heuristic. Tack et al. have presented this work in [144].

The optimisation uses the number of triangles as a cost metric, which is translated into the execution time by a performance model. A method for deriving the performance model of an embedded device using software rendering (white box modelling) and a PC using hardware accelerated rendering (black box modelling) has been presented. The accuracy of these performance models is improved with respect to previous work because of an online correction system using a recursive least square algorithm. For the software rendering, the average of the absolute values of

the errors for all tested scenes is between 3.33 % and 5% with a confidence level of 95%. Bontempi el al. [18] who have used a black box approach to derive a performance model for a texture decoder report errors between 15% and 40% for a linear model and 3.8% and 6% for a (more complex) non-linear model. Tack et al. have presented this work in [148]. Wimmer and Wonka [162] report average errors for a complete scene rendered on a Geforce 3 product up to 30% for the maximum model and 20% for the summation model, which is more or less the same as the errors on our uncorrected performance models for a Geforce FX 5900. The online correction system however heavily improves the results of the performance models.

Minimising the energy consumption is an important issue on mobile devices. Therefore, we propose to extend the optimisation algorithm to provide a constant and user requested quality and to exploit the resulting dynamic workload to minimise the energy consumption for a given execution time deadline. The presented example consumes 76% and 30% of the energy consumption that is obtained with a traditional optimisation algorithm (that uses a fixed triangle budget) for a high respectively low visual quality. Tack et al. have presented this work in [142].

8.2 Future work

The final goal of our work is to design a framework that guarantees a maximum visual quality for the given constraints on the processing power. Because polygonal (triangular) rendering was until now the most popular way to render a 3D scene, we have chosen to develop our framework for this type of rendering. However, alternative rendering methods have also been presented to avoid bottlenecks due to 3D polygonal rendering on e.g. mobile devices. For instance, Duguet et al. [40] have proposed to use point based rendering, Diepstraten et al. [38] and Lamberti et al. [79] have proposed to use remote rendering and Chang et al. [29] have proposed to use image based rendering. A very interesting path for future work is to integrate these different rendering techniques into the optimisation framework.

Currently, the visibility extraction of the optimisation framework is only able to eliminate invisible triangles for static content. Hence, a second topic of future work could be to support easy visibility extraction for animated content.

Because a GPU is currently the only hardware accelerated platform that is used for rendering 3D content, a performance model was derived for a GPU using traditional rendering techniques. However, different rendering techniques, such as tile based rendering [118], have been introduced. A performance model is needed to support these platforms; hence, a needed path for future work is therefore to derive a performance model for new types of GPUs.

A performance model for software rendering was derived because many embedded devices did not have a GPU at the time of the research. This performance model

was however only developed for a Pentium IV and XScale processor. Future work should investigate how this performance model behaves on other types of platforms like VLIWs (e.g. TI C6x processors), array processors (e.g ADRES [92]), etc.

The proposed quality metric performs better than the previous proposed error metrics, but many issues remain unsolved. One of the most important remaining issues is the influence of the texture on the visual quality: texture shifting because of the simplification process, texture masking and the influence of the texture filters on the visual quality. The work of Williams et al. [160] and Ferwerda [45], on objective quality models based on a study of human vision, are important sources for the integration of these effects into the quality model. Also the work of Watson et al. [157] on subjective quality modelling can be an important source for a further improvement of the quality model.

An important issue of view-dependent scalable content is the mismatch in performance between the CPU (content adaptation) and the GPU (rendering). Because view-dependent content assigns the triangles to the object regions that are really important for the visual quality, an interesting path for future work could be to investigate how the current platforms could be changed to decrease the overhead for view-dependent LOD. This work could for instance be important for mobile devices where the energy consumption is decreased and the performance increased by limiting the work to what is really important for the visual quality. The work of Bolz et al. [14, 15] on the evaluation of subdivision surfaces on the Graphical Processing Unit is an important reference for this research.

Currently, the energy consumption is only minimised by controlling the geometric input, i.e. the number of triangles. However, the energy consumption depends on many factors. For instance, Euh et al. [41] has proposed a texture filtering that adapts its accuracy based on an estimate of its impact on the visual quality. An interesting path for future work could therefore be to integrate this work for gaining more energy for the 3D rendering on mobile devices.

Appendix A

Presentation of the content

This appendix discusses the content that is used in this PhD. manuscript to show the results of the proposed framework. Section A first introduces the content and the animations and Section A then shows the uncontrolled rendering time for the content and animations used in Chapter 6.

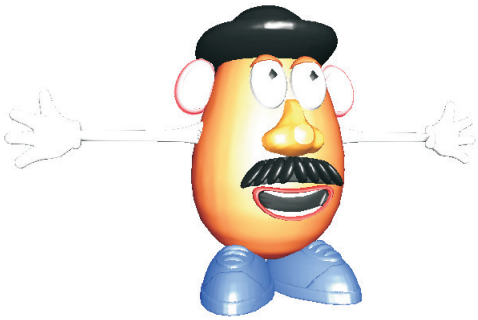
The content

A.1.1 The Mr. Potato scene

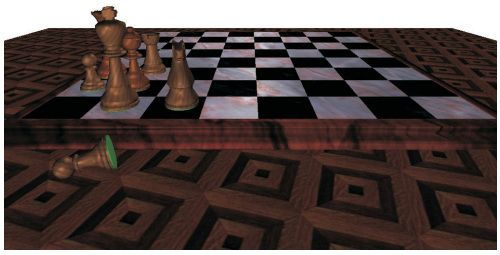
Figure A.1(a) shows the simplest scene that is used in this text to show the results of the proposed framework. It consists of a single object with 15 independent meshes. If the viewpoint is located in the front of the object (viewpoint for Figure A.1(a)), all meshes are visible a, but if it is located on the left or right side of the object, the framework eliminates the invisible arm and ear and if it is located in the back of the object, the framework eliminates the nose, eyes and mouth. The animation that is used in the scene starts from the front viewpoint, then first rotates the object 360° to change the number of visualized meshes (number of triangles) and then zooms out from the initial front viewpoint to decrease the number of pixels.

A.1.2 The chess scene

Figure A.1(b) shows the second scene that is used in this text to show the results of the proposed framework. It consists of a 9 objects and 10 meshes (the checkerboard has two meshes for the frame and the checks). The objects are not combined into one object, because one must be able to remove the chess pieces while playing a chess game or to remove the checkerboard after playing the game. All objects are always visible except if the viewpoint is located under the table that hides the



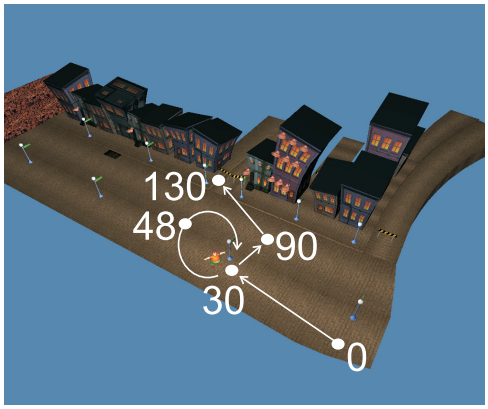
(a) Mr Potato



(b) Chess



(c) Fruit



(d) City scene with animation

Figure A.1: The different scenes that are used in this manuscript for the execution time modelling and control.

checkerboard. The animation is the same as the animation for the Mr. Potato scene that is described in the previous subsection and starts from the viewpoint that is used for Figure A.1(b).

A.1.3 The fruit scene

Figure A.1(c) shows the third scene that is used is used in this text to show the results of the proposed framework. It consists of a 22 objects and 22 meshes. The objects are not combined into one object, because one must be able to remove the fruit from the plate. The grapes and cherries are for some viewpoint hidden after the apple, the pear and the banana. All the fruit is hidden if the viewpoint is located under the plate. The animation is also the same as the animation for the Mr. Potato scene that is described in the previous subsection and starts from the viewpoint that is used for Figure A.1(b).

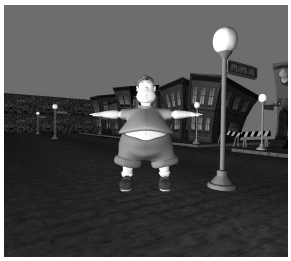
A.1.4 The city scene

Figure A.1(d) shows the fourth and most complex scene that is used in this text to show the results of the proposed framework. It consists of a 46 objects and 111 meshes. The animation is also shown in Figure A.1(d). The user starts from point 1 in Figure A.1(d) and then walks towards the character. This increases the importance of the character in the scene. Then, the user walks around the character to trigger the visibility tests of the framework for this object. For example, the eyes, nose, etc. are invisible from the back of the character and are eliminated by the framework. The user then walks away from the character until it is invisible, which results in a huge execution time drop. The user then further walks in the city to increase the importance and resolution of the lampposts in the back of the city.

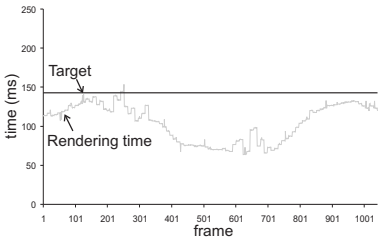
The uncontrolled execution time

Three different scenes are used for presenting the results of the optimization framework on three different platforms. The first platform is an embedded device, called Xingu, with an XScale processor running at 600 MHz. The rendering on this device is done in software because it lacks a hardware accelerator. The second platform is a PC that runs the same application as Xingu (software rendering) and the third platform is a PC with a recent hardware accelerator (Geforce FX 5900).

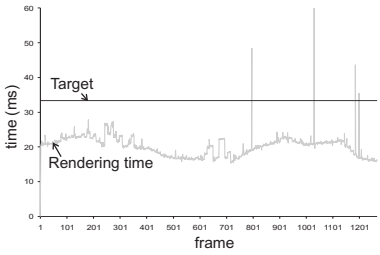
The first scene is the city for which the uncontrolled execution time is illustrated in Figure A.2 The second scene is the chess scene for which the uncontrolled execution time is illustrated in Figure A.3 and the third scene is the Mr. Potato scene for which the uncontrolled execution time is shown in A.4.



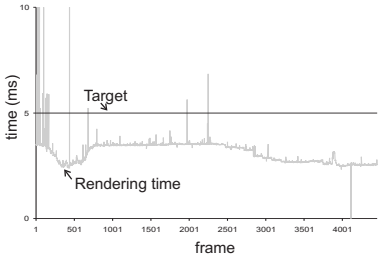
(a) The city scene



(b) Software: Xingu

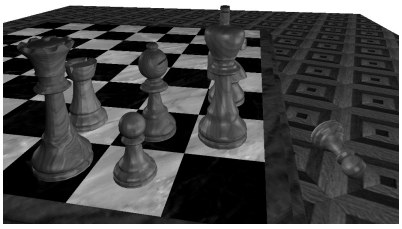


(c) Software: PC

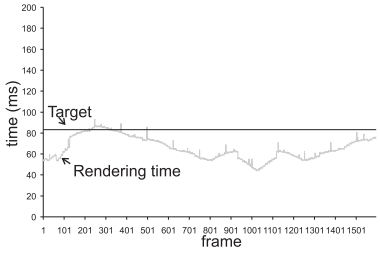


(d) Hardware: Geforce FX 5900

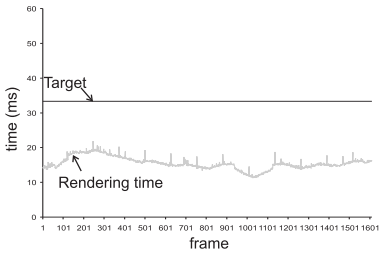
Figure A.2: Uncontrolled execution time for the city scene on the different platforms.



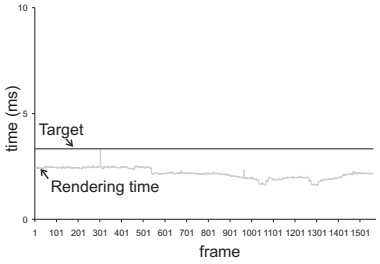
(a) The chess scene



(b) Software: Xingu

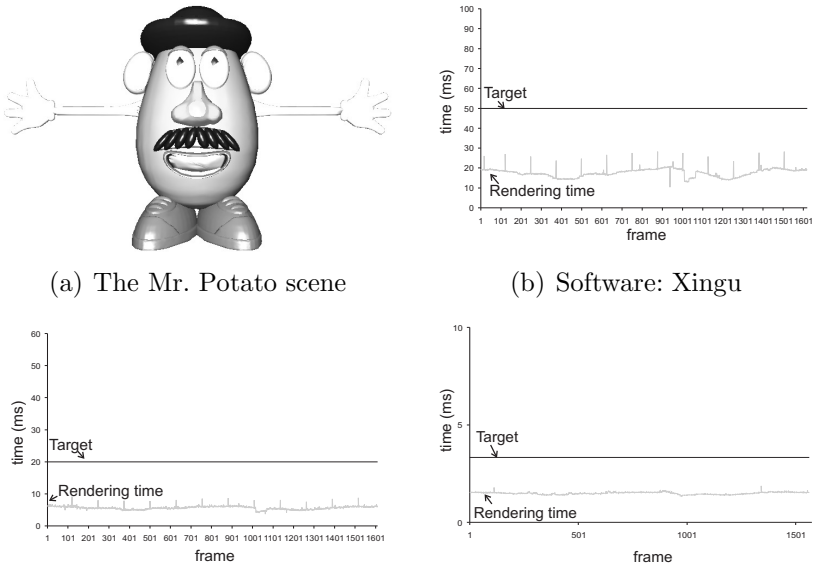


(c) Software: PC



(d) Hardware: Geforce FX 5900

Figure A.3: Uncontrolled execution time for the Chess scene on the different platforms.



(a) The Mr. Potato scene (b) Software: Xingu

(c) Software: PC (d) Hardware: Geforce FX 5900

Figure A.4: Uncontrolled execution time for the Mr. Potato scene on the different platforms.

Bijlage B

Nederlandstalige samenvatting

Terminologie

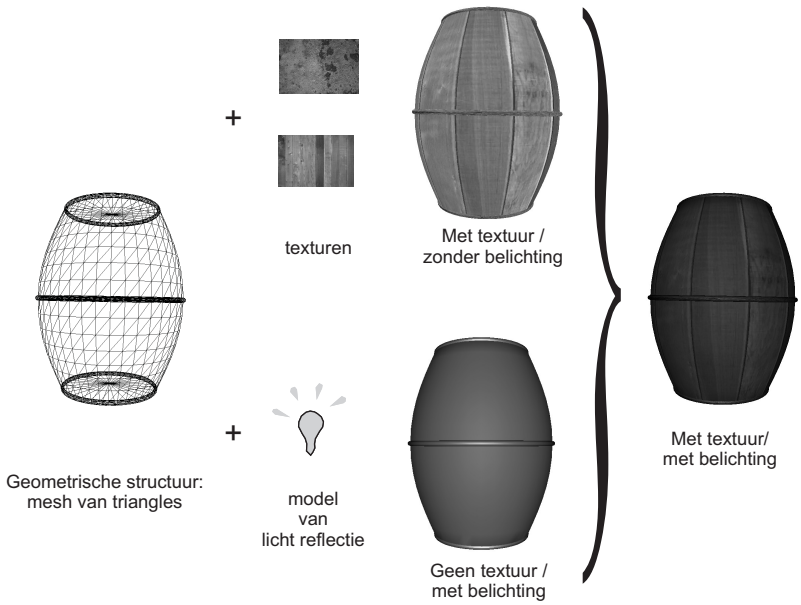
Bij wijze van inleiding volgt een korte beschrijving van de terminologie die in deze samenvatting gebruikt wordt. Meer informatie over '3D computer graphics' kan gevonden worden in de boeken van Foley et al. [46] en Hill [63].

'3D computer graphics' is een domein in de computerwetenschappen waarin het opslaan, manipuleren en visualiseren van drie dimensionale wiskundige modellen van een virtuele wereld bestudeerd worden. Typische toepassingen zijn: computerspellen, film en televisie, e-commerce, wetenschappelijke visualisatie, enz. Deze toepassingen kunnen verdeeld worden in twee categorieën met totaal verschillende kenmerken en doelstellingen: (i) foto-realistische toepassingen, die een zeer hoge visuele kwaliteit vereisen, maar een hogere rekentijd en financiële kost toelaten; (ii) interactieve toepassingen, die bestemd zijn voor de consumentenmarkt en dus een lage financiële kost vereisen, maar ook een lagere visuele kwaliteit toelaten. Deze thesis focust op het laatste type toepassingen.

Er bestaan verschillende technieken om een 3D object te modelleren. Bij het opstellen van het model kan men er in vele toepassingen vanuit gaan dat het inwendige van het object onzichtbaar is, waardoor het voldoende is om enkel het oppervlak van het object te beschrijven. Om een oppervlak van een object te beschrijven bestaan er opnieuw verschillende methoden. De meest populaire is het gebruik van een stuksgewijs lineaire benadering met driehoeken, in deze tekst een set van driehoeken genoemd. De voornaamste reden van deze populariteit is eenvoud: de visualisatie van driehoeken is namelijk eenvoudig te implementeren in hardware.

Een set van driehoeken wordt dikwijls voorgesteld met een paar $\{V, K\}$, waarin V een verzameling punten in de ruimte is en K de verzameling van driehoeken, die bepalen hoe de punten uit V verbonden zijn. Zoals voorgesteld in figuur B.1, beschrijft een set van driehoeken enkel de geometrie van een object, maar bevat het geen enkele informatie over de kleur van het object. De kleurinformatie wordt daarom

dikwijls toegevoegd met een wiskundig model van de belichting in combinatie met één of meerdere texturen. Dit zijn afbeeldingen die als een soort behangpapier op het oppervlak van het object gekleefd worden.



Figuur B.1: Een 3D model waarbij geometrie en kleur gecombineerd worden.

Het aantal driehoeken in K en het aantal punten in V zijn belangrijke parameters voor de rekentijd die nodig is om het 3D model te visualiseren. Het veranderen van het aantal punten en driehoeken van het object is daarom een belangrijke methode om de rekentijd te controleren. Technieken die een set driehoeken vereenvoudigen genereren een nieuwe set M_i (figuur B.2). Deze set M_i is een benadering is van de originele set M, maar bestaat uit minder driehoeken en heeft bijgevolg een lagere kwaliteit. Als deze vereenvoudigde set M_i deel is van een reeks van steeds sterker vereenvoudigde sets, noemt men M_i een 'Level Of Detail' (LOD).



Figuur B.2: De vereenvoudiging van een set driehoeken toegepast op het bekende Stanford konijn.

Hoofdstuk 1: Inleiding

In de intelligente omgevingen van de toekomst zal de gebruiker een centrale rol krijgen in het ontwerp van ICT-toepassingen. De gebruiker zal op een eenvoudige manier informatie kunnen opvragen en aanbieden, waar en wanneer hij dit zelf wenst. Bijgevolg zullen willekeurige netwerken en toestellen gebruikt worden voor het transport, de interactie en visualisatie van deze informatie.

Door de steeds wisselende context waarin multimediatoepassingen gebruikt worden, is het voor de aanbieders van deze toepassingen een hele uitdaging om in alle omstandigheden een goede kwaliteit te garanderen. Het antwoord op deze uitdaging ligt in een geavanceerde techniek die de nodige bandbreedte en complexiteit van de informatie aan de beschikbare netwerk- en toestelmogelijkheden aanpast. Dit proces wordt dikwijls 'Quality of Service' (QoS) genoemd. Een belangrijk element in QoS is het maximaliseren van de kwaliteit ondanks de beperkte mogelijkheden van enerzijds het netwerk en anderzijds het toestel, zonder dat de complexiteit van het QoS-algoritme te hoog wordt.

Het meest bekende aspect van QoS is netwerk-QoS. Netwerk-QoS is een kwalitatief en kwantitatief performantiecontract tussen de aanbieder van de dienst en de gebruikte toepassing. Netwerk-QoS is echter niet het enige belangrijke aspect. De eigenschappen van de gebruikte toestellen zoals schermgrootte, performantie en resolutie variëren sterk. Terminal-QoS is dan ook nodig om de kwaliteit te maximaliseren ondanks de beperkte mogelijkheden van het gebruikte toestel. Om zowel netwerk- als terminal-QoS te ondersteunen heeft het MPEG-21 multimedia framework Digital Item Adaptation (DIA) gedefinieerd. DIA past een stroom bits (gecomprimeerde multimedia data) aan aan de beschikbare opslag- en transmissiecapaciteit en de beschikbare performantie.

Niet alle problemen kunnen echter opgelost worden door het aanpassen van een stroom bits. De rekentijd voor de visualisatie van een 3D wereld is naast het aantal driehoeken ook sterk afhankelijk van het gezichtspunt van de gebruiker. De rekentijd kan daarom zeer sterk variëren voor dezelfde data bekeken vanuit verschillende gezichtspunten. De OpenGL Performance Characterization Organization heeft variaties in de visualisatietijd gemeten van meer dan een ordegrootte. '3D graphics data' met een vaste LOD of kwaliteit, die voor alle gezichtspunten een goede rekentijd geeft, zal door de variatie een te lage kwaliteit hebben voor de meeste gezichtspunten. Een continue aanpassing van de informatie aan de beschikbare rekenkracht van het toestel is dus noodzakelijk voor een optimale kwaliteit en performantie.

Omdat het onderzoek in het domein van QoS zeer uitgebreid is, hebben we ons onderzoek toegespitst op één belangrijk aspect, namelijk de controle van de nodige rekenkracht door de controle van de complexiteit van de '3D graphics data'.

Hier volgen de belangrijkste bijdragen van ons werk. Een eerste bijdrage is de ontwikkeling van een snel optimalisatie-algoritme dat gebruik maakt van een complexe,

maar realistische kwaliteitsmaat. De complexiteit van dit algoritme is vergelijkbaar met de complexiteit van eerdere voorgestelde algoritmen, maar het algoritme levert een betere kwaliteit voor dezelfde visualisatietijd. We hebben de complexiteit van het eigenlijk algoritme beperkt door alle complexe onderdelen vooraf uit te voeren. De resultaten van de voorbereidende faze worden eenvoudig gecombineerd tot een globale keuze van de kwaliteit voor de volledige virtuele wereld.

Een tweede bijdrage bestaat uit een verbetering van de standaard 'MPEG-4 decoder' voor 'Wavelet Subdivision Surfaces (WSS)' en een selectie van bruikbare adaptatiemethoden, die gebaseerd is op een studie van de complexiteit.

De verbetering van bestaande rekenkrachtmodellen voor recente '3D graphics architecturen' is een derde bijdrage.

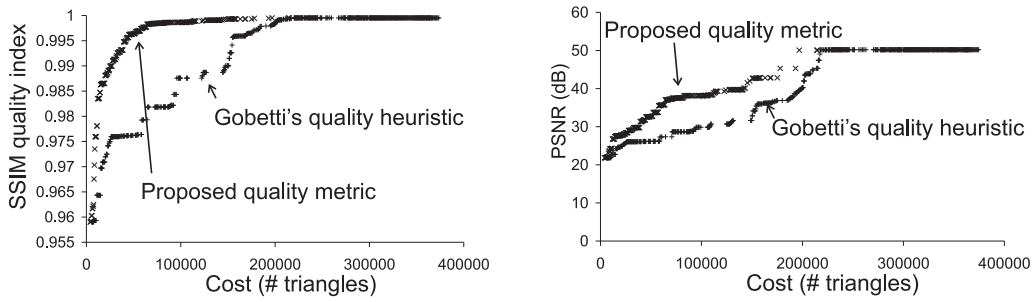
Tot slot volgt een vierde bijdrage: de uitbreiding van het voorgestelde algoritme om het energieverbruik te minimaliseren met een vooraf bepaalde constante kwaliteit en rekentijd. Het voorbeeld in deze thesis geeft een energiewinst van 24% en 70% voor een hoge respectievelijk lage visuele kwaliteit.

In hoofdstuk 1 van de thesis wordt het kader van het onderzoek geschetst en wordt achtergrondinformatie gegeven. In sectie 1.1 wordt onze motivatie besproken en wordt de probleemstelling geformuleerd. In Sectie 1.2 wordt werk complementair aan het onze voorgesteld. In sectie 1.4 wordt een globaal overzicht gegeven en sectie 1.5 eindigt met de samenvatting van onze bijdragen.

Hoofdstuk 2: Optimalisatie-algoritme

In hoofdstuk 2 wordt het door ons ontwikkelde optimalisatie-algoritme beschreven. Dit algoritme past automatisch de Level Of Detail (LOD) van de '3D graphics data' aan aan de beschikbare rekenkracht van het toestel. Vanwege het interactieve karakter van de '3D graphics toepassing' is een snel algoritme vereist. In het verleden bewandelde men vaak de weg van vereenvoudigde, maar onnauwkeurige kwaliteits- en performantiemodellen. Wij hebben ervoor gekozen om toch complexere kwaliteitsmodellen te gebruiken, waarbij we de complexe delen verschoven hebben naar een voorbereidende faze. Het eigenlijke algoritme wordt daardoor gereduceerd tot een eenvoudige combinatie van de resultaten uit dat voorbereidend werk. Zoals geïllustreerd in figuur B.3, zorgt het gebruik van een realistische kwaliteitsmaat voor een betere visuele kwaliteit en dezelfde visualisatietijd als voordien. De optimalisatietijd is echter vergelijkbaar met de optimalisatietijd van de andere algoritmen.

De link tussen de voorbereidende faze en het eigenlijke algoritme is een distortiecurve, die enkel optimale punten bevat met een maximale kwaliteit voor een minimale kost. Deze distortiecurve kan ook Paretocurve genoemd worden. Omdat ze enkel de optimale punten bevat is het immers onmogelijk de kwaliteit te verbeteren zonder ten minste één kostparameter te verhogen.



(a) SSIM index

(b) PSNR

Figuur B.3: Vergelijking van het optimalisatie-algoritme met het algoritme van Gobbetti: (a) Vergelijking op basis van de SSIM kwaliteitsindex en (b) vergelijking op basis van de PSNR metriek.

In sectie 2.1 van dit hoofdstuk wordt eerst complementair werk over eerder uitgevonden optimalisatietechnieken beschreven. Daarna wordt in sectie 2.2 een overzicht gegeven van ons algoritme. De voorbereidende faze wordt beschreven in sectie 2.3 en het eigenlijke algoritme in sectie 2.4. In sectie 2.5 worden een aantal voorbeelden gegeven en in sectie 2.6 wordt dit hoofdstuk geëindigd met de conclusies.

Hoofdstuk 3: Performantiemodellen

Ons optimalisatie-algoritme, beschreven in hoofdstuk 2, gebruikt het aantal driehoeken als een platform-onafhankelijke kostparameter. In Hoofdstuk 3 beschrijven we hoe deze parameter de visualisatietijd beïnvloedt. We onderscheiden twee verschillende aanpakken: enerzijds kan de rekentijd continu opgemeten worden en wordt het budget van de driehoeken voor de volgende beelden aangepast op basis van deze metingen; anderzijds kan het budget van de driehoeken berekend worden met een wiskundig model.

De eerste methode wordt vaak gebruikt in toepassingen met een grote consistentie tussen opeenvolgende beelden. Als de opeenvolgende beelden echter te veel verschillen zal deze methode resulteren in oscillaties en 'overshoot'. Wanneer de gebruiker bijvoorbeeld een kamer van een huis verlaat en een andere kamer binnen stapt, dan zullen de gevisualiseerde data zeer bruusk veranderen op een zeer korte tijd. Aangezien metingen van de visualisatietijd in de eerste kamer niet kunnen gebruikt worden om de data in de tweede kamer aan te passen is hier een mathematisch model van de visualisatietijd vereist.

In hoofdstuk 3 bespreken we daarom de rekentijdmodellen voor de platformen, die in deze thesis gebruikt worden om het optimalisatie-algoritme te illustreren. Eerst wordt een methode voorgesteld om het rekentijdmodel af te leiden voor de software-

visualisatie op een mobiel toestel. De belangrijkste uitdaging hierbij was de balans vinden tussen de nauwkeurigheid en de complexiteit van het model. Daarna wordt een methode voorgesteld voor het afleiden van het rekentijdmodel voor hardware-visualisatie. Twee verschillende modellen, die initieel ongeveer dezelfde nauwkeurigheid gaven, werden getest. Aangezien de nauwkeurigheid van beide modellen beperkt bleek, was een 'online correctiesysteem', dat het model continu aanpast aan metingen van de rekentijd, vereist. Na het toepassen van dit 'online correctiesysteem' werd één model geselecteerd voor het optimalisatie-algoritme.

Sectie 3.1 van hoofdstuk 3 begint met een beschrijving van complementair werk over rekentijdmodellen voor '3D visualisatie'. Tijdens het onderzoek was '3D hardware' nog niet beschikbaar voor mobiele toestellen, waardoor een rekentijdmodel voor een software-visualisatie nodig was. In sectie 3.2 wordt daarom een methode voorgesteld om zo'n model af te leiden. De rekentijd modellen voor 3D hardware worden beschreven in sectie 3.3 en de conclusies worden voorgesteld in sectie 3.4.

Hoofdstuk 4: Kwaliteitsmodellen

De doelstelling van 3D grafische toepassingen is de beste visuele kwaliteit leveren rekening houdend de beperkingen van het gebruikte toestel. Het bepalen van de kwaliteit gebeurt in interactieve toepassingen door een heuristiek die de visuele kwaliteit van de uiteindelijke afbeelding inschat aan de hand van een aantal eenvoudige parameters. Omdat 3D grafische toepassingen interactieve toepassingen zijn, is eenvoud een belangrijke voorwaarde voor de bruikbaarheid van deze heuristiek. De uitdaging is dus om de beste, maar eenvoudigste kwaliteitsmaat te vinden.

Omdat een eenvoudige evaluatie van de kwaliteitsmaat noodzakelijk is in interactieve toepassingen, wordt de nauwkeurigheid van het kwaliteitsmodel dikwijls opgeofferd voor eenvoud. Nochtans bestaan er realistische kwaliteitsmaten die het verschil (de fout) tussen twee modellen van 3D objecten kunnen opmeten. Deze kwaliteitsmaten worden bijvoorbeeld gebruikt bij het vergelijken van algoritmen om 3D objecten te vereenvoudigen. Aan deze kwaliteitsmaten zijn een paar nadelen verbonden. De complexiteit van deze maten is namelijk te groot om rechtstreeks gebruikt te worden in interactieve toepassingen. Daarnaast vergelijken ze het volledige object zonder rekening te houden met de manier waarop een gebruiker naar het object kijkt, waardoor sterk afwijkende maar slecht zichtbare delen van het object een te grote invloed krijgen in de maat.

Wij stellen daarom voor om de complexe delen van deze kwaliteitsmaat te berekenen in een voorbereidende faze [1]. Het eigenlijke werk wordt daardoor beperkt tot een eenvoudige combinatie van de zichtbare delen van de 3D objecten [2].

[1] Oplossing voor het eerste nadeel.
[2] Oplossing voor het tweede nadeel.

In een eerste sectie van hoofdstuk 4 wordt een kort overzicht gegeven van de parameters die de kwaliteit van 3D grafische toepassingen beïnvloeden. De beschikbare heuristieken worden in sectie 4.2 besproken en in sectie 4.3 wordt ons gezichtspuntafhankelijke kwaliteitsmodel voorgesteld. Ten slotte wordt dit hoofdstuk beëindigd met de conclusies in sectie 4.5.

Hoofdstuk 5: Multi-resolutie representaties

Het optimalisatie-algoritme, behandeld in hoofdstuk 2, kent een budget van driehoeken toe aan ieder object in de virtuele wereld. In dit hoofdstuk bekijken we verschillende technieken die gebruikt kunnen worden om de geometrische structuur van een object aan te passen zodat het opgebouwd is met het juiste aantal driehoeken. Mede omdat toegang tot de 3D data over een netwerk vereist is, hebben we gekozen voor een multiresolutie-techniek met een uitstekende compressie: MPEG-4 Wavelet Subdivision Surfaces (WSS). WSS laten echter enkel toe om het aantal driehoeken te vermenigvuldigen of te delen met een factor 4. Daarom hebben we de bestaande 'MPEG-4 decoder' uitgebreid zodat ook niet-uniforme WSS worden ondersteund. Deze tonen de belangrijke delen van een 3D object op een hogere resolutie en de minder belangrijke delen op een lagere resolutie.

In sectie 5.1 worden eerst de bestaande technieken voor multiresolutie-representaties van 3D objecten beschreven. Met sectie 5.2 worden dan de nodige bouwblokken toegevoegd aan de standaard 'MPEG-4 decoder' om een niet-uniforme decoder te bekomen. De complexiteit van onze voorgestelde decoder en de bruikbaarheid van onze methoden voor het optimalisatie-algoritme komen aan bod in sectie 5.3. In sectie 5.4 stellen we een nieuwe methode voor die een belangrijk deel van de CPU-belasting elimineert. Ten slotte volgt sectie 5.5 met de conclusies.

Hoofdstuk 6: De optimalisatie voor opeenvolgende beelden

In dit hoofdstuk 6 worden alle voorgaande hoofdstukken samengebracht tot één volledig systeem. We illustreren ook hoe de verschillende methoden om 3D objecten aan te passen, beschreven in hoofdstuk 5), gebruikt moeten worden. Een niet-uniforme aanpassing van de objecten laat een zeer nauwkeurige instelling van het aantal driehoeken toe, maar heeft ook de grootste complexiteit. Het is daarom aan te raden de aanpassingen te verdelen over verschillende opeenvolgende beelden. Het succes van deze verdeling is echter afhankelijk van de variaties in de gevisualiseerde 3D virtuele wereld. De laagste complexiteit wordt bekomen wanneer de aanpassing met kleine stappen en dus traag gebeurt. Als de veranderingen in de gevisualiseerde data echter groot zijn is het mogelijk dat het beeld volledig veranderd is voordat de

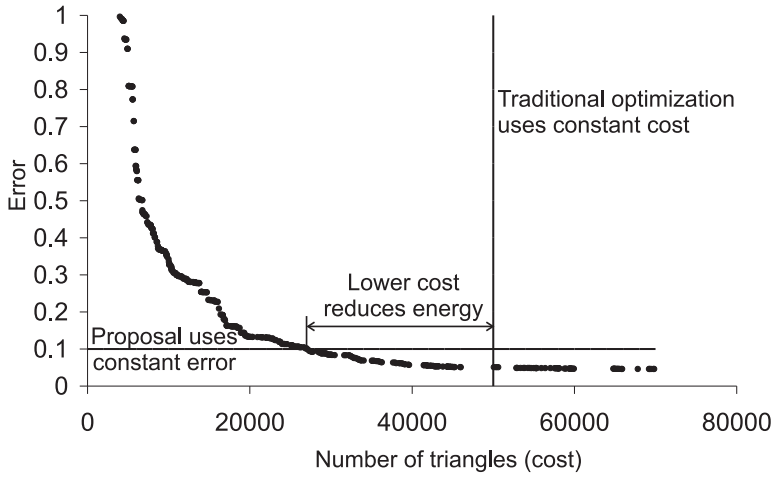
aanpassingen afgelopen zijn. Dit resulteert vaak in 'overshoot' en oscillaties. In deze situaties moet de stapgrootte vergroot worden, wat de snelheid verbetert maar ook de complexiteit vergroot. Bij zeer grote stappen is het daarom beter om over te schakelen naar uniforme aanpassingen die heel eenvoudig zijn, maar die het aantal driehoeken minder nauwkeurig instellen.

In sectie 6.1 wordt het blokdiagram van het volledige systeem gepresenteerd om de data aan te passen aan de mogelijkheden van het toestel. De secties 6.2 en 6.3 behandelen de controle van de rekentijd met niet-uniforme en uniforme 'Wavelet Subdivision Surfaces'. In deze secties worden ook de resultaten van de gecontroleerde rekentijd gegeven. In de laatste sectie 6.5 volgen de conclusies.

Hoofdstuk 7: Uitbreiding van het algoritme, controle van het energieverbruik

Mobiele toestellen, zoals zakcomputers en mobiele telefoons, zijn voldoende krachtig geworden om eenvoudige 3D toepassingen toe te laten. Het meeste onderzoek in het domein van 3D visualisatie voor mobiele toestellen concentreert zich heden op het ontwikkelen van architecturen die een maximale performantie hebben voor een minimaal energieverbruik. Er is echter nog veel werk nodig om de visualisatie van 3D data met een hoge kwaliteit toe te laten op mobiele toestellen. Vanuit het applicatiestandpunt is het een uitdaging om de beste kwaliteit te genereren met de beperkte middelen van de mobiele telefoon. De algoritmen die dit optimalisatieprobleem oplossen zijn echter ontwikkeld voor toestellen die aangesloten zijn op het elektriciteitsnet en houden dus geen rekening met het energieverbruik.

In hoofdstuk 7 stellen we daarom een uitbreiding voor van het optimalisatie-algoritme om het energieverbruik te minimaliseren. Omdat het veranderen van de kwaliteit voor de gebruiker storend kan zijn, kiezen we er voor om de kwaliteit van de 3D virtuele wereld constant te houden. De kwaliteitswaarde kan door de gebruiker zelf bepaald worden. Door de constante kwaliteit varieert de belasting van het systeem, met een minimalisatie van het energieverbruik tot gevolg. Figuur B.4 illustreert het verschil tussen ons voorstel en bestaande optimalisatietechnieken. De traditionele optimalisatietechnieken verhogen de visuele kwaliteit tot de maximale rekentijd bereikt is. Deze rekentijd wordt in figuur B.4 voorgesteld met het aantal driehoeken. Deze techniek resulteert vaak in een minimale verhoging van de visuele kwaliteit bij een sterke verhoging van de rekentijd. Het is echter vaak beter om de visuele kwaliteit op een constant niveau te houden en zo de lagere rekentijd te gebruiken om het energieverbruik te laten dalen. De voorbeelden in deze thesis tonen een energiewinst van 24% als een hoge kwaliteit nodig is en een energiewinst van 70% indien de kwaliteit verlaagd mag worden. De keuze van de kwaliteit hangt volledig van de gebruiker af.



Figuur B.4: Het verschil tussen de voorgestelde en traditionele optimalisatie-algoritmen. De voorgestelde methode houdt de kwaliteit constant en gebruikt de lagere rekentijd om de energie te minimaliseren terwijl de traditionele methoden de kwaliteit maximaliseren

In sectie 7.1 beschrijven we complementair werk aan het onze op het vlak van energieverbruik. De resultaten van dit hoofdstuk werden bekomen met een simulator die beschreven is in sectie 7.2. De uitbreidingen van het optimalisatie-algoritme zijn beschreven in sectie 7.3 en ten slotte worden de conclusies gepresenteerd in sectie 7.4.

Bibliography

[1] T. Akenine-Möller and J. Ström. Graphics for the masses: A hardware rasterization architexture for mobile phones. In *SIGGRAPH 03, Computer Graphics Proceedings*, volume 22, pages 801–808, 2003.

[2] Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. In *SIGGRAPH 99, Computer Graphics Proceedings*, pages 307–316, 1999.

[3] I. Antochi, B.H.H. Juurlink, A. G. M. Cilio, and P. Liuha. Trading efficiency for energy in a texture cache architecture. In *Proceedings of the 2002 Euromicro conference on Massively-parallel computing systems*, pages 189–196, April 2002.

[4] ARM. Arm 3d graphics acceleration flyer. available at: www.arm.com/products/solutions/3DGraphics.html.

[5] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. Mesh: Measuring errors between surfaces using the hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, volume I, pages 705 – 708, 2002.

[6] ATI. Ati imageon 2300. available at: www.ati.com/products/imageon2300.pdf.

[7] M. Avilés, F. Morá, and N. Garcia. Progressive lower trees of wavelet coefficients: Efficient spatial and snr scalable coding of 3d models. In *PCM '05*, 2005.

[8] F. Benichou and G. Elber. Output sensitive extraction of silhouettes from polygonal geometry. In *PG '99: Proceedings of the 7th Pacific Conference on Computer Graphics and Applications*, page 60, 1999.

[9] H. Biermann, A. Levin, and D. Zorin. Piecewise smooth subdivision surfaces with normal control. In *SIGGRAPH '00: Computer Graphics Proceedings*, pages 113–120, 2000.

[10] S. Bischoff, L. P. Kobbelt, and H. P. Seidel. Towards hardware implementation of loop subdivision. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 41–50, 2000.

[11] S. Biyd and L. Vandenberghe. *Convex optimization.* Cambridge University Press, 2004.

[12] E.H. Blake. A metric for computing adaptive detail in animated scenes using object-oriented programming. In *Eurographics Conference Proceedings*, pages 295–307, 1987.

[13] D. Blythe and A. Munshi. *OpenGL ES Common/Common-Lite Profile Specification version 1.1.02.* The Khronos group, 2004.

[14] J. Bolz and P. Schröder. Rapid evaluation of catmull-clark subdivision surfaces. In *Web3D '02: Proceedings of the ninth international conference on 3D Web technology*, pages 11–17, 2002.

[15] J. Bolz and P. Schroöder. Evaluation of subdivision surfaces on programmable graphics hardware. available at: www.multires.caltech.edu/pubs/, 2004.

[16] G. Bontempi. Modeles stochastiques ii. Technical report, Université Libre de Bruxelles, 2004-2005.

[17] G. Bontempi and M. Birattari. From linearization to lazy learning: A survey of divide-and-conquer techniques for nonlinear control. *International journal of computational cognition*, 3(1), 2005.

[18] G. Bontempi and G. Lafruit. Enabling multimedia qos control with black-box modelling. In W. Liu I.D. Bustard and R.Sterritt, editors, *Software 2002: Computing in an Imperfect World, Lecture Notes in Computer Sciene*, pages 46–59, 2002.

[19] J. Bormans, J. Gelissen, and A. Perkis. Mpeg-21: The 21st century multimedia framework. *IEEE Signal Processing Magazine*, 20:53–62, 2003.

[20] B. Bougard, G. Lenoir, W. Eberle, F. Catthoor, and W. Dehaene. A new approach to dynamically trade off performance and energy consumption in wireless communication systems. In *IEEE Symposium on Signal Processing Systems (SiPs'03)*, 2003.

[21] S. Boyd and L. Vandenberghe. *Convex optimization.* Cambridge University press, 2004.

[22] S. Brandt, G. Nutt, T. Berk, and M. Humphrey. Soft real-time application execution with dynamic quality of service assurance. In *Sixth International Workshop on Quality of Service (IWQoS 98)*, pages 154–163, 1998.

[23] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Transanctions on Graphics*, 23(3):777–786, 2004.

[24] F. Morán Burgos. Hierarchical 3d mesh coding with subdivision surfaces. In *IWSNHC3DI'99*, pages 189–192, 1999.

[25] F. Morán Burgos. *Hierarchical Modelling of 3D Objects with Subdivision Surfaces*. PhD thesis, Universidad Politécnica de Madrid, 2001.

[26] F. Morán Burgos, P. Gioia, M. Steliaros, M. Bourges-Sevenier, and N. Garcia. Subdivision surfaces in mpeg-4. In *Proceedings of the International Conference on Image Processing*, pages III–5– III–8, 2002.

[27] Marco Campi. Performance of rls identification algorithms with forgetting factor: a phi-mixing approach. *Journal of Mathematical Systems, estimation and control*, 4(3):1–25, 1994.

[28] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355, 1978.

[29] C. Chang and S. Ger. Enhancing 3d graphics on mobile devices by image-based rendering. In *IEEE Pacific Rim Conference on Multimedia*, pages 1105–1111, 2002.

[30] The OpenGL Performance Characterization. Specviewperf and specglperf.

[31] S. Chen and K. Nahrstedt. An overview of quality-of-service routing for the next generation high- speed networks: Problems and solutions. *IEEE Network Magazine, Special Issue on Transmission and Distribution of Digital Video*, pages 64–79, 1998.

[32] Y. Chen, Z. Zhong, T. Lan, S. Peng, and Kees van Zon. Regulated complexity scalable mpeg-2 video decoding for media processors. *IEEE transactions on circuits and systems for video technology*, 12(8):678–688, august 2002.

[33] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. In *Computer Graphics Forum*, pages 167–174, June 1998.

[34] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[35] Futuremark corporation. 3d mark. available at: www.futuremark.com.

[36] D. Crisu, S. D. Cotofana, S. Vassiliadis, and P. Liuha. Graal - a development framework for embedded graphics accelerators. In *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pages 1366–1367, February 2004.

[37] S. Daly. The visible differences predictor: An algorithm for the assessment of image fidelity. *Digital Images and Human Vision, MIT press*, pages 179–206, 1993.

[38] J. Diepstraten, M. Görke, and T. Ertl. Remote line rendering for mobile devices. In *Proceedings of Computer Graphics International (CGI'04)*, pages 454–461, 2004.

[39] D. Doo and M. Sabin. Analysis of the behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design*, 10(6):356–360, 1978.

[40] F. Duguet and G. Drettakis. Flexible point-based rendering on mobile devices. *IEEE Computer Graphics and Applications*, 24(4), July-August 2004.

[41] J. Euh, J. Chittamuru, and W. Burleson. Power-aware 3d computer graphics rendering. *The Journal of VLSI Signal Processing*, 39:15–33, 2005.

[42] R. Fernando, editor. *GPU Gems Programming Techniques, Tips, and tricks for Real-Time Graphics*. Addison-Wesley, 2004.

[43] R. Fernando and M. J. Kilgard. *The Cg Tutorial, The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[44] J. A. Ferwerda. Elements of early vision for computer graphics. *IEEE Computer Graphics and Application*, 21(5):22–33, 2001.

[45] J. A. Ferwerda, S. N. Pattanaik, P. Shirley, and D. P. Greenberg. A model of visual masking for computer graphics. In *SIGGRAPH 97, Computer Graphics Proceedings*, pages 143–152, August 1997.

[46] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics principles and practice*. The System Programming Series. Addison-Wesley, 1990.

[47] MPEG: Moving Picture Experts Group; formally ISO/ IEC JTC1/ SC29/ WG11. *ISO/IEC 14496 Information technology - Coding of audio-visual objects - Part 16: Animation Framework eXtension (AFX)*. ISO/IEC standard, 2003.

[48] MPEG: Moving Picture Experts Group; formally ISO/ IEC JTC1/ SC29/ WG11. *ISO/IEC 14496 Information technology - Coding of audio-visual objects - Part 10: Advanced Video Coding*. ISO/IEC standard, 2004.

[49] MPEG: Moving Picture Experts Group; formally ISO/ IEC JTC1/ SC29/ WG11. *ISO/IEC 21000 Information technology - Multimedia framework (MPEG-21) - Part 10: Digital Item Adaptation*. ISO/IEC standard, 2004.

[50] MPEG: Moving Picture Experts Group; formally ISO/IEC JTC1/SC29/ WG11. *ISO/IEC 14496 Information technology - Coding of audio-visual objects - Part 2: Visual*. ISO/IEC standard, 2001.

[51] JPEG: Joint Photographic Experts Group; formally ISO/IEC SC29/ WG1. *ISO/IEC 15444 - JPEG 2000 image coding system - Part 1: Core coding system*. ISO/IEC standard, 2000.

[52] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithms for interactive frame rates during visualization of complex virtual environments. In *SIGGRAPH 93, Computer Graphics Proceedings*, pages 247–254, August 1993.

[53] T. A. Funkhouser, C. H. Squin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *1992 Symposium on Interactive 3D Graphics*, pages 11–20, 1992.

[54] M. Garland and P. Heckbert. Surface Simplification Using Quadric Error Metrics. Surface simplification using quadric error metrics. In *SIGGRAPH 97, Computer Graphics Proceedings*, 1997.

[55] G.Lafruit, N. Pham Ngoc W., Van Raemdonck N., Tack J., and Bormans. Terminal qos for real-time 3d visualization using scalable mpeg-4 coding. *IEEE Trans. Circuits and Systems for Video Technology*, 13(11):1136–1143, 2003.

[56] E. Gobbetti and E. Bouvier. Time-critical multiresolution scene rendering. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 123–130, 1999.

[57] E. Gobbetti and E. Bouvier. Time-critical multiresolution rendering of large complex models. *Journal of Computer-Aided Design*, 32(13):785–803, 2000.

[58] GPGPU. General purpose programming using graphics hardware. available at: www.gpgpu.org.

[59] I. Guskov, K. Vidimče, W. Sweldens, and P. Schröder. Normal meshes. In *SIGGRAPH '00: Computer Graphics Proceedings*, pages 95–102, 2000.

[60] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th ISCA*, June 1997.

[61] B. Hamman. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11:197–214, 1994.

[62] C. Hentschel, R. J. Bril, Y. Chen, R. Braspenning, and Tse-Hua Lan. Video quality-of-service for consumer terminals - a novel system for programmable components. *IEEE Transactions on Consumer Electronics*, 49(4):1367–1377, 2003.

[63] F.S. Hill. *Computer Graphics Using Open GL.* Prentice-Hall, inc., 2001.

[64] H. Hoppe. Progressive meshes. In *SIGGRAPH 92, Computer Graphics Proceedings*, pages 99–108, 1996.

[65] H. Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH 97, Computer Graphics Proceedings*, pages 189–198, 1997.

[66] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *SIGGRAPH 93, Computer Graphics Proceedings*, pages 19–26, 1993.

[67] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH 02, Computer Graphics Proceedings*, pages 693–702, 2002.

[68] T. Ibaraki, T. Hasegawa, K. Teranaka, and J. Iwase. The multiple choice knapsack problem. *Journal of Operations Research Society of Japan*, 21:59–94, 1978.

[69] IBM. White paper: Ibm rational purifyplus. available at: www3.software.ibm.com/ibmdl/pub/software/rational/web/datasheets/version6/pplus.pdf.

[70] Silicon Graphics Inc. Graphics library programming tools and techniques. Document #007-1489-01, 1992.

[71] P. A. Ioannou and J. Sun. *Robust Adaptive Control.* Prentice Hall, Inc, 1996.

[72] K. Iourcha, K. Nayak, and Z. Hong. System and method for fixed-rate block-based image compression with inferred pixel values. US Patent 5,956,431.

[73] C.R. Johnson. *Lectures on Adaptive Parameter Estimation.* Prentice Hall, 1988.

[74] S. Khan. *Quality adaptation in a multisession multimedia system: Model, algorithms and architecture.* PhD thesis, University of Victoria, 1998.

[75] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. In *SIGGRAPH 00, Computer Graphics Proceedings*, pages 271–278, 2000.

[76] H. Kim, C. Joslin, T. Di Giacomo, S. Garchery, and N. Magnenat-Thalmann. Adaptation mechanism for three dimensional content within the mpeg-21 framework. In *Proceedings of Computer Graphics International (CGI'04)*, pages 462–469, 2004.

[77] D. Kirk. Geforce 3 architecture overview. available at: developer.nvidia.com.

[78] R. Koenen. Overview of the mpeg-4 standard. available at: www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm, March 2002.

[79] F. Lamberti, C. Zunino, A. Sanna, A. Fiume, and M. Maniezzo. An accelerated remote graphics architecture for pdas. In *Proceedings of ACM Web3D symposium*, pages 55–62, 2003.

[80] A. LaMothe. *Tricks of the 3D Game Programming Gurus: Advanced 3D Graphics and Rasterization*. Sams publishing, 2003.

[81] A. Lee, H. Moreton, and H. Hoppe. Displaced subdivision surfaces. In *Proceedings of ACM SIGGRAPH 2000*, pages 85–94, 2000.

[82] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. *Computer Graphics*, 32(Annual Conference Series):95–104, 1998.

[83] P. Leroy. Pocketgl. available at: www.sundialsoft.freeserve.co.uk/.

[84] B. Li. An analysis and comparison of two visual discrimination models. Master's thesis, University of Oregon, 1997.

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

[86] C. T. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, 1987.

[87] J. Lubin. A visual discrimination model for imaging system design and evaluation. *Vision Models for Target Detection and Recognition, World Scientific*, pages 245–283, 1995.

[88] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97, Computer Graphics Proceedings*, pages 199–208, 1997.

[89] D. Luebke, M. Reddy, J.D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail For 3D Graphics*. Elsevier Science (USA), 2003.

[90] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Symposium on Interactive 3D Graphics*, pages 95–102, 211, 1995.

[91] M. Mattavelli and S. Brunetton. Implementing real-time video decoding on multimedia processors by complexity prediction techniques. *IEEE Transactions on Consumer Electronics*, 44(3):760–767, 1998.

[92] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconigurable matrix. In *FPL*, 2003.

[93] Microsoft. Pix. The DirectX Software Development Kit, version 9.0.

[94] Microsoft. Windows directx and direct3d. available at: msdn.microsoft.com/ library.

[95] Microsoft. Windows mobile directx and direct3d. available at: msdn.microsoft. com/library.

[96] Microsoft. Accurately profiling direct3d api calls. available at: msdn.microsoft. com/library, February 2005.

[97] R. Mohan, J. R. Smith, and C. Li. Adapting multimedia internet content for universal access. *IEEE Transactions On Multimedia*, 1(1):104–114, 1999.

[98] D.S. Moor and G. P. McCabe. *Introduction to the Practice of Statistics; third edition*. W.H. Freeman and Company, 1999.

[99] T. A. Mŏller and E. Haines. *Real-Time Rendering, Second Edition*. A.K. Peters Ltd, 2002.

[100] N. P. Ngoc. *QoS Management at end Systems for Advanced Multimedia Applications*. PhD thesis, Katholieke Universiteit Leuven, 2004.

[101] N. Pham Ngoc, W. Van Raemdonck, G. Lafruit, G. Deconinck, and R. Lauwereins. A qos framework for interactive 3d applications. In *10th International Conference on Computer Graphics and Visualization.*, pages 317–324, 2002.

[102] T. D. Nielsen. A very brief note about gradient descent search. www.cs.auc. dk/ tdn/Teaching/BSS04/Notes/gradient.pdf, 2004.

[103] NVidia. Handheld goforce 3d core. available at: www.nvidia.com/page/ goforce_3d_4500.html.

[104] NVidia. Nvidia goforce 3d technology. available at: www.nvidia.com/page/ handheld.html.

[105] NVidia. Nvidia sdk. available at: developer.nvidia.com.

[106] NVidia. User guide: Nvperfhud 2.0 heads-up display for performance analysis. available at: developer.nvidia.com.

[107] NVidia. User guide: Nvperfkit 1.0 nvidia performance toolkit. available at: developer.nvidia.com.

[108] Opensg.org. The opensg forum white paper. available at: www.opensg.org/.

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

[110] Bitboys Oy. The g34 and g40 graphics processors. available at: www.bitboys.com/products.php.

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

[112] R. Pajarola and J. Rossignac. Squeeze: Fast and progressive decompression of triangle meshes. In *Proceedings of Computer Graphics International Conference*, page 173182, 2000.

[113] V. Pareto. Cours deconomie politique. Cours l'universit de Lausanne, 1896.

[114] B. Paul. The mesa 3d graphics library. available at: www.mesa3d.org.

[115] J. Peng, C.S. Kim, and C.J. Kuo. Technologies for 3d mesh compression: A survey. *Journal of Visual Communication and Image Representation*, 2003.

[116] S. Peng. Complexity scalable video decoding via idct data pruning. In *IEEE International Conference on Consumer Electronics Digest of Technical Papers*, pages 74–75, 2001.

[117] M. Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley, 2005.

[118] PowerVR. The powervr mbx core. available at: www.powervr.com.

[119] M. Reddy. *Perceptually Modulated Level of Detail for Virtual Environments.* PhD thesis, University of Edinburgh, 1997.

[120] J. Reichel. *Complexity Related Aspects of Image Compression.* PhD thesis, École polytechnique fédérale de Lausanne, 2001.

[121] J. Rohlf and J. Helman. Iris performer: A high performance multiprocessing toolkit for realtime 3d graphics. In *SIGGRAPH 94, Computer Graphics Proceedings*, pages 381–395, July 1994.

[122] J. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In *Conference on Geometric Modeling in Computer Graphics*, pages 455–465, 1993.

[123] M. Roy, S. Foufou, and F. Truchetet. Generic attribute deviation metric for assessing mesh simplification algorithm quality. In *Proceedings of IEEE International Conference on Image Processing*, pages 817–820, September 2002.

[124] S. Rusinkiewicz and M. Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Computer Graphics Proceedings*, pages 343–352, 2000.

[125] R. H. Saavedrea-Barrera. *CPU performance evaluation and execution time prediction using narrow spectrum benchmarking.* PhD thesis, University of California at Berkeley, 1992.

[126] B. Sabata, S. Chatterjee, M. Davis, J. J. Sydir, and T. F. Lawrence. Taxonomy for qos specifications. In *IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 1997.

[127] A. Said and W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6:243–250, june 1996.

[128] I. A. Salomie. *Extraction, Hierarchical Representation and Flexible Compression of Surface Meshes Derived from 3D Data.* PhD thesis, Vrije Universiteit Brussel, 2005.

[129] I. A. Salomie, A. Munteanu, A. Gavrilescu, G. Lafruit, P. Schelkens, R Deklerck, and J. Cornelis. Meshgrid a compact, multiscalable and animation-friendly surface representation. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY*, 14(7):950–966, 2004.

[130] P. Sander, J. Snyder, S. Gortler, and H. Hoppe. Texture mapping progressive meshes. In *SIGGRAPH 01, Computer Graphics Proceedings*, pages 409–416, 2001.

[131] J. Sanneblad and L. E. Holmquist. Designing collaborative games on handheld computers. In *Sketches and Applications program of ACM 2003 Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2003)*, 2003.

[132] Bruce J. Schachter. *Computer Image Generation.* Krieger Publishing Co., Inc., 1983.

[133] W. J. Schroeder, J. A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH 92, Computer Graphics Proceedings*, pages 65–70, 1992.

[134] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (version 1.5).* Silicon Graphics inc., 1993-2003.

[135] J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *Graphics Hardware workshop 2004*, 2004.

[136] G. Shen, G. Gao, S. L., H. Shum, and Y. Zhang. Accelerate video decoding with generic gpu. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):685–693, 2005.

[137] J. Sohn, J. Woo, M. Lee, H. Kim, R. Woo, and H. Yoo. A 50mvertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. In *Solid-State Circuits Conference, ISSCC 2005*, 2005.

[138] H. Sowizral, K. Rushforth, and M. Deering. *The Java 3D (TM) API Specification (2nd Edition)*. Addison-Wesley, 2000.

[139] A. Stevens. Arm 3d graphics solutions white paper. available at: www.arm.com.

[140] Superscape. Swerve3d. available at: www.superscape.com.

[141] N. Tack. Performance modelling for the geforce fx 5900. Technical report, IMEC/KU. Leuven, 2003.

[142] N. Tack, G. Lafruit, F. Catthoor, and R. Lauwereins. A content quality driven energy management system for mobile 3d graphics. In *IEEE Workshop on Signal Processing Systems (SIPS)*, pages x–y, 2005.

[143] N. Tack, G. Lafruit, F. Catthoor, and R. Lauwereins. Eliminating cpu overhead for on-the-fly content adaptation with mpeg-4 wavelet subdivision surfaces. *submitted to IEEE Transaction on Consumer Electronics*, 2005.

[144] N. Tack, G. Lafruit, F. Catthoor, and R. Lauwereins. Pareto based optimization of multi-resolution geometry for real time rendering. In *Proceedings of ACM Web3D symposium*, pages 19–27, 2005.

[145] N. Tack, G. Lafruit, F. Catthoor, and R. Lauwereins. Platform independent optimization of multi-resolution 3d content for enabling universal media access. *submitted to The Visual Computer*, 2005.

[146] N. Tack, G. Lafruit, and E. Delfosse. Complexity assessement of mpeg-4 mesh subdivision tools. MPEG contribution: ISO/IEC JTC1/SC29/WG11/MPEG02, 2002.

[147] N. Tack, G. Lafruit, and R. Lauwereins. Visual and complexity analysis of the extended loop subdivision scheme. In *International Conference on Geometric Modeling and Graphics (GMAG)*, pages 168–174, 2003.

[148] N. Tack, F. Moran, G. Lafruit, and R. Lauwereins. Rendering time modeling and control for mobile terminals. In *Proceedings of ACM Web3D symposium*, pages 109–117, 2004.

[149] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[150] P. Trancoso and M. Charalambous. Exploring graphics processor performance for general purpose applications. In *Euromicro conference on digital system design, DSD 2005*, 2005.

[151] Stanford University. The stanford 3d scanning repository. available at: graphics.stanford.edu/data/3Dscanrep/.

[152] A. Vahidi, A. Stefanopoulou, and H. Peng. Recursive least squares with forgetting for online estimation of vehicle mass and road grade: Theory and experiments. *to appear in Journal of Vehicle Dynamics*.

[153] S. Venkatasubramanian. The graphics card as a stream computer. Technical report, AT & T Labs Research, 2004.

[154] A. Vetro and C. Timmerer. Digital item adaptation: Overview of standardization and research activities. *IEEE Transactions on Multimedia*, 7(3):418–426, 2005.

[155] J. Wang, T. T. Wong, P. A. Heng, and C. S. Leung. Discrete wavelet transform on gpu. In *ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–41, 2004.

[156] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):204–241, 2004.

[157] B. Watson, A. Friedman, and A. McGaffey. Measuring and predicting visual fidelity. In *SIGGRAPH 01, Computer Graphics Proceedings*, pages 213–220, 2001.

[158] Web3D. *X3D framework - Part 1: Architecture and base components - FPDAM Amendment 1 - ISO/IEC 19775-1 (X3D functional specification).* ISO/IEC standard, 2005.

[159] H.M. Will. Vincent: A 3d rendering library for mobile devices. available at: ogl-es.sourceforge.net/.

[160] N. Williams, D. Luebke, J. Cohen, M. Kelley, and B. Schubert. Perceptually guided simplification of lit, textured meshes. In *Proceedings of the 2003 ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 2003.

[161] M. Wimmer and D. Schmalstieg. Load balancing for smooth lods. Technical report, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1998.

[162] M. Wimmer and P. Wonka. Rendering time estimation for real- time rendering. In *Proceedings of the Eurographics Symposium on Rendering*, pages 118–129, 2003.

[163] C. Wong, P. Marchal, and P. Yang. Task concurrency management methodology to schedule the mpeg4 im1 player on a highly parallel processor platform. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 170–177, 2001.

[164] R. Woo, S. Choi, J. Sohn, S. Song, and H. Yoo. A 210-mw graphics lsi implementing full 3-d pipeline with 264 mtexels/s texturing for mobile multimedia applications. *Solid-State Circuits, IEEE Journal of*, 39(2):358 – 367, Feb 2004.

[165] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 327–335., 1996.

[166] D. Xu, D. Wichadakul, and K. Nahrstedt. Resource-aware configuration of ubiquitous multimedia services. In *IEEE International Conference on Multimedia and Expo (II)*, pages 851–854, 2000.

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

[168] D. Zorin, P. Schröder, and W. Sweldens. Interpolating subdivision for meshes with arbitrary topology. In *SIGGRAPH '96: Computer Graphics Proceedings*, pages 189–192, 1996.

[169] D. Zorin, P. Schröder, and W. Sweldens. Interactive multiresolution mesh editing. In *SIGGRAPH '97: Computer Graphics Proceedings*, pages 259–268, 1997.

Nicolaas Tack was born on January 26, 1976, in Kortrijk. He received the B.S. degree from the Katholieke Hogeschool Brugge Oostende, Belgium in June 1998 and the M.Eng. degree in electrical engineering from the Katholieke Universiteit Leuven in June 2001. In August 2001, he started as a researcher at the Multimedia group of the DESICS department of IMEC, where he worked towards the Ph.D. degree in electrical engineering at the Katholieke Universiteit Leuven on an I.W.T.-Vlaanderen (Institute for the promotion of Innovation through Science and Technology in Flanders) grant. His research interests focus on terminal QoS and resource management for three-dimensional scalable coding and rendering applications.