UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARILENA MAULE

Multi-Fragment Visibility Determination in the Context of Order-Independent Transparency Rendering

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Prof. PhD. João Luiz Dihl Comba
Advisor

Prof. PhD. Rafael Piccin Torchelsen
Coadvisor

PhD. Rui Manuel Ribeiro de Bastos
Coadvisor

Porto Alegre, March 2015

CIP – CATALOGING-IN-PUBLICATION

Maule, Marilena

Multi-Fragment Visibility Determination in the Context of Order-Independent Transparency Rendering / Marilena Maule. – Porto Alegre: PPGC da UFRGS, 2015.

133 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: João Luiz Dihl Comba; Coadvisor: Rafael Piccin Torchelsen. ; Coadvisor: Rui Manuel Ribeiro de Bastos.

1. Visibility. 2. Multi-fragment rendering. 3. Order-independent transparency. 4. Anti-aliasing. 5. Depth-fighting. I. Comba, João Luiz Dihl. II. Torchelsen, Rafael Piccin. III. Bastos, Rui Manuel Ribeiro de.

“So long, and thanks for all the fish.”
— DOLPHINS

ACKNOWLEDGEMENTS

I would like to express my gratitude...

- To my family, for the constant support. Specially to my father, Tanquilo Maule, and my sister, Ivania Maule.

- To my advisors, Professors João Luiz Dihl Comba, and Rafael Piccin Torchelsen, and Ph.D. Rui Bastos, for the good will of sustaining the prolonged meetings every Tuesday, for the patience in those times when not even I had patience with myself, and for guiding me well beyond their academic responsibilities.

- To my colleagues and friends in the Computer Graphics group, for the chatting, ideas, suggestions, debuggings, and proof-reading of my papers. Special thanks to Bernardo Hens and Vinicius Azevedo for the proof-reading of this dissertation.

- To the members of the examination board, Professors Waldemar Celes Filho (PUC-RJ), Marcelo Walter (UFRGS), Manuel Menezes de Oliveira Neto (UFRGS), and Adriane Parraga (UERGS), for the tips for improving this thesis text and presentation.

- To Inf-UFRGS, for providing and maintaining the infrastructure where my research was developed. And, also, for the additional financial support.

- To NVIDIA for the donation of a GeForce GTX TITAN to my research.

- To CNPq for four years of scholarship, and CAPES for the other year.

Ph.D. Thesis - II/UFRGS

ABSTRACT

Multi-fragment effects, in the computer-generated imagery context, are effects that determine pixel color based on information computed from more than one fragment. In such effects, the contribution of each fragment is extracted from its visibility with respect to a point of view. Seen through a pixel's point of view, the visibility of one fragment depends on its spatial relationship with other fragments. This relationship can be reduced to the problem of sorting multiple fragments. Therefore, sorting is the key to multi-fragment evaluation.

The research on this dissertation is focused on two classical multi-fragment effects: order-independent transparency and anti-aliasing of transparent fragments. While transparency rendering requires sorting of fragments along the view ray of a pixel, anti-aliasing increases the problem complexity by adding spatial information of fragments with respect to the pixel area. This dissertation contribution relies on the work towards the development of a solution for the visibility of fragments that can take advantage of the transformation and lighting pipeline implemented in current GPUs.

We describe both transparency and aliasing problems, for which we discuss existing solutions, analyzing, classifying and comparing them. The analysis associates solutions to specific applications, comparing memory usage, performance, and quality. The result is a general view of each field: which are the current state-of-the-art capabilities and in which direction significant improvements can be made.

As part of this dissertation, we proposed two novel techniques for order-independent transparency rendering. We show how to achieve the minimum memory footprint for computing exact transparency in a bounded number of geometry passes; allowing increasing scene complexity and image resolution to be feasible within current hardware capabilities. Additionally, we demonstrate that, for most scenarios, the front-most fragments have the greatest impact on the pixel color. We also show how the perspective we propose has inspired recent transparency techniques.

The research includes the investigation of a novel anti-aliasing approach for transparent fragments. Through the use of a single sample per fragment, we aim at reducing memory footprint while improving performance and quality. Preliminary experiments show promising results, in comparison with a well established and largely used anti-aliasing technique.

Keywords: Visibility, multi-fragment rendering, order-independent transparency, anti-aliasing, depth-fighting.

Tese de Doutorado - II/UFRGS
Determinação de Visibilidade de Efeitos Multi-Fragmentos no Contexto De Transparência Independente de Ordem

RESUMO

No contexto de imagens geradas por computador, efeitos multi-fragmento são aqueles que determinam a cor do pixel baseados em informações computadas a partir de mais de um fragmento. Nesse tipo de efeito, a contribuição de cada fragmento é extraída de sua visibilidade com respeito a um determinado ponto de vista. Observando uma sequencia de fragmentos vista através de um pixel, a visibilidade de um fragmento depende da sua relação espacial com os demais fragmentos. Essa relação pode ser reduzida ao problema de ordenação de múltiplos fragmentos. Portanto, ordenação é essencial para correta avaliação de efeitos multi-fragmento.

A pesquisa desta tese foca em dois problemas multi-fragmento clássicos: transparência independente de ordem e anti-aliasing de fragmentos transparentes. Enquanto o efeito de transparência necessita de ordenação de fragmentos ao longo do raio de visualização do pixel, anti-aliasing aumenta a complexidade do problema ao adicionar informação espacial do fragmento com respeito à área do pixel. A contribuição desta tese é o desenvolvimento de uma solução para visibilidade de fragmentos que pode tirar proveito do pipeline de transformação e iluminação, implementando nas GPUs de hoje.

Nós descrevemos ambos os problemas de transparência e anti-aliasing, discutindo soluções anteriores, além de classificá-las e compará-las. Nossa análise associa soluções a implementações específicas, comparando uso de memória, desempenho e qualidade de imagem. Os documentos resultantes fornecem uma visão geral das áreas abordadas, contendo: qual é o estado-da-arte atualmente, o que ele é capaz de fazer e quais são suas limitações, ou seja, onde melhorias são possíveis.

Como parte integrante desta tese, nós propomos duas novas técnicas para processar transparência independente de ordem. Nós mostramos como obter o menor consumo de memória para cálculo exato de transparência, em um número finito de passos de geometria; permitindo aumento da complexidade das cenas representadas e da resolução da imagem final, em relação aos métodos anteriores, dada uma determinada configuração de hardware. Adicionalmente, demonstramos que, para a maior parte dos casos, os fragmentos mais próximos ao observador tem maior impacto sobre a cor final do pixel. Também mostramos como esta perspectiva sobre o problema inspirou novas técnicas.

A pesquisa também inclui a investigação de uma nova abordagem para anti-aliasing para fragmentos transparentes. Através do uso de uma única amostra por fragmento, nosso objetivo é reduzir o consumo de memória enquanto melhoramos desempenho e qualidade. Experimentos preliminares apresentam resultados promissores em comparação com a técnica mais usada para anti-aliasing.

Palavras-chave: Visibilidade, multi-fragmentos, transparência independente de ordem, anti-aliasing, depth-fighting.

LIST OF ABBREVIATIONS AND ACRONYMS

AA	Anti-aliasing
API	Application programming interface
BTF	Back-to-front
CPU	Central processing unit
CSAA	Coverage-sampled anti-aliasing
DAEAA	Directionally-adaptive edge anti-aliasing
DEAA	Distance-to-edge anti-aliasing
DFB	Dynamic fragment buffer / Deep fragment buffer
DFS	Depth-first search
DLAA	Directionally localized anti-aliasing
DS	Deferred shading
EQAA	Enhanced-quality anti-aliasing
FIFO	First in first out
FPS	Frames per second
FSAA	Full-scene anti-aliasing
FTB	Front-to-back
FXAA	Fast approximate anti-aliasing
GAA	Geometric anti-aliasing
GBAA	Geometry buffer anti-aliasing
GPAA	Geometric post-processing anti-aliasing
GPGPU	General-purpose graphics processing unit
GPU	Graphics processing unit
HT	Hybrid transparency
IAA	Image post-processing anti-aliasing
MLAA	Morphological anti-aliasing
MRT	Multiple render target

MSAA	Multi-sample anti-aliasing
OIT	Order-independent transparency
OGSS	Ordered grid super sampling
PMLAA	Practical morphological anti-aliasing
PPGC	Programa de pós-graduação em computação
RGSS	Rotated grid super sampling
RMW	Read modify write
SPF	Seconds per frame
SM	Stream multiprocessor
SMAA	Enhanced subpixel morphological anti-aliasing
SP	Stream processor
SPA	Streaming processor array
SRAA	Subpixel reconstruction anti-aliasing
SSAA	Super sampling anti-aliasing
T&L	Transformation and lighting
TPC	Texture/processor cluster

LIST OF SYMBOLS

α	absorbance
C	color
δ	coverage
d	average number of transparent layers ($d \leq l$)
Δ	variation
e	eye
f	min(k, d)
H	screen height
i	light intensity
k	buffer entries per pixel
l	number of transparent layers
L_a	absorbed light
L_i	incident light
L_p	propagated light
L_r	reflected light
L_t	transmitted light
m	number of objects
n	number of geometric primitives ($n >> m$)
p	pixel size (Bytes)
S	number of anti-aliasing samples per pixel
s	number of samples per pixel
σ	shininess
τ	transmittance
v	visibility
W	screen width
z	depth

LIST OF FIGURES

1.1 Depth-sorted order required for transparency rendering 18

2.1 CPU x GPU architecture comparison 22
2.2 Transformation and lighting pipeline 23
2.3 Pipeline on GPU 26
2.4 GPU architecture 27

3.1 Light interaction with a medium 34
3.2 Color spectrum and light wavelengths 35
3.3 Color filtering example 35
3.4 Difference between opaque and transparent rendering 36
3.5 Blending evaluation for green, blue, and red triangles 39
3.6 Fragment re-ordering example 42
3.7 A-buffer 42
3.8 Z^3 43
3.9 R-buffer 44
3.10 Stencil-routed A-buffer 45
3.11 FreePipe 46
3.12 Per-pixel linked list 47
3.13 Adaptive transparency 48
3.14 Per-pixel paged linked list 48
3.15 Virtual pixel maps 49
3.16 Dual depth peeling 50
3.17 Bucket sort histogram 51
3.18 Bucket sort color evaluation 51
3.19 K-buffer 52
3.20 Stochastic transparency 54
3.21 Silhouette-opaque transparency 55
3.22 Comparative chart 60
3.23 DFB pipeline 61
3.24 DFB storage 63
3.25 DFB FPS for synthetic quads 65
3.26 DFB memory management costs 66
3.27 DFB performance for dragons scene 67
3.28 DFB performance for scene with dragons and grass 68
3.29 DFB performance for power plant walkthrough 69
3.30 DFB alpha quality evaluation 70
3.31 DFB performance for dragons walkthrough 70

3.32 HT transmittance function 73
3.33 HT transmittance function approximation 75
3.34 HT image quality for hair scene 77
3.35 HT image quality for dragons scene 78
3.36 HT and AT approximations of the transmittance function compared . 78
3.37 HT error estimations 79
3.38 HT performance for hair walkthrough 81
3.39 HT performance for dragons scene 82
3.40 HT performance for power plant walkthrough 82
3.41 HT temporal artifact 83
3.42 Techniques classified by correctness of results 85
3.43 Techniques classified by order-dependency 86
3.44 Techniques classified by the capacity of capturing or approximating the transmittance-to-eye visibility reduction of fragments. 86
3.45 Techniques classified by time coherence 87
3.46 Techniques classified by memory consumption 88
3.47 Techniques classified by best performance 88
3.48 Techniques classified by worst performance 89

4.1 2D aliasing effect 92
4.2 Digital image acquisition (photography) 93
4.3 Aliasing effect in image acquisition 93
4.4 Moiré animation 93
4.5 Aliasing problem in rendering 94
4.6 Aliasing example: jagged edges 94
4.7 Aliasing example: discontinuity 95
4.8 Aliasing example: complex details 95
4.9 Aliasing example: texture 96
4.10 Critical angle for anti-aliasing 97
4.11 SSAA sample distribution 98
4.12 FSAA sampling approaches 100
4.13 MLAA aliasing patterns 101
4.14 DEAA overview 103
4.15 ASAA comparison 108
4.16 ASAA triangle dilation 112
4.17 ASAA square-triangle intersection cases 113
4.18 ASAA triangle coverage over the pixel circle 114
4.19 ASAA pixel coverage from multiple dtes 115
4.20 ASAA exact compositing 116
4.21 ASAA compositing 117
4.22 ASAA sampling comparison with MSAAx8 119
4.23 ASAA zoomed results 121

LIST OF TABLES

3.1 Numerical example of a pixel evaluated with the **correct** BTF order. . . 39
3.2 Numerical example of a pixel evaluated **out-of-depth-order** with the BTF blending equation. 39
3.3 Parameters used by different methods. 40
3.4 Transparency techniques complexity comparison 57
3.5 Hypothetical scene parameterization 60
3.6 DFB performance comparison 64
3.7 Acronyms for transparency rendering techniques. 84

4.1 Anti-aliasing comparison table 105
4.2 ASAA list of symbols 109
4.3 ASAA features comparison with MSAAx8 121
4.4 ASAA result: antena 122
4.5 ASAA result: crane 123
4.6 ASAA result: chess 123

GLOSSARY

A-buffer buffer-based OIT technique. 41, 99
adaptive transparency buffer-based transparency technique. 47
aliasing problem caused by insufficient sampling. 29

bucket sort depth peeling transparency technique. 50

clipping clipping stage of the pipeline. 24
CPU central processing unit. 22
CSAA coverage sampling anti-aliasing. 98

DAEAA directionally-adaptive edge anti-aliasing. 98
depth fighting when more than one fragment of the same pixel are mapped to the same depth. 29
depth peeling multipass over the geometry approach. 49
DLAA directionally localized anti-aliasing. 102

FIFO buffer-based OIT technique class. 44
fragment processor fragment stage of the pipeline. 25
freePipe buffer-based transparency technique. 46

geometry processor geometry stage of the pipeline. 24
Gouraud shading per vertex shading computation. 25
GPU graphics processing unit. 21, 25

image-space queries hybrid transparency technique. 51
interactive order-independent transparency depth peeling OIT technique. 49

k-buffer hybrid transparency technique. 52

linked lists buffer-based OIT technique. 46

MLAA morphological anti-aliasing. 100
MSAA multi sampling anti-aliasing. 97
multi-fragment when more than one fragment contributes to the pixel color. 18

per-pixel paged linked lists buffer-based OIT technique. 48
Phong illumination model model for local illumination. 25
Phong shading per fragment shading computation. 25
pipeline transformation and lighting graphics pipeline. 22
PMLAA practical morphological anti-aliasing. 101

rasterizer rasterization stage of the pipeline. 24

shadow area not reached by light due to obstruction. 29
silhouette-opaque probabilistic transparency technique. 55
SMAA subpixel morphological anti-aliasing. 99
SRAA multi sampling anti-aliasing. 99
SSAA subpixel-reconstruction anti-aliasing. 97
stencil-routed buffer-based transparency technique. 45
stochastic transparency probabilistic transparency technique. 55

transparency property that allows light to pass through materials. 28, 33, 35

vertex processor vertex stage of the pipeline. 23
virtual pixel maps depth peeling OIT technique. 49

weighted average sort-independent transparency technique. 53
weighted sum sort-independent transparency technique. 53

Z^3 buffer-based transparency technique. 43

CONTENTS

1 INTRODUCTION 17
1.1 Problem Statement 17
1.2 Thesis Statement 19
1.3 Thesis Contributions 19
1.4 Document Organization 20

2 PROBLEM CONTEXT AND CHARACTERIZATION 21
2.1 Graphics Pipeline and GPU Architecture 21
2.1.1 Graphics Pipeline 22
2.1.2 GPU Architecture 25
2.2 Multi-Fragment Rendering 27
2.2.1 Multi-Fragment Generation 28
2.2.2 Multi-Fragment Processing: The Sorting Problem 30
2.3 Summary 32

3 TRANSPARENCY RENDERING 33
3.1 Transparency Problem 33
3.1.1 Transparency Physics 33
3.1.2 Transparency Rendering 35
3.2 Transparency Techniques 40
3.2.1 Geometry-Sorting Methods 40
3.2.2 Fragment-Sorting Methods 41
3.2.3 Hybrid Methods 51
3.2.4 Depth-Sorting-Independent Methods 53
3.2.5 Probabilistic Methods 54
3.2.6 Summary Comparison 56
3.3 Dynamic Fragment Buffer 61
3.3.1 Fragment Counting 62
3.3.2 Prefix Sum 62
3.3.3 Storing Shaded Fragments 62
3.3.4 Sorting and Blending Fragments 62
3.3.5 Extended Dynamic Fragment Buffer 63
3.3.6 Experiments 63
3.3.7 Discussion 68
3.4 Hybrid Transparency 72
3.4.1 HT Core 72
3.4.2 HT Tail 73

3.4.3 Core, Tail, and Background Compositing 74
3.4.4 Implementation Details 75
3.4.5 Experiments and Discussion 75
3.4.6 HT Limitation 83
3.5 Classification of Transparency Techniques by Features 84
3.5.1 Image Quality 84
3.5.2 Memory Consumption 87
3.5.3 Performance 88
3.6 Summary 89

4 ANTI-ALIASED RENDERING 91
4.1 Aliasing Problem 91
4.1.1 Aliasing Physics 91
4.1.2 Aliasing in Computer-Generated Images 92
4.2 Anti-Aliasing Techniques and OIT Discussion 96
4.2.1 Full-Scene Anti-Aliasing (FSAA) 96
4.2.2 Image Post-Processing 100
4.2.3 Geometric Anti-Aliasing 103
4.2.4 Discussion 104
4.3 Area-Sample AA for OIT Attempt 107
4.3.1 Area-Sampling Anti-Aliasing 109
4.3.2 Conservative Rasterization 110
4.3.3 Computing Attributes for Dilated Vertices 111
4.3.4 Computing Fragment Coverage Over The Pixel 113
4.3.5 Compositing 115
4.3.6 Experiments and Discussion 118
4.4 Summary 124

5 FINAL CONSIDERATIONS 125

REFERENCES 128

1 INTRODUCTION

Simulation of multi-fragment effects significantly improves realism in computer-generated images, being largely used by the entertainment industry to create animated movies, special effects and games, or used as a visualization tool for exploring virtual structures and data relationships. The high processing time and memory required to produce plausible multi-fragment effects, as transparency, is the reason why such effects are not largely used in real-time applications, such as games. Offline renderings can better afford the price and often use transparency to improve realism—e.g. movies. Despite the demand for efficient transparency algorithms and the intensive research in the area, computer graphics still lack a final solution to the problem.

Transparency rendering is one instance of a class of problems that require visibility computation at more than one level. While the simplest opaque rendering only needs to determine the front-most fragment, multi-fragments effects, such as transparency, need to determine partial visibility for several fragments and, then, compose them together to reach the final pixel color. A very similar problem is the shadow rendering that, as transparency, needs to determine the partial visibility of several fragments with respect to the light instead of the camera. Other examples of multi-fragment visibility problems include:

- Anti-aliasing: in multi-sampling approaches, samples may come from different fragments, and compete for visibility through depth-test.
- Depth-fighting: geometry in the scene may contain co-planarity, or the limited precision of interpolation and depth test may generate co-planarity, in such cases more than one fragment may be the front-most and a semantic must be defined to address this problem. Otherwise, the incoming order of fragments will determine the winner and order-dependent artifacts may appear.
- Constructive Solid Geometry: rendering can be used to perform boolean operations among complex geometries through the approach of continuously find the next visible fragment and warping the new geometry around the results.
- Depth-of-field effect: simulation of fragments outside the depth of field reduces the fragments visibility to simulate a blur effect. By being partially visible, such fragments need to be blended using transparency equations.

1.1 Problem Statement

Our goal is to determine pixel colors based on fragments visibilities . For that, consider a raster-based pipeline implemented in modern GPUs and accessed through a graphics API. Scenes are described in 3D world coordinates, which are transformed by the view and the projection matrices. The resulting geometry is decomposed into a flow of fragments, with 3D coordinates in screen space and attributes interpolated from the original vertices. Implementations of the pipeline guarantee that given the same 3D scene with the same matrices and same API states, the same fragments will be generated, though not necessarily in the same order.

When considering variations of opacity and coverage of fragments, more than one fragment may contribute to the pixel color. This is called *multi-fragment* effect. A fragment may be partially visible in a pixel either if it is transparent or if it partially covers the pixel. In this case, back-most fragments are visible through fragments with partial visibility. However, fragments with partial visibility also reduce the visibility of those behind it in view direction. Therefore, fragments must be composited in depth-sorted order as defined by [Porter and Duff 84]. For simplicity, the same opacity A is used for weighing the three (R,G, B) color channels.

We define order independence, or determinism, as the property of generating the same result for the same input, even when the input order changes. This is important in OIT and AA rendering due to the depth-sorting requirement for compositing. When fragments are composited out-of-order, artifacts may be noticed when their visibilities are not properly addressed. A common artifact is a surface appearing in front (most visible) of the others when, in the scene, it should be behind.

Because the number of fragments generated per pixel at each frame is unpredictable, and may vary drastically for animations and different points of view, memory management is essential. Considering that memory is a finite hardware resource, collecting all fragments for posterior sorting and compositing is not always feasible. Alternatives working on fixed memory budgets must select fragments on-the-fly, leaning towards performance or image quality.

Transparency is a physical property of materials which allows light to pass through them, allowing us to see whatever is behind objects composed by such materials. This property is very important to correctly simulate the appearance of real objects and visual phenomena. Though the z-buffer can satisfactorily handle opaque primitives, efficient rendering of transparency remains an open problem. The **core bottleneck** is the depth-sorted order required to correctly evaluate visibility through an unknown number of layers with different opacities. Figure 1.1 shows the correct compositing order and the artifacts generated by improper blending order.

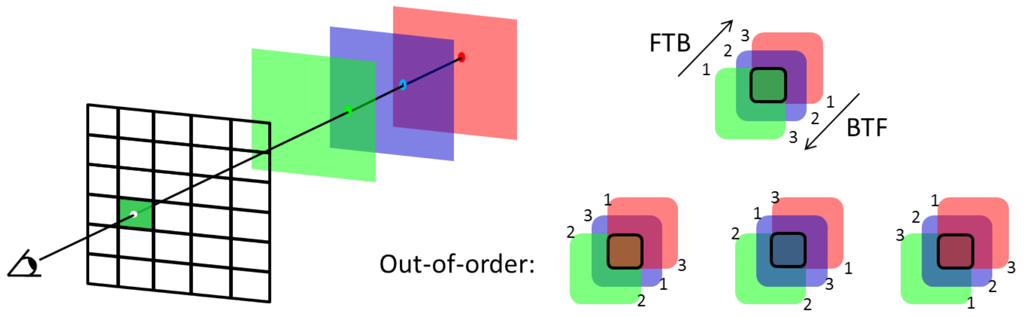


Figure 1.1: Depth-sorted order required for transparency rendering: (left) the view ray intercepting 3D primitives, first the green, then the blue and the red ones; (right) the correct front-to-back (FTB)(and back-to-front (BTF)) fragment compositing and the wrong out-of-order compositing. Numbers indicate primitive arrival order.

Due to the undetermined number of transparent fragments involved in the color of a pixel, anti-aliasing becomes particularly challenging. In basic rasterization, a fragment samples attributes (color, depth, normal, ..) at the pixel center. In multisampling (MS), coverage is sampled at infinitely small dots scattered through the pixel area. Differently, conservative rasterization (CR) generates all fragments that intersect the pixel area. For both, attributes are sampled at the pixel center. Generating, storing and processing multiple samples for transparency rendering results in an overhead for performance and memory. Through CR fragment's coverage can be evaluated differently, allowing one single sample per fragment to be sufficient. However, correct composition of single-sampled fragments with partial coverage is highly susceptible to numeric errors and still an open problem.

1.2 Thesis Statement

Compute the pixel color closest from the exact color from the compositing of several fragments with different visibilities (depth order, coverages and opacities) and attributes (color, shading, etc), within a given memory budget.

1.3 Thesis Contributions

During the development of this thesis, we arrived at a number of intermediary results that contribute to the understanding of the visibility problem and the approaches used to solve it. We started by grouping transparency rendering techniques into a concise survey that exposes different approaches to the problem (MAULE et al., 2011). The survey also compares the algorithms with respect to three key features: image quality, performance, and memory consumption; providing clues to which technique is better for a given rendering scenario. As will be described, the diverse approaches try to balance these three features. However, this has been proven to be a very hard task and what we see in the literature is severe degradation of one or more of these features when one of them is favored.

In the effort to improve image quality for transparency rendering, we also developed a tutorial complementing the transparency survey and including anti-aliasing approaches (MAULE et al., 2012a). High-quality anti-aliasing for transparency requires per-sample storage, sorting, and compositing. This significantly increases the costs of rendering, when compared to opaque anti-aliasing. In our tutorial, we present anti-aliasing approaches with alternatives to transparency rendering, pointing their advantages and drawbacks. As the transparency rendering problem alone, anti-aliasing for transparency also aims the balance between processing, quality and memory. And, again, we can see a clear imbalance where at least one of the three suffers severe penalties.

Along the research, we developed three techniques; two targeting the order-independent transparency problem alone, and a third novel approach to the anti-aliasing of transparent rendering. The first OIT approach produces exact colors, at the expense of performance and, particularly, memory (MAULE et al., 2012b, 2014). To achieve high image quality, our approach stores all transparent fragments. It has a performance penalty of an extra geometry pass to provide a tighter memory management, when compared to similar techniques; however, it still requires unbounded memory.

The second OIT approach make use of observations gathered during the development of the first to operate in bounded memory (MAULE et al., 2013). It significantly reduced memory requirements while still improving image quality when compared to similar techniques. Its major drawbacks are performance and temporal instability. The performance is penalized by synchronization of concurrent accesses to memory while temporal stability issues may arise in certain specific conditions.

The third technique combines OIT and AA visibilities. The key idea is to provide AA using a single sample per fragment to reduce storage and compositing costs. To do that, we designed an approximation for the area sampling of the coverage of a fragment over the pixel area. However, we still have problems compositing fragments that are continuous in the mesh into a full fragment.

1.4 Document Organization

We characterize the environment over which this thesis was developed and the key problem it addresses in Chapter 2. The image generation system (graphics pipeline) and hardware (GPU) over which it is implemented are described in Section 2.1. While the problems involved in multi-fragment rendering, and their relationship to the rendering environment, are detailed in Section 2.2.

The transparency-rendering effect is addressed in Chapter 3. First, the physical effect of transparency and its computer simulation are exposed in Section 3.1. Then, a literature review is given in Section 3.2. This review includes transparency techniques within our proposed classification, which categorizes techniques according to the approach used in handling the problem. It also includes analysis and comparison of techniques. What follows are two solutions proposed within this thesis. The first technique, called dynamic fragment buffer (DFB), focusing on memory reduction for exact transparency rendering, is detailed in Section 3.3. The second technique, called hybrid transparency (HT), performing approximate transparency rendering on a small memory budget, is presented in Section 3.4. At last, all transparency rendering techniques are classified according to their capabilities, based on memory footprint, performance, and image quality. The result, an overview of the current state of the research field, is presented in Section 3.5.

Anti-aliasing of computer-generated images is covered in Chapter 4. The aliasing artifact is described and explained in Section 4.1, which also includes the problem definition of anti-aliasing for rendered images. Techniques that address aliasing for raster-based pipelines are discussed in Section 4.2. This section presents techniques classified and compared by approaches adopted, and discuss their applicability for aliasing removal of transparent scenes. Later, we present a novel solution for anti-aliasing of transparent fragments in Section 4.3. This techniques is currently being developed, but preliminary experiments reveal promising results. Summary, discussion, and conclusions are presented in the last Chapter 5.

2 PROBLEM CONTEXT AND CHARACTERIZATION

In this section, we present the environment in which this thesis was developed. This comprehend the hardware and rendering system. We also define the problem of multi-fragment rendering, while pointing out the challenges of solving it in the chosen environment.

We decided to work with the transformation and lighting pipeline implemented on modern GPUs (graphics processing units). This pipeline has a series of steps involving manipulation and mapping of geometries, generation and shading of fragments, and visibility determination. Since this pipeline is efficiently implemented in hardware, for the purposes of our research, we made use of the GPUs programmability to replace pieces of it and investigate the possibilities for implementing multi-fragment effects efficiently.

The challenges involved in compositing multiple fragments in GPUs arise from the data dependency among fragments, characterized by the depth sorting required for visibility computation. The hardware was designed to process a flow of fragments, which are constantly being generated and discarded, and multi-fragment dependencies do not fit naturally in this context. Fragments visibilities depend on fragments already processed, as well as fragments that were not generated yet. This creates a memory management problem, since the number of fragments generated for each frame is unpredictable and unbounded. Ignoring or mishandling these dependencies often results in severe artifacts.

Another issue comes from the massive parallel nature of GPUs, which favors independent fragment processing. The dependencies inserted by the multi-fragment processing generate concurrency hazards, caused by fragments trying to update the same pixel at the same time. These situations must be managed through synchronization tools, which serialize the processing, guaranteeing correct results at the cost of loss of parallelism and, consequently, performance loss. Concurrency management of fragments is one of the major bottlenecks faced by multi-fragment effects in massively parallel architectures.

Both pipeline and hardware are not devoted to handling multi-fragment effects, although they've been rapidly evolving to support such effects. One of the contributions of our research is the investigation of several approaches to handling multi-fragment effects in current graphics hardware.

2.1 Graphics Pipeline and GPU Architecture

Graphics processing units (GPUs (NICKOLLS; KIRK, 2009)) are highly parallel hardware, with specialized functions for image generation through rasterization. Among other image-enhancement effects, they implement the transformation and lighting graphics pipeline. This chapter presents the basics of this pipeline, which was used in the development of our research, as well as some relevant details of the hardware architecture.

CPUs (central processing unit) nowadays are composed by multi-cores, in which each core resembles an SISD (single instruction, single data) machine, characterizing sequential processing (this is actually an over-simplified view, because in reality we have pipelining and other techniques which improve performance by inserting some parallelism at instruction level). On the other hand, the set of multi-cores that compose a CPU characterize an MIMD machine, where multiple sets of instructions are executed in parallel over multiple sets of data. Each CPU core is completely independent of the others, unless the software says otherwise. In such cases, there are synchronization tools that one can use to communicate between cores and avoid concurrency hazards.

The model through which GPUs operate is different. The GPU is divided into SMs (streaming multiprocessors), which are blocks of highly multithreaded SPs (streaming processors) that execute the same instruction set in an SIMT fashion (single instruction multiple threads). These SMs are independent and asynchronous, which means that one SM may be executing a different instruction set from another SM, characterizing an MIMD model. GPUs also provide tools for communication and synchronization of threads, both inside the same SM and among different SMs.

In conclusion, while a CPU is a set of MIMD machine composed of SISD cores, a GPU is an MIMD machine composed of SIMT blocks composed of SISD processing units. Figure 2.1 shows a high-level architecture comparison between CPU and GPU core organizations.

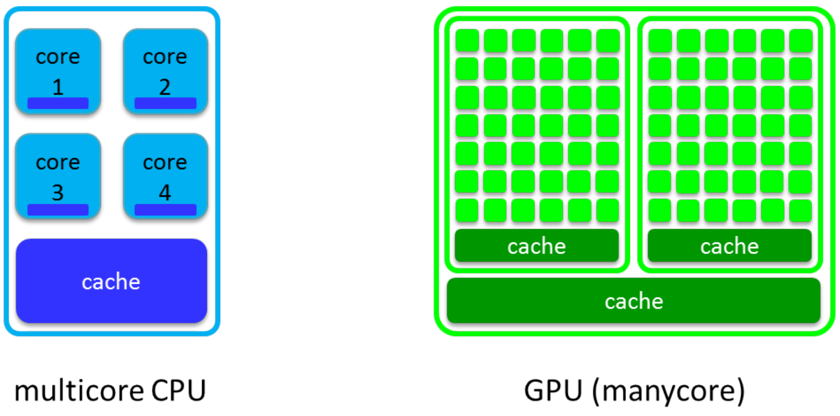


Figure 2.1: CPU x GPU architecture comparison: a modern CPU is composed of a few high-speed cores while GPU achieves performance through a massive number of slower cores.

Both CPU and GPU are MIMD machines, both are able to execute multiple instructions on a multiset of data, but the model differs from CPU to GPU. While CPUs perform processing on one or more sequential threads, prioritizing the performance of each thread, GPUs prioritizes the performance of the task to be done, executed by several parallel threads.

2.1.1 Graphics Pipeline

This section gives an overview of the stages of the *Transformation and Lighting* (T&L) pipeline implemented in modern GPUs (FOLEY et al., 1990; HEARN; BAKER, 1997; WRIGHT et al., 2010; BAILEY; CUNNINGHAM, 2011; SHREINER et al., 2013; SEGAL; AKELEY, 2014). This pipeline is composed by five main stages: vertex processing, geometry processing, clipping, rasterization, and fragment processing. There are other stages that can be enabled for rendering of special effects and optimizations, but they are not relevant in the context of this thesis, therefore, will not be discussed in this section.

The graphics pipeline input is a stream of vertices, along with their connections and attributes, camera and light descriptions, which represent a virtual scene, usually in 3D space. The resulting output of this pipeline is a 2D image, which can be seen as a matrix of pixels, representing the scene through the view specified by the camera. Figure 2.2 summarizes the five main stages of the raster-based graphics pipeline discussed in this section.

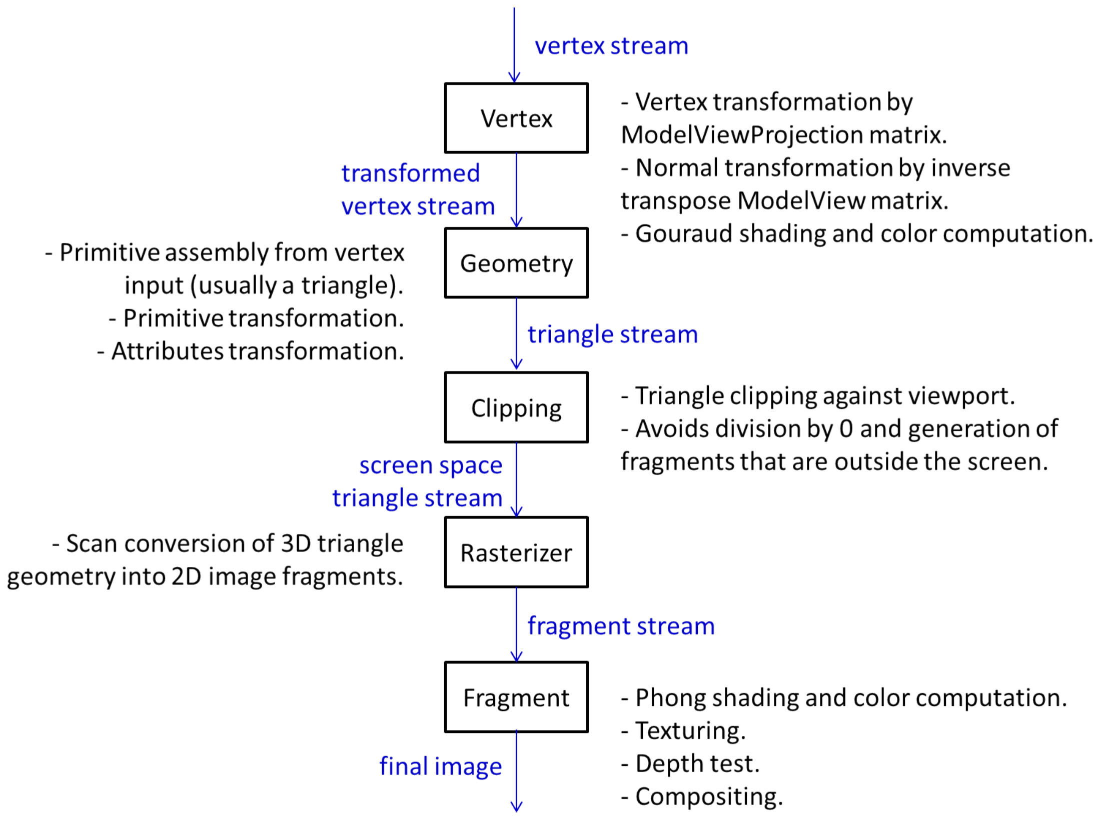

vertex stream
Vertex
- Vertex transformation by ModelViewProjection matrix.
- Normal transformation by inverse transpose ModelView matrix.
- Gouraud shading and color computation.
transformed vertex stream
Geometry
- Primitive assembly from vertex input (usually a triangle).
- Primitive transformation.
- Attributes transformation.
triangle stream
Clipping
- Triangle clipping against viewport.
- Avoids division by 0 and generation of fragments that are outside the screen.
screen space triangle stream
Rasterizer
- Scan conversion of 3D triangle geometry into 2D image fragments.
fragment stream
Fragment
- Phong shading and color computation.
- Texturing.
- Depth test.
- Compositing.
final image

Figure 2.2: Transformation and Lighting Pipeline main stages: streams of vertices are transformed and assembled into strings of triangles that are rasterized into fragments, which are shaded and composited to form the final pixel color.

2.1.1.1 Vertex Processing

The input for the vertex processor stage is one vertex, whose coordinates are specified with respect to its own local coordinate system, along with its set of attributes (normals, colors, texture coordinates, etc.) The output of this stage is a set of vertices coordinates in different coordinate systems and transformed attributes.

Upon entering the vertex stage, vertices are transformed by the ModelView matrix, which is the combination of Model and View matrices. The **Model** matrix specifies transformations that map vertices to a common world coordinate system. These transformations are very useful for detaching rendering coordinate system (world) from different modeling coordinate systems (local). For example, for rendering multiple instances of the same model into different scales, positions and orientations. Values in the **View** matrix are defined by the camera position and orientation. They produce transformations that map vertices from the world coordinate system to the camera coordinate system.

Vertices normals also need to be transformed to the camera space. However, the ModelView matrix is appropriate for transforming points, not directions; therefore, the normals are mapped by the inverse transpose of the ModelView matrix. These transformations map vertices to a convenient space, where projection can take place.

The **Projection** matrix varies according to the kind of projection that is intended. We will be using perspective projection, which approximates the projection done by a pinhole camera. This camera is specified by its field of view and aspect ratio between the projection plane axis, and also by a near and far clipping planes. These values inside the Projection matrix transform the vertices into a homogeneous space, where clipping against the view volume (defined by the same matrix) will be easily performed.

2.1.1.2 Geometry Processing

The geometry processor handles the assembly of vertices into geometric primitives, usually triangles. Triangles are the primitive most commonly used because they are guaranteed to be planar and have only one possible vertex connection, which facilitates assembly and rasterization. In this stage, vertices can also be further transformed, taking into account the proprieties of the primitive to which they belong. In this case, vertices attributes must be transformed accordingly.

2.1.1.3 Clipping

The clipping stage is responsible for defining which portions of the geometry are inside the view frustum. In 3D rendering, the view frustum is a 6-sided bounding volume defined by the projection matrix. Everything that is outside the frustum is guaranteed to be not visible in by the camera. This stage receives a geometric primitive as input and outputs a clipped primitive.

When a triangle arrives at the clipping stage, its vertices are tested against the six clipping planes of the view frustum. Vertices outside the frustum are clipped and new vertices are generated at the intersections of the triangle edges with the clipping planes. This procedure may generate a more complex primitive, which will be triangulated and each triangle will be sent separately to the next stage.

After clipping, each vertex goes through the perspective division. This operation consists of the division of the vertex coordinates by its homogeneous coordinate w. Because of the transformations on the projection matrix, the perspective division will transform the view frustum (which, at this stage looks like a pyramid with its top chopped off) into a cube with bounding going from -1 to +1 in each dimension. This cube characterizes the normalized device coordinates space (NDC), from which mapping to the screen is trivial.

The mapping from NDC space to screen space is done by the **Viewport** matrix. This matrix contains the information of where the first pixel must be drawn, as well as how many pixels exist in each screen dimension. The final output of the clipping stage is one, or more, triangles whose vertices coordinates are already mapped into screen space, ready to be rasterized.

2.1.1.4 Rasterizer

Rasterization, or scan conversion, is the processing of converting a geometric primitive into a set of fragments. Upon receiving a triangle, the rasterizer will select the top most and the left-most vertices to start the scanning. Then, it will step over the left most edge, increasing the y coordinate pixel by pixel, using the Bresenham algorithm (BRESENHAM, 1965). At each y coordinate, attributes are interpolated along the edges. The rasterized, then, will follow the line connecting left and right edges, again pixel by pixel, interpolating attributed between the two points over the edges.

2.1.1.5 *Fragment Processing*

The last stage of the pipeline is the fragment processor. The basic tasks of this stage include visibility determination, texture mapping, shading (when using Phong shading) and compositing, among others. It uses the attributes interpolated by the rasterizer to compute the fragment color. The fragment visibility depends on other fragments and is the main topic of discussion in this thesis.

2.1.1.6 *Illumination*

When using the Gouraud shading approach (FOLEY et al., 1990), the lighting of the scene is computed by the vertex stage. Each vertex is shaded and its colors will proceed through the pipeline to be interpolated along the triangle fragments. This approach is cheaper than the Phong shading approach (FOLEY et al., 1990), which performs shading per fragment, at the fragment stage. Although both approaches may use the same lighting equations, usually Phong illumination model (FOLEY et al., 1990), the quality of highlights (specular light reflection) computed through Gouraud shading is compromised at vertices and edges.

The Phong illumination model provides an empirical equation to compute shading at a given surface point. Transcribed in Equation 2.1,

$$I_p = k_a i_a + \sum_{t}^{lights} (k_d(\hat{L}_t \cdot \hat{N}_t) i_{m,d} + k_s(\hat{R}_t \cdot \hat{V})^{\sigma} i_{m,s}), \tag{2.1}$$

it includes the material (k) reflections constants for diffuse ($_d$), specular ($_s$) and ambient ($_a$) reflections, and the material shininess (σ). These interact with lights (t) intensities (i) according to the relations between the direction of incidence of light (L) and the surface normal (N), and between the reflection (R) and the view (V) directions.

The ModelViewProjection matrix makes the vertices ready to be mapped to the 2D screen (after perspective division). However, for the Phong shading approach, where vertex attributes are interpolated along fragments and shading is computed per fragment, it is appropriate for the vertices positions to be in world or camera space (as well as the lights illuminating the scene). In this case, vertex coordinates transformed by the ModelView (or only the Model) matrix are also sent to survey-dataflow-experimentsthe next stage of the pipeline.

2.1.2 GPU Architecture

GPUs are massively parallel processors that have special functions to run an accelerated implementation of the transformation and lighting pipeline. They were initially created to be specialized graphics hardware, but have grown to be ubiquitous parallel processors of general purpose. This section provides an overview of the architecture of GPUs and how they implement the graphics pipeline.

GPUs started as graphics co-processors, specialized hardware focused in graphics and video management. They had special processors for each phase of the pipeline. Then the GPU architecture was unified: general processors were adopted for processing of programmable phases of the pipeline, while performance-critical phases still have specialized hardware. Figure 2.3 shows an abstraction of how the pipeline (first presented in Figure 2.2) is implemented on modern GPUs.

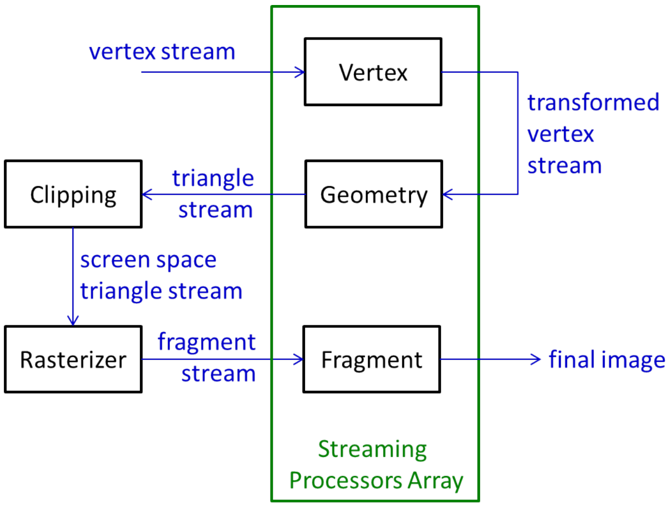


Figure 2.3: Graphics pipeline on GPU: programmable phases of the pipeline (vertex, geometry and fragment) are executed on processors of general purpose, while performance-critical phases are executed in fixed units (clipping and rasterizer) (figure modified from (NICKOLLS; KIRK, 2009)).

The programmable processing units of the GPU are called streaming processors array (SMA). The SMA contains a number of processors organized hierarchically. Each SMA contains eight texture/processor clusters (TPC), with one specialized texture unit. This unit is shared by two stream multiprocessors (SM), which are groups of stream processors (SP), or cores. Each one of these cores is capable of managing a number of parallel threads executing the same instruction set. A diagram showing how these elements are organized and share resources is presented in Figure 2.4. The strategy of grouping elements allows the management of shared resources, as specialized units and local memory, thus, better utilizing the chip area. This organization allows the GPU to be scaled with great flexibility, varying the number of processing units according to the targeted platform and market segment.

With processing units of general purpose, GPUs are well equipped to address massive parallel problems, which do not need to be graphics-related. Through the driver that implements the APIs (application programming interface), the control units can switch the processing flux to graphics (e.g. (OPENGL, 2014) and (DIRECTX, 2014) APIs) or general purpose computing (e.g. (CUDA, 2014) and (OPENCL, 2014) APIs). Graphics and general purpose APIs offer different synchronization and concurrency management tools. In the development of this thesis, we used both operating modes to process different stages of our solutions, exploiting the full potential of GPU's parallelism.

Even with the increasing programmability, GPUs continue to be well suited for graphics, which contains high levels of parallelism. For example, all vertices are independently transformed by matrices and can all run in parallel. Fragments are also mostly independent; except when it comes to visibility determination. In this case, fragment processing within the same pixel may be serialized for visibility computation, thus, running concurrently. However, fragments from different pixels hardly interfere with each other, allowing a greater level of parallelism, free of concurrency.

In the next section, we define and exemplify scenarios in which multiple fragments contribute to the same pixel color. Computing the visibility which weighs fragment contribution is the central topic of this thesis. We also highlight the main challenges to be faced when implementing multi-fragment effects in today's GPU architecture.

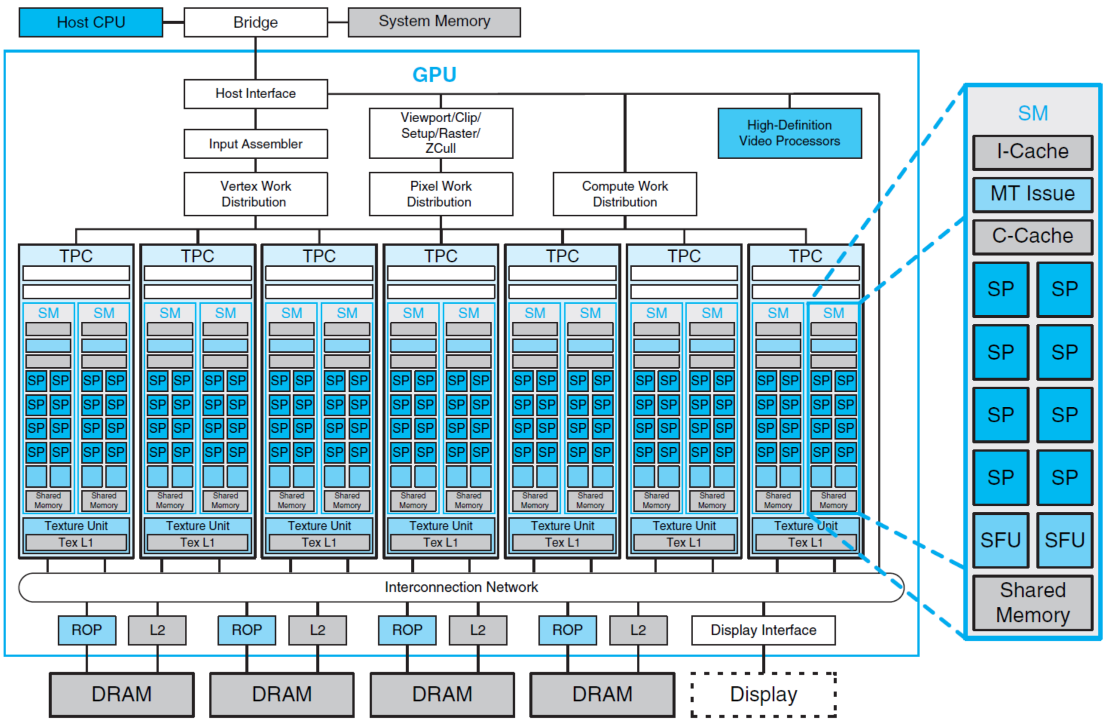


Figure 2.4: GPU architecture: a number of threads run the same instruction set inside each SP, which is organized in blocks inside SMs. SMs are grouped inside a TPC, where they share a texture unit. A set of TPCs is called a SPA, which executes the programmable phases of the graphics pipeline. The architecture also presents fixed units for control and graphics. Note that most of the chip is intended to perform computations, instead of data storage (figure from (NICKOLLS; KIRK, 2009)).

2.2 Multi-Fragment Rendering

Multi-fragment rendering is characterized by the compositing of attributes from more than one fragment to form the final pixel color. The weight of each fragment depends on its visibility, which in turn depends on the fragment coverage over the pixel area, the coverage of fragments in front of it, the fragment own opacity, and the opacity of fragments in front of it (PORTER; DUFF, 1984). In shadow effects, the shading of the fragment depends on other fragments. Important effects that increase realism and image quality are composed by multi-fragments. Examples of such effects are transparency, translucency, constructive solid geometry, depth of field, anti-aliasing, depth-fighting management and shadows, among others.

The most basic form of rendering involves rasterizing only those fragments that cover the pixel center, and, among them, choosing the one closest to the eye, while all the others are discarded. In this mode, the pixel will receive the color computed from the attributes of a single fragment, independently of its coverage or of other fragments present in the scene (FOLEY et al., 1990; WRIGHT et al., 2010). The realism and image quality achieved by this simplified form of rendering is highly limited. Using the information of more than one fragment per pixel can greatly improve the quality of the final result.

The simplified rendering is straightforward. It presents predictable performance and memory consumption. When we start considering more than one fragment at each pixel, we do not know beforehand how many fragments will need to be evaluated and composited. Depending on the scene and viewpoint, the number of fragments per pixel may vary drastically in the same frame, reducing efficiency of memory and work balance of threads.

More often than not, multi-fragment compositing depends on the relative position of fragments with respect to each other, and with respect to a view reference, implying some kind of sorting. The basic rendering mode also uses sorting in the form of depth test, which selects the front-most fragment, discarding all the others. In multi-fragment rendering, fragments are not so prone to be discarded, and an unpredictable amount of memory would be required to store all of them for proper sorting.

Different approaches and strategies are used to evaluate multi-fragments visibilities, depending on the constraints of the effect being processed. Commonly, some sort of simplification is applied to ease computer management and achieve real-time performance (e.g. ignoring refraction when handling transparency). In this section, we will exemplify different visibility relationships among fragments through effects that require multi-fragments (SHREINER et al., 2013). These examples will demonstrate the necessity for sorting, which, combined to the execution environment (T&L pipeline in GPU), implies a series of challenges that are discussed next.

2.2.1 Multi-Fragment Generation

Fragments are samples of the scene that may contribute to a pixel color. How these samples are taken and evaluated depends on the rendering effect being applied. This section illustrates fragments generation using different multi-fragment effects as examples (COZZI; RICCIO, 2012). These effects may consider compositing full-covered fragments along depth, fragments with partial coverage or both cases simultaneously. Considering the effects targeted in our research, **transparency** composes fragments along depth, while **anti-aliasing** accounts for multi-fragment coverage. For anti-aliasing of transparent materials, fragments are generated and processed in both dimensions: depth and coverage.

2.2.1.1 Fragments Along Depth

The rasterization process generates a series of fragments for each pixel. For now, let's consider that each fragment covers the entire pixel area. In this case, we have a flow of fragments distributed in depth that intersect the pixel view ray. How these fragments will be combined to compose the pixel depends on the effect being applied. Here we discuss examples of effects using multi-fragments along the depth of a pixel.

In transparency rendering, the fragments must be sorted in depth so they can be composited using a blending equation (PORTER; DUFF, 1984). This effect is commonly simplified by disregarding the refraction of materials and its effects on the transmitted light. Details on transparency rendering and techniques are given in Chapter 3. The same process is used for translucency, with the difference that the thickness of objects will change the perceived opacity. Another simplification here is ignoring subsurface scattering and translating thickness directly to the distance between the front and back fragments of the object. Techniques for both effects often use approximations to limit resource usage and improve performance. For example, storing and processing only the k front-most fragments, which tend to be the most visible ones.

In the case of constructive solid geometry (CSG), where the final goal is a series of boolean operations among objects, simplifications are not straightforward. All fragments must be stored, and the algorithm must keep tracking of pairs of fragments that delimit the object surface, the first is the front fragment, and the last is the back fragment, like in the parenthesis balancing problem. For that, fragments must be in the proper depth order and contain their respective object ids, for identification. These pairs are then composited using the boolean operations to decide which fragments will survive and display the combined geometry.

A tricky problem occurs when two or more fragments have the same depth. This may happen in two situations: (i) the geometry was specified with coplanar surfaces, or (ii) the limited precision in depth interpolation causes close fragments to be mapped to the same depth value. When this happens, a depth order must be established from another attribute. If no approach is implemented, the first (or last) fragment to be processed will appear in front of the others, which may generate order-dependent artifacts known as flickering, as the surfaces may swap their processing order, causing depth fighting. Approaches to address this problem include prioritizing the fragments with the smallest (biggest) object id, or the fragment which is the most front-facing (whose normal has the smallest dot product with the view direction).

Another class of problems changes the fragments shading not with respect to the viewer, but with respect to the light. If a fragment is visible by the light it is illuminated, otherwise, it is in shadow. This is the most common formulation for the shadow problem, which is the basis for the shadow mapping algorithm (WILLIAMS, 1978). The key idea is to render the scene from the light point of view, generating a depth map, which in turn is consulted during rendering to check if the fragment is in shadow. This approach inspired a series of shadow mapping algorithms, handling different aspects of the problem. Some address the difference in resolution from the light map to the camera, while others focus on smoothing the edges of the shadows to approximate an area light effect with umbra regions. An interesting effect is obtained when casting shadows from transparent surfaces. The light map will be required to store not only the closest depth, but all depths with the respective colors and in depth-sorted order. The filtering process, when consulting this deep shadow map, will generate fragments with colored shadows depending on the depth order, opacity, color and distance of the fragments captured by the light.

2.2.1.2 Fragments with Partial Coverage

In this section, let's consider that fragments may not cover the entire pixel area, leaving space for contributions of other fragments that are behind in view direction, in the same depth or intersecting the fragment being discussed. Taking into account the coverage of a fragment over the pixel area is an important tool for anti-aliasing.

Aliasing occurs when the sampling rate is not enough to represent the scene. This means that a single sample per pixel will most likely generate aliasing artifacts. An approach to alleviate this problem is to take more than one sample per pixel, commonly from more than one fragment, and compositing these samples together into the pixel color to generate smooth transitions. The most known approach is the multi-sampling anti-aliasing, which samples some attributes just once at the pixel center, while other attributes are sampled multiple times through the pixel area. Each sample selects the front-most fragment through depth test. More details on anti-aliasing and its techniques in Chapter 4.

2.2.1.3 Multi-Fragments in Depth and Coverage

Now, let's consider compositing multi-fragments both in depth and in coverage. An example of effect combination is the anti-aliasing of transparent fragments. While the transparency effect requires fragments to be depth-sorted and composited, anti-aliasing requires the coverage of each fragment to be taken into account during the compositing. A common approach is to take multiple coverage samples per fragment while sorting and blending per sample, before compositing the samples together. This approach does not handle all sources of aliasing, interpenetration for instance, but provides a good balance between quality, performance and memory costs. Anti-aliasing of transparent fragments is further discussed in Section 4.2.

Another example of effect that requires combination of multi-fragments in both depth and coverage is the depth of field effect. This effect blurs fragments in certain depth ranges to simulate out-of-focus objects, as would happen is a lenses system were used instead of a pinhole model. This causes fragments to bleed into neighboring pixels, where they must be blended according to their opacity and depth relation to the neighbor fragment list.

2.2.2 Multi-Fragment Processing: The Sorting Problem

All effects that exemplify the multi-fragment rendering problem in the previous section require some level of depth sorting for determining fragment visibility. That is because multi-fragment effects intrinsically denote spatial relationship among fragments. Such relationships are important to be represented because they help our visual system to understand the 3D context seen through a 2D display.

The parallel pipeline on GPU is optimized to handle a flux of fragments: a fragment is generated, evaluated and written or discarded. The problem with sorting fragments is that the order of one fragment depends on the order of others. But these others are already gone or were not generated yet. Unless all fragments are stored for posterior processing, all the data needed for sorting will not be available at the same time.

If, in one hand, sorting performed with partial data cannot offer guarantees of correctness. On the other hand, store all fragments generated in a frame may require an awfully large amount of memory. Attempts to select a bounded set of fragments to be stored during rendering, besides being based on partial data, have an extra penalty on parallelism because the selection must be performed concurrently. When considering generating the fragments multiple times, quality can be achieved at a low memory footprint. However, performance is penalized, since the scene is to be processed through the pipeline an unknown number of times.

2.2.2.1 Memory

Memory is a limited physical resource. There are techniques to expand the working memory to slower, but much bigger, medias, like virtualization for instance. However, this kind of memory management is too slow to be used in real-time rendering.

Storage of all fragments for evaluation of multi-fragment effects is viable when the scene being rendered is small and predictable. Otherwise, we are in danger of running out of memory. Since multi-fragment effects are not the only computations requiring memory, lack of memory during rendering may generate a problem in any part of the pipeline, including management and control, which would cause the whole application to crash.

However, being able to gather all required information to perform sorting would be the ideal scenario, both in terms of performance and image quality, rare are the cases when we can do so. With increasing displays and multi-displays resolution, and increasing scenes complexities, memory is an essential resource that must be used wisely, not freely. This, inevitably, implies tradeoffs among memory, performance and quality.

2.2.2.2 Performance

Performance on GPU does not depend only on the amount of computation. It depends on the balance of work among threads and the balance of resources for threads that share the same building blocks (SMs). It also depends on the constraints that threads have to run freely, without depending on data or results from other threads.

A batch of work is executed in the GPU by grouping threads in warps. All the threads in a warp must finish their tasks before the work can continue. If a thread has more work than its companions, all the other threads will have to wait for it to finish. A solution for this issue is to divide the work into smaller chunks and run more threads. But this is not always trivial or feasible.

Balancing the amount of resources required by each thread within the amount of available resources in hardware is another challenge. For instance, if each thread needs to access and process a fairly big chunk of data, fewer threads will fit in a block of shared cache. This causes the warps to be smaller, sub-utilizing the GPU infrastructure and leaving fewer threads available to hide memory latency.

GPUs rely on the availability of a large number of threads to replace those threads that block waiting for data. Although generating enough threads is not a problem for multi-fragment effects, these threads usually present lots of dependencies. When they block, the reason is often a concurrent access. This means that it is common for threads to be blocked waiting for the same resource, serializing their executions. As explained in Section 2.1, GPU cores have slower clock cycles than CPU cores, making up for this lower frequency through high numbers of cores (parallelism). Therefore, loss of parallelism on GPU implies huge performance penalties.

Another performance penalty usually applied to overcome limited resources is multi-pass, or rework. In multi-fragment effects this means passing the geometry through the pipeline several times, at each time selecting and accumulating a piece of information. For scenes with large amounts of geometry, this is obviously not a good thing to do if you can avoid it.

2.2.2.3 Quality

When memory is saved, quality is the first to suffer. If the performance is also spared, we will observe further quality loss. Finding a good balance among the three (memory, performance and quality) is not trivial. It depends mostly on the application goals (visualization, interaction or offline rendering), the effects being handled and the targeted platform, which determines resource availability (high performance, mobile, etc.).

For visualization applications, performance and memory can be penalized. However, a minimum performance and memory are still required. This is not true for interactive applications, where performance must be higher. In this case, quality has a very low priority. Unlike offline rendering, to which quality is essential, and everything else can be largely spared.

On the effects arena, each multi-fragment effect has its own range of techniques varying tradeoffs among memory, performance, and quality. For the effects addressed in this thesis, techniques details can be found in Section 3 for transparency, and Section 4 for anti-aliasing.

2.3 Summary

All effects and techniques discussed in this chapter greatly improve image quality and realism, generating more pleasant and credible figures. They do this by considering the contributions of more than one fragment per pixel, thus the name multi-fragment rendering. The most important aspect of these effects is the visibility determination for each fragment, through coverage, opacity, and distance to the light with relation to other fragments in the same pixel. Or even in other pixels, when fragments are blurred. We conclude that sorting is fundamental for determining the relationship among fragments and, ultimately, their visibility and shading.

As we saw in Section 2.1, the hardware is projected mainly to handle a flow of fragments, where each fragment is processed independently. Multi-fragment effects, by introducing visibility dependency among fragments, are presented with the challenge of efficiently managing concurrency and resources in an environment that favors independence between fragments. This challenge motivates the development of this thesis: the search for an efficient scheme to achieve order-independent transparency with anti-aliasing using current graphics hardware.

3 TRANSPARENCY RENDERING

This chapter presents ideas, concepts, and approaches on the transparency rendering problem. We start by defining the concept of transparency, how transparency we see in the real world is modeled through physical equations and how this can be used to generate images on a computer (Section 3.1). Then, we expose transparency rendering techniques present in the literature, classified by the approach used and compared with respect to performance, memory consumption and image quality (Section 3.2).

What follows are two techniques developed within this thesis. The first one focuses on image quality and achieved the best memory footprint for massive models (Section 3.3). The second technique focuses on memory reduction and achieves the best image quality among bounded-memory techniques (Section 3.4). At last, we classify transparency rendering techniques with respect to their capabilities: how they handle memory, and which performance and quality they can achieve (Section 3.5).

3.1 Transparency Problem

Transparency is the property that enables light to pass through materials. Nature presents several materials and phenomena containing transparent properties, some of them within our visible spectrum that enable us to see what exist behind them. Physical models can describe with rich details what happens to each light beam when crossing a material, however, they must be grossly simplified to be efficiently simulated in a computer for real-time applications.

3.1.1 Transparency Physics

When light comes into contact with matter, four main reactions are commonly expected:

- the light may be **reflected**, returning to the medium where it came from;
- the light may be **absorbed** by the matter, becoming another form of energy;
- the light may be **propagated** into the matter medium, or;
- the light may be **transmitted** through the matter to the media behind it.

In real matter, all sort of reaction is expected at some degree (not only these listed here, but the others are not interesting for our discussion), depending on the matter properties and the properties of the medium where the light comes from, as shown in Figure 3.1.

A material is said to be **transparent** when light can be transmitted through it, as the portion L_t of L_i is transmitted through the medium in Figure 3.1. If the light is scattered in its way through the medium (one incident light beam generates several "little" transmitted light beams in different directions) we say that the medium has the property of **translucency** instead of transparency. An example of a transparent object is the glasses

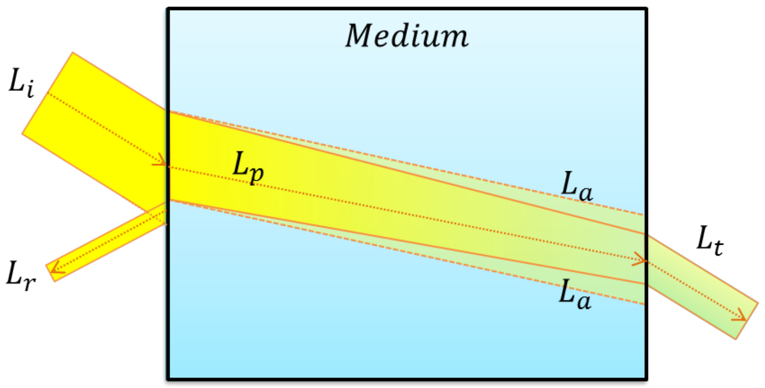


Figure 3.1: Light interaction with a medium: a portion of the incident light L_i is reflect (L_r), a portion is propagated through the medium (L_p), some is absorbed by the medium (L_a) and the remaining is transmitted beyond the medium (L_t). In this diagram, the thickness of the light beam represents its intensity, the changes in direction illustrate the refraction and reflection laws and the color change illustrates the filtering properties of the medium for some wavelengths.

lens, through which one can see clearly. An example of a translucent object is a plant leave; one can see the light pass through it, but cannot distinguish what is behind it. When a material does not permit light to pass through it, we say this material is **opaque** (no L_t). An example of an opaque object is a coin; even if one puts it over a little flashlight, no light will be seen through the coin.

The intensity of the propagated light which will be transmitted depends on how much of it will be absorbed by the medium. The amount of light absorbed is expressed in terms of percentage by the material coefficient of **absorbance** ($\alpha \in [0, 1]$, or something between 0% and 100%). As a complement of it, a material has a coefficient of **transmittance** ($\tau = 1 - \alpha$), which is the portion of light transmitted (not absorbed) by the material.

As the light interacts with the material properties during propagation, it can change its own properties, such as direction and color (wavelength absorption). The change of direction of a light beam is called **refraction** and is modeled by the Snell's law:

$$n_1 sin\theta_1 = n_2 sin\theta_2. \tag{3.1}$$

Where n_1 and n_2 are the measured coefficients of refraction of the leaving and entering media, respectively, and θ_1 and θ_2 are the angles of incidence and propagation, respectively. Most of the transparent materials, such as our glasses lens example, refracts the transmitted light, and the degree of refraction (e.g. optical correction) is calculated using this formula. What this formula does not account for is the possible scattering property of some materials, so it cannot completely represent light propagation through translucent materials.

Different materials have preferences to which wavelength they will absorb most. The wavelength is recognized by our visual system as the color of light, so, the wavelength absorption property determines the color filtering of the light beam passing through the material. When the material reflects short wavelengths, absorbing the others, our visual system recognizes it as being of a bluish color, when reflecting mean wavelengths is recognized as greenish and long as reddish. Figure 3.2 shows an illustration of wavelengths absorption and the colors it may cause our visual system to perceive.

Each time the light is transmitted, its properties change (here we are interested in its color property). So, if, for example, a white light is transmitted by a yellow surface, the

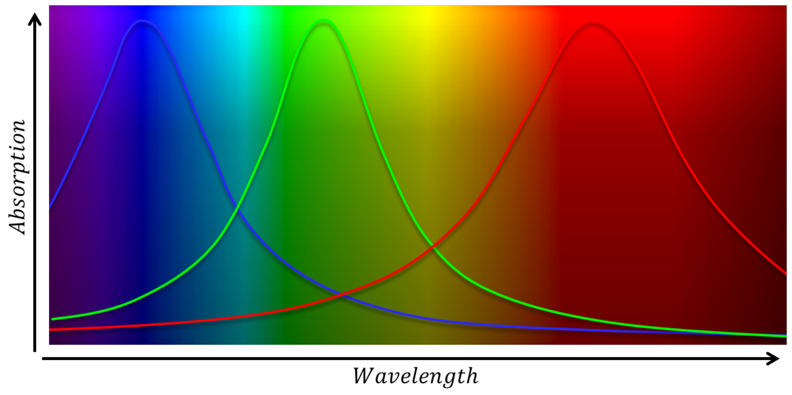


Figure 3.2: Color spectrum and light wavelengths: materials which absorb short wavelengths are perceived as bluish, while those who absorb mean and long wavelengths are perceived as greenish and reddish, respectively. The other colors of the visible spectrum are perceived by the combined absorption of different wavelengths with different intensities.

light becomes yellow. If then, this yellow light is transmitted by a pink surface, our eye will perceive it as reddish light. Figure 3.3 illustrates this example.

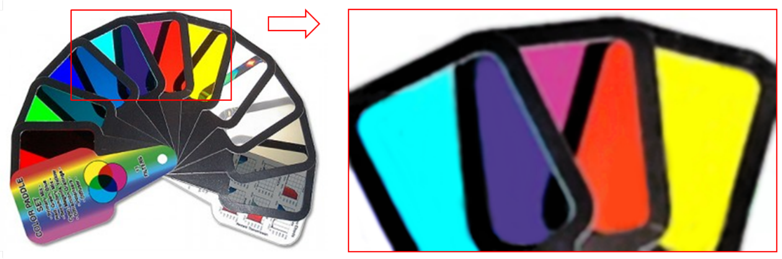


Figure 3.3: Color filtering example: at left there is a set of color filtering sheets for different wavelengths, the overlap of different sheets will produce different colors. At right, an example of a pink sheet over a yellow one producing reddish color, and a light blue sheet over the pink one producing violet.

This color filtering property of transparent materials is the main motivator of this thesis proposal. *Which properties will be present in the light arriving at the eye after being transmitted through several transparent objects of different materials?* Or, simply, determine which color the eye will perceive is the main goal of this thesis.

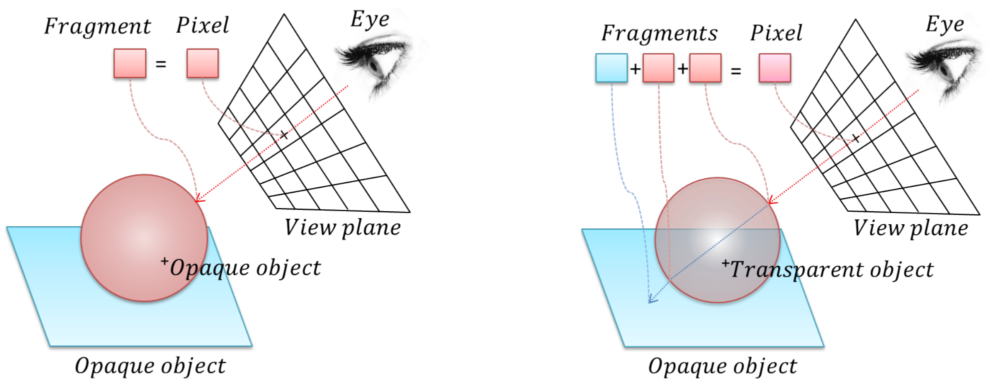
More information and details about light interaction with matter can be found in (HECHT, 2002),(PEDROTTI; PEDROTTI, 1993),(FOX, 2007), and about the subject of light perception in (GEGENFURTNERS; SHARPE, 1999).

3.1.2 Transparency Rendering

In the effort of representing the real world using a computer, the transparency property must be accounted for. Some examples of real transparent materials commonly rendered are water, glass, and plastic. Despite that, some rendering effects also use transparency to accelerate processing and to improve image quality. An example of such use of transparency is in the rendering of foliage, when instead of using geometry, a texture with an alpha mask is applied (the texels outside the leave are completely transparent). Some translucent materials (those which scatter the transmitted light) are often simplified and rendered as if they are transparent; examples are smoke and jelly. The transparency effect itself is roughly simplified and often only accounts for changes in the color property of light.

Physically, each material absorbs some wavelengths more than others. When rendering transparency, this feature is commonly simplified and all wavelengths are absorbed equally. Then, the absorbance property of the virtual surfaces can be encoded as a fourth color channel, known as alpha channel, and each fragment generated by such surfaces will have an RGBA color.

Rendering of a virtual scene can be resumed to choose the right color to each screen pixel. This is done by choosing the color from the front-most surface, the first which the view ray of the pixel comes into contact with. If the surface is opaque, it occludes everything that exists behind it. When the transparency effect is involved, the pixel color comes from more than one surface. The colors of the surfaces must be combined with their respective weights, which accounts for their visibility. Figure 3.4 presents two diagrams describing how (a) opaque and (b) transparent rendering work.



(a) Opaque rendering (b) Transparent rendering

Figure 3.4: Difference between opaque and transparent rendering: the pixel color is defined by the surface(s) intercepted by the view ray supported by the eye and the pixel center points. (a) when the view ray of a pixel intercepts the first surface and this is opaque, the view ray terminates and the surface color is given to the pixel. (b) when the view ray intercepts a first surface which is transparent, the ray is transmitted (or propagated), collecting the colors from all surfaces in the order they are intercepted, until find an opaque one; then the pixel color is composed by the collected colors with their respective visibility weights.

Transparency rendering is often simplified by considering the objects as infinitely thin surfaces, accounting only for the view ray interaction of the surfaces and ignoring the media inside the object. In such cases, the objects are commonly represented by triangle meshes without internal information. Interactions of the view ray with the media inside the objects are developed for volume rendering, with techniques to account for participating media.

The order in which the view ray intercepts the surfaces determines their visibility. A surface seen through another has its visibility reduced by the opacity of the surface in front if it (in view direction). In other words, the light leaving a surface is absorbed by the surfaces in front of it before reaching the eye, reducing its contribution to the pixel color.

This reduction is considered by the blending equations proposed by (PORTER; DUFF, 1984), which describe the visibility reduction one surface inflicts over another behind it and how their colors must be weighed using the alpha channel. A destiny color (C_{dst}), coming from the compositing of two fragments, is the front-most fragment color (C_{front}) weighted by its own absorbance (α_{front}), plus the back-most fragment color (C_{back}) weighted by its own absorbance (α_{back}) and by the reduction imposed by the absorbance of the front-most fragment, which is the front-most transmittance ($\tau_{front} = 1 - \alpha_{front}$):

$$C_{dst} = C_{front} \times \alpha_{front} + C_{back} \times \alpha_{back} \times (1 - \alpha_{front}). \tag{3.2}$$

From this equation we can conclude that: (i) the visibility of a surface i (v_i) is determined by its own opacity (α_i), (ii) the visibility of a surface i (v_i) is also determined by the transmittance in front of it in the path to the eye e (τ_{e_i}), and (iii) the final pixel color (C_{pixel}) comes from the sum of all of its n fragment colors (C_i) weighted by their respective visibilities (v_i):

$$v_i = \alpha_i \times \tau_{e_i}; \tag{3.3}$$

$$C_{pixel} = \sum_{1}^{n} C_i \times v_i. \tag{3.4}$$

There are two forms of synthesizing a virtual scene into an image: (i) by ray-tracing or (ii) by rasterization. Both apply the previous equations to compose transparent fragments, what differs are the techniques used to determine the fragments visibilities. In the next sections, we discuss the peculiarities of such techniques.

3.1.2.1 Ray-Tracing

The ray tracing algorithm (SHIRLEY; MORLEY, 2003) launches rays from the eye to the scene, simulating the reverse path made by the light ray. The first intersection of the ray with an object in the scene will define the pixel attributes. Physical models are used to evaluate these attributes, however, to consider transparency effects, additional rays must be cast from the intersection point through the object, see Figure 3.4.

Ray-tracing (as similar approaches) presents recursive algorithms to apply rendering equations (equations which aim to determine the amount of light leaving any point in the scene). The rendering equation proposed by (WHITTED, 1980) does account for the transmission term, but does not contain any function describing the specificities of each real material (is not physically-based in the materials real properties), limiting the realism it is able to generate. The rendering equation proposed by (KAJIYA, 1986), describes how we can compute the light intensity leaving any specific point of a virtual surface along any direction; however, it is limited by the use of the bi-directional reflectance distribution function (BRDF is a function, or table, describing the light behavior for each incident and reflected direction, collected from real material samples), which describes the light interaction with the surface only in terms of reflection, without considering important factors such as propagation and transmission.

The work of (JENSEN et al., 2001) proposes a bidirectional surface scattering reflectance distribution function (BSSRDF) which does account for propagation and transmission of light, improving realism. The computational cost for the evaluation of these rendering equations is very high, because they account for the integration of many interactions of light (light coming from lights sources, reflections, transmissions, etc). In practice, the algorithms are simplified to account only for few interactions.

To combine the properties of each intersection point and properly evaluate the final color, values must to be computed in view direction proximity order, in other words, in the order in which they are intercepted by the ray. This is done in ray-tracing by geometry sorting techniques.

Light interaction between distinct objects is expensive but easily handled by global illumination models. When dealing with local models (such the Phong illumination model commonly applied in raster-based pipelines), the techniques apply different approaches to guarantee the correct mix of the fragments. Most of them implies in some kind of ordering, with a high granularity level by sorting the fragments of each pixel, or in a less accurate manner by sorting the geometry.

3.1.2.2 Rasterization

Differently from the ray-tracing, the image generation by rasterization processes the geometry generating fragments, which are routed to the corresponding pixel by their x and y projected coordinates. For opaque rendering, the z-buffer (CATMULL, 1978) is used to determine the front-most fragments and discard the remaining ones which are routed to the same pixel. For transparent rendering, there is no standard solution (as the z-buffer is for opaque rendering). Several techniques were (and still are being) proposed, each one with some limitation. Most of them tries to take advantage of the growing programmability of graphics hardware, arising new challenges, such as control the concurrent memory access in order to avoid read-modify hazards.

In raster-based pipelines, the computation of transparency is simplified based on the assumption that there is no refraction when light passes from one medium to another. In this formulation, computing the correct transparency requires properly blending fragment colors, considering their surface opacities and the order they are intercepted by the view ray, as described in (PORTER; DUFF, 1984). For transparency to be correctly computed, fragments must be composed in the same order of their distance to the camera, either in front-to-back (FTB) or back-to-front (BTF)[1] ordering.

Fragments combined in unsorted depth order might have their contributions improperly evaluated. For example, consider the computation of a pixel color involving three fragments. Given that the closest and farthest fragments are almost totally transparent, and the middle fragment is opaque, if the opaque fragment is the last one to be composited, the incorrect contribution of the farthest fragment (which should be occluded) can not be discarded.

Figure 3.5(a) illustrates the expected result after rasterization of three triangles. In this example, the green triangle is the closest to the camera, while the red one is the farthest. The BTF blending equation expects triangles in the following order: red, blue, and green (as in the Painter's Algorithm (FOLEY et al., 1990)). This way, the green contribution is correctly identified as the nearest to the camera. Figure 3.5(b) illustrates color blending in incorrect depth order. Note that the green triangle is not drawn in front of the others, due to the out-of-order blending. Table 3.1 and 3.2 give numerical examples for both situations.

The main problem for raster-based rendering of transparency, is the depth sorting required to evaluate the visibility of fragments; more specifically, the transmittance in the path from the fragments to the eye (τ_{e_i}). The next section describes techniques proposed in the literature for solving raster-based transparency rendering; all of them try to balance between accuracy, memory consumption, and processing time. It is very difficult to generate correct results in real time without extrapolate the amount of memory used.

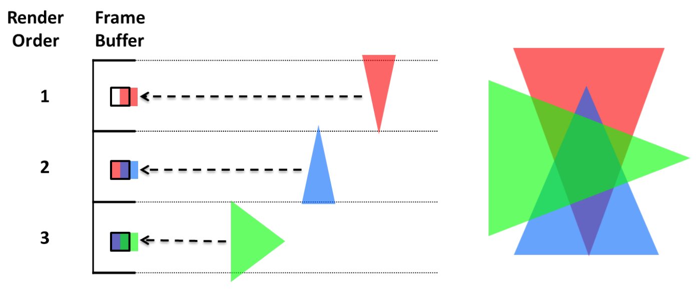
[1]Back-to-front blending equation(PORTER; DUFF, 1984): $C'_{dst} = (1-\alpha)\, C_{dst} + \alpha\, C_{frag}$

Fragment	RGBA	Depth	$BTF_{blended\ result}$
background	(1, 1, 1, 1)	∞	(1, 1, 1, 1)
1	(1, 0, 0, 0.4)	3	(1, 0.6, 0.6, 0.76)
2	(0, 0, 1, 0.4)	2	(0.6, 0.36, 0.76, 0.616)
3	(0, 1, 0, 0.4)	1	(0.36, 0.616, 0.456, 0.53)

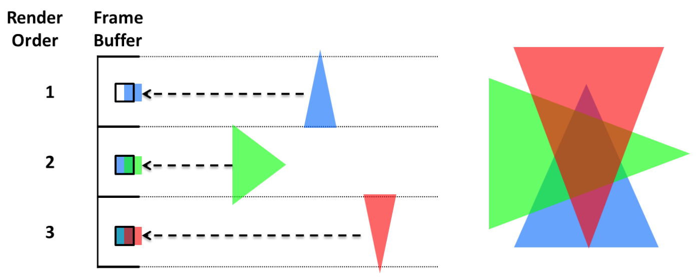
Table 3.1: Numerical example of a pixel evaluated with the **correct** BTF order. Each line represents a fragment blending order (top to bottom).

Fragment	RGBα	Depth	$BTF_{blended\ result}$
background	(1, 1, 1, 1)	∞	(1, 1, 1, 1)
1	(0, 0, 1, 0.4)	2	(0.6, 0.6, 1, 0.76)
2	(0, 1, 0, 0.4)	1	(0.36, 0.76, 0.6, 0.616)
3	(1, 0, 0, 0.4)	3	(0.616, 0.456, 0.36, 0.53)

Table 3.2: Numerical example of a pixel evaluated **out-of-depth-order** with the BTF blending equation. Each line represents a fragment and the blending order (top to bottom).



(a) Depth ordering: correct blending



(b) Out-of-order: incorrect blending

Figure 3.5: Blending evaluation for green, blue, and red triangles (in FTB viewing order). Diagrams on left illustrate the rasterization order. The arrow length represents triangles depth. Images on right are rendering results after blending. Primitives processed in-depth order (a) produce correct results, differently from objects processed in out-of-order (b).

3.2 Transparency Techniques

Transparency is an important effect for several graphics applications. Correct transparency rendering requires fragment sorting, which can be more expensive than sorting geometry primitives, and can handle situations that might not be solved in geometry space, such as object interpenetration. In this section, we survey different transparency techniques and analyze them in terms of processing time, memory consumption, and accuracy. Ideally, the perfect method computes correct transparency in real-time with low memory usage. However, achieving these goals simultaneously is still a challenging task. We describe features and trade-offs adopted by each technique, pointing out pros and cons that can be used to help with the decision of which method to use in a given situation.

Transparency is the physical property of materials that allows light to pass through objects. This property is important to estimate the appearance of real objects, and it is largely used to denote the relationship among structures in visual interaction. The focus of this survey is to summarize and describe specific techniques for rendering transparent objects using raster-based pipelines as described in Section 3.1.2.2.

As we can conclude, sorting is the main topic of the techniques we survey, and we use it as the criteria to classify methods into the following categories:

- **Geometry-Sorting**: sort geometry (meshes or primitives) before rasterization;
- **Fragment-Sorting**: sort rasterization fragments before blending, using buffer-based or depth peeling;
- **Hybrid-Sorting**: combine geometry-sorting with fragment-sorting;
- **Depth-Sorting-Independent**: blend fragments without considering their depth-order;
- **Probabilistic**: estimate visibility without sorting.

We organize our presentation using the above classification in the sections that follow. To clarify the discussion, parameters used by the methods are given in Table 3.3.

W:	screen width,
H:	screen height,
S:	number of samples for super sampling,
m:	number of objects,
n:	number of geometry primitives ($n >> m$),
p:	pixel size: 12B (8B: RGB 8b per channel, 8b alpha, 4B depth value),
l:	number of transparent layers,
d:	average number of transparent layers ($d < l$),
k:	buffer entries per pixel (in fragments),
f:	min(k, d),
s:	samples per pixel.

Table 3.3: Parameters used by different methods.

3.2.1 Geometry-Sorting Methods

The geometry-sorting methods render transparent objects (usually composed of triangle meshes) by either sorting the objects, or at a finer granularity, by sorting their primitives (i.e. triangles). Both approaches are simple to implement and can directly leverage the alpha-blending support on GPUs (Graphics Processing Units). These methods sort objects or primitives before rasterization and are arguably the most widely used transparency technique in games.

Two situations can lead to image artifacts when using geometry-sorting methods:

interpenetrating geometry and out-of-order arrivals. Interpenetrating geometry makes it difficult to properly sort objects and primitives, which leads to blending artifacts. Out-of-order arrival can arise when the technique sorts entire meshes instead of triangles, which can also lead to blending artifacts. Both situations can be properly handled by sorting at the fragment level.

3.2.1.1 Object Sorting

This approach sorts objects as single entities, usually by the centroid of their meshes. It is prone to artifacts due to the out-of-order arrival of fragments, since object ordering is approximate and does not guarantee correct ordering at the primitive level.

Object sorting methods first sort all the m objects in O($m \log m$), followed by rasterization of object primitives using O($n + WH$) operations, and blending using O(WHd) operations. The scene is rendered in one geometry pass, with no need for extra memory, apart from the color and depth buffers, which require O(WHp) bytes.

3.2.1.2 Primitive Sorting

Sorting geometry at the primitive level is the finest granularity that can be obtained in object space. It allows solving the out-of-order problem when no interpenetration occurs. One way to solve interpenetration cases is to split primitives, which might have a high computational cost and still generate z-fighting problems. Another option is to solve interpenetration analytically at even higher costs.

Primitive sorting requires sorting n primitives (triangles) in O($n \log n$) operations. The remaining steps and analysis are analogous to the object-sorting given above.

3.2.2 Fragment-Sorting Methods

Fragment-sorting methods compute transparency by z-sorting at fragment level before blending. Figure 3.6 shows an example of out-of-order rendering of triangles, where fragments are sorted before blending. We further subdivide the fragment-sorting methods into two categories:

- **Buffer-Based** methods store and sort the fragments before blending;
- **Depth Peeling** methods extract depth order implicitly through a multi-pass rendering approach.

The main advantage of fragment-sorting methods is the quality of the image, often superior to all other methods. On the other hand, the computational cost and/or memory footprint, due to sorting, is considerably higher.

3.2.2.1 Buffer-Based Methods

Buffer-based methods use a buffer to store fragments while they are generated. After rasterization, a sorting step computes the correct blending ordering. The advantage of these methods is image quality, when compared to sorting-based methods like geometry-sorting, and performance, when compared to depth peeling methods, at the expense of high memory consumption. Ideas for handling the storage issue in several methods are given below.

3.2.2.1.1 A-buffer

The A-buffer (CARPENTER, 1984) (accumulation, area-averaged, and anti-aliased buffer) was the first method to address the order-independent transparency problem, but its

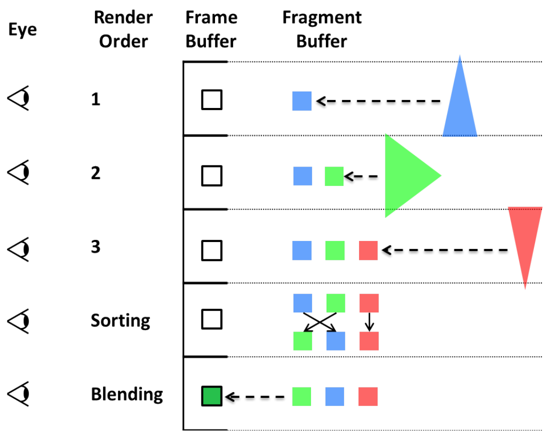


Figure 3.6: Fragment re-ordering example. A buffer holds all out-of-order fragments. A post-rasterization phase sorts these fragments by depth. Finally, the fragments are correctly blended in depth order.

original focus was a hidden surface algorithm with support for anti-aliasing. The algorithm first stores *all* fragments per pixel during rendering, followed by a sorting step to reorder fragments by depth. In a final step, fragments associated to each pixel are blended in depth-sorted order.

The A-buffer stores for each pixel of the final image either a color or a list of all fragments associated with that pixel, as shown in Figure 3.7. Fragments are appended to the appropriate pixel list as they are generated during rasterization. After rasterization, the list of fragments of each pixel is sorted and blended in FTB (or BTF) order.

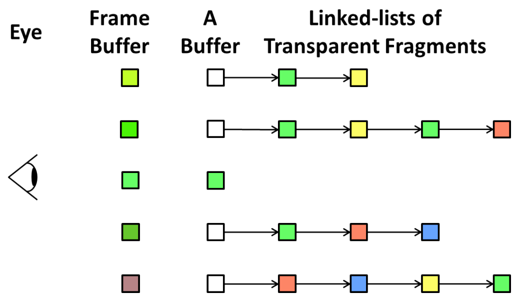


Figure 3.7: A-buffer stores for each pixel with transparency either a color (for opaque pixels) or a linked list of fragments.

The high image quality, at the cost of performance, makes the A-buffer suitable for offline applications, such as animation movies and special effects. The method represents a robust solution for order-independent transparency, visibility, and anti-aliasing. However, it uses unbounded memory and requires random memory access imposed by the linked lists of fragments. Such limitations place constraints on the achievable performance of the method and are addressed by the techniques that follow.

The A-buffer first rasterizes all geometry primitives in O$(n + WH)$ operations, generating and storing *all* the O(d) fragments per pixel. After rasterization, the fragment list of one pixel is sorted in O$(d \log d)$ operations, and traversed to blend fragments in O(d). For the entire screen, it takes O$(WHd \log d)$ operations for sorting and O(WHd) operations for blending. The scene is rendered with one single geometry pass, with *all* fragments stored on a per-pixel basis, taking O$(WHdp)$ bytes.

3.2.2.1.2 Z^3

Similarly to the A-buffer, the Z^3 technique (JOUPPI; CHANG, 1999) uses a buffer for holding multiple fragments per pixel; those are also sorted before blending. However, instead of relying on unbounded memory, the Z^3 constrains the number of fragments stored per pixel. This constraint alleviates memory requirements, but it introduces image artifacts.

Instead of keeping all transparent fragments generated by rasterization for each pixel, the Z^3 algorithm holds only k fragments per pixel. When rasterization generates more than k fragments for a given pixel, an overflow occurs. Upon overflow, once the pixel fragments in the buffer are sorted, the nearest fragments are blended and stored back in the buffer, allowing the algorithm to continue (without requiring more memory). The choice of which fragments to blend is guided by the number of samples covered by each fragment. Fragments with fewer samples have less contribution in the final color, and therefore their blending when overflow occurs is bound to generate less error. Once rasterization finishes and overflows are properly handled, fragments left in the buffer are blended in FTB order.

Besides carefully choosing the fragments to blend (upon overflow), the Z^3 algorithm tracks information about the depth of each fragment, which includes three depth values and a set of coverage samples. The depth values—i.e., a central z and two slopes (in x and y directions)—enable a better treatment of interpenetrating fragments by reconstructing good approximations of the depth in each sample. Figure 3.8 illustrates the Z^3 data structure.

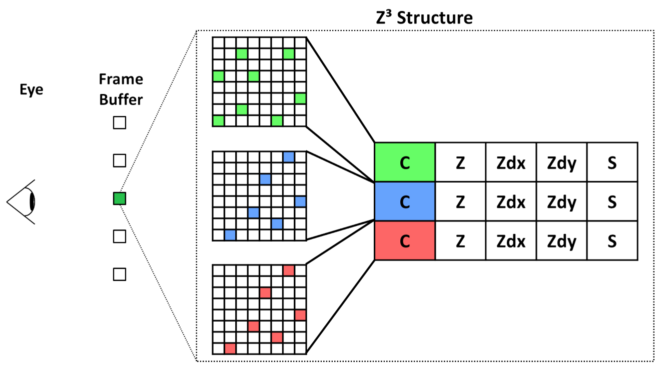


Figure 3.8: Z^3 structure stores information about fragments: color samples (C), central depth (z), slopes in x and y directions (Z_{dx} and Z_{dy}), and a stencil bit (S). Figure adapted from (JOUPPI; CHANG, 1999).

The Z^3 technique has lower memory cost than the A-buffer, since it uses a fixed amount of memory to evaluate pixel colors, compared to the unbounded memory of the A-buffer. In addition, Z^3 improves the precision of the blending by storing more depth information about the fragment samples. The main drawback is on image quality, due to the incorrect blending when overflows occur, and on performance, because it needs atomic operations to test and write into the buffer.

Z^3 computes transparency in one geometry pass with O($n + WH$) raster operations, storing O(WHd) fragments into k slots per pixel, which takes O($WHkp$) bytes. While fragments are stored, their depth-order is updated at a cost of O($WHd \log f$) operations, since during overflow one fragment may be compared against f previously stored values to find the two nearest to be blended. Z^3 processes all fragments and consequently blending takes O(WHd) operations.

3.2.2.1.3 FIFO Buffers

In this section we review two FIFO buffer methods (F-buffer and R-buffer) that use intermediate buffers to store fragment information produced by rasterization. Both methods are inspired on the A-buffer, but use different approaches to store the incoming fragments. A similar architectural proposal is presented in Amor et al.(AMOR et al., 2006).

The F-buffer (fragment-stream-buffer) (MARK; PROUDFOOT, 2001) stores *all* incoming fragments in a FIFO buffer during rasterization. Once rasterization ends, the fragments in the F-buffer are processed in a multi-pass fashion, which peels one transparent layer of the scene at each pass by testing against a depth map. Rejected fragments are sent to another F-buffer, which serves as input in the next pass. Selected fragments compose the transparent layer and are blended to an accumulation buffer. When the F-buffer is empty, the accumulated color buffer is written into the framebuffer.

The R-buffer (WITTENBRINK, 2001) is a modified A-buffer that stores all fragments of a scene sequentially in an FIFO buffer, instead of using a linked list approach. Order-independent transparency is computed in multiple passes over the stored fragments, similarly to the F-buffer, but using two R-Buffers and an extra Z-buffer. In the first pass, the closest opaque fragment is captured in the framebuffer and the un-occluded transparent fragments are stored in the R-buffer.

Multiple passes process the fragments captured in the R-buffer. In each pass, the input R-buffer is traversed to identify the farthest transparent fragment of each pixel to blend into the framebuffer while writing the remaining fragments into an output R-buffer. The input and output R-buffers switch roles at each pass, and computation proceeds until the output R-buffer is empty (Figure 3.9).

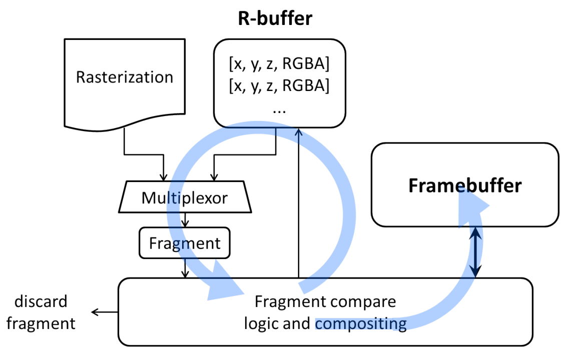


Figure 3.9: R-buffer interface, with recirculating pipeline, can perform OIT in one single geometry pass, with many passes over the stored fragments. The multiplexor selects the input from the rasterizer or the R-buffer. Fragments are tested and blended, or sent back (recirculated) to the R-buffer. Figure adapted from (WITTENBRINK, 2001).

The methods described in this section are similar. Both methods use a single geometry pass to capture all fragments of a scene, followed by multiple sorting and blending passes over the fragments stored in FIFO buffers. The main difference between them is that the F-buffer does not allow writing into the frame buffer until the processing is done, whereas the R-buffer operates directly in the framebuffer.

Like the A-buffer (CARPENTER, 1984), the main drawback of both methods is the unbounded memory usage, since these methods need to store all incoming fragments, which can exhaust memory. Differently from the original A-buffer, these approaches do not use linked lists, which is emulated by storing additional information per fragment and performing multiple passes over the stored data. In a multi-thread system, they have the advantage of not suffering from read-after-write hazards. This is caused by many threads trying to store fragments into the same pixel list, as fragment storage is not directly associated to each individual pixel.

FIFO techniques handle transparency in a single geometry pass with O($n+WH$) raster operations. The generated fragments are stored in the FIFO buffer in O(WHd) operations taking O($WHdp$) bytes. In a post-rasterization phase, sorting is performed in O(WHd^2) operations due to the disassociation of fragments to their pixel location. During sorting, fragments are blended in O(WHd) operations.

3.2.2.1.4 Stencil-routed A-buffer

The A-buffer variation given by Myers and Bavoil (MYERS; BAVOIL, 2007) uses the stencil buffer to route fragments into a multi-sample buffer. It leverages multi-sample anti-aliasing (MSAA) to capture multiple fragments per pixel, which are later processed by a pixel shader for sorting and blending.

The scene is first rasterized to capture up to k fragments per pixel, where k is the maximum number of samples available in the MSAA implementation. Each incoming fragment is routed to a sample of the MSAA buffer. If there are more than k fragments per pixel, overflow occurs and an additional geometry pass is needed. Fragment routing uses a stencil mask that stores an incoming fragment in the next free MSAA sample, building a vector of fragments per pixel. Once all fragments are captured, a full-screen quadrilateral is rendered with a pixel shader that reads all fragments of a given pixel; it sorts and blends them accordingly. Figure 3.10 illustrates the stencil-routed algorithm for a single pixel.

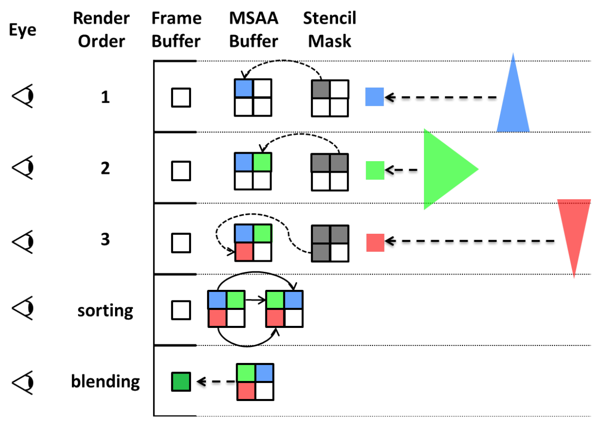


Figure 3.10: Stencil-routed evaluation for one pixel. The stencil mask routes fragments into the MSAA buffer. The fragments are then sorted and blended accordingly, in a post-rasterization stage.

Since the stencil-routed A-buffer uses hardware resources, the use of MSAA and stencil buffer is not supported by this method. In terms of performance, the Stencil-Routed A-buffer is up to k times faster than the basic *depth peeling* method to be described in Section 3.2.2.2. It stores up to k fragments per pixel on each geometry pass, using O($n + WH$) raster operations per pass. When the number of transparent fragments exceeds the k value, the algorithm needs O(l/k) geometry passes to properly evaluate transparency, with l limited by the 8bit-word of the stencil buffer. Writing of all fragments takes O($WHkp$) bytes. A subsequent sorting of all lists (in chunks of k fragments per pass) takes O($d \log k$) operations per pixel, followed by O(WHd) blending operations.

3.2.2.1.5 FreePipe

Liu et al. (LIU et al., 2010) proposed *FreePipe*, an implementation of the entire graphics pipeline in CUDA. FreePipe allowed two variations of the A-buffer to be explored, based on a fixed-size vector of k fragments per pixel (Figure 3.11).

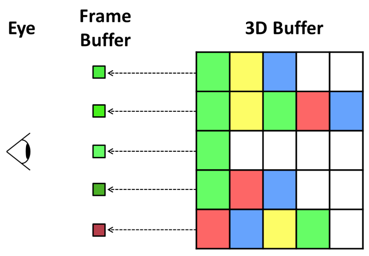


Figure 3.11: Side view of the buffer used in the freePipe approach, which has a fixed number of slots per pixel to store fragments.

The first approach tests each incoming fragment against the corresponding buffer entry (using atomic operations available in graphics hardware) to find its corresponding z-sorted location in the list. If keeping the correct sorting order is needed, fragments already stored in the buffer are moved to other positions. This data movement increases memory traffic, but results in the sorted list of fragments per pixel—i.e., no sorting is needed, after the rasterization pass is finished. It can also lead to blocking threads that compete to store fragments into the same pixel list, because the store operation must be atomic. At the end of this process, the fragments are ready for blending.

The second approach uses a counter index per-pixel, which indicates the next available position in the pixel list to store the incoming fragment. Only this counter is updated atomically. The fragments are stored in order of arrival rather than in sorted order, avoiding more atomic operations (and their hazards). In a post-processing phase, the fragments are sorted and then blended.

The limitation of both approaches is the large memory requirements because of the fixed-size buffer, as well as loss of fragments when overflow occurs. The first approach can detect pixel saturation and early terminate the processing of additional fragments. The second approach, similarly to the A-buffer, captures all fragments per pixel, before they are sorted. Both approaches allocate the same number of entries per pixel, regardless of whether or not the pixel has transparent fragments, leading to waste of memory. Atomic operations degrade performance, since they serialize threads associated to the same pixel locations.

Both approaches compute transparency in a single geometry pass with O($n + WH$) raster operations, using O($WHkp$) bytes and performing blending in O(WHk) operations. The first one sorts fragments at generation time, comparing against previously stored values. This requires O(df) operations, and does not need sorting before blending. The second one stores fragments upon generation and sorts them in O($f \log f$) in a separate step after rasterization.

3.2.2.1.6 Linked Lists

The creation and the update of linked lists entirely on GPUs, using atomic operations in shader programs, is explored in the proposal of Yang et al. (YANG et al., 2010). Since current GPUs do not allow dynamic memory allocation, two buffers are used to emulate the A-buffer linked lists. The first buffer holds, for each pixel, an index to the last rasterized fragment. This index points to a location in the second buffer, which keeps all fragments generated for all pixels. Upon the arrival of a new fragment, the index is updated to point to the new entry, which receives the previous index. Thus, each entry points to the previously-rasterized fragment, forming a linked list. Figure 3.12 shows the first buffer, called *head pointer*, the second buffer, *node buffer*, and their states after the rasterization of three triangles, where two transparent fragments are generated for a pixel.

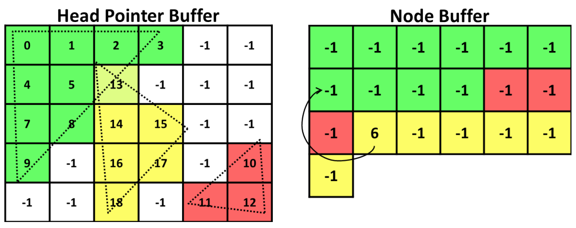


Figure 3.12: Head pointer buffer and node buffer states after inserting the green, red, and yellow triangles, respectively. The head pointer buffer points to the last fragment inserted for each pixel in the node buffer. The last fragment inserted in the node buffer points to the previous fragment. Figure adapted from (YANG et al., 2010).

Since the node buffer is shared, a single global counter is used to index the next available position where a thread can store an incoming fragment. This shared counter needs to be updated atomically, which represents a bottleneck, since all threads trying to store a new fragment will be blocked here. The random memory accesses of the linked list also degrades performance. The main advantages of this method include the fact that no memory is allocated for pixels or layers that are not covered (the Node Buffer size may be controlled to fit the number of fragments generated), and the way it leverages the graphics pipeline already optimized in hardware.

3.2.2.1.7 Adaptive Transparency (AT)

Adaptive transparency (SALVI; MONTGOMERY; LEFOHN, 2011) is an OIT technique that combines fragments out of depth-sorted order by processing the geometry twice. The first geometry pass computes and stores a visibility function, which is used to weigh the composition of shaded fragments in the second pass.

First, the geometry is rendered without shading. The depth value of the fragments is used to build a visibility function inside a fixed size buffer per pixel, which corresponds to storing the visibility and depth of some fragments in depth-sorted order. During the insertion of the fragment as a node of the list, visibility is computed by combining the fragment alpha and the visibility list using the principles described by Porter and Duff (PORTER; DUFF, 1984). In the case of overflow, the node removed is the one that causes the smallest modification in the area below the function. As shown in Figure 3.13, visibility may be over or underestimated. In the second geometry pass, the depth of the shaded fragment is used to evaluate its visibility according to the previously computed function. This can be achieved by searching the list for the depth interval containing the fragment z, and getting the visibility at that point.

Since the technique only stores depth and visibility values, it uses less memory than methods that also store fragment colors. The price of using less memory is an extra geometry pass to shade the fragments.

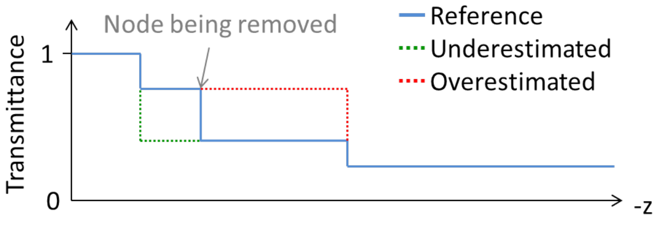


Figure 3.13: Adaptive transparency visibility compression: A node removal may lead to an overestimation (red), or to an underestimation (green), of the remaining visibility, indicated in the transmittance axis. Image adapted from (SALVI; MONTGOMERY; LEFOHN, 2011).

3.2.2.1.8 Per Pixel Paged Linked Lists (PPPLL)

PPPLL (CRASSIN, 2010a) is an extension of the Per Pixel Linked Lists (PPLL), proposed by Yang et al. (YANG et al., 2010). While PPLL builds a linked list of nodes with one fragment per node, PPPLL allows storing more fragments per node, favoring memory spatial locality.

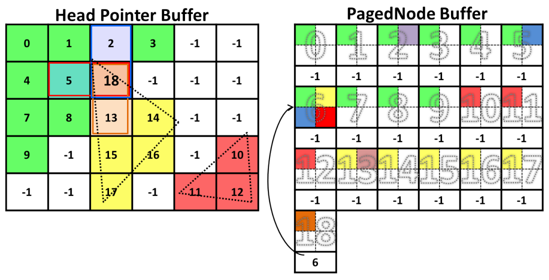


Figure 3.14: Per-pixel paged linked lists: example showing three transparent triangles and three transparent lines, which are illustrated over the Head Pointer Buffer. The **Head Pointer Buffer** keeps a pointer to the PagedNode Buffer, which indexes the page of the last received fragment from the respective pixel. For example, number 18 in the Head Pointer Buffer indicates that the last fragment for that pixel entry is stored in page 18 of the PagedNode Buffer. The **PagedNode Buffer** keeps fragment attributes in pages of size four per node, with one index to the next page and if there are no more pages, it indicates with -1. For example, in page 18, the number 6 indicates that the continuation of the list is in page 6. Image modified from (MAULE et al., 2011).

The technique works in a single geometry pass, by storing all incoming fragments in a shared buffer. These fragments are blended in a post-processing phase. To emulate the linked list per pixel, it requires two buffers. The first one has one entry per pixel, which points to the head of the pixel list into the shared buffer. The shared buffer has nodes composed by a pointer to the next node and a page with up to k fragments. When the page of a node is full, a new fragment causes the allocation of a next node, which is indexed by the previous node pointer. Figure 3.14 illustrates the technique.

Node allocation is protected by critical sections, which reduces parallelism. Since it does not count fragments, in overflow cases the re-allocation of the shared buffer is heuristical and can be repeated until the storage contains the required size.

3.2.2.2 Depth Peeling-Based Methods

Depth peeling methods extract layers of visibility from a graphical model using multiple rendering passes. Layers are captured in depth order, which eliminates the need to sort the resulting fragments. Transparency is computed by blending transparent layers in FTB or BTF order. In this section, we summarize three depth peeling variations.

3.2.2.2.1 Virtual Pixel Maps

The virtual pixel maps (VPM) introduced by Mammen (MAMMEN, 1989) proposed the first depth peeling technique. Transparency is computed in multiple geometry passes that extract one depth layer in each pass. The first geometry pass renders all opaque objects into an opaque pixel map, which stores a depth and a color value per pixel. At each subsequent geometry pass, the un-occluded fragments (with respect to the currently captured layer) are tested to find the nearest to the previously captured layer. After each pass, selected fragments are blended with the color already stored, and corresponding depth values updated. The process ends when no more fragments are blended, or when the number of passes exceeds a user-defined threshold. Figure 3.15 shows the evaluation for a pixel using VPM.

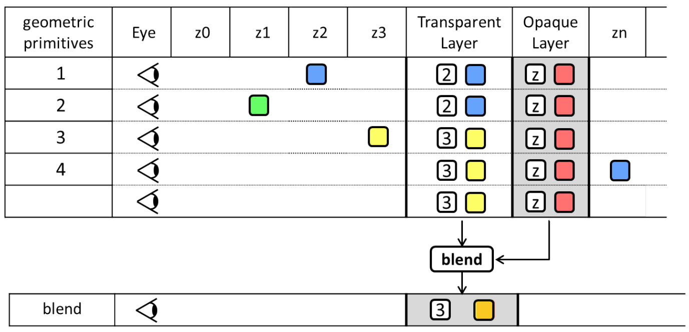


Figure 3.15: VPM transparency evaluation for one pixel. In each pass, the visible fragment nearest to the opaque layer is blended to the opaque fragment and its depth is updated. Figure adapted from (MAMMEN, 1989).

VPM does not have large memory requirements, because it needs only one extra z-buffer and one extra framebuffer to peel the occluded fragments and accumulate the nearest fragments colors. Memory requirements double the color and depth buffer O(WHp) bytes. The geometry passes can be costly for scenes with high depth complexity. For each transparent layer, VPM needs a geometry pass, totalizing O($l(n + WH)$) raster operations for all layers, at the same time that layers are blended in O(WHd).

3.2.2.2.2 Interactive Order-Independent Transparency

The interactive order-independent transparency (iOIT) technique presented by Everitt in (EVERITT, 2001) uses a depth peeling approach that leverages the implementation in graphics hardware of the shadow-mapping algorithm (WILLIAMS, 1978). It starts by using the z-buffer to extract the transparent layer nearest to the eye. The following passes use the previous z-buffer values to find the subsequent nearest layers by eliminating all fragments that are in front or match the current layer. Remaining fragments are depth sorted to generate the frontmost fragment behind the current layer—i.e. the next depth layer. After each geometry pass, the n^{th} layer is ready to be blended in the color buffer.

An approach to peel two depth layers in each geometry pass is proposed by Bavoil and Myers (BAVOIL; MYERS, 2008). The first and last visible layers are computed in each geometry pass (an example is given in Figure 3.16). Since it processes depth peeling simultaneously in FTB and BTF fashion, it can be twice faster than the iOIT depth peeling.

The advantage of iOIT techniques over Virtual Pixel Maps is the FTB approach, which enables early termination when pixel opacity saturates. This means that iOIT needs up to l geometry passes to cover all layers. Similarly to VPM, iOIT does not have large memory requirements, but it also incurs in repeated geometry passes. The remaining steps and analysis are analogous to VPM described above.

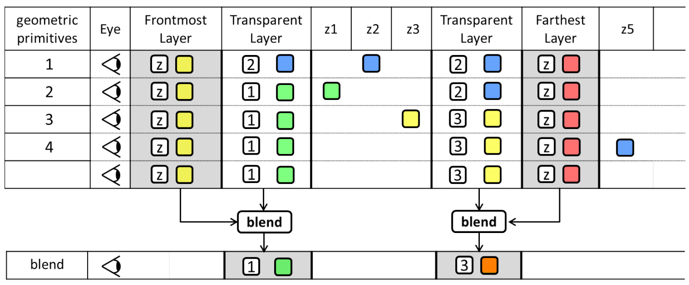


Figure 3.16: Dual depth peeling transparency evaluation for a pixel. In each geometry pass, the visible fragment farthest from the eye is composed into the farthest layer and its depth is updated. In the same pass, the visible fragment nearest to the eye is blended into the frontmost layer and its depth is updated. This example has the same configuration used in Figure 3.15. Figure adapted from (BAVOIL; MYERS, 2008).

3.2.2.2.3 Bucket Sort

Two GPU-based bucket sort (BS) implementations were proposed by Liu et al. in (LIU et al., 2009) to peel multiple depth layers per geometry pass. It allows peeling up to 32 layers in each pass using multiple render targets to store buckets.

In the first implementation, the depth range is uniformly subdivided and each subdivision is mapped to one bucket. Incoming fragments are mapped to the bucket corresponding to its depth value. A collision occurs when more than one fragment is mapped to the same bucket, which may be alleviated using a two-pass approach over the geometry. The first geometry pass renders the scene bounding box into bucket intervals to obtain a depth estimate for each pixel. The depth range is subdivided according to fragment occupation into fixed intervals. In the second pass, each incoming fragment is mapped into the bucket corresponding to its depth interval. See Liu et al. (LIU et al., 2009) for more details and the mapping equation.

In the fixed approach, some buckets might remain empty while others are overloaded, according to the depth complexity distribution of the scene. Instead of finding the ideal subdivision that maps a fragment into each bucket, the second implementation uses an adaptive approach. To create intervals that better fit the depth density distribution, an equalized-depth histogram of the geometry distribution is computed in a first geometry pass (Figure 3.17). This histogram is used in the second pass to map incoming fragments into the correct depth buckets (Figure 3.18).

The bucket sort is suitable for scenes with uniform depth distributions. For scenes with a high concentration of fragments in a small depth interval, or for scenes with high depth ranges, it may generate artifacts due to the irregular mapping of fragments into buckets. The advantage is the low cost in mapping fragments ready to be blended.

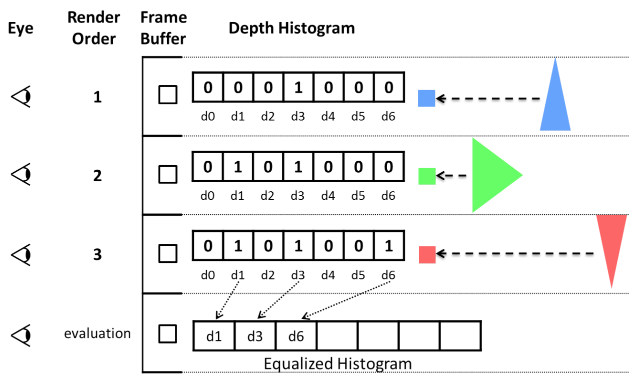


Figure 3.17: Equalized histogram evaluation. The depth histogram is a binary vector, with each entry corresponding to one depth value. A value 0 means that there is no fragment mapped to that entry, whereas a value 1 means that at least one fragment is mapped to that depth range. For each incoming fragment, its z value is used to set a bit indicating its presence in the depth histogram. At the end of the geometry pass, the equalized histogram is computed with the depth values that are set to 1 in the depth histogram.

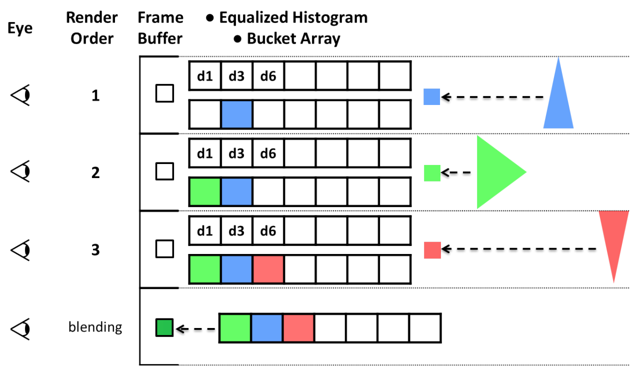


Figure 3.18: Color evaluation for a pixel using BS. The incoming fragments are mapped by the equalized-depth histogram into the correct buckets by testing their depth values.

The bucket sort technique needs two geometry passes. One to calculate the equalized histogram, and another to render the scene; both with O$(n + WH)$ raster operations. The second geometry pass takes O(WHd) operations to route fragments into O$(WHkp)$ bytes. Since fragment lists are generated in order, there is no need for sorting, and lists are blended in O(WHf) operations.

3.2.3 Hybrid Methods

The methods discussed so far compute transparency by sorting geometry primitives (before rasterization) or raster fragments. In such methods, only one of these two approaches is explored. On the other hand, hybrid methods compute transparency using both approaches. In this section, we review two hybrid methods.

3.2.3.0.4 Image-Space Queries

The image-space queries (ISQ) algorithm (GOVINDARAJU et al., 2005) describes an approach for ordering non interpenetrating geometry models. ISQ is similar to the primitive-sorting techniques described in Section 3.2.1.2, since it computes a BTF order of primitives before rasterization. However, instead of sorting primitives according to the distance to the eye in object space, ISQ uses occlusion queries in image-space to computer a correct BTF ordering among primitives. Cycles in the visibility ordering can be generated, but they are identified and resolved using occlusion queries over the triangles of objects involved in cycles.

The primitive rendering order is computed by sorting pairs of primitives using their normalized depth value. The comparison uses the occlusion query capability of GPUs by simply rendering two primitives, and evaluating the smallest depth to define the triangle closer to the viewpoint. Sorting all pairs of primitives in this fashion gives a visibility graph to render in BTF ordering.

ISQ rasterizes n primitives, by pairs, to perform visibility comparison, which takes O($n \log n$) rendering operations. Once the primitive list is sorted, it is rasterized in O($n + WH$) operations. Since most objects are already sorted due to temporal coherence between frames, the algorithm uses the list from previous frames as input for the next frame sorting, therefore, execution is expected to run in linear complexity. Memory requirements are minimal, only the standard color and rendering buffers.

3.2.3.0.5 The k-Buffer

A technique to render unstructured grids was described by (CALLAHAN et al., 2005) and later generalized by (BAVOIL et al., 2007) to handle multi-fragment effects (including transparency). The core of the technique is the k-buffer, a fixed-size fragment buffer that holds up to k fragments per pixel. The technique first uses an approximate sorting in object-space, which is not guaranteed to produce the correct ordering, but it allows for fragments to be generated during rasterization in a nearly-sorted fashion. This important nearly-sorted property allows the sorting to be concluded in image-space with a k-buffer, which only needs to have as many entries as necessary to fix the ordering of samples. Fragments are composited when the k-buffer becomes full (Figure 3.19), to make room for other incoming samples, and after all fragments are generated (which requires the k-buffer entries to be flushed).

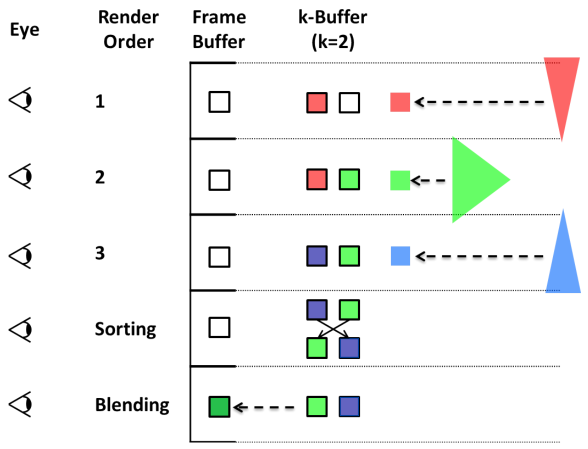


Figure 3.19: Color evaluation for one pixel using a k-buffer with two entries per pixel. While the buffer has free entries, fragments are stored in **incoming** order. In the case of overflow, the incoming fragment is tested and combined with the z-nearest entry. In a post-rasterization phase, remaining entries are sorted and blended.

The k-buffer handles interpenetrating geometry, since fragment sorting is involved. The final image quality depends on how well object sorting reduces the number of out-of-order fragments. In situations resulting in poor fragment sorting, the number of entries in the k-buffer might be smaller than necessary, which might introduce artifacts due to out-of-order blending. Also, in its proposition, the algorithm was prone to artifacts caused by read-modify-write hazards during k-buffer updates. This can be overcome today with atomic operations, at the expense of increased processing time due to the cost incurred by such operations.

Sorting is done in object space with an external radix sort in O(m) operations. Geometry is rasterized in O($n + WH$) and fragments are stored into k slots, with possible blending (in overflow cases), costing O($WHd \log f$) operations and O($WHkp$) bytes. Final sorting of k elements takes O($WHf \log f$) and O(WHf) blending operations.

3.2.3.0.6 Multi-layer alpha blending

Inspired by our hybrid transparency technique (HT - see Section 3.4), multi-layer alpha blending (MLAB) is a hybrid approach that replaces HT's OIT tail (back-most fragments) by an order-dependent approach. As HT, MLAB stores the k front-most fragments and, differently from HT, MLAB composites the back-most fragments in arrival order. This approach is susceptible to artifacts when fragments are composited out of depth-sorted order. The computational complexity and memory costs of MLAB are equivalent to those presented by HT.

3.2.4 Depth-Sorting-Independent Methods

The methods described in this section do not require sorting prior to blending fragments. Instead, they use special compositing equations to blend fragments in incoming (unsorted) order. This approach can lead to incorrect transparency results, when the fragments to be blended have very distinct colors and transparencies.

The weighted sum method (MESHKIN, 2007) blends incoming fragments in arrival order using Equation 3.5:

$$C_{dst} = \sum(\alpha_{src} \times C_{src}) + C_{bg} \times (1 - \sum \alpha_{src}) \tag{3.5}$$

where C_{dst} is the resulting color, α_{src} and C_{src} are, respectively, the opacity and the color of the incoming fragments, and C_{bg} is the background contribution. For low alpha values, this approximation is acceptable, but increasing alpha values might lead to overly dark or overly bright results.

The weighted average (BAVOIL; MYERS, 2008) also employs a single-pass blending approach using Equations 3.6, 3.7, and 3.8:

$$C = \frac{(\sum(RGB)\alpha)}{(\sum \alpha)} \tag{3.6}$$

$$\alpha_{dst} = \sum(\frac{\alpha}{N}) \tag{3.7}$$

$$C_{dst} = \frac{C\alpha \times ((1 - (1 - \alpha)^N))}{\alpha + C_{bg} \times (1 - \alpha)^N} \tag{3.8}$$

where RGB represents the three color channels, α is the opacity, N is the number of fragments for the evaluated pixel, and C_{bg} represents the background contribution.

The weighted blended OIT (MCGUIRE; BAVOIL, 2013) technique is an improvement over the weighted average technique. It modifies the equation to achieve correct background visibility, and adds a depth-weighting term that decreases a fragment visibility as its depth increases.

Sorting-independent methods are fast, single-pass, and do not need additional buffers, since fragments are merged upon generation. They are suitable for particle rendering due to the high quantity of particles commonly simulated, and the acceptable error tolerance for this effect. Both methods compute transparency in one geometry pass O($n + WH$), which generates O(WHd) fragments blended using O(WHd) operations. In terms of memory, the basic color and depth buffers O(WHp) bytes are sufficient.

3.2.5 Probabilistic Methods

This section describes techniques that approximate transparency using probabilistic approaches. The main advantage of the stochastic and probabilistic methods is the low processing cost, since pixels are evaluated in a fixed number of geometry passes and fragments are not sorted. The main drawback of these methods is the noise associated to using random sampling. One approach to reducing noise is to increase the number of samples, which increases memory consumption or processing time.

3.2.5.1 Stochastic Transparency

Stochastic Transparency (ENDERTON et al., 2010) is a stochastic solution for transparency which does not require sorting. The algorithm treats transparency as the number of samples covered by the fragment color. For example, an $\alpha = 0.5$ means, roughly, 50% of the pixel samples being covered by a fragment.

This method implements screen-door transparency (MULDER; GROEN; WIJK, 1998) with a random sub-pixel stipple pattern. The original screen-door technique uses a bitmask to select pixels that are not colored, producing holes in the object rasterization proportional to its transparency. The stochastic algorithm uses the hardware MSAA to apply screen-door transparency at samples instead of pixels. Samples are later blended to define the pixel colors, and an example is given in Figure 3.20.

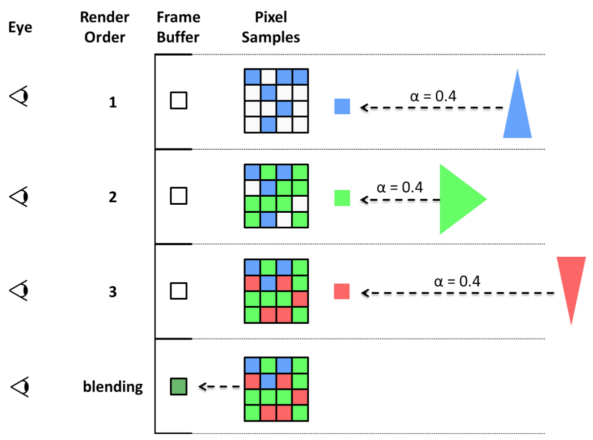


Figure 3.20: Stochastic evaluation of one pixel color. Fragment coverage uses the fragment α as estimate of fragment visibility. A random distribution is applied to avoid using the same bitmask.

A pixel color is defined by s samples, each with its own opacity α interpreted as the probability of receiving the color from an incoming fragment. On average, the number of samples updated with the fragment color is $r = s \times \alpha$. The first pass in the algorithm renders opaque geometry and the background, while transparent objects are handled in subsequent passes. The second rendering pass uses shaders to accumulate the opacity at each pixel, and depths at each sample. These values are used to approximate fragment visibility, in a third rendering pass, without sorting.

The main drawback of stochastic transparency is the noise introduced by the stochastic sample selection scheme, which can lead to flickering in dynamic scenes. The noise can be attenuated by increasing the number of samples. The advantage is the flexibility to improve quality or performance by choosing the number of samples to be used.

This approach uses three geometry passes with O($n + WHs$) operations to compute transparency, generating O(WHd) fragments. For each fragment, it tests s samples that may receive the incoming color from the fragment in O(WHs) operations. After rasterization, samples are merged in O(WHs) operations. Memory consumption is given by the samples used per pixel: O($WHsp$) bytes.

3.2.5.2 Silhouette-Opaque Transparency

The silhouette-opaque method (SOT) (SEN; CHEMUDUGUNTA; GOPI, 2003) is a screen-door approach defined in image-space (similar to (ENDERTON et al., 2010)). Each triangle in the original scene is discretized in smaller primitives. The mesh discretization (screen-door construction) computes a sample distribution per triangle, according to its area in screen-space. To compute this area, the closest point to the viewpoint in the triangle is identified, and the triangle is rotated around this point until its normal becomes parallel to the viewing direction and facing the viewer. The triangle is rasterized to a temporary super-sampled frame-buffer using current camera parameters, and samples are defined by the number of pixels covered by the triangle. Thus, sampling does not depend on triangle orientation.

After a distribution is defined, some samples are removed based on the α value of the triangle so that the number of samples is proportional to the triangle opacity. Figure 3.21 illustrates the entire process, which has to be repeated for every frame. The drawback of SOT is the limitation to scene size imposed by the number of primitive samples to be generated, as well as the noise incurred by the probabilistic sampling. The advantage is the fast rendering due to the single geometry pass and lack of sorting.

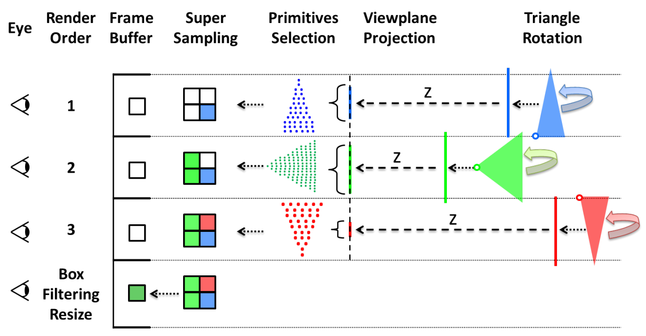


Figure 3.21: SOT color evaluation for one pixel: The triangle is rotated to be parallel to the viewing plane and projected from the distance of the nearest point of the triangle. The projected area is used to estimate the primitives (geometric points) size and quantity, which approximates the amount of light passing through the object. The triangle is rendered into a super-sampled texture, which is resized by a filtering box to compose the pixel color.

The k point samples from each one of the n triangle primitives are rasterized in

O($kn + WHS$) operations. This generates O(WHS) fragments in a S-super-sampled image, which is resized by a box filter in O(WHS) operations. Memory consumption is given by the samples taken per pixel: O($WHSp$) Bytes.

3.2.6 Summary Comparison

In this section we present a comparison among methods with respect to image quality, performance, and memory consumption. To allow for direct comparison, we first summarize in Table 3.4 the complexity analysis discussed for each method in the previous sections.

We continue the comparison among methods in this section by looking at the type of rendering artifacts the methods can generate, an important aspect to be considered when selecting which method to use. We follow this discussion with an estimate of performance using a hypothetical scene that pushes the limits in terms of transparency computation, and present an illustrative chart that compares performance and quality of all methods.

			Memory Consumption	**Computational Cost**				
				Geometry Passes	**Pre Sorting**	**Raster Operations**	**Fragment Sorting**	**Blending**
Geometry		Object(BERG et al., 2000)	$O(WHp)$	1	$O(m \log m)$	$O(n + WH)$	N/A	$O(WHd)$
		Primitive(BERG et al., 2000)	$O(WHp)$	1	$O(n \log n)$	$O(n + WH)$	N/A	$O(WHd)$
Fragment	Buffer Based	A-buffer(CARPENTER, 1984)	$O(WHdp)$	1	N/A	$O(n + WH)$	$O(WHd \log d)$	$O(WHd)$
		Z^3(JOUPPI; CHANG, 1999)	$O(WHkp)$	1	N/A	$O(n + WH)$	$O(WHd \log f)$	$O(WHd)$
		FIFO(MARK; PROUDFOOT, 2001)(WITTENBRINK, 2001)	$O(WHdp)$	1	N/A	$O(n + WH)$	$O(WHd^2)$	$O(WHd)$
		Stencil R.(MYERS; BAVOIL, 2007)	$O(WHkp)$	$\lceil l/k \rceil$	N/A	$O(n + WH)$	$O(WHd \log k)$	$O(WHd)$
		FreePipe(LIU et al., 2010)	$O(WHkp)$	1	N/A	$O(n + WH)$	$O(WHf \log f)$	$O(WHf)$
		Linked Lists(YANG et al., 2010)	$O(WHdp)$	1	N/A	$O(n + WH)$	$O(WHd \log d)$	$O(WHd)$
	Depth Peeling	VPM(MAMMEN, 1989)	$O(WHp)$	l	N/A	$O(n + WH)$	*implicit*	$O(WHd)$
		IOIT(EVERITT, 2001)	$O(WHp)$	$\leq l$	N/A	$O(n + WH)$	*implicit*	$O(WHd)$
		Bucket Sort(LIU et al., 2009)	$O(WHkp)$	2	N/A	$O(n + WH)$	$O(WHd)$	$O(WHk)$
Hybrid		ISQ(GOVINDARAJU et al., 2005)	$O(WHp)$	1	$O(n \log n)$	$O(n + WH)$	N/A	$O(WHd)$
		K-buffer(BAVOIL et al., 2007)	$O(WHkp)$	1	$O(m)$	$O(n + WH)$	$O(WHd \log f)$	$O(WHf)$
		MLAB(SALVI; VAIDYANATHAN, 2014)	$O(WHkp)$	1	1	$O(n + WH)$	$O(WHd \log f)$	$O(WHf)$
Depth-Sorting Independent		W. Sum(MESHKIN, 2007)	$O(WHp)$	1	N/A	$O(n + WH)$	N/A	$O(WHd)$
		W. Average(BAVOIL; MYERS, 2008)	$O(WHp)$	1	N/A	$O(n + WH)$	N/A	$O(WHd)$
		W. Blended(MCGUIRE; BAVOIL, 2013)	$O(WHp)$	1	N/A	$O(n + WH)$	N/A	$O(WHd)$
Probabilistic		Stochastic(ENDERTON et al., 2010)	$O(WHsp)$	3	N/A	$O(n + WHs)$	N/A	$O(WHs)$
		SOT(SEN; CHEMUDUGUNTA; GOPI, 2003)	$O(WHSp)$	1	N/A	$O(kn + WHS)$	N/A	$O(WHS)$

Table 3.4: Complexity comparison. Columns: methods taxonomy; memory consumption (in bytes); geometry passes (count the required number of traversals over the geometry); pre-sorting (sort complexity in object-space); raster operations (count operations to rasterize geometry primitives); fragment-sorting (sort complexity in image-space); and blending (counts blending operations to combine colors). N/A = not applicable.

3.2.6.1 Correct Transparency Rendering

The correct computation of transparency discussed in the methods that we survey is associated to processing *all* transparency layers in *depth-sorted order* with respect to the viewer. Out-of-order blending of fragments might generate incorrect results, as explained in Section 1.

The algorithms that can perform correct transparency computations (given enough time and memory) are: A-buffer, FIFO, Linked Lists, VPM, and iOIT. A-buffer and Linked Lists are the fastest in this set, since they only need one geometry pass to generate and store the transparent fragments for each pixel before blending. On the other hand, FIFO solutions do not associate the fragments to their pixels, requiring more passes over the fragments to properly sort them. This has a higher computation cost compared to A-buffer and Linked Lists. The disadvantage of these algorithms (A-buffer, Linked List, and FIFO) is the amount of memory required to store all fragments before processing them (sorting and blending).

Memory is not a limitation for VPM and iOIT, since they accumulate transparent layers in multiple passes over the geometry, at the expense of a high computational cost. The iOIT has the advantage of processing layers in FTB order, which allows for early termination when pixels saturate, which is not possible in VPM due to its BTF processing.

Correct transparency is costly in both memory and computation aspects. It is appropriate for offline rendering, where image quality surpasses computational cost.

3.2.6.2 Artifacts in Transparency Rendering

Rendering artifacts appear when pixels are colored incorrectly, usually because of out-of-order arrival of fragments, overflow, or insufficient sampling. Artifacts can range from plausible results to completely incorrect pixel colors. Depending on the application, artifacts may be tolerable, especially when comparing the performance gain against the methods that guarantee correct results. We discuss in more details the types of artifacts that can arise.

3.2.6.2.1 Artifacts due to out-of-order arrival

Methods that compute transparency, by blending fragments in incoming order, require geometry sorting before rendering to order fragments by depth. Causes for out-of-order arrival are approximate sorting and interpenetration of primitives, leading to pixels with incorrect colors and possible flickering. ISQ, object and primitive sorting, Weighted Sum and Weighted Average are prone to these artifacts.

In object-sorting algorithms, the low computational cost is due to the low granularity of the sorting, which leads to triangles possibly being rasterized out of order. Increasing the granularity, by sorting at the triangle level, reduces the out-of-order of fragments, but also increases the computational cost. However, interpenetration cases can still occur. Both problems appear in ISQ and require handling overlapping objects, leading to performance loss.

Scenes with few and simple transparent geometry can take advantage of the high performance and low memory consumption of these algorithms. When the scene complexity grows and interpenetrations appear, the quality of the resulting image can decrease rapidly. A special case is when the transparency algorithm does not require any kind of sorting, like in Weighted Sum and Weighted Average. However, correct results can only be obtained for a few restrictive cases. For the general case, blending fragments out of order generates incorrect results, usually presenting itself as overly light or overly dark regions.

Blending fragments in incoming order is the fastest way to perform transparency. However, due to the number of artifacts, it is only recommended for high performance-dependent applications, such as particle simulations.

3.2.6.2.2 Artifacts due to overflow

Overflow occurs when the memory reserved to store partial information is insufficient. This happens in buffer-dependent algorithms when the number of generated fragments is greater than expected, resulting in different problems. If the farthest fragments are lost, or blended in out-of-order, the errors may be minimal, depending on their contribution and the depth complexity of the scene. If this happens to the nearest fragments, artifacts may be severe. Z^3, k-buffer, bucket sort, FreePipe and Stencil routed algorithms may present this kind of artifact.

Z^3, k-buffer and bucket sort do not discard fragments but might merge fragments in incorrect depth-order. Z^3 stores additional depth information to alleviate the incorrect blending, which still might not be enough to handle the case in which an incoming fragment is distant from the previous fragments. Bucket sort can not guarantee that more than one fragment does not fall in the same bucket, which might generate out-of-order blending. K-buffer uses prior sorting in object space to generate fragments in nearly sorted order, which allows the use of a smaller k-buffer. However, there is no guarantee on which size to use.

Such artifacts can generate incorrect blended colors, which may generate flickering when the camera moves. Z^3 and k-buffer can be parameterized to store more fragments, thus increasing computational cost and memory consumption. Bucket sort is limited by the number of render targets, which makes it hard to handle complex scenes with irregular depth distribution. FreePipe and Stencil-routed discard fragments when overflow occurs. This can cause severe artifacts if the lost fragments are the frontmost ones. The Stencil-routed limitation is the 8-bit word of the stencil buffer, unable to address more than 2^8 layers, which might lead to loss of fragments. FreePipe also can be parameterized to store more fragments to address these problems.

Scenes with few transparent layers can be handled efficiently by these algorithms. However, for scenes with irregular distribution of transparent layers across, fixed-size buffers might lead to either memory waste or overflow.

3.2.6.2.3 Artifacts due to insufficient sampling

Transparency computation can be formulated as a sampling problem. In this approach, insufficient sampling might lead to incorrect colors and aliasing, while random sampling might lead to flickering.

Both SOT and Stochastic algorithms have these artifacts. SOT is designed to render opaque silhouettes of transparent objects by simulating light passing through more material. Probabilistic sampling introduces noise, which is reduced using supersampling. Stochastic transparency uses MSAA as transparency, but requires several samples to estimate the correct fragment contribution. It can be parameterized to use more samples, which incurs in higher computational cost and memory consumption.

3.2.6.3 *Hypothetical Scene Analysis*

We estimated the performance of the algorithms in different rendering contexts aiming at providing a comparative analysis with respect to image quality and performance. We use three rendering contexts to stress different demands of quality and performance, which are based on the proposal of Foley and Feiner (FOLEY et al., 1990): offline rendering (which demands high image quality); interactive rendering (which enables user visualization); and real-time rendering (which enables user interaction with the rendering).

We created a hypothetical scene that can be parameterized to reflect the different rendering contexts described above. Table 3.5 describes the values used to parameterize the scene within each context. Offline rendering has the highest quality demand, thus we estimate a 4096x2160 window, with detailed meshes and high-quality parameterization. Interactive rendering has an average demand, thus, we estimate a 1920-1080 window, with balanced parameterization to allow real-time visualization. Real-time rendering has the highest FPS demand, thus the parameterization prioritizes performance, with reduced geometry and values for buffer size and number of samples.

Application	W(p)	H(p)	m	n	l	d	k	s
Offline	4096	2160	33	500M	200	20	1	1
Interactive	1920	1080	33	50M	200	20	d	d
Real-Time	1920	1080	33	5M	200	20	3	16

Table 3.5: Hypothetical scene parameterization

We estimated the number of operations required by each method using Equation 3.9:

$$op = geo \times (raster + frag_{sort}) + pre_{sort} + blend \tag{3.9}$$

where op is the estimated number of operations, geo is the number of geometry passes (which multiplies raster and fragment sorting), plus the costs for pre-sorting and blending.

We use the complexity analysis given in Table 3.4 as basis for estimating the number of operations required by each method using Equation 3.9. The plot in Figure 3.22 summarizes this section by presenting an approximate comparison of techniques.

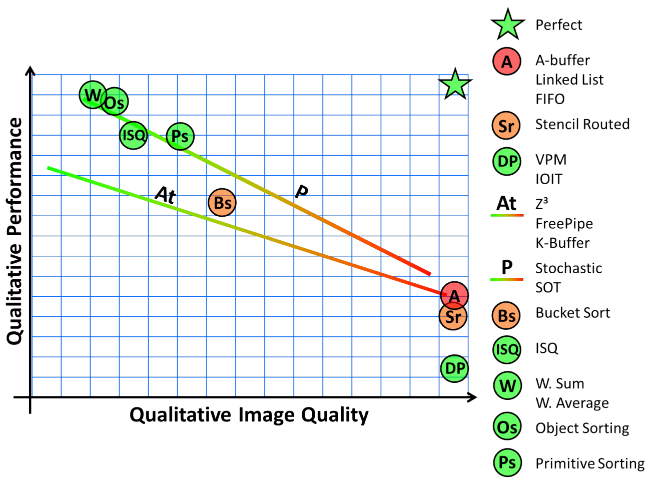


Figure 3.22: Comparative chart of techniques regarding image quality, performance, and memory consumption. The graph displays performance with respect to image quality, with colors indicating memory consumption (green = low, red = high).

3.3 Dynamic Fragment Buffer

Order-independent transparency (OIT) rendering is computationally intensive due to required sorting and sufficient memory to store fragments before sorting. This section presents Dynamic Fragment Buffer (DFB), a revamped two-pass OIT rendering technique, which performs correct blending of a large number of transparent layers at interactive frame rates. DFB self-adjusts memory allocation to handle a variable number of fragments per pixel without wasting memory.

The approach consists in a base+displacement scheme that requires a two-step computation. In the first step, only the number of layers required for each pixel is computed. Once the number of layers is known, in the second step, per-pixel layers can be evaluated and stored compactly in consecutive memory addresses. Implementing this idea efficiently on the GPU is particularly challenging, as we can observe in the implementation described in the DirectX SDK 11 (MICROSOFT, 2010), which crashes with an untreated overflow for scenes with an average of more than 8 layers per pixel.

The Dynamic Fragment Buffer (**DFB**) was designed to render images of scenes with multiple transparent layers by using a buffer-based approach for exact OIT. It provides compact storage of all fragments per pixel by using a base+displacement memory-addressing approach.

The DFB algorithm has four stages: (i) fragment counting, (ii) prefix sum, (iii) fragment shading and storing, and (iv) sorting and blending. Fragment counting performs a geometry pass to rasterize primitives (without shading), thus generating and counting the total number of fragments to be stored for each pixel. The prefix sum accumulates the per-pixel counters to compute a base address for each pixel. The base address of each pixel points to the fragment list associated to that pixel in a buffer shared by all pixels. A second geometry pass is performed to store shaded fragments at their appropriate locations in the shared buffer. The final step consists of sorting all fragments of each pixel list and blending them in depth-sorted order. Figure 3.23 shows the execution flow of DFB, along with the competing strategies B3D and PPLL, which have three stages in common: draw geometry, shade and store, and sort and blend. We detail the DFB modules below.

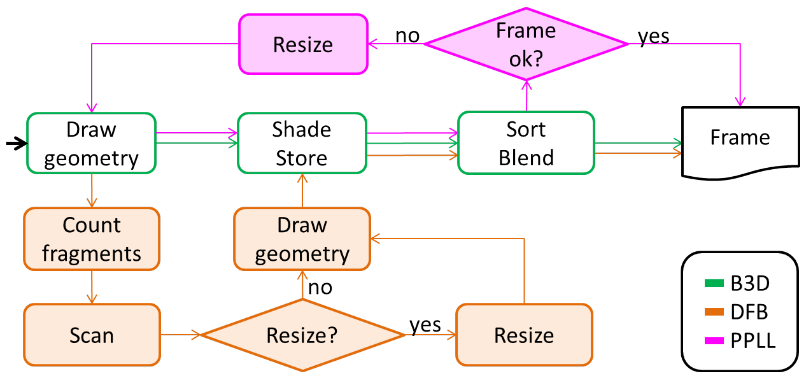


Figure 3.23: Execution flow: an overview of similarities and differences among Buffer 3D (B3D), Dynamic Fragment Buffer (DFB), and Paged per Pixel Linked Lists (PPLL).

B3D is a truncated A-buffer, with each pixel containing k slots to store fragments in incoming order. When overflow, fragments are lost. At post processing, the slots are depth-sorted before blending.

3.3.1 Fragment Counting

The first stage of the DFB algorithm uses a geometry pass to count the number of fragments per pixel by rendering the scene without shading. The algorithm prescribes that the fully opaque geometry is rendered in a previous pass. Transparent fragments are tested against the resulting depth buffer, but they do not modify it.

We use a shader to count fragments, with each fragment thread being responsible for updating a *countingBuffer* in the correspondent pixel entry. This is implemented using an atomic increment for avoiding read-after-write race conditions among concurrent threads. A 32-bit unsigned integer texture is used to represent the *countingBuffer*. The stencil buffer could also be used for this purpose, but its 8-bit representation would limit scenes to 256 layers. This limitation would prevent rendering models such as the Power Plant, which has viewpoints that generate frames with more than 500 layers (Section 3.3.6). Because all fragments are stored in incoming order, loss of important fragments may occur for any fixed-size array.

3.3.2 Prefix Sum

In the second stage, we prepare the indexing structures to store all fragments consecutively in the shared buffer. For each pixel, we define a base index that points to the address in the shared buffer where the associated fragment list starts. This is done using a prefix sum over the *countingBuffer*.

Prefix sum is implemented efficiently on the GPU using a scan operation. Starting from zero, the scan iteratively accumulates the counting of fragments from previous pixels, and results are stored in a *baseBuffer* texture. After the scan ends, the per-pixel base index, pointing to a reserved list into the shared buffer, is stored in the *baseBuffer*. The last base added to the last counter gives the total number of transparent fragments generated for the entire frame, which is needed to determine how much memory has to be allocated, and if the DFB buffer must be resized.

We use the optimized parallel scan implementation provided by the **Thrust 1.5.1** (NVIDIA, 2013) library, with some additional memory copies. Since Thrust is still being developed, the direct access of CUDA resources by Thrust operators is not available. However, we can copy a CUDA resource array, like the counting and base buffers, to a Thrust vector, where the scan operation can be performed. Once the scan finishes, data is copied back to the *baseBuffer*, to be used in subsequent stages.

3.3.3 Storing Shaded Fragments

The second geometry pass is responsible for shading fragments and storing them in the shared buffer (which we call the *Dynamic Fragment Buffer*). The per-pixel base index (*baseBuffer*), computed in the previous stage, is used to address the fragment list in the shared buffer. The base address is added to the fragment count to define the position where to store the fragment. Figure 3.24 illustrates the addressing scheme for an incoming fragment.

3.3.4 Sorting and Blending Fragments

The last stage of the algorithm sorts and blends fragments of each list in the shared buffer. The base indices are used to determine the start of each fragment list, and the counters indicate how many fragments need to be considered for each particular pixel.

A full-screen quadrilateral is used to launch a thread per pixel. Each thread sorts, in

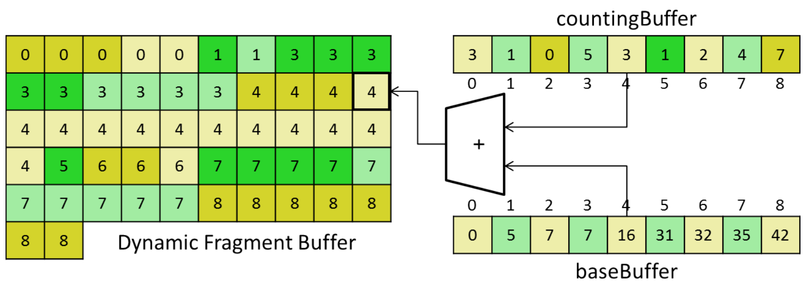


Figure 3.24: DFB storage: the address for storing a shaded fragment in the dynamic fragment buffer is computed by adding the base address and the counting index. This composed address stores the incoming fragment into a free position in the shared buffer. Yellow represents structures belonging to even pixels, while green represents the odd ones. The saturated colors indicates slots already filled with fragments. In the shared buffer, a number indicates to with pixel the entry belongs.

depth order, the stored list of shaded fragments using insertion sort. Next, in the same thread, the fragments are blended in front-to-back order. If a list is empty (no transparent layer for that pixel), no color is sent to the framebuffer.

3.3.5 Extended Dynamic Fragment Buffer

The original DFB algorithm (MAULE et al., 2012b) has a user-defined alpha value for all fragments of the entire scene. Thus, each fragment is stored in the buffer as four floating-point words of 32 bits each, capturing the RGB channels and the depth value. A single constant global alpha limits the number of scenes that can be correctly represented.

We extend the DFB fragment representation to include a 32-bit alpha per fragment–ie, $R_{32}G_{32}B_{32}A_{32}Z_{32}$. We also include another version that limits each color channel to 24 bits, where the total fragment storage for RGBA and depth is then four words of 32 bits–ie, $R_{24}G_{24}B_{24}A_{24}Z_{32}$. In addition, we include a version using only 8 bits per channel including alpha (**eDFB**, or $R_8G_8B_8A_8Z_{32}$), which reduces storage per shaded fragment to only two 32-bit words.

Memory consumption for $R_{32}G_{32}B_{32}A_{32}Z_{32}$ increases 20% with respect to the original DFB ($R_{32}G_{32}B_{32}Z_{32}$). The $R_{24}G_{24}B_{24}A_{24}Z_{32}$ version maintains the same memory consumption of RGB32b and, at the same time, adds the individual alpha per fragment. Memory consumption for eDFB is reduced by 50% with respect to the original DFB. All DFB channel formats have the advantage of correctly handling individual alphas per fragment, instead of the single global alpha value.

With different fragment sizes, performance and image quality are affected and require careful analysis. The impact of storing different fragment attributes with different sizes is presented in Section 3.3.6 and discussed in section 3.3.7.

3.3.6 Experiments

Experiments were performed on a computer with an Intel i7 980 processor and a GeForce 580 (1.5 GB working memory) running Windows 7. All techniques were implemented using GLSL 4.0, except for the DFB scan, which uses of Thrust 1.5.1 with CUDA 4.0. We guarantee correct image generation for all tests, using an offline tracker to capture the maximum number of transparent layers for each walkthrough. This number is used to set the B3D fixed-size array and GPU caches used for sorting. Only buffer-based exact-OIT algorithms are considered.

We made minimal modifications to the original DirectX Deep Frame Buffer proposal (**DxDFB** (MICROSOFT, 2010)), to load 3D models and get performance numbers. For all techniques, we used the same insertion sort algorithm. PPLL was tested in two versions: (i) four fragments per-page (**PPLL4**) and (ii) a single fragment per-page (**PLL**), emulating the original per-pixel linked-lists technique.

We measured performance with a virtual walkthrough around the scenes. The viewing changes allow the algorithms to be tested in situations with varying image coverage, visible geometry and number of transparent layers. We analyze different scenes, from simple geometric models to a complex CAD model.

Method \ Resolution	500^2	800^2	1100^2
B3D (CRASSIN, 2010b)	1250	625	345
DFB (MAULE et al., 2012b)	500	294	170
DxDFB (MICROSOFT, 2010)	11	4	N/A
PLL4 (CRASSIN, 2010a)	500	213	114

Table 3.6: Performance (in FPS) comparison for the decimated Bunny model (69K triangles, up to 14 layers). Small resolutions, in the first rows, were used to allow capturing the performance of DxDFB.

The DxDFB implementation uses a fixed-size buffer and never checks for overflows, which limits image resolution and number of transparent layers that can be handled. To avoid crashing DxDFB, we had to use a decimated version of the Stanford Bunny (with 69K triangles and up to 14 layers). Table 3.6 presents the results. DxDFB has the lowest FPS numbers among all methods and does not run at high resolutions. DFB is much faster than DxDFB and slightly faster than the two versions of PPLL being considered, but not as fast as B3D. Because the Bunny is a very simple scene, and B3D does not have the memory management overhead, B3D performance is the best. However, to guarantee full quality, B3D requires an offline counting of transparent layers (this is a pre-processing step, required once per scene, which determine the maximum depth-complexity for a walkthrough, so it is not presented in the performance evaluation). The following test cases will demonstrate the limitations of B3D.

Memory consumption of each technique depends on the attributes stored for each fragment, and can be calculated as follows:

- $B3D_{mem} \propto (width \times height) \times max\_layers$;
- $PPLL_{mem} \propto (width \times height \times percent) \times (\lceil \frac{max\_layers}{page\_size} \rceil \times page\_size)$;
- $DFB_{mem} \propto (width \times height \times percent) \times max\_layers$.

We used synthetic scenes composed of quadrilaterals for evaluating the techniques under controlled parameters, alleviating the geometry processing cost and focusing on the fragment-storage aspect. We vary the screen resolution, number of transparent layers (view-aligned quad instances), and the percentage of pixels covered on the screen. Figure 3.25 presents the performance results. Note that B3D, aside from the good performance, wastes memory in inverse proportion to screen coverage – e.g., when 25% of the screen is covered, 75% of the memory reserved by B3D is wasted. The performance advantage of DFB over the PLLs becomes evident with increasing percent of the image covered by transparent layers. That is not the case for 0% coverage, though, because DFB still has to perform

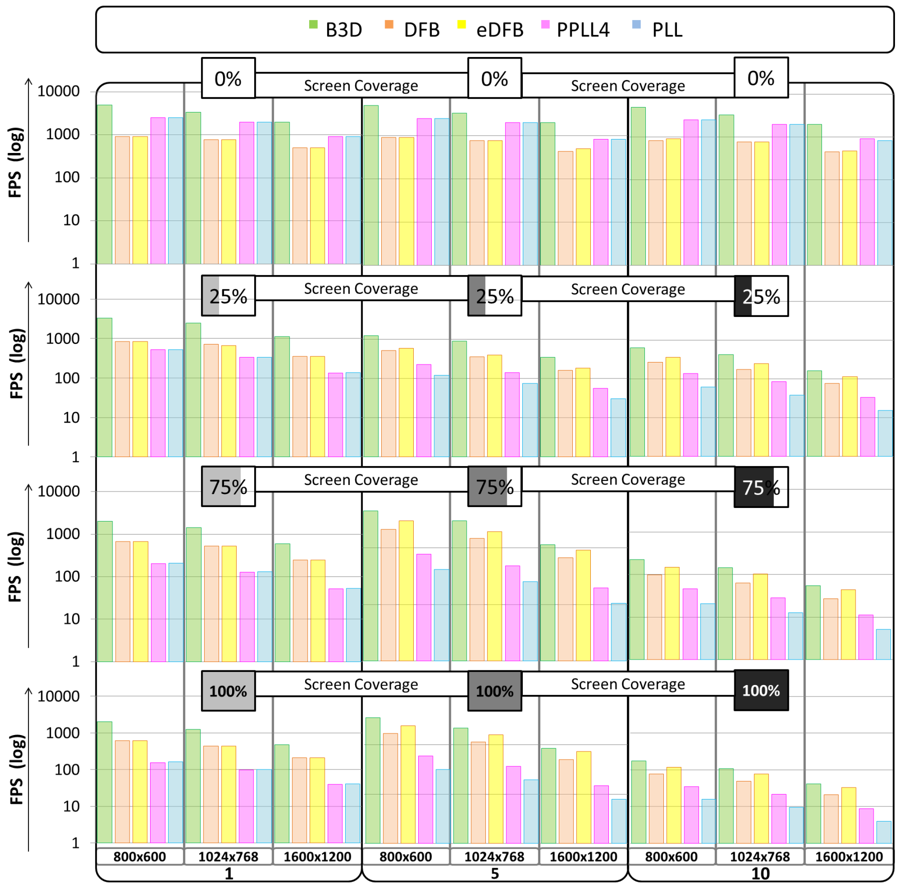


Figure 3.25: Log_{10}(FPS) for synthetic scenes. Scenes are composed of 1, 5 and 10 quad instances, covering different percentages of the screen, for three image resolutions. Illustrations of the coverage are presented above each graph and the number of quad instances (or depth complexity) is shown at the bottom of the figure.

two geometry passes, which is not required for the other techniques. Due to the reduced size of each fragment, eDFB uses less bandwidth and cache resources, thus, improving performance.

The same synthetic scenes are used to evaluate the individual costs of memory management, associated to the techniques with dynamic memory allocation. Figure 3.26 shows the time consumed by overflow checking and buffer resizing. It includes the *counting* and the *scan* stages, and the buffer resize when a future overflow is detected by the DFB. For the PLL and PPLL techniques, it includes overflow queries and the heuristic buffer resize (which simply doubles the size needed by the previous frame, not taking into account that some fragment-pages are not completely full). When the number of pixels colored by transparent fragments increases, the global counter of the paged-per-pixel linked list shared buffer becomes a major bottleneck because all threads must wait to increment it.

We evaluate performance and memory requirements during a walkthrough for scenes containing different numbers of instances of the Stanford Dragon model. The instances are disposed side by side, and the camera navigates elliptically around them. Figure 3.27 presents the results for scenes composed only by instances of the model, which contain heavy geometry and few transparent layers. Figure 3.28 presents the results for scenes composed by instances of the model in a grass field; where the grass, composed by sprites

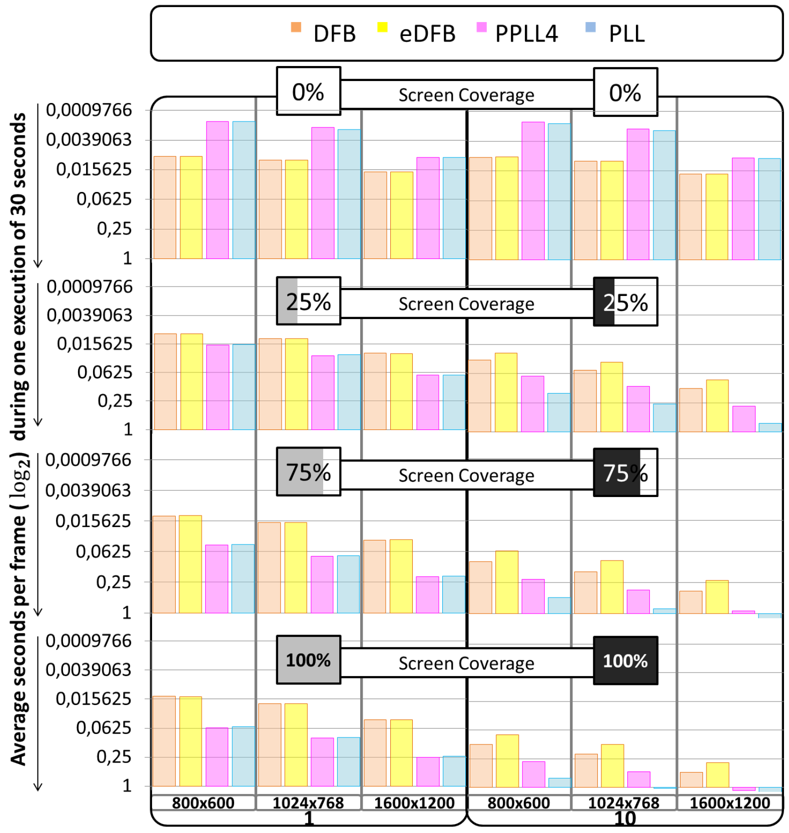


Figure 3.26: Memory management costs for synthetic scenes: costs are given in seconds for scenes with 1 and 10 quad instances, with varying resolutions and screen coverage. Only methods with dynamic memory allocation are considered.

with alpha texture, adds several transparent layers to the dragons scene.

B3D is the fastest algorithm, however, B3D cannot handle several instances nor big screen resolutions, due to lack of memory. That is the case for 10 instances with the highest resolution in Figure 3.27, and most of the tests for 5 and 10 instances in Figure 3.28. DFB presents the lowest memory consumption and the best performance among memory-efficient algorithms. The eDFB approach performs even better and halves the memory usage. In a scene with more geometry and less transparent layers, as presented in Figure 3.27, the two-pass approach of DFB may become a major bottleneck for performance as the number of geometric primitives increases, because of the two geometry steps required. However, when maintaining complex geometry and increasing transparent layers, the two-pass approach pays off in terms of performance and memory consumption, as presented in Figure 3.28.

We also performed experiments with a massive CAD model, the entire Power Plant, consisting of **12,748,510** triangles. The walkthrough varies drastically the visible amounts of geometry and transparent layers. Because this model has a large number of triangles and transparent layers (up to 514), memory requirements are high and the B3D approach is unable to run correctly due to lack of memory (it would required 2.4GB). All techniques with dynamic memory allocation can render this scene at barely-interactive rates, but only at the low resolution of 640x480.

For stress-testing the techniques, we evaluate them in a walkthrough around the Power Plant model. The time to process a frame is represented by the curves plotted on the front

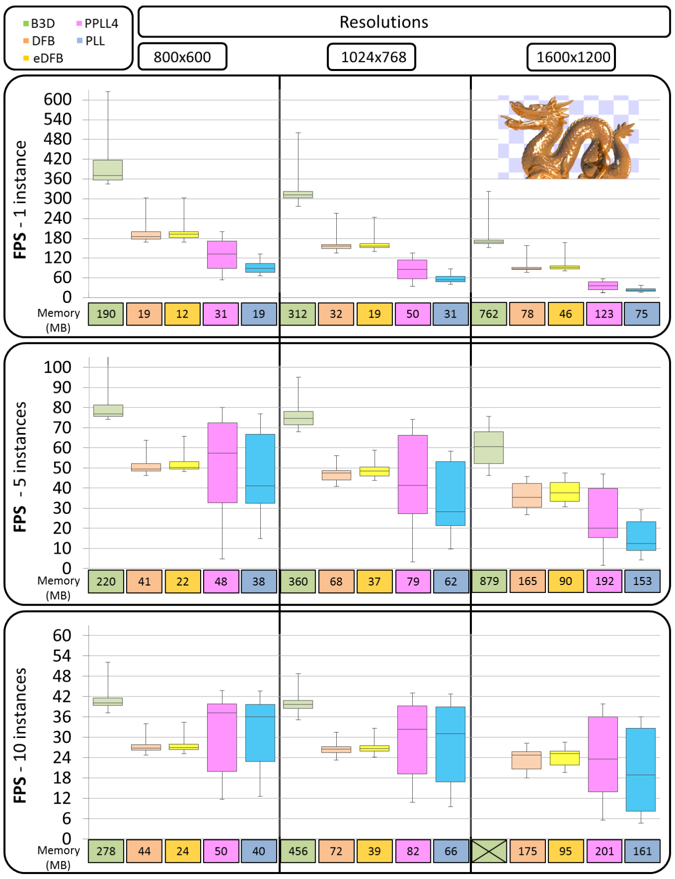


Figure 3.27: FPS and memory consumption for the Stanford Dragon model. Tests with scenes composed of 1, 5 and 10 instances of the model and three image resolutions. Each dragon model has 871,414 triangles and up to 26 transparent layers. Five instances can go up to 30 layers, and 10 instances up to 38 layers. Performance graphs have **different scales** to better present the differences among techniques when the number of instances changes.

plane of Figure 3.29. At the top, some screenshots represent the model coverage on the screen, for correlating with performance. In the background, a stacked chart presents the percent of pixels containing different number of layers.

The peaks in PPLL4 and PLL plots are caused by the hick-ups of memory reallocation (in the animated walkthrough, they represent frozen frames). We observed a correlation between peaks and the increase of depth complexity in the frames. DFB not only has a consistent superior performance, but also requires less memory; this is because overflows are easier to identify when estimating the total amount of memory required by a frame. The eDFB technique presents similar performance at half the memory usage.

A scene with two dragons in a grass field was used to demonstrate the image quality improvement of adding an alpha channel per fragment (eDFB), instead of the single global alpha value for the entire scene of the original DFB. The red and the yellow dragons have opacity of 0.15 and 0.3, respectively. The alpha texture of the grass has opacity values varying from 0 (fully transparent textels) to 1 (fully opaque textels), with transition values of opacity at the border of the grass leaf for proper transparency and anti-aliasing. Figure 3.30 shows the rendering of the scene using individual alpha (eDFB-left), using global alpha (DFB-middle) and the difference image (right).

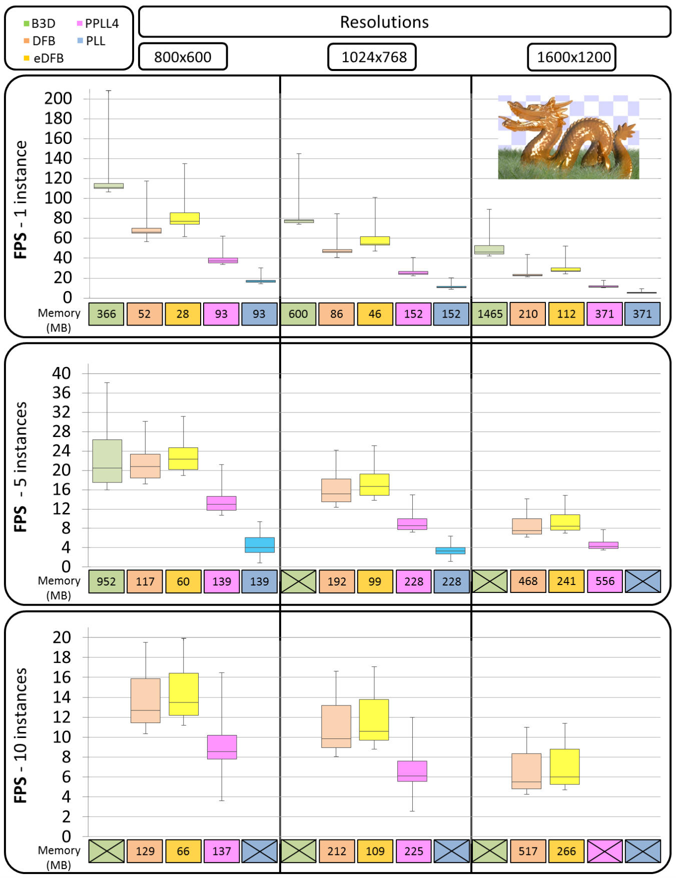


Figure 3.28: FPS and memory consumption for the Stanford Dragon model in a grass field. Tests with scenes composed of 1, 5 and 10 instances of the model and three image resolutions. The grass coverage is proportional to the space occupied by the instances. One single instance of the dragon with the grass sums up to 50 transparent layers. Five instances can go up to 130 layers, and 10 instances up to 220 layers. Performance graphs have **different scales** to better present the differences among techniques when the number of instances changes.

Using this same scene, we show performance and memory measurements for different fragment formats in Figure 3.31. Results show that, when fragment size is reduced, memory bandwidth demand is also reduced, which brings performance gains.

3.3.7 Discussion

In this section, we discuss the results in terms of geometry cost, impact of varying image resolutions, cost of memory allocation, and image quality.

3.3.7.1 Performance Impact of Geometry Processing

Scenes with a large number of triangles and few transparent layers configure the worst situation for the DFB technique due to its two geometry passes, and eDFB shares the same drawback. The impact of the two passes can be clearly seen in Figure 3.27 for the lowest resolution with 10 instances. However, when fragment processing costs increase, the benefits of a sequentially organized array of fragments per pixel pay for the extra geometry step, as shown in Figure 3.28. For the PPLL techniques, the amount of geometry influences the most when a frame must be sent again due to overflow. B3D, which uses a

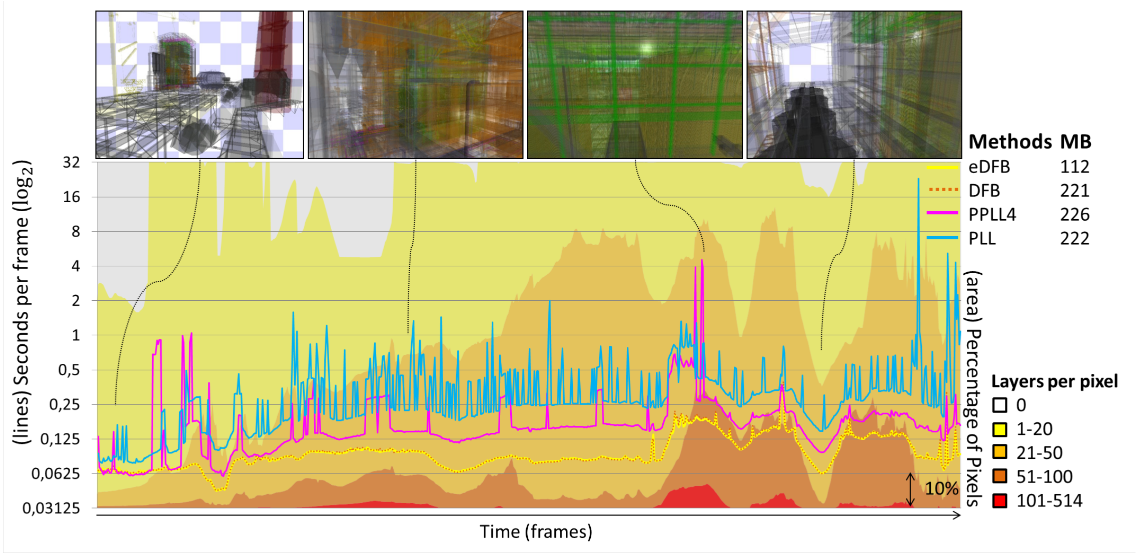


Figure 3.29: A tour around the Power Plant model: B3D is not able to handle the full Power plant with **12,748,510** triangles and up to **514** transparent layers, at 640x480 pixels. Screen shots show frames captured during the walkthrough. The graph lines show the seconds to generate each frame of the tour. Color-coded areas in the background show the percentage of pixels with different numbers of transparent layers; each horizontal strip indicates a range of 10%. Peaks in PLL and PPLL4 are caused by memory reallocation. DFB and eDFB present very similar performances, thus their lines overlap.

single geometry pass, is less affected than the others with respect to geometry costs.

3.3.7.2 Performance Impact of Image Resolution

The scan operation is the most expensive step of the DFB and eDFB, and its performance is directly dependent of the screen resolution. In our tests, the Thrust *exclusive_scan* showed to be data dependent, which means that its performance changed with the number of transparent layers in the frame.

Due to the fixed array allocation per pixel, the B3D approach requires more memory than the other techniques to keep all fragments of a given scene. This was especially important for high depth-complexity scenes, where B3D was unable to handle large resolutions without exceeding the physical memory limits.

PPLL decreases performance when image resolution increases, but the causes are different than DFB's and the performance loss is greater. To test for overflow, PPLL must wait until the GPU finishes processing the current frame, causing a pipeline flush. When more pixels are covered, the re-allocation heuristic causes the frame to be re-sent many times, always waiting to know whether there was overflow or not. Performance degradation due to such flushes can be seen in Figures 3.25, 3.26, 3.27 and 3.29. Figure 3.26 shows that the DFB and eDFB performances degrade smoothly as resolution increases, and their memory consumption is lower than the other approaches (Figure 3.27). This makes the eDFB the most suitable technique for applications that require rendering large image resolutions.

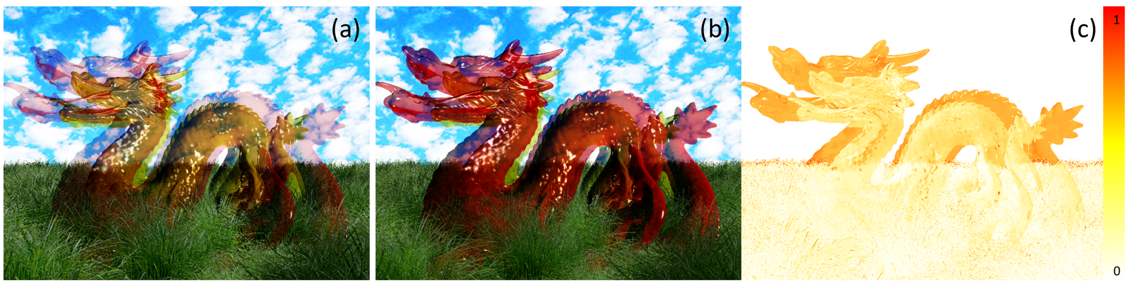


Figure 3.30: Transparent scene with different opacities: the opacity for the red dragon is 0.15, and for the yellow dragon is 0.3; the grass uses alpha texture with opacities varying from 0 to 1. (a) correct blending with individual alpha per fragment ($R_8G_8B_8A_8Z_{32}$); (b) image rendered with a global alpha for all fragments (mean alpha of the scene = 0.3 ($R_{32}G_{32}B_{32}Z_{32}$)); (c) error when rendering with global alpha (RGB distance mapped in color codes).

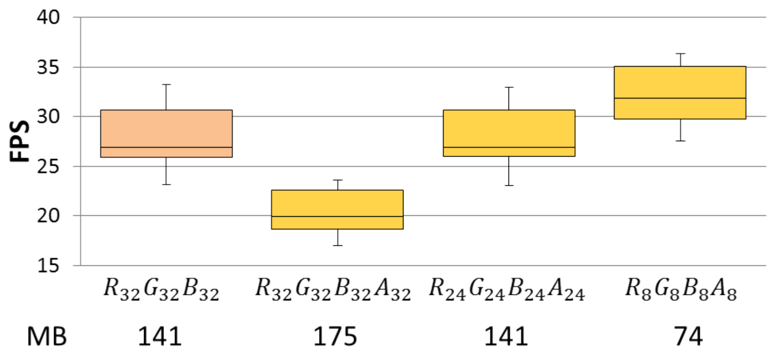


Figure 3.31: Performance and total memory consumption for a free walkthrough around two dragons in the grass scene (up to **84** layers): all versions keep a 32-bit word for depth storage. $R_{32}G_{32}B_{32}A_{32}$, using 5 words of 32b is penalized for the extra storage, while the other versions, using 4 words only, present similar performance. RGBA8b, using less bandwidth, presents better performance.

3.3.7.3 Impact of Memory Allocation

The DFB was named after its ability of adjusting itself at each frame to use the minimum memory possible, while keeping the necessary fragments to correctly evaluate OIT using less memory than previous approaches. Since it requires contiguous storage, an amount of memory must be pre-allocated to ensure contiguity.

PPLL and B3D techniques also need contiguous memory, but they cannot estimate the correct buffer size because they are unable to count the number of fragments that must be stored on each frame. Because of that, both techniques may waste memory, or drop fragments when allocated memory is exceeded (overflow).

PPLL can detect overflow cases, resize the buffer and rebuild the frame. It does not know how many fragments were lost and cannot precisely increase the buffer size, though. The buffer size is increased heuristically (for instance, its size is doubled upon overflow (CRASSIN, 2010a)) and the frame is re-generated, overflow is checked again, and the process repeats until the buffer size is large enough, or until running out of memory. The consequences of this iterative memory allocation are the elevated costs presented in Figure 3.26, and the peaks of performance loss in Figure 3.29. Because the DFB monitors closely the number of fragments per pixel, an overflow only occurs when the GPU memory is insufficient to handle the frame.

3.3.7.4 Performance Impact of Memory Addressing

Both DFB and B3D have sequential per-pixel arrays of fragments, which improves memory locality. However, B3D uses an array of fixed size, thus when arrays are processed, the look-ahead cache may copy trash from unused entries. The array organization of DFB avoids both wasting and copying trash to the cache.

The PLL main drawbacks are the bottlenecks generated due to the use of a single node counter for all pixels (leading to serialization), and the random memory accesses. Both problems are also impact PPLL, which reduces them by creating pages of consecutive fragments. However, it is still hard to define a good page size. If the page is too small, many random accesses are required to assemble the list; if the page is too big, memory is wasted and trash is copied to the cache.

B3D has better performance than DFB, because it has only one texture read to get the fragment list, whereas DFB needs two texture reads per fragment. PLL and PPLL need to follow several links of the list, using random memory accesses, to assemble the entire fragment list, which makes them slower than DFB and B3D.

3.3.7.5 Performance Impact of Fragment Size

Figure 3.31 presents performance numbers in terms of memory bandwidth, by evaluating DFB versions with different fragment sizes. The HDRI (high dynamic range imaging) eDFB version with 5 words of 32 bits per fragment ($R_{32}G_{32}B_{32}A_{32}Z_{32}$) runs significantly slower than the version with just 4 32-bit words per fragment ($R_{32}G_{32}B_{32}Z_{32}$). In terms of performance, fragment size influences bandwidth usage. In the other hand, the compact versions must perform extra bit-packing operations.

The version using 24 bits for each color channel improves image quality with individual alpha per fragment, while maintaining the original DFB performance and memory usage. This is a good option for HDR image generation.

For LDR (low dynamic range imaging) purposes, the version using only 8 bits per color channel ($R_8G_8B_8A_8Z_{32}$) has a significant memory footprint reduction with respect to the original DFB, while increasing performance because of the reduction in memory bandwidth.

3.3.7.6 Image Quality

For all techniques, image artifacts can happen if the memory available is not enough to handle all transparent fragments of the scene. By being tighter on memory allocation and handling, DFB is less susceptible to such artifacts.

The eDFB gains image quality with the addition of individual alpha channel per fragment, as presented in Figure 3.30. We concluded that $R_8G_8B_8A_8Z_{32}$ version is good enough for LDR displays because the image resulting from the difference to the high-quality $R_{32}G_{32}B_{32}A_{32}Z_{32}$ version does not present visually perceptible values. When HDR is required, Figure 3.31 shows that the best balance between performance and quality is to use the $R_{24}G_{24}B_{24}A_{24}Z_{32}$ version.

3.4 Hybrid Transparency

Hybrid transparency (HT (MAULE et al., 2013)) is an approach for real-time approximation of order-independent transparency. The hybrid approach combines an accurate compositing, of a few core transparent layers, with a quick approximation, for the remaining layers. Its main advantage, the ability to operate in bounded memory without noticeable artifacts, enables its usage with high scene complexity and image resolution, which other approaches fail to handle. Hybrid transparency is suitable for highly-parallel execution, can be implemented in current GPUs and further improved, with minimal architecture changes. We present quality, memory, and performance analysis and comparisons which demonstrate that hybrid transparency is able to generate high-quality images at competitive frames rates and with the lowest memory consumption among comparable OIT techniques.

The intrinsic goal of OIT proposals is to correctly represent a visibility function that describes the contribution of layers along depth. The visibility of a fragment is defined by the multiplication of two components: (i) the opacity of the fragment and (ii) the transmittance of the accumulation of fragments in front of it. The algebra described in (PORTER; DUFF, 1984), can be extended to compute the transmittance (τ) of a fragment i (from a depth-sorted list of fragments) using the opacities (α) in front of it:

$$\tau_i = \begin{cases} 1 & \text{if } i = 1 \\ \prod_{j=1}^{i-1}(1-\alpha_j) & \text{if } i > 1. \end{cases} \tag{3.10}$$

In order to balance image quality, memory consumption, and performance, we developed the hybrid transparency technique (HT). This OIT approach divides the transparent layers into subsets defined by the importance of their contribution to the pixel color, and applies different algorithms to compose these subsets. Subsets with major importance are composed by accurate algorithms, which are slow due to the required sorting, while less important subsets of transparent fragments can be combined using less accurate, but faster, algorithms.

Parts of a scene that are seen through multiple transparent layers get dimmed by their accumulated opacity. This suggests that fragments away from the viewer are not as important as the closer ones, and their contribution can be approximated with less impact to the pixel color. Therefore, HT splits the fragments of a pixel in two subsets: core fragments, which are closer to the viewer, and tail fragments, which are seen by the viewer through the core, thus, getting dimmed and less important to the pixel color. The two subsets are combined with the opaque background to produce the final pixel color. Figure 3.32 illustrates the monotonically-decreasing behavior of transmittance along depth, which leaded us to the selection of the core and tail subsets.

3.4.1 HT Core

The core is the subset of transparent fragments of a pixel that has main contribution to the pixel color, thus, deserving accurate blending. Leaded by the monotonically-decreasing behavior of the transmittance function along depth ((SALVI; MONTGOMERY; LEFOHN, 2011) and Fig. 3.32), we selected the front-most fragments to compose the core. Different high-quality algorithms can be used—e.g. depth peeling(EVERITT, 2001) or A-buffer(CARPENTER, 1984)—with minor modifications in order to select the core fragments.

To extract k core fragments of a pixel, we use a truncated A-buffer (k-TAB), which allows capturing and blending the k closest fragments using bounded memory. The exact

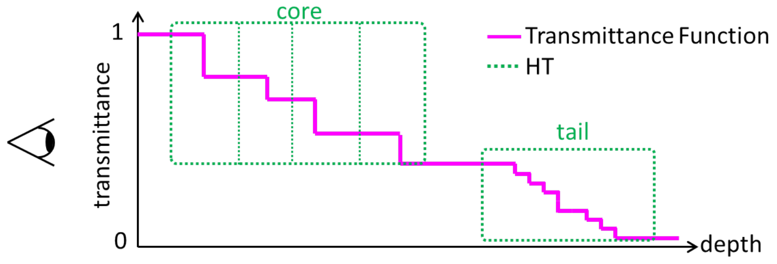


Figure 3.32: Transparent fragments are weighted by the transmittance at their respective depth. The monotonically-decreasing behavior of the transmittance function indicates that fragments closer to the viewer tend to have major visibility, thus, composing a subset of core fragments which must be combined using an accurate algorithm. The same property lead us to conclude that fragments away from the viewer tend to have less contribution (visibility) and can be approximated with less prejudice to the pixel color.

number of fragments to be handled by the core can be configured based on a memory budget, or based on information about the scene to be handled (image-quality requirement).

Using the **over** operator (from (PORTER; DUFF, 1984)), we can define the visibility (v_i) of a core fragment i, using the opacities (α) of the depth-sorted core fragments, as

$$v_i = \begin{cases} \alpha_1 & \text{if } i = 1 \\ \alpha_i \times (1 - \sum_{j=1}^{i-1} v_j) & \text{if } i > 1. \end{cases} \tag{3.11}$$

Each color (C) of the k core fragments is weighted by the visibility of the fragment, and the sum of these weighted colors compose a new layer, which we call core layer (C_{core}), described by Equation 3.12. The opacity of the core layer (α_{core}) is the accumulation of visibilities from all the k fragments (Equation 3.13), and it is used to dim the visibility of the subsequent subset of fragments (the tail).

$$C_{core} = \sum_{i=1}^{k} (C_i \times v_i) \tag{3.12}$$

$$\alpha_{core} = \sum_{i=1}^{k} v_i \tag{3.13}$$

3.4.2 HT Tail

The core will handle a fixed number of k fragments, which means that overflows are expected. If a pixel has n fragments to combine, all fragments not captured by the core ($n - k$) compose the tail subset. The tail algorithm must work on limited memory and still consider an unbounded number of fragments without overflowing. Two algorithms fit the requirements: weighted sum (WSum) (MESHKIN, 2007) and weighted averages (WAvg) (BAVOIL; MYERS, 2008). Both algorithms are fast, work in fixed memory and with only one geometry pass; because they do not perform sorting they are approximate.

Our choice was WAvg, because it provides a better color approximation (BAVOIL; MYERS, 2008). WAvg works in a single geometry pass by accumulating the colors of the fragments (C_i) weighted by their own opacities (α_i), and accumulating their opacities as well. Equation 3.14 describes the colors accumulation (C_{acc}), while Equation 3.15

describes the opacities accumulation (α_{acc}).

$$C_{acc} = \sum_{i=k+1}^{n} (C_i \times \alpha_i) \tag{3.14}$$

$$\alpha_{acc} = \sum_{i=k+1}^{n} \alpha_i \tag{3.15}$$

In a post-processing full-screen pass, the accumulated color is weighted by the accumulated opacity, producing the tail color (C_{tail}), as described by Equation 3.16. WAvg uses the counting of fragments to distribute the accumulated visibility equally among all the fragments of the pixel, composing the tail layer. This is done by dividing the accumulated opacity by the number of fragments ($t = n - k$), as in Equation 3.17.

$$C_{tail} = \frac{C_{acc}}{\alpha_{acc}} \tag{3.16}$$

$$\alpha_{avg} = \frac{\alpha_{acc}}{t} \tag{3.17}$$

Applying Equation 3.10, the transmittance of the composition (how much can be seen through the composition (τ_{tail})) is defined as

$$\tau_{tail} = (1 - \alpha_{avg})^t. \tag{3.18}$$

While the transmittance of the composition weights the next layer (background), the opacity (or absorptance: the amount of light blocked) of the composition (α_{tail}) weights the tail color. The opacity is the complement of the transmittance, so

$$\alpha_{tail} = 1 - \tau_{tail}. \tag{3.19}$$

3.4.3 Core, Tail, and Background Compositing

With the core and tail accumulated into two transparent layers, the **over** operator (PORTER; DUFF, 1984) can be used to combine them, along with the background or opaque layer. Each layer is dimmed by the layers in front of it. The tail contribution is limited by the transmittance remaining after the core compositing, and the background is weighted by the transmittance remaining after core and tail compositing.

The final color (C_{final}) accounts for the three contributions: (i) from the core (C_{core}), which is the first layer with full visibility ($v_{core} = 1$), (ii) from the tail (C_{tail}), which is dimmed by the core opacity and (iii) from the background (C_{bg}), weighted by the transmittance remaining after the compositing of the core and the tail (since $\alpha_{bg} = 1$). Equation 3.20 describe the final compositing.

$$C_{final} = C_{core} + (1 - \alpha_{core}) \times (C_{tail} + (1 - \alpha_{tail}) \times C_{bg}) \tag{3.20}$$

The HT core correctly estimates the visibility of its fragments. The HT tail correctly weights the fragments by their own opacities, but approximates their transmittance. Thus, HT approximates the transmittance function of a pixel as depicted in Figure 3.33.

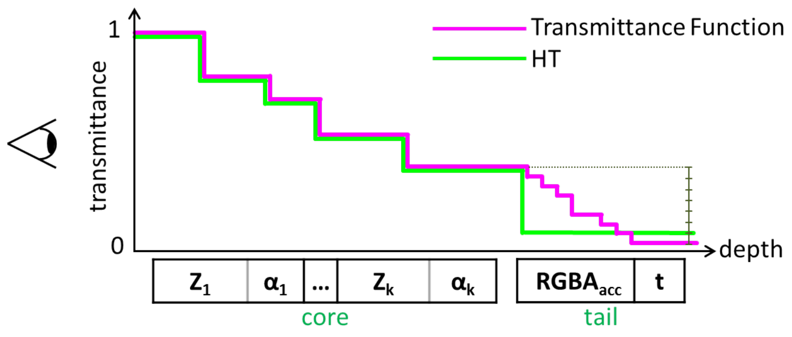


Figure 3.33: HT approximation for the transmittance function: the core correctly evaluates the first part of the transmittance function. The remaining transmittance is equally distributed among the remaining terms.

3.4.4 Implementation Details

Hybrid transparency can be implemented on current GPUs using two geometry passes. The first pass extracts the k closest fragments and is followed by a full-screen pass that computes their visibilities. The second geometry pass does the shading and accumulates the fragments into the appropriate color buffer (core or tail). A final full-screen pass composites the core, tail, and background buffers.

The first rendering pass is used only by the core, which collects the k front-most layers into a 3D texture that implements the k-TAB. Since the k-TAB is always kept sorted, the insertion of a fragment only requires to place itself in order. This is done by a single-pass bubble sort that swaps the fragments into their proper place using atomic operations. Today's GPUs do not provide atomic operations for more than one 32-bit word. Yet, 32 bits are enough to store a 24-bit depth and an 8-bit opacity (alpha) per fragment. After the first geometry pass, we draw a full-screen quadrilateral to create the corresponding per-pixel visibility function, which allows to correctly blend the k front-most fragments during the second geometry pass. The computation of the visibility function is done by traversing the list of k core fragments for each pixel and computing the visibility per depth, which is stored into another k-deep 3D texture (in order to acquire better numerical precision).

During the second geometry pass, fragments are shaded and forwarded to the core or tail processing, being composited in the respective color buffers. If the depth of a fragment is present in the k-TAB (created in the first rendering pass), the fragment is handled by the core; otherwise, it is handled by the tail. The last step combines the two transparent color buffers with the opaque one (or background). Algorithm 1 summarizes these steps—note that all steps are performed on the GPU.

3.4.5 Experiments and Discussion

We designed several tests to evaluate our proposal and check its viability for real-time OIT rendering, as well as created situations to stress-test our algorithm quality. HT is compared against the state-of-the-art OIT technique for visibility compression, the adaptive transparency (AT), which promises to work in fixed memory when critical sections become available on the GPU, and is able to produce images with barely noticeable artifacts (very reduced noise and flickering) when compared to other known techniques—e.g. Z^3 and k-buffer. A reference image generated by a GPU implementation of the A-buffer is used to evaluate quantitative image quality. We also evaluate the core and tail components of HT

Algorithm 1 Hybrid Transparency

```
1. Store: $< z, a >$
        1.1 Render Geometry
        1.2 Atomically keep the $k$ front-most fragments (core) in depth-sorted order into
    the k-TAB
2. Visibility Evaluation (core)
        2.1 Traverse the k-TAB, computing visibility at each depth and storing it into a 3D
    visibility buffer
3. Render: shade and compositing
        3.1 Render Geometry
        3.2 Composite the core fragments into color buffer $C_0$
        3.3 Accumulate overflowed fragments (tail) into color buffer $C_1$
4. Final Composite
        4.1 Composite color buffers and opaque background
```

independently, to demonstrate the advantages of the proposed hybrid combination.

Since one of the main advantages of our proposal is the ability to run in fixed memory, we evaluate buffer-overflow situations and their impact on image quality. Performance measurements are presented to demonstrate the practical application of HT for OIT rendering, along with a theoretical complexity analysis demonstrating its scalability with highly-parallel execution.

3.4.5.1 Memory Consumption Analysis

The memory consumption of HT depends on: (i) the screen dimensions (width W, and height H) and (ii) the core size (k). So, the total memory used can be described by:

$$TotalMemory = W \times H \times (2k + 2). \tag{3.21}$$

Each layer of the k-TAB stores a tuple $< z, \alpha >$, packed into a single word (4 bytes), and another word (4 bytes) for the accumulated visibility of each fragment. Two more words (8 bytes) per pixel are used to count and accumulate the color of all overflowed fragments.

The memory used by HT is comparable to the required for depth peeling and WAvg, which depend mostly on the screen resolution, and little on the scene depth complexity—these are the most memory-economic techniques available. Unlike such techniques, HT has good performance and image quality simultaneously.

Memory consumption of HT is also comparable to the theoretical proposal of AT (which is supposed to work in bounded memory using critical sections). The actual AT implementation uses linked lists to hold an unbounded number of fragments (i.e. can run out of memory). Thus, the memory usage reported for AT in section 3.4.5.3 is the size actually used to store all fragments of the scene.

3.4.5.2 Image Quality Analysis

The images generated by HT are compared against a reference image (generated with a GPU implementation of the A-buffer) and images generated by AT and by HT components (k-TAB and WAvg). We used two distinct 3D scenes for practical image comparison: (i) the hair scene, same used in the original AT paper, and (ii) multiple instances of the Stanford dragon aligned. We also include a per-pixel error analysis of the color distance from the results of the methods to the reference image, with increasing memory (k).

Figure 3.34 shows the image quality evaluation for the hair model under different memory budgets. HT presents significant quality improvement when compared to the AT approach, using only 4 slots per pixel. We can also observe that, individually, the methods that compose the core (k-TAB) and the tail (WAvg) of HT are not able to produce the same image quality. With just 4 slots, the image generated with HT is barely distinguishable from the reference.

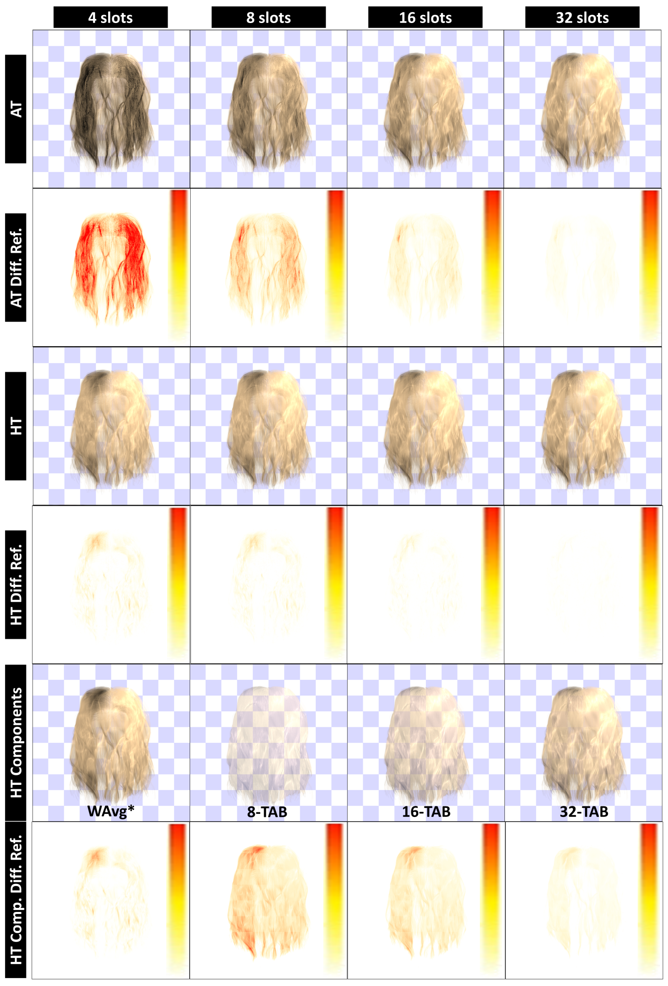


Figure 3.34: Hair model with 15,000 strands and up to 663 transparent layers: columns with increasing number of slots used to approximate OIT (per pixel). Images generated by AT are presented in the first row. The second row displays the normalized distance to the reference image in color-coded mode. Third and fourth rows displays the same results for HT. The fifth row shows images generated by the HT components, and their corresponding error in the sixth row. WAvg does not permit parameterization, being unable to be improved with more memory. The TAB is presented with 8, 16 and 32 slots storing only the front-most fragments, discarding the remaining ones. (dark.hair from Cem Yuksel)

The dragon scene is composed of 22 instances of the dragon model, with challenging opacity configurations. The aligned instances of the model have the following opacities: 0.05, 0.1, 0.05, 0.1, 0.2, 0.9, 0.2, 0.1, 0.05, 0.1, 0.05 and its repetition. This situation generates front-most fragments that are less visible, due to their low opacities (and these are stored in the HT core), leaving fragments with higher contributions to be combined by the HT tail (less precise).

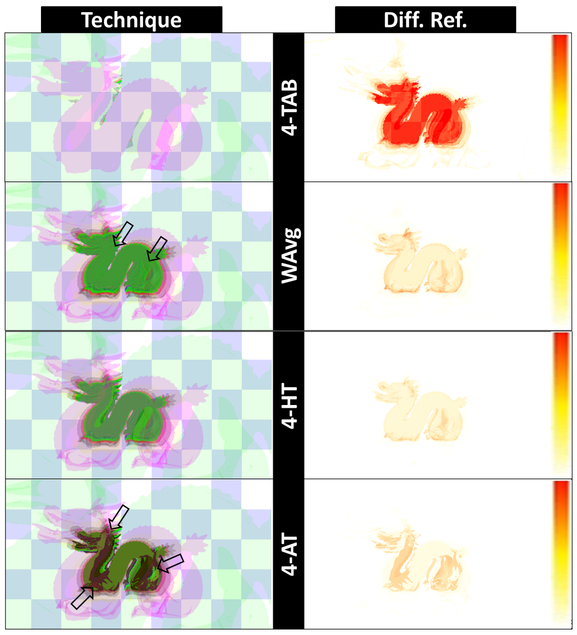


Figure 3.35: 22 aligned dragons: instances closer to the viewer have low opacities (HT core stores less important fragments), while instances farther away have high opacities (HT tail is compositing the most important terms). The first column presents images generated by each technique, followed by the color-coded error images in the second column. Even though this situation is challenging for HT, its error is smaller than the errors of other approaches, and HT visual quality is undoubtedly better.

These opacities configuration is a difficult scenario for HT. However, as results in Figure 3.35 show, no artifact is visually noticeable, especially when compared to the disturbing artifacts generated by the other techniques. 4-TAB shows severe errors, because it is unable to store enough significant layers. WAvg errors are less noticeable, but the incorrect distribution of visibility caused loss of details. The AT results show an interesting ”shadow effect caused by the heuristic used to approximate transmittance on overflow.

Both AT and HT approximate the visibility function encoded by incoming fragments using fixed-size buffers. AT discards fragments with smaller contribution to the function by always underestimating transmittance approximations. HT assumes that the front-most fragments are the most important, so their transmittance are correctly evaluated, and the tail fragments equally share the remaining transmittance. Figure 3.36 compares the approximation of the transmittance function for HT and AT.

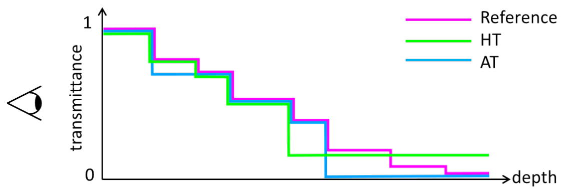
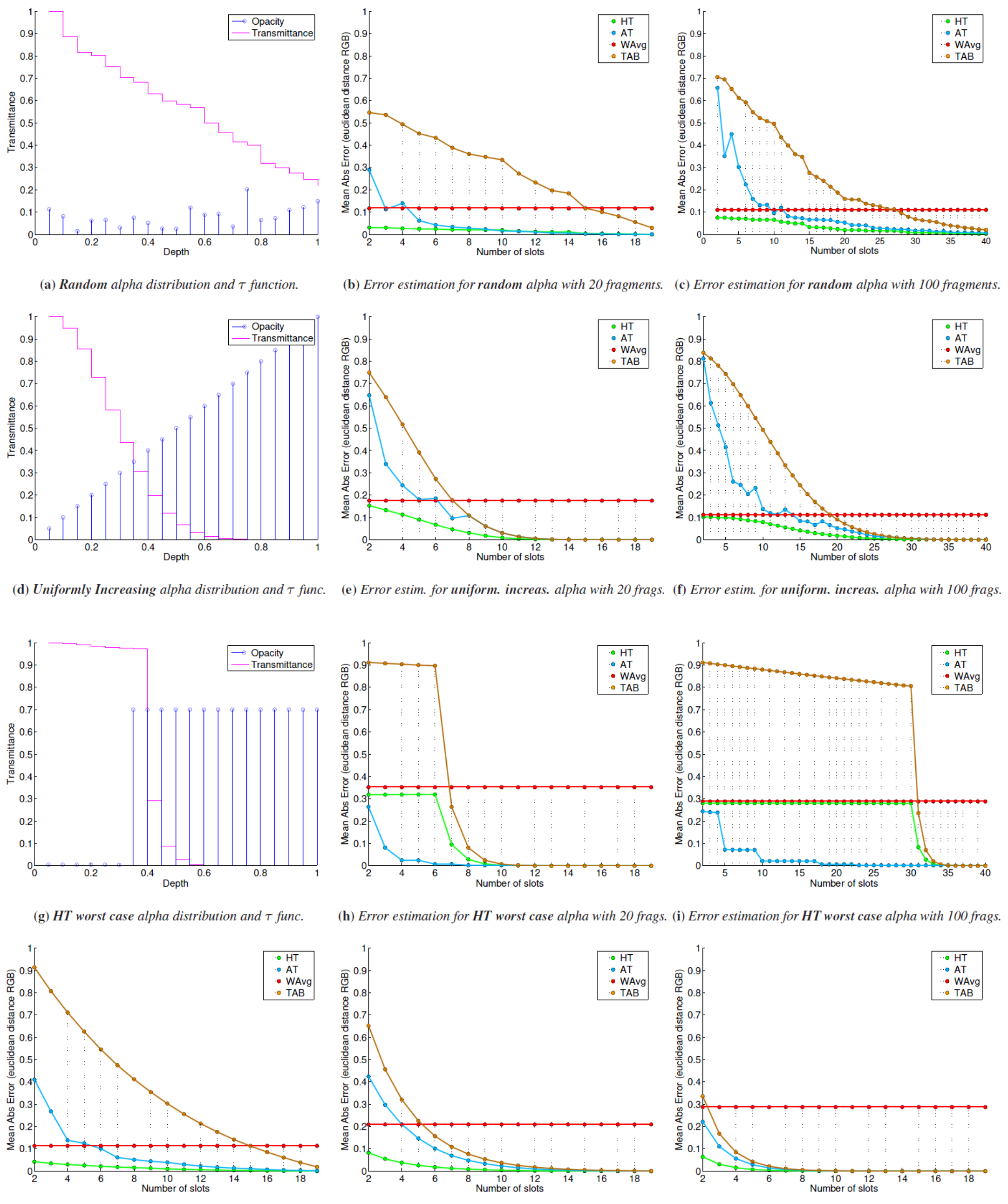


Figure 3.36: HT and AT approximations of the transmittance function using a fixed number of slots. AT underestimates the transmittance for compressed nodes of the function, while HT correctly represents the first part of the function, and distributes equally the remaining transmittance.



(a) ***Random** alpha distribution and τ function.* (b) *Error estimation for **random** alpha with 20 fragments.* (c) *Error estimation for **random** alpha with 100 fragments.*

(d) ***Uniformly Increasing** alpha distribution and τ func.* (e) *Error estim. for **uniform. increas.** alpha with 20 frags.* (f) *Error estim. for **uniform. increas.** alpha with 100 frags.*

(g) ***HT worst case** alpha distribution and τ func.* (h) *Error estimation for **HT worst case** alpha with 20 frags.* (i) *Error estimation for **HT worst case** alpha with 100 frags.*

(j) *Error estim. for **constant** alpha=0.1 with 20 frags.* (k) *Error estim. for **constant** alpha=0.3 with 20 frags.* (l) *Error estim. for **constant** alpha=0.5 with 20 frags.*

Figure 3.37: **Error estimation** for different alpha distributions with increasing number of slots (k). We simulated the composition of fragments by the algorithms 100 times, each time with random colors, and plotted average errors. Figures (a), (d), (g) show the alpha behavior and how it changes transmittance along depth. We present evaluations for 20 fragments (Figures (b), (e), (h)), and for 100 fragments (Figures (c), (f), (i)). We also present the error behavior for constant alphas with 20 fragments (Figures (j), (k), (l)).

We present a second set of experiments with simulations of different alpha behaviors, measuring the color distance to the reference. The goal is to systematically explore scenes that can better assess the image quality generated by HT and other approaches. Results are reported in Figure 3.37 for four different distributions of alpha (opacity): random, uniformly increasing, HT worst case and constant; each producing different behaviors of the transmittance function (τ).

First, we create a visibility distribution with randomly chosen alpha values, which might appear in scenes that require mixed rendering effects (e.g. hair and smoke and foliage). Results of the random distribution show that, even for small memory budgets, HT presents the smallest errors.

The second distribution has alphas uniformly increasing along depth. It is interesting to note how TAB and HT behaviors resemble the transmittance function, but the hybrid aspect of HT greatly reduces the errors, while AT presents unpredictable behavior.

The third distribution is the worst scenario for HT, where no important layer is captured by the core. This alone would not cause significant errors, but combining it with the subsequent important layers (which is the WAvg worst case) can compromise image quality. This is the single configuration in our experiments where we were able to find that HT generates higher errors than AT.

Finally, we simulated constant alphas behaviors, usually present in scenes with particles or objects of the same material. With the exception of WAvg (which does not improve upon memory increase), all techniques display an exponential error decrease with an increasing number of slots.

In summary, the experiments reported in this section show that HT has several advantages over other methods with respect to image quality. For instance, HT does not present visible flickering (caused by different approximations in consecutive frames), neither loss of details (caused by incorrect evaluation of visibilities). HT is stable and does not produce noise artifacts. In fact, HT does not generate visually-noticeable artifacts.

3.4.5.3 Performance Analysis

HT was developed to sustain a high level of parallelism. The expected execution time is dominated by the first rendering pass, in which each thread performs up to k atomic swaps to keep an average of d fragments in depth-sorted order. The k swaps are performed in parallel, so, as in a pipeline of k stages and d instructions, the complexity order is $O(k + d)$.

The AT theoretical proposal uses critical sections to select the best samples to use in the transmittance function. However, critical sections are non-parallelizable blocks of instructions. The execution time of theoretical AT also would be dominated by the first rendering pass, with cost equivalent to the serialization of d blocks of instruction, with i instructions each, over k entries, for a total cost of $O(d \times i \times k)$.

Our practical experiments were performed using a GeForce GTX 580 GPU and an Intel Core i7 CPU running Windows 7 (x64). Since the actual implementation of AT needs unbounded memory support to run in current GPUs (originally the per-pixel linked lists, PLL), we included in our experiments another version of AT using a more economic unbounded algorithm (Dynamic Fragment Buffer (DFB) (MAULE et al., 2012b)), which enabled us to compare AT with HT for a massive model (the UNC Power Plant).

For fair comparisons with AT, results for our proposal (HT) were given using three implementation variants: (i) with unbounded-memory support from PLL (HTP), (ii) with unbounded-memory support from DFB (HTD), and (iii) the original bounded-memory approach (HT). The same support variations were used for AT, called ATP, when using the original PLL support, and ATD, when using the DFB support. The numbers following the techniques abbreviations denote the number of slots used to approximate the visibility function (k).

Figure 3.38 shows the performance results, along with the memory consumption, for rendering the hair model. The FPS of each method was measured for several frames during a walkthrough around the model. Dotted lines represent the emulated versions with unbounded-memory support, among which HTD4 presents outstanding performance. The fixed-memory HT4 not only has the lowest memory consumption (at least 8x smaller), but also has performance comparable to ATP16 and HTP4 running in unbounded memory.

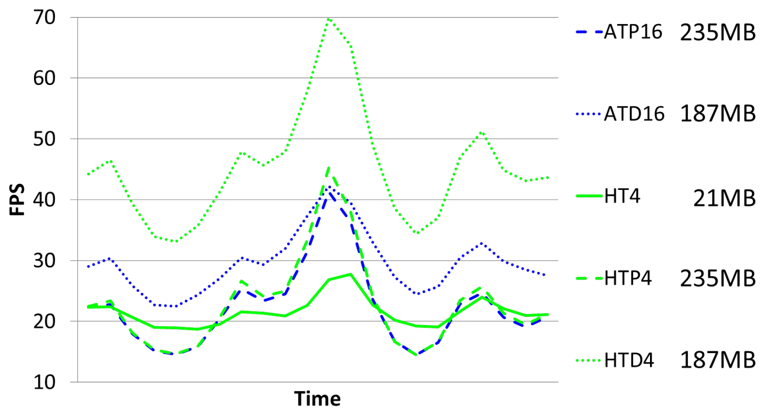


Figure 3.38: Hair walkthrough FPS: Dotted lines denote techniques executed with unbounded memory support. HT4 (full line) runs in bounded memory. (15,000 strands, up to 671 transparent layers)

A walkthrough around the scene with aligned instances of the Stanford dragon is used to test the techniques for different resolutions and number of instances. Figure 3.39 shows the performance results, and memory consumption, for increasing resolutions and two sets of instances. We display the FPS as box plots to capture the performance variation along the camera path. Versions using PLL, with only one geometry step, are faster than HT and the unbounded versions using DFB, both requiring two geometry steps. Each dragon instance has 871,414 triangles and up to 6 transparent layers, which leads us to the conclusion that the performance for this scene is dominated by geometry processing.

A future single-pass HT (which needs 64-bit atomic updates available in GPU) is expected to present great performance improvement. In the other hand, a practical bounded-memory version of AT requires serialization of code chunks (through critical sections) to correctly compute the transmittance function update. For a massively-parallel architecture, as the GPUs, serializations mean great performance loss.

In our last experiment, we measured the seconds taken to generate each frame (SPF) along a walkthrough around the entire UNC Power Plant (12,748,510 triangles, up to 588 transparent layers). This model combines the two previous tests: several transparent layers and heavy geometry. Performance results and memory consumption are presented in Figure 3.40. The versions requiring unbounded memory (dotted lines) were able to run only for the lowest resolution, running out-of-memory for the bigger ones (PLL versions were unable to handle this model). HT is able to render bigger resolutions efficiently, because it has fixed and small memory footprint; its performance is comparable to the unbounded AT, while the unbounded HT is up 2 times faster.

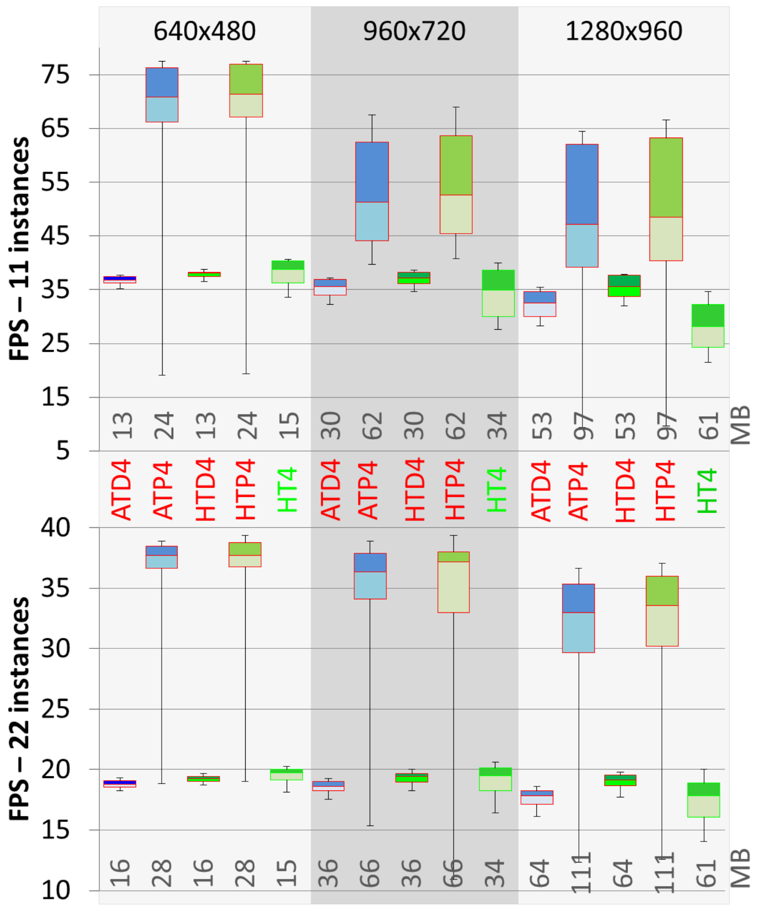


Figure 3.39: Dragons varying resolution and number of instances. The red color denotes techniques executed with unbounded-memory support. HT4, in green, runs in bounded memory. (up to 132 transparent layers and 19,171,108 triangles)

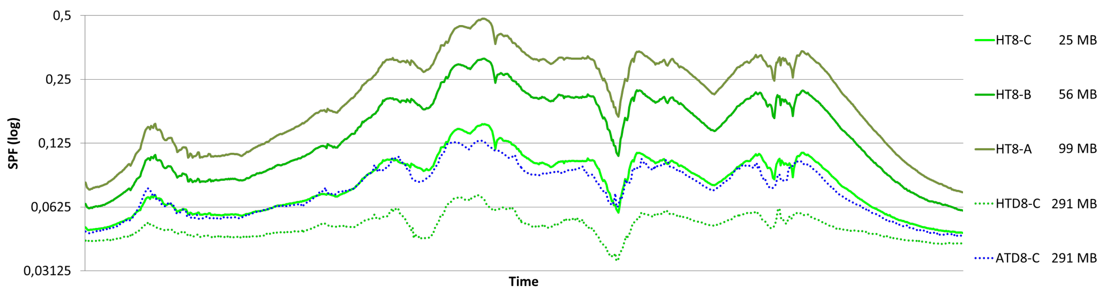


Figure 3.40: Power Plant walkthrough SPF: Dotted lines denote techniques executed with unbounded-memory support. HT8's (full lines) run in fixed-size memory. The accompanying letters A, B, and C indicate the screen resolution: A is for 1280x960, B is 960x720 and C is for 640x480. (the Power Plant scene has up to 588 transparent layers and 12,748,510 triangles)

3.4.6 HT Limitation

For some specific situations, HT fails to capture a sufficient number of significant fragments in the core, therefore compromising the visibility reduction property of transparency (or how much front-most fragments occlude those behind). When the set of fragments captured by the core change, a popping artifact may occur. This happens when the fragments in the core have low visibility, and a fragment with high opacity switches from the tail to the core, and vice versa. The perceived effect is a new surface suddenly appearing opaque, hiding those behind, or suddenly disappearing, showing those behind.

The main flaw in the HT approach is its assumption that front-most fragments are **always** the most important. Figure 3.41 shows a situation where front-most fragments have low opacity, thus, low contribution to the final pixel color. At the same time, fragments with high opacity are approximated by the HT tail (e.g. the green dragon). When the camera steps forward, a low-opacity fragment leaves the core (it went behind the camera), giving space to a high-opacity fragment to be correctly composited, and a popping artifact occurs.

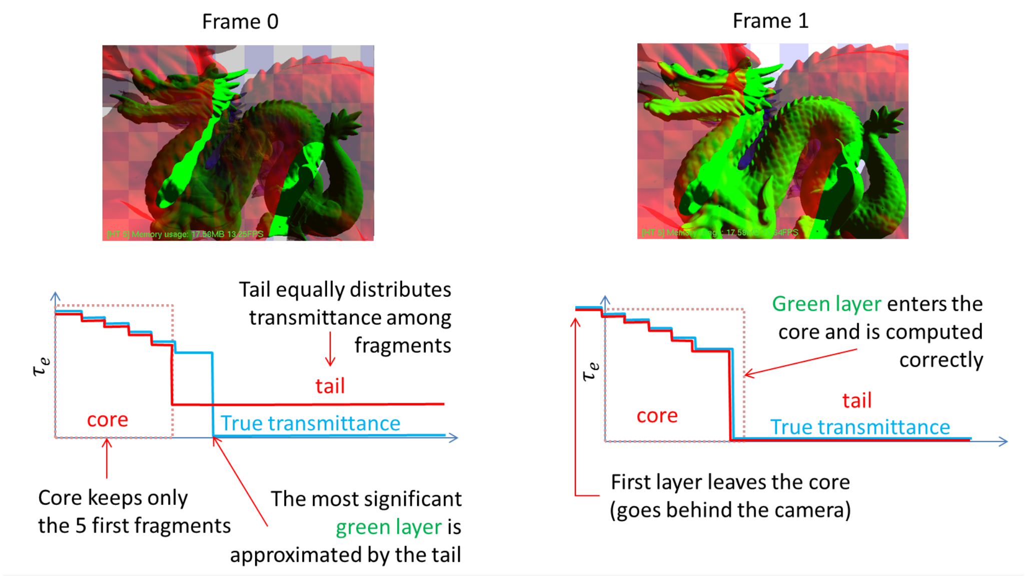

Frame 0
Frame 1
Tail equally distributes transmittance among fragments
tail
core
τ_e
True transmittance
Core keeps only the 5 first fragments
The most significant green layer is approximated by the tail
Green layer enters the core and is computed correctly
tail
core
τ_e
True transmittance
First layer leaves the core (goes behind the camera)

Figure 3.41: HT temporal artifact: when the core cannot store enough opacity, important fragments may be approximated by the tail; causing visibility artifacts (Frame 0). When, in a subsequent frame, an important fragment enters the core, a popping occurs (Frame 0 to Frame 1).

Despite our efforts, no solution to this problem was found yet. Unsuccessful attempts include combinations of core-tail segments, and heuristics to estimate the most important fragments using only local opacity and depth. Both approaches failed to estimate the visibility of important fragments after a tail segment. The same sort of artifact is also presented by a subsequent technique, MLAB, presented in Section 3.2.3.0.6.

3.5 Classification of Transparency Techniques by Features

In this section, existing techniques are classified with respect to important properties related to quality, memory, and performance. Sets are created for grouping techniques by such properties. The techniques proposed during the development of this thesis are also classified, highlighting the contributions made to the OIT rendering field. Some of the non-filled interceptions of these sets indicate opportunities for improvement.

The following section will present and discuss the classification. Acronyms presented in Table 3.7 are used to refer techniques.

Acronym	Technique
AB	A-Buffer(CARPENTER, 1984)
AT	Adaptive Transparency (SALVI; MONTGOMERY; LEFOHN, 2011)
BS	Bucket Sort(LIU et al., 2009)
DDP	Duap Depth Peeling(BAVOIL; MYERS, 2008)
DFB	Dynamic Fragment Buffer(MAULE et al., 2012b)
DP	Depth Peeling(EVERITT, 2001)
FIFO	F-Buffer and R-Buffer(MARK; PROUDFOOT, 2001)(WITTENBRINK, 2001)
FP	FreePipe(LIU et al., 2010)
HT	Hybrid Transparency(MAULE et al., 2013)
KB	k-Buffer(BAVOIL et al., 2007)
PLL	Per-Pixel Linked List(YANG et al., 2010)
SRAB	Stencil-Routed A-Buffer(MYERS; BAVOIL, 2007)
ST	Stochastic Transparency(ENDERTON et al., 2010)
WSum	Weighted Sum(MESHKIN, 2007)
WAvg	Weighted Average(BAVOIL; MYERS, 2008)
Z^3	Z^3(JOUPPI; CHANG, 1999)

Table 3.7: Acronyms for transparency rendering techniques.

3.5.1 Image Quality

Transparency rendering techniques present different levels of quality. This section discusses desired properties of transparency techniques for generating images with quality. Will be considered as correct images only those generated by the proper application of the Porter and Duff blending equation (PORTER; DUFF, 1984). Approximations will be classified with respect to the following properties:

- Order-independent transparency: determinism, or, the same input always produce the same output;
- Visibility reduction: visibility reduction of one layer over those behind it;
- Time coherence: consistency between frames.

3.5.1.1 Correct Blending

Within the context of transparency rendering, correct image is defined as the adequate application of the depth-order-dependent blending equation defined by Porter and Duff (PORTER; DUFF, 1984). Figure 3.42 presents techniques divided into two categories; those able to generate correct results for any scene are inside the green ellipse. Techniques outside the green ellipse may provide correct results, but only for specific scenarios, not for the general case. Those techniques are classified as approximations.

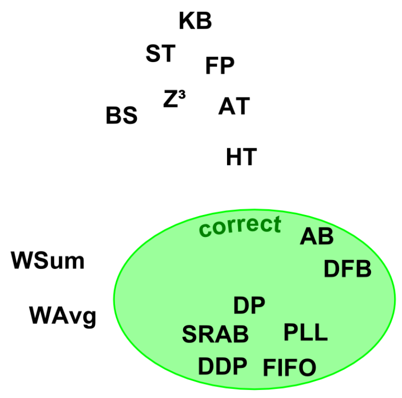


Figure 3.42: Techniques classified by correctness of results: techniques inside the green ellipse guarantee correct blending for any input case. Techniques outside the green ellipse are approximations. (correct: correct blending results for the Porter and Duff blending equation (PORTER; DUFF, 1984))

DP, DDP, and SRAB generate correct results by performing geometry passes, until all transparent layers are composited. AB, FIFO, PLL, and DFB store all incoming fragments for proper sorting and blending, generating correct results.

Approximate techniques cannot guarantee correct results for any scene. FP also stores incoming fragments, but it may lose some fragments due to lack of memory. Z^3, KB, AT and BS store a fixed number of fragments, compositing out-of-order on overflow cases. ST approximates visibility by stochastically filling fragment samples. WSum and WAvg use approximations of the blending equation; they may generate correct results only on few cases. HT is a combination of KB and WAvg, both approximations.

3.5.1.2 Order-Independent Transparency

The exactly correct result is not always required. Order-independence is one of the features desired in the approximations.

Order-independence is an important feature because it gives consistency to the results, eliminating sources of noise (flickering, salt-and-pepper, etc). Order-independence implies determinism: if the same list of fragments is processed twice, no matter the order in which these fragments are processed, the color result will be the same. Figure 3.43 shows techniques divided into two classes: those which are order-independent, inside the red ellipse, and order-dependent techniques outside the ellipse.

By definition, all techniques capable of generate correct results are order-independent. The blending equations of WSum and WAvg approximate the Porter and Duff blending equation by eliminating the depth-order-dependency, thus, the processing order of fragments does not change the result. HT is order independent because it uses KB with on-the-fly sorting to correctly evaluate the k front-most fragments and, then, sends the remaining fragments to be processed by WAvg.

The remaining techniques have their reasons for presenting order dependency. KB and Z^3, on overflow, merge the nearest fragments in incoming order. The same occurs in BS when more than two fragments are routed to the same bucket. FP drops the last incoming fragments when overflow. The incoming order of fragments changes the stochastic sampling of ST. AT builds a compressed visibility function by removing the less significant node on overflow; since this removal decision is made with partial information, the less significant node is relative to those already processed, thus, order-dependent.

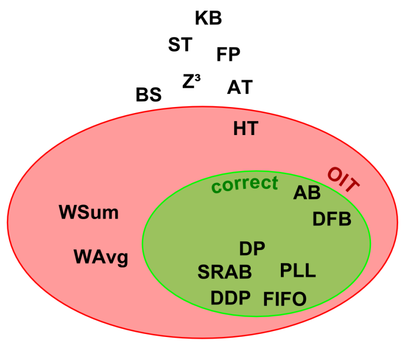


Figure 3.43: Techniques classified by order-dependency: techniques inside the red ellipse generate the same color result, no matter the incoming order of fragments. Techniques outside the red ellipse may generate different results for the same input, depending on the order in which the fragments are processed. (OIT: Order-independent Transparency)

3.5.1.3 Visibility Reduction

The depth-order dependency of transparency rendering comes from the reduction of visibility one layer imposes over those behind it. A fragment visibility depends on the transmittance on the path from the fragment to the eye, and this transmittance is determined by in-front fragments. Figure 3.44 shows, inside the light-blue ellipse, techniques able to capture or approximate transmittance-to-eye reduction. Techniques outside the light-blue ellipse cannot properly capture transmittance-to-eye reduction.

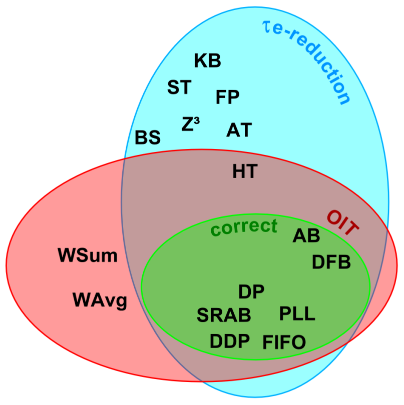


Figure 3.44: Techniques classified by the capacity of capturing or approximating the transmittance-to-eye visibility reduction of fragments. Inside the light-blue ellipse, techniques able to account for transmittance-to-eye reduction. Techniques outside the light-blue ellipse cannot properly account for transmittance-to-eye reduction. (τ_e-reduction: transmittance to the eye reduction by in-front fragments)

By definition, correct techniques evaluate the transmittance-to-eye reduction correctly. KB, BS, AT, FP, Z3, ISQ perform some kind of sorting to be able to approximate the transmittance-to-eye reduction one layer imposes over those behind it. HT core, by sorting its entries, also is able to capture this property, but the tail doesn't.

WSum and WAvg do not perform sorting; they attribute the same transmittance-to-eye to all fragments.

3.5.1.4 Time Coherence

Another relevant property is time coherence, or consistency between frames, in other words, the technique is free of flickering and popping artifacts. Figure 3.45 shows techniques classified with respect to this property: inside the orange ellipse, techniques with time coherence. Techniques outside the orange ellipse may not present flickering or popping for most inputs, but they are not temporal-artifact-free for all cases.

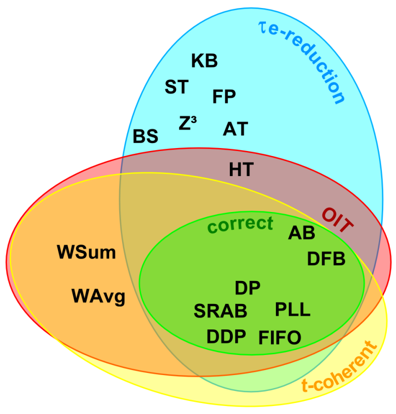


Figure 3.45: Techniques classified by time coherence: techniques inside the orange ellipse are guaranteed to be temporal-artifact-free. Techniques outside the orange ellipse may present flickering and popping artifacts. (t-coherent: time coherent, or consistency between frames)

Correct techniques are, by definition, time-coherent. WSum and WAvg order-independent blending equations are stable among frames.

HT depends on the amount of opacity it can store in the core. If, in one frame, the core stores only low-opacity fragments, important fragments may be approximated by the tail. When, in the next frame, one of these low-opacity fragments leaves the core, enabling a high-opacity fragment to be correctly evaluated, a popping occurs. By definition, techniques which results vary with the incoming order of fragments are not time-coherent.

3.5.2 Memory Consumption

Besides image quality features, are considered two other important properties for transparency techniques: memory footprint and performance. In the raster pipeline, primitives are rasterized, generating fragments which attributes are used to change pixel attributes, then the fragment is discarded to make space for new fragments. This cycle is efficiently performed in a predetermined amount of memory. A straightforward approach to handling transparency rendering is to store all fragments, thus they can be properly sorted and blended. However, this approach leads to unbounded memory requirements. It is hard to make project decisions for an application if one does not know beforehand how much memory will be required. Unpredictable memory usage also does not scale well with screen resolution and scene complexity (in terms of number of transparent layers), often causing the application to crash due to lack of physical memory (memory overflow). Therefore, the capacity of operating in a predefined bounded memory is highly desirable for a transparency technique. Figure 3.46 classifies the techniques with respect to their memory footprint. Inside the dark blue ellipse, techniques are able to run in bounded memory, so, outside the dark blue ellipse there are techniques which require unbounded memory.

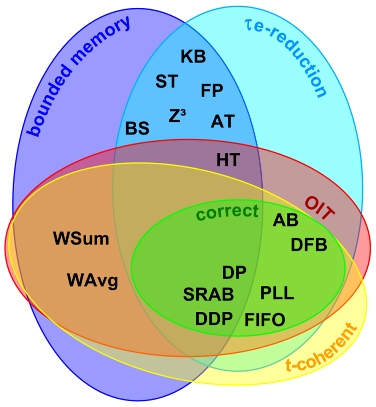


Figure 3.46: Techniques classified by memory consumption: techniques able to operate in bounded memory are inside the dark blue ellipse, and techniques that require unbounded memory are outside the dark blue ellipse. (bounded memory: pre-determined fixed-size amount of memory)

DP, DDP, and SRAB can produce the correct result in bounded memory by relying in multiple geometry passes to capture transparency layers. KB, ST, FP, BS, Z3, AT and HT approximate blending results by storing only parts of the information generated by all fragments in each pixel. WSum and WAvg approaches accumulate fragments attributes into accumulation buffers for post processing.

AB, FIFO, PLL, and DFB store all fragments in unbounded lists and buffers.

3.5.3 Performance

The optimal theoretical performance for a transparency rendering technique is linear in the number of fragments per pixel, because each fragment must be, at least, evaluated. Due to the depth order required by the blending equation, it is very difficult to conceive an approach that actually runs in linear time; most techniques perform sorting operations, elevating the complexity. Figure 3.47 shows the best performances among transparency techniques. The best performance is achieved by techniques inside the white ellipse, which only accumulate the contribution of each fragment a single time.

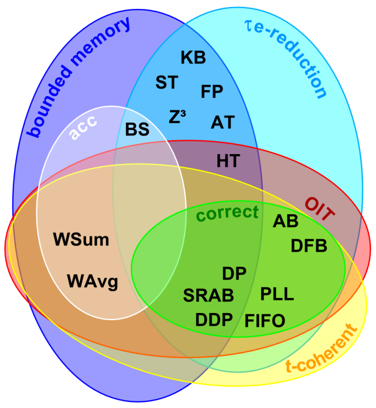


Figure 3.47: Techniques classified by best performance: techniques inside the white ellipse present the best performance because they only evaluate each fragment once. (acc: accumulation of a single contribution per fragment)

WSum and WAvg accumulate fragments contributions, so each fragment is processed only once. BS routes fragments into buckets using their depths as indexes, thus performing sorting in linear time.

DP, DDP, and SRAB need as much rendering passes as the number of transparent layers in the frame, re-evaluating all fragments at each pass. AB, FIFO, PLL and DFB store and sort all fragments in $O(n \log n)$. Z3, KB, AT, FP and HT perform on-the-fly sorting, thus fragments are tested more than once. ST evaluates each fragment three times, once per geometry step, for approximating their contribution.

The worst performance a transparency technique may present is unpredictable performance. Figure 3.48 presents techniques with unpredictable performance inside the pink ellipse.

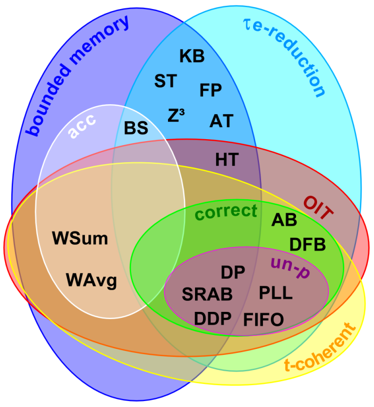


Figure 3.48: Techniques classified by worst performance: techniques inside the pink ellipse present the worst performance because it cannot be predicted. (un-p: unpredictable performance)

DP, DDP, and SRAB need as much rendering passes as the number of transparent layers in the frame, which is unpredictable, thus, presenting highly variable performance. FIFO and PLL need to store all fragments, so they may have to face overflow situations that require more memory allocation, causing the frame to be rebuilt. Since they don't know exactly how much memory must be requested, a frame may be rebuilt several times.

AB and DFB, despite using unbounded memory, have predictable performance. AB, by applying true linked-lists, may request more memory without the need to rebuild the frame. DFB, by counting the number of fragments generated for each pixel, is able to allocate the exact amount of memory required by the frame. Bounded-memory techniques performances are inversely proportional to the buffer size allocated for each pixel.

3.6 Summary

Transparency is a physical property of materials, which is desirable to be simulated on computers. This property variates how much light objects will transmit, allowing back-most fragments to be seen through front-most fragments. The compositing of a pixel color from transparent fragments is what classifies transparency as a multi-fragment effect.

Literature offers a diversified set of approaches for handling the depth sorting required to render transparency. Methods operate over different levels of the scene. They may sort the geometry or the fragments. That can be done by accumulating fragments through many geometry steps, or by storage management. If exact rendering is not required, approximations can take place. Approximate techniques can improve resources consumption through partial sorting, probabilistic sampling of opacity or alternative blending equations. As part of the development of this research, we studied, classified and compared transparency techniques. The resulting publication offers the reader an overview of the field and pathways for further research.

In this thesis, we proposed two techniques capable of managing visibility for order-independent transparency blending. The first one produces exact blending at the cost of unbounded memory. However, it is the most memory-efficient among similar techniques. It also achieves better performance for massively transparent scenes. The second technique is a hybrid approximation. By leveraging the understanding that front-most fragments are more likely to have greater visibility, we were able to significantly reduce memory usage.

The variety of transparency techniques can be scattered along three priority axes: memory management, performance, and image quality. None of the techniques proposed so far is capable of achieving satisfactory results for all three axis simultaneously, demanding application-targeted tradeoffs. However, techniques proposed in this thesis were able to move the state-of-the-art a bit forward, towards a better outcome for a larger diversity of applications.

4 ANTI-ALIASED RENDERING

This chapter presents the concept and consequences of aliasing, along with techniques proposed to solve them. We begin with the characterization of aliasing as a sampling problem that may occur either during capture or when reproducing the image. In the real world, this may happen with any acquisition device, impacting photographs quality (Section 4.1.1). In computer-generated images, there are some situations that tend to arise the aliasing problem, generating different artifacts (Section 4.1.2). For these, we expose a series of techniques present in the literature in Section 4.2.

We propose a novel technique in Section 4.3. Its goal is to reduce the number of samples required to improve quality of transparent scenes. Through that, memory footprint and bandwidth, and performance can also be improved.

4.1 Aliasing Problem

Aliasing is an artifact that occurs when a signal is not properly sampled or reconstructed. Any phenomenon that conveys measurable information can be considered a signal. We would like to capture signals for processing, transmission, and display. For that, the signal must be sampled by a digital sensor. The number of samples must be twice the signal's frequency for proper reconstruction (Nyquist rate). When this number is unfeasible, or the frequency is unknown, diversified samples distributions are used for approximating a reconstruction. When the display device does not have the capabilities required to express the signal, it generates reconstruction aliasing.

Within the context of this thesis, we are interested in image-related aliasing. A colored image can be interpreted as a function of four dimensions: two spatial dimensions (x, y), a color dimension $(wavelength)$ and an intensity dimension $(amplitude)$. The sensor and display mechanisms used to capture and present the image shall be adequate to sample and reconstruct the four components, respectively.

4.1.1 Aliasing Physics

Let's consider the case of a simple 2D signal, like a sound wave being captured by a single sensor after being emitted by one single source (COLLEGE, 2015). The wave presents an amplitude (first dimension) that changes over time (second dimension). For capturing this signal, we can use a single sensor that quantifies the amplitude once at each given time interval. The rate at which the sensor measures the signal (sampling frequency) dictates what will be the maximum achievable quality for the reconstruction. In theory, a signal can be unequivocally reconstructed if sampled at a rate at least twice its frequency (Nyquist criteria). Otherwise, more than one function will be able to fit the samples, causing aliasing. Both adequate and inadequate sampling are represented in Figure 4.1. For this sound example, the aliasing would cause you to hear a sound different from the original.

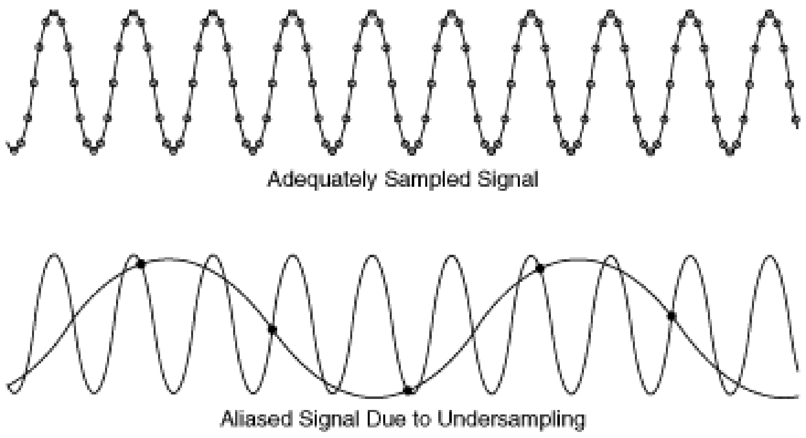


Figure 4.1: Sampling of signals represented as 2D functions. For both cases, the samples are equally-spaced. In the first image, a greater number of samples is used, capturing all the features of the original function. In the second image, too few samples are used, leading to the reconstruction of a different function (image from (SAMPLING, 2012)).

A similar process is used for image acquisition (digital photography), but involving more dimensions. For capturing a scenario of the real world into an image, modern cameras use arrays of sensors. Especial sensors are used for different colors, stimulated by different wavelengths. Sensors of different colors are uniformly distributed in a 2D matrix format, and are interpreted as sets of color channels to define a pixel in the digital image, as illustrated in Figure 4.2. These sensors are allowed to receive light stimuli for a time period (exposure time), during which they accumulate photons that are translated into electrical signals. The intensity of these signals defines the value of each color channel in the pixel.

Differently from the 2D sound example, where samples were displaced in time, for image acquisition samples are displaced in space. Therefore, the most common approach is to have the 2D array of physical sensors. Quality of the image acquired largely depends on the number of sensors in the device. If this number is not enough to capture subtle color variations presented in the scene, the reconstructed image will be aliased (see right image on Figure 4.2). Even if the number of sensors is adequate, the digital image may be aliased by the displaying device, if it does not have enough pixel units to represent all sensors sets.

Figure 4.3 shows another example of image aliasing. In this case, real photographs depict the same brick wall. The first of them shows an adequately sampled and reconstructed image, where high frequencies are distinguishable. The second photograph shows a poorly pixelized image with a wave-like aliasing artifact. This artifact is known as Moiré patterns and appears when high frequencies interfere with each other.

Aliasing is not an artifact exclusive to electronics. Any capture device is susceptible to aliasing, including the human eye. Figure 4.4 shows examples of patterns interference that can be seen by the naked eye.

4.1.2 Aliasing in Computer-Generated Images

In computer graphics we have a virtual scene, represented by geometric primitives. And we would like to display this scene in a discrete video monitor, as a 2D array of pixels. For that, we need to decide the pixels colors based on the scene attributes projected

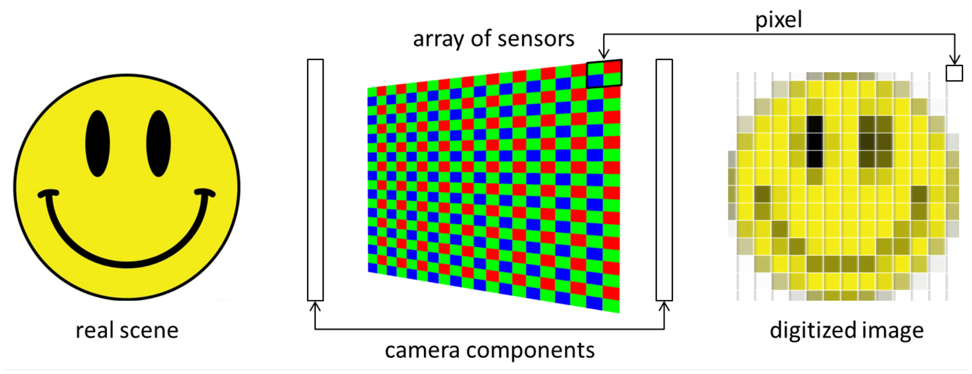


Figure 4.2: Image acquisition: an array of sensors for different wavelengths capture light intensity from a real scene. Color sensors are grouped to form image pixels. The digitized image also illustrates aliasing artifacts because the number of pixels is not enough to represent the high frequencies of the real scene.

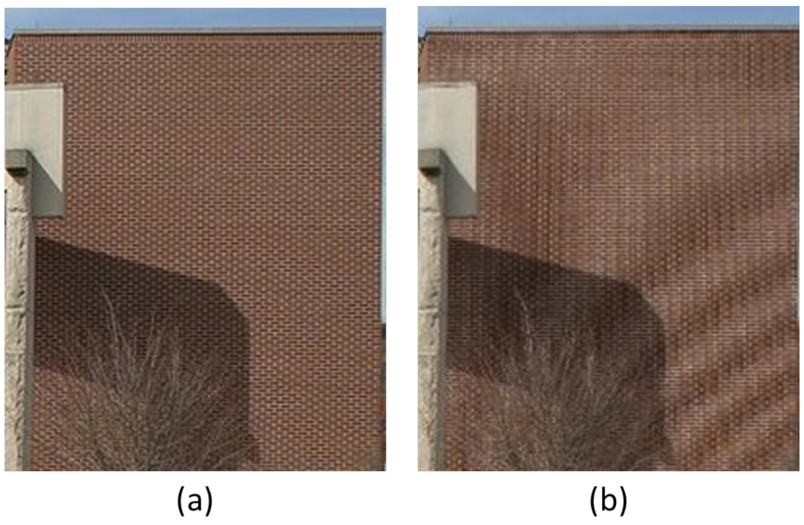


Figure 4.3: Example of aliasing in image acquisition: image (a) is a photograph taken with a proper sampling rate; while image (b) presents the result for a subsampling of the same scene. The aliasing artifact in (b) is known as Moiré pattern (image from (MOIRE, 2014)).

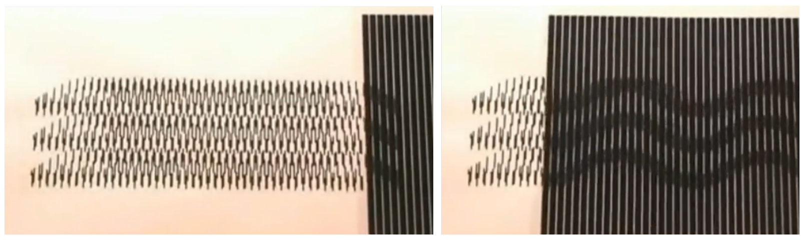


Figure 4.4: Moiré animation: patterns interference that can be seen by the naked eye. (image from (MOIRE ANIMATIONS, 2014), see the video).

onto them (see Section 2.1.1 for details), which is achieved through the sampling of the scene. Each pixel corresponds to an infinitesimal point sample over a surface in the virtual scene. This point, usually at the pixel center, determines the color and if the pixel will be colored. If a portion of the scene intersects a pixel, but does not cover its sample, it will not be captured. This causes aliasing artifacts like jagged edges, discontinuity of small objects and artifacts in regions with complex details (CROW, 1977). These artifacts can be amortized through anti-aliasing techniques, as exemplified in Figure 4.5.

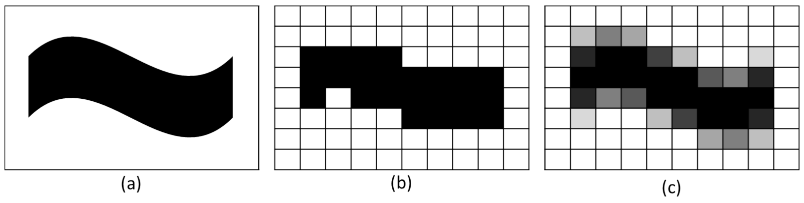


Figure 4.5: The aliasing problem in rendering: (a) curved shape to be drawn. (b) aliased result from sampling the pixels centers. (c) anti-aliased solution: each pixel receives an amount of color corresponding to the percentage of its area covered by the shape.

Jagged edge is an artifact that appears at the silhouettes of objects. I happens when small contributions from the surface boundaries fail to be captured by the pixels sampling. As result, edges present abrupt transitions that resemble staircases, because they follow the boundary of pixels in the matrix. The resulting image is very unpleasant and unrealistic. The solution is to generate a transition region, smoothing the edges and given the feeling of continuity, as shown in Figure 4.6. Section 4.2 describe techniques to achieve these smooth transitions.

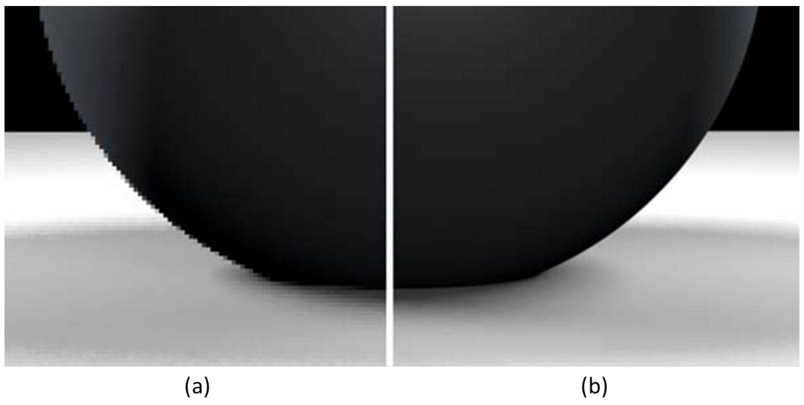


Figure 4.6: Aliasing example: jagged edges. (a) sphere rendered without AA presents jagged edges. (b) AA applied to the sphere rendering displays smooth transitions: a more pleasing and believable image.

When geometric primitives are projected onto a small or thin region of the screen, they may present discontinuity artifacts. This is because portions of small and thin primitives may fail to cover the pixel sample and, therefore, are not captured. The rendering result reconstructs a continuous surface in the scene as discontinuous segments of pixels, as shown in Figure 4.7-left. Despite being unpleasant, this artifact also may reduce the viewer's understanding of what is being presented on the image (compare antennas in Figure 4.7). In animations, the prejudice is worsened because movement (of the scene or camera) will cause different portions of the surface to be captured, and others to be missed. In a sequence of animated frames, the same portion of the surface will appear and disappear continuously, causing a very distracting artifact known as popping, or flickering. A better representation of the scene can be achieved by capturing those missed contributions, and enabling them to contribute to the pixel color by a percentage proportional to their coverage.

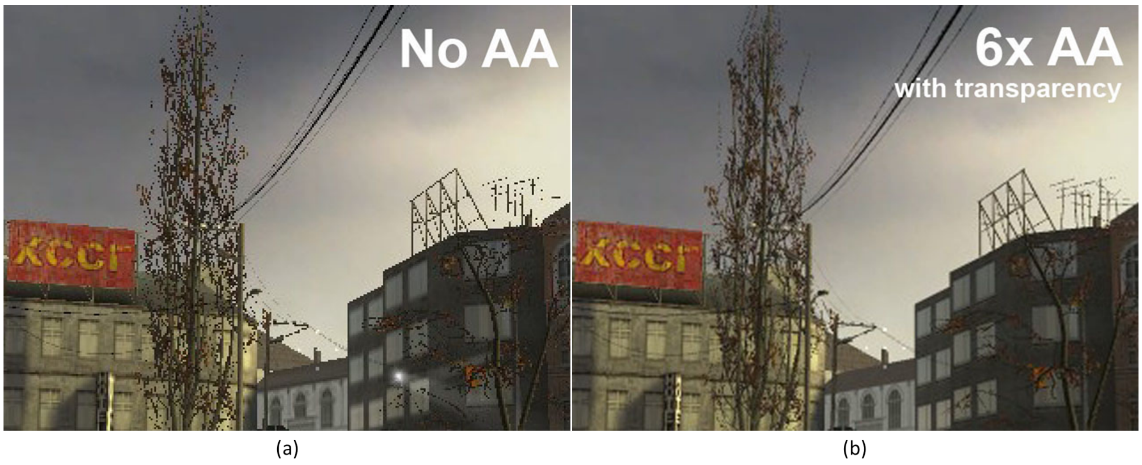


Figure 4.7: Aliasing example: discontinuity. (a) without AA, we see elements of the scene generating discontinuities: power cables and pole, trees, antennas and outdoor supports. (b) these problems are alleviated through AA, providing a more meaningful image (image from (GTA, 2012)).

A scene with complex details also may give rise to aliasing artifacts. Complex details are characterized by high frequencies presented in the scene, which combined to the sampling pattern, may produce Moiré effects. This artifact produces distinguishable patterns that are not present in the original virtual scene, and easily distract the viewer from the actual surfaces. Different AA approaches are used to fight this source of artifacts, however, there is no limit to the frequencies present in a scene so, a composition of frequencies may generate artifacts even with our better AA methods. Figure 4.8 shows an example of this kind of aliasing.

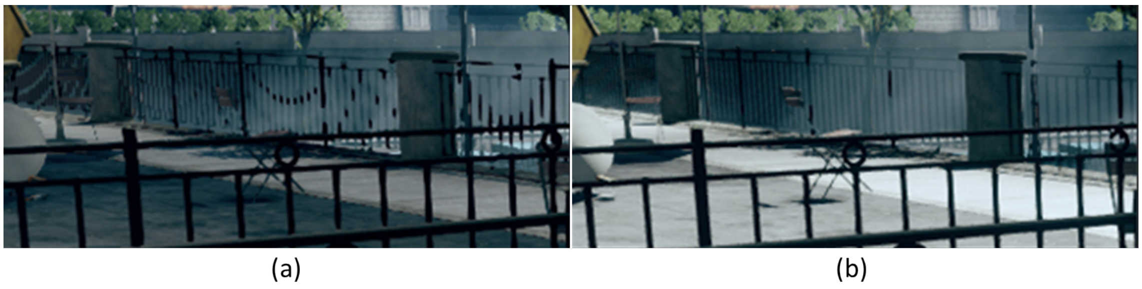


Figure 4.8: Aliasing example: complex details. (a) in the back fence, we see patterns artifacts, characterizing Moiré effect, due to improper sampling. (b) with improved sampling AA, the back fence is better represented (image from (BATTLEFIELD, 2014)).

Geometry is not the only source of aliasing. Textures mapped onto surfaces deformed by perspective projection may also generate artifacts. This happens when more than one texel map to the same pixel, but only the color of the nearest one is used. This causes details present in the texture to be missed, causing discontinuity and confusing patterns. As shown in Figure 4.9, AA through filtering is an acceptable solution. However, depending on the texture details, a higher sampling rate would be more adequate (as in the tree foliage in Figure 4.7).

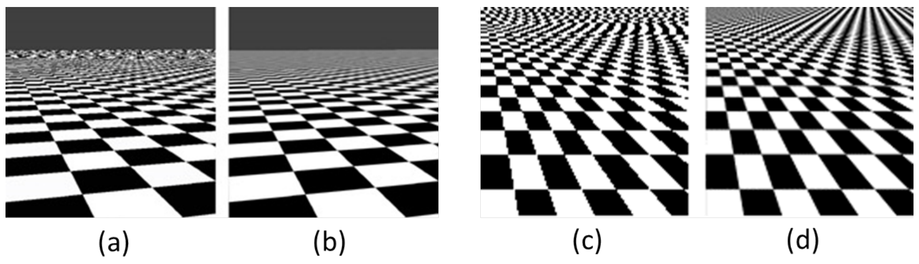


Figure 4.9: Aliasing example: texture. (a) subsampled texture mapping with aliasing artifacts. (b) texture mapping with filtering AA. (c) subsampled texture with Moiré artifacts. (d) texture mapping with least-squares AA. (image from (PHARR, 2004; PERSPECTIVE MOIRÉ, 2014)).

We focus on aliasing caused by sampling (capture) more than by displaying (reproduction) because, with constantly increasing displays resolutions, the occurrence of aliasing due to lack of pixels to represent the image is decreasing. This advent interestingly tends to shift the performance bottleneck from antialiasing some pixels to be able to generate fragments for a larger number of pixels. However, at least for now, we still are highly dependent on AA to generate quality images.

This section presented examples of common aliasing artifacts in computer generated images, pointing the issues involved in their occurrence. We showed that aliasing is unpleasant, sometimes disturbing, and that AA approaches significantly improve image quality and comprehension. Next section will describe and compare the AA techniques proposed for different kinds of aliasing and applications.

4.2 Anti-Aliasing Techniques and OIT Discussion

This section reviews techniques for treating the aliasing problem in computer-generated images. It is common for AA approaches to ignore transparency and consider all fragments as opaque. We will comment on each technique what would be required for it to support transparent fragments.

Techniques are classified by approach and compared based on their requirements and capabilities. We also discuss the usage of AA techniques with current hardware features and other desired effects.

4.2.1 Full-Scene Anti-Aliasing (FSAA)

Full scene means that all parts of the scene get some processing for AA—even if they are not visible. This is the first, simplest and most intuitive anti-aliasing approach. The key idea is to sample the scene at higher frequency than what is needed for display, by rendering to a higher resolution and, then, applying a filter to down sample to the desired resolution.

There are variations among the FSAA techniques that optimize distribution of samples inside the pixel area (BEETS, 2000). The simplest is the ordered grid (OGSS), which samples from a regular subpixel distribution. One implementation for this technique is to render to a screen n times larger than the desired resolution. For example, for a screen with size WxH, a 4-OGSS takes four samples per pixel by rendering to a framebuffer $2W$x$2H$, and downsampling with a box filter to combine each four pixels into one.

However, OGSS has problems to solve the most visible aliasing cases: nearly-vertical

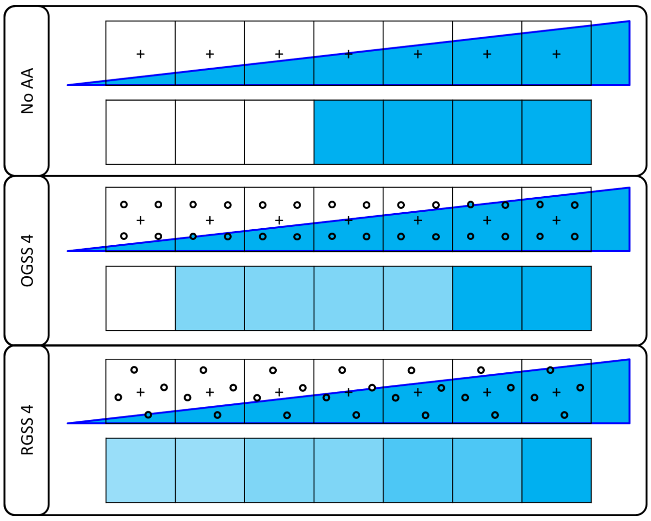


Figure 4.10: Critical edge angle for ordered grid sampling: The worst case for OGSS is nearly-vertical/horizontal edges. In such cases, OGSS loses the ability to provide all its shading levels, reducing smoothness. In the same scenario, RGSS provides more shading levels due to its rotated samples distribution. Image modified from (BEETS, 2000).

and nearly-horizontal edges, as shown in Figure 4.10. This figure also shows how the rotated grid technique (RGSS) does not share this limitation, providing more shading levels and, consequently, smoother edges for the nearly-vertical and nearly-horizontal edge cases. RGSS consists in rotating the grid of samples inside each pixel. A rotation of around 30 degrees leads to good results. This simple rotation changes the critical angle (which allows less shading levels) to a less disturbing edge angle, providing more pleasing images.

Regular patterns are easily perceived by our eyes, so, other distributions were proposed to alleviate the critical angle cases: (i) the random distribution, (ii) the Poisson distribution and (iii) the jittered sampling. Figure 4.11 illustrates these approaches.

Increasing the number of samples taken for each output (or displayable) pixel always reduces the artifacts of all distributions. To halving the aliasing, four times more samples are needed. Below, we discuss techniques that use more than one sample per output pixel with different approaches.

4.2.1.1 Super Sampling Anti-Aliasing (SSAA)

In SSAA(BEETS, 2000) the number of input pixels is increased with respect to the number of output pixels and there is a sample for each input pixel. This means that all the attributes are evaluated and stored for each sample (ex. depth, color, normal and texture coordinates. See Fig. 4.12). This approach gives the best image quality, at the cost of fully computing these attributes per sample.

4.2.1.2 Multi-Sampling Anti-Aliasing (MSAA)

The key difference between MSAA(BEETS, 2000) and SSAA is the amount of information super-sampled. In other words, MSAA is an SSAA where some pixel attributes are not evaluated for every sample, commonly this attribute is the shaded color. In MSAA, these attributes are evaluated at the center of the pixel and copied to each sample of the pixel (see Fig. 4.12).

One example of MSAA is to take four samples per pixel, computing for each sample its own depth and stencil values, but each of those samples receiving the same shaded color sampled at the pixel center. The difference from SSAA is that the shading computation is performed only once per output pixel, saving processing time; while the memory requirements are the same for both approaches.

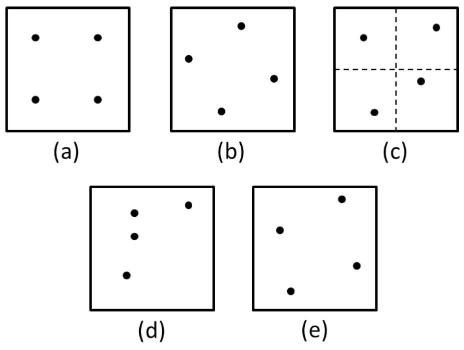


Figure 4.11: Super Sampling distributions: (a) Ordered Grid. (b) Rotated Grid. (c) Jittered Grid. (d) Random. (e) Poisson.

4.2.1.3 Coverage-Sampled Anti-Aliasing (CSAA)

This technique, called CSAA (by NVIDIA(NVIDIA, 2007)) or Enhanced-Quality Anti-Aliasing (EQAA) (by AMD(AMD, 2011)), uses the standard MSAA approach combined with extra samples per pixel to better capture pixel coverage, as shown in Figure 4.12. Some of the samples (not necessary half) capture color, depth, and location within the pixel. The remaining samples do not receive any of these attributes. They are only used to capture the fragment coverage at some location, in order to weight the contribution of the fragment to the final pixel color. In this web page (WOLIGROSKI, 2011) you can find details and image comparisons for CSAA and EQAA.

The image quality is improved by the extra coverage information, which better approximates a contribution of a fragment over the pixel area. However, the results are order-dependent since the coverage samples are not directly subject to the depth test (so, may be overwritten).

4.2.1.4 Directionally-Adaptive Edge AA (DAEAA)

DAEAA (IOURCHA; YANG; POMIANOWSKI, 2009)(JIMENEZ et al., 2011) consists of an MSAA process with improved image filtering in the final stage. The hardware-optimized MSAA is used to take samples of each pixel. These samples approximate the value of isolines passing inside a pixel, which estimate the primitive coverage and provide better color weighting.

Pixels are selected to be processed when they present different MSAA sample colors inside them, which means that they are from more than one fragment and the pixel may present aliasing (or, at least, a pixel that needs processing). Some regions, e.g. corners, may be excessively blurred, so they are masked before processing. For the remaining pixels, the gradient of primitive edges is computed by sampled isolines (see Fig. 4.12).

The isolines are calculated as straight lines, assuming low curvature along the pixel area. So, the function values, which the isolines represent, are approximated by the tangent plane at the pixel center, and resolved by least squares with the pixel samples. The last step is a stochastic integration, using a 3x3 box filter over the samples, which are weighted by the isoline length inside the pixel.

The image quality of 4xDAEAA is comparable with a 16xMSAA (IOURCHA; YANG; POMIANOWSKI, 2009), at the cost of four full-screen shader passes.

4.2.1.5 Subpixel-Reconstruction Anti-Aliasing (SRAA)

SRAA (JIMENEZ et al., 2011)(CHAJDAS; MCGUIRE; LUEBKE, 2011) technique was developed to work with deferred shading(DEERING et al., 1988)(DS). It is inspired by MLAA and MSAA techniques, and operates as a post-process over more information than the simple color buffer. The technique super samples depth and geometry normal attributes, and uses them to fill gaps among pixels of a normal size color buffer, in other words, it builds a virtual super-sampled image, then down samples to the output resolution.

A separate forward geometry pass is performed to construct the super-sampled depth and geometry normal buffers, by using the hardware MSAA. The DS pass is performed at final resolution and, after fragment shading, a super-sampled image is reconstructed by a cross-bilateral filter. The reconstruction takes depth and normal samples from a neighborhood, weighted by their distance to the pixel center. This process produces subpixel information, which is combined by a box filter to produce the final image with lower resolution.

To take advantage of this technique, a high shading costs is required in order to hide the AA poor performance, since the techniques needs an extra geometry pass to collect MSAA depths and normals, and the final filters are computationally intensive.

4.2.1.6 Enhanced Subpixel Morphological AA (SMAA)

SMAA (JIMENEZ et al., 2012) was built on top of the PMLAA pipeline, with improvements in edge detection, subpixel-feature management and temporal stability. It uses local luma contrast and a wider neighborhood search to detect edges, MSAA to improve subpixel features and temporal SSAA re-projection to provide temporal stability. The technique may operate in four different modes:

SMAA 1x: works only in the final image, so it cannot solve subpixels and temporal instability. It searches for aliased pixels by comparing the local luma contrast of the neighborhood with the PMLAA pre-defined patterns, and a new diagonal pattern. Only if the diagonal search fails the remaining patterns are verified. Sharpness of corners is improved by a wider search in the direction indicated by the detected edge, which identifies a corner and applies a reduced blur. To save processing time, only the top and left neighbors of q pixel are analyzed, the bottom and left ones are covered by other pixels.

SMAA S2x: operates over the SMAA 1x with addition of MSAA to solve subpixel aliasing. The pre-computed textures are modified to provide correct coverage for each subpixel position.

SMAA T2x: operates over the SMAA 1x with addition of temporal supersampling to solve temporal instability. The subsamples of the previous frame are projected into the current frame, weighted by their relative velocity with the current subsamples.

SMAA 4x: operates over the SMAA 1x with MSAA and temporal supersampling. This mode helps solving aliasing patterns detection, subpixel features, and temporal instability.

4.2.1.7 Accumulation, Area-averaged and Anti-aliased buffer (A-buffer)

A-buffer technique, besides the correct OIT computation, was mainly developed to solve anti-aliasing. If an opaque fragment does not cover a pixel entirely, then a list of visible fragments is built for that pixel. During the post-processing phase, the fragments in the list are combined.

Each fragment coverage is encoded in a bitmask (see Fig. 4.12), and all visible fragments are stored in a per-pixel linked list. When the geometry pass is over, the algorithm traverses the per-pixel lists, sorts them in front-to-back order and weights the fragments by their coverage when there is no interception. For interpenetrating fragments, the z_{min} and z_{max} values are used to approximate the visibility of the coverages, and weight the fragment contributions.

A-buffer produces high-quality images at the expense of unbounded memory and extra processing time.

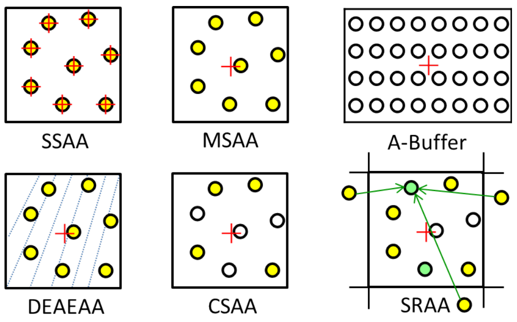


Figure 4.12: FSAA sampling approaches comparison: SSAA with full shading of all 8 samples per **pixel**. MSAA with 8 color+coverage samples and central shading per **pixel**. DAEAA with 8 MSAA samples to estimate isolines passing through the **pixel**. CSAA with 4 color+coverage samples and 4 coverage-only samples per **pixel**. SRAA with 4 color+coverage samples, 2 geometry samples, and the color reconstruction from other samples. A-Buffer has a bitmask with 8x4 coverage samples per **fragment**.

4.2.2 Image Post-Processing

This approach to handling aliasing is based in the processing of the final image. Differently from the FSAA approach, aliased pixels are detected and selected, so their processing does not interfere with the entire image. The aliasing removal of those pixels consists in attenuate the neighborhood high frequencies.

One pixel is considered aliased by the analysis of its neighborhood in the color buffer, with or without the aid of extra information, such as depth and normal. This exploration identifies high frequencies and builds a mask to select the pixels which need processing. The last step is the filtering of the selected pixels, which are combined with their neighbors in order to reduce the high frequency by smoothing them. In this phase, the absence of the selection mask would cause the entire image to be excessively blurred. As the result is approximated by blurring aliased pixels in the final image, this approach cannot handle subpixel issues.

4.2.2.1 Morphological Anti-Aliasing (MLAA)

MLAA (RESHETOV, 2009; JIMENEZ et al., 2011), is an image-based technique, which aims to minimize aliasing from edges and silhouettes. This technique was developed with the intent of removing aliasing from ray-tracing-generated images without considering more samples, which in ray-tracing are especially costly.

The working flow is simple: (i) find visible discontinuities between pixels by difference thresholds, (ii) identify aliasing patterns from these discontinuities and (iii) blend them with the neighborhood. The discontinuity can be determined by any metric, the first proposal used the sum of the 4 most significant bits of each color channel.

The image is scanned for discontinuities by comparing segments of different luminance between neighbor columns and lines, creating lists of vertical and orthogonal segments. These are classified as L, U or Z shapes, as shown in Figure 4.13. The U and Z shapes can be decomposed as two L shapes and processed separately.

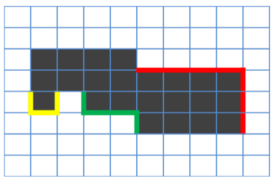


Figure 4.13: Examples of L (red), Z (green) and U (yellow) shapes detected in an aliased image by the MLAA technique.

From the L shapes, the longest edge is first selected, and it forms a triangle with the middle of the shortest edge. The triangle area is used to weight the blending with the neighboring pixels.

MLAA, when proposed, was able to significantly reduce the aliasing of ray-traced images very fast, because it only requires the color buffer to produce good results for nearly-vertical/horizontal edges, which are the most noticeable kind of aliasing.

Its limitations are, the inability to handle subpixel features, blur of border pixels even when there is no aliasing, and it is not well suited to animation due to temporal instability (because each frame is processed individually).

4.2.2.2 Practical Morphological Anti-Aliasing (PMLAA)

PMLAA (JIMENEZ et al., 2011) is a modification of the original sequentially processed MLAA, which leverage the GPU features. The technique works in the same three steps, each one was improved with relation to the original MLAA. The edge detection can make use of more information, such as depth, ids, normal and combinations of them to improve aliasing detection. The coverage estimation counts with bilinear filter and pre-computed areas acceleration. And the final blending also makes use of the fast filtering offered by the GPU hardware.

The edge detection phase masks the pixels requiring anti-aliasing, avoiding unnecessary processing. After that, the edge reconstruction takes place to estimate the coverage area. For each pixel, the algorithm searches for the end of the edge it belongs to in the top and left borders. This is done by bilinear filtering the pre-processed image, which makes possible analyze more than one pixel at time. Once the end of the edge is found, the crossing edges patterns also are established by bilinear filtering. With a small offset, the filtering is able to recognize four different types of crossing edges.

With identified edge width and crossing edges patterns, the algorithm uses these information to access a texture with pre-computed area coverage patterns. These values are summed into an accumulation buffer and used to blend the pixel neighborhood. The final phase uses the accumulated areas to weight the blending of the 4 neighbors of each pixel by bilinear filtering in sRGB space.

When compared to MLAA, PMLAA presents great improvement in terms of processing time, due to the efficient parallelization in GPU, and in terms of image quality, by using more information to correct select pixels. However, it maintains the main limitations. It may cause excessive blur in sharp edges and presents temporal instability.

4.2.2.3 Directionally Localized Anti-Aliasing (DLAA)

DLAA (JIMENEZ et al., 2011; ANDREEV, 2011) was developed for the PS3 console to handle the most disturbing aliasing type, which are the nearly vertical and horizontal edges. It was prototyped in Photoshop® (ADOBE, 2013), using high-pass filters, blur filters, contrast modifiers, thresholds, and masks. The main goal was, receiving only the final image, produce a better looking image as fast as possible.

The technique workflow is straightforward. First, a Sobel-like filter is applied to detect only vertical edges. A curved threshold function selects the desired edges by ignoring grayish values, so masking the vertical edges. A vertical blur is applied to the entire image, and the previously created mask is used to select the edges regions to be blended with the original image, producing anti-aliased vertical edges. All these filters are rotated by 90 degrees and the same process is repeated for horizontal edges. In short, the technique process can be resumed to:

1. Vertical blur filtering.
2. Vertical edge detection.
3. Threshold application over edge detection to mask vertical edges.
4. Mask use to blend vertical anti-aliased edges with the original image.
5. Repeat the process to anti-alias horizontal edges.

Different kernel sizes may be required, depending on the width of the edge. In order to detect long edges, the rate of blur is increased, then the high-pass filter is applied, followed by a contrast adjustment. In this process, only the long edges will survive, creating a long edges mask, which are blurred with a bigger kernel.

The results are comparable with MLAA, without the need of search for specific patterns, neither compute coverage estimations. As the filter was designed for nearly vertical and horizontal edges, as the MLAA, it does not perform well along diagonal aliased edges. Temporal instability and loss of subpixel features are also limitations of this technique.

4.2.2.4 Fast approXimate Anti-Aliasing (FXAA)

As the name says, FXAA (JIMENEZ et al., 2011; LOTTES, 2009) does not aim to acquire correct anti-aliasing. The proposal is a very fast algorithm which reduces some aliasing artifacts, improving image quality. It works by determining the need of anti-aliasing by local contrast examination. The selected pixels are processed by a directional edge blur filter.

The algorithm receives as input a color buffer only. The first step of the algorithm is to determine which pixels actually need anti-aliasing. In order to do that, each pixel is tested with a 4-neighbors (neighboring edges), which compares the luminance of the neighbors to verify if the contrast is higher than a user-defined threshold. The local contrast is determined by the difference between the maximum and minimum values of luminance among the current pixel and its four neighbors. If the contrast is low, the pixel is discarded from further processing.

For the pixels classified as needing anti-aliasing, a local luma gradient is computed. The directions perpendicular to the gradient are used to sample the neighborhood and blur the pixel. The user can define a scale factor, which controls how many neighbors are considered, varying from 2 to 8 samples. After performing the blur, the local contrast is tested again and, if it is too high, the default 2-samples blur is applied, ignoring the scale.

As the pixels are processed individually, this algorithm does not present good results for long nearly vertical and nearly horizontal edges, which are the most disturbing artifacts. A palliative solution is to use fractional super sampling (FSS) as input image, so, when FXAA down samples to the target resolution, these artifacts are reduced. As the other IAA techniques, this one also presents temporal instability.

4.2.3 Geometric Anti-Aliasing

Most of the aliasing in computer-generated images comes from geometric edges, so, this approach works with these edges to select and weight contributions of pixels in the color buffer.

During the geometry rasterization, the line equations of the edges are passed to the fragments. Analytically, the distances from the pixel center to the actual edge are encoded in the fragments. In the post-processing stage, the distances are used to identify aliased pixels and weight their contributions.

The post-processing stage resembles the IAA approach with distance information. Since this approach only works with geometric edges, it is not capable of handle aliasing from other sources, such as alpha texture and interpenetrations.

4.2.3.1 Distance-to-Edge Anti-Aliasing (DEAA)

DEAA (JIMENEZ et al., 2011; MALAN, 2011) is a post-processing anti-aliasing technique. It encodes in each pixel the distance to the edges of the triangle it belongs to. After rendering, these distances are analyzed to verify if the pixel needs anti-aliasing.

During the rendering, in the vertex shader, each vertex of the triangle receives either the R, G or B color. The rasterization will generate the fragments with these colors interpolated and the distance to the RGB base approximates the fragment distance to the triangle edge. Four distances are encoded in an RGBA texture of each fragment: up, down, left and right directions, each one in an 8 bits channel.

In the post-processing stage, the smallest distance of each pixel is verified, if it is less than one pixel, the current pixel is marked as belonging to a border, as shown in Figure 4.14. Only border pixels are processed. If two neighbor pixels have competing distances, which means, both indicate primitive coverage in the neighbor area, the smallest one is chosen and blended into the neighbor, pondered by the coverage area indicated by the distance.

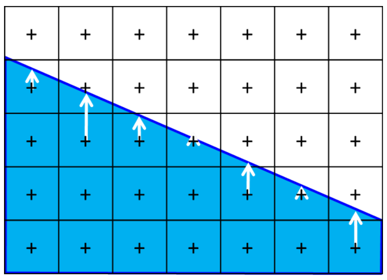


Figure 4.14: Extra information is associated to edge pixels: the distance from its center to the geometric edge. This data is used to weight the blur in a post-processing phase. Image modified from (MALAN, 2011).

This solution does not solve sub-pixels problems because no extra samples are taken. Other cases untreatable by this technique are interpenetrations (because the geometric edge is not the limit of the primitive in screen space), and cases where edges are not present, such as shadows and textures.

4.2.3.2 Geometric PostProcessing AA (GPAA)

GPAA (JIMENEZ et al., 2011; PERSSON, 2011) gets the silhouettes information from the pipeline. A preprocessing stage evaluates only the silhouettes edges, computing their equations in screen space and passing this information to the pixel shader. As the DEAA, the blur is weighted by the distance of the pixel center to the edges.

This technique presents high-quality results, with accurate coverage even in nearly vertical and horizontal edges, and temporal stability. However, the edge extraction increases memory consumption and processing time, degrading fast with geometry augment.

4.2.3.3 Geometry Buffer Anti-Aliasing (GBAA)

GBAA (JIMENEZ et al., 2011; PERSSON, 2011), differently from GPAA, stores the geometric information during the rendering, without the need for a preprocessing stage. The distances are calculated for each vertex in the vertex shader, and interpolated by the rastering process. In this sense, it is similar to DEAA, the final step after rendering checks the distances in each pixel, if smaller than half a pixel, the pixel receives anti-aliasing, so the coverage of neighboring pixels is computed and they are blended.

The memory requirement is smaller than for GPAA and the overall quality is maintained. However, the entire processing is still expensive for high demands of FPS.

4.2.4 Discussion

Current GPUs have special components to handle transparency, and others to anti-aliasing. Often, the transparency components are used to support anti-aliasing and vice-versa. Here, we will comment the usage of these resources, along with rendering issues related to the application of OIT and AA effects.

The hardware standard to support the transparency effect is the fourth color channel, called alpha channel. This channel is used to encode the opacity attribute, and it weights the contribution of the fragments to the final pixel color. The hardware also supports the blending equations proposed by Porter and Duff (PORTER; DUFF, 1984).

In order to provide anti-aliasing, the classical hardware support is for the MSAA. The components required are a multi-sampled texture, to store the samples information (depth, color, etc), and a multi-sampled rasterization algorithm. Recently, other techniques have gain support, such as CSAA, EQAA, FXAA, and others.

An intriguing problem nowadays is how to combine AA and/or OIT with deferred shading, because of the high memory consumption and technicalities involving the decoupling of samples from their originating surfaces.

Table 4.1 summarizes the main features discussed for each technique.

Class	Technique	Samples Per Pixel	Shadings Per Pixel	Geometry Passes	Pattern Detection	Pre Process	On the fly	Solve Subpixel	Order Dependent
FSAA	SSAA	N	N	1			●	●	
	MSAA	N	1	1			●	●	
	CSAA	N+K	1	1			●	●	●
	DAEAA	N	1	1			●	●	
	SRAA	N	1	2		●		●	
	SMAA	N	N	1	●		●	●	
	A-buffer	N	1	1			●	●	●
IAA	MLAA	1	1	1	●				
	PMLAA	1	1	1	●				
	DLAA	1	1	1					
	FXAA	1	1	1	●				
GAA	DEAA	1	1	1			●		
	GPAA	1	1	2		●			
	GBAA	1	1	1			●		

Table 4.1: Anti-Aliasing comparison table: techniques are clustered by class, **samples per pixel** column indicates the sampling rate per pixel, **shadings per pixel** indicates how many times the color is evaluated per pixel, **geometry passes** indicates how many times the geometry must be processed, **pattern detection** indicates the presence of a module to detect aliasing patterns in the final image, **pre-process** indicates the need for a pre-processing step, **on the fly** indicates anti-aliasing processing during the rendering, **solve subpixel** indicates if the technique is able to solve sub-pixel aliasing, as thin primitives and temporal aliasing, **order dependent** indicates if the order in which the fragments arrive impacts in the result.

4.2.4.1 AA with the aid of transparency hardware

AA techniques can make use of the opacity channel to weight the fragment visibility by its coverage over the pixel area (ex.: (NVIDIA, 2004)). The blending of such fragments become order-dependent, which means that artifacts may be generated if they are combined out of depth-sorted order.

To obtain and store information for more than one sample per pixel is costly. The storage space can be saved by encoding the coverage of the fragments into the alpha channel. For example, if 1 of 4 samples is covered, the opacity of the fragment will be multiplied by 0.25; if it was opaque, it will become transparent with 0.25% of opacity. This can be compared to what is done by coverage AA techniques, where the fragment contribution is weighted by the amount of spatial samples it covers.

This approach inserts the transparency order-dependency into the AA problem. If the fragments are combined out-of-order, their visibility will be incorrect due to the blending equation.

Rendering of billboards often uses this approach by drawing the opaque background first. For particles effects, which present alpha textures and similar colors, out-of-order algorithms may be used to approximate the result. The performance of this approach is quite superior to other anti-aliasing techniques, but the set of cases to which it is applicable is restricted.

4.2.4.2 OIT with the aid of anti-aliasing hardware

Since the graphic hardware is prepared to store more than one sample per pixel to handle AA, some OIT techniques use this memory to store transparent fragments. The storage capacity implemented for AA is adequate to compute transparency, in case the number of layers per pixels does not exceed the limit of slots for samples.

The Stencil Routed technique (MYERS; BAVOIL, 2007) is one usage proposal of the MSAA textures to handle OIT, with multiple geometry steps if necessary. This technique basically uses the high-performance of these textures to store and combine up to n fragments per geometry pass, combined with the stencil buffer for fragment counting.

Stochastic Transparency (ENDERTON et al., 2010) also make use of the MSAA hardware, but with a different approach. It requires three geometry passes to estimate visibility and generate, for each fragment, a coverage probability. When the fragment arrives, this probability is used to fill the MSAA samples individually. Higher probabilities tend to fill more samples, which mean that the fragment is more visible. As the pixels are processed individually with the computed probabilities, the generated image contains large amounts of noise.

For low amounts of transparent layers, the MSAA hardware is a fast approach. The AA of the transparent scene could be performed with an IAA technique, without the benefit of temporal stability.

4.2.4.3 AA for transparent scenes

Apply anti-aliasing to a transparent scene is an interesting problem, mainly because of the treatment for AA samples and the order dependency. A GAA technique would remove the samples management, but not the ordering problem. And an IAA approach, applied only to the final image, brings reasonable results; however, it is unable to solve temporal instability.

Buffer-based OIT needs high amounts of memory, and to take more samples to perform

AA for this approach may be impractical. For example, a 4xMSAA algorithm for opaque scene in a 640x480 screen needs 9.4MB (RGBA + depth), if storing five transparent layers per pixel with their samples, the memory requirement goes up to 46.87 MB. Now, think about a bigger screen and you will see the problem. Coverage-only techniques have been used with the recent OIT algorithms proposals; however it relies in sorting by the central z, not having the ability to solve interpenetration cases.

For depth peeling approaches, a NxSSAA would cost n times more processing at each geometry step, and the other multi-sample techniques can easily degenerate to SSAA brute-force. Again, the solution relies in applying IAA and losing temporal stability.

The GAA techniques were thought to blur surviving fragments in the color buffer, this means that it does not resolve thin primitives, neither temporal aliasing. Its application for different layers of OIT was never properly explored.

4.2.4.4 The deferred shading issue

Deferred shading (DS) is a rendering pipeline which supports expensive shading computations. A classic forward pipeline shades the fragments which pass the depth test and stores only their color, which is replaced when a new fragment passes the depth test. The DS strategy consists in storing the geometric information, into the so-called G-Buffers, when a fragment passes the depth test. Only the final visible fragments are shaded at the end of the rendering process, saving expensive computations for fragments which are not visible.

The G-Buffers represent a big memory budget, because they store all the information needed to shade the fragment. To apply SSAA in the DS pipeline would be necessary to store all the information for all the samples, implying prohibitive memory consumption. The benefits of the MSAA are lost due to the decoupling of samples from their original surface, degenerating to SSAA brute-force.

IAA techniques, due to their detachment of the pipeline, are compatible with any rendering process, including deferred shading. However, the price to use such AA approach is given up the ability to handle subpixel features and temporal instability. GAA techniques may be used efficiently.

DS pipeline is incompatible with OIT because its main idea is to avoid processing occluded fragments, while OIT implies in partial occlusion, requesting the processing of more than one fragment per pixel in depth-sorted order. The simplest way to combine them is rendering OIT in a second stage, when the opaque depth buffer is set, with a forward rendering pipeline.

Combine order-independent transparency with high-quality anti-aliasing in a deferred shading pipeline is a challenge. The issues involved are: (i) high shading costs, which motivates the use of DS pipeline, (ii) depth-sorted order-dependency, intrinsic to the OIT problem, and (iii) high memory requirements, associated to all the three.

4.3 Area-Sample AA for OIT Attempt

Edges and thin primitives represent the major sources of spatial and temporal aliasing. Thin primitives cannot be handled by image filters and are challenging even for increasing point-samples, resulting in color discontinuities and poor color transitions. Add transparency to this problem and the costs of compositing fragment samples rise rapidly. We introduce an A-buffer-based approach for anti-aliasing of opaque and transparent fragments, without the A-buffer limitation of point samples and sampling masks. For each fragment, only one sample is produced, maintaining low shading and compositing costs. All fragments that intersect a pixel are ensured to be generated through conservative rasterization. Fragments with partial coverage are detected automatically and their contributions are estimated through an accelerated area sampling. We present a proof of concept running on today's GPUs. Results show that our approach is comparable to MSAAx8 for edges and better than MSAAx8 for thin primitives, presenting smoother color transitions and color continuity.

Aliasing is one of the most studied topics of computer graphics. The continuous efforts devoted to it are justified by the severe artifacts produced by aliasing, which significantly reduce overall image quality and realism. The most frequent and disturbing forms of aliasing come from undersampling of thin primitives and geometric edges, causing popping and jagging artifacts, respectively.

no AA
2xMSAA
4xMSAA
8xMSAA
ASAA

Figure 4.15: Our accelerated area-sampling approach produces superior edge-AA. Point-sampling approaches rasterize S samples per-fragment, so they can generate up to S shades of color transition. This results in blocks of pixels with the same shade and abrupt color transitions between these blocks. Our proposed area-sampling generates smoother color transition while rasterizing only one sample per fragment. Because of its ability to capture smaller variations on pixel coverage, ASAA can paint each pixel with a different shade; compare the gray lines on the bottom of each method.

Today, the most efficient anti-aliasing (AA) approaches rely on aliasing patterns detection and smart blurs. Because they operate only on the final image, they are compatible with any sort of rendering – e.g. deferred shading, ray-tracing, transparency, etc. However, the tradeoff between performance and quality leaves these approaches lacking temporal stability and effectiveness on thin primitives.

Approaches based on higher sampling rates are more robust and stable. Significant improvement is visible by merely increasing the number of coverage point-samples. When comparing images with 2, 4 and 8 samples, the improvement is evident. Nonetheless, after 8 samples the visual impact is hardly palpable (RESHETOV, 2012). This means that, after 8 samples, a larger number of samples is required to perceive improvement in the anti-aliased image. Combining point-sampling with order-independent transparency (OIT) requires per-sampling compositing, inflating the costs of each extra sample.

As an alternative to point samples, line-samples increase the accuracy of the estimated coverage with fewer samples. They provide good AA of 2D polygons and are particularly suitable for text. However, how it would work for AA of 3D meshes is not clear yet. There is still the area sampling approach, which is commonly set aside for high-quality offline rendering due to its high computational and memory costs.

In this section we present a proposal for an area-sampling approach for real-time anti-aliasing of opaque and OIT primitives, which is currently under development. Our proposal consists of three simple phases: (i) conservative rasterization, (ii) accelerated coverage estimation and storage, and (iii) per-fragment sorting and compositing. We present renderings of opaque and transparent scenes comparing our approach to increasing coverage point-samples. Preliminary results show that area sampling can provide smoother color transitions, as can be seen in Figure 4.15, and improved color continuity for thin primitives. However, there are still unsolved issues.

Despite a rich literature on AA and OIT, both are still highly active research areas (ANDERSSON, 2012). This section addresses both problems by proposing an AA approach with native OIT support. The main feature of our proposal is its ability to handle AA with a single sample per fragment, which, combined with OIT, maintains low memory requirements and fast per-fragment blending — instead of per sample blending.

4.3.1 Area-Sampling Anti-Aliasing

Aliasing artifacts come from undersampling of the scene. Therefore, our proposal uses conservative rasterization to guarantee that all fragments that may contribute to a pixel color will be generated for processing. A novel accelerated area sampling is used for computing the fragment coverage over the pixel area. Each fragment is processed according to two visibility weights: its coverage (⥉) and its absorbance (α). These weights determine how much the fragment will contribute to the pixel color, and how much of back-most contribution will be visible through that fragment, simultaneously handling AA and OIT. A per-pixel structure stores the fragments for posterior sorting, accumulation, and compositing. Table 4.2 defines the meanings of all symbols used in this section.

⥉	=	coverage
α	=	absorbance
τ	=	transmittance
A	=	area
C	=	rgb color
d	=	pixel square diagonal ($d = \frac{\sqrt{2}}{2}$)
dte	=	distance to edge
H_s	=	screen height
N	=	number of fragments, $N \in \mathbb{N}$
P	=	projection matrix
r	=	pixel circle radius ($r = 0.5$)
θ	=	angle from circular sector
v	=	vertex from original triangle
V	=	vertex from dilated triangle
W	=	attributes weights matrix
W_s	=	screen width
X_c	=	X is in clip space
X_{pc}	=	X is attribute of pixel circle
X_v	=	X is in view space

Table 4.2: List of symbols: the empty box symbol $\square_x$ denotes any attribute belonging to x.

4.3.2 Conservative Rasterization

Conservative rasterization (CR) guarantees that all fragments of a geometric primitive will be generated, despite how small their coverage over the pixel area may be. The best approach to implementing CR would be by modifying the rasterization algorithm to include fragments that cover any part of the pixel, not only some sample. However, the rasterizer is a fixed part of graphics APIs pipelines — i.e. cannot be modified by a shader program. We use a workaround to implement CR on today's GPUs to validate our AA proposal.

The CR algorithm from GPU Gems 2 (HASSELGREN; AKENINE-MöLLER; OHLSSON, 2005) works by dilating the triangle edges in clip space. This dilation, which corresponds to half pixel in screen space, is enough to assure that small edge fragments will cover the pixel central sample and be generated by the rasterization process. The original work was not developed for final image generation; therefore, it presents some limitations that we address in this section. The main limitations are:

- No support for back-facing triangles, needed for rendering of transparent primitives.
- Dilated vertices in NDC space: for perspective-correct attributes interpolation, these vertices must be sent to the rasterizer in clip space.
- Dilated triangle is not coplanar with the original triangle: the triangles must be coplanar for the dilation of vertices attributes.
- No support for triangles clipped by the near plane: this issue prevents correct close-ups of the scene.

Here we describe how we address these issues for AA rendering. First of all, we clip the original triangle against the near plane. This generates one or two triangles, which are dilated. Algorithm 2 details the operations to dilate the *current* vertex in the geometry shader (this procedure is performed for each vertex, *previous* and *next* are the other two vertices of the original triangle). It starts by computing the equations for the edges passing through the $current$ vertex. Then, it checks if the triangle is front or back-facing, so the edges can be displaced in the correct direction for dilation (outwards the triangle). Edges are displaced by $hPixel = (\frac{1}{W_s}, \frac{1}{H_s})$. At last, the point where these displaced edges intercept is the dilated vertex.

For interpolation purposes, original and dilated triangles are required to be coplanar, so dilated attributes can be computed (see sec. 4.3.3). Then, the next step is to project the dilated triangle onto the original triangle plane. Because the dilation procedure generates a triangle orders of magnitude larger than the original, double-precision is required for this operation. Vertices of the dilated triangle are projected onto the original triangle plane in view-projection direction. In clip space, this direction is given by the dilated vertex itself, as shown by the red dotted lines in Figure 4.16. At last, we reconstruct the correct depth value of the dilated vertices for sorting. Since we have w_c, and we know that after projection $w_c = -z_v$, we can recover z_c using the third column of the projection matrix P, which is the column that transforms depth.

We identified two cases that our CR approach does not handle. First, when some of the dilated vertices falls behind the clipping plane. This causes errors when projecting the dilated triangle onto the original triangle plane. For simplicity, we go around this issue by keeping the original vertex when this situation occurs. The second issue is related to triangles that project into lines in screen space — i.e. degenerated triangles. When the triangle becomes closer to a line in screen space, we notice increasing precision errors when projecting the dilated triangle onto the original plane. When all vertices are aligned, this projection is not possible. Our results face some artifacts coming from these issues, but they will not be a problem for CR implemented in the rasterizer. That is because the rasterizer can work directly in screen space, with no need for triangle dilation and projection in clip space.

4.3.3 Computing Attributes for Dilated Vertices

CR generates a dilated triangle to be rasterized instead of the original triangle. The dilated triangle projects into an area half pixel bigger at each edge in screen space, which ensures that all fragments from the original triangle will be generated for coverage estimation. In clip space, which is the proper space for perspective-correct dilation of vertices attributes, the dilation distances are not homogeneous. Because more 3D area is projected into the same pixel when moving away from the camera, the distance for a dilated vertex increases with the vertex distance to the camera, as shown in Figure 4.16. Therefore, proper dilation of attributes is mandatory due to the non-homogeneity of the dilation operation in clip space. The dilated attributes must be so that, when the rasterizer interpolates them, at the pixels where the original vertices are projected, the interpolation will generate the original attributes values.

One approach for computing interpolated attributes during the raster process is mapping each fragment in the triangle using barycentric coordinates (which is consistent with

Algorithm 2 Vertex Dilation Extended

```
1:  **[V] = dilateVertex(tPlane, previous, current, next)**:
2:  **output** dilated vertex V from $current$ vertex
3:  **inputs**:
4:  - tPlane:   (xyzw) plane equation for original triangle
5:  - previous: (xyw) previous vertex in clip space
6:  - current:  (xyw) vertex to be dilated in clip space
7:  - next:     (xyw) next vertex in clip space
8:  Compute edges passing through current vertex
9:    $edge0 = (current - previous) \times previous$
10:   $edge1 = (next - current) \times current$
11: Test if triangle is back-facing
12:   if $tPlane.w < 0$
13: Displace the edges by + half Pixel
14:     $edge0.z += hPixel \cdot |edge0.xy|$
15:     $edge1.z += hPixel \cdot |edge1.xy|$
16: Compute the intersection point of the two edges.
17:     $V.xyw = edge1 \times edge0$
18:   else
19: Displace the edges by - half Pixel
20:     $edge0.z -= hPixel \cdot |edge0.xy|$
21:     $edge1.z -= hPixel \cdot |edge1.xy|$
22: Compute the intersection point of the two edges.
23:     $V.xyw = edge0 \times edge1$
24:   end
25: Project dilated vertex onto the original triangle plane
26: $V.xyw = V.xyw + \hat{V}.xyw \times (-\frac{(tPlane.xyz \cdot V.xyw) + tPlane.w}{tPlane.xyz \cdot \hat{V}.xyw})$
27: Depth reconstruction using the projection matrix P
28:   $V.z = P[:,3] \cdot [0, 0, -V.w, 1]$
```

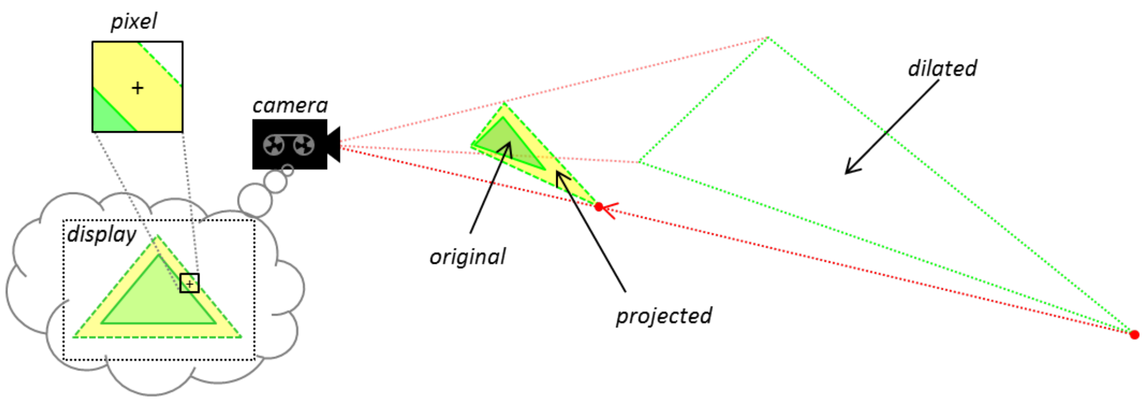


Figure 4.16: Dilation generates a triangle orders of magnitude bigger than the original (proportions distorted for exhibition). The dilated triangle is projected onto the original plane in view-projection direction (red dotted lines). Due to the perspective projection, with increasing distance to the camera, more area is projected into the same pixel. This causes the dilation to be non-homogeneous in 3D space so that, when projected, it will correspond to half pixel at each edge in the display. The pixel detailed shows a fragment that would not be generated by the rasterization of the original triangle, but is generated when we rasterize the dilated triangle.

OpenGL interpolation). This generates three weights, one for each vertex, used to weigh any attribute (color, texture coordinates, etc). We use the same approach to compute the dilated attributes; however, because by definition the dilated vertices are outside the original triangle, instead of mapping dilated vertices in the original triangle, we begin by mapping the original vertices in barycentric coordinate of the dilated triangle. This generates a set of weights $\in [a..i]$ that map the dilated vertices V_k into the original vertices v_k ($k \in [0..2]$):

$$v_0 = aV_0 + bV_1 + cV_2,$$
$$v_1 = dV_0 + eV_1 + fV_2,$$
$$v_2 = gV_0 + hV_1 + iV_2.$$

By organizing these equations in matrix format, we get a linear system in the form $WV_k = v_k$:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \cdot \begin{vmatrix} V_0 \\ V_1 \\ V_2 \end{vmatrix} = \begin{vmatrix} v_0 \\ v_1 \\ v_2 \end{vmatrix}.$$

Since we have attributes for v_k and need to compute attributes for V_k, we solve the system by inverting the matrix of weights (W):

$$W^{-1}v_k = V_k. \tag{4.1}$$

During rasterization, the original attributes will correctly map to the pixels where the original vertices project. However, if a fragment does not cover the central sample, the interpolated attribute will be extrapolated outside the primitive boundaries, assuming mistaken values. We describe how to overcome this issue in Section 4.3.4.2, depending on the presence of vertices inside the pixel area.

4.3.4 Computing Fragment Coverage Over The Pixel

This section describes our novel accelerated area sampling technique. For comparison purposes, we also describe an exact area sampling algorithm. The exact solution computes the area of the polygon resulting from the intersection between the triangle and the pixel square. Our proposed area sampling approximates the pixel square by a circle, using distance-to-edges to estimate coverage. This approximation is significantly faster than the exact approach, and the quality impact is minimal.

For both approaches, we compute the distances from the fragment center to each edge of the original triangle, and use them to identify trivially full or empty coverages. If a fragment has a negative distance, to any edge, whose module is greater than half-pixel diagonal ($d = \sqrt{2}/2$ for the exact method, and $r = 0.5$ for the circle-approximation), then it is certain that the intersection of the pixel with the triangle is empty, so the fragment is discarded. If a fragment has all distances positive and greater than half-pixel diagonal, then the pixel is completely inside the triangle and the fragment coverage is 1. Otherwise, we need to compute the area of coverage.

4.3.4.1 Coverage From Polygon Area

The fragment coverage over the pixel area can be computed from the area of the intersection of the pixel square and the original triangle. For computing the polygon resulting from this intersection, we use the Sutherland-Hodgman (FOLEY et al., 1990) polygon-clipping algorithm. For accelerating the identifications of trivial cases and clipping boundaries, we use the Cohen-Sutherland (FOLEY et al., 1990) line-clipping algorithm. This approach is consistent with the seminal work of (CATMULL, 1978). Figure 4.17 shows relevant intersection cases for this approach.

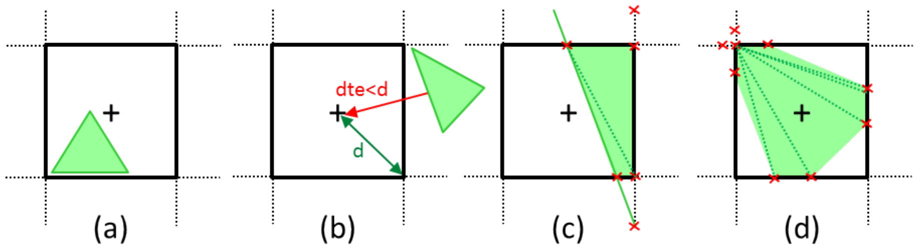


Figure 4.17: Square-triangle intersection cases: (a) All in: return the triangle area. (b) All out: return 0. (c) Most common case (single edge): compute up to 6 intersections, triangulate and return the sum of up to 2 triangle areas. (d) Worst case: compute up to 9 intersections, triangulate and return the sum of up to 5 areas.

When all the vertices are inside the square, or there is no intersection of the edges with it, the Cohen-Sutherland algorithm help us to return the coverage quicker. When we must compute the intersection of the edges with the square boundaries, Cohen-Sutherland helps to identify against which boundaries the edges need to be clipped. The worst case for this approach occurs when we have the maximum number of intersections, generating a complex polygon that needs more computations for its area evaluation. However, this is not a common case in rendering. Most of the fragments with partial coverage will be at the edges of large triangles (Fig. 4.17(c)), therefore, this is the case we optimize in the next section.

4.3.4.2 Coverage From Circle Approximation of Pixel Square

For accelerating the most common case of partial fragment coverage (edges), we approximate the pixel square area by an inscribed circle. By doing so, we can compute a good coverage approximation using relations of the circle with the distance from the pixel center to the triangle edges (dte). This is because, differently from the square, an edge passing through a circle will always generate the same covered/uncovered area, independently from the edge direction. Only the distance to the edge influences the covered/uncovered area. Our approach differs from (CATMULL, 1984), which does not use a circular area sampler, but operates through a circular Gaussian-shaped filter.

An edge passing through the circle creates a circular segment, whose area defines the coverage. If the dte is negative, meaning that the pixel center is outside the triangle, the segment area gives us the percentage of covered circle. Otherwise, if the dte is positive, the total circle area minus the segment area give us the coverage. Because the pixel-circle area ($A_{pc} \approx 0.785$) is smaller than the pixel-square area ($A_{ps} = 1$), contributions evaluated inside the pixel circle ($\uparrow_{pc}$) must be scaled by the circle area to the range [0, 1] by

$$\uparrow = \frac{\uparrow_{pc}}{A_{pc}}. \tag{4.2}$$

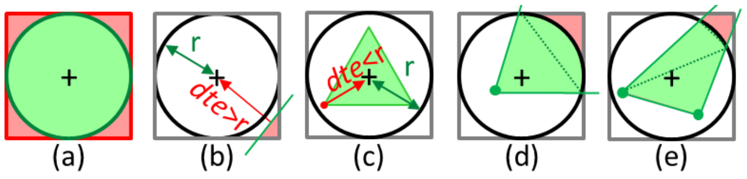


Figure 4.18: Especial cases for triangle coverage over the pixel circle: (a) 21,5% of discontinuous area at the square corners is not covered by the circle. (b) trivial reject: at least one dte places the circle outside the triangle. (c) trivial accept: all triangle vertices are inside the circle. (d) single vertex inside: the coverage comes from a triangle and a circular segment. (e) two vertices inside: the coverage comes from two triangles and a circular segment.

As shown in Figure 4.18(a), the circle approximation does not cover 21,5% of the pixel at its corners. However, the uncovered area is not continuous. So, for a triangle to be completely missed by the circle approximation, it must be inside one of the corners, whose area is only 5.4% of the pixel area. An example of contribution loss due to the circle approximation is shown in Figure 4.18(b). Notice that the triangle will not be missed by all circles, and its contribution will be accounted by neighboring pixels. Figure 4.18 also shows special cases, when one or more vertices are inside the circle.

We divide the intersection problem into five cases:

- Trivial zero-coverage: when at least one of the dtes is negative and its module is greater than the radius (Fig. 4.18(b)).
- Trivial triangle area: when all the three vertices are inside the circle (Fig. 4.18(c)). In this case, the attributes are adjusted to an average of the three vertices.
- Single vertex inside the circle: the covered area is the sum of areas from a circular segment and a triangle (Fig. 4.18(d)). In this case, the attributes are adjusted to the vertex attribute.
- Two vertices inside the circle: the coverage comes from the areas of two triangles and a circular segment (Fig. 4.18(e)). In this case, the attributes are adjusted to an average of the two vertices.

- And, for the most common case (exemplified in Figure 4.19), when one or more edges cut the circle, we evaluate the pixel coverage by removing uncovered segment areas (${}_{u}A_{seg}$) from the total circle area as:

$$\Umbrella = A_{pc} - \sum {}_{u}A_{seg}. \tag{4.3}$$

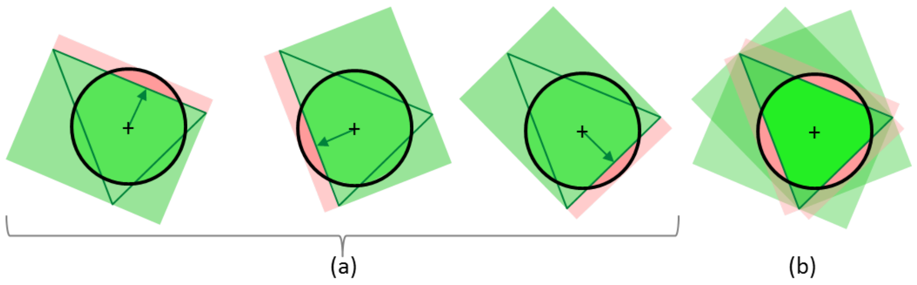


Figure 4.19: Pixel coverage from multiple dtes: (a) from each dte, we define a non-covered area (in salmon pink); (b) the total area minus the sum of non-covered areas gives us the total covered area.

The area of a circular segment is given by

$$A_{seg} = \frac{r^2}{2} \times (\theta - \sin(\theta)), \tag{4.4}$$

where r is the circle radius and θ is the angle between the lines that define the circular sector containing the circular segment. The angle θ can be computed from the dte as

$$\theta = 2\text{acos}(\frac{dte}{r}) = 2\text{acos}(2dte). \tag{4.5}$$

Therefore, we can compute the circular segment area from dte as

$$A_{seg} = \frac{r^2 \times (2\text{acos}(2dte) - \sqrt{1 - 4dte^2})}{2}. \tag{4.6}$$

In an edge case, a fragment that does not cover the pixel center has its attributes extrapolated outside the triangle boundaries. They can be adjusted using the derivatives of attributes from neighboring pixels weighed by the closest dte.

4.3.5 Compositing

This section describes how opaque and transparent fragments are composited into anti-aliased pixels. For opaque-only fragments, fragment polygons are checked for intersections with each other, and splitted into new fragment when they intersect. Then, fragments are sorted by depth and clipped against the visible area left by previous fragments, the occluded portions are discarded. The remaining fragments are weighed by their areas and composited (CATMULL, 1978).

Compositing transparent fragments is not so simple, because there may be non-occlusion areas. We do not know of any publication that describes the exact compositing for a list of interpenetrating transparent fragments. We describe a possible algorithm for evaluating the composition of transparent fragments, as polygons with exact coverage, to illustrate the complexity of the problem we are addressing.

When considering geometric coverage for AA, there are four different compositing cases between two fragments (examples can be seen in Figure 4.20):

Total: the covered areas from both fragments match. The fragments are weighted by their total coverage and blended.

None: there is no superposition of the covered areas, so neither fragment reduces the visibility of the other. The fragments are weighed by their respective coverages and accumulated.

Partial: there is a superposition of covered areas that may generate up to four regions: (i) front-fragment only, (ii) back-fragment only, (iii) front-fragment over back-fragment, and (iv) uncovered area. Each region is weighed by its coverage before compositing.

Intersection: the fragments intersect, generating up to five regions: (i) front-fragment only, (ii) back-fragment only, (iii) front-fragment over back-fragment region, (iv) back-fragment over front-fragment region, and (iv) uncovered area. Each region is weighed by its coverage before compositing.

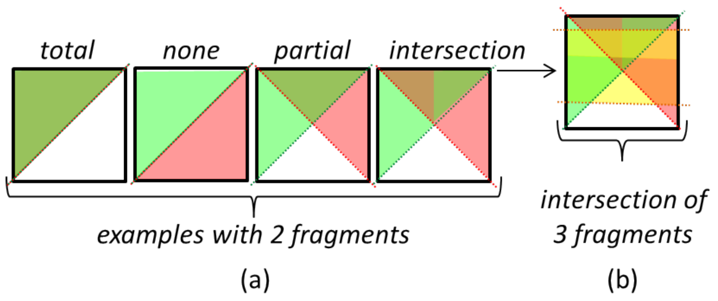


Figure 4.20: Exact compositing of transparent fragments with examples of partial coverage: (a) examples for the four possible cases of combined coverages for the two fragments (red and green). (b) example of the worst case, intersection, with three fragments (red, green and yellow), demonstrating how rapidly grows the number of different combinations of coverages (regions).

For exact accumulation of contributions from a list of transparent fragments, we would need to keep tracking and compute superpositions and intersections between each pair of fragments. Each fragment is a planar polygon, which means that the evaluation of two fragments may generate up to four covered regions (intersection case). In the worst case, all fragments intersect each other, and the cost for computing the polygons is $O(N^2)$, where N is the number of fragments being composited. The intersections of these polygons will divide the space into $O(N^3)$ regions with different contributions from each fragment, whose coverage must be computed and composited separately (Fig. 4.20(b) shows an example of these regions).

The exact approach is very costly and, as the total pixel area is being subdivided, the proportional contribution of each region becomes insignificant. An alternative is to discretize the pixel area by point samples (e.g. MSAA), but they have a high computational cost for storage, sorting and compositing of transparent fragments. That is one of the reasons why we choose to work only with the percentage of coverage. By doing so, we store, sort and compose only a single sample per fragment with cost $O(N)$.

Our approach is based in the front-to-back blending equation (FTB) (PORTER; DUFF, 1984), which defines that the contribution of each fragment is weighed by two factors: (i) the fragment local visibility, independent from other fragments, and (ii) a global visibility, which is limited by those fragments in front in the eye path. The FTB equation, proposed for compositing of transparent fragments, computes these visibilities from the absorbances (α) of fragments. The absorbance of a fragment i determines how much this fragment is

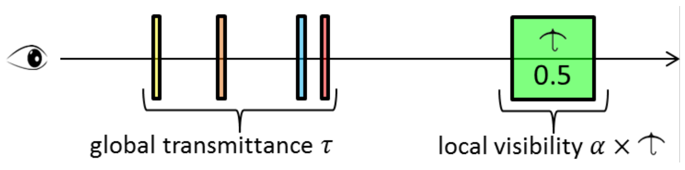

☂
0.5
global transmittance τ
local visibility $\alpha \times$ ☂

Figure 4.21: Fragment visibility weighing: the global visibility of the green fragment is the transmittance left after the accumulation of visibilities from all fragments in front of it. The local visibility of the fragment is its attribute for absorbance weighed by its coverage over the pixel area.

visible (local visibility), and how much contribution it will let pass through from back-most fragments (global visibility for those behind: $\tau_i = 1 - \alpha_i$). Similarly, the percentage of pixel area covered by a fragment (☂) can be used to weigh both local and global visibilities. For compositing of opaque and transparent fragments with partial coverage, the absorbance in the FTB equation is weighed by the coverage of the fragment:

$$_w\alpha_i = \alpha_i \times \text{☂}_i. \tag{4.7}$$

Since both α_i and ☂$_i$ values range from 0.0 to 1.0, $_w\alpha_i$ also will be in this interval, which is the same interval of the original α-weight of FTB. The diagram in Figure 4.21 shows the origins for both local and global visibility with alpha-coverage weighing.

The global visibility weighs considering coverage also derives intuitively from the original FTB. ☂ only gives us a percentage of how much pixel area the fragment is covering, we do not have the information of which portions the fragment is covering, neither its coverage relation to coverages from other fragments in the same pixel. However, we can start from the local visibility weight $_w\alpha$ and, as in the original FTB, define the transmittance through this fragment as

$$_w\tau_i = 1.0 -_w \alpha_i. \tag{4.8}$$

Therefore, $_w\tau_i$ defines how much contribution fragment i will let pass through itself from fragments behind it in view direction (see Fig. 4.21). Again, since we derived $_w\tau_i$ directly from the original FTB, it will assume the same range of valid values [0.0, 1.0]. With fragment transmittance defined, we can accumulate color contributions from a list of depth-sorted fragments as

$$C_{acc} + = C_i \times_w \alpha_i \times (1 -_w \alpha_{acc})(_w\tau), \tag{4.9}$$

and absorbance as

$$_w\alpha_{acc} + =_w \alpha_i \times (1 -_w \alpha_{acc})(_w\tau). \tag{4.10}$$

Since the coverage is already considered by the computation of the $_w\alpha$ value, its accumulation is not required.

If all edges from all triangles are dilated, not only silhouette edges, *complementary fragments* will be generated. These fragments come from different triangles, but they are contiguous in the mesh and together make a single fragment. If we compose such fragments using the approximate Equations 4.9 and 4.10, they will mistakenly reduce the visibility of their complements, that are behind in view direction. Therefore, complementary fragments must be merged before the FTB compositing, this means, combine them weighed by their respective percentage of coverage. For that, we could store information from the triangle edges and compare to find adjacent fragments. If we find a match, the face normals of the fragments must be pointing in the same direction for them to be considered complementary; otherwise, it is a false positive — e.g. the edge of a box, where one fragment is facing the camera and the other is behind, facing backwards. However, instead of storing and comparing edges, we approach the problem with a simpler approximate technique. We traverse the depth-sorted list of fragments, accumulating consecutive fragments facing the camera, while the sum of their coverages is not enough to fill one pixel. The results presented in the next section show that this approach is able to compose complementary fragments, and provide a good approximation for silhouettes.

4.3.6 Experiments and Discussion

We performed our experiments in a GTX Titan GPU (driver 332.21) and an Intel i7 (4770K) CPU running Windows 7 OS. For all approaches being compared, the fragments are stored for posterior sorting and blending. For that, we used a two-pass A-buffer implementation for GPU (MAULE et al., 2014). All implementations use OpenGL 4.4.

For the basic rendering with no AA, only fragments that cover the pixel center are generated. Each fragment stored contains depth, color, and opacity. Since all fragments have full coverage, the post processing consists of per-fragment sorting and blending. MSAA generates all fragments that cover at least one sample. Depth, color and opacity stored for the fragment come from a single sample, and a binary mask, representing coverage, is also stored. The post processing performs per-pixel sorting and per-sample blending — the list of fragments is traversed S times (S being the number of samples), accumulating covered samples, each traversal adds $1/S$ of contribution to the pixel color.

For ASAA, the attributes are also sampled at the pixel center. Besides depth, color and absorbance, ASAA also stores a value in the range [0, 1] indicating the percentage of pixel covered by the fragment, and the dot product of the normal of the primitive with the view direction. At post-processing, while sorting the list of fragments, the dot value is used to help solving depth-fighting cases: the most front-facing fragment is chosen as the closest to the viewer. After sorting, complementary fragments are accumulated before being blended, as described in Section 4.3.5; totalizing three traversals of the list of fragments.

We begin by comparing our ASAA technique against the well-established multisampling of 8 point-samples per pixel (8xMSAA) in Figure 4.22. The sampling diagrams illustrate how each technique captures and weighs fragments. Maps of captured fragments for a real scene are presented below, and the rendering results that follow show the coverage weight that each fragment received (all fragment are white and opaque). For both techniques, for a fragment to be generated by the rasterizer, the geometric primitive must cover at least one point sample. While MSAA improves fragment capture by increasing the number of point samples per pixel, ASAA uses conservative rasterization. CR dilates the primitive edges enough to guarantee that the central sample of all fragments that intersect the pixel will be generated for processing. As demonstrated in the diagrams, MSAA misses a primitive that does not cover any of its samples. We can extrapolate this example to an increasing number of MSAA samples. While quality improves, sampling cost grows and small fragments can still be undersampled. For the same primitive, ASAA discards a fragment that does not cover the circle inscribed in the pixel. This kind of loss is limited to 5.4% of the pixel area.

For weighing captured fragments, MSAA uses the ratio of covered samples to the total number of samples (S). This approach provides up to $S + 1$ shades for smoothing, causing color discontinuity between neighboring pixels with different numbers of covered samples.

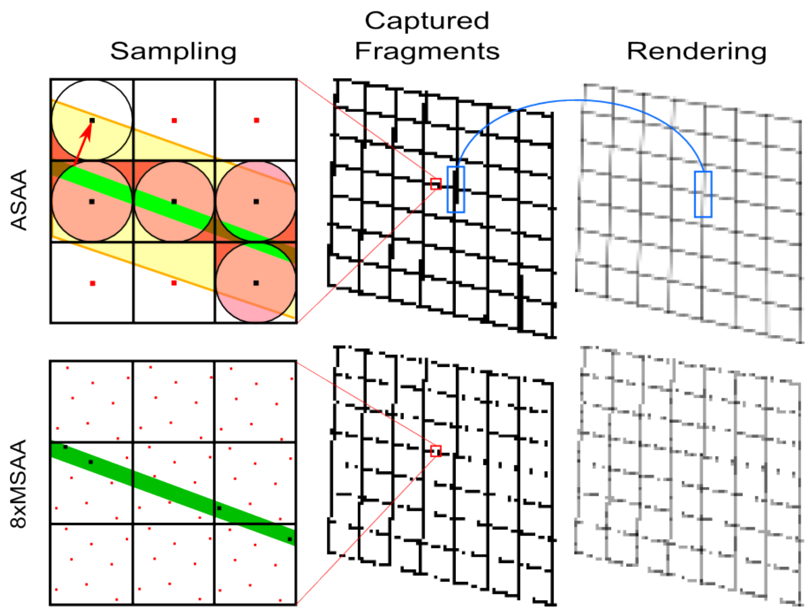


Figure 4.22: Sampling comparison: 8xMSAA versus ASAA. The sampling diagrams (left) show 3x3 pixels with their respective point samples and a green geometric primitive crossing the 3x3 pixels. The black dots indicate samples that are covered, and the red ones indicate samples that are not covered by the primitive. A pixel with only red dots does not generate any fragment—i.e., no coverage. The yellow area represents the dilated primitive, which is the one rasterized for CR. The red arrow indicates a fragment trivially discarded by a $dte < -r$ (Sec. 4.3.4.2). Red regions represent the error for the ASAA sampling approach, and the highlighted green regions represent the coverage captured by ASAA. The blue boxes indicate where contribution is splitted between pixels, generating perceptual discontinuity. ASAA can capture all fragments that contribute to the image, while 8xMSAA may miss fragments. ASAA provides superior image quality by providing smooth color variation and color continuity.

Variations in the number of covered samples, as can be seen in the horizontal lines in the rendering of Figure 4.22, may be produced by a small difference of precision when interpolating geometric primitives.

On the other hand, ASAA considers the pixel to be a circle instead of a square. With this simplification, ASAA uses the distance to the edges for estimating, with reasonable precision, the area covered by the primitive (see Sec. 4.3.4.2). This approach provides smoother color transitions than MSAA. Even so, ASAA may present perceptual discontinuity in the final image. This occurs when contribution from a thin primitive is split among two or more pixels, causing each pixel to receive very small coverage, leading to low visibility of the primitive. In the example on Figure 4.22, you can see that perceptual discontinuities in the rendering match with regions where the primitive generated more fragments in the mask of captured fragments. Notice that this issue is inherent to the discretization of the image into pixels, and not an artifact caused by ASAA sampling.

Rendering results for complex scenes, with diverse sources of geometric aliasing, are presented in Table 4.3. The power plant scene contains a series of pipes (parts of the original UNC model), with varied absorbances, which project into thin triangles in screen space. Such triangles are easily missed by point samples and, consequently, their visibility is underestimated. This scene also presents long edges, which are the most common source of aliasing, and chain-link fences that generate a complex case of aliasing known as Moiré pattern — i.e. a repetitive spatial pattern of high-frequency sampled at lower frequency. Similarly, the prison scene also presents long edges and generates thin triangles at the cell bars, which cause another example of Moiré pattern. Different from the power plant, the prison scene does not contain transparent primitives. The third scene is an illustration of four neurons and their connections. This scene combines varied absorbances with very thin and very tiny triangles, which is challenging for visibility estimation and, consequently, for color continuity. Evaluation of the techniques with respect to temporal artifacts, as flickering and popping, can be seen in the accompanying video.

Results presented in Table 4.3 correspond to 970x720 resolution. All images in this table are the direct result of rendering, and the memory consumption accounts only for fragment storage — i.e. the multisampling render-buffer object (RBO) consumption is not accounted for; that is because RBO is required for MSAA support in hardware but it is not used for image generation. Notice that the table does not compare performance measurements. Because basic rendering and 8xMSAA are implemented in hardware, they have significant performance advantages over ASAA, which is emulated with alternative tools for demonstration of its capabilities. However, despite being emulated, ASAA performs better than 8xMSAA for scenes with a very low number of triangles. We noticed an expressive performance loss of ASAA when increasing the number of triangles, which is justified by our implementation, which consists of emulating CR in the geometry shader (see Sec. 4.3.2). This indicates potential for performance improvement, just by adding CR support in the hardware rasterizer.

The maximum number of fragments per pixel (in Table 4.3) indicates how much information is available for decision-making of the pixel color. For all tested scenes, ASAA presents the largest improvements in the number of fragments being captured for processing, while its memory consumption increases proportionally. These fragments come from thin primitives and edges that point-sampling fails to capture. By capturing and compositing these contributions, ASAA provides superior image quality. For the power plant scene, the major quality impact is in the Moiré pattern caused by the yellow chain-link fence (more pronounced in the accompanying video). While 8xMSAA significantly improves over the basic rendering, it still presents the artifact, which is no longer disturbing in ASAA. A zoomed-in screenshot of this artifact is presented in Figure 4.23, which also highlights ASAA superior edge smoothing and color continuity for different parts of the power plant scene. ASAA superior quality can also be seen in the animated images provided in the supplementary material.

For the prison scene, cells can no longer be distinguished with increasing distance to the viewer. Using basic rendering, the third column of cell bars is already confusing. For the images presented in Table 4.3, the differences between ASAA and 8xMSAA are easier to spot on the enlargements in Figure 4.23. The best way to identify why ASAA is more pleasant than 8xMSAA is by trying to keep track of the holes in the cell bars (the ones used to handcuff prisoners). At the center of the MSAA image, cells become blurred regions of discontinuous colors. ASAA is able to maintain color continuity, simplifying image comprehension.

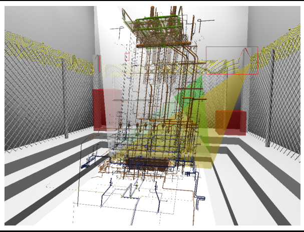
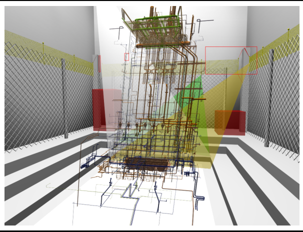
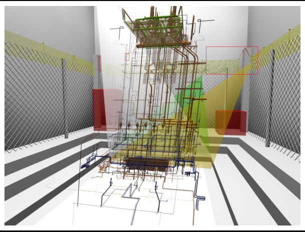
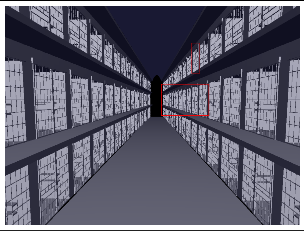
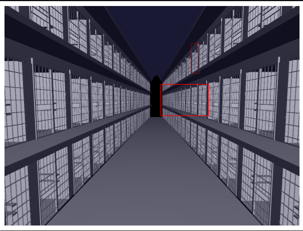
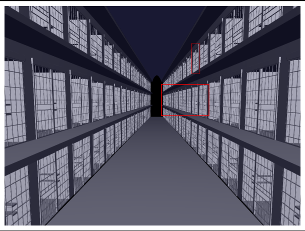
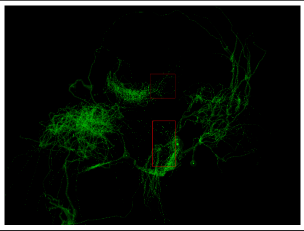
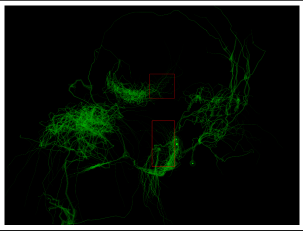
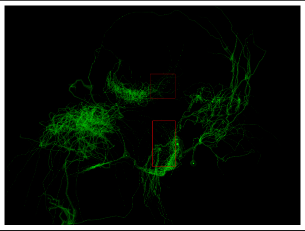
Method	**no AA**	**ASAA**	**8xMSAA**
Power Plant 496,205 t. 861,327 v.			
	1,359,606 / 41 / 31.6	3.5x / 16.78x / 3.94x	1.9x / 3x / 2.13x
Prison 151,950 t. 200,197 v.			
	3,088,660 / 27 / 64.24	1.57x / 7.3x / 2.09x	1.37x / 3.1x / 1.63x
Neurons 381,920 t. 198,000 v.			
	374,893 / 35 / 12.48	7.97x / 9.26x / 6.82x	3.17x / 3.11x / 2.82x
	total / max. / MB	**total / max. / MB**	**total / max. / MB**

Table 4.3: Feature comparison: ASAA and MSAA values are relative to the base rendering with no AA. Top row indicates to which method the column belongs. For each test scene, the first column presents the name of the scene and its geometric data. Following columns compare image quality (miniaturized images: zoom in for full resolution). Red boxes in the images indicate details enlarged in Figure 4.23. Numbers below each image indicate: the total number of fragments captured for the frame, the maximum number of fragments captured by one pixel, and memory consumption in megabytes.

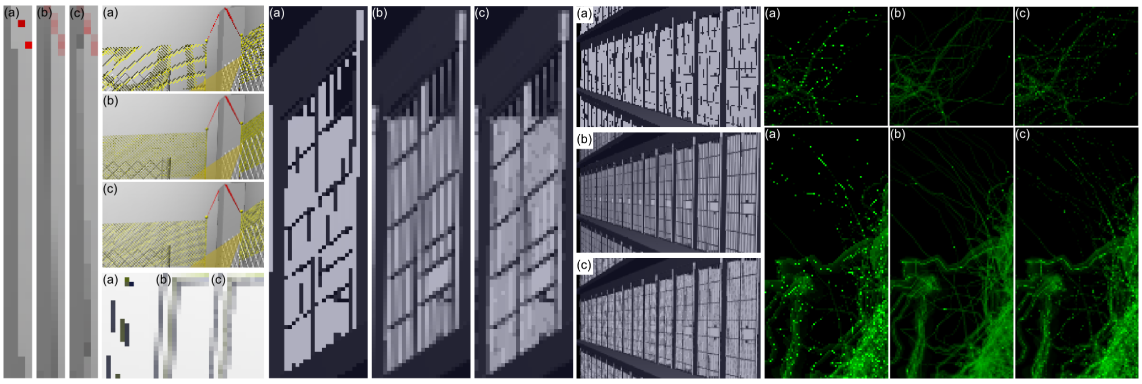


Figure 4.23: Zoomed-in details from the scenes in Table 4.3: (a) no AA, (b) ASAA, (c) 8xMSAA. Details from the power plant are on the left. First, a simple case of edge aliasing, followed by Moiré pattern and thin primitives. In the center are the details from the prison scene, comparing renderings from a single cell and from a series of cells in perspective. Details on the right are from the neurons scene. Both details compare the abilities to capture thin primitives and provide color continuity.

The neurons scene is another example of the importance of small contributions. Thus, the differences in quality are more visible in the image enlargements in Figure 4.23. By using binary coverage per pixel, the basic rendering produces an image of unreadable dotted lines. 8xMSAA significantly improves understanding of this image by capturing more fragments and, also, by providing distinct shades for primitives with different thickness in screen space. But, still, MSAA fails to capture all fragments and, with a limited number of discrete samples, under and overestimation of visibility are very frequent, leading to color discontinuity, popping and flickering (that can be seen in the accompanying video and animated images). In the image generated by ASAA, all fragments were captured and there is virtually no limit in the number of shades that a pixel can assume. These two key features provide color continuity that enables us to distinguish between structures of the scene, their format, thickness and relation to each other.

Next series of results present simpler scenes, with easily discernible features. Different resolutions were used to demonstrate how methods degenerate when there are not enough pixels to properly represent the scene, and zoomed clips are used for comparing details.

The first scene, in Table 4.4, is a TV antena. It has both thin primitives and high frequencies. For the higher resolution images, we see that ASAA results are comparable to MSAAx8. In the lower resolution images, MSAA greatly improves quality each time it doubles the number of samples, but, in this case, ASAA results are superior. When reducing the resolution, there are fewer pixels (samples) to represent the scene, making more likely for thin primitives to be missed between samples, which does not occur when using area sampling. In the zoomed snip, we see that ASAA has a smoother shading and more discernible details.

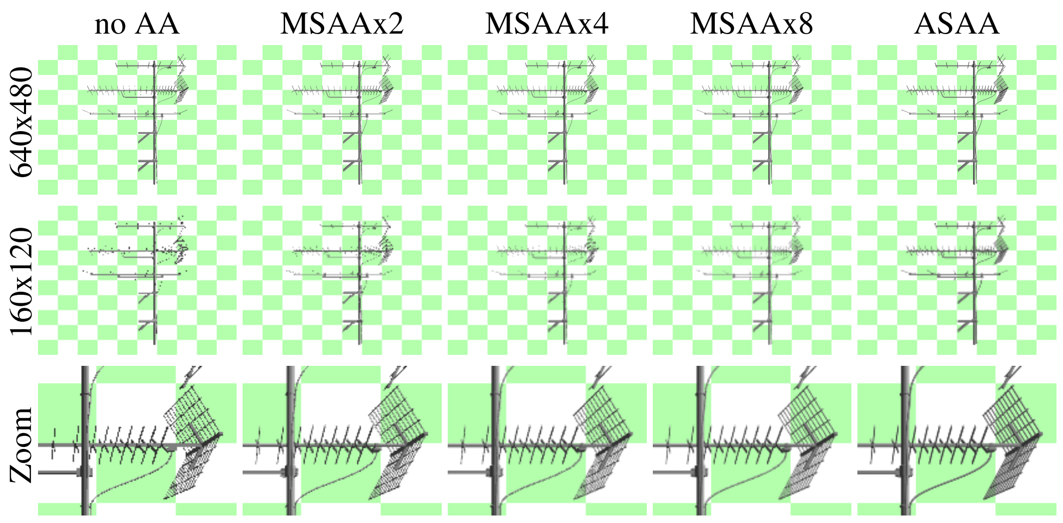


Table 4.4: Antena scene: rendering methods are denoted on the columns, and resolutions on the lines; the last line contains enlargements of part of the scene (from 640x480) (zoom in for details). ASAA superior quality is more noticeable in the lower resolution.

The second scene is a construction crane, with results presented in Table 4.5. This scene contains even thinner primitives, whose rendering is greatly improved by ASAA. However, in this scene we perceive examples of discontinuities in the primitives that are not caused by sampling, but by the contribution of the primitive being divided between pixels (see Figure 4.22 for more details on this matter). In the zoomed snip, we start to see some of the artifacts caused by currently pending issues on ASAA (dark pixels). Next scene, which contains more transparency, has more evidences about where these artifacts are coming from.

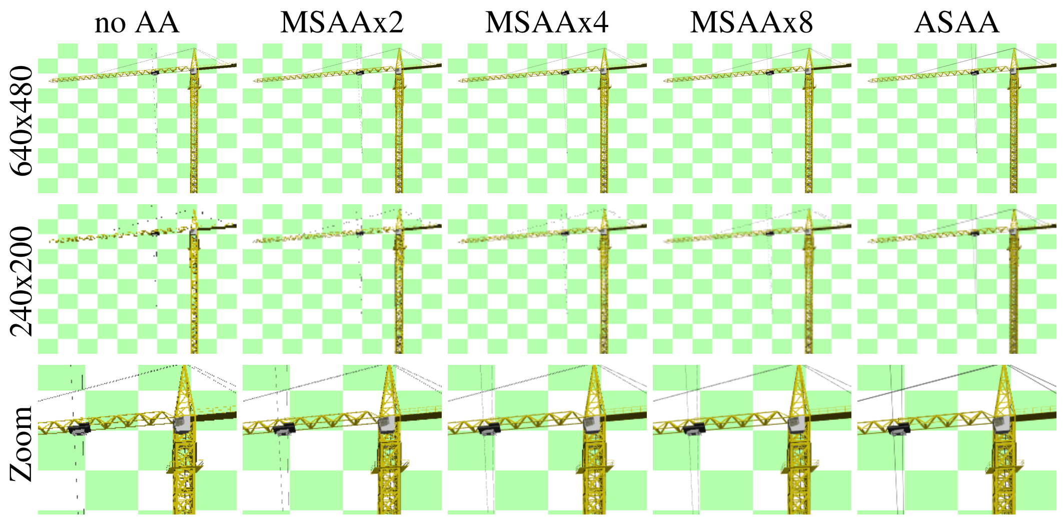


Table 4.5: Crane scene: rendering methods are denoted on the columns, and resolutions on the lines; the last line contains enlargements of part of the scene (from 640x480) (zoom in for details). Note that, besides providing better representation for thin primitives, ASAA presents some artifacts (dark pixels) from issues being investigated.

The last scene is a chess table, presented in Table 4.6. In the zoomed snip, we identify artifacts coming from two sources. When the bottom of a chess piece meets the square, we have depth fighting due to our naive approach of selecting the most front-facing fragment to be in front (so we are actually causing depth fighting in this situations). The artifacts at internal edges of the pieces meshes are more troublesome. They occur when complementary fragments are not identified, generating underestimation of visibility.

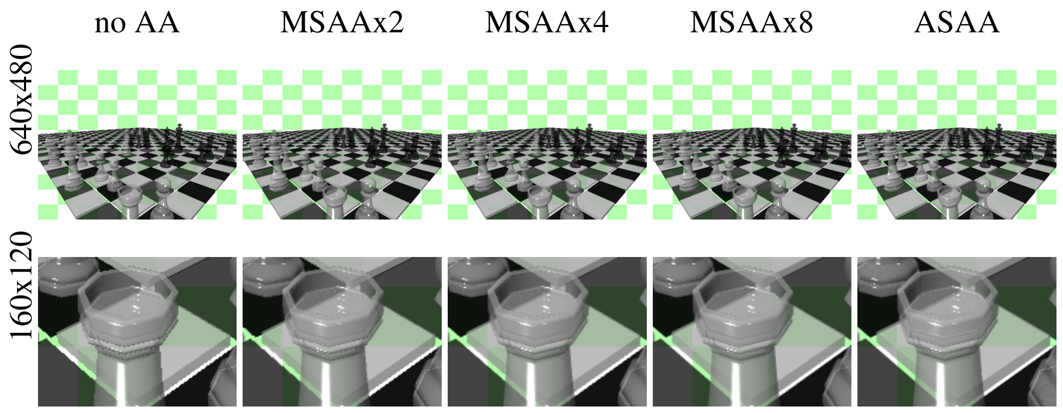


Table 4.6: Chess scene: rendering methods are denoted on the columns. The first line is for results with 640x480 pixels, and the last line contains enlargements of part of the scene (zoom in for details). ASAA presents artifacts at internal edges of the mesh, when complementary fragments from different triangles must be combined to form a single fragment.

Our investigations showed that some of the complementary fragments are missing from the list of fragments, at the post-processing stage. Further investigations indicate that some of these fragments may be generating negative coverages, which is predicted by our approach (when the triangle does not intersect the pixel) that, in response, discards the fragment. We are currently reviewing our implementations, and the theory behind them, to solve this problem.

4.4 Summary

Aliasing is the name given to artifacts that occur in computer-generated images, mainly, due to poor sampling of the virtual scene. Such artifacts greatly reduce image quality, reducing features discernment and creating uncomfortable distractions. The usage of samples from more than one primitive to compose the pixel color is what characterizes anti-aliasing as a multi-fragment effect.

Literature offers techniques classified into three main approaches: (i) full-scene AA, based on a per-pixel higher sampling rate, (ii) image post-processing AA, based on aliasing pattern detection and blur, and (iii) geometry, storing geometry information on fragments. Among these, only image post-processing approaches have full compatibility with OIT, because they are independent of the image-generation process. Most of AA techniques are focused on opaque rendering, and projected to work with the assumption that only font-most fragments are visible. Very few techniques were designed with OIT support in mind, and those who were present high memory and processing costs.

In this thesis, we propose an AA approach for minimizing memory costs in OIT rendering by using a single sample per fragment, while improving quality through an accelerated area sampling. Since at time of development there was no hardware support, we emulated our technique in GPU to validate the approach. Preliminary results show quality improvement for thin primitives and high-frequency features. Performance measurements are biased by the emulation process, but improvements on this aspect are promising with the new hardware support for conservative rasterization. This project is currently under development, with minor issues being investigated.

5 FINAL CONSIDERATIONS

Multi-fragment effects play an important role in computer graphics. Through them, we can not only improve rendering quality, but also produce more meaningful images. They are able to express relationship among virtual objects, as well as improve objects representation in a pixel display, providing more discernible shapes. By using multiple fragments, some of the global illumination effects can be captured within a graphics pipeline designed for local illumination. While global illumination takes us closer to real-world representations, the simplified pipeline in GPU is highly parallel and efficient.

However, there are challenges in implementing multi-fragment effects on GPU. They stand on three axes: performance, memory, and quality. With respect to performance, concurrency management is a key factor. Because GPUs are highly parallel, their efficiency relies in being able to execute many tasks at the same time. Resource access controls, like atomic operations and critical sections, cause major bottlenecks in the execution flow. The need to evaluate attributes for more fragments per pixel also impacts the performance negatively. Even though performance may limit the range of applications for one technique, memory is a major limiting factor for applications in general. For multi-fragment effects, partial data storage is essential for fragment shading and compositing. Efficient data structures, access patterns, and data compression strategies are important to reduce memory usage to fit within a budget. Often, correctness is sacrificed to save enough memory so the effect can be feasible. In such cases, quality is compromised. Therefore, multi-fragment techniques are developed to balance among these three factors, favoring one or other, according to the application purpose.

A variety of order-independent transparency approaches was proposed in the literature to handle the multi-fragment nature of this rendering problem. These approaches have different characteristics and priorities with respect to the three axes: performance, memory, and quality. The literature also contains several techniques that were developed for each approach, in the attempt to compensate for its deficiencies in low-priority axis. We presented in this thesis a comprehensive compendium of these techniques, classified by approach, and analyzed with respect to the three axes and algorithmic strategies. Our analysis can help choosing the most suitable technique for different applications of the OIT problem. We also presented a second classification of techniques with respect to the features they support. The set of features consists of capabilities desired in an OIT solution, as well as traits to be avoided. By grouping techniques into intersecting sets of features, we are able to highlight future pathways of research. Any intersection of desirable features that is not attended by any technique is a candidate for investigation of new techniques. Inspired by the techniques we studied, we proposed in this thesis two novel techniques.

The first technique is called dynamic fragment buffer (DFB). It is a memory-efficient algorithm to render, at interactive rates, high-quality images from scenes composed of several transparent layers. When compared to previous approaches with a fix number of fragments per pixel, DFB presents memory savings proportional to the asymmetry of layers among pixels in the frame; which, for massively transparent models, may reach hundreds of fragments. With increasing number of transparent layers and screen resolutions, a fixed number of fragments supported per pixel becomes impractical. Compared with linked lists of fragments, DFB performance advantage increases with the number of transparent layers and screen resolution. Linked lists need interactive memory allocation, random memory accesses and a single atomic operation that serializes all threads from all pixels. DFB has static memory allocation and sequential memory access per pixel, and the concurrency management bottleneck is limited to each individual pixel. Experiments results shown that DFB has the best memory usage, with the same image quality at competitive frame rates, when compared to previous techniques in the came category. Even for scenes with a large amount of geometry, like the Power Plant, DFB proved to be the best choice. The DFB extension called eDFB, with individual alpha per fragment and 8-bit color channels, fixed limitations of the original DFB and reduced memory consumption with better performance than the original DFB.

The second technique proposed is the hybrid transparency (HT). HT was a novel OIT concept that can be extended combining different techniques to balance memory consumption, performance, and image quality, the three key axes of multi-fragment rendering. The particular hybrid instance, we implemented to proving the concept, uses a truncated A-buffer (accurate) for the closest transparent layers and the weighted average technique (fast) for the remaining ones. Two unrelated and very different OIT approaches, each one with a key advantage for one of the axes; A-buffer for quality, and weighted average for performance. Both combined into a third hybrid approach that requires a low memory budget (the third axis). HT is able to handle any number of transparent layers, operating in fixed memory on current GPU technology. Our tests demonstrate the feasibility of HT, in particular, its ability to handle complex scenes and high screen resolutions, which could not be achieved by previous techniques with the same image quality. Performance results show that HT can be used in real-time applications with visually-imperceptible artifacts (no noise and no flickering). HT can be further improved: with critical sections available in GPUs, a different number of slots per pixel would be possible, thus, storing the necessary number of fragments to reach an opacity threshold, providing quality guarantees.

Recognizing the importance of image quality, we studied a second multi-fragment effect devoted to improving quality and scene representation accuracy: anti-aliasing. Not all AA approaches are committed to accuracy. In many cases, the goal is to improve only the visual appealing of the image to the human eye. For that, literature contains techniques that were developed with traits of human perception in mind, for identifying disturbing aliasing patterns and defining amelioration filters. These approaches do not recover information not sampled in the scene, they just blur the image in order to make it more pleasing to the viewer. Because sampling is expensive, these palliative techniques present a great performance advantage. However, their results are not as good as results from sampling-based approaches. That is because sample-based approaches capture and recover scene information for pixel compositing. Therefore, they can produce images with higher quality and more faithful representations of the virtual scene.

Sample-based AA approaches are not only expensive for opaque rendering, but their costs increase significantly for scenes with transparency. In contrast to image-based approaches, whose cost remains the same, sample-based approaches require more memory and more processing time to AA transparent scenes because the front-most sample is not the only one that is visible. And that applies to anti-alias techniques which support, or can be modified to support, transparency. Differently from image-based approaches, which are independent of the rendering process and can support any transparency technique, sample-based AA techniques yield superior image quality, however, they do not have a trivial mapping to transparency techniques.

Searching for a low-cost, but still high-quality, sampling-based approach for OIT, we developed the area-sampling AA technique (ASAA). This novel solution to anti-aliasing can capture all fragments from a scene while eliminating the need for more than one sample per pixel by relying on an accelerated area-sampling approach. We implemented a proof-of-concept through emulation of conservative rasterization in the geometry shader of modern GPUs. Results demonstrate superior image quality provided by our approach and additional memory consumption proportional to the number of new fragments being captured. Recent hardware support for conservative rasterization can make our solution competitive in terms of performance, and, therefore, a potential candidate for replacing multisampling as the main tool for aliasing treatment in computer-generated images. However, by the time we implemented our solution, this hardware was yet under development. ASAA quality can be further improved by, for example, treating the perception of color discontinuity that occurs when contributions from thin primitives are splitted among pixels.

The work presented in this thesis summarizes the main challenges for solving visibility for multi-fragment rendering. We identified sorting as a key element, essential to determining fragments contributions. Sorting is what causes the challenges, by implying data-dependency among fragments. These dependencies demand memory for fragments storage, until the dependencies can be solved. Therefore, adequate use of memory is the first challenge for OIT rendering. The same dependencies are responsible for performance deterioration, once they serialize fragments execution for concurrency-free memory updates, which does not suit well the massively parallel architecture of GPUs. This makes performance the second challenge to be faced by OIT techniques. However, when memory and processing time must be spared, image quality is penalized; becoming the last main challenge: sustain an acceptable level of quality while sparing resources.

All the information we gather in this thesis supplies an overview of the multi-fragment rendering problem, as well as a comprehensive view of transparency and anti-aliasing areas. Including transparency rendering techniques descriptions, classifications, and analysis, and the techniques we proposed, we have made substantial contributions to the area. Our reviews of the techniques can serve as a basis for future research, and the proposed techniques can inspire new approaches. With respect to anti-aliasing, our main contribution consists of techniques reviews, with discussions of their implementation alongside transparency. And further practical contributions are expected with our novel AA approach for OIT.

REFERENCES

ADOBE. **Photoshop**. Access date: Mar/02/2015. Available at: http://www.photoshop.com.

AMD. **EQAA Modes for AMD 6900 Series Graphics Cards**. Access date: Mar/02/2015. Available at: http://developer.amd.com/sdks/radeon/assets/EQAA%20Modes%20for%20AMD%20HD%206900%20Series%20Cards.pdf.

AMOR, M. et al. Hardware Oriented Algorithms for Rendering Order-Independent Transparency. **Comput. J.**, Oxford, UK, v.49, p.201–210, March 2006.

ANDERSSON, J. **5 Major Challenges in Real-time Rendering**. 2012.

ANDREEV, D. Directionally Localized Anti-Aliasing. **ACM SIGGRAPH Courses Presentation**, [S.l.], 2011. http://iryoku.com/aacourse/downloads/12-Anti-Aliasing-from-a-Different-Perspective-(DLAA).pptx.

BAILEY, M.; CUNNINGHAM, S. **Graphics Shaders**: theory and practice, second edition. [S.l.]: Taylor & Francis, 2011.

BATTLEFIELD. Access date: Mar/02/2015. Available at: http://international.download.nvidia.com/webassets/en_US/shared/images/guides/battlefield-3-beta/antialiasing.jpg.

BAVOIL, L. et al. Multi-fragment effects on the GPU using the k-buffer. In: I3D '07: PROCEEDINGS OF THE 2007 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, New York, NY, USA. **Anais. . .** ACM, 2007. p.97–104.

BAVOIL, L.; MYERS, K. **Order Independent Transparency with Dual Depth Peeling**. 2008.

BEETS, K. Super-sampling Anti-aliasing Analyzed. **Beyond 3D**, [S.l.], 2000. http://www.x86-secret.com/articles/divers/v5-6000/datasheets/FSAA.pdf.

BERG, M. de et al. **Computational Geometry**: algorithms and applications. 2.ed. [S.l.]: Springer-Verlag, 2000. 367p.

BRESENHAM, J. E. Algorithm for Computer Control of a Digital Plotter. **IBM Syst. J.**, Riverton, NJ, USA, v.4, n.1, p.25–30, Mar. 1965.

CALLAHAN, S. P. et al. Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering. **IEEE Transactions on Visualization and Computer Graphics**, Piscataway, NJ, USA, v.11, n.3, p.285–295, 2005.

CARPENTER, L. The A-buffer, an antialiased hidden surface method. In: SIGGRAPH '84: PROCEEDINGS OF THE 11TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES. **Anais...** [S.l.: s.n.], 1984. p.103–108.

CATMULL, E. A hidden-surface algorithm with anti-aliasing. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.12, n.3, p.6–11, Aug. 1978.

CATMULL, E. An Analytic Visible Surface Algorithm for Independent Pixel Processing. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 11., New York, NY, USA. **Proceedings...** ACM, 1984. p.109–115. (SIGGRAPH '84).

CHAJDAS, M. G.; MCGUIRE, M.; LUEBKE, D. Subpixel reconstruction antialiasing for deferred shading. In: SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES. **Anais...** [S.l.: s.n.], 2011. http://doi.acm.org/10.1145/1944745.1944748.

COLLEGE, M. **How Sound Waves Work**. Access date: Mar/02/2015. Available at: http://www.mediacollege.com/audio/01/sound-waves.html.

COZZI, P.; RICCIO, C. **OpenGL Insights**. [S.l.]: CRC Press, 2012. http://www.openglinsights.com/.

CRASSIN, C. **OpenGL 4.0+ ABuffer V2.0**: linked lists of fragment pages. Access date: Mar/02/2015. Available at: http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html.

CRASSIN, C. **Fast and Accurate Single-Pass A-Buffer using OpenGL 4.0+**. Access date: Mar/02/2015. Available at: http://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html.

CROW, F. C. The Aliasing Problem in Computer-generated Shaded Images. **Commun. ACM**, New York, NY, USA, v.20, n.11, p.799–805, Nov. 1977.

CUDA. Access date: Mar/02/2015. Available at: http://www.nvidia.com/cuda.

DEERING, M. et al. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In: COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 15., New York, NY, USA. **Proceedings...** ACM, 1988. p.21–30. (SIGGRAPH '88).

DIRECTX. Access date: Mar/02/2015. Available at: http://msdn.microsoft.com/directx.

ENDERTON, E. et al. Stochastic transparency. In: I3D '10: PROCEEDINGS OF THE 2010 ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES. **Anais...** [S.l.: s.n.], 2010. p.157–164.

EVERITT, C. **Interactive Order-Independent Transparency**. 2001.

FOLEY, J. D. et al. **Computer graphics**: principles and practice (2nd ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

FOX, M. **Optical properties of solids; 2nd ed.** Oxford: Oxford Univ. Press, 2007. (Oxford Master Series in Condensed Matter Physics).

GEGENFURTNERS, K. R.; SHARPE, L. T. **Color vision** : from genes to perception. [S.l.]: Cambridge University Press, 1999.

GOVINDARAJU, N. K. et al. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In: I3D '05: PROCEEDINGS OF THE 2005 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, New York, NY, USA. **Anais...** ACM, 2005. p.49–56.

GTA. Access date: Mar/02/2015. Available at: http://gtaforums.com/topic/514428-will-v-have-anti-aliasing/.

HASSELGREN, J.; AKENINE-MöLLER, T.; OHLSSON, L. Conservative Rasterization. In: PHARR, M.; FERNANDO, R. (Ed.). **GPU Gems 2**: programming techniques for high-performance graphics and general-purpose computation. [S.l.]: Addison-Wesley Professional, 2005.

HEARN, D.; BAKER, P. **Computer Graphics, C Version**. [S.l.]: Prentice Hall, 1997. (Prentice-Hall international editions).

HECHT, E. **Optics; 4th ed.** [S.l.]: Addison Wesley, 2002.

IOURCHA, K.; YANG, J. C.; POMIANOWSKI, A. A directionally adaptive edge anti-aliasing filter. In: CONFERENCE ON HIGH PERFORMANCE GRAPHICS. **Proceedings...** [S.l.: s.n.], 2009.

JENSEN, H. W. et al. A practical model for subsurface light transport. In: COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 28., New York, NY, USA. **Proceedings...** ACM, 2001. p.511–518. (SIGGRAPH '01).

JIMENEZ, J. et al. Filtering Approaches for Real-Time Anti-Aliasing. In: ACM SIGGRAPH COURSES. **Anais...** [S.l.: s.n.], 2011.

JIMENEZ, J. et al. SMAA: enhanced morphological antialiasing. **Computer Graphics Forum (Proc. EUROGRAPHICS 2012)**, [S.l.], 2012.

JOUPPI, N. P.; CHANG, C.-F. Z3: an economical hardware technique for high-quality antialiasing and transparency. In: HWWS '99: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS WORKSHOP ON GRAPHICS HARDWARE. **Anais...** [S.l.: s.n.], 1999. p.85–93.

KAJIYA, J. T. The rendering equation. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.20, n.4, p.143–150, Aug. 1986.

LIU, F. et al. Efficient depth peeling via bucket sort. In: HPG '09: PROCEEDINGS OF THE CONFERENCE ON HIGH PERFORMANCE GRAPHICS 2009. **Anais...** [S.l.: s.n.], 2009. p.51–57.

LIU, F. et al. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In: I3D '10: PROCEEDINGS OF THE 2010 ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES. **Anais. . .** [S.l.: s.n.], 2010. p.75–82.

LOTTES, T. **FXAA**. Access date: Mar/02/2015. Available at: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

MALAN, H. Distance-to-edge Anti-Aliasing. **ACM SIGGRAPH Courses Presentation**, [S.l.], 2011. http://iryoku.com/aacourse/downloads/10-Distance-to-edge-AA-(DEAA).pptx.

MAMMEN, A. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. **IEEE Comput. Graph. Appl.**, [S.l.], v.9, n.4, p.43–55, 1989.

MARK, W. R.; PROUDFOOT, K. The F-buffer: a rasterization-order fifo buffer for multi-pass rendering. In: HWWS '01: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS WORKSHOP ON GRAPHICS HARDWARE. **Anais. . .** [S.l.: s.n.], 2001. p.57–64.

MAULE, M. et al. A survey of raster-based transparency techniques. **Computers & Graphics**, [S.l.], v.35, n.6, p.1023 – 1034, 2011.

MAULE, M. et al. Transparency and Anti-Aliasing Techniques for Real-Time Rendering. In: SIBGRAPI CONFERENCE ON GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI), 2012, 25. **Anais. . .** [S.l.: s.n.], 2012.

MAULE, M. et al. Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer. In: SIBGRAPI CONFERENCE ON GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI), 2012, 25. **Anais. . .** [S.l.: s.n.], 2012.

MAULE, M. et al. Hybrid Transparency. In: ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES. **Anais. . .** [S.l.: s.n.], 2013.

MAULE, M. et al. Memory-optimized order-independent transparency with Dynamic Fragment Buffer. **Computers & Graphics**, [S.l.], 2014.

MCGUIRE, M.; BAVOIL, L. Weighted Blended Order-Independent Transparency. **Journal of Computer Graphics Techniques (JCGT)**, [S.l.], v.2, n.2, p.122–141, December 2013.

MESHKIN, H. **Sort-Independent Alpha Blending**. 2007.

MICROSOFT. **OIT11 Sample**. Access date: Mar/02/2015. Available at: http://msdn.microsoft.com/en-us/library/ee416572%28v=VS.85%29.aspx.

MOIRE Animations. Access date: Mar/02/2015. Available at: https://www.youtube.com/watch?v=0SLmE3-Zg94.

MOIRE. Access date: Mar/02/2015. Available at: http://en.wikipedia.org/wiki/Aliasing.

MULDER, J. D.; GROEN, F. C. A.; WIJK, J. J. van. Pixel masks for screen-door transparency. In: VISUALIZATION '98, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 1998. p.351–358. (VIS '98).

MYERS, K.; BAVOIL, L. Stencil routed A-Buffer. In: SIGGRAPH '07: ACM SIGGRAPH 2007 SKETCHES. **Anais...** [S.l.: s.n.], 2007. p.21.

NICKOLLS, J.; KIRK, D. Graphics and Cumputing GPU. In: PATTERSON, D. A.; HENNESSY, J. L. (Ed.). **Computer Organization and Design**: the hardware/software interface. 4.ed. [S.l.]: Morgan Kaufmann, 2009.

NVIDIA. Antialiasing with Transparency. **Technical Report**, [S.l.], 2004. http://lala.com.

NVIDIA. **Coverage Sampled Antialiasing**. Access date: Mar/02/2015. Available at: http://developer.download.nvidia.com/SDK/10/direct3d/Source/CSAATutorial/doc/CSAATutorial.pdf.

NVIDIA. **Thrust**. Access date: Mar/02/2015. Available at: http://code.google.com/p/thrust.

OPENCL. Access date: Mar/02/2015. Available at: https://www.khronos.org/opencl.

OPENGL. Access date: Mar/02/2015. Available at: https://www.opengl.org.

PEDROTTI, F. L.; PEDROTTI, L. S. **Introduction to optics; 2nd ed.** [S.l.]: Prentice-Hall, 1993.

PERSPECTIVE MoirÉ. Access date: Mar/02/2015. Available at: http://bigwww.epfl.ch/research/projects/all.html.

PERSSON, E. Geometry Buffer Antialiasing. **ACM SIGGRAPH Courses Presentation**, [S.l.], 2011. http://iryoku.com/aacourse/downloads/11-Geometry-Buffer-Anti-Aliasing-(GBAA).pptx.

PHARR, M. **Chapter 25 of GPU Gems**: fast filter-width estimates with texture maps. [S.l.]: Pearson Higher Education, 2004.

PORTER, T.; DUFF, T. Compositing digital images. **SIGGRAPH Comput. Graph.**, [S.l.], v.18, n.3, p.253–259, 1984.

RESHETOV, A. Morphological Antialiasing. **Intal Labs**, [S.l.], 2009. http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf.

RESHETOV, A. Reducing Aliasing Artifacts Through Resampling. In: FOURTH ACM SIGGRAPH / EUROGRAPHICS CONFERENCE ON HIGH-PERFORMANCE GRAPHICS, Aire-la-Ville, Switzerland, Switzerland. **Proceedings...** Eurographics Association, 2012. p.77–86. (EGGH-HPG'12).

SALVI, M.; MONTGOMERY, J.; LEFOHN, A. Adaptive transparency. In: ACM SIGGRAPH SYMPOSIUM ON HIGH PERFORMANCE GRAPHICS, New York, NY, USA. **Proceedings...** ACM, 2011. p.119–126. (HPG '11).

SALVI, M.; VAIDYANATHAN, K. Multi-layer Alpha Blending. In: MEETING OF THE ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, 18., New York, NY, USA. **Proceedings. . .** ACM, 2014. p.151–158. (I3D '14).

SAMPLING. Access date: Mar/02/2015. Available at: http://zone.ni.com/reference/en-XX/help/370051V-01/cvi/libref/analysisconcepts/aliasing/.

SEGAL, M.; AKELEY, K. **The OpenGL Graphics System**: a specification (core profile 4.4). [S.l.: s.n.], 2014.

SEN, O.; CHEMUDUGUNTA, C.; GOPI, M. Silhouette-Opaque Transparency Rendering. In: COMPUTER GRAPHICS AND IMAGING. **Anais. . .** [S.l.: s.n.], 2003. p.153–158.

SHIRLEY, P.; MORLEY, R. K. **Realistic Ray Tracing**. Natick, MA, USA: A. K. Peters, Ltd., 2003.

SHREINER, D. et al. **OpenGL Programming Guide**: the official guide to learning opengl, version 4.3. 8th.ed. [S.l.]: Addison-Wesley Professional, 2013.

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, New York, NY, USA, v.23, n.6, p.343–349, June 1980.

WILLIAMS, L. Casting Curved Shadows on Curved Surfaces. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 5., New York, NY, USA. **Proceedings. . .** ACM, 1978. p.270–274. (SIGGRAPH '78).

WITTENBRINK, C. M. R-buffer: a pointerless a-buffer hardware architecture. In: HWWS '01: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS WORKSHOP ON GRAPHICS HARDWARE. **Anais. . .** [S.l.: s.n.], 2001. p.73–80.

WOLIGROSKI, D. **Coverage Sampling Modes**: nvidias csaa and amds eqaa. Access date: Mar/02/2015. Available at: http://www.tomshardware.com/reviews/anti-aliasing-nvidia-geforce-amd-radeon,2868-4.html.

WRIGHT, R. S. et al. **OpenGL SuperBible**: comprehensive tutorial and reference. 5th.ed. [S.l.]: Addison-Wesley Professional, 2010.

YANG, J. C. et al. Real-time Concurrent Linked List Construction on the GPU. In: EUROGRAPHICS CONFERENCE ON RENDERING, 21., Aire-la-Ville, Switzerland, Switzerland. **Proceedings. . .** Eurographics Association, 2010. p.1297–1304. (EGSR'10).