Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA INFORMÁTICA

Hugo Ventura

Simulação da Profundidade de Campo

Orientador:
Prof. Doutor Manuel Próspero dos Santos

20 de Fevereiro de 2009

Nº do aluno: 26378
Nome: Hugo Miguel Pinto Ventura

Título da Dissertação:
Simulação da Profundidade de Campo

Palavras-Chave:

- profundidade de campo
- pós-processamento
- tempo-real
- *shaders*
- GPU
- *Frame Buffer Objects*

Keywords:

- depth of field
- post-processing
- real-time
- shaders
- GPU
- Frame Buffer Objects

Agradecimentos

Acima de tudo, gostava de agradecer à minha namorada por todo o apoio que me deu durante a realização desta Dissertação, onde mesmo não percebendo nada do que fazia ou falava, sempre me ajudou como podia, sempre presente a ouvir-me e a dar-me força, motivando-me até ao fim. Enfim, esteve sempre lá quando precisei. Carla, este trabalho é para ti!

Agradeço à minha família pois sem ela não estaria certamente aqui a escrever este agradecimento nesta Dissertação. Obrigado por estarem sempre lá e me proporcionarem, até hoje, tudo o que preciso para eu chegar onde eu quiser. O resto depende de mim, não é?

Um muito obrigado ao meu orientador, o professor Próspero dos Santos. Em primeiro pela oportunidade que me deu de elaborar uma tese de mestrado nesta área que é a Computação Gráfica, na qual tenho particular fascínio. Em segundo por todo apoio oferecido de forma a conseguir atingir os objectivos propostos. E por fim um muito obrigado pela pessoa gentil que é, ri-me mesmo muito durante as nossas conversas.

Um grande obrigado a todos os meus colegas com quem partilharam comigo esta inesquecível viagem universitária. Mencioná-los seria uma injustiça, são todos uns porreiros, sem excepção. E durante todos estes anos, mais que colegas de faculdade, foram realmente amigos sempre que precisei. Força pessoal!

Por fim, gostava de agradecer, no âmbito deste trabalho, à senhora Tianshu Zhou, e aos senhores Sungkil Lee e Angelo Theodorou pela disponibilidade evidenciada no esclarecimento de aspectos relativos a técnicas ou tecnologias de programação gráfica utilizadas. O meu obrigado também ao senhor Tudor Carean por facultar a sua aplicação a ser integrada nesta Dissertação.

Resumo

A renderização de imagens realistas é um objectivo da Computação Gráfica (CG) e, tal como em fotografia, a profundidade de campo apresenta-se como uma das suas componentes mais importantes. Com esta é proporcionada uma noção de profundidade capaz de auxiliar o sistema visual humano a interpretar correctamente os elementos de uma cena tridimensional.

Foram já propostos vários métodos para modelação de profundidade de campo mas, devido à evolução tecnológica assistida em CG superando limitações passadas, só hoje é possível introduzir correctamente o efeito de profundidade em programas interactivos como os de Realidade Virtual.

Esta Dissertação de Mestrado em Engenharia Informática visa a elaboração de uma aplicação interactiva, simuladora de uma correcta profundidade de campo em tempo-real aproximado. Conjuntamente será definida a associada metodologia, a ser facilmente integrada por um qualquer programador de aplicações gráficas.

É assim apresentado um estudo relativo a técnicas recentes para criação do efeito realístico de profundidade baseadas no método de Pós-Processamento, de modo a seleccionar e utilizar aquela que mais se enquadra nos objectivos definidos.

Numa simulação gráfica de visão realista é usual o aparecimento de incorrecções visuais que poderão facilmente interferir com a percepção do observador. Com o intuito de ser realizada uma modelação precisa da profundidade de campo, estas incoerências visuais são objecto de uma análise aprofundada, culminada na determinação das suas possíveis soluções.

De forma a garantir simulação com tempo de execução apropriado à interactividade exigida são consideradas optimizações de diversa ordem, entre as quais, o uso das potencialidades de hardware gráfico. Aproveita-se o paralelismo em GPU através de linguagem de *shading* GLSL, elevando o desempenho global quando manipulados pixels em grande número.

Abstract

The rendering of realistic images is an objective of Computer Graphics (CG) and, as in photography, the depth of field introduces itself as one of their most important components. With this a depth notion is provided, capable of helping the human visual system to properly decode the elements of a three-dimensional scene.

Several methods have been proposed to model depth of field but, because of the evident technological evolution in CG overcoming past limitations, only today it has become possible the insertion of a correct depth effect in interactive programs like the ones of Virtual Reality.

This Master Thesis, in Computer Engineering, targets the elaboration of an interactive application, which simulates an accurate depth of field in real time. At the same time an associated methodology will be defined, to be easily used by any graphics application programmer.

Therefore a survey of realistic depth effect rendering techniques, founded on the Post-Processing method, is presented in order to select and use the one that better suits all previous defined objectives.

In realistic graphics simulations of human vision is usual the appearance of incorrect visual cues that can simply disturb the observer's perception. With the intention to create a precise depth of field modeling, these visual incoherencies are object of a further deep analysis, culminating in the determination of its respective possible solutions.

To ensure a depth of field simulation with execution time suitable of the demanding interactivity, optimizations of several orders are considered, like taking advantage of graphics hardware potentialities. GPU processing parallelism is exploited using the GLSL shading language, allowing global performance enhancement when high number of pixels is manipulated.

Índice

1. Introdução 1

1.1 Introdução Geral e Motivação 1

1.2 Descrição e Contexto 2

1.3 Solução Apresentada 4

1.4 Abordagem e Principais Contribuições Previstas 5

2. Estado da Arte 7

2.1 Os Conceitos da Profundidade de Campo 7

2.2 Modelos de Câmara 10

2.2.1 Modelo *Pinhole* 11

2.2.2 Modelo *Thin Lens* 13

2.2.3 Modelo *Thick Lens* 16

2.2.4 Modelo *Geometric Lens* 17

2.2.5 Na Simulação de Profundidade de Campo 17

2.3 Principais Tipos de Técnicas 19

2.3.1 Multi-Passo 20

2.3.1.1 *Ray Tracing* Distribuído 20

2.3.1.2 *Accumulation-Buffer* 21

2.3.2 Pós-Processamento 22

2.3.3 Profundidade por Camadas 23

2.3.4 Análise Comparativa 25

2.4 Técnicas de Profundidade de Campo em Pós-Processamento 26

2.4.1 Cinco Técnicas Pós-Processamento 26

2.4.1.1 Rápida Renderização de Profundidade de Campo com *Surface Splatting* 26

2.4.1.2 Correcta Profundidade de Campo Através de Difusão Anisotrópica 28

2.4.1.3 Profundidade Interactiva por Difusão Simulada em GPU 29

2.4.1.4 Prática Profundidade de Campo por Pós-Processamento 32

2.4.1.5 Exacta Simulação de Profundidade de Campo em Tempo-real 33

2.4.2 Outras 35

2.4.3 Análise Comparativa 36

2.5 Artefactos Visuais Comuns em Pós-Processamento e Soluções 37
2.5.1 Problemas de Oclusão 38
2.5.2 Soluções 39

3. Programação em GPU 42

3.1 Importância da Execução em GPU em Simulações Visuais 43
3.2 *Shading* por GLSL 44
3.2.1 *Shaders*, Razões e Vantagens 44
3.2.2 GLSL 46
3.2.2.1 *Pipeline* Gráfico 47
3.2.2.2 Processador *Vertex* 49
3.2.2.3 Processador *Fragment* 50
3.3 CUDA 51
3.4 Compute Shader 53
3.5 Conclusão 55

4. Implementação da Profundidade de Campo 57

4.1 A Metodologia 57
4.1.1 Filtragem de Imagens 58
4.1.2 Filtro de Passagem Dupla 61
4.2 OpenGL 64
4.2.1 Frame Buffer Object 65
4.3 Fluxo de Processamento e Integração com OpenGL 67
4.3.1 Configurações 69
4.3.1.1 Inicialização de Objectos FBO 70
4.3.1.2 Inicialização de *Shaders* 72
4.3.2 Renderização *Offscreen* 75
4.3.3 Pós-Processamento 77
4.3.3.1 *Vertex Shader* 79
4.3.3.2 *Fragment Shader* 79
4.4 Optimizações 82
4.5 Interface 84

5. Testes e Resultados 89

5.1 Qualidade de Renderização 90

5.2 Desempenho de Renderização 95

5.3 Simulação em Ambiente Virtual Animado 97

5.4 Integração com Aplicações Gráficas 98

6. Conclusão 101

6.1 Trabalho Realizado 101

6.2 Contribuições 102

6.3 Trabalho Futuro 104

7. Posfácio 108

8. Bibliografia 109

8.1 Leitura Aconselhada 111

Índice de Figuras

Figura 1 – Exemplo de Profundidade de Campo 7
Figura 2 – Utilização do efeito de profundidade em cinematografia..... 8
Figura 3 – Imagem de filme *Ratatouille* da Pixar..... 9
Figura 4 – Exemplo de Profundidade de Campo de jogo *Call of Duty 4* 10
Figura 5 – Modelo de câmara *Pinhole*. Cena tridimensional gerada computacionalmente e fotografia captada utilizando sistema óptico *Pinhole* 11
Figura 6 – Princípio do modelo de câmara *Pinhole*..... 12
Figura 7 – Geografia de um sistema óptico *Thin Lens* 14
Figura 8 – Círculo de Confusão quando objecto desfocado 15
Figura 9 – Princípio da técnica *Ray Tracing* 21
Figura 10 – Imagem renderizada por *Ray Tracing* 21
Figura 11 – Profundidade de Campo e transparência 27
Figura 12 – Simulação de profundidade de campo por difusão de calor. Plano focado bastante afastado e plano focado bastante próximo..... 30
Figura 13 – Efeito *bokeh* em fotografia 31
Figura 14 – Cena 3D com efeito de profundidade de campo 34
Figura 15 – Algoritmo de detecção de arestas identifica onde se posicionam os limites dos objectos na cena..... 40
Figura 16 – Processamento interno da técnica da diferença de pixels adjacentes..... 41
Figura 17 – Comparação de operações Floating-Point por segundo em CPU e GPU..... 42
Figura 18 - *Pipeline* de processamento gráfico do OpenGL e integração dos processadores *Vertex* e *Fragment* 47
Figura 19 - *Pipeline* de processamento gráfico do OpenGL com alto nível de abstracção..... 47
Figura 20 - Organização de memória do GPU e multiprocessadores, usando CUDA 51
Figura 21 - a) Modelo de programação e execução do CUDA. b) Modelo de memória interna do CUDA 52
Figura 22 – Integração do *Compute Shader* no normal fluxo de processamento gráfico..... 54

Figura 23 – Aplicações alvo do *Compute Shader* da Microsoft 54
Figura 24 – Convolução em *Compute Shader* e sua performance prevista 55
Figura 25 – Filtragem por Convolução 59
Figura 26 – Processo de Convolução 59
Figura 27 – Exemplos de máscaras aplicadas em Convolução 60
Figura 28 – Contribuição de pixel P para o pixel com centro em C, segundo diferentes círculos de confusão 62
Figura 29 – Arquitectura *Frame Buffer Object* 66
Figura 30 – Fluxo de processamento da simulação de profundidade de campo 68
Figura 31 – Formato de um pixel de textura 72
Figura 32 – Configuração de *shaders*. a) Criação e compilação de shaders. b) Criação de Programas com anexo de *shaders* 73
Figura 33 – Processamento do filtro de passagem dupla. Amostragem de pixels vizinhos ao pixel a calcular (central). a) Primeira passagem do filtro, amostragem horizontal. b) Primeira passagem do filtro, amostragem vertical 80
Figura 34 – Interface gráfica da simulação de profundidade de campo 85
Figura 35 – Informação de desempenho mostrada na barra de título da aplicação 85
Figura 36 – Janela de controlo da simulação visual 86
Figura 37 – Mapa de processamento de filtragem 87
Figura 38 – Cenas tridimensionais. a) Figuras. b) Sala de Cubos. c) Mundo das Esferas 89
Figura 39 – Resultados visuais da simulação da profundidade de campo 91
Figura 40 – Artefacto visual inerente à metodologia apresentada, utilizando valores extremos das propriedades ópticas 92
Figura 41 – Amostragem circular de pixels vizinhos do pixel a calcular no *shader* 93
Figura 42 – Resultado visual da simulação da profundidade de campo com filtro de uma só passagem e amostragem circular 94
Figura 43 – *Tradeoff* entre o tamanho do filtro e desempenho global da simulação 97
Figura 44 – Integração da profundidade de campo numa aplicação OpenGL 100
Figura 45 – Evolução da capacidade de processamento do GPU, sendo assinalado o pico de desempenho aproximado do GPU utilizado neste trabalho, NVIDIA GeForce 8500 GT 104
Figura 46 – Imagem comparativa da aplicação Google Earth, à esquerda sem e à direita com filtragem anisotrópica 107

Índice de Tabelas

Tabela 1 – Desempenho em FPS segundo uma resolução R e N iterações 28

Tabela 2 – Desempenho em FPS segundo diferentes resoluções 31

Tabela 3 – Desempenho em milissegundos segundo diferentes resoluções 34

Tabela 4 – Desempenho de profundidade de campo da cena “Figuras” 95

Tabela 5 – Desempenho de profundidade de campo da cena “Sala de Cubos” 95

Tabela 6 – Desempenho de profundidade de campo da cena “Mundo das Esferas” 96

Índice de Equações

Equação 1 – Lei das Lentes 14

Equação 2 – Diâmetro de Círculo de Confusão 15

Equação 3 – Rácio referente à abertura óptica..... 15

Equação 4 – Distância hiperfocal 15

Equação 5 – Plano mais próximo delimitador de espaço projectado de forma nítida 16

Equação 6 – Plano mais afastado delimitador de espaço projectado de forma nítida 16

Equação 7 – Fórmula matemática para Convolução..... 59

Equação 8 – Distribuição Gaussiana 1D..... 60

Equação 9 – Separabilidade da Distribuição Gaussiana 2D 61

Equação 10 – Função de sobreposição para cálculo de peso a aplicar sobre cada pixel 62

Equação 11 – Função de intensidade luminosa para cálculo do peso a aplicar sobre cada pixel. 62

Equação 12 – Função de controlo de *intensity leakage* sobre cada pixel 63

Equação 13 – Contribuição do pixel (central) a calcular no *shader* 63

Equação 14 – Função de cálculo da contribuição de cada pixel amostrado segundo a contribuição de três factores 63

1. Introdução

1.1 Introdução Geral e Motivação

O fantástico avanço das tecnologias computacionais possibilita aplicações cada vez mais avançadas em todos os domínios da Informática. A Computação Gráfica (CG) não é excepção e, hoje, com o aumento das capacidades gráficas, habituámo-nos a ver impressionantes e superiores renderizações 3D em tempo-real, com o propósito único de produzir imagens tão realistas como as captadas pela nossa visão. Em CG, os progressos mais evidentes denotam-se principalmente na cinematografia (animações e efeitos especiais), nos jogos computorizados, Realidade Virtual (RV) ou mesmo Realidade Aumentada (RA). A sua razão recai no proporcional aumento da exigência científica na obtenção de gráficos computacionais mais realistas, em conjunto com a necessidade de uma constante inovação na comercialização de melhores e originais produtos de entretenimento, tão específicos como filmes ou videojogos.

Dentro dos infindáveis desenvolvimentos em CG a serem realizados, com o objectivo acima descrito, inclui-se a criação do efeito de profundidade de campo. Trata-se de uma componente natural da visão humana e das câmaras reais (filme e fotografia) onde, a partir de um ponto de visualização, uma zona focada manifesta total nitidez facultando uma perfeita definição de contornos dos seus objectos. Zonas que não a focada vão perdendo nitidez consoante a sua distância à mesma zona focada.

A simulação de profundidade de campo, objecto desta Dissertação, apresenta-se como um excelente contributo na área de CG com imensas hipóteses de aplicação. Não permite apenas um maior realismo de simples imagens totalmente geradas por computador, mas também a valorização de outras aplicações de outros domínios como videojogos ou Realidade Virtual. A qualidade de aplicações gráficas interactivas, nestes domínios ou em outro qualquer que

apresente ambientes virtuais computorizados, será facilmente enaltecida ao elevar a experiência vivida nos respectivos universos.

Os videojogos terão uma superior jogabilidade e, consequentemente, um valor global maior pela simples adição do efeito de profundidade, capaz de proporcionar aos jogadores um sentimento muito semelhante ao adquirido durante a visualização de uma obra cinematográfica.

Da mesma forma, com uma correcta noção de profundidade, muito beneficiarão aplicações de Realidade Virtual, área tão importante para o desenvolvimento da Computação Gráfica. O objectivo desta tecnologia é recriar a sensação de realidade ao seu utilizador num mundo totalmente virtual. Existem hoje aplicações diversas da RV em áreas tão distintas como jogos e entretenimento, simulação aérea, arquitectura e urbanismo, treino militar, entre outros, não havendo limites no mundo ou na experiência a simular. A realidade em RV será impossível de atingir caso, num ambiente totalmente virtual, não exista nenhuma possibilidade de percepção sobre as distâncias dos objectos em seu redor. A profundidade de campo em RV dará a naturalidade necessária numa simples interacção do utilizador com o universo virtual onde está inserido.

1.2 Descrição e Contexto

Uma cena sintética tridimensional, simplesmente gerada por computador, tem a particularidade de possuir uma profundidade de campo infinita e assim evidenciar todos os seus objectos perfeitamente detalhados, situação que comporta por si só um enorme irrealismo visual. Para dar o aspecto real a uma imagem sintética, esta deve ser aproximada o mais possível à imagem que um ser humano captaria com os seus próprios olhos, ou com câmaras ópticas, se o mundo virtual fosse na verdade real. Ora uma das noções mais relevantes para conceder realismo a uma imagem gráfica é a de profundidade, existente no processo natural da visualização humana quando, numa qualquer região do espaço a visão é centralizada, apresentando-se a área focada com aspecto nítido, enquanto as restantes, na mesma direcção mas com diferentes distâncias de afastamento, se exibem desfocadas. Na verdade, a acção constante num sistema óptico não é

mais do que a de atribuir importância aos pontos realmente focalizados, retirando importância a todos os pontos em redor. Com isto é sintetizada a razão fundamental pela qual nasceu o problema da profundidade de campo em CG, e a partir deste ponto a questão seria como inserir efectivamente o aspecto real, que a noção de profundidade proporciona, a gráficos computacionais.

A profundidade de campo foi conseguida em aplicações gráficas segundo métodos variados, dos quais sobressaem algumas implementações como, por exemplo, *Ray Tracing* transportando todas as propriedades dos raios de luz e capaz de produzir imagens com enorme grau de foto-realismo. Igualmente conhecidas são as implementações *Accumulation Buffer* por justaposição de várias imagens captadas de pontos ligeiramente diferentes, ou as denominadas Pós-Processamento por aplicação de um filtro sobre a imagem original, segundo um modelo de câmara de lente e abertura óptica. Ao longo dos anos foram realizadas técnicas de natureza diversa, capazes de criar profundidade de campo sobre imagens computorizadas desprovidas de realismo, mas praticamente todas patenteavam um *tradeoff* entre qualidade e rapidez de formação do resultado final.

Os avanços tecnológicos no mundo informático, e especificamente na área de CG, levaram à exigência prática de funcionalidades interactivas nas suas aplicações, apenas eficientes se aproximados os respectivos tempos de processamento aos de tempo-real. Também as técnicas de implementação da profundidade de campo, até então pouco utilizadas em aplicações gráficas devido ao seu imenso custo computacional, tiveram de evoluir e moldar-se às novas exigências. Dos tipos de implementações atrás mencionados, foram as de Pós-Processamento que melhor conseguiram elevar a rapidez de execução na modelação de profundidade de campo, adaptando-se da melhor forma à contínua evolução do hardware gráfico.

Com isto somos levados ao encontro dos objectivos capitais da presente tese. O resultado prático da elaboração desta Dissertação passa pela simulação de profundidade de campo interactiva, através dos recursos actualmente existentes na computação de gráficos computorizados. E ao mesmo tempo, apresentar uma metodologia que possa ser utilizada por um qualquer programador de aplicações, em que a profundidade de campo seja um agente absolutamente necessário. O resultado desta Dissertação, atingidos os objectivos propostos, será

apresentado como um desenvolvimento de uma dada técnica já existente, acompanhada pela respectiva metodologia detalhada. Será possibilitada então a sua implementação em uma qualquer aplicação gráfica, que queira oferecer resultados visuais aproximados ao foto-realismo. O objectivo final no desenvolvimento da profundidade de campo será o de aproximar, igualmente à realidade, os tempos em que é apresentado o efeito de profundidade em programas com intensa interactividade e exibição de resultados em tempo-real.

1.3 Solução Apresentada

No sentido de resolver o problema descrito acima, é necessário dominar todos os aspectos associados a uma simulação visual, como é o caso da profundidade de campo sobre uma cena sintética. Torna-se imprescindível, depois de compreendido o conceito de Profundidade de Campo na sua plenitude, fazer um levantamento dos tipos de técnicas simuladoras do efeito, entender os respectivos tipos de processamento e analisar seus benefícios e condicionantes.

Segundo a própria evolução deste tema ao longo dos anos, o desenvolvimento verificou-se sobretudo nas técnicas Pós-Processamento, facto que será devidamente analisado e aclarado. A elaboração desta Dissertação acompanhará assim a evolução verificada no domínio da profundidade de campo em CG com a implementação de uma técnica, também ela, baseada em Pós-Processamento. Serão dissecadas as distintas implementações utilizadas no tipo de técnica referida e, sabendo que estas se caracterizam essencialmente por um processo de filtragem, será adoptada a que mais se afigura com os intentos dos objectivos propostos.

O filtro a aplicar sobre a imagem, derivada da cena 3D, corresponde à componente mais crítica numa técnica de Pós-Processamento, com inúmeros factores a ter em conta para que a precisão do efeito de profundidade se aproxime da realidade. Estudos sobre os princípios da óptica relativos a modelos de câmara existentes terão de ser efectuados para definição das relações geométricas existentes. Poderão ser consideradas, por exemplo, a posição de captura de imagem, o plano focado no espaço da cena visualizada, e o diâmetro de abertura óptica caso o modelo em causa seja constituído por lente. Factores adicionais como a iluminação, o tratamento

de texturas ou o modo concreto do processo de aplicação de filtro são contemplados, resultando numa metodologia clara a ser usada em outras aplicações de criação de gráficos computacionais.

Também a possibilidade de existência de artefactos visuais comuns às técnicas hoje existentes será analisada, e consequentes hipóteses de resolução serão apresentados para que a simulação implementada se apresente visualmente correcta. Quaisquer procedimentos de correcção de artefactos visuais, i.e., irregularidades de acordo com a realidade, necessitam de ser integrados no algoritmo de aplicação do filtro gerador do efeito de profundidade de campo. Isto deverá ser realizado sem nunca desprezar a necessidade da execução se processar com rapidez na ordem de tempo-real, numa aplicação interactiva onde o plano ou objecto focado poderá ser alterado consoante a vontade do utilizador.

Por fim, no sentido de se obter *frame rate* de tempo-real durante a simulação da profundidade de campo, será necessário proceder a optimizações entre as quais a eficaz utilização das capacidades GPU. Os níveis de desempenho de processamento gráfico são hoje bastante superiores às de qualquer CPU devido à sua arquitectura estar desenhada para uma elevada computação paralela. As características vantajosas oferecidas por tecnologias GPU podem, e devem, ser aproveitadas na execução de programas que manifestem uma forte componente interactiva, tal como aquele realizado para esta Dissertação.

A aplicação apresentará uma interface simples e eficaz, para o utilizador, composta por uma janela de visualização da simulação e outra com todas as funcionalidades de controlo. À partida poderá ser escolhida, de entre várias cenas de características diferentes, a desejada para visualização do efeito de profundidade de campo com a subsequente possível manipulação do plano focado e com a exibição de resultados de desempenho entre outras funcionalidades disponíveis.

1.4 Abordagem e Principais Contribuições Previstas

A simulação de profundidade de campo não é novidade em CG, com os seus resultados mais evidentes a resultarem da sua aplicação em cinematografia, mais especificamente na criação de animações geradas inteiramente por gráficos computacionais. Imagens sintéticas cada

vez mais realistas são desta forma proporcionadas. Mas os avanços das tecnologias informáticas, e a consequente necessidade de inovação nos seus programas, traçaram um novo objectivo no âmbito do tema desta Dissertação: a simulação interactiva. Como consequência, técnicas como *Ray Tracing* tiveram de ser abdicadas na criação da profundidade de campo, porque a qualidade alcançada exigia um grande custo computacional, completamente inaceitável devido à enorme quantidade de processamento exigida. Foi então principiado um grande desenvolvimento noutro tipo de técnicas para simulação de profundidade de campo, as de Pós-Processamento, conseguindo combinar a precisão com a rapidez. Esta Dissertação apresentará uma solução dentro do contexto apresentado, baseada no presente Estado de Arte relativo à geração computacional da profundidade de campo.

Atingir os objectivos descritos anteriormente representa, por si só, uma contribuição de desenvolvimento, nesta área tão importante de GG como é a correcta simulação visual realista. Globalmente, o trabalho realizado consistirá em adaptar e evoluir uma existente implementação de profundidade de campo simulada. As principais contribuições previstas sintetizam-se essencialmente por:

- Apresentar uma metodologia genérica para simulação de profundidade de campo, a ser facilmente implementada por um qualquer programador de aplicações;
- Optimizar e superar limitações da técnica adaptada;
- Proporcionar uma simulação interactiva com frequência de apresentação de imagens adequada, próxima de tempo-real, por meio de hardware e tecnologias gráficas mais recentes;
- Promover futuros desenvolvimentos em áreas como Realidade Virtual ou Realidade Aumentada.

2. Estado da Arte

2.1 Os Conceitos da Profundidade de Campo

Profundidade de campo é, segundo J. Demers [Dem04], o efeito resultante do processo de focagem de objectos que se encontrem numa dada extensão, exibindo-se perfeitamente nítidos com uma alta definição dos seus contornos (aspecto *sharp*). Por outro lado, objectos que se distanciem da extensão focada apresentam-se naturalmente mais desfocados (aspecto *blur*), como se pode ver na Figura 1. Deste modo é oferecida ao observador uma indispensável percepção de profundidade para interpretação geométrica ou mesmo para cálculo da posição relativa de objectos numa cena.

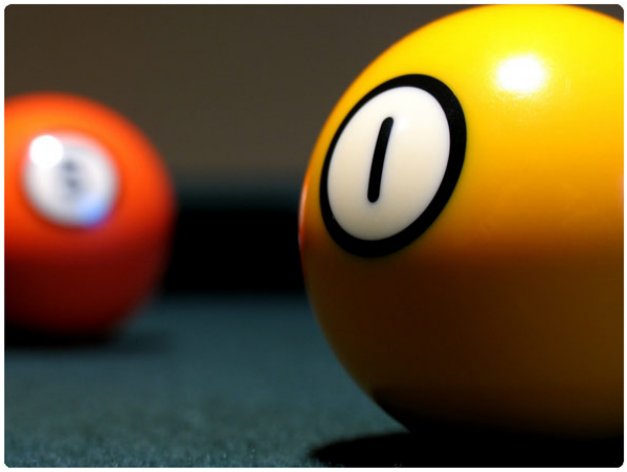


Figura 1 – Exemplo de Profundidade de Campo

A profundidade de campo é evidenciada principalmente nos domínios da fotografia e da cinematografia, com destaque da sua presente aplicação em jogos de computador e da sua valiosa contribuição nas áreas de Realidade Virtual e Realidade Aumentada. É essencial, numa dada cena, direccionar a atenção do observador, espectador ou jogador para um certo espaço ou

objecto e facultar ao mesmo tempo uma melhor noção de profundidade. Focando, por exemplo, uma pessoa no meio de uma multidão, não só a orientação visual será imediatamente guiada para essa pessoa como também para o contexto à sua volta, i.e., todo o ambiente que a rodeia.

Para obtenção de uma ideia conceptual mais correcta relativamente à profundidade de campo, pode-se exemplificar através dos princípios da fotografia. Em fotografias de tipo paisagem (*landscape*), a profundidade deve normalmente ser a máxima, consoante a câmara, pois nestes casos deseja-se focar o possível de uma cena consideravelmente distante. Já em fotografias de tipo retrato (*portrait*), a profundidade deve ser pequena, onde existe a necessidade de focar uma pessoa ou objecto que se encontra, provavelmente, algo perto do ponto de captura da foto.

No domínio da cinematografia, a profundidade de campo é notada de forma semelhante à fotografia, com a particularidade da captação de imagem ser contínua e o plano focado poder ser alterado como e quando desejado, alterando a noção de profundidade em conjunto com movimento da cena. Em qualquer obra cinematográfica é notória a utilização da profundidade de campo para guiar a atenção do espectador onde, por vezes, é simulado o campo de visão de uma personagem do filme e focado exactamente o local onde a mesma está a olhar. É dado um excelente exemplo de utilização do efeito de profundidade de campo em cinematografia numa sequência visualizada na Figura 2 do filme *I, Robot*, realizado por Alex Proyas (2004). Nesta, o actor (Will Smith) encontra-se focado e no momento em que passa por detrás do robot, mais próximo da câmara, o realizador transfere a atenção do espectador centrada no actor para o robot, precisamente por focar o plano onde este último se situa.

Figura 2 – Utilização do efeito de profundidade em cinematografia

Ainda em cinematografia, a maioria das animações são hoje criadas inteiramente através de processos digitais, com imagens geradas por computador. A profundidade de campo é um dos efeitos principais na procura de imagens sintéticas cada vez mais realistas como é possível verificar na Figura 3. De facto, é neste domínio que muitos desenvolvimentos gráficos são conseguidos com superiores movimentos, expressões faciais, iluminações, ambientes, entre outros, frequentemente envolvidos pelas leis da Física avistando o real cada vez mais perto.

Figura 3 – Imagem de filme *Ratatouille* da Pixar

Actualmente, com os avanços alcançados em termos de capacidades computacionais, começaram já a ser comercializados jogos com a singularidade de ser visualizada a profundidade de campo em tempo-real, na constante procura de aproximar os jogos das experiências cinematográficas. Essencial em jogos em primeira pessoa, exactamente porque a sua concepção é a de visualizar o ponto de vista do protagonista, como se a visibilidade da personagem fosse exactamente a do jogador. Jogos como *Call of Duty 4: Modern Warfare* (2007) ou *Crysis* (2007), exemplificam perfeitamente a aplicação de profundidade de campo em tempo-real em CG (Figura 4).

Figura 4 – Exemplo de Profundidade de Campo de jogo *Call of Duty 4*

Com a capacidade de simulação de profundidade em tempo-real como é o caso de videojogos, também a sua aplicação em Realidade Virtual é necessariamente crescente. De facto, para uma aplicação deste género se mostrar verdadeiramente realista esta terá de apresentar um verosímil efeito de profundidade para que a percepção do utilizador no mundo virtual não seja prejudicada e, por isto, não diminuída a qualidade da experiência.

Pode-se então constatar a real importância que a profundidade de campo tem para centralizar a atenção do observador para um corpo específico num qualquer espaço visualizado. Mas igualmente importante será conhecer os mecanismos de ocorrência do fenómeno objecto desta tese. Demers [Dem04] afirma que o efeito da profundidade de campo nasce da natureza física de lentes de máquinas fotográficas, de câmaras de filmar ou até da visão humana. Estas relações serão explanadas na subsecção 2.2.

2.2 Modelos de Câmara

Muitas aplicações gráficas com renderização realista, onde se insere o tema desta Dissertação, necessitam primariamente de definir um modelo de câmara real a simular. Geralmente gráficos de computadores carecem da profundidade de campo finita presente em câmaras reais. Melhoramentos contínuos são verificados na computorização de modelos ópticos para simulações exactas em computação gráfica interactiva. A sua finalidade é a obter

renderizações foto-realistas, permitindo igualmente combinar cenas reais e sintéticas (totalmente gerada computacionalmente) para ambientes de Realidade Aumentada ou ainda para estúdios virtuais [Hei97].

De seguida será feita uma revisão dos modelos de câmara mais importantes, actualmente existentes no âmbito da renderização gráfica. Seguindo-se de uma análise sobre qual destes proporciona uma melhor renderização do efeito profundidade de campo, segundo a sua própria natureza e suas características, e a que melhor se adapta aos intentos desta Dissertação.

2.2.1 Modelo *Pinhole*

O modelo de câmara *Pinhole* – expressão que significa "buraco de agulha" – tradicionalmente usada em computação gráfica (Figura 5), é também aquela usada para referir a Fotografia Estenopeica, reportando-se a pequeno furo.

Figura 5 – Modelo de câmara *Pinhole*. Cena tridimensional gerada computacionalmente (esquerda) e fotografia captada utilizando sistema óptico *Pinhole* (direita)

Uma câmara *Pinhole* não possui qualquer tipo de lente, tratando-se, numa visão simplista, de uma caixa totalmente fechada e opaca com um buraco de tamanho ínfimo num dos seus lados. A luz, ao atravessar o buraco, projecta uma imagem verticalmente inversa, no lado oposto da caixa. O princípio deste modelo pode ser visualizado na Figura 6. Devido ao buraco ser infinitesimamente pequeno, qualquer ponto do filme só pode ser atingido por um único raio

de luz, de uma direcção apenas. Todos os raios de luz criam desta forma uma imagem totalmente nítida com uma, teórica, infinita profundidade de campo [Kos01], como já constatado na Figura 5. A grande vantagem da utilização deste modelo é a sua simplicidade, e captar fotografias baseadas nas câmaras *Pinhole* torna-se assim uma experiência bastante elementar e económica.

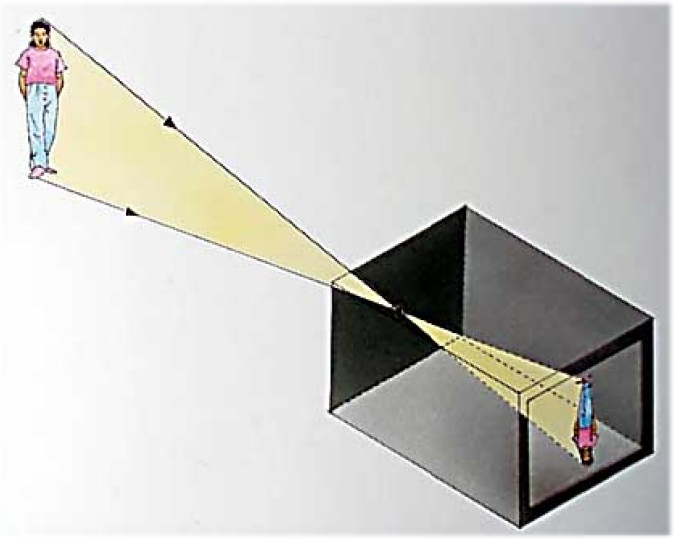


Figura 6 – Princípio do modelo de câmara *Pinhole*

Segundo W. Heidrich *et al*. em [Hei97], a projecção computacional verificada neste modelo pode ser descrita como uma transformação de perspectiva. Com parâmetros diversos como as dimensões do filme (película fotográfica) e a distância que separa o plano do filme ao orifício, juntamente com os adicionais planos de *clipping* mais próximo e longínquo respectivamente, delimitadores da zona visível ao observador.

Embora seja bastante fácil construir uma câmara deste modelo, e a teoria necessária para a obtenção de uma imagem seja desprovida de qualquer complexidade, já a obtenção de imagens de óptima qualidade pode tornar-se pouco prática por variadas razões. A mais importante será a pouca luz que recai sobre o filme devido ao orifício ser tão pequeno e, por esta razão, o tempo de exposição ser bastante longo para que se atinja a qualidade de imagem desejada. Outra questão prende-se com o facto de uma câmara *Pinhole* ter a sua abertura de diâmetro finito, que causará a perda de detalhe nos objectos que se encontrem extremamente perto, ainda que este fenómeno não seja facilmente notado [Kos01]. Se, de algum modo, o tamanho do orifício e o tempo de exposição não forem convenientemente balanceados, a qualidade de nitidez acaba por se perder.

As câmaras de modelo *Pinhole*, pela sua essência, restringem a sua utilização para fins artísticos e experimentais, resultando na obtenção de imagens extremamente peculiares.

2.2.2 Modelo *Thin Lens*

O modelo de câmara *Thin Lens*, conhecido de forma idêntica como o modelo das "Lentes Finas", que tal como indica o seu nome é possuidor de uma lente e mostra-se muito mais complexo e menos económico que o modelo anterior. Isto, também pela razão, de se apresentar bastante mais realista no universo de sistemas ópticos. Lentes reais têm uma abertura óptica (*aperture*) de tamanho finito, usualmente controlada em fotografia por um dispositivo físico denominado por diafragma. Ainda assim, a aproximação a este modelo, acaba por ser a mais simples dentro dos modelos de simetria circular e abertura óptica finita.

O princípio fundamental do modelo de câmara em causa, está associado à ténue espessura da lente, sendo mesmo ignorada. A luz que passa pela lente é refractada num único plano, o plano principal, e movida numa linha recta.

Para uma melhor compreensão, descrevendo o estudo efectuado em [Hei97] e seguindo a Figura 7, a luz vinda de um ponto **Q** do espaço de objecto – campo real onde os objectos físicos estão dispostos, ou todo o lado esquerdo da lente na Figura 7 – é focado num único ponto no espaço de imagem – extensão espacial onde será projectada a imagem, ou todo o lado direito da lente na Figura 7 – e vice-versa. Luz incidente do espaço de objecto paralelo ao eixo óptico é focada no ponto de focagem **F'** no espaço de imagem. Enquanto luz paralela do espaço de imagem é focada no ponto **F** no espaço de objecto. Refira-se ainda que ambos **F** e **F'** assentam sobre o eixo óptico perpendicular à lente. A lente foca todos os raios paralelos com o seu eixo num ponto que se encontra a uma certa distância do seu centro, esta denominada por comprimento focal, e representado por *f*.

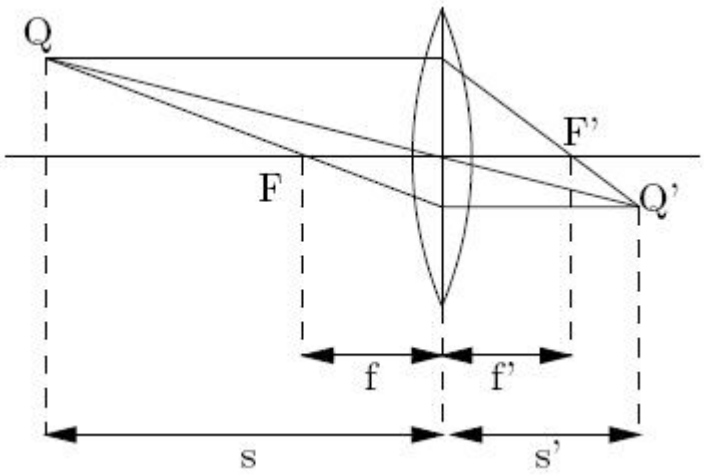


Figura 7 – Geografia de um sistema óptico *Thin Lens*

A partir da informação anterior é então possível entender como a imagem de uma cena é criada num plano de filme, a uma distância *s'* do plano da lente, dados a abertura óptica e o comprimento focal *f*. Para renderização computacional de imagens simulando o modelo *Thin Lens*, revela-se conveniente a especificação da distância *s* do plano de focagem em vez do comprimento focal, e por isso é salientada na Figura 7. Esta distância pode ser facilmente derivada da relação evidenciada na Equação 1, conhecida igualmente por "lei das lentes" [Kos01], relacionando a distância *s*, entre objecto e lente, e a distância *s'*, entre lente e imagem.

$$\frac{1}{s} + \frac{1}{s'} = \frac{1}{f}$$

Equação 1 – Lei das Lentes

Considerando agora a Figura 8 de [Dem04], sabe-se que múltiplos raios de luz dispersados a partir de um ponto de um objecto passam pela lente. Caso todos estes raios convirjam para um único ponto no plano de imagem a uma distância ***V*** da lente, o respectivo ponto do objecto define-se como focado. De outra forma, se os raios de luz provenientes do objecto não se reunirem num mesmo ponto do plano de imagem, o mesmo ponto do objecto não se encontra no plano focado da cena em questão. Significando isto que o objecto está fixado antes ou depois do plano focado, ou seja, desfocado. Os raios de luz provenientes de um ponto não focado intersectam o plano de imagem numa área circular, tipicamente conhecida como Círculo de Confusão. O diâmetro deste círculo, ***C***, pode ser calculado através da Equação 2.

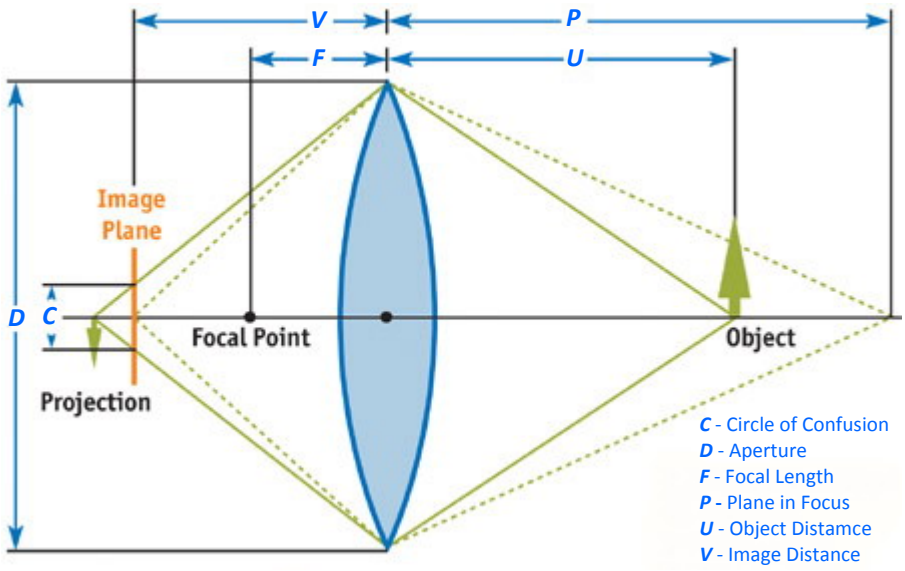


Figura 8 – Círculo de Confusão quando objecto desfocado

$$C = \left| D \frac{F(P-U)}{U(P-F)} \right|$$

Equação 2 – Diâmetro de Círculo de Confusão

$$a = \frac{f}{D}$$

Equação 3 – Rácio referente à abertura óptica

A grandeza ***D*** refere-se ao diâmetro da lente, representando da mesma forma a máxima abertura óptica. Esta grandeza, na prática, nunca é usada senão para se conhecer uma outra grandeza associada à abertura óptica, designada tanto por ***a*** como por ***k***, que define o rácio entre o comprimento focal e o diâmetro efectivo da lente (Equação 3).

Qualquer ponto com um diâmetro do Círculo de Confusão menor que um determinado máximo é considerado como estando focado. É possível calcular a distância, de focagem da lente, que irá projectar pontos até à infinidade, mais exactamente até ao raio máximo possível do Círculo de Confusão, C_{max}. A isto pode-se chamar de distância hiperfocal ***H***, calculado utilizando a Equação 4.

$$H = \frac{f^2}{C_{\max} k}$$

Equação 4 – Distância hiperfocal

R. Kosara, em [Kos01], considera ainda este número ***H*** interessante por ser relevante em fotografias de tipo paisagem, onde normalmente se requer a maior profundidade de campo permitida pelo sistema óptico. E também porque a partir deste número, os planos mais próximo, $\boldsymbol{u}_{\mathbf{near}}$, e mais afastado, $\boldsymbol{u}_{\mathbf{far}}$, que delimitam o espaço projectado de forma nítida na imagem, podem ser facilmente expressados. Caso a lente esteja focada para uma distância $u = \frac{H}{x}$ então os planos estarão a distâncias definidas pelas Equações 5 e 6.

$$u_{\mathrm{near}} = \frac{H}{x+1} \quad .$$

Equação 5 – Plano mais próximo delimitador de espaço projectado de forma nítida

$$u_{\mathrm{far}} = \frac{H}{x-1}$$

Equação 6 – Plano mais afastado delimitador de espaço projectado de forma nítida

2.2.3 Modelo *Thick Lens*

O modelo *Thick Lens*, tal como referido em [Hei97], manifesta ser um avanço nos modelos de lentes de simetria circular, onde a espessura da lente não pode mais ser ignorada como o era no modelo *Thin Lens*. Este é maioritariamente usado em sistemas ópticos mais complexos, muitos, compostos por várias lentes, expondo-se como uma aproximação mais exacta, mais real. Em termos de renderização, este modelo pode ser tratado de modo semelhante ao modelo *Thin Lens*, exceptuando as alterações nas disposições paralelas ao eixo óptico. Com lentes reais é normal a imagem resultante manifestar vários tipos de aberrações (difracções cromáticas, distorções geométricas, etc.). Mas, numa simulação computacional, para a que fossem visualizados estes efeitos seria necessário um modelo de câmara mais sofisticado, pois o modelo *Thick Lens*, tal como *Thin Lens*, garante simplesmente imagens desprovidas de qualquer tipo de distorção.

2.2.4 Modelo *Geometric Lens*

A aproximação ao modelo *Geometric Lens* permite, por sua vez, a simulação de todo o tipo de aberrações que nos modelos estudados anteriormente se indicava como impraticável. Também ao contrário aos modelos precedentes, este já é capaz de sustentar lentes sem simetria circular, como por exemplo as lentes bi-focais ou as lentes progressivas comummente aplicada aos dispositivos ópticos tanto usados pelo ser humano, os óculos. W. Heidrich *et al*., em [Hei97], constata que este modelo é estabelecido numa descrição geométrica completa das lentes em conjunto com o seu índice de refracção, com raios de luz incidentes traçados através da verdadeira geometria da lente, dobrando-se nas superfícies desta de acordo com a mudança no índice de refracção aludido.

2.2.5 Na Simulação de Profundidade de Campo

Tendo presente que uma gráfica simulação da profundidade de campo deve ter como base um modelo de câmara, poderão ser obtidas, depois de revistos os modelos mais importantes, algumas conclusões concretas sobre qual a mais indicada para os trabalhos desta Dissertação.

Verificou-se que a maioria dos métodos de renderização de imagens de cenas 3D, utiliza o modelo *Pinhole*, onde qualquer imagem se apresenta nítida em toda a sua profundidade, isto é, vemos os objectos na sua perfeição quer estando mais afastado ou mais próximo, afectando a percepção do observador e reduzindo o realismo da imagem considerada. Este modelo não demonstra praticamente nenhum tipo de efeito de profundidade de campo, exactamente o resultado oposto ao requerido nesta Dissertação, e assim este é forçosamente descartado para atingir o objectivo delineado.

Todos os seguintes modelos revistos partilhavam um facto em comum, a posse de uma lente, factor suficiente para uma aproximação ao sistema óptico do ser humano. Analogamente aos sistemas ópticos e seus dispositivos, a lente do olho (lente de uma câmara) permite a luz atravessar até à retina (filme) para formar a imagem captada. Esta é ao mesmo tempo focada, tendo em conta a abertura da pupila (abertura óptica de um dispositivo fotográfico). Na retina, a

imagem formada por mais de cem milhões de células foto receptoras transformam as ondas luminosas em impulsos electroquímicos, que são posteriormente descodificados pelo cérebro. Conclui-se assim que qualquer um dos sistemas ópticos possuidores de lente estão habilitados a renderizar profundidade de campo.

A partir das considerações revistas, dentro dos vários modelos de câmara com suporte de lente, o modelo *Thin Lens* apresenta-se como a mais vantajosa na aplicação prática nesta Dissertação, pelas seguintes razões:

- *Thin Lens* é um modelo de câmara similar à própria visão humana com o sistema de lente e abertura óptica, capaz de simular correctamente a realidade, com a profundidade de campo como o seu aspecto mais proeminente. Dentro dos modelos capazes de apresentar um efeito de profundidade de campo preciso, é o *Thin Lens* aquele que conseguirá mais facilmente aproximar-se às frequências de tempo-real devido a uma inferior complexidade na sua computação;

- Outros modelos simulam também outros efeitos, não visíveis em *Thin Lens*, tais como a difracção, distorções geométricas ou aberrações cromáticas, mas estes são raramente notados. Definida a profundidade de campo como o aspecto mais proeminente numa visualização realista, poderão ser ignorados outros modelos que simulem outros efeitos sem um interesse objectivo nesta Dissertação. Logo, são descartadas também implementações, desnecessariamente, bastante mais complexas;

- Apenas a difracção provocada pela abertura óptica interfere, teoricamente, com o efeito de profundidade de campo, pois se esta encurtar o seu diâmetro abaixo de um mínimo específico, a difracção criada provocará um aumento do Círculo de Confusão [Kos01]. De facto, numa lente real, a abertura óptica interfere de um modo mais negativo que a difracção, por se tratar de um polígono de, geralmente, sete ou oito vértices. Mesmo assim, se estivermos perante uma simulação, com a abertura óptica funcional, em que padrões de difracção possam ser visualizados, estes só serão realmente visíveis para luzes monocromáticas. Reforça-se mais uma vez que se torna escusado optar por um modelo de câmara mais complexo que o *Thin Lens*;

- Modelos de câmara que permitam a simulação em tempo-real de efeitos cada vez mais autênticos são ainda complicados de implementar por todos os factores críticos à computação relacionados com iluminação e os próprios princípios físicos da luz. Ponderando, em particular, a simulação da profundidade de campo em tempo-real, verifica-se que esta não está, ainda hoje, tão desenvolvido como seria talvez desejado mas passos seguros estão a ser dados nesse sentido. Perante os recursos computacionais disponíveis no presente, aplicações interactivas como videojogos, começaram já a adoptar o efeito de profundidade, elevando naturalmente a sua qualidade com a nova aparência realista, antes inexistente. Desenvolver um simulador interactivo baseado em modelos como *Thick Lens* ou, principalmente, *Geometric Lens* resultariam em custos de renderização elevados. Certamente que deitariam por terra a tentativa de simular, neste caso, profundidade de campo em tempos de execução interactivos, mesmo tendo em conta a capacidade computacional dos dias de hoje.

Também por algumas destas razões, a maioria dos métodos para modelação de profundidade de campo em computação utilizam os princípios do modelo de câmara *Thin Lens* como veremos mais à frente neste documento. E naturalmente será, da mesma forma, o modelo considerado durante o restante trabalho efectuado nesta Dissertação, como uma das bases para a requerida implementação.

2.3 Principais Tipos de Técnicas

A profundidade de campo foi já objecto de trabalho no passado, com a implementação de inúmeras aproximações a renderizações de imagens com profundidade de campo finita. Entre estas, serão aprofundadas de seguida técnicas de Multi-Passo, de Pós-Processamento e por camadas (*layers*). Todas habilitadas a simular a noção de profundidade em imagens sintéticas, a partir de uma cena gráfica tridimensional, com variados graus de qualidade.

2.3.1 Multi-Passo

Globalmente as técnicas Multi-Passo conseguem obter modelações de profundidade de campo mais próximas do real, como iremos verificar nesta subsecção. Devido à sua natureza, a aplicação de numerosos passos algorítmicos sobre uma dada cena 3D, resulta numa melhor qualidade final com o aumento de passos efectuados.

Irão ser focados os trabalhos:

- *Ray Tracing* Distribuído [Coo84];
- *Accumulation-Buffer* [Hae90].

2.3.1.1 *Ray Tracing* Distribuído

A técnica de renderização avançada, *Ray Tracing* Distribuído [Coo84], é tido em conta, para cada pixel, a possível reflexão especular envolvendo todos os objectos em cena (Figura 9 e 10), bem como a refracção da luz nos materiais transparentes. São traçados raios visuais atravessando cada pixel de imagem do plano de projecção, na existência de tantos raios quantos os pixels projectados, derivados de toda a lente e não de um único ponto desta [Dem04]. A cor final de cada pixel é calculada através da média de intensidades dos raios distribuídos [Zho07]. Como se está, objectivamente, a modelar a luz transportada para a câmara torna-se provável a geração de simulações de profundidade de campo mais exactas quanto possíveis [Dem04]. Este tipo de técnica é tão satisfatório que admite a projecção de efeitos como sombras suaves ou mesmo *motion blur*, em que o movimento de um objecto provoca na imagem uma impressão de desfocagem arrastada. Devendo-se isto exclusivamente à capacidade de tratamento de cada pixel independentemente dos pixels vizinhos.

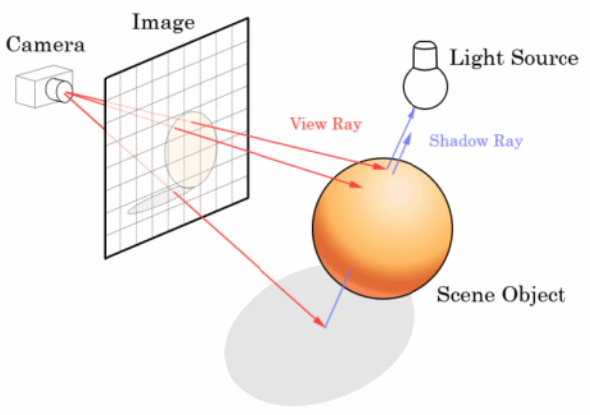


Figura 9 – Princípio da técnica *Ray Tracing*

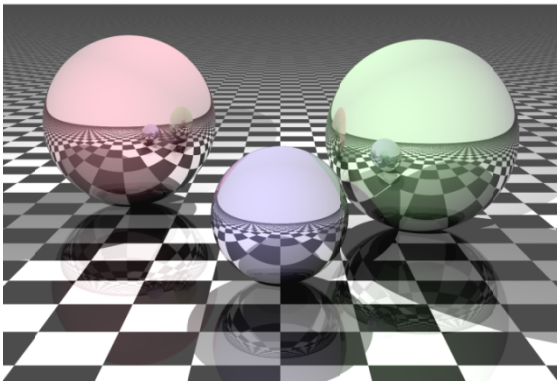


Figura 10 – Imagem renderizada por *Ray Tracing*

O processamento na técnica *Ray Tracing* é um excelente exemplo de uma relação qualidade/eficiência, pois garante a melhor qualidade de imagem possível em troca de uma excessiva computação. As suas velocidades de execução estão ainda afastadas das exigidas em simulações interactivas. A qualidade da imagem final, através desta aproximação, é elevada ao ponto de serem exibidos apenas suaves níveis de ruído (interferência na integridade da imagem) na sua presença, mesmo quando utilizado baixo número de raios que o aconselhável para a criação correcta de profundidade de campo.

2.3.1.2 *Accumulation-Buffer*

Segundo [Hae90], *Accumaltion-Buffer* permite simular a profundidade de campo através de uma aproximação da técnica de *Ray Tracing*, revista na secção 2.3.1.1, auxiliada pelo tradicional *z-buffer* (dados de profundidade) e por um *buffer* especial, o *accumulation-buffer*. A partir de uma abertura óptica de tamanho finito é renderizada uma cena múltiplas vezes, segundo o modelo *Pinhole* (secção 2.2.1), a partir de posições de câmara ligeiramente diferentes. É efectuada a amostragem (*sampling*) necessária por toda a lente [Hae90]. Todas as imagens renderizadas são assim combinadas num *buffer* acumulativo específico para o efeito, o já mencionado *accumulation-buffer*.

A profundidade de campo neste tipo de técnica exprime-se mais verdadeira, quantos mais passos de renderização forem efectuados, constatando-se que o número de passos necessários é proporcional à área do Círculo de Confusão em causa. Numa cena com uma profundidade acentuada, pode ser necessário um número de passos completamente proibitivo, quer em termos dos recursos computacionais existentes, como também dos exigentes níveis de desempenho impostos pela interactividade da aplicação, caso presente. Grandes círculos de confusão podem interferir na precisão do *accumulation-buffer* e gerar artefactos visuais na imagem final, isto é, incongruências na imagem que não deveriam ocorrer [Zho07]. Artefactos de várias espécies poderão ser formados como, por exemplo, *ghosting* onde um objecto é desvanecido em posições distintas à efectivamente captada, transmitindo a noção de um fantasma pairado na sua proximidade.

A técnica aqui revista é das poucas que potencialmente, conforme afirmado em [Dem04], consegue modelar em tempo-real uma verdadeira profundidade de campo, i.e., renderizar correctamente a visibilidade e sombras de uma câmara possuidora de lente e diafragma. Infelizmente, neste momento a capacidade desta técnica para oferecer bons resultados exige ainda um número de passos considerável que a afasta das frequências de tempo-real desejadas.

2.3.2 Pós-Processamento

As implementações da profundidade de campo através dos conotados métodos Pós-Processamento, caracterizam-se por seguirem um mesmo procedimento. Primariamente é renderizada uma cena para o *frame buffer*, e só posteriormente é processada a imagem da cena resultante, exibindo finalmente a imagem com o uma manifesta noção de profundidade. Geralmente, técnicas Pós-Processamento são conduzidas por meio de métodos de filtragem digital, razão por serem também conhecidas como técnicas de pós-filtragem. Este tipo de técnica tem sido fortemente empregue em trabalhos mais recentes, relativos ao tema objecto desta Dissertação. As suas vantagens prendem-se sobretudo por uma certa facilidade de implementação de uma correcta simulação, e por um reduzido custo computacional que é, normalmente, independente da complexidade da cena, segundo Zhou *et al*. [Zho07]. A essência da metodologia adoptada em Pós-Processamento, conjuntamente com um intenso

desenvolvimento aproveitando as crescentes capacidades de hardware gráfico, fazem da mesma a mais qualificada para a obtenção de profundidade de campo em *frame rates* de tempo-real.

A maioria das técnicas Pós-Processamento baseia-se no trabalho primário de M. Potmesil e I. Chakravarty [Pot82], ao estenderem a projecção tradicional de uma câmara *Pinhole* para um modelo de câmara mais realista, o modelo *Thin Lens* (secção 2.2.2). Causando uma aproximação a efeitos que uma lente real e pertencente abertura óptica proporcionam, sem a preocupação de modelar defeitos visuais inerentes à própria lente. Potmesil *et al*. criam então o processamento posterior à geração da cena tridimensional com a necessidade de guardar cada amostra de ponto, retendo informação relativa à posição, visibilidade e intensidade de cor. Cada ponto amostrado é convertido para um Círculo de Confusão, cujas dimensão e distribuição de intensidade são determinadas pela profundidade do ponto e das características do modelo de câmara utilizado. Assim, cada pixel na rasterização da imagem final é calculado pela acumulação das distribuições de intensidade dos círculos de confusão sobrepostos, isto para todos os pontos amostrados. A geração de imagens sintéticas, incorporando os factores relativos ao sistema óptico, permite a sua adaptação na recriação de diversos efeitos cinematográficos entre os quais a profundidade de campo. Posteriormente, Rokita consegue melhorar a eficiência da técnica de Potmesil em [Rok93], e atingir mais tarde a profundidade de campo com taxas de execução apropriadas para aplicações de realidade virtual [Rok96], dando uma excelente contribuição neste domínio particular de CG.

Embora praticamente todas as técnicas de Pós-Processamento sigam um procedimento global idêntico, existem ainda inúmeras variantes neste tipo de técnica. A maioria das variações ocorre na fase de filtragem. Foram publicados já distintos algoritmos para amostrar e pesar intensidades de pixels, baseados na profundidade ou informação relacionada com a profundidade obtida aquando da renderização inicial [Zho07].

2.3.3 Profundidade por Camadas

Segundo C. Scofield em [Sco92], técnicas processadas por camadas (*layers*) de imagem em profundidade podem ser vista como um compromisso entre as técnicas *Ray Tracing* [Coo84]

e Pós-Processamento [Pot82]. Isto por facultar profundidade de campo, por exemplo, não interferindo no processo de *shading* e respeitando, ao mesmo tempo, os princípios de uma lente óptica, oferecedora de diferentes vistas do ambiente em redor. A solução proposta assenta numa aproximação 2D com manipulação de camadas de cenas 3D, praticável de ser identicamente efectuada através de editores de imagens, como exemplo o Photoshop.

Com este conceito, Scofield num procedimento a envolver três fases divide, no seu trabalho em [Sco92], uma cena em várias camadas, cada uma com pixels possuidores da mesma, ou semelhante, distância da câmara (primeira fase). Cada camada é renderizada separadamente e aplicado o respectivo efeito embaciado consoante a profundidade (segunda fase), de forma a ser dado o efeito desfocado desejado. Todas as camadas geradas podem assim ser compostas numa imagem final dando a impressão de profundidade de campo (terceira fase).

Como cada camada possui uma imagem separada dos objectos visíveis na cena, os contornos destes adquirem uma visibilidade correcta, mas também problemas resultam deste processamento. A maioria das irregularidades derivam principalmente por cada objecto possuir uma profundidade representativa estabelecida através da sua média de profundidades, quando na realidade deveria ter em conta as profundidades das suas áreas na cena em questão. Alguns dos problemas associados esta técnica, tendo também como referência a análise a este método efectuada por J. Demers em [Dem04], podem-se sintetizar pelos seguintes pontos:

- Podendo os objectos serem repartidos por múltiplas camadas em profundidade, existirá sempre a hipótese de serem exibidas descontinuidades nos limites dos mesmos [Zho07];

- Numa camada, o efeito desfocado é aplicado de forma uniforme e objectos que se estendem demasiadamente em profundidade aparentam não possuir profundidade de campo algum, especialmente em porções de cena mais próximas do observador;

- Ambientes sintéticos onde os seus objectos se estendam por toda a profundidade da cena mencionada, como pavimentos ou paredes, caso sejam divididos em planos de igual profundidade, poderão surgir artefactos de complexa resolução. Um objecto dividido por múltiplas camadas diferentemente desfocadas, causará o erro de alterar o

seu verdadeiro formato. Pior artefacto surge, na divisão de um objecto por camadas, quando são incorrectamente considerados outros objectos como fazendo parte do objecto dividido, por serem muito pequenos. Poderão ser confundidos objectos pequenos como pertencentes a um objecto maior atrás destes, e ao serem misturados na mesma camada terão, incorrectamente, o mesmo tratamento.

2.3.4 Análise Comparativa

Dentro destas três metodologias revistas para modelação da profundidade de campo, pode-se concluir que técnicas de Multi-Passo conseguem efectivamente criar as imagens mais fotorrealistas que a computação gráfica permite. Mas neste caso a qualidade nasce somente em contrapartida de um intenso processamento. A qualidade resultante é claramente proporcional ao número de passagens algorítmicas, facto que as descarta definitivamente na sua aplicação em implementações de programas interactivos. Pois a execução arrasta-se levando a exibição da profundidade de campo para *frame rates* bastante distantes dos de tempo-real

A geração de profundidade de campo por camadas denota uma enorme possibilidade de ocorrência de diversos artefactos visuais, de médio ou difícil grau de resolução. Torna-se assim ineficaz na modelação correcta da profundidade de campo, sem qualquer menção a tempos de execução.

As técnicas de Pós-Processamento pela sua essência têm a capacidade de conjugar a precisão da simulação em causa com uma rapidez moldada às especificações de aplicações possuidoras de funcionalidades interactivas. A interactividade exige as imagens apresentadas sejam constantemente renderizadas, daí ser necessária uma metodologia que melhor se adapte e ofereça um resultado final balanceando de igual forma a sua qualidade e eficiência. E a metodologia Pós-Processamento vai de encontro aos principais objectivos de uma simulação de profundidade de campo, como também aos desta Dissertação.

2.4 Técnicas de Profundidade de Campo em Pós-Processamento

Nesta subsecção será feita uma revisão de cinco, das mais recentes, técnicas de Pós-Processamento. Serão analisadas diferentes formas de abordar o mesmo problema através das mais desenvolvidas metodologias para simulação de profundidade de campo em tempo-real. Por fim, apresentar-se-á uma análise comparativa entre as técnicas revistas para determinação da que melhor respeita e garante, respectivamente, todos os requisitos e objectivos desta Dissertação.

2.4.1 Cinco Técnicas Pós-Processamento

2.4.1.1 Rápida Renderização de Profundidade de Campo com *Surface Splatting*

Os autores J. Krivanek e J. Zara [Kri03] desenvolvem um rápido algoritmo que renderiza a profundidade de campo para superfícies ponto a ponto (*point-based*), assentando na técnica de filtragem *surface splatting* proposta por Zwicker *et al.* [Zwi01]. Esta técnica renderiza superfícies, opacas ou transparentes, de agrupamento de pontos sem qualquer conectividade. Em *surface splatting*, *splats* são definidos como pontos do modelo, com uma normal e um raio associados, onde é aplicada uma função Gaussiana radialmente simétrica.

Apesar deste método ser conotado como Pós-Processamento, distingue-se por cada ponto ser desfocado antes da formação da imagem final, evitando o processo de focagem em separado, como acontece regularmente na maioria dos métodos de Pós-Processamento. Outra singularidade é o facto de a simulação da profundidade de campo na suportar transparência, como é possível verificar na Figura 11.

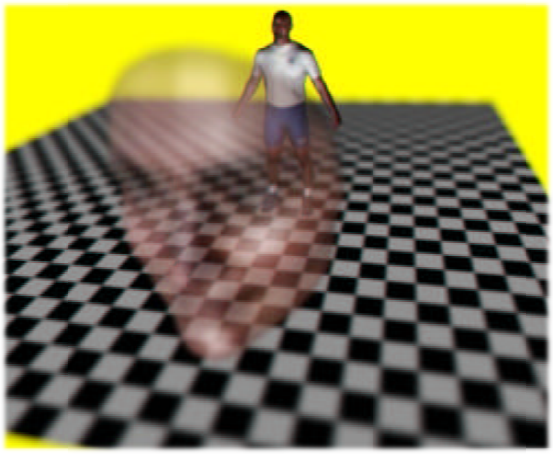


Figura 11 – Profundidade de Campo e transparência

Na sua essência, o algoritmo de Krivanek *et al.* desfoca *splats* individuais envolvendo os mesmos com um filtro Gaussiano, em vez de desfocar a imagem propriamente dita. Sintetizando o processamento, cada *splat* é amplificado proporcionalmente à quantidade de desfocagem requerida de acordo com a sua profundidade na cena.

De modo a acelerar o desempenho da técnica, é usado um baixo nível de detalhe para as superfícies desfocadas, tornando a velocidade do algoritmo independente da quantidade de desfocagem consoante a profundidade.

Devido ao facto de cada ponto ser desfocado antes da formação da imagem final, este método possibilita a resolução de incoerências visuais como oclusão parcial e *intensity leakage*. Estes não são mais que artefactos visuais muito comuns nas técnicas de Pós-Processamento que serão estudadas em maior detalhe na subsecção 2.5 deste documento. As desvantagens capitais da técnica de Krivaneck *et al.* são, primeiro, a sua alta sensibilidade, prejudicial na regularização das amostras (*samples*) posicionais na superfície dos objectos e, segundo, a presença ocasional de artefactos. Estes são mais evidentes em objectos complexos com bastantes curvas porque, com opções técnicas respeitantes à união/separação de superfícies no *A-buffer* (*buffer* de visibilidade *anti-alias*), certos artefactos chegam mesmo a ser impossíveis de evitar quando existem superfícies posicionadas perto umas de outras, ou até intersectadas.

2.4.1.2 Correcta Profundidade de Campo Através de Difusão Anisotrópica

Bertalmío *et al*., em [Ber04], introduz uma nova equação diferencial parcial (EDP) para difusão anisotrópica, aplicada a imagens 2D renderizadas por modelo de câmara *Pinhole*. Isto, desfocando as imagens segundo a informação de profundidade presente no *z-buffer* da cena 3D correspondente.

O objectivo, dos autores deste método, é adequar uma difusão anisotrópica a uma imagem em que cada ponto possua um nível de desfocagem correspondente ao diâmetro do seu respectivo Círculo de Confusão.

Os dados de entrada necessários ao algoritmo do método em questão são a imagem texturizada 2D da cena 3D, e o *z-buffer* desta. Pela razão de ambos se localizarem totalmente em GPU, possibilita também que o algoritmo corra inteiramente no GPU. Facto que aumenta drasticamente o rendimento ao evitar comunicações com o CPU através do *bus* de dados.

Todo o processo ocorrido em GPU em conjunto com um esquema numérico adequado à EDP referida, e adaptado perfeitamente às características da linguagem de hardware gráfico programável de CG, possibilita uma renderização em tempo-real.

Na Tabela 1 é possível ver os testes de desempenho realizados ao algoritmo. Usando uma placa gráfica NVIDIA GEFORCE FX 5950 Ultra, são mostrados os FPS (*frames* por segundo) para três diferentes resoluções segundo diferente número de iterações por *frame*. Nas imagens geradas é notada uma realística profundidade de campo, embora seja possível aumentar o número de iterações e consequente desfocagem de imagem. Neste caso, custos superiores de *frame rate* seriam verificados.

R/N	5	10	15	20
640x480	35.7	21.7	15.4	11.9
800x600	21.5	13.4	9.7	7.7
1024x768	13.5	9.0	6.0	4.9

Tabela 1 – Desempenho em FPS segundo uma resolução R e N iterações

O algoritmo proposto atinge bem os fenómenos ópticos reais, sem artefactos visuais e inconsistências de profundidade presentes noutras aproximações, como *intensity leakage*.

Segundo os autores a grande vantagem desta técnica é igualmente a fonte de maior inconveniência, ao ser executada totalmente em GPU o seu sobrecarregamento é enorme. Assim, desenvolver a técnica mostra-se uma tarefa pouco acessível pela complexidade de programação para GPU. Hoje esta questão é desvalorizada pela apresentação de linguagens de programação de alto nível para GPU, com interfaces de certo modo mais simplificadoras.

2.4.1.3 Profundidade Interactiva por Difusão Simulada em GPU

M. Kass, A. Lefohn e J.Owens, em [Kas06], desenvolveram uma aproximação à profundidade de campo numa tentativa de criar uma ferramenta para criação de animação cinematográfica. O seu intuito seria o de atingir de maneira mais facilitada imagens de acordo com a visão de realizadores ou directores de fotografia. Esta é anunciada pelos autores como suficientemente boa para pré-visualização de filmes animados, possível de ser computada interactivamente em GPU. Os criadores de uma animação cinematográfica, na posse da aplicação baseada nesta técnica, poderão especificar as configurações de câmara e capturar imagens com a abertura óptica da lente e com o ponto de foco desejado, de forma totalmente interactiva. É obtido um nível superior de controlo artístico sobre as cenas tridimensionais, que na área da animação cinematográfica é deveras importante. Depois de parametrizada cada cena da animação na aplicação sugerida, é então utilizada a técnica de *Ray Tracing* distribuído (secção 2.3.1.1) para computação das imagens finais exibidas no filme em questão.

A profundidade de campo é modelada por Kass *et al*. como uma difusão de calor, onde o tamanho do Círculo de Confusão corresponde ao calor do pixel. Quando o factor de aquecimento é nulo também a desfocagem o é para os pixels relativos, garantindo o correcto processamento dos contornos entre objectos nítidos e objectos muito desfocados que se encontrem por trás.

A solução em estudo direcciona-se para uma série de sistemas lineares tridimensionais, que implementam efectivamente filtros recursivos independentes, computados em tempo constante por pixel, autónomo do tamanho dos círculos de confusão processados [Kas06].

O processamento deste método divide ainda a cena em *layers* para simular a profundidade de campo de modo mais preciso, superando artefactos visuais e a visualização incorrecta de objectos segundo as suas profundidades e o ponto de foco. Permite-se assim simular a oclusão parcial e visualizar objectos que se encontrem por detrás de um objecto opaco e que, entretanto, se tornou parcialmente transparente. Isto acontece numa situação em que o objecto é bastante pequeno relativamente à abertura óptica da câmara, e se encontra a uma profundidade muito próxima da captura de imagem (Figura 12).

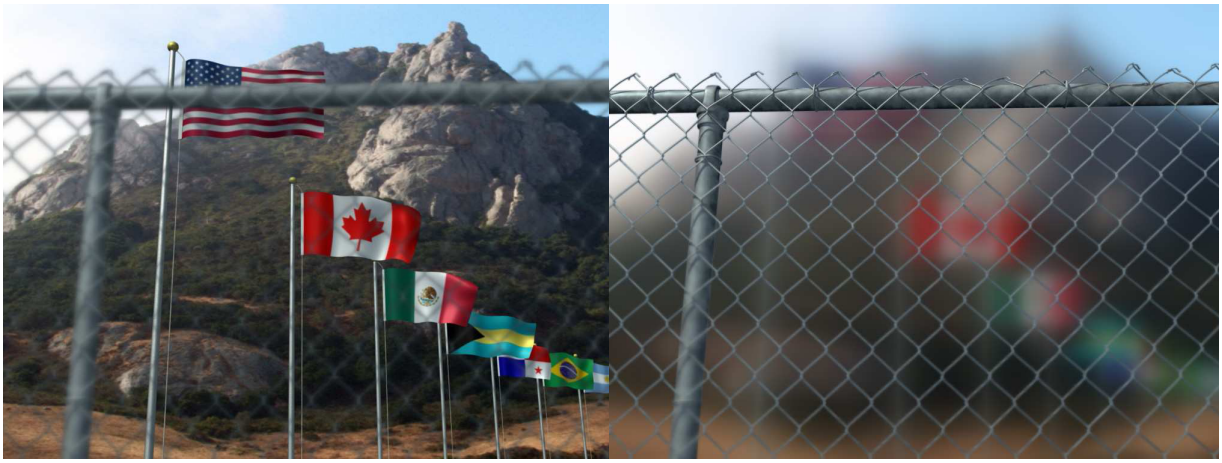


Figura 12 – Simulação de profundidade de campo por difusão de calor. Plano focado bastante afastado (esquerda) e plano focado bastante próximo (direita)

Executada a aplicação num computador 2.4 GHz Athlon 64 FX-53, Windows XP como sistema operativo e utilizando uma placa gráfica NVIDIA Geforce 7800, foram obtidos os resultados apresentados na Tabela 2.

Resolution	FPS
256x256	80-90
512x512	21-23
1024x1024	6-7
1024x768 (with the separate foreground layer added for the flag images)	3-4

Tabela 2 – Desempenho em FPS segundo diferentes resoluções

Ao contrário de grande maioria de métodos anteriores, este consegue atingir alta qualidade e velocidade interactiva ao mesmo tempo [Kas06]. Entre várias opções de implementação eficientes, é utilizado um tempo de computação constante por pixel, e são tomados padrões de gestão de memória, inteiramente adequados face à realidade do GPU.

Por fim, na técnica apresentada por Kass *et al*., a equação de calor aproxima-se a uma específica função Gaussiana, caracterizada pelo aparecimento do efeito de *bokeh*, verificando-se uma distorção exagerada dos planos não focados. No limite, a forma da abertura óptica é evidenciada por toda a zona desfocada da imagem (Figura 13). Se forem utilizadas outras distribuições de calor, outros efeitos visuais não pretendidos poderão surgir, facto este problemático para a execução da simulação de profundidade de campo.

Figura 13 – Efeito *bokeh* em fotografia

2.4.1.4 Prática Profundidade de Campo por Pós-Processamento

A técnica de profundidade de campo agora apresentada direcciona-se particularmente para o universo dos jogos de computadores e consolas, principalmente para os de estilo *"Primeira-Pessoa"*. Neste tipo de jogos o jogador controla a visão da personagem no jogo, por isso torna-se essencial a profundidade de campo para que a mesma se torne tão real quanto possível.

E. Hammon e a sua equipa [Ham07] desenvolveram uma técnica para o jogo, de contexto de guerra, Call of Duty 4: Modern Warfare, com o intuito de dar aos jogadores uma rica e imersiva experiência cinematográfica. Consegue simular uma profundidade de campo de óptima qualidade com a vantagem de praticamente não afectar o desempenho total do sistema.

A solução de Hammon *et al*. descarta, em situações particulares, a simulação correcta da profundidade de campo em prol de uma melhor experiência de entretenimento proporcionada ao jogador. Na fase de implementação, quando objectos não focados se posicionavam à frente de objectos focados, eram gerados artefactos visuais na jogabilidade. Isto levou Hammon a decidir simplesmente por desfocar as descontinuidades visíveis nestes casos, descartando uma aproximação fisicamente correcta. Os pixels do contorno de um objecto desfocado, à frente de um outro focado, perdem nitidez considerando os diâmetros dos seus círculos de confusão e dos respectivos pixels vizinhos.

Em resumo, o algoritmo cria o efeito de profundidade numa imagem seguindo várias fases de processamento. Depois de amostrados os círculos de confusão dos objectos mais próximos de visualização, é então desfocado a contígua imagem com os círculos de confusão, calculados através de funções Gaussianas. Posteriormente é obtido o real Círculo de Confusão a partir da imagem desfocada e da imagem não desfocada. Finalmente é aplicada uma desfocagem específica e crucial para que a cena resultante possua uma profundidade de campo uniforme e uma variação de cores correcta, conforme o grau de desfocagem correspondente [Ham07].

Esta técnica de Pós-Processamento apresenta características de acordo com a realidade dos jogos computorizados na primeira pessoa. Por exemplo, no jogo ao qual se destina este método, a personagem principal possui uma arma, precisamente a principal fonte de interacção

com o jogador. Deste modo são verificados problemas na conjugação da profundidade de campo no mundo do jogo e da profundidade referente ao modelo de visão do jogador. Por esta causa, é dada a possibilidade de configuração separada das duas profundidades de campo e, assim, é descartada mais uma vez a simulação correcta da profundidade de campo.

No decurso do jogo, é exibido um efeito de profundidade de campo bem conseguido denotando suavidade com um custo de 1 a 1,5ms apenas, ou seja, tempos de processamento ideais para uma aplicação interactiva como é o caso. Para este teste foi utilizada uma Radeon X1900 (e uma equivalente Geforce 7900) como placa gráfica, a uma resolução de 1024x768. Por outro lado, é exibido *intensity leakage* e outro problema inerente à própria técnica, manifestando-se a perda de nitidez em toda a imagem, e não em certos planos apenas, quando o raio de desfocagem é muito grande.

2.4.1.5 Exacta Simulação de Profundidade de Campo em Tempo-real

Neste último método analisado, realizado por T. Zhou *et al*. [Zho07], é revista uma implementação por Pós-Processamento capaz de simular uma exacta profundidade de campo com *frame rates* apropriados para aplicações interactivas.

Esta solução, muito recente, apresenta a inovação de utilizar directamente a informação contida no *z-buffer*, evitando qualquer intervenção no processo de renderização da cena referida. São reduzidos passos extra de renderização para recolha os dados necessários ao processamento seguinte, frequentes na maioria das restantes técnicas deste tipo. Este factor, não só provoca um menor custo de execução como também simplifica a sua implementação e, consequentemente, a integração com outras aplicações de renderização de imagens.

O Pós-Processamento, em si, é caracterizado pela aplicação de um filtro de dois passos (vertical e horizontal) capaz de efectuar amostragem de um vasto número de pixels, impossível de ser atingido por um filtro de passo único. O filtro afecta cada pixel amostrado ao considerar a sobreposição dos círculos de confusão dos pixels vizinhos, a intensidade da luz e a possibilidade de existência de *intensity leakage*.

Zhou *et al*. conseguem, de facto, criar uma técnica de excelente desempenho. Beneficia da sua implementação extremamente optimizada, capaz de ser parcialmente executada em GPU, e dos recentes desenvolvimentos de hardware gráfico e à sua acessibilidade através de APIs desprovidas de específicas complexidades anteriormente verificadas. Utilizando um Pentium IV 2.8GHz, 1 GB RAM, placa gráfica nVIDIA 6800 e sistema operativo Linux, verificaram-se os resultados apresentados na Tabela 3, referentes à cena visualizada na Figura 14. É concluído, em [Zho07], que o custo do pós-processamento é independente da complexidade da cena em causa, escalado de acordo com a variação de resolução.

Figura 14 – Cena 3D com efeito de profundidade de campo

Resolution	Frame time (no DOF)	Frame time (DOF)	DOF cost
512 × 512	4.5	10.9	6.4
1024 × 512	4.7	17.7	13.0
1024 × 1024	5	30.1	25.1

Tabela 3 – Desempenho em milissegundos segundo diferentes resoluções

Esta técnica tem a vantagem de atingir uma simulação precisa e de ser executada em tempo-real, sem qualquer perda de intensidade na sobreposição de objectos em profundidade. Ainda assim, ostenta algumas limitações como é o caso da dimensão do filtro aplicado. Quando este aumenta de tamanho o desempenho global é prejudicado consideravelmente. Torna-se um

problema ainda maior quando se está perante cenas com grande profundidade, causador de uma inevitável contrapartida entre a correcta simulação e a sua eficiência de execução.

2.4.2 Outras

Na simulação de profundidade de campo muitas outras soluções foram já implementadas a partir do trabalho desenvolvido por Potmesil [Pot82], mas por se desviarem dos objectivos reais desta Dissertação ou pelo facto de já terem sido objecto de desenvolvimento para outras soluções, não são incluídas como matéria investigada nesta preparação de Dissertação. Contudo, dentro das muitas soluções mencionadas, refere-se a de T. Kosloff *et al*. [Kosl07] e a de P. Fearing [Fea96] que introduzem uma visão inovadora, respectivamente, quanto à aplicação de profundidade de campo em CG e quanto a um diferente conceito de implementação e processamento.

Um avanço recente na simulação de profundidade de campo é apresentado pelos autores Todd Kosloff e Brian Barsky em [Kosl07]. Invocam a questão de que a criação do efeito de profundidade, obedecendo às leis físicas de um modelo de câmara de lente e diafragma, é limitada em termos das capacidades correntes da própria Computação Gráfica. Neste sentido, Kosloff e Barsky sugerem uma nova forma de introduzir a noção de profundidade em imagens graficamente sintéticas. Utilizam difusão anisotrópica (também em [Ber04]) para generalizar a profundidade de campo que permite o utilizador especificar, de modo independente, o grau de desfocagem em cada ponto, objecto ou plano do espaço tridimensional [Kosl07]. Com isto é então permissível, por exemplo, atrair a atenção do observador para uma única pessoa ou mesmo para distintas pessoas afastadas e com diferentes distâncias da câmara, no meio de uma multidão, desfocando toda a cena restante. Embora esta implementação consiga simular uma correcta profundidade de campo, não incide no objectivo desta Dissertação nem procura uma execução em tempo-real.

Uma outra implementação bastante referenciada é a de Paul Fearing [Fea96], centrando o estudo num modo alternativo de criação de profundidade de campo em tempo-real. Face à grande quantidade de computação na recriação do efeito, Fearing atribui uma específica ordem

de importância a cada um dos pixels da imagem sintética. É dada prioridade à computação da profundidade de campo na área da cena onde o efeito é mais notado, processando as áreas menos importantes caso haja tempo suficiente. Isto é, o tempo de execução é definido à partida, e nesta base o algoritmo aplica à cena o efeito de profundidade da melhor forma possível, através da ordem de importância referida. Nunca atingindo uma simulação de elevada qualidade sem alguma perda de eficiência, ainda assim esta modelação consegue obter níveis de execução equiparados aos de tempo-real, ao evitar a contínua computação de enormes quantidades de pixels que se verificam não terem sido alterados em relação a uma imagem anterior. Trata-se, por isto, de uma implementação particularmente eficaz em sequências de *frames* relacionados [Fea96], demonstrando-se como uma ferramenta a ponderar essencialmente em animação computorizada.

2.4.3 Análise Comparativa

Esta comparação entre técnicas sugere uma necessidade de verificar qual das técnicas estudas é a que melhor se adequa como ponto de partida à elaboração desta Dissertação.

As técnicas revistas possuem aspectos distintos e, algumas, propósitos aplicacionais dissemelhantes. Embora todas tenham o objectivo de atingir uma melhor e mais rápida simulação de profundidade de campo, algumas como [Zho07], [Ber04] e [Kri03] são criadas com o intuito único de evoluir a geração e visualização realística de cenas sintéticas tridimensionais em CG. Outras, como [Kas06] e [Ham07], são realizadas atendendo a requisitos específicos. A técnica de [Kas06] surge com a necessidade de uma ferramenta de pré-visualização de cenas de animação cinematográfica antes da sua renderização final. E em [Ham07] é concebida uma técnica para um jogo computacional, totalmente adaptada às características da própria jogabilidade.

A solução apresentada em [Ham07] tem a grande desvantagem de descartar a simulação correcta da profundidade de campo por esta não se adequar à realidade do jogo em si. Quanto à técnica de [Kri03], requisitando o domínio da técnica *surface splatting* revela-se algo instável com artefactos resultantes do aumento da complexidade dos objectos em cena. As técnicas de [Ber04] e [Kas06], através da difusão anisotrópica, apresentam-se como boas soluções, de

resultados bastante satisfatórios. Mas é a técnica de [Zho07] que procura efectivamente a precisão da simulação em termos físicos, perante um sistema de lente e diafragma utilizando uma rigorosa informação de profundidade dada pelo *z-buffer*. Estas últimas três técnicas referidas fazem uso das capacidades do GPU, executadas parcial ou totalmente no mesmo, com tempos de resposta bem superiores a outras técnicas. Ainda neste ponto, Zhou *et al*. resolve problemas que outras técnicas evidenciam quando utilizam capacidades especiais do GPU ou da API gráfica, prejudicando as suas integrações em aplicações de renderização.

Ainda que havendo hipótese de ser realizada uma comparação a partir dos resultados apresentados em cada implementação, esta torna-se bastante relativa devido às respectivas tecnologias utilizadas serem significativamente diferentes. De facto, é proibitivo efectuar-se conclusões partindo de resultados obtidos sem se estar na presença dos mesmos recursos de processamento gráfico. Infelizmente só [Zho07], [Ber04] e [Kri03] realizaram estudos alargados de desempenho às suas próprias cenas, sendo que [Ham07] fez uma breve nota de tempo de execução e [Kas06] o estudo da quantidade de *frames* por segundo (FPS) em cada execução. Ainda assim, é concluído que as soluções apresentadas por [Zho07] e [Ham07] denotam tempos de execução mais adequados à interactividade, se for implicada a complexidade das cenas em causa. Em [Zho07] é mostrado que o custo de pós-processamento é independente da complexidade da cena, enquanto em [Ham07] não é feita nenhuma referência neste sentido mas esta, como já indicado, não simula uma correcta profundidade de campo.

Pela sua qualidade, eficiência e simplicidade conclui-se que a modelação efectuada por [Zho07], num excelente exemplo de Pós-Processamento, é a que mais se adequa aos objectivos pretendidos por esta Dissertação.

2.5 Artefactos Visuais Comuns em Pós-Processamento e Soluções

Os algoritmos Pós-Processamento denotam, pela sua natureza de implementação, problemas causadores de inconsistências perceptivas da profundidade, denominados também por artefactos visuais. Geralmente a maioria dos artefactos evidenciados partem de problemas

relacionados com a oclusão de objectos na cena gerada, tendo em conta o ponto onde a mesma cena é captada ou visualizada.

2.5.1 Problemas de Oclusão

Todas as simulações por Pós-Processamento revistas aplicam um modelo de câmara com lente e abertura óptica finita, em que qualquer ponto captado no plano de imagem corresponde à agregação de luz proveniente de cada ponto da lente [Kosl07]. Facilmente evidenciam incorrecções visuais não coerentes com a realidade que estão associadas a situações de oclusão entre objectos, verificadas em imagens renderizadas com profundidade de campo.

Em imagem real, numa cena com objectos que se ocluam, raios de luz provenientes do objecto ocluído conseguem circundar o referido oclusor e agregarem-se longe do centro da lente do sistema óptico utilizado. Isto demonstra que o objecto em *background* é apenas visível em certos pontos da lente com evidência de um efeito suave de semi-transparência nos contornos do objecto oclusor não focado. Define-se assim o conceito "oclusão parcial".

O efeito de oclusão parcial é genuinamente criado em Óptica, e por isso deve fazer parte de uma qualquer simulação de profundidade de campo que se defina por correcta [Kosl07]. Concretamente, para simular o efeito agora analisado é necessária informação relativa a porções não visíveis de objectos ocluídos na cena 3D, para assim, através de cálculos geométricos, apresentar as zonas parcialmente ocluídas na imagem final. Este processamento é uma das maiores dificuldades na modelação de profundidade de campo por Pós-Processamento. Em simulação é fulcral a existência de oclusão parcial quando um objecto muito fino, ou de pequenas dimensões, se posiciona muito próximo da lente a ocluir um outro objecto de dimensões superiores. Neste caso o último objecto estará parcialmente ocluído na zona de contornos do objecto oclusor se existir uma grande abertura da lente, consentindo a recepção de raios de luz do mesmo objecto em *background.* Uma situação de oclusão parcial permite assim a percepção do respectivo fragmento de cena ocluído.

Outro problema perante objectos ocluídos em profundidade de campo é a chamada *intensity leakage*, que se caracteriza por uma fuga de intensidade de cores de pixels quando

processada a desfocagem. Se numa imagem, um objecto focado em *foreground* encontra-se sobreposto a um objecto desfocado em *background* (ou vice-versa), as cores do objecto desfocado poderão ser misturadas com as do objecto focado. A simulação torna-se desta forma incorrecta comparativamente à uma situação idêntica em óptica real.

2.5.2 Soluções

Com a primeira simulação de profundidade de campo em CG, desenvolvida por Potmesil e Chakravarty [Pot82] através de uma aproximação de Pós-Processamento, eram apresentados também os primeiros artefactos visuais frequentes neste tipo de técnica. Posteriormente, Shinya [Shi94] consegue resolver os problemas de oclusão apontados ao trabalho de Potmesil com a utilização de um *buffer* de distribuição de raios de luz. Mais concretamente um *buffer* de sub-pixels para cada pixel para rastrear todos os raios de luz inseridos nesse pixel pela lente [Bar05].

Quanto à oclusão parcial, ainda hoje a solução apresentada por Shinya carece de suporte hardware, e sobrecarrega o computador em termos de processamento com de custos de memória associados. A simulação ressente-se no seu desempenho, tornando-a proibitiva se integrada em aplicações de intensa interactividade. De uma forma geral, as técnicas Pós-Processamento não tentam resolver o problema da oclusão parcial [Zho07] pela complexidade e carga de execução. Ainda assim, soluções como as de Krivanek *et al*. [Kri03] e de Kass *et al*. [Kas06] conseguem resolver aceitavelmente questões de oclusão e serem, ao mesmo tempo, consideravelmente rápidas. Neste âmbito, a solução de Krivanek assenta na técnica de filtragem *surface splatting,* enquanto a de Kass resulta da aplicação de um filtro extra, exclusivo ao tratamento de questões de oclusão parcial.

Relativamente a artefactos por *intensity leakage*, dificilmente é hoje deixada por tratar em qualquer simulação de profundidade de campo por Pós-Processamento. Mas as soluções neste ponto não apresentam, globalmente, uma forma comum de ser resolvida, mas sim maneiras adaptadas à implementação efectuada que evitem a fuga de intensidade entre objectos sobrepostos. Por exemplo, em [Zho07], Zhou *et al*. têm, no seu filtro aplicado à imagem 2D original, uma componente específica para tratar apenas da questão de *intensity leakage*. Nesta

garante-se a não contribuição dos pixels de um objecto para além do plano focado, no cálculo dos pixels de um objecto focado.

Foi investigado a existência de trabalhos existentes relativamente aos problemas salientados nesta subsecção, ressaltando um apresentado por Barsky *et al*. [Bar05], com o objectivo único de resolver os mesmos artefactos visuais já considerados. A sua solução passa essencialmente por definir se uma porção da imagem pertence a um ou mais objectos. Os problemas levantados pelos autores são os de oclusão e os de discretização, i.e., derivados do algoritmo separar a imagem em sub-imagens por profundidade. Desta separação ressalta a questão para determinar se numa imagem dois pixels adjacentes pertencem a objectos diferentes (em planos diferentes), ou se pertencem ao mesmo objecto (no mesmo plano de profundidade). Nasce então o novo conceito "identificação inequívoca de objecto" e nele são consideradas, pelos autores, duas aproximações: a "técnica de detecção de arestas" e a "técnica de diferença de pixels adjacentes".

A execução da técnica de detecção de arestas utiliza a imagem original e o seu mapa de profundidade. A partir de uma sub-imagem (um plano da imagem em profundidade), mais especificamente da região formada pelos pixels delimitadores da sub-imagem corrente, a região é estendida até às arestas detectadas nos passos anteriores do algoritmo (Figura 15). Depois da expansão e da região estar completamente envolvida por arestas intersectadas, são excluídas as regiões da mesma sub-imagem que não pertencem ao objecto definido pela região previamente estendida.

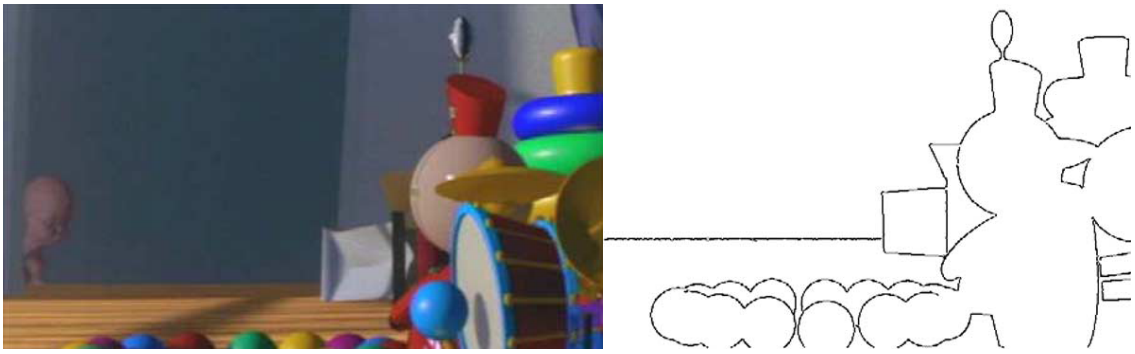


Figura 15 – Algoritmo de detecção de arestas identifica onde se posicionam os limites dos objectos na cena

Por sua vez, a técnica da diferença de pixels adjacentes (Figura 16) consegue utilizar todas as intersecções entre um raio infinito da câmara e a cena, e definir os objectos na sua totalidade. Porque é dado ao algoritmo o acesso total à informação de espaço da imagem sintética [Bar05]. O método consegue resolver perfeitamente os artefactos visuais ao atribuir, inicialmente, o identificador do objecto a cada pixel respectivo. Torna-se assim infalível a perfeita definição da posição de cada objecto e dos seus pixels numa imagem graficamente computada.

Figura 16 – Processamento interno da técnica da diferença de pixels adjacentes

Estes algoritmos não têm em conta a sua execução em tempo-real, e ainda que estejam sujeitos a erros associados à separação da imagem em sub-imagens estes são raros e imperceptíveis no resultado final. Mais comprometedor é a situação de um mapa de profundidade com ruído, pois como o algoritmo depende especialmente deste dado de entrada, a sua má qualidade acarretará um resultado com menor qualidade em ambas as técnicas apresentadas [Bar05].

3. Programação em GPU

A implementação em hardware gráfico é a primeira escolha quando em causa está a rapidez de execução na geração de imagens computorizadas. Hoje, os GPUs oferecem, comparativamente aos CPUs, incríveis recursos quer para processamento gráfico quer não gráfico, como se pode verificar na Figura 20 [CUD08].

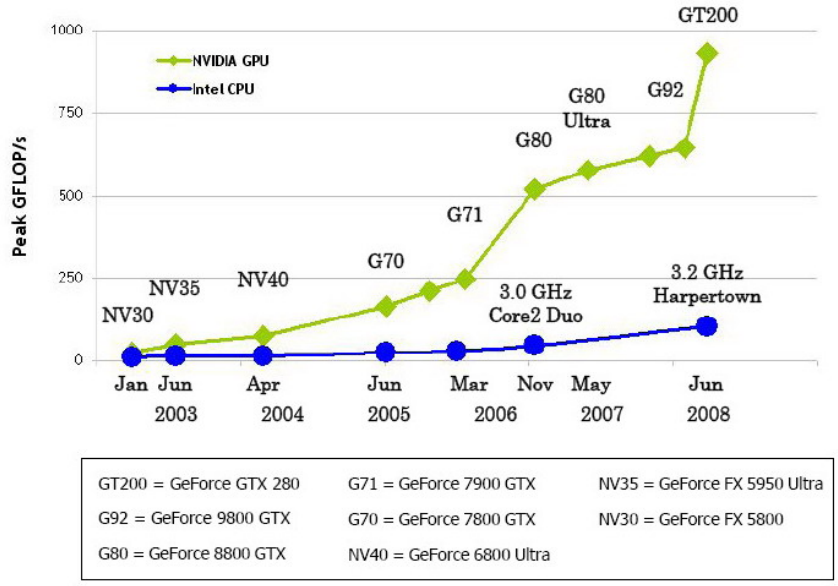


Figura 17 – Comparação de operações Floating-Point por segundo em CPU e GPU

O hardware gráfico está optimizado para máxima eficiência no processamento de imagens através de renderização e *shading*. Operações sobre imagens são efectuadas muito mais rapidamente num GPU em relação a um CPU convencional. Os CPUs estão desenhados para alto desempenho em operações sequenciais, contrariamente aos GPUs que são optimizados para um alto paralelismo no processamento. Em aplicações que processam grandes conjuntos de dados,

pode ser utilizado o modelo de programação paralela através da execução por *threads*. Adicionalmente, GPUs programáveis são caracterizados pelo seu custo reduzido, são prontamente disponíveis, facilmente actualizáveis e, sobretudo, compatíveis com múltiplos sistemas de operação e arquitecturas de hardware [Har02].

No passado, as grandes desvantagens dos GPUs eram designadamente a complexidade de programação e a falta de alta precisão em fragmentos de operações e em questões de armazenagem de dados. Estas duas desvantagens relacionavam-se porque o esforço em assegurar a precisão apontada resultava num acréscimo de dificuldade na programação [Har02]. Hoje a questão programática foi superada com o emergir das linguagens de programação de gráficos de alto-nível, como por exemplo GLSL [Ros06] ou CUDA da NVIDIA [CUD08].

3.1 Importância da Execução em GPU em Simulações Visuais

Uma simulação gráfica interactiva da visualização (humana ou qualquer modelo de câmara) engloba questões diversas a serem tratadas, onde o tempo de execução é o ponto capital. Sem taxas de tempo de execução ao nível de tempo-real, perde-se a interactividade pretendida, e quantas mais tarefas forem simultaneamente efectuadas, maior a dificuldade na procura de uma simulação visualmente correcta e eficiente.

A programação em GPU de uma simulação visual é suficiente para garantir que a mesma será executada nos tempos mais baixos permitidos pela tecnologia gráfica utilizada, tornando-se um aspecto quase imprescindível ao sucesso final da mesma. A renderização de imagens de uma simulação visual pode ser total ou parcialmente movida para o GPU.

No âmbito da recriação da profundidade de campo em aplicações gráficas com enorme interacção com utilizador, o aproveitamento das vantagens oferecidas pela programação em GPU é já um requisito obrigatório. Simulações de profundidade de campo, como os de [Ber04], [Kas06] ou [Zho07], são exemplos de implementações adaptadas a este modelo de programação em GPU. Por esta razão, a modelação do efeito de profundidade a desenvolver nesta Dissertação não poderia deixar ignorada toda esta questão. Estudos foram então efectuados no sentido de

uma compreensão global relativo ao funcionamento das melhores e mais recentes soluções de programação em GPU.

3.2 *Shading* por GLSL

3.2.1 *Shaders*, Razões e Vantagens

Um *shader* sintetiza um conjunto de instruções para criação de efeitos de renderização de uma imagem tridimensional sobre hardware gráfico, adicionando sombreamento e iluminação aos respectivos pixels impondo um resultado fotorrealista de uma forma mais flexível. As tecnologias de *shaders* e as técnicas possíveis através de *shading*, são extensamente utilizadas em jogos, e é já uma componente indispensável em qualquer aplicação gráfica envolvendo objectos 3D [Ros06].

Na sua essência, os *shaders* permitem programar estágios específicos do processamento gráfico directamente em GPU, e com um certo grau de paralelismo da sua execução é obtida uma maior flexibilidade na obtenção de efeitos superiores. Mas um *shader* apresenta a particularidade no seu processamento de apenas proceder à modificação de pixels se estes forem desenhados, ou seja, se tiver sido invocado um comando para desenhar uma imagem em ecrã, por exemplo. Cada pixel de uma imagem antes ser apresentado em ecrã é tratado pelo *shader* respectivo, ao ponto de na necessidade de nada desenhar, i.e., apresentar ecrã todo preto, um quadrilátero terá sempre de ser desenhado, neste caso com todos os pixels de cor preta.

Antes da existência de *shaders* as suas funcionalidades eram conseguidas através de programação na linguagem *Assembly*, tornando-se desde logo uma tarefa bastante complexa. Para ser superada esta enorme desvantagem foram criadas linguagens próprias para programação de *shaders* de maneira muito mais intuitiva, não afectando nenhum dos restantes processos até à renderização final de uma imagem sintética. Existem diversas interfaces para programação de *shaders* como a linguagem *shader* Cg, HLSL ou GLSL. Enquanto Cg foi desenvolvida pela NVIDIA, muito idêntica ao GLSL mas esta específica à integração com OpenGL, por sua vez a

linguagem HLSL (*High Level Shader Language*) foi lançada pela Microsoft como componente integrante do seu pacote de funcionalidade gráficas DirectX, adaptada à API Direct3D.

A programação de *shaders* pode hoje ser feita através de linguagens de um já considerável alto-nível como é o caso de GLSL com o estilo semelhante à linguagem C. Talvez o grande inconveniente durante a programação de *shaders* prenda-se com a evidente limitação nos processos de *debug* (depuração), na determinação e correcção de erros durante a execução da aplicação. Facto um pouco generalizado a qualquer modelo de programação paralelo. Duas das soluções mais consistentes nesta matéria são oferecidas através do *Shader Debugger* [Deb09], criado pela própria NVIDIA, e através do *glslDevil* [Str07], criado pela Universidade de Estugarda, Alemanha. A primeira suporta *debug* de pixel nas linguagens como HLSL e Cg, a segunda foi criada especificamente para GLSL, embora só actue sobre uma restrita lista de chamadas do OpenGL para desenho gráfico.

Vários efeitos obtidos a partir de *shaders* eram possíveis apenas por implementação em software mas sempre de uma forma limitada. A partir do momento em que os *shaders* possibilitaram implementações aproveitando a aceleração do hardware gráfico, o desempenho total de uma renderização foi drasticamente aumentado, e a utilização do CPU pôde ser atenuada executando desta forma outras tarefas [Ros06]. Facilmente com os, cada vez mais, sofisticados hardwares gráficos, mais complexas poderão ser as técnicas de renderização a partir de *shaders* e, destes, resultarem aplicações de tempo-real.

Para se ter uma ideia do que os *shaders* são realmente capazes, veja-se a seguinte pequena lista de efeitos possíveis [Ros06]:

- Aumentar o realismo de materiais como tintas, madeira, metais, pedra, etc.;
- Simulação avançada de fenómenos naturais como água, fogo, fumo, nevoeiro;
- Renderização avançada de efeitos como iluminação, *ray tracing*, etc.;
- Iluminação mais realística com superiores efeitos luminosos e sombras suaves;
- Manipulação de texturas dinamicamente geradas (2D ou 3D);
- Melhores efeitos de animação com interpolação de *frames*, sistemas de partículas, ou mesmo com procedimentos de definição de movimento;
- Processamento de imagem por convolução, aplicação de máscaras e filtros complexos.

3.2.2 GLSL

A linguagem de alto nível, para programação de gráficos, *OpenGL Shading Language*, também conotada por GLSL, é na verdade o conjunto de duas linguagens próximas. Permitem a criação de *shaders* para os processadores programáveis contidos no fluxo de processamento (*pipeline*) do OpenGL. As duas linguagens referidas para programação de *shaders* distinguem-se segundo o processador a que se destina: *vertex* ou *fragment*. O OpenGL oferece os mecanismos necessários à criação, à compilação e à ligação (*link*) dos respectivos *shaders*, formando o código executável denominado por Programa. Respeitando algumas limitações relativas ao tipo de *shader* em causa, estes podem receber dados de entrada enviados pela própria aplicação OpenGL.

A linguagem de alto nível GLSL é parte integrante do OpenGL, cruzando-se com esta durante a execução das suas aplicações. Refira-se que o OpenGL dispõe da sua própria API independente para a criação de imagens e gráficos 3D. A linguagem dos *shaders* é ligeiramente diferente do OpenGL, mas baseada igualmente na linguagem C/C++, com idêntico fluxo de controlo. É, no entanto, um pouco mais restrito em termos de tipos de dados e as funções são chamadas apenas para retorno de valores.

Antes de ser aprofundado o conhecimento específico à programação dos *vertex* e *fragment shaders*, o programador deverá compreender o *pipeline* de processamento gráfico. São assim verificados os contextos em que se inserem os distintos *shaders*, as suas capacidades e limitações globais. Por fim serão descritos separadamente os processadores relativos aos dois tipos de *shaders* existentes e suas capacidades.

Depois de compreendida toda a funcionalidade fixa do processamento gráfico, torna-se mais simples verificar as vantagens proporcionadas pela programação de *shaders* através de GLSL.

3.2.2.1 *Pipeline* Gráfico

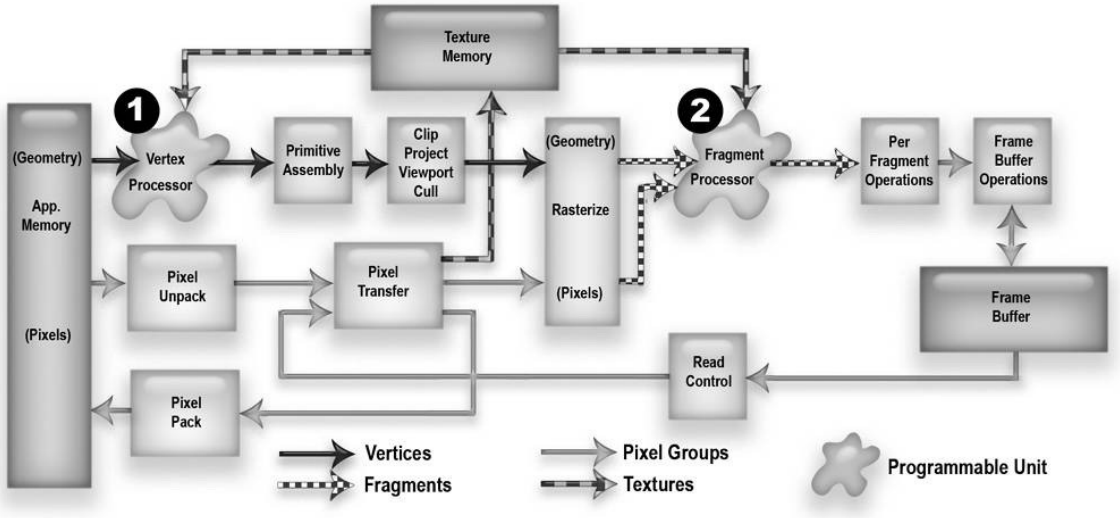


Figura 18 - *Pipeline* de processamento gráfico do OpenGL e integração dos processadores *Vertex* (1) e *Fragment* (2)

O diagrama acima demonstra como os processadores programáveis, definidos em GLSL, integram o fluxo global de processamento do OpenGL. Esta linguagem foi cuidadosamente desenhada para permitir implementações em hardware, realizando os processamentos de vértices e fragmentos de forma paralela. Pode ser visualizada na Figura 18, a localização exacta onde os *shaders* vão actuar durante o processo de renderização de uma imagem, desde o as instruções de desenho até ser guardada no *frame buffer* a informação final dos pixels a apresentar em ecrã.

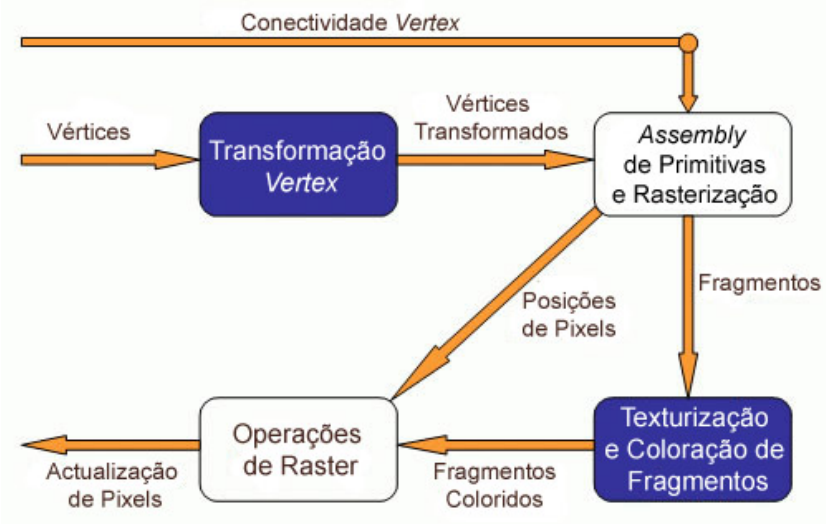


Figura 19 - *Pipeline* de processamento gráfico do OpenGL com alto nível de abstracção

A Figura 19 apresenta agora uma versão mais simplificada do *pipeline* gráfico, apresentada em [GLS09], com um superior nível de abstracção, onde são distinguidos os estágios mais importantes de processamento, e os respectivos dados que se movem através dos mesmos. São eles:

- **Transformação *Vertex*:** No contexto do GLSL, um *vertex* é todo um conjunto de atributos associados a cada vértice dos objectos a desenhar. Atributos como a posição no espaço, a cor, a normal, as coordenadas de texturas entre outros. Nesta fase são efectuadas operações sobre cada *vertex* como transformações posicionais, adição de iluminação ou mesmo geração e transformação de coordenadas de texturas.

- ***Assembly* de Primitivas e Rasterização:** Vértices transformados e toda a informação de conectividade entre os mesmos são os dados de entrada neste estágio. É aqui que todas as primitivas são agregadas para serem executadas. A Rasterização determina os fragmentos (informação de cor, normal, texturas, etc.) e posições de pixels por primitiva. Um fragmento (ou *fragment*) pode ser então definido como uma porção de dados que será usada na determinação ou actualização da cor de um certo pixel final do *frame buffer*. São computados de forma apropriada os atributos de cada *fragment*, através da informação de conectividade *vertex* computados na fase "Transformação *Vertex*". Os dados de saída nesta fase será uma das duas seguintes hipóteses:
    - A posição dos fragmentos no *frame buffer*;
    - Os valores interpolados para cada *fragment*.

- **Texturização e Coloração de Fragmentos:** Nesta fase é determinada a cor final de cada fragmento. A cor de um pixel, anteriormente processada através da interpolação, pode agora ser combinada com um *texel* (unidade de textura). Da mesma forma, poderão ser manipulados para cada pixel valores referentes à profundidade ou quantidade de nevoeiro (primitiva *Fog*), por exemplo.

- **Operações de Raster:** A última fase do fluxo de processamento gráfico é naturalmente a rasterização final, a partir da localização dos pixels e dos valores de cor e profundidade por *fragment* como dados de entrada. São executados uma série de testes por *fragment* (*Scissor*, *Alpha*, *Stencil*, *Depth*) e a sua informação é então usada para actualizar o valor de cada pixel de acordo com o corrente modo *blend*. Este é o modo que define a forma como é fixado o novo valor do respectivo pixel no *frame buffer*, que só nesta fase pode ser realmente acedido e alterado.

3.2.2.2 Processador *Vertex*

O processador *Vertex* representa a unidade programável que opera sobre os vértices e todos os dados a estes associados. Os *vertex shaders* são executados neste processador depois de compilados e ligados (*link*). Os dados de entrada para *shaders* deste tipo podem ser posição de vértices, cor, normais entre outros, dependendo da informação enviada da própria aplicação OpenGL [Kes06].

Os *vertex shaders* podem, por exemplo, executar tarefas relacionadas com:

- Transformações de posições usando as matrizes de projecção e vista de modelo;
- Transformações da normal;
- Transformação e geração de coordenadas de texturas;
- Iluminação por *vertex*;
- Cor.

Refira-se, no entanto, que as capacidades mencionadas apenas são aproveitadas se utilizado um *vertex shader*. Nesta situação toda a funcionalidade fixa do OpenGL, relativa às tarefas referidas, irá ser substituída pelo *shader*.

O processador apenas tem acesso ao estado actual do OpenGL, daí as suas naturais limitações, como por exemplo o não acesso ao *frame buffer* neste estágio do processamento

gráfico. O processador opera um *vertex* de cada vez, não substituindo operações gráficas que utilizem informação de vários vértices ao mesmo tempo [Kes06].

3.2.2.3 Processador *Fragment*

O processador *Fragment* opera com base em *fragments*, seus valores e dados associados. Este processador é designado a compilar e executar *fragment shaders*.

Num *fragment shader* apenas é permitido o acesso às coordenadas dos pixels de um fragmento, invalidada qualquer possibilidade de alteração e transformação das mesmas. Não é permitido também o acesso a fragmentos vizinhos, i.e., apenas um fragmento é operado de cada vez neste processador usando unicamente a sua informação. Os dados de entrada são os valores interpolados resultantes da fase anterior do fluxo de processamento gráfico, como por exemplo as normais, cores e posições de *vertex*.

Neste processador podem ser executadas, por exemplo, tarefas relacionadas com:

- Computação de cor, profundidade e coordenadas de texturas por pixel;
- Manipulação e aplicação de texturas;
- Computação do atributo Nevoeiro.

A computação realizada num *fragment shader* tem a finalidade de, posteriormente à sua execução, actualizar valores em memória de pixels de texturas ou *frame buffer*, dependendo do tipo de comando OpenGL invocado e do respectivo estado global OpenGL [Kes06].

Neste processador, e de forma análoga ao processador *Vertex*, só na existência de um *fragment shader* será anulada a mesma funcionalidade suportada pelo OpenGL. Caso o *shader* exista e seja executado então todos os efeitos desejados têm de ser implementados por completo no *shader*.

3.3 CUDA

Antes de surgir este novo modelo de programação em hardware, que expõe o GPU como um autêntico dispositivo genérico de computação paralela de dados, apresentava-se problemática a programação em GPU. Esta tinha de ser efectuada através da API de hardware gráfico, intrincado de dominar e inadequado para aplicações não gráficas. O CUDA (*Compute Unified Device Architecture*), da NVIDIA [CUD08], veio solucionar as contrariedades passadas com um estilo de programação bem mais flexível através da sua API. Esta trata-se de uma extensão à linguagem de programação C, com uma linha de aprendizagem mínima, óptima principalmente para iniciados, levando ao futuro aumento de desenvolvimento da computação em GPU. Com esta nova tecnologia a gestão do acesso ao GPU é deixada à responsabilidade do mecanismo de *multi-casting* do sistema operativo, por meio de diversas aplicações CUDA/gráficas, executadas concorrentemente [CUD08].

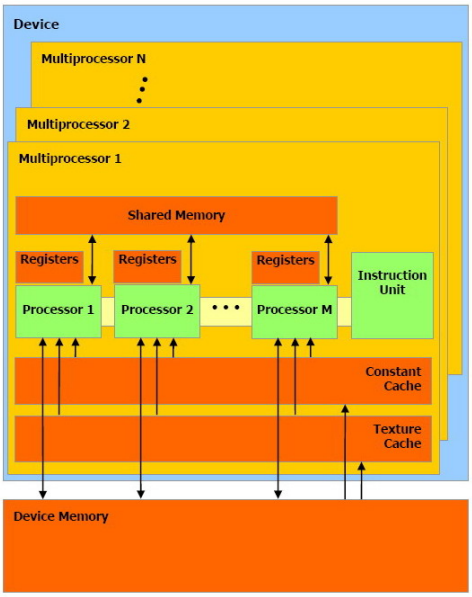
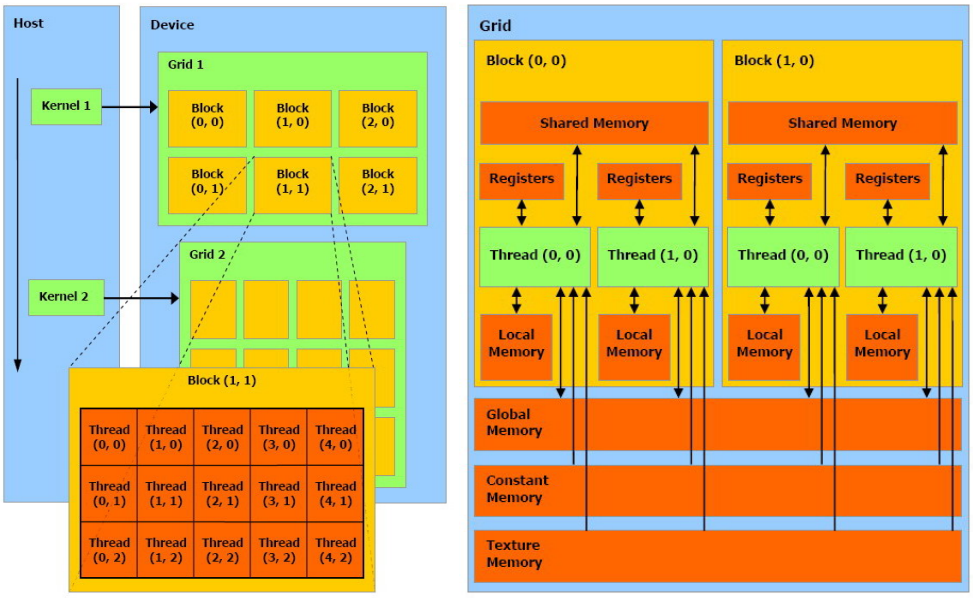


Figura 20 - Organização de memória do GPU e multiprocessadores, usando CUDA

Sabe-se à partida que CUDA oferece o GPU como um co-processador de dados paralelizados para o CPU. Mas aprofundando a análise à arquitectura e funcionamento do CUDA verifica-se que neste âmbito o GPU não é mais que num conjunto de multiprocessadores com arquitectura SIMD, Figura 20.



a) **b)**

Figura 21 - a) Modelo de programação e execução do CUDA. b) Modelo de memória interna do CUDA

No contexto do CUDA e seguindo o esquema da Figura 21-a), o CPU é designado por *host* e o GPU por *device*, este último capaz de executar funções denominadas por *kernel*. Cada *kernel* tem à sua disposição um número definido de *threads*, que por sua vez estão agrupados em blocos (*blocks*), estes são ainda organizados por grelhas (*grids*). Variadíssimos aspectos nascem desta arquitectura como a possível sincronização e partilha de memória entre *threads* dentro do mesmo bloco, mas o mesmo não acontece entre diferentes blocos, Figura 21-b). A invocação de um *kernel* não bloqueia de forma alguma a execução no *host* e assim ambos podem executar as suas rotinas paralelamente. Por fim a sincronização entre *device* e *host* é realizada implicitamente quer em leitura ou escrita em memória GPU, sabendo que esta só pode ser acedida se nenhum *kernel* estiver activo, facto que implica o bloqueio do CPU até que nenhum *kernel* esteja a ser executado [Jen08].

Foi visto anteriormente que a execução de uma aplicação simulatória, como a da profundidade de campo, pode ocorrer totalmente em GPU, mas com o CUDA poderão ser executadas apenas porções da aplicação consoante a necessidade. A capacidade do CUDA é estendida a um patamar em que uma função, localizada com código executado com elevada redundância, pode ser individualmente carregada para o GPU e processada a partir de muitos *threads*, definindo-se a função em questão como um *kernel*.

3.4 Compute Shader

A Microsoft, numa medida para reforçar a sua forte posição no mundo do GPGPU (uso do GPU para programação geral e não exclusivamente gráfica), irá lançar o conceito tecnológico *Compute Shader* como parte integrante do pacote DirectX 11. No entanto é dada a noção que o *Compute Shader* não terá um propósito tão generalizado quanto a solução mais forte no mercado, o CUDA da NVIDIA. Isto porque a sua apresentação e publicação, pelo qual se baseia esta análise [Boy08], foi realizada numa importante conferência de jogos a nível mundial, o *Gamefest*. Foram focadas variadas maneiras de como melhorar a experiência de jogabilidade, tornando a física dos jogos e efeitos mais realistas e oferecendo uma avançada componente para pós-processamento de imagem. Outra razão para se adivinhar uma maior incidência desta tecnologia no domínio dos jogos é o facto de se saber, à partida, que o DirectX é um pacote de APIs que facilitam o desenvolvimento de jogos. Interface gráfica capaz de padronizar as comunicações entre software e hardware na integração de suas aplicações com o sistema operativo da Microsoft, o Windows. Muitas vezes é referido o DirectX como Direct3D, uma das APIs do DirectX, por ser talvez a mais importante no desenvolvimento de jogos e certamente a mais publicada.

Numa breve alusão às funções vantajosas deste novo modelo de programação e processamento para GPU, não será obrigatório desenhar um quadrado como o era usando simples *shaders*. O *Compute Shader* permitirá, por exemplo, invocar *threads* conforme a necessidade existindo zona de memória partilhada entre os mesmos. Partilhados serão também os registros entre *threads* aliviando a pressão computacional sobre os mesmos, ao reduzir processamento redundante ou mesmo a quantidade de operações I/O. A disponibilização de estruturas de dados mais flexíveis e a permissão de acessos *input/output* desprovidos de qualquer ordem são factores valorizadores deste novo atributo do DirectX 11 [Boy08]. O *Compute Shader* executa as suas rotinas numa ajustada integração com o Direct3D e seus recursos.

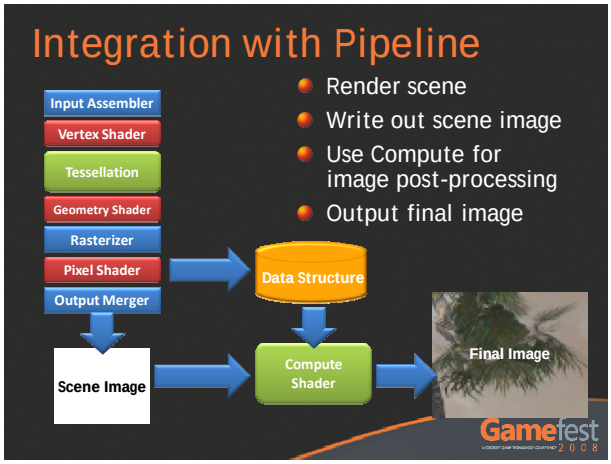


Figura 22 – Integração do *Compute Shader* no normal fluxo de processamento gráfico

Podemos visualizar na Figura 22 a forma como o *Compute Shader* irá funcionar. No normal fluxo de processamento gráfico é notória a habilidade de serem extraídas estruturas de dados, de acordo com a informação relativa aos pixels. O *Compute Shader* processa, finalmente, os pixels antes da imagem resultante ser devolvida ao restante processamento gráfico por defeito do Direct3D.

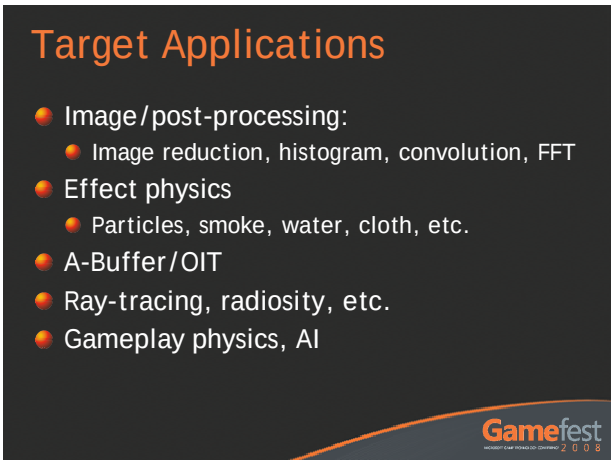


Figura 23 – Aplicações alvo do *Compute Shader* da Microsoft

Foram conduzidos esforços pela Microsoft, com o *Compute Shader*, no sentido de melhorar qualquer pós-processamento a realizar sobre cada imagem (Figura 23), por onde passa também algumas das melhorias visuais na jogabilidade de jogos futuros. E neste contexto se insere o tema desta Dissertação, ou seja, a simulação de profundidade de campo por Pós-

Processamento. O seu desempenho pode ser extremamente beneficiado se aproveitadas as capacidades evidenciadas pelo *Compute Shader* durante a devida implementação. Um alto desempenho em técnicas de filtragem como convolução (Figura 24), poderá facultar a criação de profundidade de campo mais rapidamente. A disponibilização de mais tempo para execução 3D significa a possibilidade de aumentar a qualidade final do efeito pretendido.

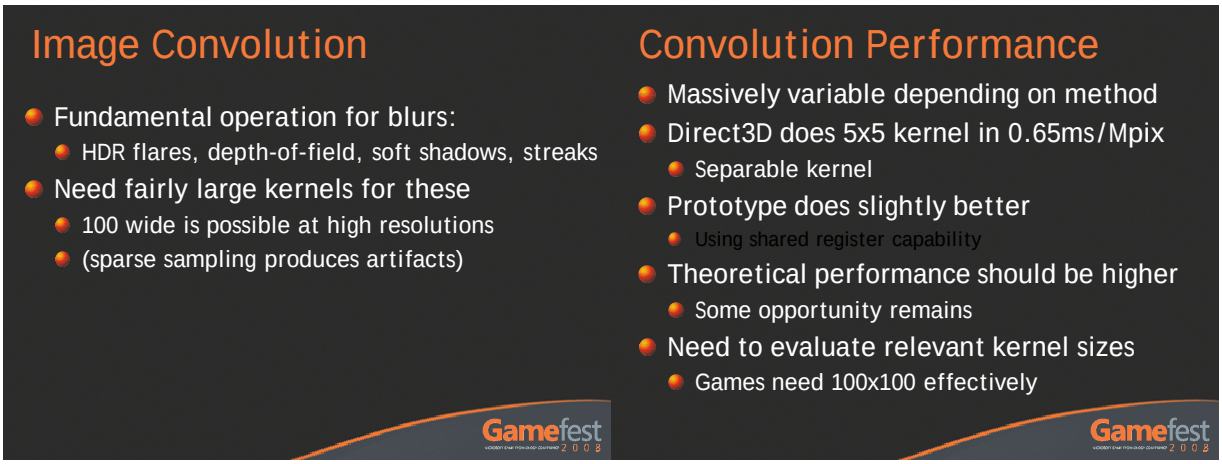


Figura 24 – Convolução em *Compute Shader* e sua performance prevista

O *Compute Shader* possui a grande capacidade de escalar o paralelismo existente numa aplicação (neste caso se desenvolvido com DirectX) às capacidades gráficas do hardware onde a mesma é executada, sem a necessidade de mudanças de código muitas vezes complicadas. De um modo global, o que o *Compute Shader* oferece acima de tudo é flexibilidade na programação em GPU prevendo-se, a curto ou médio prazo, que efeitos físicos acelerados por GPU se tornem uma norma em aplicações gráficas como jogos, e assim favorecido ainda mais a difusão da tecnologia DirectX.

3.5 Conclusão

Depois de uma primeira abordagem a este problema, apenas duas hipóteses foram consideradas, GLSL e CUDA. Um estudo mais aprofundado sobre GLSL foi realizado desde o início dos trabalhos, principalmente pela sua utilização na técnica de profundidade de campo [Zho07] no qual se baseou a simulação implementada nesta Dissertação. Posteriormente foram analisadas as características do CUDA e que vantagens poderiam oferecer em comparação ao

GLSL. Já *Compute Shader* torna-se, naturalmente, uma tecnologia a ser investigada em trabalho futuro depois de o DirectX 11 da Microsoft ser lançado no mercado.

Concluiu-se que o desempenho será sempre optimizado se bem aproveitadas as potencialidades disponibilizadas pelo CUDA de acordo com os requisitos da aplicação. Mostra-se também ser especialmente indicado para manipulação de imagens, com o mapeamento de blocos de imagens e pixels para *threads* de processamento paralelo. Globalmente a utilização de CUDA insere-se perfeitamente no contexto desta Dissertação, sendo uma excelente solução para aceleração da aplicação em GPU. A possibilidade CUDA acabou por ser descontinuada nos trabalhos desta Dissertação por limitações temporais à conclusão da mesma.

Mesmo descartada a implementação integral através de CUDA, foi investigada a possibilidade de serem combinadas as suas capacidades com GLSL. Embora seja possível integrar ambos, nenhuma clara vantagem foi verificada no uso de CUDA para melhoramento da execução de *shaders*. As duas técnicas de programação em GPU estão indicadas, no fundo, para executarem o mesmo procedimento paralelo de filtragem de todos os pixels de uma imagem.

Conclui-se que são obtidos resultados óptimos se utilizado apenas GLSL ou apenas CUDA na modelação da profundidade de campo. Esta última possibilidade foi deixada para trabalho futuro, explorando vantagens oferecidas pela própria plataforma como a utilização de distintos objectos *kernel*, memória partilhada, sincronização de *threads* ou ainda a comunicação entre processos. Contudo, nenhuma garantia é dada para que melhor desempenho seja obtido directamente pelo uso de CUDA. Uma comparação realizada por [Won08] entre CUDA e *shaders* para computação geral em GPU (GPGPU), mostrou que, à parte das diferenças inerentes às respectivas arquitecturas, teoricamente o CUDA é tão eficiente como *shaders*.

4. Implementação da Profundidade de Campo

Considerando todo o estudo efectuado e apresentado nos pontos anteriores deste documento, apresenta-se agora a metodologia e sua implementação para simulação de profundidade de campo. Descrevem-se de forma aprofundada as tecnologias e técnicas de programação utilizadas, assim como outros inúmeros detalhes específicos à implementação.

Esta secção focará aspectos relacionados com:

- A técnica usada para modelação da profundidade de campo com revisão do processo de filtragem;
- A biblioteca gráfica OpenGL na qual foi criada a aplicação simuladora. Apresentação de todos os aspectos importantes à execução da metodologia considerada, como por exemplo o uso de objectos FBO;
- O fluxo de processamento da metodologia com análise pormenorizada de todas as fases de execução da simulação. Pontos como configurações, integração de *shaders* e OpenGL, ou renderização por FBO serão explanados;
- Optimizações realizadas;
- Interface gráfica ao dispor do utilizador do simulador de profundidade de campo.

4.1 A Metodologia

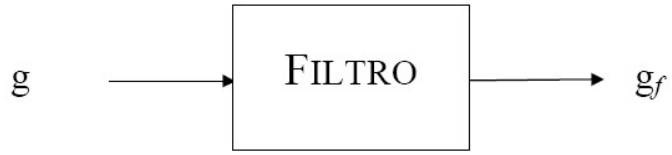
Analisadas e comparadas cinco das melhores e mais eficazes técnicas Pós-Processamento para simulação da Profundidade de Campo (subsecção 2.4 deste documento), determinou-se a implementação da profundidade de campo baseada no método sugerido por Zhou *et. al.* [Zho07].

Assim a metodologia apresentada passa essencialmente pela aplicação de um filtro de passagem dupla sobre uma imagem 2D, captada através da teórica existência de uma câmara fotográfica e suas esperadas propriedades. Durante a aplicação do filtro são modificados todos os pixels de uma imagem segundo a teoria óptica estudada (subsecção 2.2.2 deste documento). Para uma perfeita compreensão da metodologia implementada, serão dissecadas todas as particularidades do filtro referido.

4.1.1 Filtragem de Imagens

No contexto desta Dissertação, referir que se obtém o efeito de profundidade de campo através de filtragem quer dizer, na realidade, que a imagem originalmente renderizada será processada e recriada segundo os parâmetros definidos segundo um dado filtro. Qualquer representação gráfica de um objecto ou cena tridimensional apresenta-se sob a forma de uma imagem de duas dimensões, sendo o pixel a sua menor unidade. Sabendo isto, pode-se definir a filtragem de uma imagem como a modificação dos seus pixels baseada numa função sobre os respectivos vizinhos locais. Desta forma um pixel "filtrado" tem sempre um valor dependente do contexto em que ele se encontra na imagem original.

Uma das formas de implementação deste tipo de filtragem, e também a mais utilizada em sistemas de processamento de imagens, é denominada por Convolução. Trata-se de uma simples operação matemática que faculta uma maneira muito específica de multiplicar uma matriz (imagem a ser tratada) por outra de tamanho mais reduzido (máscara ou *kernel*). O esquema de filtragem por Convolução pode ser verificado na Figura 25.



$g_f = g * h$ onde:

$g_f \rightarrow$ imagem filtrada
$g \rightarrow$ imagem original
$h \rightarrow$ máscara
$* \rightarrow$ Convolução

Figura 25 – Filtragem por Convolução

$$g_f(l,p) = \sum_{m=-int(w/2)}^{int(w/2)} \sum_{n=-int(w/2)}^{int(w/2)} g(l+m, p+n)\, h(m,n)$$

Equação 7 – Fórmula matemática para Convolução

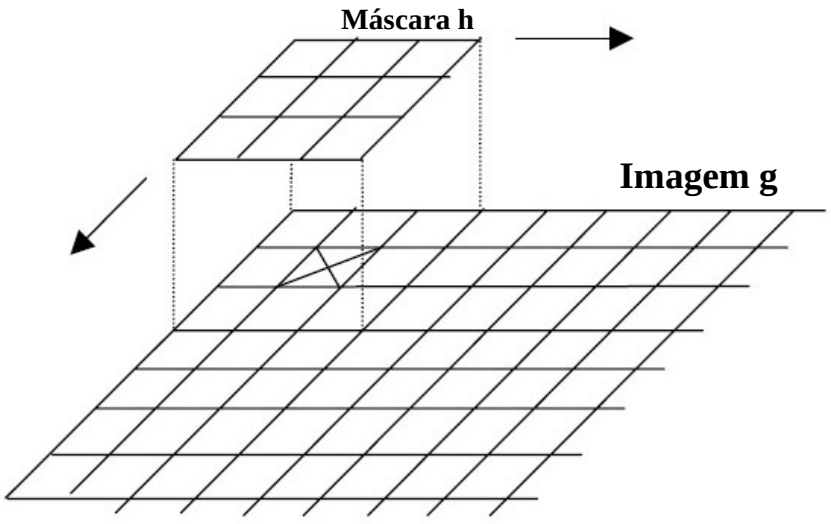


Figura 26 – Processo de Convolução

A Convolução discreta pode ser descrita segundo a formulação matemática apresentada na Equação 7. Já na Figura 26 verifica-se o mecanismo para realização de convolução de *h* sobre *g*, onde cada pixel da imagem (marcado com "X") irá possuir um novo valor resultante da soma ponderada dos seus pixels vizinhos segundo contribuições estabelecidas pela máscara usada. O termo "filtro" é atribuído às próprias máscaras usadas na filtragem, sempre de dimensões ímpares, dimensão 3 no exemplo da Figura 26. Existem variadíssimos tipos de filtros que podem ser utilizados numa operação de convolução, como por exemplo os três exemplos apresentados na Figura 27.

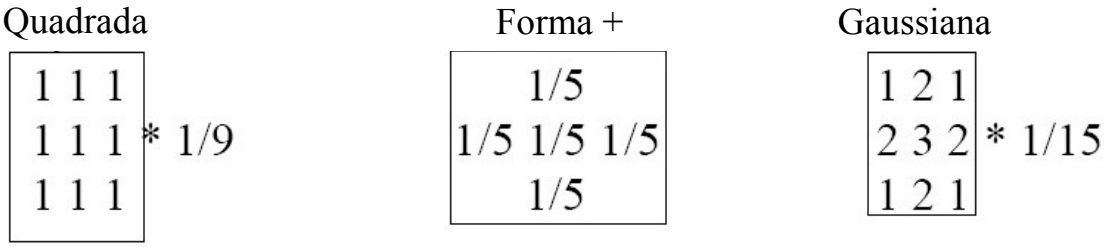


Figura 27 – Exemplos de máscaras aplicadas em Convolução

No contexto desta Dissertação irá ser utilizado um filtro de tipo semelhante ao Gaussiano, mais concretamente um filtro Gaussiano separável. Uma distribuição Gaussiana em uma dimensão (1D) tem a forma evidenciada na Equação 8, onde σ é o desvio padrão da distribuição. Foi já demonstrado, pelas Equações 8 e 9, que uma distribuição Gaussiana em 2D é exactamente equivalente à multiplicação das duas distribuições 1D das direcções vertical e horizontal, ou seja, é considerada separável. Esta característica permite que o filtro seja aplicado à matriz de pixels (imagem 2D) interpretada de maneira horizontal (sequência de linhas de pixels), e posteriormente aplicado à mesma matriz horizontalmente interpretada (por colunas de pixels). A sua propriedade comutativa permite a ordem ser invertida sem alteração do resultado final.

$$g_{1D}(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

Equação 8 – Distribuição Gaussiana 1D

$$g_{2D}(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \left(g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \right) \times \left(g(y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}} \right) = g_{1D}(x) \times g_{1D}(y)$$

Equação 9 – Separabilidade da Distribuição Gaussiana 2D

4.1.2 Filtro de Passagem Dupla

A modelação do efeito de profundidade foi alcançado através da aplicação de um filtro de passagem dupla, semelhante a um típico filtro Gaussiano separável, sobre uma imagem 2D renderizada. Este tipo de filtro, apresentado por Zhou *et al*. em [Zho07], permite a amostragem de um maior número de pixels comparativamente com filtros de uma única passagem que facilmente evidenciam imperfeições visuais associados a *aliasing*.

Uma primeira passagem sobre a imagem original é efectuada, numa direcção pré-determinada (horizontal ou vertical). A imagem resultante desta primeira passagem será a imagem de entrada para a segunda, ou seja, a segunda fase deste processo de filtragem será feita sobre os pixels resultantes da primeira passagem, na direcção que não a mesma da primeira passagem.

Os pixels são processados de forma idêntica em ambas as passagens. Cada pixel (central) é determinado através da soma ponderada dos valores dos pixels vizinhos segundo três pesos específicos analisados em baixo. Por fim, a soma total é normalizada ao ser dividida pela contribuição total dada pela vizinhança do pixel central.

Os pesos atribuídos aos pixels vizinhos (amostrados), aquando do processo de filtragem, regem-se segundo três factores de controlo: o factor da sobreposição; o da intensidade luminosa; e o da fuga de intensidade. No cálculo relativo aos três factores, o peso é dado entre os valores zero e um, atribuindo assim contribuições entre o nulo até um total obtido por multiplicação dos três factores.

O factor da sobreposição, dado pela Equação 10 e visualizado na Figura 28, define a contribuição que o Círculo de Confusão de um pixel amostrado P dá ao pixel central C, aquele a ser calculado. Um pixel vizinho P contribuirá totalmente para o pixel C se o raio do seu Círculo de Confusão r_p for maior ou igual à distância que separa ambos. O mesmo pixel P não contribuirá de forma alguma para C se o raio r_p for menor que a distância de separação. E no caso restante, a diferença entre a distância dos dois pixels e o Círculo de Confusão de P define a contribuição deste último para C.

$$O(r_p) = \begin{cases} 0, & r_p \leq d_p \\ r_p - d_p, & d_p \leq r_p < d_{p+1} \\ 1, & r_p \geq d_{p+1} \end{cases},$$

Equação 10 – Função de sobreposição para cálculo de peso a aplicar sobre cada pixel

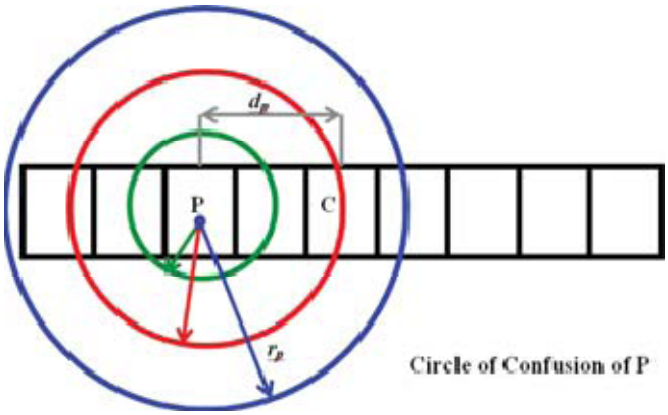


Figura 28 – Contribuição de pixel P para o pixel com centro em C, segundo diferentes círculos de confusão

O segundo factor, calculado a partir da Equação 11, refere-se à intensidade da luz sobre cada pixel. Proposto por Zhou *et. al.* baseando-se em [Che87], é usado um diferencial de intensidade uniforme pelo Círculo de Confusão do pixel amostrado em causa, resultando a função de intensidade luminosa na proporção do inverso do quadrado do seu raio [Zho07].

$$I(r_p) \propto \frac{1}{r_p^2}$$

Equação 11 – Função de intensidade luminosa para cálculo do peso a aplicar sobre cada pixel

Por fim, o terceiro e último factor a pesar na filtragem de cada pixel da cena sintética tridimensional é o de dispersão de cor, isto é, a fuga de intensidade de pixels em situações de oclusão entre objectos. A contribuição segundo este factor é determinada pela função definida na Equação 12, que ajusta o peso do pixel amostrado P, por um factor baseado no tamanho do Círculo de Confusão do pixel central, se P se encontrar mais longe da câmara que o plano de focagem. Garante-se assim que objectos para além do plano focado não contribuem para objectos focados evitando artefactos por *intensity leakage*. Por outro lado P contribui com a sua cor por inteiro se estiver a uma distância da câmara menor ou igual que o plano focado. Isto garante que os pixels amostrados contribuem sempre na desfocagem de pixels mais distantes em profundidade (*background*) se estiverem antes do plano em foco.

$$L(z_p) = \begin{cases} \propto r_c, & z_p > z_f \\ 1 & z_p \leq z_f \end{cases}$$

Equação 12 – Função de controlo de *intensity leakage* sobre cada pixel

Sintetizando, a cor de cada pixel (central) é misturada segundo o produto dos pesos atribuídos por cada um dos três factores revistos representado na Equação 14. De notar que o peso assinalado ao pixel central resume-se simplesmente pela sua intensidade luminosa (Equação 13). Depois de somadas todas as contribuições dos pixels vizinhos amostrados ao pixel central, é finalmente renormalizado o seu valor, ou seja, é dividido pelo total de contribuições somadas. Finalmente garante-se que o resultado final respeita os limites definidos e aceites pelo hardware gráfico, remetendo então a cor final do pixel central para o *frame buffer* [Zho07].

$$W(C) = I(r_c)$$

Equação 13 – Contribuição do pixel (central) a calcular no *shader*

$$W(P) = O(r_p) \times I(r_p) \times L(z_p)$$

Equação 14 – Função de cálculo da contribuição de cada pixel amostrado segundo a contribuição de três factores

4.2 OpenGL

O algoritmo para renderização de profundidade de campo apresentado foi implementado usando o OpenGL (*Open Graphics Library*), uma interface de software para dispositivos de hardware gráfico [Sob03]. A sua API (*Application Programming Interface*) disponibiliza inúmeros comandos distintos que permitem criar aplicações tridimensionais interactivas. O OpenGL tornou-se realmente uma biblioteca padrão na indústria de desenvolvimento de aplicações, por razões como:

- O razoável nível de aprendizagem proporcionado pelo OpenGL, utilizando linguagem de programação de alto nível (C/C++);
- OpenGL está hoje bastante desenvolvido, disponibilizando muita documentação;
- Extrema estabilidade nos seus processos, apresentando resultados visuais muito satisfatórios.

O OpenGL fornece acesso a, praticamente, todos os recursos de qualquer hardware gráfico pois as suas funcionalidades foram desenvolvidas de forma a serem independentes da interface gráfica usada. Isto significa que uma aplicação OpenGL pode ser executada em múltiplas plataformas de hardware gráfico sem qualquer alteração de código.

Internamente o OpenGL é uma máquina de estados, determinando quando e o que o OpenGL deve realizar. Todas as suas características, como cor, posição, luz, propriedades de materiais, transformações, objectos desenhados, projecções, etc., são configuráveis através de variáveis ou modos de estado [Sob03]. O programador tem controlo total sobre todos os modos e estados existentes numa aplicação por si criada e, naturalmente que, estes só terão efeito caso estejam ligados, possibilitada ainda a sua modificação em qualquer estágio do processamento.

O fluxo de rotinas do OpenGL obedece à execução de uma certa ordem de operações ou estágios conhecido como o *pipeline* de renderização do OpenGL, revisto na subsecção 3.2.2.1 deste documento.

É comum verificar as aplicações OpenGL aproveitarem as funcionalidades disponibilizadas pela biblioteca GLUT na implementação de um sistema de janelas

independente. Esta oferece um conjunto de ferramentas especificamente desenhadas para escrita de programas gráficos OpenGL.

Uma aplicação OpenGL possui um mínimo de requisitos para a sua perfeita execução, criando o chamado "contexto OpenGL" onde deve ser respeitada a estrutura básica que passa pela execução dos seguintes passos:

- Configuração e abertura da janela da aplicação;
- Inicialização dos estados no OpenGL;
- Registro das funções de "*callback*";
- Renderização;
- Redimensionamento;
- Definição de entradas de dados: teclado, rato, etc.;
- Entrada no *loop* de processamento de eventos OpenGL.

4.2.1 Frame Buffer Object

A extensão do OpenGL *Frame Buffer Object*, também conhecida por FBO, representa a técnica de programação usada para definir o destino de desenho de uma renderização que não o *frame buffer* que o OpenGL disponibiliza. Um *frame buffer* não é mais que um conjunto de *buffers* lógicos (*color*, *depth*, *stencil* e *accumulation*), juntamente com o estado que define onde se fará a renderização de saída do OpenGL. A funcionalidade FBO necessita de uma biblioteca de extensão ao OpenGL, o GLEW (*OpenGL Extension Wrangler Library*), ajudando na pesquisa e carregamento de extensões OpenGL durante a execução da aplicação.

A grande vantagem na utilização de FBO prende-se com a natural melhoria de desempenho de aplicações que necessitem de aceder à informação da imagem renderizada. Por exemplo, uma aplicação para criação de efeitos visuais, como acontece nesta Dissertação, necessitará de processar a informação a partir da imagem previamente renderizada. No caso do OpenGL é realizada uma cópia dos dados contidos no *frame buffer* para texturas, através das funções *glCopyTexImage2D* ou *glCopyTexSubImage2D*. A utilização de FBO permite evitar esta

sequência de acções e guarda, no acto da renderização da cena 3D, a informação relativa a cor ou profundidade, etc., directamente em texturas inicialmente definidas. A diminuição de comandos resulta num menor uso de memória e numa maior rapidez de processamento [FBO06]. Este processo funciona como uma espécie de renderização *offscreen*, i.e., não perceptível ao utilizador, ocorrendo antes de qualquer amostra de resultados na janela da aplicação.

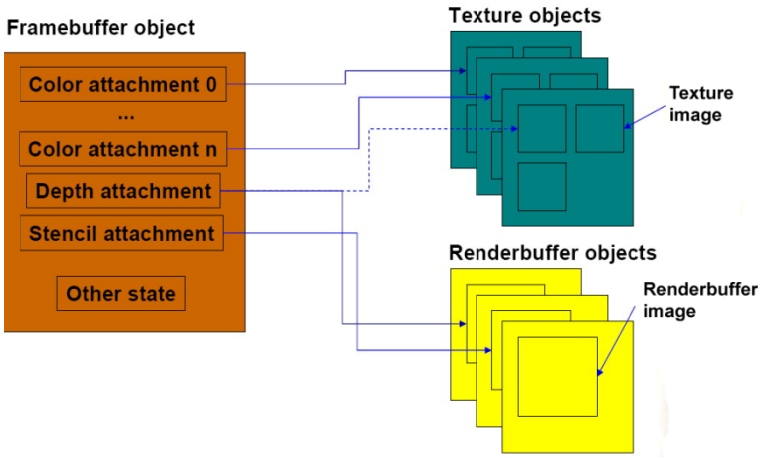


Figura 29 – Arquitectura *Frame Buffer Object*

A Figura 29 esquematiza a arquitectura FBO. Um objecto FBO necessita de ter definida a imagem ou imagens a si anexadas, por meio de texturas ou de um outro tipo de objecto OpenGL, o *Renderbuffer*. O propósito de um *Renderbuffer* é o encapsulamento de uma única imagem 2D, semelhante a uma textura na medida em que ambos podem ser independentemente alocados e partilhados dentro de múltiplos contextos [FBO06]. Um objecto *Renderbuffer* guarda os dados de pixels resultantes de uma renderização, correspondente ao *buffer* lógico anexado (*depth* ou *stencil*), e não pode ser usado como texturas. Estas associam-se normalmente ao *buffer* de cor por permitirem a necessária leitura em processamento posterior.

Esta arquitectura de execução torna-se mais eficiente no aproveitamento das vantagens FBO porque os recursos são assim partilhados dentro de um mesmo contexto. Oferece flexibilidade de programação controlando o que renderizar para que *buffers* durante execução da aplicação gráfica. Um único objecto FBO consegue guardar, se exigido, toda a informação associada aos diferentes *buffers* lógicos do OpenGL no acto da renderização 3D.

Um exemplo de utilização de FBO numa aplicação OpenGL poderá ser a seguinte:

- Criar um objecto FBO;
- Ligar o *color buffer* a uma textura;
- Ligar o *depth buffer* a um *Renderbuffer*;
- Renderizar uma cena tridimensional directamente para os *buffers* do objecto FBO criado;
- Processar um efeito visual sobre a imagem desenhada na textura de cor, a partir da informação contida nos *buffers* ligados ao objecto FBO;
- Renderizar a textura de cor resultante. Um pixel *shader* poderá ser utilizado para este efeito.

A utilização de objectos FBO demonstra-se então muito vantajosa na programação de aplicações gráficas que envolvam texturas dinâmicas, técnicas multi-passo (*anti-aliasing*, *motion blur*), efeitos por processamento de imagens (filtragem como a profundidade de campo), GPGPU entre outros.

4.3 Fluxo de Processamento e Integração com OpenGL

A implementação da simulação de profundidade de campo nesta Dissertação, baseia-se na técnica sugerida por Zhou *et al.* em [Zho07]. Esta faz proveito, essencialmente, das capacidades de renderização tridimensional do OpenGL e suas bibliotecas (GLU e GLUT).

O OpenGL fornece um vasto conjunto de funções que permitem gerir praticamente todos os recursos oferecidos pelo hardware gráfico, podendo ser complementado pelas bibliotecas GLU e/ou GLUT. A biblioteca GLU (*OpenGL Utility Library*) disponibiliza ao programador rotinas de alto-nível para desenho gráfico, que o próprio OpenGL não dispõe. Já a biblioteca GLUT (*OpenGL Utility Toolkit*) permite a criação de simples interfaces para interacção com as aplicações OpenGL, podendo gerir as mesmas entre diferentes janelas. A API da biblioteca GLUT caracteriza-se pela sua portabilidade, facilitando a execução de um programa OpenGL em qualquer PC independentemente do sistema operativo presente.

No contexto desta implementação, são requeridas ainda as bibliotecas GLUI e GLEW. A biblioteca GLUI é baseada na GLUT e permite que uma aplicação OpenGL seja controlada interactivamente através de botões, selectores, escolhas múltiplas e *spinners*, de forma totalmente independente da janela principal. Já a biblioteca GLEW, não é mais que uma plataforma cruzada C/C++ que ajuda na pesquisa e carregamento de extensões do OpenGL.

A modelação da profundidade de campo aqui exposta é reconhecida como Pós-Processamento, pois toda a criação do efeito de profundidade é executada depois da renderização da cena tridimensional. Para que se tenha uma noção, o mundo 3D em que ocorre a simulação não tem qualquer limitação, podendo ser, por exemplo, um simples cubo de uma certa cor, ou mesmo uma sala de museu com imensos objectos, texturas, materiais e iluminações diversas.

Veremos nesta subsecção como é integrada na aplicação OpenGL o processamento de *shaders* através da tecnologia GLSL, e como são efectuadas renderizações directas para objectos FBO, na execução do algoritmo criador do efeito de profundidade numa cena sintética 3D.

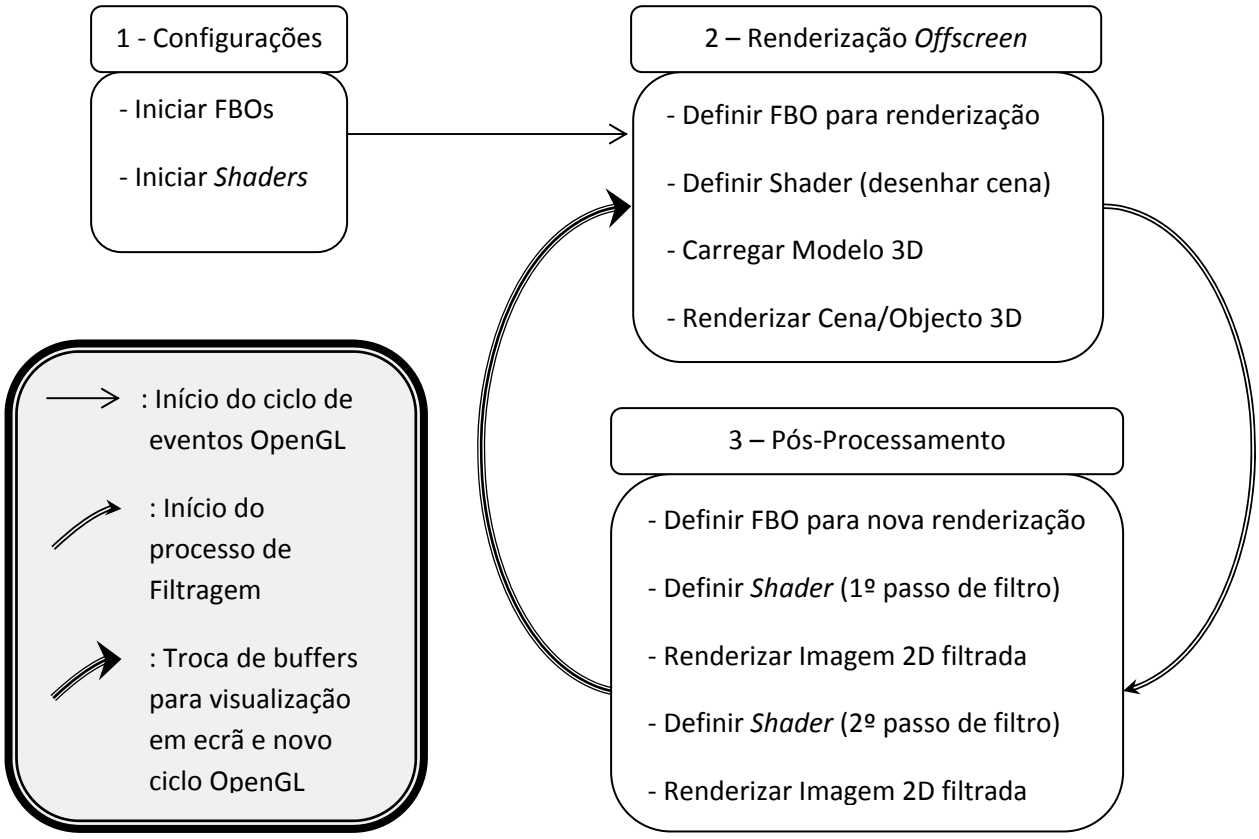


Figura 30 – Fluxo de processamento da simulação de profundidade de campo

A técnica implementada para simulação visual do efeito da profundidade de campo processa-se de acordo com o diagrama apresentado na Figura 30. Uma descrição específica e detalhada sobre os pontos mais importantes de todo este procedimento será de seguida efectuada.

4.3.1 Configurações

As configurações do contexto OpenGL para a execução da simulação de profundidade de campo englobaram os seguintes aspectos:

- Activação dos modos de exibição das imagens renderizadas na janela da aplicação. São estes:
    - Janela com duplo *buffer*. São criados dois *buffers* de cor, um *front buffer* e um *back buffer*. O objectivo é evitar artefactos por *flickering* (cintilação de imagens), ao desenhar a cena 3D no *back buffer* não visível em ecrã e só então copiado totalmente para o *front buffer*, este visível ao utilizador. As renderizações para os respectivos *buffers* são feitas automaticamente, mas a cópia do *back buffer* para *front buffer* é feita de forma explicita pela invocação do comando OpenGL *glutSwapBuffers()*. Duplo *buffer* é deste modo imprescindível em cenas animadas evitando, na presença de um só *buffer*, a possível espera e visualização do desenho em ecrã quando em causa está uma cena de renderização complexa;
    - Sistema de cores RGBA (Red Green Blue Alpha). Compreende o modelo RGB misturando as três cores primárias (vermelho, verde e azul), e também o factor alfa que indicará a informação de opacidade de cada pixel;
    - Teste de profundidade. Na renderização de uma imagem será guardada, para cada pixel, a distância do ponto respectivo ao visualizador na perspectiva definida. Estes dados são bastante importantes para a execução da técnica de profundidade de campo aqui detalhada.

- Definição da posição e tamanho da janela;

- Especificação das funções associadas aos eventos de renderização em OpenGL, como o desenho 3D, a transformação da janela por parte do utilizador, ou o tratamento de teclas pressionadas;
- Criação da interface gráfica que possibilitará manipular os vários parâmetros ópticos, simulando a câmara que estará a captar a imagem visionada na janela da aplicação;
- Inicialização de extensões da biblioteca GLEW que permitirão a utilização de *shaders*;
- Inicialização dos objectos FBO juntamente com as texturas, ou objectos *Renderbuffer* associados, para guardar o mapa de cor da cena 3D captada;
- Inicialização dos *shaders* a serem utilizados durante o processo global de renderização.

4.3.1.1 Inicialização de Objectos FBO

A inicialização de objectos FBO (*Frame Buffer Objects*) deve ser realizada de acordo com o procedimento planeado à partida para renderização de imagens. É importante determinar o número de objectos FBO e quais os anexos para guardar imagens que cada um desses objectos irá ter. Recorde-se que cada objecto poderá anexar a si texturas ou objectos *Renderbuffer* para conter o mapa de informação de renderização para cada pixel da cena tridimensional.

Assim, tal como efectuado nesta implementação, um objecto FBO pode ser gerado a partir do comando:

- *void GenFramebuffersEXT(sizei n, uint *framebuffers framebuffers)*

Antes de ser efectuada a associação de uma imagem ou *Renderbuffer* a um objecto FBO, tal como outra qualquer operação, deve ser utilizado o comando:

- *void BindFramebufferEXT(enum target, uint framebuffer framebuffer)*

Uma criação de objecto *Renderbuffer* e a sua anexação a um qualquer FBO deverá utilizar, respectivamente, os seguintes métodos disponibilizados pela sua API. A geração de um objecto Renderbuffer é feito invocando o comando:

- *void GenRenderbuffersEXT(sizei n, uint *renderbuffers renderbuffers)*

A definição do actual objecto *Renderbuffer* para consentir as posteriores operações realizadas é feita através da função:

- *void BindRenderbufferEXT(enum target, uint renderbuffer renderbuffer)*

Para se definir o formato do tipo de dados que o Renderbuffer em causa irá guardar (profundidade entre outros) é definido utilizando o método:

- *void RenderbufferStorageEXT(enum target, enum internalformat, sizei width, sizei height)*

Finalmente um objecto Renderbuffer é anexado ao um objecto FBO através do seguinte comando:

- *void FramebufferRenderbufferEXT(enum target, enum attachment, attachment, enum renderbuffertarget, uint renderbuffer)*

Por sua vez, texturas 2D podem ser igualmente anexas aos objectos FBO, tal como aconteceu no âmbito da implementação gráfica desta Dissertação. Mas as texturas apenas são usadas em objectos FBO como *buffers* de cor, ou seja, guardam o valor de todos os pixels da imagem 2D renderizada, que seriam normalmente guardados no *frame buffer* do OpenGL. As texturas depois de criadas podem ser anexas ao objecto FBO utilizando o seguinte comando:

- *void FramebufferTexture2DEXT(enum target, enum attachment, enum textarget, uint texture, int level)*

Por fim, e depois de criados e anexados aos objectos FBO todos os *buffers* necessários, é exigível a verificação de qualquer possível erro antes de executar o resto da aplicação, usando o método:

- *enum CheckFramebufferStatusEXT(enum target)*

Terminada a descrição do processo de configuração de objectos FBO veremos agora um pouco melhor a criação de texturas na implementação realizada. Todas as texturas foram geradas com idênticos parâmetros e propriedades. Por uma razão de optimização a ser explicada na subsecção 4.4 deste documento, as texturas utilizadas foram definidas com o formato RGBA16F. Cada pixel é assim guardado, na textura, num grupo de quatro valores reais correspondendo aos

quatro canais do sistema de cor RGBA, cada componente do pixel dispondo de 16 bits, num total de 64 bits por pixel (Figura 31). As texturas são definidas com o mesmo tamanho da janela em que vai ser renderizada a imagem final, na resolução pré-determinada.

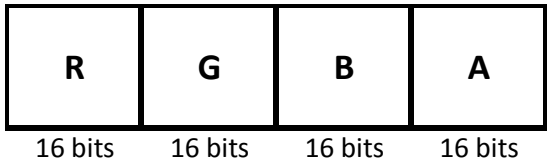


Figura 31 – Formato de um pixel de textura

Como será visto mais à frente, as texturas associadas aos objectos FBO, aquelas que conterão as imagens renderizadas, irão ser filtradas e desenhadas sobre uma superfície geométrica, um quadrilátero. Esta situação comporta problemas de correspondência de *texels* (unidade de texturas), dos mapas de texturas para os devidos pixels no ecrã onde normalmente uma textura será estendida ou encurtada consoante a geometria de destino. Esta deformação da textura original é compensada através da chamada "Filtragem de Textura", processo em que é calculado o fragmento de cor de um mapa de textura esticado ou encolhido segundo os respectivos filtros *magnification* ou *minification* [Wri07].

Para as texturas geradas são parametrizados os referentes "filtros de textura" a serem aplicados de forma linear, ou seja, é aplicada a média ponderada dos *texels* circundantes à coordenada corrente de textura, uma interpolação linear. Garante-se que uma textura esticada apresente um aspecto ofuscado que, na maioria das vezes, contribui para um resultado mais realista do que outros tipos de filtragem de texturas, como é o caso da filtragem por aproximação ao pixel vizinho mais próximo.

4.3.1.2 Inicialização de *Shaders*

Para a implementação conduzida nesta Dissertação foram utilizados três *shaders*, um para renderização da imagem captada da cena tridimensional e outros dois relativos à aplicação do

efeito de profundidade à imagem anterior, segundo o filtro de passagem dupla (subsecção 4.1.2 deste documento). Os *shaders* necessitam de serem configurados antes da sua utilização durante a execução da aplicação. A inicialização de um *shader* engloba todo o processo desde a própria criação do *shader* até ao ligamento (*link*) dos objectos gerados pelo compilador, juntando-os num único programa executável.

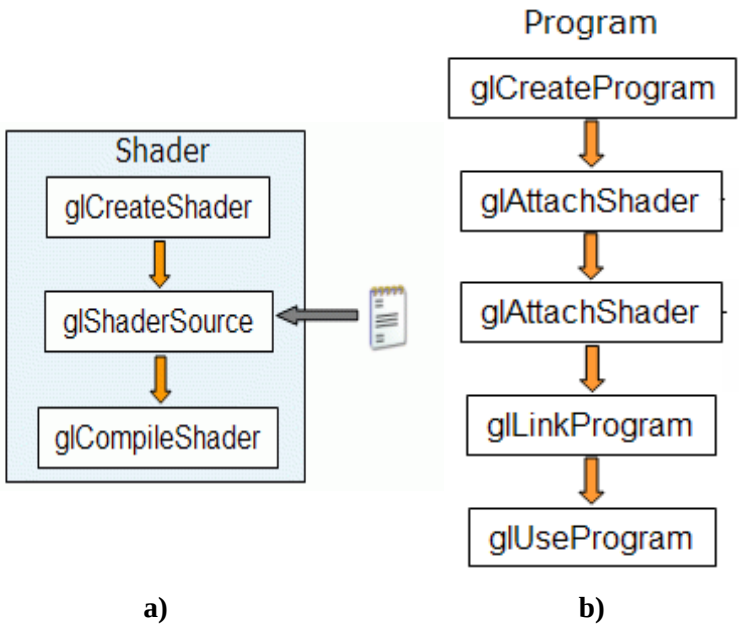


Figura 32 – Configuração de *shaders*. a) Criação e compilação de shaders.
b) Criação de Programas com anexo de *shaders*

Como visto anteriormente, quando se refere um *shader*, está-se normalmente a mencionar um par de *shaders* (*vertex*/*fragment*) e este par é denominado na implementação como um Programa. Veremos de seguida os passos necessários até um Programa estar pronto a ser utilizado, segundo a linguagem de *shading* GLSL [Ros06], como esquematizado pela Figura 32.

Inicialmente é criado o objecto que conterá o *shader* através da função que devolverá o processo associado ao *shader* criado:

- *GLuint glCreateShader(GLenum shaderType)*

Na aplicação do filtro de passagem dupla, geradora de profundidade de campo, foram utilizados dois pares de *shaders* (pares vertex/*fragment shader*), e assim, dois Programas. Mas apenas três criações de *shaders* foram efectivamente necessárias, pois nos dois Programas o

vertex shader é exactamente o mesmo. Poderão ser criados quantos *shaders* se quiserem, tendo em conta que um Programa só pode conter uma função *main* no conjunto dos *vertex shaders* existentes, regra idêntica é aplicada no conjunto dos *fragment shaders* presentes.

De seguida à criação de cada *shaders* é a si anexado o código fonte com as respectivas instruções de processamento, fazendo utilização da função *glShaderSource*, e finalmente compilado com a instrução *glCompileShader*. Para uma melhor especificação:

- *void glShaderSource(GLuint shader, int numOfStrings, const char **strings, int *lenOfStrings)*

- *void glCompileShader(GLuint shader)*

Com os *shaders* tratados e compilados, segue-se a criação dos Programas necessários. Foram necessários dois, cada um representa, respectivamente, as duas passagens do filtro que caracterizam a técnica de simulação de profundidade de campo aqui descrita. A instrução *glCreateProgram* permitirá criar o objecto que conterá o par de *shaders* e devolverá o processo associado:

- *GLuint glCreateProgram(void)*

Não existe nenhum limite para a criação de Programas mas apenas se pode utilizar um em cada momento de desenho 3D. Tal como esta técnica demonstrará, os shaders garantem grande flexibilidade na sua utilização. Durante o processo de renderização de uma imagem pode ser trocada a execução de um *shader* por outro, ou simplesmente não usar nenhum continuando a execução através da funcionalidade fixa do OpenGL, no tratamento *vertex* e *fragment*.

A criação de um Programa permite então anexar a si os *shaders* criados nos passos anteriores pelo uso da função *glAttachShader*:

- *void glAttachShader(GLuint program, GLuint shader)*

Cada Programa só terá de anexar a componente do fluxo de processamento gráfico que se quer manipular, só *vertex*, só *fragment* ou ambas. Nesta implementação, os dois Programas possuem o par de *shaders* embora anexem o mesmo *vertex shader* por ser igual. Tal como em C se podem utilizar várias classes, cada Programa pode anexar quantos *shaders* se quiserem mas, para cada um dos dois tipos de *shaders*, só pode haver uma instrução *main*.

A ligação (*link*) final de cada Programa é conseguida por uso da função *glLinkProgram* especificada da seguinte forma:

- *void glLinkProgram(GLuint program)*

O processo de inicialização de *shaders* termina com a definição de todas as conhecidas "variáveis uniformes", utilizadas durante a execução dos devidos *shaders*. Em GLSL este tipo de variáveis permitem introduzir, a partir do processamento OpenGL, valores necessários a cálculos internos aos *shaders*, que se mantenham constantes ao longo de uma primitiva, imagem ou mesmo durante uma cena inteira. Para este efeito é utilizada a função *glGetUniformLocation* que requisita espaço em memória para alocar o valor a levar para o *shader*, sendo o método *glUniform* aquele que guardará, de facto, o valor no espaço alocado. Estas duas últimas funções podem ser assim pormenorizadas:

- *GLint glGetUniformLocation(GLuint program, const char *name)*

- *GLint glUniform{1,2,3,4}{f,I,v}(GLint location, GLsizei count, GLfloat *v)*

Como exemplo de uma variável imprescindível aos cálculos efectuados nos *shaders* de criação de profundidade de campo, verifica-se a distância corrente a que se encontra o plano focado da cena tridimensional visionada. Esta pode ser dinamicamente alterada pelo próprio utilizador e assim é usada directamente pelos respectivos *shaders* na renderização seguinte.

4.3.2 Renderização *Offscreen*

Depois de iniciado o ciclo infinito para eventos do OpenGL, a função definida para renderização de imagem será também invocada interminavelmente até à terminação do programa. Nesta função inicia-se o estágio de processamento da simulação implementada onde se realiza a renderização prévia àquela a ser exibida em ecrã e que irá oferecer a imagem base para o pós-processamento do estágio seguinte a este.

O objectivo principal nesta fase é renderizar a cena ou objecto 3D usando objectos FBO que guardarão informação respectiva nos seus *buffers*, a ser posteriormente processada na criação de profundidade de campo. Para isto se concretizar é essencial a execução de passos

como, por exemplo, a indicação do objecto FBO a ser usado para guardar a informação de renderização (método *glBindFramebufferEXT*). Também deverá ser indicado o *buffer* para onde a escrita do *fragment shader* deve ser direccionada, condizendo com o mesmo *buffer* anexado durante as configurações de objectos FBO. Note-se que podem ser escritos mais que um *buffer* ao mesmo tempo, dois por exemplo para guardar informação relativa a cor e profundidade. Os possíveis métodos utilizados são os seguintes:

- *void glDrawBuffer(GLenum mode)*

ou

- *void glDrawBuffers(GLsizei n, const GLenum *bufs)*

De seguida é necessário alterar o estado do *shader* encarregado desta primeira renderização para activo, i.e., aquele que irá ser usado durante o desenho da cena 3D, através da instrução:

- *void glUseProgram(GLuint prog)*

Finalmente são definidas as propriedades de projecção e perspectiva utilizada em cada cena a renderizar, e finalmente desenhados os objectos ou cena 3D, de forma normal como em qualquer programa OpenGL. Uma cena tridimensional é concebida a partir das primitivas geométricas (desenhar pontos, linhas, figuras geométricas) disponibilizadas pelo OpenGL. Pode ainda ser atingido maior realismo com adição de cor simples ou segundo uso tipos de materiais, iluminação, texturas ou outros, sobre todos os objectos em cena. O carregamento de modelos 3D mais complexos pode ser efectuado antes do acto de desenho em si. A imagem desenhada é naturalmente desprovida de profundidade de campo, apresentando-se como uma típica imagem *Pinhole*, e por isso será processada no estágio seguinte para obtenção do efeito de profundidade desejado segundo as propriedade ópticas definidas.

O *shader* utilizado para renderização da cena deverá ser programado consoante a cena tridimensional criada. Cada cena tem as suas únicas características e como, neste caso, é usado um *shader* para renderização, qualquer necessário processamento referente a primitivas por *vertex* ou *fragment* terão de ser programadas obrigatoriamente nos *shaders*. Por exemplo, se for

desejada a existência de iluminação no ambiente virtual criado, a mesma terá de ser programada directamente no *vertex shader* usada nesta fase de processamento da profundidade de campo.

Embora os *shaders* para renderização da cena 3D sejam aqui distintos consoante as suas próprias propriedades, todas possuem um processamento idêntico, ou seja, é guardada a informação de cor e profundidade de cada pixels na textura destinada à mesma renderização. O valor de profundidade em cada ponto da imagem é obtido, nos *shaders*, através das variáveis uniformes disponibilizadas pelo GLSL, acedendo desta forma ao estado corrente do OpenGL. No *vertex shader*, a multiplicação da matriz *modelview* do OpenGL (modelação e de visualização), através da variável uniforme gl_*ModelViewMatrix*, com a posição *vertex* calculado, através da variável uniforme *gl_vertex*, dará a posição real do ponto em causa em relação ao ponto de vista corrente. Desta operação é extraída apenas o valor de profundidade e transferida para o respectivo processamento *fragment*, através de variáveis *varying* do GLSL que permitem a passagem de informação interpolada de *vertex shader* para o *fragment shader*. Finalmente, no processamento por fragmento, os valores de cor e profundidade são escritos na textura determinada nas configurações de objectos FBO.

4.3.3 Pós-Processamento

Nesta fase depois de renderizados todos os objectos 3D, perfeitamente detalhados quaisquer que sejam as suas distâncias à teórica câmara, é então dado início à técnica de Pós-Processamento para criação de profundidade de campo.

O conhecimento dos procedimentos de interacção entre *shaders* e objectos FBO no acto da renderização é novamente necessário neste estágio. O próximo passo é utilizar a imagem resultante da fase anterior e aplicar a primeira parte do processo de filtragem, ou seja, filtrar a imagem 2D apenas num eixo (horizontal). É determinado o objecto FBO e o destino de renderização da próxima imagem filtrada, e ao mesmo tempo definido o *shader* que será encarregue deste primeiro passo do filtro aplicado. O *shader* irá ter como dados de entrada a textura com a informação da imagem original da cena 3D, usando uma variável uniforme de GLSL. De seguida é desenhada a imagem filtrada e esta, por sua vez, também guardada para a

segunda parte do processo de filtragem. Note-se que agora não são objectos 3D a serem desenhados, como na subsecção anterior "Renderização *Offscreen*", mas sim um simples polígono (quadrilátero) incorporando a imagem processada. Agora, para que a imagem seja correctamente renderizada ocupando toda a janela da aplicação, as configurações de projecção e perspectiva de visualização definidas para a cena 3D deixaram de fazer sentido, e assim foram carregadas as respectivas matrizes identidade.

O segundo passo deste filtro é executado de forma semelhante com a diferença de a sua renderização já não necessitar de um objecto FBO por ser a última e aquela a ser exibida em ecrã, devendo por isso ser escrita directamente no *frame buffer* do próprio OpenGL. Como já referido anteriormente, o *shader* respectivo à segunda passagem da filtragem aplicada utilizou como dados de entrada não a imagem da cena 3D original, mas sim a última imagem a ser renderizada, resultado da primeira passagem do filtro de profundidade de campo.

Este estágio de processamento termina com a chamada de instrução *glutSwapBuffers*, a mesma que provoca a troca da última cena processada do *back buffer* para o *front buffer* (como explicado na subsecção 4.3.1) e ser apresentada em ecrã na janela da simulação.

De seguida serão revistos os pormenores relativos à execução interna aos *shaders* encarregues de modelarem a correcta profundidade de campo em imagens sintéticas.

Os dois Programas usados nesta fase, referentes aos dois passos do filtro implementado, são idênticos na sua essência, possuindo ligeiras diferenças nalguns aspectos relacionados com o respectivo tratamento a que se propõem. Apenas o *fragment shader* difere de um Programa para outro nas duas passagens de filtragem. O mesmo *vertex shader* é utilizado nos diferentes Programas por não existir nenhum processamento específico aos vértices (iluminação por exemplo) para além das transformações geométricas requeridas. É sobre o *fragment shader* que irá ser efectuada uma análise mais profunda por conter o algoritmo fulcral à criação do efeito de profundidade de campo, no tratamento das cores da imagem 2D gerada a partir da cena 3D previamente construída. Relativamente aos *fragment shaders* dos dois Programas utilizados, o primeiro avalia as contribuições dos pixels vizinhos horizontais no seu processamento de todos os pixels da imagem, enquanto o segundo considera os pixels vizinhos verticais.

Os *shaders* tanto da primeira passagem como da segunda são muito semelhantes e, por esta razão, será efectuada uma descrição de ambos os tipos (*vertex* e *fragment*) de ambas as passagens ao mesmo tempo.

4.3.3.1 *Vertex Shader*

Foi criado um *vertex shader* e anexado aos dois Programas a executar durante a simulação, e embora exista este não possui nenhuma instrução que contribua para a técnica implementada. Apenas é feita a chamada da função (*ftransform*) que garante um óptimo desempenho e que os valores atribuídos aos dados de vértices sejam exactamente os mesmos caso fosse usada a funcionalidade fixa de *vertex* do OpenGL. Funcionalidade esta que será substituída se um *vertex shader* for usado durante a execução da aplicação gráfica.

4.3.3.2 *Fragment Shader*

O *fragment shader* aplicado, no âmbito da metodologia apresentada nesta Dissertação, irá processar da mesma forma todos os pixels da imagem previamente renderizada pelo OpenGL, de maneira a produzir a mesma imagem mas agora com o efeito de profundidade de campo. Isto considerando as propriedades da câmara óptica, como veremos já de seguida.

Através de variáveis uniformes da linguagem GLSL [Ros06], são utilizados os seguintes dados de entrada, indispensáveis à simulação pretendida:

- Uma textura com o mapa de cor da imagem 2D a filtrar;
- A distância à câmara do plano focado na cena 3D, definido interactivamente pelo utilizador;
- O valor de um pixel escalado para um *texel*, para cálculo de coordenadas da textura a filtrar. É utilizada a largura ou altura consoante a primeira ou segunda passagem do filtro. Como visível na Figura 33, o *fragment shader* da primeira passagem processará cada pixel (central) avaliando os pixels vizinhos horizontais (esquerda e direita do pixel central). Por sua vez, o *fragment*

shader da segunda passagem executará o mesmo processo mas considerando os pixels vizinhos verticais (cima e baixo do pixel central). Esta variável define em que dimensão da imagem a execução se irá realizar, se vertical ou horizontal;

- Factor de multiplicação para determinação dos círculos de confusão de cada ponto da imagem visualizada. Este valor mantém-se constante para cada *frame* segundo o valor de distância focal estabelecida pelo utilizador, e por isso pode ser movido para fora do *shader* tal como a variável anterior, libertando assim a mesma computação por todos os pixels de uma imagem;
- Tamanho do filtro, escolhido pelo utilizador, que poderá ser aumentado ou reduzido variando de igual maneira o número de pixels amostrados.

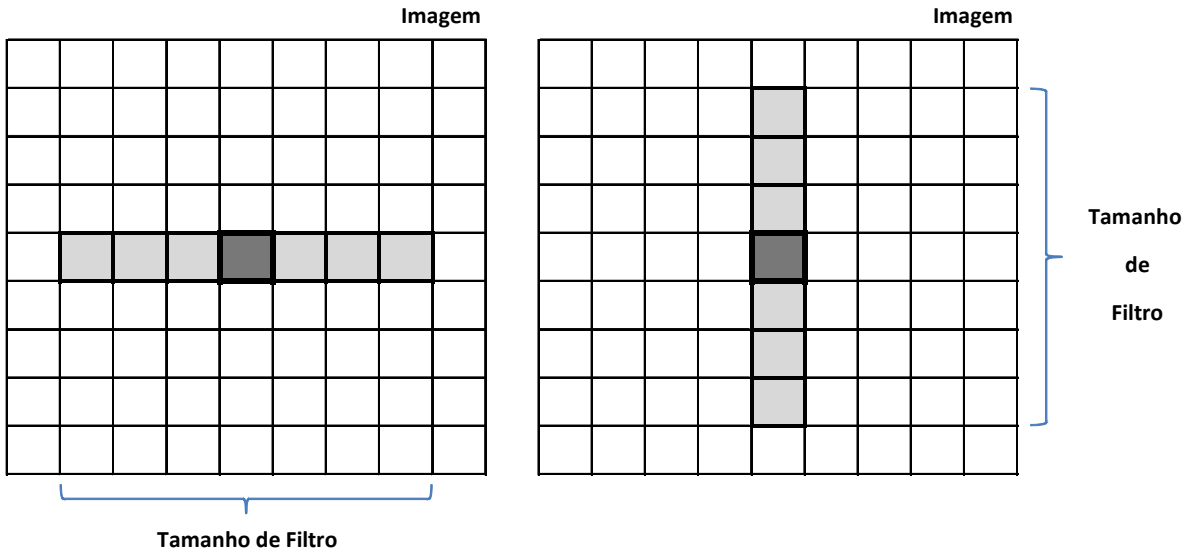


Figura 33 – Processamento do filtro de passagem dupla. Amostragem de pixels vizinhos ao pixel a calcular (central). a) Primeira passagem do filtro, amostragem horizontal. b) Primeira passagem do filtro, amostragem vertical

O pseudo-código descritivo do processamento executado pelo *fragment shader*, em ambas as passagens do filtro de imagem, é o seguinte:

```
{
    obter a cor do pixel-central;
    obter a profundidade do pixel-central;
    calcular o CdC (Círculo de Confusão) para o pixel-central;
    calcular o peso de intensidade para o pixel-central;
    adicionar à cor final a cor do pixel-central consoante a sua intensidade;
    para ( cada pixel-amostrado, vizinho do pixel-central)
    {
        obter a cor do pixel-amostrado;
        obter profundidade do pixel-amostrado;
        calcular o CdC para o pixel-amostrado;
        calcular a contribuição de sobreposição do pixel-amostrado;
        calcular a contribuição de intensidade do pixel-amostrado;
        calcular a contribuição de dispersão do pixel-amostrado;
        calcular o peso final do pixel-amostrado;
        adicionar à cor final, a cor do pixel-amostrado já ponderado através dos 3 pesos
        calculados;
    }
    renormalizar a cor final referente ao pixel-central;
    saída da cor final;
}
```

A obtenção de cada pixel (central) verifica a avaliação e contribuição dos pixels vizinhos (amostrados) segundo três factores (sobreposição, intensidade e dispersão), tal como explicitado na subsecção 4.1.2 desta Dissertação.

4.4 Optimizações

A implementação realizada no trabalho, aqui apresentado, teve desde o início o intuito de melhorar o mais possível a metodologia Pós-Processamento considerada, Zhou *et al*. [Zho07], quer em qualidade quer em eficiência. Com este objectivo, optimizações de diversa ordem foram efectuadas, não só durante a implementação mas também posteriormente à implementação da profundidade de campo. São descritos de seguida os mais importantes melhoramentos operados sobre a programação da técnica em si, relativamente ao OpenGL e *shaders*, sendo alguns sugeridos pelos criadores desta metodologia por filtro de passagem dupla [Zho07]. Refira-se que nem todas as optimizações efectuadas por Zhou *et al*. puderam ser igualmente realizadas nesta implementação pela existência de diferenças no próprio desenho de ambas as soluções, como o tamanho máximo do filtro. A implementação da simulação aqui descrita tem a capacidade de elevar o tamanho máximo do filtro a um nível consideravelmente alto, podendo este ser definido pelo utilizador durante a execução.

No contexto OpenGL a grande melhoria efectuada foi a utilização dos chamados objectos FBO (*Frame Buffer Objects*), explicitados na subsecção 4.2.1. Estes objectos representam destinos para onde o OpenGL pode desenhar que não o ecrã, ou seja, renderizações para destinos que não o *buffer* padrão deste sistema, o *frame buffer*. Ao associar uma textura a um objecto FBO foi assim possível renderizar a cena 3D directamente para a textura totalmente *off-screen*, evitando uma desnecessária cópia do *frame buffer* para textura. Esta é a grande vantagem deste tipo de objectos que tornam a execução naturalmente mais eficiente.

Mas a utilização de objectos FBO pode ser estendida de forma a obter-se ainda melhores resultados de desempenho. Dependendo do tipo e número de *buffers* necessários à execução da respectiva aplicação, um utilizador pode configurar objectos FBO de maneira a aproveitar as diversas formas que estes objectos oferecem para cambiar entre destinos de renderização. Pode ser utilizado, por exemplo, um único FBO aproveitando os destinos de renderização por si permitidos e definindo qual ou quais estão activos em cada momento. Ou podem ser utilizados distintos objectos FBO, contendo cada um o seu destino para renderização, seja referente a informação de cor, profundidade ou qualquer outro tipo de informação. Foi exactamente esta

última hipótese, aquela implementada por ser descrita como a mais vantajosa no que diz respeito a desempenho [Gre05].

As restantes optimizações de maior importância centraram-se na própria programação dos *shaders* utilizados nesta implementação e na interacção com a execução do OpenGL. A execução foi grandemente optimizada pela utilização do canal alfa nas texturas, formatadas no sistema de cor RGBA, onde foi guardada o mapa de cor da imagem captada sobre a cena 3D. Normalmente o canal alfa está associado à informação relativa a transparência, mas como este efeito não foi tratado nesta implementação aproveitaram-se assim estes bits para fixar o valor de profundidade de cada pixel renderizado, evitando a existência de outra textura para este efeito. Como a informação de cor é guardada numa textura, associada a um objecto FBO, foi preciso definir o formato da textura de forma a permitir guardar o valor total de profundidade no canal alfa. Cada pixel obedece ao sistema RGBA, possuindo normalmente apenas 8 bits (RGBA8). Isto teve de ser alterado para que a informação total, relativa a cor e profundidade, fosse guardada com sucesso ao espaço de memória reservado a cada pixel. Assim foram definidas texturas com o formato RGBA16F, passando cada pixel a ocupar 64 bits de memória, 16 para cada canal (R, G, B e A). A renderização da imagem para textura é feita por meio de um *shader* específico à passagem do filtro em causa, que guarda a informação de cor (RGB) nos primeiros três canais e a informação de profundidade no último canal (A) de cada pixel. Este procedimento possibilitou descartar uma textura que seria necessária para guardar apenas dados de profundidade por pixel, e evitou, consequentemente, mais uma renderização para a referida textura, suprimindo desta forma o dobro das leituras de pixels a texturas (*texture lookups*).

Outra optimização realizada prende-se com a tentativa de reduzir ao máximo a computação efectuada por fragmento, através do *fragment shader* encarregue de filtrar a imagem original renderizada. Desta forma, valores que se mantêm constantes por cada *frame* foram calculados fora do *shader* e passados para este através de variáveis uniformes a partir da execução do OpenGL. Foi descoberto, durante esta implementação e depois de alguns testes, que o desempenho do processamento de um *fragment shader* é significativamente afectado pela existência de código condicional. Isto levou à remoção deste tipo de código na amostragem de pixels vizinhos relativos ao pixel processado, no filtro de passagem dupla. Esta acção mostrou-se

bem mais eficiente, embora levando a que fossem efectuadas todas as rotinas sobre pixels que à partida não contribuíam para o pixel central e que podiam ser evitadas com condições.

Ainda nos *shaders*, aproveitaram-se as várias capacidades matemáticas disponibilizadas pelo GPU, mais concretamente pela linguagem de programação GLSL a partir das funções disponibilizadas pela respectiva API e principalmente pelos vantajosos vectores GPU, tipo de dados contendo um grupo de números reais, *vec2* com dois, *vec3* com três e por fim *vec4* com quatro números reais. A API referente à linguagem GLSL oferece ainda inúmeras funções que permitem a execução de operações, sobre todas as componentes dos vectores GPU, de forma paralela (quatro no máximo, com os respectivos *vec4*).

Procedeu-se a um esforço de simplificação global do código criado relativo a todo o processamento OpenGL para desenho das cenas tridimensionais testadas e processamento de *shaders* para criação do efeito de profundidade.

Foram ainda consumadas várias tentativas de optimização da própria qualidade da profundidade de campo apresentada, através dos princípios ópticos utilizados, mas sem sucesso porque a técnica utiliza correctamente a teoria óptica do modelo de câmara *Thin Lens*. Como veremos na secção "Testes e Resultados" o único problema desta técnica diz respeito à não separabilidade do filtro utilizado e não na aplicação das leis ópticas. Notou-se sim uma melhoria de qualidade quando experimentada uma semelhante versão do filtro de passagem dupla, mas agora efectuada numa só passagem. Como se previa, foi melhorada a qualidade mas piorado tenuemente o desempenho global, a ver na secção 5.1.

4.5 Interface

O simulador de profundidade de campo implementado oferece uma simples interface gráfica, criada a partir da biblioteca GLUI. Esta interface compreende duas janelas gráficas: a janela de visualização da simulação e uma outra para controlo de propriedades de simulação, como é visível segundo a Figura 34.

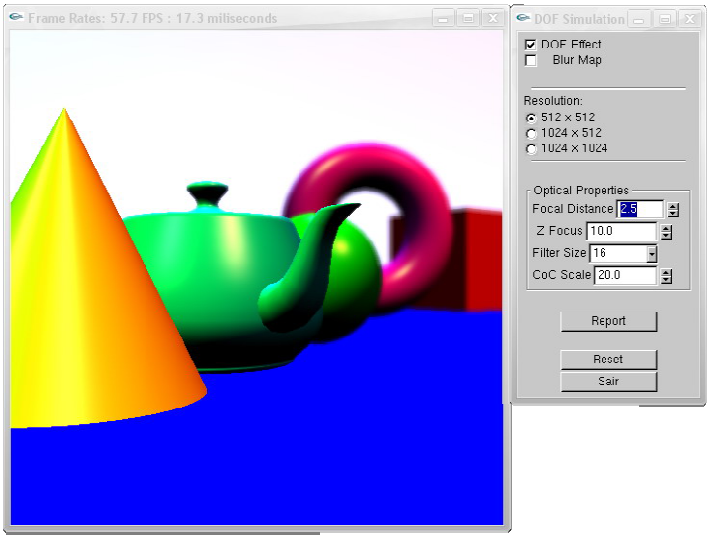


Figura 34 – Interface gráfica da simulação de profundidade de campo

A janela principal permitirá visualizar a simulação da profundidade de campo, e durante o tempo que a aplicação estiver a ser executada, irão sendo apresentadas, no espaço reservado ao título da janela, as taxas de *frames* por segundo (FPS) e os tempos de apresentação de cada *frame* (em milissegundos), como ilustrado na Figura 35.

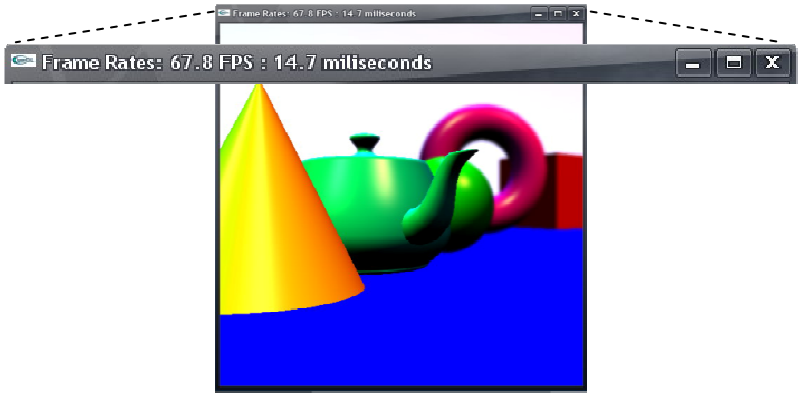


Figura 35 – Informação de desempenho mostrada na barra de título da aplicação

A segunda janela referida contém todos os meios de controlar a simulação de acordo com as escolhas exclusivas a cada utilizador.

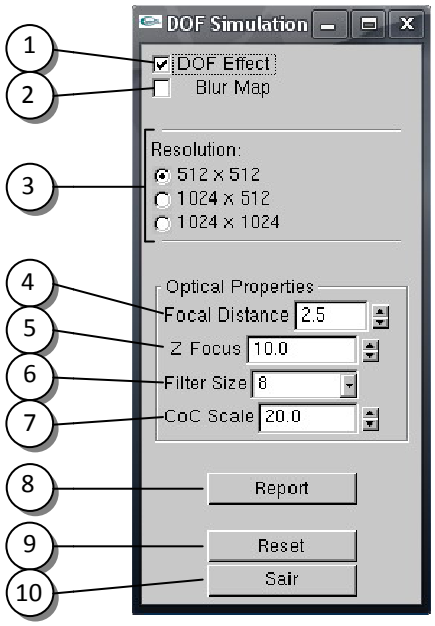


Figura 36 – Janela de controlo da simulação visual

A Figura 36 mostra-nos a sequência pela qual serão descritas as funcionalidades dos diversos controladores, como botões, caixas de verificação, caixas de selecção, entre outras relativas à simulação da profundidade de campo. As funcionalidades interactivas são:

1. Definição de visualização da cena 3D com ou sem efeito de profundidade de campo;

2. Visualização do mapa da imagem num sistema de cores preto e branco, mostrando as áreas de maior processamento de profundidade de campo, consoante as propriedades ópticas definidas. Zonas a preto correspondem a áreas totalmente nítidas, incluídas no plano focado e por isso com menor processamento de filtragem. As restantes zonas serão mais claras quanto mais afastadas do plano focado. Quanto mais claros forem os pontos da imagem, maiores os seus círculos de confusão, indicando assim as zonas da imagem onde maior processamento se concentrará (Figura 37);

Figura 37 – Mapa de processamento de filtragem

3. Determinação da resolução da janela principal. Com esta funcionalidade a aplicação poderá apresentar simulações com resoluções de ordem:
    - 512 x 512;
    - 1024 x 512;
    - 1024 x 1024.

4. Escolha do valor referente à teórica distância focal, i.e., a distância até à lente, posicionada entre a lente e o plano de imagem. Determina a quantidade da luz convergida (ao focar) ou difundida (ao desfocar), num sistema óptico. Como exemplo, uma diminuição deste valor levará a um aumento do poder de focalização óptico;

5. Determinação do plano focado. O valor apresentado corresponde à distância real da cena 3D onde é desejado que se apresentem os objectos focados. Simula o plano onde o utilizador estaria a olhar com os seus próprios olhos;

6. Definição do tamanho do filtro a utilizar nos cálculos de profundidade de campo, correspondendo ao tamanho máximo do Círculo de Confusão calculado para cada

pixel. Isto é, se um pixel da imagem filtrada possui um Círculo de Confusão muito grande, apenas serão ponderadas as contribuições de pixels vizinhos abrangidos pelo tamanho definido neste ponto. Um aumento deste valor levará à contribuição de um maior número de pixels vizinhos durante o processamento do filtro de dupla passagem, e assim a diminuição de desempenho global;

7. Variação de escala do Círculo de Confusão. Com este factor é possível obter diferentes níveis de profundidade de campo, pois as contribuições de pixels vizinhos serão alteradas dinamicamente, aumentando ou diminuindo o raio dos círculos de confusão dos pixels amostrados;

8. Escrita de relatórios de desempenho para ficheiro de texto pré-determinado;

9. Reinicialização de todas as opções do painel de controlo da simulação para os valores por defeito;

10. Sair da aplicação.

5. Testes e Resultados

Nesta secção descrevem-se todos os testes, resultados e análises realizadas sobre a simulação de profundidade de campo implementada. A partir de três diferentes cenas 3D, mostradas na Figura 38, foi efectuado um estudo focando a qualidade e desempenho de renderização da técnica desenvolvida.

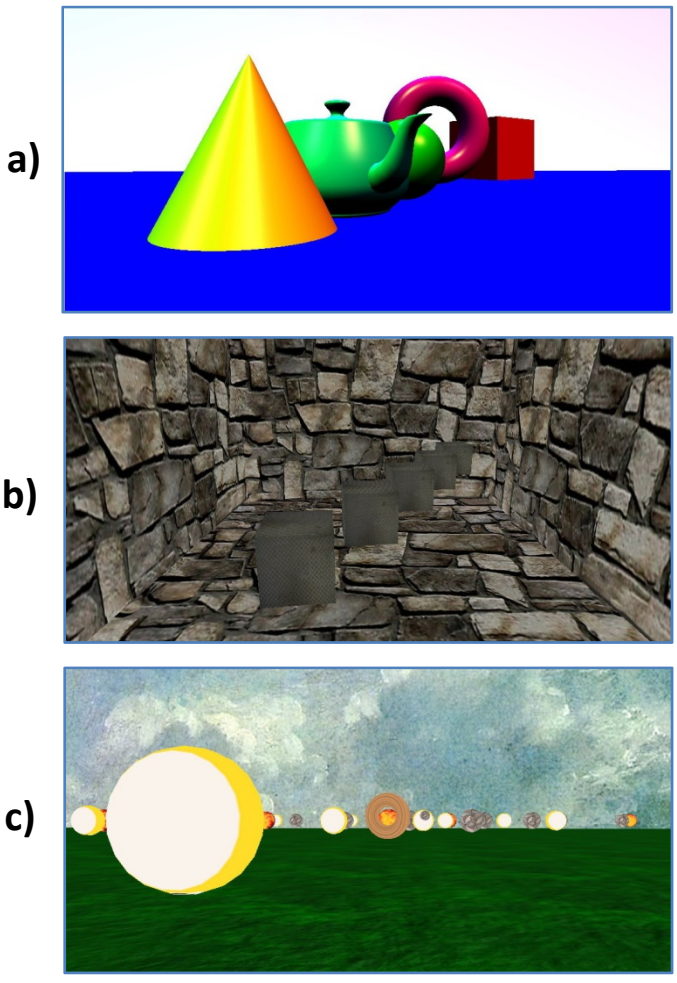


Figura 38 – Cenas tridimensionais. a) Figuras. b) Sala de Cubos. c) Mundo das Esferas

A necessidade de simular o efeito de profundidade sobre diferentes cenas tridimensionais prende-se com o facto de serem assim permitidos testes mais abrangentes à própria aplicação. As cenas, de distinta complexidade gráfica, possuem variações principalmente ao nível da utilização de cores e texturas, bem como, ao nível da quantidade, pormenorização e posicionamento de objectos em profundidade. As três cenas testadas nesta implementação podem ser descritas da seguinte forma:

- **"Figuras"**: Cena possuidora de iluminação, constituída por cinco objectos (cone, bule, esfera, quadrado e torus), sem qualquer textura aplicada, ou seja, usando apenas uma única cor para cada objecto, Figura 38-a). Cena com relativa profundidade;

- **"Sala de Cubos"**: Cena totalmente texturizada, isto é, todos os objectos possuem texturas. Cinco cubos estão dispostos em profundidade, dentro de uma sala (outro cubo). As paredes da sala e os cubos apresentam texturas diferentes. Cena com pouca profundidade, Figura 38-b);

- **"Mundo das Esferas"**: Cena bastante mais ampla, constituída por 250 esferas, um torus e um pavimento criado com cerca de 1680 triângulos. Todos os objectos apresentam aplicação de texturas numa cena com grande profundidade, Figura 38-c);

Será, por fim, analisado o comportamento da metodologia para profundidade de campo apresentada num ambiente virtual animado, e o grau de dificuldade na sua integração com uma aplicação gráfica externa aos trabalhos realizados nesta Dissertação.

5.1 Qualidade de Renderização

Como se trata de uma simulação visual, o teste principal é, de facto, verificar a qualidade visual do efeito de profundidade comparando os níveis de nitidez objecto focado com a restante cena desfocada. Outra forma de análise, à qualidade do efeito de profundidade gerado, pode ser conseguida se confrontados os seus resultados com a familiar profundidade de campo notória em imagens fotográficas ou obras cinematográficas. E neste ponto a qualidade da profundidade de campo sobre as cenas tridimensionais equipara-se perfeitamente à qualidade do mesmo efeito em

Fotografia, por exemplo, indo ao encontro dos objectivos propostos inicialmente, como se pode verificar pela Figura 39.

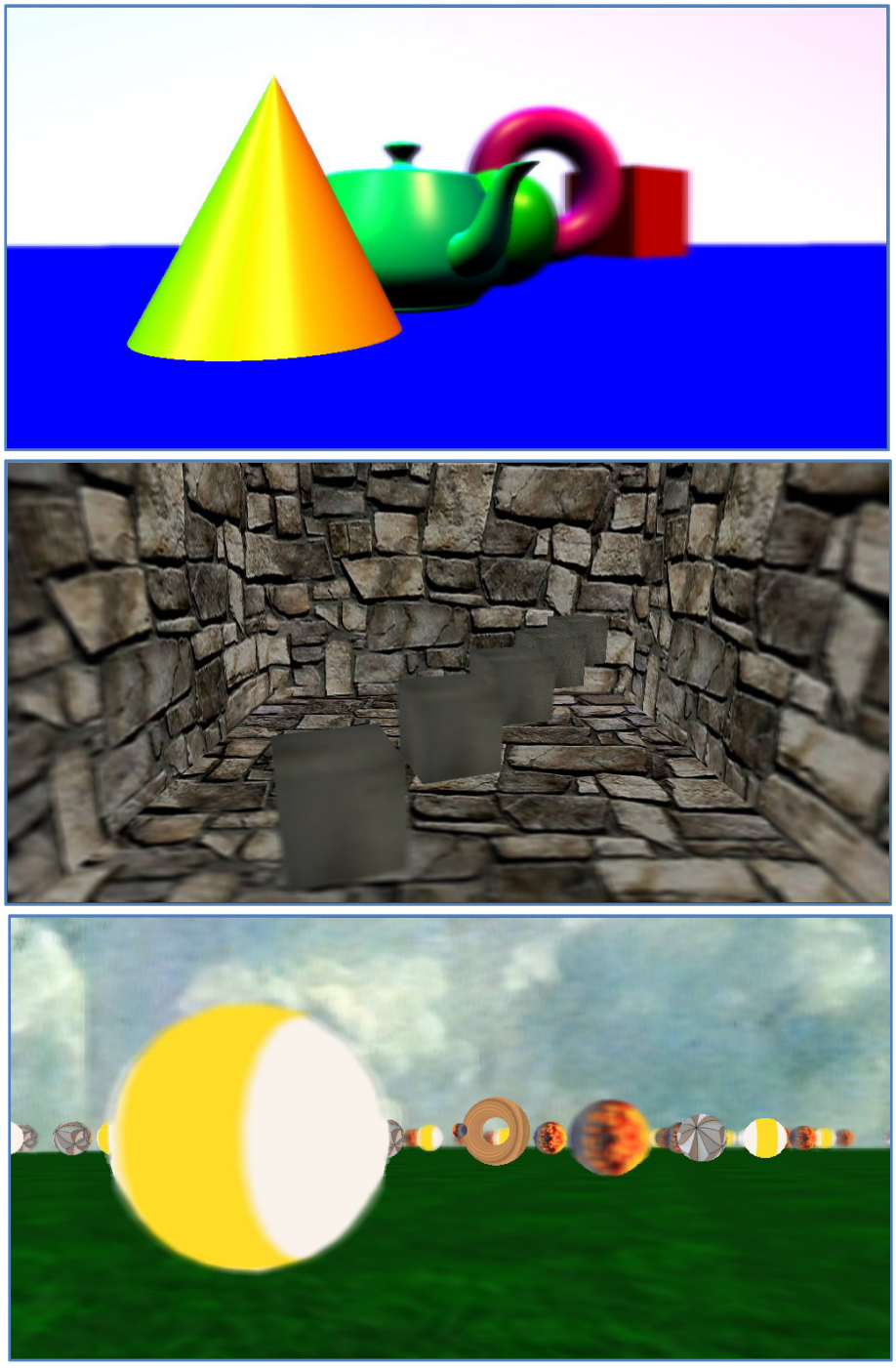


Figura 39 – Resultados visuais da simulação da profundidade de campo. Na primeira cena o cone é focado. Na cena seguinte é focado o último cubo. E na última cena é focado o torus.

Mas a real avaliação do nível de qualidade de profundidade de campo desta técnica, no universo da Computação Gráfica, apenas pode ser medida se comparada, nas mesmas condições, com outras técnicas Pós-Processamento existentes. A publicação bastante recente de uma nova técnica Pós-Processamento para profundidade de campo, Lee *et. al.* em [Lee08], faz este tipo de comparação onde se inclui o método de Zhou *et. al*. [Zho07], a mesma adoptada neste trabalho. No seu estudo comparativo é sugerido a não precisão da técnica de Zhou *et al*., facto também confirmado durante a realização desta Dissertação, embora confirmada como uma das mais capacitadas técnicas simulatórias da profundidade de campo.

A metodologia usada na modelação de profundidade de campo, apesar de aplicar um filtro de forma idêntico aos filtros Gaussianos separáveis, não o é. Isto porque cada uma das duas passagens do filtro (processamento vertical e horizontal) pondera as contribuições de pixels amostrados de forma dinâmica, quando na realidade deveria aplicar uma mesma função Gaussiana em ambas as passagens do filtro. Esta situação só se torna problematicamente visível em condições excepcionais, mais concretamente, quando definidas propriedades ópticas e de filtro com valores elevados (Figura 40). No entanto estas apenas são perceptíveis ao olho humano se consideravelmente próximos da câmara. A variação de escala do Círculo de Confusão apresenta-se como uma das propriedades críticas para a qualidade de renderização, isto face a aplicação de valores elevados, pelo facto do filtro ter um tamanho limite. Quando o factor de escala é excessivamente aumentado, cada pixel necessita de um correspondente maior número de contribuições de pixels vizinhos do que o filtro permite, levando ao aparecimento de artefactos visuais distorcendo a imagem final.

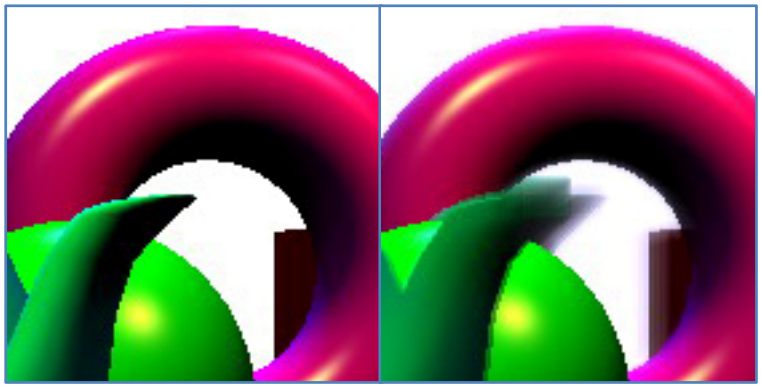


Figura 40 – Artefacto visual inerente à metodologia apresentada, utilizando valores extremos das propriedades ópticas

De forma a confirmar visualmente a imprecisão devido ao filtro ser aplicado separadamente, considerando as componentes horizontal e vertical, decidiu-se efectuar uma abordagem à implementação de um filtro de uma só passagem. Este novo filtro envolve os mesmos princípios de aplicação que o filtro de passagem dupla no cálculo da profundidade de campo. Modificou-se apenas a ponderação de contribuições de pixels vizinhos, pois agora teriam de ser realizadas amostragens de pixels na horizontal e vertical do pixel a calcular, isto numa só passagem de processamento sobre a imagem 2D original. Neste filtro de uma única passagem, a cor final de cada pixel é então calculado a partir das contribuições de pixels vizinhos segundo uma amostragem circular, como ilustrado pela Figura 41. Experimentou-se este tipo de amostragem pois seria impossível efectuar uma amostragem rectangular à volta do pixel a calcular sem afectar gravemente o desempenho da simulação. Menos pixels são assim amostrados, numa forma circular, estando estes posicionados mais proximamente do pixel central, o que torna o resultado final mais suave e real, como se verá de seguida.

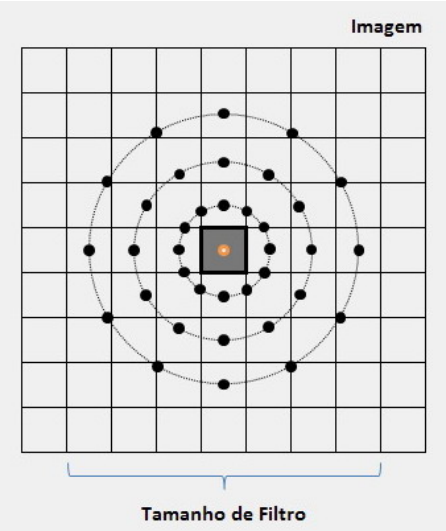


Figura 41 – Amostragem circular de pixels vizinhos do pixel a calcular no *shader*

Notou-se acima de tudo uma melhoria no efeito de desfocagem, corrigindo o erro visual apresentado na Figura 40, comprovado pela Figura 42. Como esperado, esta abordagem teve um custo de eficácia na sua execução, que a deixa sem grande margem de progressão para integração em aplicações interactivas em tempo-real. Conclui-se que este filtro de uma passagem necessita de amostrar muitos mais pixels vizinhos que o filtro de passagem dupla para atingir uma melhor

qualidade visual. Para chegar a valores de FPS mais elevados é exigível uma amostragem mais restrita afectando, naturalmente, a qualidade da simulação.

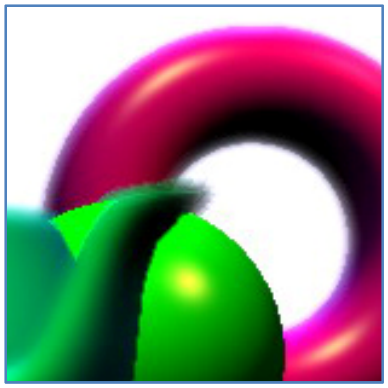


Figura 42 – Resultado visual da simulação da profundidade de campo com filtro de uma só passagem e amostragem circular

Embora a técnica elaborada não apresente artefactos visuais respeitantes a fugas de cor (*intensity leakage*), devido ao respectivo tratamento, este não trata de problemas de oclusão parcial. Como já explanado neste documento, a oclusão parcial demonstra-se bastante complicada de resolver em metodologias Pós-Processamento, como a implementada nesta Dissertação, onde a imagem a filtrar é resultante de um único ponto de vista. Para simular correctamente a oclusão parcial seria imprescindível a informação de porções de objectos que estão, de facto, ocluídos, e assim inexistente nos distintos *frame buffers*. Não foram efectuados trabalhos de resolução desta questão.

O simulador de profundidade de campo implementado da forma explanada na secção 4, permite ajustar, através de um painel de controlo, todas as suas variáveis, podendo o utilizador definir o efeito de profundidade que melhor preenche os seus intentos. A profundidade de campo tem como objectivo tornar uma aplicação gráfica mais realista, mas o contexto da própria aplicação pode requerer diferentes qualidades de profundidade de campo. Podem-se desejar e apresentar divergentes efeitos de profundidade, quer em simples programas para ensinamento de Computação Gráfica (básicos e leves), quer em Realidade Virtual (complexos e extremamente realistas) ou até em jogos computorizados (condições específicas para cada tipo de jogo). Isto torna o simulador bastante flexível, onde é permitido ao utilizador escolher o plano focado,

variar a distância focal, aumentar o Círculo de Confusão por pixel, e alterar igualmente o tamanho do filtro aplicado.

5.2 Desempenho de Renderização

Os testes de desempenho realizados, sobre o simulador implementado, focaram o grau de rapidez de execução com a alteração do plano de focagem definido pelo utilizador, permitindo então concluir se, na percepção humana, a aplicação está efectivamente a executar o efeito de profundidade aproximadamente em tempo-real.

Os resultados apresentados nas Tabelas 4, 5 e 6, foram obtidos a partir de um computador com processador Intel 1.86 GHz Core2Duo, 2 GB RAM e uma placa gráfica NVIDIA GeForce 8500GT, executando o sistema operativo Windows XP.

	Sem Profundidade de Campo		Com Profundidade de Campo		Custo	
Resolução	FPS	Tempo por frame	FPS	Tempo por frame	FPS	Tempo
512 x 512	96	10,3	58	17,1	38	6,8
1024 x 512	54	18,5	31	32,1	23	13,6
1024 x 1024	26	37,6	16	62,9	10	25,3

Tabela 4 – Desempenho de profundidade de campo da cena "Figuras" (tempos em milissegundos)

	Sem Profundidade de Campo		Com Profundidade de Campo		Custo	
Resolução	FPS	Tempo por frame	FPS	Tempo por frame	FPS	Tempo
512 x 512	460	2,2	83	12,0	377	9,8
1024 x 512	265	3,8	44	22,5	221	18,7
1024 x 1024	142	7,0	23	43,1	119	36,1

Tabela 5 – Desempenho de profundidade de campo da cena "Sala de Cubos" (tempos em milissegundos)

	Sem Profundidade de Campo		Com Profundidade de Campo		Custo	
Resolução	FPS	Tempo por frame	FPS	Tempo por frame	FPS	Tempo
512 x 512	76	13,1	48	20,5	28	7,4
1024 x 512	46	21,8	24	40,8	22	19
1024 x 1024	36	27,5	14	71,4	22	43,9

Tabela 6 – Desempenho de profundidade de campo da cena "Mundo das Esferas" (tempos em milissegundos)

Os resultados registados foram obtidos mantendo as mesmas propriedades ópticas e de filtro para todas as cenas. Pode-se concluir, numa primeira análise, que o desempenho do filtro não só varia com o incremento de resolução de renderização, como seria natural, mas também com as características desiguais que cada cena apresenta em relação às outras. A principal propriedade, das cenas 3D, directamente relacionada com o desempenho geral da simulação é a profundidade a que se estendem todos os seus objectos. Quanto mais distantes da câmara estiverem posicionados os objectos de uma cena, superior será o número de cálculos efectuados, pela existência de pixels com maior Círculo de Confusão e assim realizarem uma amostragem de pixels vizinhos também maior.

De todos os parâmetros variáveis da simulação, o tamanho máximo do filtro, e por conseguinte o diâmetro máximo de Círculo de Confusão, é aquele que afecta directamente o seu desempenho. O tamanho de filtro e o desempenho global da aplicação estão correlacionados ao ponto de um destes factores necessitar de ser sacrificado em prol do outro. Exemplificando esta situação de compromisso entre tamanho de filtro e desempenho, o aumento do tamanho do filtro levará a que se registe um maior número de leituras de cor sobre texturas de renderização, e na execução dos *shaders* são, estas leituras, as operações mais custosas. A relação entre o tamanho máximo do Círculo de Confusão e o desempenho é demonstrado na Figura 43, testado no "Mundo das Esferas", onde é necessário maior filtro possível devido à cena se estender bastante em profundidade.

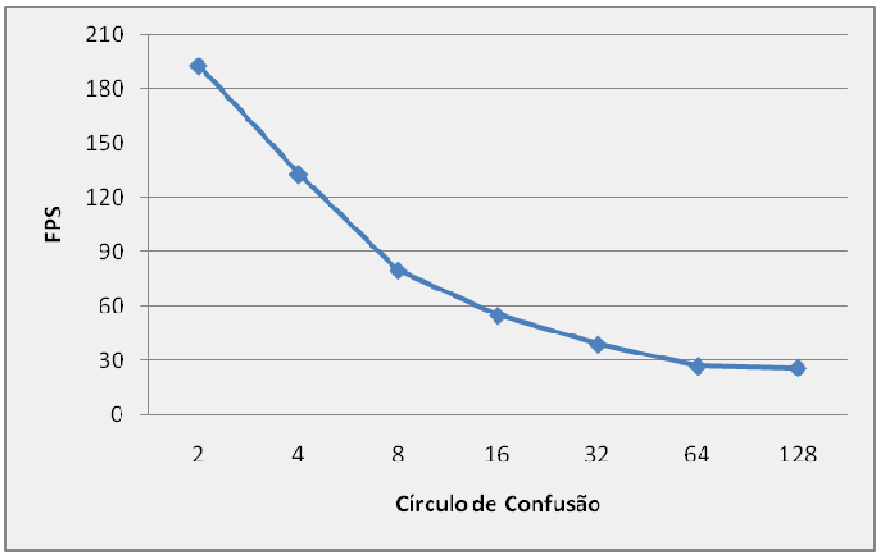


Figura 43 – *Tradeoff* entre o tamanho do filtro e desempenho global da simulação

Por fim, realizou-se o último teste de desempenho, à aplicação criada, com o objectivo de verificar as reais vantagens oferecidas pela utilização de objectos FBO. Foi assim utilizada a implementação da profundidade de campo, de uma primeira fase desta Dissertação, sem a existência de objectos FBO, e comparada com a sua versão final. A partir de testes sobre a cena "Sala de Cubos", com as três distintas resoluções permitidas, foram registadas melhorias de desempenho entre os 40% e os 50% de valores FPS médios, comprovando desta forma a enorme vantagem no uso da arquitectura FBO na renderização 3D.

Os resultados de desempenho aqui apresentados confirmam uma simulação de profundidade de campo em tempo-real aproximado. Os valores de desempenho são suficientes para se concluir que, em conjunto com as tecnologias gráficas de hoje, a técnica implementada possa ser usada sem graves custos em qualquer aplicação de tempo-real, onde se adequa, pelo menos, a apresentação de 60 FPS durante toda a sua execução.

5.3 Simulação em Ambiente Virtual Animado

Sabendo que a profundidade de campo será imprescindível e comum em aplicações gráficas de tempo-real num futuro bem próximo, foi aproveitada a simulação implementada, para

simular também animação de objectos e movimentação do ponto de vista (ou câmara) na cena tridimensional.

A cena "Sala de Cubos" apresenta cubos giratórios que se movem também em profundidade, dando outra dimensão da experiência visual da profundidade de campo. Só um plano se encontra focado enquanto os cubos se vão movendo em fila, para trás e para a frente, passando por esse mesmo plano, ora mostrando aspecto nítido, ora aspecto desfocado.

Por outro lado, a cena "Mundo das Esferas" habilita o utilizador a uma mesma experiência que tanto um jogo ou a Realidade Virtual podem oferecer, ou seja, a liberdade de movimentos e de definição do plano focado, no ambiente gráfico sintético em causa. Naturalmente que esta comparação é feita apenas ao nível da experiência sensorial (visão), pois quer um jogo ou uma aplicação de RV envolvem inúmeros aspectos não considerados nesta simples simulação, como som sincronizado com eventos, técnicas de interacção com utilizador, entre outros. Não são sequer efectuadas comparações de renderização com as aplicações referidas actuais (jogos, RV, RA), possuidoras de universos 3D extremamente complexos e definidos pela utilização das capacidades mais avançadas que as placas gráficas oferecem, também elas cada vez mais poderosas.

A animação nas cenas referidas apresentou-se com uma excelente qualidade, não sendo notada qualquer espécie de inconsistências visuais no redesenhamento constante de várias *frames* por segundo. Isto habilita, de maneira ainda mais concreta, a integração do método de profundidade de campo, apresentado nesta Dissertação, com qualquer tipo de aplicação gráfica. A técnica adoptada neste trabalho, para geração de profundidade de campo, é mesmo das poucas técnicas Pós-Processamento mais recentes que não apresenta qualquer espécie de incoerência da imagem visualizada em ambientes virtuais dinâmicos [Lee08]. Incoerência como a cintilação durante o desenho de novos frames é totalmente inexistente na simulação testada.

5.4 Integração com Aplicações Gráficas

O último teste realizado sobre a metodologia apresentada, para criação gráfica da profundidade de campo, cingiu-se à própria integração com qualquer aplicação que não as

geradas nos trabalhos efectuados desta Dissertação. Uma aplicação gráfica, concebida também através de OpenGL, foi-me gentilmente cedida por Tudor Carean, no seguimento de conversas, por correio electrónico, sobre carregamento de modelos 3D para aplicações gráficas, sendo o seu contacto obtido em [Car09]. Esta apresenta-se com relativo grau de complexidade devido a aspectos como o carregamento de um modelo 3D, mais concretamente umas ruínas no meio de uma montanha, o envolvimento de gravidade, colisões, sombras realistas e o livre controlo e visualização, através de teclado e rato, em tudo idêntico ao estilo de jogo *First-Person Shooter*. A integração provou-se ser bastante acessível para qualquer programador de gráficos, sendo apenas necessário inicializar os *shaders* e objectos FBO, e integrar na renderização da cena os passos da metodologia apresentada nesta Dissertação, mais concretamente na secção 4 deste documento. Refira-se que o plano focado foi determinado pelo centro da imagem, ou seja, a partir do movimento do rato, por parte do utilizador, a imagem apresentará o plano focado consoante a distância do ponto central da imagem virtual ao ponto de vista em questão.

A única desvantagem encontrada diz respeito à própria exigência da utilização de *shaders* durante a renderização da imagem, que desta forma neutraliza toda a programação por *vertex* efectuado na aplicação OpenGL como aconteceu, neste caso, com toda a iluminação realista existente na cena tridimensional original. A iluminação na aplicação integrada necessitaria de ser programada integralmente no respectivo *vertex shader* durante a criação da aplicação, e não no programa OpenGL.

A execução da profundidade de campo nesta aplicação, com taxas de execução afastadas de tempo-real, mostrou-se ainda assim bastante eficaz. Depois de integrada com a aplicação referida, apenas foi apresentado um custo de cerca 5 FPS.

O resultado desta experiência, i.e., integração da metodologia de profundidade de campo numa aplicação gráfica, pode ser visualizado através da Figura 44, onde se direcciona o centro da imagem para 3 planos distintos em termos de profundidade da cena visualizada, focando, respectivamente, esses mesmos planos. Com isto, prova-se nesta Dissertação que a metodologia, apresentada na secção 4 deste documento, possibilita uma correcta, qualitativa e eficaz simulação da profundidade de campo em qualquer aplicação gráfica que apresente ambientes virtuais, executada em tempo-real aproximado por uso de recentes tecnologias gráficas.

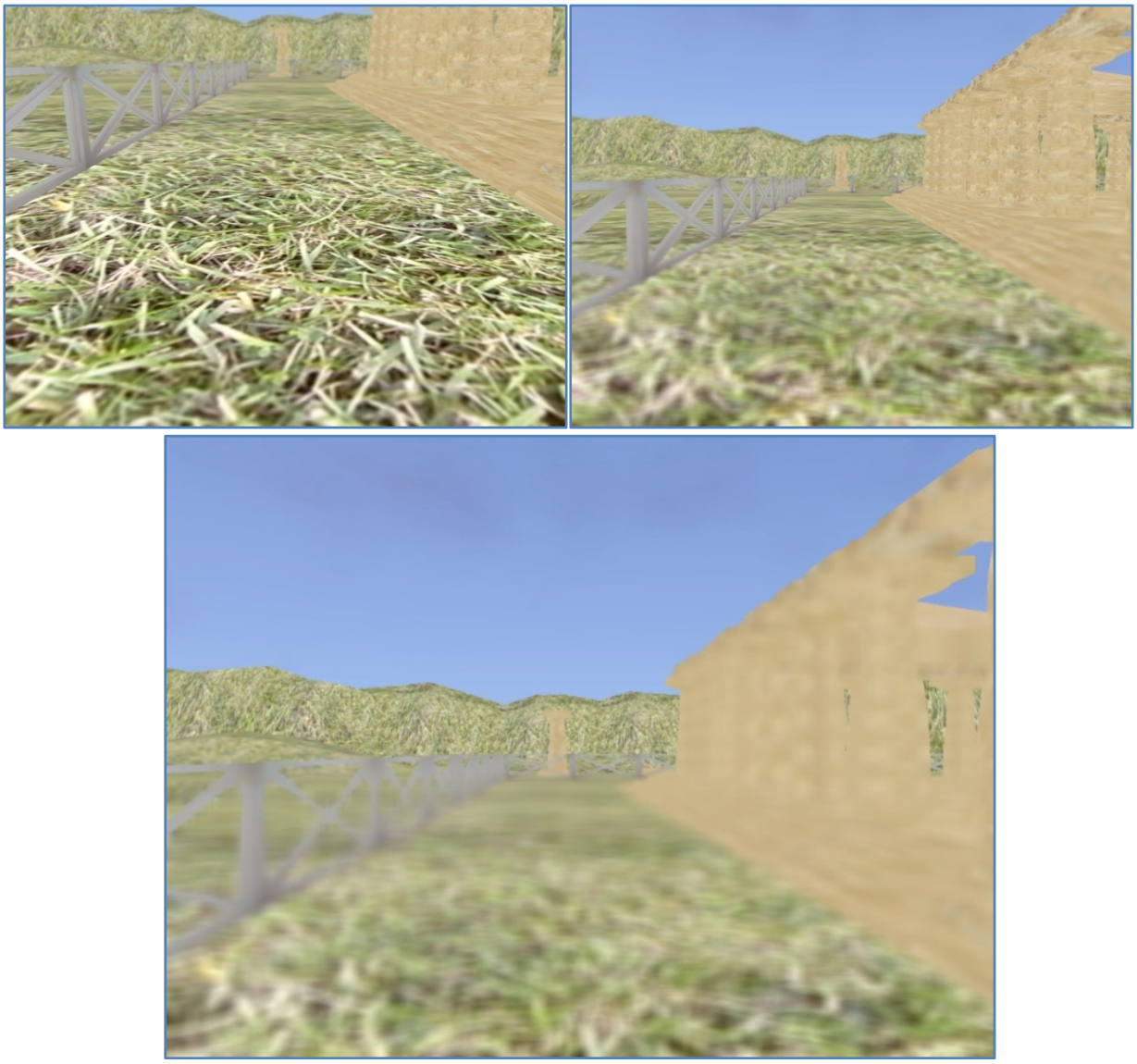


Figura 44 – Integração da profundidade de campo numa aplicação OpenGL

6. Conclusão

6.1 Trabalho Realizado

O tema apresentado nesta Dissertação nasceu da ideia de introduzir o conceito de profundidade de campo em aplicações criadas por meios e conhecimentos da Computação Gráfica. Com este fim, foi elaborado um trabalho de estudo e apresentação prática de uma metodologia capaz de recriar o efeito da profundidade de campo em cenas sintéticas tridimensionais, desprovidas de qualquer distinção de importância dos objectos visualizados.

Primariamente, foram explorados os princípios ópticos por detrás do fenómeno natural que é a profundidade de campo para então determinar a forma de modelar a mesma por meios computacionais.

O estado da arte relativa ao tema desta Dissertação mostrou-nos algumas das mais recentes técnicas Pós-Processamento para profundidade de campo, tendo sido adoptada a técnica de Zhou *et al.* [Zho07] essencialmente por apresentar um melhor equilíbrio entre qualidade e eficiência. A técnica baseia-se em cálculos precisos fundados a partir do modelo de câmara *Thin Lens*, pós-processando a imagem *Pinhole* gerada computacionalmente com um filtro de passagem dupla, aplicado através de *shaders*.

O processo de *shading* foi realizado utilizando as capacidades de aceleração gráfica em GPU oferecidas pela linguagem de programação GLSL [Ros06], especificamente adaptada à linguagem OpenGL facilitando a criação e interacção de *shaders* com as suas aplicações. Os *shaders* permitem ao programador modificar o *pipeline* gráfico durante execução de programas, tanto ao nível de processamento de vértices e transformações (*vertex shader*) como ao nível de

processamento de cor relativamente a fragmentos (*fragment shader*), paralelizando intensamente a execução.

O filtro de passagem dupla é aplicado de uma forma semelhante aos filtros Gaussianos separáveis. Uma imagem 2D renderizada é interpretada como 1D e então processada separada e consecutivamente nas duas dimensões possíveis, horizontal e vertical. A filtragem pondera a contribuição dos pixels vizinhos a cada pixel (central) segundo 3 factores distintos, estabelecidos a partir dos respectivos círculos de confusão consoante a sua informação de profundidade. Os 3 factores são: a sobreposição, a intensidade e a dispersão de cor.

Várias optimizações foram experimentadas, tanto em termos de qualidade da profundidade de campo como do próprio desempenho, mas apenas estas últimas tiveram sucesso, especialmente com a utilização de objectos FBO [FBO06]. Este tipo de objectos possibilita associar diferentes destinos de renderização *off-screen*, guardando ao mesmo tempo diferentes tipos de informação como cor, profundidade entre outros. Mostra-se ideal na execução da técnica proposta, por necessitar da imagem pré-renderizada para posterior filtragem com o efeito de profundidade de campo. A imagem previamente renderizada pode ser guardada directamente para uma textura, evitando assim os passos de renderização para *frame buffer* do OpenGL e a necessária cópia para a textura de *input* ao algoritmo de geração do efeito de profundidade.

6.2 Contribuições

Comprova-se nesta Dissertação que se torna possível, através de técnicas e tecnologias gráficas recentes, a apresentação de uma metodologia para modelação gráfica da profundidade de campo capaz de ser integrada por qualquer programador nas suas aplicações gráficas. Consideram-se assim atingidos os objectivos principais definidos à partida.

A profundidade de campo implementada com base no algoritmo de Zhou *et al*. [Zho07] não é precisa. Isto porque o filtro de passagem dupla é aplicado como qualquer outro filtro Gaussiano separável mas, de facto, não o é. O filtro usa, sim, cálculos precisos de teoria óptica mas que não são correctamente utilizados na prática, ainda que incorrecções visuais só sejam descortinadas pelo olhar humano quando valores elevados de propriedades ópticas são

empregados. Comprovou-se pelos testes realizados (secção 5) que esta imprecisão não evitou que a técnica adoptada apresentasse muito bons resultados, óptima para tornar qualquer aplicação gráfica mais realista com a geração do efeito de profundidade numa cena tridimensional.

Como alternativa ao filtro de passagem dupla, foi implementada semelhante metodologia de filtragem mas de uma única passagem (secção 5.1), ou seja, não separável, melhorando o efeito de profundidade mas resultando num típico exemplo de compromisso entre qualidade e eficiência.

A metodologia de criação gráfica da noção de profundidade, apresentada nesta Dissertação, oferece níveis de desempenho aproximados a tempo-real tal como pretendido à partida. Mas o resultante desempenho da profundidade de campo nesta implementação depende directamente do tamanho do filtro de dupla passagem, que necessita de ser limitada para satisfazer taxas de *frames* na ordem de tempo-real. Esta limitação poderá ter as suas repercussões negativas na qualidade do efeito de profundidade em cenas 3D, se apresentados objectos distribuídos por uma grande profundidade. Confirmou-se que esta específica situação, caso presente, não é facilmente perceptível ao olho humano em aplicações interactivas como verificado na integração apresentada na secção 5.4, onde nenhum artefacto visual foi detectado. Os testes efectuados, sobre a técnica de profundidade de campo criada neste trabalho, demonstraram que a limitação do filtro, da metodologia de profundidade de campo, não afasta qualquer aplicação de execuções em tempo-real aproximado. No entanto, não é possível garantir execuções adequadas a aplicações de Realidade Virtual em geral, prevendo-se que este problema seja facilmente superado com as maiores capacidades computacionais hoje apresentadas e em constante evolução.

Constata-se ainda que a plataforma em que se desenvolveu a implementação, e da qual foram obtidos os respectivos registos de resultados, denota capacidades (gráficas e de processamento) algo inferiores às oferecidas pelos mais avançados componentes de hardware comercializados nos dias de hoje, como indicado na Figura 45.

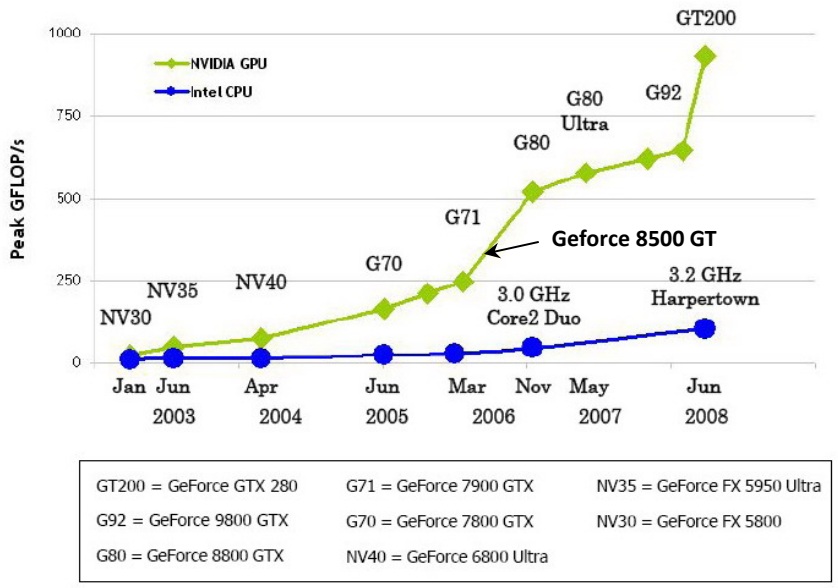


Figura 45 – Evolução da capacidade de processamento do GPU, sendo assinalado o pico de desempenho aproximado do GPU utilizado neste trabalho, NVIDIA GeForce 8500 GT

Comprovou-se uma fácil integração com outras aplicações gráficas, com ténue afectação no desempenho total de execução (secção 5.4). O maior requisito na utilização desta metodologia será o domínio da linguagem de programação GLSL, para uso de técnicas de *shading* em GPU.

É dado desta forma um agradável contributo na área de Computação Gráfica pela implementação de uma metodologia capaz de recriar a noção de profundidade em ambientes gráficos virtuais através de recentes técnicas e tecnologias gráficas. A sua qualidade e eficácia permitem naturais desenvolvimentos em áreas como Realidade Virtual ou Realidade Aumentada, quando pretendidas visualizações perfeitamente realistas.

6.3 Trabalho Futuro

Os testes previstos foram todos realizados, mas propõe-se para trabalho futuro o uso de cenas tridimensionais de alta qualidade, detalhe e definição, extraindo visualmente ainda mais a capacidade da metodologia apresentada. Para isto será essencial o carregamento de modelos 3D

para memória durante a execução da aplicação, interpretando ficheiros de criação de modelos tridimensionais em programas como "Autodesk 3ds Max".

Em trabalho futuro, para além de se efectuarem testes com cenas 3D de elevada composição e detalhe, pode ainda ser melhorada a técnica em si para resultados de melhor qualidade da profundidade de campo. Poderão, por exemplo, serem focados esforços no sentido de tratar a existência de transparência nos objectos 3D presentes e tratar situações de oclusão parcial, com um pouco de acréscimo no seu desempenho global.

Muito poucas técnicas de Pós-Processamento consideram a transparência como um dos agentes presentes na cena tridimensional, como é o caso de Krivanek *et al*. [Kri03]. Na metodologia implementada, a única textura onde são comprimidos os dados de cor e profundidade passaria a ser insuficiente. Tornar-se-ia obrigatória mais uma textura para criação do mapa de informação relativo ao nível de transparência de cada pixel, para os consequentes cálculos por fragmento nos *shaders*, durante a mistura final da sua cor.

Também a oclusão parcial é ainda algo ignorada nas inúmeras soluções para modelação da profundidade de campo ao longo da evolução da Computação Gráfica, pois é de resolução algo complexa para que seja incluído em aplicações de grande interactividade. A prática impossibilidade de apresentação de uma perfeita e rigorosa oclusão parcial por Pós-Processamento, que utiliza apenas um ponto de vista, é superada por soluções simuladoras de um efeito semelhante. A solução na metodologia apresentada nesta Dissertação poderia passar, em trabalhos futuros, pela aplicação de um filtro específico para processamento exclusivo do espaço visível em *foreground* (próximos do ponto de captação da imagem), igualmente efectuada nas metodologias [Kas06] e [Lee08].

Acredita-se que a implementação englobou as melhores técnicas de optimização existentes no presente face às linguagens de programação utilizadas, prevendo-se uma natural melhoria em termos da constante evolução das capacidades gráficas de hardware. Em alternativa às tecnologias utilizadas seria interessante, em trabalho futuro, implementar totalmente a profundidade de campo usando outras soluções gráficas como é o caso do CUDA da NVIDIA [CUD08] ou *Compute Shader* [Boy08] da Microsoft. Enquanto o *Compute Shader* não foi ainda comercializado, já em relação à tecnologia CUDA existem dúvidas nas suas capacidades de

elevar o desempenho comparativamente com as capacidades GLSL, quando estudos recentes afirmaram que o desempenho global entre GLSL e CUDA é teoricamente idêntico [Won08]. Afirmação reforçada pela contínua escolha de *shaders* na implementação de novas técnicas de simulação de profundidade de campo, como a de Lee *et al*. [Lee08].

Por fim, seria também interessante adaptar a metodologia apresentada neste trabalho à técnica de interpolação de *mipmaps* filtrados anisotropicamente, evoluindo talvez para a geração de um novo algoritmo para simulação do efeito de profundidade. A interpolação de *mipmaps* e respectivas filtragens anisotrópicas deram origem à mais correcta e eficiente simulação de profundidade de campo por Pós-Processamento, criada por Sungkil Lee *et al*. em [Lee08]. Esta parece ser aquela que mais se aproxima de renderizações multi-passo, como *Ray Tracing* distribuído.

Mipmap é uma técnica bastante utilizada em CG na filtragem de texturas, e não é mais que uma colecção de imagens associadas a uma dada textura. São diferentes versões da textura principal, pré calculadas com resoluções diferentes, que permitem que seja escolhida aquela de tamanho mais favorável ao procedimento na dimensão a que se destina a sua aplicação. Reduzem a probabilidade de artefactos visuais por *aliasing*, e melhoram os tempos de execução processando um muito menor número de *texels*, comparativamente com o uso de uma única textura.

Um filtro anisotrópico garante informação de direcção para cada *texel*, aumentando de forma acentuada o detalhe na renderização da imagem filtrada, pois evita distorções anteriormente visíveis nas superfícies de objectos 3D, a partir de projecções oblíquas. Este tipo de projecções é proporcionado, por exemplo, pela aplicação Google Earth (Figura 46) cuja função é apresentar um modelo tridimensional do globo terrestre.

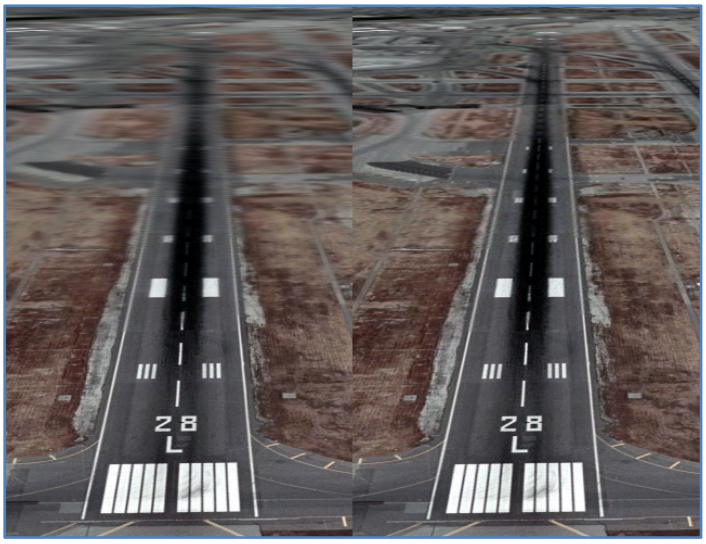


Figura 46 – Imagem comparativa da aplicação Google Earth, à esquerda sem e à direita com filtragem anisotrópica

As propostas de trabalho futuro apresentadas nesta secção, nomeadamente sobre a resolução de oclusão parcial, a utilização de novas tecnologias gráficas, como *Compute Shader*, ou mesmo a aplicação de filtragens anisotrópicas sobre *mipmaps* farão, certamente, a simulação de profundidade de campo apresentada nesta Dissertação acompanhar a evolução computacional gráfica. Permitindo, assim, aproximar cada vez mais a imagem sintética à nossa percepção da realidade.

7. Posfácio

A motivação na realização desta Dissertação foi sempre enorme desde o seu início. Tratou-se de uma excelente oportunidade de aprofundar conhecimentos e descobrir novas e mais avançadas matérias numa das minhas áreas preferidas, como é o caso da Ciência e Tecnologia da Programação, mais concretamente Computação Gráfica, que sempre me fascinou. A área e o tema desta Dissertação não podiam ir mais ao encontro das minhas ambições para uma tese de Mestrado em Engenharia Informática.

É com agrado que verifico o cumprimento dos objectivos propostos. A vontade de fazer ainda mais e melhor é grande, demonstrada pelo trabalho futuro a realizar, especificamente definido tendo esta Dissertação como ponto de partida. Toda a investigação e análise em volta da computorização do conceito de profundidade de campo apresentada neste documento, juntamente com a respectiva aplicação prática, contribuirão certamente no seu contínuo desenvolvimento científico.

Foram por mim adquiridos importantes conhecimentos na criação de uma aplicação gráfica interactiva capaz de simular a profundidade de campo, por meio de técnicas e tecnologias actuais de Computação Gráfica. Por conseguinte, obtive experiência preciosa para uma possível carreira profissional enveredada na área, acompanhada de um grande sentimento de realização pessoal no término de mais este ciclo académico que é o Mestrado em Engenharia Informática.

8. Bibliografia

[Bar05] B. Barsky, M. Tobias, D. Chu, D. Horn: Elimination of Artifacts Due to Occlusion and Discretization Problems in Image Space Blurring Techniques. Graphical Models. Vol. 67, No. 6, p. 584-599. 2005.

[Ber04] M. Bertalmio, P. Fort, D. Sanchez-Crespo: Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. Em Proceedings of Second International Symposium on 3D data Processing, Visualization and Transmission 2004. p. 767–773, 2004.

[Boy08] Chas. Boyd: Direct3D 11 Compute Shader, More Generality for Advanced Techniques. Gamefest 2008. Microsoft Game Technology Conference. 2008.

[Car09] Tudor Carean. 3D Coding Tutorials. 2009. http://www.3dcodingtutorial.com

[CUD08] NVIDIA CUDA Programming Guide 2.1. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf

[Deb09] NVIDIA Shader Debugger. 2009. http://developer.nvidia.com/object/nv_shader_debugger_home.html

[Dem04] J. Demers: Depth of Field: A Survey of Techniques. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. R. Fernando. Addison-Wesley. p. 375-390. 2004.

[FBO06] Framebuffer object technical paper. 2006. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt

[Fea96] P. Fearing: Importance ordering for real-time depth of field. In Proceedings of the Third International Conference on Computer Science. p. 372–380. 1996.

[GLS09] GLSL Tutorial.2009. http://www.lighthouse3d.com/opengl/glsl/

[Gre05] Simon Green: The OpenGL Framebuffer Object Extension. NVIDIA Corporation. Game Developers Conference. 2005.

[Hae90] P. Haeberli, K.Akeley: The accumulation buffer: hardware support for high-quality rendering. Computer Graphics (SIGGRAPH ’90 Proceedings). Vol. 24. Nº4. p. 309–318. 1990.

[Ham07] Jr. Earl Hammon: Practical Post-Process Depth of Field. *GPU Gems 3*. Hubert Nguyen. Addison Wesley Professional. cap. 28. 2007.

[Hei97] W. Heidrich, P. Slusallek, H. Seidel.: An Image-Based Model for Realistic Lens Systems em Interactive Computer Graphics. *Proceedings of Graphics Interface*. p. 68-75. 1997.

[Jen08] Jens Breitbart: A framework for easy CUDA integration in C++ applications. Kassel University. Kassel, Germany. 2008.

[Kas06] M. Kass, A. Lefohn, J. Owens: Interactive Depth of Field Using Simulated Diffusion on a GPU. Relatório técnico. Pixar Animation Studios. 2006.

[Kes06] J. Kessenich, D. Baldwin, R. Rost: The OpenGL Shading Language. 2006.

[Kos01] R. Kosara. Semantic Depth of Field – Using Blur for Focus+Context Visualization. Tese de Doutoramento, Vienna University of Technology. Áustria. p. 14-22. 2001.

[Kosl07] Todd J. Kosloff, Brian A. Barsky: An Algorithm for Rendering Generalized Depth of Field Effects Based on Simulated Heat Diffusion. Departamento EECS, Universidade de California, Berkeley, EUA. 2007.

[Kri03] J. Krivanek, J. Zara: Fast depth-of-field rendering with surface splatting. In Proceedings of Computer Graphics International, p. 196–201, 2003.

[Lee08] S. Lee, G. J. Kim, S. Choi: Real-Time Depth-of-Field Rendering Using Anisotropically Filtered Mipmap Interpolation. IEEE Transactions on Visualization and Computer Graphics. September/October 2008.

[Pot82] M. Potmesil, L. Chakravarty: Synthetic image generation with a lens and aperture camera model. ACM Transactions on Graphics 1(2): 85–108, 1982.

[Ros06] Randi J. Rost: OpenGL® Shading Language, Second Edition. Addison-Wesley. 2006.

[Sco92] C. Scofield: $2^{1/2}$D Depth of Field Simulation for Computer Animantion. D. Kirk. *Graphics Gems III*, Morgan Kaufmann. p. 36–38. 1992.

[Shr05] D. Shreiner, M. Woo, J. Neider, T. Davis: OpenGL Programming Guide (Fifth Edition). Addison-Wesley, Boston, Massachusetts, 2005.

[Sob03] Marcionílio Barbosa Sobrinho: Tutorial de Utilização de OpenGL. Centro Universitário de Belo Horizonte. 2003.

[Str07] M. Strengert, T. Klein, and T. Ertl: A Hardware-Aware Debugger for the OpenGL Shading Language. In ACM Siggraph/Eurographics Workshop on Graphics Hardware 2007. San Diego, California, USA, pp.81-88. 2007. http://www.vis.uni-stuttgart.de/glsldevil/index.html.

[Won08] Tien-Tsin Wong: Shader Programming vs CUDA. The Chinese University of Hong Kong. 2008.

[Wri07] Richard S. Wright, Jr.; Benjamin Lipchak; Nicholas Haemel: OpenGL SuperBible 4th Comprehensive Tutorial and Reference. Addison-Wesley. 2007.

[Zho07] T. Zhou, J. X. Chen, M. Pullen: Accurate Depth of Field Simulation in Real Time. *Computer Graphics Forum*. Blackwell Publishing. Vol. 26. p.15-23. 2007.

8.1 Leitura Aconselhada

[Che03] J. Chen: Guide to Graphics Software Tools. Springer Verlag, 2003.

[Coo84] R. Cook, T. Porter, L. Carpenter: Distributed Ray Tracing. ACM SIGGRAPH Computer Graphics 18(3): p. 137–145, 1984.

[Har02] M. Harris, G. Coombe, T. Scheuermann, A. Lastra: Physically-Based Visual Simulation on Graphics Hardware *Proc. SIGGRAPH/ Eurographics Workshop Graphics Hardware,* p. 1-10. 2002.

[Rok93] P. Rokita: Fast Generation of Depth-of-Field Effects in Computer Graphics. Computers & Graphics, Vo1.17 (5). p. 593-595. 1993.

[Rok96] P. Rokita: Generating depth-of-field effects in virtual reality applications. IEEE Computer Graphics and Applications 16(2): p. 18–21, 1996.

[Shi94] M. Shinya: Post-filtering for depth of field simulation with ray distribution buffer. Proceedings Graphics Interface'94, p. 59–66, 1994.

[Zwi01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. Em Siggraph 2001 Proceedings. 2001.