Francisco de Sousa Pereira

UTILIZAÇÃO EFICIENTE DE MÚLTIPLOS COMPONENTES DE HARDWARE PARA A SÍNTESE DE IMAGEM E ANIMAÇÕES

UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

VILA REAL, 2008

UTILIZAÇÃO EFICIENTE DE MÚLTIPLOS COMPONENTES DE HARDWARE PARA A SÍNTESE DE IMAGEM E ANIMAÇÕES

Tese apresentada à Universidade de Trás-os-Montes e Alto Douro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Informática, realizada sob a orientação do Professor Doutor Luís Gonzaga Mendes Magalhães, Professor Auxiliar, e do Professor Doutor João Paulo Fonseca da Costa Moura, Professor Auxiliar, ambos do Departamento de Engenharias da Universidade de Trás-os-Montes e Alto Douro, de acordo com o dispostos no Decreto-Lei 216/92.

Francisco de Sousa Pereira

Aos meus pais,

pela dedicação e apoio incondicional.

RESUMO

Palavras chaves: Processamento paralelo, computação gráfica; abstracção; múltiplos recursos.

Com o advento de sistemas computacionais a um preço acessível e cada vez mais poderosos em termos de cálculo, incluindo computadores pessoais (PC) de "*desktop*", a geração de imagens sintéticas de cenários complexos, utilizando iluminação global, é uma meta alcançável. No entanto, devem ser tidos em conta alguns pormenores no que respeita à simulação correcta dos mecanismos de transporte da luz baseados em modelos físicos, para atingir interacção em tempo real.

O poder computacional oferecido pelos actuais sistemas computacionais não é só derivado do aumento das capacidades dos CPU que os constituem, mas também da utilização de outros componentes dos sistemas, como as placas gráficas e outros componentes adicionais de processamento. A implementação de algoritmos de "*ray tracing*" nos referidos sistemas computacionais focam tipicamente a utilização exclusiva de um único componente, ou realizam tarefas muito específicas em determinados componentes. Este tipo de abordagem leva a uma falta de balanceamento ou a um mau balanceamento na distribuição de tarefas, conduzindo em casos mais extremos à marginalização de determinados componentes.

Neste trabalho apresenta-se um modelo para a gestão dos componentes num PC standard de uma forma eficiente e balanceada. No centro desta abordagem está a separação do ciclo principal da síntese da imagem e dos componentes de processamento especifico, através de um modelo por camadas com uma interface abstracta e independente dos recursos físicos. Assim isolando-se a camada lógica da implementação das primitivas que compõem o processo de síntese. Espera-se que este tipo de organização venha a providenciar vantagens ao nível da

portabilidade, facilidade de expansão a novos componentes com maior capacidade de processamento, optimizações ao nível dos recursos/componentes, balanceamento de tarefas e modularidade.

ABSTRACT

Keywords: Parallel computing; computer graphics; abstraction; multiples resources.

With the advent of affordable high computational power systems, including standard PC desktop configurations, computer graphics image production of complex models, using global illumination, is in our grasp. Nevertheless, some details must be taken into account when physically correct simulation of light transport, in real-time interaction, is to be achieved.

The computational power provided by current computer systems is not only supported by fast improvement in CPU capabilities, but also by the inclusion in such systems of additional hardware such as GPUs and add-on boards, sometimes even more powerful than the CPU itself. Implementations of ray tracing algorithms in such systems usually targets one of the available devices, or allocates specific tasks to each type of resources. This approach tends to drive the system to unbalanced work distribution and leads, in extreme cases, to marginalization of some hardware components.

The work proposed in this thesis presents a model for efficient PC multiple devices management. This approach focuses in separating the main rendering algorithm from the low level computer graphics routines and device particular specifications, through an abstract layer model, independent of the hardware physical characteristics. In this way it is possible to isolate the specification of a rendering API from the rendering process. This type of organization will provide advantages at several levels, namely portability, smooth integration of new and more powerful hardware, code optimizations at resources level, load balancing and modularity.

AGRADECIMENTOS

Na elaboração deste trabalho várias foram as pessoas que me apoiaram e ajudaram, sobre as mais diversas formas. Por este motivo gostaria de agradecer às seguintes pessoas pelo seu empenho, sabendo de antemão que é impossível citar todos aqueles que contribuíram. Para estes desde já as minhas sinceras desculpas pelo facto.

Aos meus orientadores Professor Doutor Luís Gonzaga Mendes Magalhães e Professor Doutor João Paulo Fonseca da Costa Moura, ambos do Departamento de Engenharias da Universidade de Trás-os-Montes e Alto Douro, pelo encorajamento e apoio que dispensaram na realização deste trabalho, muito para além do que era esperado.

Aos colegas do Departamento de Engenharias da Universidade de Trás-os-Montes e Alto Douro, pelos comentários, sugestões e apoio prestado.

À Universidade de Trás-os-Montes e Alto Douro, nas pessoas do seu Magnífico Reitor, Professor Doutor Armando Mascarenhas Ferreira e do Coordenador Científico do Departamento de Engenharias, Professor José Afonso Moreno Bulas Cruz, colega e amigo nesta caminhada.

Um agradecimento ainda a todos aqueles que criam e disponibilizam software de livre acesso e utilização. Saliento aqueles que vêm produzindo as mais variadas ferramentas para Linux, incluindo o próprio sistema, e ainda os que promovem a sua distribuição.

Finalmente, gostaria de agradecer aos meus pais, Maria de Sousa Freitas e João Pereira, pelo seu apoio, sem o qual este trabalho não seria possível.

ÍNDICE

1 Introdução 1
2 Revisão bibliográfica 5
2.1 Rendering 6
2.1.1 Ray tracing 9
2.1.2 Back traycing 11
2.1.3 Operações básicas no *rendering* 14
2.1.4 Intersecções 15
2.1.5 Geometria 17
2.1.6 Estruturas de representação em *ray tracing* 19
2.1.7 Formas de aceleração dos cálculos 23
2.1.8 Iluminação de um ponto 25
2.2 Algoritmos de iluminação global 28
2.2.1 Irradiância (*Radiance Irradiance Cache*) 28
2.2.2 Radiosiodade 30
2.2.3 Photon Mapping 32
2.3 Computação paralela 34
2.4 Abordagens específicas 42
2.4.1 Placas gráficas 43
2.4.2 Placas auxiliares de processamento 47
2.4.3 Implementação em CPU com optimizações 49
2.4.4 O projecto RoD 54
3 Modelo Proposto 57
3.1 Introdução 57
3.2 Definição do modelo proposto 58
3.2.1 Definição de termos e conceitos do modelo 60
3.2.2 Descrição global do HRA 62
3.2.3 Classificação das primitivas 64
3.2.4 Trabalho requerido num "*rendering*" 65
3.2.5 Comparação com outras abordagens e novas tecnologias 67

3.3 Especificação 68
3.3.1 Casos-de-Uso 70
3.3.2 Diagramas de actividades 74
3.3.3 Diagramas de classes 79
3.3.4 Diagrama de estados 83
3.4 Interacção do modelo 86
3.4.1 Cálculo da intersecção de um raio primário com um objecto 87
3.4.2 Cálculo da intersecção de um raio secundário com um objecto 93
3.4.3 Cálculo da iluminação especular num ponto 95
3.4.4 Cálculo da iluminação difusa num ponto 96
3.4.5 Outros exemplos para outros tipos de operações 97
3.5 Problemas associados a ambientes heterogéneos 98
3.5.1 Representação dos dados 98
3.5.2 Modelos de comunicação heterogéneos 100
3.5.3 Modelos de programação heterogéneos 100
3.6 Desenho dos elementos do HRA 101
3.6.1 Primitivas 102
3.6.2 Unidades de trabalho 105
3.6.3 Contentores para unidades de trabalho e resultados 106
3.6.4 Representação dos dados 109
3.6.5 Gestão e prioritização de unidades de trabalho 109
3.6.6 Comunicação entre o HRA e os recursos 110
3.7 Questões funcionais e tecnológicas de implementação 114
4 Trabalho experimental 117
4.1 Introdução 117
4.2 Implementação do *renderer* 118
4.2.1 Dados básicos e estruturas simples 119
4.2.2 Programação modular 121
4.3 Validação experimental 123
4.3.1 Metodologia definida para a validação experimental 123
4.3.2 Cenários utilizados 125
4.3.3 Plataformas utilizadas 128
4.4 Configurações definidas 128

4.4.1 Configuração do *renderer* só no CPU sem HRA (R+P) 129
4.4.2 Configuração CPU com HRA (R+H1+P)..... 132
4.4.3 Configuração CPU e GPU com HRA (R+H*+P)..... 136
4.4.4 Configuração GPU com HRA (R+H.GPU+P) 138
4.5 Análise aos resultados das experiências realizadas..... 139
4.5.1 Comparação entre CPU com e sem HRA 140
4.5.2 Comparação HRA com um ou dois recursos 141
4.6 Conclusões..... 142
5 Conclusões e trabalho futuro..... 143
5.1 Trabalho futuro 146
Referências..... 147
Apêndice A – Recursos utilizados..... 157
Apêndice B – Exemplo de primitiva 159
Glossário de termos e conceitos 169

LISTA DE TABELAS

Tabela 1 – Configuração das plataformas (hardware) utilizadas....128
Tabela 2 – Configurações gerais para as experiências com os cenários A e B....129
Tabela 3 – Quadro resumo dos resultados "CPU sem HRA"....131
Tabela 4 – Quadro de resultados "CPU sem HRA" (plataforma 1)....131
Tabela 5 – Quadro de resultados "CPU sem HRA" (plataforma 2)....132
Tabela 6 – Quadro resumo dos resultados "CPU com HRA"....134
Tabela 7 – Quadro de resultados "CPU com HRA" (plataforma 1)....135
Tabela 8 – Quadro de resultados "CPU com HRA" (plataforma 2)....135
Tabela 9 – Quadro resumo dos resultados "CPU + GPU com HRA"....137
Tabela 10 – Quadro de resultados "CPU com HRA incluindo GPU" (plataforma 1)....137
Tabela 11 – Quadro de resultados "CPU com HRA incluindo GPU" (plataforma 2)....138
Tabela 12 – Comparação das diferentes configurações para o cenário A e B (plataforma 1)....139
Tabela 13 – Comparação das diferentes configurações para o cenário A e B (plataforma 2)....140
Tabela 14 – Características do computador 1....157
Tabela 15 – Características do computador 2....158
Tabela 16 – Características da placa gráfica designada de GPU 1....158
Tabela 17 – Características da placa gráfica designada de GPU 2....158

LISTA DE FIGURAS

Figura 1 – Equação de *rendering* (Kajiya 1986) 10
Figura 2 – Raios primários e secundários no *back tracing* 12
Figura 3 – Uma aproximação geométrica de uma esfera através de polígonos 17
Figura 4 – Geometria de um golfinho representada por um conjunto de polígonos [Img01] 21
Figura 5 – Raio oposto à normal num ponto 26
Figura 6 – BSDF [Img02] 26
Figura 7 – Irradicância e zona de influência das amostras (Ward 1988) 30
Figura 8 – Múltiplas interacções utilizado o método da radiosidade [Img03] 31
Figura 9 – Iluminação directa apenas vs. inclusão da radiosidade [Img03] 31
Figura 10 – Casuísticos utilizando *photon mapping* [Img04] 33
Figura 11 – Visão geral da interacção no projecto RoD 55
Figura 12 – HRA *dataflow* 62
Figura 13 – Trabalho realizado em função do tipo de unidades e sua frequência 65
Figura 14 – Sistema em função do tipo recursos e sua multiplicidade 66
Figura 15 – Correlação de eficiência entre um tipo de unidade de trabalho e um tipo de recurso 67
Figura 16 – Caso-de-uso "Processar UT" pelo *renderer* 71
Figura 17 – Caso-de-uso "Obter resultados" pelo *renderer* 73
Figura 18 – Caso-de-uso "Iniciar sistema" realizado pelo HRA 74
Figura 19 – Diagrama de actividades para o processamento de uma UT 76
Figura 20 – Diagrama de actividades para inicialização do sistema 78
Figura 21 – Diagrama de classes do modelo proposto 81
Figura 22 – Diagrama de estados de uma unidade de trabalho 84
Figura 23 – Diagrama de estados de um Contentor 85
Figura 24 – Diagrama de estados do HRA 85
Figura 25 – Diagrama de estados de um Recurso 86
Figura 26 – Divisão do plano imagem. 88
Figura 27 – Distribuição de raios que passam por um *pixel* 89
Figura 28 – Intersecção de um raio com uma superfície 90
Figura 29 – Raios secundários definidos segundo o ângulo de incidência e BRDF (Reinhard 1994) 93

Figura 30 – Divisão e dependências entre tipos de primitivas definidas 104
Figura 31 – Composição geral de uma unidade de trabalho..... 106
Figura 32 – Modelo de comunicação *sliding window* [Img05]..... 112
Figura 33 – Código ilustrativo da definição de tipo de dados de base..... 120
Figura 34 – Código ilustrativo da definição das estruturas complexas..... 120
Figura 35 – Código (declaração) ilustrativo da relativa independência dos dados ao alto nível 121
Figura 36 – Código (função) ilustrativo da relativa independência dos dados a alto nível 122
Figura 37 – Cenário A com esferas e pirâmides e inter reflexões (CPU)..... 126
Figura 38 – Cenário B com esferas e inter reflexões (CPU+GPU) 127
Figura 39 – Interacção entre níveis na configuração CPU sem HRA 130
Figura 40 – Interacção entre níveis na configuração CPU com HRA 133
Figura 41 – Interacção entre níveis na configuração CPU + GPU com HRA..... 136

1 INTRODUÇÃO

Desde o seu aparecimento, a computação gráfica esteve sempre ligada a desenvolvimentos tecnológicos ao nível do hardware, essencialmente pela sua natureza de apresentação visual e necessidade de poder de cálculo intensivo. Quando Kajiya apresentou da equação de *rendering*, envolvendo as contribuições de todas as fontes de luz para cada ponto de cada superfície (luz reflectida e luz auto-transmitida), tornou-se claro que o esforço de cálculo necessário para gerar a síntese de uma imagem seria elevado, mesmo utilizando resoluções reduzidas na imagem produzida.

A equação apresentada também sugere a existência de uma grande quantidade de cálculos independentes entre si. Esta característica é um aspecto importante, que determinou desde cedo o recurso a soluções de implementação de síntese de imagem (*rendering*) em arquitecturas inspiradas no paradigma de computação paralela. Esta independência de dados não só se manifesta num plano mais geral, na soma das várias componentes de luz num determinado ponto (luz reflectida e luz auto-transmitida), como também na contribuição de cada fonte de

luz para a iluminação de cada ponto. Existe ainda um terceiro plano de independência, pois a síntese de cada ponto da imagem não é directamente dependente dos valores dos pontos vizinhos.

O advento de sistemas computacionais com custos relativamente reduzidos, o que os torna acessíveis a novos públicos, permitiu um conjunto de desenvolvimentos tecnológicos que colocam hoje ao alcance de uma maior faixa de utilizadores equipamentos com desempenho da ordem de vários Gigaflops (*floating point opertations per second*).

Esta situação poderá levar a uma mudança de paradigma, com os algoritmos de rasterização adoptados nos equipamentos actuais ceder lugar aos algoritmos de *rendering* rápido de cenários complexos, utilizando iluminação global.

Com efeito, o desempenho oferecido por um computador actual não se deve apenas ao contínuo aumento da frequência de relógio do CPU, mas também ao rápido desenvolvimento das capacidades de outros componentes fundamentais do sistema computacional, incluindo as placas gráficas (GPU) e outras placas adicionais de processamento, bem como também aos sistemas de comunicação entre componentes.

A nível da computação gráfica, a abordagem tradicional de implementação de um algoritmo de *ray tracing* centrava esforços na exploração das potencialidades de um único recurso (ou tipo de recurso), ou procurava atribuir um conjunto de tarefas diferenciadas a tipos específicos de hardware. Esta abordagem conduz normalmente a situações de insuficiente aproveitamento dos recursos disponíveis, nomeadamente pela falta de balanceamento do trabalho de cálculo atribuído aos vários recursos. Tipicamente, cada tipo de recurso desempenha um conjunto específico de tarefas (e.g. a placa gráfica calcula intersecções, o CPU assegura a gestão de processo).

Assim, cada vez mais, os projectistas colocam esforços no desenvolvimento de sistemas mais completos, explorando a cooperação entre unidades funcionais, em detrimento de uma visão somente focada no aumento de velocidade do CPU.

Surgem sistemas computacionais com uma maior complexidade, agregando unidades funcionais diversas, cuja programação exige novas abordagens.

É importante definir estratégias e organizações capazes de utilizar e gerir os recursos disponíveis em computadores mais recentes, de uma forma eficiente e balanceada. Neste sentido, nesta tese é proposta a criação de uma interface abstracta entre o ciclo principal de interacção de *rendering* de alto nível e o hardware disponível, capaz de assegurar uma maior independência entre estes dois níveis. É definido um modelo de reserva e gestão de recursos, designado abreviadamente por HRA, acrónimo da designação "*Hardware Resource Allocation model*", que permite uma distribuição adequada do esforço computacional pelos recursos disponíveis.

O modelo proposto apresenta um conjunto de características, com reflexos a vários níveis:

- Portabilidade ao nível de vários aspectos, como a utilização de novos e mais poderosos recursos à medida que este forem disponibilizados;
- Optimização do código ao nível dos recursos físicos;
- Modularidade e balanceamento de trabalho;
- Aspectos mais práticos relativos à estratégia de gestão e implementação em ambientes heterogéneos abaixo do nível de abstracção, onde a performance é um aspecto chave.

O presente documento encontra-se organizado em cinco capítulos, para além da bibliografia, um conjunto de apêndices e um glossário.

No capítulo seguinte, capítulo 2, é apresentada uma revisão bibliográfica, onde são abordados alguns conceitos gerais de computação gráfica, com ênfase na síntese de imagem, recorrendo a modelos físicos da simulação do comportamento da luz. São abordados aspectos relevantes para a síntese de imagem recorrendo ao paradigma de processamento paralelo. São também

referidas abordagens adoptadas para a aceleração do processo de *rendering* em equipamentos específicos e soluções mistas.

No capítulo 3 é proposto e especificado um modelo de reserva e gestão de recursos, o HRA "*Hardware Resource Allocation model*", expondo este conceito e abordando problemas relacionados com a utilização de ambientes compostos por um conjunto de recursos de hardware heterogéneos, como seja o caso da representação de dados, os paradigmas de programação, a comunicação entre os vários recursos. São ainda propostas soluções e analisadas questões de engenharia emergentes.

O capítulo 4 é referente ao trabalho experimental realizado no sentido de corroborar o modelo proposto no capítulo anterior. É descrita a metodologia adoptada, apresentadas as experiências e configurações realizadas, bem como a análise dos resultados obtidos.

Finalmente, no capítulo 5, são apresentadas as conclusões sobre o trabalho desenvolvido e modelo proposto, para além de propostas de trabalho futuro relacionadas com melhoramentos e adaptações de algumas das características do modelo.

2 REVISÃO BIBLIOGRÁFICA

O presente capítulo tem como objectivo apresentar ao leitor uma breve descrição do estado da arte no campo da computação gráfica, mais especificamente na utilização de múltiplos recursos para a aceleração do processo de *rendering* em ambientes potencialmente paralelos. O capítulo encontra-se estruturado da seguinte forma: No sentido de enquadrar o leitor com o tema abordado é feita uma abordagem de uma visão mais genérica da computação gráfica, que inclui alguns factos históricos relevantes no ponto 2.1. Ainda dentro deste ponto é apresentada alguma teoria e técnicas de base utilizadas na área, para um melhor enquadramento de opções ou abordagens utilizas, referidas quer neste capítulo quer no capítulo 3, referente ao modelo proposto.

No ponto 2.2, é apresentada um breve descrição das características principais dos algoritmos de *rendering* com inclusão da componente de iluminação global, para além das componentes directa e indirecta, no sentido de expor especificidades destes tipos algoritmos e soluções desenvolvidas com o objectivo de acelerar o processo de obtenção do resultado final em tempo real. Ainda dentro

desta perspectiva, no sentido de complementar a informação fornecida, o ponto 2.3 abordada a temática da computação paralela aplicada à computação gráfica. Neste ponto são referidos aspectos ligados à evolução tecnológica dos sistemas e modelos utilizados, familiarizando o leitor mais distante desta área com alguma terminologia e conceitos adoptados na área da computação paralela.

Finalmente no ponto 2.4 e respectivos sub pontos, são abordados trabalhos desenvolvidos em aspectos específicos e em elementos distintos do processo de *rendering*, tendo em conta recursos e tecnologias disponíveis no mercado, no sentido de aumentar a eficiência dos sistemas e minorar os tempos de cálculo. Estas abordagens versam a optimização da implementação do algoritmo de *rendering* no CPU, com a utilização de recursos massivamente paralelos ou adaptação da configuração do hardware em relação à tarefa a realizar. No fim deste ponto são referidas algumas soluções híbridas que utilizam mais do que uma das abordagens referidas.

Ainda dentro deste ponto e na perspectiva do *rendering* e dos projectos descritos, é apresentado o projecto onde foi desenvolvido algum do trabalho nesta tese, composto pelas várias facetas e vertentes: O projecto foi desenvolvido, incluindo aspectos desde do sistema de visão humano até ao paralelismo dos recursos utilizados no processo de *rendering*.

2.1 Rendering

Noção no sentido mais lato do termo, *rendering* é o processo de geração de uma imagem bidimensional, representativa da observação de um cenário do ponto de vista do olho humano ou de uma câmara, baseado na descrição contextual da informação sobre geometria, texturas, luzes e comportamento óptico dos elementos presentes no referido cenário (Kajiya 1986; Wald 2004; Benthin 2006). Alguns autores aplicam o termo de uma forma mais restritiva (Wald 2004), associando o *rendering* ao conjunto de algoritmos de *ray tracing* existentes. Neste sentido restrito, o *rendering* é equivalente ao termo *ray tracing*, deixando de fora

os processos de rasterização, os quais, por vezes, são considerados por estes autores como uma aproximação do que se pode obter com um algoritmo de *ray tracing*. Na presente tese o termo *rendering* é aplicado no sentido do domínio *ray tracing*, salvo indicação em contrário.

A evolução tecnológica dos sistemas informáticos em diversas vertentes providenciou às áreas científicas afins, isto é, as que utilizam sistemas computacionais, uma alargada capacidade de desenvolvimento, quer em termos de capacidade de cálculo (Wald 2004; Benthin 2006), quer em termos de dispositivos de aquisição de dados e apresentação ou representação de resultados. Esta evolução fez com que a qualidade apresentada, grandemente influenciada pela quantidade de cálculos efectuados e iterações pretendidas, se estendam para novos limites. Mesmo assim, em algumas soluções existentes, as suas respostas não se coadunam com as exigências actuais de qualidade, em termos de tempo ou de cálculos necessários para composição final da imagem ou as animações.

É cada vez maior a utilização da computação gráfica, nomeadamente dos algoritmos de *rendering*, em vários aspectos da vida social (CGSociety 2005; CGSociety 2008) como sejam a simulação (Plachetka 2001; ArchitectureWeek 2002; Kurenov 2006), cinema (Gutierrez 2004; Christensen 2006; Pixar 2008; Pixar 2008), jogos (Sung 2007; EA 2008) entre outros. A interacção requerida neste tipo aplicações veio colocar uma forte exigência na diminuição do tempo de cálculo, mais especificamente nos algoritmos baseados em modelos físicos do transporte da luz (Ward 1999). A inclusão de mais componentes ou refinamentos dos cálculos (Ward 1988; Ward 1992) na síntese de imagens e animações implica a utilização de recursos cada vez mais eficientes e poderosos em termos de poder de cálculo, mas também recorrendo à integração e utilização de múltiplos componentes e características dos modelos de iluminação em simultâneo.

É possível afirmar que duas abordagens foram seguidas no que respeita à geração da imagem a partir de algoritmos de *rendering*:

- A criação de ferramentas *off-line* para a síntese de imagens e animações de elevada qualidade, considerando que a qualidade deriva do número de aproximações e componentes utilizadas (Ward 1998);
- Aplicações interactivas que minimizam determinados aspectos ou reduzem a qualidade final, permitindo a interactividade pretendida (Parker 1999; Luebke 2001; Wald 2003; Günther 2004; Pomi and Slusallek 2004; Schmittler 2004).

O fosso que separa estas duas abordagens tem vindo a ser reduzido (Wald), sendo possível, hoje em dia, dotar as aplicações interactivas de um grau elevado de realismo (Wald 2004), recorrendo a componentes de hardware standard ou específico (Purcell; Wald 2005), à adaptação dos algoritmos de *rendering* a paradigmas específicos (Purcell 2004), ou através da optimização dos fluxos de dados e instruções no hardware subjacente (Wald).

Tradicionalmente, em aplicações interactivas, os algoritmos de rasterização para a produção de imagens e animações têm predominado sempre que a componente interactiva é importante, como seja o caso das plataformas para jogos. Esta situação é explicada pelo facto de os algoritmos de rasterização produzirem resultados num tempo adequado e com qualidade bastante aceitável para os fins pretendidos.

A evolução tecnológica permite hoje a utilização de algoritmos baseados na simulação do comportamento físico da luz, nomeadamente de algoritmos de *ray tracing* em alternativa aos tradicionais algoritmos de rasterização. Duas razões importantes explicam esta tendência:

- Nos algoritmos de rasterização é necessário recorrer a artifícios para contemplar aspectos tais como sombras, transições suaves entre sombras e objectos, ou inclusão da componente de iluminação global, enquanto nos algoritmos de *ray tracing* estas componentes são parte integrante na própria definição do algoritmo genérico;

- As dimensões das imagens produzidas têm vindo a aumentar, como resposta ao aumento das dimensões dos dispositivos de apresentação (*displays*) (Chen 2001) e à crescente procura de maior detalhe nos cenários reproduzidos. Acontece que os algoritmos de rasterização são caracterizados por uma progressão linear do tempo de síntese da imagem em relação às dimensões e ao detalhe das imagens, enquanto que os algoritmos de *ray tracing* se caracterizam por uma progressão logarítmica do tempo de síntese da imagem, em relação a complexidade geométrica e ao número de fontes de luz do modelo (Wald). Para além deste aspecto, é possível, no *ray tracing*, reutilizar diversos cálculos anteriormente efectuados quer nas componentes de iluminação directa e indirecta, quer na componente de iluminação global (Ward 1998), reduzindo ainda mais o tempo de síntese global.

2.1.1 Ray tracing

O processo de *ray tracing* (Kuchkuda 1988) para a produção de imagens sintetizadas em computador designa um conjunto de métodos (Wald). Estes métodos baseiam-se na projecção ou "lançamento" de vários raios com origem num ponto do espaço tridimensional. O objectivo é determinar as contribuições directas das fontes de luz para a iluminação no plano imagem do ponto de vista escolhido (raios primários), as contribuições indirectas provenientes de outros objectos através do processo de retransmissão da luz recebida (raios secundários) e da contribuição da iluminação global. A componente de iluminação global pode ser determinada através de algoritmos de amostragem do cenário e interpolação de valores intermédios em falta (Ward 1988), ou pela inclusão de fontes de luz virtuais que simulam esta contribuição (Cohen 1993).

O processo de cálculo das contribuições de cada fonte de luz para a iluminação de cada ponto do plano da imagem é realizado de uma forma mais eficiente pelo método denominado de *back tracing*. Os cálculos das contribuições

são efectuados começando no destino, o ponto de vista do observador ou do equipamento de visão, para as fontes de luz. Outras contribuições, indirectas, são calculadas de modo idêntico de uma forma recursiva, isto é, tomando como ponto de partida o ponto de intersecção do raio inicial com a superfície intersectada. Este processo é mais expedito do que o método de *forward tracing* em que é realizada a análise dos efeitos da emissão de todas as fontes de luz para todo o espaço 3D, incluindo os efeitos de retransmissão da luz nos objectos presentes no cenário. Neste caso são calculadas "todas" as imagens e não só a imagem visível do ponto de vista do observador. Este método é irrealizável no *ray tracing*, excepto no caso de se optar por muitas limitações, o que torna a imagem produzida de qualidade duvidosa [1].

$$L_o(x, \mathbf{w}) = L_e(x, \mathbf{w}) + \int_{\Omega} f_r(x, \mathbf{w}', \mathbf{w}) L_i(x, \mathbf{w}')(\mathbf{w}' \cdot \mathbf{n}) d\mathbf{w}'$$

Figura 1 – Equação de *rendering* (Kajiya 1986)

Todas as contribuições para a iluminação de um ponto anteriormente referidas são sumariadas na equação da Figura 1, apresentada por Kajiva (Kajiya 1986). A energia luminosa total que sai do ponto *x* na direcção *w* é igual a soma da energia emitida com o somatório das contribuições da energia recebida segundo o ângulo *w´* dentro do domínio de integração Ω. A energia retransmitida é uma parte da energia recebida, modelado pelo índice de reactância, f_r, que por sua vez é dependente do ângulo incidente, ângulo de saída e posição considerada. Esta componente é ainda influenciada pelo produto interno do vector incidente com normal à superfície *n*, no ponto *x*.

Como se pode depreender do processo exposto anteriormente, uma das operações básicas do método de *ray tracing* prende-se com a determinação da intersecção dos raios primários de luz com as superfícies dos objectos. Uma outra

[1] Alguns algoritmos poderão utilizar parcialmente, para determinadas situações, o método *forward tracing*, como seja o caso da técnica de mapeamento de fotões (*photon mapping*) Jensen, H. W. (2001). Realistic Image Synthesis Using Photon Mapping, AK Peters, Ltd..

operação semelhante consiste na determinação da intersecção dos *shadow rays* (Reinhard 1999) ou raios secundários com as superfícies dos objectos. Designam-se por *shadow rays* os raios "lançados" a partir dos pontos de intersecção dos raios iniciais com as superfícies do espaço 3D, no processo de *back traycing*. Estes raios só são efectivamente úteis se intersectarem um objecto ou um ponto de luz, situação na qual serão efectuados os respectivos cálculos.

2.1.2 Back traycing

O método de *back tracing* é utilizado nos algoritmos *ray tracing*, tendo o nome sido atribuído devido ao facto de funcionar no sentido inverso ao trajecto dos raios em relação ao fenómeno físico. Os cálculos de iluminação são efectuados partindo do ponto de vista de onde se pretende visualizar o cenário, até chegar às fontes de luz, passando pelos objectos iluminados directamente e indirectamente, bem como os raios provenientes directamente das fontes de luz ou alterados por outros objectos. Este método implica um menor número de cálculos relativamente a outras abordagens, pois uma grande parte dos raios "lançados" contribuem para a síntese da imagem pretendida (Ward 1998; Shirley 2003; Pharr 2004), do ponto de vista definido, nomeadamente os raios que intersectam os objectos ou pontos de luz.

Como se pode observar na Figura 2, são gerados raios com origem no ponto de vista que passam pelo plano da imagem a sintetizar. Por sua vez, a partir destes raios gera-se um conjunto de raios denominados de *shadow rays*. Sempre que seja encontrado (intersectado) um objecto do modelo, no ponto de intersecção mais próximo do ponto de vista do raio com o objecto, são "lançados" novos raios com origem no referido ponto de intersecção, de uma forma semelhante aos raios primários gerados a partir do ponto de vista. No caso de ser encontrada uma fonte de luz é então calculada a respectiva contribuição da componente de iluminação directa.

Se o raio projectado tem um ângulo oposto ao raio incidente em relação à normal da superfície no ponto de intersecção, essa componente é definida como componente especular, caso seja um raio com um ângulo diferente e intersecte uma fonte de luz, o valor calculado é definido como componente difusa.

Analisando a Figura 2 poder-se-á observar que o raio *r2* é um raio de componente especular, supondo que o seu ângulo é o oposto ao ângulo do raio incidente *r1*, em relação à normal no ponto de intersecção de *r1* com o objecto (fonte de luz) em forma de esfera *F1*. O raio *r6*, não oposto ao raio *r1* no ponto de intersecção, é uma contribuição de luz difusa de uma segunda fonte de luz *F2*.

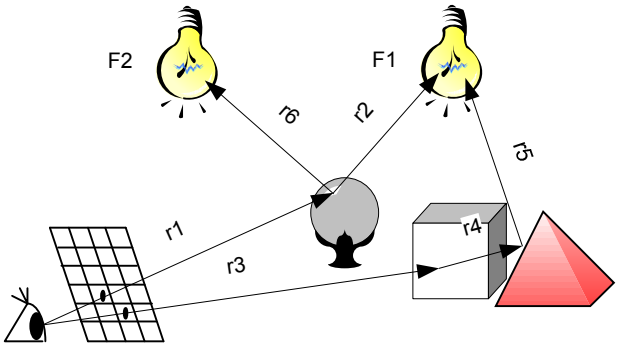

Figura 2 – Raios primários e secundários no *back tracing*

Os *shadow rays* são gerados de uma forma aleatória mas seguindo uma função de distribuição predefinida (Ward 1988; Shirley 2003; Pharr 2004). O objecto deve ser amostrado de uma forma correcta e expressiva para que as contribuições da iluminação no ponto de intersecção do raio resultem numa aproximação com o menor erro possível em relação ao valor real. Existem diversas abordagens e metodologias no que se refere à distribuição das direcções dos raios gerados (Shirley 2003), apresentando cada uma delas vantagens e desvantagens em aspectos como tempo do processo de *rendering* ou "qualidade" obtida nas diferentes componentes de iluminação, nomeadamente nas componentes de iluminação directa e indirecta (Shirley 2003).

O processo de *back tracing* é repetido de uma forma recursiva, com os *shadow rays* tomando o lugar dos raios primários e produzindo mais *shadow rays*, caso seja encontrado um objecto. Cada vez que este processo é repetido é incrementado o valor do "*bounce*" do raio. Este valor de "*bounce*", na realidade um valor máximo predefinido para este, é utilizado como parâmetro para quebrar o algoritmo recursivo.

Um valor mais elevado para o número máximo de *bounces*, implica uma maior aproximação dos valores calculados em relação aos valores expectáveis para a representação da síntese de imagem obtida. No entanto, o poder de cálculo necessário aumenta de uma forma exponencial em relação ao valor máximo de *bounces* predefinido, no caso de não se utilizar optimizações ao nível algorítmico (Ward 1999).

Após ser atingido o número máximo de *bounces* ou se encontrar uma fonte de luz o processo recursivo é finalizado, devolvendo ao nível de interacção anterior o cálculo da iluminação directa, no caso de ser uma fonte de luz, ou iluminação indirecta já calculada no caso de um objecto que retransmite luz recebida. No final vão ser consideradas todas as contribuições de todas as componentes da iluminação para cada *pixel* do plano imagem definido.

É de salientar que na realidade não é projectado apenas um raio por *pixel* do plano imagem definido para um ponto de vista, mas sim vários e com distribuições distintas no plano constituído pela área do *pixel* em questão, de forma a evitar efeitos de *aliasing* (Ward 1998; Shirley 2003; Pharr 2004). Dentro deste aspecto, número de raios por *pixel*, são ainda englobados os tópicos de sob-amostragem e sub-amostragem, como forma de melhorar a qualidade visual de um determinado pormenor ou evitar demasiados cálculos num local de pouca importância em relação à tarefa para a qual a imagem é produzida (Cater 2003).

O número de raios utilizados por *pixel* pode variar dinamicamente, no sentido em que este valor pode depender dos dados independentes descritivos do cenário ou da progressão dos valores calculados no decorrer da execução do processo de

rendering. Esta abordagem poderá ser intitulada de amostragem variável ou *dynamic sampling* [2].

2.1.3 Operações básicas no *rendering*

No essencial o processo de *rendering* de um modelo é composto por duas operações elementares:

- Cálculo dos pontos de intersecção de cada raio, raio de luz ou outro raio, por exemplo um *shadow ray*, com os objectos;
- Cálculo da iluminação para cada um dos referidos pontos, devida a luz directa ou indirecta das fontes de luz, e iluminação global caso seja considerada, tendo em conta o tipo de material e o seu comportamento fotométrico.

Todos os outros tipos de operação existem para coadjuvar estas duas primitivas essenciais, como o seja o atravessamento da estrutura de dados representativa da geometria do modelo do cenário ou pré-processamento para agilizar e minimizar os cálculos efectuados no processo de *rendering* ou o cálculo da normal num ponto de uma superfície.

O algoritmo de *ray tracing*, no seu essencial, procura saber quais os raios relevantes que passam pelo plano imagem, e seus derivados, que intersectam objectos do modelo, calculando de seguida as contribuições dos outros objectos e de fontes de luz para o valor da iluminação do ponto na superfície do objecto tento em conta o ponto de vista e características do material.

A partir do processo global anteriormente descrito é possível aduzir algumas características do referido algoritmo que podem e foram aproveitadas para melhorar a sua eficácia:

[2] Este conceito é distinto da sob-amostragem variável e a sub-amostragem variável de forma separada.

- O cálculo da intersecção de um raio é independente de outro qualquer raio, o que torna o algoritmo facilmente paralelizável (Chalmers 2002);
- O cálculo da contribuição de uma fonte de luz num ponto de uma superfície é independente das outras fontes de luz, dado que o valor final é igual à soma algébrica de todas as contribuições (sistema linear). Novamente será possível paralelizar o cálculo das componentes. Ainda dentro deste aspecto, como o sistema é linear, a ordem de cálculo das componentes é irrelevante (comutatividade da soma), o que facilita a distribuição de carga computacional por diversos elementos;
- Como um raio pode gerar um novo conjunto de raios para refinamento do valor calculado, o algoritmo é recursivo. Por este motivo, para além da simplicidade das operações básicas, é fácil de implementar o algoritmo de base numa grande variedade de equipamentos computacionais;

Uma abordagem mais profunda destes temas e uma indicação da terminologia sobre computação paralela, mais especificamente aplicada à computação gráfica, é referida no ponto 2.3.

2.1.4 Intersecções

O cálculo das intersecções no processo de *ray tracing* é baseado na fórmula $R(t) = O + tD$ (Shirley 2003), em que $R(t)$ denomina o trajecto do raio no espaço tridimensional, O o ponto de origem do raio, quando t toma o valor zero e D o vector da direcção do raio. O valor escalar das componentes do vector D não são fundamentais, pois apenas são utilizados para determinar o valor de t, tal que $R(t)$ seja um ponto de intersecção do raio numa superfície de um objecto do espaço 3D. O intervalo de valores da variável t situa-se entre 0, origem, e um valor t_{max}. O valor t_{max} é definido como um valor para o qual o raio está fora da zona definida no modelo do cenário. Na realidade não são testados todos os valores de t no intervalo definido, o que é impossível pois t, segundo a definição, é um valor contínuo. Na realidade são calculados os valores de t que são solução do sistema

de equações composto pela descrição geométrica da superfície ou superfícies de um objecto e a equação da recta que representa o raio que se pretende testar na direcção e sentido definidos.

Duas observações devem ser consideradas em relação ao método descrito. No algoritmo de *ray tracing* os raios podem ter a sua origem na superfície de um objecto, como seja o caso dos raios denominados de secundários, já referidos anteriormente. Para evitar o fenómeno de auto oclusão (Shirley 2003; Wald 2004), e dado que numa implementação prática os valores numéricos são discretos, o intervalo para t é definido como sendo um valor entre um mínimo definido no sistema, denominado de ε, e t_{max}. A segunda observação prende-se com os valores negativos de t. Neste caso, o raio representado é o mesmo, apenas no sentido oposto àquele que pretendemos testar. Desta forma, apenas os valores de t dentro do intervalo $[\varepsilon, t_{max}]$ são considerados válidos na solução do sistema de equações descrito anteriormente, e caso existam isso significa que existe intersecção do raio pretendido com a superfície do objecto testado. No caso de se obter múltiplas soluções de t dentro do intervalo definido, é considerada como solução correcta aquela que apresenta o valor mas baixo de t, pois esse valor representa a primeira intersecção do raio com o objecto, mais propriamente com a superfície do objecto.

Em implementações práticas deste tipo de algoritmo o conjunto de superfícies utilizado, relativamente às equações representativas das referidas superfícies, é limitado. É utilizada uma aproximação de todos os objectos através de um conjunto de um só tipo de superfície, como seja os casos de Wald et al. (Wald) e das placas gráficas (polígonos) (Buck 2004; Wald 2005), até a um leque mais diversificado de superfícies como seja o caso do software Radiance (Ward 1998). Sendo os objectos representados pela junção de uma ou mais formas 3D ou limitados por superfícies 2D no espaço 3D, o teste de intersecção de um raio com um objecto é efectuado nas formas ou superfícies que representam o objecto.

2.1.5 Geometria

A geometria, descrição da forma física dos objectos, e a sua representação ao nível computacional tem uma grande influência na qualidade da síntese de imagem obtida e na quantidade de cálculos necessária para se obter a referida imagem.

A geometria definida para um objecto é uma aproximação da representação física, exemplo na Figura 3, excepto no caso de objectos com representação perfeita na geometria do sistema utilizado, como é por exemplo uma esfera perfeita que é representada correctamente em plataformas ou sistemas de *rendering* que suportam a representação matemática de uma esfera, como seja o caso do pacote de software Radiance (Ward 1998). Não são considerados como erros de representação as aproximações e limitações no cálculo devidos à representação discreta e à quantificação de números reais em sistema computacionais.

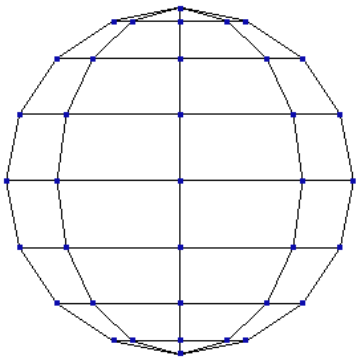

Figura 3 – Uma aproximação geométrica de uma esfera através de polígonos

Algumas plataformas utilizam formas básicas limitadas a apenas um só elemento de representação de base, ver Figura 3, como seja o caso do subsistema gráfico utilizado na plataforma PC, nomeadamente nas placas gráficas. Nesta plataforma são utilizados polígonos bidimensionais, quase sempre triângulos por questões de optimização e aceleração do cálculo, para representar a geometria dos objectos presentes num cenário. Este tipo de abordagem deve-se essencialmente

da baixa programabilidade do hardware subjacente ao nível das estruturas mais complexas e na redução substancial da implementação dos algoritmos, código, equações para múltiplas formas de base. Este tipo de abordagem usa essencialmente dados simples, pontos representados por matrizes de valores reais e apresenta um pequeno *footprint* (quantidade de instruções) na implementação dos algoritmos definidos. Isso não significa que o hardware actual não seja capaz de contornar estas limitações (Buck 2004). No entanto, o desenvolvimento e evolução do referido hardware tem em conta a compatibilidade com os sistemas/versões anteriores.

Outro aspecto importante na descrição da geometria está relacionado com a representação do modelo e organização das formas de base dos objectos no espaço tridimensional. É possível utilizar representações auxiliares para acelerar, no sentido de minimizar, o tempo total de cálculo das intersecções dos raios com as superfícies dos objectos. Talvez a melhor abordagem para elucidar a utilização de formas auxiliares será a utilização de um exemplo. Consideremos um objecto com múltiplas formas de base, como seja o caso de uma árvore. Em vez de testar se um raio intersecta cada uma das superfícies que compõem o objecto, é possível envolver, limitar, o espaço 3D do objecto por uma única forma básica, uma primeira aproximação mais rude da geometria. Essa forma pode ser um paralelepípedo (6 rectângulos 2D) que envolve todas as superfícies da árvore. No caso de um raio não intersectar o paralelepípedo envolvente, é garantido que não intersecta qualquer um dos polígonos que representam a geometria da árvore, reduzindo significativamente o número de cálculos necessários (Shirley 2003).

Estas estruturas auxiliares podem ser aplicadas à divisão do espaço 3D, em vez de envolver objectos individuais (Shirley 2003; Pharr 2004). Neste caso, as estruturas auxiliares limitam diversos objectos, ou partes de objectos se todo o modelo for transformado nas suas formas básicas, como seja o caso de polígonos. Neste tipo de abordagem, divisão do espaço 3D, temos como exemplos a utilização de *octrees* (Ward 1998) e *btrees* (Pharr 2004), que são descritas mais detalhadamente no ponto 2.1.7.

2.1.6 Estruturas de representação em *ray tracing*

Existe um conjunto de representações elementares que devem ser consideradas no desenvolvimento de um *ray tracer*. De seguida são indicadas quais e em que situações estas representações são utilizadas:

- Escalar – Valor elementar a partir do qual são definidos todos ou quase todos os outros elementos ou estruturas de representação dentro do sistema desenvolvido. É possível optar por um valor do tipo inteiro de forma a maximizar a rapidez de cálculo em detrimento da precisão dos cálculos e consequente aumento do erro de aproximação. Também se optar-se pela representação dos valores pela representação do tipo real em vírgula flutuante (IEEE 1985), tendo esta última representação uma maior precisão mas consequente aumento no tempo de cálculo para o mesmo conjunto de dados. Sempre que a qualidade da imagem sintetizada, entenda-se aproximação mais correcta aos valores esperados, é um factor preponderante, a utilização de representações numéricas de maior precisão é importante. No caso de se pretender minimizar o tempo de cálculo, a escolha da representação através de valores inteiros é favorecida. Devido ao desenvolvimento da micro-electrónica e processadores ou arquitecturas dedicadas ao cálculo de valores reais (*float*) e matrizes de reais (IntelCorporation 2006), o fosso diferencial em termos de tempo de cálculo entre as duas abordagens tem vindo a diminuir. Também o tipo de plataforma onde se pretende implementar o *ray tracer* e o suporte para um determinado tipo de representação pode condicionar a escolha (Tisnovsky 2002);
- Ponto – É um elemento definido por três escalares que representam as coordenadas de um ponto no espaço 3D. Em algumas implementações (Tisnovsky 2002), o tipo de representação utilizada não é a mesma para os diferentes tipo de estruturas ou cálculos, dentro do mesmo tipo de característica representada. A titulo de exemplo em determinadas

implementações os pontos são representados por valores inteiros, sendo os cálculos efectuados na representação de vírgula flutuante (Zemcik 2003), ou utilizados factores escalares do tipo real (*float*), onde é efectuada a conversão dos dados e termos para valores inteiros, efectuando-se o cálculo utilizando aritmética inteira (Zemcik 2003);

- Vector – É definido por um coeficiente escalar para cada dimensão em coordenadas cartesianas ou dois ângulos e a norma do vector em coordenadas esféricas, no caso 3D (Pharr 2004). Na representação em coordenadas esféricas os ângulos são definidos em relação a dois dos três planos *XY*, *YZ* e *XZ*. São considerados vectores sem ponto de aplicação. Os vectores são utilizados na maioria dos casos para indicar a direcção e sentido de um raio, sendo a norma relevante quando se pretende calcular em que local um raio intersecta uma superfície (Shirley 2003);
- Raio – Essencialmente composto por um ponto de origem, um valor escalar que representa a progressão temporal e um vector que indica a direcção e sentido de progressão do raio, sendo a sua norma normalmente normalizada (valor unitário);
- Superfície – É representada por uma equação ou conjunto de equações representativas da superfície no espaço 3D. Estas equações matemáticas são codificadas no sistema, na linguagem de implementação do mesmo, sendo apenas necessário guardar os coeficientes representativos da equação ou equações. A titulo de exemplo, uma esfera é representada pela equação $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$, em que os coeficientes escalares representativos são x_0, y_0, e z_0, o centro (ponto), e r_0 o raio da esfera. Desta forma, cada tipo de superfície é implementada através da codificação intrínseca das expressões de cálculo da intersecção de um raio e cálculo da normal num ponto da superfície;
- Polígono/triângulo – Este tipo de superfície, com expressão bidimensional, é das mais utilizadas no *rendering* pois em muitas ocasiões é a única

superfície com representação directa no sistema, entenda-se implementada directamente em código no sistema. Embora seja uma porção de um plano, é representada utilizando um conjunto de três ou mais pontos co-planares, denominados de vértices, em que os segmentos de recta entre os vértices definem os limites do segmento do plano pertencente ao objecto. É considerado como pertencente à superfície do objecto o segmento interior do plano definido pelos vértices e limitado pelas rectas. No caso de conter mais do que três vértices é obrigatório considerar a ordem dos vértices, de forma a claramente indicar as rectas que definem os limites da superfície a considerar. As rectas são consideradas entre dois vértices consecutivos e entre o último e o primeiro;

- Malha ou *Mesh* – É uma forma complexa composta por múltiplos polígonos adjacentes, usualmente triângulos, em que vários dos vértices são partilhados por polígonos adjacentes. Por uma questão de optimização, a malha é representada por conjunto ordenado de pontos (vértices) únicos e uma lista de sequências de índices (número inteiro) indicativos dos vértices pertencentes a cada polígono. A Figura 4 apresenta a geometria de um golfinho forma por uma *mesh* de triângulos;

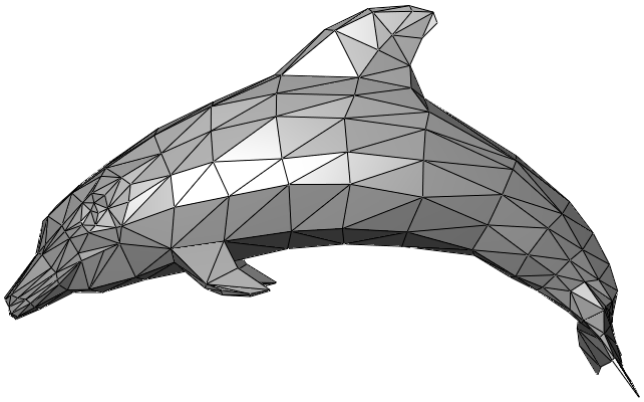

Figura 4 – Geometria de um golfinho representada por um conjunto de polígonos [Img01]

- Representação da cor – A grande maioria dos sistemas de *ray tracing*, utilizam o modelo RGB (*Red, Green and Blue*) (Hauta-Kasari 1988;

EastmanKodakCompany 1999), em que é utilizado um valor escalar para a quantificação das componentes Vermelha, Verde e Azul, Os escalares pode ser representados por um valor real em formato de vírgula flutuante ou inteiro (Schwarz 1987). A cor pode ainda ser representada por uma sequência de bits, conjunção da representação da parte inteira das respectivas componentes (Gonzalez 2006);

- Propriedades dos materiais – Cada material deve especificar o seu comportamento em relação à luz para cada componente de iluminação. Se o modelo de cor utilizado é o RGB, então cada componente de iluminação tem um conjunto de factores escalares, um para cada cor primária. O ponto 2.1.8 fornece mais informação sobre as componentes de iluminação utilizadas no *ray tracing*;
- *Pixel* – Elemento do domínio da imagem representado por uma cor, mais propriamente pelo valor de uma cor (Gonzalez 2006);
- Plano imagem – É representado por uma matriz bidimensional de *pixeis* e utilizado para armazenar o resultado da síntese. Também deve conter informação sobre a sua localização no espaço 3D aquando da síntese da imagem;
- Ponto de vista – No essencial este elemento é composto por um ponto e uma direcção/sentido. Também pode ter associado um plano imagem perpendicular à direcção do ponto de vista e respectiva distância do plano imagem ao ponto de vista. Pode-se considerar que este é o modelo mais simples de uma câmara;
- Câmara – Este elemento pode conter um conjunto de características diversas tornando-o em algumas situações bastante complexo. Pode incluir e modelar o plano imagem, ponto de focagem, lentes, obturação. Todos os elementos de base são do tipo escalar, ou compostos por estruturas mais complexas com escalares como campos (Shirley 2003).

As representações aqui apresentadas apenas são indicativas de elementos genéricos e fundamentais na implementação de um sistema de *ray tracing*. Foram omitidas estruturas e características de sistemas ou implementações específicos, como seja o caso de sistemas que entram em consideração com o comprimento de onda da luz (luz espectral) (Peercy 1995) ou polarização da luz (Tannenbaum 1994). Normalmente são situações de refinamento do algoritmo de base ou representação distinta da luz. Também não foram referidas as estruturas não directamente relacionadas com a representação dos elementos fundamentais, como seja o caso das estruturas de aceleração referidas no ponto seguinte.

2.1.7 Formas de aceleração dos cálculos

Como foi referido no ponto 2.1.5, é possível reduzir o número de cálculos totais recorrendo a estruturas auxiliares que subdividem o espaço 3D ou utilizando formas mais simples, uma primeira aproximação, que envolvem os objectos. Quer na subdivisão do espaço, quer na utilização de formas que envolvem todo o objecto, se a estrutura ou forma envolvente não é atravessada por um raio, então nenhum dos objectos interiores é atravessado pelo referido raio. Este processo evita que sejam efectuados cálculos de intersecção desnecessários, reduzindo de uma forma considerável os cálculos (Ward 1998; Shirley 2003; Pharr 2004).

A razão entre o número de objectos ou superfícies contidas num elemento da estrutura auxiliar e o número de operações de atravessamento/teste de intersecção da estrutura auxiliar deve ser elevada, mas também limitada, por forma a utilizar esta técnica de uma forma recursiva e reduzir o número de testes de intersecção efectuados. Uma razão baixa entre os elementos envolvidos e a representação do espaço envolvente, não é benéfico.

A título de exemplo, se a forma envolvente contém apenas duas formas básicas, será efectuado sempre o cálculo da intersecção com a forma envolvente e, caso o teste seja positivo, serão efectuados os testes com cada uma das duas formas envolvidas, num total de três cálculos. No caso do teste ser negativo

apenas será efectuado um cálculo. Fazendo uma análise estatística grosseira, atribuindo 50% de probabilidade a cada caso, o teste de intersecção positivo terá que efectuar um cálculo suplementar, dois para formas envolvidas e um para forma envolvente, no caso negativo reduz-se a um cálculo, um para a forma envolvente. O ganho de menos um cálculo obtido em 50% dos casos é anulado pelo cálculo supérfluo dos outros 50% dos casos.

O raciocínio anterior pode ser aplicado à subdivisão do espaço 3D. Em determinados pacotes de software, como seja o caso do Radiance (Ward 1998), é possível predefinir, como parâmetro, o número máximo de formas primitivas contidas em cada subdivisão. Caso a subdivisão contenha mais objectos que o número máximo definido, são criados novos sub espaços conforme o algoritmo definido. Este tipo de abordagem produz uma estrutura de dados em árvore (Pharr 2004), em que o número de ramos por nó depende do algoritmo escolhido para a subdivisão. Por exemplo, a subdivisão do espaço em *octree* (Ward 1998) divide o paralelepípedo do espaço 3D do modelo definido segundo as três dimensões/eixos, criando 8 sub espaços por cada espaço considerado. A decisão sobre o local onde deve ser efectuada a divisão do espaço e em que eixos deve ser o espaço dividido pode ser realizada de uma forma estática, por exemplo dividindo em dois sub espaços iguais segundo todos eixos. Numa abordagem dinâmica, a divisão depende do número de objectos e das posições que estes ocupam no sub espaço (Chalmers 2002; Shirley 2003). Também é possível que um objecto pertença a mais do que um sub espaço ou existir uma sobreposição entre os sub espaços, tudo depende o algoritmo de subdivisão utilizado (Shirley 2003).

Deve-se ter em atenção que todo este processo requer um pré-processamento que implica um atraso temporal em relação ao início propriamente dito do processo de *ray tracing* que produz os resultados pretendidos. Se a opção de dividir o espaço 3D for definida através de um algoritmo de divisão dinâmico, com maior complexidade, em vez de valores ou regras fixas, esta pode acarretar a necessidade de um maior poder de cálculo e consequente maior atraso, mais tempo, na fase de pré-processamento. A

quantificação da razão entre o ganho obtido na fase de *ray tracing* em contraste com o *overhead* devido ao pré-processamento não tem uma fórmula matemática simples para ser estimada dada a quantidade de parâmetros e algoritmos possíveis, mas também é muito dependente do modelo geométrico de cada cenário que se pretende sintetizar [3].

2.1.8 Iluminação de um ponto

A outra operação elementar do *ray tracing* é o calculo da iluminação. A iluminação de um ponto é composta pelo cálculo e soma de todas as contribuições da luz directa, indirecta e, caso seja considerada, da componente de iluminação global. O cálculo de cada uma das componentes é realizado do seguinte modo:

- Iluminação directa especular – Partindo do raio incidente, na perspectiva do *back tracing*, e do ponto da superfície onde esta foi intersectada pelo raio, é calculada a normal à superfície no referido ponto. Caso exista uma fonte de luz na direcção do ângulo oposto ao do raio incidente segundo a normal, consultar Figura 5, é determinada a quantidade de luz que essa fonte de luz envia na direcção definida e com base nas propriedades reflectoras e de retransmissão do material da superfície, é calculado o valor da componente especular de iluminação. Caso a superfície contenha uma componente de difracção é necessário calcular onde o raio difractado sai da superfície para calcular se existe uma fonte de luz nessa direcção, e se existe então é calculada a sua contribuição com base nas propriedades refractárias do material. Por vezes, como forma de simplificar o processo, a propriedade de transparência de um material não tem em conta o efeito da diminuição ou aumento da velocidade da luz e respectivo desvio do raio em relação à direcção da normal, no caso da passagem entre materiais com diferentes

[3] As formas geométricas aqui indicadas são na verdade estruturas computacionais. Não foram referidas no ponto 2.1.6, pois são considerados artifícios computacionais para melhorar a eficiência, redução de tempo, dos algoritmos de *ray tracing* e não estruturas que representam qualquer tipo de geometria específica dos objectos do cenário.

densidades. Nesta simplificação o raio incidente e o raio refractado pertencem à mesma recta;

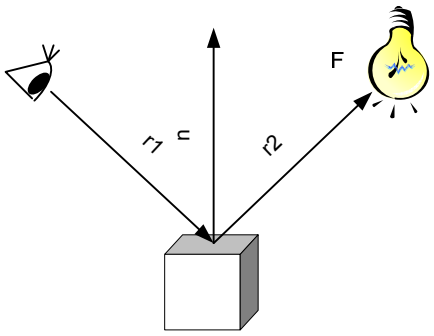

Figura 5 – Raio oposto à normal num ponto

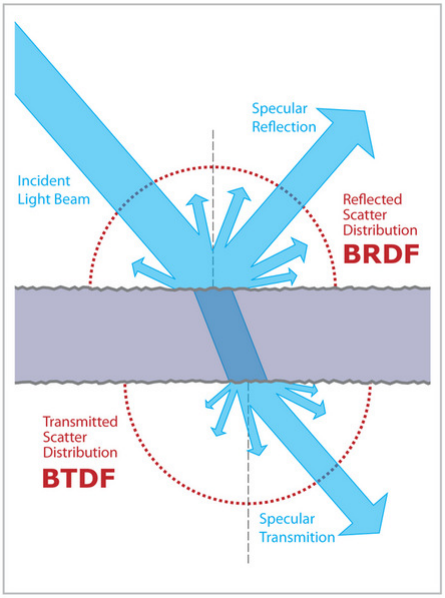

Figura 6 – BSDF [Img02]

- Iluminação directa difusa – utilizando o mesmo processo da componente especular, são calculadas as contribuições das fontes de luz agora não no ângulo oposto ao raio incidente em relação à normal, mas sim do ângulo entre a normal e o raio que intersecta uma fonte de luz. Para saber qual o efeito dessa fonte de luz no ponto pode ser utilizada uma função de distribuição de reflectância bidireccional (BRDF - *Bidirectional Reflectance Distribution Function*) (Kajiya 1986; Shirley 2003; Pharr 2004), uma das

componentes apresentada na Figura 6, considerando ainda o factor dado pelo co-seno do ângulo entre a direcção da luz e a normal no ponto (Kajiya 1986). Numa aproximação mais grosseira, considerando o BRDF unitário para todas as direcções, multiplicando a contribuição apenas pelo co-seno do ângulo entre a direcção da luz e a normal no ponto;

- Iluminação indirecta – Para além das componentes directas também é possível incluir na iluminação de um ponto, a luz retransmitida por outros objectos do modelo. Exemplos deste tipo de objectos incluem espelhos ou superfícies espelhadas (com um valor bastante elevado de componente especular). O cálculo da contribuição deste tipo de fontes de luz secundárias pode ser efectuado da mesma forma que a contribuição das fontes de luz directas, na sua componente especular e difusa (Ward 1998; Parker 1999; Pharr 2004);
- Iluminação global – Esta componente é referida em diversas ocasiões (Ward 1998; Tole 2003) como aquela que apresenta uma substancial diferença entre os algoritmos de *ray tracing* e os restantes, no que toca a suavidade das transições entre sombras e objectos, e do contraste nos limites entre objectos (Ward 1998). Normalmente produz um efeito de "amaciamento" das transições entre os *pixeis* do plano imagem representativos dos objectos e sombras. Funciona como uma componente de baixa frequência ou uma neblina de luz com valores de baixa intensidade, ou pelo menos de baixa variação espacial, em todo o cenário 3D. A sua representação é efectuada através de um conjunto de formas geométricas 3D, como semiesferas (Ward 1998), e um valor de iluminação ambiente no volume da influência (forma 3D). Esta componente é aprofundada no ponto seguinte, 2.2.

A iluminação de um ponto é obtida pela soma das componentes anteriormente descritas, que são aproximações ao modelo físico. Dependendo da complexidade do sistema implementado, poderá conter ou não todas as

componentes referidas, com uma aproximação maior ou menor, por possível amostragem de fontes de luz e grau de resolução utilizada nos cálculos de determinadas componentes da iluminação, como por exemplo o raio da esfera de influência representativa de um valor da componente de iluminação global (Ward 1998).

2.2 Algoritmos de iluminação global

Os algoritmos de iluminação global têm ganho algum protagonismo nos últimos anos em relação a outras soluções, nomeadamente em relação a rasterização, devido principalmente ao aumento da complexidade geométrica nos cenários utilizados mas também da qualidade/percepção visual pretendida para as imagens e animações geradas. Como foi referido anteriormente, no final do ponto 2.1, o diferencial entre o poder de cálculo necessário numa solução de rasterização e uma solução de *ray tracing* é cada vez mais ténue. Prevê-se que dentro em breve os algoritmos de *ray tracing* sejam menos custosos, em termos de cálculos necessários, do que os algoritmos de rasterização (Wald 2004). Esta tendência é devida principalmente aos cálculos adicionais necessárias no algoritmos de rasterização para incluir simular os mesmos efeitos produzidos pelos algoritmos de *ray tracing*, mas também pelo aumento significativo da complexidade geométrica, do número de fontes de luz e das dimensões da imagem final sintetizada.

Relativamente a componente de iluminação global, nos algoritmos de *ray tracing*, três abordagens de base são utilizadas em grande parte das implementações efectuadas: Irradiância (Radiance *Irradiaince Cache*), Photon Mapping e Radiosidade. Dentro dos pontos seguintes são apresentados as três abordagens referidas.

2.2.1 Irradiância (*Radiance Irradiance Cache*)

A técnica conhecida como *Irradiance Cache* é utilizada para o cálculo da componente de iluminação global no pacote de software Radiance (Ward 1998;

Ward 1999; Wald 2004). Esta técnica recorre a um modelo de aproximações e amostragem do meio das contribuições indirectas dos objectos do cenário em cada ponto das superfícies analisadas, utilizando o método de Monte Carlo (Shirley 2003). O algoritmo é fortemente influenciado pela forma como são reutilizados os dados previamente calculados tendo em conta dois factores:

- Os valores calculados são independentes do ponto de vista, pois é a contribuição é omnidireccional (Ward 1998; Ward 1999);
- A contribuição desta componente para a iluminação de um ponto é habitualmente reduzida, sendo que uma aproximação dentro de certos parâmetros/limites não altera significativamente o resultado produzido (Ward 1988).

Na implementação prática desta técnica, é utilizada uma estrutura de dados que armazena os valores calculados para cada ponto do espaço 3D processado e o raio de influência em que o valor é considerado válido, mais propriamente uma esfera de influência. Sempre que se pretenda calcular o valor da irradiância num determinado ponto, verifica-se primeiro se o ponto está na zona de influência de um ou mais valores previamente calculados e já armazenados na referida estrutura de dados. Caso esta situação se verifique, então é feita uma interpolação do valor pretendido a partir de todas as amostras consideradas influentes, valores de irradiância adjacentes ao ponto que se pretende calcular em que o ponto está dentro da sua esfera de influência. Dado que o valor da irradiância é omnidireccional, a interpolação é efectuada pela média das irradiâncias adjacentes, ponderadas pela distância das amostras consideradas até ao ponto em questão. Este valor não é armazenado na estrutura de dados da irradiância pois é uma interpolação de valores e não um valor realmente calculado pelas contribuições de iluminação no ponto. Caso o ponto considerado não esteja no domínio de influência de um conjunto mínimo de valores previamente calculados, então é feita uma amostragem do meio através da projecção de raios para calcular a contribuição indirecta dos outros objectos na iluminação do ponto pretendido. Este

valor calculado é armazenado na estrutura de dados, denominada de *Irradiance Cache*, para futura utilização em interpolações.

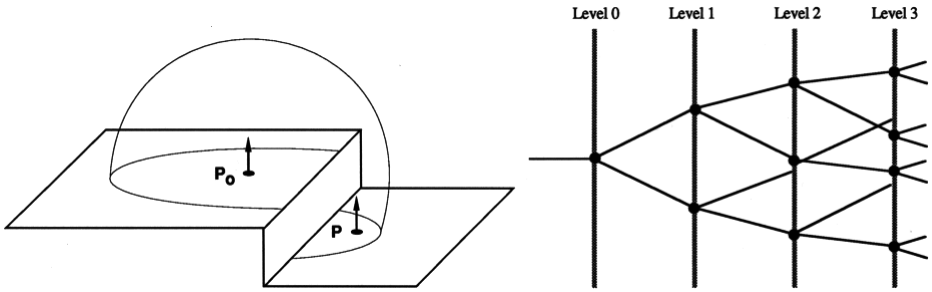

Figura 7 – Irradicância e zona de influência das amostras (Ward 1988)

Na Figura 7 – Irradicância e zona de influência das amostras (Ward 1988) os níveis representam raios e as intersecções pontos calculados.

Este método apresenta grandes vantagens na síntese de animações, pois como foi referido os valores calculados são independentes do ponto de vista. Por esse motivo é possível reutilizar valores previamente calculados na síntese de imagens anteriores com pontos de vista distintos. Esta assunção é valida para cenários em que geometria e as fontes de luz não mudam de posições (estáticas), o que não acontece em modelos com alteração das características fotométricas das fontes de luz ou da variação da geometria por deformação ou movimento dos objectos do cenário. É possível ainda alguma reutilização, desde que exista movimentos de pequenas dimensão ou movimentos em uniformidade de diversos objectos (Wald 2003).

2.2.2 Radiosiodade

O processo de cálculo da contribuição da componente de iluminação global denominado de radiosidade é efectuado com base na teoria de transferência de energia (Cohen 1993; Pharr 2004) entre superfícies. Neste caso mais propriamente dito transferência de energia luminosa ou mais especificamente reflexão de luz difusa entre superfícies. As superfícies são dividas em *patches*, segmentos com

composição bidimensional, que recebem energia luminosa proveniente de outros *patchs* considerados. O cálculo da energia recebida tem em conta o ângulo entre a normal à superfície receptora e a direcção que une as duas superfícies.

O algoritmo é baseado em sucessivas aproximações da iluminação calculada no plano imagem definido para o ponto de vista considerado, como se pode observar na Figura 8.

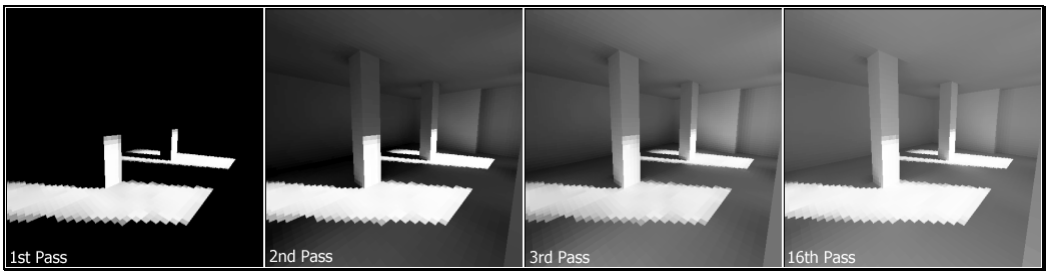

Figura 8 – Múltiplas interacções utilizado o método da radiosidade [Img03]

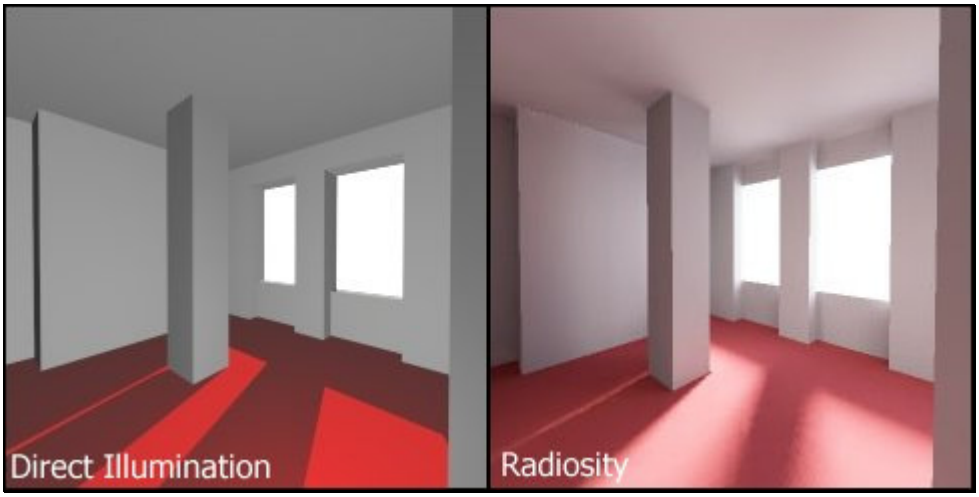

Figura 9 – Iluminação directa apenas vs. inclusão da radiosidade [Img03]

O algoritmo utiliza o método dos coeficientes finitos aplicado ao integral das contribuições, calculando a quantidade de energia transferida entre *patches* (segmentos). No primeiro passo de interacção apenas as superfícies auto iluminadas (fontes de luz) transmitem energia para os restantes segmentos considerados. Num segundo passo são consideradas as contribuições de

segmentos iluminados para outros segmentos e assim sucessivamente nos passos seguintes até atingir um valor ou condição de paragem predefinido pelo algoritmo.

É necessário considerar apenas o diferencial de transferência de energia entre passos, pois caso a energia transferida de um segmento para outro já tenha sido considerada num passo anterior, apenas o diferencial entre a interacção anterior e a actual deve ser calculado.

Considera-se que o fluxo de energia é efectuado em apenas um sentido, partindo do segmento que emite para o segmento que recebe, tendo em conta as características fotométricas difusas do material do receptor e o ângulo entre a referida direcção considerada e a normal no segmento receptor [4].

Em relação a decomposição deste algoritmo em elementos para a sua implementação num ambiente paralelo, pode-se constatar que existe um elevado grau de independência entre os múltiplos cálculos, principalmente ao efeito da propriedade comutativa da soma de valores. Como os restantes métodos apresentados, esta abordagem é facilmente paralelizável.

2.2.3 Photon Mapping

O processo de *photon mapping* é constituída por dois passos (Jensen 2001). No primeiro passo são "lançados fotões a partir das fontes de luz para o espaço 3D e sempre que estes encontram uma superfície de um objecto do cenário é tomada uma de três opções, sendo que a selecção é determinada através de um processo baseado no algoritmo de Monte Carlo:

- Fotão armazenado – O fotão é adicionado a uma estrutura de dados ligada a superfície, acumulando a sua intensidade com o valor previamente armazenado resultante de valores, fotões, previamente considerados. Esta estrutura serve como um acumulador de fotões ligada as superfícies dos objectos do cenário;

[4] Esta abordagem, relativamente à componente de iluminação global, é do tipo *forward tracing*.

- Fotão reflectido – Neste caso o fotão é reflectido utilizando o modelo fotométrico da superfície onde o fotão intersectou o objecto. O fotão, com as novas características devido a sua reflexão, é processado como seja de um fotão proveniente de uma fonte de luz primária, "lançado" para o meio. Existe um número máximo de *bounces*, semelhante ao referido no ponto 2.1.2, por forma a tornar o algoritmo finito, limitando o número de acções recursivas possíveis;
- Fotão retransmitido - O fotão é absorvido pela superfície, dando origem a um novo fotão que é criado devido ao processo de retransmissão do material considerando, através de um dos processos de retransmissão considerados, incluindo refracção (Dmitriev 2002; Günther 2004).

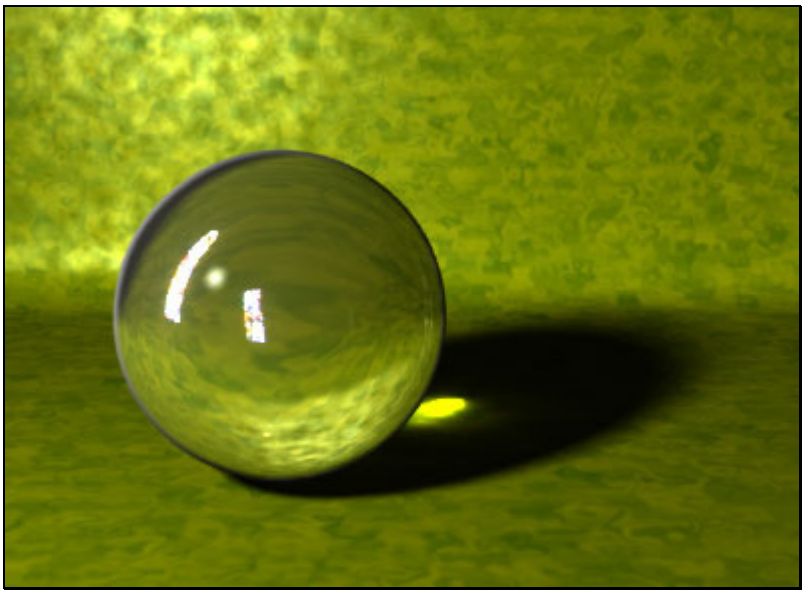

Figura 10 – Casuísticos utilizando *photon mapping* [Img04]

No segundo passo é feito a síntese de imagem recorrendo as contribuições directas e indirectas das fontes primárias e secundárias, mas considerando também as contribuições dos fotões acumulados nas superfícies como de fontes de luz primárias se tratassem. Esta técnica é frequentemente utilizada em situações

que se pretenda incluir os efeitos casuísticos presentes no cenário, como seja o caso da Figura 10.

Como se pode depreender pelo algoritmo apresentado, dada a independência de cálculos entre fotões e a propriedade associativa da soma, este algoritmo é de fácil adaptação a uma plataforma paralela de computação, no que respeita ao primeiro passo do algoritmo enunciado. O segundo passo também é paralelizável, pois trata-se de um algoritmo de *ray tracing* tradicional.

2.3 Computação paralela

A computação gráfica de elevado realismo é uma área que desenvolve algoritmos e métodos para sintetizar imagens de modelos artificiais de uma forma o mais realista possível. Este tipo de algoritmos são conhecidos pelo elevado grau de incerteza do acesso aos dados e elevada complexidade computacional. A síntese de uma imagem de elevada qualidade pode demorar horas, ou mesmo dias. O processamento paralelo (Duncan 1990; Chalmers 2002) oferece o potencial necessário para resolver problemas desta complexidade num intervalo aceitável (Chalmers 2002).

O significativo incremento em poder computacional nos sistemas informáticos e o leque de recursos disponíveis num sistema computacional moderno, providenciam uma plataforma para o desenvolvimento de *ray tracers* com adaptações específicas. A implementação de aplicações deste género tem visado vários tipos de hardware geralmente presente num sistema informático (Parker 1999; Chalmers 2002; Zemcik 2003; Buck 2004), incluindo elementos com um único processador (Wald 2001) ou múltiplos elementos de processamento homogéneos (Wald 2001; Chalmers 2002), recursos específicos como placas de processamento dedicado (Zemcik 2003) ou placas gráficas (Buck 2004). O desenho deste tipo de implementações é normalmente adaptado para o hardware específico ao nível da estruturação do algoritmo de *rendering/ray tracing* e das estruturas de dados associadas, por questões de eficiência. Esta abordagem habitualmente foca

apenas uma classe específica de hardware num sistema informático, embora o sistema possa conter os recursos com capacidades capazes de contribuir para o processo de síntese de imagem. A portabilidade da implementação para outras arquitecturas ou outros recursos não é tida em conta, pois não é um assunto fulcral pelo menos no estágio de desenho da implementação. Mais detalhes sobre estas abordagens específicas serão descritos no ponto 2.4, relativo a implementações em hardware específico.

Dado que a implementação de um *ray tracer* pode seguir uma abordagem com base no paradigma do paralelismo, já que o algoritmo genérico de *ray tracing* é facilmente paralelizável (Benthin 2002; Chalmers 2002), surge a necessidade de apresentar alguma terminologia dentro da área antes de prosseguir para implementações/abordagens específicas.

O primeiro aspecto a ter em conta a quando da transformação/adaptação de um algoritmo genérico expresso de uma forma sequêncial é a forma como são divididos os elementos constituintes, quer ao nível dos dados quer ao nível operações realizadas sobre os dados.

- Divisão espacial (domínio dos dados) – A divisão é efectuada ao nível dos dados, sendo possível aplicar a mesma operação ou algoritmo específico a um conjunto de dados similares. Um exemplo deste tipo de divisão, é a efectuada nos processadores vectoriais (Chalmers 2002; CIO 2006). É possível aplicar a mesma instrução a um vector de dados, como seja o caso da multiplicação de todos os valores por um valor escalar. Neste caso temos um *speedup* obtido teoricamente igual ao número de valores que podemos operar em simultâneo (Chalmers 2002);
- Divisão temporal (divisão algorítmica ou funcional) – É possível em determinadas situações realizar distintas tarefas sobre o mesmo dado inicial, ou decompor uma tarefa de grande dimensão num conjunto de tarefas de menor dimensão. O exemplo mais característico deste tipo

de abordagem é uma linha de montagem, em que existem vários itens em distintas fases de processamento ao longo da linha. O *speedup* obtido teoricamente é igual ao número de etapas que compõem a linha de montagem, supondo tempos iguais para cada etapa e sem tempos de atraso entre etapas. Esta abordagem é utilizada por exemplo para aumentar a eficiência dos processadores, *pipelining*, permitindo que diversas instruções sejam executadas em simultâneo, mas em que cada uma destas instruções está numa etapa distinta (Chalmers 2002).

Esta duas divisões são as mais generalistas, sendo possível as combinar e especializar de vários modos, obtendo abordagens mistas ou especializadas. Dentro desta variações incluem-se as seguintes (Chalmers 2002):

- *Data Driven* – O sistema é composto por elemento que divide o conjunto global de dados a serem operados pelos elementos de processamento (EP ou PE do inglês *Processing Element* (Chalmers 2002)). É possível efectuar uma divisão única dos dados primário e os distribuir pelos elementos de processamento (EP), no entanto caso a carga computacional exigida seja distinta ou os elementos de processamento seja distintos em termos de capacidade de processamento, surgirá uma assimetria no tempo de trabalho de cada para cada elemento de processamento/dados à processar. O tempo de finalização é o tempo do para que mais demorou, levando a que o tempo entre o término do trabalho realizado por cada elemento de processamento e o tempo do término do último elemento de processamento seja "desperdiçado" em espera. A gestão torna-se mais complexa caso seja necessário processar dados secundário (resultados de processamento de outros dados);
- *Demand Driven* – Esta abordagem é semelhante à anterior, excepto de em vez do gestor decidir de que forma é efectuada a distribuição dos

os dados, são os elementos de processamento que pedem ao gestor um bloco de dados quando estão inactivos. Sempre que é produzido um dado secundário susceptível de ser processado, este é enviado para o gestor que o coloca na fila de espera de processamento. Neste caso é mais fácil gerir dados secundários, bem como minimizar a assimetria entre o final de processamento dos diferentes elementos, dependendo do grau de divisão dos dados (granulosidade);

- *Fork and Join* – No campo do domínio funcional, a primeira abordagem é efectuar uma divisão das diversas tarefas principais e independentes em problemas de menor dimensão. Esta é uma abordagem de cima para baixo (*top down*), dividindo o algoritmo geral em etapas ou ramos independentes, atribuindo a cada EP um ou mais elementos desta subdivisão. A granularidade, nesta abordagem, é de blocos de dimensão considerável relativamente ao problema global. No final é necessário recolher os resultados de cada etapa ou ramo identificado e executado, por forma a obter a solução final pretendida. Em alguma medida esta divisão pode levar ao mesmo efeito indicado no caso *Data Driven*, pois como o tempo de processamento de cada elemento independente pode ser bastante díspar em relação aos dos outros elementos. É possível que diversos EP terminem antes do último EP, desperdiçando-se por este motivo a capacidade de cálculo disponível;
- *Data flow* – Também do campo funcional, o algoritmo deve ser decomposto nas suas operações mais elementares, definindo-se bem os caminhos que os dados devem percorrer para serem produzidos os resultados pretendidos. Dada a elevada divisão no domínio funcional é mais fácil de distribuir e balancear a carga distribuída pelos EP disponíveis.

Dos elementos expostos anteriormente, várias características e aspectos foram enunciados que determinam problemas e quantificadores ligados à

abordagem ou implementação de um algoritmo num ambiente paralelo. Seguidamente apresenta-se algumas características e aspectos relativos ao paralelismo de uma forma mais concreta e fundamentais na análise da implementação e eficiência da paralelização de um algoritmo.

- *Speedup* – Este valor é definido pela razão entre o tempo total da execução do algoritmo na sua forma sequêncial num só elemento de processamento e a execução paralela em múltiplos elementos, até a obtenção de todos os resultados pretendidos. Um valor de 100% indica que não foi obtido nenhum ganho ou perda de eficiência;
- Balanceamento – Aquando da divisão de tarefas por múltiplos elementos, o tempo de processamento de uma unidade de trabalho por cada elemento é dependente das suas características intrínsecas, essencialmente o poder de cálculo relativamente ao tipo de dados e operações efectuadas sobre os dados, bem como da dimensão do bloco de dados a processar. Sempre que seja necessário sincronizar todo os sistema, como no final da execução de todo o algoritmo, os elementos com processamento mais rápido e sem mais unidades de trabalho para processar ficam "parados" sem produzir trabalho efectivo até ao fim efectivo do algoritmo;
- Granularidade – Valor associado à dimensão relativa dos elementos devida à decomposição do algoritmo e dos dados, respectivamente no domínio funcional e espacial (Chalmers 2002; Boosten 2003). Um valor baixo indica que é mais fácil distribuir pelos elementos de processamento as unidades de trabalho mas que geralmente implica um aumento nas comunicações entre o controlador ou controladores e os elementos de processamento. Tendo em conta a relação entre os tempos dedicados às comunicações e o processamento efectivo das unidades de trabalho, bem como o modelo de comunicações definido, poderá existir uma adaptação da granularidade a cada situação em

pricinpalmente se for de baixa dimensão. É sempre possível aglomerar múltiplas unidades num bloco de maior dimensão e trata-las como de uma só unidade se trata-se;

- Rácio comunicações/execução – Sempre que um dado ou conjunto de dados, incluindo código do algoritmo, não esteja presente na memória directamente acedida pelo elemento de processamento, é necessário obter o referido dado do controlador ou de outro elemento de processamento. À comunicação de um pacote de dados existem dois tempos associados e distintos:
    - Tempo de estabelecimento ou atraso da comunicação (latência) (Tanenbaum 1988; Cater 2003);
    - Tempo necessário para a efectiva transferências dos dados (*data rate*).

  Existe ainda um terceiro factor que influência a eficiência da comunicação que é devido ao tipo de modelo de comunicações utilizado (Tanenbaum 1988; Chalmers 2002);

- Consistência e dependência de dados – A existência de múltiplas cópias dos dados primários e a necessidade de replicação de dados secundários pelos vários elementos de processamento requer uma forma de sincronismo do valor dos dados (Jansen 1998). A alteração de um dado por um elemento do sistema implica que todos os outros elementos do sistema sejam alertados dessa alteração antes de utilizar o dado em qualquer cálculo. Por vezes é necessário alertar outros elementos que o valor de um determinado dado partilhado vai ser alterado, antes da efectiva aliteração, prevenindo desta forma a utilização de um valor que será inválido num tempo próximo. Esta dependência de dados pode levar à paragem de alguns elementos de processamento devido a espera da validação de um dado necessário para o processamento. A reorganização de unidades de trabalhos e

flexibilização de qual o trabalho a ser realizado em cada elemento de processamento, ou mesmo o agrupamento de trabalho por mapeamento de dados independentes em grupos para o cálculo, pode ajudar nesta problemática (Chalmers 2002);

- Controlador do sistema – O modelo de distribuição escolhido pode variar do simples modelo cliente/servidor ou clientes/servidor, em que neste caso o servidor é o controlador, ou então um modelo em que todos os clientes também são controladores (Chalmers 2002), neste último caso a parte de gestão do sistema (controlador) encontra-se distribuída por todos os elementos. Ainda é possível optar por um modelo misto, com múltiplos controladores distribuídos ou múltiplos níveis com múltiplos clientes com gestão própria, agrupados em grupos e um ou mais controladores por grupo;
- Vizinhança – Entende-se como vizinhança a distância lógica de dependências das unidades de trabalho e/ou o caminho lógico necessário percorrer para enviar um dado de elemento de processamento para outro ou para o seu controlador. A subdivisão de um espaço ou de um conjunto de trabalho também efectuado por afinidade de dados, como seja o caso de dividir um espaço imagem de síntese em áreas contíguas. Quando é feita uma organização em grupos de elementos de processamento, deve ser tido em conta os dados que estes vão partilhar;
- Componente não paralelizável – Numa análise mais rigorosa da decomposição e adaptação de um algoritmo sequêncial é possível denotar a existência de uma componente paralelizável e outra não paralelizavel (Chalmers 2002). A componente não paralelizável, devido essencialmente a dependência entre dados, pode ser composta por vários segmentos de algoritmo não sequênciais. Esta componente pode implicar pontos de sincronismo intermédios entre todos os

elementos de processamento, apresentando o mesmo problema de balanceamento referido na finalização global do algoritmo.

A adaptação de um algoritmo sequêncial deve ter em conta os aspecto referidos, pois nem sempre a paralelização do algoritmo leva a um ganho, ou pelo menos um ganho significativo, devido por exemplo ao *overhead* nas comunicações entre elementos de processamento, em grande medida associado a granularidade da divisão de trabalho e necessidade de sincronismo de dados.

Ainda dentro do paralelismo podemos utilizar a nomenclatura definida por Flynn (Duncan 1990; Chalmers 2002), no que diz respeito à classificação da organização das operações e dos dados num modelo computacional de Von Newmann (Backus 1978; Chalmers 2002):

- SISD (*Single Instruction Single Data*) – É o formato mais simples e regularmente utilizado. Em cada instante de processamento é aplicada um só operação a apenas um só dado. Algoritmo sequêncial;
- SIMD (*Single Instruction Multiple Data*) – Neste caso é aplicado uma só operação a múltiplos dados homogéneos (do mesmo tipo). Uma das formas de paralelismo mais utilizado, como por exemplo nos processadores (Chalmers 2002; CIO 2006; IntelCorporation 2006) e placas gráficas (Nvidia 2008; Nvidia 2008);
- MISD (*Multiple Instruction Single Data*) – Situação em que é aplicado um conjunto de operações a apenas um dado, produzindo vários resultados simultaneamente. Forma menos utilizada no mundo real;
- MIMD (*Multiple Instruction Multiple Data*) – Conjunto das duas formas anteriores, aplicação de múltiplas operações com múltiplos dados;
- SPMD (*Single Program Multiple Data*) – Aplicação de um programa, conjunto de operações sequênciais, a um conjunto de dados.

É ainda de realçar que podem existir diferentes níveis de paralelismo utilizado, desde de instruções apresentadas ao CPU (Wald 2004) até a uma visão

global do sistema implementado (Koholka 1999). Mais pormenores sobre este níveis de paralelismo e abordagens utilizadas podem ser consultadas no ponto seguinte.

2.4 Abordagens específicas

Nos pontos anteriores desse capítulo foram abordados temas, termos e conceitos relativos à computação gráfica numa visão mais geal. O restante do capítulo é dedicado às abordagens e conceitos específicos, quer em termos de plataformas e hardware utilizado, que em termos de algoritmos ou adaptações específicas.

A síntese de imagem e animações de alta fidelidade tem sido abordada quer no domínio exclusivo da implementação no CPU quer em hardware dedicado específico. No campo da implementação exclusiva no CPU, a abordagem rege-se essencialmente pela adaptação e optimização do algoritmo de *rendering* e respectivas estruturas de dados referidas em pontos anterior à arquitectura onde a síntese da imagem o animação será realizada. Estas optimizações implicam a utilização de instruções específicas do CPU e de um grau de paralelismo elevado, ao nível da utilização dos diferentes elementos do CPU e ao nível do emparelhamento/reorganização de dados, que garantem uma maior eficiência na síntese da imagem, no sentido de realizar o mesmo conjunto de cálculos mas num mais curto espaço de tempo (Wald 2002). O tipo de hardware mais utilizado na implementação de um *renderer*, para além do referido CPU, é o GPU. Este apresenta uma configuração muito favorável a implementação de *ray tracers*, mesmo no que toca a implementação de *ray tracer* que inclui uma componente de iluminação global. Por fim, temos outros recursos não específicos da área da computação gráfica, que também são utilizados na implementação de um *ray tracer*, como seja ocaso de placas de processamento auxiliar. Os pontos seguintes apresentam trabalhos desenvolvidos nas três vertentes apresentadas: CPU; GPU; Placa adicionais de processamento.

2.4.1 Placas gráficas

A arquitectura estabelecida actualmente no que respeita ao hardware dedicado para o processamento gráfico, denominado como placa gráfica [5], é principalmente dirigida para o processo de rasterização de imagens partindo de um conjunto de geometria e texturas definidas para um cenário. No entanto, como se pode observar pelas características apresentadas nos últimos modelos de placas gráficas lançadas no mercado, como seja a inclusão de um conjunto de 256 "*stream processors*" (Buck 2004; Nvidia 2008), denota uma tendência para uma maior gama de aplicabilidade deste tipo de recursos em problemas fora do âmbito da computação gráfica, e mais propriamente fora do processo de rasterização.

Fora do âmbito dos problemas clássicos em que as placas gráficas podem ser utilizadas encontra-se um conjunto de algoritmos, também clássicos, como seja o caso da *Fast Fourier Transform* (FFT), do cálculo matricial e Álgebra Linear básica (Buck 2004). A implementação destes algoritmos em placas gráficas demonstraram que a sua eficiência rivaliza com as implementações standard no CPU numa grande variedade de problemas do foro do cálculo matemático. A principal vantagem das placas gráficas sobre os modelos genéricos de CPUs é a presença de um elevado número de unidades de vírgula flutuante e cálculo de inteiros, para além do também elevado nível de paralelismo, quer espacial quer temporal, que beneficia amplamente as operações sobre matrizes e valores escalares.

Embora as placas gráficas não estejam ainda focadas para implementação de algoritmos da área do *ray tracing*, várias são as primitivas partilhadas entre os algoritmos de rasterização e os algoritmos de *ray tracing*, fazendo com que este tipo de hardware se apresente atractivo para a implementação dos referidos algoritmos de *ray tracing*. O rápido desenvolvimento nas arquitecturas das placas gráficas, nomeadamente no poder de cálculo e na razão entre o poder de cálculo e o custo, torna ainda mais fácil e apetecível o desenvolvimento aplicações na área do *ray tracing*.

[5] *Graphics card* ou GPU - *Graphics Processing Unit* na nomenclatura inglesa.

Uma abordagem mais genérica com o objectivo de criar uma plataforma ou ambiente de desenvolvimento fácil foi desenvolvida por Buck et al. (Buck 2004), baseada no paradigma de *streaming* que é bastante adaptável à arquitectura seguida nas placas gráficas.

A definição de uma linguagem de *streaming* de uso generalista por parte de Buck et al. (Buck 2004) para o modelo de programação das placas gráficas, demonstra a versatilidade e o poder do hardware gráfico presente na grande maioria dos computadores pessoais de média e alta gama. A linguagem definida aproveita a arquitectura do tipo SIMD presente nas placas gráficas, os diferentes processadores de vértices e de fragmentos, a memória para texturas e registos internos, para implementar um ambiente de programação orientado ao paradigma de *streaming*.

As *streams* são matrizes de elementos de tipo de dados homogéneos que podem ser processadas em paralelo por módulos de código denominados de *kernels*. O resultado da aplicação dos *kernels* aos elementos de uma *stream* de entrada pode ser ligado a outros *kernels* para processamento posterior. Parâmetros de entrada simples podem ser colocados nos registos internos da placa gráfica, com acesso apenas de leitura. Toda a especificidade do hardware onde a aplicação é executada é ocultada pelo compilador/*linker* e bibliotecas da linguagem definida, restando ao utilizador apenas implementar os *kernels* com os algoritmos que pretende executar e indicar as respectivas entradas e saídas, *streams*.

Este tipo de abordagem tem uma boa adaptação a problemas com um comportamento do tipo SIMD, no entanto subsiste um problema de implementação que pode prejudicar a eficiência. Este problema é a falta de instruções de *branching* (salto do programa para outro bloco de código) nos processadores de vértices e de fragmentos das placas gráficas, que leva à execução completa de um fragmento de código, *kernel* neste caso, mesmo que não seja necessário. Esta situação impõe determinadas limitações na implementação de algoritmos que

dependam muito deste tipo de facilidade como seja o caso de percorrer uma octree.

Seguindo esta abordagem Purcell et. al (Purcell 2004) implementou um *ray tracer* fotorealista em tempo real. Segundo o autor a tecnologia VLSI (*Very-Large-Scale Integration*) atingiu um patamar capaz de colocar num só chip o poder computacional necessário para uma implementação de *ray tracer* em tempo real.

Ainda segundo o autor, a maioria dos *renderers* fotorealísticos de alta fidelidade geram imagens através de técnicas de iluminação global em cima da componente de *ray tracing*. O *ray tracing* em tempo real tem sido o objectivo procurado pela comunidade de computação gráfica desde de há muitos anos. Infelizmente é um processo bastante oneroso em termos de poder computacional.

Purcell (Purcell 2004) apresentada um modelo abstracto de um *stream processor* para as recentes placas gráficas (GPU), permitindo que o GPU execute programas neste modelo definido. Descreve-se a implementação do *ray tracer* segundo o modelo de *streaming* no GPU, providenciando uma análise sobre a largura de banda e requisitos computacionais da implementação. O *ray tracer* implementado é utilizado também para avaliar a simulação de GPUs com modelos de execução avançados.

A utilização de técnicas de *photon mapping* no modelo implementado também foi estudada e avaliada. Finalmente são feitas considerações relativamente a tendências relativas à utilização do modelo computacional de *streaming*. Os resultados apresentados mostram que um *ray tracer* baseado no GPU e seguindo uma abordagem de num modelo de *streaming* tem o potencial de rivalizar ou mesmo superar algoritmos implementados no CPU, sem fundamentar a necessidade de aquisição de novo hardware, diminuindo desta forma o fosso entre *rendering* realístico e interactivo.

Purcell (Purcell 2004) ainda refere que apenas em anos recentes os supercomputadores e agrupamentos (*clusters*) de PCs (Samanta 2000; Samanta

2000) atingiram um nível suficiente para demonstrar aplicações de algoritmos de *ray tracing* com taxas interactivas que incluem a componente de iluminação global.

Ocorreu recentemente um avanço tecnológico na área do hardware gráfico, em que os pipeline funcionais fixos foram substituídos por processadores programáveis de vértices e fragmentos. Num futuro próximo, é provável que o pipeline das placas gráficas evolua para um processador de *streaming* mais genérico, capaz de realizar operações mais complexas que o *rendering* de triângulos (Purcell 2002).

Purcell avaliou a tendência na programabilidade de tais pipelines gráficos e a possibilidade de implementar um *ray tracer* neste tipo de hardware. Utilizado o simulador definido, é analisada a eficiência de uma implementação de um algoritmo *ray tracing* na nova geração de placas gráficas programáveis. Também é feita a comparação entre o hardware que não permite o *branching* utilizando uma implementação de múltipla passagens e uma arquitectura que permite o *branching*. Purcell (Purcell 2002) ainda apresenta como é possível implementar outras algoritmos de *ray tracing*, como seja o caso do *whitted ray tracing*, *path tracing* e algoritmos de *rendering* híbridos, utilizando a abordagem proposta. Finalmente demonstra que o *ray tracing* em hardware gráfico poderá ser mais rápido em relação a implementações baseadas no CPU, bem como possa ser uma abordagem competitiva em relação ao modelo de *rendering* de triângulos.

Segundo Thompson et al. (Thompson 2002), recentemente as arquitecturas de hardware para gráficos tiveram a sua ênfase na versatilidade, oferecendo novas e avançadas formas de programar e reconfigurar o pipeline do processo gráfico. Também é estudado se é ou não possível utilizar as arquitecturas de hardware gráfico em problemas e algoritmos de índole genérico, tradicionalmente implementados em processadores vectoriais genéricos.

Thompson et al. apresenta uma plataforma de programação que é aplicada a uma grande variedade de problemas, incluindo a multiplicação de matrizes e 3-SAT. Comparando a eficiência de resultados na implementação em hardware

gráfico, com a implementação standard em CPU, é demonstrado que existe uma melhoria significativa do tempo de execução em muitos dos casos, bem como deixa espaço para futuros melhoramentos. São também analisados pontos onde se detectou constrangimentos no pipeline do hardware gráfico, sendo propostas ligeiras extensões as actuais arquitecturas por forma a contornar estes "*bottlenecks*", que podem levar a um aumento significativo do poder de cálculo na implementação de algoritmos genéricos. Com base nos resultados obtidos e as tendências na área ao nível da microarquitectura, acredita-se que o hardware gráfico será cada vez mais importante na elevada eficiência dos sistemas computacionais standard.

A visão apresentada neste ponto tem em conta apenas a implementação dos algoritmos de *ray tracing* exclusivamente em placas gráficas, considerando apenas o processamento necessário para produzir os resultados pretendidos. Foi desprezado a intervenção do resto do sistema, nomeadamente o CPU, na configuração, programação e comunicações entre componentes do sistema.

2.4.2 Placas auxiliares de processamento

Mais recentemente o aparecimento de algum hardware específico ou mesmo de uso generalista sobre a forma de placas de expansão para a plataforma PC, levou a implementação de algoritmos de *ray tracing* no referido hardware (Schmittler 2004). A rápida evolução da micro-electrónica, com já foi referido no ponto anterior, veio trazer para o mercado equipamento com maiores potencialidades e mais recursos, como seja o caso dos *Digital Signal Processors* (DSP) e *Field Programmable Gate Arrays* (FPGA).

Zemcik et al. (Zemcik 2003) implementou o pipeline para um *renderer* de partículas numa placa auxiliar de processamento que incluí um DSP e uma FPGA. O DSP é essencialmente responsável pela interface, gestão do sistema e programação da FPGA. A FPGA é reprogramada com o algoritmo pretendido, neste caso de *rendering*, que acede a uma DRAM local da placa para obter dados e

armazenar resultados. A principal vantagem na utilização da FPGA é a sua adaptação ao nível das instruções ao problema que se pretende solucionar, sendo fácil criar instruções específicas, como do tipo SIMD, para o *renderer* de partículas. A comunicação entre o sistema anfitrião (PC) e a placa é feita através de uma porta série ou de um porta Ethernet de 10Mbit. Uma nova versão desta arquitectura suporta uma ligação via *slot* de expansão PCI e contém múltiplos conjuntos de DSP e FPGA integrados no mesmo equipamento (Camea 2008).

A FPGA providencia uma forma simples e poderosa de implementar instruções específicas para um determinado problema com um impacto pouco significativo no pipeline do algoritmo pretendido. O paralelismo ao nível dos dados e o pipeline ao nível das instruções pode ser programado directamente na FPGA por forma a obter uma maior performance na especificidade do algoritmo implementado. No entanto a implementação só de operações isoladas sobre valores escalares, como inteiros e valores de vírgula flutuante, não é aconselhável pois existe hardware específico com melhor performance.

As FPGAs também foram utilizadas por Schmittler (Schmittler 2004) para a implementação de um *ray tracer* que já tinha sido implementado na plataforma PC por Wald (Wald 2004). Segundo Schmittler o *ray tracing* em tempo real estabeleceu-se com uma alternativa viável aos sistemas gráficos 3D baseados na rasterização. No entanto a implementação em software dos *ray tracers* é ainda limitada, quando implementada para os CPUs standard, requerendoneste caso ainda um número considerável de CPUs para obter taxas consideras de tempo real. Schmittler et al. (Schmittler 2004) propõe um protótipo com um pipeline completo de *ray tracing* numa única FPGA. Com um relógio de somente 90MHz, bastante inferior ao relógio dos CPU actuais na ordem do GHz, o protótipo é capa de atingir taxas de 20 à 60 *frames* por segundo numa vasta gama de cenários 3D, incluído suporte para texturas, múltiplas fontes de luz e múltiplos níveis de reflexão e transparência.

Uma característica interessante do desenho deste protótipo é a reutilização da unidade de transformação, relembre-se que o sistema está implementado numa matriz de portas lógicas (FPGA), necessária para o suporte de cenários dinâmicos de outras tarefas, incluído o cálculo de uma forma eficiente da intersecção de um triângulo por um raio bem como o cálculo de sombras [6].

2.4.3 Implementação em CPU com optimizações

Ao nível do CPU foram obtidos resultados que demonstram um significativo aumento da eficiência, menores tempos de *rendening*, mantendo os pipelines do CPU e unidades funcionais na máxima das capacidades. Esta "quase sempre máxima capacidade" é conseguida através da alteração da ordem sequêncial do algoritmo de *rendering* e adaptação das estruturas de dados utilizadas. Exemplos deste tipo de optimizações são o cuidado de alinhar a estrutura de dados à dimensão das linhas de *cache* do CPU e à utilização de paralelismo ao nível das instruções (Wald 2001; Wald 2003). Os mais recentes processadores incluem no seu *instruction set* instruções do tipo SIMD, como seja o caso das instruções SEE presentes nos processadores Pentium™ da Intel© e 3Dnow™ da AMD©.

Wald et al. (Wald 2003) implementou um *ray tracer* optimizado para arquitectura Pentium™, essencialmente ao nível da representação dos dados em estruturas mais simples e a utilização de instruções do tipo SIMD. A coerência de raios foi utilizada para compactar quatro intersecções com um triângulo, num mapeamento directo da pipeline do processador, tirando partido das instruções SEE (SIMD) presentes na arquitectura. Num nível hierárquico mais alto as estruturas de aceleração utilizadas foram adaptadas à configuração do 1º e 2º nível de *cache* do processador no que respeita ao alinhamento das linhas de *cache* e aos dados dos nós das estruturas de aceleração. O tempo de acesso de um nível de *cache* é sensivelmente na ordem de uma magnitude inferior para o próximo nível de memória (Wald 2003). Na componente de iluminação dos pontos da

[6] Mesmo não considerando o suporte adicional para cenários dinâmicos, o hardware proposto tem uma redução de custos 68% em relação a outras soluções analisadas.

imagem foi utilizada uma abordagem semelhante no que respeita ao teste de intersecção, no entanto devido à reorganização dos dados o ganho de eficiência não foi significativo na mesma ordem de grandeza.

O ganho de eficiência combinado do *ray tracer* implementado, comparando-o com os pacotes de software Rayshade (Rayshade 2008) and POV-ray (Pov-ray 2008), foi na ordem de uma ordem de magnitude. Comparando os resultados de uma rasterização com o *ray tracer* implementado, verificou-se que o *ray tracer* é mais eficiente em cenários com um número elevado de polígonos de representação, nomeadamente acima de um milhão de polígonos. Esta situação deve-se principalmente à natureza de crescimento logarítmico e linear na quantidade de cálculos relativamente ao número de polígonos de uma cena, no *ray tracing* e na rasterização respectivamente (Wald 2003).

Ainda segundo Wald (Wald 2004), um dos objectivos mais ambicionados na comunidade da computação gráfica é a geração de realismo em tempo real, isto é, sintetizar imagens com um aspecto realista com ferramentas dentro da definição de tempo real. Actualmente, virtualmente todas as abordagens no sentido do tempo real utilizam hardware específico, essencialmente baseado na abordagem da rasterização de triângulos. Infelizmente, embora esta tenha apresentado uma tremenda evolução nos últimos anos, para muitas aplicações. Esta tecnologia já chegou aos limites em termos de complexidade do cenário, efeitos suportados e realismo atingível.

Uma alternativa a rasterização de triângulos é o algoritmo de *ray tracing*, que é bem conhecido pela sua melhor escalabilidade, quer em termos da dimensão do modelo do cenário, quer no poder de cálculo requerido. No entanto, o *ray tracing* também é exigente em termos de cálculo computacional e por esse motivo é utilizado principalmente no *rendering* de elevada fidelidade em sistemas *off-line*. A tese de Wald (Wald 2004) foca a sua atenção na questão sobre se o *ray tracing* terá brevemente um maior papel em aplicações interactivas e como este objecto é atingido. Para esse fim, é discutido o sistema em tempo real de *ray tracing*

RTRT/OpenRT (Dietrich 2003; Wald 2003; Pomi and Slusallek 2004; Wald 2004; Wald 2005; Wald 2005), um software baseado num *ray tracer* em tempo real que permite obter *framerates* a taxas interactivas, em sistemas baseados nos CPU actuais. Em particular, é discutido o desenho global do sistema, a eficiência da implementação dos algoritmos fundamentais de *ray tracing*, técnicas para manusear de forma eficiente os modelos de cenários dinâmicos, uma eficiente paralelização da plataforma, e uma API de baixo nível semelhante ao OpenGL.

No seu conjunto, todas estas técnicas produzem um mecanismo completo de *rendering* em tempo real que suporta cenários bastante complexos, de elevada fiabilidade e de iluminação correcta, para além da simulação correcta do comportamento da luz, com taxas interactivas. Na parte final da sua tese são discutidas as implicações e o potencial de um sistema de *ray tracing* em tempo real com iluminação global e como a disponibilização desta tecnologia pode proporcionar o objecto enunciado: obter iluminação global de uma forma interactiva – a simulação correcta do comportamento da luz a taxas interactivas.

No que respeita às plataformas específicas Benthin et al. (Benthin 2006) refere que nas últimas três décadas o aumento performance no CPU foi obtida essencialmente através do aumento da frequência de relógio. Este aumento na frequência de relógio veio aumentar o consumo de energia e respectiva dissipação de calor, que de certa forma limita a continução deste tipo de abordagem, fazendo como que os responsáveis pelo design dos de CPU enveredar por outros caminhos alternativos para providenciar mais poder de cálculo. Em particular abordam três vias: o modelo computacional de *streaming*, unidades com instruções do tipo SIMD aplicadas a elementos semelhantes a vectores, e arquitecturas *multicore*.

Um exemplo particular deste tipo de arquitectura multicore é o *Cell Broadband Engine Architecture* (CBEA), que oferece um poder de cálculo na ordem dos 200 GFlops num circuito integrado a funcionar a velocidade de relógio de 3,2 GHz. O Cell Processor oferece um elevado potencial para aplicações de cálculo intensivo como seja o caso do *ray tracing*, mas também apresenta novos desafios

devido a sua arquitectura não convencional. Benthin et al. descreve a implementação de *ray tracer* em tempo real (Benthin 2006) no Cell Processor (Hofstee 2005; Williams 2006), utilizando uma combinação de rotinas nucleares optimizadas de baixo nível, um modelo de *streaming*, *cache* explicito e uma abordagem em software em para emular virtualmente o Hyper-Threading™ (IntelCorporation 2003), minimizando os efeitos de latência do acesso via DMA, foi possível atingir uma performance superior de uma ordem de magnitude em relação a uma implementação num CPU standard.

Schmittler et al. (Schmittler 2002) refer que o algoritmo de *ray tracing* é bem conhecido pela sua habilidade de gerar imagens de alta qualidade e a sua flexibilidade para incorporar efeitos de iluminação avançados. O *ray tracing* interactivo foi amplamente demonstrado em *clusters* de computadores e super computadores, mas em hardware que suporta directamente o *ray tracing* é difícil de implementar.

Os autores apresentam uma nova arquitectura para o *ray tracing* em tempo real, escalável, modular e bastante eficiente. Esta arquitectura atinge uma elevada eficiência com baixos requisitos de largura de banda em termos de memória, juntando os raios de uma forma eficiente.

A arquitectura é facilmente configurável para suportar uma variedade de cargas. Para modelos de cenários do género OpenGL ela oferece uma performance semelhante às soluções mais avançadas no domínio da rasterização. A arquitectura também suporta todas as características do *ray tracing*, incluído sombras exactas, reflexões e refracções, bem como é capaz de processar modelos como milhões de triângulos. A arquitectura e a sua eficiência foi analisada em diferentes configurações, baseado em simulações precisas e cíclicas.

Outra arquitectura para o *ray tracing* de plataforma é apresentada por Dietrich et al. (Dietrich 2003), o OpenRT. Segundo os autores há mais de uma década, grafismos interactivos foram modelados através de triângulos e sua respectiva rasterização, correspondente à API do OpenGL (SGI 2008). Segundo

Dietrich et al., o *ray tracing* interactivo tem vindo a apresentar-se como uma realidade disponível em diferentes tipos de plataformas, devido a sua superior escalabilidade, usabilidade e eficiência, é francamente provável que o *ray tracing* se apresente como uma alternativa viável no mundo das aplicações de gráficos por computador. Sendo desejável que este tipo de tecnologia suporte num padrão com a sua bem conhecida API, como a quase standard interface OpenGL, será difícil e complicado implementar uma interface semelhante, pois o standard OpenGL está fortemente associado como o processo de rasterização, tornando-o pouco adequado para o *ray tracing*.

É proposto uma nova API, designada de OpenRT. Esta nova API foi desenhada para ser similar ao OpenGL quanto possível, enfatizando no entanto nas características do *ray tracing* interactivo.

Sendo do mesmo nível de simplicidade de utilização e aprendizagem do OpenGL, o OpenRT oferece todas as vantagens do *ray tracing*, como a visibilidade implícita, instanciação, a liberdade de lançar raios arbitrários e programar o modelo de iluminação.

Numa versão mais prática Wald et al. (Wald 2003) propõe uma implementação de um *ray tracer* numa rede de computadores homogéneos. Devido ao seu significativo e elevado nível de paralelismo, o *ray tracing* sempre foi uma plataforma atractiva para investigadores na área do processamento paralelo. Com os recentes avanços no domínio do software e do hardware, é agora possível criar imagens sintetizadas de elevada qualidade a taxas interactivas em *clusters* de PCs. Wald et al. apresenta o estado da arte do *ray tracing* distribuído e interactivo baseado na descrição de aspectos distribuídos do OpenRT (Dietrich 2003), um modelo para um motor de *rendering* interactivo. É demonstrado a sua escalabilidade, e são apresentados vários exemplos de aplicação.

Informações sobre os aspectos anteriormente referidos e outros podem ser encontrados no *STAR – State of The Art Report* de Wald et al. (Wald 2003) na conferência da *Euruographics 2003*.

2.4.4 O projecto RoD

O trabalho desenvolvido desta tese em parte realizado num projecto designado de "*Rendering on Demand*" (RoD) (RoD 2006), cujo o principal objectivo é produzir animações de alta fidelidade em tempo real.

Nesse sentido foram definidas três vertentes com o intuito de melhorar os tempos de execução, reduzindo o tempo global de *rendering* de uma animação. As três vertentes definidas são:

- Percepção – Evitar cálculos desnecessários em zonas da imagem onde os utilizadores não distinguem diferenças significativas entre uma síntese de baixa qualidade e uma síntese de qualidade completa (Dumont 2002; Cater 2003; Sundstedt 2004);
- Processamento paralelo ao nível de múltiplos sistemas informáticos – Providenciar a colaboração efectiva de múltiplos nós computacionais para um resultado comum numa rede de computadores, com uma gestão eficiente de tarefas e dados relativos ao trabalho em questão dentro de um conjunto de recursos disponíveis (Chalmers 2002; Wald 2002; Wald 2003; Debattista 2005; Benthin 2006);
- Processamento paralelo ao nível dos recursos de computador – Utilizar de uma forma eficiente os recursos disponíveis num sistema informático, maximizando a contribuição de cada um e de todos os recursos, para o objectivo comum da síntese de imagem.

O trabalho desenvolvido como suporte da presente tese, teve com incidência o paralelismo ao nível dos diferentes recursos presentes num sistema informático, tendo no entanto em conta sempre a visão global de obter uma aplicação que preserve uma elevada qualidade em tempo real.

Nesse sentido foram introduzidos restrições no modelo definido, ao nível da execução em tempo real e prioridade entre as tarefas executadas.

Uma melhor visão das dependências e interface entre as diferentes camadas/agrupamentos de ferramentas do projecto RoD é apresentada na Figura 11 – Visão geral da interacção no projecto RoD.

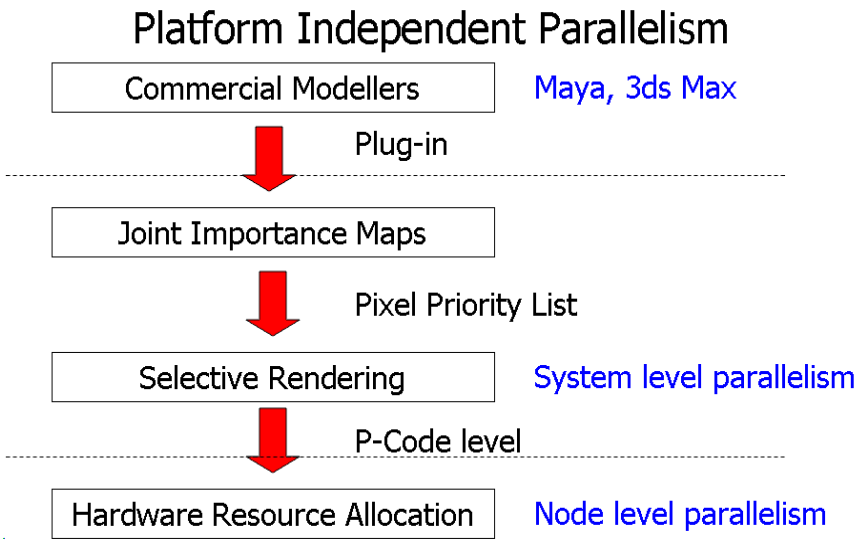

Figura 11 – Visão geral da interacção no projecto RoD

2.4.4.1 Abstracção, suporte e integração

Uma característica com muita ênfase neste projecto é referente à independência da implementação dos sistema em relação aos recursos e equipamentos utilizados actualmente existentes. Como será possível verificar, mais propriamente nos pontos seguintes deste capítulo, outras abordagens tendem maximizar as características específicas de determinados tipos de recursos. O projecto RoD pretende ser mais independente das tendências actuais ou mais "tentadoras" do momento, sem deixar de as abordar.

Nesse contexto existe uma tentativa de uma abstracção mais distante de quaisquer tipo de tecnologia ou abordagem específica, especificando apenas grandes blocos do processo de síntese de imagem definido uma interface standard entres os respectivos blocos.

2.4.4.2 Percepção e prioridade do *rendering* de *pixeis*

Com forma de reduzir a quantidade de cálculos necessários para obter um resultado considerado cognitivamente semelhante a solução final de *rendering* completa de uma imagem, foram efectuados estudos por parte de elementos do grupo com seja o caso de Sundstedt et al. (Sundstedt 2004), que por sua vez utilizar modelos e técnicas definidas por Cater et al. (Cater 2003), Myszkowski et al. (Myszkowski 1998), Yee et al. (Yee 2001) e Adelson et al. (Adelson 2000).

Dada a menor relevância destas questões para a presente tese, sugere-se que o leitor mais curioso sobre estes temas, consulte os documentos referidos.

3 MODELO PROPOSTO

3.1 Introdução

No presente capítulo é proposto um modelo de alocação dinâmica de recursos num sistema de *rendering*, suportado em múltiplos e distintos equipamentos, com um objectivo final comum: A síntese de uma imagem ou sequências de imagens através de um algoritmo de *rendering* fotorealista, utilizando todos os recursos disponíveis num sistema computacional e capazes de contribuir de uma forma colaborativa e substancial para o resultado final.

O modelo proposto baseia-se sobretudo na organização do fluxo de trabalho proveniente do *renderer*, entenda-se o ciclo principal de *rendering,* e a utilização dos recursos existentes, pelo que, quer a sua especificação quer a sua descrição reflectem esse ponto de vista. Isto significa que as matérias intrínsecas à síntese de imagens ou animações e ao processo de *rendering*, não relevantes para a descrição do referido modelo, não serão abordadas. Em grande parte os assuntos relacionados com o processo de *rendering* propriamente dito já foram abordados com algum detalhe no capítulo 2.

Na primeira parte do capítulo é efectuada uma descrição genérica do modelo proposto e o seu enquadramento no processo de *rendering*. Na segunda parte é apresentada uma especificação formal do modelo de reserva e gestão de recursos proposto nesta tese e designado por HRA, acrónimo da designação "*Hardware Resource Allocation model*". Com este intuito, são apresentados diversos diagramas do modelo que o especificam seguindo a metodologia UML - *Unified Modeling Language* (Arlow 2005). Por último, é efectuada uma descrição mais técnica do modelo. Esta descrição inicia-se com a apresentação de alguns exemplos de utilização, os quais pretendem complementar a compreensão do funcionamento do modelo, terminando o capítulo com a discussão de alguns aspectos próprios de ambientes heterogéneos.

3.2 Definição do modelo proposto

O modelo proposto na presente tese pode ser observado segundo um conjunto de perspectivas, mas sempre com um objectivo final, o de providenciar uma estrutura para uma plataforma de *rendering* com um grau de independência elevado relativamente ao hardware disponível instalando no nó de computação considerado. Um nó é um sistema computacional individual com múltiplos recursos de hardware com capacidade de colaborar no algoritmo de *rendering* de uma forma significativa.

Pode-se observar o modelo segundo uma perspectiva organizacional, que define quais as componentes que compõem o modelo, que interacção é permitida entre as diferentes componentes e suas respectivas funcionalidades e objectos. Este modelo organizacional é designado de "*Hardware Resources Allocator*" (HRA), que providência uma abstração do hardware presente no nó em relação ao corpo principal do algoritmo de *rendering*.

Do ponto de vista conceptual também foi reorganizado algoritmo de *rendering*, através da classificação das distintas componentes fundamentais em

termos de recursividade, dimensão (código necessário na sua implementação) e respectiva complexidade.

Estas duas perspectivas foram consideradas fundamentais, no entanto através destas, outras perspectivas podem ser consideradas, com base por exemplo em paradigmas de programação do hardware subjacente, ou interacção e organização dos dados e elementos funcionais dentro de cada recurso de hardware considerado. No entanto, não seria incorrecto considerar estas perspectivas como fundamentais na abordagem conceptual, não sendo no entanto esta a abordagem seguida.

Como foi referido na parte final do capítulo 2, as abordagens anteriores têm privilegiado soluções específicas e direccionadas para arquitecturas de hardware em particular (Schmittler 2002; Wald 2003; Purcell 2004; Schmittler 2004; Benthin 2006). O baixo nível de abstracção/independência relativamente à arquitectura implica alguma limitação na portabilidade das referidas soluções, embora seja possível adaptar parte ou partes das soluções implementadas para outras plataformas/hardware (Schmittler 2002; Wald 2005; Benthin 2006). Estas abordagens seguem uma optimização segundo a perspectiva de eficiência (rapidez de obtenção dos resultados) do algoritmo de *rendering*

Na implementação de uma solução para um problema, o binómio portabilidade/eficiência deve ser considerado bem como as respectivas consequências. Em soluções com uma perspectiva de curto prazo e com o objectivo de serem rápidas, quer ao nível da performance quer ao nível do tempo da sua implementação, a balança tende para a eficiência, resultando normalmente num prejuízo no que diz respeito à portabilidade. Para soluções de média e longa duração o peso da portabilidade ganha significativa relevância. Também deve ser considerado que nem sempre a eficiência e a potabilidade são factores totalmente complementares, dependendo do tipo de problema e da abordagem seguida. No caso da presente abordagem, foi considerado um sistema de média ou longa

duração em plataformas heterogenias, e por consequência dada ênfase predominante ao factor portabilidade.

3.2.1 Definição de termos e conceitos do modelo

Dento do modelo desenvolvido foram considerados distintos aspectos, quer de funcionamento quer de organização. Os pontos seguintes pretendem definir de uma forma clara estes aspectos e apresentar alguns conceitos.

3.2.1.1 Primitiva

É considerado como primitiva, no sistema definido, o fragmento de código/algoritmo que através da sua execução é capaz de produzir resultados com base num conjunto de dados de entrada, vazio ou não. Como exemplo simples pode ser referida a primitiva que gera números aleatórios no sistema, que neste caso não contém parâmetros (dados) de entrada.

3.2.1.2 Unidades de trabalho

No contexto do modelo apresentado, uma unidade de trabalho (UT) é composta pelos dados e pela primitiva que se pretende aplicar aos dados de entrada de forma a produzir resultados. Os dados resultantes são considerados como parte integrante da primitiva. Para uma unidade de trabalho ser distinta basta que os dados utilizados ou a primitiva sejam distintos.

Em termos práticos a unidade de trabalho pode conter a informação ligeiramente distinta da aqui referida, por questões de optimização/eficiência, mas de uma forma geral a representação de uma unidade de trabalho no sistema tem uma associação directa ou indirecta com as representações dos elementos que a compõem, dados e primitiva.

3.2.1.3 Contentores de unidades de trabalho e de resultados

Os contentores de unidades de trabalho e de resultados, são estruturas de dados capazes de conter e gerir de uma forma simples as unidades de trabalho geradas pelo sistema e apresentadas ao elemento que gere a execução dos trabalhos, o HRA. O contentor de unidades de trabalho também serve como um mecanismo de contenção na produção de unidades de trabalho, pois numa implementação a sua dimensão é finita o que leva a que quando o contentor se encontre no estado cheio não seja possível que o ciclo principal de *rendering* produza unidades de trabalho enquanto não forem removidos um número suficiente de unidades do contentor.

Da parte do contentor de resultados não existe qualquer tipo de limitação, pois não deve existir contenção nos resultados produzidos. É responsbilidade do ciclo principal de *rendering* remover os resultados do contentor de resultados. Para facilitar a remoção de resultados do contentor de resultados, o contentor de unidades de trabalho deve providenciar uma funcionalidade para detectar se este último pode aceitar um número pré-definido de unidades de trabalho ou não. Esta facilidade pode efectuada por exemplo através de uma chamada não bloqueante na colocação de unidades de trabalho no contentor. Desta forma, o ciclo principal de *rendering* pode tentar obter resultados do contentor de resultados caso não consiga submeter novas unidades de trabalho, evitando *deadlocks* (Tanenbaum 2001).

3.2.1.4 Ciclo principal de *rendering*

O elemento responsável pela gestão global do processo de *rendering* é denominado no modelo proposto de ciclo principal de *rendering*. Este é responsável por definir e produzir as unidades de trabalho que são necessárias para obter o resultado pretendido tendo em conta os parâmetros gerais definidos no processo de *rendering*, como o cenário, o número máximo de *bounces*, o posicionamento do ponto de vista e suas características, entre outros.

3.2.2 Descrição global do HRA

O *Hardware Resource Allocation* , designado por HRA, foi desenvolvido com o intuito de providenciar uma interface abstracta entre o ciclo principal de *rendering* de alto nível e o hardware disponível, como pode ser observado na Figura 12 – HRA *dataflow*. O tipo de hardware considerado no desenvolvimento do HRA inclui elementos standards presentes em qualquer configuração de um computador pessoal, *PC - Personal Computer,* nomeadamente o CPU e uma placa gráfica, mas não limitado a estes.

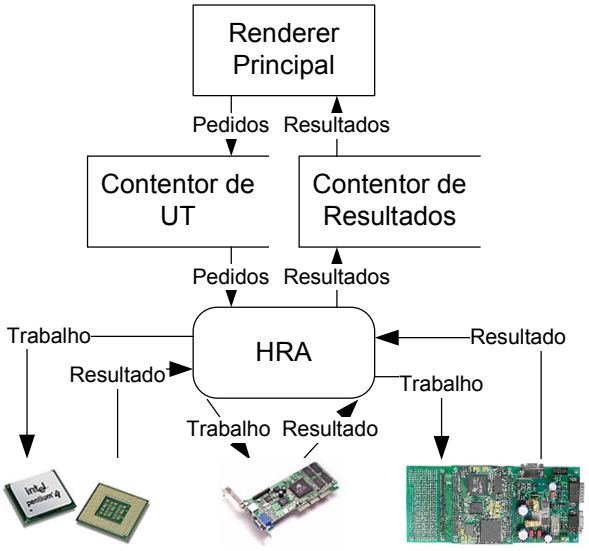

Figura 12 – HRA *dataflow*

O modelo proposto prevê que as necessidades computacionais decorrentes da execução do algoritmo de *ray tracing* sejam expressas na forma de unidades de trabalho, que são colocadas num contentor (Contentor de UT). Estas unidades de trabalho, mais propriamente a componente da primitiva da unidade de trabalho, são classificadas em 3 grupos: i) primitivas básicas; ii) primitivas complexas não recursivas; iii) primitivas complexas recursivas.

De acordo com o definido no modelo, as unidades de trabalho são retiradas do contentor e de forma "ordenada", "entregues" aos recursos de hardware para

processamento, sendo os resultados deste processamento colocados num outro contentor denominado de contentor de resultados.

Para além das habituais optimizações ao nível do código e dos dados nas primitivas de *rendering* de baixo nível, o HRA deve manter os recursos em funcionamento durante a maior parte do tempo de *rendering*, com o intuito de maximizar o trabalho e a eficiência do sistema numa visão mais global.

O HRA também tem conhecimento do estado de cada recurso, mais precisamente que unidades de trabalho, dados e primitivas implementadas que estão presentes na memória local de cada recurso. A interface entre o ciclo principal do *renderer* e o HRA é providenciada através de dois contentores. Um para a colocação das unidades de trabalho e outro para a colecta dos respectivos resultados. Embora o segundo contentor seja referido como contentor dos resultados, na realidade ele contém unidades de trabalho, pois estas, quando são criadas contêm espaço para armazenar os resultados dos cálculos específicos efectuados para a unidade de trabalho em questão.

O HRA obtém as unidades de trabalho do contentor de entrada e processa-os tendo em conta o tipo de unidade de trabalho, o estado e reserva actual dos recursos. Esta unidade ou conjunto de unidades de trabalho é enviado para o recurso num ou mais pacotes de dados, tendo em conta a estratégia de comunicação definida para o recurso em questão. Depois de finalizado o processamento da unidade, ou unidades de trabalho, o recurso devolve o resultado ao HRA, que por sua vez o coloca no contentor de resultados de onde o ciclo principal de *rendering* (*renderer*) o deve retirar.

Cada recurso contém uma parte do processo global de *rendering* e dados parciais necessários ao processamento das unidades de trabalho que lhe foram submetidas. Toda a gestão global do processo e estado dos recursos está presente no HRA.

Para maximizar o poder de cálculo providenciado pelos recursos presentes no sistema é de extrema importância garantir que o trabalho é dividido de uma forma

uniforme pelos recursos disponíveis (Chalmers 2002; Plachetka 2002). Considerando um ambiente heterogéneo relativamente aos recursos disponíveis, vários aspectos devem ser considerados, como a representação dos dados em cada tipo de recurso, a dimensão dos mesmos, o modelo de comunicação utilizado entre o recurso e o HRA, bem como o paradigma de programação específico utilizado pelo recurso.

O propósito global do HRA é o de utilizar todos os recursos disponíveis com um objectivo comum: conseguir uma performance elevada, minimizando, dentro de uma margem aceitável, o *overhead* criado pela necessidade de conversão, gestão e comunicação dos dados.

3.2.3 Classificação das primitivas

Segundo o ponto de vista conceptual foram classificadas as distintas componentes fundamentais do algoritmo de *rendering*, com base em três características: iteração da componente (primitiva); dimensão da sua implementação em distintos tipos de hardware, tendo em conta o respectivo paradigma de programação; complexidade do algoritmo e dos dados fornecidos/produzidos pela execução da primitiva.

Nesta análise foi considerado agrupar as diferentes primitivas nos seguintes três grupos:

- Primitivas de baixo nível – São primitivas que requerem apenas possíveis parâmetros de entrada e produzem resultados de complexidade baixa. A sua dimensão é reduzida, na ordem das dezenas de linha de código, com um *footprint* normalmente inferior a 1 kilobyte em termos de código executável no recurso considerado. As primitivas incluídas neste grupo não recorrem a outras primitivas de *rendering* na sua implementação;
- Primitivas de alto nível não recursivas – fazem parte deste grupo as primitivas que recorrem ao código das primitivas de baixo nível na sua

implementação ou processam dados de complexidade superior. O algoritmo das primitivas deste grupo não pode conter elementos susceptíveis de gerar iteração nos cálculos ou produzir recursividade no cálculo, a qual é difícil de controlar em termos de tempo de execução ou na necessidade de obtenção de dados suplementares para a sua finalização;

- Primitivas de alto nível com recursividade- Neste grupo foram consideradas todas as primitivas que não se podem classificar nos outros dois grupos, nomeadamente as primitivas de gestão do processo geral de *rendering* ou que produzem novos dados a serem processados, ou ainda as que necessitam de dados suplementares durante a sua execução.

3.2.4 Trabalho requerido num "*rendering*"

A equação apresentada na Figura 13 tem o intuito de representar o trabalho que é necessário realizar na síntese de uma imagem ou animação em função de um conjunto de unidades de trabalhos e suas respectivas frequências. No limite o tempo necessário para realizar o trabalho é igual ao somatório do tempo individual de cada tipo de unidade de trabalho e a sua respectiva frequência, considerando que o tempo de unidades de trabalho do mesmo tipo é igual ou é média dos tempos. No entanto cada unidade pode ser processada pelo sistema considerado dentro do espaço temporal de outra ou outras unidades de trabalhos, desta forma o tempo que decorre desde do inicio do trabalho até a sua conclusão é no máximo igual ao somatório dos tempos parciais.

$$T = \sum_{i=1}^{n} F_i P_i$$

Figura 13 – Trabalho realizado em função do tipo de unidades e sua frequência

Na equação da figura anterior o termo *T* representa o tempo total consumindo no processo de *rendering*, F_i a frequência da primitiva do tipo *i* (número de vezes que foi invocada) e U_i o tempo médio da execução do referido tipo de primitiva. O intervalo de tempo entre o inicio e o fim do processamento pode ser inferior ao termo *T* quando as unidades podem ser processadas em paralelo (dentro do intervalo de processamento de outra unidade).

Um sistema, nó de processamento, é composto por um conjunto de recursos com capacidade para realizar o trabalho requerido pelo conjunto de unidades de trabalho que compõem o trabalho global. Capacidade individual de cada tipo recurso pode não conter os requisitos suficientes para processar todos os tipos de unidades de trabalho, no entanto somatório da capacidades presentes no sistema deve incluir todos os requisitos necessários.

A Figura 14 é uma representação da composição de um sistema em função do tipo de recursos nele presente e sua respectiva multiplicidade. A equação é apenas uma representação e não uma equação linear num determinado domínio quantitativo.

$$S = \sum_{j=1}^{m} M_i R_i$$

Figura 14 – Sistema em função do tipo recursos e sua multiplicidade

Nesta equação o termo M_i representa a multiplicidade do recurso do tipo R_i e o termo *S* representa a capacidade do sistema em termos de números de recursos disponíveis para processamento.

Já a equação da figura seguinte representa a correlação de eficiência (E_{ij}) de um recurso do tipo *j* e unidades de trabalho do tipo *i*. Este pode ser atribuído ou calculado através de experimentação num implementação.

$$E_{ij} = U_i \otimes R_j$$

Figura 15 – Correlação de eficiência entre um tipo de unidade de trabalho e um tipo de recurso

Os valores de E_{ij} podem ser utilizados no algoritmo de atribuição de unidades de trabalho aos recursos existente no sistema, por forma a maximizar a eficiência geral do sistema.

3.2.5 Comparação com outras abordagens e novas tecnologias

Tendo em conta os elementos focados nos pontos anteriores, as abordagens focadas no Capítulo 2 e outras semelhantes (David Coulthurst 2008), seguem a da perspectiva da execução em paralelo, intervalo de tempo do processo de rendering inferior ao somatório dos tempos parciais das unidades de trabalho, ou optimização do código e dados aos recursos disponíveis.

No entanto, o carácter global deste tipo de abordagens impede a sua transposição fácil para outras plataformas ou recursos específicos.

O modelo proposto não implica o descartar total das técnicas desenvolvidas por outras abordagens, no entanto devido ao objectivo global de obter a melhor eficiência possível com o tipo de recursos visados, existe uma grande dependência entre todas as componentes da implementação do renderer e o formato dos dados utilizados.

As optimizações sugeridas por Wald et al. (Wald 2003) pode ser utilizadas nas primitivas de baixo nível de forma a melhorar o coeficiente de correlação de uma primitiva com o tipo de recursos. Na realidade grande parte das abordagens utilizam este coeficiente de correlação como elemento fundamental na eficiência obtida (Wald 2003; David Coulthurst 2008).

Surgem no mercado novas propostas de soluções de convergência transparente entre o domínio do CPU de GPU por exemplo. Dentro destas novas abordagens e plataformas apresenta-se o OpenCL (Saha 2009) e a plataforma

CUDA (Halfhill 2008). Ambas as tecnologias/plataformas pretende providenciar ao programador um ambiente comum de programação e uma adaptação automática a um ambiente heterogéneo de recursos (entenda-se GPU e CPU actualmente).

3.3 Especificação

A especificação do modelo proposto é apresentada nos pontos seguintes e pretende definir de uma forma simples mas suficientemente clara os requisitos funcionais do modelo, através de diagramas de casos-de-uso, o seu comportamento dinâmico, recorrendo a diagramas de estados, e a estrutura do modelo, com diagramas de classes.

Os diversos diagramas são complementados com um texto descritivo sucinto. O objectivo do texto complementar é de elucidar sobre alguns detalhes que por ventura não se apresente de uma forma óbvia no respectivo diagrama ou que possam suscitar dúvidas para um leitor menos familiarizado com a metodologia de especificação UML.

O conjunto de diagramas aborda o quê, como e quem interage com o sistema, bem como o que o modelo se propõe realizar. No entanto, antes de focar qualquer um destes tópicos será importante definir alguns requisitos pretendidos e relacionados com o problema em questão:

- O sistema deve ser capaz de interagir e utilizar um leque alargado de recursos de hardware heterogéneos, que estão presentes num sistema informático designado de "standard" (i.e. PC);
- As funcionalidades providenciadas e implementadas em cada componente devem permitir que num caso extremo o processo de *rendering* seja totalmente realizado no referido componente. No outro extremo desta visão, a implementação pode ser apenas parcial, incluindo para isso um conjunto reduzido de funcionalidades;

- Tendo em conta o requisito anterior, o sistema como um todo deve providenciar todos os tipos de funcionalidades requeridas para o *rendering*. A totalidade das primitivas de *rendering* deve estar presente no conjunto total dos recursos, mesmo que cada recurso individual apenas contemple um subconjunto das primitivas;
- O modelo de programação, as especificações e características de cada recurso devem ser mantidos independentes do funcionamento do sistema em geral. Estas características individuais não podem limitar a funcionalidade ou implementação pretendida para o *renderer* ou funcionamento do próprio HRA;
- O tempo para a realização das tarefas a vários níveis deve ser controlável ou conhecido, permitindo desta forma um controlo sobre o processo global de *rendering*. Este processo global, a produção de uma imagem ou animações, pode, num caso limite, necessitar de uma resposta em tempo real;
- Novos recursos de hardware que surjam no mercado devem ser fáceis de adicionar e incluir no sistema definido, requerendo apenas conhecimento específico sobre o recurso de hardware e a sua interface com o sistema;
- O resultado obtido com base nas mesmas premissas, o mesmo cenário e as mesmas características pretendidas, entre diferentes sistemas deve ser semelhante, excluindo pequenos pormenores como seja o caso de cálculos que utilizam valores pseudo-aleatórios intrínsecos ao processo de *rendering*;
- São de excluir situações fisicamente impossíveis de realizar, como seja o caso de exigir um poder de cálculo muito superior ao que o recursos de hardware pode providenciar;

- Optimizar o processo de *rendering*, num perspectiva temporal, mas sem comprometer as premissas definidas nomeadamente em termos de produção de resultados finais. Por exemplo, pode ser alterada a sequência dos cálculos ou a inclusão de parâmetros que alterem o número de cálculos requeridos de uma forma dinâmica, desde que o resultado pretendido, a produção de uma imagem num período definido de tempo, seja mantido.

3.3.1 Casos-de-Uso

Os diagramas de casos-de-uso apresentados para o modelo proposto pretendem definir os casos relevantes no sistema definido. Não é uma especificação cabal e exaustiva todo o modelo, mas sim o que se considera essencial ou exemplificativo do ponto de vista de uma possível implementação do modelo, ou mais propriamente da funcionalidade que o HRA pretende implementar.

3.3.1.1 Submissão de uma Unidade de Trabalho

Como já foi referido anteriormente em diversos pontos da presente tese, o processo completo de *rendering* pode ser decomposto em elementos de menor dimensão e de alguma forma independentes entre distintos elementos. Estes elementos ou unidades de trabalho (abreviadas com UT em trechos do texto subsequente), podem incluir, por exemplo, a intersecção de um raio com uma superfície ou a coloração de um ponto da superfície de um objecto do cenário tendo em conta a iluminação de uma fonte de luz e características fotométricas do material iluminado e da fonte de luz.

A definição das unidades de trabalho é abordada de uma forma mais pormenorizada na parte final deste capítulo, no entanto, por agora podemos considerá-las como instâncias de primitivas de *rendering*.

Neste primeiro caso-de-uso, Figura 16, o *renderer*, entenda-se o ciclo principal de *rendering,* submete uma ou mais unidades de trabalho ao HRA para que estas sejam processadas (cálculo de resultados). Este caso-de-uso incluí outros casos-de-uso, nomeadamente o caso-de-uso sobre a decisão por parte do HRA a que recurso de hardware esta unidade de trabalho será atribuída ou ainda o caso-de-uso relativo à organização as unidades submetidas até ao momento e ainda não atribuídas aos recursos de hardware.

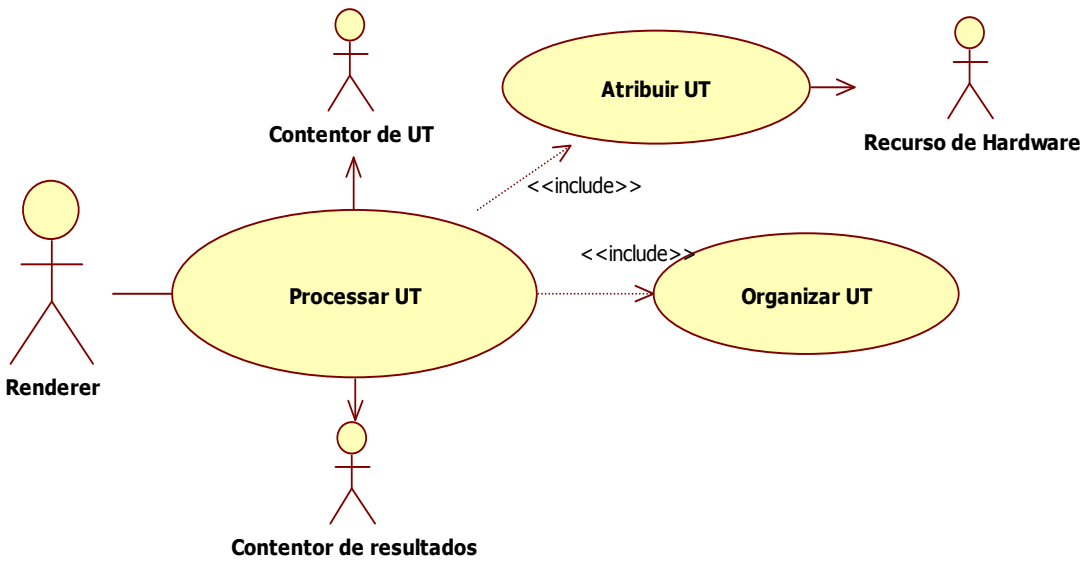

Figura 16 – Caso-de-uso "Processar UT" pelo *renderer*

O caso-de-uso "Organizar UT" é sempre incluído quando é submetida uma ou mais unidades de trabalho, pois existe uma alteração no conjunto de UT contidas no contentor de UT. O HRA poderá decidir, tendo em conta as novas características do conjunto de UT e parâmetros de comportamento do sistema, organizar ou manter a ordem e organização corrente do conjunto de UT. Por este motivo o caso-de-uso é incluído no caso-de-uso "Processar UT" e não uma extensão condicional (Arlow 2005).

Os contentores são considerados como actores neste caso-de-uso pois devem gozar de alguma autonomia em relação ao HRA, em termos de gestão interna. Os

contentores poderiam ser considerados com sendo parte do sistema caso sejam directamente geridos pelo HRA.

De salientar que como pré-condição deste caso-de-uso, no caso do contentor de UT não poder aceitar as UT submetidas porque não tem capacidade actual de armazenamento, o *renderer* é bloqueado até que algumas das UT anteriormente submetidas sejam processadas, existindo por consequência capacidade de acolhimento para as UT submetidas [7].

3.3.1.2 Obtenção das unidade de trabalho processadas (resultados)

O *renderer* necessita de recolher os resultados do processamento das unidades de trabalho anteriormente submetidas. Este caso-de-uso é distinto do anterior porque a submissão de UT e a obtenção dos respectivos resultados não é síncrona ou totalmente sequêncial.

A título de exemplo, o resultado de uma unidade de trabalho específica pode ser obtido após o resultado de outra unidade de trabalho submetida posteriormente, devido à alteração da sequência de processamento decidida pelo HRA tendo em conta o conjunto de unidade de trabalho e respectivo tipo de cada unidade de trabalho num determinado momento.

A Figura 17 apresenta os dois actores considerados, o *renderer* e o contentor com os resultados das UT processadas, que interagem entre si através do HRA. Cada unidade de trabalho deve conter um identificador que associa o pedido com o resultado do respectivo processamento.

[7] Em suma o renderer é bloqueado, isto é, é adormecido para não produzir mais UT durante um intervalo de tempo.

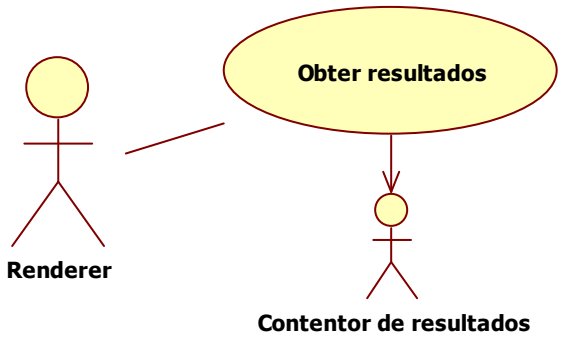

Figura 17 – Caso-de-uso "Obter resultados" pelo *renderer*

A obtenção dos resultados, mais propriamente o pedido dos resultados, por parte do *renderer* (ciclo principal de *rendering*) pode depender de diversos aspectos, como sejam o tempo de espera, a quantidade e a ordem dos referidos resultados. Os detalhes sobre estes assuntos serão discutidos mais em pormenor na última parte do capítulo numa vertente mais técnica e menos formal.

Em termos gerais, tendo em conta os aspectos anteriormente referidos, o *renderer* questiona o contentor de resultados, por todos os resultados ou apenas uma parte deles que no limite pode ser apenas um resultado específico de uma unidade de trabalho submetida, sendo que o contentor lhe devolve os resultados pedidos caso estes existam. O que deve ser efectuado com esse ou esses resultados é da responsabilidade do ciclo principal do *renderer*.

Como pré-condição óbvia, se se pode chamar realmente uma pré-condicão, o pedido de processamento de uma unidade de trabalho deve ser submetido antes de ser solicitado o seu respectivo resultado.

3.3.1.3 Iniciar sistema

O caso-de-uso "Iniciar sistema" aqui apresentado, Figura 18, embora seja em termos temporais o primeiro caso realizado no sistema, a sua importância para a compreensão dos casos-de-uso anteriores é diminuta. É apresentado com o intuito de esclarecer alguns pormenores do modelo proposto.

Figura 18 – Caso-de-uso "Iniciar sistema" realizado pelo HRA.

Na realidade, são especificados dois casos-de-uso, um responsável pela pesquisa e por possíveis inicializações do hardware (recursos de hardware), o outro pela configuração do sistema com os parâmetros definidos nos ficheiros de configuração.

No caso-de-uso "Configurar sistema", o HRA deve criar as estruturas de funcionamento necessárias ao correcto funcionamento dos recursos de hardware, como seja o caso do contentor de UT e o contentor de resultados, e configurar os parâmetros do sistema com base nos dados descritos nas configurações definidas pelo utilizador e ficheiros de configuração.

No caso-de-uso "Configurar recursos" inclui a inicialização dos recursos e o estabelecimento dos mecanismos de controlo e comunicação entre o HRA (sistema) e os recursos de hardware presentes. Pode ainda incluir a pesquisa de algum tipo particular de hardware não especificado na configuração inicial do sistema. As estruturas de comunicação com os recursos de hardware presentes do lado do HRA também são criadas dentro deste caso-de-uso.

3.3.2 Diagramas de actividades

As actividades consideradas nos próximos pontos incluem, dentro da medida do possível, os elementos abordados nos casos-de-uso anteriormente descritos. São considerados os aspectos com uma importância mais significativa, ficando os pormenores da implementação relegados para a parte técnica do capítulo.

3.3.2.1 Submissão de uma unidade de trabalho e a obtenção do respectivo resultado

O diagrama da Figura 19 indica as actividades envolvidas na criação e processamento de uma UT, e consequente obtenção do resultado por parte do *renderer*, envolvendo o HRA e os recursos de hardware disponíveis no sistema.

O actor recurso de hardware, referido no diagrama, representa um recurso genérico, apenas com capacidade de processar a unidade de trabalho submetida pelo HRA ao referido recurso. A decisão sobre qual o recurso de hardware com capacidade para efectuar o processamento e que não esteja ocupado por outros processamentos, é realizada ao nível do HRA na actividade designada por "Organizar UT".

Os contentores das UT e dos resultados são considerados como parte do HRA e, por consequência, as funcionalidades providenciadas por estes elementos são parte da funcionalidade providenciada pelo próprio HRA. O diagrama de classes apresentado mais adiante inclui esta integração, descrita de uma forma mais clara e formal.

Por uma questão de simplicidade, o diagrama de actividades apresentado apenas refere-se à submissão de uma única unidade de trabalho ao HRA, não considerando se o sistema se encontra no início, no final ou em qualquer outro momento da produção da imagem ou animação, dada a reduzida importância desse facto no diagrama de actividades apresentado.

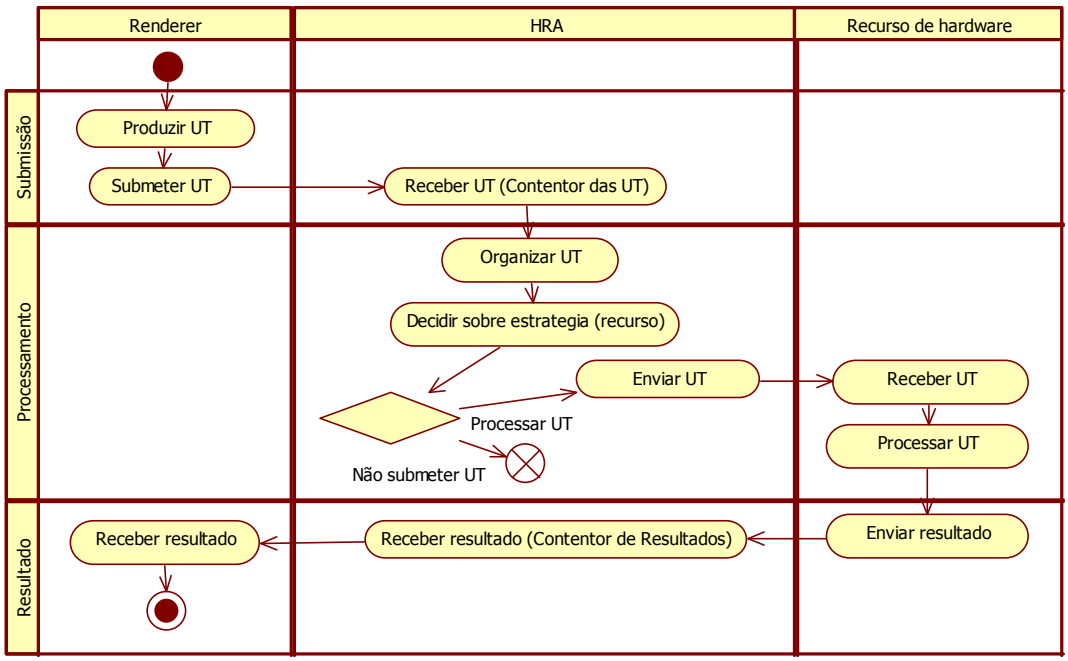

Figura 19 – Diagrama de actividades para o processamento de uma UT

Seguindo a sequência lógica das actividades apresentadas no diagrama, o *renderer* produz uma unidade de trabalho com base num conjunto de dados do cenário, na configuração do sistema e na primitiva que se pretende aplicar aos referidos dados, por exemplo o cálculo da normal num ponto de uma superfície.

Esta unidade de trabalho é submetida ao HRA através da sua colocação no contentor de UT, sendo que a actividade "Receber UT", por parte do HRA, é apenas a sinalização ao HRA que existe uma nova unidade de trabalho no contentor das UT.

Tendo em conta a quantidade e tipo de UT presentes no contentor das UT, o HRA efectua uma organização das referidas UT com base num conjunto de parâmetros, com por exemplo o tipo das UT, o equipamento disponível no momento da organização ou as prioridades associadas às UT em espera.

Uma vez organizadas, agrupadas e ordenadas as UT, o HRA deve decidir sobre qual a melhor estratégia para enviar a unidade de trabalho, ou conjunto de UT, tendo em conta o estado dos recursos de hardware, a estratégia definida, a

carga actual (contentor de UT), e os modelos de comunicação a usar entre o HRA e o recurso. Tendo em conta os parâmetros indicados, o HRA poderá decidir entre as situações:

- Não enviar a unidade de trabalho submetida, ficando esta a aguardar uma decisão posterior, despoletada noutra submissão ou na finalização do processo de *rendering*;
- Enviar a unidade de trabalho para processamento no recurso de hardware seleccionado, com base nas premissas do sistema: Estado do recursos, capacidade de realização do processamento por parte do recurso e conjunto de UT no contentor;
- Enviar a unidade de trabalho em conjunto com outras UT para um determinado recurso, com as premissas do ponto anterior, mas aplicadas a um conjunto de UT.

O algoritmo para a organização das UT deve garantir que nenhuma unidade de trabalho ficará indefinidamente no contentor das UT. Para além deste facto, deve existir um mecanismo que permita efectuar o "*flush*", isto é, processar as UT remanescentes no contentor, mesmo que não sejam submetidas novas UT.

Da parte do recurso de hardware, este deve receber as UT enviadas através do mecanismo de comunicação definido e efectuar o respectivo processamento. O resultado é enviado para o HRA para ser colocado no contentor de resultados, que por fim será removido do contentor pelo *renderer*.

3.3.2.2 Configurar sistema

Numa fase inicial o sistema deve ser configurado tendo em conta os parâmetros definidos pelo utilizador e as características do sistema. Quando se refere o sistema, significa incluir todos elementos presentes, desde do HRA aos recursos de hardware. O diagrama da Figura 20, apresenta as actividades relativas

à inicialização do HRA e dos recursos do sistema, incluindo as transições de estado entre as actividades.

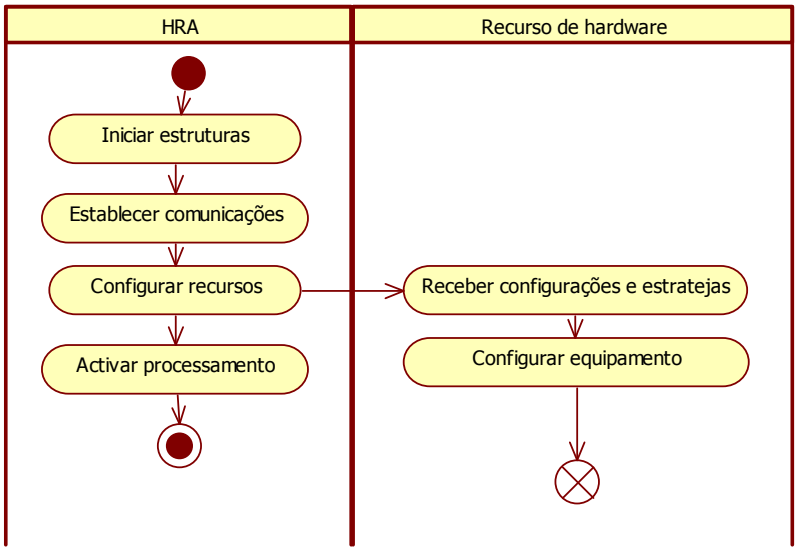

Figura 20 – Diagrama de actividades para inicialização do sistema

Quando é inicialmente invocado, o HRA efectua um determinado conjunto de operações, nas quais se inclui a criação das estruturas necessárias ao funcionamento do sistema (e.g. contentores). De seguida, tendo em conta as configurações indicadas através dos dados fornecidos pelo utilizado ou ficheiros de configuração e estado de recursos fisicamente existentes, são criados os canais de comunicação com os recursos bem como são definidas as políticas associadas à comunicação com cada tipo de recurso.

Após estabelecidas as comunicações, é realizada a configuração de cada recurso presente e incluído no sistema de *rendering*. Esta fase de configuração pode envolver o envio de parâmetros de configuração e do código para implementação da gestão do próprio recurso de hardware. Tudo depende da complexidade e programabilidade de cada recurso. Um recurso com um funcionamento mais simples pode requerer apenas um conjunto de valores iniciais, não existindo desta forma qualquer actividade por parte do recurso, ou então é

necessário enviar um conjunto de instruções que efectuam a configuração e gestão do recurso em equipamentos mais complexos e independentes do sistema principal.

Após este conjunto de actividades por parte do HRA, o sistema está pronto para funcionar, passando-se à fase de *rendering* propriamente dita.

3.3.3 Diagramas de classes

Os diagramas de classes pretendem definir o comportamento de cada elemento relevante ao sistema, nomeadamente na perspectiva das funcionalidades que cada elemento disponibiliza para realizar as actividades e os requisitos definidos num problema em concreto.

A definição das classes foi baseada essencialmente nos diagramas de casos-de-uso e de actividades anteriormente apresentados, bem como nos requisitos expressos em pontos anteriores do presente capítulo.

No caso relativo ao modelo proposto foram consideradas as seguintes classes:

- Unidade de trabalho – A classe que modela uma unidade de trabalho é composta pelos dados e primitiva (código) necessária para realizar os cálculos conducentes à obtenção dos resultados. Também é possível determinar o estado em que se encontra a unidade de trabalho, isto é, se já foi ou não processada e se já foi atribuída a um recurso físico ou não;
- Contentor – Elemento responsável por armazenar as unidades de trabalho submetidas ao sistema mas ainda por processar, mas também o elemento que armazena as unidades de trabalho já processadas. Dadas as circunstâncias enunciadas o sistema é composto por duas instâncias desta classe, o contentor de UT submetidas e o contentor de resultados (unidades de trabalho já processadas);

- HRA – Classe que define o comportamento do HRA em geral, incluindo os elementos de configuração do sistema. É também composto pelos dois contentores anteriormente referidos, que por sua vez são compostos pelas unidades de trabalho;
- Recurso – Esta classe modela o comportamento e funcionalidades dos recursos de hardware com capacidade de processamento de unidades de trabalho. A Classe Recurso é parcialmente implementada no sistema que contém o HRA e outra parte da implementação será efectuada no próprio recurso físico. As funcionalidades, nomeadamente de gestão do recurso, podem ser implementadas do lado do HRA ou do próprio recurso de hardware, dependendo da complexidade e programabilidade do referido recurso.

As quatro classes apresentadas formam o núcleo do modelo proposto. O *renderer* ou a implementação das primitivas de *rendering* referidas por múltiplas vezes no decorrer do presente texto, não se encontram enquadradas nas classes apresentadas, pois são tratados como dados do modelo e não como implementação do modelo proposto em si. Uma evidência desse tipo de tratamento, a representação das primitivas de *rendering* como dados, é formalizada na classe UT (Figura 21). O atributo "código" é a implementação ou um apontador/identificador para o código da primitiva aplicável aos dados, atributo "dados" da unidade de trabalho.

O diagrama de classes da Figura 21 é uma representação gráfica em UML dos diagramas de classes anteriormente descritos. Analisando mais profundamente o diagrama apresentado, é possível observar que a classe HRA é composta por elementos da classe Recursos e da classe Contentor, sendo esta última classe, classe Contentor, composta por elementos da classe UT (unidade de trabalho).

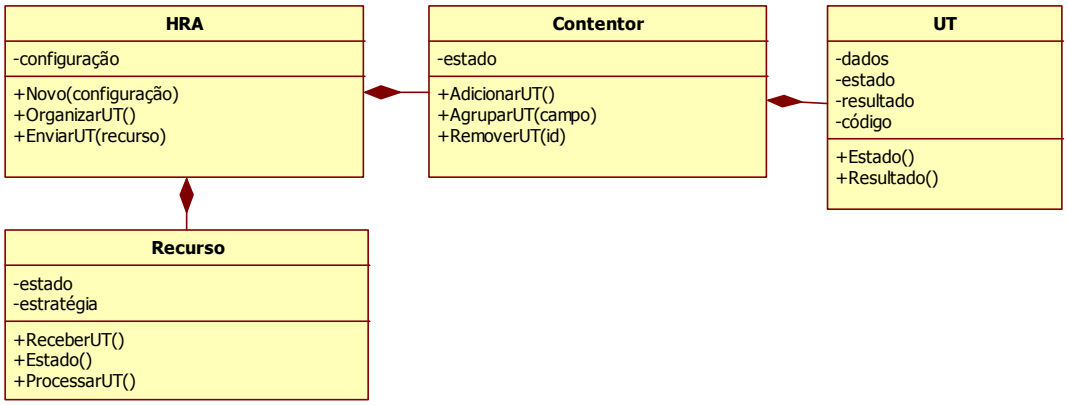

Figura 21 – Diagrama de classes do modelo proposto

Numa perspectiva mais detalhada do funcionamento de cada classe temos os seguintes atributos e operações:

- Classe "**UT**":
  - Atributo "**dados**" – conjunto de dados necessários para realizar um primitiva específica, podendo incluir raios, descrição parcial de superfícies, dados de iluminação e dados dos materiais que compõem a superfície. No limite pode ser apenas um apontador/identificador de um conjunto de dados;
  - Atributo "**código**" – código da primitiva que aplicado aos dados anteriormente referidos, produz o resultado pretendido. Devido à natureza heterogénea do sistema, este atributo poderá ser apenas um identificador da primitiva, estando o código colocado numa zona de memória específica do recurso de hardware onde a primitiva é processada;
  - Atributo "**resultado**" – conjunto de dados reservado para colocar o resultado da aplicação da primitiva designada aos dados da unidade de trabalho;

- Atributo "**estado**" – indicação do estado em que a unidade de trabalho se encontra. Poderá assumir um dos seguintes valores: não atribuída, não processada ou processada;
- Operação "**Estado**" – Obtém o estado da unidade de trabalho, contido no atributo "estado";
- Operação "**Resultado**" – Obtém o resultado da unidade de trabalho, caso esta tenha o resultado já calculado.

- Classe "**Contentor**":
  - Atributo "**estado**" – indicação do estado do contentor. Este poderá estar vazio, com uma quantidade definida de elementos ou cheio;
  - Operação "**AdicionarUT**" – Adiciona uma unidade de trabalho específica ao contentor;
  - Operação "**AgruparUT**" – Reorganiza a ordem das UT presentes no contentor com base num campo específico;
  - Operação "**RemoverUT**" – Remove do contentor uma unidade de trabalho específica contida no contentor, através da sua identificação (id).
- Classe "**Recurso**":
  - Atributo "**estado**" – indicação do estado em que o recurso de hardware se encontra. O estado poderá ser "não utilizado" ou com uma carga de processamento atribuído, entre apenas uma unidade de trabalho até ao limite máximo de UT definido para cada tipo de recurso;
  - Atributo "**estratégia**" – conjunto de informações sobre a estratégia de comunicação e processamento implementada no recurso;

- Operação "**ReceberUT**" – Obtém uma unidade de trabalho do HRA para processamento;
- Operação "**Estado**" – Obtém o estado actual do recurso, incluindo processamento e estado da unidade de trabalho no recurso;
- Operação "**ProcessarUT**" – operação interna de indicação para o processamento da unidade de trabalho.

- Classe "**HRA**":
  - Atributo "**configuração**" – conjunto de informações relativas à configuração actual do HRA;
  - Operação "**NovoHRA**" – define um novo HRA, com base numa configuração definida. Embora o sistema standard apenas utilize uma única instância do HRA, o modelo proposto poderá conter múltiplas instâncias em funcionamento colaborativo;
  - Operação "**OrganizarUT**" – organiza as UT dos contentores existentes com base num estratégia definida na configuração do HRA;
  - Operação "**RemoverUT**" – remove uma unidade de trabalho do contentor de resultados para serem devolvidas ao ciclo principal de *rendering*.

Os diagramas de classes apresentados, mais propriamente os atributos e operações indicadas, não pretendem ser uma descrição exaustiva do modelo proposto, mas tem o intuito de providenciar uma visão alargada do funcionamento pretendido.

3.3.4 Diagrama de estados

Os diagramas de estados que seguidamente se apresentam, são referentes às classes e diagramas referidos nos pontos anteriores da especificação.

Novamente, o conjunto de diagramas que se apresenta não pretendem ser uma exposição exaustiva de todas as variantes de todas as classes e respectivas funcionalidades, mas sim dar uma visão global do processamento e gestão de uma unidade de trabalho no sistema, bem como o respectivo funcionamento do modelo em situações consideradas fundamentais e exemplificativas desse mesmo modelo.

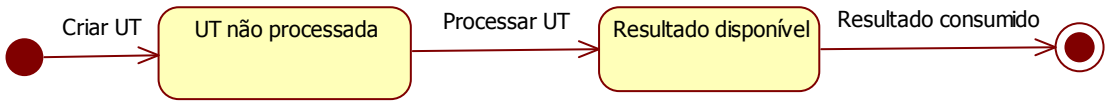

Figura 22 – Diagrama de estados de uma unidade de trabalho

O diagrama de estados de uma unidade de trabalho (Figura 22) é simples. Uma unidade de trabalho é criada com os dados, código da primitiva desejada e espaço para guardar os resultados pretendidos. Quando os resultados são utilizados (consumidos) depois do processamento da unidade de trabalho, esta deixa de existir.

O número de estados de um Contentor (Figura 23) é também reduzido, mas com diversas transições. O Contentor ou contém UT ou então está vazio. Um contentor termina a sua existência se estiver vazio e for dada a indicação que o processo de *rendering* foi finalizado.

Existe ainda a transição de "OrganizarUT", que não altera o estado do Contentor, pelo menos de uma forma significativa. Apenas altera a ordem das UT presentes no contentor. É também de realçar que esta transição só tem significado no Contentor de UT, e não no Contentor de resultados, pois a ordem dos resultados, compreenda-se UT, deste último não é relevante.

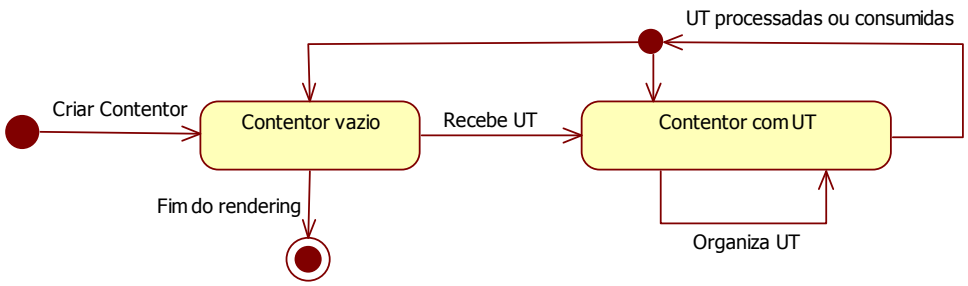

Figura 23 – Diagrama de estados de um Contentor

Os estados do HRA são dependentes do estado do contentor de UT e do contentor de resultados. Quando o HRA recebe uma unidade de trabalho, esta é colocada no contentor das UT e quando terminar o seu processamento o contentor dos resultados recebe a referida unidade de trabalho. A sinalização de fim de *rendering* é considerada quando ambos os contentores estiverem vazios.

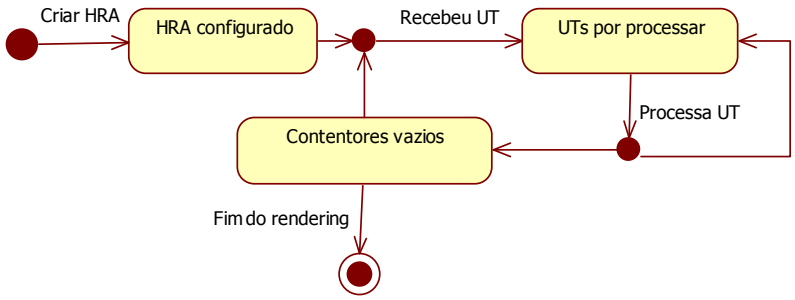

Figura 24 – Diagrama de estados do HRA

O estado de um Recurso (Figura 25) é partilhado entre o sistema anfitrião e o recurso físico que representa. A maioria das acções de transição são efectuadas internamente no próprio recurso de hardware e não no HRA. Da mesma forma que no estado do HRA, o estado final do recurso é alcançado quando é finalizado o *rendering* do cenário.

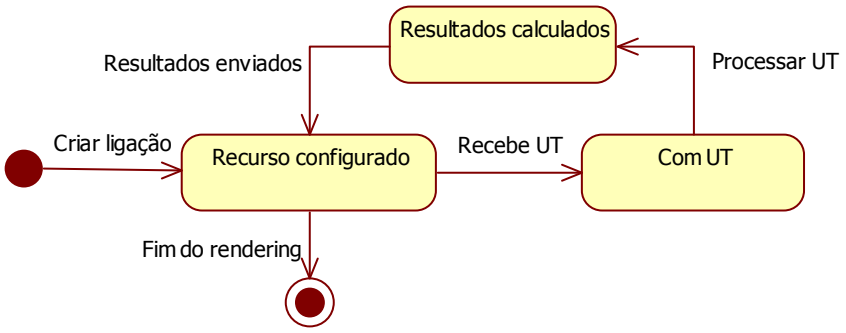

Figura 25 – Diagrama de estados de um Recurso

Dependendo do tipo de recurso é possível que exista um estado de "desconfiguração", isto é, para deixar de utilizar o recurso pode ser necessário efectuar algumas acções de finalização. Neste caso existe mais um estado, "Recurso desactivado", entre a transição "Fim de rendering" e o estado final. Salienta-se novamente que é este estado extra é apenas dependente do tipo de recurso.

3.4 Interacção do modelo

Para uma melhor compreensão do funcionamento do sistema de *rendering* proposto como um todo, desde do ciclo principal do *renderer* até a utilização específica dos recursos de hardware disponíveis, apresentam-se seguidamente quatro exemplos de interacção do modelo para diferentes situações de cálculos de distintos tipos de raios.

Os dois primeiros exemplos estão relacionados com a fase inicial do processo de *rendering* de um cenário, nomeadamente o cálculo de visibilidade de um conjunto de objectos ou em particular de um objecto. Especificamente o primeiro exemplo tem em consideração os raios que são projectados do ponto de vista do observador e que passam pelo plano 2D onde é calculada a imagem (Ward 1998; Shirley 2003; Pharr 2004). O segundo tem por base os raios que são projectados a partir dos pontos de intersecção dos objectos que contribuem para a imagem final,

para os restantes elementos do cenário (Ward 1998; Shirley 2003; Pharr 2004). Esta componente pretende estimar/amostrar quais os objectos do cenário que contribuem com iluminação indirecta para um ponto de intersecção inicial, ponto este pertencente a um objecto directamente visualizado do ponto de vista considerado.

O segundo grupo de exemplos, o terceiro e o quarto, estão relacionados com o cálculo da contribuição directa e indirecta das fontes de luz presentes no cenário, os quais implicam uma maior complexidade algorítmica em relação aos dois anteriores exemplos apresentados.

Em qualquer um dos exemplos, a descrição apresentada segue a sequência temporal dos cálculos (sequência lógical) e respectivas componentes, partindo do ciclo principal de *rendering* para os recursos, terminando com a devolução dos resultados calculados pelos recursos ao ciclo principal.

3.4.1 Cálculo da intersecção de um raio primário com um objecto

O processo de síntese da imagem de um determinado cenário, inicia com a geração de raios primários que passam pelas áreas definidas por cada *pixel* que compõem a imagem final, partindo do ponto de vista definido pelo utilizador. Tendo em conta a resolução definida, em *pixeis*, e dimensão da imagem do plano imagem, são criadas múltiplas áreas contíguas, áreas que representam a área de influência de cada *pixel*. A Figura 26 explica graficamente este processo, onde apenas é representado um único raio, *r1*, passando pela área definida de um só ponto da matriz imagem [8].

[8] A pequena oval escura junto do raio pretende indicar em que local o raio intersecta o plano imagem e por consequência a área do *pixel* que será calculado, mais propriamente a coloração do referido *pixel*.

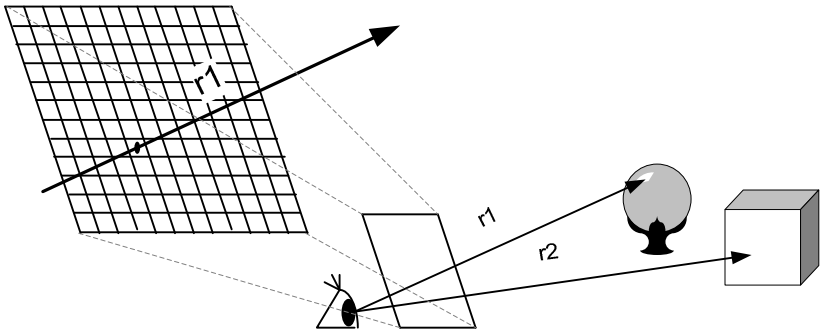

Figura 26 – Divisão do plano imagem.

Esta etapa não implica necessariamente a criação de unidades de trabalho no sentido definido anteriormente no capítulo, pois esta etapa é apenas necessária no início do processo de *rendering* de uma imagem do cenário, sendo o seu custo, em termos de poder computacional, relativamente pouco significativo em relação aos restantes cálculos. Esta divisão do plano imagem pode ser implementada directamente pelo ciclo principal do *renderer* ou então por uma questão de opção, pode ser implementada através de uma única primitiva o que por sua vez dará origem a uma unidade de trabalho única no sistema. Neste último caso, é criada uma unidade de trabalho que será passada ao HRA via contentor de unidades de trabalho.

Após serem definidas as áreas para cada *pixel* do plano imagem, o ciclo principal do *renderer* vai criar um conjunto de raios primários que partem do ponto de vista definido e que atravessam a área de um *pixel* e apenas um só *pixel*. O número de raios gerados para cada *pixel* é definido como um parâmetro de configuração do processo de síntese. Por exemplo, caso seja definido que para cada *pixel* deverão ser projectados 10 raios, e utilizando uma resolução de 100 por 100 *pixeis*, o total de raios primários criados será de 100.000. O processo de criação de cada raio primário, tendo em conta uma área e um ponto vista, vai gerar uma unidade de trabalho. Todas as unidades de trabalho são colocadas no contentor de UT, que são retiradas do contentor pelo HRA quando submetidas para processamento.

A distribuição espacial dos número de raios que passam pela área de um *pixel* pode ser efectuada de diversas formas, no sentido de evitar efeitos de *aliasing*. Mais informação sobre diferentes tipos de distribuições e respectivas vantagens e desvantagens podem ser consultados em (Ward 1998; Shirley 2003; Pharr 2004). Na Figura 27 está representada uma distribuição aleatória de 4 raios que passam pela área de um ponto.

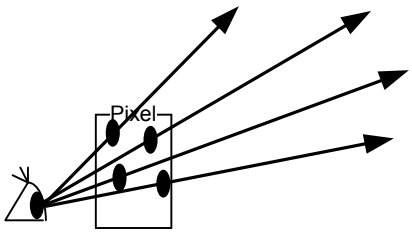

Figura 27 – Distribuição de raios que passam por um *pixel*

O HRA pode optar por retirar estas unidades de trabalho do contentor de UT de duas formas:

- Retirar cada unidade de trabalho individualmente e submeter essa unidade de trabalho a um recurso livre que implementa a funcionalidade requerida;
- Retirar um conjunto de UT com os mesmos dados, isto é, o mesmo ponto de vista e a mesma área, enviando para um recurso livre um pacote compacto com múltiplas UT mas apenas com uma única referência aos dados comuns, o ponto de vista e a área do pixel. A selecção de múltiplas UT com o mesmo conjunto de dados não é complexa, pois a submissão por parte do *renderer* é normalmente feita de forma sequêncial, para o mesmo *pixel*, por uma questão de implementação simplificada do algoritmo.

Em qualquer uma das situações, os dados passados pelo HRA aos recursos têm em conta a existência, ou não, de memória local no recurso, do tipo de recurso para onde é enviada a unidade ou as unidades de trabalho, do dado ou da primitiva a executar. Apenas é necessário enviar o identificador do dado ou da primitiva, caso o código da primitiva ou o dado já esteja presente na memória local

do recurso. Mesmo na primeira vez é possível passar apenas um identificador da primitiva a aplicar pois é possível que alguns recursos implementem nativamente (em hardware) a primitiva desejada, como seja o caso das placas gráficas que implementam uma alargada gama de primitivas de *rendering*.

Depois de calculado o raio, ou os raios solicitados, por parte do recurso, estes são devolvidos ao HRA utilizando a estratégia de comunicação definida para o recurso em questão. Por sua vez, o HRA coloca o resultado no contentor de resultados, que será posteriormente removido pelo ciclo principal de *rendering* (*renderer*).

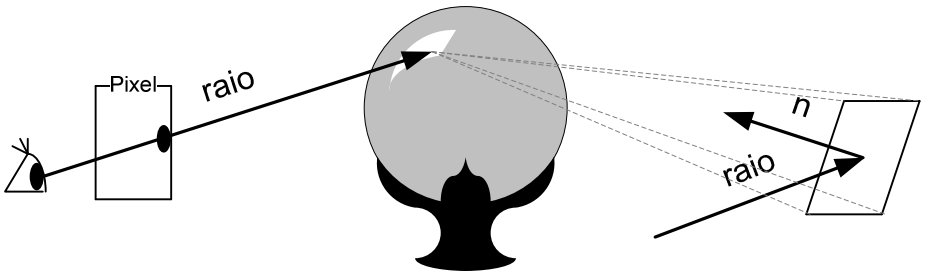

Figura 28 – Intersecção de um raio com uma superfície

Numa segunda fase, o ciclo principal inicia a produção de unidades de trabalho que vão inquirir se cada um dos raios, já disponíveis, passa por algum objecto do cenário como se pode observar no caso especifico apresentado na Figura 28. Existem quatro abordagens que podem ser seguidas para esta fase:

1. Para cada raio é criada uma unidade de trabalho que questiona toda a geometria do cenário se existe uma intersecção desta geometria com o referido raio. Nesta situação, compete ao HRA decompor a geometria em elementos fundamentais, para que possam ser processados por recursos que não implementem a intersecção com formas complexas ou objectos constituídos por múltiplas formas. O HRA pode sempre tomar a decisão de não decompor a geometria em formas básicas, utilizando apenas os

recursos que implementem a primitiva com suporte para uma complexidade de geometria elevada;

2. A segunda abordagem, será delegar a responsabilidade de decompor a geometria global para o ciclo principal de *rendering*. Neste caso, alguma da complexidade do algoritmo geral é mantida no ciclo principal, o que facilita a gestão dos recursos por parte do HRA e melhora o balanceamento de trabalho pelos recursos, devido a uma menor granularidade (menor dimensão) das unidades de trabalho (Chalmers 2002);
3. Na terceira abordagem, em vez de se optar por testar a intersecção de um raio com todos os objectos do cenário, é efectuado o teste de um objecto com todos os raios primários. Esta abordagem requer que todos os raios primários estejam disponíveis no início desta fase. Nesta situação é possível que exista a perda de alguma eficiência, pois é criado um ponto de sincronização, que pode levar a que alguns recursos não sejam utilizados na etapa anterior (Chalmers 2002) ou que terminem a sua parte do trabalho relativamente mais cedo em relação ao ponto de sincronização;
4. Por último, pode-se optar por calcular numa só primitiva todas as intersecções de todos os raios com a geometria total do cenário. Esta abordagem apresenta todas desvantagens das abordagens 1 e 3, mas pode ter uma implementação muito eficiente em determinados tipos de hardware. Uma abordagem semelhante foi utilizada por Longhurst et al. (Longhurst, Debattista et al. 2005), para o cálculo dos objectos de um cenário que interceptam os raios primários, mas apenas o raio primário que passa pelo centro do *pixel* é considerado. É devolvida uma matriz imagem com identificação de que cada objecto é intersectado em cada *pixel*. Longhurst et al. também realizou sob-amostragem do cenário, no entanto, segundo Shirley e Morley (Shirley 2003), a distribuição linear de raios apresenta desvantagens em determinados aspectos.

Entre a decomposição total da geometria e a utilização de um bloco de dados com toda a geometria ou totalidade dos raios, é possível encontrar um ponto de equilibro entre a eficiência do sistema e o *overhead* introduzido pelo mesmo.

No caso do HRA decompor a unidade de trabalho, colocada pelo ciclo principal no contentor das UT, em unidade de trabalho de menor complexidade, nomeadamente decompor a geometria, estas novas UT serão enviadas para os recursos disponíveis e capazes de as processar. Depois de calculadas as intersecções pelos recursos, os resultados são conjugados na unidade de trabalho original, que por sua vez é colocada no contentor de resultados e retirada pelo ciclo principal de *rendering*.

No caso de não existir uma decomposição da geometria por parte do HRA, apenas é feita a avaliação e gestão das UT submetidas pelo ciclo principal, relativamente à distribuição destas pelos recursos disponíveis, não existindo, por consequência, a conjugação de múltiplos resultados de várias UT. Os resultados são passados na mesma configuração da qual são obtidos dos recursos.

Finalmente, na última etapa deste exemplo, é necessário determinar em que ponto da superfície de cada objecto é que cada raio primário a intersecta. Os cálculos para determinar se um objecto é interceptado por um raio, são os mesmos que são utilizados na determinação do ponto da superfície do objecto onde o raio a intersecta, diferindo apenas no resultado final. No primeiro caso, se intercepta, o resultado é um valor booleano que indica se foi ou não intersectado o objecto, enquanto que no segundo caso são devolvidas as coordenadas do ponto onde foi intersectado pela primeira vez o objecto.

Apenas por uma questão de eficiência e por forma a evitar repetição de cálculos, a primitiva indica se um objecto é intersectado por um raio, devolvendo um ponto nulo quando não existe intersecção (um apontador nulo em termos de implementação em linguagem C), ou então as coordenadas do ponto de intersecção em caso afirmativo. Esta técnica, ou técnicas muito semelhantes são

utilizada por outros autores (Shirley 2003), com o objectivo de diminuir o tempo de cálculo total.

3.4.2 Cálculo da intersecção de um raio secundário com um objecto

O cálculo de raios secundários é bastante semelhante ao exemplo relatado anteriormente, no que se refere às primitivas utilizadas, diferindo no entanto na distribuição dos raios e respectiva utilização. No caso dos raios primários, é feita uma distribuição dos raios através de uma função pseudo-aleatória ou pelo particionamento da área ocupada por um *pixel* (Ward 1998; Shirley 2003; Pharr 2004), enquanto que no caso dos raios secundários são criados um número predefinido de raios por cada raio primário que atinge um objecto.

Os raios secundários têm origem no ponto de intersecção do raio primário com o objecto, com uma distribuição pseudo-aleatória ou linear na semiesfera (Kajiya 1986; Ward 1988; Shirley 2003; Pharr 2004) com orientação no sentido da normal à superfície e centrada no ponto de intersecção, tendo os raios secundários como origem esse mesmo ponto. Figura 29 apresenta um exemplo de uma distribuição.

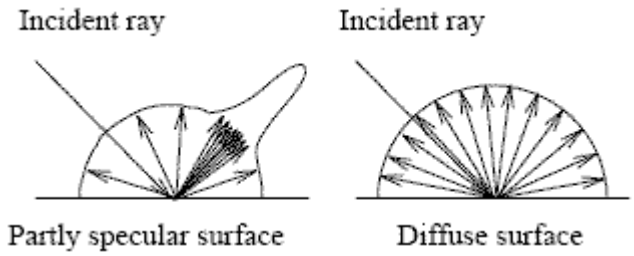

Figura 29 – Raios secundários definidos segundo o ângulo de incidência e BRDF (Reinhard 1994)

Os raios secundários são utilizados para determinar quais as fontes de luz que contribuem para a iluminação directa mas também podem ser utilizados para "amostrar" o cenário, mais propriamente os objectos que podem contribuir para a luz indirecta no ponto da superfície em questão. Os raios secundários podem por sua vez produzir mais raios secundários, dependendo do número de *bounces*

(Ward 1998; Shirley 2003; Pharr 2004) definidos para o sistema. Este processo é recursivo e interactivo. Segundo o modelo proposto, toda a interacção e recursividade deve ser tratada pelo *renderer*, tratado no sentido de produzir as unidades de trabalho correspondentes.

Exposto o algoritmo subjacente aos raios secundários, o processo desenvolve-se quando o ciclo principal pretende criar um conjunto de raios secundários por cada intersecção de um raio primário com um objecto. De uma forma semelhante ao caso anteriormente exposto, Cálculo da intersecção de um raio primário com um objecto, existem 4 abordagens para esta etapa:

1. Utilizar a primitiva de intersecção de um raio com uma forma geométrica simples para criar a respectiva unidade de trabalho;
2. Utilizar uma primitiva mais complexa com um raio e uma geometria complexa ou total;
3. Utilizar uma primitiva complexa com múltiplos raios e um objecto, que compõem a unidade de trabalho;
4. Utilizar uma primitiva bastante complexa e extensa com um conjunto de raios e toda a geometria. Esta abordagem não goza do mesmo princípio de eficiência enunciado no exemplo dos raios primários, pois a distribuição de raios é efectuada por uma semi-esfera e não um plano (matriz imagem). Não é do conhecimento do autor, a existência de alguma implementação de uma primitiva deste tipo em hardware existente, directa ou indirectamente.

As abordagens dos pontos 3 e 4 são mais complexas e com resultados menos eficientes em relação ao caso dos raios primários porque existe uma menor congruência dos raios (Chalmers 2002). Os raios secundários assumem todas as direcções da semiesfera definida e a estrutura que contém os pontos de intersecção, não representado através de uma matriz de *pixeis* mas sim uma matriz de ângulos sólidos.

Independentemente da abordagem utilizada, os passos são semelhantes aos seguidos nos raios primários. O ciclo principal envia a unidade de trabalho para o contentor das UT, sendo a unidade de trabalho retirada pelo HRA para efectuar o seu processamento, segundo a distribuição de trabalho definida, num recurso disponível e capaz de processar a unidade de trabalho em questão. Quando o resultado ficar disponível, este é devolvido ao HRA através do modelo de comunicação definido e colocado por este no contentor dos resultados. O ciclo principal remove os resultados obtidos e com base na natureza interactiva e recursiva do algoritmo geral, decide se lança novos raios secundários a partir dos novos pontos intersecções, com base na configuração e especificações iniciais. Esta decisão pode também ser baseada no limite temporal das primitivas ou na necessidade de uma resposta em tempo real por parte do sistema.

3.4.3 Cálculo da iluminação especular num ponto

Os dois casos anteriormente referidos, os procedimentos estavam relacionadados com a geometria, no entanto, numa segunda etapa do algoritmo de *rendering*, é necessário calcular as contribuições das fontes de luz e dos objectos, directamente ou indirectamente, para a coloração de cada ponto do plano da imagem tendo em conta as propriedades fotométricas dos objectos e das fontes de luz intervenientes. Não existindo uma dependência directa entre a iluminação de cada *pixel* e os seus vizinhos (Reinhard and Jansen; Chalmers 2002; Dietrich 2003; Wald 2003; Wald 2005), é possível intercalar cálculos de visibilidade e intersecção de outros *pixeis* com cálculos de iluminação.

Após a definição da direcção de um raio, primário ou secundário, e respectiva intersecção deste raio com uma superfície da geometria do cenário, o ciclo principal necessita de determinar a contribuição de cada fonte de luz no ponto, produzindo as unidades de trabalho conducentes à determinação do valor da coloração devida à contribuição directa. Entre os dados das UT produzidas encontram-se o ponto e as propriedades do material no ponto, mais propriamente

a componente especular. Na maioria dos casos os valores para a coloração e propriedades do material são representados no formato RGB (Schwarz 1987; Hauta-Kasari 1988; EastmanKodakCompany 1999).

No caso das fontes de luz serem representadas por superfícies e não por pontos é feita uma amostragem múltipla do objecto auto iluminado (Ward 1998), que produzirá uma unidade de trabalho complexa (geometria complexa e múltiplos raios) ou um conjunto de UT com geometria simples ou complexa, e um raio por cada unidade de trabalho. As considerações apresentadas para os casos da intersecção de raios primários e secundários, relativamente à complexidade geométrica e inclusão de múltiplos raios numa só unidade de trabalho, também devem ser aplicáveis no funcionamento do sistema a este nível.

As UT produzidas são colocadas no contentor das UT pelo ciclo principal, em que o HRA é responsável por remover e gerir, enviando para um recurso livre e capaz de produzir o resultado pretendido. Novamente, o modelo de comunicação definido entre o recurso específico e o HRA será utilizado para recolher o resultado ou conjunto de resultados. Salienta-se ainda, como já foi referido anteriormente, que pode existir uma mistura entre vários tipos de UT processadas, de distintas etapas do processo de síntese de imagem. O resultado da unidade de trabalho, já no domínio do HRA, é colocado no contentor de resultados, onde o *renderer* o deverá recolher.

3.4.4 Cálculo da iluminação difusa num ponto

No exemplo anterior foi considerado que o *renderer* gera as UT referentes à contribuição da componente especular. Em algumas abordagens, esta componente é modelada pela composição de outros parâmetros, como acontece no caso do pacote de software Radiance (Ward 1992; Ward 1998). Neste software, são utilizados coeficientes das propriedades de alguns tipos de materiais como seja o caso do "metal" e "madeira" (*wood*), responsáveis pela representação da componente especular e difusa dos materiais. Esta é uma outra representação da

combinação das propriedades fotométricas do material, matematicamente semelhante à mudança de eixos em coordenadas cartesianas.

Tendo em conta a componente difusa dos materiais, o cálculo da luz difusa é efectuado de uma forma aproximada, multiplicando os seus coeficientes pelo co-seno do ângulo entre a normal e o raio de luz que atinge a superfície. O processo é bastante semelhante ao descrito no ponto anterior, à excepção do coeficiente angular indicado.

O contentor de UT pode conter, em fila de espera, cálculos para a componente especular e a para componente difusa de pontos distintos do cenário, existindo apenas dependência da intersecção de um raio primário ou secundário com um objecto, na composição destas unidades de trabalho. É possível organizar unidades de trabalho com dados comuns mas com cálculos distintos, componente especular e componente difusa, de forma a minimizar representações e tempos gastos em comunicações entre o HRA e os recursos.

Finamente, considerando estas duas componentes, sem descartar outras possíveis interacções, como por exemplo da componente de iluminação global, é possível calcular o resultado da contribuição de todas as componentes em cada *pixel* do plano imagem pretendido.

3.4.5 Outros exemplos para outros tipos de operações

Os quatro exemplos anteriormente apresentados podem ser considerados entre os fundamentais para síntese de uma imagem. No entanto facilmente pode estender o processo de composição e de decomposição de elementos, os cálculos de raios primários e secundários para a componente de iluminação global. As operações aritméticas utilizadas são semelhantes em estruturas de dados semi-complexas que podem ser decompostas em elementos de mentor dimensão e facilmente paralelizável, por exemplo o algoritmo de iluminação global denominado de Irradiância (*Radiance Irradiance Cache*) apresentado no ponto 2.2.1 e utilizado no software Radiance (Ward 1998).

3.5 Problemas associados a ambientes heterogéneos

A utilização de um ambiente heterogéneo de recursos capazes de contribuir para o *rendering* de uma imagem ou animação traduz-se, a vários níveis, na necessidade de resolver problemas que surgem associados a este tipo de organização. Os problemas que podem surgir são multiplicas e de natureza diversa. De entre eles destacam-se:

- A representação dos dados – A representação dos dados, não só dos dados primitivos como seja o caso dos valores do tipo real (*float*) ou inteiro, podem ser ligeiramente distintos em distintos recursos, na gama e representação intrínseca, mas também pelo suporte para estruturas mais complexas de alto nível baseadas nos dados do tipo primitivos;
- Transferência de dados entre elemento principal e recursos – A forma como os dados e o código a executar em cada recurso são transferidos do elemento principal, normal do CPU/memória principal do sistema, para aos elementos locais do recurso pode ser diverso, mais propriamente o modelo de comunicação a utilizado;
- Paradigma de programação – Aos dois aspectos anteriores junta-se o problema relativo ao paradigma de programação específico de cada recurso. Este aspecto pode limitar ou constringir parte do algoritmo global que se pretende implementar;

Os pontos seguintes apresentam uma visão mais detalhada dos aspectos referidos relativamente ao desenho e implementação de um sistema de *rendering*, num ambiente heterogéneo de recursos.

3.5.1 Representação dos dados

Existe sempre um compromisso entre a portabilidade de um sistema e a sua implementação eficiente, quer em termos de algoritmo quer em termos de

estruturas de dados utilizadas. Um aspecto essencial neste caso é a representação dos dados designados de primitivos ou elementares, no que respeita à sua resolução e tipo de dado. Por exemplo, um *renderer* pode ser implementado recorrendo a dados do tipo inteiro ou na representação de valores em vírgula flutuante, valores do tipo real. A utilização do formato de vírgula flutuante garante um resultado final mais correcto do *rendering* de cenários com uma alta fidelidade relativamente à utilização de inteiros, pois a gama de valores e aproximações de valores devido cálculo aritméticos apresentam um menor erro (IEEE 1985).

Algumas placas gráficas utilizam representações não standard na aritmética de vírgula flutuante, utilizando dados do tipo real com 16 bit, denominado de *half resolution* (meia resolução), ou de 20 bit por questões de eficiência (Hutchins 2007). Esta diversidade de representações e gamas pode agravar ainda mais a colaboração entre recursos para um objectivo comum.

Num nível mais abstracto do hardware, as estruturas que representam os elementos de um cenário, como seja o caso de formas geométricas ou raios, podem ser reorganizadas internamente e adaptadas ao hardware utilizado, tendo em conta questões de *cache* local ou aproveitando a estrutura do pipeline de baixo nível que suporta o paralelismo ao nível das instruções do hardware (Wald 2003).

Quando se utiliza um conjunto de hardware heterogéneo, o realinhamento e reorganização dos dados em tempo de execução do *renderer* para cada recurso específico pode consumir um tempo demasiado elevado, criando desta forma um *overhead* significativo capaz de anular qualquer ganho obtido pela optimização do código e/ou adaptação das estruturas de dados, quer primitivas de complexidade algorítmica mais elevada.

Também devem ser considerados os diferentes esquemas de *cache* dados utilizados entre os vários níveis do *renderer*, desde do ciclo principal até à memória local de cada recurso. São um exemplo deste tipo situação as estruturas de aceleração, como seja o caso das *octrees*.

3.5.2 Modelos de comunicação heterogéneos

A utilização de um simples modelo de comunicação do tipo cliente/servidor (Tanenbaum 1988) nas comunicações entre recursos e o HRA pode não potenciar uma utilização eficiente do respectivo recurso em algumas situações ou de uma forma global. A elevada gama de tempos de latência nas comunicações num ambiente heterogéneo é um exemplo dessa situação. O tempo de latência na comunicação pode variar de alguns micro segundos, no caso de memória partilhada em que intervêm apenas mecanismos de sincronização com *mutexes* ou semáforos (Tanenbaum 2001), até vários milissegundos numa comunicação via ligação Ethernet (Tanenbaum 1988).

O modelo de comunicação deve permitir o envio de unidades de trabalho sem a confirmação de resultados de unidades anteriores ainda que pendentes, e que não influenciam as unidades de trabalho que se pretende submeter. Para além disso, o sistema de comunicações deve evitar qualquer tipo de contenção e ser suficientemente rápido para providenciar que os pacotes de dados necessários aos recursos sejam rápidos, isto é, os recursos com um elevado poder de cálculo de unidades de trabalho. Uma abordagem do tipo *pipelined* na comunicação com recursos mais lentos é mais apropriada, permitindo o envio consecutivo de unidades de trabalho sendo o fluxo de dados controlado pela capacidade da linha de comunicação e não pelo processamento das unidades.

A estratégia de comunicação deve ter em conta a necessidade de manter o funcionamento de cada recurso próximo do valor máximo de eficiência no decorrer do *rendering*, para todos os recursos envolvidos no processo.

Mais informação e estratégias sobre este tópico são focados no ponto 3.6.6 - Comunicação entre o HRA e os recursos.

3.5.3 Modelos de programação heterogéneos

Existem actualmente diferentes tipos de hardware disponíveis nas configurações denominadas computadores de secretária (*desktop computers*),

capazes de providenciar uma significativa capacidade de cálculo, para além de equipamentos adicionais que facilmente se podem adquirir para estas plataformas. No entanto esta variedade apresenta uma contrariedade, nomeadamente implica uma gama elevada de modelos de programação distintos em cada equipamento utilizado. Os modelos de programação passam pelo paradigma clássico de um programa sequêncial com uma descrição imperativa, ou pelo modelo de processamento de *stream*, até à reconfiguração do equipamento e instruções em tempo de execução.

3.5.3.1 Adaptação do hardware à tarefa

A proliferação de FPGAs (Camea 2008) em diversos tipos de placas adicionais de processamento veio trazer para o mundo das aplicações mais comuns, um modelo de programação novo que permite a configuração do hardware, à tarefa que será executada, em tempo de inicializacão, ou mesmo a própria reconfiguração durante a execução dos diversos estágios de uma tarefa. Este modelo apresenta uma mistura entre o paradigma clássico de programação imperativa e modular com a reprogramação/reorganização do recurso ou recursos associados.

A inclusão de diferentes paradigmas de programação não devem interferir com o algoritmo principal do *renderer*, que por defeito é executado no CPU. A utilização de primitivas básicas simples, permite uma mais fácil e simples adaptação na implementação das primitivas em cada um dos distintos recursos disponíveis num ambiente heterogéneo, supondo que este ambiente apresenta distintos paradigmas de programação.

3.6 Desenho dos elementos do HRA

Nos pontos seguintes são aprestadas as linhas gerais tomadas no desenho do HRA, que tentam minorar os problemas enunciados anteriormente, com maior enfeze no ponto 3.5 – Problemas associados a ambientes heterogéneos, para além

de descrever de uma forma organizada os aspectos considerados importantes no desenvolvimento do sistema de *rendering* suportado numa plataforma com recursos heterogéneos. Outros aspectos mais generalistas, como seja o caso da portabilidade, eficiência e de poder ser escalável também são mencionados.

3.6.1 Primitivas

O balanceamento de trabalho entre diferentes recursos de hardware, quer seja num ambiente heterogéneo ou homogéneo, é mais fácil de atingir quando a execução de uma tarefa pode ser decomposta em blocos de trabalho de granularidade reduzida (Chalmers 2002), considerando granularidade reduzida a dimensão reduzida dos trechos de código (*small footprint*). Tendo em conta este aspecto é possível catalogar ou classificar o processo de *ray tracing* em três grupos de primitivas: primitivas de baixo nível; primitivas de alto nível; primitivas de alto nível com recursividade.

No grupo das primitivas de baixo nível podem ser incluídas todas as primitivas básicas ligadas ao *ray tracing*, como seja o caso da intersecção de um raio com uma forma de representação geométrica simples ou a iluminação de um ponto devido a uma fonte de luz. As primitivas incluídas neste grupo são normalmente fáceis de implementar num vasta variedade de hardware, mesmo que o hardware apresente uma reduzida quantidade de memória local disponível ou de um reduzido suporte (*small footprint*) para uma formas de programação mais complexa e extensiva. Considera-se que não existe dependência directa entre os resultados da execução de uma primitiva e os dados de entrada necessários para a execução de outras primitivas.

As dependências são tratadas ao nível da reserva de unidades de trabalho, por parte do HRA e do ciclo principal de *rendering*. Este tipo de abordagem facilita uma distribuição mais uniforme de unidades de trabalho pelos conjuntos de recursos disponíveis.

As estruturas de dados mais complexas e os algoritmos de maior complexidade, que por sua vez são compostos por primitivas de baixo nível, podem ser implementados com uma única primitiva ao nível do HRA. Um exemplo deste tipo de primitiva é o atravessamento de uma forma geométrica complexa, para calcular qual o ponto onde é interceptado um raio de luz.

O aspecto mais importante na classificação como primitiva básica é a sua total independência de outras tarefas ou primitivas dentro do sistema. Foi definida a primitiva com a nomenclatura de "auto contida". Isto significa que quando a primitiva básica é iniciada, isto é, iniciada a sua execução, a primitiva tem todos os dados necessários para a sua completa finalização e todo o código necessário para a sua execução está presente no recurso onde a unidade de trabalho está a ser processada. Nenhum outro tipo de primitiva necessita de ser invocada, a não ser que esta seja uma parte integrante da própria primitiva. A primitiva também não requer directamente que seja criada outra unidade de trabalho durante a execução da primitiva devido a resultados intermédios. Essa responsabilidade é do *renderer* principal.

Existe também uma limitação temporal na execução da primitiva, isto é, à primitiva está sempre associado um tempo máximo de execução derivado da natureza de tempo real definido na especificação na abordagem do problema. Esta regra excluí as primitivas que fazem recursividade directa sobre si próprias.

Todas as primitivas e código de natureza recursiva, devido ao algoritmo de iluminação global por exemplo, são executadas no ciclo principal do *renderer*, que por sua vez produzirá todas as unidades de trabalho implementadas pelas primitivas necessárias, definidas anteriormente e derivadas dos resultados parciais obtidos pelas primitivas (UT) processadas.

A decomposição das tarefas do algoritmo de *rendering* em três classes, primitivas de baixo nível (básicas), primitivas de alto nível não recursivas e primitivas de alto nível de gestão e criação de unidades de trabalho, permite a definição de uma API (*Aplication Programing Interface*) que pode ser totalmente

ou parcialmente implementada em cada recurso. A implementação desta API em cada recurso não necessita de ser total, devido a limitações de memória local ou programabilidade do respectivo recurso. Neste caso, o HRA deve ter conhecimento de quais as primitivas implementadas através da sua configuração, com por exemplo uma lista de primitivas implementadas em cada tipo de recurso.

A Figura 30 apresenta a hierarquia e divisão da classificação definida para as primitivas de *rendering*.

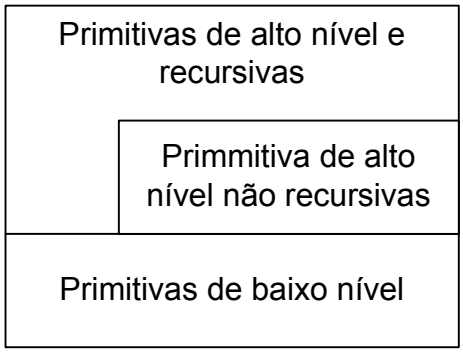

Figura 30 – Divisão e dependências entre tipos de primitivas definidas

A marginalização de parte do hardware e a falta de balanceamento no sistema ocorre quando todas as unidades de trabalho existentes num determinado intervalo de tempo não podem ser processadas por um recurso específico e disponível. Uma forma de contornar este problema é aceitar unidades de trabalho referentes a tarefas produzidas mais recentemente e iniciá-las antes do tempo, no recurso disponível e capaz de as processadas.

Este tipo de solução não é trivial pois requer um acréscimo de complexidade na componente algorítmica relativa à gestão do sistema, acréscimo esse que poderá absorver qualquer ganho conseguido pela execução fora de ordem (Chalmers 2002) de determinadas unidades de trabalho.

Outra solução, é aumentar a capacidade do contentor das unidades de trabalho, aumentando a possibilidade da existência de unidades de trabalho com primitivas diferentes e capazes de serem realizadas num recurso disponível. Esta

última abordagem pode implicar uma diminuição da interactividade do sistema global e falhas na resposta em tempo real. Estas questões serão aprofundadas no ponto 3.6.3.

3.6.2 Unidades de trabalho

O trabalho submetido ao HRA é apresentado sobre a forma de unidades de trabalho. Uma unidade de trabalho é composta por um conjunto de dados e por uma primitiva, descrita no ponto anterior, que será aplicada aos dados da unidade de trabalho. Um exemplo simples de uma unidade trabalho é o teste da intersecção de um raio com um objecto. O código do teste é a primitiva e a representação geométrica do objecto e do raio são os dados da unidade de trabalho.

Na realidade, na implementação do sistema, o código da primitiva não está directamente embutido na unidade de trabalho, mas sim é utilizado uma referência, sob a forma de um identificador único, da primitiva em questão. Este tipo de abordagem oferece algumas vantagens, como seja o caso da transmissão de pacotes de dados mais pequenos entre HRA e recursos, e uma menor dimensão na representação das unidades de trabalho. Outra vantagem neste tipo de representação é um menor gasto da memória local de cada recurso, pois não é necessário a implementação completa da primitiva dentro da unidade de trabalho, mas sim um apontador (identificador) para o código local da primitiva dentro do recurso. Esta vantagem é ainda mais relevante em recursos que utilizam um esquema de memória partilhada com o HRA. O próprio HRA pode utilizar este tipo de representação para efectuar o agrupamento e gestão das unidades de trabalho através do tipo de primitiva que deve ser aplicada aos dados de um conjunto de unidades de trabalho. Esta situação optimiza a utilização de recursos e minimiza a dimensão dos pacotes de dados enviados, e por consequência o tempo de comunicação, não sendo necessário enviar múltiplas descrições do código, mas sim

apenas uma, caso o código não esteja presente no recurso que processa as unidades de trabalho.

A utilização de uma referência em vez da descrição completa de dados numa unidade de trabalho, apresenta as mesmas vantagens referidas relativamente à utilização de uma referência para as primitivas.

Como uma unidade é definida pela referência da primitiva e pela referência aos dados que a primitiva é aplicada, como se pode observar na Figura 31, as unidades de trabalho podem ser vistas como uma função do código e dos dados. Com esta visão sobre as unidades de trabalho, os dois parâmetros podem ser utilizados no algoritmo de ordenação e prioritização das unidades presentes no contentor de unidades de trabalho, que liga o HRA ao ciclo principal do *renderer*, bem como no algoritmo de envio de unidades de trabalho para os recursos.

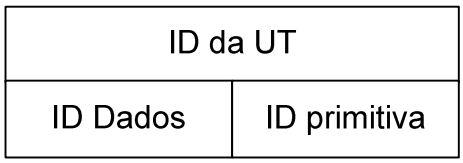

Figura 31 – Composição geral de uma unidade de trabalho

3.6.3 Contentores para unidades de trabalho e resultados

A comunicação e o fluxo de dados entre o *renderer* principal e o HRA é implementado através de duas estruturas de dados denominadas de contentores. O contentor das unidades de trabalho é de dimensão limitada, para garantir uma forma de controlo de fluxo (Tanenbaum 1988; Tanenbaum 2001). Quando o *renderer* tenta colocar uma ou mais unidades no contentor, a chamada feita é do tipo bloqueante (Tanenbaum 1988; Tanenbaum 2001; Plachetka 2002). Isto significa que no caso do contentor não tenha capacidade para alberga mais unidades, ou mais concretamente para as unidades requeridas, o *renderer* vai ficar bloqueado só sendo reactivado quando existir uma determinada capacidade para

mais unidades, nomeadamente para as unidades requeridas pelo *renderer* principal.

Na implementação efectuada, a capacidade do contentor das unidades de trabalho é estaticamente definida no início do funcionamento do sistema. Também foram definidos os estados de vazio e cheio, o contentor é considerado vazio quando o número de unidades que podem ser submetidas é superior a um valor predefinido mas sempre igual ou inferior à capacidade máxima do mesmo. No limite, o contentor é considerado vazio com capacidade para mais 1 unidade de trabalho, no caso extremo sem nenhuma unidade de trabalho. O contentor é considerado cheio quando ultrapassar ou igualar um valor previamente definido, o valor uma UT num extremo e totalmente ocupado com valor máximo da sua capacidade no outro extremo. Este tipo de abordagem evita parcialmente que o *overhead* gasto ao bloquear e acordar (*sleep* e *wakeup*) (Tanenbaum 2001) do *renderer* seja excessivo, dando uma margem de manobra entre o estado cheio e vazio.

O valor da capacidade máxima do contentor das unidades de trabalho pode afectar a interactividade e a resposta em tempo real do sistema, bem como as definições da noção de cheio e vazio. Estes valores podem fazer com que o ciclo principal do *renderer*, que faz a interface como o mundo exterior, permaneça inactivo durante um tempo demasiado elevado. Este aspecto pode diminuir a percepção de interactividade e tempo real pretendida. Um equilíbrio entre a eficiência, utilizando capacidades maiores, e interactividade, utilizando valores de menor dimensão, deve ser futuramente analisado e estudado.

Num futuro desenvolvimento do sistema, através do *profilling* das unidades de trabalho, disponibilidade combinada dos recursos utilizados e resposta do sistema, os valores da capacidade máxima, noção de cheio e vazio do contentor das unidades de trabalho poderão ser alterados dinamicamente durante o decorrer do processo de *rendering*, com base numa estratégia ou perfil predefinido.

Na implementação actual ambos os contentores residem numa zona de memória partilhada entre o ciclo principal do *renderer* e o HRA, ambos estão implementados no CPU, com controlo de acesso efectuado por *mutexes* (Butenhof 1997; Tanenbaum 2001).

Também é possível efectuar outra melhoria de eficiência na organização no contentor de unidades de trabalho. Não é fora do comum que um conjunto de unidades de trabalho contenham algumas semelhanças, quer relativamente à primitiva ou primitivas a aplicar quer ao nível dos dados, e que sejam submetidas conjuntamente. Para acomodar esta situação foi implementada uma função capaz de submeter um conjunto de unidades de trabalho com referência à mesma primitiva ou mesmo conjunto de dados. Este tipo de opção também facilita o algoritmo de prioritização e gestão do HRA, pois as unidades presentes no contentor de unidades de trabalho apresentam uma preordenação natural, tendo como consequência um menor tempo gasto na ordenação e agrupamento de unidades de trabalho que são submetidas aos recursos.

A gestão do contentor de resultado é bastante mais simples pois não carece de qualquer limitação de capacidade, apenas limitada pela dimensão de memória disponível no sistema, ou de ordenação e prioritização dos dados. As condições e parâmetros definidos para o contentor de unidades de trabalho influenciam indirectamente a quantidade de resultados obtidos, embora não existindo nenhum parâmetro directo que limite a capacidade do contentor de resultados. Os resultados removidos (na realidade é removida a referência aos dados pois estes é que são utilizados) é da responsabilidade do *renderer* principal. Na realidade o contentor de resultados contém unidades de trabalho mas já processadas, pois como já foi referido as unidades de trabalho incluem uma zona para armazenar os resultados da aplicação da primitiva aos dados.

3.6.4 Representação dos dados

A representação de dados primitivos deve ser considerada num ambiente heterogéneo com uma elevada diversidade de elementos de hardware, nomeadamente na representação binária. Problemas como a dimensão e alinhamento (Wald 2004; Camea 2008) dos dados devem ser resolvidos antes destes serem enviados para os recursos. Para circunscrever os problemas de alinhamento, a implementação de dados básicos de *rendering*, como pontos, intervalos, *pixeis* e cores devem ser representados sobre a forma de matrizes. Estruturas de alto nível e mais complexas que utilizam elementos heterogéneos na sua composição, são formados pelos elementos de base anteriormente indicados. Todos os dados básicos, inicialmente referidos, utilizam o dado primitivo do tipo real (representação em vírgula flutuante) presente em todos os equipamentos utilizados, pois apresenta as características mais condizentes com os resultados espectáveis no *rendering* de cenários com uma elevada fidelidade.

3.6.5 Gestão e prioritização de unidades de trabalho

A principal função do HRA, para além de providenciar um nível de abstracção entre o hardware e o *renderer* principal, é de gerir, de uma forma eficiente e balanceada, todos os recursos presentes no sistema, por forma a garantir o menor tempo possível na apresentação dos resultados pretendidos.

Num ambiente paralelo e heterogéneo a ordem pela qual as unidades de trabalho são processadas não é imperativa, pelo menos a uma pequena escala. Esta presunção permite que seja efectuada uma reorganização na ordem de processamento das unidades de trabalho, permitindo melhorar ou mesmo em alguns casos maximizar quer a utilização dos pipelines dos recursos, quer a quantidade de recursos utilizados simultaneamente.

Uma submissão em grupo dentro de um pacote melhora a gestão e eficiência de utilização da memória e respectiva comunicação aos recursos. Apenas uma cópia dos dados ou do código das primitivas é necessária, melhorando, para além

das comunicações, a estratégia e taxa de acerto (Tanenbaum 2001) na memória de *cache* local de cada recurso.

O desenho do HRA segue as especificações definidas no projecto RoD (RoD 2006), referido no ponto 2.4.4. Foi definido que todas as unidades de trabalho submetidas pelo *renderer* devem ser processadas. Nesse sentido cada unidade de trabalho tem associado um intervalo máximo, ou tempo de validade, dentro do qual devem ser realizadas. Caso uma ou mais unidades não sejam processadas, devido à chegada de novas unidades de trabalho com maior prioritárias, dentro do intervalo definido, as unidades com o tempo esgotado passam automaticamente para a prioridade mais elevada, mesmo em relação à prioridade indicada pelo *renderer*.

Para evitar atrasos elevados no processamento de unidades de trabalho, o valor do tempo de validade deve ser relativamente pequeno em relação ao tempo em que é necessário o seu resultado no *renderer*. A influência da capacidade do contentor das unidades de trabalho neste tempo de validade é tida em conta no início do processo de *rendering*. Este valor pode ser ajustado dinamicamente ao longo do processo em conjunção com a capacidade do contentor das unidades de trabalho e a noção de vazio e cheio anteriormente referidas, num futuro desenvolvimento do sistema.

3.6.6 Comunicação entre o HRA e os recursos

Quando analisadas as comunicações entre o HRA e os recursos, duas propriedades apresentam-se como importantes. A largura de banda (Tanenbaum 1988; Tanenbaum 2001) e o tempo de latência da comunicação (Tanenbaum 1988; Tanenbaum 2001). Tendo em conta estes dois parâmetros, diferentes estratégias de comunicação podem ser definidas:

- Um simples modelo cliente/servidor pode ser implementado para uma comunicação com um elevado débito, largura de banda e uma latência muito baixa. A unidade de trabalho ou unidades de trabalho são

enviadas independentemente da dimensão do respectivo pacote de dados. Como exemplo desta situação temos o CPU virtual, que é implementado através de um *thread* (Butenhof 1997; Tanenbaum 2001) de processamento no CPU. O resultado do processamento das unidades de trabalho é colocado directamente na memória logo que seja realizado o respectivo cálculo. Neste caso, a latência é muito pequena e limitada apenas pelo tempo de acesso à memória partilhada e da activação do mecanismo de sincronização. Quando não existe trabalho, o CPU virtual está adormecido (*sleep*), dando lugar aos restantes *threads* em funcionamento no sistema, como seja o caso do HRA e do ciclo principal do *renderer*;

- Para recursos com uma latência elevada, o modelo anterior não providencia uma interacção capaz de manter o pipeline do recurso próximo da capacidade máxima, a não ser que o rácio comunicação versus processamento (Tanenbaum 1988; Chalmers 2002) seja muito baixo. O tempo em que o recurso espera por novas unidades de trabalho e devolve os resultados pode ser de uma magnitude tal que torna a eficiência de utilização do recurso muito baixa e desta forma prejudicar a eficiência global do sistema. Nesta situação deve ser utilizado, na comunicação, um protocolo capaz de permitir o envio de várias unidades de trabalho sem confirmação através do envio do respectivo resultado. Um mecanismo semelhante ao protocolo *sliding window* (Tanenbaum 1988), utilizado no conjunto de protocolos TCP/IP (Tanenbaum 1988), será uma boa solução, ver Figura 32. Cada pacote é numerado sequencialmente, até um valor máximo de pacotes sem confirmação. Conforme vão chegando os resultados dos pacotes enviados, é confirmado o seu processamento, podendo assim serem enviados mais pacotes com novas unidades de trabalho. O HRA envia novos pacotes conforme a disponibilidade da comunicação e a gestão de unidades de trabalho por si efectuada. O número de pacotes

máximos que podem ser enviados sem confirmação é definido no início do processo, no entanto, tal como outros aspectos referidos anteriormente, o valor pode ser ajustado de uma forma dinamicamente tendo em conta as características dos recursos e perfil do trabalho executado em cada situação. Obviamente as estratégias para esta gestão dinâmica carecem de um estudo mais aprofundado.

Na implementação efectuada não foi considerada a existência de possíveis erros de retransmissão ou perda de pacotes, dada que a sua existência não foi verificada. No entanto isso não garante que não possa suceder noutras futuras implementações ou plataformas. Neste caso será necessário recorrer a mecanismos e algoritmos de correcção de erros em equipamentos mais susceptíveis a estas situações (Tanenbaum 1988).

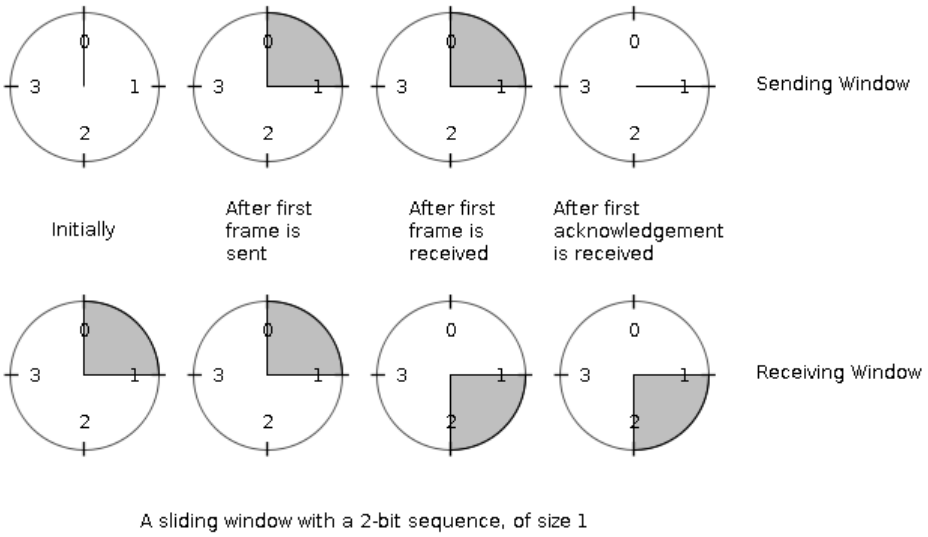

Figura 32 – Modelo de comunicação *sliding window* [Img05]

O protocolo *sliding window* (Tanenbaum 1988) específica a forma de corrigir os erros nos pacotes e possíveis perdas de pacotes. Também a complexidade do protocolo de comunicação e a programabilidade que é possível no recurso considerado devem ser ponderados na decisão de que tipo de protocolo de comunicação deve ser utilizado entre o recurso e o HRA.

Para recursos com uma largura de banda bastante reduzida outras opções podem ser consideradas. Nesse sentido o HRA deve maximizar o número de unidades de trabalho por pacote, na perspectiva da dimensão do pacote. Uma forma de colocar um maior valor no pacote de dados consiste em enviar múltiplas unidades de trabalho com a mesma referência à primitiva ou aos dados. Desta forma pode ser criado um pacote com uma única referência aos dados ou código.

Outro aspecto que pode melhorar a comunicação, e da responsabilidade do HRA, está relacionado com o não envio, ou melhor o não reenvio de dados e primitivas já presentes na memória local do recurso utilizado.

Esta abordagem, técnica de *caching*, diminui significativamente a dimensão do pacote nomeadamente no que se refere às primitivas, pois uma referência indicadora pode apenas conter alguns bytes enquanto o código de uma primitiva pode conter na ordem das dezenas às centenas de bytes, ou mesmo milhares de bytes. Um aspecto que limita esta abordagem é a capacidade da memória local do recurso, que deve descartar dados ou código presentes na memória, para acomodar os novos dados ou primitivas.

A escolha entre um simples modelo cliente/servidor ou a utilização de um protocolo do tipo *sliding window* deve ser ponderada considerando os vários aspectos referidos, como seja o caso da programabilidade do recurso, memória local disponível, largura de banda e tempo de latência na comunicação.

A comunicação entre recursos e o HRA não se confina ao envio de unidades de trabalho e recepção de resultados. Pacotes e protocolos adicionais para a gestão dos recursos podem ser implementados, com o objectivo de melhorar a eficiência e controlo dos recursos. Este conjunto extra de comunicações pode e deve seguir as linhas gerais definidas para a comunicação das unidades de trabalho e resultados. É de ter em conta que grande parte da gestão e das decisões são tomadas pelo HRA tendo em conta o estado dos contentores e dos recursos.

3.7 Questões funcionais e tecnológicas de implementação

A aplicação do modelo proposto apresenta algumas vantagens funcionais em diversos aspectos que melhoram, quer a sua própria implementação, quer uma futura expansão do sistema.

Ao nível da implementação, as primitivas catalogadas como pertencentes à classe de primitivas básicas (baixo nível) e classe das primitivas complexas mas não recursivas devem implementadas em linguagem C standard, formando uma bibliotecas standard de utilização (uma API), de fácil portabilidade para diversos tipos de recursos. A versão para o CPU inclui todo o conjunto de primitivas definidas e funciona como guia para implementação das versões para outros recursos, mesmo que estes recursos sigam um paradigma de programação distinto do CPU. É possível utilizar ou reutilizar a implementação das primitivas para o CPU noutros recursos distintos devido à sua reduzida dimensão (*small footprint*). No caso da classe de primitivas complexas, como estas são compostas por primitivas básicas, a sua adaptação é indirectamente efectuada pela adaptação das primitivas básicas e alguma adaptação da composição das primitivas básicas.

A decomposição do algoritmo geral de *rendering* em blocos de menor dimensão e a não obrigatoriedade da implementação completa de todo o conjunto de primitivas em todos os tipos de recursos, traduz-se na facilidade de implementar apenas parcialmente o conjunto de primitivas definido num determinado tipo de recurso. Esta abordagem traz duas vantagens imediatas:

- Não é necessário a completa implementação da especificação do *renderer* no hardware existente ou futuramente disponibilizado, encurtando os ciclos de desenvolvimento e utilização de novos recursos adicionados ao sistema;
- É possível definir apenas as primitivas que são mais eficientemente realizadas em hardware específico e optimizar desta forma a sua implementação.

Obviamente que esta abordagem, de implementar apenas parcialmente o conjunto das primitivas definidas num ou mais recursos, incorre na desvantagem de criar uma possível paragem no processamento de unidades de trabalhos em recursos específicos prejudicando a eficiência global do sistema. Pode não ser possível ter em cada momento do processo de *rendering* as unidades de trabalhos do tipo correcto para uma implementação parcial efectuada um dos recursos disponíveis. No caso de não existir unidades de trabalho compatíveis com a implementação específica de um determinado recurso, num determinado momento, este recurso não será utilizado e consequentemente afectará a eficiência global do sistema.

4 TRABALHO EXPERIMENTAL

4.1 Introdução

O principal objectivo do presente capítulo é descrever a implementação de um *renderer*, com base no modelo proposto no capítulo anterior, bem como apresentar os dados obtidos com distintas configurações para corroborar as premissas do referido modelo. De uma forma mais objectiva e específica pretende-se corroborar um objectivo fundamental: utilizar de uma forma cooperativa e eficiente um conjunto de recursos computacionais de um computador "standard", com o intuito de produzir uma imagem de alta-fidelidade num tempo comparativamente inferior em relação ao menor tempo gasto em qualquer um dos recursos utilizados, quando este é utilizado de uma forma individual para o *rendering* de um cenário.

Neste sentido, para validar a implementação, foram efectuadas diversas experiências, delineadas com base em múltiplas configurações e implementações específicas do referido *renderer* no sentido de corroborar, com os resultados obtidos, os pressupostos definidos nos objectivos desta tese.

Relativamente a organização do capítulo, ela está estruturada da seguinte forma: É apresentado inicialmente uma breve descrição da implementação onde é indicada a metodologia seguida, a plataforma utilizada, os pressupostos definidos na implementação do *renderer* e a caracterização de cada uma das configurações definidas. Finalmente, são apresentados de uma forma objectiva os resultados obtidos e respectivas comparações directas entre esses mesmos resultados das experiências nas diferentes configurações consideradas.

4.2 Implementação do *renderer*

A implementação do modelo especificado no capítulo 3 teve em conta vários aspectos, quer ao nível da codificação quer ao nível do objectivo pretendido. Nesse sentido, a implementação foi definida de uma forma modular, com o intuito de isolar possíveis influências não desejáveis nos resultados obtidos e provocados por uma dependência forte entre os distintos componentes de software que compõem o sistema. A divisão estrutural ou funcional utilizada para implementação, e já descrita no modelo proposto, é a seguinte:

- Primitivas básicas (baixo nível) de computação gráfica;
- Primitivas de alto nível não recursivas;
- Primitivas de alto nível recursivas e de gestão geral do processo de *rendering*;
- Primitivas de gestão dos contentores, recursos e comunicações entre os diversos elementos do sistema.

Esta definição ou estruturação também facilita o intuito de criar e isolar os diferentes elementos presentes entre distintas implementações bem com também facilita a comparação entre implementações, como seja o caso da implementação do *renderer* com o CPU sem HRA e a implementação do *renderer* em conjugação com HRA e outros recursos de processamento.

Numa vertente mais prática da implementação, a plataforma de desenvolvimento utilizada teve por base a linguagem C (ANSI C), com recurso ao compilador "GNU C compiler" numa plataforma Linux, nomeadamente devido à sua portabilidade e suporte em múltiplas plataformas. Para a placa gráfica foram utilizados kits de desenvolvimento providenciados pelos fabricantes das placas gráficas utilizadas (Nvidia 2008). Existem, no entanto, kits de desenvolvimento que podemos denominar de "mais avançados e genéricos" que permitem a programação directamente em linguagem C para as placas gráficas da Nvidia (Nvidia 2008).

Esta divisão modular também providenciou uma forma simples de desenvolver um *renderer* minimalista, isto é, sem recorrer necessariamente à implementação de algoritmos complexos de *rendererring*, mas sem comprometer uma possível implementação com algumas opções avançadas, dado que foram implementados um conjunto alargado gama de primitivas de *rendering*, nomeadamente de baixo nível e de alto nível não recursivo. Neste sentido, é possível construir um *renderer* personalizado, bastando para isso implementar o algoritmo das primitivas de alto nível recursivas e de gestão geral do processo de *rendering* pretendido [9].

4.2.1 Dados básicos e estruturas simples

Como já tinha sido referido no capítulo 3, no campo de questões práticas, a representação dos dados, mesmo dos dados compostos ou designados de complexos, assenta num tipo de dado de base comum. Os trechos de código apresentados da Figura 33 e da Figura 34 são exemplo da representação de diversos elementos usuais na implementação de um *renderer* recorrendo a um linguagem funcional e estruturada, como o caso da linguagem C que foi utilizada nesta implementação.

[9] Essa situação, foi efectuada na implementação de um *renderer* utilizando só o CPU para a primeira experiência delineada no ponto 4.4.1.

```
typedef double Real;                                        /* base type value */

typedef Real Coord;

typedef Real Interval[2];

typedef Coord Vector2D[2];

typedef Coord Vector3D[3];

typedef Coord Point2D[3];

typedef Coord Point3D[3];

typedef Real Color;

typedef Color RGB[3];
```

Figura 33 – Código ilustrativo da definição de tipo de dados de base

Como é possível verificar na Figura 33, as estruturas de base representativas de dados ditos primitivos são derivados do tipo real e sobre a forma de matriz, devido essencialmente a questões de alinhamento de dados entre diversas plataformas e recursos distintos de hardware. Quando os dados são definidos sobre a forma de uma matriz, todos os elementos da matriz são contíguos, enquanto quando se recorre uma estrutura com dados homogéneos não é possível garantir o mesmo alinhamento dos referidos dados em distintas plataformas. As estruturas de dados com campos heterogéneos não implicam qualquer tipo de alinhamento, como se pode verificar pelo código apresentado na Figura 34.

```
typedef struct Ray
{
  Point3D o;                                            /* point of origin */
  Vector3D d;                                       /* direction of the ray */
} Ray;

typedef struct Cache
{
  Real where;                                        /* where ray intercept */
  Vector3D normal;                             /* normal in interception point */
  Ray *ray;                                          /* which ray intercepted */
  RGB color;                                             /* what color there */
} Cache;
```

Figura 34 – Código ilustrativo da definição das estruturas complexas

4.2.2 Programação modular

Considerando ainda o código apresentado nas Figura 33 e Figura 34, é possível verificar uma hierarquia de complexidade crescente, isto os dados de complexidade média são representados por matrizes de dados simples, as estruturas de maior complexidade do sistema são composta por estruturas heterogenias de complexidade média e possivelmente baixa.

Ainda relativamente ao aspecto modular, é possível verificar que foram declaradas como genéricos os identificadores para as estruturas de dados complexas e respectivas funções de processamento, que desta forma minimizam o conhecimento da estrutura interna de determinados tipos de dados. Os trechos de código apresentados nas Figura 35 e Figura 36, apresentam esta abordagem, processamento genérica, recorrendo a apontadores do tipo *void* para os elementos pretendido. De certa forma é a abordagem parcialmente seguida em linguagens orientadas aos objectos, mas recorrendo ao C standard.

```
typedef struct Object
{
  void *data;
  int (*hit)(void *data, Ray *ray);
} Shape;
```

Figura 35 – Código (declaração) ilustrativo da relativa independência dos dados ao alto nível

No caso do trecho de código contemplado pela Figura 36 é apresentada a primitiva (função) que adiciona uma unidade de trabalho, na realidade um item genérico, a um contentor já definido e já configurado no início do funcionamento do sistema de *rendering*.

Em termos mais descritivos o dado "data" é adicionado ao fim da lista. O código associado aos *mutexes*, no inicio e final da função, é utilizado para sincronização e controlo do contentor, pois este elemento (contentor) pode ser acedido por distinto elementos (*threads*) em funcionamento no sistema.

```
int AddBinNodeData(Bin bin, void *data)
{
  int status = 0;
  BinNode *node;

  if (bin)
   {
    pthread_mutex_lock(&(bin->mutex));

    if (bin->size >= bin->high)                          /* bin almost full */
      pthread_cond_signal(&(bin->work));
    if (BinFull(bin))                                   /* if bin full wait */
      pthread_cond_wait(&(bin->empty), &(bin->mutex));

    node = bin->free;                        /* get a node from the free list */
    bin->free = node->next;

    node->data = data;
    node->next = NO_LINK;
    if (bin->first == NO_LINK)           /* bin empty then last is also first */
      bin->first = node;
     else                                    /* if not update the actual last */
      (bin->end)->next = node;
    bin->end = node;
    (bin->size)++;
    status = 1;
    pthread_mutex_unlock(&(bin->mutex));
   }

  return (status);
}
```

Figura 36 – Código (função) ilustrativo da relativa independência dos dados a alto nível

Como já foi referido a abordagem utilizada, de alguma forma na linha da programação orientada aos objectos, permite uma fácil adaptação do código e dos dados representativos das estruturas para diferentes recursos, bem como a alteração da representação geométrica, por exemplo. Desta forma os módulos definidos limitam a necessidade, em grande parte, da dependência do conhecimento em pormenor da estrutura e funcionamento interno de outros módulos que possam utilizar ou com quem possam interagir.

A título de exemplo, um módulo implementado e designado por "shapes", que combina as formas geométricas de representação standard (triângulos, cones, cilindros, cubos, polígonos, *meshs*, etc), pode ser facilmente substituído por outro

módulo, por exemplo o designado por "nurbs" (*Non Uniform Rational Basis Spline*). Neste último caso, a representação da geometria dos objectos é efectuada através de curvas e superfícies paramétricas. Para um funcionamento correcto do novo módulo, este deve implementar com a mesma interface (nomes e parâmetros das primitivas implementadas).

Ainda é possível misturar diversas representações ou criar um novo conjunto de representações, desde que estas apresentem uma interface semelhante à definida para o nível superior. Desde que implementem a mesma funcionalidade mas como base em primitivas e dados diferentes, o HRA e o *renderer* principal não impõem qualquer outro tipo de condicionalismos ou restrições.

A estrutura da representação dos dados em camadas e a reduzida dimensão das primitivas permitem, com alguma facilidade, transformar e adaptar o código, e as representações para distintos e diversos recursos e plataformas, ou para novos paradigmas no campo do *rendering*.

Num futuro desenvolvimento poder-se-á definir um sistema de inserção dinâmica de módulos, isto é, em tempo de execução, para sistemas de *rendering* em funcionamento ou produção continua (*dynamic linking* ((FSF) 2008)). No sistema presentemente implementado, a inserção e remoção de módulos é feita de uma forma estática, ou seja, é necessário recompilar e ligar (*link*) ((FSF) 2008) o programa em questão sempre que sejam feitas alterações ou modificação dos módulos que o compõem.

4.3 Validação experimental

4.3.1 Metodologia definida para a validação experimental

Com o objectivo de validar o modelo proposto no capítulo 3, através de uma validação experimental das principiais características do referido modelo, foram definidas três tipos de configuração na organização do módulos de software

referidos no ponto 4.2, nomeadamente o ciclo principal do *renderer*, o HRA e o conjunto de primitivas de *rendering* (baixo e alto nível não recursivas).

Nesse sentido foram consideradas três configurações:

- *Renderer* com as primitivas (R+P) – É a configuração normal em qualquer tipo de implementação em que temos um ciclo principal de *rendering* que invoca as primitivas de *rendering*, cabendo ao ciclo principal de *rendering* a gestão quer dos dados que da organização e ordem de processamento;
- *Renderer* com HRA e as primitivas utilizando apenas um só recurso (R+H1+P)– Nesta configuração a componente de gestão da invocação e gestão de processamento são delegadas ao módulo HRA. O ciclo principal apenas produz as unidades de trabalho que são necessárias para a tarefa pretendida. Apenas um só recurso do tipo CPU virtual é utilizado;
- *Renderer* com HRA e primitivas processadas em múltiplos recursos (R+H*+P) – Esta última engloba todos os recursos considerados e respectivas multiplicidade (número de instancias de cada tipo de recurso), o ciclo principal de *rendering* e a gestão efectuada pelo HRA.

Estas configuração foram concebidas, num sentido mais prático, para analisar os resultados dos tempos de síntese da imagem segundo duas vertentes:

- O impacto da introdução do HRA no sistema – Contabilizar o *overhead* devido à introdução da gestão efectuada pelo HRA e a sua interface com o recurso CPU virtual. Neste sentido, foram comparados os resultados das configurações *renderer* com as primitivas, e *renderer* com as primitivas com a gestão dos recursos efectuada pelo HRA;
- O impacto da adição de novos recursos - Quantificar os ganhos obtidos pela introdução de recursos adicionais no processo de *rendering*, em relação à utilização de um único recurso, com uma única instância do

recurso. A configurações utilizada para vertente foi a do *renderer* com primitivas mais HRA, versão utilizando apenas o CPU virtual e versão utilizando o CPU virtual e GPU.

Também foram utilizados dois cenários distintos nas experiências realizadas com características fotométricas e geométricas distintas, mais propriamente a complexidade geométrica, o número de fontes de luz e as componentes de iluminação consideradas. Estas características afectam o comportamento dos algoritmos de *rendering* utilizados [10].

Também por precaução estatística e com forma de minorar a interferência de elementos estranhos ao processo em si, foi considerando para cada configuração e com cada cenário considerado a repetição do processo de síntese da imagem múltiplas vezes nas mesmas aparentes condições, removendo da média aritmética final os tempos as amostras com maior e menor valor. As discrepâncias entre tempos das diversas amostras, por vez elevadas, podem ser derivadas de factores externos ao processo com seja situações causadas pelo acesso e políticas de *cache* do disco, da memória primária ou do CPU, por exemplo. As influências nos tempos de síntese também podem ser causadas pela execução de outros processos alheios às experiências em sistemas operativos multiprogramados, como seja o caso da plataforma utilizada, Linux (distribuição Fedora 9 (Fedora 2008)), mesmo não permitindo ou não habilitando o funcionamento de algumas das características e serviços na configuração de base da referida plataforma.

4.3.2 Cenários utilizados

As imagens da Figura 37 e Figura 38 representam os dois cenários utilizados nas diferentes configurações e experiências definidas nos pontos seguintes.

É de salientar que as imagens resultantes das três configurações, com o mesmo cenário, apresentam um aspecto semelhante em termos visuais, com

[10] O capítulo 2 deve ser consultado no caso de se pretender informação mais detalhada sobre os aspectos referidos e respectivas consequências no processo de *rendering*.

diferenças marginais. Por esse motivo se apresenta apenas uma imagem de cada cenário (A e B) e não todas as imagens obtidas para o mesmo cenário.

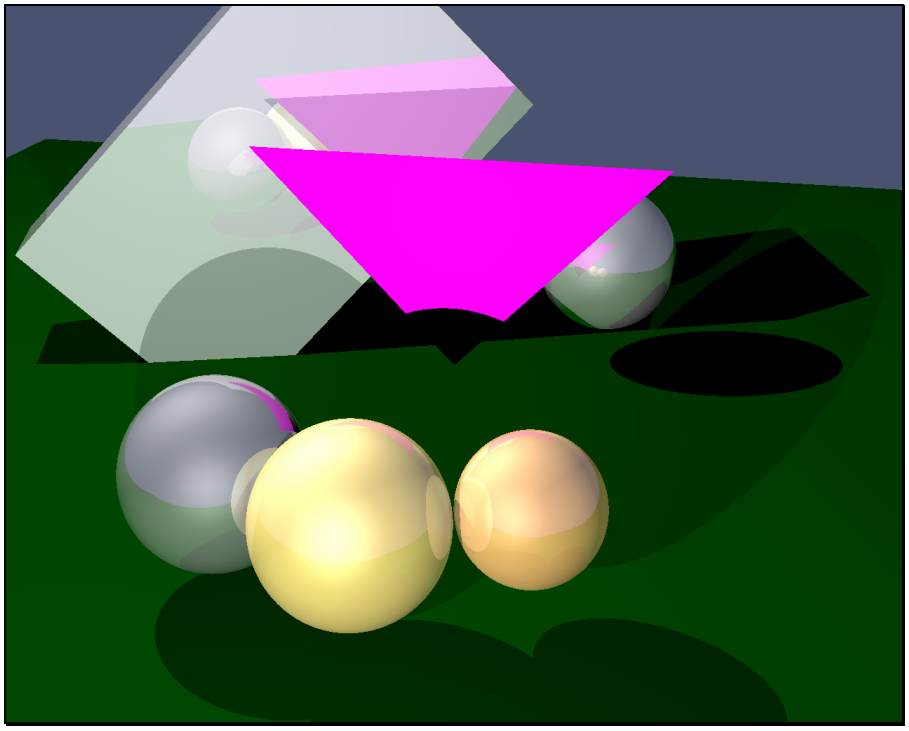

Figura 37 – Cenário A com esferas e pirâmides e inter reflexões (CPU)

As diferenças marginais referidas são devidas principalmente à utilização de valores aleatórios, utilizados em funções de distribuição e de amostragem no algoritmo de *rendering.* Outro possível factor para estas ligeiras diferenças pode ser associado à ordem pela qual os raios secundários são processados nas diferentes configurações e recursos.

Para comparar as imagens de cada cenário nas diversas configurações foi utilizado um método simples e rápido, que poderá ser considerado não totalmente correcto do ponto de vista perceptual (Myszkowski 1998; Yee 2001). Esse método consistiu-se na realização de uma simples diferença entre imagens resultantes, imagem diferencial entre imagens do mesmo cenário, considerando ainda a média e desvio padrão dos *pixeis* de cada uma das imagens comparadas. As diferenças entre *pixeis* nas mesmas coordenadas de duas imagens foram sempre inferior a

0,8% (2/255), na comparação directa da mesma componente de cada cor de cada *pixel*.

Foi considerado não recorrer a métodos mais complexos, mas provavelmente mais correctos em termos perceptuais, como seja o caso do método definido por Myszkowski et al. (Myszkowski 1998) ou o método indirecto utilizando por Yee et al. (Yee 2001), para determinar se as imagens obtidas são semelhantes.

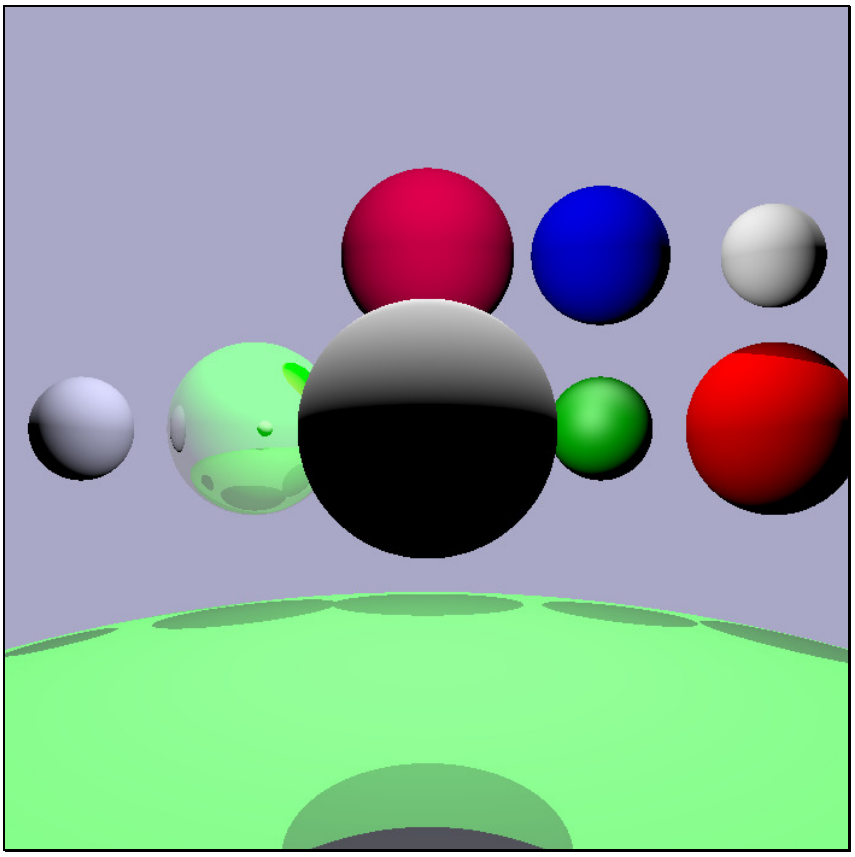

Figura 38 – Cenário B com esferas e inter reflexões (CPU+GPU)

De salientar que quando se refere o mesmo cenário, refere-se que se considerou as mesmas condições de *rendering*, como seja o caso do ponto de vista e outros parâmetros iniciais definidos para as experiências realizadas, a mesma descrição geométrica e fotométrica. As condições distintas entre configurações são relativas apenas a diferentes formas de funcionamento no sistema de *rendering*, nomeadamente das componentes de software utilizadas e possível ordem pela qual

os cálculos da síntese de imagem são realizados (devido a prioridades no cálculo e recursos de hardware disponíveis).

4.3.3 Plataformas utilizadas

Para realizar o conjunto de testes com vista à obtenção de resultados foram consideradas as três configurações do *renderer*, mais propriamente das componentes de software e recursos de processamento utilizados. Estas configurações foram testadas em duas plataformas de hardware sintetizadas na Tabela 1. Para informações mais detalhas sobre os equipamentos utilizados deve ser consultado o apêndice A.

Plataforma 1	Pentium 4 3GHz / 1GiB RAM (DDR1-400MHz) Placa Gráfica XFX 6600GT / AGP
Plataforma 2	Pentium 4 3GHz / 1GiB RAM (DDR2-533MHz) Placa Gráfica XFX 8600GT / PCI Express

Tabela 1 – Configuração das plataformas (hardware) utilizadas

Uma breve descrição das configurações consideradas e os principais objectivos da comparação de resultados entre configurações no que respeita a síntese de uma imagem, já foram sumariamente referidas no ponto 4.3.1.

Os pontos seguintes apresentam em maior detalhe as características das configurações consideradas, envolvendo desde aspectos de âmbito genérico como seja a interacção entre camadas de software, ou questões mais práticas como seja o paralelismo implícito.

4.4 Configurações definidas

Relativamente a organização e utilizado do módulos de software definidos, forma consideradas configurações distintas. Os módulos de software considerados foram o ciclo principal de *rendering* (designado por *renderer*), o HRA e as

primitivas de *rendering*, considerando neste ultimo módulo o conjunto das primitivas de baixo nível e alto nível não recursivas.

A Tabela 2 apresenta as parâmetros gerais utilizados nas experiências das diversas configurações para a síntese das imagens no *renderer* implementado com vista a obtenção de tempos relacionados com o processo de *rendering*.

Configuração	
Número de raios	100000000
Número máximo de *bounces*	5
Cenário	A e B

Tabela 2 – Configurações gerais para as experiências com os cenários A e B

Relativamente a apresentação dos dados obtidos em cada configuração e a respectiva configuração, expostas nos seguintes pontos, esta estruturada com inicio no fluxo e interconexão entre as distintas camadas de software, seguido pelo resumo dos dados dependes obtidos, como seja média e desvio padrão, finalizando por uma tabela geral dos dados directamente obtidos nas experiências, incluídos os dados dependentes anteriormente referidos.

4.4.1 Configuração do *renderer* só no CPU sem HRA (R+P)

A primeira configuração experimental envolveu apenas a utilização do ciclo principal de *rendering* em conjunção com as primitivas de *rendering*. O ciclo principal de *rendering* efectua a invocação directa das primitivas de uma forma sequêncial sem recorrer ao HRA. O *renderer* utiliza apenas um único *thread* (Butenhof 1997) e o CPU como elemento único de processamento de dados. A interacção entre níveis para esta configuração, mais propriamente entre primitivas e ciclo principal de *rendering*, é apresentada na Figura 39.

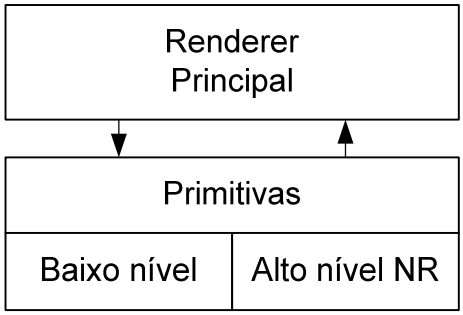

Figura 39 – Interacção entre níveis na configuração CPU sem HRA

Esta configuração não implica qualquer tipo de paralelismo, explícito ou directamente implícito. Poderá no entanto ocorrer algum paralelismo não directamente explicito, nomeadamente ao nível da execução das instruções no CPU, ou entre o sistema, a memória e o CPU. Uma das precauções consideradas para tentar evitar algum paralelismo no sistema, foi o de não habilitar a opção de Hyper-Threading™ da Intel© (IntelCorporation 2003) na BIOS (Basic Input/Ouput System) (Wikipedia 2008) dos equipamentos utilizados. Esta tecnologia efectua uma reorganização das instruções ao nível do CPU por forma a utilizar de uma forma mais eficiente as diversas unidades funcionais internas do CPU, mais propriamente do Pentium™ 4 da Intel© (IntelCorporation 2003). Esta tecnologia faz com que o sistema apresente com dois CPU (virtuais) ao nível do sistema operativo e aplicações.

Nesta primeira configuração, o ciclo principal de *rendering* invoca directamente as primitivas necessárias à síntese do cenário seleccionado, as primitivas de baixo e alto nível sem recursividade, designadas de Alto nível NR na figura anterior. As referidas primitivas pertencem ao conjunto que será utilizado no contentor do HRA, na configuração seguinte, CPU com HRA. A componente de gestão do *rendering* e a recursividade é controlada inteiramente pelo ciclo principal de *rendering*, cabendo a este decidir como a síntese da imagem é efectuada e a ordem pela qual os cálculos deverão ser efectuados.

A Tabela 3 apresenta de uma forma resumida os parâmetros definidos para a realização deste tipo de experiência, considerando os valores de média sem o

máximo e o mínimo tempo obtido, para as duas plataformas consideradas e dois cenários utilizados.

CPU sem HRA	**Dados**	
	Cenário A	**Cenário B**
Número de raios	100000000	100000000
Máximo de *bounces*	5	5
Recursos	CPU	CPU
Média (sem max e min)		
Plataforma 1	**201,13**	**166,81**
Plataforma 2	**154,99**	**127,63**

Tabela 3 – Quadro resumo dos resultados "CPU sem HRA"

CPU sem HRA PC1	**Tempo de síntese (s)**	
	Cenário A	**Cenário B**
Experiência nº 1	221,42	176,11
Experiência nº 2	201,53	163,81
Experiência nº 3	198,35	167,29
Experiência nº 4	200,33	168,10
Experiência nº 5	203,75	165,21
Experiência nº 6	201,95	168,39
Experiência nº 7	199,44	162,44
Experiência nº 8	200,03	167,32
Experiência nº 9	199,18	168,17
Experiência nº 10	202,81	166,18
Média	***202,88***	***167,30***
Desvio Padrão	***6,39***	***3,49***
Média (sem max e min)	***201,13***	***166,81***

Tabela 4 – Quadro de resultados "CPU sem HRA" (plataforma 1)

Já as Tabela 4 e Tabela 5 apresentam os valores médios do tempo de *rendering* para os dois cenários e duas plataforma utilizadas, seguindo a

metodologia definida, bem como valores individuais de cada experiências realizada, no inicio da tabela, bem como o desvio padrão obtido de todos os valores medidos.

CPU sem HRA PC2	Tempo de síntese (s)	
	Cenário A	Cenário B
Experiência nº 1	164,25	130,44
Experiência nº 2	156,12	127,11
Experiência nº 3	156,15	128,99
Experiência nº 4	154,22	127,64
Experiência nº 5	155,47	127,01
Experiência nº 6	153,93	126,87
Experiência nº 7	155,84	128,29
Experiência nº 8	154,10	127,66
Experiência nº 9	153,18	127,98
Experiência nº 10	154,11	126,76
Média	***155,74***	***127,88***
Desvio Padrão	***3,01***	***1,08***
Média (sem max e min)	***154,99***	***127,69***

Tabela 5 – Quadro de resultados "CPU sem HRA" (plataforma 2)

A principal medição efectuada nesta configuração foi o tempo global decorrido na síntese da imagem, isto é, desde a invocação do *renderer* até ao finalizar do processo de síntese da imagem, incluindo guardar a imagem em suporte secundário (disco).

4.4.2 Configuração CPU com HRA (R+H1+P)

Nesta configuração, o HRA é introduzido entre o ciclo principal de *rendering* e as primitivas necessárias para a síntese da imagem final. O *renderer* produz as unidades de trabalho e em vez de invocar directamente as primitivas, estas são colocadas no contentor de UT para que o HRA faça a sua gestão, sendo que a

invocação das primitivas efectuada pelo HRA. O sentido de invocação e recolha de resultados está representada na Figura 40.

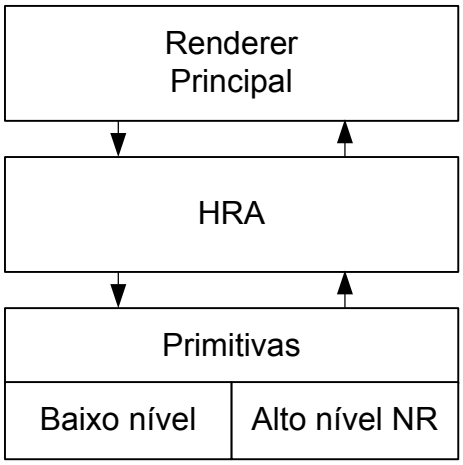

Figura 40 – Interacção entre níveis na configuração CPU com HRA

Nesta configuração o HRA foi utilizado apenas com um único recurso de processamento, o CPU, definido como CPU virtual que utiliza um *thread* próprio para calcular os resultados das unidades de trabalho submetidas e posteriormente recolhidos pelo ciclo principal do *renderer* no contentor de resultados.

Nesta configuração foram limitados os artifícios ou facilidades do sistema que potenciam o paralelismo entre *renderer*, o HRA e o processamento das unidades de trabalho, como seja o caso do Hyper-Threading™ da Intel© (IntelCorporation 2003), que não foi habilitado como na configuração anterior.

Em termos de funcionamento existem três *threads*: Um para o ciclo principal de *rendering*, um para o HRA e outro para o CPU virtual. Esta configuração envolve paralelismo explicito e obrigatório entre os três componentes considerados.

A Tabela 6 apresenta os valores médios do tempo de *rendering* e características utilizadas na configuração descrita, conforme a metodologia definida, paras duas plataformas e dois cenários considerados.

CPU com HRA	Dados	
	Cenário A	Cenário B
Número de raios	100000000	100000000
Máximo de *bounces*	5	5
Recursos	CPU	CPU
Média (sem max e min)		
Plataforma 1	**234,73**	**188,71**
Plataforma 2	**172,99**	**149,40**

Tabela 6 – Quadro resumo dos resultados "CPU com HRA"

Mesmo tentando evitar o paralelismo explicitamente dentro dos componentes e entre eles, este ocorre pois os mais recentes sistemas operativos efectuam a gestão de processos, neste caso dos *threads*, de forma a maximizar a utilização eficiente dos recursos do sistema, sejam entradas e saídas de hardware, a memória e o CPU, o que leva que as aplicações usufruam deste paralelismo intrínseco, quer internamente à aplicação ou entre aplicações ou dispositivos utilizados.

Existe nesta configuração, mesmo não sendo pretendido, três elementos que podem funcionar potencialmente em paralelo, ainda que existam algumas dependências entre eles. O *renderer* necessita de produzir unidades de trabalho para o HRA, que por sua vez as envia para o CPU virtual.

As Tabela 7 e Tabela 8 apresenta os valores médios do tempo de *rendering* e características utilizadas na configuração descrita para os cenários considerados e plataforma 1 e 2 respectivamente, seguindo a metodologia definida. Os valores das experiências individuais para esta configuração podem ser observados no inicio da tabela.

CPU com HRA PC1	Tempo de síntese (s)	
	Cenário A	**Cenário B**
Experiência nº 1	254,35	192,19
Experiência nº 2	221,05	188,15
Experiência nº 3	219,19	190,91
Experiência nº 4	222,54	185,75
Experiência nº 5	224,45	189,15
Experiência nº 6	246,13	187,23
Experiência nº 7	239,51	188,99
Experiência nº 8	240,23	189,39
Experiência nº 9	241,77	188,76
Experiência nº 10	242,17	187,07
Média	***235,14***	***188,76***
Desvio Padrão	***11,64***	***1,78***
Média (sem max e min)	***234,73***	***188,71***

Tabela 7 – Quadro de resultados "CPU com HRA" (plataforma 1)

CPU com HRA PC1	Tempo de síntese (s)	
	Cenário A	**Cenário B**
Experiência nº 1	176,21	151,91
Experiência nº 2	173,15	150,98
Experiência nº 3	172,91	149,76
Experiência nº 4	172,49	149,71
Experiência nº 5	174,77	148,98
Experiência nº 6	171,03	149,21
Experiência nº 7	170,91	148,79
Experiência nº 8	171,53	149,13
Experiência nº 9	172,11	148,62
Experiência nº 10	173,79	148,10
Média	***172,89***	***149,52***
Desvio Padrão	***1,59***	***1,09***
Média (sem max e min)	***172,72***	***149,40***

Tabela 8 – Quadro de resultados "CPU com HRA" (plataforma 2)

4.4.3 Configuração CPU e GPU com HRA (R+H*+P)

Na terceira configuração, para além do ciclo principal de *rendering*, da inclusão do HRA e da utilização do CPU virtual como recurso de processamento, foi adicionado o GPU como segundo recurso de processamento. Este alinhamento pode ser observado na Figura 41, que indica também a sequência de operações efectuadas nesta configuração.

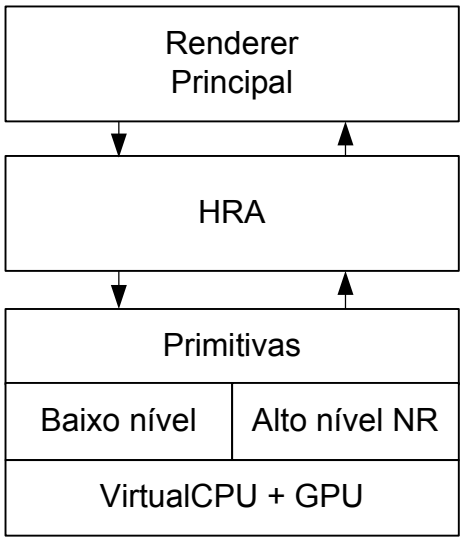

Figura 41 – Interacção entre níveis na configuração CPU + GPU com HRA

Nesta configuração estão em funcionamento três *threads* no CPU para além dos elementos presentes no GPU, sendo estes últimos elementos controlados pelo HRA. Nesta situação é difícil evitar o paralelismo, pois basta que o GPU processe unidades de trabalho em simultâneo com outras operações que o CPU possa estar a realizar, quer seja no *thread* do *renderer*, no *thread* do HRA ou no *thread* atribuído CPU virtual. Definir um mecanismo para impedir este tipo de paralelismo é contrário ao princípio definido e mecânica pretendida, ou seja utilizar múltiplos recursos que contribuem para um resultado comum, com os recursos em funcionamento simultâneo.

A Tabela 9 apresenta os valores médios do tempo de *rendering*, as características e parâmetros utilizados nesta última configuração. A metodologia seguida foi a mesma das configurações anteriores.

CPU + GPU com HRA	**Dados**	
	Cenário A	**Cenário B**
Número de raios	100000000	100000000
Máximo de *bounces*	5	5
Recursos	CPU + GPU	CPU + GPU
Média (sem max e min)		
Plataforma 1	**163,19**	**139,53**
Plataforma 2	**123,00**	**95,37**

Tabela 9 – Quadro resumo dos resultados "CPU + GPU com HRA"

CPU com HRA PC1+GPU1	**Tempo de síntese (s)**	
	Cenário A	**Cenário B**
Experiência nº 1	181,27	153,11
Experiência nº 2	162,13	140,12
Experiência nº 3	164,97	139,18
Experiência nº 4	160,64	140,36
Experiência nº 5	160,53	138,53
Experiência nº 6	162,74	139,85
Experiência nº 7	166,11	138,87
Experiência nº 8	161,28	139,21
Experiência nº 9	163,93	140,11
Experiência nº 10	163,75	138,15
Média	***164,74***	***140,75***
Desvio Padrão	***5,78***	***4,18***
Média (sem max e min)	***163,19***	***139,53***

Tabela 10 – Quadro de resultados "CPU com HRA incluindo GPU" (plataforma 1)

As Tabela 10 e Tabela 11 apresentam os valores médios do tempo de *rendering* para os dois cenários e duas plataforma utilizadas, seguindo a metodologia definida, bem como valores individuais de cada experiência realizada.

CPU com HRA PC2+GPU2	**Tempo de síntese (s)**	
	Cenário A	**Cenário B**
Experiência nº 1	124,10	97,22
Experiência nº 2	119,13	94,11
Experiência nº 3	122,92	96,11
Experiência nº 4	123,14	95,54
Experiência nº 5	122,85	95,80
Experiência nº 6	124,18	95,12
Experiência nº 7	123,10	94,83
Experiência nº 8	121,98	95,12
Experiência nº 9	122,36	94,78
Experiência nº 10	123,52	95,69
Média	***122,73***	***95,43***
Desvio Padrão	***1,36***	***0,81***
Média (sem max e min)	***123,00***	***95,37***

Tabela 11 – Quadro de resultados "CPU com HRA incluindo GPU" (plataforma 2)

4.4.4 Configuração GPU com HRA (R+H.GPU+P)

Esta última configuração não foi considerada pois não representa nem pode avaliar o objectivo principal que é pretendido com o modelo proposto no capítulo 3 da presente tese. No entanto, não deixa de ser uma configuração possível. Por exemplo, caso se pretenda implementar um sistema específico de *rendering* com base no modelo proposto, em que o CPU é utilizado apenas para o ciclo principal do *renderer* e para o HRA, sendo as unidades de trabalho processadas apenas num conjunto de placas de processamento de elevado poder de cálculo, por exemplo.

4.5 Análise aos resultados das experiências realizadas

O conjunto de resultados obtidos das experiências efectuadas estão resumidos nas Tabela 12 e Tabela 13. Estes resultados foram também parcialmente apresentados na descrição de cada configuração definida anteriormente.

A Tabela 12 apresenta o conjunto de valores das diferentes configurações obtidos após a realização das experiências tendo em conta a metodologia definida, mais especificamente para o cenário A na plataforma 1.

PC1+GPU1	**Tempo de síntese (s)**		
	CPU sem HRA	**CPU com HRA**	**CPU+GPU c/ HRA**
Número de raios	100000000	100000000	100000000
Máximo de *bounces*	5	5	5
Número de recursos	1	1	2
Cenário A			
Média (sem max e min)	201,13	234,73	163,19
Valor relativo	**100%**	**117%**	**81%**
Cenário B			
Média - max - min	166,81	188,71	139,53
Valor relativo	**100%**	**113%**	**84%**

Tabela 12 – Comparação das diferentes configurações para o cenário A e B (plataforma 1)

Do conjunto das duas tabelas é possível retirar algumas conclusões, mais propriamente na comparação dos tempos médios da síntese "da mesma imagem", apresentados e calculados segundo a metodologia indicada no ponto 4.3.1.

Também é de salientar que os parâmetros iniciais definidos em cada configuração foram iguais, como se pode observar pelas Tabela 12 e Tabela 13, excepto como é obvio no número de recursos de processamento utilizados na

configuração "CPU+GPU com HRA". O que se pretendeu foi medir o tempo utilizado para a síntese da imagem cada configuração com o mínimo ou praticamente nenhuma alteração possível nos parâmetros que poderiam influenciar, directamente ou indirectamente, o tempo global de síntese da imagem.

PC2+GPU2	**Tempo de síntese (s)**		
	CPU sem HRA	**CPU com HRA**	**CPU+GPU c/ HRA**
Número de raios	100000000	100000000	100000000
Máximo de *bounces*	5	5	5
Número de recursos	1	1	2
Cenário A			
Média (sem max e min)	154,99	172,72	123,00
Valor relativo	**100%**	**111%**	**79%**
Cenário B			
Média - max - min	127,69	149,40	95,37
Valor relativo	**100%**	**117%**	**75%**

Tabela 13 – Comparação das diferentes configurações para o cenário A e B (plataforma 2)

4.5.1 Comparação entre CPU com e sem HRA

Uma comparação entre os valores obtidos na configuração "CPU sem HRA" e "CPU com HRA", tendo como base o tempo da primeira configuração, mostrar que a introdução do HRA no sistema introduziu um *overhead* entre 13 e 17%. Este aumento do tempo de síntese pode ser explicado pela introdução de uma camada extra de software entre o ciclo principal do render e as primitivas, utilizando o mesmo recurso.

Em vez das primitivas serem invocadas directamente pelo *renderer*, na configuração de base "CPU sem HRA", as primitivas e os respectivos dados são colocados no contentor de UT, que por sua vez são atribuídas pelo HRA ao recurso

denominado de CPU virtual. Este procedimento só por si adiciona um processamento adicional ao sistema, quando limitado o paralelismo no sistema.

Com a evolução dos CPUs para múltiplos cores (AMD 2008; IntelCorporation 2008) e com suporte para tecnologias como Hyper-Threading™ da Intel© (IntelCorporation 2003), é possível utilizar vários CPUs virtuais em funcionamento em paralelo, com apenas uma simples alteração de um parâmetro, o número de CPU virtuais que se pretende utilizar para processamento. Como foi referido em múltiplas ocasiões este parâmetro poderá ser configurado dinamicamente ao longo do processo de síntese em futuras implementações do modelo, recorrendo a *profiling* do poder de processamento em cada momento. Para criar um novo CPU virtual basta iniciar o novo *thread*, com a função inicial "VirtualCPU()" definida.

4.5.2 Comparação HRA com um ou dois recursos

Numa segunda perspectiva, a introdução de um segundo elemento de processamento de UT, na configuração "CPU+GPU com HRA", dotou o sistema global de maior poder de processamento o que se traduz numa redução do tempo total de síntese entre 16 a 19% em relação ao tempo de base da configuração "CPU sem HRA", ou entre de 29 e 36 pontos de base (113%-84% e 117%-81%).

Numa comparação directa de tempos entre a configuração "CPU com HRA" e "CPU+GPU com HRA" as reduções são da ordem dos 24% e 30% do tempo de síntese.

Neste caso, devido ao paralelismo entre o CPU e o GPU, foi possível realizar os cálculos, ou aproximadamente os mesmos cálculos, num tempo menor. Nesta comparação apenas foi introduzido um novo tipo de recurso e apenas uma unidade deste tipo de recurso, o GPU. A adição de múltiplos recursos do mesmo tipo, como por exemplo o GPU (Nvidia 2008; Nvidia 2008), ou outro tipo de recurso como seja o caso de placas auxiliares de processamento, pode aumentar a eficiência do sistema significativamente.

Esta adaptação de escala em termos de recursos só poderá ser comprovada recorrendo a equipamentos com as características anteriormente enunciadas. No entanto a estrutura modular e a grande independência de dados habituais no processo de *rendering*, expostos pelo modelo proposto, sugere que essa adaptação e experimentação pode levar a resultados bastantes positivos.

4.6 Conclusões

Neste capítulo foi apresentado o trabalho experimental realizado no sentido de corroborar a tese apresentada para o modelo proposto. Foi possível ilustrar a forma como a introdução de uma camada abstracta entre a nível físico de hardware e o nível lógico do programa melhora a organização e eficiência do algoritmo de *rendering* em ambientes como aptidão para o paralelismo.

A implementação efectuada do modelo proposto no capítulo 3 teve em conta os elementos fundamentais do modelo, não tendo em conta aspectos relacionados com optimizações ao nível do hardware ou ao nível do algoritmo de gestão de recursos e do contentor de UT. Estes e outros itens podem e poderão ser estudados e implementados de forma a criar um *renderer* multidimensional, no sentido dos recursos, com uma eficiência ainda superior a obtida na implementação efectuada. É de salientar que algumas das optimizações referidas, nomeadamente ao nível das primitivas de *rendering* e do hardware, não implicam necessariamente uma alteração dos dados comparativos obtidos entre diferentes configurações das experiências realizadas, pois estas optimizações reflectem-se de igual modo em qualquer uma das configurações consideradas.

5 CONCLUSÕES E TRABALHO FUTURO

O modelo proposto no capítulo 3 pretende responder às questões de portabilidade, expansão e gestão eficiente dos recursos para um *renderer* multidimensional, na perspectiva dos recursos disponíveis num sistema computacional standard. O modelo proposto também apresenta uma distanciação e maior abstracção em relação a outras abordagens mais específicas e intrinsecamente integradas com os recursos utilizados. Nessa perspectiva, o modelo segue o caminho da definição de camadas de funcionalidade e respectiva abstracção entre níveis.

Com base nas características e premissas enunciadas no capítulo 3 - Modelo Proposto e no exposto no capítulo 4 - Trabalho experimental foram identificadas alguns características do modelo proposto que importa realçar:

- Portabilidade – O modelo proporciona várias características a diversos níveis neste aspecto, a portabilidade. De salientar a reduzida dimensão das primitivas básicas que compõem parcialmente as unidades de trabalho, proporcionando uma fácil adaptação da sua implementação

noutros equipamentos, mesmo utilizando paradigmas de programação distintos.

- Implementação parcial - Outra característica definida no modelo é possibilidade de uma implementação parcial das primitivas em novos recursos. Novos equipamentos poderão ser incluídos na plataforma sem necessidade de uma implementação completa das primitivas de base. Esta característica é vantajosa do ponto de vista económico e de desenvolvimento faseado. É possível utilizar desde logo um equipamento, implementando apenas algumas primitivas e meios de comunicação como o HRA. Uma implementação parcial poderá, como foi referido no capítulo 3, ter reflexos na distribuição de carga/balanceamento no sistema.
- Balanceamento de trabalho – Na presença de múltiplos elementos de processamento, num ambiente paralelo, existe a necessidade de distribuir o trabalho requerido por todos elementos disponíveis (Chalmers 2002; Plachetka 2002; Plachetka 2002). O modelo proposto no capítulo 3 aborda esta condicionante em múltiplos aspectos, desde a dimensão reduzida das primitivas, à utilização de *caching* local, *pipeline* nas comunicações, processamento fora de ordem, entre outros;
- Paralelismo intrínseco – A divisão em módulos do sistema e funcionamento semi-independente entre eles favorece a eficiência global, nomeadamente devido às tendências dos sistemas operativos modernos (cada vez como um maior suporte para múltiplos processos), na absorção de tempos de acessos a dispositivos e memórias, mas também ao paralelismo emergente dos recursos (como seja CPUs com Hiper-Treading™ e multicore, GPUs massivamente paralelos, acesso concorrente à memória, entre outros);

- Definição de uma API standard de primitivas – A estrutura e divisão modular dos elementos e primitivas de *rendering*, fomentam a interoperabilidade e expansão do sistema. Será fácil acrescentar novos níveis de paralelismo, como por exemplo a utilização de *clusters* de processamento com múltiplos computadores (e.g. CPU virtual remota);
- Optimizações de código – Com a independência de código, utilizando versões especializadas das primitivas para cada tipo de recurso e com a reduzida dimensão das primitivas, é possível realizar optimizações ao nível do código e alinhamento local dos dados para obter uma melhor performance do sistema global.

O trabalho desenvolvido nesta tese sugere que, através da inclusão de um elemento de abstracção entre o ciclo principal de *rendering* e um conjunto de recursos heterogéneo disponíveis, é possível reduzir o tempo total para a síntese de uma imagem. O modelo proposto apresenta algumas vantagens sobre outras abordagens mais rígidas mas com o mesmo objectivo. Um aspecto importante do sistema é a sua independência de hardware específico, portabilidade, especificada no seu desenho e derivada da divisão do processo de *rendering* em três níveis com unidades de trabalho pequenas e simples. A interoperabilidade entre diferentes tipos de hardware é importante porque não limita à utilização de um sistema semelhante numa variedade de plataformas. A reduzida dimensão das primitivas definidas e a implementação parcial do conjunto de primitivas existentes é muito útil para teste e implementação em novos tipos de hardware e potencialmente mais poderosos. Desta forma, o ciclo de desenvolvimento e teste pode ser substancialmente simplificado e reduzido.

A introdução do HRA não se limita a gerir a distribuição de trabalho entre os recursos disponíveis, mas também apresenta benefícios ao nível da optimização de código, reorganização da ordem da execução de primitivas e melhoria das estratégias de *cache* local e global.

5.1 Trabalho futuro

Dentro da evolução futura do modelo, provavelmente a questão mais referida ao longo do presente texto é relativa à adaptação dinâmica de múltiplas características ou parâmetros do modelo proposto. Alguns módulos ou camadas do sistema são passíveis de ser desenhados por forma a apresentar um comportamento dinâmico, adaptando-se as suas características ao estado do processo de *rendering* e comportamento do hardware.

Um exemplo ilustrativo desta abordagem é a dimensão do contentor de unidades de trabalho. Uma adaptação do número de UT pendentes pode influenciar duas características complementares. A eficiência derivada do agrupamento e reordenação das UT em espera e a inactividade do sistema. Uma maior dimensão do contentor pode implicar mais tempo de espera para as UT pendentes, enquanto uma menor dimensão pode levar a um processamento mais rápidos das UT em fila de espera. O sistema poderá tentar beneficiar a eficiência ou a interactividade dinamicamente, e desta forma cobrir os campos das plataformas de *rendering* interactivo ao *rendering* de pormenor em plataformas *off-line*.

Outros aspectos desta da abordagem, como o *profiling*, deverão ser objecto de trabalho futuro, como o objectivo de melhorar aspectos como sejam as comunicações ou optimização parcial de código, organização de dados, entre outros.

REFERÊNCIAS

(FSF), F. S. F. (2008). "GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)." Retrieved 2208-10-13, 2008, from http://gcc.gnu.org/.

Adelson, E. H. (2000). Lightness Perception and Lightness Illusions. The New Cognitive Neurosciences**:** 339-351.

AMD. (2008). "Quad-Core AMD Processors - AMD Opteron™ & AMD Phenom™." Retrieved 2008-10-13, from http://multicore.amd.com/us-en/AMD-Multi-Core.aspx.

ArchitectureWeek. (2002). "ArchitectureWeek - Tools - Rendering View by View." Retrieved 2008-10-13, 2008, from http://www.architectureweek.com/2002/1113/tools_2-2.html.

Arlow, J. N., Ila (2005). UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design, Addison-Wesley Professional.

Backus, J. (1978). "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs." Commun. ACM **21**(8): 613-641.

Benthin, C. D., Tim Wald, Ingo Slusallek, Philipp (2002). Interactive Headlight Simulation – A Case Study of Interactive Distributed Ray Tracing. Fourth Eurographics Workshop on Parallel Graphics and Visualization.

Benthin, C. W., I. Scherbaum, M. Friedrich, H. (2006). "Ray Tracing on the Cell Processor." Interactive Ray Tracing 2006, IEEE Symposium on: 15-23.

Boosten, M. (2003). Fine-Grain Parallel Processing on a Commodity Platform: a Solution for the ATLAS Second Level Trigger. **Ph.D.**

Buck, Y. F., Tim Horn, Daniel Sugerman, Jeremy Fatahalian, Kayvon Houston, Mike Hanrahan, Pat (2004). "Brook for GPUs: stream computing on graphics hardware." SIGGRAPH '04: ACM SIGGRAPH 2004 Papers: 777-786.

Butenhof, D. R. (1997). Programming with POSIX threads. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Camea. (2008). "Camea – signal and image processing." Retrieved 2008-10-13, 2008, from http://www.camea.cz/en/products/technologies/computation-accelerators/.

Cater, K. C. A. W., G. (2003). Detail to Attention: Exploiting Visual Tasks for Selective Rendering.

CGSociety. (2005). "CGSociety - CG Retrospective 2005." Retrieved 2008-10-13, 2008, from http://features.cgsociety.org/story_custom.php?story_id=3306.

CGSociety. (2008). "Welcome to The Computer Graphics Society." Retrieved 2008-10-13, 2008, from http://www.cgsociety.org/.

Chalmers, A. D., Timothy Reinhard, Erik (2002). Practical Parallel Rendering AK Peters Ltd.

Chen, H. C., Yuqun Finkelstein, Adam Funkhouser, Thomas Li, Kai Liu, Zhiyan Samanta,Rudrajit Wallace, Grant (2001). "Data Distribution Strategies for High-Resolution Displays." Computers & Graphics **25**: 811--818.

Christensen, P. H. F., J. Laur, D.M. Batali, D. (2006). "Ray Tracing for the Movie `Cars'." Symposium on Interactive Ray Tracing **0**: 1-6.

CIO. (2006). "Cray to Unite Scalar, Vector Computer Architectures." Retrieved 2008-10-13, 2008, from http://www.cio.com/article/19292/Cray_to_Unite_Scalar_Vector_Computer_Architectures.

Cohen, M. F., Wallace, John Hanrahan, Pat (1993). Radiosity and realistic image synthesis. San Diego, CA, USA, Academic Press Professional, Inc.

David Coulthurst, P. D., Kurt Debattista, Simon McIntosh-Smith and Alan Chalmers (2008). Parallel Path Tracing using Incoherent Path-Atom Binning. Spring Conference on Computer Graphics 2008.

Debattista, K. S., Veronica Pereira, Francisco Chalmers, Alan (2005). "Selective Parallel Rendering for High-Fidelity Graphics." Proceedings of Theory and Practice of Computer Graphics 2005: 59-66.

Dietrich, A. W., Ingo Benthin, Carsten Slusallek, Philipp (2003). The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. OpenSG Symposium.

Dmitriev, K. B., Stefan Myszkowski, Karol Seidel, Hans-Peter (2002). Interactive Global Illumination Using Selective Photon Tracing. Thirteenth Eurographics Workshop on Rendering.

Dumont, R. P., Fabio Ferwerda, James A. (2002). "Perceptually-Driven Decision Theory for Interactive Realistic Rendering." ACM Transactions on Graphics **22**: 152--181.

Duncan, R. (1990). "A Survey of Parallel Computer Architectures." Computer **23**(2): 5-16.

EA (2008). "EA Portugal - FIFA 09, Need for Speed Undercover, Spore, Skate 2."

EastmanKodakCompany (1999). Reference Input Medium Metric RGB Color Encoding (RIMM RGB) White Paper.

Fedora. (2008). "Fedora Project." Retrieved 2008-10-13, from http://fedoraproject.org/.

Gonzalez, R. C. W., Richard E. (2006). Digital Image Processing (3rd Edition). Upper Saddle River, NJ, USA, Prentice-Hall, Inc.

Günther, J. W., Ingo Slusallek, Philipp (2004). Realtime Caustics Using Distributed Photon Mapping. Eurographics Symposium on Rendering.

Gutierrez, M., Thalmann, D Vexo, F (2004). "Augmented CGI Films." Second International Conference on Intelligent Access of Multimedia Documents: 53-64.

Halfhill, T. (2008). "Parallel Processing with CUDA." Microprocessor Journal.

Hauta-Kasari, M. P., J. Jaaskelainen, T. Lenz, R. (1988). "Spectral based Analysis of Color Images." Retrieved 2008-10-13, 2008, from http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel2/220/1164/00028405.pdf?arnumber=28405.

Hofstee, H. P. (2005). "Power Efficient Processor Architecture and The Cell Processor." HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture: 258-262.

Hutchins, E. A. A., Brian K. (2007). "US Patent 7199799 - Interleaving of pixels for low power programmable processor." Retrieved 2008-10-13, 2008, from http://www.patentstorm.us/patents/7199799/description.html.

IEEE. (1985). "IEEE 754: Standard for Binary Floating-Point Arithmetic." Retrieved 2008-10-13, 2008, from http://grouper.ieee.org/groups/754/.

IntelCorporation. (2003). "Hyper-Threading Technology." Retrieved 2008-10-13, 2008, from http://www.intel.com/technology/platform-technology/hyper-threading/.

IntelCorporation (2006). Extending the World's Most Popular Processor Architecture.

IntelCorporation (2008) "Intel® Core™2 Extreme Processor QX9000 Series, Intel® Core™2 Quad Processor Q9000, Q9000S, Q8000, and Q8000S Series." **Volume**, DOI:

Jansen, F. W., Reinhard, Erik (1998). "Data Locality in Parallel Rendering." Second Eurographics Workshop on Parallel Graphics and Visualisation: 1-15.

Jensen, H. W. (2001). Realistic Image Synthesis Using Photon Mapping, AK Peters, Ltd.

Kajiya, J. T. (1986). "The rendering equation." SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques **20**(4): 143-150.

Koholka, R. M., Heinz Goller, Alois (1999). MPI-parallelized Radiance on SGI CoW and SMP. Parallel Computation, 4th International ACPC Conference, Salzburg, Austria.

Kuchkuda, R. (1988). "An Introduction to Ray Tracing." Theoretical Foundations of Computer Graphics and CAD: 1039-1060.

Kurenov, S. N. P., Sukitti Kim, Minho Peters, Jorg Cendan, Juan C. (2006). "Simulation for Training with the Autosuture Endo Stitch Device." Surgical Innovation **13**(4): 1-5.

Longhurst, P., K. Debattista, et al. (2005). Snapshot: A rapid technique for driving a selective global illumination renderer. WSCG 2005.

Luebke, D. P. H., Benjamin (2001). "Perceptually-Driven Simplification for Interactive Rendering." Rendering Techniques: 223-234.

Myszkowski, K. (1998). The Visible Differences Predictor: applications to global illumination problems. Eurographics Workshop on Rendering Techniques, G. Drettakis and N. Max, Eds.

Nvidia. (2008). "Download CUDA Code - complete toolkit, free of charge, for creating derivative works in computing, whether it is academic, commercial, or personal." Retrieved 2008-10-13, 2008, from http://www.nvidia.com/object/cuda_get.html.

Nvidia. (2008). "GeForce 8 Series." from http://www.nvidia.com/page/geforce8.html.

Nvidia. (2008). "GeForce 8800." Retrieved 2008-10-13, 2008, from http://www.nvidia.com/page/geforce_8800.html.

Nvidia. (2008, 2008/05/09). "NVIDIA GeForce 9800 GX2." Retrieved 2008-10-13, 2008, from http://www.nvidia.com/object/geforce_9800gx2.html.

Nvidia. (2008). "NVIDIA Scene Graph Programming Library and SDK." Retrieved 2008-10-13, 2008, from http://developer.nvidia.com/object/nvsg_home.html.

Nvidia. (2008). "Quadro Plex 2200 D2." Retrieved 2008-10-13, 2008, from http://www.nvidia.com/object/product_quadroplex_2200_d2_us.html.

Parker, S. P., William Sloan, Peter-Pike J. Shirley, Peter Smits, Brian Hansen, Charles (1999). "Interactive ray tracing." Symposium on Interactive 3D Graphics: 119-126.

Peercy, M. S. M. Z., Benjamin R. Zhu, Daniel (1995). "Interactive full spectral rendering." SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics: 67-ff.

Pharr, M. H., Greg (2004). Physically Based Rendering: From Theory to Implementation. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc.

Pixar. (2008). "Pixar's RenderMan®." Retrieved 2008-10-13, 2008, from https://renderman.pixar.com/.

Pixar. (2008). "Pixar Animation Studios." Retrieved 2008-10-13, 2008, from http://www.pixar.com/.

Plachetka, T. (2002). Perfect Load Balancing for Demand-Driven Parallel Ray Tracing. Euro-Par 2002.

Plachetka, T. (2002). (Quasi-) Thread-Safe PVM and (Quasi-) Thread-Safe MPI without Active Polling. Euro PVM/MPI 2002.

Plachetka, T. S., Olaf Albracht, Frank (2001). The HiQoS Rendering System. SOFSEM 2001.

Pomi, A. and P. Slusallek (2004). Interactive Mixed Reality Rendering in a Distributed Ray Tracing Framework. IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2004, Student Colloquium.

Pov-ray. (2008). "POV-Ray - The Persistence of Vision Raytracer." Retrieved 2008-10-13, 2008, from http://www.povray.org/.

Purcell, T. J. (2004). Ray Tracing on a Stream Processor. Department of Computer Science. Stanford, University of Stanford. **Ph.D.:** 113.

Purcell, T. J. B., Ian R. Mark, William Hanrahan, Pat (2002). "Ray Tracing on Programmable Graphics Hardware." ACM Transactions on Graphics **21**(3): 703--712.

Rayshade. (2008). "Rayshade Homepage." Retrieved 2008-10-13, 2008, from http://graphics.stanford.edu/~cek/rayshade/.

Reinhard, E. (1999). Scheduling and Data Management for Parallel Ray Tracing. Department of Computer Science. Bristol, UK, University of Bristol. **Ph.D.**

Reinhard, E. and F. W. Jansen Hybrid Scheduling for Efficient Ray Tracing of Complex Images.

Reinhard, E. T., Lucas U.W. Jansen, Frederik (1994). Environment Mapping for Efficient Sampling of the Diffuse Interreflection.

RoD. (2006). "RoD." Retrieved 2008-10-13, 2008, from http://www.3cresearch.co.uk/item/12.

Saha, B. Z., Xiaocheng Chen, Hu Gao, Ying Yan, Shoumeng Rajagopalan, Mohan Fang, Jesse Zhang, Peinan Ronen, Ronny Mendelson, Avi (2009). "Programming model for a heterogeneous x86 platform." PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation: 431-440.

Samanta, R. F., Thomas Li, Kai Singh,Jaswinder Pal (2000). Sort-First Parallel Rendering with a Cluster of PCs. In SIGGRAPH 2000 Technical sketches**:** 26--0.

Samanta, R. F., Thomas Li, Kai Singh,Jaswinder Pal (2000). Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs.

Schmittler, J. W., Ingo Slusallek, Philipp (2002). SaarCOR — A Hardware Architecture for Ray Tracing. Graphics Hardware.

Schmittler, J. W., Sven Wagner, Daniel Paul, Wolfgang J. Slusallek, Philipp (2004). Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. Graphics Hardware.

Schwarz, M. W. C., William B. Beatty, John C. (1987). "An experimental comparison of RGB, YIQ, LAB, HSV, and opponent color models." ACM Trans. Graph. **6**(2): 123-158.

SGI. (2008). "OpenGL - The Industry Standard for High Performance Graphics." Retrieved 2008-10-13, 2008, from http://www.opengl.org/.

Shirley, P. M., R. Keith (2003). Realistic Ray Tracing. Natick, MA, USA, A. K. Peters, Ltd.

Sundstedt, V. C., Alan Cater, Kirsten Debattista, Kurt (2004). "Top-Down Visual Attention for Efficient Rendering of Task Related Scenes." VMV 2004 - Vision, Modelling and Visualization.

Sung, K. S., Peter Rosenberg, Becky Reed (2007). "Experiencing aspects of games programming in an introductory computer graphics class." SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education: 249-253.

Tanenbaum, A. S. (1988). Computer networks: 2nd edition. Upper Saddle River, NJ, USA, Prentice-Hall, Inc.

Tanenbaum, A. S. (2001). Modern Operating Systems: 2nd edition. Upper Saddle River, NJ, USA, Prentice-Hall, Inc.

Tannenbaum, D. C. T., Peter Wozny, Michael J. (1994). "Polarization and birefringency considerations in rendering." SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques: 221-222.

Thompson, C. J. H., Sahngyun Oskin, Mark (2002). "Using modern graphics architectures for general-purpose computing: a framework and analysis." MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture: 306-317.

Tisnovsky, P. H., Adam (2002). "Adaptive algorithm for vector field interpolation based on octree structure." SCCG '02: Proceedings of the 18th spring conference on Computer graphics: 151-156.

Tole, P. P. (2003). Two Algorithms for Progressive Computation of Accurate Global Illumination, Cornell University.

Wald, I. (2004). Realtime Ray Tracing and Interactive Global Illumination. Max-Planck-Institut f¨ur Informatik. Saarbrucken, Germany, Saarland University.

Wald, I. (2005). The OpenRT-API. SIGGRAPH '05, Los Angeles, California, ACM Press.

Wald, I. (2005). Siggraph 2005 Course on Interactive Ray Tracing The OpenRT API. SIGGRAPH '05: ACM SIGGRAPH 2005 Courses**:** 21.

Wald, I. B., Carsten Dietrich, Andreas Slusallek, Philipp (2003). "Interactive Ray Tracing on Commodity PC clusters - State of the Art and Practical Applications." Lecture Notes on Computer Science **2790**: 499--508.

Wald, I. B., Carsten Slusallek, Philipp (2002). A Flexible and Scalable Rendering Engine for Interactive 3D Graphics, Computer Graphics Group, Saarland University.

Wald, I. B., Carsten Slusallek, Philipp (2003). Distributed Interactive Ray Tracing of Dynamic Scenes. PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, IEEE Computer Society.

Wald, I. B., Carsten Slusallek, Philipp (2003). Interactive Global Illumination in Complex and Highly Occluded Environments. Eurographics Symposium on Rendering 2003.

Wald, I. K., Thomas Benthin, Carsten Keller, Alexander Slusallek, Philipp (2002). Interactive Global Illumination using Fast Ray Tracing. Eurographics Workshop on Rendering.

Wald, I. P., Timothy J.Schmittler, Jörg Benthin, Carsten Slusallek, Philipp (2003). Realtime Ray Tracing and its use for Interactive Global Illumination. EUROGRAPHICS 2003 - STAR – State of The Art Report.

Wald, I. S., Philipp (2001). State of the Art in Interactive Ray Tracing. EUROGRAPHICS ’01 STAR – State of The Art Report.

Wald, I. S., Philipp Benthin, Carsten Wagner, Markus (2001). Interactive Rendering with Coherent Ray Tracing. EG 2001 Proceedings, Blackwell Publishing. **20(3):** 153-164.

Ward, G. J. (1988). "A Ray Tracing Solution for Diffuse Interreflection." ACM on Computer Graphics **22**(4): 85-92.

Ward, G. J. (1992). Mesuring and Modeling Anisotropic Refection. SIGGRAPH92.

Ward, G. L. S., Rob (1998). Rendering with radiance: the art and science of lighting visualization. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc.

Ward, G. S., Maryann (1999). "The Holodeck Ray Cache: An Interactive Rendering System for Global Illumination in Nondiffuse Environments." ACM Transactions on Graphics **18(4)**: 361-398.

Wikipedia. (2008). "BIOS." from http://en.wikipedia.org/wiki/BIOS.

Williams, S. S., John Oliker, Leonid Kamil, Shoaib Husbands, Parry Yelick, Katherine (2006). "The potential of the cell processor for scientific computing." CF '06: Proceedings of the 3rd conference on Computing frontiers: 9-20.

Yee, H. P., Sumanta Greenberg, Donald P. (2001). Spatiotemporal Sensitivity and Visual Attention for Efficient Rendering of Dynamic Environments.

Zemcik, P. T., Pavel Herout, Adam (2003). "Particle rendering pipeline." SCCG '03: Proceedings of the 19th spring conference on Computer graphics: 165-170.

Fonte das Imagens

[Img01] Wikipedia, "Polygon Mesh" (http://en.wikipedia.org/wiki/Polygon_mesh).

[Img02] Wikipedia, "Bidirectional_scattering_distribution_function" (http://en.wikipedia.org/wiki/Bidirectional_scattering_distribution_function).

[Img03] Wikipedia, "Radiosity" (http://en.wikipedia.org/wiki/Radiosity_%283D_computer_graphics%29).

[Img04] Wikipedia, "Photon Mapping" (http://en.wikipedia.org/wiki/Photon_mapping).

[Img05] Wikipedia, "Sliding Window Protocol" (http://en.wikipedia.org/wiki/Sliding_Window_Protocol).

APÊNDICE A – RECURSOS UTILIZADOS

O presente apêndice apresente as características técnicas dos equipamentos e recursos utilizados na experiências conduzidas no capítulo 4 (Trabalho experimental).

Computador 1	
Processador	Intel Pentium 4
Memória	1GiB DDR400/Dual Channel (3-4-4-8)
Memória Virtual	3GiB (1GiB física + 2GiB de *swap* em disco)
Interface do GPU	AGP 8x
Sistema	Linux / Fedora 9 / Kernel 2.6.
Disco/FS	Seagate SATA 250MB/Ext3

Tabela 14 – Características do computador 1

Computador 2	
Processador	Intel Pentium 4
Memória	1GiB DDR400/Dual Channel (3-4-4-8)
Memória Virtual	*3GiB (1GiB física + 2GiB de swap em disco)*
Interface do GPU	PCI Express 16x
Sistema	Linux / Fedora 9 / Kernel 2.6.
Disco/FS	Seagate SATA 250MB/Ext3

Tabela 15 – Características do computador 2

Placa Gráfica XFX 6600GT (GPU1)	
Série	GeForce 6 Series
Memória	128 MiB DDR2
Pixel shaders	8
Vertex shaders	3
Frequência	400 MHz / 1000 MHz
Interface	AGP 8x

Tabela 16 – Características da placa gráfica designada de GPU 1

Placa Gráfica XFX 8600GT (GPU2)	
Série	GeForce 8 Series
Memória	512 MiB DDR3
Pixel shaders	Não aplicável (utiliza os *stream processors*)
Vertex shaders	Não aplicável (utiliza os *stream processors*)
Stream Processors	32
Frequência	540 MHz / 1180 MHz / 700 MHz
Interface	PCI Express 16x

Tabela 17 – Características da placa gráfica designada de GPU 2

APÊNDICE B – EXEMPLO DE PRIMITIVA

Neste apêndice apresenta-se alguns exemplos ilustrativos do código que implementa uma primitiva e da representação de dados, bem como das primitivas do HRA.

Alguns exemplos são apenas trechos de código e não ficheiro correspondente na sua totalidade, pois pretende apresentar a estrutura modular e simples da implementação dos elemento básicos e com facilidades de portabilidade.

btypes.h

```
#ifndef _BTYPES_H_                          /* avoid more than one inclusion */
#define _BTYPES_H_

typedef enum _BOOL {FALSE = 0, TRUE = 1} BOOL;

typedef double Real;

typedef Real Coord;

typedef enum _COORD {XCOORD = 0, YCOORD = 1, ZCOORD = 2} COORD;
typedef Real Interval[2];

#endif
```

vector.h

```
#ifndef _VECTOR_H_                          /* avoid more than one inclusion */
#define _VECTOR_H_

#include <math.h>

#include "btypes.h"


typedef Coord Vector2D[2];

typedef Coord Vector3D[3];

typedef Coord Point2D[3];

typedef Coord Point3D[3];


#define SetV2D(v, x, y)          (v)[XCOORD] = x;\
                                 (v)[YCOORD] = y

#define SameV2D(v1, v2)          (((v1)[XCOORD] == (v2)[XCOORD]) &&\
                                  ((v1)[YCOORD] == (v2)[YCOORD]))

#define DiffV2D(v1, v2)          (((v1)[XCOORD] != (v2)[XCOORD]) ||\
                                  ((v1)[YCOORD] != (v2)[YCOORD]))

#define AddV2D(r, v1, v2)        (r)[XCOORD] = (v1)[XCOORD] + (v2)[XCOORD];\
                                 (r)[YCOORD] = (v1)[YCOORD] + (v2)[YCOORD]

#define SubV2D(r, v1, v2)        (r)[XCOORD] = (v1)[XCOORD] - (v2)[XCOORD];\
                                 (r)[YCOORD] = (v1)[YCOORD] - (v2)[YCOORD]

#define ScalarMulV2D(r, v, t)    (r)[XCOORD] = (v)[XCOORD] * (t);\
                                 (r)[YCOORD] = (v)[YCOORD] * (t)
```

```
#define ScalarDivV2D(r, v, t)    (r)[XCOORD] = (v)[XCOORD] / (t);\
                                 (r)[YCOORD] = (v)[YCOORD] / (t)

#define SquareLenghtV2D(v)       ((v)[XCOORD] * (v)[XCOORD] +\
                                  (v)[YCOORD] * (v)[YCOORD])

#define LenghtV2D(v)             sqrt(SquareLenghtV2D(v))

#define DotProdV2D(v1, v2)       ((v1)[XCOORD] * (v2)[XCOORD] +\
                                  (v1)[YCOORD] * (v2)[YCOORD])


#define SetV3D(v, x, y, z)       (v)[XCOORD] = x;\
                                 (v)[YCOORD] = y;\
                                 (v)[ZCOORD] = z

#define SameV3D(v1, v2)          (((v1)[XCOORD] == (v2)[XCOORD]) &&\
                                  ((v1)[YCOORD] == (v2)[YCOORD]) &&\
                                  ((v1)[ZCOORD] == (v2)[ZCOORD]))

#define DiffV3D(v1, v2)          (((v1)[XCOORD] != (v2)[XCOORD]) ||\
                                  ((v1)[YCOORD] != (v2)[YCOORD]) ||\
                                  ((v1)[ZCOORD] != (v2)[ZCOORD]))

#define AddV3D(r, v1, v2)        (r)[XCOORD] = (v1)[XCOORD] + (v2)[XCOORD];\
                                 (r)[YCOORD] = (v1)[YCOORD] + (v2)[YCOORD];\
                                 (r)[ZCOORD] = (v1)[ZCOORD] + (v2)[ZCOORD]

#define SubV3D(r, v1, v2)        (r)[XCOORD] = (v1)[XCOORD] - (v2)[XCOORD];\
                                 (r)[YCOORD] = (v1)[YCOORD] - (v2)[YCOORD];\
                                 (r)[ZCOORD] = (v1)[ZCOORD] - (v2)[ZCOORD]

#define ScalarMulV3D(r, v, t)    (r)[XCOORD] = (v)[XCOORD] * (t);\
                                 (r)[YCOORD] = (v)[YCOORD] * (t);\
                                 (r)[ZCOORD] = (v)[ZCOORD] * (t)

#define ScalarDivV3D(r, v, t)    (r)[XCOORD] = (v)[XCOORD] / (t);\
                                 (r)[YCOORD] = (v)[YCOORD] / (t);\
                                 (r)[ZCOORD] = (v)[ZCOORD] / (t)

#define SquareLenghtV3D(v)       ((v)[XCOORD] * (v)[XCOORD] +\
                                  (v)[YCOORD] * (v)[YCOORD] +\
                                  (v)[ZCOORD] * (v)[ZCOORD])

#define LenghtV3D(v)             sqrt(SquareLenghtV3D(v))

#define DotProdV3D(v1, v2)       ((v1)[XCOORD] * (v2)[XCOORD] +\
                                  (v1)[YCOORD] * (v2)[YCOORD] +\
                                  (v1)[ZCOORD] * (v2)[ZCOORD])

#define ExProdV3D(r, v1, v2)     (r)[XCOORD] = (v1)[YCOORD] * (v2)[ZCOORD] -\
                                               (v1)[ZCOORD] * (v2)[YCOORD];\
                                 (r)[YCOORD] = (v1)[ZCOORD] * (v2)[XCOORD] -\
```

```
                                                (v1)[XCOORD] * (v2)[ZCOORD];\
                                    (r)[ZCOORD] = (v1)[XCOORD] * (v2)[YCOORD] -\
                                                (v1)[YCOORD] * (v2)[XCOORD];

void UnitV2D(Vector2D *v);
void UnitV3D(Vector3D *v);

#endif
```

vector.c

```
#include "vector.h"

/*****************************************************************************
 * Aim: Transform a 2D vector into a unit vector
 * Parameters:
 *   v - given 2D vector
 * Output: none
 *****************************************************************************/
void UnitV2D(Vector2D *v)
{
  Coord k;

  k = (Coord)1 / LenghtV2D(*v);
  ScalarMulV2D(*v, *v, k);
}

/*****************************************************************************
 * Aim: Transform a 3D vector into a unit vector
 * Parameters:
 *   v - given 3D vector
 * Output: none
 *****************************************************************************/
void UnitV3D(Vector3D *v)
{
  Coord k;

  k = (Coord)1 / LenghtV3D(*v);
  ScalarMulV3D(*v, *v, k);
}
```

triangle.h

```
#ifndef _TRIANGLE_H_                        /* avoid more than one inclusion */
#define _TRIANGLE_H_

#include "../rgb.h"
#include "../ray.h"
#include "../cache.h"
```

```
typedef struct _Triangle
{
  Point3D p[3];
  RGB rgb;
} Triangle;


BOOL HitTriangle(Triangle *shape, Ray *ray, Interval interval, Cache *cache);

#endif
```

triangle.c (parcial)

```
#include "triangle.h"

/*****************************************************************************
 * Aim: Test if a ray hit the triangle
 * Parameters:
 *   shape - triangle
 *   ray - pointer to the ray to test
 *   interval - interval of the ray to test
 *   cache - caching structure to be fill
 * Output: FALSE if not or TRUE if hit
 *****************************************************************************/
BOOL HitTriangle(Triangle *shape, Ray *ray, Interval interval, Cache *cache)
{
  BOOL out = FALSE;
  Real tval;
  Real dx0, dy0, dz0, dx1, dy1, dz1, dx2, dy2, dz2, x, y, z;
  Real demon, beta, gamma;
  Vector3D temp1, temp2;

  dx0 = shape->p[0][XCOORD] - ray->o[XCOORD];
  dy0 = shape->p[0][YCOORD] - ray->o[YCOORD];
  dz0 = shape->p[0][ZCOORD] - ray->o[ZCOORD];
  dx1 = shape->p[0][XCOORD] - shape->p[1][XCOORD];
  dy1 = shape->p[0][YCOORD] - shape->p[1][YCOORD];
  dz1 = shape->p[0][ZCOORD] - shape->p[1][ZCOORD];
  dx2 = shape->p[0][XCOORD] - shape->p[2][XCOORD];
  dy2 = shape->p[0][YCOORD] - shape->p[2][YCOORD];
  dz2 = shape->p[0][ZCOORD] - shape->p[2][ZCOORD];

  x = dy2 * ray->d[ZCOORD] - dz2 * ray->d[YCOORD];
  y = dz2 * ray->d[XCOORD] - dx2 * ray->d[ZCOORD];
  z = dx2 * ray->d[YCOORD] - dy2 * ray->d[XCOORD];

  demon = dx1 * x + dy1 * y + dz1 * z;
  beta = (dx0 * x + dy0 * y + dz0 * z) / demon;

  if ((beta > 0.0) && (beta < 1.0f))
   {
```

```
    x = dy1 * dz0 - dz1 * dy0;
    y = dx0 * dz1 - dz0 * dx1;
    z = dx1 * dy0 - dx0 * dy1;

    gamma = (ray->d[ZCOORD] * z + ray->d[YCOORD] * y + ray->d[XCOORD] * x)
            / demon;

    if ((gamma > 0.0) && ((beta + gamma) < 1.0f))
     {
      tval = -(dz2 * z + dy2 * y + dx2 * x) / demon;
      if ((tval >= interval[0]) && (tval <= interval[1]))
       {
        out = TRUE;
        if (cache)
         {
          cache->where = tval;
          SubV3D(temp1, shape->p[1], shape->p[0]);
          SubV3D(temp2, shape->p[2], shape->p[0]);
          ExProdV3D(cache->normal, temp1, temp2);
          UnitV3D(&(cache->normal));
          cache->ray = ray;
          CopyRGB(cache->color, shape->rgb);
         }
       }
     }
   }

  return (out);
}
```

hra.h

```
#ifndef _HRA_H_                              /* avoid more than one inclusion */
#define _HRA_H_

typedef struct _WorkUnit
{
  int type;                                /* type of function and limitations */
  int (*func)(void *);                                    /* function to apply */
  void *data;                                                 /* apply to data */
} WorkUnit;

typedef WorkUnit WU;

void *HRAmain(void *input);
void *VirtualCPU(void *input);

#endif
```

bin.h

```
#ifndef _BIN_H_                                /* avoid more than one inclusion */
#define _BIN_H_

#include <stdarg.h>

typedef struct _BinNode
{
  void *data;
  struct _BinNode *next;
} BinNode;

typedef struct _BinStruct
{
  pthread_mutex_t mutex;
  pthread_cond_t no;
  pthread_cond_t full;
  pthread_cond_t empty;
  int status;                                        /* bin status for disposal */
  int size;                                              /* number of WU in bin */
  int maxsize;                                             /* maximum WU in bin */
  int low;                              /* low threshold for considering empty */
  int high;                             /* high threshold for considering full */
  BinNode *first;                                                  /* first WU */
  BinNode *end;                                                     /* last WU */
  BinNode *free;                                             /* first free slot */
  BinNode *buffer;                                            /* the bin buffer */
#ifdef DEBUG
  int count;                                          /* number of WU processed */
#endif
} BinStruct;

typedef BinStruct * Bin;

enum BIN_LOCATION {BIN_START, BIN_END};

#define BIN_IN_USE        1
#define BIN_NO_USE        0

#define NO_BIN            NULL
#define NO_LINK           NULL

#define BinSize(bin)     ((bin)->size)
#define BinFull(bin)     ((bin)->size == (bin)->maxsize)
#define BinHasWork(bin)  ((bin)->size)
#define BinEmpty(bin)    (!((bin)->size))
#define BinMaxSize(bin)  ((bin)->maxsize)
#define BinStat(bin)     ((float)(bin)->size/(bin)->maxsize)
#define BinInUse(bin)    ((bin)->status)

Bin NewBin(int maxsize, double empty, double full);
int AddBinNodeData(Bin bin, void *data);
int GetBinData(Bin bin, void **data);
void FreeBin(Bin bin);
```

```
int GetBinBlockData(Bin bin, int max, void **data);
int GetBinBlockDataByType(Bin bin, int max, void **data, void *type);

#ifdef DEBUG
#define PrintBinConfig(who, bin)  printf("%s: Total=%d, Max=%d, low=%d, high=%d,
WU=%d, status=%d\n",\
                                          who, (bin)->count, (bin)->maxsize, \
                                          (bin)->low, (bin)->high, \
                                          (bin)->size, (bin)->status)
void BinShow(Bin bin);
#endif

#endif
```

bin.c (parcial)

```
#ifdef DEBUG
#include <stdio.h>
#endif
#include <stdlib.h>
#include "bin.h"

#include <pthread.h>

/*****************************************************************************
 * Aim: create a new bin with a maximum size
 * Parameters:
 *   maxsize - maximum elements that the bin can hold
 * Output: pointer the new bin or NULL if error
 *****************************************************************************/
Bin NewBin(int maxsize, double empty, double full)
{
  int i;
  Bin bin;
  BinNode *ptr;

  if ((bin = (Bin)malloc(sizeof(BinStruct))) != NULL)
   {
    if ((ptr = (BinNode *)malloc(sizeof(BinNode) * maxsize)) != NULL)
     {
      for(i = 0 ; i < maxsize - 1; i++)
        ptr[i].next = ptr + i + 1;                        /* link free nodes */
      ptr[maxsize - 1].next = NO_LINK;

      pthread_mutex_init(&(bin->mutex), NULL);           /* inicialize mutexs */
      pthread_cond_init(&(bin->work), NULL);
      pthread_cond_init(&(bin->empty), NULL);

      bin->status = BIN_IN_USE;
      bin->size = 0;
      bin->maxsize = maxsize;
      bin->low = (int)(empty * maxsize);
```

```
      bin->high = (int)(full * maxsize);
      bin->buffer = ptr;
      bin->free = ptr;
      bin->first = NO_LINK;
      bin->end = NO_LINK;
#ifdef DEBUG
      bin->count = 0;
#endif
     }
     else
     {
      free(bin);
      bin = NULL;
     }
   }

  return (bin);
}

/*****************************************************************************
 * Aim: add a data to the begining of a bin
 * Parameters:
 *   bin - the bin
 *   data - pointer to the data
 * Output: 0 if not in 1 if in
 *****************************************************************************/
int AddBinNodeData(Bin bin, void *data)
{
  int status = 0;
  BinNode *node;

  if (bin)
   {
    pthread_mutex_lock(&(bin->mutex));

    if (bin->size >= bin->high)                          /* bin almost full */
     {
      pthread_cond_signal(&(bin->work));
      if (BinFull(bin))                                 /* if bin full wait */
        pthread_cond_wait(&(bin->empty), &(bin->mutex));
     }

    node = bin->free;                       /* get a node from the free list */
    bin->free = node->next;

    node->data = data;
    node->next = NO_LINK;
    if (bin->first == NO_LINK)           /* bin empty then last is also first */
      bin->first = node;
     else                                   /* if not update the actual last */
      (bin->end)->next = node;
    bin->end = node;
    (bin->size)++;
    status = 1;
```

```
    pthread_mutex_unlock(&(bin->mutex));
   }

  return (status);
}
```

GLOSSÁRIO DE TERMOS E CONCEITOS

O presente glossário não é uma glossário exaustivo de todos os termos utilizados na teste, ou na área científica onde este se enquadra. Pretende sim apresentar de um forma sucinta o sentido dos termos utilizados na implementação e especificação do modelo propostos, bem como esclarecer o sentido exacto da utilização alguns termos de computação gráfica, computação paralela, comunicações de dados, no sentido da sua utilização ou ainda esclarecer um leitor menos familiarizado numa das áreas referidas.

Os termos definidos nesta secção são utilizados ao longo da tese, com o sentido com que devem ser compreendidos, dentro do contexto onde são referidos.

A

Amostragem

Processo de recolher amostras seguindo um algoritmo, distribuição ou função matemática, por forma a reconstruir o sinal sem perda de informação relevante.

Aliasing

Interferência da componente de alta frequência de um sinal (e.g. imagem) na representação e amostragem do referido sinal.

Anti-aliasing

Forma, técnica ou distribuição para evitar as interferência da componente de alta frequência de um sinal amostrado.

B

Btree

Estrutura de dados para organizar a representação de um cenário numa estrutura em árvore, sendo a divisão espacial efectuada em apenas segundo um dos eixos em cada interacção (divisão de um espaço 3D em dois, através de um plano).

C

Cenário

Conjunto de dados representativos da geometria e propriedades fotométricas dos materiais dos objectos, e fontes de luzes de um espaço 3D considerado.

Ciclo principal de rendering

Define a parte do algoritmo de *rendering* responsável pela definição de que raios, cálculos de iluminação e componentes do algoritmo de *rendering* devem ser efectuadas no processo de síntese de imagem ou animações.

Contentor

Estrutura capaz de armazenar as unidades de trabalho produzidas e respectivos resultados. No modelo proposto existem dois contentores: contentor das unidades de trabalho e contentor dos resultados. Consultar termo **unidade de trabalho**.

Correlação de eficiência

Factor relativo que indica a adaptação de uma primitiva de *rendering* com um tipo específico de recurso. O valor pode ser atribuído empiricamente ou através de experimentação. Ver termo **Primitiva**.

Componente difusa

Ver termo **Difusa (componente de luz)**

Componente directa

Ver termo **Directa (componente de luz)**

D

Difusa (componente de luz)

Componente da luz reflectida considerado os ângulos não opostos ao raio incidente e a normal no ponto de incidência.

Directa (componente de luz)

Componente da luz reflectida considerado o ângulo oposto ao raio incidente e a normal no ponto de incidência.

E

Escalar

Elemento numérico básico na composição de todos os dados do modelo proposto, por defeito equivalente a representação standard de um valor do tipo real.

Espectral (luz)

Decomposição da luz no domínio da frequências ou comprimentos de onda.

F

Fidelidade (alta)

Propriedade com alguma subjectividade na classificação da qualidade de uma imagem ou sequência de imagens relativamente a factores visuais e interpretação visual do cérebro humano.

Fonte de luz

Elemento de um cenário com iluminação própria ou auto iluminado. Pode ser representado apenas por um único ponto em sistema mais simples ou pela distribuição da emissão de luz nas superfícies de um objecto 3D do cenário.

Footprint

Quantidade de memória necessária para armazenar um trecho de código ou de dados num determinada arquitectura de um processador ou recurso.

G

Gama

Cardinal do conjunto com ordem de uma representação ou propriedade dentro de um sistema.

L

Latência

Ver **tempo de latência**

M

MISD

Multiple Instruction Single Data - Aplicação de um conjunto de instruções a um dado no mesmo ciclo de processamento.

MIMD

Multiple Instruction Multiple Data - Aplicação de um conjunto de instruções a um conjunto de dados homogéneos no mesmo ciclo de processamento.

O

Octree

Estrutura de dados para organizar a representação de um cenário numa estrutura em árvore, sendo a divisão espacial efectuada em todos os eixos em cada interacção (divisão de um espaço 3D em oito sub espaços).

Objecto

Elemento de um cenário que representa um entidade física definida, composto através da representação de um ou mais formas 3D elementares.

P

Primitiva

Conjunto de código que realiza uma parte do algoritmo de *rendering*, com parâmetros de entrada e resultados. O código pode ser distinto em diferentes recursos de hardware, no entanto é considerado como sendo sempre a mesma primitiva.

Primitiva auto-contida

Primitiva de *rendering* dependente apenas dos parâmetros de entrada (e.g. raios e superfícies) e que não invoca ou não necessita qualquer outra primitiva para ser totalmente realizada. Ver **primitiva**.

R

Recurso

Recurso físico (hardware) com capacidade capaz de efectuar processamentos de todas ou parte do conjunto de primitivas definidas. Dependo do contexto onde é referido, pode indicar a instância ou elemento de

um tipo específico de recursos físico ou representar a classe de um tipo específico de recursos.

Renderer

Dependo do contexto pode se referir a todo o código do software de *rendering*, ou apenas ao ciclo principal que inclui as primitivas complexas recursivas e geração de das unidades de trabalho.

RGB

Representação da coloração de um *pixel* ou propriedades fotométricas do tipo de material dos objectos que compõem um cenário.

S

SIMD

Single Instruction Multiple Data – Aplicação de tipo de instrução a um conjunto de dados homogéneos no mesmo ciclo de processamento (e.g. somar um valor constante a um matriz de inteiros, a todos os elementos simultaneamente).

SISD

Single Instruction Single Data – Aplicação de um instrução a um único dado em cada ciclo de processamento. Caso género na programação sequêncial.

SPMD

Single Program Multiple Data – Aplicação de um conjunto de instruções (programa ou trecho de código) a um conjunto possivelmente heterogéneo de dados.

T

Tempo de latência

Diferencial entre o tempo do inicio da transmissão de uma trama/pacote no emissor e o inicio da recepção da referida trama ou pacote por parte do receptor.

Tempo de comunicação

Diferencial entre o tempo do inicio da transmissão de uma trama/pacote no emissor e fim da transmissão dessa mesma trama/pacote por parte do emissor.

Thread

Elemento semelhante a um processo UNIX e que reside dentro processo. No entanto contém informação partilhada com o processo, com seja o caso dos segmentos de código e dados, mas tendo uma apontador para o código à executar e um segmento de *stack* próprio. Por vezes denominado de processo leve em determinada bibliografia.

U

Unidade de trabalho ou UT

Unidade para processamento composto por um conjunto de dados e pela primitiva que deve ser aplicada aos referidos. O conjunto de dados deve ser suficiente para completar na totalidade o processamento da primitiva.

V

Vector

Elemento geométrico utilizado em múltiplas representações no modelo propostos. Composto por uma matriz de 3 elementos do tipo Escalar. Ver **Escalar**.

W

Work unit

Ver **Unidade de trabalho**

X

XYZ (coordenadas)

Representação geométrica de objectos no espaço tridimensional de elementos 3D.