COPPE/UFRJ

GRIDRT: UMA ARQUITETURA PARALELA PARA RAY TRACING UTILIZANDO VOLUMES UNIFORMES

Alexandre Solon Nery

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Nadia Nedjah

Rio de Janeiro
Fevereiro de 2010

GRIDRT: UMA ARQUITETURA PARALELA PARA RAY TRACING UTILIZANDO VOLUMES UNIFORMES

Alexandre Solon Nery

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Nadia Nedjah, Ph.D.

Prof. Luiza de Macedo Mourelle, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
FEVEREIRO DE 2010

Solon Nery, Alexandre

GridRT: Uma Arquitetura Paralela para Ray Tracing utilizando Volumes Uniformes/Alexandre Solon Nery. – Rio de Janeiro: UFRJ/COPPE, 2010.

XIII, 137 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Nadia Nedjah

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2010.

Referências Bibliográficas: p. 76 – 80.

1. Traçado de Raios. 2. Arquitetura. 3. FPGA. I. Maia Galvão França, Felipe *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

"O computador surgiu para solucionar os problemas que antes não existiam."

Autor desconhecido.

Agradecimentos

Agradeço a Deus, a meus pais e demais familiares, pelo apoio incondicional por todas as etapas da minha vida. À Lucimara e à sua família, pela compreensão e carinho ao longo desta fase de dedicação aos estudos. Aos Professores Felipe, Nadia, Luiza e Maltar, que desde o início desta pesquisa estiveram presentes, dispostos a me ajudar e orientar. Aos meus amigos, pelo companheirismo, pelos momentos de descontração e conselhos em momentos difíceis.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

GRIDRT: UMA ARQUITETURA PARALELA PARA RAY TRACING UTILIZANDO VOLUMES UNIFORMES

Alexandre Solon Nery

Fevereiro/2010

Orientadores: Felipe Maia Galvão França
Nadia Nedjah

Programa: Engenharia de Sistemas e Computação

Entre os objetos de estudo da Computação Gráfica destaca-se a visualização em tempo real de cenas tridimensionais. Nesse sentido, procura-se renderizar tais cenas tridimensionais em imagens que sejam o mais próximas possível da realidade. Isto é feito através de algoritmos de *Iluminação Global*, tais como o *Traçado de Raios*, *Traçado de raios Monte Carlo* e *Radiosidade*. No entanto, aliar a síntese de imagens de alta qualidade e desempenho não é uma tarefa trivial. O algoritmo de traçado de raios é capaz de avaliar todas as características da cena a fim de compor a informação da cor de um *pixel* da imagem final, cujo processo é custoso. Em contra partida, este algoritmo pode ser facilmente paralelizado. Dessa forma, algumas implementações paralelas deste algoritmo em software estão atingindo desempenho satisfatório e, portanto, é de se esperar que uma implementação em hardware atinja desempenho igual ou melhor, em tempo real. Diante disso, neste trabalho, apresenta-se uma arquitetura paralela para traçado de raios, utilizando estrutura de aceleração de Volumes Uniformes. Esta arquitetura não depende da plataforma em que será implementada e é capaz de realizar os cálculos de interseção em paralelo. São apresentadas duas implementações da arquitetura em software: uma usando OpenMP e a outra OpenMPI, além de uma implementação em FPGA.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

GRIDRT: A PARALLEL ARCHITECTURE FOR RAY TRACING USING UNIFORM GRIDS

Alexandre Solon Nery

February/2010

Advisors: Felipe Maia Galvão França
Nadia Nedjah

Department: Systems Engineering and Computer Science

One of the main subjects of research in Computer Graphics is the visualization of three dimensional scenes, so that the produced image is as close as possible to reality. This goal can be accomplished through *Global Illumination* algorithms, such as *Ray Tracing*, *Path Tracing* and *Radiosity*. However, it is not easy to achieve high speed rendering and image fidelity. The ray tracing algorithm is capable of evaluating the scene characteristics in order to compute the color of a single image pixel. Thus, it is a high cost computational process. However, parallel implementations of ray tracing have been achieving a satisfactory performance, as the algorithm is embarrassingly parallel. Thus, a custom parallel design in hardware is expected to achieve or surpass real time performance. Therefore, in this work we present a parallel architecture for ray tracing, using a spatial subdivision technique known as Uniform Grids. Such architecture is independent of any platform and is capable of performing parallel intersection calculations. Also, two parallel implementations in software are presented: one using OpenMP and the other OpenMPI, as well as an FPGA implementation.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Em Computação Gráfica, um dos maiores desafios é a produção de imagens de qualidade fotográfica a partir de uma cena tridimensional. O desafio é mais agudo quando se trata de uma produção em tempo real, ou seja, a uma taxa de pelo menos 60 imagens por segundo (*frames per second* - fps), garantindo interatividade entre o usuário e a visualização da imagem.

Tais objetivos ainda estão longe de serem alcançados, principalmente quando se trata de aliar desempenho e qualidade de imagem. Os processadores gráficos utilizam algoritmos baseados em *Iluminação Local*, que dominam o mercado há pelo menos duas décadas. O processo de produção de imagem nesse modo é denominado de *rasterização* [1]. Este termo é geralmente aplicado ao processo de conversão de uma informação vetorial (e.g. descrições geométricas) em uma imagem. Neste modelo de implementação, o *hardware* não tem conhecimento acerca da cena toda. Sua funcionalidade é a projeção ou conversão em *pipeline* de polígonos para coordenadas de uma imagem, a uma alta vazão (na ordem de centenas de milhares de triângulos por segundo). Portanto, sem o conhecimento das características da cena e da posição dos demais objetos, a imagem gerada carece de detalhes importantes, como sombras e reflexões [2].

Os algoritmos capazes de produzir imagens de melhor qualidade são aqueles que se enquadram na categoria de algoritmos de *Iluminação Global*. Nestes algoritmos, a cena toda é avaliada para compor cada elemento da imagem final. Um desses algoritmos é chamado de Traçado de Raios [3–5], (*Ray Tracing*). Para isso, são lançados raios, i.e. vetores, em direção à cena tridimensional, a fim de coletar informações da cena, permitindo a composição de cada pixel. O trajeto dos raios é traçado à medida que estes colidem com os objetos da cena. Tais trajetos são constantemente computados por meio de onerosos cálculos de interseção, sendo que esses são realizados para cada ponto de interseção encontrado. As informações do objeto atingido são então usadas para colorir um pixel da imagem, referente ao raio traçado. Por isso, o algoritmo de traçado de raios produz diretamente simulações de sombra, reflexão

e refração. No entanto, em implementações simplistas do algoritmo de traçado de raios, o tempo de produção de uma imagem é proporcional ao número de objetos e raios traçados, o que acarreta um alto custo computacional [1, 4].

Apesar disso, o algoritmo de traçado de raios é paralelizável, uma vez que cada raio pode ser tratado independentemente dos demais. A escalabilidade deste algoritmo é quase linear para até 16 processadores disponíveis [6–8], o que o torna atraente para ser implementado em arquiteturas paralelas, tais como as unidades de processamento gráfico (*Graphics Processing Units* - GPUs) e as arquiteturas customizadas em FPGA (*Field Programmable Gate Array*).

Os processadores gráficos caracterizam-se pelo processamento em *pipeline* para processar um grande conjunto de dados em paralelo, inclusive com o uso de instruções SIMD (*Single Instruction Multiple Data*) de 4 vias, cujos dados são geralmente vetores representados em ponto flutuante [1]. Graças à introdução de estágios programáveis no *pipeline* das GPUs, tornou-se possível o desenvolvimento de aplicações de propósito geral que são capazes de executar nos processadores gráficos, inclusive o próprio traçado de raios [9–11]. Isso tem ocorrido gradualmente e continua a evoluir [1], tendendo a substituir por completo os estágios fixos do *pipeline* por estágios programáveis [12]. Além disso, por meio da linguagem CUDA [13] da Nvidia ou OpenCL [14] é ainda mais fácil desenvolver aplicações voltadas para GPUs.

As FPGAs, por outro lado, são dispositivos programáveis, nos quais qualquer sistema digital pode ser implementado através de uma *linguagem de descrição de hardware*, HDL [15], tendo disponíveis os recursos requeridos. O sistema é convertido para funções lógicas que executam diretamente no hardware, geralmente operando a baixas frequências. Porém, dependendo da quantidade de recursos disponíveis na FPGA, partes do sistema podem ser replicadas e paralelizadas, compensando a baixa frquência de operação. Logo, uma aplicação pode ser mapeada em *hardware* e paralelizada da maneira que for mais adequada, principalmente por meio da replicação de elementos processadores e/ou de unidades funcionais [16, 17]. Diante disso, o algoritmo de traçado de raios é um forte candidato para ser implementado em FPGA [18]. Por exemplo, o paralelismo do algoritmo pode ser explorado através da replicação de unidades aritméticas para execução de cálculos de interseção em paralelo ou, até mesmo, através da construção de uma arquitetura completa para executar o traçado de raios completo [19, 20].

1.1 Motivação

O desempenho do algoritmo de traçado de raios é baixo [1, 3, 4], principalmente em termos de taxa de atualização de quadros (*frames per second* - *fps*). Uma taxa de atualização de pelo menos 60 quadros por segundo é necessária para garantir

interatividade e visualização contínuas em tempo real. Esse desempenho é dificilmente atingido com esse algoritmo, ainda que sejam utilizadas estruturas de dados que reduzem a quantidade de cálculos de interseção e algoritmos paralelos. Apesar disso, algumas implementações paralelas [8, 21] são capazes de atingir taxas de até 10 quadros por segundo, explorando coerência de cache assim como o uso de instruções especiais do processador [22, 23].

Embora a arquitetura das GPUs esteja evoluindo para dar suporte a outros tipos de aplicação, o modelo de programação é fortemente ligado ao processamento de dados em fluxo contínuo (*Stream Processing*). Deste modo, um fluxo de dados de entrada é processado em diferentes estágios do *pipeline* do processador gráfico. Cada um destes estágios realiza uma tarefa específica do processo de rasterização, para que uma imagem seja produzida ao longo da travessia dos dados no *pipeline*. Mais recentemente, estes estágios puderam ser programados para realizar uma tarefa de propósito geral. Além disso, alguns dos estágios são subdivididos em estágios ainda menores e contando com diversas unidades funcionais operando em paralelo, como por exemplo várias unidades de ponto flutuante [1].

Diante deste cenário, o desempenho do algoritmo de traçado de raios em GPU continua insatisfatório e geralmente inferior ao desempenho de implementações paralelas do mesmo [2]. A fim de garantir eficiência, as GPUs precisam acessar blocos lineares de memória. Isto é uma das razões para o baixo desempenho do traçado de raios nesta arquitetura, pois o acesso à memória no traçado de raios tende a ser aleatório [2]. Além disso, o algoritmo de traçado de raios é recursivo e, logo, seu desempenho depende do tratamento das instruções de controle de fluxo, funções estas que são executadas eficientemente por processador de propósito geral [24]. Os primeiros modelos de GPUs com estágios programáveis não contemplavam instruções de laços de repetição nem predição de desvio [9]. Somente a partir de 2004, as GPUs passaram a incluir algumas destas funções [1]. Ainda, a ausência de estruturas de pilha também dificulta a implementação de fluxos recursivos, de modo que as informações sobre o estado de execução do traçado de raios deve ser armazenada em memória de textura da GPU [9, 10]. Segundo o fabricante Nvidia, a geração de processadores gráficos *GF100* será a primeira a introduzir suporte à recursividade em hardware através de sua arquitetura *Fermi* [25].

Por essas razões, as FPGAs representam uma alternativa atrativa de desenvolvimento do traçado de raios para execução em hardware, permitindo a exploração de algumas das deficiências dos processadores gráficos para produzir uma arquitetura mais rápida quanto à execução do algoritmo completo ou pelo menos de seu caminho crítico. Nesse sentido, implementações paralelas do algoritmo em FPGA já alcançaram taxas de atualização de imagens adequadas para visualização em tempo real [19, 20], operando a frequências de apenas 60 a 90MHz, que são muito menores

se comparada à frequência de operação em torno de 500MHz das GPUs. Em termos de recursos de hardware, a área ocupada por um processador gráfico também é geralmente maior.

1.2 Objetivo

O objetivo deste trabalho é projetar uma arquitetura paralela, GridRT [26, 27], para a execução do algoritmo de traçado de raios, independente da plataforma de implementação. A arquitetura explora o uso de estruturas espaciais de dados. Nela, a cena a ser processada é repartida em regiões de tamanho regular, nas quais residem uma parcela da cena. Cada uma destas é alocada a um elemento processador, possibilitando cálculos de interseção em paralelo para aqueles raios que atravessam um conjunto de elementos processadores do volume.

1.3 Contribuições

Uma arquitetura paralela para acelerar os cálculos de interseção do algoritmo de traçado de raios foi desenvolvida, independente de plataforma. Esta arquitetura é baseada na estrutura de aceleração de Volumes Uniformes, a fim de reduzir os cálculos de interseção e paralelizá-los de acordo com a mesma, favorecendo a síntese de cenas grandes e esparsas. Duas implementações paralelas em software desta arquitetura também foram desenvolvidas, assim como uma implementação em hardware. O potencial de paralelismo da arquitetura é promissor, de forma a prover escalabilidade e processamento paralelo de raios.

1.4 Metodologia

A arquitetura GridRT foi desenvolvida através da linguagem de descrição de hardware VHDL (*Very high speed integrated circuits Hardware Description Language*) e simulada no ModelSim Xilinx Edition 6.3c [28], independentemente da plataforma em que seria implementada. Nesta etapa, foram simuladas configurações de quatro e oito processadores. Três componentes adicionais se encarregavam de gerar os raios primários, atravessá-los no Volume Uniforme e encaminhar estes dados para processamento pelo conjunto de processadores.

Antes da etapa de síntese da arquitetura em hardware, duas implementações em software foram desenvolvidas na linguagem C++, utilizando *Open Multi-Processing* (OpenMP) [29] e *Open Message Passing Interface* (OpenMPI) [30] para dar suporte ao paralelismo. Ambas as versões foram executadas em uma arquitetura Intel™

Corei7 de 2.26GHz, para configurações de 8,12,27,36 e 64 processos/threads. A cena *Stanford Bunny* [31] foi utilizada nos testes em software e em hardware, sendo que neste último a mesma foi introduzida nas memórias dos elementos processadores no momento da especificação do hardware.

Uma vez testada, a arquitetura GridRT em hardware foi conectada ao microprocessador MicroBlaze [32], programado para executar as funções de geração de raios primários e atravessamento do volume uniforme. Desta forma, o GridRT age como co-processador do MicroBlaze, conectado a este via um barramento dedicado de alta velocidade. Tudo isso foi feito por meio do *Xilinx Platform Studio*. Finalmente, a síntese destes componentes foi realizada pelo XST [33] (*Xilinx Synthesis Technology*), configurado para otimização (redução) de área.

Ao término da síntese, os componentes, compilados para funções lógicas, foram mapeados em uma FPGA Virtex-5, modelo XC5VFX70T [34]. Uma vez mapeados, o MicroBlaze iniciou o processamento do algoritmo de traçado de raios, comunicando-se com o co-processador GridRT, responsável pelos cálculos de interseção do traçado de raios. Os resultados destes cálculos foram retornados ao MicroBlaze, que por sua vez os enviou à uma estação de trabalho via porta rs232. Nesta estação, o aplicativo *HyperTerminal* coletou os resultados, i.e. os pontos de interseção, e os apresentou em sua tela.

1.5 Organização da dissertação

O Capítulo 2 introduz o algoritmo de traçado de raios e suas principais características. Em seguida, o Capítulo 3 apresenta uma revisão bibliográfica dos trabalhos mais relevantes encontrados na litetura, que também buscaram soluções para o algoritmo de traçado de raios em arquiteturas paralelas. O Capítulo 4 descreve a arquitetura paralela GridRT independentemente de detalhes de implementação, enquanto os Capítulos 5 e 6 descrevem as soluções implementadas em software e hardware, respectivamente. Finalmente, o Capítulo 7 conclui este trabalho e apresenta direções para possíveis trabalhos futuros.

Capítulo 2

Traçado de Raios

Neste capítulo será apresentado o algoritmo de Traçado de Raios (*Ray Tracing*), assim como as suas principais características. Maiores detalhes podem ser encontrados em livros especializados sobre o assunto [3, 4, 12].

2.1 Introdução

O Traçado de Raios é um algoritmo para visualização de uma cena tridimensional. Ele produz uma imagem correspondente da cena através de raios lançados de uma *câmera virtual* apontada em direção à cena que se deseja visualizar, como mostra a Fig.2.1. Esta câmera faz o papel de observador e os raios determinam quais os objetos visíveis da cena. Para isto, o algoritmo calcula o menor ponto de interseção entre um raio e cada objeto da cena, o que corresponde a encontrar o objeto mais próximo do observador, isto é, o objeto visível.

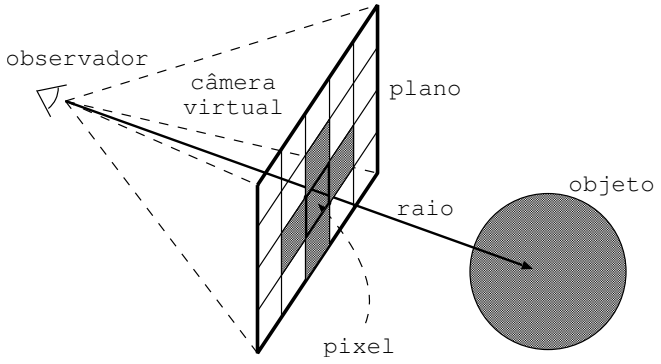


Figura 2.1: Exemplo de projeção de raios.

Cada raio atravessa um *pixel*[1] do plano da câmera virtual, onde a imagem é capturada. Observe que o número de raios lançados corresponde ao número de pixels

[1]É o menor elemento de uma imagem ou dispositivo de exibição, ao qual é possível atribuir uma cor. Um conjunto de pixels forma uma imagem inteira.

do plano de visão da câmera virtual e que quanto maior a quantidade destes, melhor a qualidade (resolução) da imagem produzida. Uma vez encontrada a menor interseção entre um raio e a cena, a cor do objeto atingido é atribuída ao pixel em questão, ainda segundo a Fig.2.1. Esse procedimento também é conhecido como algoritmo de Projeção de Raios (*Ray Casting*), que é uma versão reduzida do algoritmo de Traçado de Raios, visto que outros objetos da cena não contribuem para a coloração da imagem.

O algoritmo de traçado de raios deve ser capaz de interagir com a cena toda, a fim de compor a cor de um dado pixel. Para isso, a trajetória de um raio é determinada pelo cálculo de interseção entre o raio e os objetos da cena. No ponto de interseção encontrado, o algoritmo coleta as informações necessárias sobre o objeto (e.g. cor, material, vetor normal) e, com base nestas informações, calcula a nova direção do raio ou não. Na Fig.2.2, por exemplo, um raio atinge a superfície de um objeto polido e é refletido, o que o faz atingir um segundo objeto.

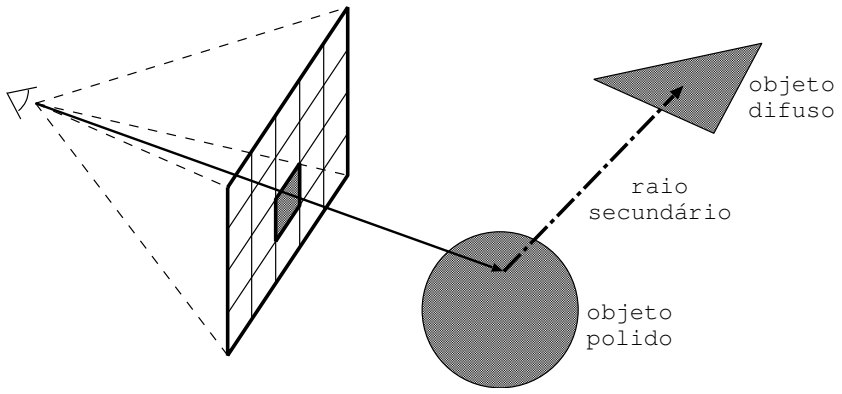


Figura 2.2: Exemplo de traçado de raios.

Essa situação pode ser interpretada como a reflexão do segundo objeto sob a superfície do primeiro, o que altera a composição da cor do pixel correspondente ao raio traçado. Caso a superfície desse outro objeto também fosse polida, o raio seria novamente refletido e o algoritmo procederia assim por diante até encontrar um objeto difuso (não reflexivo) ou nenhuma interseção. Para objetos transparentes, um procedimento semelhante é aplicado: em uma colisão, o raio atravessa o objeto conforme as leis de refração (vide Seção 2.2.2) e em seguida pode atingir ou não outros objetos da cena. Em caso afirmativo, modifica-se a coloração do pixel correspondente ao raio traçado.

Além disso, para cada ponto de interseção um raio também é gerado em direção a cada fonte de luz da cena, como será visto na Seção 2.2.3. Estes raios são utilizados para verificar se o ponto de interseção encontrado recebe luz de alguma das fontes luminosas da cena. Desse modo, se o ponto em questão não recebe luz, então o mesmo encontra-se em sombra. O algoritmo também calcula a contribuição das

fontes de luz para a coloração da superfície no ponto de interseção, a fim de que a representação do objeto seja fiel à posição das fontes luminosas. Senão, os objetos teriam uma aparência achatada na imagem produzida e não seriam afetados pelas fontes de luz. Este tópico, conhecido por *Shading Model*, será abordado em maiores detalhes na Seção 2.4.

O algoritmo de traçado de raios é, portanto, uma simulação dos trajetos percorridos pelos raios em uma cena tridimensional. O algoritmo encaixa-se na categoria de modelos de *Iluminação Global* que, ao contrário dos modelos de *Iluminação Local*, considera a interação entre todos os elementos de uma cena para a composição da imagem, o que a confere um alto grau de realismo. Neste contexto, entende-se a palavra *realismo* como sendo a produção de efeitos importantes de sombra, reflexão e refração, que são inerentes ao algoritmo de traçado de raios, isto é, tais efeitos são resultados naturais da execução do algoritmo.

2.2 O Algoritmo

O algoritmo de traçado de raios é geralmente estruturado em funções, cada qual especializada em uma tarefa. A primeira delas, descrita no Algoritmo 2.1, encarrega-se de gerar os raios iniciais que atravessam os pixels da câmera virtual, em busca dos primeiros pontos de interseção. Tais raios são descritos na Seção 2.2.1.

```
   Entrada: cena
   Saída: imagem
1  para cada pixel da câmera virtual faça
2      iteração := 0;
3      raio := o + t · d;            /* calcular vetor do raio inicial */
4      imagem[pixel] := trace(raio,iteração);
```

Algoritmo 2.1: Algoritmo de traçado de raios (*Whitted-Style*)

Observe que o valor da iteração inicializado na linha 2 é argumento da função *trace* da linha 4, descrita no Algoritmo 2.2. Esta função traça recursivamente a trajetória de um raio inicial e o valor da iteração controla a profundidade desta recursividade, ou seja, quantas reflexões ou refrações o raio pode sofrer ao longo do seu trajeto. Logo, quando um valor máximo for atingido, a função retorna a cor de fundo padrão da imagem, geralmente preto.

Para se determinar a trajetória do raio são realizados testes de interseção, a cada iteração. Assim, quando o menor ponto de interseção for encontrado, a função *shade* na linha 10 do Algoritmo 2.2 recebe os resultados da interseção. Nesta função, descrita no Algoritmo 2.3, a posição de cada fonte de luz é avaliada em relação ao ponto de interseção, a fim de descobrir se o ponto recebe luz direta de alguma fonte

```
   Entrada: raio, iteração
   Saída: cor
 1 se iteração > max então
 2     retorna preto ;                /* geralmente é a cor de fundo */
 3 senão
 4     menor := ∞;
 5     para cada objeto da cena faça
 6         t := interseção(raio, cena[obj]);
 7         se t < menor e t > 0 então
 8             menor := t ;     /* atualiza o menor valor encontrado */
 9             resultado := cena[obj] ;        /* armazena o objeto e suas
               propriedades */
10     cor := shade(resultado,menor,iteração);
11     retorna cor;
```

Algoritmo 2.2: Função recursiva (*trace*) do traçado de raios.

luminosa. Para isso, são realizados novos testes de interseção contra cada raio de sombra, conforme a Seção 2.2.3. Portanto, caso o ponto de interseção receba luz direta, a cor da superfície no ponto indicado é somada às cores das fontes de luz visíveis a partir dele.

Em seguida, o material de que é constituído o objeto é avaliado. Caso ele seja reflexivo, um novo raio é calculado na direção da reflexão do raio incidente. Por outro lado, se o objeto for transparente, um novo raio é calculado segundo a lei de refração. Em ambos os casos, a função *trace* é executada recursivamente para o novo raio calculado e incrementando o identificador da iteração. A reflexão e refração de um raio são discutidas na Seção 2.2.2.

```
   Entrada: objeto,ponto de interseção,iteração
   Saída: cor
 1 cor = resultado.cor;
 2 para cada fonte de luz faça
 3     calcular o vetor do raio de sombra;
 4     se não existem objetos no caminho do raio então
 5         cor = cor + luz;
 6 se resultado.material é espelhado então
 7     novo raio := nova direção ;                  /* raio refletido */
 8     cor := cor + trace(novo raio, iteração + 1);
 9     retorna cor
10 se resultado.material é transparente então
11     novo raio := nova direção ;                  /* raio refratado */
12     cor := cor + trace(novo raio, iteração + 1);
13     retorna cor
```

Algoritmo 2.3: Função de iluminação (*shade*) do traçado de raios.

Cada uma destas funções desempenha um papel importante do algoritmo de traçado de raios. Juntas, elas descrevem o funcionamento do algoritmo e englobam os procedimentos de cálculo de interseções e iluminação, que serão abordados nas Seções 2.3 e 2.4, respectivamente. Este conjunto de algoritmos foi primeiramente descrito por Turner Whitted [35] e recebeu o nome de *Whitted-Style Ray Tracer*.

2.2.1 Raio Primário

Os *raios primários* são aqueles que partem do observador em direção à cena. Um raio primário pode gerar um raio secundário a cada interseção. Neste contexto, a *Câmera Virtual* faz, conforme mostra a Fig. 2.3, o papel de observador. Ela é composta de um ponto de origem, conhecido por "olho" do observador, e também é definida por uma janela formada de vários pixels: o *plano de visão*.

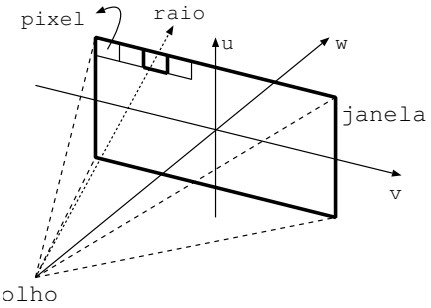


Figura 2.3: Raio primário no plano de visão de um observador.

Cada raio primário atravessa um pixel do plano de visão e é composto por um ponto de origem e um ponto de destino, que juntos definem sua direção. Cada ponto, por sua vez, é composto por três coordenadas (x, y, z) no plano Cartesiano. O vetor de direção é dado pela subtração do ponto de destino do ponto de origem. Logo, um raio é definido pela Equação 2.1, em sua forma paramétrica, onde o é a origem, t é o parâmetro e d é o vetor de direção normalizado.

$$r = o + t \cdot \overrightarrow{d} \tag{2.1}$$

De acordo com a Equação 2.1, parametrizada em t, qualquer ponto do raio pode ser encontrado variando o valor do parâmetro t, se o vetor de direção d estiver normalizado. Quando t é inferior a zero, significa que o ponto encontra-se antes da origem do raio, enquanto que para valores superiores a zero, o ponto encontra-se além da origem. Por fim, se t for igual a zero, o ponto corresponde à origem do raio, conforme mostra a Fig. 2.4.

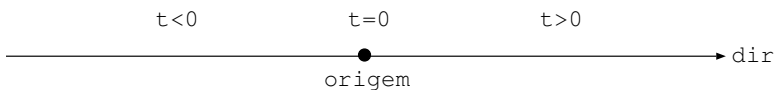


Figura 2.4: Valores de t para a equação paramétrica do raio.

2.2.2 Raio Secundário

Os *raios secundários* são gerados a partir de um ponto de interseção, embora também sejam definidos de acordo com a Equação 2.1. A direção de um raio secundário é calculada segundo as propriedades do objeto no ponto de interseção encontrado. Por isso, dependendo das características do objeto, este raio pode sofrer reflexão ou refração. Em ambos os casos, o raio incidente e o vetor normal na superfície do objeto são utilizados para calcular a nova direção.

Portanto, os raios secundários são responsáveis por introduzir a influência que os demais objetos da cena exercem sob o pixel correspondente ao raio primário que os gerou. A quantidade de raios secundários que serão gerados precisa ser limitada, caso contrário, a computação sugerida é infinita. Por exemplo, duas superfícies espelhadas refletiriam um raio inúmeras vezes entre si, o que tornaria esta computação inviável.

Limitando a quantidade de raios secundários a zero, permite-se tratar apenas os raios primários para formação da imagem, retomando ao algoritmo de Projeção de raios introduzido na Seção 2.1. Por outro lado, quanto maior a quantidade de raios secundários permitidos, maior será a qualidade da cena, pois mais objetos poderão contribuir para a formação da imagem. Esta quantidade de raios secundários é controlada pelo número da iteração na função recursiva do Algoritmo 2.2.

Reflexão

A reflexão de um raio ocorre em superfícies espelhadas (polidas). Neste caso, o material do objeto possui um índice de reflexão que indica o quanto de luz é refletido do ambiente que o cerca, ou seja, de outros objetos ao seu redor. Este índice, no intervalo fechado $\mathbb{R}[0, 1]$, define a parcela de luz que será refletida a partir do raio incidente. Quando um objeto é difuso, esta taxa corresponde a zero e, portanto, não há reflexão alguma.

Um raio secundário, também definido de acordo com a Equação 2.1, é refletido a partir de um ponto de interseção segundo o ângulo formado entre o raio incidente e o vetor normal da superfície do objeto. Isto é ilustrado na Fig. 2.5. A esse tipo de reflexão dá-se o nome de *Reflexão Especular Perfeita*, uma vez que considera objetos perfeitamente polidos (*Perfect Specular*), quando na realidade os objetos possuem imperfeições que podem alterar a trajetória de reflexão.

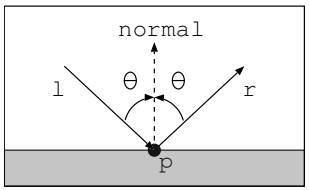


Figura 2.5: Exemplo de raio secundário de reflexão.

Sejam l, r e n três vetores co-planares, que representam o raio de luz incidente, o raio refletido e a normal da superfície do ponto de interseção p, respectivamente. O vetor r pode ser obtido através Equação 2.2,

$$r = l - 2(n \cdot l)n \tag{2.2}$$

onde $(n \cdot l)$ corresponde à operação de produto escalar (*Dot Product*) entre os vetores n e l. Esta operação é descrita na Equação 2.3, para as coordenadas (x, y, z).

$$n \cdot l = n_x l_x + n_y l_y + n_z l_z \tag{2.3}$$

Refração

A refração ocorre devido à diferença de velocidade de propagação da luz em diferentes meios, cujos índices de refração absolutos são medidos em relação à velocidade da luz no vácuo. Logo, seja c a velocidade de propagação da luz no vácuo e seja v a velocidade de propagação no meio (e.g. ar, água, gelo, etc.), a Equação2.4 apresenta esta relação.

$$\eta = \frac{c}{v} \tag{2.4}$$

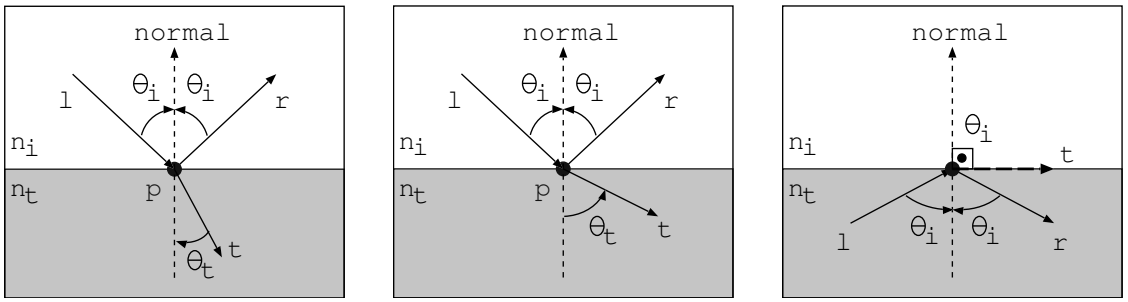
A fim de traçar os raios que atravessam objetos transparentes é preciso determinar o vetor de transmissão t, ilustrado na Fig.2.6. Para isso, utiliza-se a Equação 2.2 (vetor de reflexão) associada à *Lei de Snell*, uma vez que esta relaciona o ângulo de incidência θ_i e de transmissão θ_t aos seus respectivos meios de transmissão η_i e η_t, conforme a Equação 2.5.

$$\frac{sen(\theta_i)}{sen(\theta_t)} = \frac{\eta_t}{\eta_i} = \eta \tag{2.5}$$

Deste modo, de posse do ângulo de refração e da equação de reflexão, a expressão para t é descrita segundo a Equação2.6. Maiores detalhes em [1, 3].

$$t = \frac{1}{\eta} l - \left(\cos(\theta_t) - \frac{1}{\eta} \cos(\theta_i) \right) n \tag{2.6}$$

Quando um raio é transmitido de um índice de refração menor para um maior, o vetor de transmissão t aproxima-se do vetor normal, conforme mostra a Fig.2.6a. Porém, quando um raio é transmitido de um índice de refração maior para um menor, o vetor distancia-se do vetor normal, como é ilustrado na Fig.2.6b. Por fim, o fenômeno de reflexão total ocorre quando o ângulo de refração $\theta_t > \frac{\pi}{2}$, segundo a Fig.2.6c.



(a) Refração no ponto p, onde $n_1 < n_2$.

(b) Refração no ponto p, onde $n_1 > n_2$.

(c) Reflexão interna total no ponto p.

Figura 2.6: Fenômenos da refração e reflexão total.

Observe que esta é uma descrição para efeitos de transparência simples. Outros tipos de transparência mais realistas seguem as *Equações de Fresnel*[3, 4], mas que ainda não estão contempladas neste trabalho. No futuro, o suporte a efeitos de transparência complexos pode ser incluído.

2.2.3 Raio de Sombra

Para cada ponto de interseção encontrado é gerado um *raio de sombra* em direção a cada uma das fontes de luz da cena. A existência de algum objeto na trajetória de um raio de sombra indica que o ponto da superfície do objeto de onde o raio partiu não receberá iluminação direta de uma fonte de luz, embora ainda possa receber de outras fontes luminosas, caso existam. Considere a cena da Fig.2.7. A partir dos pontos de interseção p e q são gerados raios de sombra em direção a uma fonte de luz. Neste caso, o ponto q encontra-se na sombra, pois há um objeto no trajeto do raio até a luz.

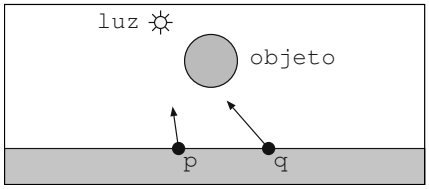


Figura 2.7: Exemplo de raio secundário de sombra.

2.3 Algoritmos de Interseção

A maioria das aplicações de Computação Gráfica fazem uso intensivo de rotinas para cálculo de interseção, especialmente em aplicações para detecção de colisões [1]. O algoritmo de traçado de raios faz uso de métodos para o cálculo do ponto de interseção entre um raio e um objeto, se existir. Para cada tipo de objeto (e.g. esferas, triângulos, cubos, etc.) deve existir um método específico para calcular a interseção com um raio. Neste trabalho, considera-se o uso de triângulos para a composição da cena, visto que é possível representar qualquer objeto com o uso de triângulos [4].

É notável a importância dos cálculos de interseção para o funcionamento do algoritmo de traçado de raios, sem os quais este não existiria. Esta seção apresenta inicialmente uma breve explicação a respeito do triângulo e de sua representação matemática, seguida de uma explicação mais detalhada do algoritmo de interseção raio-triângulo.

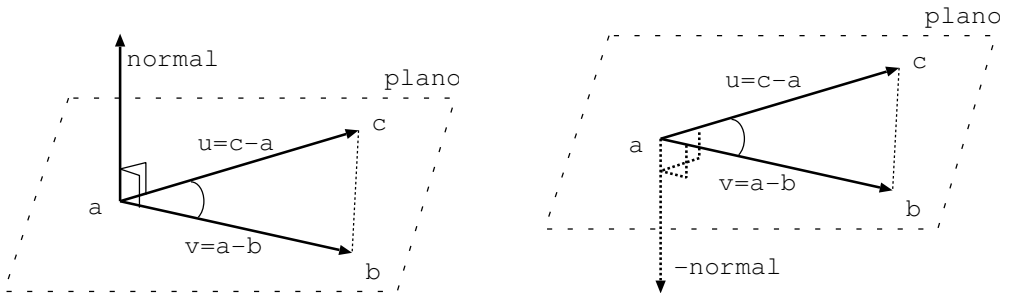
2.3.1 O Triângulo

A maioria dos objetos são representados por uma coleção de triângulos que compartilham vértices (pontos) entre si. Este conjunto representa um objeto fechado e é conhecido também por *Triangle Mesh* [4]. A Fig. 2.8 apresenta a imagem do objeto fechado *Stanford Bunny* [31], um modelo tridimensional geralmente utilizado em aplicações de computação gráfica.

Figura 2.8: Objeto fechado *Stanford Bunny*.

Um triângulo é definido por três vértices ($\mathbf{a}$,$\mathbf{b}$,$\mathbf{c}$) não co-lineares no espaço, ou seja, vértices que não estão dispostos sob a mesma linha. Desta maneira, tais vértices configuram um plano que, por sua vez, configura um vetor $\mathbf{n}$ normal ao plano

(perpendicular a ele). Este vetor normal pode ser calculado pelo produto vetorial (*Cross Product*) entre duas arestas (vetores) de um triângulo. O vetor normal também pode ser pré-computado e armazendo com os dados do triângulo, ao custo de um maior espaço de armazenamento.



(a) Produto vetorial $v \times u$. (b) Produto vetorial $u \times v$.

Figura 2.9: Representação do Triângulo.

Seja o triângulo $\triangle\mathbf{abc}$, conforme a Fig. 2.9a. Sejam os vetores $\mathbf{u} = \mathbf{c} - \mathbf{a}$ e $\mathbf{v} = \mathbf{b} - \mathbf{a}$. O produto vetorial $v \times u$ produz o vetor normal $\mathbf{n}$. Se a ordem da operação for invertida para $u \times v$, o vetor normal será produzido no sentido inverso, segundo a Fig.2.9b. A operação realizada por um produto vetorial é apresentada na Equação 2.7.

$$u \times v = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x) \tag{2.7}$$

Para garantir que o vetor normal de cada uma das faces dos triângulos esteja apontando para fora do objeto, os vértices da Equação 2.7 devem estar ordenados no sentido anti-horário, como mostra a Fig. 2.10. É dessa forma que distingui-se os triângulos visíveis dos não visíveis.

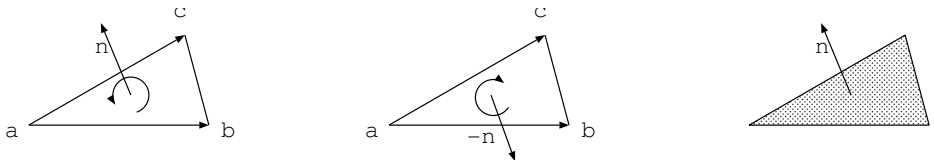


Figura 2.10: Da esquerda para a direita: Sentido anti-horário, sentido horário e a face apontando para o observador.

2.3.2 Interseção Raio-Triângulo

O algoritmo de traçado de raios dedica pelo menos 95% do tempo gasto na sua execução para calcular interseções [35], o que sugere que estes cálculos precisam

ser rápidos. Geralmente, os algoritmos de interseção raio-triângulo envolvem duas etapas:

1. Descrobrir o ponto de interseção entre o raio e o plano formado pelo triângulo;
2. Verificar se este ponto está localizado dentro do triângulo.

Na Fig.2.11, fica claro que, apesar de ambos os raios r_1 e r_2 colidirem contra o plano do triângulo, apenas r_2 o atravessa de fato.

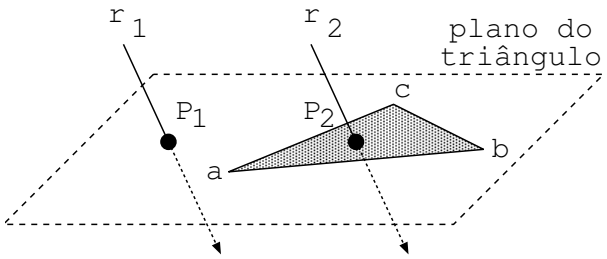


Figura 2.11: Interseção entre o raio r_2 e o triângulo $\triangle\mathbf{abc}$.

A fim de determinar com mais rapidez a interseção raio-triângulo, utiliza-se o conceito de coordenadas baricêntricas, que são coordenadas definidas pelos vértices de um triângulo ou de qualquer outro *simplex*, (e.g. tetraedro)[4]. A medida em que o algoritmo prossegue, são realizados testes que verificam a localização do ponto em relação ao triângulo, contribuindo à determinação precoce se há ou não interseção. Este procedimento é comum na literatura[4, 19, 36, 37].

Considere o triângulo $\triangle\mathbf{abc}$, e um ponto P cujas coordenadas baricêntricas correspondem à $p(\alpha, \beta, \gamma)$. Qualquer ponto do triângulo pode ser calculado utilizando a Equação 2.8.

$$p(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \tag{2.8}$$

Estas coordenadas são homogêneas, i.e. $\alpha+\beta+\gamma = 1$, permitindo que a Equação 2.8 seja re-escrita para a Equação 2.9.

$$p(\alpha, \beta, \gamma) = a + \beta(b - a) + \gamma(c - a) \tag{2.9}$$

Igualando-se as Equações 2.1 e 2.9, é possível determinar se um ponto do raio é também um ponto no plano, de acordo com a Equação 2.10. Observe que este ponto encontra-se dentro do triângulo para valores positivos de β e γ, assim como para $\beta + \gamma < 1$ [36].

$$\underbrace{o + t \cdot d}_{\text{Eq. do raio}} = \underbrace{a + \beta(b - a) + \gamma(c - a)}_{\text{Eq. do triângulo}} \tag{2.10}$$

Reordenando os termos da Equação 2.10 e escrevendo-os para cada componente (x, y, z), a Equação 2.10 torna-se o Sistema de Equações 2.11, cujas soluções para β,γ e t podem ser encontradas pela *Regra de Cramer*.

$$\begin{bmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_z & d_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - o_x \\ a_y - o_y \\ a_z - o_z \end{bmatrix} \tag{2.11}$$

Logo, resolvendo pela *Regra de Cramer*:

$$\beta = \frac{\begin{vmatrix} a_x - o_x & a_x - c_x & d_x \\ a_y - o_y & a_y - c_y & d_y \\ a_z - o_z & a_z - c_z & d_z \end{vmatrix}}{|A|}$$

$$\gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - o_x & d_x \\ a_y - b_y & a_y - o_y & d_y \\ a_z - b_z & a_z - o_z & d_z \end{vmatrix}}{|A|}$$

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - o_x \\ a_y - b_y & a_y - c_y & a_y - o_y \\ a_z - b_z & a_z - c_z & a_z - o_z \end{vmatrix}}{|A|}$$

onde,

$$A = \begin{vmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_z & d_z \end{vmatrix}$$

Uma vez encontrado o parâmetro t, basta substituí-lo na Equação 2.1 para determinar o ponto de interseção correspondente.

2.4 O Modelo de Sombreamento

O modelo de sombreamento, *Shading Model*, determina a intensidade de luz que será emitida para cada ponto de interseção encontrado, considerando a posição das fontes de luz da cena e o material do objeto atingido, além da posição do observador em relação a estas fontes. Tudo isto confere ao objeto uma aparência sólida e convincente.

Dentre os modelos existentes, o de Phong [38] é o mais conhecido. Neste, são con-

sideradas três componentes de iluminação: ambiente, difusa e especular, conforme a Fig. 2.12. Observe que a componente ambiente apresenta apenas o contorno do objeto, porque não utiliza a localização das fontes de luz nem as propriedades dos objetos para calcular a luminosidade. Por outro lado, a componente difusa mostra o volume do objeto, suas regiões iluminadas e sombreadas, devido ao posicionamento das fontes de luz em relação ao vetor normal de cada triângulo do objeto. Finalmente, a componente especular simula a reflexão da luz em relação à posição do observador e das fontes de luz, incluindo um efeito de brilho que varia segundo o posicionamento da câmera virtual.

(a) Ambiente (b) Difusa (c) Especular

Figura 2.12: Componentes do Modelo de Phong.

Ao total, a intensidade da luz no modelo de Phong para um ponto de interseção e uma fonte de luz é dada pela soma das três componentes, de acordo com a Equação 2.12, onde os termos k_a,k_d e k_e representam os coeficientes de iluminação ambiente, difusa e especular, variando no intervalo fechado $\mathbb{R}[0,1]$. Os termos I_a,I_d e I_e são as intensidades ou cores das componentes, geralmente no padrão *RGB* (*Red,Green,Blue*) e com cada cor variando no intervalor fechado $\mathbb{R}[0,1]$.

$$I_p = \underbrace{I_a k_a}_{\text{ambiente}} + \underbrace{I_d k_d\,(cos\alpha)}_{\text{difusa}} + \underbrace{I_e k_e\,(cos^e\beta)}_{\text{especular}} \tag{2.12}$$

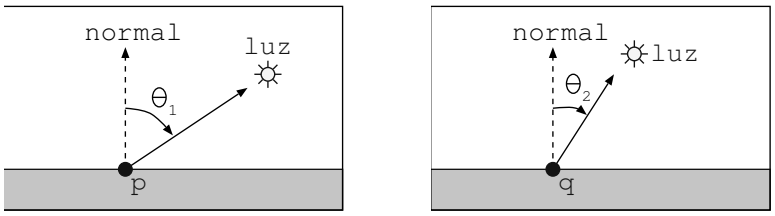
2.4.1 Componente Ambiente

A *componente ambiente* é uma aproximação da iluminação natural que cada objeto recebe do ambiente que o cerca e independe da sua geometria, conforme mostra a Fig.2.12a. Logo, esta componente não utiliza a localização das fontes de luz nem tão pouco a geometria (i.e. vetor normal) dos objetos e, por isso, ela permanece constante.

2.4.2 Componente Difusa

A *componente difusa* calcula a intensidade de luz refletida de forma uniforme em todas as direções, para um dado ponto de interseção, cujo resultado é ilustrado na Fig.2.12b. Ela depende apenas do ângulo entre o vetor de incidência da luz e o vetor normal da superfície. A base da componente difusa é a *Lei de Lambert*, que

relaciona a intensidade de luz refletida ao cosseno do ângulo formado pela normal da superfície e o vetor da fonte luminosa. Observe na Fig.2.13a que quanto maior o ângulo (θ_1), menor a incidência direta de luz no ponto p. Porém, quanto menor o ângulo (θ_2), maior a incidência direta da luz no ponto q, segundo a Fig.2.13b.



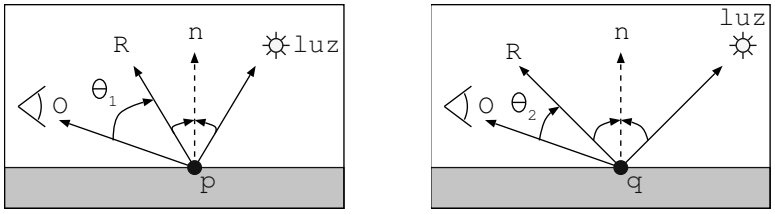
(a) Ângulo θ_1 aberto, menor incidência de luz no ponto p.

(b) Ângulo θ_2 fechado, maior incidência de luz no ponto q.

Figura 2.13: Incidência da luz nos pontos de interseção p e q.

2.4.3 Componente Especular

A *componente especular* simula refletores imperfeitos, ou seja, à maneira de superfícies polidas e depende do posicionamento entre observador, objeto e fonte de luz, cujo resultado é ilustrado na Fig.2.12c.



(a) Ângulo θ_1 aberto, menor a intensidade da reflexão no ponto p.

(b) Ângulo θ_2 fechado, maior a intensidade da reflexão no ponto q.

Figura 2.14: Intensidade da reflexão nos pontos de interseção p e q.

Esta componente depende da posição do observador, objeto e fonte de luz. A intensidade de luz refletida varia de acordo com o cosseno do ângulo formado entre o vetor de reflexão (R) da luz incidente e o vetor de direção do observador (O). A intensidade da reflexão deve ser maior para ângulos fechados, próximos do vetor de reflexão (R). Neste caso, o observador estaria olhando mais diretamente para a reflexão da luz incidente, como mostra a Fig.2.14b.

Por outro lado, quanto maior o ângulo, menor deve ser a intensidade da reflexão emitida para o observador, segundo a Fig.2.14a. Para isso, o expoente da componente especular, onde $e > 0$, controla a intensidade da reflexão, que deve ser maior quando o observador se encontrar próximo ao vetor de reflexão. Este expo-

ente acentua a queda da intensidade de luz refletida na medida em que o ângulo aumenta.

Capítulo 3

Implementações existentes

Os primeiros trabalhos sobre o uso da técnica de traçado de raios para renderização em tempo real datam da década de 80 [3] e surgiram da necessidade de sintetizar imagens mais realistas, diante das limitações do modelo de computação local, conhecido como *rasterização*. Neste modelo, muito esforço de programação é necessário para se incluir simulações de sombra, reflexão e refração que, por outro lado, são produtos diretos da execução do algoritmo de traçado de raios. Essencialmente, existem duas formas de acelerar o traçado de raios:

1. Acelerar e/ou reduzir o cálculo de interseções;

2. Paralelização.

Entre estas opções, a paralelização é geralmente a melhor forma de alcançar alto desempenho em traçado de raios. O algoritmo é altamente paralelizável, uma vez que cada raio pode ser processado independentemente, assim como os resultados podem ser produzidos em qualquer ordem. Nesse sentido, o algoritmo de traçado de raios é, há algum tempo, alvo de implementações paralelas em Clusters [8] e Sistemas de Memória Compartilhada [24], aliados às demais técnicas de aceleração por subdivisão espacial. A ideia geral é distribuir os raios a serem computados entre os processadores, de forma que cada um fica responsável por sintetizar uma parte da cena. A aceleração alcançada é muitas vezes linear ao número de processadores envolvidos na computação, o que torna o algoritmo bastante escalável [6, 7].

Diante disso, já há algum tempo acredita-se na substituição das técnicas de rasterização pela técnica de traçado de raios, principalmente diante do advento dos CMPs (*Chip Multiprocessors*) [6, 39]. Porém, a evolução dos processadores gráficos (*Graphics Processing Unit* - GPUs) nas últimas décadas foi extraordinariamente maior, em termos de número de transistores e GFLOPs [13] . Graças à introdução de estágios programáveis do *pipeline* dos processadores gráficos, estes passam a viabilizar computações de propósito geral e também o traçado de raios [9].

Contudo, os processadores gráficos também apresentam características que ainda os tornam inadequados para o processamento de traçado de raios em tempo real, como a ausência de recursividade em hardware e o acesso linear à memória [1, 2, 9]. Por isso, também existe uma vertente dedicada à pesquisa de arquiteturas paralelas em hardware para acelerar o algoritmo em questão [19, 20]. Tais arquiteturas fazem o uso de FPGAs para a prototipagem de um hardware dedicado. Nas seções seguintes serão analizados os aspectos destas três possibilidades de implementação do algoritmo: software, GPUs e FPGA.

3.1 Soluções em Software

São muitas as soluções do algoritmo de traçado de raios em software, já que a maior parte delas não se preocupa com a síntese em tempo real. A esse tipo de renderização dá-se o nome de *off-line rendering*, que não envolve processamento interativo. Por isso, é ideal para a produção de filmes, cujas cenas são produzidas quadro a quadro no decorrer de algumas semanas ou meses [1, 40]. Porém, não significa que a implementação do algoritmo não precisa ser eficiente, pois ainda assim é necessário utilizar algoritmos paralelos e estruturas de subdivisão espacial para se atingir um desempenho razoável. O POV-Ray [41](*Persistence of View Ray Tracing*) e o Yafaray [42] são duas implementações populares do algoritmo de traçado de raios. Em nenhuma das implementações existe a preocupação com a renderização em tempo real e até a versão 3.6 do POV-Ray não há suporte para arquiteturas paralelas, ou seja, o algoritmo é puramente sequencial. Já o Yafaray suporta processamento paralelo com o uso de threads.

Para a computação em tempo real, o foco dos trabalhos existentes era otimizar o algoritmo da melhor maneira possível para o processador x86, procurando "encaixar" o algoritmo por meio do alinhamento dos dados da memória com a *cache* e do uso de conjuntos de instruções especiais, como o SSE (*Streaming SIMD Extensions*) da Intel^{TM} [22]. Em um destes trabalhos [21], utiliza-se instruções SIMD para computar a intercessão entre um conjunto de raios e um triângulo, atingindo um desempenho três vezes superior ao do teste de interseção convencional. Além disso, o cuidado com a coerência de *cache* se fez necessário visto que o acesso à memória no traçado de raios é muitas vezes aleatório, em desacordo com o princípio de localidade. Tais características, aliadas ao uso de algoritmos de subdivisão espacial da cena, conferiram um ganho de desempenho de mais de uma ordem de magnitude, comparado ao desempenho do POV-Ray [41].

Conforme uma outra proposta [8], foi possível atingir desempenho em tempo real com uma abordagem paralela em um *Cluster* de sete computadores de dois núcleos cada, também utilizando coerência de *cache*, para visualização de grandes modelos

tridimensionais. Nesta implementação, a cena foi subdivida segundo a estrutura de aceleração de árvores BSP (*Binary Space Partitioning Trees*), pré-processada e armazenada inteiramente em um servidor. Deste modo, os dados são requisitado sob demanda pelos clientes durante os cálculos de interseção. As taxas de atualização da imagem variaram de 3 a 5 quadros por segundo. Com o uso de instruções SIMD, os autores acreditam que esta taxa de atualização poderia ter atingido até 12 quadros por segundo, visto que o desempenho das interseções é quase três vez maior.

Em um *sistema de memória compartilhada* [43] foi possível sintetizar imagens a uma taxa de até 20 quadros por segundo, para configurações de até 128 processadores em uma máquina SGI Origin 2000. A escalabilidade também apresenta um comportamento quase linear em relação ao número de processadores utilizados para a computação do traçado de raios.

3.2 Soluções em GPU

Graças ao modelo de processamento em fluxo (*Stream Processor*), as GPUs são altamente eficazes para tipos de computação que envolvem processamento SIMD de um grande volume de dados em ponto flutuante, o que é bastante comum em aplicações de computação gráfica. A introdução de estágios programáveis no *pipeline* dos processadores gráficos [1] possibilitou a execução de outros tipos de computação, tais como simulações científicas e o próprio traçado de raios.

Para tanto, tais aplicações precisam se adaptar a este modelo de programação, de tal maneira que a computação deve ser decomposta em estágios, chamados de *kernels* (núcleos). As primeiras propostas de decomposição do traçado de raios em GPU foram elaboradas por Purcell et al. [9] e Carr et al. [24]. A proposta de Purcell et al. sugere que todas as etapas do traçado de raios sejam feitas em GPU, incluindo a criação de raios primários, o atravessamento da estrutura de aceleração, os cálculos de interseção e o modelo de coloração dos *pixels*. Desta forma, o algoritmo de traçado de raios foi decomposto nestes quatro *kernels*. O primeiro deles gera os raios primários segundo a configuração da câmera virtual. Em seguida, estes raios são processados pelo *kernel* de transpasse da estrutura de aceleração de volumes uniformes, produzindo uma lista de raios associados aos *voxels* por eles percorridos. O próximo *kernel* calcula o menor ponto de interseção de cada raio e gera também uma lista de interseções associadas aos respectivos objetos atingidos (triângulos). Esta lista é finalmente processada pelo último *kernel*, que calcula a contribuição da informação no ponto de interseção para colorir o respectivo *pixel*. Um trabalho semelhante também foi elaborado por Kristof [10].

A proposta de Carr et al. sugere o mapeamento em GPU apenas para os cálculos de interseção. Dessa maneira, a GPU se torna um co-processador SIMD especia-

lizado em executar interseções em um grande volume de dados, enquanto a CPU processa os dados de entrada e os resultados da computação deste co-processador.

Uma questão importante levantada por ambos os trabalhos se refere a quão adequada é a arquitetura das placas gráficas para o processamento do traçado de raios, pois sabe-se que este algoritmo faz uso intensivo de instruções de desvio e recursividade, o que ainda não é o forte das GPUs [1, 25]. Carr et al. lembra que tais funções são melhor executadas pela CPU e, por este motivo, utiliza a GPU apenas para os cálculos de interseção. Purcell et al. cita a ausência de instruções de desvio e recursividade, além da baixa quantidade de registradores, como fatores que prejudicam o desempenho do algoritmo de traçado de raios em GPU. Estas questões também já foram levantadas por outros trabalhos [19, 20], que sugerem a criação de arquiteturas específicas para o traçado de raios, como será visto na Seção 3.3.

3.3 Soluções em FPGA

O uso de hardware dedicado para aceleração de aplicações tem sido bastante comum, principalmente quando um algoritmo, ou parte dele, pode ser paralelizado e mapeado em hardware [16], como é o caso de muitos algoritmos de processamento de imagens, criptografia, dinâmica de fluídos, entre outros [17]. Dessa forma, as FPGAs atuam como co-processadores do sistema e geralmente executam o caminho crítico do algoritmo que se deseja acelerar. Uma abordagem mais radical é projetar o sistema todo em hardware, ou seja, uma arquitetura completa.

Pavel Zemcik [18] faz uma avaliação dos diferentes tipos de algoritmos para processamento gráfico que podem ser implementados em FPGA, mostrando alguns exemplos de implementação e um protótipo de arquitetura de traçado de raios. Grep Humphreys e Scott Ananian [44] propuseram uma arquitetura paralela baseada em DSPs (*Digital Signal Processors*). Tal arquitetura é formada por um conjunto de DSPs conectados a uma memória, que contem os dados da cena. Os raios são enviados através de um barramento serial para cada DSP, que agem como processadores independentes de cálculos de interseção. Ou seja, cada DSP recebe um raio distinto e os testes de interseção acontecem em paralelo contra a cena inteira, cujos dados são acessados sincronamente pelo conjunto de DSPs. O foco desta arquitetura é a flexibilidade e o baixo custo de implementação, em detrimento do desempenho.

Uma alternativa simples e direta é realizar os cálculos de interseção em FPGA, que é a funcionalidade que consome mais tempo do algoritmo. No trabalho de Tim Todman e Wayne Luk [45], o cálculo de interseções com esferas foi mapeado em hardware, utilizando a linguagem de descrição Handel-C. Com o hardware limitado a 16MHz, foi possível alcançar a taxa de 5 milhões de interseções por segundo, que é um valor muito baixo se comparado ao trabalho de Purcell et al. em GPU. Uma

abordagem semelhante também foi feita por Cameron [7], que utiliza as FPGAs disponíveis em um Cray XD-1 para calcular interseções entre raios e cônicas, alcançando uma aceleração de 77% em relação a mesma implementação sem o hardware dedicado.

Os trabalhos mais relevantes do traçado de raios em FPGA são atribuidos a Schmittler et al.[20] e Woop et al[19], sendo este último uma evolução do primeiro, através da inclusão de unidades programáveis da arquitetura. Neles, uma arquitetura completa para traçado de raios foi desenvolvida, com suporte a todas as características do algoritmo, inclusive a modificação das cenas em tempo real. Com frequências variando entre 66MHz a 90MHz para um processador de raios, foi possível renderizar cenas em tempo real, atingindo taxas de atualização de imagens de até 20 quadros por segundo, o que é considerada uma taxa de atualização decente, mas ainda insatisfatória. Para tanto, os autores utilizaram-se de organização da *cache* para garantir coerência entre os raios, unidades de cálculo de interseção raio-triângulo e subdivisão espacial da cena por meio de árvores BSP, mais especificamente as árvores *KD*. Além disso, a arquitetura é escalável, podendo ser adicionado mais processamento.

Tendo como base estes trabalhos, foi produzida a arquitetura do GridRT [26, 27], com o principal diferencial de utilizar a estrutura de subdivisão espacial de Volumes Uniformes. A regularidade desta estrutura possibilita o rápido atravessamento da cena assim como uma rápida repartição da mesma. Alguns recursos do algoritmo de traçado de raios estão previstos para serem incluídos na arquitetura em trabalhos futuros, como os efeitos de refração. Apesar desta arquitetura ainda não permitir a síntese de cenas em tempo real, ela apresenta um modelo de paralelismo intuitivo e promissor, como será mostrado no Capítulo 4. O ponto central do GridRT é o processamento paralelo de cálculos de interseção para um dado raio, podendo atingir taxas de aproximadamente 77 mil interseções por segundo para cada elemento processador. Além disso, duas implementações paralelas em software foram produzidas, tomando por base os modelos de programação MPI e OpenMP, afim de avaliar algumas características da arquitetura, conforme descrito no Capítulo 5.

Capítulo 4

A Arquitetura GridRT

O modelo arquitetural do GridRT, *Grid Ray Tracing*, será discutido neste Capítulo independente da plataforma em que será implementado, servindo como embasamento para os Capítulos 5 e 6.

4.1 Volumes Uniformes

A fim de que seja viável implementar o algoritmo de traçado de raios, tanto em software quanto em hardware, é preciso utilizar uma estrutura de repartição espacial da cena a ser processada. Caso contrário, o desempenho do algoritmo é diretamente proporcional à quantidade de objetos da cena.

Existem diversas estruturas que podem ser empregadas, por exemplo: Volumes Hierárquicos Limitados (BVH) [3] , Particionamento Binário do Espaço (BSP) [3] e Volumes Uniformes (Uniform Grids) [4, 46]. Tais estruturas apresentam em comum o fato de restringir a busca por interseções a apenas uma parte da cena, ou seja, somente aos objetos que se encontram na direção de um dado raio. Para isso, a cena é subdividida em regiões (*voxels*). Em cada região há uma lista dos objetos totalmente ou parcialmente contidos nela. Tal subdivisão é feita com base em uma heurística ou em padrões regulares, como é o caso dos Volumes Uniformes, mostrado na Fig. 4.1.

Nesta divisão regular, cada raio atravessa a estrutura da malha regular em sequência e, para cada voxel visitado, são realizados os devidos testes de interseção. Assim, quando uma interseção for encontrada, os demais voxels não precisam ser visitados, pois esta interseção é com certeza a menor, uma vez que a busca é feita a partir da origem do raio e distanciando-se da mesma. Portanto, nenhuma outra interseção adiante daquela que já foi encontrada será menor que ela. No exemplo da Fig. 4.1, os voxels são visitados em ordem até que uma interseção é encontrada no voxel 6 e, então, a busca é encerrada para o raio correspondente.

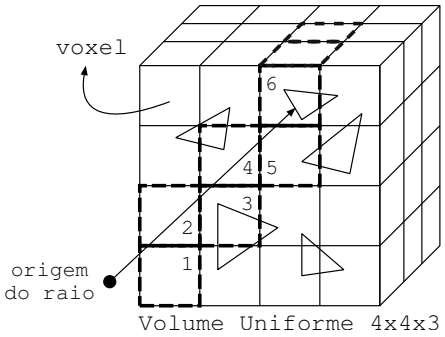


Figura 4.1: Exemplo de particionamento por Volumes Uniformes.

A preferência pelo uso dos volumes uniformes se deve principalmente a melhor adaptação da estrutura para implementação em hardware [9], graças à sua regularidade. Isto também favorece a rápida construção da estrutura e ao rápido atravessamento da cena. A estrutura pode ser representada através de um vetor unidimensional, no qual cada índice corresponde a um voxel e dá acesso à sua respectiva lista de objetos. Dessa forma, os volumes uniformes proporcionam um meio rápido e eficiente de atravessamento dos voxels, cujo tempo de acesso é constante para qualquer tipo de raio (primário ou secundário). Este algoritmo de atravessamento foi descrito por Fujimoto et al. [46] e otimizado por Amantides e Woo [47]. A construção do volume uniforme não é foco desta discussão e maiores detalhes podem ser encontrados em [4].

No caso de volumes hierárquicos, como as árvores BSP ou BVH, o tempo de acesso aos voxels varia de acordo com o raio, principalmente para os raios secundários, cujo atravessamento pode ser iniciado a partir de um ramo da árvore. Logo, é preciso realizar uma busca na árvore a fim de identificar em qual ramo se iniciará o atravessamento do raio e os cálculos de interseção. Apesar disso, os volumes hierárquicos geralmente levam à um desempenho ligeiramente superior a dos volumes uniformes. No entanto, em um estudo detalhado a respeito das estruturas de aceleração, Havran et al.[48] e Kalos et al. [49] demonstraram que é pouca a diferença de desempenho entre estas. Geralmente, tal diferença varia de acordo com as propriedades da cena e também da heurística utilizada para a repartição espacial.

4.2 Paralelismo

A arquitetura GridRT [26, 27] explora o cálculo de interseções em paralelo, para todos os voxels que são traspassados por um raio. Ou seja, o atravessamento da malha indicará quais os voxels que devem ser selecionados para que ocorram os cálculos das interseções. Cada voxel é mapeado em um Elemento de Processamento (EP) cuja funcionalidade principal é calcular a interseção entre um raio e a parcela

da cena correspondente. No exemplo da Fig.4.2, observa-se que duas interseções foram encontradas, t_1 e t_2, cada uma pelos elementos de processamento 5 e 6, respectivamente. Obviamente, t_1 é menor que t_2. No entanto, ambos os elementos de processamento são incapazes de tomar esta decisão sem terem conhecimento dos valores um do outro.

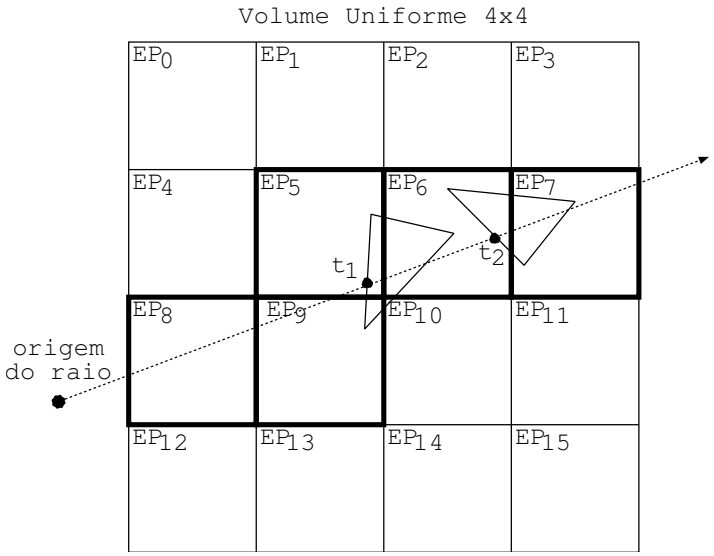


Figura 4.2: Voxels mapeados em Elementos de Processamento.

Desse modo, não há como distinguir diretamente qual das interseções calculadas é a menor, já que esta propriedade é automaticamente garantida pela ordem de atravessamento sequencial dos voxels. A princípio, uma solução simples envolveria a troca das interseções encontradas entre todos os EPs. Logo, ao término da computação em um raio, cada EP teria as interseções encontradas pelos demais e, portanto, poderia determinar se a sua interseção é a menor ou não. Em caso afirmativo, o EP então iria proguedir para a computação restante, senão seu resultado seria descartado e um novo raio seria processado. Apesar de viável, esta solução prevê a troca e comparação de informações, que pode ser um processo oneroso, considerando que as interseções são definidas por suas coordenadas, representadas em ponto flutuante. Além disso, todos os EPs teriam que aguardar pelo término do processamento daquele com mais interseções a computar.

Portanto, a solução adotada retoma à ordem do traspasse para determinar o resultado correto e, ainda assim, manter o paralelismo dos testes de interseção. Logo, em um primeiro passo, a malha é atravessada por um raio para que uma lista com a sequência dos voxels (identificadores de EPs) seja gerada. Em seguida, esta lista é transmitida aos EPs, que a consultam para verificar se devem participar da computação ou não, ou seja, se os seus respectivos identificadores estão listados ou não. Para aqueles EPs identificados na lista, inicia-se a computação de interseções

em paralelo e, caso uma interseção seja encontrada, o EP em questão deve atuar de acordo com as seguintes possibilidades:

1. Primeiro da lista: o EP envia um sinal (mensagem) de interrupção ao próximo da lista, que por sua vez o retransmitirá ao próximo e abortará sua computação, e assim por diante, até que o último seja informado. Imediatamente após enviar o sinal de interrupção, o primeiro pode prosseguir sua computação utilizando seu resultado, já que nenhum outro poderia ser menor que este (graças à sua ordem na lista). No exemplo apresentado na Fig. 4.3, o EP_3 encontrou uma interseção, em um momento T_1, e enviou um sinal de interrupção ao próximo (EP_4), que foi retransmitido até o último. Em um momento T_2, todos os EPs adiante foram interrompidos, enquanto apenas o EP_3 continua a computação.

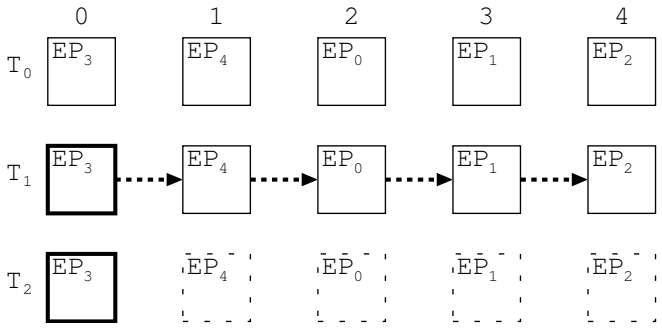


Figura 4.3: Interseção no primeiro da lista de atravessamento.

2. Último da lista: o EP deve apenas aguardar por um sinal de confirmação ou interrupção do EP anterior. Note que ele não poderá prosseguir até que receba algum destes sinais, que podem ser uma retransmissão de interrupção ou uma indicação de que todos os anteriores terminaram a computação sem resultados. No caso de interrupção, o EP aborta sua computação. Já no caso de término, o EP tem certeza de que pode progredir utilizando seu resultado, já que nenhum EP anterior encontrou uma interseção menor. Na Fig.4.4, o EP_2 encontra uma interseção e aguarda pelo sinal de término ou interrupção, para decidir se aborta ou continua a computação.

3. Intermediário da lista: o EP envia um sinal de interrupção ao próximo da lista e aguarda por um sinal de confirmação ou interrupção do anterior. No caso de confirmação, a computação prossegue, senão é abortada assim como nos casos anteriores. A Fig 4.5 exemplifica esta situação, na qual o EP_0 encontra uma interseção, interrompe o próximo da lista e aguarda pela confirmação ou interrupção do EP_4.

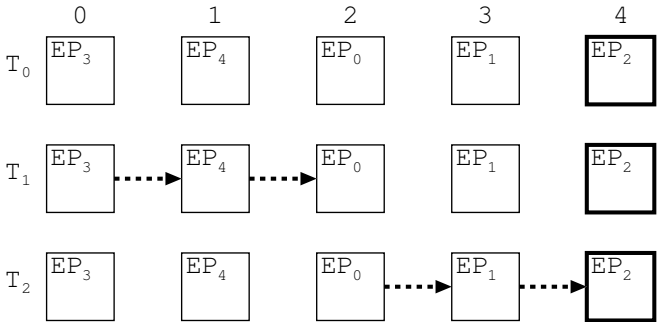


Figura 4.4: Interseção no último da lista de atravessamento.

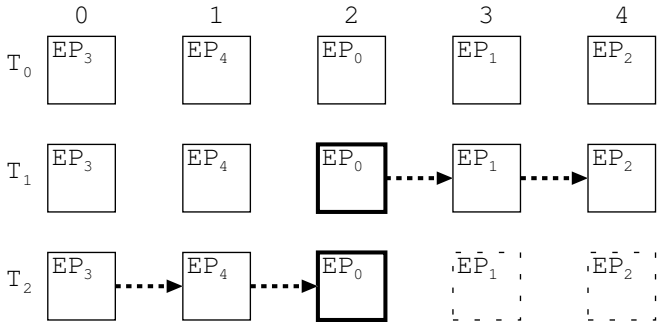
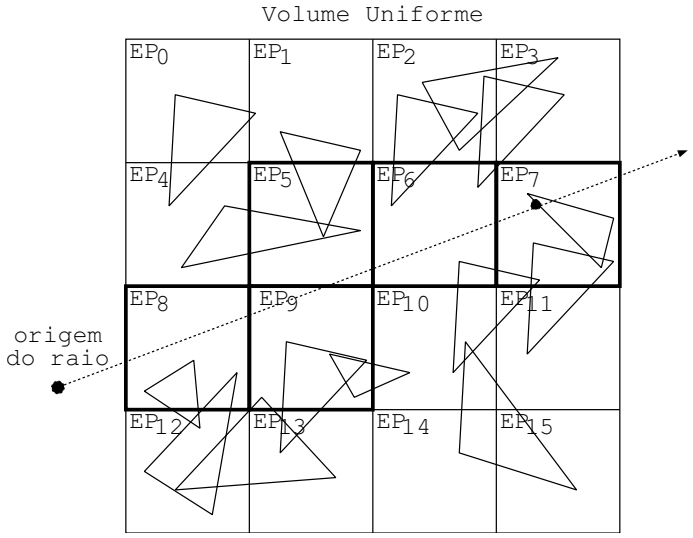


Figura 4.5: Interseção no meio da lista de atravessamento.

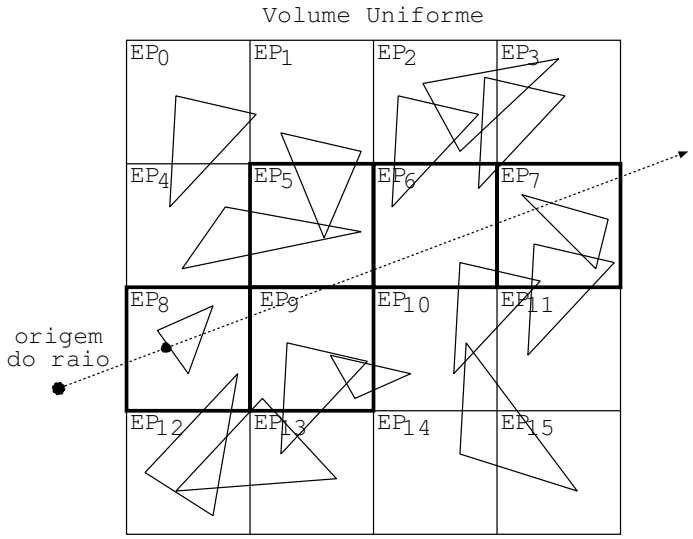
4.3 Expectativas de desempenho

Neste modelo de paralelismo, espera-se obter uma performance favorável para os raios cujas interseções ocorrem mais profundamente no volume, ou seja, para aqueles raios que penetram no interior da malha até encontrar uma interseção, conforme mostra a Fig.4.6a. Note que, neste caso, o cálculo de interseções em paralelo favorece o adiantamento do resultado que será encontrado pelos últimos elementos de processamento, ao contrário do que ocorre no algoritmo sequencial. Portanto, o modelo paralelo é ideal para as cenas com vários objetos e que estes estejam bem distribuídos pelo espaço, aumentando as chances de se encontrar interseções profundas no volume. Este tipo de cena ocorre frequentemente na computação gráfica [1, 3].

O pior caso, por outro lado, ocorre para cenas pequenas e geralmente aglutinadas no espaço. Nelas, as interseções são encontradas pelos primeiros elementos de processamento do volume, conforme mostra a Fig.4.6b. Dessa forma, as demais interseções computadas em paralelo são descartadas e, a princípio, o tempo de execução é equivalente ao do elemento de processamento com mais interseções computadas (mais objetos).



(a) Melhor caso



(b) Pior caso

Figura 4.6: Comparação entre o melhor e o pior caso.

Capítulo 5

GridRT em Software

A implementação da arquitetura GridRT em software segue o modelo arquitetural apresentado no Capítulo 4, com pequenas modificações no modelo de comunicação entre os elementos de processamento. Para a implementação em OpenMP (*Open Multi-Processing*), a comunicação se dará pelo compartilhamento de memória, enquanto que para a implementação em MPI (*Message Passing Interface*) se dará pela troca de mensagens. Note que para ambas implementações, o algoritmo de construção dos volumes uniformes foi omitido.

5.1 OpenMP

O OpenMP [29] é uma API (*Application Programming Interface*) para programação *multi-thread* via memória compartilhada. Uma thread principal cria um conjunto de threads subordinadas e a tarefa é executada em paralelo. Esta criação de threads ocorre por meio de uma diretiva do pré-processador, que indicará a seção do código que deverá executar em paralelo, de acordo com a linha 3 do Algoritmo 5.1. Nela, as variáveis *a* e *b* são privadas (únicas para cada thread) enquanto a variável *c* é compartilhada entre todas as threads.

```
1 início
      /* seção sequencial                                          */
2     int a,b,c;
3     #pragma omp parallel private(a, b) shared(c)
4     início
         /* seção paralela executada por todas as threads          */
5     fim
      /* seção sequencial                                          */
6 fim
```

Algoritmo 5.1: Exemplo OpenMP

No exemplo da Fig.5.1, uma tarefa principal (sequencial) se divide em sub-

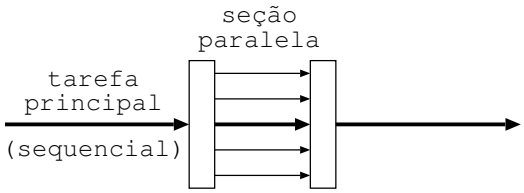


Figura 5.1: Criação de sub-tarefas pelo OpenMP.

```
   Entrada: cena, raio
   Saída: cor do pixel
 1 numcells := grid.getNumCells();
 2 omp_set_num_threads(numcells) ;                    /* set #threads */
 3 shared_res[numcells] ;                  /* resultado de cada thread */
 4 para cada raio faça
 5     lista := grid.traverseGrid(ray) ;     /* lista de atravessamento */
 6     #pragma omp parallel private(tid, resultado)
 7     início
 8         tid := omp_get_thread_num() ;                 /* thread id */
 9         triangles := grid.getTriangleArray(tid) ;    /* thread objs.  */
10         para cada triângulo faça
11             resultado := interseção(raio,triângulo);
12             se resultado ≠ null então
13                 shared_res[tid] := resultado;
14             i := 0;
15             enquanto lista[i] ≠ tid faça
16                 se shared_res[lista[i]] ≠ null então
17                     shared_res[tid] := null;
18                     break ;    /* na verdade, sair do para interno */
19                 i := i+1;
20     fim
21     Continuar a computação;
```

Algoritmo 5.2: Algoritmo de Traçado de Raios em OpenMP

tarefas, que ao final da seção paralela são sincronizadas e terminadas, restando apenas a tarefa principal.

Diante disso, a implementação da arquitetura GridRT em OpenMP modela cada elemento de processamento em uma thread, que fica responsável por calcular interseções entre um raio e sua parte da cena. Após cada teste de interseção, a thread em questão realiza uma busca no vetor de resultados compartilhados para verificar se deve ou não interromper sua computação, conforme o Algoritmo 5.2. Os únicos dados privados são o identificador da thread e o resultado da interseção do elemento de processamento. Os demais dados são compartilhados.

No Algoritmo 5.2, para cada raio a ser processado, a thread principal o atravessa pelo volume uniforme (grid) a fim de determinar a lista de voxels (threads) que serão

acessados. Em seguida, inicia-se a bifurcação de tarefas, pelas quais uma thread identificada por *tid* ficará responsável (linha 6). Depois, cada thread acessa a parte da cena correspondente ao seu identificador (linha 9) e, para cada triângulo, realiza o cálculo de interseção. Caso uma interseção seja encontrada, ela é armazenada em um vetor compartilhado (*shared_res*), indexado pelo identificador da thread. Tal vetor servirá para que cada thread possa verificar se alguma outra anterior a ela na lista já encontrou uma interseção. Em caso afirmativo, a tarefa em questão descartará sua interseção e abortará os cálculos restantes (linha 18). O algoritmo completo é contemplado no Apêndice A.

A Fig.5.2 apresenta um exemplo de execução em OpenMP. Nela, uma cena foi subdividida em quatro regiões, indexadas de 0 a 3, e a cada uma delas foi associado um conjunto de triângulos, de acordo com o algoritmo de construção do volume uniforme. Suponha a lista de atravessamento $L = (2, 0)$ e que uma interseção foi encontrada pela tarefa 2. Logo, a tarefa 0 interromperá sua execução no momento em que verificar a lista L e descobrir que a tarefa 2 já encontrou uma interseção.

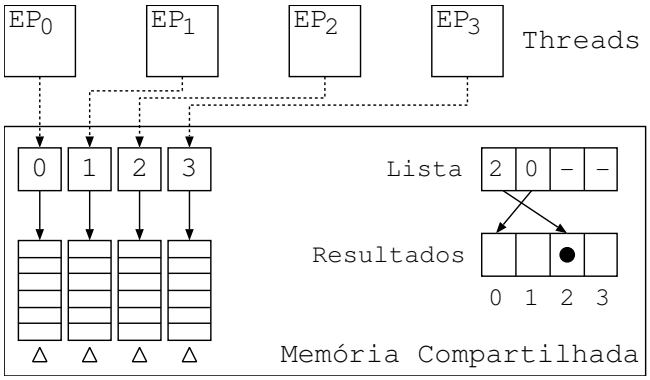


Figura 5.2: Modelo de implementação OpenMP.

5.2 OpenMPI

O OpenMPI [30] é uma API para programação paralela por troca de mensagens entre processos de uma mesma máquina ou entre processos de máquinas remotas, como na Fig.5.3. Tais mensagens podem ser bloqueantes ou não-bloqueantes. No primeiro caso, o processo fica bloqueado aguardando pelo envio ou recebimento da mensagem, enquanto que no segundo caso o processo continua imediatamente após colocar a mensagem no buffer de saída ou ler do buffer de entrada. Neste último caso, se a mensagem não estiver presente, o processo continua a execução do algoritmo, sendo necessária uma verificação posterior da mensagem requisitada.

Neste modelo paralelo por troca de mensagens, os elementos de processamento são mapeados em *Processos* e, geralmente, um programa paralelo em MPI é descrito

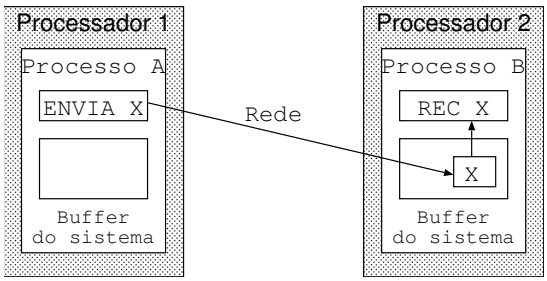


Figura 5.3: Modelo de programação paralela por troca de mensagens.

conforme o Algoritmo 5.3. No início, todos os processos devem invocar a função inicializadora *MPI_Init* e, também, as funções *MPI_Comm_size* e *MPI_Comm_rank* para determinar o número total de processos e o respectivo identificador do processo em questão. O mesmo ocorre no final com a função MPI_Finalize. Nota-se que um processo encarrega-se de ler os dados de entrada do algoritmo e distribuí-los aos demais processos. Durante a execução, os processos devem ou não trocar mensagens entre si dependendo da implementação paralela da aplicação.

```
1 MPI_Init(...);
2 MPI_Comm_size(MPI_COMM_WORLD,&size);
3 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
4 se rank = 0 então
      /* O Processo Mestre lê os dados de entrada e os distribui
         para os demais processos                                  */
      /* Computação e Troca de mensagens...                        */
5 senão
      /* O Processo Escravo recebe os dados do Mestre e processa
         sua parte em paralelo com os demais                       */
      /* Computacão e Troca de mensagens...                        */
6 MPI_Finalize();
```

Algoritmo 5.3: Exemplo MPI

A implementação em MPI da arquitetura GridRT é mais semelhante ao modelo arquitetural do Capítulo 4, pois os sinais de interrupção são simulados através de mensagens. Também, um processo fica encarregado de criar os raios primários, atravessá-los e enviá-los juntamente com a lista de atravessamento aos processos que farão parte da computação dos respectivos raios. Portanto, um processo adicional é criado especificamente para estas funções e é aquele que possui o maior identificador entre os demais. O Algoritmo 5.4 apresenta uma visão geral do GridRT paralelo em MPI. Na linha 5, inicia-se a parte do algoritmo referente ao processo principal, enquanto que na linha 10 inicia-se o referente aos demais processos.

Cada processo escravo recebe um raio e a lista correspondente. Se o processo estiver identificado na lista, o mesmo dá inicio aos testes de interseção. Note que, ao

```
   Entrada: cena, raio
   Saída: cor do pixel
 1 MPI_Init(&argc,&argv);
 2 MPI_Comm_size(MPI_COMM_WORLD,&size);
 3 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 4 Define MESTRE (size-1);
 5 se rank = MESTRE então
 6     Cria os raios primários;
 7     Atravessa cada raio no Grid;
 8     Envia cada raio aos processos;
 9     Envia a lista correspondente a cada raio aos processos;
10 senão
11     para cada raio recebido faça
12         se id. do processo estiver listado então
13             resultado := intersecão(triângulo);
14             se resultado ≠ null então
15                 Determina a posição do processo na lista;
16                 se primeiro então
17                     Envia mensagem de interrupção ao próximo;
18                 se último então
19                     Recebe uma mensagem do processo anterior;
20                     Lê a mensagem e interrompe se necessário;
21                 senão
22                     Recebe uma mensagem do processo anterior;
23                     Encaminha a mensagem ao próximo da lista;
24                     Lê a mensagem e interrompe se necessário;
25             se resultado ≠ null então
26                 Continua a computação;
27 MPI_Finalize();
```

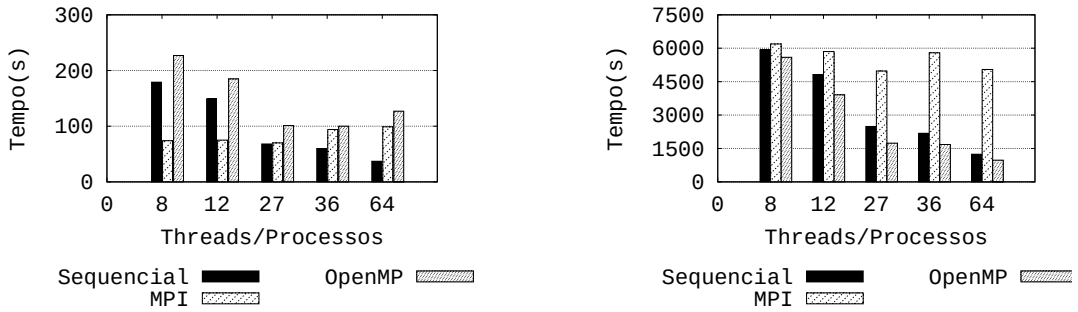
Algoritmo 5.4: Algoritmo de Traçado de Raios em MPI

contrário do que ocorre no Algoritmo 5.2 (OpenMP), todos os testes de interseção são realizados para então iniciar a troca de mensagens. Desta forma, reduz-se a quantidade de troca de mensagens que, do contrário, seria elevada. Ao término das interseções, cada processo age de acordo com a sua posição na lista de atravessamento. Aquele que for o primeiro da lista, verifica o resultado da sua interseção e envia uma mensagem de interrupção ou término para o seguinte. O último, por sua vez, aguarda uma mensagem de interrupção ou término para, então, decidir se mantém seu resultado ou descarta-o. Finalmente, os intermediários da lista devem aguardar uma das mensagens do anterior, repassá-las adiante e decidir se mantém ou descarta seu resultado. O algoritmo completo pode ser visto no Apêndice B.

5.3 Resultados

Nesta seção são apresentados os resultados da execução dos algoritmos OpenMP e MPI, nas Figuras 5.4a e 5.4b, para 320×240 raios primários. Tais resultados foram obtidos em uma arquitetura Intel Core i7 2.26GHz, com quatro núcleos capazes de executar até oito threads em paralelo. Cada execução refere-se a uma configuração do volume uniforme (malha) nos tamanhos de 8, 12, 27, 36 e 64 voxels. Consequentemente, o número de threads/processos deve corresponder ao tamanho do volume definido, mantendo a relação de 1:1 entre voxels e elementos de processamento.

Apesar da arquitetura suportar até oito processos em execução paralela, no máximo oito estarão de fato processando cálculos de interseção, em um total de 64 processos. Os demais estarão "desativados" (em *espera ocupada*). Isto se deve ao número de elementos da lista de atravessamento, cujo tamanho máximo para estas configurações é menor que oito. Ou seja, todos os raios atravessados nos volumes uniformes de tamanho 8, 12, 27, 36 e 64 poderão ativar no máximo oito elementos de processamento.



(a) 67K triângulos.

(b) 1.20M triângulos.

Figura 5.4: Traçado de raios sequencial vs. paralelo.

Os resultados apresentados na Fig.5.4a comparam a execução sequencial e paralela para uma cena de 67K triângulos, que representa apenas um coelho (*Stanford Bunny*). Neste caso, o algoritmo sequencial demonstrou uma performance melhor em relação ao algoritmo OpenMP, especialmente para um volume uniforme de 64 voxels (threads). Isto se deve ao pequeno volume de dados a serem paralelizados, que não compensam o custo da paralelização: a cada cálculo de interseção, o vetor de resultados compartilhados deve ser analisado. Por outro lado, o algoritmo MPI demonstrou uma melhora inicial em relação ao sequencial, para configurações de 8 e 12 voxels (processos). A partir de então, o algoritmo sequencial vence o MPI. Tal desempenho inicial superior é devido à execução paralela de todos os cálculos de interseção, para que depois seja determinado o menor deles. Logo, o custo de

sincronização é menor. No entanto, na medida em que cresce o tamanho da malha (volume uniforme), também aumenta a quantidade de mensagens bloqueantes entre os processos e, consequentemente, diminui-se a performance do algoritmo.

Para cenas maiores e esparsas, os resultados são promissores. A Fig.5.4b compara a execução sequencial e paralela para uma cena de 1.2M triângulos, que representa 18 coelhos (*Stanford Bunny*) espalhados pelo espaço (Fig.5.6b). Neste caso, o algoritmo paralelo OpenMP apresentou um desempenho melhor aos demais, atendendo às expectativas mostradas na Seção 4.3: cenas grandes e esparsas favorecem o cálculo em paralelo de interseções, antecipando os resultados das interseções dos elementos de processamento identificados pela lista de atravessamento de um raio. Os resultados da Fig.5.5 indicam que, na medida em que o tamanho da cena cresce, a performance do algoritmo paralelo OpenMP melhora em relação ao sequencial, para cenas maiores que 134K. Estes resultados foram obtidos em uma arquitetura Athlon X2 de dois núcleos.

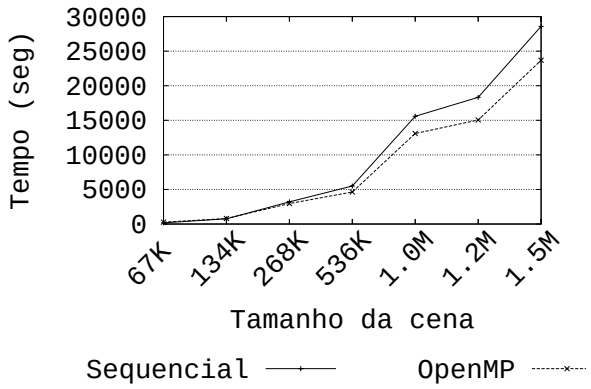
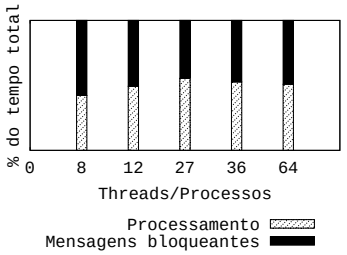
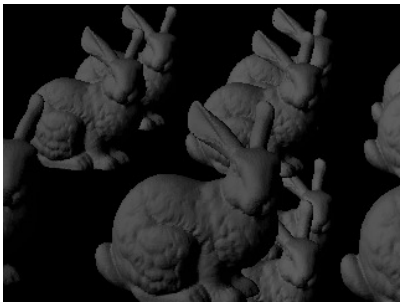


Figura 5.5: Relação entre o aumento da cena e a performance do algoritmo OpenMP.

No entanto, o algoritmo paralelo em MPI não correspondeu aos resultados esperados. O uso de mensagens bloqueantes para a troca de informações (sinalizações) prejudica a performance geral do algoritmo, conforme a Fig.5.6a. Portanto, este algoritmo precisa ser revisado e, possivelmente, alterado para utilizar mensagens não-bloqueantes. A Tabela 5.1 apresenta os tempos de execução para os algoritmos descritos nesta seção.



(a) Mensagens bloqueantes em MPI.



(b) 18 objetos *Stanford Bunny*.

Figura 5.6: *Overhead* das mensagens bloqueantes e renderização de 18 coelhos.

Tabela 5.1: Tempos de execução* para renderização de cenas de 67K e 1.2M triângulos.

Grid	Cena de 67K			Cena de 1.2M		
	Sequencial	MPI	OpenMP	Sequencial	MPI	OpenMP
8	179	74	227	5931	6191	5594
12	149	75	185	4819	5851	3909
27	68	70	101	2489	4977	1740
36	60	94	100	2182	5797	1678
64	37	99	127	1242	5047	975

*Todos os tempos estão em segundos. Resolução: 320×240.

Capítulo 6

GridRT em Hardware

Assim como a implementação em software apresentada no capítulo anterior, a implementação em hardware da arquitetura GridRT será descrita conforme o modelo no Capítulo 4. Apenas quatro processadores serão implementados e conectados ao processador *MicroBlaze* da Xilinx™, devido a restrições de recursos do hardware na FPGA usada. No entanto, uma seção será dedicada a demonstrar a escalabilidade da arquitetura.

O hardware a que se refere este trabalho é uma FPGA (*Field Programmable Gate Array*), modelo Virtex-5 da Xilinx™. Portanto, uma breve explicação da FPGA e de suas características mais importantes será feita. Além disso, o modelo de paralelismo implementado em hardware, por meio de sinais de interrupção, também merece atenção especial. A implementação em hardware foi completamente descrita em VHDL [15](*Very high speed integrated circuits Hardware Description Language*) e o código para cada um dos componentes pode ser encontrado no Apêndice C. Todas as simulações foram feitas no ModelSim Xilinx™ Edition 6.3c [28].

6.1 FPGA - Field Programmable Gate Arrays

As FPGAs são dispositivos lógicos programáveis, nos quais uma matriz de blocos lógicos configuráveis (*Configurable Logic Blocks* - CLBs), conectados por canais também programáveis, pode ser configurada para desempenhar um conjunto de funções lógicas e dar origem a qualquer tipo de sistema digital [50]. Dependendo da sua densidade em termos de CLBs, uma FPGA pode ser utilizada para criar até mesmo processadores complexos e protótipos de novas arquiteturas, mesclando flexibilidade e desempenho adequados às necessidades do sistema.

Tais dispositivos reconfiguráveis são geralmente empregados na etapa de prototipação do sistema, que precede a etapa de produção de um ASIC (*Application-Specific Integrated Circuits*), cujo custo de produção é proibitivo para a criação de um único chip para fins de validação. Logo, se necessário, o protótipo pode ser corrigido e

atualizado, por meio de ajustes na sua especificação descrita em uma linguagem de descrição de hardware [15, 51] (*Hardware Description Language* - HDL). Uma especificação descreve cada um dos componentes do sistema, em diferentes níveis de abstração: seja no nível de portas lógicas ou no nível de componentes mais complexos e no nível de comportamento. Em seguida, um software sintetizador de hardware, como o XST [33](*Xilinx Synthesis Technology*), se encarrega de compilar a especificação do hardware em funções lógicas que possam ser mapeadas nos blocos lógicos da FPGA. O processo todo pode ser resumido em três etapas:

- Mapeamento: as funções lógicas são mapeadas em CLBs;
- Alocação: as CLBs são implementadas na FPGA;
- Roteamento: realiza as conexões entre as CLBs.

As vantagens de se utilizar uma FPGA para a implementação de um sistema digital vão além dos benefícios da prototipação. Dependendo da quantidade de recursos (blocos lógicos) disponíveis em um chip, é possível replicar diversos componentes do sistema e realizar operações da arquitetura em paralelo e em *pipeline*. Logo, apesar das baixas frequências de operação das FPGAs (em torno de 500MHz), uma aplicação paralela mapeada em hardware pode atingir um fator de aceleração muitas vezes superior ao de implementações em processadores modernos [16, 17].

6.1.1 Blocos Lógicos

As células lógicas (CLs) são os elementos fundamentais de uma FPGA. Geralmente estas são agrupadas em parcelas (*Slices*), que por sua vez são agrupadas em blocos lógicos configuráveis (CLBs), formados por 2 ou 4 slices dependendo do tipo da FPGA. Um sistema é construído a partir da cofiguracão dos CLBs disponíveis. Geralmente, os blocos lógicos são compostos de uma tabela de pesquisa (*look-up table* - LUT) de 4 a 6 entradas, circuitos de seleção (multiplexadores) e flip-flops, de acordo com o esboço na Fig.6.1.

Durante a programação de um CLB, a LUT pode ser configurada para realizar alguma função lógica e o flip-flop permite que o CLB se comporte de maneira síncrona (com uso do clock) ou combinatorial, de acordo com a escolha do multiplexador. A saída conecta-se a qualquer segmento adjacente, que também conecta-se a canais programáveis entre os CLBs. Na periferia, os blocos de E/S (*Input Ouput Blocks* - IOBs) permite realizar a interface do sistema com o mundo externo a FPGA.

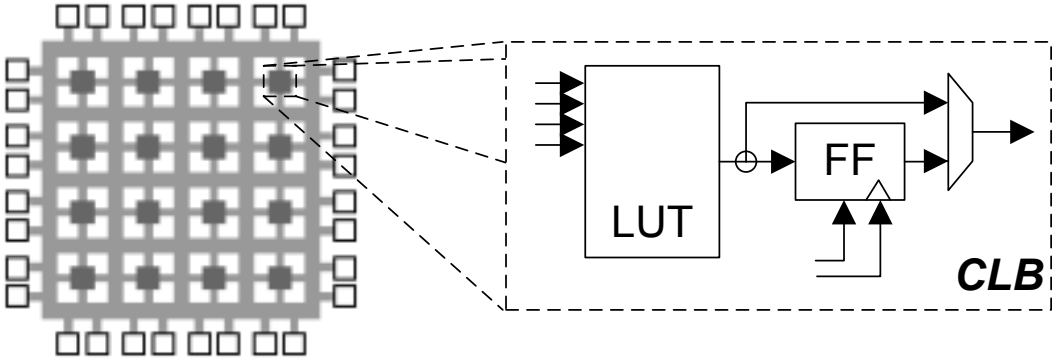


Figura 6.1: Arquitetura de uma célula lógica.

6.1.2 Blocos de Memória

Algumas FPGAs possuem blocos de memória incorporados, geralmente de 36Kb cada. Estes blocos são disponibilizados com o propósito de compor partes do sistema que está sendo projetado, tais como filas, pilhas, vetores, etc. Além disso, os blocos podem ser unidos para compor memórias ainda maiores, com até 4 portas de endereçamento, dependendo do fabricante e do modelo da FPGA.

No entanto, essas memórias não estão disponíveis em larga escala, mas apenas em quantidades pequenas, em torno de 10Mb. Se necessário, blocos de memória também podem ser projetados usando os próprios CLBs, consumindo blocos lógicos do dispositivo (principalmente LUTs). Estas memórias são chamadas de *memórias distribuídas*.

6.1.3 Xilinx Virtex-5

A plataforma alvo de implementação da arquitetura GridRT é a placa de desenvolvimento ML507 [52], que possui uma FPGA do tipo Virtex-5[34] modelo XC5VFX70T. A Tabela 6.1 apresenta os principais recursos desta FGPA, enquanto a Fig.6.2 exibe um diagrama de blocos da plataforma ML507. Nota-se nesta figura que a FPGA é ligada aos diversos dispositivos da plataforma. Dentre estes, será utilizado neste trabalho a porta serial RS232 para comunicação de dados (apresentação de resultados) a uma estação de trabalho.

Cada CLB da Virtex-5 é organizada em dois *slices*. O mais simples, chamado de *SLICEL*, possui quatro LUTs de seis entradas cada, quatro flip-flops e vários multiplexadores. Estes componentes são utilizados para prover funções lógicas, aritméticas e de armazenamento. Outros *slices*, conhecidos por *SLICEM*, dão suporte a memórias distribuídas e registradores de deslocamento. Além disso, um bloco de memória tem um tamanho de 36 Kb e, ao total, existem 148 deles.

Tabela 6.1: Principais recursos da FPGA XC5VFX70T.

Recursos		XC5VFX70T
Lógica	Slices	11200
	Células lógicas	71680
	Flip-Flops	44800
Memória	Distribuída	820 Kb
	BRAM	5328 Kb
Incorporado em Hardware	DSP48E	128
	PowerPC 440	1

Os recursos incorporados à FPGA são componentes físicos de propriedade intelectual, como o PowerPC e os blocos processadores de sinais digitais (DSP48E). Portanto, eles não consomem blocos lógicos da FPGA e podem atingir frequências de 450 a 550 MHz. Assim, o processador PowerPC pode ser programado para desempenhar alguma função secundária, enquanto um componente dedicado em FPGA executa a função principal através de um projeto paralelo e mais eficiente. Mais adiante, na Seção 6.2.1, um esquema semelhante será discutido para a implementação do GridRT.

Os blocos de processamento digital carregam um multiplicador 25×18, além de um somador e um acumulador. Logo, eles também podem ser empregados para acelerar operações aritméticas ou para atender a outros propósitos.

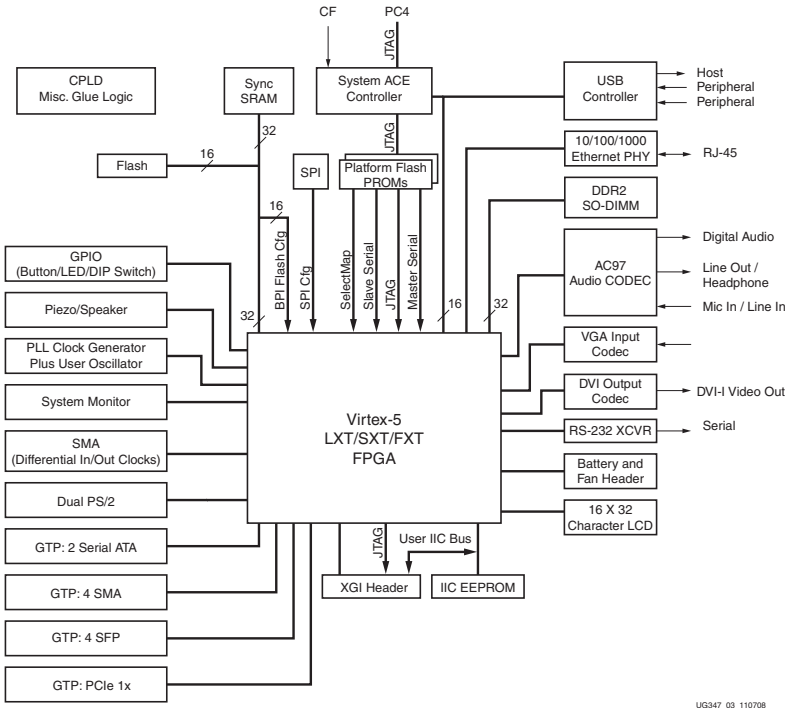
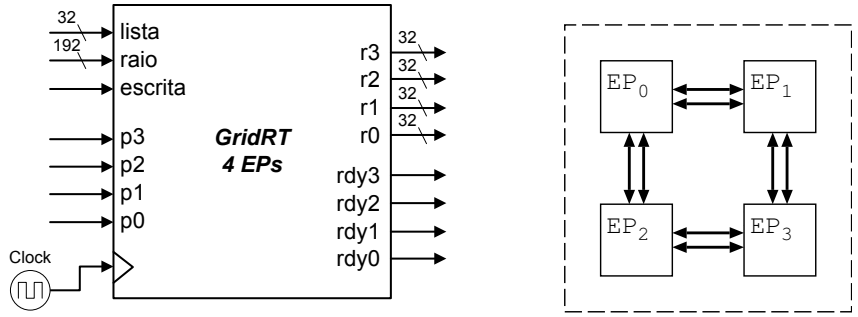


Figura 6.2: Plataforma de desenvolvimento ML507 [52].

6.2 Implementação em FPGA

A Arquitetura GridRT é encapsulada em um componente previsto para auxiliar a computação do algoritmo de traçado de raios em hardware, através de cálculos de interseção em paralelo. Neste componente existe um conjunto de processadores que realizam a computação paralela dos raios recebidos, sendo que a cena é repartida entre eles, semelhante à estrutura de Volumes Uniformes descrita no Capítulo 4. Cada processador é interligado ao seu vizinho direto, por meio de duas linhas de interrupção, a fim de que sejam capazes de decidir a respeito do resultado correto, ou seja, qual o menor ponto de interseção computado. A Fig.6.3a é um exemplo de componente GridRT e sua interface. Já a Fig.6.3b exibe uma configuração incluindo quatro processadores.



(a) O componente GridRT

(b) Implementação de 4 processadores

Figura 6.3: As interfaces do componente GridRT e sua configuração de 4 processadores.

O raio é recebido pelo componente, acompanhado da lista de atravessamento. Note que os dados da cena já devem se encontrar repartidos entre cada processador, nas suas respectivas memórias. Isto é feito diretamente na atualização das memórias, durante a etapa de codificação. Cada raio tem uma largura de 6×32 bits, que representam as coordenadas de origem e destino do raio. Já a lista de atravessamento tem uma largura de 4×8 bits, que representam o total de até 8 identificadores de 4 bits, suficiente para suportar uma arquitetura de até 8 processadores. Um identificador é um inteiro com sinal, cujos valores positivos são usados para selecionar o caminho percorrido pelas interrupções, de acordo com a ordem dos identificadores na lista de atravessamento. Os valores negativos são usados pelo primeiro e último processador da lista, para determinar o caminho das interrupções nas extremidades. Maiores detalhes a respeito das interrupções são encontrados na Seção 6.2.4.

Um sinal de escrita armazena estes dados de entrada nos registradores dos elementos processadores e, em seguida, os mesmos devem ser avisados através dos sinais p_3, p_2, p_1 e p_0. Isto é feito para que exista sincronia entre o sinal de escrita dos dados

e o início do processamento. Além disso, tais sinais também servem para indicar quais os elementos processadores que estão identificados na lista. Para isso, a lista é pré-processada e os respectivos sinais produzidos, conforme a forma de onda na Fig.6.4. O restante dos elementos processadores permanecerão ociosos (em espera ocupada), até que a computação paralela termine e uma outra escrita de dados seja feita nos registradores, indicando a existência de um novo raio e uma nova lista.

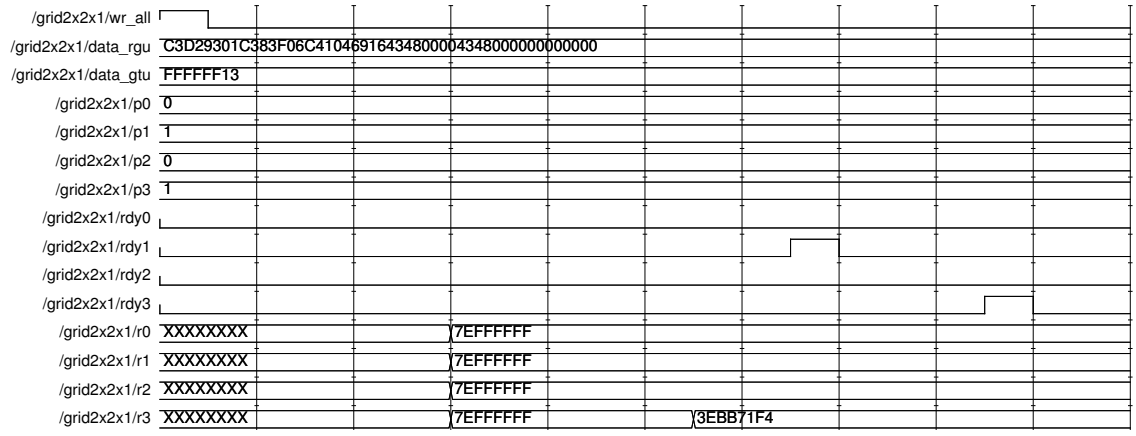


Figura 6.4: Simulação do GridRT.

Terminada a computação, aqueles processadores que foram ativados retornam um sinal de término, cada um: rdy_3, rdy_2, rdy_1 e rdy_0. Os resultados também se encontram disponíveis em r_3, r_2, r_1 e r_0. No exemplo apresentado na forma de onda da Fig.6.4, observe que os elementos processadores 1 e 3 foram ativados. Mais adiante, o elemento processador 3 encontrou um resultado, em r_3, e interrompeu o elemento processador 1, segundo suas respectivas ordens na lista de atravessamento. Ambos retornam os sinais de término rdy_1 e rdy_3. A maneira pela qual os processadores decidem qual o resultado correto também é descrita na Seção 6.2.4.

6.2.1 O MicroBlaze

O MicroBlaze [32] é um microprocessador de propriedade intelectual da Xilinx^{TM}, que pode ser sintetizado em dispositivos reconfiguráveis. Ele opera com um conjunto de instruções reduzido (RISC) e é otimizado para ser implementado em FPGAs do próprio fabricante, com suporte a operações em ponto flutuante e pipelining de até cinco estágios, além de outras características. Como este microprocessador é sintetizado na FPGA, o mesmo consome uma parte dos recursos disponíveis, o que é uma desvantagem, como será explicado na Seção 6.3.

Porém, sua principal vantagem encontra-se na interface de comunicação ponto-a-ponto, conhecida por *Fast Simplex Link* (FSL) [53], que permite a conexão de um componente qualquer, identificado como co-processador, ao microprocessador. Estes

canais dedicados são ideais para estender a capacidade computacional do microprocessador através de componentes destinados a realizar uma determinada computação com mais rapidez, geralmente através de paralelismo.

Logo, tendo em mãos o componente acelerador do algoritmo de traçado de raios, que executará o caminho crítico do algoritmo, ou seja, os cálculos de interseção, é possível alimentá-lo com dados oriundos do MicroBlaze via um canal FSL. O conjunto de instruções do microprocessador possui instruções para ler dados da porta FSL e gravá-los em um registrador do processador, assim como instruções para enviar dados de um registrador para a porta FSL. Estas funcionalidades apresentam variantes bloqueantes e não-bloqueantes[32]. Além disso, outros componentes podem ser conectados ao microprocessador por meio de outros barramentos, de acordo com a Fig.6.5.

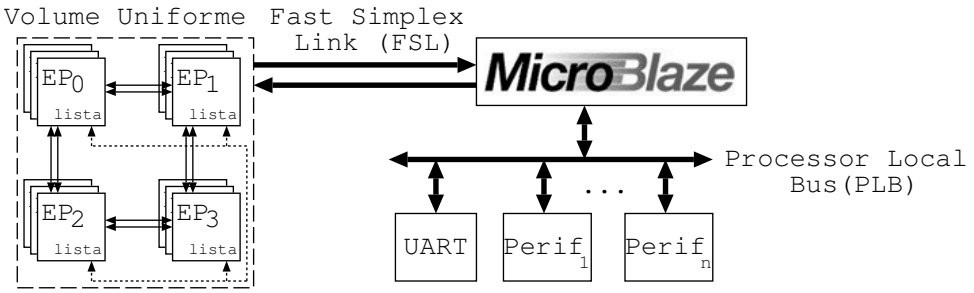


Figura 6.5: O MicroBlaze conectado ao GridRT e periféricos.

Deste modo, o resultado da computação pode ser transmitido para um estação de trabalho por meio de uma interface UART (*Universal Asynchronous Receiver/Transmitter*) e interpretado por um software na contraparte. O *XPS UART Lite* [54] é um exemplo de componente de propriedade intelectual da Xilinx™ que conecta-se ao barramento PLB (*Processor Local Bus*) do MicroBlaze e realiza esta interface de comunicação entre o processador e a estação de trabalho.

6.2.2 Comunicação via FSL

Um canal FSL disponibiliza um meio de comunicação dedicado e uni-direcional entre dois componentes quaisquer. Este canal é baseado em filas (FIFO) e pode conter entre 1 e 8K palavras de 32 bits. Um componente age como mestre quando insere dados na fila e o outro escravo retirando os dados da mesma, conforme a interface descrita na Fig.6.6.

Uma operação de inserção na fila é controlada pelo sinal FSL_M_Write, que escreve o dado do FSL_M_Data quando o sinal FSL_M_Full não estiver ativo, indicando que a fila não está cheia. Junto com o dado, também é inserido o bit de controle FSL_M_Control, que o acompanha durante toda a fila. Geralmente

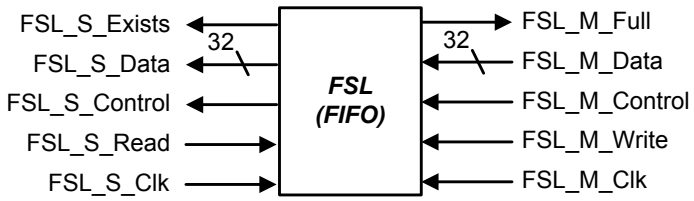


Figura 6.6: Interface FSL.

este bit é usado pelo componente escravo para indicar o início e o término de uma transmissão. A operação de remoção da fila é comandada pelo sinal FSL_S_Read, que lê o dado do FSL_S_Data e o bit de controle FSL_S_Control caso o sinal FSL_S_Exists estiver ativo, indicando a existência de dados na fila. Maiores detalhes sobre a operação da comunicação via FSL em [53].

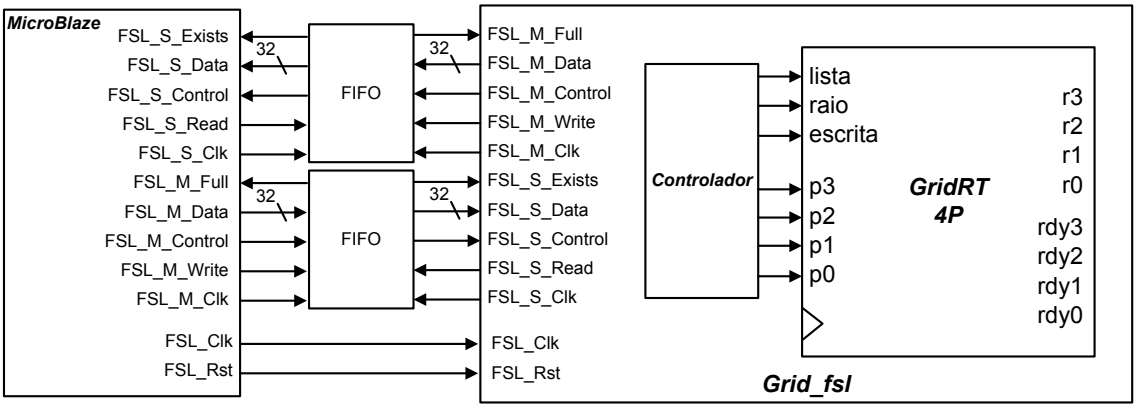


Figura 6.7: Comunicação FSL.

Portanto, dois canais FSL fazem a comunicação entre o MicroBlaze e o componente GridRT, um em cada sentido. Para isso, o microprocessador disponibiliza duas instruções para inserção (*put*) e remoção (*get*) de dados das respectivas filas. Do lado do co-processador, este é encapsulado em um outro componente que realiza as operações de leitura e escrita das filas, através de uma máquina de estados que repassa os dados para o GridRT. Porém, como a transmissão de dados ocorre em palavras de 32 bits, os dados precisam ser lidos, armazenados e então enviados para o componente acelerador. Por exemplo, sabe-se que o raio tem uma largura de 6×32 bits e, por isso, é preciso ler todas as 6 palavras antes de ativar o sinal de escrita para que estes dados sejam disponibilizados ao GridRT. O mesmo é verdadeiro para a operação de escrita dos resultados, uma vez que quatro resultados, que correspondem aos pontos de interseção encontrados por cada processador, precisam ser inseridos por vez. Os sinais FSL_S_Clk e FSL_M_Clk são utilizados para que as operações ocorram em sincronia.

A latência de leitura e escrita do canal FSL é muito pequena. Se as filas não forem implementadas utilizando blocos de memória (BRAMs), o dado inserido pelo mestre se encontra disponível para o escravo logo após a transição negativa do sinal de escrita. Caso contrário, a latência é de apenas um ciclo adicional. Analogamente, novos dados são disponibilizados em FSL_S_Data e FSL_S_Control logo após a transição negativa do sinal de leitura. O código VHDL do componente que realiza a interface FSL entre o MicroBlaze e o GridRT encontra-se no Apêndice C.5.

A programação do microprocessador pode ser feita em ANSI C/C++ e compilada pelo XPS (*Xilinx Platform Studio*), que gera para cada componente incluído os respectivos *drivers* de controle. Neste caso, o MicroBlaze executa uma parte do algoritmo de traçado de raios, deixando o caminho crítico de cálculos de interseção para ser executado pelo hardware dedicado, GridRT. O algoritmo completo que é executado no microprocessador encontra-se no Apêndice D, enquanto a parte principal é apresentada no Algoritmo 6.1.

```
1  para cada pixel da Câmera Virtual faça
       /* gera um raio primário                                    */
2      unsigned input[0] = x0.u;
3      unsigned input[1] = y0.u;
4      unsigned input[2] = z0.u;
5      unsigned input[3] = xd.u;
6      unsigned input[4] = yd.u;
7      unsigned input[5] = zd.u;
       /* atravessa o volume uniforme                              */
8      traverseGrid(&eye, &dest, &p0, &p1,list, LISTSIZE);
9      unsigned input[6] = reduceList(list,LISTSIZE) ;        /* lista */
10     se input[6] != 0xFFFFFFFF então        /* lista não-vazia */
11         grid_fsl(input_id,output_id,input,output);
12     xil_printf("Resultado[1] = 0x%08x",output[1]);
13     xil_printf("Resultado[2] = 0x%08x",output[2]);
14     xil_printf("Resultado[3] = 0x%08x",output[3]);
15     xil_printf("Resultado[4] = 0x%08x",output[4]);
16     eraseList(list,LISTSIZE) ;                    /* apaga a lista */
```

Algoritmo 6.1: Algoritmo de Traçado de Raios no MicroBlaze

Observe que a comunicação FSL (linha 11) ocorre na função *grid_fsl*, cujos dados de entrada e saída são armazenados nos vetores *input* e *output*, respectivamente. Tal função realiza as chamadas às instruções *put* e *get* do canal FSL. Além disso, a função *xil_printf* realiza a comunicação com a estação de trabalho via a UART (Fig.6.5). Nela, os dados são interpretados e impressos na tela.

6.2.3 O Elemento de Processamento

Nas implementações em software, o elemento de processamento foi implementado na forma de Processos e Threads, enquanto que nesta implementação em hardware o elemento de processamento é de fato um processador do estilo MIPS [55], controlado por um pequeno conjunto de instruções em multi-ciclo. Uma visão geral do processador é mostrada na Fig.6.8, na qual os números representam as principais micro-ordens da micro-instrução.

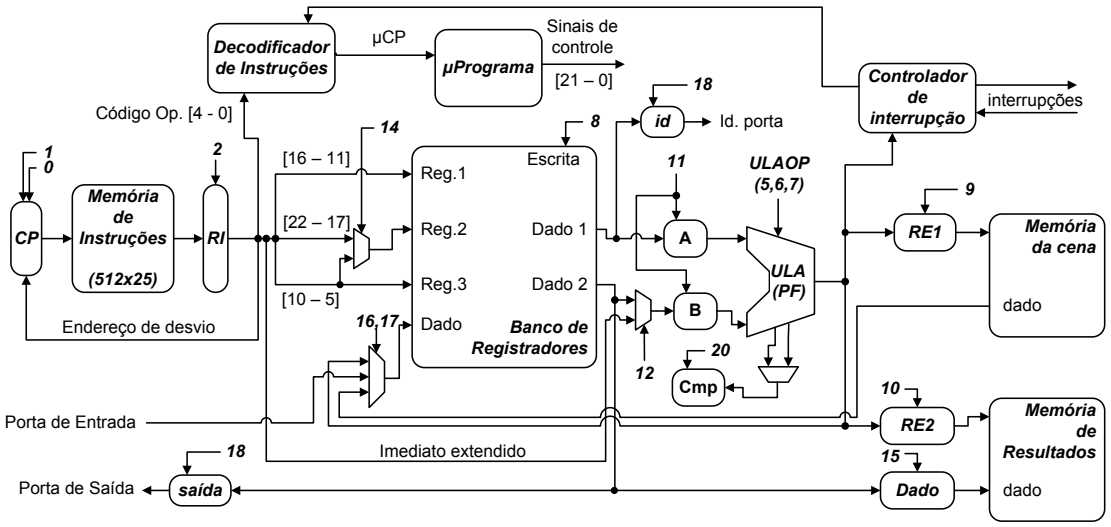


Figura 6.8: O Elemento de Processamento.

Cada processador é capaz de executar um programa de no máximo 512 instruções, o que é suficiente para armazenar o algoritmo de cálculo de interseções. Isto se deve ao pequeno tamanho deste algoritmo e também para fins de economia de recursos, como será mostrado na Seção 6.3.1. Uma instrução é formada de 25 bits e armazenada na memória de instruções, com capacidade de 12.5Kb. Além disso, um micro-programa realiza o controle do elemento processador. Cada micro-instrução tem uma largura de 23 bits e o micro-programa armazena até 128 micro-instruções, o que resulta em uma memória de 2.875 Kb.

A cena é armazenada na *Memória da Cena*, cuja capacidade máxima é de 256 Kb. Uma vez que um triângulo consome 288 bits (9 coordenadas de 32 bits), então a memória é capaz de armazenar aproximadamente 910 triângulos. O resultado da computação, que representa a cor de um pixel no padrão RGB de 32 bits, é armazenado na *Memória de Resultados*, cuja capacidade é de 256 Kb. Logo, 8192 pixels podem ser armazenados nessa memória. Contudo, uma imagem com uma resolução de 320×240 *pixels* não caberia na memória de resultados e, por isso, elas são utilizadas apenas como um repositório temporário de resultados parciais. No futuro, os resultados produzidos por cada elemento processador deverão ser armazenados

em uma memória secundária. O mesmo ocorre para a memória da cena, que não é capaz de armazenar uma cena grande e, portanto, o elemento processador deverá no futuro buscá-las em memória secundária. Todas as memórias apresentadas são implementadas em FPGA através de blocos de memória BRAMs, sem que seja necessário consumir LUTs dos blocos lógicos.

Conjunto de Instruções

Todas as operações são realizadas entre registradores ou entre um registrador e um valor imediato, de acordo com a Tabela 6.2, na qual as instruções em destaque correspondem a instruções para operações em ponto flutuante, com precisão simples (32 bits). Além disso, outras instruções dão suporte a desvios, comparações e E/S de dados.

As instruções têm comprimento de 25 bits e formato fixo, divididos em três tipos conforme a Tabela 6.3. Os primeiros 5 bits da instrução codificam a operação a ser executada, enquanto os demais identificam os registradores de origem e destino, assim como valores imediatos e endereços de desvio.

Tabela 6.3: Formato das Instruções.

Formato	Instrução [0-24]				
R	[0-4] Operação	[5-10] R_D	[11-16] R_S	[17-22] R_T	[23-24] -
I	[0-4] Operação	[5-10] R_D	[11-16] R_S	[17-24] Imediato	
J	[0-4] Operação	[5-13] Endereço			[14-24] -

As instruções do tipo R realizam operações entre dois registradores, identificados pelos campos R_S e R_T, e armazenam o resultado da operação em um terceiro registrador, identificado por R_D. Neste caso, os dois bits restantes são inutilizados. Já as instruções do tipo I realizam operações entre um registrador e um valor imediato, identificados pelos campos R_S e $Imediato$. Este último tem comprimento de 8 bits, mas é extendido para 32 bits no ato da operação. O registrador identificado por R_D armazena o resultado da operação. Por fim, as instruções do tipo J são utilizadas para desvios no programa e apenas identificam o endereço para o qual o contador de programa deverá apontar. O campo de endereço tem comprimento máximo de 9 bits, uma vez que a memória de instruções armazena no máximo 512 instruções. O restante dos bits da instrução são inutilizados.

Tabela 6.2: Conjunto de Instruções disponíveis para cada EP.

Instrução	**Formato**	**Operação**	**Descrição**
add	R	$R_d = R_t + R_s$	Soma o conteúdo de R_t e R_s em R_d
sub	R	$R_d = R_t - R_s$	Subtrai o conteúdo de R_t e R_s em R_d
and	R	$R_d = R_t$ 'e' R_s	Lógica 'E' entre R_t e R_s em R_d
or	R	$R_d = R_t$ 'ou' R_s	Lógica 'OU' entre R_t e R_s em R_d
addfp	R	$R_d = R_t + R_s$	Soma o conteúdo de R_t e R_s em R_d
subfp	R	$R_d = R_t - R_s$	Subtrai o conteúdo de R_t e R_s em R_d
mulfp	R	$R_d = R_t \times R_s$	Multiplica o conteúdo de R_t e R_s em R_d
divfp	R	$R_d = R_t/R_s$	Divide o conteúdo de R_t e R_s em R_d
sqrtfp	R	$R_d = \sqrt{R_t}$	Calcula a raiz quadrada de R_t em R_d
addi	I	$R_d = R_t + Imm$	Adiciona a R_t um valor imediato em R_d
load	I	$R_d = mem[R_t + Imm]$	Carrega o conteúdo do endereço $[R_t + Imm]$ em R_d
st	I	$mem[R_t + Imm] = R_s$	Armazena o conteúdo de R_s no endereço $[R_t + Imm]$
input	R	$R_d = portid[R_s]$	Carrega no registrador R_d o conteúdo da porta de E/S R_s
output	R	$portid[R_s] = R_t$	Carrega o conteúdo do registrador R_t na porta de E/S R_s
cmp	R	$R_d(<, >, =)R_t$	Compara o conteúdo de R_t e R_s
cmpfp	R	$R_d(<, >, =)R_t$	Compara o conteúdo de R_t e R_s
jl	J	$CP = Addr$	Salta para o endereço Addr se $R_d < R_t$
je	J	$CP = Addr$	Salta para o endereço Addr se $R_d = R_t$
jg	J	$CP = Addr$	Salta para o endereço Addr se $R_d > R_t$
j	J	$CP = Addr$	Salta para o endereço Addr

Contudo, nota-se que as instruções de comparação não geram um resultado para ser armazenado em um registrador de destino, apesar destas instruções serem classificadas como do tipo R. Os dois registradores a serem lidos estão sempre posicionados nos campos R_S e R_T e, por isso, um multiplexador realiza a troca de posição dos campos R_D e R_T, possibilitando a leitura de dois registradores pelas instruções de comparação. Esta troca é realizada pelo sinal 14 da micro-instrução.

Ao total, existem 64 registradores de 32 bits em um elemento processador, dos quais somente 46 estão disponíveis para escrita pelo conjunto de instruções. Os outros registradores são utilizados para armazenar valores constantes e dados do raio a ser processado, assim como sua lista de atravessamento, conforme a Tabela 6.4. Esta grande quantidade de registradores se faz necessária, pois o algoritmo de interseção também precisa de pelo menos 25 registradores para armazenar resultados intermediários da computação. Além disso, 64 registradores oferecem maior flexibilidade para uma possível expansão da lista de atravessamento e, consequentemente, da arquitetura.

Tabela 6.4: Registradores especiais.

Registrador	**Dado**
0	Constante 0
1	Constante 1.0 (ponto flutuante)
2	Identificador do EP
25	Maior número possível em ponto flutuante
26	Coordenada X da origem do raio
27	Coordenada Y da origem do raio
28	Coordenada Z da origem do raio
29	Coordenada X do destino do raio
30	Coordenada Y do destino do raio
31	Coordenada Z do destino do raio
34-41	Lista de atravessamento
42	Identificador do EP Anterior
43	Identificador do EP Posterior

As memórias da cena e dos resultados são acessadas por meio das instruções de *load* e *store*, respectivamente. No entanto, os registradores de endereço de ambas as memórias têm uma largura de 13 bits, visto que as memórias têm uma capacidade de somente 256Kb. Portanto, o endereço contido em um registrador de 32 bits é truncado para os 13 bits menos significativos. No futuro, caso as memórias possam conter mais dados, estes registradores também serão expandidos.

Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética (ULA) do elemento processador realiza as operações necessárias para a execução das instruções. Nela estão embutidos componentes somadores, subtratores, multiplicadores e comparadores. Além disso, existem quatro unidades para cálculos em ponto flutuante da Xilinx™ [56], uma para cada operação: soma, multiplicação, divisão e raiz quadrada. Porém, a subtração é feita pela inversão de um dos operandos de entrada, através dos sinais 3 e 4 da micro-instrução.

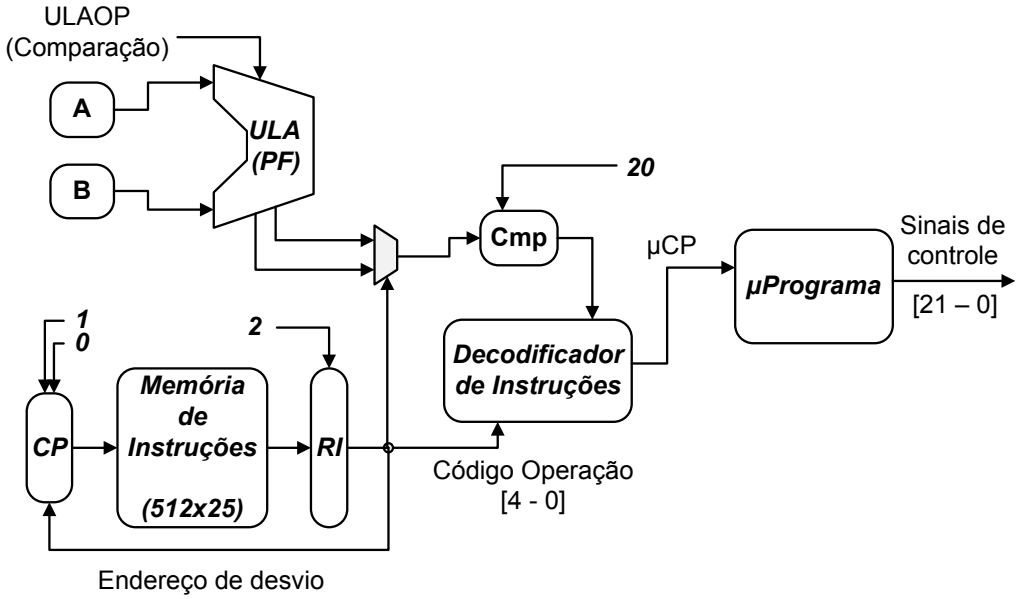


Figura 6.9: Unidade Lógica e Aritmética.

As operações realizadas pela ULA são codificadas pelos sinais 5,6 e 7 da micro-instrução, identificados pelo campo UlaOP da Tabela 6.5.

Tabela 6.5: Códigos de operação da ULA.

UlaOP	Operação	Descrição
000	add	Soma os operandos A e B
001	sub	Subtrai os operandos A e B
010	and	Operação lógica 'e' dos operandos A e B
011	or	Operação lógica 'ou' dos operandos A e B
100	addfp	Soma em ponto flutuante dos operandos A e B
101	mulfp	Multiplicação em ponto flutuante dos operandos A e B
110	divfp	Divisão em ponto flutuante dos operandos A e B
111	sqrtfp	Raiz quadrada em ponto flutuante do operando A

A representação dos dados em ponto flutuante segue o padrão IEEE-754 [57], em precisão simples. Assim, um valor é retratado em 32 bits, dos quais o mais

significativo indica se o número é negativo (1) ou positivo (0). O restante é divido em duas partes: os oito mais significativos representam o expoente do número, enquanto os demais menos significativos representam a mantissa. Por isso, a comparação de dois valores nesta representação não é simples como a comparação entre inteiros, cuja posição dos bits mais significativos são suficientes para indicar qual o maior e menor operando. Logo, para os dados em ponto flutuante, a comparação é feita através de uma subtração e, em seguida, o resultado desta operação é examinado e codificado. Para isto, um terceiro componente avalia o resultado da subtração da seguinte maneira:

- Se $(A - B)$ é positivo, então conclui-se que $A > B$.
- Se $(A - B)$ é negativo, então conclui-se que $A < B$.
- Se $(A - B)$ é nulo, então conclui-se que $A = B$.

Tal avaliação, utilizando-se do formato em ponto flutuante, examina o bit de sinal para descobrir se o resultado é positivo (s = 0) ou negativo (s = 1). Ao mesmo tempo, uma operação lógica 'ou' é realizada nos demais bits (mantissa e expoente) para descobrir se o resultado é nulo. A Tabela 6.6 resume esta codificação.

Tabela 6.6: Codificação do resultado da comparação.

Codificação	**Descrição**
00	$A = B$
01	$A > B$
10	$A = B$
11	$A < B$

A comparação entre inteiros utiliza uma unidade de comparação e o resultado é codificado em 2 bits, também de acordo com a Tabela 6.6. O resultado de ambas as codificações é multiplexado pela operação que está sendo realizada (entre inteiros ou em ponto flutuante) e armazenada em um registrador, que é lido pelo decodificador de instruções durante uma operação de desvio (Fig. 6.9). A carga no registrador de comparação é controlada pelo sinal 20 da micro-instrução.

Por exemplo, sejam os operandos A = 2.3434, B = 1.2043 e a operação de comparação **cmpfp A,B**. O sinal 11 da micro-instrução sinaliza a carga destes operandos, simultaneamente. A subtração produzirá o resultado C = 1.1391, cuja representação binária em precisão simples é:

$$\underbrace{0}_{\text{sinal}} \,|\, \underbrace{01111111}_{\text{expoente}} \,|\, \underbrace{00100011100111000000111}_{\text{mantissa}}$$

Logo, é possível concluir que $A > B$, visto que o sinal é positivo (s = 0) e que a operação lógica 'ou' entre os demais bits é diferente de zero, gerando a codificação de resultado 01 no registrador CMP, para uso posterior pelo decodificador de instruções. A Fig.6.10 apresenta uma simulação desta operação, na qual os operandos e o resultado estão representados em hexadecimal. Note que o resultado da subtração (0x3F91CE07) gerou corretamente a codificação de comparação 01 após três ciclos, representada pelo sinal *comparefp*.

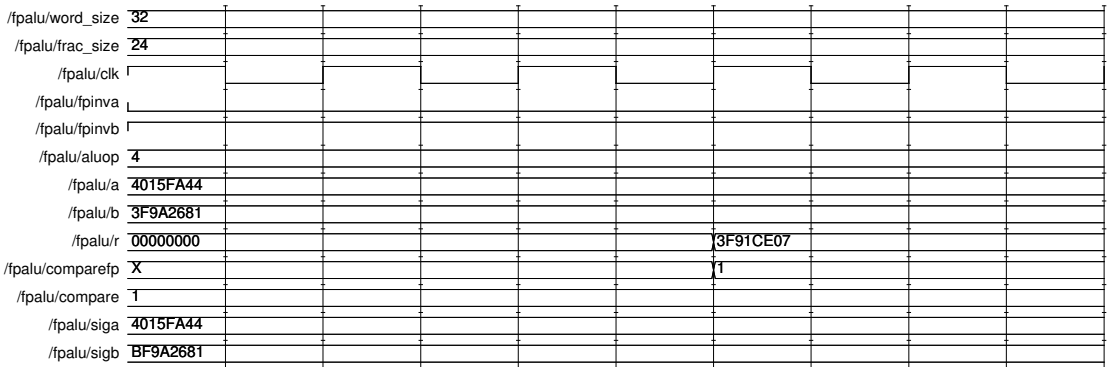


Figura 6.10: Simulação da operação de comparação.

As unidades de ponto flutuante da Xilinx^{TM} utilizam o padrão IEEE 754 e podem ser personalizadas para atender aos requisitos do hardware, permitindo a configuração da latência, taxa de ciclos, precisão, etc. Cada um destes parâmetros tem um impacto no custo de área e desempenho das unidades. No caso da unidade de multiplicação da ULA, utilizam-se blocos de processamento digital (DSP48E) incorporados à FPGA para reduzir a latência desta operação e também reduzir seu custo. Os resultados são apresentados na Seção 6.3.

Controle

O controle do elemento de processamento é efetuado através de um micro-programa, formado por micro-instruções de 23 bits. Cada micro-instrução corresponde a um conjunto de sinais que executam uma instrução do programa (add, sub, etc.). O contador do micro-programa seleciona a micro-instrução a ser executada e o desvio de fluxo no micro-programa é feito pelos dois bits mais significativos da micro-instrução, conforme especificado no Algoritmo 6.2.

A penúltima micro-ordem da micro-instrução reinicia o contador do micro-programa, dando início a um nova etapa de busca da próxima instrução, enquanto a última micro-ordem (MSB) sinaliza o desvio para o endereço de uma nova micro-instrução, recebido do *Decodificador de Instruções*. A cada ciclo, o contador do micro-programa é incrementado. A Tabela 6.7 apresenta todos os sinais que com-

```
   Dados: microprograma vd, contador i
1  process(clk)
2  início
3      se rising_edge(clk) então                    /* transição alta */
4          i <= i + 1 ;                            /* incrementa μCP */
5          se vd(i)(22)='1' então                  /* último bit = '1' */
6              i <= conv_integer(micropc) ;          /* desvia μCP */
7          se vd(i)(21)='1' então                  /* penúltimo bit = '1' */
8              i <= 0 ;                              /* reinicia μCP */
9  fim
10 control <= vd(i)(20 downto 0) ;                 /* Sinais de controle */
```

Algoritmo 6.2: Controle de desvio no micro-programa

põem a micro-instrução. O código em VHDL do micro-programa completo é contemplado no Apêndice C.1.

A decodificação de uma instrução é feita pelo *decodificador de Instruções*, no momento em que uma nova instrução é trazida e armazenada no registrador de instrução (RI), na Fig.6.11. Então, o decodificador examina o código de operação da instrução e gera o endereço da micro-instrução correspondente (μCP). Durante a etapa de decodificação, o MSB da micro-instrução de decodificação é ativado para desviar o fluxo de execução do microprograma para a nova micro-instrução.

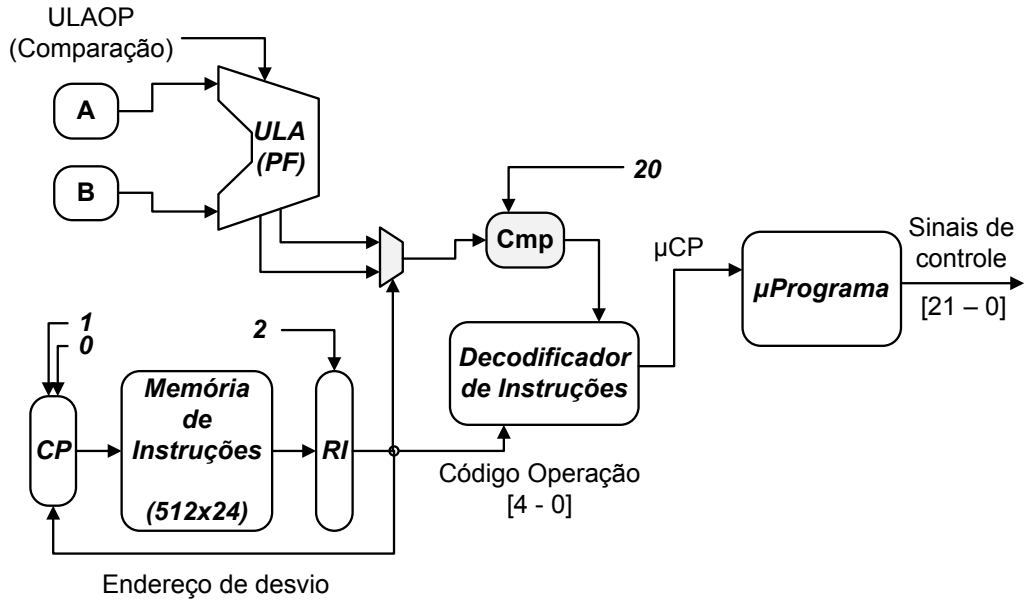


Figura 6.11: Componentes de controle.

Considere, por exemplo, a execução da instrução de soma **add**. Primeiro, o contador do microprograma deverá apontar para a micro-instrução de busca, que ativará a carga (μ-ordem 2) de uma nova instrução da memória no registrador de

Tabela 6.7: Sinais da Microinstrução.

Sinal	Descrição
0	Incrementa o contador do programa
1	Carrega um novo endereço no contador do programa
2	Carrega uma nova instrução no registrador de instruções
3	Inverte o operando B (em ponto flutuante)
4	Inverte o operando A (em ponto flutuante)
5,6,7	Codificam a operação a ser realizada pela ULA
8	Ativa a escrita ao banco de registradores
9	Carrega o resultado da ULA no registrador de endereços (MC)
10	Carrega o resultado da ULA no segundo registrador de endereços (MR)
11	Carrega novos operandos nos registradores A e B
12	Seleciona o operando que será carregado no registrador B (do banco de registradores ou um imediato)
13	Carrega o registrador de seleção de porta de entrada
14	Alterna entre o registrador de destino e um registrador de origem para acessar o banco de registradores
15	Carrega o resultado da ULA no registrador de dados da memória de resultados
16,17	Seleciona o dado a ser escrito no banco de registradores
18	Carrega o registrador de seleção de porta de saída
19	Sinaliza o término do programa em execução
20	Armazena o resultado de uma operação de comparação
21	Reinicia o contador do microprograma (nova etapa de busca da próxima instrução)
22	Desvia o fluxo de execução para uma nova micro-instrução

instruções.

$$vd(0) <= 00000000000000000000 \underbrace{1}_{\text{Carga RI}} 00$$

A cada ciclo o contador do microprograma é incrementado, fazendo-o apontar para a próxima micro-instrução, que decodificará (μ-ordem 22) a nova instrução e ao mesmo tempo incrementará (μ-ordem 0) o contador de programa.

$$vd(1) <= \underbrace{1}_{\text{Decodifica Op.}} 00000000000000000000 \underbrace{1}_{\text{Incrementa CP}}$$

O sinal de decodificação desviará o contador do microprograma para a primeira micro-instrução que executará a instrução **add**. Em um primeiro ciclo, os operandos são carregados (μ-ordem 11) nos registradores A e B.

$$vd(3) <= 00000000000 \underbrace{1}_{\text{Carga A e B}} 00000000000$$

No ciclo seguinte, o resultado é armazenado (μ-ordem 8) no banco de registradores e o contador do microprograma é reiniciado (μ-ordem 21). Ao total, a instrução **add** consome quatro ciclos para ser buscada, decodificada e executada.

$$vd(4) <= 0 \quad \underbrace{1}_{\text{Reinicia } \mu CP} \quad 000000000000 \quad \underbrace{1}_{\text{Carga no Banco}} \quad 00000000$$

As instruções de comparação têm o resultado carregado no registrador CMP (μ-ordem 20), para que o decodificador decida entre executar a micro-instrução de desvio correspondente ou continuar o fluxo de execução do microprogama, que eventualmente será reiniciado e buscará a próxima instrução da memória de instruções. Caso o desvio seja tomado, um novo endereço será carregado no contador de programa (μ-ordem 1) pela micro-instrução de desvio. Por isso, antes de toda instrução de desvio, é preciso existir uma instrução de comparação, para que a tomada de decisão possa ser realizada.

Algoritmo da Memória de Instruções

O algoritmo presente na Memória de Instruções é responsável, principalmente, pelos cálculos de interseção. Suas demais funções realizam tarefas de controle, E/S de dados e gerenciamento da lista de atravessamento.

A primeira parte do algoritmo aguarda por um dado proveniente da porta de E/S. Este dado sinaliza que a carga do raio e da respectiva lista de atravessamento foi executada e que, portanto, o algoritmo pode dar continuidade ao restante da computação. Em seguida, determina-se a posição do elemento de processamento na lista. A partir dela, o EP identifica qual o EP anterior e posterior a ele, armazenando seus identificadores nos registradores R_{42} e R_{43}, respectivamente. Tais registradores permitem o chaveamento dos sinais de interrupção, enquanto o controlador de interrupção fica incumbido de processá-las, sem a interferência do programa.

Em seguida, lê-se a quantidade de triângulos existentes na Memória da Cena e, sucessivamente, as coordenadas de um triângulo, caso exista. Senão, o algoritmo termina sem calcular interseções, desviando para a instrução de término (*hlt*).

A seção do algoritmo responsável pelos cálculos de interseção é então iniciada, utilizando os registradores especiais da Tabela 6.4 e os dados lidos da Memória da Cena. Em três momentos o algoritmo verifica os resultados parciais da computação a fim de determinar se deve ou não continuar, i.e. se os dados computados até então ainda são válidos. Em caso afirmativo, o algoritmo carrega mais dados (coordenadas do triângulo) da Memória da Cena e realiza uma novo teste de interseção, caso ainda existem dados. Senão, o algoritmo termina.

O resultado da computação é armazenado no registrador R_{24}, que é enviado ao Controlador de Interrupções para que este possa interromper os próximos elementos

processadores da lista de atravessamento.

6.2.4 Paralelismo e o Controlador de Interrupção

Todos os elementos de processamento que são atravessados por um mesmo raio realizam os cálculos de interseção em paralelo. No entanto, é preciso determinar a menor das interseções entre as obtidas pelos processadores, a fim de identificar o objeto mais próximo do observador. Caso contrário, um objeto localizado em um plano posterior seria identificado, gerando um resultado inconsistente.

Este problema, discutido na Seção 4.2, pode ser solucionado por meio da sequência de atravessamento de um raio. Esta ordem, intrínseca dos volumes uniformes, garante que os cálculos de interseção sejam efetuados na ordem em que se afastam da origem do raio. Assim, a primeira interseção encontrada é a menor de todas, tornando desnecessária a busca por interseções nos demais processadores atravessados pelo raio.

Para realizar essa operação, cada elemento de processamento é conectado ao seu vizinho direto por meio de duas linhas de interrupção, que são selecionadas (chaveadas) de acordo com a lista de atravessamento que cada um recebe. A Fig.6.12 apresenta este chaveamento para uma configuração de 4 processadores. Os registradores R_{42} e R_{43} selecionam o processador anterior e posterior, de acordo com a ordem na lista.

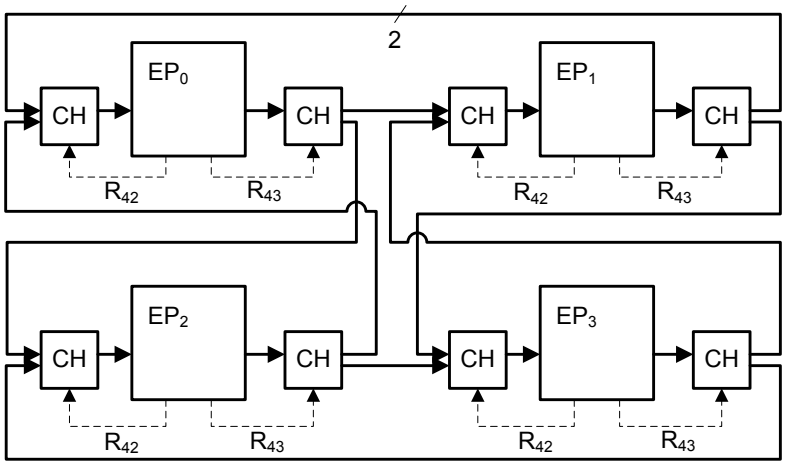


Figura 6.12: Linhas de Interrupção (2 bits) para 4 processadores.

O *Controlador de Interrupção* (CI) dos processadores do GridRT se encarrega de tratar os sinais recebidos, processá-los e retransmití-los, conforme a ordem do seu respectivo processador na lista. Tal controlador é modelado como uma máquina de estado. Desta forma, as interrupções são tratadas e despachadas mais rapidamente, ao contrário do que ocorreria se fossem processadas pelo elemento processador atra-

vés de rotinas de interrupção. A Fig.6.13 mostra os componentes envolvidos no processo de interrupção.

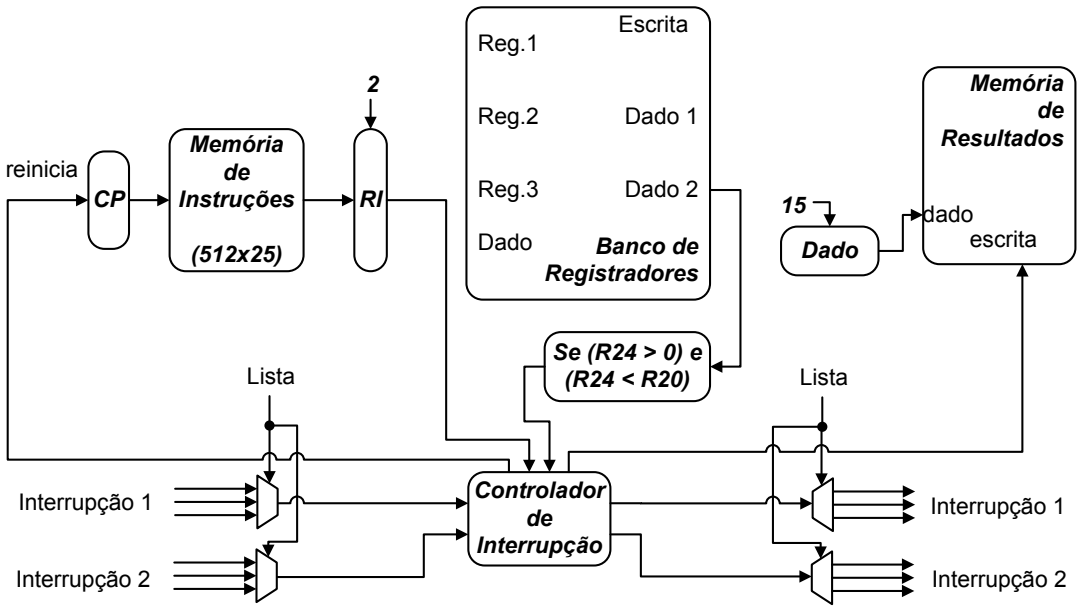


Figura 6.13: Controlador de Interrupção e via de dados do elemento processador.

As interrupções do tipo 1 correspondem à suspensão da computação, enquanto aquelas de tipo 2 correspondem ao término da computação. As seguintes possibilidades são previstas pelo controlador de interrupção:

1. Quando uma interrupção de suspensão (*interrupt1_in*) é recebida pelo elemento processador, significa que o anterior encontrou uma interseção e está suspendendo a execução dos EPs conseguintes na lista, como mostra a forma de ondas na Fig.6.14. Logo, ao receber esta interrupção, o controlador reinicia o contador de programa (*rst_pc*) e re-transmite a interrupção para o próximo processador na lista (*interrupt1_out*). Dessa maneira, não há mais nada a se fazer a não ser aguardar o próximo raio a ser atravessado e processado.

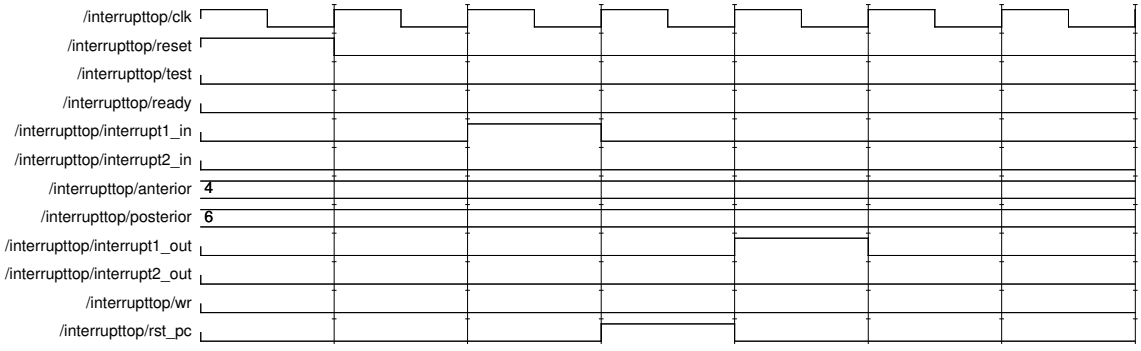


Figura 6.14: Simulação da interrupção de suspensão.

2. Uma outra situação ocorre quando um elemento processador encontra uma interseção (*test*), que a armazena no registrador R_{24}. Neste caso, o controlador de interrupções envia uma interrupção ao próximo da lista (*interrupt1_out*). No entanto, observe que o elemento processador em questão não pode assumir que já detém a menor interseção até que ele receba uma interrupção de término (*ready*), informando que os anteriores terminaram sem sucesso, caso esses existam. Assim que receber esta interrupção, o controlador confirma a escrita do resultado na *Memória de Resultados*, ativando o sinal *wr* e, ao final do programa, uma instrução sinaliza o término do processamento. A Fig.6.15 demonstra a forma de onda de interrupção para esta situação. Note que o sinal de escrita só é enviado para a memória de resultados com a chegada do sinal de interrupção de término (*interrupt2_in*).

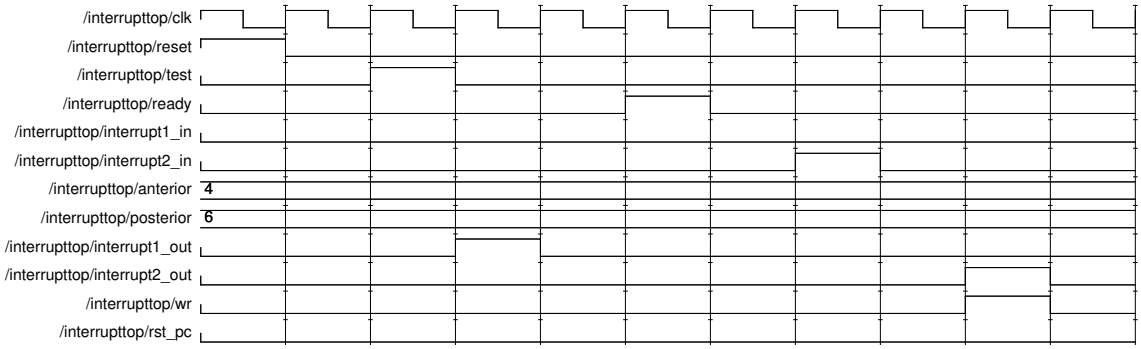


Figura 6.15: Simulação da interrupção de término.

3. Se o processador concluir sua computação sem encontrar interseções, a última instrução sinaliza o fim do processamento (*ready*) e o controlador aguarda o recebimento da interrupção de término do anterior (*interrupt2_in*), conforme mostra a forma de onda na Fig.6.16. Após a chegada da interrupção aguardada, a mesma é re-transmitida adiante (*interrupt2_out*).

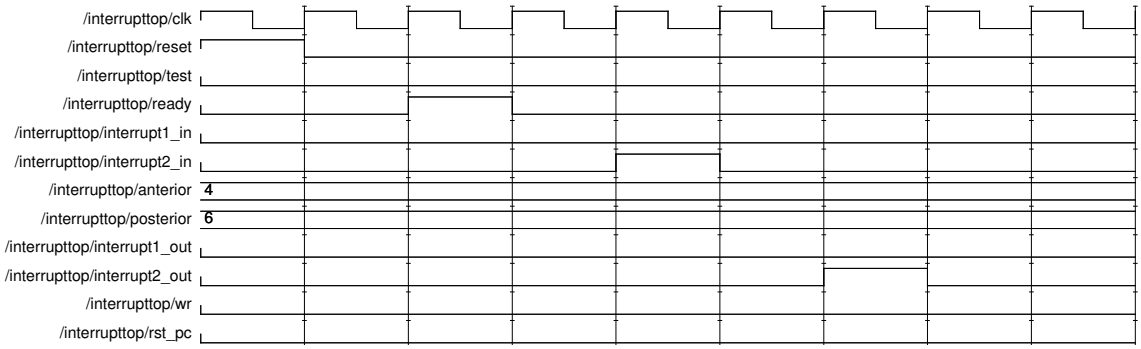


Figura 6.16: Simulação do término da computação sem interseções.

6.2.5 Escalabilidade

Através da repartição da cena em volumes uniformes (malha uniforme) busca-se uma relação ótima entre o número de *voxels* e o tamanho da cena. Suponha, por exemplo, que uma cena tenha sido repartida em duas regiões iguais. Esta repartição reduz a quantidade de passos necessários ao algoritmo de atravessamento do raio, visto que no máximo duas regiões poderão ser atravessadas por um raio. No entanto, como a cena foi repartida entre apenas duas regiões, a quantidade de objetos em cada uma é alta e, consequentemente, o número de testes de interseção também. Por outro lado, suponha que a mesma cena tenha sido repartida em quatro regiões. Logo, a quantidade máxima de passos do algoritmo de atravessamento aumenta, mas o número de objetos em cada região diminui, assim como o número de testes de interseção. Portanto, encontrar a relação ótima não é uma tarefa trivial, principalmente quando se trata de uma arquitetura modelada em hardware, cujo tamanho do volume uniforme tem impacto no custo de área final. Porém, é consenso que uma quantidade reduzida de repartições implica em mais objetos por repartição, o que também implica em mais cálculos de interseção. Tal processamento de interseções é mais oneroso que o algoritmo de atravessamento do volume e, sempre que possível, deve ser reduzido [2, 4].

Na arquitetura GridRT a quantidade de repartições da cena equivale ao número de elementos processadores necessários que, por sua vez, têm impacto no custo de área final. Uma solução ótima busca determinar a melhor relação de objetos para cada *voxel*. Uma delas [4] sugere que o número de repartições em cada direção seja definida segundo a Equação 6.1, onde n é o número de objetos da cena. Os parâmetros w_x, w_y e w_z representam o comprimento do volume uniforme em cada direção e o parâmetro m é somente um fator de multiplicação para variação da quantidade de repartições. O volume uniforme produzido tem o tamanho definido por $n_x \times n_y \times n_z$.

$$\begin{aligned} s &= \sqrt[3]{(w_x w_y w_z)/n} \\ n_x &= \lfloor m w_x / s \rfloor + 1, \\ n_y &= \lfloor m w_y / s \rfloor + 1, \\ n_z &= \lfloor m w_z / s \rfloor + 1, \end{aligned} \tag{6.1}$$

Por exemplo, a cena *Stanford Bunny* [31] contem 69451 triângulos e o comprimento do volume uniforme em cada direção (w_x, w_y, w_z) corresponde a $(77.849503, 77.166809, 60.336647)$. Estes comprimentos são dados pela subtração da coordenada máxima e mínima do volume uniforme total. Deste modo, a repartição descrita na Equação 6.1 aplicada à esta cena produziria um volume uniforme de $45 \times 45 \times 35$ voxels, para um fator de multiplicação $m = 1$. Tal repartição equi-

valeria a um conjunto de 70875 elementos processadores, o que é inviável diante das limitações de recursos das FPGAs mais modernas.

Comprimento da Lista e dos Identificadores

O número de elementos processadores do GridRT interfere no comprimento do identificador de cada processador, no tamanho da lista de atravessamento e na quantidade de elementos chaveadores de interrupção que dela dependem.

Por exemplo, para uma configuração de até 4,8 e 16 elementos processadores, os identificadores devem ter comprimento de 2,3 e 4 bits, respectivamente. Logo, para N processadores, o comprimento do identificador corresponde a $\lceil \log_2 N \rceil$. Porém, um bit adicional é necessário para distinguir os valores positivos e negativos. Portanto, o comprimento final corresponde a $\lceil \log_2 N \rceil + 1$.

Quanto ao tamanho da lista, o mesmo depende da quantidade de repartições (processadores) em cada direção, mas, no pior caso, o tamanho da lista nunca excederá a soma das repartições em cada direção. De acordo com o exemplo na Fig.6.17, um raio partindo do ponto A em direção ao ponto B percorreria a diagonal do volume uniforme.

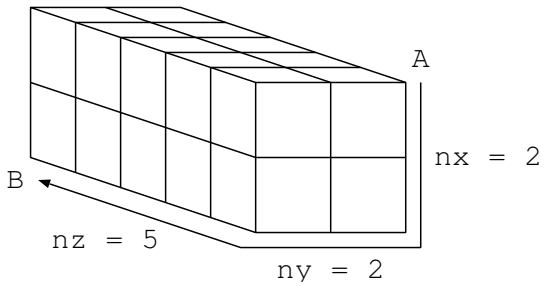


Figura 6.17: Comprimento máximo da lista de atravessamento.

No entanto, o caminho pode ser interpretado de uma forma diferente, ou seja, o raio precisou percorrer dois EPs no eixo x, mais dois EPs no eixo y e finalmente cinco EPs no eixo z, totalizando nove posições. Deste modo, para um repartição de $2 \times 2 \times 5$ processadores o tamanho máximo da lista é de $2 + 2 + 5$ identificadores. Logo, o tamanho da lista é descrito pela Equação 6.2.

$$T = n_x + n_y + n_z \tag{6.2}$$

Uma lista de oito posições (T = 8) já foi implementada neste trabalho para uma possível expansão da arquitetura, a curto prazo. Os identificadores de 4 bits totalizam uma lista de atravessamento com comprimento de 4×8 bits. Caso mais processadores possam ser incluídos, será preciso rever o tamanho da lista e de cada identificador. Por exemplo, uma arquitetura de 16 processadores, cujo volume fosse

repartido em $4 \times 4 \times 1$, precisaria de identificadores de $\lceil \log_2 16 \rceil + 1$ bits e de uma lista de 4+4+1 posições, totalizando uma lista de atravessamento com comprimento de 5×9 bits.

Em vista disso, como a lista é armazenada em registradores do processador, o seu tamanho tem impacto direto na quantidade necessária de registradores. No caso exemplificado de 16 processadores, apenas um 1 registrador adicional seria necessário para conter a lista de nove elementos.

Caminho das Interrupções

A seleção do caminho percorrido pelas interrupções é determinado pelos identificadores dos processadores presentes na lista. Esta é gerada a partir do atravessamento de cada raio contra o volume uniforme, a fim de determinar os processadores que devem participar da computação de interseção do raio.

Cada processador recebe e envia interrupções de e para um elemento chaveador, cuja seleção do caminho ocorre usando os identificadores dos elementos processadores anterior e posterior em relação ao correspondente na lista, conforme mostra a Fig.6.18.

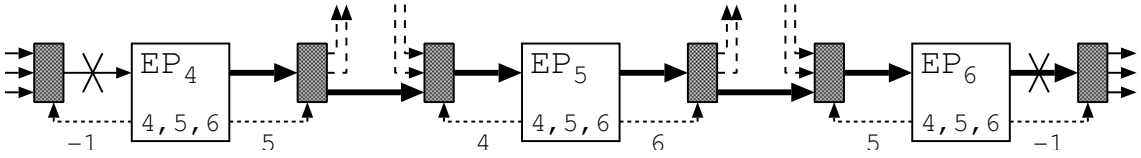


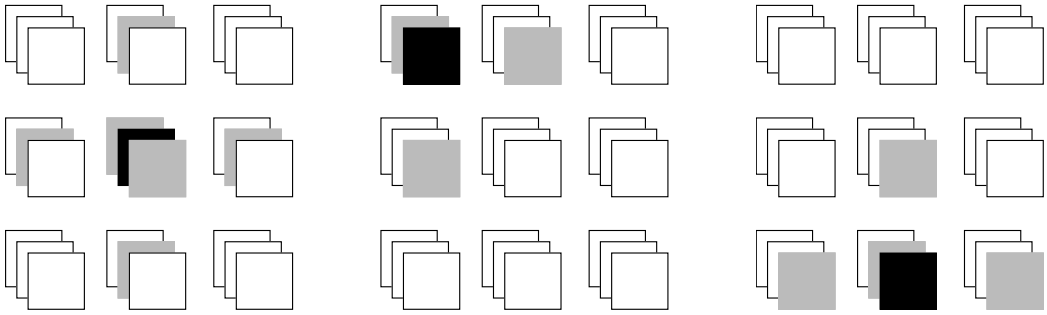
Figura 6.18: Caminho das interrupções.

Observe que apenas os elementos processadores da lista $L = (4, 5, 6)$ participam da computação e selecionam o caminho das interrupções. Ainda, aqueles localizados nas extremidades da lista utilizam um valor negativo (-1) para desativar qualquer sinal proveniente do EP anterior ou enviar ao EP posterior. Isto é feito para garantir que nenhum sinal interfira na computação dos elementos processadores ativos, pois ainda que alguns dos EPs estejam desativados, seus controladores de interrupção não estão e, portanto, podem processar qualquer sinal que chegue até eles.

É importante observar que um volume uniforme é sempre construído obedecendo à regularidade, isto é, a repartição que ocorre em cada um dos eixos é estendida aos demais, de maneira a formar um cubo ou um paralelepípedo retângulo. Por esse motivo, nunca existirão volumes disformes, constituídos de 3,5,7 ou 11 processadores, por exemplo.

Logo, dependendo da disposição dos processadores no volume uniforme, o número de vizinhos diretos que podem se comunicar com um elemento processador em

particular pode chegar a no máximo seis, que corresponde ao caso em que o processador se encontra dentro do volume uniforme, conforme destacado na Fig.6.19a. Deste modo, existem seis elementos chaveadores encarregados de receber as interrupções dos vizinhos e outros seis elementos chaveadores encarregados de enviar outras interrupções aos mesmos, totalizando uma dúzia.



(a) Meio (b) Canto (c) Borda

Figura 6.19: Tipos de disposição dos processadores.

Na posição do processador exibido na Fig.6.19b, existem três vizinhos diretos, totalizando seis elementos chaveadores. Por fim, o processador da Fig.6.19c tem quatro vizinhos, o que totaliza oito elementos chaveadores.

A conexão entre as linhas dos elementos chaveadores aos seus respectivos elementos processadores ainda é um processo manual, realizado durante a etapa de construção do volume uniforme, ou seja, durante a codificação da arquitetura GridRT em VHDL. Uma vez conectados, o elemento chaveador apenas realiza a seleção da linha de interrupção indicada na lista para o elemento processador correspondente.

6.3 Resultados

A arquitetura GridRT foi inteiramente descrita através da linguagem VHDL, simulada por meio do ModelSim XE 6.3c e sintetizada pelo XST, cujo dispositivo alvo foi uma FPGA Virtex-5 modelo XC5VFX70T. Os principais recursos desta plataforma de desenvolvimento encontram-se descritos na Seção 6.1.3.

Devido às limitações de recursos, somente a configuração de 4 processadores foi sintetizada e programada na FPGA. Contudo, uma outra configuração de 8 processadores também foi sintetizada para estimativa do custo de área e escalabilidade da arquitetura, conforme será explicado na Seção 6.3.1. Além diso, a cena processada é uma versão em baixa resolução do *Stanford Bunny*, um coelho do repositório de modelos tridimensionais do Laboratório de Computação Gráfica de Stanford [31]. De modo geral, tal cena é empregada para testes em diferentes áreas da computa-

ção gráfica e disponibilizada em diferentes resoluções, conforme os dados listados na Tabela 6.8. Logo, como em cada processador cabem aproximadamente 910 triângulos, foi utilizada a cena de 948 triângulos, repartida entre os 4 processadores da arquitetura.

Tabela 6.8: Resoluções disponíveis da cena *Stanford Bunny*.

Contagem de triângulos	69451	16301	3851	948
Tamanho	19Mb	4.5Mb	1Mb	266Kb

O desempenho da arquitetura foi alvo de avaliação, embora a pequena quantidade de processadores aliada a uma baixa frequência de operação do sistema (50MHz) tenham contribuído para o elevado tempo de execução do algoritmo, como será explicado na Seção 6.3.2.

6.3.1 Consumo de área

Avaliar o consumo de área de um projeto em FPGA significa contabilizar os recursos por ele consumidos, tais como LUTs, latches, flip-flops, etc. A partir desta avaliação é possível prever quais são as partes do projeto que carecem de otimização (redução de área), assim como o grau de escalabilidade da arquitetura proposta. Quando o projeto é submetido ao sintetizador (XST), dá-se início à etapa de análise de recursos, uma vez que é nesta etapa que a descrição do hardware em uma HDL é convertida em termos de funções lógicas mapeáveis em CLBs. Portanto, já é possível se ter uma ideia do consumo do sistema sem que ele seja transcrito para a FPGA.

Inicialmente, a arquitetura GridRT não previa o uso do microprocessador MicroBlaze. As funções de transpassamento do volume uniforme e de geração de raios primários eram feitas por dois componentes adicionais, além de uma unidade de sincronização, como é ilustrado na Fig. 6.20. Tais componentes também são processadores MIPS que operam com um conjunto reduzido de instruções, com a exceção da unidade de controle, que é uma simples máquina de estados. Os processadores são semelhantes ao elemento de processamento descrito na Seção 6.2.3. A principal diferença é a ausência da *Memória da Cena* e *Memória de Resultados*. As estatísticas da síntese para as configurações de 1 e 8 processadores são apresentadas na Tabela 6.9, assim como a estatística para a configuração completa.

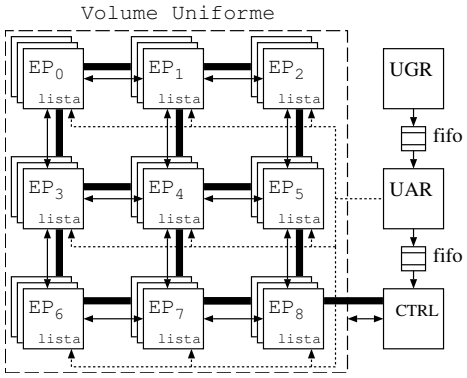


Figura 6.20: Arquitetura GridRT, sem o MicroBlaze.

Tabela 6.9: Síntese de 1EP, 8EPs e do sistema completo.

	Arquitetura GridRT		
	1EP	**8EPs**	**Completo**
Registradores	2781	22088	35725
LUTs	3975	31468	44343
Pares completos LUT-FF	1845	14580	18436
BRAM/FIFO	17	72	74
DSP48Es	4	32	43

Diante destes resultados, observa-se que o consumo de recursos dos componentes adicionais é elevado. A diferença de tamanho do conjunto de 8 processadores para o sistema completo é 20% a 40% maior, sendo consumidas, por exemplo, mais 12875 tabelas *look-up*. Portanto, o uso do MicroBlaze como substituto destes componentes justifica-se através do seu reduzido consumo de recursos, otimizado para dispositivos da Xilinx^{TM}. Por exemplo, a quantidade de LUTs utilizadas pelo microprocessador é aproximadamente 2305, como mostra a Tabela 6.10 para uma implementação da arquitetura com 4 processadores. Nesta tabela, também observa-se a previsão do gasto de recursos das unidades de ponto flutuante e outros componentes de E/S de dados, como o $rs232\_uart$. Estes e alguns outros componentes podem ser visualizados no diagrama de blocos da Fig. 6.22, que também apresenta as conexões e interfaces dos mesmos.

Além dos dados da Tabela 6.10, a síntese também fornece uma estimativa de frequência máxima de operação do sistema. Neste caso, tal estimativa foi de 50.246 MHz com um período de 19.902 ns, o que correspondeu ao funcionamento correto da implementação a 50 MHz.

Após a fase de síntese, as funções lógicas, mapeadas em blocos lógicos (CLBs), podem ser transcritas para a FPGA, embora ainda existam outros passos de otimi-

Tabela 6.10: Relatório da síntese para 4 processadores, 1 MicroBlaze e periféricos.

Componente	**Flip-Flops**	**LUTs**	**BRAMs**
system	14452	19471	52
microblaze_0_wrapper	2143	2305	
microblaze_0_to_grid_fsl_0_wrapper	7	44	
grid_fsl_0_wrapper	6927	6956	36
grid_fsl_0_wrapper_floating_point_5	140	430	
grid_fsl_0_wrapper_floating_point_4	284	494	
grid_fsl_0_wrapper_floating_point_3	385	780	
grid_fsl_0_wrapper_floating_point_2	183	164	
grid_fsl_0_wrapper_floating_point_1	140	430	
grid_fsl_0_to_microblaze_0_wrapper	7	44	
xps_timer_0_wrapper	357	285	
proc_sys_reset_0_wrapper	67	51	
mdm_0_wrapper	119	112	
clock_generator_0_wrapper			
rs232_uart_1_wrapper	146	126	
lmb_bram_wrapper			16
ilmb_cntlr_wrapper	2	6	
dlmb_cntlr_wrapper	2	6	
dlmb_wrapper	1	1	
ilmb_wrapper	1	1	
mb_plb_wrapper	145	342	

zação a serem executados entre estas etapas. A Tabela 6.11 apresenta um resumo sobre os recursos consumidos na FPGA XC5VFX70T através da implantação do sistema completo de 4 processadores.

Na Tabela 6.11, percebe-se que a taxa de utilização de *Slices* é alta (68%), o que constitui um fator determinante para impedir que a arquitetura possa ser expandida, pois são neles que residem as LUTs e os Flip-Flops, sem os quais o sistema não pode crescer. Os recursos consumidos pelas implementações de 1 e 4 processadores são ilustradas na Fig. 6.21a e Fig. 6.21b, respectivamente. A área da configuração de 4 processadores é até 3.24 vezes maior em comparação àquela de 1 processador, ambas incluindo um MicroBlaze, barramentos e periféricos. Os valores dos gráficos da Fig. 6.21 são descritos na Tabela 6.12.

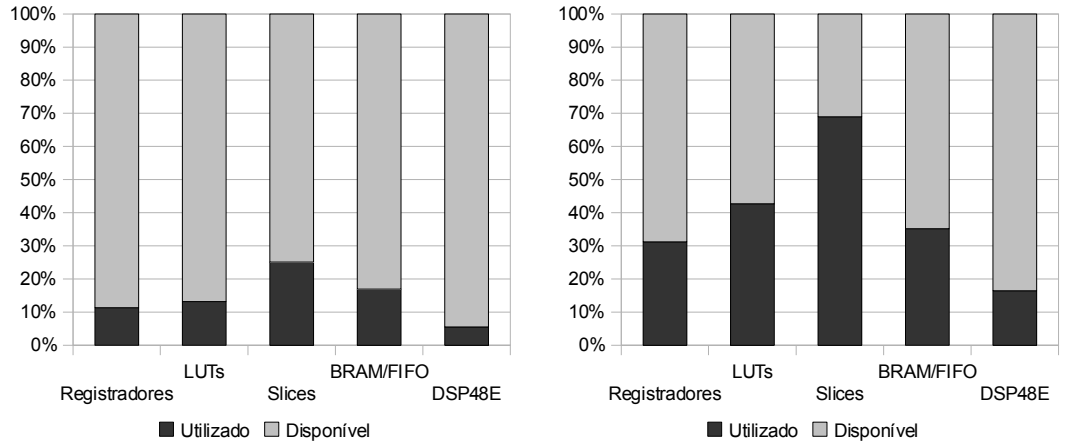
Tabela 6.11: Consumo de área para 4 processadores, 1 MicroBlaze e periféricos.

Recursos	**Utilizado**	**Disponível**	**Utilização**
Número de registradores dos Slices	13986	44800	31%
Usados como Flip-Flops	13984		
Usados como Latch-thrus	13984		
Número de LUTs dos Slices	19128	44800	42%
Usados como lógica	18246	44800	40%
Usados como Saída O6	16587		
Usados como Saida O5	264		
Usados como Saida 05 e 06	1395		
Usados como Memória	651	13120	4%
Usados como Dual-Port	64		
Usados como Saída 05 e 06	64		
Usados como Shift-Register	587		
Usados como Saída 06	586		
Usados como Saída 05	1		
Exclusivos route-thru	231		
Número de Route-thrus	1263	89600	1%
Número de Slices ocupados	7720	11200	68%
Número de SLICEMs ocupados	338	3280	10%
Número de Pares LUT-FF	25040		
Com um FF inutilizado	11054	25040	44%
Com uma LUT inutilizada	5912	25040	23%
Com pares LUT-FF completos	8074	25040	32%
Número de IOBs delimitados	4	640	1%
Número de BlockRAM/FIFO	52	148	35%
Total de memória usada (KB)	1872	5328	35%
Número de DSP48Es	21	128	16%

Tabela 6.12: Utilização dos recursos da configuração de 1 e 4 processadores, com MicroBlaze e periféricos.

	1 EP	**4 EPs**	**Recursos da FPGA**	**Aumento**
Registradores	5053	13986	44800	2.76×
LUTs	5898	19128	44800	3.24×
Slices	2811	7720	11200	2.74×
BRAM/FIFO	25	58	148	2.08×
DSP48E	7	21	128	3×

Comparada à arquitetura de Schmittler et al. [20] e de Woop et al. [19], de apenas um processador, o GridRT de 4 processadores utiliza menos que a metade das unidades de ponto flutuante. No entanto, o consumo de memória total é pelo menos oito vezes maior, como mostra a Tabela 6.13. Uma vez que o GridRT assemelha-



(a) Configuração com 1 processador, MicroBlaze e periféricos

(b) Configuração com 4 processadores, MicroBlaze e periféricos

Figura 6.21: Utilização de recursos para 1 e 4 processadores, com uso do MicroBlaze e periféricos.

se à estrutura de volumes uniformes, sua escalabilidade foi, desde o início, foco de preocupação, pois a repartição da cena pode atingir a casa das dezenas ou até centenas de processadores em cada uma das três direções da repartição. Portanto, o elemento de processamento foi projetado visando a economia e simplicidade, a fim de não prejudicar a escalabilidade.

Tabela 6.13: Unidades de ponto flutuante.

Operação	**SaarCOR [20]**	**RPU [19]**	**GridRT 4P**
ADD	16	20	8
MUL	11	20	4
DIV	5	4	4
CMP	13	4	0
SQRT	0	0	4
Total	**45**	**48**	**20**
Total MEM.	**75KB**	**251.9KB**	**2MB**

O alto consumo de memória é, no entanto, devido ao armazenamento da cena e dos resultados nos blocos de memória dos processadores da arquitetura GridRT, ainda que tais dados estejam repartidos entre os processadores. Isso também prejudica a escalabilidade da arquitetura e restringe o tamanho das cenas que podem ser processadas por ela. No futuro, a cena pode ser armazenada em uma memória secundária e blocos de dados carregados nos respectivos elementos processadores à medida que forem por eles requisitados. Desta forma, cenas maiores poderão ser

processadas, sem prejudicar a escalabilidade.

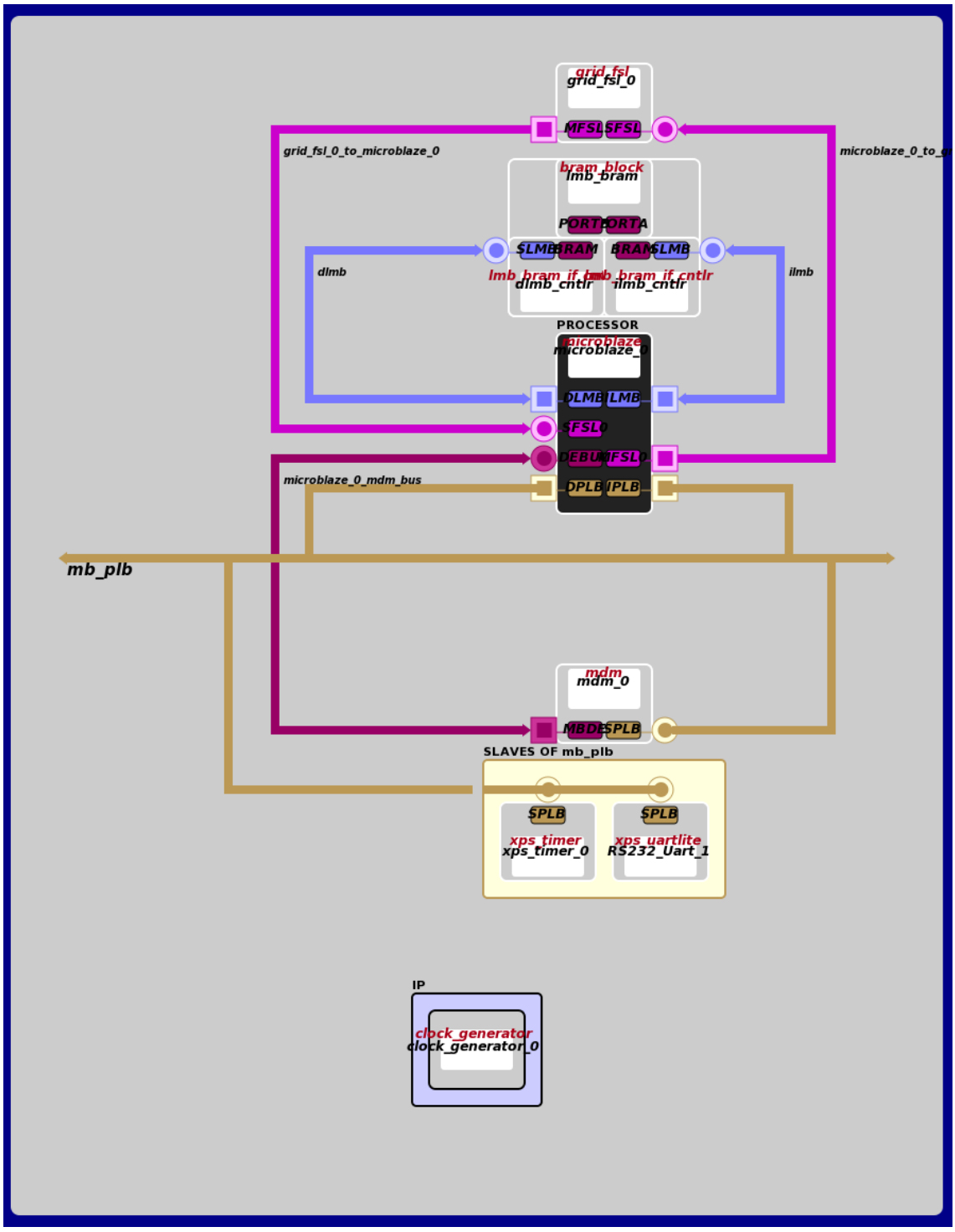


Figura 6.22: Diagrama de blocos do sistema.

6.3.2 Desempenho

O resultado de desempenho da arquitetura GridRT ainda não é adequado para o processamento de cenas tridimensionais em tempo real, principalmente devido

ao uso de somente quatro processadores e ao baixo poder computacional de cada um deles, que foi simplificado para privilegiar a escalabilidade da arquitetura. Além disso, a frequência máxima atingida foi de apenas 50MHz, embora a FPGA Virtex-5 tenha capacidade para operar a até 500MHz.

As operações de soma, subtração e multiplicação, em ponto flutuante, tiveram suas latências definidas para 3 ciclos, visto que uma baixa latência minimiza o consumo de recursos, mas reduz a frequência de operação da unidade [56]. Essa decisão pode ter contribuído para a redução da frequência do sistema. Da mesma forma, as operações de divisão e raiz quadrada tiveram suas latências definidas para 9 ciclos, pois uma latência ainda menor reduziu fortemente a frequência de operação destas unidades. A Tabela 6.14 apresenta a quantidade de ciclos necessários para a execução de cada instrução. Contudo, nota-se que as instruções de ponto flutuante, assim como as demais, apresentam ciclos adicionas referentes a outros sinais de controle. Maiores detalhes são disponíveis consultando o microprograma no Apêndice C.1.

Tabela 6.14: Ciclos de execução das instruções.

Ciclos	**Instruções**
3	add, sub, and, or, cmp, jl, je, jg, j
4	addi, store, input, output
6	addfp, subfp, mulfp, cmpfp
7	load
12	divfp, sqrtfp

O cálculo de uma interseção gasta 648 ciclos, incluindo o tempo de carga dos dados da memória para o banco de registradores. Logo, a uma frequência de 50 MHz, cujo período é de 20 ns, são realizadas aproximadamente 77 mil interseções por segundo, em cada processador. Desse modo, os quatro processadores são capazes de realizar em paralelo 308 mil interseções por segundo. Além disso, 528 ciclos correspondem às operações em ponto flutuante, dentre os 648 ciclos do cálculo de interseção. Logo, são realizadas de 5.4 a 21.6 milhões de operações em ponto flutuante por segundo (MFLOPs). A Tabela 6.15 mostra uma comparação da arquitetura GridRT com as de Schmittler et al. [20], Woop et al. [19] e NVidia 5900 FX [13].

Tabela 6.15: Comparação de desempenho.

Arquitetura	**GridRT**	**SaarCOR [20]**	**RPU [19]**	**Nvidia 5900 FX [13]**
Nº de EPs	4	1	1	varios
FLOPs	5.4∼16.2 M	4 G	2.9 G	200 G
Freq. operação (MHz)	50	90	66	500
Nº de FPUs	20	45	48	400
Operação	Multi-ciclo	*Pipeline*	*Pipeline*	*Pipeline*

Observa-se que a quantidade de unidades de ponto flutuante empregada em cada arquitetura contribui para um desempenho melhor, principalmente para a arquitetura Nvidia. Além disso, um consumo de 648 ciclos por cálculo de interseção é elevado se comparado ao consumo de 44 ciclos de uma implementação com instruções SSE (*Streaming SIMD Extensions*) da Intel™ [21]. A esta velocidade de 44 ciclos, a arquitetura GridRT produziria 2.5 milhões de cálculos de interseção por segundo, para cada elemento processador.

Um outro fator decisivo é a operação em *pipeline* do elemento processador, que aumenta significativamente o desempenho do sistema, como pode ser visto nas outras arquiteturas. Já no GridRT, o controle dos processadores ainda é multi-ciclo, embora as unidades de ponto flutuante da Xilinx™ já possam operar em *pipeline*, isto é, a cada ciclo novos operandos podem ser introduzidos, cujos resultados são obtidos de acordo com a latência da operação.

Capítulo 7

Conclusão e Trabalhos futuros

A arquitetura GridRT e seu modelo de paralelismo foram apresentados. Ela é semelhante à estrutura de aceleração de volumes uniformes, ou malhas uniformes, com suporte a cálculos paralelos de interseção raio-triângulo. Para isso, são usados sinais de interrupção entre os elementos de processamento, para sincronização das suas operações.

Duas implementações desta arquitetura foram desenvolvidas em software para demonstração do modelo de paralelismo. A primeira, usando OpenMP, utiliza *threads* para realizar os cálculos de interseção em paralelo e simular a sinalização das interrupções via compartilhamento de memória. Já a segunda, usando OpenMPI, emprega a troca de mensagens entre *processos* para sinalização de interrupções e processamento paralelo das interseções. Nas duas implementações esperava-se obter aceleração em relação a uma versão sequencial, devido ao benefício do adiantamento dos cálculos de interseção, graças ao paralelismo. No entanto, somente a implementação em OpenMP obteve aceleração, ao contrário da implementação em OpenMPI. O motivo desta queda de desempenho está no uso de rotinas de troca de mensagem bloqueante. Estes aguardam a confirmação das mensagens enviadas ou o recebimento de uma mensagem para só então continuar a execução do algoritmo.

Uma terceira implementação da arquitetura, dessa vez em hardware, descrita inteiramente em VHDL, teve como alvo uma FGPA Virtex-5, modelo *XC5VFX70T*. Os elementos processadores implementados seguem o modelo MIPS e são controlados por um pequeno conjunto de instruções, com suporte a operações em ponto flutuante, por meio de unidades aritméticas da Xilinx. Um controlador de interrupções, presente em cada elemento processador, realiza as tarefas de sinalização e tratamento das interrupções, à parte do controle principal do processador. Desse modo, a arquitetura foi implementada com quatro elementos processadores, interligados por linhas de interrupção, que são chaveadas através de uma lista de atravessamento do raio em consideração. Tais processadores formam o co-processador GridRT, especializado em cálculos de interseção em paralelo. Este, se comunica com

o microprocessador MicroBlaze, que realiza as funções de atravessamento do volume uniforme e a geração dos raios primários, além de permitir a interface com os outros periféricos da arquitetura, conforme foi apresentado no Capítulo 6.

Apesar da simplicidade dos elementos de processamento em hardware, o consumo de área limitou a escalabilidade da arquitetura a apenas quatro processadores. Além disso, o foco dado à economia de recursos, contribuiu para o baixo desempenho desta configuração, conforme apresentado na Seção 6.3.1. Os resultados sugerem o uso de mais unidades de ponto flutuante por processador, assim como a inclusão de uma estrutura que permita o processamento dos dados em *pipeline*. Contudo, a inclusão de mais elementos processadores também se faz necessária, visto que a quantidade de repartições do volume uniforme é um fator determinante para que se possa atingir um desempenho de processamento adequado à síntese de imagens em tempo real, como foi visto na Seção 6.2.5.

O paralelismo da arquitetura GridRT é promissor, diante do resultado obtido pela implementação em OpenMP. Apesar do baixo desempenho da arquitetura implementada em hardware, o projeto pode ser aprimorado e otimizado. Por exemplo, o uso de memória secundária pode reduzir o tamanho das memórias de cada elemento processador e, consequentemente, reduzir o consumo de recursos. Além disso, as unidades em ponto flutuante podem ser adequadas para uma precisão menor, mas ainda suficiente para a correta operação do traçado de raios. Tudo isto reduziria o consumo de área e possibilitaria a inclusão de mais elementos de processamento, o que contribuiria para reduzir a quantidade de cálculos de interseção. No futuro, é preciso explorar ainda mais o paralelismo da arquitetura, visto que alguns elementos processadores ficam ociosos durante o processamento de um dado raio. A princípio, é possível que mais de um raio seja processado em paralelo, desde que os identificadores dos elementos processadores listados sejam diferentes. Por exemplo, sejam $L_1 = (0, 1, 3)$ e $L_2 = (2, 4)$ as respectivas listas de atravessamento de dois raios distintos. Observa-se que todos os identificadores das listas L_1 e L_2 são diferentes e que, portanto, ativam elementos de processamento distintos.

Referências Bibliográficas

[1] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. *Real-Time Rendering*. 3ª ed. Natick, MA, USA, A. K. Peters, Ltd., 2008.

[2] FRIEDRICH, H., GÜNTHER, J., DIETRICH, A., et al. "Exploring the Use of Ray Tracing for Future Games". In: *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pp. 41–50, New York, NY, USA, 2006. ACM Press.

[3] GLASSNER, A. S. *An introduction to ray tracing*. 1ª ed. London, UK, UK, Academic Press Ltd., 1989.

[4] SUFFERN, K. *Ray Tracing from the Ground Up*. 1ª ed. Natick, MA, USA, A. K. Peters, Ltd., 2007.

[5] PARKER, S., PARKER, M., LIVNAT, Y., et al. "Interactive Ray Tracing for Volume Visualization", *IEEE Transactions on Visualization and Computer Graphics*, v. 5, pp. 238–250, 1999.

[6] HURLEY, J. "Ray Tracing Goes Mainstream", *Intel Technology Journal*, v. 9, n. 2, pp. 99–107, Maio 2005.

[7] CAMERON, C. B. "Using FPGAs to Supplement Ray-Tracing Computations on the Cray XD-1", *HPCMP Users Group Conference*, v. 0, pp. 359–363, 2007.

[8] WALD, I., SLUSALLEK, P., BENTHIN, C. "Interactive Distributed Ray Tracing of Highly Complex Models". In: *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pp. 277–288. Springer, 2001.

[9] PURCELL, T. J., BUCK, I., MARK, W. R., et al. "Ray tracing on programmable graphics hardware". In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, p. 268, New York, NY, USA, 2005. ACM.

[10] RALOVICH, K. "Implementing and Analyzing a GPU Ray Tracer". In: *CESCG '07: Central European Seminar on Computer Graphics for students*, p. 6, 2007.

[11] CHRISTEN, M. *Ray Tracing on GPU*. Ms.c., University of Applied Sciences Basel, 2005.

[12] SLUSALLEK, P., SHIRLEY, P., MARK, W., et al. "Introduction to real-time ray tracing". In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, p. 1, New York, NY, USA, 2005. ACM.

[13] NVIDIA. "Nvidia Corporation homepage". , 2010. Disponível em: <`http://www.nvidia.com/`>. Acesso em: janeiro de 2010.

[14] KHRONOS. "OpenCL - The open standard for parallel programming of heterogeneous systems". , 2009. Disponível em: <`http://www.khronos.org/opencl/`>. Acesso em: janeiro de 2010.

[15] CHU, P. P. *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. New York,NY, Wiley-Interscience, 2008.

[16] EL-GHAZAWI, T., EL-ARABY, E., HUANG, M., et al. "The Promise of High-Performance Reconfigurable Computing", *Computer*, v. 41, n. 2, pp. 69–76, 2008.

[17] LU, S.-L. L., YIANNACOURAS, P., KASSA, R., et al. "An FPGA-based Pentium®in a complete desktop system". In: *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pp. 53–59, New York, NY, USA, 2007. ACM.

[18] ZEMCIK, P. "Hardware acceleration of graphics and imaging algorithms using FPGAs". In: *SCCG '02: Proceedings of the 18th spring conference on Computer graphics*, pp. 25–32, New York, NY, USA, 2002. ACM.

[19] WOOP, S., SCHMITTLER, J., SLUSALLEK, P. "RPU: a programmable ray processing unit for realtime ray tracing". In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 434–444, New York, NY, USA, 2005. ACM.

[20] SCHMITTLER, J., WOOP, S., WAGNER, D., et al. "Realtime ray tracing of dynamic scenes on an FPGA chip". In: *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 95–106, New York, NY, USA, 2004. ACM.

[21] WALD, I., SLUSALLEK, P., BENTHIN, C., et al. "Interactive Rendering with Coherent Ray Tracing". In: *Computer Graphics Forum*, pp. 153–164, 2001.

[22] MAXIM, S., ALEXEI, S., ALEXANDER, K. "Ray-Triangle Intersection Algorithm for Modern CPU Architectures". In: *Proceedings of GraphiCon, 2007*, pp. 33–39, 2007.

[23] WALD, I., FRIEDRICH, H., MARMITT, G., et al. "Faster Isosurface Ray Tracing Using Implicit KD-Trees", *IEEE Transactions on Visualization and Computer Graphics*, v. 11, n. 5, pp. 562–572, 2005.

[24] CARR, N. A., HALL, J. D., HART, J. C. "The ray engine". In: *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[25] NVIDIA. "Whitepaper Nvidia GF100". , 2010. Disponível em: <`http://www.nvidia.com/object/IO_86775.html`>. Acesso em: janeiro de 2010.

[26] NERY, A. S., NEDJAH, N., FRANÇA, F. M. G. "GridRT: A massively parallel architecture for ray-tracing using uniform grids". In: *DSD '09: Euromicro Conference on Digital System Design*, pp. 211–216, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[27] NERY, A. S., NEDJAH, N., FRANÇA, F. M. G. "A massively parallel hardware architecture for ray-tracing", *International Journal of High Performance Systems Architecture 2009 - Vol. 2, No.1 pp. 26 - 34*, pp. 26–34, 2009.

[28] GRAPHICS, M. "ModelSim Xilinx Edition". , 2009. Disponível em: <`http://www.xilinx.com/ise/verification/mxe_details.html`>. Acesso em: março de 2008.

[29] OPENMP. "The OpenMP API specification for parallel programming". , 2008. Disponível em: <`http://openmp.org/wp/`>. Acesso em: agosto de 2009.

[30] OPENMPI. "Open Source Message Passing Interface". , 2004. Disponível em: <`http://www.open-mpi.org/`>. Acesso em: agosto de 2009.

[31] LABORATORY, S. C. G. "The Stanford 3D Scanning Repository". , 2009. Disponível em: <`http://www-graphics.stanford.edu/data/3Dscanrep/`>. Acesso em: fevereiro de 2009.

[32] XILINX. *MicroBlaze Processor Reference Guide*, 2008. Disponível em: <`http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`>. Acesso em: novembro de 2009.

[33] XILINX. *Xilinx Synthesis Technology: User Guide*, 2008. Disponível em: <`http://www.xilinx.com/itp/xilinx5/pdf/docs/xst/xst.pdf`>. Acesso em: janeiro de 2009.

[34] XILINX. *Virte-5 user guide UG190 (v5.1)*, 2009. Disponível em: <http://www.xilinx.com/support/documentation/user_guides/ug190.pdf>. Acesso em: novembro de 2009.

[35] WHITTED, T. "An improved illumination model for shaded display", *Commun. ACM*, v. 23, n. 6, pp. 343–349, 1980.

[36] SHIRLEY, P. "Ray-Object Intersections". In: *Realistic Ray Tracing*, 1ª ed., pp. 34–38, Natick, MA, USA, A. K. Peters, Ltd., 2000.

[37] SHIRLEY, P. "Polygon Intersection via Barycentric Coordinates". , 1992. Disponível em: <http://tog.acm.org/resources/RTNews/html/rtnv5n3.html#art4>. Acesso em: agosto de 2008.

[38] PHONG, B. T. "Illumination for computer generated pictures", *Commun. ACM*, v. 18, n. 6, pp. 311–317, 1975.

[39] WALD, I., BENTHIN, C., SLUSALLEK, P. "Distributed Interactive Ray Tracing of Dynamic Scenes". In: *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, p. 11, Washington, DC, USA, 2003. IEEE Computer Society.

[40] CHRISTENSEN, P., FONG, J., LAUR, D., et al. "Ray Tracing for the Movie 'Cars"', *Symposium on Interactive Ray Tracing*, pp. 1–6, 2006.

[41] OF VISION RAYTRACER PTY. LTD., P. "POV-Ray: Persistence of View Ray Tracing". , 2009. Disponível em: <http://www.povray.org/>. Acesso em: maio de 2009.

[42] GUSTAVO, B., BERT, B., MICHELE, C., et al. "YafaRay". , 2009. Disponível em: <http://www.yafaray.org/>. Acesso em: maio de 2009.

[43] PARKER, S., MARTIN, W., PIKE J. SLOAN, P., et al. "Interactive Ray Tracing". In: *In Symposium on interactive 3D graphics*, pp. 119–126, 1999.

[44] HUMPHREYS, G., ANANIAN, C. S. *TigerSHARK: A Hardware Accelerated Ray-tracing Engine*. Relatório técnico, Princeton University, 1996.

[45] TODMAN, T., LUK, W. "Reconfigurable Designs for Ray Tracing", *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, v. 0, pp. 300–301, 2001.

[46] FUJIMOTO, A., TANAKA, T., IWATA, K. "ARTS: accelerated ray-tracing system", *Tutorial: computer graphics; image synthesis*, pp. 148–159, 1988.

[47] AMANATIDES, J., WOO, A. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *Eurographics '87*, pp. 3–10, 1987.

[48] HAVRAN, V., PRIKRYL, J., PURGATHOFER, W. *Statistical Comparison of Ray-Shooting Efficiency Schemes*. Relatório técnico, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2000. Disponível em: <http://www.cg.tuwien.ac.at/research/publications/2000/Havran-2000-SCR/>.

[49] SZIRMAY-KALOS, L., HAVRAN, V., BALÁZS, B., et al. "On the efficiency of ray-shooting acceleration schemes". In: *SCCG '02: Proceedings of the 18th spring conference on Computer graphics*, pp. 97–106, New York, NY, USA, 2002. ACM.

[50] WILSON, P. *Design Recipes for FPGAs*. 1ª ed. Burlington, MA, UK, Newnes, 2007.

[51] CHU, P. P. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Newark, NJ, Wiley-IEEE Press, 2006.

[52] XILINX. "ML505/ML506/ML507 Evaluation Platform User Guide". , 2008. Disponível em: <http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf>. Acesso em: novembro de 2009.

[53] XILINX. *Fast Simplex Link v2.11b*, 2009. Disponível em: <http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf>. Acesso em: novembro de 2009.

[54] XILINX. *XPS UARTLite*, 2009. Disponível em: <http://www.xilinx.com/support/documentation/ip_documentation/xps_uartlite.pdf>. Acesso em: novembro de 2009.

[55] HENNESSY, J. L., PATTERSON, D. A. *Computer organization and design: the hardware/software interface*. 3ª ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2004.

[56] XILINX. "Floating-Point Operator v5.0". , 2008. Disponível em: <http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf>. Acesso em: janeiro de 2009.

[57] KOREN, I. *Computer arithmetic algorithms*. 2ª ed. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 2001.

Apêndice A

Código fonte da implementação em OpenMP

```
#define PI (355/113)

#include "Point.h"
#include "IPoint.h"
#include "Vector3D.h"
#include "Triangle.h"
#include "Mesh.h"
#include "Ray.h"
#include "Viewport.h"
#include "Camera.h"
#include "World.h"
#include <string.h>
#include <float.h>
#include "Grid.h"

#include <stdlib.h>
#include <omp.h>

#include "image.hpp"
#include "png.hpp"

float clamp(float x, float min, float max) {
  return (x < min ? min : (x > max ? max : x));
}

float maior(float a, float b) {
```

```
  if (a > b)
    return a;
  else
    return b;
}

int main(int argc, char ** argv) {
  int i = 0;
  int j = 0;
  int tid = 0; //private thread id
  int numcells = 0;
  time_t total_time = 0;
  char path[100];
  bool flag = false;

  strcpy(path,
      "models/bunny/reconstruction/bun_zipper.ply");
  World * nworld = new World();

  if (nworld == NULL) {
    printf(" Cannot create world! \n");
    return 0;
  }

  printf("World created...\n");
  World * world = nworld->readFilePLY(path);
  delete nworld;

  printf(" Done reading scene ...\n");

  Light * light = world->getLight();
  Camera * camera = world->getCamera();
  Viewport * viewport = camera->getViewport();

  png::image<png::rgb_pixel>
      image(viewport->getWidth(),
          viewport->getHeight());

  Mesh * mesh = world->getMesh();
```

```
Point *min = mesh->getMin();
Point *max = mesh->getMax();
min->print();
max->print();
delete min;
delete max;

Grid * grid = new Grid();
grid->setupCells(*mesh);

//SET the number of threads to be equal to the number of voxels
numcells = grid->getNumCells();
omp_set_num_threads(numcells);

printf("Creating rays...\n");
camera->generateRays(); //must be called before getRays();

printf("Rays: \n");
Ray *** rs = camera->getRays();

printf("Rays created...\n");

float ka = 0.2;
float kd = 0.03;
float ks = 0.003;
float ls = 0.02;
int e = 2;
Color cd = Color(1, 1, 1);

IPoint * res = NULL;
IPoint ** shared_res = new IPoint*[numcells];
int * traverse = new int[numcells];

float t = FLT_MAX;
int n = 0;

for (n = 0; n < numcells; n++)
  shared_res[n] = NULL;
```

```
  Triangle ** tri = NULL;

#pragma omp parallel shared(grid) private(tid,tri)
  {
    tid = omp_get_thread_num();
    printf("Thread %d reporting.\n", tid);
  }

  IPoint * resultado = NULL;
  float tt = FLT_MAX;
  unsigned int w = 0;

  total_time = time(NULL);
  for (i = 0; i < viewport->getWidth(); i++) {
    printf("i = %d\n", i);
    for (j = 0; j < viewport->getHeight(); j++) {

      traverse = grid->traverseGrid(*rs[i][j]);

#pragma omp parallel shared(shared_res,grid,numcells,traverse)
                    private(tid,tri,flag,resultado,tt,w)
      {
        tid = omp_get_thread_num();
        //traverse = grid->traverseGrid(*rs[i][j]);
        tri = grid->getTriangleArray(tid);

        if (tri != NULL) {
          for (n = 0; n < numcells; n++) {
            if (tid == traverse[n]) {
              tt = FLT_MAX;
              resultado = NULL;
              flag = false;

              for (w = 0; w
                  < grid->getTriangleArraySize(
                      tid); w++) {
                if (tri[w] != NULL) {
                  resultado = tri[w]->intersect(
                      *rs[i][j]);
```

```
            if (resultado != NULL) {
              if (resultado->getT() < tt
                  && resultado->getT() > 0) {
                tt = resultado->getT();
                shared_res[tid] = resultado;
              }
            }

          }

          int q = 0;
          for (q = 0; traverse[q] != tid; q++) {
            if (shared_res[traverse[q]]
                != NULL) {
              shared_res[tid] = NULL;
              flag = true;
            }
          }

          if (flag == true)
            break;

        }
      }
    }
  }
  delete[] tri;
}

t = FLT_MAX;
for (n = 0; n < numcells; n++) {
  if (shared_res[n] != NULL) {
    if (shared_res[n]->getT() < t
        && shared_res[n]->getT() > 0) {
      t = shared_res[n]->getT();
      res = shared_res[n];
    }
  }
}
```

```
if (res != NULL) {

  //Shade function
  //compute light dir
  float x = light->getX() - res->getX();
  float y = light->getY() - res->getY();
  float z = light->getZ() - res->getZ();
  Point dir(x, y, z);

  Vector3D vaux = Vector3D();
  vaux.setDir(dir);

  Vector3D *n = res->getIntersectedNormal();
  float dot = vaux.dotProduct(*n);

  //ambient
  float r = cd.getRed() * ka
      * light->getRed() * ls;
  float g = cd.getGreen() * ka
      * light->getGreen() * ls;
  float b = cd.getBlue() * ka
      * light->getBlue() * ls;

  //lambertian
  if (dot > 0) {
    float r2 = (kd * cd.getRed() / PI);
    float g2 = (kd * cd.getGreen() / PI);
    float b2 = (kd * cd.getBlue() / PI);

    //specular
    Vector3D * incidence =
        rs[i][j]->getDir();

    float dot2 = incidence->dotProduct(*n);
    dot2 *= 2;

    n->scalarMult(dot2);
    incidence->scalarMult(-1);
```

```
  incidence->add(*n);

  x = camera->getEyeX() - res->getX();
  y = camera->getEyeY() - res->getY();
  z = camera->getEyeZ() - res->getZ();
  dir.setX(x);
  dir.setY(y);
  dir.setZ(z);
  vaux.setDir(dir);

  dot2 = vaux.dotProduct(*incidence);
  dot2 = -1;

  if (dot2 > 0) {
    dot2 = pow(dot2, e);

    r2 += ks * dot2;
    g2 += ks * dot2;
    b2 += ks * dot2;
  }

  r2 *= ls * light->getRed() * dot;
  g2 *= ls * light->getGreen() * dot;
  b2 *= ls * light->getBlue() * dot;

  r = r + r2;
  g = g + g2;
  b = b + b2;

  delete incidence;

}

float m = maior(maior(r, g), b);
if (m > 1.0) {
  r = r / m;
  g = g / m;
  b = b / m;
```

```
      }

      image[j][i] = png::rgb_pixel(r * 250, g
          * 250, b * 250);

      delete n;
    } else {
      image[j][i] = png::rgb_pixel(0, 0, 0);
    }

    for (n = 0; n < numcells; n++) {
      shared_res[n] = NULL;
    }

    res = NULL;
  }
}

total_time = time(NULL) - total_time;
printf("\nTotal time = %ld\n", total_time);
image.write("output.png");

delete camera;
delete world;

camera = NULL;
world = NULL;

for (i = 0; i < viewport->getWidth(); i++) {
  for (j = 0; j < viewport->getHeight(); j++) {
    delete rs[i][j];
  }
}

for (i = 0; i < viewport->getWidth(); i++)
  delete rs[i];

delete[] rs;
rs = NULL;
```

```
  delete viewport;
  viewport = NULL;

  delete mesh;
  mesh = NULL;

  printf("Done...\n");

  return 0;

}
```

Apêndice B

Código fonte da implementação em MPI

```
//#include <mpi.h>

#define PI (355/113)

#include "Point.h"
#include "IPoint.h"
#include "Vector3D.h"
#include "Triangle.h"
#include "Mesh.h"
#include "Ray.h"
#include "Viewport.h"
#include "Camera.h"
#include "World.h"
#include <string.h>
#include <float.h>
#include "Grid.h"

#include <stdlib.h>
#include <mpi.h>
#include <time.h>

#include "image.hpp"
#include "png.hpp"

float clamp(float x, float min, float max) {
  return (x < min ? min : (x > max ? max : x));
```

```
}

float maior(float a, float b) {
  if (a > b)
    return a;
  else
    return b;
}

int main(int argc, char ** argv) {
  int i = 0;
  int j = 0;

  int * cells_size = NULL;
  int total_cells = 0;
  float * cena = NULL;
  float * rdata = NULL;
  Triangle ** cell_triangles = NULL;
  int * traverse = NULL;
  float lowest = FLT_MAX;

  int rank, size, rc, tag, tag1, tag2, tag3,
      trav_size;
  tag = 0;
  tag1 = 1;
  tag2 = 2;
  tag3 = 3;
  trav_size = 0;
  MPI_Status status;
  MPI_Request req1, req2, req3, req4, req5, req6,
      req7;
  int primeira = 0;

  rc = MPI_Init(&argc, &argv);
  if (rc != MPI_SUCCESS) {
    printf("Cannot initialize MPI...\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
    return 0;
  }
```

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

char path[100];

float ka = 0.2;
float kd = 0.03;
float ks = 0.03;
float ls = 0.2;
int e = 2;
Color cd = Color(1, 1, 1);

//dragon and bunny
Point * laux = new Point(1000, 1000, 1000);
Color * caux = new Color(1, 1, 1);

Light * light = new Light(*laux, *caux);
delete laux;
delete caux;

//bunny
Point location(-100, 200, 500);
Point lookat(-26.677546, 150.07388, 8.952819);

Viewport * viewport = new Viewport(240, 320);
Camera * camera = new Camera(location, lookat,
    *viewport);

double total_time = 0;

double int_time = 0;
double total_int_time = 0;

double traverse_time = 0;
double total_traverse_time = 0;

double send_ray_time = 0;
double total_send_ray_time = 0;
```

```
int nr = 0;

if (rank == size - 1) {
  printf("\tMASTER %d reporting...\n", rank);
  strcpy(path,
      "models/bunny/reconstruction/bun_zipper.ply");

  World * world = new World();
  printf("\nReading Scene...\n");
  World * nworld = world->readFilePLY(path);
  delete world;

  printf("\nScene succesfully read...\n");

  Mesh *mesh = nworld->getMesh();
  Point *min = mesh->getMin();
  Point *max = mesh->getMax();
  min->print();
  max->print();
  delete min;
  delete max;
  Grid * grid = new Grid();
  grid->setupCells(*mesh);

  cells_size = grid->getCellsSize();
  total_cells = grid->getNumCells();

  grid->print();

  delete nworld;
  delete mesh;

  printf("Broadcasting...\n");
  for (int proc = 0; proc < size - 1; proc++) {
    MPI_Send(&total_cells, 1, MPI_INT, proc,
        tag, MPI_COMM_WORLD);
    MPI_Send(cells_size, total_cells, MPI_INT,
        proc, tag, MPI_COMM_WORLD);
```

```
  cena = grid->getTriangleFloatArray(proc);
  MPI_Send(cena, cells_size[proc] * 9,
      MPI_FLOAT, proc, tag, MPI_COMM_WORLD);
  delete cena;

}

printf("Creating rays...\n");
camera->generateRays(); //must be called before getRays();

printf("Rays: \n");
Ray *** rs = camera->getRays();

printf("Rays created...\n");
delete camera;
MPI_Barrier(MPI_COMM_WORLD);
total_time = time(NULL);
for (i = 0; i < viewport->getWidth(); i++) {
  printf("i = %d\n", i);
  for (j = 0; j < viewport->getHeight(); j++) {
    nr++;
    traverse = grid->traverseGrid(*rs[i][j]);

    for (int proc = 0; proc < size; proc++) {
      traverse_time = MPI_Wtime();
      MPI_Send(traverse, total_cells,
          MPI_INT, proc, tag1 + nr,
          MPI_COMM_WORLD);
      traverse_time = MPI_Wtime()
          - traverse_time;
      total_traverse_time += traverse_time;
    }
    trav_size = 0;
    for (int count = 0; count < total_cells; count++) {
      if (traverse[count] != -1) {
        trav_size++;
        rdata = rs[i][j]->getFloatArray();
        send_ray_time = MPI_Wtime();
```

```
            MPI_Send(rdata, 6, MPI_FLOAT,
                traverse[count], tag2 + nr,
                MPI_COMM_WORLD);

            send_ray_time = MPI_Wtime()
                - send_ray_time;
            total_send_ray_time += send_ray_time;
            delete rdata;
          }
        }

        delete traverse;
      }

    }

    printf(" %d terminou \n", rank);
    MPI_Barrier(MPI_COMM_WORLD);

    total_time = time(NULL) - total_time;
    printf("\nTotal time: %.2f", total_time);

    printf("\nSend ray time: %.2f",
        total_send_ray_time);
    printf("\nSend traverse time: %.2f",
        total_traverse_time);

    delete viewport;
    delete grid;

    MPI_Finalize();
  } else //PE
  {
    printf("\tPE %d reporting...\n", rank);
    MPI_Recv(&total_cells, 1, MPI_INT, size - 1,
        tag, MPI_COMM_WORLD, &status);

    cells_size = new int[total_cells];
```

```
MPI_Recv(cells_size, total_cells, MPI_INT,
    size - 1, tag, MPI_COMM_WORLD, &status);
printf("\tPE %d number of objects = %d\n",
    rank, cells_size[rank]);

cena = new float[cells_size[rank] * 9];
MPI_Recv(cena, cells_size[rank] * 9,
    MPI_FLOAT, size - 1, tag, MPI_COMM_WORLD,
    &status);

printf("\tPE %d Scene received...\n", rank);

cell_triangles
    = new Triangle*[cells_size[rank]];

j = 0;
for (i = 0; i < cells_size[rank] * 9; i = i
    + 9) {
  Point *v0 = new Point(cena[i + 0], cena[i
      + 1], cena[i + 2]);
  Point *v1 = new Point(cena[i + 3], cena[i
      + 4], cena[i + 5]);
  Point *v2 = new Point(cena[i + 6], cena[i
      + 7], cena[i + 8]);

  cell_triangles[j] = new Triangle(*v0, *v1,
      *v2);
  j++;

  delete v0;
  delete v1;
  delete v2;
}

delete cena;

printf("\tPE %d Scene triangles done...\n",
    rank);
MPI_Barrier(MPI_COMM_WORLD);
```

```
int interrupt, interrupt2_in, interrupt2_out =
    -1;
int interrupt_back = -1;

for (int ii = 0; ii < viewport->getWidth(); ii++) {
  printf(" %d ii = %d\n", rank, ii);
  for (int jj = 0; jj < viewport->getHeight(); jj++) {
    nr++;

    traverse = new int[total_cells];
    MPI_Recv(traverse, total_cells, MPI_INT,
        size - 1, tag1 + nr, MPI_COMM_WORLD,
        &status);

    int maximo = 0;
    trav_size = 0;
    for (i = 0; i < total_cells; i++) {
      if (traverse[i] != -1) {
        trav_size++;
        if (cells_size[traverse[i]] > maximo) {
          maximo = cells_size[traverse[i]];
        }
      } else {
        break;
      }
    }

    Ray * rr = NULL;
    interrupt2_in = -1;
    interrupt2_out = -1;
    interrupt = -1;
    //Receive Ray Data
    for (i = 0; i < trav_size; i++) {
      if (traverse[i] == rank) {
        //receive ray data
        rdata = new float[6];
        MPI_Recv(rdata, 6, MPI_FLOAT, size
            - 1, tag2 + nr, MPI_COMM_WORLD,
            &status);
```

```
Point * o = new Point(rdata[0],
    rdata[1], rdata[2]);
Point * d = new Point(rdata[3],
    rdata[4], rdata[5]);
rr = new Ray(*o, *d);
delete o;
delete d;
delete rdata;

//begin intersection
lowest = FLT_MAX;
IPoint *aux = NULL;
IPoint *res = NULL;

if (rr != NULL) {
  for (j = 0; j < cells_size[rank]; j++) {
    if (cell_triangles[j] != NULL) {
      aux
          = cell_triangles[j]->intersect(
              *rr);

      if (aux != NULL) {
        if (aux->getT() < lowest) {
          lowest = aux->getT();
          res = aux;
        }
      }
    }
  }
}

//Interrupts
if (trav_size > 1) {
  if (res != NULL) {
    if (i == 0) {
      interrupt = -2;
      int_time = MPI_Wtime();
      MPI_Send(&interrupt, 1,
          MPI_INT, traverse[i + 1],
```

```
        traverse[i] + nr,
        MPI_COMM_WORLD);
    int_time = MPI_Wtime()
        - int_time;
    total_int_time += int_time;

    lowest = res->getT();
  } else if (i == trav_size - 1) {

    int_time = MPI_Wtime();
    MPI_Recv(&interrupt, 1,
        MPI_INT, traverse[i - 1],
        traverse[i - 1] + nr,
        MPI_COMM_WORLD, &status);
    int_time = MPI_Wtime()
        - int_time;
    total_int_time += int_time;

    if (interrupt == -1)
      lowest = res->getT();
    else
      break;
  } else {
    int_time = MPI_Wtime();
    MPI_Recv(&interrupt, 1,
        MPI_INT, traverse[i - 1],
        traverse[i - 1] + nr,
        MPI_COMM_WORLD, &status);
    int_time = MPI_Wtime()
        - int_time;
    total_int_time += int_time;

    interrupt_back = interrupt;
    interrupt = -2;

    int_time = MPI_Wtime();
    MPI_Send(&interrupt, 1,
        MPI_INT, traverse[i + 1],
        traverse[i] + nr,
```

```
        MPI_COMM_WORLD);
    int_time = MPI_Wtime()
        - int_time;
    total_int_time += int_time;

    if (interrupt_back == -1)
      lowest = res->getT();
    else
      break;
  }
} else {
  if (i == 0) {
    interrupt = -1;

    int_time = MPI_Wtime();
    MPI_Send(&interrupt, 1,
        MPI_INT, traverse[i + 1],
        traverse[i] + nr,
        MPI_COMM_WORLD);
    int_time = MPI_Wtime()
        - int_time;
    total_int_time += int_time;
  } else if (i == trav_size - 1) {
    int_time = MPI_Wtime();
    MPI_Recv(&interrupt, 1,
        MPI_INT, traverse[i - 1],
        traverse[i - 1] + nr,
        MPI_COMM_WORLD, &status);
    int_time = MPI_Wtime()
        - int_time;
    total_int_time += int_time;
  } else {
    int_time = MPI_Wtime();
    MPI_Recv(&interrupt, 1,
        MPI_INT, traverse[i - 1],
        traverse[i - 1] + nr,
        MPI_COMM_WORLD, &status);
    int_time = MPI_Wtime()
        - int_time;
```

```
        total_int_time += int_time;

        //aqui so repassa adiante
        int_time = MPI_Wtime();
        MPI_Send(&interrupt, 1,
            MPI_INT, traverse[i + 1],
            traverse[i] + nr,
            MPI_COMM_WORLD);
        int_time = MPI_Wtime()
            - int_time;
        total_int_time += int_time;

        if (interrupt == -2)
          break;
      }
    }

}

if ((res != NULL) && (lowest > 0)
    && (interrupt != -2)) {

  //Shade function
  //compute light dir
  float x = light->getX()
      - res->getX();
  float y = light->getY()
      - res->getY();
  float z = light->getZ()
      - res->getZ();
  Point dir(x, y, z);

  Vector3D vaux = Vector3D();
  vaux.setDir(dir);

  Vector3D *n =
      res->getIntersectedNormal();
  float dot = vaux.dotProduct(*n);
```

```
float r = cd.getRed() * ka
    * light->getRed() * ls;
float g = cd.getGreen() * ka
    * light->getGreen() * ls;
float b = cd.getBlue() * ka
    * light->getBlue() * ls;

if (dot > 0) {
  float r2 =
      (kd * cd.getRed() / PI);
  float g2 = (kd * cd.getGreen()
      / PI);
  float b2 = (kd * cd.getBlue()
      / PI);

  Vector3D * incidence =
      rr->getDir();

  float dot2 =
      incidence->dotProduct(*n);
  dot2 *= 2;

  n->scalarMult(dot2);
  incidence->scalarMult(-1);

  incidence->add(*n);

  x = camera->getEyeX()
      - res->getX();
  y = camera->getEyeY()
      - res->getY();
  z = camera->getEyeZ()
      - res->getZ();
  dir.setX(x);
  dir.setY(y);
  dir.setZ(z);
  vaux.setDir(dir);

  dot2
```

```
            = vaux.dotProduct(*incidence);
          dot2 = -1;

          if (dot2 > 0) {
            dot2 = pow(dot2, e);

            r2 += ks * dot2;
            g2 += ks * dot2;
            b2 += ks * dot2;
          }

          r2 *= ls * light->getRed() * dot;
          g2 *= ls * light->getGreen()
              * dot;
          b2 *= ls * light->getBlue() * dot;

          r = r + r2;
          g = g + g2;
          b = b + b2;

          delete incidence;

        }

        float m = maior(maior(r, g), b);
        if (m > 1.0) {
          r = r / m;
          g = g / m;
          b = b / m;
        }

        //cor: r,g,b
        delete n;

      }
    }
  }

if (rr != NULL)
```

```
          delete rr;

        if (traverse != NULL)
          delete traverse;

       }//end for j

    }//end for i


    printf(" %d terminou, int_time: %.2f \n",
        rank, total_int_time);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();

  }

  return 0;

}
```

Apêndice C

Código fonte dos componentes VHDL

C.1 Microprograma

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity program is
port(
 clk : in std_logic;
 micropc : in std_logic_vector(6 downto 0);
 control : out std_logic_vector(20 downto 0)
);
end program;

architecture beh of program is

signal i : integer :=0;
type mem_rom is array(0 to 97) of std_logic_vector(22 downto 0);
signal vd: mem_rom;

begin

 vd(0) <=  "00000000000000000000100"; --busca
 vd(1) <=  "00000000000000000000001"; --decodifica
 vd(2) <=  "10000000000000000000000";
```

```
vd(3) <=  "00000000000100000000000"; --add
vd(4) <=  "00000000000000100000000";
vd(5) <=  "01000000000000000000000";
vd(6) <=  "00000000000100000100000"; --sub
vd(7) <=  "00000000000000100100000";
vd(8) <=  "01000000000000000100000";
vd(9) <=  "00000000000100001000000"; --or
vd(10) <= "00000000000000101000000";
vd(11) <= "01000000000000001000000";
vd(12) <= "00000000000100001100000"; --and
vd(13) <= "00000000000000101100000";
vd(14) <= "01000000000000001100000";
vd(15) <= "00000000000100010000000"; --addfp
vd(16) <= "00000000000000010000000";
vd(17) <= "00000000000000010000000";
vd(18) <= "00000000000000010000000";
vd(19) <= "00000000000000110000000";
vd(20) <= "01000000000000010000000";
vd(21) <= "00000000000100010001000"; --subfp = soma invertendo B
vd(22) <= "00000000000000010001000";
vd(23) <= "00000000000000010001000";
vd(24) <= "00000000000000010001000";
vd(25) <= "00000000000000110001000";
vd(26) <= "01000000000000010001000";
vd(27) <= "00000000000100010100000"; --mulfp
vd(28) <= "00000000000000010100000";
vd(29) <= "00000000000000010100000";
vd(30) <= "00000000000000010100000";
vd(31) <= "00000000000000110100000";
vd(32) <= "01000000000000010100000";
vd(33) <= "00000000000100011000000"; --divfp
vd(34) <= "00000000000000011000000";
vd(35) <= "00000000000000011000000";
vd(36) <= "00000000000000011000000";
vd(37) <= "00000000000000011000000";
vd(38) <= "00000000000000011000000";
vd(39) <= "00000000000000011000000";
vd(40) <= "00000000000000011000000";
vd(41) <= "00000000000000011000000";
```

```
vd(42) <= "00000000000000011000000";
vd(43) <= "00000000000000111000000";
vd(44) <= "01000000000000011000000";
vd(45) <= "00000000000100011100000"; --sqrtfp
vd(46) <= "00000000000000011100000";
vd(47) <= "00000000000000011100000";
vd(48) <= "00000000000000011100000";
vd(49) <= "00000000000000011100000";
vd(50) <= "00000000000000011100000";
vd(51) <= "00000000000000011100000";
vd(52) <= "00000000000000011100000";
vd(53) <= "00000000000000011100000";
vd(54) <= "00000000000000011100000";
vd(55) <= "00000000000000111100000";
vd(56) <= "01000000000000011100000";
vd(57) <= "00000000001000000000000"; --addi
vd(58) <= "00000000001100000000000";
vd(59) <= "00000000001000100000000";
vd(60) <= "01000000001000000000000";
vd(61) <= "00000010001000000000000"; --lw
vd(62) <= "00000010001100000000000";
vd(63) <= "00000010001000000000000";
vd(64) <= "00000010001001000000000";
vd(65) <= "00000010001000000000000";
vd(66) <= "00000010001000100000000";
vd(67) <= "01000010001000000000000";
vd(68) <= "00000000000000000000000"; --jump
vd(69) <= "00000000000000000000010";
vd(70) <= "01000000000000000000000";
vd(71) <= "00000000100100000000000"; --cmp_fp
vd(72) <= "00000000100000000000000";
vd(73) <= "00000000100000000000000";
vd(74) <= "00000000100000000000000";
vd(75) <= "00100000100000000000000";
vd(76) <= "01000000100000000000000";
vd(77) <= "00000000100100000000000"; --cmp
vd(78) <= "00100000100000000000000";
vd(79) <= "01000000100000000000000";
vd(80) <= "00001000000000000000000"; --get ray
```

```
 vd(81) <= "00000000000000000000000";
 vd(82) <= "00000000000000000000000";
 vd(83) <= "00000000000000000000000";
 vd(84) <= "01001000000000000000000";
 vd(85) <= "01000000000000000000000";
 vd(86) <= "00000000101100000000000"; --st
 vd(87) <= "00000001101010000000000";
 vd(88) <= "01000000101000000000000";
 vd(89) <= "00000000000000000000000";
 vd(90) <= "00000100000000000000000"; --input
 vd(91) <= "00000100000000100000000";
 vd(92) <= "01000100000000000000000";
 vd(93) <= "00000000000000000000000";
 vd(94) <= "00000000110000000000000"; --output
 vd(95) <= "01001000100000000000000";
 vd(96) <= "01000000100000000000000";
 vd(97) <= "00010000000000000000000"; --hlt

 process(clk)
  begin
   if(rising_edge(clk))then
    i <= i + 1;

    if(vd(i)(22)='1')then
     i <= conv_integer(micropc);
    end if;

    if(vd(i)(21)='1')then
     i <= 0;
    end if;

   end if;

 end process;
 control <= vd(i)(20 downto 0);
end beh;
```

C.2 Decodificador de Instruções

```
library ieee;
use ieee.std_logic_1164.all;

entity instrDecoder is
port(
 opcode : in std_logic_vector(4 downto 0);
 c : in std_logic_vector(1 downto 0);
 micropc : out std_logic_vector(6 downto 0);
 muxOrder : out std_logic
);
end instrDecoder;

architecture beh of instrDecoder is

constant add : std_logic_vector(4 downto 0) := "00000";
constant sub : std_logic_vector(4 downto 0) := "00001";
constant or2 : std_logic_vector(4 downto 0) := "00010";
constant and2 : std_logic_vector(4 downto 0) := "00011";
constant addfp : std_logic_vector(4 downto 0) := "00100";
constant subfp : std_logic_vector(4 downto 0) := "00101";
constant mulfp : std_logic_vector(4 downto 0) := "00110";
constant divfp : std_logic_vector(4 downto 0) := "00111";
constant sqrtfp : std_logic_vector(4 downto 0) := "01000";
constant addi : std_logic_vector(4 downto 0) := "01001";
constant cmp : std_logic_vector(4 downto 0) := "01010";
constant je : std_logic_vector(4 downto 0) := "01011";
constant jl : std_logic_vector(4 downto 0) := "01100";
constant cmpfp : std_logic_vector(4 downto 0) := "01101";
constant lw : std_logic_vector(4 downto 0) := "01110";
constant st : std_logic_vector(4 downto 0) := "01111";
constant input : std_logic_vector(4 downto 0) := "10000";
constant output : std_logic_vector(4 downto 0)  := "10001";
constant hlt : std_logic_vector(4 downto 0)  := "10010";
constant j : std_logic_vector(4 downto 0)  :=  "10011";
constant jg : std_logic_vector(4 downto 0)  := "10100";

begin
```

```
process(opcode,c)
begin
 case opcode is
  when lw =>
   micropc <= "0101101";
  when st =>
   micropc <= "0111111";
  when add =>
   micropc <= "0000011";
  when sub =>
   micropc <= "0000101";
  when or2 =>
   micropc <= "0000111";
  when and2 =>
   micropc <= "0001001";
  when addfp =>
   micropc <= "0001011";
  when subfp =>
   micropc <= "0001111";
  when mulfp =>
   micropc <= "0010011";
  when divfp =>
   micropc <= "0010111";
  when sqrtfp =>
   micropc <= "0100001";
  when addi =>
   micropc <= "0101011";
  when cmp =>
   micropc <= "0110111";
  when cmpfp =>
   micropc <= "0110011";
  when je =>
   if(c = "00" or c = "10")then --A=B
    micropc <= "0110001"; --jump 49
   else
    micropc <= "0111110"; --continue 62
   end if;
  when jl =>
   if(c = "11")then --A<B
```

```
     micropc <= "0110001"; --jump 49
    else
     micropc <= "0111110"; --continue 62
    end if;
   when jg =>
    if(c = "01")then --A>B
     micropc <= "0110001"; --jump 49
    else
     micropc <= "0111110"; --continue 62
    end if;

   when j =>
    micropc <= "0110001"; --49

   when input =>
    micropc <= "1000011"; --67

   when output =>
    micropc <= "1000111"; --71

   when hlt =>
    micropc <= "1001010"; --74

   when others =>
    micropc <= "ZZZZZZZ";
  end case;
end process;
end beh;
```

C.3 Memória de Instruções

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity instrMem is
port(
 clk : in std_logic;
```

```
 addr : in std_logic_vector(8 downto 0);
 data : out std_logic_vector(24 downto 0)
);
end instrMem;

architecture beh of instrMem is
constant ADDR_WIDTH : integer := 9;
constant DATA_WIDTH : integer := 25;

type rom_type is array(0 to 2**ADDR_WIDTH-1) of
     std_logic_vector(DATA_WIDTH-1 downto 0);

signal instr_mem: rom_type;

begin

 instr_mem(0) <= "0000000000000000000000000";
 instr_mem(1) <= "0000000001100101100000010";
 instr_mem(2) <= "0000000000000010000110000";
 instr_mem(3) <= "0000000010000100000001010";
 instr_mem(4) <= "0000000000000000001001011";
 instr_mem(5) <= "0000000000001010001001010";
 instr_mem(6) <= "0000000000000001011001011";
 instr_mem(7) <= "0000000000001010001101010";
 instr_mem(8) <= "0000000000000001100101011";
 instr_mem(9) <= "0000000000001010010001010";
 instr_mem(10) <= "0000000000000001110001011";
 instr_mem(11) <= "0000000000001010010101010";
 instr_mem(12) <= "0000000000000001111101011";
 instr_mem(13) <= "0000000000001010011001010";
 instr_mem(14) <= "0000000000000010001001011";
 instr_mem(15) <= "0000000000001010011101010";
 instr_mem(16) <= "0000000000000010010101011";
 instr_mem(17) <= "0000000000001010100001010";
 instr_mem(18) <= "0000000000000010100001011";
 instr_mem(19) <= "0000000000001010100101010";
 instr_mem(20) <= "0000000000000010101101011";
 instr_mem(21) <= "0000000000000000001010011";
 instr_mem(22) <= "1111111100000010101001001";
```

```
instr_mem(23) <= "00100011000000101011100000";
instr_mem(24) <= "00000000000000010110110011";
instr_mem(25) <= "00100010000000101010000000";
instr_mem(26) <= "00100100000000101011100000";
instr_mem(27) <= "00000000000000010110110011";
instr_mem(28) <= "00100011000000101010000000";
instr_mem(29) <= "00100101000000101011100000";
instr_mem(30) <= "00000000000000010110110011";
instr_mem(31) <= "00100100000000101010000000";
instr_mem(32) <= "00100110000000101011100000";
instr_mem(33) <= "00000000000000010110110011";
instr_mem(34) <= "00100101000000101010000000";
instr_mem(35) <= "00100111000000101011100000";
instr_mem(36) <= "00000000000000010110110011";
instr_mem(37) <= "00100110000000101010000000";
instr_mem(38) <= "00101000000000101011100000";
instr_mem(39) <= "00000000000000010110110011";
instr_mem(40) <= "00100111000000101010000000";
instr_mem(41) <= "00101001000000101011100000";
instr_mem(42) <= "00000000000000010110110011";
instr_mem(43) <= "00101000000000101010000000";
instr_mem(44) <= "11111111000000101011101001";
instr_mem(45) <= "00000000101100101100000011";
instr_mem(46) <= "00000000101100101101011110";
instr_mem(47) <= "00000000101101010000001010";
instr_mem(48) <= "00000000000001111110101011";
instr_mem(49) <= "00000001101100000011101110";
instr_mem(50) <= "00000010101100000010001110";
instr_mem(51) <= "00000011101100000010101110";
instr_mem(52) <= "00000100101100000011001110";
instr_mem(53) <= "00000101101100000011101110";
instr_mem(54) <= "00000110101100000100001110";
instr_mem(55) <= "00000111101100000100101110";
instr_mem(56) <= "00001000101100000101001110";
instr_mem(57) <= "00001001101100000101101110";
instr_mem(58) <= "00000110000011000011000101";
instr_mem(59) <= "00000111000100000011100101";
instr_mem(60) <= "00001000000101001000000101";
instr_mem(61) <= "00001001000011001001001101";
```

```
instr_mem(62) <= "00001010000100001010001010";
instr_mem(63) <= "00001011000101001011001010";
instr_mem(64) <= "00011010000011000011001010";
instr_mem(65) <= "00011011000100000100001010";
instr_mem(66) <= "00011100000101000101001010";
instr_mem(67) <= "00011111001010001100001100";
instr_mem(68) <= "00001011011110001101001100";
instr_mem(69) <= "00001101001100001100001010";
instr_mem(70) <= "00011111000100001101001100";
instr_mem(71) <= "00000101011110001110001100";
instr_mem(72) <= "00001110001101001101001010";
instr_mem(73) <= "00000101001010001110001100";
instr_mem(74) <= "00001011000100001111001100";
instr_mem(75) <= "00001111001110001110001010";
instr_mem(76) <= "00001000011110001111001100";
instr_mem(77) <= "00011111000111010000001100";
instr_mem(78) <= "00010000001111001111001010";
instr_mem(79) <= "00001011000111010000001100";
instr_mem(80) <= "00001000001010010001001100";
instr_mem(81) <= "00010001010000010000001010";
instr_mem(82) <= "00001100000110010001001100";
instr_mem(83) <= "00001111001001010010001100";
instr_mem(84) <= "00010000011101010011001100";
instr_mem(85) <= "00010010010001010001001000";
instr_mem(86) <= "00010011010001010001001000";
instr_mem(87) <= "00010001000001010001001110";
instr_mem(88) <= "00001100000011010010001100";
instr_mem(89) <= "00001101001001010011001100";
instr_mem(90) <= "00001110011101010100001100";
instr_mem(91) <= "00010011010010010010001010";
instr_mem(92) <= "00010100010010010010001010";
instr_mem(93) <= "00010001010010010011001100";
instr_mem(94) <= "00000000010011000000001101";
instr_mem(95) <= "00000000000000111100101100";
instr_mem(96) <= "00000101000111010010001100";
instr_mem(97) <= "00001000000100010100001100";
instr_mem(98) <= "00010100010010010010001010";
instr_mem(99) <= "00001101000110010100001100";
instr_mem(100) <= "00001111000011010101001100";
```

```
 instr_mem(101) <= "00011101010010010110000110";
 instr_mem(102) <= "00010101010100010100000100";
 instr_mem(103) <= "00010110010100010100000100";
 instr_mem(104) <= "00010001010100010100000110";
 instr_mem(105) <= "00000000010100000000001101";
 instr_mem(106) <= "00000000000000111100101100";
 instr_mem(107) <= "00010100010011010011000100";
 instr_mem(108) <= "00000000000001010011011101";
 instr_mem(109) <= "00000000000000111100101100";
 instr_mem(110) <= "00001110000110010100000110";
 instr_mem(111) <= "00010010001001010100100110";
 instr_mem(112) <= "00010000000011010110000110";
 instr_mem(113) <= "00010101010100010100000101";
 instr_mem(114) <= "00010110010100010100000100";
 instr_mem(115) <= "00010001010100010100000110";
 instr_mem(116) <= "00000000010100000000001101";
 instr_mem(117) <= "00000000000000111100101100";
 instr_mem(118) <= "00000000010100011000011101";
 instr_mem(119) <= "00000000000000111100110100";
 instr_mem(120) <= "00000000010100011000000010";
 instr_mem(121) <= "11111111101101101101010011";
 instr_mem(122) <= "00001001101100101100010011";
 instr_mem(123) <= "00000000000000010110101010";
 instr_mem(124) <= "00000000000000011000101100";
 instr_mem(125) <= "00000000000000000000010010";
 instr_mem(126) <= "00000000011001011000000010";
 instr_mem(127) <= "00000000000000000000110011";

 process(clk)
 begin
  if(rising_edge(clk))then
   data <= instr_mem(conv_integer(addr));
  end if;
 end process;
end beh;
```

C.4 Registrador

```
library ieee;
use ieee.std_logic_1164.all;

entity reg is
generic(N: integer := 32);
port(
 load,clk : in std_logic;
 e : in std_logic_vector(N-1 downto 0);
 s : out std_logic_vector(N-1 downto 0)
);
end reg;

architecture beh of reg is
begin
 process(load)
  begin
   if(rising_edge(clk))then
    if(load = '1')then
     s <= e;
    end if;
   end if;
 end process;
end beh;
```

C.5 Comunicação GridRT-FSL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity grid_fsl is
port
(
-- DO NOT EDIT BELOW THIS LINE ----------------------
-- Bus protocol ports, do not add or delete.
 FSL_Clk        : in    std_logic;
```

```
 FSL_Rst        : in    std_logic;
 FSL_S_Clk      : out   std_logic;
 FSL_S_Read     : out   std_logic;
 FSL_S_Data     : in    std_logic_vector(0 to 31);
 FSL_S_Control  : in    std_logic;
 FSL_S_Exists   : in    std_logic;
 FSL_M_Clk      : out   std_logic;
 FSL_M_Write    : out   std_logic;
 FSL_M_Data     : out   std_logic_vector(0 to 31);
 FSL_M_Control  : out   std_logic;
 FSL_M_Full     : in    std_logic
-- DO NOT EDIT ABOVE THIS LINE ----------------------
);

 attribute SIGIS : string;
 attribute SIGIS of FSL_Clk : signal is "Clk";
 attribute SIGIS of FSL_S_Clk : signal is "Clk";
 attribute SIGIS of FSL_M_Clk : signal is "Clk";

end grid_fsl;

architecture EXAMPLE of grid_fsl is

-- Total number of input data.
constant NUMBER_OF_INPUT_WORDS  : natural := 8;

-- Total number of output data
constant NUMBER_OF_OUTPUT_WORDS : natural := 8;

type STATE_TYPE is (Idle, Read_Inputs, Process_List, Compute, Write_Outputs);

signal state        : STATE_TYPE;

signal result          : std_logic_vector(0 to 31);

signal x0 : std_logic_vector(31 downto 0);
signal y0 : std_logic_vector(31 downto 0);
signal z0 : std_logic_vector(31 downto 0);
```

```
signal xd : std_logic_vector(31 downto 0);
signal yd : std_logic_vector(31 downto 0);
signal zd : std_logic_vector(31 downto 0);

signal sig_ray : std_logic_vector(32*6-1 downto 0);

signal list : std_logic_vector(0 to 31);
signal ctrl : std_logic_vector(0 to 31);

signal sig_wrall, sig_complete : std_logic;

signal sigRes0, sigRes1, sigRes2, sigRes3 : std_logic_vector(31 downto 0);
signal sigP0,sigP1,sigP2,sigP3 : std_logic_vector(2 downto 0);
signal p0,p1,p2,p3 : std_logic;
signal sigRdy0,sigRdy1,sigRdy2,sigRdy3 : std_logic;

-- Counters to store the number inputs read & outputs written
signal nr_of_reads  : natural range 0 to NUMBER_OF_INPUT_WORDS - 1;
signal nr_of_writes : natural range 0 to NUMBER_OF_OUTPUT_WORDS - 1;

component processor is
 generic(PID: natural := 0;
  WORD_SIZE : natural := 32;
  FRAC_SIZE : natural := 24;
  LSIZE : natural := 8;
  LDSIZE : natural := 4);
 port(
  clk, wr_all : in std_logic;
  shAddr : in std_logic_vector(12 downto 0);
  shdo : out std_logic_vector(WORD_SIZE-1 downto 0);
  rgu_data : in std_logic_vector(WORD_SIZE*6 - 1 downto 0);
  gtu_data : in std_logic_vector(31 downto 0);

  ant_sel ,pos_sel: out std_logic_vector(3 downto 0);

  interrupt1,interrupt2 : in std_logic;
  interrupt1_out,interrupt2_out : out std_logic;
  complete : out std_logic;
```

```
  portid : out std_logic_vector(2 downto 0);
  in_port : in std_logic_vector(2 downto 0);
  out_port : out std_logic_vector(2 downto 0);
  resultado : out std_logic_vector(WORD_SIZE-1 downto 0)
);
end component;

component grid2x2x1 is
 generic(WORD_SIZE : natural := 32; FRAC_SIZE : natural := 24);
 port(
  clk, wr_all : in std_logic;
  data_rgu : in std_logic_vector(WORD_SIZE*6 - 1 downto 0);
  data_gtu : in std_logic_vector(31 downto 0);
  p0,p1,p2,p3: in std_logic_vector(2 downto 0);
  rdy0,rdy1,rdy2,rdy3 : out std_logic;
  res0,res1,res2,res3 : out std_logic_vector(WORD_SIZE-1 downto 0)
);
end component;

type list_type is array(0 to 7) of integer range -7 to 7;
signal ready_seq : std_logic_vector(3 downto 0);
signal vList : list_type;

begin

grid: grid2x2x1 generic map (32,24) port map(
 clk => FSL_Clk,
 wr_all => sig_wrall,
 data_rgu => sig_ray,
 data_gtu => list,
 p0 => sigP0,
 p1 => sigP1,
 p2 => sigP2,
 p3 => sigP3,
 rdy0 => sigRdy0,
 rdy1 => sigRdy1,
 rdy2 => sigRdy2,
 rdy3 => sigRdy3,
 res0 => sigRes0,
```

```
 res1 => sigRes1,
 res2 => sigRes2,
 res3 => sigRes3
);


Complete_Status: process(sigRdy0,sigRdy1,sigRdy2,sigRdy3,FSL_Clk,sig_wrall) is
begin
 if(rising_edge(FSL_Clk))then
  if(FSL_Rst = '1' or sig_wrall = '1')then
   ready_seq <= "0000";
  else
   if(sigRdy0 = '1')then
    ready_seq(0) <= '1';
   end if;

   if(sigRdy1 = '1')then
    ready_seq(1) <= '1';
   end if;

   if(sigRdy2 = '1')then
    ready_seq(2) <= '1';
   end if;

   if(sigRdy3 = '1')then
    ready_seq(3) <= '1';
   end if;
  end if;
 end if;
end process Complete_Status;

FSL_S_Read  <= FSL_S_Exists
               when (state = Read_Inputs) else '0';

FSL_M_Write <= not FSL_M_Full
               when (state = Write_Outputs and sig_complete = '1') else '0';

FSL_M_Data <= result;
```

```
sig_ray <= zd & yd & xd & z0 & y0 & x0;

sig_wrall <= '1' when state = Compute else '0';
sig_complete <= '1' when ((ready_seq(0) = p0) and (ready_seq(1) = p1) and
                          (ready_seq(2) = p2) and (ready_seq(3) = p3) and
                          (state = Write_Outputs)) else '0';

sigP0 <= "001" when (state = Write_Outputs and p0 = '1') else "000";
sigP1 <= "001" when (state = Write_Outputs and p1 = '1') else "000";
sigP2 <= "001" when (state = Write_Outputs and p2 = '1') else "000";
sigP3 <= "001" when (state = Write_Outputs and p3 = '1') else "000";

The_SW_accelerator : process (FSL_Clk) is

begin  -- process The_SW_accelerator
 if FSL_Clk'event and FSL_Clk = '1' then     -- Rising clock edge
  if FSL_Rst = '1' then               -- Synchronous reset (active high)
  -- CAUTION: make sure your reset polarity is consistent with the
  -- system reset polarity
  state        <= Idle;
  --result          <= (others => '0');
  else
   case state is
   when Idle =>
    if (FSL_S_Exists = '1') then
     state       <= Read_Inputs;
     --result      <= (others => '0');
     nr_of_reads <= NUMBER_OF_INPUT_WORDS - 1;
     nr_of_writes <= NUMBER_OF_OUTPUT_WORDS - 1;

     p0 <= '0';
     p1 <= '0';
     p2 <= '0';
     p3 <= '0';

     vList(0) <= 0;
     vList(1) <= 0;
     vList(2) <= 0;
     vList(3) <= 0;
```

```
  vList(4) <= 0;
  vList(5) <= 0;
  vList(6) <= 0;
  vList(7) <= 0;

 end if;

when Read_Inputs =>
 if (FSL_S_Exists = '1') then
  if(nr_of_reads = 7)then
   x0 <= std_logic_vector(unsigned(FSL_S_Data));
   nr_of_reads <= nr_of_reads - 1;
   elsif(nr_of_reads = 6)then
   y0 <= std_logic_vector(unsigned(FSL_S_Data));
   nr_of_reads <= nr_of_reads - 1;
   elsif(nr_of_reads = 5)then
   z0 <= std_logic_vector(unsigned(FSL_S_Data));
   nr_of_reads <= nr_of_reads - 1;
   elsif(nr_of_reads = 4)then
   xd <= std_logic_vector(unsigned(FSL_S_Data));
   nr_of_reads <= nr_of_reads - 1;
   elsif(nr_of_reads = 3)then
   yd <= std_logic_vector(unsigned(FSL_S_Data));
   nr_of_reads <= nr_of_reads - 1;
   elsif(nr_of_reads = 2)then
   zd <= std_logic_vector(unsigned(FSL_S_Data));
   nr_of_reads <= nr_of_reads - 1;
   elsif(nr_of_reads = 1)then
   list <= std_logic_vector(unsigned(FSL_S_Data));

   vList(0) <= to_integer(signed(FSL_S_Data(0 to 3)));
   vList(1) <= to_integer(signed(FSL_S_Data(4 to 7)));
   vList(2) <= to_integer(signed(FSL_S_Data(8 to 11)));
   vList(3) <= to_integer(signed(FSL_S_Data(12 to 15)));
   vList(4) <= to_integer(signed(FSL_S_Data(16 to 19)));
   vList(5) <= to_integer(signed(FSL_S_Data(20 to 23)));
   vList(6) <= to_integer(signed(FSL_S_Data(24 to 27)));
   vList(7) <= to_integer(signed(FSL_S_Data(28 to 31)));
```

```
   nr_of_reads <= nr_of_reads - 1;
  else
   ctrl <= std_logic_vector(unsigned(FSL_S_Data));
   state <= Process_List;
  end if;
 end if;

when Process_List =>
 for i in 0 to 7 loop
  if(vList(i) = 0)then
   p0 <= '1';
  elsif(vList(i) = 1)then
   p1 <= '1';
  elsif(vList(i) = 2)then
   p2 <= '1';
  elsif(vList(i) = 3)then
   p3 <= '1';
  end if;
 end loop;

 state <= Compute;

when Compute => --wr_all = 1
 state <= Write_Outputs;

when Write_Outputs =>
 if(FSL_M_Full = '0' and sig_complete = '1')then
  if(nr_of_writes = 7)then
   result <= sigRes0;
   nr_of_writes <= nr_of_writes - 1;
  elsif(nr_of_writes = 6)then
   result <= sigRes1;
   nr_of_writes <= nr_of_writes - 1;
  elsif(nr_of_writes = 5)then
   result <= sigRes2;
   nr_of_writes <= nr_of_writes - 1;
  elsif(nr_of_writes = 4)then
   result <= sigRes3;
   nr_of_writes <= nr_of_writes - 1;
```

```
      elsif(nr_of_writes = 3)then
       result <= sigRes0;
       nr_of_writes <= nr_of_writes - 1;
      elsif(nr_of_writes = 2)then
       result <= sigRes0;
       nr_of_writes <= nr_of_writes - 1;
      elsif(nr_of_writes = 1)then
       result <= sigRes0;
       nr_of_writes <= nr_of_writes - 1;
      else
       result <= sigRes0;
       state <= Idle;
      end if;
     end if;

    end case;
   end if;
 end if;
end process The_SW_accelerator;
end architecture EXAMPLE;
```

Apêndice D

Algoritmo de traçado de raios no MicroBlaze

```
// Located in: microblaze_0/include/xparameters.h
#include "xparameters.h"
#include "stdio.h"
#include "math.h"
#include "xutil.h"
#include "grid_fsl.h"
#include "xtmrctr.h"

// Function that performs initialization of the timer.
// Resets timer to 0
// Starts timer

void Start_Timer() {
  XTmrCtr_mSetLoadReg(XPAR_XPS_TIMER_0_BASEADDR,
      XPAR_XPS_TIMER_0_DEVICE_ID, 0);
  XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,
      XPAR_XPS_TIMER_0_DEVICE_ID, XTC_CSR_LOAD_MASK);
  XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,
      XPAR_XPS_TIMER_0_DEVICE_ID, 0x00);
  XTmrCtr_mEnable(XPAR_XPS_TIMER_0_BASEADDR,
      XPAR_XPS_TIMER_0_DEVICE_ID);
}

// This function stops timer and returns final value of the timer.
unsigned int Stop_Timer() {
  XTmrCtr_mDisable(XPAR_XPS_TIMER_0_BASEADDR,
```

```
      XPAR_XPS_TIMER_0_DEVICE_ID);
  return XTimerCtr_mReadReg(XPAR_XPS_TIMER_0_BASEADDR,
      XPAR_XPS_TIMER_0_DEVICE_ID, XTC_TCR_OFFSET);
}

unsigned int Get_Timer() {
  return XTimerCtr_mReadReg(XPAR_XPS_TIMER_0_BASEADDR,
      XPAR_XPS_TIMER_0_DEVICE_ID, XTC_TCR_OFFSET);
}

//====================================================

// --- estruturas ---

struct po {
  float x;
  float y;
  float z;
}typedef point;

struct vi {
  int width;
  int height;
}typedef viewport;

struct ve {
  float x;
  float y;
  float z;
}typedef vector;

// --- funÁ?es ---
float dist(point * o, point * d);
void normalize(vector *v);
float dotProduct(vector *a, vector *b);
void crossProduct(vector *a, vector *b, vector *r);

void traverseGrid(point *o, point *d, point *p0, point *p1,
    int *list, int listsize);
```

```
float maximo(float a, float b);
float minimo(float a, float b);
int inside(point *p, point *p0, point *p1);
float clamp(float x, float min, float max);
void printList(int * list, int size);
void eraseList(int * list, int size);
unsigned int reduceList(int * list, int size);

void teste();

// --- var global ---
int nx = 2;
int ny = 2;
int nz = 1;

#define TRUE (0==0)
#define FALSE (0==1)
#define LISTSIZE 8

// --- main ---
int main(void) {
  union ufloat {
    float f;
    unsigned u;
  } x0, y0, z0, xd, yd, zd, ux, uy, uz, vx, vy, vz, wx, wy,
      wz, di;

  // --- declaraÁ?es ---
  unsigned int input[8];
  unsigned int output[8];
  unsigned int start_timer_value, end_timer_value;
  int i = 0;
  int j = 0;
  float xv = 0.0f;
  float yv = 0.0f;
  float distance = 0.0f;
  int list[LISTSIZE] = { 0xf, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf,
      0xf };
```

```
point lookat;
point eye;
point dest;
point p0;
point p1;
vector upv;
vector u;
vector v;
vector w;
viewport view;

// --- atribuiÁoes + algoritmo ---

// --- GRID SIZE ---
//(-93.14661,33.620388,-56.644012) bunny lowest res
p0.x = -93.14661f;
p0.y = 33.620388f;
p0.z = -56.644012f;

//(58.159115,181.89702,57.80081) bunny lowest res
p1.x = 58.159115f;
p1.y = 181.89702f;
p1.z = 57.80081f;

// --- camera start---
view.width = 320;
view.height = 240;

lookat.x = -26.198248f;
lookat.y = 93.17701f;
lookat.z = 7.3524833f;

eye.x = 0.0f;
eye.y = 200.0f;
eye.z = 200.0f;

distance = dist(&eye, &lookat);

upv.x = 0.0f;
```

```
upv.y = 1.0f;
upv.z = 0.0f;

w.x = eye.x - lookat.x;
w.y = eye.y - lookat.y;
w.z = eye.z - lookat.z;
normalize(&w);

crossProduct(&w, &upv, &u);
normalize(&u);

crossProduct(&u, &w, &v);
normalize(&v);

//--- camera end ---
print("-- Ray Tracer --\r\n");

x0.f = eye.x;
y0.f = eye.y;
z0.f = eye.z;

input[0] = x0.u; //eye.x 0
input[1] = y0.u; //eye.y 200
input[2] = z0.u; //eye.z   200
input[7] = 0x00000000; //ctrls (not used)

ux.f = u.x;
uy.f = u.y;
uz.f = u.z;

vx.f = v.x;
vy.f = v.y;
vz.f = v.z;

wx.f = w.x;
wy.f = w.y;
wz.f = w.z;

di.f = distance;
```

```
xil_printf("u = 0x%08x 0x%08x 0x%08x \r\n", ux.u, uy.u,
    uz.u);
xil_printf("v = 0x%08x 0x%08x 0x%08x \r\n", vx.u, vy.u,
    vz.u);
xil_printf("w = 0x%08x 0x%08x 0x%08x \r\n", wx.u, wy.u,
    wz.u);
xil_printf("distance = 0x%08x \r\n", di.u);

Start_Timer();
start_timer_value = Get_Timer(); // record starting time

for (i = 0; i < view.width; i++) {
  xil_printf(">>linha : %d \r\n ", i);
  for (j = 0; j < view.height; j++) {
    // --- 125 ciclos daqui ---
    xv = i - (view.width / 2);
    yv = (view.height / 2) - j;

    dest.x = u.x * xv + v.x * yv - w.x * distance;
    dest.y = u.y * xv + v.y * yv - w.y * distance;
    dest.z = u.z * xv + v.z * yv - w.z * distance;

    xd.f = (dest.x - eye.x); //DIR = (Dest - orig)
    yd.f = (dest.y - eye.y); //DIR
    zd.f = (dest.z - eye.z); //DIR

    // --- 125 ciclos ate aqui ---

    input[3] = xd.u;
    input[4] = yd.u;
    input[5] = zd.u;

    traverseGrid(&eye, &dest, &p0, &p1, list, LISTSIZE);
    input[6] = reduceList(list, LISTSIZE);

    if (input[6] != 0xFFFFFFFF) {
      grid_fsl(XPAR_FSL_GRID_FSL_0_INPUT_SLOT_ID,
          XPAR_FSL_GRID_FSL_0_OUTPUT_SLOT_ID, input,
```

```
            output);
      }
      xil_printf(" Results[1-4] = 0x%08x 0x%08x 0x%08x 0x%08x \r\n",
                          output[1],output[2],output[3],output[4]);

      eraseList(list, LISTSIZE);
    }
  }

  end_timer_value = Stop_Timer();

  //xil_printf(" Tempo inicial: %d \r\n",start_timer_value);
  //xil_printf(" Tempo final: %d \r\n",end_timer_value);
  xil_printf(" Cycles : %d \r\n", (end_timer_value
      - start_timer_value));

  print("-- Finished --\r\n");
  return 0;
}

float dist(point *o, point *d) {
  float x = d->x - o->x;
  float y = d->y - o->y;
  float z = d->z - o->z;

  return sqrt(x * x + y * y + z * z);
}

void normalize(vector *v) {
  float length = v->x * v->x + v->y * v->y + v->z * v->z;
  length = sqrt(length);

  *(&v->x) = v->x / length;
  *(&v->y) = v->y / length;
  *(&v->z) = v->z / length;
}

float dotProduct(vector *a, vector *b) {
  return (a->x * b->x + a->y * b->y + a->z * b->z);
```

```
}

void crossProduct(vector *a, vector *b, vector *r) {
  *(&r->x) = a->y * b->z - a->z * b->y;
  *(&r->y) = a->z * b->x - a->x * b->z;
  *(&r->z) = a->x * b->y - a->y * b->x;
}

float maximo(float a, float b) {
  if (a > b)
    return a;
  else
    return b;
}

float minimo(float a, float b) {
  if (a < b)
    return a;
  else
    return b;
}

int inside(point *p, point *p0, point *p1) {
  if (p->x > p0->x && p->x < p1->x) {
    if (p->y > p0->y && p->y < p1->y) {
      if (p->z > p0->z && p->z < p1->z) {
        return TRUE;
      }
    }
  }
  return FALSE;
}

float clamp(float x, float min, float max) {
  return (x < min ? min : (x > max ? max : x));
}

void printList(int * list, int size) {
  int i = 0;
```

```
  xil_printf(" list: \r\n");
  for (i = 0; i < size; i++) {
    xil_printf(" %d ", list[i]);
  }
  xil_printf(" \r\n ");
}

void eraseList(int * list, int size) {
  int i = 0;
  for (i = 0; i < size; i++) {
    list[i] = 0xf;
  }
}

unsigned int reduceList(int * list, int size) {
  unsigned int result = list[0];
  unsigned int i;

  for (i = 1; i < size; i++) {
    result = (result << 4) + list[i];
  }

  return result;
}

void traverseGrid(point *o, point *d, point *p0, point *p1,
    int *list, int listsize) {
  vector dir;

  dir.x = d->x - o->x;
  dir.y = d->y - o->y;
  dir.z = d->z - o->z;

  float a = 1.0f / dir.x;
  float b = 1.0f / dir.y;
  float c = 1.0f / dir.z;

  float tx_min, ty_min, tz_min;
  float tx_max, ty_max, tz_max;
```

```
float t0, t1;

int ix, iy, iz;

if (a >= 0) {
  tx_min = (p0->x - o->x) * a;
  tx_max = (p1->x - o->x) * a;
} else {
  tx_min = (p1->x - o->x) * a;
  tx_max = (p0->x - o->x) * a;
}

if (b >= 0) {
  ty_min = (p0->y - o->y) * b;
  ty_max = (p1->y - o->y) * b;
} else {
  ty_min = (p1->y - o->y) * b;
  ty_max = (p0->y - o->y) * b;
}

if (c >= 0) {
  tz_min = (p0->z - o->z) * c;
  tz_max = (p1->z - o->z) * c;
} else {
  tz_min = (p1->z - o->z) * c;
  tz_max = (p0->z - o->z) * c;
}

t0 = maximo(maximo(tx_min, ty_min), tz_min);
t1 = minimo(minimo(tx_max, ty_max), tz_max);

if (t0 < t1) {
  point p;

  float dtx = (tx_max - tx_min) / nx;
  float dty = (ty_max - ty_min) / ny;
  float dtz = (tz_max - tz_min) / nz;

  float tx_next, ty_next, tz_next;
```

```
int ix_step, iy_step, iz_step;
int ix_stop, iy_stop, iz_stop;

p.x = o->x + t0 * dir.x;
p.y = o->y + t0 * dir.y;
p.z = o->z + t0 * dir.z;

if (inside(&p, p0, p1)) {
  ix = (int) floor(clamp((o->x - p0->x) * nx / (p1->x
      - p0->x), 0, nx - 1));
  iy = (int) floor(clamp((o->y - p0->y) * ny / (p1->y
      - p0->y), 0, ny - 1));
  iz = (int) floor(clamp((o->z - p0->z) * nz / (p1->z
      - p0->z), 0, nz - 1));
} else {
  ix = (int) floor(clamp((p.x - p0->x) * nx / (p1->x
      - p0->x), 0, nx - 1));
  iy = (int) floor(clamp((p.y - p0->y) * ny / (p1->y
      - p0->y), 0, ny - 1));
  iz = (int) floor(clamp((p.z - p0->z) * nz / (p1->z
      - p0->z), 0, nz - 1));
}

if (dir.x > 0) {
  tx_next = tx_min + (ix + 1) * dtx;
  ix_step = +1;
  ix_stop = nx;
} else {
  tx_next = tx_min + (nx - ix) * dtx;
  ix_step = -1;
  ix_stop = -1;
}

if (dir.y > 0) {
  ty_next = ty_min + (iy + 1) * dty;
  iy_step = +1;
  iy_stop = ny;
} else {
  ty_next = ty_min + (ny - iy) * dty;
```

```
  iy_step = -1;
  iy_stop = -1;
}

if (dir.z > 0) {
  tz_next = tz_min + (iz + 1) * dtz;
  iz_step = +1;
  iz_stop = nz;
} else {
  tz_next = tz_min + (nz - iz) * dtz;
  iz_step = +1;
  iz_stop = nz;
}

// --- GRID TRAVERSAL ---
int cont = LISTSIZE - 1;
while (TRUE) {
  int index = ix + nx * iy + nx * ny * iz;

  if (cont >= 0)
    list[cont] = index;

  if (tx_next < ty_next && tx_next < tz_next) {
    cont--;
    tx_next += dtx;
    ix += ix_step;
    if (ix == ix_stop)
      return;
  } else if (ty_next < tz_next) {
    cont--;
    ty_next += dty;
    iy += iy_step;
    if (iy == iy_stop)
      return;
  } else {
    cont--;
    tz_next += dtz;
    iz += iz_step;
    if (iz == iz_stop)
```

```
            return;
        }
      }
    }
  }
```