4
Processamento Paralelo

Como apresentado no Capítulo 3, o algoritmo de mapeamento de textura com relevo é divido em cinco passos: reinicialização de recursos, configuração da câmera, cálculo de *lookup tables*, pré *warp* e mapeamento de textura. Tal algoritmo recebe como dados de entrada a posição corrente p do observador, um quadrilátero q e uma textura com relevo $\{i_s, K_s\}$. O passo de pré *warp* é responsável por produzir uma imagem de saída i_t que represente uma visualização parcial do mapeamento de $\{i_s, K_s\}$ sobre q visto de p. Desse modo, uma vez que a imagem i_t tenha sido gerada seu conteúdo pode ser transferido para a memória de textura da placa de vídeo e, finalmente, mapeada sobre o quadrilátero q para que uma visualização correta seja obtida.

Como um dos objetivos deste trabalho é atestar a possibilidade de uso da técnica de mapeamento de textura com relevo em aplicações com requerimento de tempo real, é necessário que tal mapeamento consuma o menor tempo de processamento possível.

Neste capítulo, são tratadas as questões relacionadas com a otimização do processamento empregado durante a amostragem e reconstrução de imagens a partir de texturas com relevo visando diminuir o tempo de execução requerido por tal procedimento. Mais especificamente, são apresentadas duas abordagens paralelas onde o foco do paralelismo se concentra nos quatro primeiros passos do algoritmo de mapeamento de textura com relevo. Para isto, inicialmente é apresentado um breve resumo de conceitos relacionados à computação paralela.

4.1
Conceitos Relacionados com Paralelismo

Segundo Carissimi *et al.* [7], um *processo* é definido como um programa em execução. Desta forma, o algoritmo de mapeamento de textura com relevo, quando executado, passa a ser um processo. Todo processo possui um *fluxo de execução*. Por sua vez, uma *thread* nada mais é do que um fluxo de execução. Na maior parte das vezes, cada processo é formado por um conjunto de recursos mais uma única *thread*.

4.1.1
Categorias de Sistemas

Segundo Andrews [4] todo programa paralelo se encaixa numa das três categorias: *Sistema Multi-Threaded*, *Sistema Distribuído* e *Computação Paralela*.

O termo *Sistema Multi-Threaded* geralmente significa que um programa contém mais processos do que processadores para executar suas *threads*. Tais sistemas têm como objetivo gerenciar múltiplas tarefas independentes.

Quando se trata de um *Sistema Distribuído* componentes executam em máquinas conectadas por uma rede de comunicação local ou global. Conseqüentemente, os processos se comunicam por troca de mensagens.

Em *Computação Paralela* os programas são executados em processadores paralelos para alcançar altas taxas de desempenho e, geralmente, existem tantos processos quanto processadores, tentando resolver um problema de forma mais rápida ou um problema maior na mesma quantidade de tempo.

4.1.2
Modelos de Programação

Segundo Andrews [4], um programa paralelo pode ser caracterizado pelo modelo de programação ao qual pertence: *variáveis compartilhadas*, *troca de mensagem*, *coordenação* e *dados paralelos*.

No modelo de *variáveis compartilhadas* os processos envolvidos na comunicação trocam informações por meio de uma memória compartilhada. Sendo assim, a passagem de dados ocorre quando um processo escreve em uma variável que será lida por outro processo.

Quando os processos envolvidos na comunicação pertencem a máquinas diferentes (sistema distribuído) o compartilhamento de memória deixa de existir. Neste caso, a comunicação ocorre por *troca de mensagem*.

Um modelo é dito ser de *coordenação* quando existe um espaço de dados compartilhados sobre o qual operam primitivas como mensagens. Geralmente, tais modelos se baseiam em estruturas de dados que são conceitualmente compartilhadas mas que poderiam ser distribuídas fisicamente.

Finalmente, no modelo de *dados paralelos* cada processo executa as mesmas operações sobre diferentes partes do dado compartilhado. Na literatura, costuma-se usar o acrônimo SPMD (do inglês, *Single Program Multiple Data*) para caracterizar este tipo de modelo.

4.2 Hyper-Threading

Hyper-Threading é uma nova tecnologia desenvolvida pela Intel que permite que aplicações *Multi-Threaded* executem suas *threads* em paralelo [23]. Mais especificamente, tal tecnologia faz com que um único processador físico seja visto por uma aplicação como dois processadores lógicos. Isto é alcançado duplicando-se a arquitetura de estado em cada processador lógico, sendo que estes processadores compartilham um único conjunto de recursos físicos destinados à execução dos mesmos. A título de exemplo, a Figura 4.1 ilustra a arquitetura de dois processadores físicos: um sem a tecnologia de *Hyper-Threading* (a) e outro com tal tecnologia (b).

Do ponto de vista de um *software*, isto significa que sistemas operacionais e programas de usuários podem atribuir processos ou *threads* para os processadores lógicos como se estes fossem processadores físicos convencionais de um sistema multi-processador. Do ponto de vista de uma arquitetura, isto significa que as instruções nos processadores lógicos serão avaliadas simultaneamente sobre os recursos de execução compartilhados.

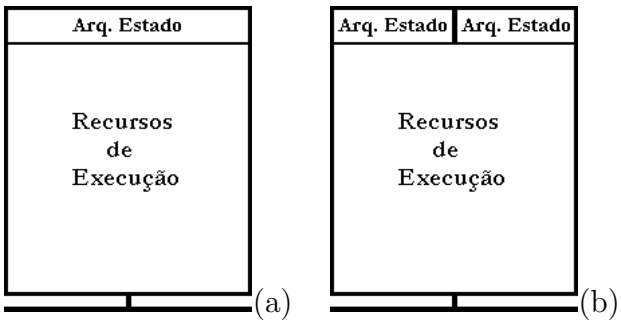

Figura 4.1: Arquitetura de processadores. A Figura (a) representa um processador sem a tecnologia de *Hyper-Threading* enquanto que a Figura (b) representa um processador com a tecnologia de *Hyper-Threading*.

4.3 Otimização do Processo de Amostragem e Reconstrução

No contexto de aplicações gráficas em tempo real, costuma-se utilizar uma divisão do *pipeline* de renderização em três estágios conceituais: *aplicação*, *geometria* e *rasterização* [2]. Atualmente, os dois últimos estágios são implementados em *hardware* pela placa gráfica, enquanto que o primeiro estágio é implementado na CPU.

Esta mesma divisão pode ser utilizada para representar todo o processamento envolvido durante o mapeamento de textura com relevo. O diagrama da Figura 4.2 ilustra os estágios conceituais de acordo com a abordagem convencional descrita no Capítulo 3.

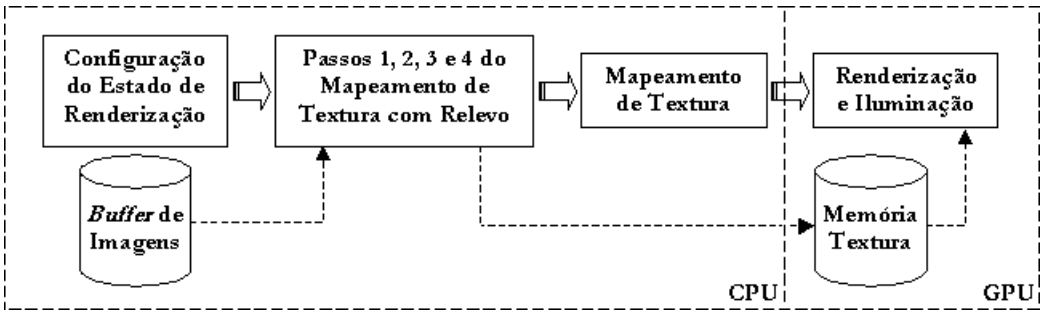

Figura 4.2: Diagrama do processo convencional envolvido na amostragem e reconstrução de imagens a partir de texturas com relevo.

Neste diagrama, o estágio da aplicação é executado de forma seqüencial e compreende todos os passos realizados na CPU. Os estágios de geometria e rasterização são implicitamente representados pelo passo de **renderização e iluminação**, realizado na GPU.

Por definição, a velocidade de um *pipeline* é determinada pelo seu estágio mais lento, independente de quão rápido são os outros estágios. Geralmente, este estágio mais lento é denominado de *gargalo*.

Segundo Akenine-Möller & Haines [2], o primeiro passo na otimização de um *pipeline* é localizar o gargalo. Tal localização pode ser efetuada através da realização de um conjunto de testes propostos em [2].

Analisando-se os algoritmos descritos no Capítulo 3, tem-se o sentimento de que o gargalo do processo de amostragem e reconstrução a partir de texturas com relevo é o estágio da aplicação, uma vez que todos os *texels* do mapa de normal e do mapa de cor são processados uma vez por quadro (do inglês, *frame*). Dessa maneira, decidiu-se num primeiro instante realizar os testes referentes a este estágio.

Uma maneira de verificar se o estágio da aplicação limita a velocidade de renderização é enviar dados através do *pipeline* de forma que os outros estágios realizem pouco ou nenhum trabalho. Em *OpenGL* isto pode ser alcançado substituindo-se todas as chamadas de `glVertex3f` e `glNormal3f` por chamadas de `glColor3f`. Dessa forma, o trabalho de envio de dados da CPU permanece inalterado enquanto que o trabalho de envio e recebimento de dados nos estágios da geometria e rasterização é altamente reduzido. O estágio da geometria não realiza operações de *clipping*, mapeamento de coordenadas da tela, nem quaisquer transformações sobre vértices e normais. O estágio da rasterização não recebe quaisquer vértices do estágio da geometria e, portanto, não recebe primitivas para renderizar. Logo, se o desempenho não melhorar, então o gargalo é o estágio da aplicação.

Através da realização deste teste para várias texturas de entrada, constatou-se que o estágio da aplicação é realmente o gargalo. Tais testes são apresentados com mais detalhes no Capítulo 5.

Diante das observações anteriores, decidiu-se implementar duas abordagens paralelas para o processo envolvido na amostragem e reconstrução de imagens a partir de texturas com relevo. No que se segue, é apresentada para cada abordagem a motivação que permitiu sua implementação, a caracterização quanto ao tipo de sistema e ao tipo de modelo de programação, bem como a descrição da metodologia adotada para sua realização.

4.3.1

Abordagem Paralela

Com o intuito de otimizar o processamento empregado na CPU, criou-se uma *thread* de CPU para executar somente os quatro primeiros passos do algoritmo de mapeamento de textura com relevo. Com o auxílio da tecnologia de *Hyper-Threading* tal *thread* pode ser executada em paralelo com o processo convencional da CPU, permitindo um ganho considerável de tempo. O diagrama da Figura 4.3 ilustra o novo procedimento.

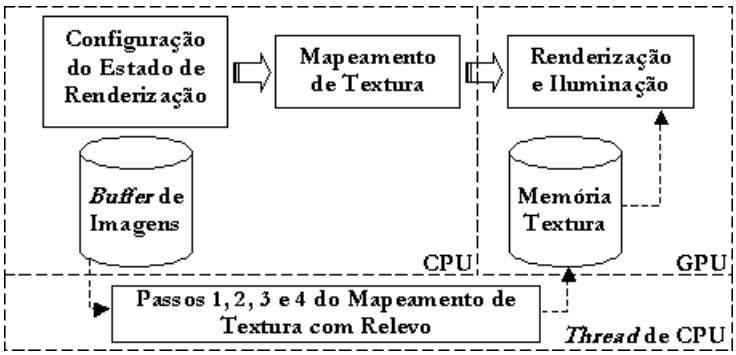

Figura 4.3: Diagrama do processo paralelo envolvido na amostragem e reconstrução de imagens a partir de texturas com relevo.

Para garantir que o resultado final seja correto, além de criar a *thread* de CPU é necessário sincronizá-la com o restante dos processos em andamento. Tal sincronização é representada pela máquina de estados ilustrada na Figura 4.4.

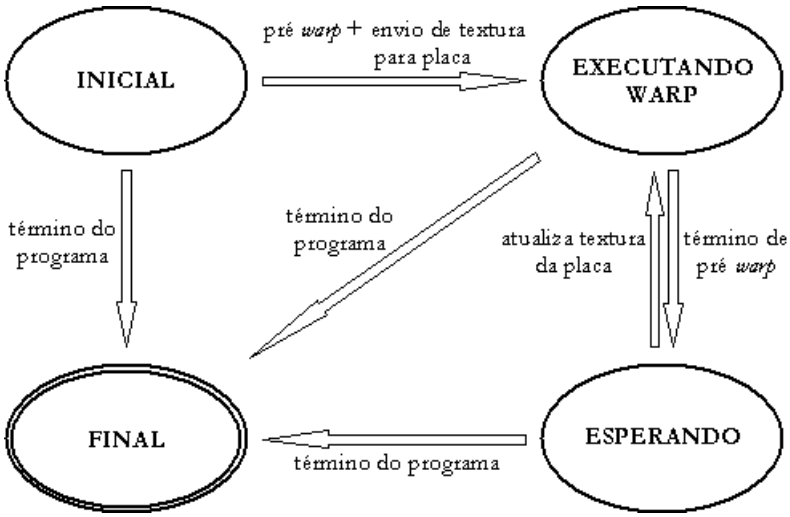

Figura 4.4: Máquina de estados da *thread* de CPU.

No momento em que a *thread* é criada seu estado corrente é o estado **inicial**. A transição do estado **inicial** para o estado **executando warp** é feita através da primeira realização da operação de *warp* seguida do envio da imagem resultante desta operação para a memória de textura da placa de vídeo. No restante da execução, o autômato permanece num laço entre os estados **executando warp** e **esperando**, salvo em situações de término de programa em que a *thread* de CPU é levada para o estado **final**. No estado **executando warp** o processo realiza a operação de pré *warping* que, uma vez terminada, causa a transição para o estado **esperando**. Neste estado, a nova imagem resultante é enviada para a memória de textura, desta forma, atualizando o seu conteúdo. Uma vez que a memória de textura é atualizada ocorre a transição para o estado **executando warp** indicando que uma nova operação de *warping* pode ser efetuada.

A *thread* de CPU é criada após a instanciação do quadrilátero sobre o qual a imagem resultante do *warping* é mapeada. Dessa maneira, o algoritmo de inicialização apresentado no Capítulo 3 é modificado para conter a operação que cria a *thread* de CPU. A nova versão do pseudo-código de inicialização é descrita pelo Algoritmo 19.


```
algoritmo inicializacao(quad, texRel)
    1 Alocação de memória;
    2 Inicialização de variáveis;
    3 quad.C ← quad.v3;
    4 quad.a ← (quad.v2 − quad.v3)/(texRel.mCor.largura);
    5 quad.b ← (quad.v0 − quad.v3)/(texRel.mCor.altura);
    6 quad.f ← (quad.a × quad.b)/‖quad.a × quad.b‖;
    7 Cria a thread de CPU threadCPU com prioridade normal;
fim
```

Algoritmo 19: Pseudo-código para instanciação do quadrilátero `quad` modificado para conter a operação que cria a *thread* de CPU.

Assumindo-se que a máquina, utilizada para executar o algoritmo objeto deste trabalho, possui a tecnologia de *Hyper-Threading*, um processador pode ser designado a permanecer exclusivamente dedicado à *thread* de CPU sem que o desempenho do *pipeline* gráfico seja prejudicado. Portanto, a *thread* pode ser criada com a prioridade normal (linha **7** do Algoritmo 19).

A transição do estado **inicial** para o estado **executando warp** ocorre durante a execução da rotina principal de desenho realizada uma vez por quadro. O Algoritmo 20 apresenta um pseudo-código para esta rotina.

Na linha **1** é configurado o estado de renderização da API *OpenGL*.

```
algoritmo draw(quad, texRel, cam)
 1  Configura o estado de renderização da API OpenGL;
 2  Ativa programa de vértice;
 3  Ativa programa de fragmento;
 4  Se é a primeira execução de draw então
 5     reinicializacao(quad,texRel);
 6     configCamera(quad,cam);
 7     inicLookupTables(quad);
 8     amostragem1D(quad,texRel);
 9     enviaTextura(quad);
10     estadoThread ← EXECUTANDO_WARPING;
11  senão
12     Se estadoThread = ESPERANDO então
13        atualizaTextura(quad);
14        estadoThread ← EXECUTANDO_WARPING;
15     fim_se
16  fim_se
17  desenhaQuad(quad);
18  Desativa programa de fragmento;
19  Desativa programa de vértice;
fim
```

Algoritmo 20: Pseudo-código para a rotina principal de desenho executada uma vez por quadro.

Tal configuração consiste na habilitação de operações como *blending*, iluminação, *culling*, testes de profundidade, entre outras. Nas linhas **2** e **3** os programas de vértice e fragmento para o cálculo de iluminação por *pixel* são ativados, respectivamente. Na primeira vez que a rotina `draw` é executada, os quatro primeiros passos do algoritmo de mapeamento de textura com relevo (linhas **5** a **8**) são realizados e, em seguida, a textura resultante do *warping* é enviada para a placa de vídeo através da rotina `enviaTextura`. Esta última rotina é implementada via a função `glTexImage2D` disponibilizada através da API *OpenGL* (veja a função `mapeiaTextura` no Capítulo 3). Uma vez que a textura é enviada para a placa de vídeo, a transição do estado **inicial** para o estado **executando warping** pode ser efetuada (linha **10**). Nas circunstâncias em que a rotina `draw` é executada pela segunda vez em diante, necessita-se verificar se o estado corrente da *thread* de CPU é **esperando**. Em caso afirmativo, significa que a *thread* realizou uma operação de *warping* e, conseqüentemente, a textura residente na placa de vídeo pode ser atualizada através da rotina `atualizaTextura` (linha **13**). Neste caso, esta atualização é efetuada pela função `glTexSubImage2D` também disponibilizada pela API *OpenGL* (novamente, veja a função `mapeiaTextura` no Capítulo 3). Em seguida, a *thread* de CPU sofre uma transição do estado

esperando para o estado **executando warping** e uma nova operação de *warping* pode ser realizada. Finalmente, na linha **17**, o quadrilátero com a textura mapeada pode ser desenhado e, nas linhas **18** e **19**, os programas de fragmento e vértice são desativados, respectivamente. Note que durante toda a realização da rotina `draw`, a *thread* de CPU é executada em paralelo.

As transições de estado remanescentes são descritas pelo Algoritmo 21, onde a *thread* de CPU é implementada.

```
algoritmo threadCPU(quad, texRel, cam)
   1 Enquanto verdadeiro faça
   2    Se estadoThread = EXECUTANDO_WARPING então
   3       reinicializacao(quad,texRel);
   4       configCamera(quad,cam);
   5       inicLookupTables(quad);
   6       amostragem1D(quad,texRel);
   7       estadoThread ← ESPERANDO;
   8    senão se estadoThread = FINAL então
   9       retorna 0;
  10    fim_se
  11 fim_enquanto
fim
```

Algoritmo 21: Pseudo-código para a *thread* de CPU.

O laço da linha **1** é responsável por fazer com que a *thread* execute até que seu estado corrente seja modificado para o estado **final**. Durante a realização do laço, se o estado corrente é **executando warping**, os quatro primeiros passos do algoritmo de mapeamento de textura com relevo são executados (linhas **3** a **6**) e a transição para o estado **esperando** é realizada (linha **7**) indicando que uma operação de atualização de textura (linha **13** do Algoritmo 20) pode ser realizada. Para o caso em que o estado corrente da *thread* é o estado **final**, a execução da rotina `threadCPU` é finalizada.

Levando-se em conta o processamento envolvido na CPU e que dispõe-se da tecnologia de *Hyper-Threading*, esta nova abordagem pode ser caracterizada como pertencente à categoria de *computação paralela*, uma vez que dois processos são executados em paralelo por dois processadores: um para a *thread* de CPU e outro para o processo principal da CPU. Quanto ao modelo de programação, a nova abordagem pode ser caracterizada como pertencente ao modelo de *variáveis compartilhadas*, pois, a troca de informação entre ambos os processos ocorre por meio de um *buffer* de imagem que ora é atualizado pela *thread* de CPU, ora é enviado para a placa de vídeo pelo processo principal da CPU.

4.3.2
Abordagem Multi-Threaded

Além da abordagem anterior, o uso da tecnologia de *Hyper-Threading* permite que haja um processamento paralelo para diferentes partes dos dados de entrada. Mais especificamente, é possível dividir a textura de entrada $\{i_s, K_s\}$ em duas partes e executar, simultaneamente, os quatro primeiros passos do algoritmo de amostragem e reconstrução para cada uma destas partes. Conseqüentemente, um pós processamento é necessário para unir os dois resultados obtidos numa única textura resultante para que a mesma possa ser mapeada sobre o quadrilátero q. O diagrama da Figura 4.5 ilustra este procedimento.

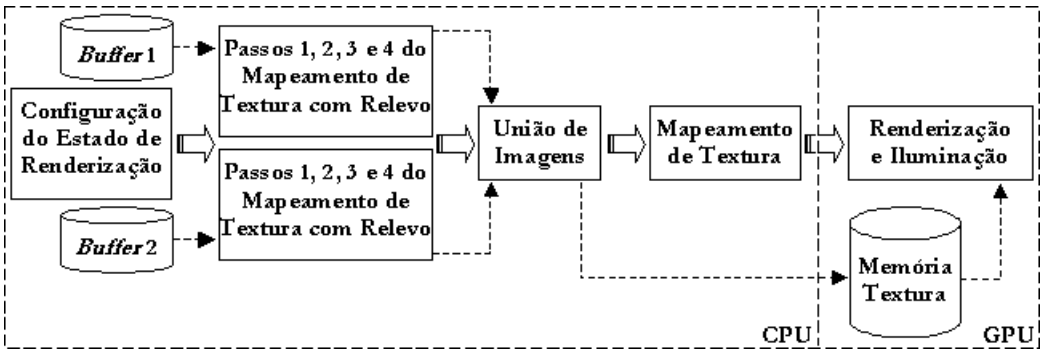

Figura 4.5: Diagrama do processo *multi-threaded* envolvido na amostragem e reconstrução de imagens a partir de texturas com relevo.

Inicialmente, é necessário dividir o *buffer* de imagens em duas partes. Vale notar que o termo *buffer de imagens* compreende o mapa de normal com profundidade juntamente com o mapa de cor utilizados como entrada para o algoritmo de mapeamento de textura com relevo. A divisão do *buffer* de imagens é descrita no Algoritmo 22.

Os laços das linhas **1** e **2** iteram, respectivamente, sobre as linhas e as colunas dos mapas de entrada. O comando **Se** da linha **3** realiza a divisão dos mapas em relação à coluna do meio, ou seja, a metade esquerda dos mapas de `texRel` é copiada para `texRel1` (linhas **4** e **5**), enquanto que a metade direita é copiada para `texRel2` (linhas **8** e **9**).

Repare que os mapas de textura de `texRel1` e `texRel2` possuem as mesmas dimensões que os mapas de `texRel`, ou seja, o número de *texels* dos mapas de `texRel1` e `texRel2` é o mesmo dos mapas de `texRel`, e não a metade. Isto procede devido aos comandos das linhas **6** e **10**. O primeiro comando é responsável por preencher a metade direita dos mapas

```
algoritmo divideBuffer(texRel)
 1 Para v ← 0; v < altura; v ← v + 1 faça
 2   Para u ← 0; u < largura; u ← u + 1 faça
 3     Se u < largura/2 então
 4       texRel1.mNormal[v, u] ← texRel.mNormal[v, u];
 5       texRel1.mCor[v, u] ← texRel.mCor[v, u];
 6       texRel2.mNormal[v, u] ← {0⃗, 255};
 7     senão
 8       texRel2.mNormal[v, u] ← texRel.mNormal[v, u];
 9       texRel2.mCor[v, u] ← texRel.mCor[v, u];
10       texRel1.mNormal[v, u] ← {0⃗, 255};
11     fim_se
12   fim_para
13 fim_para
fim
```

Algoritmo 22: Pseudo-código para dividir o *buffer* de imagens de entrada para o algoritmo de mapeamento de textura com relevo, onde $\{\vec{0}, 255\}$ representa um *texel* inválido.

de `texRel1` com *texels* inválidos enquanto que o segundo comando executa esta mesma operação para a metade esquerda dos mapas de `texRel2`. Veja a Figura 4.6.

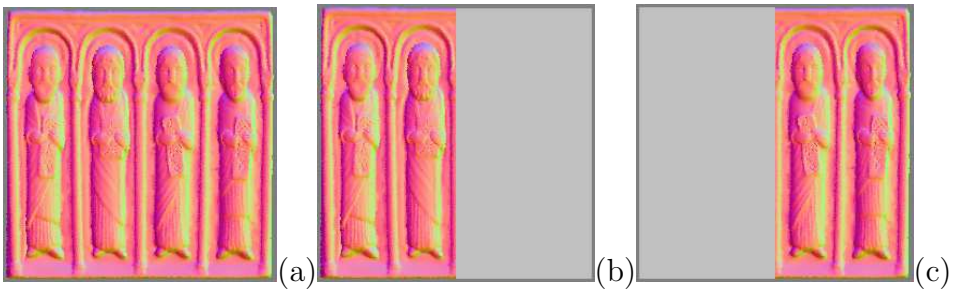

Figura 4.6: Ilustração do processo de divisão do mapa de normal com profundidade. A Figura (a) representa o mapa de normal original, a Figura (b) representa o mapa de normal contido em `texRel1` e a Figura (c) representa o mapa de normal contido em `texRel2`.

Isto é necessário porque durante o pré *warping* das texturas de `texRel1` e `texRel2`, alguns *texels* poderiam deslocar-se além dos limites do plano de suporte de tais texturas, se estas realmente tivessem a metade do tamanho da textura original (veja a Seção 3.7). Conseqüentemente, as imagens resultantes do pré *warping* compreenderiam uma visão incompleta da superfície representada fazendo com que o resultado da união de ambas também representasse uma visão incompleta. Veja a Figura 4.7.

No entanto, somente manter o tamanho original das texturas não

Figura 4.7: Visão incompleta da superfície representada. Devido ao *warping*, os *texels* deslocaram-se para a direita. Na metade esquerda, a imagem foi cortada devido ao suporte da mesma estender-se somente até a metade da textura original. Na metade direita, um espaço em branco surge uma vez que as amostras que preencheriam tal espaço foram perdidas devido à restrição de tamanho do suporte da metade esquerda.

garante que o resultado final seja correto. É preciso realizar uma operação extra para permitir que o resultado final compreenda uma visão correta da superfície representada. Tal operação é descrita no Algoritmo 23, que realiza a união das imagens resultantes do *warping* dos mapas contidos em `texRel1` e `texRel2`.

Os parâmetros `quad1` e `quad2` contêm as imagens resultantes do *warping* aplicado aos mapas de `texRel1` e `texRel2`, respectivamente. Para que o resultado final seja correto, é necessário unir o conteúdo das imagens de `quad1` e `quad2` e armazenar o resultado desta união em `quad`. Dessa forma, nas linhas **3** a **6** são obtidos os *texels* correntes, do mapa de normal e do mapa de cor, contidos em `quad1` e `quad2`. Como no caso do algoritmo `divideBuffer`, o comando **Se** da linha **7** realiza a divisão dos mapas em relação à coluna do meio, ou seja; se $u <$ largura/2 então preenche-se a metade esquerda das texturas de `quad` com o conteúdo das imagens de `quad1`, caso contrário, preenche-se a metade direita de tais texturas com o conteúdo das imagens de `quad2`. Além disso, é preciso realizar a operação extra, citada anteriormente, que compreende os testes das linhas **8** e **16**. No caso da linha **8**, é necessário verificar se o valor de profundidade do *texel* corrente de `quad1` é inválido. Em caso afirmativo, o valor contido neste *texel* é irrelevante para a imagem final. Se, além disso, o valor de profundidade do *texel* corrente de `quad2` é válido, então significa que o valor contido em tal *texel* representa uma amostra válida cujo deslocamento ultrapassou a coluna do meio. Conseqüentemente, este *texel* deve ser atribuído a `quad` e sua presença na imagem final garante que a região branca ilustrada na Figura 4.7

```
algoritmo uneBuffers(quad1, quad2)
 1 Para v ← 0; v < altura; v ← v + 1 faça
 2   Para u ← 0; u < largura; u ← u + 1 faça
 3     {N1, D1} ← quad1.mNormalP[v, u];
 4     C1 ← quad1.mCorP[v, u];
 5     {N2, D2} ← quad2.mNormalP[v, u];
 6     C2 ← quad2.mCorP[v, u];
 7     Se u < largura/2 então
 8       Se D1 = 255 e D2 ≠ 255 então
 9         quad.mNormalP[v, u] ← {N2, D2};
10         quad.mCorP[v, u] ← C2;
11       senão
12         quad.mNormalP[v, u] ← {N1, D1};
13         quad.mCorP[v, u] ← C1;
14       fim_se
15     senão
16       Se D2 = 255 e D1 ≠ 255 então
17         quad.mNormalP[v, u] ← {N1, D1};
18         quad.mCorP[v, u] ← C1;
19       senão
20         quad.mNormalP[v, u] ← {N2, D2};
21         quad.mCorP[v, u] ← C2;
22       fim_se
23     fim_se
24   fim_para
25 fim_para
fim
```

Algoritmo 23: Pseudo-código para unir os *buffers* de `quad1` e `quad2`.

não apareça. O mesmo é válido para o teste da linha **16**, invertendo-se `quad1` para `quad2` e vice-versa.

A rotina `draw`, descrita na Seção 4.3.1, deve ser modificada para corresponder a esta nova abordagem. O Algoritmo 24 apresenta um pseudo-código para a nova rotina.

As modificações em relação à rotina da Seção 4.3.1 se concentram entre as linhas **5** e **16**, o restante permanece inalterado e, portanto, não requer considerações adicionais. Na primeira vez que a rotina `draw` é executada, os quatro primeiros passos do algoritmo de mapeamento de textura com relevo (linhas **5** e **6**) são realizados para `quad1` e `quad2`. Uma vez que o processo de *warping* é concluído, a união das imagens transformadas de `quad1` e `quad2` pode ser armazenada em `quad`. Em seguida, a textura resultante de tal união é enviada para a placa de vídeo. Uma vez que a textura é enviada para a placa de vídeo, a transição do estado **inicial** para o estado **executando**

```
algoritmo draw(quad, texRel, cam)
 1  Configura o estado de renderização da API OpenGL;
 2  Ativa programa de vértice;
 3  Ativa programa de fragmento;
 4  Se é a primeira execução de draw então
 5     quatroPrimeirosPassos(quad1);
 6     quatroPrimeirosPassos(quad2);
 7     quad ← uneBuffers(quad1,quad2);
 8     enviaTextura(quad);
 9     estadoThread1 ← EXECUTANDO_WARPING;
10     estadoThread2 ← EXECUTANDO_WARPING;
11  senão
12     Se estadoThread1 = ESPERANDO e
           estadoThread2 = ESPERANDO então
13        quad ← uneBuffers(quad1,quad2);
14        atualizaTextura(quad);
15        estadoThread1 ← EXECUTANDO_WARPING;
16        estadoThread2 ← EXECUTANDO_WARPING;
17     fim_se
18  fim_se
19  desenhaQuad(quad);
20  Desativa programa de fragmento;
21  Desativa programa de vértice;
fim
```

Algoritmo 24: Pseudo-código para a rotina principal de desenho executada uma vez por quadro modificada de acordo com a nova abordagem.

warping pode ser efetuada (linhas **9** e **10**) para ambos os quadriláteros. Nas circunstâncias em que a rotina `draw` é executada pela segunda vez em diante, necessita-se verificar se o estado corrente da *thread* de CPU, em ambos os quadriláteros, é **esperando**. Em caso afirmativo, significa que cada *thread* realizou uma operação de *warping* e, conseqüentemente, o resultado de cada *warping* pode ser unido para que a textura residente na placa de vídeo seja atualizada (linha **14**). Em seguida, cada *thread* de CPU sofre uma transição do estado **esperando** para o estado **executando warping** e uma nova operação de *warping* pode ser realizada.

Novamente, levando-se em conta o processamento empregado na CPU, esta nova abordagem pode ser caracterizada como pertencente à categoria de *sistema multi-threaded*, uma vez que existem somente dois processadores de CPU para executar três processos: duas *threads* de CPU mais o processo principal. Quanto ao modelo de programação, a nova abordagem pode ser caracterizada como pertencente ao modelo de *dados paralelos*, pois,

diferentes partes de uma imagem são processadas pelo mesmo conjunto de operações, no caso, os quatro primeiros passos do algoritmo de mapeamento de textura com relevo.

4.4 Discussão

Quando se trata de sistemas multi-processadores para aplicações gráficas de tempo real existem basicamente duas abordagens que podem ser utilizadas: *processamento paralelo* e *pipeline de multi-processadores*.

Nas seções anteriores foram apresentadas duas abordagens de processamento paralelo, onde o foco do paralelismo se concentrou nos quatro primeiros passos do algoritmo de mapeamento de textura com relevo.

Com o uso da tecnologia de *Hyper-Threading*, é possível montar um *pipeline* na CPU composto de dois estágios, um para cada processador disponível. O primeiro estágio seria responsável por realizar os três primeiros passos do algoritmo de mapeamento de textura com relevo mais o *warping* horizontal. O segundo estágio seria responsável por realizar o *warping* vertical juntamente com o restante das operações necessárias. Veja a Figura 4.8

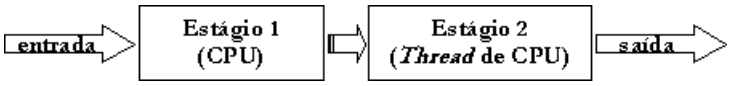

Figura 4.8: *Pipeline* de multi-processadores para uma arquitetura com *Hyper-Threading*.

Desta forma, a velocidade de execução em relação à abordagem seqüencial provavelmente aumentará uma vez que o trabalho necessário para realizar a amostragem e reconstrução a partir de texturas com relevo será dividido em estágios de *pipeline* que são executados em paralelo. No entanto, comparando-se esta nova abordagem com a abordagem paralela, a latência[1] tende a ser maior no *pipeline* de multi-processadores, uma vez que o número de estágios a serem percorridos é maior.

[1] É o tempo decorrido entre uma solicitação de usuário e o recebimento do resultado final.