8 Implementação e Resultados Práticos

...there's a difference between knowing the path and walking the path. (The Matrix)

8.1 Framework utilizado

Para o sistema de visualização deste trabalho utilizou-se um *framework* capaz de visualizar objetos 3D de forma interativa. Os cenários podem ser modelados utilizando o software 3DSMAX e devem ser diferenciados os objetos que serão tratados por geometria com os que serão tratados como impostores como relevo. A estrutura básica do *framework* está ilustrada na figura 8.1:

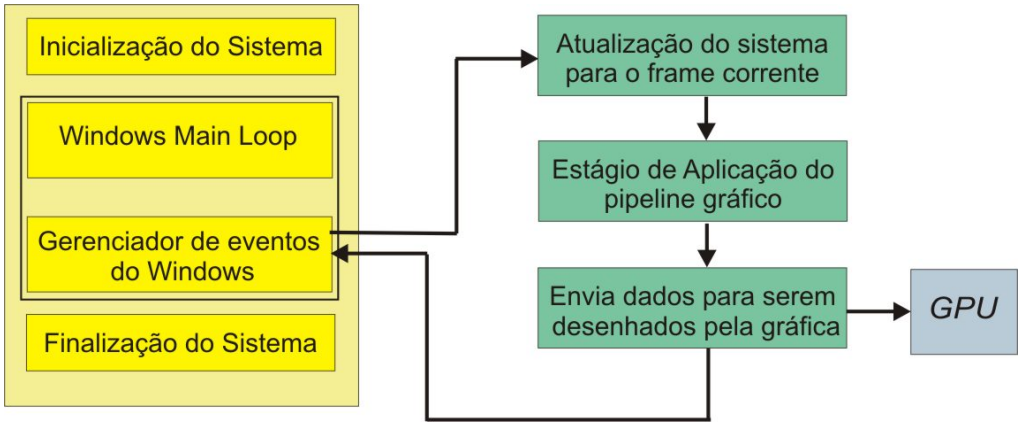


figura 8.1 – Diagrama da estrutura do *framework* utilizado para as implementações que se seguem

O sistema inicializa todas as variáveis globais antes de entrar no *loop* principal do *windows*. Uma vez dentro deste *loop*, para cada *frame*, executa-se a sequência da direita do diagrama, que a grosso modo pode ser resumido como sendo o estágio de aplicação do *pipeline* gráfico. No término deste estágio, antes de voltar ao início do *loop* principal, enviam-se os resultados para serem desenhados pela placa gráfica, correspondendo aos estágios de geometria e rasterização, que estão ambas a cargo da GPU. Uma vez feito este envio, a CPU já terminou sua parte do *pipeline* gráfico e apenas a placa gráfica possui tarefas pare

serem feitas. Neste âmbito, pode-se encarar o sistema como uma máquina paralela temporal, uma vez que o processador pode iniciar o processo de novas etapas da visualização enquanto a GPU está tratando a visualização dos dados anteriores.

8.2 Implementação básica do estágio de *pre-warping*

Foram implementadas quatro versões da técnica de mapeamento de texturas com relevo (Fonseca, 2004), variando o processo de amostragem e reconstrução da imagem. A estrutura do *framework* implementado é da seguinte forma:

Relief_Mapping (sprite, Relief, camera) // Chamado a cada frame
Reinicialização dos parâmetros para a câmera corrente
Configuração do polígono da textura com relevo para nova posição
Constrói_Look_Up_Table
Amostragem e Reconstrução da imagem
Mapeamento da textura com pre-warping sobre o polígono

Para todas as versões, procurando otimizar o cálculo das equações 3-1 e 3-2, que devem ser calculadas para cada *texel* do mapa de relevos, lançou-se mão de uma *lookup table*. Esta tabela deve ser reconstruída cada vez que qualquer parâmetro da câmera destino seja alterado e é feito pelo seguinte algoritmo (Oliveira, 2000):

Função Constrói_Look_Up_Table
$e_1 = -(\vec{c} \cdot \vec{a})/(\vec{a} \cdot \vec{a})$ *// e_1 e e_2 são as coordenadas do ponto epipolar*
$e_2 = -(\vec{c} \cdot \vec{b})/(\vec{b} \cdot \vec{b})$ $k_3 = 1/(\vec{c} \cdot \vec{f})$ *//k_1, k_2, k_3 são as constântes da eq. 3-1*
$k_1 = e_1 . k_3$ *// referentes à posição corrente da câmera*
$k_2 = e_2 . k_3$
Para i=0 até i >= 255 faça
Depth = Q(i) / 255 // Profundidade aproximada para um
// índice quantizado i
Coef_1[i] = k_1(Depth) // Coeficientes das Equações 3-1 e 3-2
Coef_2[i] = k_2(Depth)
Coef_3[i] = 1/(1+ k_3)(Depth)

Através desta *look up table*, os cálculos para cada *texel* que correspondiam a

$$u_i = \frac{u_f - k_1 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)} \quad \text{e} \quad v_i = \frac{v_f - k_2 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)}$$

são substituídos pelas seguintes operações:

u_{next} = *(u + Coef_1[Depth_Texel]) x Coef_3[Depth_Texel]*
v_{next} = *(v + Coef_2[Depth_Texel]) x Coef_3[Depth_Texel]*

o que se resume a 2 adições e 2 multiplicações por *texel*.

8.2.1 Amostragem Unidimensional Realizada em dois passos

O primeiro método consiste em criar dois passos totalmente separados e independentes para o processo de amostragem e reconstrução (Oliveira, 2000). Para esta versão cria-se uma função que realiza o cálculo de *warping* apenas para as linhas e outra que, terminada a primeira, realiza o cálculo de *warping* apenas para as colunas, conforme ilustra a figura 8.2. Como foi discutido na seção 4.2, ao realizar o *warping* de um *texel*, podem surgir buracos, ou seja, dois *texels* que eram vizinhos na textura original podem se afastar de tal forma que surgem *pixels* entre os dois que antes não existiam. (O tamanho destes buracos é delimitado pela medida de erro apresentada no capítulo 5, de forma que nunca são maiores do que um determinado valor estipulado).

Para solucionar este problema, o algoritmo implementado cria *texels* para esta região, fazendo uma interpolação linear entre os dois elementos pertencentes à imagem original. Esta interpolação deve ser efetuada não apenas para a cor dos *texels*, mas também para o valor das normais e da profundidade. Neste método, as interpolações horizontais e verticais são feitas separadamente. É importante notar que, ao atingir o passo vertical, faz-se uma interpolação sobre outra já feita no passo anterior.

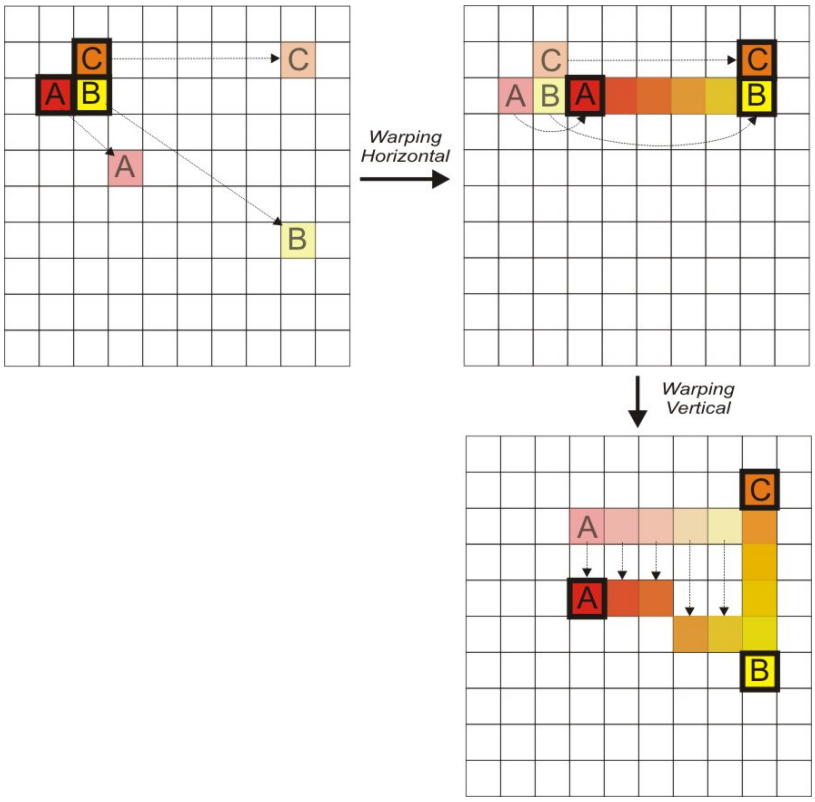


figura 8.2 – O método da amostragem unidimensional dividido em dois passos realiza primeiro o *warping* para as linhas, fazendo a interpolação necessária para preencher os buracos que surgem devido ao distanciamento de dois *texels* que eram vizinhos na imagem original. Depois de processar toda a imagem, realiza-se um segundo *warping*, restrito às colunas, também fazendo a interpolação de *texels* no caso de afastamentos dos mesmos.

Este método possui um inconveniente, especialmente mais notório, em locais de interpolação que existe maior descontinuidade na informação de profundidade. Este efeito pode-se tornar mais proeminente em linhas retas, que tenderão a parecer curvas. O erro se deve sobretudo ao fato de se realizar uma interpolação (no passo vertical) sobre um valor que já era interpolado (no passo horizontal)[*].

8.2.2 Amostragem Assimétrica Realizada em dois passos

Nesta versão da implementação das texturas com relevo se procura sanar o erro obtido na primeira abordagem. Neste caso, evita-se realizar uma interpolação

[*] Uma discussão mais detalhada das limitações desta abordagem pode ser obtida em (Oliveira, 99) e em (Fonseca, 04).

sobre outra, armazenando o valor final da coordenada vertical do *texel* já no primeiro passo do *pre-warping* (Oliveira, 2000), ou seja, no passo horizontal. Para isto, deve-se utilizar uma tabela auxiliar responsável por armazenar estes resultados. Denomina-se a esta versão de assimétrica porque o passo horizontal não é simétrico ao vertical.

8.2.3 Amostragem Realizada em Dois Passos com Compensação de Deslocamento

Neste método, sugerido e demonstrado em (Oliveira, 1999), utiliza-se um cálculo mais exato para a interpolação linear no cálculo dos valores de profundidade para os *texels* interpolados durante o passo horizontal. Esta interpolação, para os *texels* que estão entre A e B (figura 8.2) é dada pela equação 8-1:

$$disp(t) = \frac{\beta_1 - t\beta_2 - (1-t)\beta_3}{\gamma_1 + t\gamma_2 + (1-t)\gamma_3} \tag{8-1}$$

Onde *disp* (*t*) é o valor de profundidade interpolado, $\beta_1 = v_{fA}(1 + k_3 disp(B) - v_{pA}$, $\beta_2 = v_{fB} - v_{pA} + k_2 disp(B)$, $\beta_3 = v_{pA} k_3 disp(B)$, $\gamma_1 = v_{pA} k_3 - (1 + k_3 disp(B))k_2$, $\gamma_2 = (v_{fB} - v_{pA} + k_2 disp(B))k_3$ e $\gamma_3 = v_{pA}.k_3^2.disp(B)$, sendo que se denomina de v_{fN} ao *pixel N* da imagem fonte e v_{pN} ao *pixel N* da imagem com o *pre-warping*. Os demais coeficientes são os mesmos da equação 3-1.

8.2.4 Amostragem Intercalada Realizada em um Passo

Nas implementações descritas ainda é possível de ser observada a presença de ruídos durante o *pre-warping*. Tais ruídos surgem pelo fato de que múltiplas amostras são mapeadas sobre um mesmo *pixel* na imagem resultante, durante a primeira fase do *pre-warping*. Ao se realizar a segunda etapa, alguns *texels* que se tornam necessários para se fazer a interpolação já não existem, pois foram ocludidos. Neste método implementado, utilizou-se a solução indicada em (Oliveira, 1999) para este problema: intercalar os passos horizontal e vertical do *pre-warping*. Para cada *texel* intermediário produzido no passo horizontal realiza-

se imediatamente sua interpolação dentro da coluna apropriada. Como cada iteração vertical recebe e processa os *texels* numa ordem compatível de oclusão, a visibilidade correta é preservada na imagem resultante.

Este método, diante dos três apresentados anteriormente, é o que menos operações realiza na implementação do *pre-warping*, devido às intercalações. Por esta razão é o que possui maior velocidade, como se pode verificar na tabela 8.1, e é o método utilizado para o restante do desenvolvimento que se apresenta nesta dissertação.

Algoritmo	Tempo médio de processamento (ms)	Desvio Padrão (ms)
Amostragem unidimensional	47.406	3.001
Amostragem Assimétrica	47.859	4.612
Amostragem com compensação	49.625	6.075
Amostragem intercalada	40.859	7.734

Tabela 8.1 – Tempo obtido para realizar o *pre-warping* utilizando cada um dos 4 algoritmos de amostragem discutidos previamente. O modelo usado para esta medida é o da figura 5.4, numa resolução de 256 x 256 e utilizando uma máquina com processador Pentium IV 2.6 GHz.

8.3 *Pre-warping* serial no *pipeline* gráfico

Numa primeira implementação, inseriu-se toda a etapa de *pre-warping* das texturas com relevo no estágio de aplicação, conforme ilustra a figura 8.3:

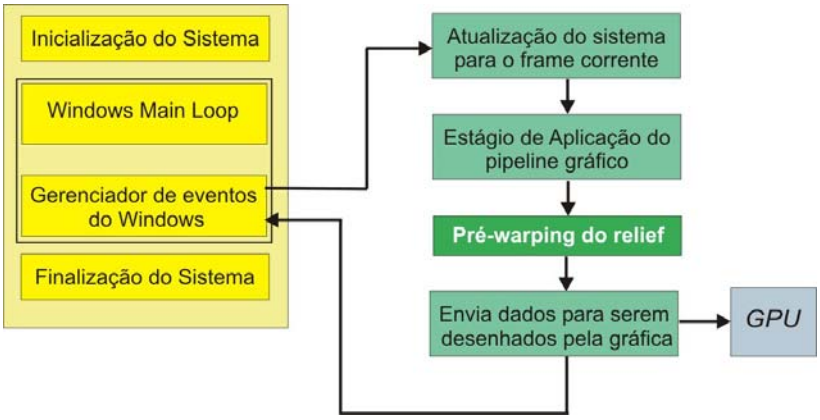


figura 8.3 – Primeira implementação do mapeamento de texturas com relevo utilizando o *framework* apresentado. Apenas se mostra no diagrama a etapa de *pre-warping*, já que a etapa de texturização está a cargo da GPU.

Nesta implementação não houve nenhuma preocupação com paralelismo. Assim sendo, enquanto se está realizando a etapa de *pre-warping*, todo o sistema está parado, à espera do resultado a ser gerado. Ao se fazer isto, pôde-se observar que a etapa de *pre-warping* tornou-se o gargalo de toda a aplicação, ou seja, ela tornou-se a responsável por restringir a performance do sistema. Num Pentium IV, 3GHz com uma RADEON 9700, enquanto numa cena sem objetos com texturas com relevo a performance era de 95 FPS (*frames* por segundo), ao se inserir um objeto deste tipo a performance caiu para 11 FPS.

Além disso, nesta primeira implementação utiliza-se uma textura com relevo já pronta e armazenada na memória. Como existe apenas uma textura para o objeto, torna-se impossível vê-lo de qualquer ângulo. Desta maneira, o *framework* restringe o movimento da câmera. A figura 8.4 mostra um diagrama das etapas envolvidas no processo, bem como uma identificação do local onde esta etapa é executada: CPU ou GPU.

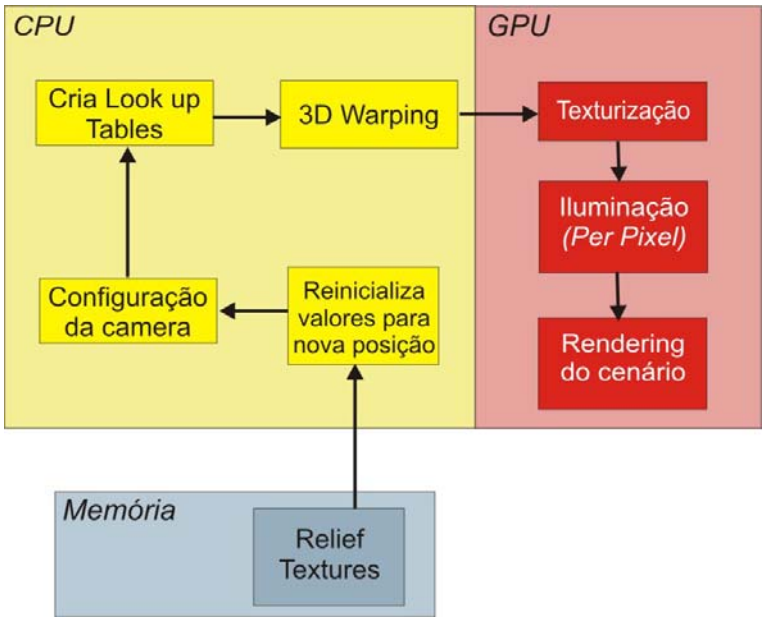


figura 8.4 – Distribuição entre CPU e GPU na implementação do sistema.

8.4 *Pre-warping* com *Time-Slice* fixo

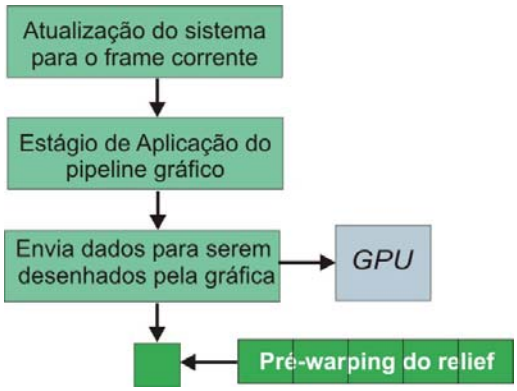
Numa segunda implementação, procurando encarar um sistema com CPU e GPU como uma máquina paralela, criou-se uma distribuição baseada em *time-slice*: dividiu-se a etapa de *pre-warping* em vários pedaços. Para cada *frame*, faz-se com que a CPU realize o *pre-warping* de uma parte da textura. Apenas ao

terminar o *pre-warping* por completo se envia esta textura para a placa gráfica, de modo a tratar da etapa de texturização. Perceba-se no diagrama da figura 8.5 (a) que o *pre-warping* foi inserido após enviar os dados para a GPU. Fez-se isto porque se observou que neste momento a CPU estava com maior tempo ocioso. Enquanto a GPU realiza todo o estágio de geometria e de rasterização, a CPU está tratando de uma parte do *warping*. Ao utilizar esta abordagem, a taxa de visualização da cena subiu para 75 FPS, mas a taxa de atualização das texturas com relevo caiu para 6 por segundo. Embora a atualização destas texturas tenha sido menor, esta abordagem foi melhor porque comprometeu menos a interatividade da aplicação e a visualização dos objetos geométricos.

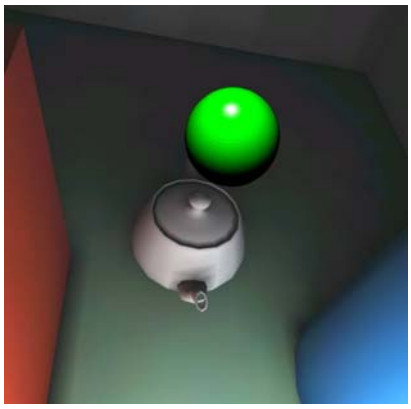
8.5 *Pre-warping* com *time-slice* variável

Uma variação feita na implementação anterior foi a de não fixar o tempo concedido ao *warping*: permite-se que a CPU faça o cálculo de *warping* apenas enquanto dura o processo de visualização da placa gráfica. Desta maneira, garante-se que a CPU apenas está processando a textura com relevo enquanto a GPU está de fato ocupada. Quando o *framework* volta a precisar da CPU, esta interrompe o *warping* que estava realizando. Sempre que se termina uma varredura completa da textura com relevo, a CPU avisa que no próximo frame a GPU pode usar uma textura nova para o objeto com relevo, que está disponível na memória RAM. (constatou-se que não é conveniente realizar o *warping* diretamente na memória de vídeo, pois isto interrompe o processamento que está ocorrendo na GPU).

Esta implementação deixou a taxa de FPS praticamente igual à que existia antes de se inserir o objeto de textura com relevo, mas por outro lado, a taxa de atualização da textura com relevo ficou mais lenta (em torno de 3 a 4 atualizações por segundo, para a cena da figura 8.5). Vale ressaltar também que quanto mais complexa for a cena para a GPU, mais tempo a CPU tem para realizar o *pre-warping* e, portanto, maior será a taxa de atualizações da textura com relevo.



(a)



(b)

figura 8.5 – (a) Estrutura do *framework* para *time slice*. (b) enquanto a cena é desenhada a uma taxa de 75 FPS, a esfera, que é representada por textura com relevo, é atualizada a 6 vezes por segundo, na primeira versão desta implementação. Já na segunda versão, a taxa ficou em torno de 95 FPS, mas a taxa de *warpings* por segundo caiu para 3.

8.6 *Pre-warping* com *multi-threading*

Numa terceira abordagem implementou-se o mapeamento de texturas com relevo utilizando *multi-threading*. Neste caso, o *pre-warping* ficou a cargo de um *thread* exclusivo para isto (para tanto é necessário utilizar uma máquina com mais de um processador; no caso deste trabalho utilizou-se um processador com *hyper-threading*). Com isto um processador fica dedicado ao *pipeline* de visualização padrão, sem ser interrompido nenhuma vez para fazer as tarefas relacionadas ao mapeamento de texturas com relevo.

Através de mensagens, este processador avisa ao *thread* se houve mudanças de posição do observador, o que requer que um novo *warping* seja criado para a textura. O *thread*, ao terminar de fazer o *pre-warping*, avisa ao processador do *framework*, que por sua vez, permite que o *thread* envie o resultado para a GPU. Este processo de sincronização é implementado através de uma máquina de estados, ilustrado na figura 8.6. Novamente vale ressaltar que o *warping* foi feito inteiramente na memória RAM, para que o processamento da placa gráfica não seja interrompido a todo instante. Apenas quando se termina uma sequência de *warping*, realiza-se a transferência para a memória de textura.

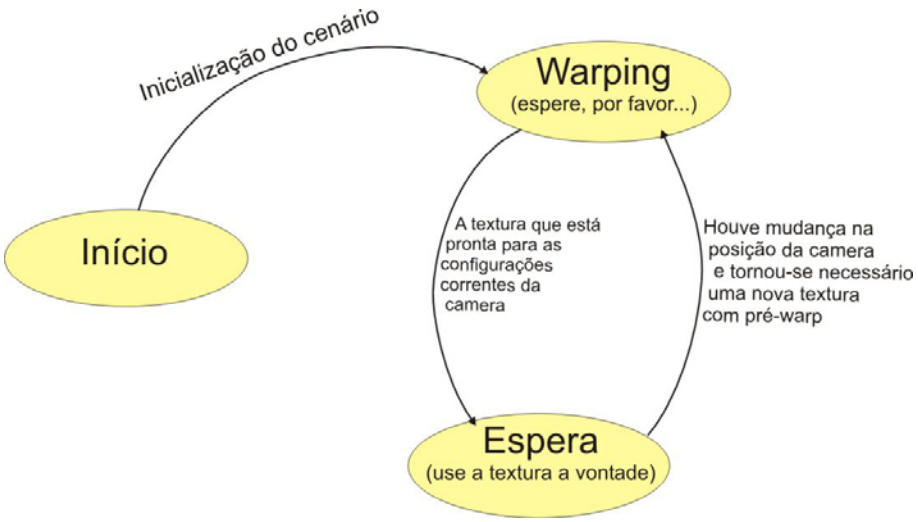


figura 8.6 – Máquina de estados responsável por sincronizar o processo de *pre-warping*.

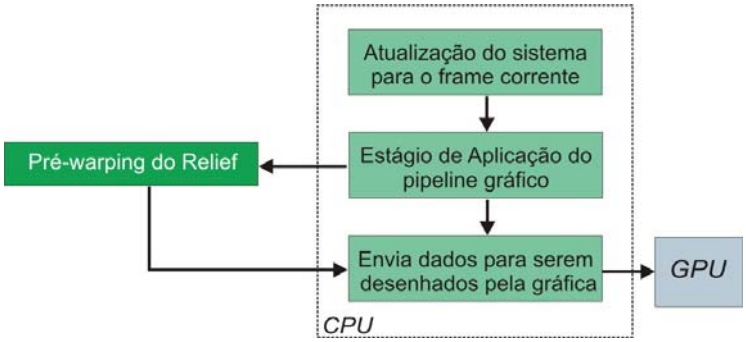


figura 8.7 – Estrutura do *framework* para texturas com relevo utilizando *multi-thread*.

A máquina de estados é controlada pelo *framework* (figura 8.8). Perceba-se que não há mais nenhuma conexão entre o laço de visualização do cenário normal e o laço do *pre-warping*. A máquina de estados garante que não se passe para a placa gráfica uma textura com relevo que está no meio do processo de *pre-warping*.

Testou-se este sistema também numa máquina sem possuir *hyper-threading*. Neste caso, o resultado foi superior ao apresentado no *pre-warping* com *time-slice* variável. Nesta versão deu-se ao processo de *pre-warping* uma prioridade pequena, de forma que o sistema operacional gerenciasse os momentos em que a CPU tem tempo disponível para este processo. Assim sendo, sempre que a CPU encontra tempo ocioso passa a dedicar-se ao *pre-warping*. Observou-se que numa aplicação de visualização com o *framework* desenvolvido, sem *culling* por software, o tempo ocioso da CPU chega a 90% para uma cena de 1000 polígonos.

Desta forma, o tempo que existe à disposição para o passo de amostragem e reconstrução da textura com relevo é grande.

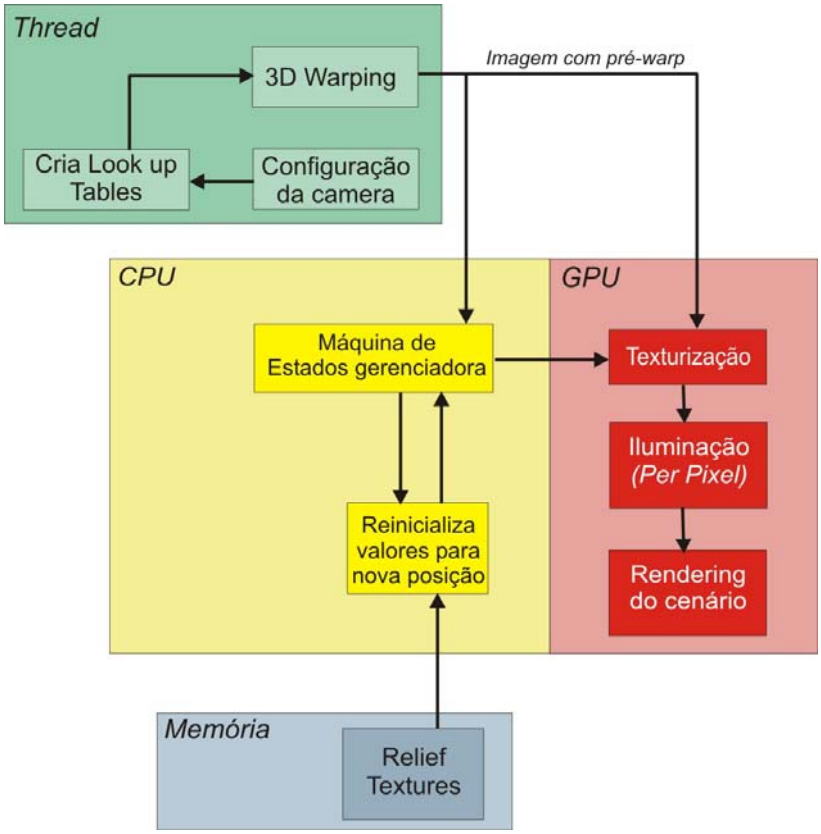


figura 8.8 – Arquitetura mais detalhada do *framework* com implementação de processo distribuído para o *pre-warping*.

8.7 *Pre-warping* multi-processado

Na implementação anterior é apresentada uma arquitetura que cria um processo dedicado à etapa de *pre-warping*. Esse método é especialmente interessante quando se dispõe de uma máquina com dois processadores ou com tecnologia de *hyper-threading*, mas não é capaz de explorar sistemas que possuem mais do que dois processadores. A abordagem presente é a que pode ser considerada com mais propriedade de paralela, pois divide a textura com relevo em *n* pedaços, onde *n* é o número de processadores disponíveis. Neste caso, cada um se encarrega do *pre-warping* de um pedaço. Quando todos terminam, um processo deve se responsabilizar por unir os diversos resultados numa só textura. Este método é classificado como paralelismo espacial, de acordo com as categorias apresentadas na seção 7.3.

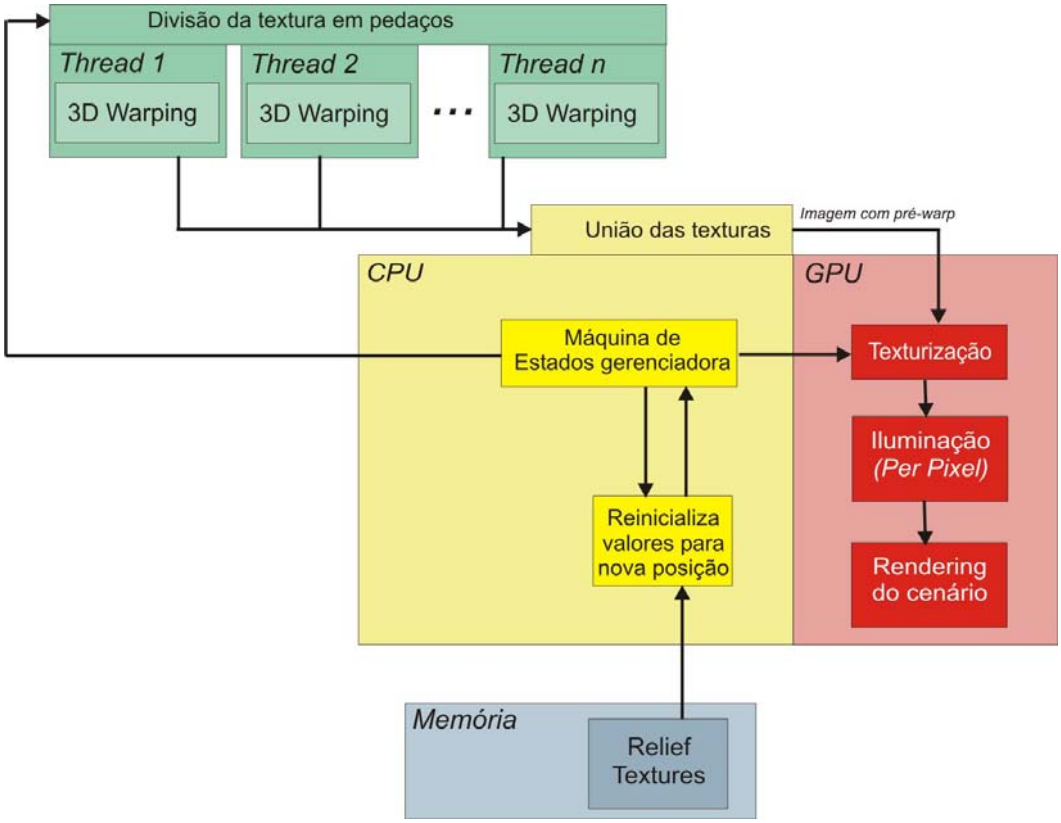
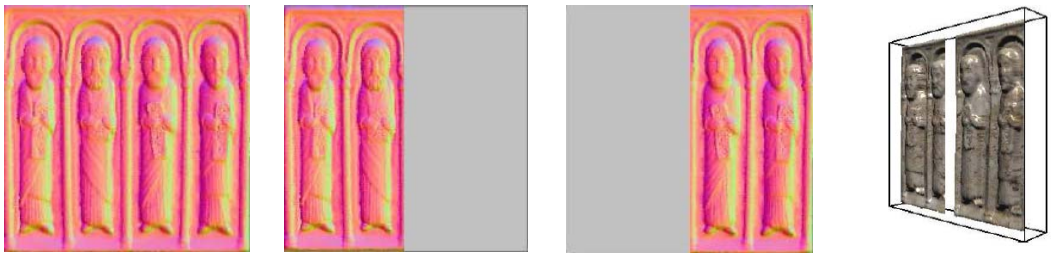


figura 8.9 – Arquitetura do sistema para multi-processamento. Cada *thread* recebe um pedaço da textura original e se encarrega de fazer o *pre-warping* apenas neste pedaço.

Deve-se ressaltar que o processo de divisão da textura com relevo não consiste simplesmente em quebrar a imagem original em *n* pedaços menores. Ao se fazer o *pre-warping* de um pedaço, pode ocorrer de que um *texel* seja deslocado para uma região que pertence a outro *thread*. Se o processo não possuir acesso a esta parte, o *texel* que se está movendo é simplesmente perdido. O efeito disto consiste em "buracos" nas emendas dos pedaços, conforme se pode ver na figura 8.10 (d).



(a) (b) (c) (d)

figura 8.10 – Ao se dividir a imagem em *n* partes, o *warping* de uma pode "invadir" o espaço que era de outra, ocorrendo a perda destes *texels* e formando um "buraco" na imagem com relevo.

Para evitar este problema, a divisão da textura com relevo em pedaços deve ser feita da seguinte forma:

- Alocar um espaço de memória para cada processo com a mesma resolução da textura com relevo original;
- Inicializar os valores de profundidade para todos os *texels* como sendo um valor inválido;
- Copiar as informações da textura original apenas para a região que é usada para este pedaço.

Estes passos podem ser observados na figura 8.10. Em (a) está representada a textura original. (b) e (c) representam dois pedaços diferentes que serão enviados a processadores distintos para sofrerem o *pre-warping*. A área cinza de cada uma destas imagens indica que há valores de profundidade equivalentes a *texels* inválidos. Fazendo-se isto, permite-se que cada processo escreva valores nos *texels* da região de outra imagem, caso seja necessário.

O processo necessário para unir as texturas com o *pre-warping* terminado também deve levar em conta esta possível invasão de áreas. O algoritmo abaixo descreve esta união:

Para i de 1 a n faça
Para cada texel pertencente ao pedaço i faça
Se texel é válido então copie este texel para textura final
Senão verificar se pedaço vizinho apresenta este mesmo texel como válido
Caso seja válido, copie-o para a textura final

A tabela 8.2 mostra resultados obtidos para 3 implementações: *pre-warping* serial, *pre-warping* com *multi-threading* e *pre-warping* paralelo. Neste caso, a medida do tempo é feita pela taxa de FPS do processo de visualização completa e não do número de *warpings* por segundo que estão sendo feitos.

Pode-se perceber que o desempenho obtido no *pre-warping* paralelo é muito inferior ao obtido com a abordagem *multi-threading*. Isto ocorre porque no primeiro existem 3 processos para 2 processadores (um processo do *framework* e dois processos do *pre-warping*) e no segundo 2 processos para 2 processadores. Deve-se ter em conta que na abordagem paralela existe ainda o processo de separação e junção das texturas, que não é necessário para o *multi-threading*.

Outra observação que se faz é de que o valor de FPS corresponde ao do *framework* como um todo, e não à taxa de *warpings* por segundo que se está realizando. Esta tabela mostra que, para a arquitetura de *hardware* escolhida, o segundo método deixa o *framework* mais livre para o processamento do restante da cena.

Abordagem	Taxa de FPS	Desvio padrão
pre-warping Serial	24.93	1.54
pre-warping Multi-threading	96.45	5.01
pre-warping multiprocessado	31.09	7.67

tabela 8.2 – Comparação da performance obtida para as diversas abordagens do *pre-warping*. A textura com relevo utilizada é a mesma da tabela 8.1, com um Pentium IV 2.6 GHz com *hyper-threading*.

Outra abordagem, que pode ser mais eficiente, porém não implementada, consiste em criar *n threads*, deixando cada um responsável pela linha *i mod Altura_Imagem*, sendo *i* o número do processo e *Altura_Imagem* a resolução vertical da textura com relevo. Utilizando o método de amostragem intercalada (seção 8.2.4), poderia-se escrever o resultado diretamente na textura final, não sendo necessário realizar a etapa de junção.

8.8 *Pre-warping* com atualização dinâmica dos Impostores com relevo

Nesta versão, implementou-se primeiramente a estimativa de Schaufler, de forma a minimizar o número de *warpings* realizados sobre o impostor. Quando o *warping* não for necessário, o impostor é tratado como um simples *sprite*. Isto é especialmente conveniente para o caso de se implementar um sistema de previsão para gerar texturas com relevo, em máquinas que apenas possuem 2 processadores (e.g., processadores com *hyper-threading*), pois neste caso pode-se liberar o *thread* de *warping* para esta função. Apenas com esta otimização, observou-se a seguinte melhoria, para a câmera parada:

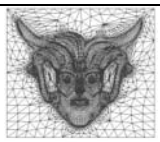
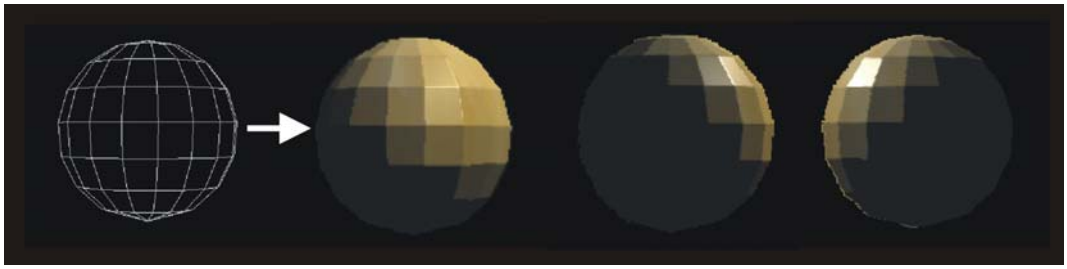
Objeto	FPS (sem Schaufler)	FPS (com Schaufler)
	35	120

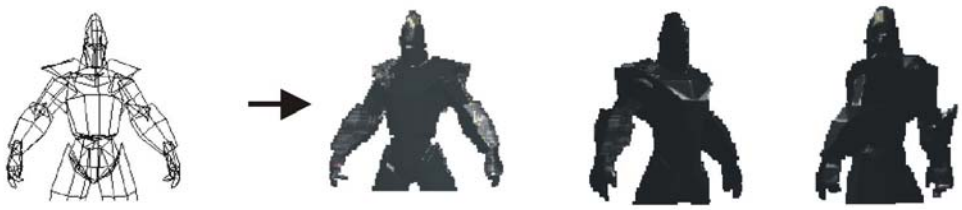
tabela 8.3 – Comparação de performance com implementação do teste de Schaufler para otimizar número de *warpings*. Os testes foram feitos sem ativar o *hyper-threading*. Esta otimização está apresentada com detalhes na seção 5.3.

Sempre que ocorrer um movimento do observador e o teste de Schaufler falhar, então o sistema permite que se inicie um *thread* para realizar um *warping* sobre a textura com relevo. Quando este termina, aplica-se o teste da métrica apresentado na seção 5.4. Caso se comprove que o erro seja maior do que o valor tolerado, inicia-se o cálculo de uma nova textura com relevo, que é usada tão logo fique pronta.

A figura 8.11 mostra dois exemplos de impostores com relevo: Em (a) foram necessários 14 atualizações para se dar uma volta completa sobre a esfera, que é um objeto com topologia simples, já em (b) foram necessárias 26 atualizações para o mesmo movimento sobre um objeto com geometria complexa.



(a)



(b)

figura 8.11 – Dois exemplos de Impostores com relevo: (a) representa uma geometria simples, onde são necessárias 14 atualizações para se dar uma volta por completo, já no caso de (b) se representa uma geometria complexa e são necessárias 26 atualizações para que o erro do *warping* passe desapercebido ao longo de uma volta completa.

No capítulo 5.1 discutiu-se a necessidade de gerar uma nova textura com relevo para o objeto, quando estes critérios de erro forem atingidos. Nesta implementação criou-se um sistema paralelo que gera uma nova textura com relevo quando o observador chega a uma região crítica. Esta região corresponde a um local próximo ao de um ponto em que o erro superará o valor tolerado pela equação (5-7). Caso existam apenas dois processadores, o *thread* para a geração da nova textura com relevo deve ter prioridade menor do que o processo de *pre-warping*.

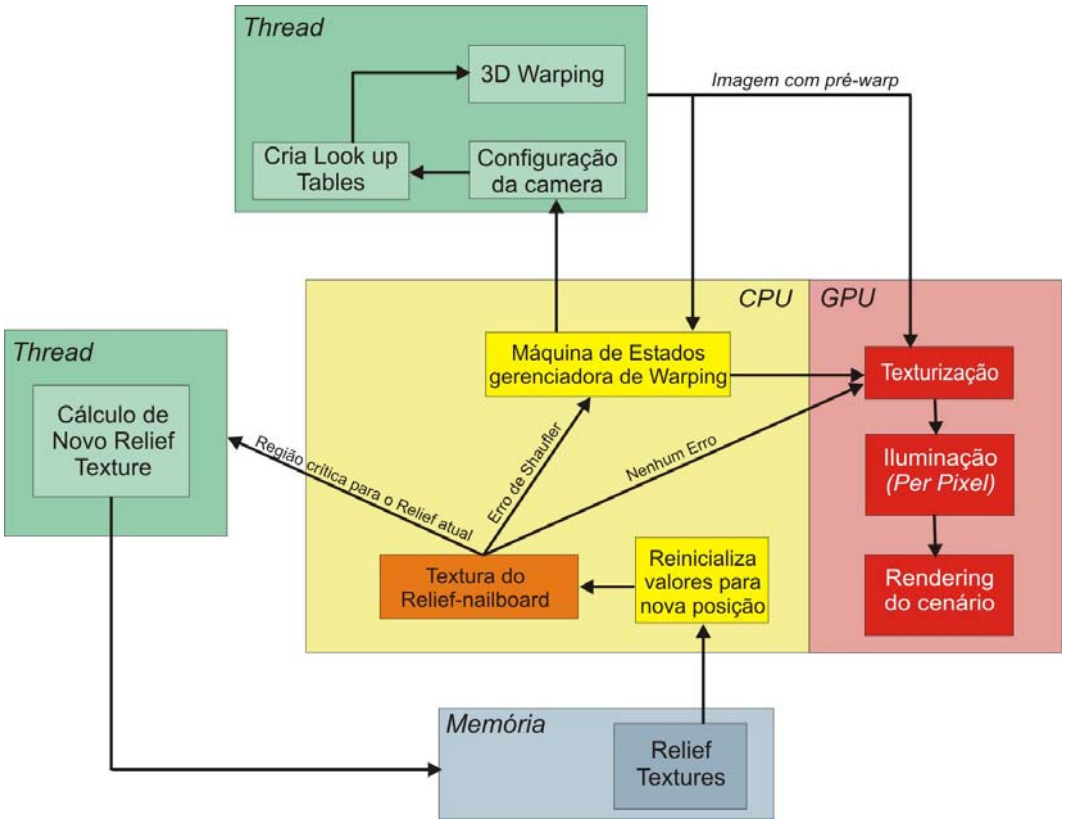


figura 8.12 – Arquitetura do sistema com otimização e métricas de erro.

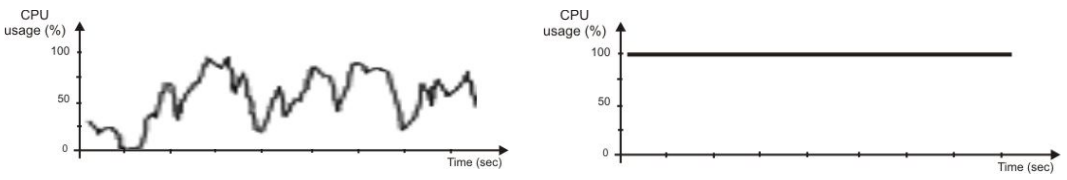
A geração de uma nova textura corresponde a uma visualização do objeto pela CPU, podendo-se utilizar qualquer algoritmo que forneça a profundidade de um ponto. Denominam-se os programas que realizam este cálculo de *software shaders*. Como foi discutido, estes possuem a capacidade de calcular efeitos que podem ser impossíveis de serem obtidos pela GPU. Neste trabalho implementou-se um *software shader* para *ray-tracing*, omitindo a computação da iluminação, já que este fica a cargo do *pixel shader*, utilizando o mapa de normais do objeto em questão. Este é um algoritmo que não é conveniente de ser implementado na placa

gráfica, por envolver recursão (até a data do presente trabalho as linguagens de programação para placa gráfica impossibilitam que haja recursão). A figura 8.13 mostra um mesmo objeto sendo renderizado pelo *software shader* e pela GPU.

figura 8.13 – O personagem da esquerda foi renderizado pelo *software shader* de *Ray-tracing* e o personagem da direita foi renderizado diretamente pela GPU.

O tempo consumido para se realizar o warping da uma imagem, que é de 256 x 256 pixels, num Pentium IV 2.6GHz foi de 11 ms, resultando numa taxa de 91 warpings por segundo. Já o tempo para gerar um novo impostor com relevo pelo *software shader* foi de 228 ms, para um nível de Ray-tracing, num objeto composto por 1500 polígonos. Para manter um erro de warping acumulado quase imperceptível, o valor colocado para a heurística gerou 17 imagens ao dar uma volta completa sobre o objeto da figura 8.13. Para esta cena, a taxa de frames por segundo foi de 131 no caso de ser totalmente renderizada em GPU. Já para o caso de se usar software shader, a taxa foi de 124. Apesar de serem valores muito próximos, esta pequena perda se deve especialmente ao tempo de transferência que ocorre da imagem na memória do sistema para a memória de vídeo.

A figura 8.14 mostra o tempo de consumo de CPU para ambos os casos.



(a) (b)

figura 8.14 – Tempo de consumo de CPU sem o *software shader* (a) e com o *software shader* (b)

Pelo exemplo apresentado, pode-se comprovar que houve um pequeno comprometimento da taxa de frames por segundos ao inserir um *software shader* responsável por gerar impostores com o algoritmo de ray-tracing. Entretanto o acréscimo de precisão e realismo na imagem gerada é consideravelmente maior. Manter a taxa de frames por segundos praticamente inalterada, porém acrescentando recursos de visualização mais realistas é justamente uma das principais propostas dos impostores com relevo. A figura 8.14 mostra que o processamento extra para obter tal resultado foi retirado do tempo que antes estava sendo desperdiçado pela CPU.

As texturas com relevo que foram usadas e deixaram de valer, podem ser armazenadas num *cache*, juntamente com as posições no espaço onde são válidas. Estas posições podem ser geradas através de volumes, estimados a partir da métrica de erro apresentada. Para objetos com superfícies contínuas, e portanto com valores pequenos no gradiente da profundidade, estes volumes tendem a ser maiores. Objetos com muitas descontinuidades possuem um mosaico maior de volumes ao seu redor (figura 4.3 e 8.11). Desta forma, antes de efetivamente ser gerado uma nova textura com relevo, verifica-se se há um volume na nova posição do observador e, neste caso, utiliza-se uma textura previamente calculada ao invés de gerar uma nova.

Quando o espaço de memória disponível para as texturas com relevo estiver saturado e for necessário descartar alguma textura para dar espaço a uma nova, é conveniente analisar o volume que se encontra mais distante da posição corrente do observador ou analisar a direção do movimento e estimar o local menos provável de se estar nos próximos *frames*. Esta idéia, bem como outras possíveis melhorias, estão discutidas no próximo capítulo: Conclusão.