3
Amostragem e Reconstrução de Imagens

O Capítulo 2 apresentou a formalização dos conceitos envolvidos no mapeamento de textura com relevo. Dentre estes, descreveu o processo de *warping*, isto é, como determinar as coordenadas de pontos infinitesimais na imagem destino a partir de pontos na imagem fonte. Neste capítulo, há uma discussão acerca das variantes de implementação da técnica utilizada nesta dissertação.

O processo de criar novas imagens a partir de imagens discretas por meio de transformações espaciais é composto de dois estágios: *amostragem* e *reconstrução*.

A estrutura unidimensional das equações de pré *warping* possibilita a amostragem ser implementada como um processo de dois passos utilizando operações unidimensionais ao longo de linhas e colunas.

No entanto, devido à natureza geométrica do processo de amostragem a partir de texturas com relevo, *pixels* adjacentes na imagem fonte podem se tornar não-consecutivos após a avaliação das equações de pré *warping*. O estágio de reconstrução consiste na interpolação de tais amostras não-adjacentes.

O que se segue é uma discussão detalhada das estratégias de amostragem e reconstrução de imagens a partir de texturas com relevo, bem como de suas vantagens e desvantagens.

3.1
O Problema Computacional

Segundo Cormen *et al.* [13], informalmente, um *algoritmo* é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como *entrada* e produz algum valor ou conjunto de valores como *saída*. Também, pode-se visualizar um algoritmo como uma ferramenta para resolver um *problema computational*.

No caso deste trabalho, o problema computacional envolvido é o seguinte: ***Dados a posição corrente*** p ***do observador, um quadrilátero*** q ***e uma textura com relevo*** $\{i_s, K_s\}$ ***como entrada; obter uma imagem*** i_t ***que represente a visualização correta do mapeamento de*** $\{i_s, K_s\}$ ***sobre*** q ***visto de*** p.

3.2
Representação Computacional dos Dados de Entrada

De acordo com o problema computacional definido na seção anterior; a posição corrente p do observador, o quadrilátero q e a textura com relevo $\{i_s, K_s\}$ são os dados de entrada para o problema.

3.2.1
Posição corrente do Observador

Para que uma cena 3D possa ser visualizada na tela de um computador, é necessário que exista um modelo de câmera que descreva corretamente o processo de visualização. Além disso, é necessário dispor da *transformação de modelagem* que descreve a maneira como um objeto, especificado no espaço do objeto, é posicionado no espaço desta cena virtual.

A posição corrente do observador nada mais é do que a posição da câmera virtual utilizada para visualizar a cena 3D e, dessa maneira, pode ser representada por um ponto do espaço $\mathbb{R}^3$.

O modelo de câmera associado, geralmente o modelo de projeção perspectiva ilustrado na Seção 2.2, descreve uma transformação geométrica,

mais conhecida como *transformação de visualização*, que define a orientação de visualização. Tal transformação pode ser representada por uma matriz 4x4.

Da mesma forma que o modelo de câmera, a transformação de modelagem também pode ser representada por uma matriz 4x4. Por questões de eficiência [16], as transformações de modelagem e visualização são combinadas em uma única matriz denominada matriz *ModeloVisão* (do inglês, *ModelView*).

Logo, a estrutura de dados `Camera` representa computacionalmente as informações relacionadas com a posição do observador e direção de visualização.

Tipo `Camera`
{
  `posicao` $= \{p \mid p \in \mathbb{R}^3\}$;
  `modeloVisao` $= \{m_{i,j} \in \mathbb{R} \mid 1 \leq i \leq 4 \text{ e } 1 \leq j \leq 4\}$;
};

3.2.2 Textura com Relevo

Como definido no Capítulo 2, uma textura com relevo é composta de cor, profundidade e informação de câmera. Porém, somente com tais informações não é possível capturar efeitos dependentes do ponto de vista e da direção de iluminação. Uma possível solução para representar tais efeitos é o uso de mapas de normais em conjunção com mapeamento de textura com relevo.

Dessa maneira, no caso específico deste trabalho, um mapa de normal é associado a uma textura com relevo. A informação de normal é combinada com profundidade gerando um mapa de normal com profundidade, que é representado como uma imagem RGBA onde o canal *alpha* armazena valores de profundidade e os canais de cor armazenam o vetor normal. A informação de cor é armazenada num mapa de textura convencional que é representado como uma imagem RGB. O que se segue é a representação computacional de uma imagem RGB, de uma imagem RGBA e de uma textura com relevo, respectivamente, onde $\mathbb{P} = \{x \in \mathbb{N} \mid 0 \leq x \leq 255\}$.

Tipo `ImgRGB`
{
  `largura` = $\{w \in \mathbb{N} \mid w$ é a extensão horizontal em *pixels*$\}$;
  `altura` = $\{h \in \mathbb{N} \mid h$ é a extensão vertical em *pixels*$\}$;
  `mapa` = $\{m_{v,u} \in \mathbb{P}^3 \mid 1 \leq v \leq$ `altura` e $1 \leq u \leq$ `largura`$\}$;
};

Tipo `ImgRGBA`
{
  `largura` = $\{w \in \mathbb{N} \mid w$ é a extensão horizontal em *pixels*$\}$;
  `altura` = $\{h \in \mathbb{N} \mid h$ é a extensão vertical em *pixels*$\}$;
  `mapa` = $\{m_{v,u} \in \mathbb{P}^4 \mid 1 \leq v \leq$ `altura` e $1 \leq u \leq$ `largura`$\}$;
};

Tipo `TexturaRelevo`
{
  `mNormal` = `ImgRGBA`;
  `mCor` = `ImgRGB`;
};

É importante ressaltar que os efeitos dependentes do ponto de vista e da direção de iluminação contribuem na aparência de superfícies não-difusas e o uso de mapas de normais produz somente soluções para iluminação local, uma vez que normais são propriedades geométricas de superfícies e portanto não mudam com o ponto de vista.

Tipicamente, uma única textura é utilizada para adicionar detalhes a múltiplas superfícies. Por exemplo, uma única textura de tijolos pode ser utilizada na parte exterior de todas as paredes de um prédio em construção. Além de cor, profundidade e normal; texturas com relevo possuem um modelo de câmera de projeção paralela associado. Enquanto os valores de profundidade descrevem a distância entre as amostras e um plano de referência, os parâmetros do modelo de câmera especificam a posição, orientação e dimensões da textura com relevo em um mundo virtual. Mais precisamente, o ponto $\dot{C}$ determina a posição; o vetor $\vec{f}$ define a orientação, e os vetores $\vec{a}$ e $\vec{b}$ determinam as dimensões. Conseqüentemente, alterando-se os parâmetros de câmera apropriadamente, uma textura com relevo pode ser instanciada múltiplas vezes da mesma maneira que uma textura convencional. Por esta razão, os parâmetros do modelo de câmera não são armazenados na estrutura de dados `TexturaRelevo`, e sim na estrutura de dados `Quadrilatero`, que é descrita na próxima seção.

3.2.3
Quadrilátero

O mapeamento de textura com relevo é um mapeamento planar, ou seja, é uma transformação geométrica que associa a cada ponto p_i um elemento de textura, tal que p_i faça parte de um determinado conjunto de pontos do espaço da cena 3D. Mais especificamente, estes pontos devem pertencer a um mesmo plano.

Dessa maneira, durante o processo de amostragem e reconstrução da imagem resultante, deve haver uma entidade planar sobre a qual uma textura com relevo possa ser mapeada. Tal entidade planar, geralmente, é um quadrilátero cuja representação computacional utilizada é a apresentada pela estrutura de dados `Quadrilatero`; onde v_0, v_1, v_2 e v_3 são os vértices. Além disso, para que texturas com relevo possam ser utilizadas como primitivas de modelagem e renderização, é necessário que a informação de câmera seja associada à entidade `Quadrilatero` de modo que uma textura com relevo possa ser reutilizada em vários quadriláteros. Porém, somente desacoplar a informação de câmera da textura com relevo não é o suficiente para garantir que esta pode ser reutilizada, pois, a informação contida no mapa de normal está coerente com a orientação original da superfície a ser representada. Dessa forma, é preciso um cuidado especial durante o cálculo de iluminação que será discutido na Seção 3.6.

Tipo `Quadrilatero`
{
- `v0` $= \{p \in \mathbb{R}^3 \mid p$ é o canto inferior esquerdo$\}$;
- `v1` $= \{p \in \mathbb{R}^3 \mid p$ é o canto inferior direito$\}$;
- `v2` $= \{p \in \mathbb{R}^3 \mid p$ é o canto superior direito$\}$;
- `v3` $= \{p \in \mathbb{R}^3 \mid p$ é o canto superior esquerdo$\}$;
- `C` $= \{\dot{C} \in \mathbb{R}^3 \mid \dot{C}$ é o COP do modelo de câmera $K_s\}$;
- `a` $= \{\vec{a} \in \mathbb{R}^3 \mid \vec{a}$ determina a dimensão horizontal de $K_s\}$;
- `b` $= \{\vec{b} \in \mathbb{R}^3 \mid \vec{b}$ determina a dimensão vertical de $K_s\}$;
- `f` $= \{\vec{f} \in \mathbb{R}^3 \mid \vec{f} = \left(\vec{a} \times \vec{b}\right) \div \|\vec{a} \times \vec{b}\|\}$;
- `c` $= \{\vec{c} \in \mathbb{R}^3 \mid \vec{c} = \dot{C} -$ (posição corrente do observador)$\}$;
- `mNormalI` = `ImgRGBA`;
- `mCorI` = `ImgRGB`;
- `mNormalP` = `ImgRGBA`;
- `mCorP` = `ImgRGB`;

};

Os *buffers* de imagem `mNormalI` e `mNormalP` representam a informação

sobre os mapas de normais no espaço de imagem intermediário e transformado, respectivamente, cujo entendimento ficará mais claro no decorrer do capítulo. O mesmo se aplica a `mCorI` e `mCorP`.

3.3
Processo de Aquisição de Texturas com Relevo

Uma imagem com profundidade é uma imagem digital com um modelo de câmera associado onde cada elemento do espaço de cor é aumentado para incluir um valor escalar representando a distância, no espaço Euclideano, entre o ponto amostrado e uma entidade de referência (Seção 2.2). No caso de texturas com relevo, o modelo de câmera é o de projeção paralela.

No processo de aquisição de texturas com relevo, utilizou-se um *plugin* do 3D *Studio MAX*®, implementado por Fabio Policarpo [29], que dados um modelo poligonal, uma caixa envolvente (*bounding box*) para este modelo e uma câmera de projeção paralela, armazena a imagem resultante da visualização do modelo poligonal, através da câmera, em uma imagem RGBA. Nos três primeiros canais de cada *texel* são armazenadas as componentes do vetor normal ao modelo poligonal naquele ponto. No canal *alpha* são armazenados valores de profundidade quantizados, que variam de 0 a 255, da seguinte maneira: para uma dada orientação de visualização, *texels* que representam pontos pertencentes à face frontal da caixa recebem o valor 0, os que representam pontos pertencentes à face posterior da caixa recebem o valor 254 e, finalmente, os que representam pontos localizados entre ambas as faces recebem um valor entre 0 e 254 de acordo com a sua posição em relação à caixa envolvente. Um valor extra, no caso 255, é reservado para amostras que não são parte do modelo representado, tais como valores de *background*. Vale notar que o termo entidade de referência apresentado na definição de uma imagem com profundidade, neste caso, é representado pela face frontal da caixa envolvente.

É importante salientar que na ausência de tal quantização os valores de profundidade associados a cada *texel* teriam que ser armazenados como um valor ponto-flutuante e, conseqüentemente, o requerimento de armazenamento para tal informação equivaleria à quantidade necessária para o dado de normal (32 *bits* por amostra); e, como bem observado por Oliveira [26], transferir uma textura com relevo, nestas condições, da memória principal

para a memória de textura iria requerer duas vezes tanta largura de banda e espaço quanto uma textura regular. Dessa maneira, além de reduzir tais requerimentos, o uso de valores de profundidade quantizados ajuda a melhorar a coerência da memória *cache*, uma vez que a profundidade e o vetor normal associados a um *texel* são sempre utilizados juntos.

3.4
Amostragem e Reconstrução a partir de Texturas com Relevo

A Figura 3.1 ilustra a estrutura do algoritmo de mapeamento de textura com relevo realizado em cinco passos. Inicialmente, faz-se uma reinicialização dos recursos (basicamente *buffers* de imagens e *lookup tables*) utilizados durante a execução do algoritmo. Em seguida, realiza-se a configuração do modelo de câmera K_s onde os parâmetros $\dot{C}$, $\vec{a}$, $\vec{b}$, $\vec{f}$ e $\vec{c}$ são transformados pela matriz *ModeloVisão*. Anteriormente ao passo de pré *warping*, é necessário efetuar o cálculo de algumas *lookup tables*. O passo de pré *warping* é implementado em duas fases, uma horizontal e outra vertical. Finalmente, realiza-se o mapeamento de textura convencional.

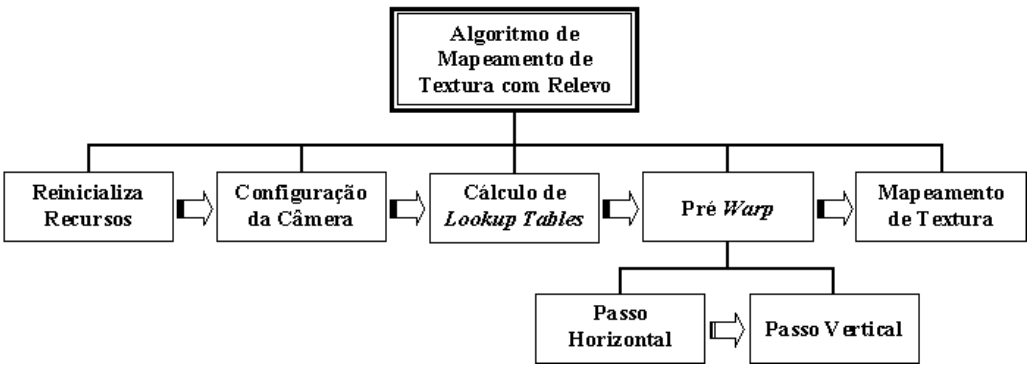

Figura 3.1: Estrutura do algoritmo de mapeamento de textura com relevo.

As seções seguintes tratam das variantes de implementação do processamento empregado durante a etapa de pré *warping*. Para cada variante de implementação é apresentado um pseudo-código que descreve o processo de amostragem e reconstrução de imagens a partir de texturas com relevo. Posteriormente, as etapas restantes são apresentadas com mais detalhes.

3.4.1

Amostragem Unidimensional Realizada em Dois Passos

A ordem compatível de oclusão especifica a ordem na qual os *texels* de uma textura com relevo podem ser processados de maneira que a visibilidade correta seja alcançada para pontos de vista arbitrários. Como ilustrado na Figura 2.10 (Capítulo 2, página 37), a linha epipolar divide a imagem fonte em no máximo quatro regiões.

Assumindo-se que tais regiões estejam definidas de antemão, sejam (min_u, min_v) e (max_u, max_v) o canto superior esquerdo e o canto inferior direito de uma destas regiões (Figura 3.2), respectivamente; e sejam $step_u$ e $step_v$ o deslocamento horizontal e vertical durante o processo de *warping* para tal região, respectivamente.

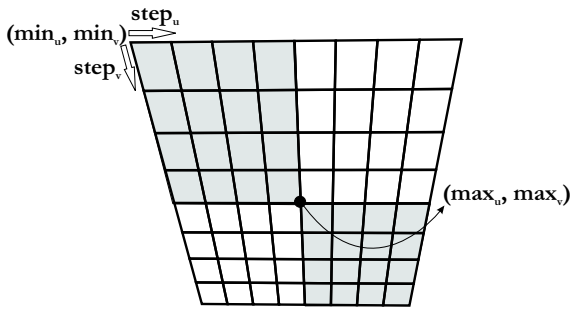

Figura 3.2: A região superior esquerda é definida pelos pontos (min_u, min_v) e (max_u, max_v). $step_u$ e $step_v$ definem o deslocamento horizontal e vertical do processo de *warping* para tal região, respectivamente.

O Algoritmo `warpHorizontal`, na página 48, descreve o passo horizontal do estágio de pré *warping* de uma textura com relevo.

Neste algoritmo, `texRel` é um objeto do tipo `TexturaRelevo` e armazena a textura com relevo propriamente dita, `quad` é um objeto do tipo `Quadrilatero` e armazena os parâmetros do modelo de câmera associado a esta entidade bem como os *buffers* `mNormalI` e `mCorI` que são os mapas de normal e cor transformados após a realização do passo horizontal do estágio de pré *warping*, N é uma variável que armazena um vetor normal, D armazena um valor de profundidade e C armazena uma cor.

O laço da linha **1** é responsável por iterar sobre as linhas da textura `texRel`, enquanto que no laço da linha **3** as colunas são iteradas. A variável booleana `flag` manipula a inicialização dos valores de u_{prev}, $\{N_{prev}, D_{prev}\}$ e C_{prev}. Se o valor contido em `flag` é **falso** significa que uma inicialização deve ocorrer, caso contrário, nenhuma inicialização é realizada. Isto evita

```
algoritmo warpHorizontal(min_u, min_v, max_u, max_v, step_u, step_v,
quad, texRel)
 1  Para v ← min_v; v ≠ max_v; v ← v + step_v faça
 2     flag ← falso;
 3     Para u ← min_u; u ≠ max_u; u ← u + step_u faça
 4        {N_in, D_in} ← texRel.mNormal[v, u];
 5        C_in ← texRel.mCor[v, u];
 6        Se D_in = 255 então
 7           Se flag = verdadeiro então
 8              flag ← falso;
 9           fim_se
10           continue;
11        senão se D_in = 0 então
12           quad.mNormalI[v, u] ← {N_in, D_in};
13           quad.mCorI[v, u] ← C_in;
14           u_prev ← u;
15           {N_prev, D_prev} ← {N_in, D_in};
16           C_prev ← C_in;
17           Se flag = falso então
18              flag ← verdadeiro;
19           fim_se
20           continue;
21        fim_se
22        u_next ← Eq_2_14(u, D_in);
23        Se flag = falso então
24           flag ← verdadeiro;
25           u_prev ← u_next;
26           {N_prev, D_prev} ← {N_in, D_in};
27           C_prev ← C_in;
28        fim_se
29        reconstrucao(v, step_u, u_prev, u_next, N_prev, D_prev, C_prev, N_in,
     D_in, C_in,quad);
30        u_prev ← u_next;
31        {N_prev, D_prev} ← {N_in, D_in};
32        C_prev ← C_in;
33     fim_para
34  fim_para
fim
```

Algoritmo 3: Pseudo-código para o passo horizontal do estágio de pré *warping* de uma textura com relevo. As variáveis N, D e C armazenam o vetor normal, a profundidade e a cor de um determinado *texel*, respectivamente.

que ocorra interpolação entre *texels* pertencentes a regiões desconexas[1] em

[1]Duas regiões numa linha (ou coluna) da imagem são desconexas se, e somente se, tais regiões são separadas por uma seqüencia de *texels* cujo valor de profundidade seja inválido, isto é, igual a 255.

uma linha da imagem.

Nas linhas **4** e **5** são obtidos os *texels* na posição (u, v) do mapa de normal e do mapa de cor do objeto `texRel`. Uma vez que tais elementos foram obtidos, é necessário verificar o valor de profundidade encontrado. Se tal valor é igual a 255 a amostra corrente pertence ao *background* e nenhum processamento é necessário na linha corrente. Se o valor é 0 o *warping* não é requerido e os *texels* $\{N_{in}, D_{in}\}$ e C_{in} podem ser armazenados em `mNormalI` e `mCorI`, respectivamente. Finalmente, se o valor de profundidade é diferente de 0 e 255 a amostragem é realizada através da avaliação da equação de pré *warping* e o estágio de reconstrução é executado. Toda esta verificação está descrita entre as linhas **6** e **21** do Algoritmo 3.

A posição do *texel* transformado pelo *warping* é calculada com base na coluna e no valor de profundidade correntes, de acordo com a Equação (2-14) (linha **22** do Algoritmo 3). Entre as linhas **23** e **28** ocorre a verificação do valor de `flag`, pois, se este é **falso** é necessário inicializar as variáveis N_{prev}, D_{prev} e C_{prev} que serão posteriormente utilizadas na etapa de reconstrução.

No estágio de reconstrução (Algoritmo 4), os valores de profundidade, normal e cor entre os *texels* anterior e corrente são linearmente interpolados.

algoritmo `reconstrucao`(v, $step_u$, u_{prev}, u_{next}, N_{prev}, D_{prev}, C_{prev}, N_{in}, D_{in}, C_{in}, quad)
1 $i \leftarrow (u_{prev} + step_u)$;
2 **Para** $u_{out} \leftarrow i$; $u_{out} \neq u_{next}$; $u_{out} \leftarrow u_{out} + step_u$ **faça**
3 quad.mNormalI$[v, u_{out}] \leftarrow$ `lerp`$(\{N_{prev}, D_{prev}\}, \{N_{in}, D_{in}\})$;
4 quad.mCorI$[v, u_{out}] \leftarrow$ `lerp`(C_{prev}, C_{in});
5 **fim_para**
fim

Algoritmo 4: Pseudo-código para o estágio de reconstrução do passo horizontal do pré *warping* de uma textura com relevo, onde `lerp`(a, b) realiza a interpolação linear entre os valores a e b.

Finalmente, nas linhas **30**, **31** e **32**, do algoritmo `warpHorizontal`, os valores das amostras, após o processo de amostragem, são armazenados para uso durante o próximo passo.

Uma vez que o passo horizontal tenha processado todas as linhas, o mesmo algoritmo (substituindo-se a Equação (2-14) pela Equação (2-15), trocando as variáveis u e v entre si e utilizando como texturas de entrada os mapas `mNormalI` e `mCorI`) é aplicado às colunas da imagem resultante, conseqüentemente manipulando o deslocamento vertical dos

texels. A Figura 3.3 ilustra o processo inteiro para um *texel* B.

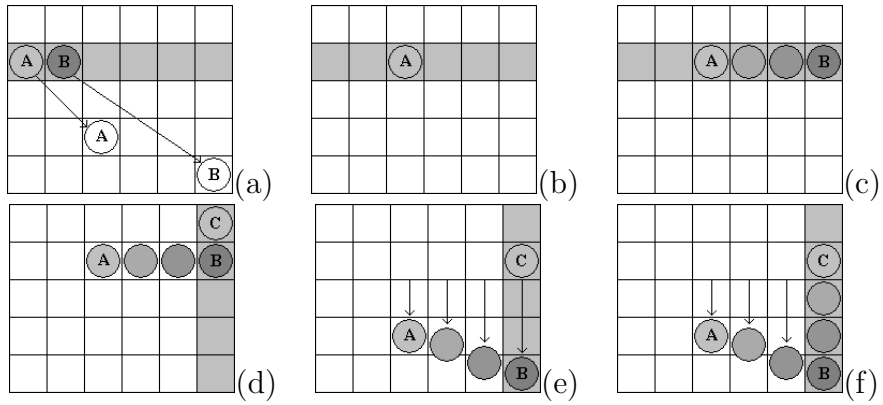

Figura 3.3: *Warping* de um *texel*. (a) *Texels* fonte A e B e suas posições finais. (b) O primeiro *texel* da linha corrente é movido para sua coluna final. (c) O próximo *texel* é movido para sua coluna final e valores de profundidade, normal e cor são interpolados. (d) Após o passo horizontal, o *texel* C é adjacente ao *texel* B. (e) Passo vertical: coordenadas de linhas são computadas utilizando os valores de profundidade interpolados e, ao longo de cada coluna, *texels* são movidos para suas linhas finais. (f) Valores de cor e normal são interpolados. (*Imagem adaptada de [26].*)

A Figura 3.3(a) mostra dois *texels* adjacentes A e B e suas posições após o pré *warping*. No passo horizontal o *texel* A é movido para sua coluna final (Figura 3.3(b)) e, uma vez que o mesmo acontece para o *texel* B, valores de profundidade, normal e cor são interpolados durante a reconstrução (Figura 3.3(c)). Seja C o *texel* acima de B após a realização do passo horizontal (Figura 3.3(d)). Durante o segundo passo, os valores de profundidade interpolados são utilizados para computar a coordenada de linha final de todos os *texels*. Ao longo das colunas, cada *texel* é movido para sua linha final (Figura 3.3(e)) e valores de cor e normal são interpolados (Figura 3.3(f)).

3.4.1.1

Limitações

Apesar de apresentar resultados de alta qualidade visual, o Algoritmo 3 é propenso ao aparecimento de ruídos resultantes de oclusão. Por exemplo, considere a Figura 3.4 onde a linha epipolar cruza a imagem em questão no canto inferior esquerdo. A área diagonal corresponde a uma região profunda, enquanto que o resto da superfície tem profundidade igual a zero. Seguindo-se um caminho hipotético, o *texel t* moverá para a esquerda

durante o passo horizontal e, após algumas iterações, será sobrescrito por outro *texel* com deslocamento igual a zero. Neste caso, nenhuma informação estará disponível para o passo vertical.

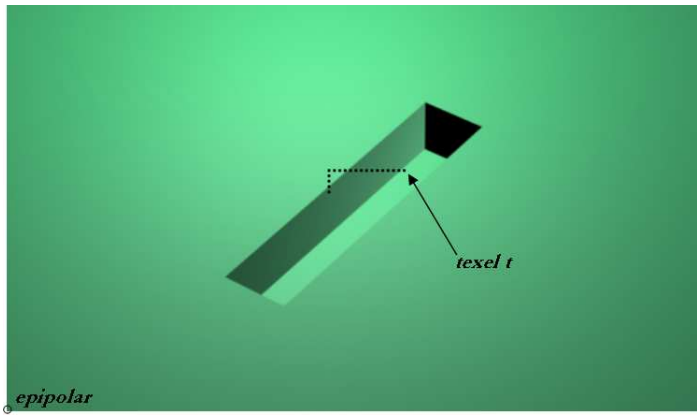

Figura 3.4: Ruídos resultantes de oclusão. A área diagonal corresponde a uma região profunda, enquanto o restante tem profundidade igual a zero. De acordo com o *warping* o *texel* t moverá para a esquerda e será sobrescrito por um outro *texel* com profundidade igual a zero. Neste caso, nenhuma informação estará disponível para o passo vertical. (*Imagem adaptada de [26].*)

Segundo Oliveira [26], além dos erros resultantes de oclusão, durante a execução do Algoritmo 3 podem aparecer ruídos resultantes de erros de interpolação de cor. Tais artefatos ocorrem porque o passo de pré *warping* não preserva a topologia da imagem, ou seja, *texels* que mantém uma relação de vizinhança na imagem fonte, após a realização do pré *warping*, podem perdê-la.

Ainda há problemas resultantes de distorção não-linear. O mais comum é a aparência curvada de linhas que, antes da realização do pré *warping*, eram retas. As seções 3.4.2 e 3.4.3 apresentam mais detalhes sobre o problema bem como possíveis soluções.

3.4.2

Amostragem Assimétrica Realizada em Dois Passos

A causa de distorções não-lineares no Algoritmo 3, segundo Oliveira [26], é o uso de uma *transformação linear fracionária*[2] para computar as coordenadas dos *texels* durante o segundo passo do pré *warping*.

[2] A razão de duas funções lineares.

Como discutido na Seção 3.4.1, durante o passo horizontal, valores de profundidade são interpolados linearmente (linha **3** do Algoritmo 4). Tal interpolação linear é descrita pela Equação (3-1).

$$displ(t) = (1-t)displ_{prev} + (t)displ_{in},\ t \in [0,1] \tag{3-1}$$

No segundo passo do pré *warping*, os valores de profundidade interpolados são utilizados para obter as coordenadas da linha final como

$$v_p(t) = \frac{v + k_2 displ(t)}{1 + k_3 displ(t)}$$

$$v_p(t) = \frac{v + k_2((1-t)displ_{prev} + (t)displ_{in})}{1 + k_3((1-t)displ_{prev} + (t)displ_{in})} \tag{3-2}$$

onde $displ(t)$ é o valor de profundidade interpolado computado através da Equação (3-1), e $v_p(t)$ é a coordenada da linha de um *texel* na imagem resultante após o pré *warping*. A não linearidade da Equação (3-2) é, deste modo, a causa das distorções.

A solução mais direta para evitar distorções não-lineares é substituir a interpolação e o armazenamento de valores de profundidade pela interpolação e armazenamento das coordenadas da linha final no decorrer do primeiro passo. Tais valores de linha, então, tornam-se imediatamente disponíveis para o segundo passo.

O Algoritmo 5, na página 53, apresenta o pseudo-código para o passo horizontal do *warping* assimétrico onde `vIntermediário` é um arranjo bidimensional, com uma entrada para cada *texel*, que armazena as coordenadas de linha interpoladas.

Como no Algoritmo 3, os laços das linhas **1** e **3** são responsáveis por iterar sobre as linhas e as colunas da textura `texRel`, respectivamente.

Além da coordenada da coluna (linha **24** do Algoritmo 5), a coordenada da linha final, para onde o *texel* corrente deve se deslocar, também é calculada. Tal cálculo é realizado com base na linha e no valor de profundidade correntes, neste caso, de acordo com a Equação (2-15) (linha **25** do Algoritmo 5). Note que, ao contrário da abordagem vertical para o Algoritmo 3, onde as coordenadas da linha final são calculadas com base nos valores de profundidade interpolados, neste procedimento as coordenadas da linha final são calculadas com base nos valores de profundidade da imagem

```
algoritmo warpHorizontal1(min_u, min_v, max_u, max_v, step_u,
step_v, quad, texRel)
 1  Para v ← min_v; v ≠ max_v; v ← v + step_v faça
 2     flag ← falso;
 3     Para u ← min_u; u ≠ max_u; u ← u + step_u faça
 4        {N_in, D_in} ← texRel.mNormal[v, u];
 5        C_in ← texRel.mCor[v, u];
 6        Se D_in = 255 então
 7           Se flag = verdadeiro então
 8              flag ← falso;
 9           fim_se
10           continue;
11        senão se D_in = 0 então
12           quad.mNormalI[v, u] ← {N_in, D_in};
13           quad.mCorI[v, u] ← C_in;
14           vIntermediário[v, u] ← v;
15           u_prev ← u;
16           v_prev ← v;
17           {N_prev, D_prev} ← {N_in, D_in};
18           C_prev ← C_in;
19           Se flag = falso então
20              flag ← verdadeiro;
21           fim_se
22           continue;
23        fim_se
24        u_next ← Eq_2_14(u, D_in);
25        v_next ← Eq_2_15(v, D_in);
26        Se flag = falso então
27           flag ← verdadeiro;
28           u_prev ← u_next;
29           v_prev ← v_next;
30           {N_prev, D_prev} ← {N_in, D_in};
31           C_prev ← C_in;
32        fim_se
33        reconstrucao1(v, step_u, u_prev, u_next, v_prev, v_next, N_prev, D_prev,
     C_prev, N_in, D_in, C_in,quad);
34        u_prev ← u_next;
35        v_prev ← v_next;
36        {N_prev, D_prev} ← {N_in, D_in};
37        C_prev ← C_in;
38     fim_para
39  fim_para
fim
```

Algoritmo 5: Pseudo-código para o passo horizontal do *warping* assimétrico. As variáveis N, D e C armazenam o vetor normal, a profundidade e a cor de um determinado *texel*, respectivamente.

fonte e, conseqüentemente, a não linearidade da Equação (3-2) é evitada.

No estágio de reconstrução (Algoritmo 6), além dos valores de profundidade, normal e cor serem interpolados, a coordenada da linha na iteração anterior e da linha na iteração corrente, obtidas através da avaliação da Equação (2-15), são também linearmente interpoladas para uso durante o passo vertical (linha **5** do Algoritmo 6).

algoritmo `reconstrucao1`(v, $step_u$, u_{prev}, u_{next}, v_{prev}, v_{next}, N_{prev}, D_{prev}, C_{prev}, N_{in}, D_{in}, C_{in}, quad)
1 $i \leftarrow (u_{prev} + step_u)$;
2 **Para** $u_{out} \leftarrow i$; $u_{out} \neq u_{next}$; $u_{out} \leftarrow u_{out} + step_u$ **faça**
3 quad.mNormalI$[v, u_{out}] \leftarrow$ `lerp`$(\{N_{prev}, D_{prev}\}, \{N_{in}, D_{in}\})$;
4 quad.mCorI$[v, u_{out}] \leftarrow$ `lerp`(C_{prev}, C_{in});
5 vIntermediário$[v, u_{out}] \leftarrow$ `lerp`(v_{prev}, v_{next});
6 **fim_para**
fim

Algoritmo 6: Pseudo-código para o estágio de reconstrução do passo horizontal do *warping* assimétrico, onde `lerp`(a, b) realiza a interpolação linear entre os valores a e b.

Existem algumas vantagens em calcular ambas as coordenadas do *texel* transformado no primeiro passo do algoritmo. Além de eliminar a distorção, o denominador das Equações (2-14) e (2-15) é computado somente uma vez por *texel* ao invés de duas vezes. O custo extra de calcular a coordenada da linha final durante o primeiro passo é compensado por não ter de calculá-la durante o passo vertical.

3.4.3

Amostragem Realizada em Dois Passos com Compensação de Deslocamento

Uma solução alternativa para o problema da distorção não-linear, como sugere Oliveira [26], é utilizar um método mais exato do que a interpolação linear no cálculo dos valores de profundidade para os *texels* interpolados durante o passo horizontal.

Dessa maneira, sejam A e B os *texels* anterior e corrente do passo horizontal, respectivamente (Figura 3.3(a)). Uma vez que o *warping* envolve três espaços de imagem distintos — definidos por Oliveira [26] como *source*, *intermediate* e *pre-warped* — subscritos serão utilizados para identificá-los.

Subscritos também serão utilizados para identificar cada *texel*. Conseqüentemente, por exemplo, (u_{sA}, v_{sA}) se refere ao *texel* A no espaço da imagem fonte. Da mesma forma, (u_{iB}, v_{iB}) se refere ao *texel* B no espaço da imagem intermediária, e (u_{pA}, v_{pA}) se refere ao *texel* A no espaço da imagem transformada pelo *warp*.

De acordo com a linha **5** do Algoritmo 6 da seção anterior, as coordenadas de linha são interpoladas como:

$$v_p(t) = (1-t)v_{pA} + (t)v_{pB},\ t \in [0,1] \tag{3-3}$$

Substituindo-se a equação de pré *warping* $v_{pB} = \frac{v_{sB}+k_2 displ_B}{1+k_3 displ_B}$ dentro da Equação (3-3) produz

$$v_p(t) = (1-t)v_{pA} + t\left(\frac{v_{sB}+k_2 displ_B}{1+k_3 displ_B}\right) \tag{3-4}$$

A Equação (3-4) é claramente linear uma vez que k_3 e $displ_B$ são ambas constantes, e tal equação pode ser reescrita como:

$$v_p(t) = \frac{v_{pA} + (v_{sB} - v_{pA})\,t + ((1-t)v_{pA}k_3 + tk_2)\,displ_B}{1 + k_3 displ_B} \tag{3-5}$$

Sabe-se, da Equação (3-2), que a coordenada da linha para onde o *texel* corrente deverá deslocar-se durante o passo vertical é dada por:

$$v_p(t) = \frac{v_{sB} + k_2 displ(t)}{1 + k_3 displ(t)} \tag{3-6}$$

Desse modo, um valor de profundidade compensado pode ser obtido resolvendo-se as Equações (3-5) e (3-6) para $displ(t)$, o que proporciona:

$$displ(t) = \frac{\beta_1 - t\beta_2 - (1-t)\beta_3}{\gamma_1 + t\gamma_2 + (1-t)\gamma_3} \tag{3-7}$$

onde $\beta_1 = v_{sB}(1 + k_3 displ_B) - v_{pA}$, $\beta_2 = v_{sB} - v_{pA} + k_2 displ_B$, $\beta_3 = v_{pA}k_3 displ_B$, $\gamma_1 = v_{pA}k_3 - (1 + k_3 displ_B)k_2$, $\gamma_2 = (v_{sB} - v_{pA} + k_2 displ_B)k_3$ e $\gamma_3 = v_{pA}k_3^2 displ_B$.

Os novos valores de profundidade, $displ(t)$, são computados no passo horizontal do pré *warping* e armazenados no arranjo `displCompensada` como descrito no Algoritmo 7, na página 56.

```
algoritmo warpHorizontal2(min_u, min_v, max_u, max_v, step_u,
step_v, quad, texRel)
 1  Para v ← min_v; v ≠ max_v; v ← v + step_v faça
 2     flag ← falso;
 3     Para u ← min_u; u ≠ max_u; u ← u + step_u faça
 4        {N_in, D_in} ← texRel.mNormal[v, u];
 5        C_in ← texRel.mCor[v, u];
 6        Se D_in = 255 então
 7           Se flag = verdadeiro então
 8              flag ← falso;
 9           fim_se
10           continue;
11        senão se D_in = 0 então
12           quad.mNormalI[v, u] ← {N_in, D_in};
13           quad.mCorI[v, u] ← C_in;
14           displCompensada[v, u] ← lookupTableDispl[D_in];
15           u_prev ← u;
16           v_prev ← v;
17           {N_prev, D_prev} ← {N_in, D_in};
18           C_prev ← C_in;
19           Se flag = falso então
20              flag ← verdadeiro;
21           fim_se
22           continue;
23        fim_se
24        u_next ← Eq_2_14(u, D_in);
25        v_next ← Eq_2_15(v, D_in);
26        Se flag = falso então
27           flag ← verdadeiro;
28           u_prev ← u_next;
29           v_prev ← v_next;
30           {N_prev, D_prev} ← {N_in, D_in};
31           C_prev ← C_in;
32        fim_se
33        reconstrucao2(v, step_u, u_prev, u_next, v_prev, v_next, N_prev, D_prev,
     C_prev, N_in, D_in, C_in,quad);
34        u_prev ← u_next;
35        v_prev ← v_next;
36        {N_prev, D_prev} ← {N_in, D_in};
37        C_prev ← C_in;
38     fim_para
39  fim_para
fim
```

Algoritmo 7: Pseudo-código para o passo horizontal do *warping* com compensação de valores de profundidade.

No passo horizontal, as únicas diferenças entre as abordagens assimétrica (Algoritmo 5) e com compensação de deslocamento (Algoritmo 7) são as das linhas **14** e **33** dos respectivos pseudo-códigos.

Na linha **14** do Algoritmo 7, uma vez que D_{in} é igual a zero, o *warping* não é requerido e o valor de profundidade compensado pode ser armazenado em `displCompensada`. A tabela `lookupTableDispl` é um arranjo unidimensional com 255 entradas, onde `lookupTableDispl[i]` representa o valor real de profundidade associado ao valor quantizado `i`, ou seja, se `lookupTableDispl[17] = 19.6` então o *texel*, cuja distância entre a amostra associada e a face frontal da caixa envolvente é 19.6, armazena como profundidade quantizada o valor 17. Maiores detalhes sobre o cálculo de `lookupTableDispl` são dados posteriormente.

Na linha **33** é executada a etapa de reconstrução da abordagem com compensação de deslocamento, descrita pelo pseudo-código do Algoritmo 8.

algoritmo `reconstrucao2`(v, $step_u$, u_{prev}, u_{next}, v_{prev}, v_{next}, N_{prev}, D_{prev}, C_{prev}, N_{in}, D_{in}, C_{in}, quad)
1 $i \leftarrow (u_{prev} + step_u)$;
2 **Para** $u_{out} \leftarrow i$; $u_{out} \neq u_{next}$; $u_{out} \leftarrow u_{out} + step_u$ **faça**
3 quad.mNormalI$[v, u_{out}] \leftarrow$ `lerp`$(\{N_{prev}, D_{prev}\}, \{N_{in}, D_{in}\})$;
4 quad.mCorI$[v, u_{out}] \leftarrow$ `lerp`(C_{prev}, C_{in});
5 displCompensada$[v, u_{out}] \leftarrow$ `Eq_3_7`(v_{prev}, v, D_{in});
6 **fim_para**
fim

Algoritmo 8: Pseudo-código para o estágio de reconstrução do passo horizontal do *warping* com compensação de deslocamento, onde `lerp`(a, b) realiza a interpolação linear entre os valores a e b.

Dessa maneira, no decorrer do primeiro passo do estágio de pré *warping*, o valor compensado de profundidade é computado utilizando-se a Equação (3-7) e armazenado em `displCompensada` para o *texel* interpolado. Durante o segundo passo, as coordenadas da linha final, para onde o *texel* corrente deverá deslocar-se, são obtidas utilizando-se a Equação (3-6) em conjunção com os valores armazenados em `displCompensada`.

3.4.4
Amostragem Intercalada Realizada em Um Passo

Auto-oclusão é causada por mapeamentos bidimensionais não-injetores. Tais mapeamentos podem fazer com que múltiplas amostras sejam mapeadas para o mesmo *pixel* sobre a image resultante. Desse modo, durante o estágio de pré *warping*, se um cuidado apropriado não é tomado, amostras poderiam ser sobrescritas durante o primeiro passo, não estando disponíveis para o segundo passo (Figura 3.4, página 51).

Para resolver este problema, Oliveira [26] sugere uma solução capaz de manipular um número arbitrário de auto-oclusões e que não requer comparação de valores de profundidade. Tal solução consiste na intercalação dos passos horizontal e vertical do pré *warping*.

A etapa de amostragem não requer nenhuma consideração adicional e pode ser realizada como descrito no Algoritmo 7, a menos das linhas **14**, que neste caso não existe, e **33**, que representa a etapa de reconstrução onde a intercalação de fato ocorre.

À medida que cada *texel* intermediário é produzido no passo horizontal, este é imediatamente interpolado e deslocado para a linha apropriada. Tal procedimento é descrito com maiores detalhes pelo Algoritmo 9, na página 59.

Uma vez que cada iteração vertical recebe e processa o seus *texels* na ordem compatível de oclusão, a visibilidade correta é preservada na imagem resultante. Além disso, como cada *texel* é processado imediatamente após a sua criação, nenhuma informação é sobrescrita e auto-oclusões são evitadas.

3.5
Framework de Amostragem e Reconstrução

As seções anteriores trataram da etapa de pré *warping* do algoritmo de mapeamento de textura com relevo. Nesta seção, é estudado o *framework* completo para tal algoritmo e, conseqüentemente, as etapas restantes (Figura 3.1, página 46) são apresentadas com mais detalhes.

Por questões de simplicidade, sem perda de generalidade, será utilizada para realização do pré *warping* a amostragem unidimensional realizada

```
algoritmo reconstrucao3(step_u, step_v, u_prev, u_next, v_prev, v_next,
N_prev, D_prev, C_prev, N_in, D_in, C_in, quad)
    1  i ← (u_prev + step_u);
    2  Para u_out ← i; u_out ≠ u_next; u_out ← u_out + step_u faça
    3      N_out ← lerp(N_prev, N_in);
    4      C_out ← lerp(C_prev, C_in);
    5      v_out ← lerp(v_prev, v_next);
    6      i ← (V[u_out] + step_v);
    7      Para v_final ← i; v_final ≠ v_next; v_final ← v_final + step_v faça
    8          quad.mNormalP[v_final, u_out] ← lerp(N[u_out], N_out);
    9          quad.mCorP[v_final, u_out] ← lerp(C[u_out], C_out);
   10     fim_para
   11     N[u_out] ← N_out;
   12     C[u_out] ← C_out;
   13     V[u_out] ← v_out;
   14 fim_para
fim
```

Algoritmo 9: Pseudo-código para o estágio de reconstrução da abordagem intercalada. N[u_{out}], C[u_{out}] e V[u_{out}] são a normal, a cor e a coordenada da linha da última amostra deslocada para a coluna u_{out}, respectivamente.

em dois passos (Seção 3.4.1). Os *frameworks* utilizados em conjunção com as demais abordagens podem ser deduzidos a partir do procedimento apresentado nesta seção.

Antes que o algoritmo de mapeamento de textura com relevo seja efetuado é necessária a execução de algumas operações. Tais operações incluem alocação de memória, inicialização de variáveis e instanciação de um quadrilátero associado a uma textura com relevo. Na instanciação do quadrilátero em questão, os parâmetros do modelo de câmera K_s são computados baseados na posição, orientação e dimensão deste e armazenados para uso posterior como descrito no Algoritmo 10.

```
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‖;
fim
```

Algoritmo 10: Pseudo-código para instanciação do quadrilátero `quad`.

No algoritmo de mapeamento de textura com relevo propriamente

dito, é necessário reinicializar os recursos utilizados durante a execução do mesmo. Tais recursos são basicamente *buffers* de imagens e *lookup tables*. O pseudo-código apresentado no Algoritmo 11 realiza a reinicialização requerida para a amostragem unidimensional.

```
algoritmo reinicializacao(quad, texRel)
  1 Para v ← 0; v ≤ texRel.mCor.altura; v ← v + 1 faça
  2   Para u ← 0; u ≤ texRel.mCor.largura; u ← u + 1 faça
  3     atribui 255 para profundidade de quad.mNormalI[v, u];
  4     atribui 255 para profundidade de quad.mNormalP[v, u];
  5   fim_para
  6 fim_para
fim
```

Algoritmo 11: Reinicialização dos recursos do algoritmo de mapeamento de textura com relevo, onde `mNormalI` e `mNormalP` representam os mapas de normais no espaço de imagem intermediário e transformado, respectivamente.

Tal reinicialização é necessária para evitar que o valor de profundidade corrente, obtido pelo processo de leitura do *buffer* `mNormalI` durante o passo vertical, não contenha informação escrita durante amostragens anteriores. O mesmo é válido para o *buffer* `mNormalP` durante a etapa de mapeamento de textura.

Uma vez que a etapa de reinicialização tenha sido concluída, os parâmetros do modelo de câmera K_s do quadrilátero `quad` devem ser transformados de acordo com a configuração corrente de visualização. Os valores transformados são então utilizados para instanciar a textura com relevo associada. Todo o processo é descrito no Algoritmo 12, na página 61.

Como discutido na Seção 3.3, os valores de profundidade armazenados são valores quantizados. Uma vez que somente 255 valores diferentes são utilizados, poderia ser vantajoso inicializar *lookup tables* para evitar cálculos repetitivos bem como as divisões por *texel*. O Algoritmo 13, na página 61, apresenta um pseudo-código para inicializar tais tabelas durante o pré *warping*. Os índices dos valores quantizados são utilizados para indexar as tabelas.

Nas linhas **1** e **2** são calculadas as coordenadas do ponto epipolar. A partir deste ponto são calculados os coeficientes `k1`, `k2` e `k3` que representam a informação de câmera associada ao processo. Então, em cada iteração do laço da linha **6**, é calculado o valor de profundidade aproximado para um dado índice quantizado `i` bem como são armazenados em *lookup tables* os

```
algoritmo configCamera(quad, cam)
    1 trans ← componente de translação de cam.modeloVisao;
    2 rot ← {cam.modeloVisao} − {trans};
    3 quad.C ← (rot)(quad.C);
    4 tCOP ← (rot)(cam.posicao);
    5 tCOP ← tCOP + trans;
    6 quad.c ← quad.C - tCOP;
    7 quad.a ← (rot)(quad.a);
    8 quad.b ← (rot)(quad.b);
    9 quad.f ← (quad.a × quad.b)/‖quad.a × quad.b‖;
fim
```

Algoritmo 12: Configuração do modelo de câmera associado ao quadrilátero `quad`. `trans` e `rot` representam as componentes de translação e rotação da matriz `modeloVisao`, respectivamente. `tCOP` representa o centro de projeção da imagem destino de acordo com a configuração corrente de visualização.

```
algoritmo inicLookupTables(quad)
    1 e[0] ← (−quad.c · quad.a)/(quad.a · quad.a);
    2 e[1] ← altura − (−quad.c · quad.b)/(quad.b · quad.b);
    3 k3 ← 1/(quad.c · quad.f);
    4 k1 ← e[0]k3;
    5 k2 ← e[1]k3;
    6 Para i ← 0; i < 255; i ← i + 1 faça
    7     d ← (i)(qs)/255;
    8     rTable[i] ← (k1)(d);
    9     sTable[i] ← (k2)(d);
   10     tTable[i] ← 1/(1 + (k3)(d));
   11 fim_para
fim
```

Algoritmo 13: Inicialização de *lookup tables*. A variável `e` armazena as coordenadas do ponto epipolar. `k1`, `k2` e `k3` representam a informação de câmera associada ao processo. `d` armazena o valor de profundidade aproximado para um dado índice quantizado `i`. `rTable`, `sTable` e `tTable` armazenam os numeradores e o recíproco do denominador das equações de pré *warping* (2-14) e (2-15).

numeradores e o recíproco do denominador das equações de pré *warping* (2-14) e (2-15). `qs` é o passo da quantização e representa a distância entre as faces frontal e posterior, da caixa envolvente, utilizadas no processo de aquisição da textura com relevo.

Desse modo, o pré *warping* de um *texel* pode ser obtido com duas adições e duas multiplicações, da seguinte maneira:

Uma vez que se tenha as coordenadas do ponto epipolar, a textura

$u_{next} = (u +$ `rTable[`D_{in}`])*tTable[`D_{in}`];`
$v_{next} = (v +$ `sTable[`D_{in}`])*tTable[`D_{in}`];`

com relevo `texRel` pode ser dividida em regiões e o processo de *warping* pode ser aplicado a cada uma destas regiões. O Algoritmo 14, na página 63, apresenta um pseudo-código que realiza tal procedimento.

Nas linhas **3** a **7** é realizado o *warping* horizontal da região superior esquerda de `texRel` enquanto que nas linhas **8** a **11**, **12** a **16** e **17** a **20** é realizado o *warping* horizontal das regiões inferior esquerda, superior direita e inferior direita de `texRel`, respectivamente. O mesmo procede para as linhas **21** a **38**, só que neste caso é efetutado o *warping* vertical.

Finalmente, com o término do processo de *warping*, a textura resultante pode ser mapeada sobre o quadrilátero `quad` de acordo com a rotina `mapeiaTextura`, na página 64.

Na primeira vez que a função `mapeiaTextura` é executada, os *buffers* `mCorP` e `mNormalP`, no espaço de imagem transformado, são transferidos para a memória de textura da placa de vídeo através da API *OpenGL* [39]. Uma vez que tais *buffers* são transferidos, as outras execuções de `mapeiaTextura` são responsáveis por atualizar o conteúdo da memória de textura fazendo uso, novamente, da API *OpenGL*.

Desse modo, o procedimento completo para realização do mapeamento de textura com relevo é descrito pelo Algoritmo 15, na página 64.

```
algoritmo amostragem1D(quad, texRel)
 1 largura ← (texRel.mCor.largura - 1);
 2 altura ← (texRel.mCor.altura - 1);
 3 min_u ← 0;   max_u ← MIN(e[0],largura);
 4 min_v ← 0;   max_v ← MIN(e[1],altura);
 5 Se min_u < max_u e min_v < max_v então
 6    warpHorizontal(min_u,min_v,max_u,max_v,+1,+1,quad,texRel);
 7 fim_se
 8 min_v ← MAX(e[1]-1,0);   max_v ← altura;
 9 Se min_u < max_u e min_v < max_v então
10    warpHorizontal(min_u,min_v,max_u,max_v,+1,−1,quad,texRel);
11 fim_se
12 min_u ← MAX(e[0]-1,0);   max_u ← largura);
13 min_v ← 0;   max_v ← MIN(e[1],altura);
14 Se min_u < max_u e min_v < max_v então
15    warpHorizontal(min_u,min_v,max_u,max_v,−1,+1,quad,texRel);
16 fim_se
17 min_v ← MAX(e[1]-1,0);   max_v ← altura;
18 Se min_u < max_u e min_v < max_v então
19    warpHorizontal(min_u,min_v,max_u,max_v,−1,−1,quad,texRel);
20 fim_se
21 min_u ← 0;   max_u ← MIN(e[0],largura);
22 min_v ← 0;   max_v ← MIN(e[1],altura);
23 Se min_u < max_u e min_v < max_v então
24    warpVertical(min_u,min_v,max_u,max_v,+1,+1,quad,texRel);
25 fim_se
26 min_v ← MAX(e[1]-1,0);   max_v ← altura;
27 Se min_u < max_u e min_v < max_v então
28    warpVertical(min_u,min_v,max_u,max_v,+1,−1,quad,texRel);
29 fim_se
30 min_u ← MAX(e[0]-1,0);   max_u ← largura);
31 min_v ← 0;   max_v ← MIN(e[1],altura);
32 Se min_u < max_u e min_v < max_v então
33    warpVertical(min_u,min_v,max_u,max_v,−1,+1,quad,texRel);
34 fim_se
35 min_v ← MAX(e[1]-1,0);   max_v ← altura;
36 Se min_u < max_u e min_v < max_v então
37    warpVertical(min_u,min_v,max_u,max_v,−1,−1,quad,texRel);
38 fim_se
fim
```

Algoritmo 14: Pseudo-código para amostragem 1D realizada em dois passos onde MIN(a,b) e MAX(a,b) retornam o menor e o maior valor dentre a e b, respectivamente.

```
void mapeiaTextura(Quadrilatero quad)
{
  static s_primeiraVez = true;
  if (s_primeiraVez)
  {
    glGenTextures(1,&corId);
    glBindTexture(GL_TEXTURE_2D,corId);
    glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,quad.mCorP.largura,
      quad.mCorP.altura,0,GL_RGB,GL_UNSIGNED_BYTE,;
      quad.mCorP);
    glGenTextures(1,&normalId);
    glBindTexture(GL_TEXTURE_2D,normalId);
    glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,
      quad.mNormalP.largura,quad.mNormalP.altura,0,GL_RGBA,
      GL_UNSIGNED_BYTE,quad.mNormalP);
    s_primeiraVez = false;
  }
  else
  {
    glBindTexture(GL_TEXTURE_2D,corId);
    glTexSubImage2D(GL_TEXTURE_2D,0,0,0,quad.mCorP.largura,
      quad.mCorP.altura,GL_RGB,GL_UNSIGNED_BYTE,quad.mCorP);
    glBindTexture(GL_TEXTURE_2D,normalId);
    glTexSubImage2D(GL_TEXTURE_2D,0,0,0,
      quad.mNormalP.largura,quad.mNormalP.altura,GL_RGBA,
      GL_UNSIGNED_BYTE,quad.mNormalP);
  }
  glBegin(GL_QUADS);
    glTexCoord2f(0.0f,0.0f); glVertex3fv(quad.v0);
    glTexCoord2f(1.0f,0.0f); glVertex3fv(quad.v1);
    glTexCoord2f(1.0f,1.0f); glVertex3fv(quad.v2);
    glTexCoord2f(0.0f,1.0f); glVertex3fv(quad.v3);
  glEnd();
}
```

```
algoritmo mapeiaTexturaRelevo(quad, texRel, cam)
  1 reinicializacao(quad,texRel);
  2 configCamera(quad,cam);
  3 inicLookupTables(quad);
  4 amostragem1D(quad,texRel);
  5 mapeiaTextura(quad);
fim
```

Algoritmo 15: Pseudo-código para o mapeamento de textura com relevo.

3.6
Cálculo de Iluminação por Pixel

O uso de mapas de normais em conjunção com mapeamento de textura com relevo permite que o cálculo de iluminação seja efetuado por *pixel*. Para realização deste cálculo, adotou-se a linguagem de *shading* Cg [16] da NVidia.

Cg é uma linguagem de programação de alto nível que permite ao programador a capacidade de definir o comportamento de certas partes do *pipeline* gráfico, por isso o termo *pipeline programável*. Segundo Fernando & Kilgard [16], a tendência do *hardware* gráfico é adquirir cada vez mais unidades programáveis. Atualmente, este *hardware* disponibiliza duas unidades deste tipo: o *processador de vértices* e o *processador de fragmentos*.

O processador de vértices recebe como entrada os atributos de cada vértice da cena 3D que se quer renderizar. Tais atributos são geralmente posição, cor, coordenada de textura, entre outros. O processador de vértices então busca a próxima instrução, pertencente ao programa de vértice definido pelo usuário, e a executa. Este procedimento é repetido até o término do programa.

Quando o programa de vértice termina, o vértice processado é enviado para o estágio de *confecção de primitivas e rasterização* onde ocorre a conversão de vértices em primitivas seguida da conversão destas primitivas em fragmentos[3]. Dessa maneira, o fragmento é o dado de entrada para o processador de fragmentos que funciona de modo semelhante ao processador de vértices.

As seções seguintes tratam dos programas de vértice e de fragmento que realizam o cálculo de iluminação por *pixel* e o mapeamento de textura com relevo.

3.6.1
Sistemas de Coordenadas do Pipeline Gráfico

Para facilitar o entendimento dos programas de vértice e fragmento, e por questões de notação, segue abaixo um diagrama ilustrativo dos sistemas

[3]Entidade gerada pela rasterização de uma primitiva e candidata a se tornar um *pixel* na imagem final.

de coordenadas e das transformações pelas quais os vértices são processados através do *pipeline* gráfico. Mais detalhes sobre o assunto podem ser obtidos em [16].

Figura 3.5: Sistemas de coordenadas e transformações pelas quais os vértices são processados através do *pipeline* gráfico. (*Imagem adaptada de [16].*)

Como discutido anteriormente, é desejável que uma textura com relevo possa ser reutilizada para adicionar detalhes a vários quadriláteros. Para que isto seja possível, é necessário desacoplar a informação de câmera, que define o posicionamento e a orientação da superfície subjacente no mundo, da textura com relevo. No entanto, o mapa de normal, que continua associado à estrutura de dados `TexturaRelevo`, é dependente da orientação original da superfície subjacente representada. Dessa forma, para os casos em que a textura com relevo é mapeada sobre quadriláteros de orientação não compatível com a orientação original da superfície subjacente, o cálculo de iluminação fornecerá resultados errôneos. Isto ocorre porque o *vetor de luz* e o *vetor half-angle*, utilizados no cálculo de iluminação por *pixel*, não mais compartilham um sistema de coordenadas consistente com as normais no mapa de normal, impedindo a reutilização da textura com relevo.

3.6.1.1

Iluminação no Espaço da Textura

Uma solução é transformar os vetores de luz e *half-angle* para o sistema de coordenadas do mapa de normal. Tal sistema de coordenadas é conhecido como *espaço da textura* (algumas vezes denotado *espaço tangente*). A transformação que converte um vetor no espaço do objeto para o espaço da textura exige o cálculo de dois vetores conhecidos como *vetor tangente* e *vetor binormal*. Para maiores detalhes desta transformação consultar [16].

3.6.2

O programa de Vértice

O programa de vértice essencialmente transforma os vetores utilizados no cálculo de iluminação para o espaço da textura, como descrito no Algoritmo 16.

```
void vertexProg(float4 position : POSITION,
                float3 normal : NORMAL,
                float2 texCoord : TEXCOORD0,
                float3 tangent : TEXCOORD1,
                float3 binormal : TEXCOORD2,
            out float4 oPosition : POSITION,
            out float2 oTexCoord : TEXCOORD0,
            out float3 oLightDirection : TEXCOORD1,
            out float3 oEyeDirection : TEXCOORD2,
        uniform float3 lightPosition
        uniform float3 eyePosition
        uniform float4x4 modelViewProj)
{
  oPosition = mul(modelViewProj,position);
  oTexCoord = texCoord;
  oLightDirection = lightPosition - position;
  oEyeDirection = eyePosition - position;
  float3x3 rot = float3x3(tangent,binormal,normal);
  oLightDirection = mul(rot,oLightDirection);
  oEyeDirection = mul(rot,oEyeDirection);
}
```

Algoritmo 16: Programa de vértice para o cálculo de iluminação por *pixel*.

O programa `vertexProg` recebe como entrada oito parâmetros: `position`, `normal`, `texCoord`, `tangent`, `binormal`, `lightPosition`, `eyePosition` e `modelViewProj`. Os dois primeiros parâmetros representam, respectivamente, a posição e o vetor normal unitário associados ao vértice atual, ambos no espaço do objeto. O terceiro parâmetro é o atributo do vértice que armazena as coordenadas de textura associadas ao mesmo. Os parâmetros `tangent` e `binormal` são vetores unitários no espaço do objeto que juntamente com o vetor `normal` formam uma base para o espaço da textura. Finalmente, os três últimos parâmetros, que são globais para todos os vértices (por isso a palavra chave `uniform`), representam, respectivamente, a posição da fonte de luz no espaço do objeto, a posição

do observador no espaço do objeto e a matriz *ModeloVisão* concatenada com a *matriz de projeção*.

O programa então multiplica o vértice corrente pela matriz `modelViewProj` transportando-o para o espaço de *clip* para que posteriormente o mesmo possa ser rasterizado. O resultado desta operação é armazenado no parâmetro de saída `oPosition`. Em seguida, as coordenadas de textura são armazenadas em `oTexCoord` para uso posterior no programa de fragmento. Logo após, são calculados os vetores no espaço do objeto que representam a direção da luz e a direção do observador em relação ao vértice em questão. No entanto, é necessário transformar estes vetores para o espaço da textura, e a matriz `rot` armazena tal transformação. No que se segue, os vetores `oLightDirection` e `oEyeDirection` são transportados para o espaço da textura para posteriormente serem utilizados no programa de fragmento.

3.6.3
O programa de Fragmento

A cálculo de iluminação por *pixel* é efetivamente realizado no programa de fragmento, como mostra o Algoritmo 17, na página 69.

Os três primeiros parâmetros deste algoritmo são as coordenadas de textura, a direção da luz no espaço de textura e a direção do observador no espaço de textura que foram retornados pelo programa de vértice e posteriormente interpolados pelo *pipeline* gráfico. O parâmetro `oColor` é o único parâmetro de saída do programa e armazena a cor final do fragmento. Os dois próximos parâmetros definem o modelo de luz da cena corrente. Os parâmetros `ka`, `ks`, `ke` e `shininess` são, respectivamente, a componente ambiente, a componente especular, a componente emissora e a intensidade do brilho do material associado ao objeto que se quer gerar. E, finalmente, os dois últimos parâmetros são o mapa de normal com profundidade e o mapa de textura transformados após o *warping*, respectivamente.

Inicialmente, obtém-se o *texel* corrente do mapa de normal associado às coordenadas de textura armazenadas em `texCoord`. Repare que o *texel* obtido sofre uma expansão. Esta expansão é necessária porque o conteúdo do mapa de normal é armazenado como um valor *unsigned* e, portanto, estes valores ficam restritos ao intervalo $[0, 1]$. Devido às normais serem

```
void fragmentProg(float2 texCoord : TEXCOORD0,
                  float3 lightDirection : TEXCOORD1,
                  float3 eyeDirection : TEXCOORD2,
              out float3 oColor : COLOR,
          uniform float3 globalAmbient,
          uniform float3 lightColor,
          uniform float3 ka,
          uniform float3 ks,
          uniform float3 ke,
          uniform float shininess,
          uniform sampler2D normalMap,
          uniform sampler2D textureMap)
{
  float4 normalTexel = expand(tex2D(normalMap,texCoord));
  if (normalTexel.w == 1.0)
    discard;
  else
  {
    float3 textureTexel = tex2D(textureMap,texCoord);
    float3 L = normalize(lightDirection);
    float3 N = normalize(normalTexel.xyz);
    float3 H = normalize(L + normalize(eyeDirection));
    TMaterial mat;
    mat.ka = ka;
    mat.kd = textureTexel;
    mat.ks = ks;
    mat.ke = ke;
    mat.shininess = shininess;
    oColor.xyz = lighting(mat,lightColor,globalAmbient,L,
      N,H);
    oColor.w = 1.0;
  }
}
```

Algoritmo 17: Programa de fragmento para o cálculo de iluminação por *pixel*.

vetores normalizados, cada componente deveria ter um intervalo $[-1, 1]$. Para permitir que o programa de fragmento opere corretamente, normais são expandidas da seguinte maneira:

```
float4 expand(float4 normal)
{
  return (normal - 0.5)*2.0;
}
```

É importante observar que para que esta expansão funcione é

necessário que o dado de normal, tido como entrada para o algoritmo de amostragem a partir de texturas com relevo, sofra uma contração antes de ser armazenado no *buffer* de entrada [16].

Com o *texel* corrente do mapa de normal em mãos é possível testar a viabilidade do seu valor de profundidade. Lembre-se que um valor de profundidade é inválido se, e somente se, é igual a 255 (Seção 3.3) no intervalo $[0, 255]$. Dessa maneira, no caso do programa de fragmento, um valor de profundidade é inválido quando este é igual a 1 no intervalo $[-1, 1]$. Portanto, o comando `if` do Algoritmo 17 nada mais é do que uma verificação da viabilidade do valor de profundidade corrente. Se este é inválido, o fragmento associado é descartado; do contrário, o cálculo de iluminação pode ser efetuado.

Uma vez que a viabilidade do valor de profundidade é certificada, o *texel* corrente do mapa de textura é obtido da mesma forma que no mapa de normal. Este *texel* é então posteriormente utilizado como a componente difusa do material durante o cálculo de iluminação propriamente dito. O restante do programa realiza as operações necessárias para um cálculo de iluminação convencional (função `lighting`) e, portanto, não requer nenhuma consideração adicional. Maiores detalhes podem ser obtidos no Capítulo 5 de [16].

3.7 Discussão

Durante o pré *warping*, amostras poderiam ser mapeadas além dos limites do plano de suporte da imagem fonte, fazendo com que a imagem resultante produzisse uma visão incompleta da superfície representada (Figura 3.6).

A ocorrência de tais situações depende da configuração de visualização e do tamanho dos deslocamentos dos *pixels*. Para solucionar o problema Oliveira [26] sugere o uso de seis texturas com relevo, cada uma associada a uma das faces da caixa envolvente do objeto a ser representado. Novas visões podem ser obtidas transformando-se as texturas com relevo de acordo com o *warp* e mapeando as imagens resultantes sobre as faces da caixa. Entretanto, somente realizar o *warping* de cada textura sobre a corresponde face da caixa não é o bastante para produzir o resultado desejado.

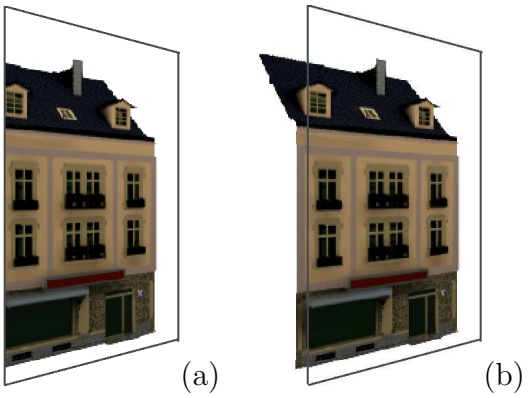

Figura 3.6: Captura de amostras além do limite do plano de suporte da imagem fonte. Uma visão incompleta da superfície é obtida (a) se amostras mapeadas para fora do plano da imagem fonte não forem consideradas (b). (*Imagem obtida em [26].*)

A solução é, além da face corrente processar também suas faces adjacentes. A Figura 3.7 ilustra uma divisão do espaço do objeto dentro de regiões numeradas. O Algoritmo 18, na página 72, apresenta um pseudo-código para renderizar esta nova representação.

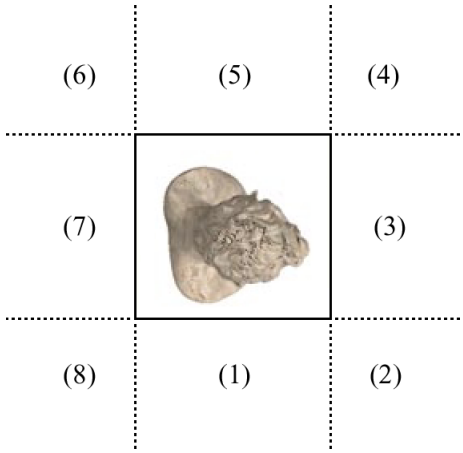

Figura 3.7: Divisão do espaço do objeto dentro de regiões numeradas. Vista de cima de um objeto envolvido por uma caixa. (*Imagem adaptada de [26].*)

De acordo com o Algoritmo 18, o modo como a renderização ocorre depende da posição da câmera em relação à caixa envolvente do objeto a ser visualizado. Por exemplo, se a câmera se encontra numa região ímpar, conforme a Figura 3.7, as texturas associadas às faces esquerda, direita e frontal são transformadas pela operação de *warping* e os respectivos resultados são mapeados sobre a face frontal.

A alteração da estrutura topológica da imagem após a realização da amostragem unidimensional é a causa de erros de interpolação de cor

```
algoritmo renderizaCaixa(face[6], texRel[6], cam)
   1 Se cam.position ∈ região ímpar então
   2     pré warp texRel[esquerda] para face[frente];
   3     pré warp texRel[direita] para face[frente];
   4     pré warp texRel[frente] para face[frente];
   5 senão
   6     pré warp texRel[esquerda] para face[direita];
   7     pré warp texRel[direita] para face[direita];
   8     pré warp texRel[direita] para face[esquerda];
   9     pré warp texRel[esquerda] para face[esquerda];
  10 fim_se
fim
```

Algoritmo 18: Pseudo-código para renderizar seis texturas com relevo, cada uma associada a uma das faces da caixa envolvente do objeto a ser representado.

(Seção 3.4.1.1). Embora a topologia da imagem possa ser preservada com o uso de estruturas de dados adicionais, a introdução desta complexidade extra poderia acarretar na perda de algumas vantagens resultantes da simplicidade de tal abordagem.

Para finalizar a discussão sobre o assunto, vale notar que na Equação (3-7), onde calcula-se um valor de profundidade compensado a fim de evitar ruídos causados por distorção não-linear, a maioria dos termos podem ser reutilizados, tais como $k_2 displ_B$, $k_3 displ_B$ e $(1 + k_3 displ_B)$. Além disso, uma vez que estes termos contêm um valor de profundidade agregado parece vantajoso armazená-los em *lookup tables*.