3
Descrição do Algoritmo

Neste capítulo vamos descrever as idéias principais do algoritmo proposto por Lindstrom & Pascucci e estudado neste trabalho.

Em síntese, o objetivo deste algoritmo é gerar uma malha com o menor número possível de triângulos e que ao mesmo tempo seja uma boa aproximação para a malha completa. O algoritmo trata com um terreno representado por amostras regularmente espaçadas, ou seja, dispostas no formato de uma grade.

Uma restrição é que esta grade de pontos deve necessariamente formar uma matriz bidimensional com $(2^{n/2} +1)$ vértices em cada direção, onde *n* é o número de níveis de refinamento, necessariamente maior ou igual a 2. Desta forma, quando o terreno não se encaixa neste formato é preciso estendê-lo fazendo a alocação de uma área de memória às vezes bem maior que o tamanho dos dados do próprio terreno. Porém, depois de executado o refinamento, as áreas sem dados reais podem ser cobertas com uma textura transparente e não haverá nenhum prejuízo mais sério que algum relativo desperdício de memória e de processamento.

Em primeiro lugar, destacamos que se trata de um algoritmo dependente da visão, o que significa dizer que a malha de triângulos gerada dependerá do ponto onde o observador estiver localizado. Afinal de contas, a cada quadro é feita a verificação de quais são os vértices ativos, isto é, que deverão estar presentes na malha final.

Esta verificação é feita projetando na tela um erro no espaço do objeto, armazenado em tempo de pré-processamento, no próprio vértice. Somente se este erro projetado for maior que uma tolerância previamente especificada é que o vértice deverá ser ativado. O erro no espaço do objeto de um vértice *i*, é, em princípio, a diferença vertical entre a altura real de *i* e altura aproximada no ponto correspondente a *i* quando *i* não está presente na malha.

Destacamos ainda que se trata de um algoritmo de refinamento. Ele inicia com uma malha-base grosseira, em geral contendo quatro triângulos. Em seguida,

PUC-Rio - Certificação Digital Nº 0115656/CA

estes triângulos vão sendo divididos, conforme o critério de refinamento e de modo que a malha vá se adaptando ao terreno. Assim, a decisão de dividir ou não um triângulo dependerá das posições do observador e do próprio triângulo.

A forma de executar a divisão de um triângulo não é inovadora. Ela é feita através do esquema de partição da maior aresta. Todos os triângulos aqui são retângulos e isósceles. Quando um vértice é inserido na metade da hipotenusa de um triângulo, dois outros triângulos, também isósceles, são criados. Note que isto já sugere algum procedimento recursivo para fazer tal divisão dos triângulos.

Além disso, note que a malha produzida por este esquema pode ser representada por um grafo acíclico direcionado (DAG – sigla que vem do inglês *directed acyclic graph*) de vértices. Neste grafo, convencionamos que uma aresta (*i, j*) partindo do vértice *i* em direção ao vértice *j* indica que *i* é pai de *j*. Além disso, a aresta (*i, j*) partindo de *i* para um dos seus filhos *j* corresponde à partição de um triângulo onde *j* é inserido na metade da hipotenusa deste triângulo, conforme mostrado na Figura 7.

Deste modo, repartindo recursivamente a malha-base grosseira, criaremos uma malha onde todos os vértices que não são folhas e nem estão localizados na fronteira possuem exatamente quatro filhos e dois pais. Enquanto isto, os vértices localizados na fronteira da malha possuem apenas dois filhos e um pai. Finalmente, os vértices-folhas são aqueles que não possuem filhos.

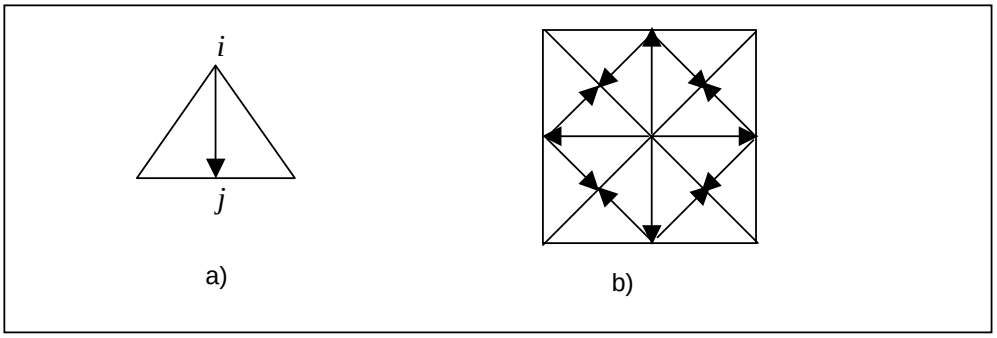


Figura 7 - a) Aresta de *i* para *j* no DAG; b) Destacando arestas no DAG de uma malha.

Outro ponto importante identificado verificando-se a natureza desta malha é que se um vértice for inserido e seu pai não, isto gerará um vértice T indesejável e uma possível fenda na malha final. Então, para assegurar uma malha válida, a seguinte propriedade deve ser mantida: se um vértice está ativo, isto é, presente na

PUC-Rio - Certificação Digital Nº 0115656/CA

malha final, então os seus pais (e por indução todos os seus ancestrais) também estarão.

O problema é que neste grafo direcionado acíclico (DAG) pode-se alcançar um vértice filho sem, obrigatoriamente, passar por todos os seus ancestrais. Esquemas de dependências explícitas entre vértices têm sido propostos para resolver este problema. Todavia, tal abordagem é considerada computacionalmente ineficiente e cara do ponto de vista de armazenamento.

Em vez disto, uma forma mais elegante e eficiente é a utilização de esquemas implícitos de dependências entre vértices. Desta forma, uma importante contribuição de Lindstrom e Pascucci [2] é que eles propõem uma maneira razoavelmente simples, sem a exigência de um pré-processamento pesado e embutida no próprio critério de refinamento. A seguir descrevemos como isto pode ser feito, mas já adiantamos que um aninhamento dos erros é utilizado.

PUC-Rio - Certificação Digital Nº 0115656/CA

3.1. Refinamento

O objetivo do refinamento realizado aqui é seguir dividindo a malha-base de forma a ir ajustando-a ao terreno e ao mesmo tempo garantir uma malha contínua. Para isto, Lindstrom & Pascucci criaram um esquema de refinamento independente da métrica de erro utilizada e baseado em esferas aninhadas.

Já vimos que uma malha contínua e sem vértices T é obtida quando nenhum filho é introduzido sem que seus pais e seus ancestrais também o sejam. Além do mais, poderíamos assegurar isto se conseguíssemos aninhar os erros no espaço da tela, ou seja, assegurando que o erro de um vértice-pai seja maior ou igual ao erro de todos os seus descendentes. Afinal, um vértice estará ativo ou inativo dependendo do valor do seu erro no espaço da tela comparado à tolerância especificada.

Portanto, o primeiro passo na tentativa de, em última análise, garantir uma malha contínua consiste em propagar os erros no espaço do objeto dos filhos para os pais (e todos os seus ancestrais). Seja δ_i o erro no espaço do objeto do vértice i, ou seja, a diferença vertical entre a altura real de i e a altura aproximada no ponto correspondente a i quando i não está presente na malha. Assim, considerando que

este erro já tenha sido calculado e armazenado para todos os vértices da malha, poderemos fazer a propagação de erros do seguinte modo:

$$\delta_i = \begin{cases} \delta_i, \text{ se } i \text{ é um vértice-folha} \\ \max(\delta_i, \delta_j), \text{ para todo } j \text{ descendente de } i \end{cases}$$

Deste modo, já temos os erros no espaço do objeto aninhados. Porém, esta condição ainda não é suficiente, pois ainda não conseguimos aninhar os erros no espaço da tela, conforme desejado. Observe que um filho pode estar arbitrariamente próximo do observador e seu pai arbitrariamente longe, de modo que, ao serem projetados na tela, o erro do pai ainda será menor que o do filho. Isto poderá implicar o filho ser ativado sem a ativação do pai e, portanto, gerar o indesejável vértice T.

Então, ainda é necessário definir uma esfera envolvente para cada vértice da malha, englobando o próprio vértice e todos os seus descendentes. Sendo r_i o raio da esfera centrada no vértice i, teremos que:

$$r_i = \begin{cases} 0, \text{ se } i \text{ for um vértice-folha} \\ \max(\| p_i - p_j \| + r_j), \text{ para todo } j \text{ descendente de } i \end{cases}$$

A Figura 8 mostra, no plano, estas esferas envolventes para um pequeno trecho de malha. Observe que, para deixar a figura mais simples e clara, as esferas foram representadas por círculos.

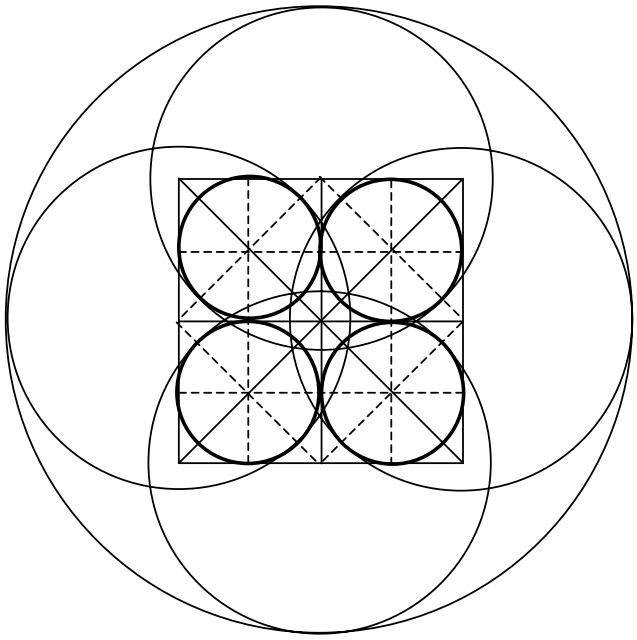


Figura 8 - Malha com esferas envolventes

PUC-Rio - Certificação Digital Nº 0115656/CA

Agora, definimos o erro no espaço da tela do vértice *i* como proporcional ao seu erro no espaço do objeto e inversamente proporcional à distância até o ponto de vista do observador subtraída do raio da esfera envolvente de *i*. Assim, finalmente temos que o erro no espaço da tela do vértice *i* será maior ou igual ao erro de *j*, para todo *j* descendente de *i* (pois o raio de *i* é maior que o raio de *j*). Conseqüentemente, se *j* estiver ativo, então todos os seus ancestrais também estarão e garantimos uma malha final contínua, conforme desejado.

Vale ressaltar que tanto o raio da esfera envolvente quanto o erro no espaço do objeto de cada vértice devem ser calculados e propagados adequadamente na malha, tal como descrito anteriormente, em tempo de pré-processamento. Além, é claro, dos dados do terreno, estes são os únicos termos necessários para executar o procedimento de refinamento recursivo, conforme será mostrado mais adiante. O pseudo-código para o pré-processamento também será apresentado mais adiante.

PUC-Rio - Certificação Digital Nº 0115656/CA

3.2. Métricas de Erro

A escolha de qual métrica de erro utilizar, em princípio, é ortogonal ao algoritmo de refinamento que vem sendo descrito aqui. A condição é que a posição do erro no espaço do objeto possa ser incluída em uma esfera envolvente. Além disso, deve-se decidir antecipadamente se será utilizado um erro medido em termos absolutos ou relativos, pois os erros precisam ser propagados em tempo de pré-processamento.

Então, dado um erro no espaço do objeto (δ_i), o algoritmo dependente da visão o projeta na tela (para obter ρ_i) e a forma mais simples de fazer isto é:

$$\rho_i = \frac{\lambda \delta_i}{\|p_i - o\|} \qquad \text{(eq. 1)}$$

onde $\lambda = \frac{\omega}{\varphi}$ e ω é a largura da tela em *pixels*, φ é o campo de visão ou a abertura da câmera e *o* é o ponto onde está localizado o olho do observador .

Como o objetivo é maximizar ρ_i e ρ_i será máximo quando $\|p_i - o\|$ for mínimo, então se *o* não pertence à esfera envolvente de *i*, então $\|p_i - o\| - r_i$ maximizará o erro no espaço da tela. Para *o* dentro da esfera, este termo é zero e o vértice é ativado. Desta forma, temos que:

$$\rho_i = \frac{\lambda \delta_i}{\|p_i - o\| - r_i} \qquad \text{(eq. 2)}$$

3.3. Pré-Processamento

Antes de começar a executar o refinamento em tempo real é preciso preparar os dados do terreno, colocando-os no formato exigido pelo algoritmo de refinamento. Veremos que comparativamente com outros algoritmos para visualização de terrenos, aqui o pré-processamento é relativamente simples e não muito demorado. A complexidade computacional deste pré-processamento é $O(m^2)$ para um terreno de dimensões (m+1) x (m+1), onde m é uma potência de dois, portanto, já incluindo os dados fictícios eventualmente necessários para que o terreno se encaixe no formato (2^n +1) x (2^n +1). Já vimos que cada amostra ou vértice deve armazenar consigo, além da sua altura (no caso, a coordenada z), também um raio e um erro no espaço do objeto.

Em linguagem C, podemos dizer que cada vértice pode ser representado pela seguinte estrutura:

```
Struct VerticeTrn
{
  float z;        /* altura */
  float sigma; /* erro no espaço do objeto */
  char ativo;  /* indica se o vértice estará presente na malha final atual */
  float r;        /* raio do vértice */
};
VérticeTrn **MatrizTrn;
```

Assim, uma matriz bidimensional desta estrutura (MatrizTrn neste caso) representará o terreno e as coordenadas x e y serão, implicitamente, representadas pelos índices dentro desta matriz.

As coordenadas x, y e z devem ser lidas na entrada e a variável *ativo* será atualizada durante o refinamento. Portanto, em suma, resta ao pré-processamento calcular e armazenar adequadamente o erro no espaço do objeto e o raio para cada um dos vértices do terreno.

Desta forma, o pré-processamento começa calculando o erro no espaço do objeto para cada vértice. Inicialmente, ainda não nos preocupamos com o requisito de que o erro de um vértice-pai deve ser maior ou igual ao erro de todos os seus descendentes. Vale destacar que esta etapa foi inspirada na estrutura de blocos

PUC-Rio - Certificação Digital Nº 0115656/CA

criada no trabalho feito por Lindstrom et al.[5] em 1996 e depois detalhado no capítulo 11 de [10]. Aqui não existe mais a necessidade de uma estrutura física para blocos, mas a malha final continua com o mesmo formato e não incorremos em nenhum erro ao pensarmos nela como dividida em blocos. Enfim, mais adiante ficará claro que pensar na malha como sendo constituída por blocos é apenas um artifício para facilitar esta etapa de pré-processamento.

Todavia, se estamos falando em blocos, então é necessário discutirmos alguns conceitos relacionados a eles. No trabalho citado anteriormente, o terreno inteiro é visto como um grande bloco que é formado por quatro blocos menores e recursivamente cada um destes quatro blocos também é formado por quatro outros blocos. A recursão pára quando o bloco possui dimensão três, ou seja, contém 3x3 amostras, e não faz mais sentido dividi-lo em quatro blocos. A Figura 9 ilustra o bloco-pai, envolvendo todo o terreno, e seus quatro filhos (o direito superior - DS, o esquerdo superior - ES, o direito inferior - DI e o esquerdo inferior - EI).

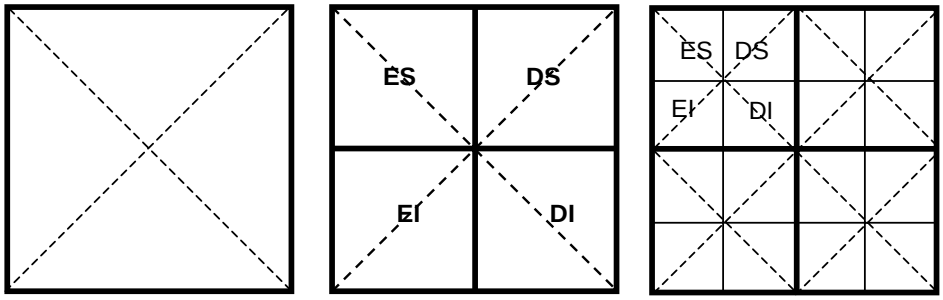


Figura 9 - Terreno dividido recursivamente em blocos.

Uma vez visto como o terreno pode ser virtualmente dividido em blocos, agora destacamos que existem dois tipos de blocos: pares e ímpares. Adiantamos que esta classificação permite identificar quem são os dois pais do vértice central de um bloco, conforme pode ser visto nas figuras 10b e 10c. Esta informação será útil mais adiante. O bloco inicial englobando o terreno inteiro e todo bloco que é filho direito superior (DS) ou esquerdo inferior (EI) é necessariamente par. Por outro lado, todo bloco que é filho esquerdo superior (ES) ou direito inferior (DI) é necessariamente ímpar. O segundo ponto é que os quatro vértices-filhos do vértice central de um bloco estão sempre posicionados conforme mostrado na figura 10a. A posição dos pais do vértice central, porém, varia dependendo de se o bloco que o engloba é par ou ímpar, conforme ilustrado nas figuras 10b e 10c. A dimensão

PUC-Rio - Certificação Digital Nº 0115656/CA

do bloco varia de acordo com a posição dele na árvore de blocos. O bloco grande, envolvendo o terreno inteiro e raiz da árvore, possui dimensão (2^n+1) x (2^n+1). Já os blocos-folhas possuem dimensão 3x3 e os blocos intermediários possuem dimensões intermediárias.

Um bloco completo contém todos os vértices, como na Figura 10. Contudo, um bloco não necessariamente precisa ser completo. O vértice central e seus 4 filhos podem eventualmente serem retirados. Repare que, quando um vértice é retirado do bloco, a altura no ponto correspondente a ele será aproximada pela média das alturas de seus dois vizinhos dentro do bloco. A Figura 11 ilustra este esquema para um bloco ímpar. Para um bloco par é análogo, só que o vértice central será eventualmente aproximado pelos vértices localizados na diagonal oposta.

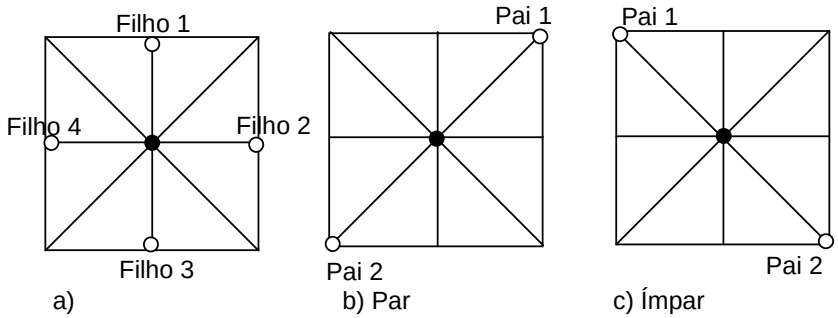


Figura 10 - Vértices de um bloco.
Em a) destaque para os quatro filhos do vértice central. Em b) e c) destaque para os pais do vértice central e para um bloco par e ímpar, respectivamente.

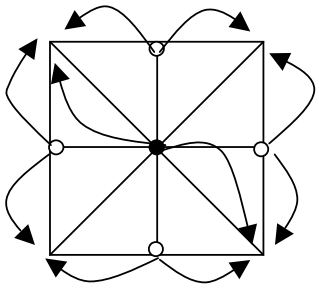


Figura 11 - Destacando os vértices que aproximam os vértices eventualmente ausentes. As setas partem do vértice que pode estar ausente e vai até os vértices que o aproximam.

Desta forma, o procedimento que calcula o erro no espaço do objeto para todos os vértices percorre o terreno recursivamente como se ele estivesse dividido em blocos. Uma vez identificado um bloco, já sabemos quais os vértices que

PUC-Rio - Certificação Digital Nº 0115656/CA

aproximarão a altura de cada um dos cinco vértices que podem estar ausentes. Então, o erro no espaço do objeto será a diferença de altura entre a altura original do vértice e a média de alturas entre os dois vértices que o aproximarão caso ele esteja ausente no bloco. O pseudo-código para este cálculo é mostrado a seguir.

```
IniciaErros(int ileft, int jlower, int dim, int paridade)
        Se (dim > 2)
        {
                /* Bloco-Filho Superior Esquerdo */
                IniciaErros(ileft, jlower – dim, dim/2, 1);
                /* Bloco-Filho Superior Direito */
                IniciaErros(ileft + dim, jlower - dim, dim/2, 0);
                /* Bloco-Filho Inferior Esquerdo */
                IniciaErros(ileft, jlower, dim/2, 0);
                /* Bloco-Filho Inferior Direito */
                IniciaErros(ileft + dim, jlower, dim/2, 1);
        }

        Para cada um dos 5 vértices (com coordenadas x,y) que podem ser retirados do
        bloco
                Erro = CalculaErroObj(Altura_vertDir, Altura_vertEsq, Altura_Vertice);
                MatrizTrn[x][y].sigma = Erro;

float CalculaErroObj (float Alt_Dir, float Alt_Esq, float Alt_Vértice)
        Media = (Alt_Dir + Alt_Esq)/2;
        ErroObj = || Alt_Vértice – Media||;
        Retorne ErroObj;
```

Note que inicialmente o procedimento IniciaErros é chamado com os seguintes parâmetros: 0 para *ileft*, tamanho da matriz em potência de 2 que envolve o terreno inteiro para *jlower* e para *dim*, e finalmente 1 para *paridade*. Em outras palavras, inicialmente o procedimento é chamado para o bloco maior e a partir daí é como se ele fosse subdividindo o terreno em blocos menores (veja a Figura 12).

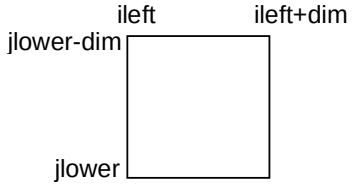


Figura 12 - Bloco processado pelo procedimento IniciaErros

É importante também iniciar todos os raios com valor zero. Uma vez feito isso, enfim é o momento de atualizar os erros no espaço do objeto e os raios de todos os vértices de modo que o erro e o raio de qualquer vértice sejam sempre

PUC-Rio - Certificação Digital Nº 0115656/CA

maiores ou iguais a todos os seus descendentes. Em outros termos, vamos propagar os erros e os raios dos filhos para os pais.

Lembre-se que permanecemos pensando na malha como virtualmente constituída por blocos. Então, podemos simultaneamente ir atualizando o erro no espaço do objeto e o raio de todos vértices, agora percorrendo a hierarquia de malhas no sentido de baixo para cima. Assim, começamos processando os blocos menores e vamos subindo até atingir o bloco que engloba o terreno inteiro, e neste caso todos os erros e raios já estarão corretos e o pré-processamento pode ser encerrado.

Embora continue sendo fundamental saber se um bloco é par ou ímpar (para identificar os pais do vértice central do bloco), como estamos indo de baixo para cima não sabemos quem é filho superior, inferior, direito ou esquerdo, então não podemos usar isto para descobrir se o bloco é par ou ímpar. Por outro lado, repare que sempre começamos com um bloco par, no sentido da esquerda para a direita e de cima para baixo. Além disso, a partir daí eles se alternam, um bloco par seguido por um bloco ímpar, depois outro bloco par e assim por diante, não importa em qual resolução estejamos. (Veja a Figura 13).

A seguir apresentamos o pseudo-código para a propagação simultânea dos erros no espaço do objeto e dos raios dos filhos para os pais, como acabamos de descrever.

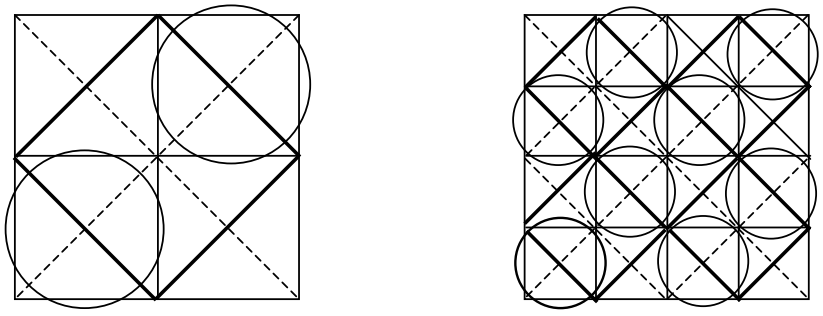


Figura 13 - Os blocos ímpares estão circulados e os pares não.

```
PropagaErrosRaios(int ileft, int jlower, int dimAtual)
Enquanto (dimAtual <= dimMatriz)
{
ipar = 1;
Para (i = ileft; i <= (dimMatriz - dimAtual); i=i+dimAtual)
{
        jpar = -1;
            Para (j = jlower; j >= (dimAtual); j=j-dimAtual)
```

PUC-Rio - Certificação Digital Nº 0115656/CA

```
                {
                        paridade = jpar * ipar;
                        PropagaRaiosBloco(i, j, dimAtual, paridade);
                        PropagaErrosBloco(i, j, dimAtual, paridade);
                        jpar = jpar * (-1);
                }/* fim Para j...*/
                ipar = ipar *(-1);
        }/* fim Para i... */
        dimAtual = (dimAtual*2);
}/* fim Enquanto */

PropagaRaiosBloco(int ileft, int jlower, int dimAtual, int paridade)
  Obtém coordenadas e raio do vértice central do bloco atual;
  dist = 0;
  Para cada um dos 4 filhos do vértice central
  {
        obtém coordenadas e raio do filho;
        Se (dist < distância até o filho + raio do filho)
                dist = distância até o filho + raio do filho;
  }
  if (dist > raio do vértice de central)
        raio do vértice central = dist;
  Obtém coordenadas dos 2 vértices pais do vértice central;
  Se (paridade= 1) /*Bloco é ímpar */
  {
        xpai1 = ileft + dimAtual;
        ypai1 = jlower - dimAtual;
        xpai2 = ileft;
        ypai2 = jlower;
  }
  senão /*se bloco é par */
  {
        xpai1 = ileft;
        ypai1 = jlower - dimAtual;
        xpai2 = ileft + dimAtual;
        ypai2 = jlower;
  }
  Se (existe pai1 )
        Se (raio do pai1 < (raio do vértice + distância do vértice até pai1))
                raio do pai1 = (raio do vértice + distância do vértice até pai1);
  Se (existe pai2 )
        Se (raio do pai2 < (raio do vértice + distância do vértice até pai2))
                raio do pai2 = (raio do vértice + distância do vértice até pai2);

PropagaErrosBloco(int ileft, int jlower, int dimAtual, int paridade)
Obtém coordenadas e erro do vértice central do bloco atual;
erro = 0;
  Para cada um dos 4 filhos do vértice central
  {
        obtém coordenadas e erro do filho;
        Se (erro < erro do filho)
                erro = erro do filho;
  }
  Se (erro > erro do vértice de central)
        erro do vértice central = erro;
  Obtém coordenadas dos 2 vértices pais do vértice central;
  Se (existe pai1 )
        Se (erro do pai1 < (erro do vértice central))
                erro do pai1 = (erro do vértice central);
  Se (existe pai2 )
```

PUC-Rio - Certificação Digital Nº 0115656/CA

```
Se (erro do pai2 < (erro do vértice central))
erro do pai2 = (erro do vértice central);
```

Inicialmente PropagaErrosRaios é chamado com os seguintes parâmetros: 0 para *ileft*, dimensão da matriz envolvendo o terreno todo para *jlower* e para *dimAtual*.

3.4. Procedimento para Refinamento em Tempo Real

O algoritmo de refinamento, aqui denominado RefinaMalha, inicia com uma malha-base contendo quatro triângulos, conforme mostrado na Figura 14 e pode ser descrito pelo seguinte pseudo-cógido:

```
RefinaMalha (V, n)
  paridade (V) = 0;
  V = ( iso, iso);
  RefinaSubMalha (V, ic, is, n);
  AnexaStrip (V, ise,1);
  RefinaSubMalha (V, ic, il, n);
  AnexaStrip (V, ine,1);
  RefinaSubMalha (V, ic, in, n);
  AnexaStrip (V, ino,1);
  RefinaSubMalha (V, ic, io, n);
  AnexaStrip (V, iso,1);
```

Aqui a variável *n* é o número de níveis de refinamento, i_c o vértice localizado no centro da malha, i_{so}, i_{se}, i_{ne} e i_{no} os quatro vértices localizados nos cantos da grade e, por fim, i_s, i_l, i_n e i_o são os vértices introduzidos no primeiro nível de refinamento, conforme mostrado na Figura 14. O vetor de triângulos inicia com duas cópias do vértice localizado no canto inferior esquerdo da malha para permitir que o teste na primeira linha do procedimento AnexaStrip possa ser executado. No momento da renderização o primeiro vértice do vetor pode, então, ser descartado.

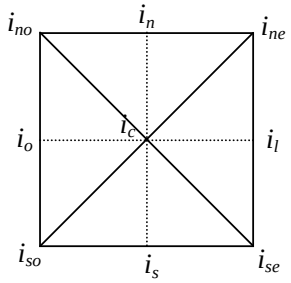


Figura 14 - Malha-base.

PUC-Rio - Certificação Digital Nº 0115656/CA

Internamente, o algoritmo RefinaMalha invoca o procedimento RefinaSubMalha para cada um dos quatro triângulos mostrados na Figura 14. Este procedimento realiza o percurso mais interno na hierarquia de malhas e f_d e f_e são os vértices filhos de j no DAG (grafo direcionado acíclico) do triângulo atual (veja a Figura 15). RefinaSubMalha é então chamado recursivamente com j como o novo vértice-pai para cada um dos dois novos triângulos criados.

Observe que um vetor de vértices é retornado pelo procedimento.

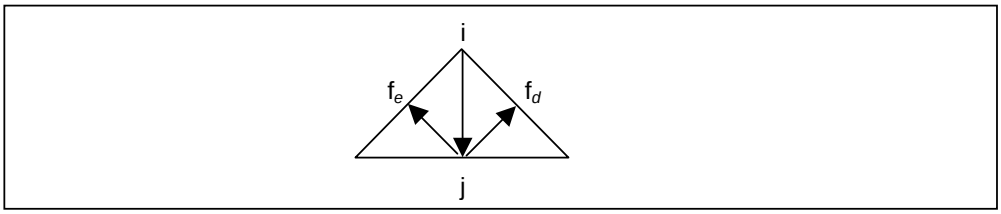
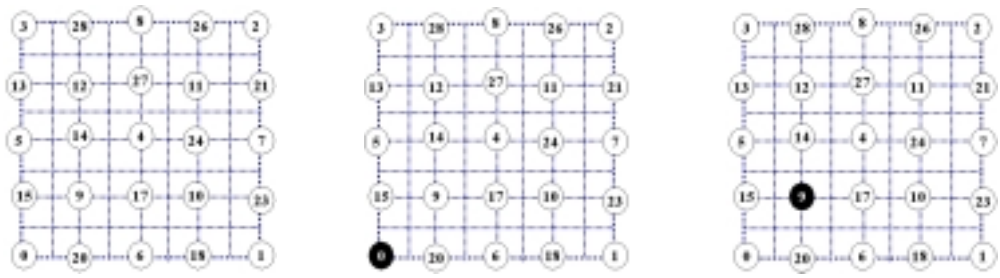


Figura 15 - Destacando filhos direito e esquerdo do vértice j.

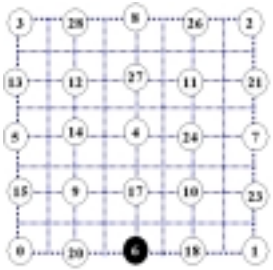
```
RefinaSubMalha (V, i, j, l)
    Se (l > 0) && (ativo(i) ==1))
        RefinaSubMalha(V,j, fe, l-1);
        AnexaStrip(V, i,(l % 2));
        RefinaSubMalha(V,j,fd, l-1);

AnexaStrip (V, v, p)
  Se (( v != vn-1) && (v != vn)
            Se  (p != paridade(V))
                    paridade (V) = p;
            senão
                    V = (V, vn-1);
            V = (V, v);
```

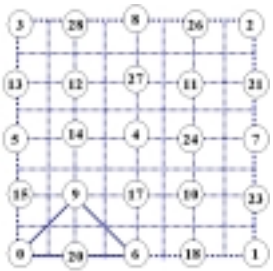
Através da sequência de imagens da Figura 16, tentamos ilustrar a *renderização* de uma *strip* de triângulos produzida pelo procedimento de refinamento descrito nesta seção. A *strip* deste exemplo constrói a malha mostrada na figura t39 e que, conforme pode ser observado, mapeia em cima de uma grade.

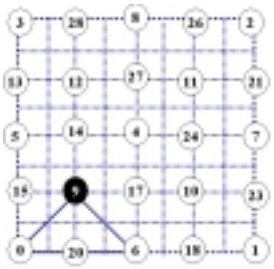


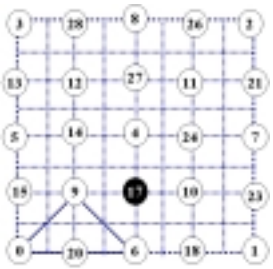
t1) t2) t3)

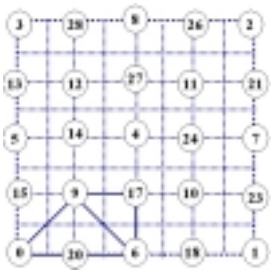
PUC-Rio - Certificação Digital Nº 0115656/CA

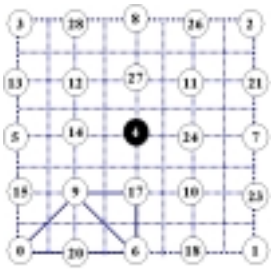

t4)

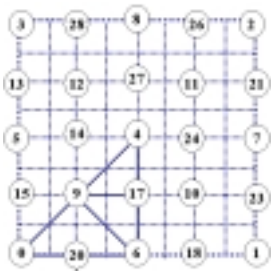

t5)

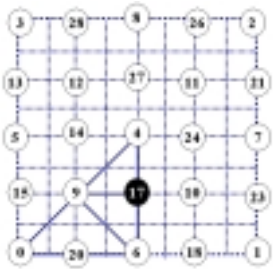

t6)

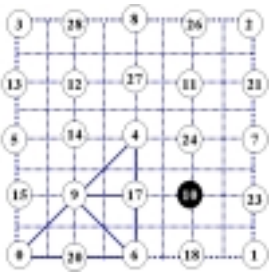

t7)

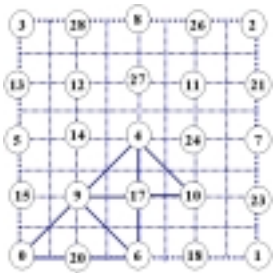

t8)

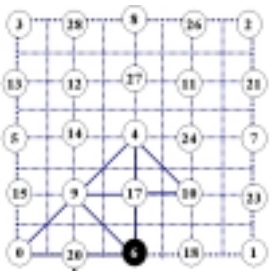

t9)

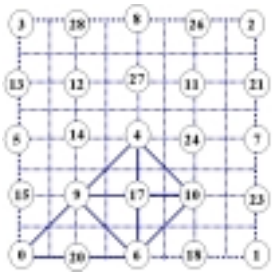

t10)

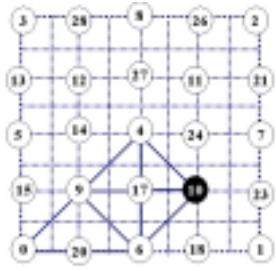

t11)

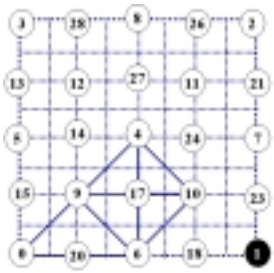

t12)

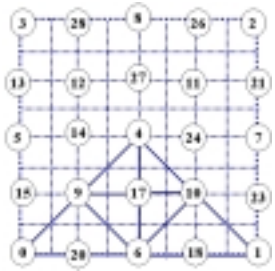

t13)


t14)

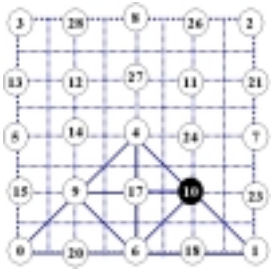

t15)

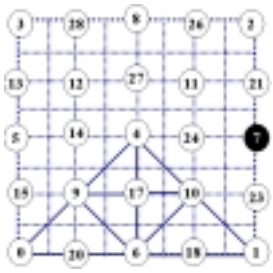

t16)

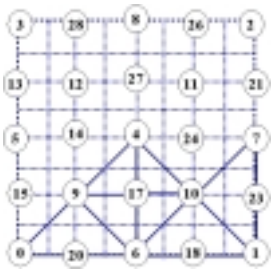

t17)

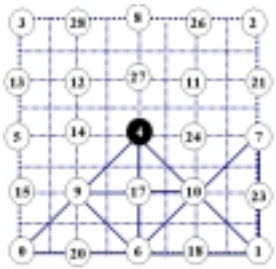

t18)

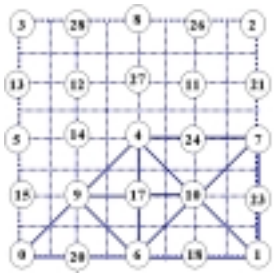
PUC-Rio - Certificação Digital Nº 0115656/CA

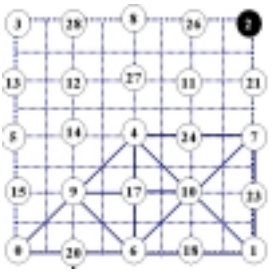

t19)

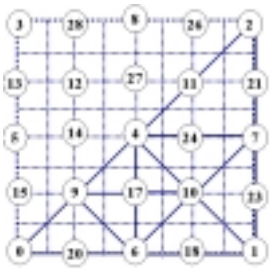

t20)

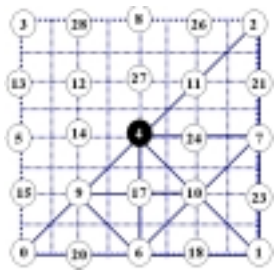

t21)

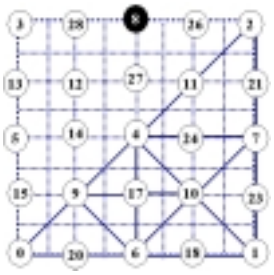

t22)

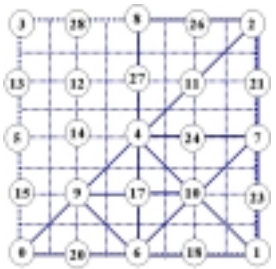

t23)

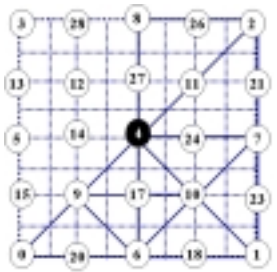

t24)

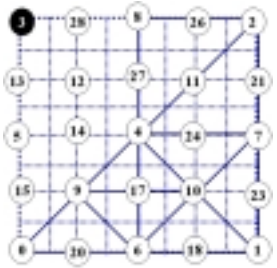

t25)

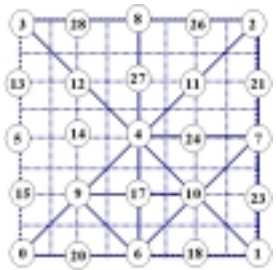

t26)

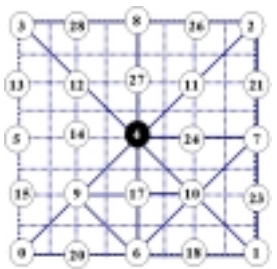

t27)

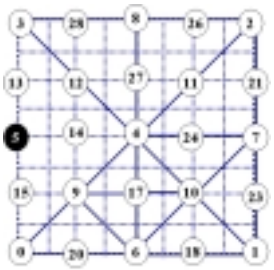

t28)


t29)

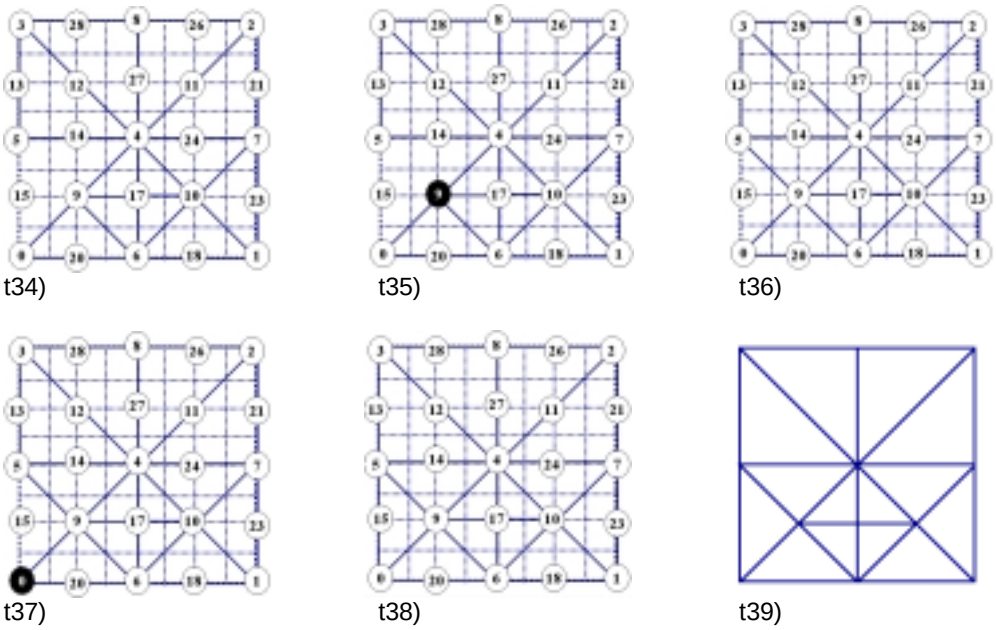

t30)


t31)


t32)


t33)

PUC-Rio - Certificação Digital Nº 0115656/CA



Strip de vértices: 0, 9, 6, 9, 17, 4, 17, 10, 6, 10, 1, 10, 7, 4, 2, 4, 8, 4, 3, 4, 5, 9, 0

Figura 16 – Animação da *renderização* de uma *strip* de triângulos.

Note que os procedimentos de refinamento e renderização podem ser paralelizados. Contudo, nossa implementação não explora esta potencialidade do algoritmo.

3.5. Procedimento para *Culling* da Malha fora da Visão

No algoritmo sendo descrito neste capítulo, o *culling* da malha fora do volume de visão pode ser feito simultaneamente com o refinamento, explorando a natureza hierárquica da malha e das esferas aninhadas. A partir daí, podem-se descartar grandes regiões sempre que possível. Para adicionar o *culling* ao refinamento básico, basta trocar, no procedimento RefinaMalha, as chamadas para RefinaSubMalha por chamadas para o procedimento RefinaSubMalhaVisivel, cujo pseudo-código é apresentado abaixo.

```
RefinaSubMalhaVisivel (V, i, j, l, dentro)
  Se (dentro[k] p/ k de 0 a 5)        // totalmente dentro
       RefinaSubMalha (V, i, j, l);
  Senão se (l > 0) && (ativo(i)) &&  (Visível(i, dentro)) //se intercepta
     RefinaSubMalhaVisível(V,j, fe, l-1);
     AnexaStrip(V, i,(l % 2));
     RefinaSubMalhaVisível(V,j, fd, l-1);
```

PUC-Rio - Certificação Digital Nº 0115656/CA

```
Visivel (i, dentro)
   Para cada plano do frustum de visão <nk, dk>
  Se (não dentro[k])
        s = nk • pi + dk;
              Se (s > ri)
             return falso;
      Se (s < -ri)
             dentro[k] = verdadeiro;
   retorne verdadeiro;
```

O procedimento de *culling* descrito aqui usa os seis planos do *frustum* ou volume de visão. Por conta disso, as equações implícitas de cada um deles devem ser calculadas antecipadamente, no espaço do objeto, e passadas ao longo do refinamento. A seção seguinte apresenta o pseudo-código para obter estas equações. Além disso, é necessário manter uma variável sinalizadora (*flag*) para cada um dos planos, indicando se a esfera centrada no ponto está completamente dentro do volume com relação a cada plano. Assim, como a esfera centrada em i contém i e todos os seus descendentes, então se a esfera de i estiver completamente dentro do volume o vértice i e todos os seus descendentes também estarão e não é mais necessário fazer testes de visibilidade para os descendentes de i. Neste caso, a partir daí pode-se chamar o procedimento RefinaSubMalha em lugar de RefinaSubMalhaVisivel.

Se, por outro lado, a esfera centrada em *i* estiver completamente fora do volume de visão, então significa que *i* e todos seus descendentes podem ser descartados e o refinamento pode terminar.

É importante destacar que este *culling* não introduz vértices T, pois as esferas aninhadas se encarregam de manter os relacionamentos entre vértices-pais e os filhos. Em outros termos, repare que aqui também nenhum filho é introduzido na malha final sem que seus ancestrais também sejam.

Para efeito ilustrativo, considere a Figura 17. Uma possibilidade de gerar um vértice T nesta malha seria incluir o vértice *j* sem incluir um dos seus pais i_1 ou i_2. Contudo, note que é impossível imaginar algum plano que deixe *j* do seu lado interior sem, simultaneamente, cortar a esfera envolvente de i_1 e deixar a esfera de i_2 completamente no seu interior ou vice-versa. Neste caso, se pelo critério de erro *j* for ativado, então tanto i_1 quanto i_2 também serão e não teremos vértice T.

PUC-Rio - Certificação Digital Nº 0115656/CA

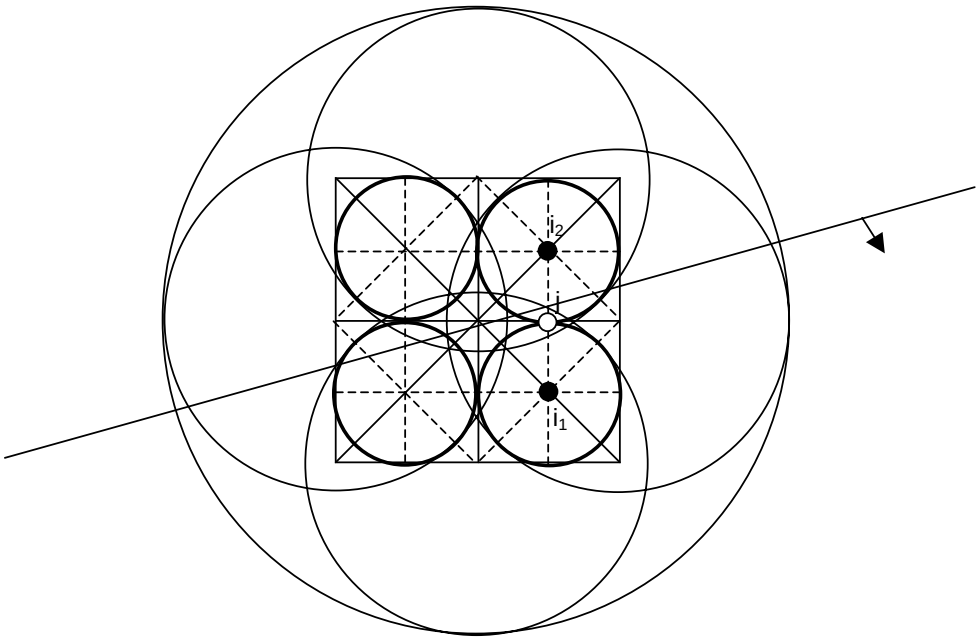


Figura 17 - Esferas Envolventes Aninhadas

3.6. Calculando o *Frustum* de Visão

Cada um dos seis planos que formam o *frustum* de visão, ou seja os planos *near*, *far*, *left*, *right*, *bottom* e *top* (conforme a Figura 18), possui uma equação que pode ser escrita da seguinte forma, em coordenadas no espaço do objeto:

$$ax + by + cz + d = 0$$

Também podemos escrever que $n^T \bullet p' = 0 \Rightarrow \begin{bmatrix} a & b & c & d \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$

Onde n^T é a normal transposta do plano e p é um ponto deste mesmo plano.

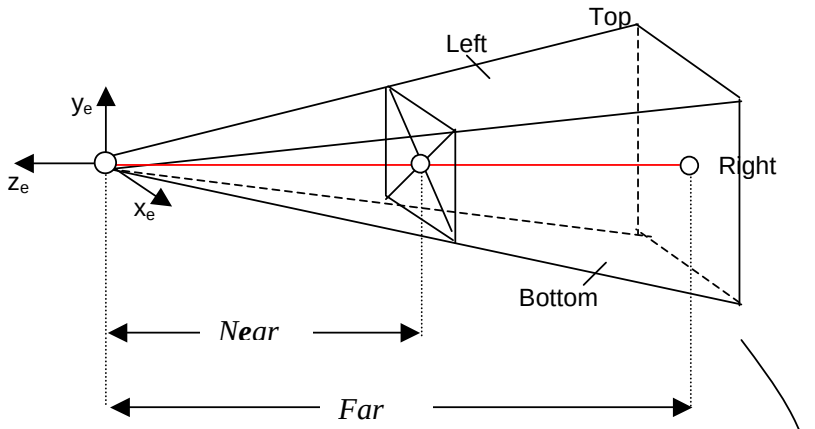
Depois que cada plano do *frustum* é multiplicado pelas matrizes MODELVIEW e PROJECTION do OpenGL, obtemos os chamados planos de *clipping*. Poderíamos dizer, então, que aqui a equação de cada plano seria algo do tipo:

$$a'x' + b'y' + c'z' + d' = 0$$

PUC-Rio - Certificação Digital Nº 0115656/CA

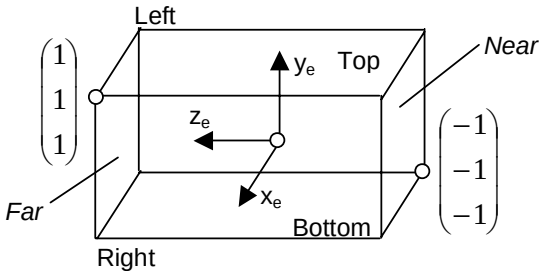
De forma análoga à anterior, podemos escrever $n'^{T} \bullet p = 0 \Rightarrow$

$$\begin{bmatrix} a' & b' & c' & d' \end{bmatrix} * \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = 0$$



a) Sistema de coordenadas do objeto.

M = Projection * Modelview



b) Sistema de coordenadas de *clipping*.

Figura 18 - *Frustum* de Visão

Neste caso, o *frustum* é mapeado para coordenadas dentro de um cubo com coordenadas entre –1 e + 1 (conforme mostrado na figura 18b). Na realidade o *clipping* é feito em coordenadas homogêneas, antes da divisão por w, para evitar que objetos que estejam atrás do olho apareçam na cena. Assim, podemos escrever os planos de *clipping* como:

Near: z + w = 0

Far: z – w = 0

Left: x + w = 0

Right: x – w = 0

PUC-Rio - Certificação Digital Nº 0115656/CA

Bottom: y + w = 0

Top: y – w = 0

Ou seja, podemos montar a seguinte tabela:

	a'	*b*'	*c*'	*d*'
Near	0	0	-1	-1
Far	0	0	1	-1
Left	-1	0	0	-1
Right	1	0	0	-1
Bottom	0	-1	0	-1
Top	0	1	0	-1

Tabela 1 - Planos que formam o *frustum* de visão

Vamos chamar de *M* a matriz MODELVIEW multiplicada pela matriz PROJECTION. Sabemos que *M* é quem faz a transformação do sistema de coordenadas do objeto para o sistema de coordenadas de *clipping* (as coordenadas entre –1 e +1), isto é:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = M * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Então, a equação de cada plano do *frustum* no espaço do objeto também pode ser escrita como:

$$\begin{bmatrix} a & b & c & d \end{bmatrix} * M^{-1} * M * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

pois $M^{-1} * M = I$, onde *I* é a matriz identidade.

Substituindo $M * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ por $\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$, obtemos $\begin{bmatrix} a & b & c & d \end{bmatrix} * M^{-1} * \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = 0$

Disto, podemos tirar que $\begin{bmatrix} a' & b' & c' & d' \end{bmatrix} = \begin{bmatrix} a & b & c & d \end{bmatrix} * M^{-1}$ ou

PUC-Rio - Certificação Digital Nº 0115656/CA

$$\begin{bmatrix} a' \\ b' \\ c' \\ d' \end{bmatrix} = M^{-T} * \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \Rightarrow n' = M^{-T} * n \text{ ou } \boxed{n = M^T * n'}$$

Assim, através da equação obtida acima e da tabela montada anteriormente, podemos obter os planos que formam o *frustum* de visão no espaço do objeto, conforme exigido no procedimento que faz o refinamento da malha de triângulos simultaneamente com o *culling* do trecho da mesma que cai fora do *frustum* de visão.

Se $M = \begin{bmatrix} mpv[0][0] & mpv[0][1] & mpv[0][2] & mpv[0][3] \\ mpv[1][0] & mpv[1][1] & mpv[1][2] & mpv[1][3] \\ mpv[2][0] & mpv[2][1] & mpv[2][2] & mpv[2][3] \\ mpv[3][0] & mpv[3][1] & mpv[3][2] & mpv[3][3] \end{bmatrix}$, então

$$M^T = \begin{bmatrix} mpv[0][0] & mpv[1][0] & mpv[2][0] & mpv[3][0] \\ mpv[0][1] & mpv[1][1] & mpv[2][1] & mpv[3][1] \\ mpv[0][2] & mpv[1][2] & mpv[2][2] & mpv[3][2] \\ mpv[0][3] & mpv[1][3] & mpv[2][3] & mpv[3][3] \end{bmatrix}$$

Para o plano *Near* na tabela anterior, podemos escrever:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = M^T \begin{bmatrix} 0 \\ 0 \\ -1 \\ -1 \end{bmatrix}$$

De onde obtemos que:

$a = -mpv[2][0] - mpv[3][0]$

$b = -mpv[2][1] - mpv[3][1]$

$c = -mpv[2][2] - mpv[3][2]$

$d = -mpv[2][3] - mpv[3][3]$

Para o plano *Far* na tabela anterior, podemos escrever:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = M^T \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix}$$

De onde obtemos que:

$a = mpv[2][0] - mpv[3][0]$

PUC-Rio - Certificação Digital Nº 0115656/CA

$b = mpv[2][1] - mpv[3][1]$

$c = mpv[2][2] - mpv[3][2]$

$d = mpv[2][3] - mpv[3][3]$

Para os planos *Left/Right* na tabela anterior, podemos escrever:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = M^T \begin{bmatrix} \pm 1 \\ 0 \\ 0 \\ -1 \end{bmatrix}$$

De onde obtemos que:

$a = \pm mpv[0][0] - mpv[3][0]$

$b = \pm mpv[0][1] - mpv[3][1]$

$c = \pm mpv[0][2] - mpv[3][2]$

$d = \pm mpv[0][3] - mpv[3][3]$

Por fim, para os planos *Bottom/Top* na tabela anterior, podemos escrever:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = M^T \begin{bmatrix} 0 \\ \pm 1 \\ 0 \\ -1 \end{bmatrix}$$

De onde obtemos que:

$a = \pm mpv[1][0] - mpv[3][0]$

$b = \pm mpv[1][1] - mpv[3][1]$

$c = \pm mpv[1][2] - mpv[3][2]$

$d = \pm mpv[1][3] - mpv[3][3]$

Como $norma = \sqrt{a^2 + b^2 + c^2}$

Então para obter um vetor unitário do tipo $(\hat{a}, \hat{b}, \hat{c})$ e uma equação $\hat{a}x + \hat{b}y + \hat{c}z + \hat{d} = 0$, devemos fazer

$\hat{a} = \frac{a}{norma}$; $\hat{b} = \frac{b}{norma}$; $\hat{c} = \frac{c}{norma}$ e $\hat{d} = \frac{d}{norma}$, para cada um dos os planos citados.

Assim, finalmente, apresentamos a seguir o pseudo-código para obter as equações dos seis planos que formam o *frustum* de visão no espaço do objeto.

PUC-Rio - Certificação Digital Nº 0115656/CA

```
ComputaPlanosFrustum()
    Obtém matrizes MODELVIEW e PROJECTION do OpenGL;
    mpv = PROJECTION * MODELVIEW;
    Para (i = 0; até i != 6; i=i+1)  /* Para cada um dos 6 planos do Frustum */
    {
        k = i / 2;
        Para (j = 0; até j != 4; j = j+1) /*Para cada componente da equação */
        { Se ((i mod 2) = 0)
            fator = +1;
          senão
                fator = -1;
          componente[j] = fator*mpv[k][j] - mpv[3][j];
        }
        norma=sqrt((componente[0]* componente[0]) + (componente[1]*
componente[1])  + (componente[2]* componente[2]));
        nx[i] = (float) (componente[0]/norma);
        ny[i] = (float) (componente[1]/norma);
        nz[i] = (float) (componente[2]/norma);
        d[i] =  (float) (componente[3]/norma);
    }
```

PUC-Rio - Certificação Digital Nº 0115656/CA

3.7. Procedimento para *Geomorphing*

O *geomorphing* é uma espécie de animação para suavizar as transições na geometria da malha durante um sobrevôo e que na prática tenta evitar artefatos visuais como o surgimento ou desaparecimento brusco de montanhas. No algoritmo estudado neste trabalho, foi incorporado um *geomorphing* baseado na posição do observador. Assim, o parâmetro de *geomorphing* para um vértice é uma função da distância dele até o observador.

Deste modo, em vez de definir um valor de tolerância para o erro no espaço da tela, define-se uma faixa de tolerância (τ_{min}, τ_{max}) e quando o erro no espaço da tela do vértice cai nesta faixa o *geomorphing* é aplicado.

O parâmetro *t* para cada vértice pode ser definido como:

$$t = \frac{\rho - \tau_{\min}}{\tau_{\max} - \tau_{\min}} \qquad \text{(eq. 3)}$$

Onde ρ é o erro no espaço da tela, τ_{min} é a tolerância mínima e τ_{max} a tolerância máxima especificada.

Então, quando *t* for menor ou igual a zero o vértice estará inativo, enquanto para *t* maior ou igual a 1 o vértice estará ativo e será desenhado com sua altura original. Para *t* entre zero e um, porém, a altura do vértice será dada por:

$$z(t) = tz_i + (1-t)\frac{z_e + z_d}{2} \qquad \text{(eq. 4)}$$

onde z_i é a altura real do vértice i e z_e e z_d são as alturas dos vértices (direito e esquerdo) que formam a aresta que aproximará a altura de i se i for removido da malha. Como as alturas destes vértices (z_e e z_d) também podem ter passado por um *geomorphing*, ou seja, pode acontecer uma espécie de *geomorphing* em cascata, então os próprios valores de z_e e z_d devem ser passados como parâmetro no procedimento que executa o percurso na hierarquia de malhas.

Agora, usando a definição de *t* e do erro projetado na tela(ρ) temos:

$$t = \frac{\lambda\frac{\delta}{(d-r)} - \tau_{\min}}{\tau_{\max} - \tau_{\min}} \qquad \text{(eq. 5)}$$

Então, definindo uma faixa (d_{min}, d_{max}) para a distância entre o vértice e o observador, temos:

1) $d = d_{min}$ quando $t = 1$, ou seja, o vértice será desenhado com a sua altura real.

Então, substituindo *t* pelo valor 1 e *d* por d_{min} na (eq.5), teremos:

$$1 = \frac{\lambda\frac{\delta}{(d_{\min}-r)} - \tau_{\min}}{\tau_{\max} - \tau_{\min}} \Rightarrow \tau_{\max} - \tau_{\min} = \frac{\lambda\delta}{d_{\min}-r} - \tau_{\min} \Rightarrow d_{\min} - r = \delta\frac{\lambda}{\tau_{\max}} \Rightarrow$$

$$d_{\min} = \frac{\lambda}{\tau_{\max}}\delta + r = \nu_{\min}\delta + r \qquad \text{(eq. 6)}$$

2) $d = d_{max}$ quando $t = 0$, ou seja, o vértice estará inativo e nem será desenhado.

Então, analogamente ao caso anterior, substituindo *t* pelo valor 0 e *d* por d_{max} na (eq.5) teremos:

$$0 = \frac{\lambda\frac{\delta}{(d_{\max}-r)} - \tau_{\min}}{\tau_{\max} - \tau_{\min}} \Rightarrow 0 = \frac{\lambda\delta}{d_{\max}-r} - \tau_{\min} \Rightarrow d_{\max} - r = \delta\frac{\lambda}{\tau_{\min}} \Rightarrow$$

$$d_{\max} = \frac{\lambda}{\tau_{\min}}\delta + r = \nu_{\max}\delta + r \qquad \text{(eq. 7)}$$

Como ρ e *d* são inversamente proporcionais, $\rho = \tau_{min}$ quando $d = d_{max}$ e $\rho = \tau_{max}$ quando $d = d_{min}$, então em vez de escrever *t* como na (eq. 3), poderíamos, de maneira equivalente, escrevê-lo como:

PUC-Rio - Certificação Digital Nº 0115656/CA

$t = \frac{d - d_{max}}{d_{min} - d_{max}}$ ou, equivalentemente (após multiplicar ambos os lados da equação por –1): $t = \frac{d_{max} - d}{d_{max} - d_{min}}$

Finalmente, escolhendo trabalhar com as distâncias ao quadrado (para não nos preocuparmos com questões de sinal), temos que *t* pode ser definido como:

$$t = \frac{d_{max}^2 - d^2}{d_{max}^2 - d_{min}^2} \quad \text{(eq. 8)}$$

Finalmente, note, no pseudo-código do refinamento básico, que precisamos alterar o procedimento AnexaStrip que recebe o índice do vértice como parâmetro para que, em vez disso, ele agora receba as coordenadas x, y e z do vértice a ser anexado no vetor de triângulos a ser renderizado.

Em seguida apresentamos o pseudo-código para o *geomorphing* tal como descrito acima:

```
RefinaSubMalhaMorph (V, i, j, l, z_l, z_a, z_r)
   t = Morph(i);
   Se (l > 0) && (t > 0))
       z = t*z_i + (1-t)*z_a;
       m = (z_l + z_r)/2;
       RefinaSubMalhaMorph(V,j, f_e, l-1, z_l,m, z);
       AnexaPonto(V, x_i, y_i, z,(l % 2));
       RefinaSubMalhaMorph(V,j,f_d, l-1,z, m, z_r);

Morph (i)
  d =  ||p_i – o||;
  d_max = ν_max* δ_l + r_i;
  Se  (d < d_max)
     d_min = ν_min * δ_i + r_i;
     Se (d > d_min)
         retorne (d_max^2 – d^2)/(d_max^2 - d_min^2);
     senão
       retorne 1;
  senão
    retorne 0;
```

PUC-Rio - Certificação Digital Nº 0115656/CA