# 4
# Algoritmo de Visualização

Utilizar a malha completa para a visualização de um terreno pode ser impraticável no caso de terrenos muitos grandes. Por isso, precisa-se encontrar uma forma de visualizar um terreno grande utilizando um número limitado de triângulos. Isto significa que se deve descartar triângulos que contribuam pouco ou nada para a imagem final.

Ao utilizar uma métrica de erro dependente do ponto de vista do observador para a eliminação de triângulos, pode-se dar mais importância a triângulos que estejam mais próximos do observador ou que sejam mais importantes para caracterizar o terreno, como montanhas. Dessa forma, pode-se produzir uma imagem final do terreno de qualidade bem próxima à da imagem que seria gerada utilizando todas as amostras do terreno.

Este capítulo introduz uma solução para a visualização de terrenos dependente do observador. Na primeira seção será apresentado um algoritmo para visualização em memória principal, detalhando como é feito o percurso da *quadtree* para a visualização garantindo a ausência de falhas na superfície do terreno graças a seu critério de refinamento e cálculo de visibilidade. Na segunda seção, é mostrada uma adaptação do algoritmo para uma visualização eficiente utilizando o acesso aos dados em memória secundária, apresentando a predição como um dos artifícios para atingir tal eficiência.

## 4.1
## Visualização em Memória Principal

O algoritmo proposto nesta seção supõe que todos os dados relativos ao terreno podem ser armazenados em memória principal. Ele foi desenvolvido tendo em mente a sua adaptação posterior ao acesso aos dados em memória secundária, que será apresentada na Seção 4.2.

### 4.1.1 Métrica de Erro

Conforme mencionado anteriormente, a métrica de erro utilizada é extremamente importante para um correto julgamento da contribuição de um ladrilho na imagem final apresentada na tela. A métrica utilizada neste trabalho é a proposta por Lindstrom *et al.* [18], e que consiste em projetar na tela o erro no espaço do objeto a partir de um ponto específico. A projeção utilizada apresenta um erro isotrópico, ou seja, o erro é o mesmo independentemente da direção de visualização. Dado um erro no espaço do objeto $\epsilon$ e uma distância $d$ entre o observador e o ponto de onde o erro está sendo projetado, o erro projetado $\rho$ pode ser calculado segundo a Equação 4-1.

$$\rho = \lambda \frac{\epsilon}{d} \tag{4-1}$$

onde $\lambda = \frac{w}{2 \tan \gamma/2}$, w é a altura da janela de visualização em pixels e $\gamma$ é o ângulo de abertura da câmera.

A partir do erro projetado calculado, pode-se obter uma avaliação do erro associado ao ladrilho dada uma tolerância $\tau$ em pixels especificada pelo usuário, conforme apresentado na Equação 4-2 [18]. A manipulação algébrica apresentada evita o uso da raiz quadrada no cálculo da distância.

$$\begin{aligned} \text{AvaliaErro} &\iff \rho > \tau \\ &\iff \lambda \frac{\epsilon}{d} > \tau \\ &\iff \frac{\lambda}{\tau} \epsilon > d \\ &\iff (\nu\epsilon)^2 > d^2 \end{aligned} \tag{4-2}$$

O erro no espaço do objeto utilizado para a projeção é o erro máximo de cada ladrilho. Este é projetado na tela a partir do ponto da caixa envolvente mais próximo ao observador, conforme ilustrado na Figura 4.1. Deve-se atentar para o fato de que este ponto não é o vértice mais próximo, e sim o ponto mais próximo na superfície da caixa envolvente.

Como os erros no espaço do objeto e as caixas envolventes estão aninhados, pode-se garantir que o erro projetado cresce monotonicamente com o erro no espaço do objeto. Esta característica é importante para garantir que nós de níveis inferiores (próximos à raiz) tenham sempre um

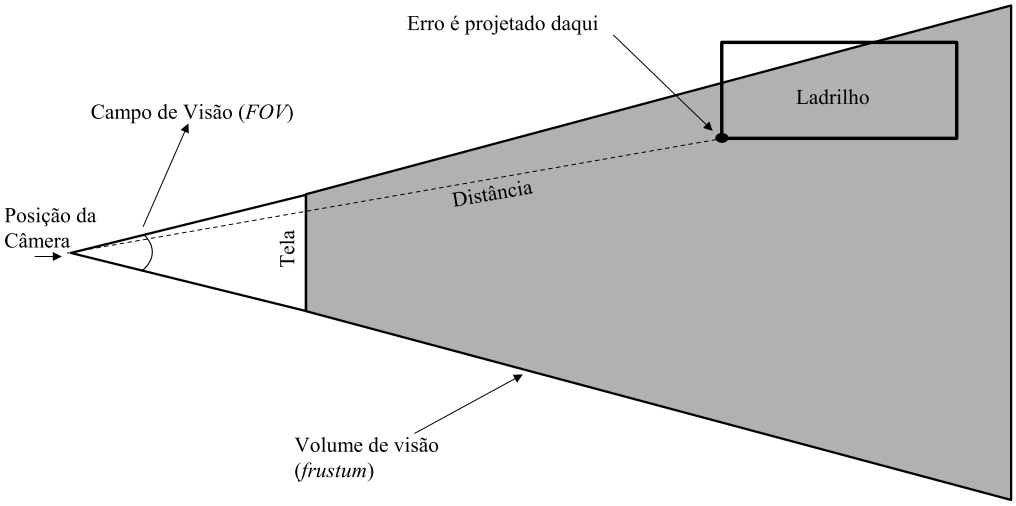



Figura 4.1: Projeção do erro na tela

erro projetado maior do que os de níveis superiores.



### 4.1.2 Algoritmo de Refinamento

O algoritmo de construção da malha visualizada é um algoritmo de refinamento. Isto quer dizer que ele trabalha no sentido de cima para baixo, ou seja, ele inicia o percurso a partir de um nível de resolução mínimo (a raiz da árvore) e vai refinando a malha de acordo com a métrica de erro até que a malha final seja obtida. Além disso, o algoritmo que será apresentado não mantém nenhuma coerência entre quadros consecutivos.

Antes de apresentar o algoritmo, é necessário introduzir o conceito de *nó ativo*. Um nó é considerado ativo se for visível quando foi visitado pelo algoritmo que percorre a estrutura hierárquica. Isto é, um nó que foi visitado, mas nesse momento determinou-se que ele não é visível, não é considerado ativo. A informação de nó ativo fica armazenada em cada nó na forma de um número de quadro. Cada quadro é numerado incrementalmente e, dessa forma, um nó está ativo se o seu número de quadro for o mesmo do quadro atual. Assim, não é preciso reiniciar o estado dos nós a cada execução do algoritmo.

A visibilidade dos nós é calculada fazendo-se o teste da caixa envolvente do nó contra o volume de visão. Este cálculo, também chamado de *frustum culling*, é baseado no algoritmo proposto por Assarsson *et al.* [4].

A execução do algoritmo de visualização proposto neste trabalho inicia-se pela função `DesenhaTerreno`. Inicialmente, esta função chama a função `Percorre`, que faz o percurso da *quadtree* marcando os nós ativos. Em

seguida, a função `ColetaNos` é chamada para coletar os nós que serão usados para a visualização da malha. Por fim, os nós coletados são desenhados.

```
DesenhaTerreno(Quadtree, parametrosVisao)
{
  raiz = Quadtree.raiz();
  Ativa(raiz);
  Percorre(raiz, parametrosVisao);

  listaNos = ColetaNos(Quadtree, parametrosVisao);

  Enquanto(!Vazia(listaNos))
  {
    no = RemoveNoLista(listaNos);
    Desenha(no);
  }
}
```

Como mencionado anteriormente, a função `Percorre` é encarregada de marcar os nós ativos, sendo este procedimento recursivo. Inicialmente, o nó que não está visível é descartado. Em seguida, marca-se o nó atual como ativo, examina-se a métrica de erro do nó ou ladrilho atual com a função `AvaliaErro`, implementada conforme a Equação 4-2, e decide-se se o nó deve ser refinado ou não. A função `AvaliaErro` retorna verdadeiro quando a tolerância de erro especificada não foi atendida (deve ser feita uma divisão no ladrilho para reduzir o erro) e retorna falso caso contrário. Em caso positivo o procedimento é repetido recursivamente para cada um dos filhos do nó atual. A recursão termina quando o nó atual é uma folha ou quando a tolerância de erro projetado foi atendida.

```
Percorre(no_atual, parametrosVisao)
{
  Se (!Visivel(no_atual, parametrosVisao))
    retorna;

  Ativa(no_atual);
  Se (!Folha(no_atual) && AvaliaErro(no_atual, parametrosVisao))
  {
    Para cada filho de no_atual
      Percorre(filho, parametrosVisao);
  }
}
```

Os nós utilizados para o desenho da malha são os nós ativos de maior profundidade, ou seja, os nós ativos cujos filhos estão todos inativos ou

os nós que forem folhas ativas. Seguindo esta regra, a função `ColetaNos` percorre a árvore fazendo a coleta desses nós.

```
ColetaNos(Quadtree, parametrosVisao)
{
  listaNos = NovaLista();
  pilhaNos.Empilha(Quadtree.raiz());
  Enquanto (!Vazia(pilhaNos))
  {
    no = DesempilhaTopo(pilhaNos);

    Se (Folha(no))
    {
      Adiciona(no, listaNos);
      continua;
    }

    contaAtivos=0;
    Para filho de no
    {
      Se EstaAtivo(filho)
      {
        pilhaNos.Empilha(filho);
        contaAtivos++;
      }
    }

    Se (contaAtivos == 0)
      Adiciona(no, listaNos);
  }
  retorna listaNos;
}
```



### 4.1.3 Eliminação de Falhas

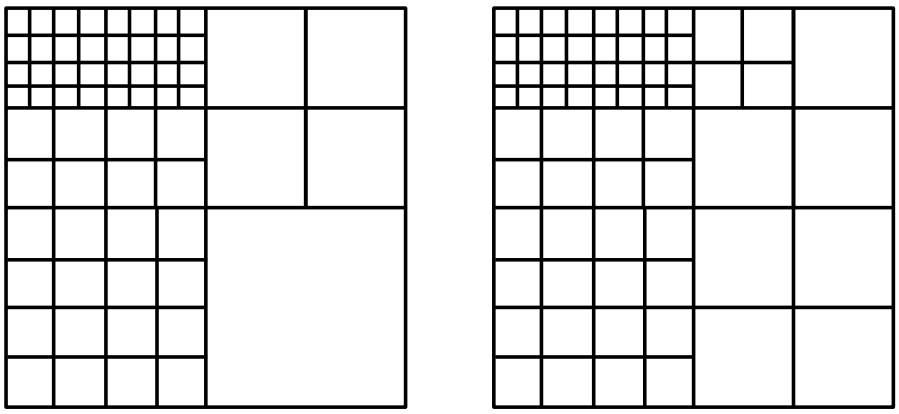
O algoritmo de refinamento apresentado na seção anterior é baseado somente na métrica de erro. Assim, podem ocorrer grandes diferenças em nível de detalhe entre ladrilhos próximos, principalmente porque a rugosidade do ladrilho é levada em consideração no cálculo do erro projetado. Quando isto ocorre, aparecem falhas entre ladrilhos vizinhos.

Alguns algoritmos de visualização de terrenos preenchem falhas fazendo "muros"verticais em torno das bordas do ladrilho, mas este tipo de solução produz artefatos perceptíveis na superfície do terreno. Outros

métodos envolvem criar triângulos para preencher os buracos entre os ladrilhos ou modificar a geometria de um dos ladrilhos adjacentes para produzir uma vizinhança sem falhas. Embora funcionem, estes métodos foram evitados por consumir muito processamento da *CPU*.

A solução adotada propõe corrigir as falhas sem modificar a geometria de nenhum ladrilho. Isto pode ser feito garantindo que ladrilhos vizinhos não apresentem uma diferença de nível maior do que 1. Com isso, usando um esquema de indexação alternativo, cada ladrilho pode se adaptar ao seu vizinho de resolução inferior conforme será mostrado na Seção 4.1.4.

Quando um algoritmo de refinamento de árvore, como o da Seção 4.1.2, apresenta ladrilhos cuja diferença de nível entre vizinhos pode ser maior do que 1, diz-se que a árvore não está balanceada. Este caso está ilustrado na Figura 4.2(a). O objetivo desta seção é alterar o algoritmo proposto para sempre gerar uma *quadtree* balanceada, como a da Figura 4.2(b).



4.2(a): 4.2(b):

Figura 4.2: Balanceamento de *quadtrees*: (a) *Quadtree* não balanceada; (b) *Quadtree* balanceada

Röttger et al. [24] propõem uma forma de garantir o balanceamento da árvore através de uma métrica de erro modificada, mas este tipo de solução não resolveria totalmente o problema pois, como será mostrado na Seção 4.2.1, a presença ou não de ladrilhos em memória será um fator que alterará o balanceamento da árvore.

A solução utilizada foi proposta por Hill [13] e consiste em modificar a função `Percorre`, acrescentando uma chamada para uma nova função `Divide`, que é encarregada de fazer a divisão do nó atual de modo a garantir a diferença de um nível entre vizinhos ativos visíveis. Portanto, a função `Percorre` chama a função `Divide` antes da chamada recursiva de cada filho para garantir o balanceamento. As funções `DesenhaTerreno` e `ColetaNos` permanecem sem qualquer modificação.

```
Percorre(no_atual, parametrosVisao)
{
  Se(!Visivel(no_atual, parametrosVisao))
    retorna;
  Se (!Folha(no_atual) &&
      AvaliaErro(no_atual, parametrosVisao))
  {
    Divide(no_atual, parametrosVisao);
    Para cada filho de no_atual
      Percorre(filho, parametrosVisao);
  }
}
```

A função `Divide` garante a diferença de 1 nível entre ladrilhos adjacentes ao primeiramente dividir de forma recursiva todos os vizinhos ativos visíveis de um nó que não estão no mesmo nível que ele, para depois executar a divisão do nó em questão.

```
Divide(no_atual, parametrosVisao)
{
  Para cada vizinho ativo de no_atual
  {
    Se (Visivel(vizinho, parametrosVisao) && (Nivel(vizinho) < Nivel(no_atual)))
      Divide(vizinho, parametrosVisao);
  }

  // Faz divisão de no_atual
  Para cada filho de no_atual
  {
    Se (Visivel(filho, parametrosVisao))
      Ativa(filho);
  }
}
```

Pode-se perceber neste novo algoritmo que o teste de visibilidade pode ocorrer mais de uma vez para cada nó, acarretando uma redução no desempenho. Para resolver tal problema, o resultado de um teste de visibilidade é armazenado em cada ladrilho junto com o número do quadro em que o teste foi realizado. Dessa forma, pode-se rapidamente saber se o teste já foi executado e, em caso positivo, somente retornar o resultado armazenado.

### 4.1.4 Processamento do Ladrilho

Conforme apresentado na seção anterior, dada uma árvore balanceada, a malha final deve ser visualizada sem falhas. Para isto, um ladrilho deve ser capaz de se adaptar aos seus ladrilhos vizinhos. Isto é feito conforme explicado a seguir.

Cada ladrilho é representado como uma grade regular com triangulação completa, ou seja, todos os vértices fazem parte da malha de visualização. Um esquema de indexação alternativo é usado para criar triângulos maiores ao longo da borda do ladrilho mais refinado, para que este se adapte ao seu vizinho menos refinado. Dessa forma, a adaptação ao vizinho sempre é feita com o ladrilho de nível alto adaptando-se a um ladrilho de nível baixo, conforme ilustrado na Figura 4.3.

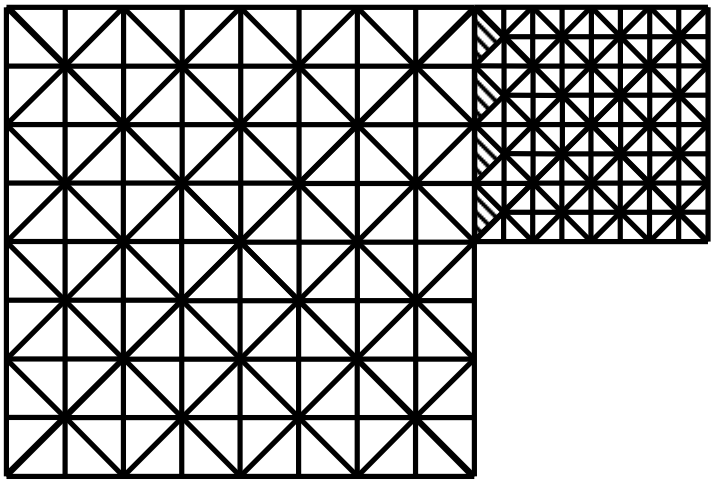


Figura 4.3: Adaptação ao vizinho: os triângulos hachurados não foram subdivididos para evitar falhas na malha final

Com isso é possível fazer a adaptação entre ladrilhos adjacentes sem ter que modificar nenhuma informação de vértice. Entretanto, como cada ladrilho pode possuir quatro vizinhos, seriam necessários 16 indexações diferentes para cobrir todas as combinações de vizinhanças. Para reduzir a quantidade de indexações, a malha do ladrilho é então dividida em quatro sub-malhas na forma de triângulo, conforme ilustrado na Figura 4.4.

Com esse tipo de divisão, é possível criar duas indexações diferentes que representem as duas triangulações necessárias para cada sub-malha. A triangulação ilustrada na Figura 4.5(a) faz adaptação da vizinhança com ladrilhos vizinhos de nível igual ou superior (no caso do vizinho de nível superior, será ele quem deverá fazer a adaptação). A triangulação da Figura 4.5(b) adapta-se a vizinhos de nível inferior. Dessa forma, somente

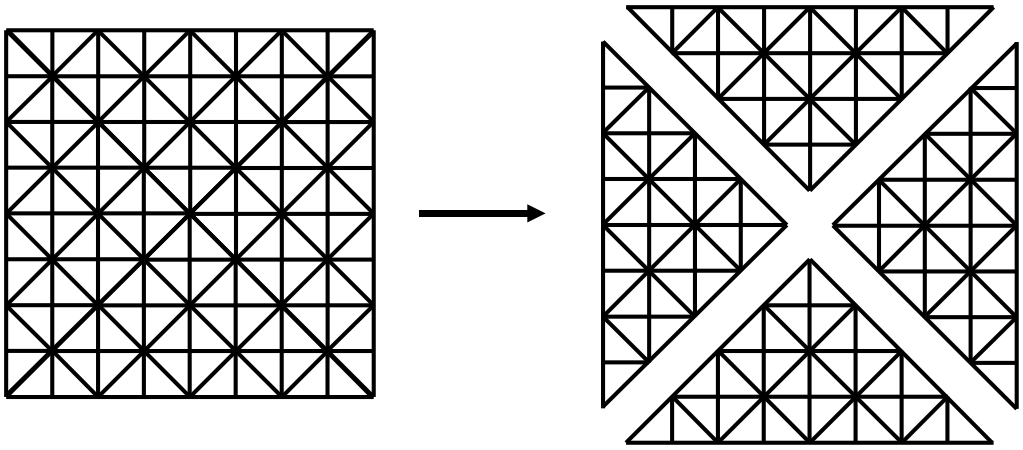
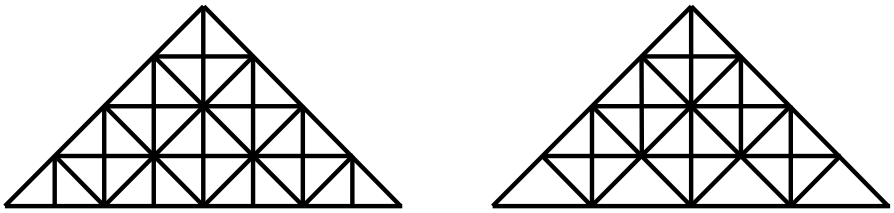


Figura 4.4: Particionamento do ladrilho em quatro triângulos

8 indexações diferentes têm que ser criadas. Elas são criadas no início da execução do programa, tendo como entrada a resolução do ladrilho, e são selecionadas em tempo de execução por cada ladrilho com base nos seus vizinhos. Para que este tipo de triangulação seja possível e ao mesmo tempo a adaptação aos vizinhos seja consistente, a grade regular que representa o ladrilho deve ter a resolução $2^n + 1 \times 2^n + 1$.



4.5(a): 4.5(b):

Figura 4.5: Dois tipos de triangulações pré-calculadas: (a) Vizinho tem mesma resolução; (b) Vizinho tem a resolução inferior

A partir das indexações apresentadas, o desenho do ladrilho é feito utilizando uma malha 2D com a resolução do ladrilho. Como todos os ladrilhos do terreno são de dimensões iguais, essa malha 2D é única para todos os ladrilhos. Com o mesmo argumento, as indexações das triangulações também são únicas. A malha 2D apresenta valores x e y entre 0 e 1 nas duas dimensões, sendo que fatores de translação e escala irão posicionar e redimensionar a malha para que o desenho do ladrilho seja correto. Com o intuito de minimizar a transferência de dados para a placa gráfica *(GPU)*, as coordenadas x e y da malha 2D e as indexações que formam as triangulações explicadas anteriormente são armazenadas em memória de vídeo.

Para obter a posição 3D de cada vértice, os dados de altura de cada ladrilho em particular são enviados como coordenada de textura para a

*GPU*. O uso de coordenadas de textura justifica-se pois pode-se enviar somente uma coordenada por vértice. Em seguida, um *vertex program* atribui a coordenada de textura de um vértice à sua coordenada z, obtendo a posição em 3 dimensões desejada.

A seguir, é apresentado o código do *vertex program* para o cálculo da posição 3D em *GLSL* [20], a linguagem de *shader* do *OpenGL*. Deve-se atentar para o fato de que as transformações de translação e escala estão aplicadas à matriz de transformação `gl_ModelViewProjectionMatrix`.

```
void main()
{
  vec4 vertexData;
  vertexData.x = gl_Vertex.x;
  vertexData.y = gl_Vertex.y;
  vertexData.z = gl_MultiTexCoord1.x;
  vertexData.w = 1.0;

  // saída
  gl_Position = gl_ModelViewProjectionMatrix * vertexData;
}
```

## 4.2 Visualização em Memória Secundária

Conforme exposto anteriormente, a quantidade de dados que um terreno pode utilizar facilmente excede a capacidade da memória principal de um computador. Dessa forma, é necessário um sistema para gerenciar o carregamento/descarregamento dos ladrilhos.

Fazer o carregamento dos ladrilhos no início de cada quadro é uma solução que praticamente inviabiliza obter um desempenho em tempo real pois, dependendo da velocidade de navegação do observador, muitos ladrilhos novos terão que ser carregados e isto demandará um tempo considerável, quebrando a fluidez de execução do programa.

A solução proposta é fazer o carregamento dos ladrilhos para a memória de forma assíncrona, utilizando duas linhas de execução *(threads)*. Dessa forma, é possível paralelizar operações de *IO*(entrada-saída) e manter um desempenho de tempo real para a visualização, mesmo que a atualização da geometria ocorra com menos freqüência. Uma linha de execução é a de gerenciamento de ladrilhos, responsável por carregar/remover ladrilhos para/da memória. Esta linha de execução utiliza um mecanismo de predição de movimento da câmera para carregar ladrilhos que possam ser utilizados

em um futuro próximo e remover ladrilhos que provavelmente não serão necessários.

A outra linha de execução é a de visualização, responsável por visualizar o terreno, fazendo o cálculo do erro projetado, eliminando ladrilhos não visíveis e balanceando a estrutura de *quadtree* para eliminar falhas ou vértices T na superfície do terreno, levando em consideração que os ladrilhos podem não estar carregados em memória principal. A visualização pode ser feita de duas formas distintas: baseada no erro máximo tolerado ou na quantidade máxima de polígonos a ser processada.

Deve-se acrescentar para cada nó da árvore uma indicação de que o ladrilho correspondente está carregado ou não. Não há comunicação explícita entre as linhas de execução, ou seja, não são enviadas mensagens entre ambas. A comunicação é feita através de uma memória compartilhada protegida por seções críticas. As seções críticas garantem que condições de disputa *(race conditions)* não possam ocorrer.



### 4.2.1 Linha de Execução de Visualização

O algoritmo de visualização é uma adaptação do algoritmo apresentado na Seção 4.1. Esta adaptação consiste em garantir que somente os nós que estejam carregados em memória principal sejam utilizados. É necessário também garantir o balanceamento levando em consideração esta restrição. Dessa forma, mesmo que a linha de execução de gerenciamento de ladrilhos não consiga trazer para a memória todos os ladrilhos necessários, haverá uma visualização suave e sem falhas, ainda que em uma resolução mais baixa. A visualização em resolução mais baixa geralmente não é perceptível, pois a falta de ladrilhos ocorre normalmente em casos de navegação em alta velocidade. Além disso, pode-se utilizar a predição de movimento de câmera para minimizar a ocorrência desses casos, conforme será apresentado na Seção 4.2.2.

O algoritmo que será apresentado utiliza a função `Divide` de forma diferente. Esta passa a retornar um valor booleano que indica se o ladrilho pode ser dividido ou não, e é acrescentada como condição adicional de divisão na função `Percorre`, além das condições do nó não ser folha e atender à tolerância desejada. Além disso, a função `Divide` muda de comportamento, só podendo fazer a divisão do nó se todos os filhos visíveis estão carregados. O trecho de código que faz este teste, ou seja, o trecho

entre o teste de carregamento dos filhos e a ativação deles deve ser feito em seção crítica.

Segue abaixo o pseudo-código do algoritmo.

```
Divide(no_atual, parametrosVisao)
{
  Para cada vizinho ativo de no_atual
  {
    Se (Visivel(vizinho, parametrosVisao) && (Nivel(vizinho) < Nivel(no_atual)))
      Se (!Divide(vizinho, parametrosVisao))
        retorna falso;
  }

  InicializaSeçãoCritica();

  Se existe algum filho visível não carregado
    retorna falso;

  Para cada filho de no_atual
  {
    Se (Visivel(filho, parametrosVisao))
      Ativa(filho);
  }
  FinalizaSeçãoCritica();

  retorna verdadeiro;
}

Percorre(no_atual, parametrosVisao)
{
  Se(!Visivel(no_atual, parametrosVisao))
    retorna;
  Se (!Folha(no_atual) &&
      AvaliaErro(no_atual, parametrosVisao) &&
      Divide(no_atual, parametrosVisao))
  {
    Para cada filho de no_atual
      Percorre(filho, parametrosVisao);
  }
}
```

### Visualização Limitada por Quantidade de Polígonos

Com os algoritmos apresentados anteriormente, a qualidade da imagem obtida é especificada por uma tolerância de erro máximo projetado na tela. Entretanto, dependendo de fatores como a complexidade do terreno, o

poder computacional da *GPU* e *CPU*, o tamanho da janela de visualização, entre outros, pode-se obter um desempenho inaceitável para a tolerância de erro especificada. Contudo, a maioria das aplicações deseja manter um nível mínimo de taxa em quadros por segundo para que o usuário tenha uma experiência de navegação ininterrupta. Visando resolver este tipo de problema, um algoritmo foi desenvolvido para visualizar o terreno em uma taxa mínima de quadros por segundo especificada, cometendo o menor erro projetado possível.

Na maioria das aplicações, pode-se assumir que a taxa de quadros por segundo é inversamente proporcional à quantidade de polígonos desenhados. O algoritmo que será apresentado, portanto, permite especificar uma quantidade máxima de polígonos, proporcionando uma taxa mínima em quadros por segundo. Inicialmente, converte-se a quantidade máxima de polígonos em número máximo de ladrilhos. Isto pode ser feito de forma trivial, dado que a quantidade de polígonos em cada ladrilho é aproximadamente a mesma (é aproximada devido às possíveis adaptações a vizinhos, que podem ser consideradas mínimas).

O algoritmo utiliza uma fila de prioridade (heap) em que a ordenação é feita pelo ladrilho de maior erro. Assim, os ladrilhos mais importantes, ou seja, os que provocariam o maior erro, têm prioridade no processo de divisão. Para que a ordenação seja possível, é necessário obter o erro projetado de cada ladrilho. A função que executa tal ação é a `ErroNo`, calculada segundo a Equação 4-1.

O algoritmo inicia-se pela função `DesenhaTerreno`, que é semelhante ao `Percorre` apresentado nas seções anteriores. A principal diferença é que `DesenhaTerreno` é executado de forma iterativa, utilizando a fila de prioridade, e não recursiva. É acrescentada também uma condição extra de parada, que é quando o limite máximo de ladrilhos é atingido.

```
DesenhaTerreno(Quadtree, parametrosVisao, maxNumLadrilhos)
{
  raiz = Quadtree.raiz();
  // Só inicia o procedimento se a raiz estiver carregada
  Se (!EstaCarregado(raiz))
    retorna;

  Ativa(raiz);
  contagemLadrilhos = 1;

  erro = ErroNo(raiz);
  heap.Adiciona(erro, raiz);
  Enquanto(!Vazia(heap) && (maxNumLadrilhos >= contagemLadrilhos))
```

```
  {
    no_atual, erro = heap.RemovePrimeiro();
    Se (!Visivel(no_atual, parametrosVisao))
      retorna;
    Se (!Folha(no_atual) &&
        erro > parametrosVisao.tau  &&
        Divide(no_atual, parametrosVisao, maxNumLadrilhos))
    {
      Para cada Filho de no_atual
      {
        erro = ErroNo(Filho);
        heap.Adiciona(erro, Filho);
      }
    }
  }

  listaNos = ColetaNos(Quadtree, parametrosVisao);

  Enquanto(!Vazia(listaNos))
  {
    no = RemoveNoLista(listaNos);
    Desenha(no);
  }
}
```

A contagem dos ladrilhos que serão usados na visualização da malha é feita com uma pequena alteração na função `Divide`, através de um contador que é incrementado quando cada ladrilho muda seu estado para ativo. Deve-se atentar para o fato de que quando um ou mais filhos de um pai é ativado, deve-se decrementar o contador de uma unidade, pois o pai não será mais utilizado para a visualização.

```
Divide(no_atual, parametrosVisao, maxNumLadrilhos)
{
  Para cada vizinho ativo de no_atual
  {
    Se (Visivel(vizinho, parametrosVisao) && (Nivel(vizinho) < Nivel(no_atual)))
      Se (!Divide(vizinho, parametrosVisao))
        retorna falso;
  }

  Se (filhos visíveis de no_atual não estão carregados)
    retorna falso;

  ladrilhoAdicionado = falso;
  Para cada filho de no_atual
```

```
    Se Visivel(filho, parametrosVisao)
    {
      Se (!EstaAtivo(filho))
      {
        contagemLadrilhos++;
        ladrilhoAdicionado = verdadeiro;
      }
      Ativa(filho);
    }

  // Se algum filho foi adicionado,
  // o pai não pode ser mais contado como nó desenhado
  Se (ladrilhoAdicionado)
    contagemLadrilhos--;

  retorna verdadeiro;
}
```



### 4.2.2 Linha de Execução de Gerenciamento de Ladrilhos

Esta linha de execução é dividida em três partes principais: gerenciamento de ladrilhos, algoritmo de carregamento de ladrilhos e predição de movimento de câmera.

#### Gerenciamento de Ladrilhos

Inicialmente, deve-se incluir nos ladrilhos a funcionalidade de carregamento/descarregamento das alturas a partir da memória secundária para que o gerenciamento de ladrilhos seja possível. Os ladrilhos carregados em memória são controlados por uma fila de prioridade de tamanho fixo, sendo este tamanho o número máximo de ladrilhos que podem estar carregados em memória. Isto quer dizer que assim que a fila estiver cheia, o primeiro ladrilho da fila será descarregado para a inclusão de um novo ladrilho.

A política de liberação de ladrilhos é feita utilizando o conceito de ladrilho em uso. A variável que indica se um ladrilho está ativo indica também qual o último quadro em que ele foi usado. Um ladrilho é considerado em uso enquanto a diferença entre o número do quadro atual e o número do quadro ativo indicado é menor ou igual a um número $N$. Isso é necessário porque, mesmo depois de terminar a rotina de visualização, a $GPU$ pode ainda estar utilizando o ladrilho em paralelo. Geralmente, o valor usado é 5. Utilizar um valor um pouco maior para $N$, por exemplo entre 50 e 100,

pode ser uma boa estratégia pois a possibilidade de um ladrilho que ficou inativo por uns poucos quadros voltar a ficar ativo é grande. Portanto, um ladrilho só pode ser removido caso ele não esteja em uso.

A fila de prioridade é implementada como um *heap*, em que a ordenação de cada ladrilho é determinada pelo número do último quadro em que o ladrilho foi usado ponderado por um valor proporcional ao seu nível de profundidade na árvore. Isto é feito para que os ladrilhos mais antigos e de maior profundidade sejam os primeiros a sair da fila. Para que o *heap* não tenha que ser atualizado a cada vez que um ladrilho é utilizado, a condição de prioridade é relaxada. A posição do ladrilho no *heap* é calculada somente no momento da inserção, e uma verificação posterior de uso é feita no momento da remoção. Caso o ladrilho não possa ser removido, ele é reinserido no *heap*, refazendo-se o cálculo de prioridade com o seu número de quadro ativo atualizado.

Segue abaixo o pseudo-código da função que adiciona ladrilhos, seguindo a política de liberação apresentada acima.



```
AdicionaLadrilho(numQuadroAtual, ladrilhoRequisitado)
{
  Se (heap.Tamanho() >= numMaxLadrilhos)
  {
    ladrilhoRemovido = falso;
    Faça
    {
      ladrilho = heap.RemovePrimeiro();

      // Função Descarrega é internamente feita dentro de seção crítica
      Se (ladrilho.Descarrega())
        ladrilhoRemovido = verdadeiro;
      Senão
        heap->Insere(ladrilho.UltimoQuadroUsado(), ladrilho);
    }
    Enquanto(!ladrilhoRemovido);
  }
  ladrilhoRequisitado.Carrega();
  heap.Insere(numQuadroAtual, ladrilhoRequisitado);
  retorna verdadeiro;
}
```

### Algoritmo de Carregamento

O algoritmo de carregamento é feito utilizando adaptações do algoritmo de visualização apresentado na Seção 4.2.1. A primeira adaptação é

que agora a função `Divide_carregamento`, ao invés de ativar um ladrilho, carrega para a memória principal os ladrilhos filhos que não estejam carregados e que estejam visíveis. Adicionalmente, `Divide_carregamento` não retorna mais um booleano, pois a divisão é sempre possível.

Está apresentado abaixo o pseudo-código do algoritmo:

```
Divide_carregamento(no_atual, parametrosVisao)
{
  Para cada vizinho carregado de no_atual
  {
    Se ((Visivel(vizinho, parametrosVisao)) &&
        (Nivel(vizinho) < Nivel(no_atual)))
      Divide_carregamento(vizinho, parametrosVisao);
  }

  Para cada Filho de no_atual
  {
    Se ((Visivel(Filho, parametrosVisao)) &&
        (!Filho.EstaCarregado()))
      AdicionaLadrilho(parametrosVisao.quadroAtual, Filho);
  }
}

Percorre_carregamento(no_atual, parametrosVisao)
{
  Se (!Visivel(no_atual, parametrosVisao))
    retorna;

  Se (!Folha(no_atual) && AvaliaErro(no_atual, parametrosVisao))
  {
    Divide_carregamento(no_atual, parametrosVisao);
    Para cada filho de no_atual
      Percorre_carregamento(filho, parametrosVisao);
  }
}
```

### Predição de Movimento de Câmera

O objetivo da predição é fazer uso de uma posição futura de câmera para pré-carregar os dados da memória secundária para a memória principal. Dessa forma, este mecanismo pretende garantir que sempre que um dado, no caso deste trabalho um ladrilho, é necessário, ele já esteja presente em memória principal.

Existem algumas possibilidades para fazer esse carregamento prévio dos ladrilhos, como, por exemplo, utilizar um volume de visão exagerado

na linha de execução de carregamento de ladrilhos, em relação o volume de visão usado na visualização. Outra possibilidade é abrir ou fechar o volume de visão dependendo da velocidade de navegação. Neste trabalho, em vez de fazer isso, durante o procedimento de *culling* é utilizada a esfera que engloba a caixa envolvente de cada ladrilho, para assim obter uma avaliação relaxada de visibilidade. Além disso, decidiu-se fazer uma predição mais apurada do volume de visão. Foram feitas as predições de posição e direção de visualização da câmera utilizando cinemática.

Inicialmente, os parâmetros de câmera são atualizados na linha de execução de visualização. Essa atualização é repassada à linha de execução de gerenciamento, sendo então calculada a velocidade de deslocamento, levando em consideração as posições e os tempos entre atualizações consecutivas. Em seguida, é feita a predição de posição utilizando a velocidade de deslocamento calculada e fazendo uma projeção da posição atual em um tempo $T$ à frente do tempo atual. A predição de direção de visualização é feita de forma semelhante, utilizando uma cinemática aproximada de quatérnios.

Deve-se atentar para o fato de que a predição somente é feita na posição e direção de visualização, sendo que outros parâmetros, como ângulo de visão, plano *near*, plano *far*, entre outros, são assumidos como fixos. Por fim, o volume de visão futuro obtido é utilizado nos algoritmos para carregar novos ladrilhos.