3
Estrutura de Dados

Tendo em vista o que foi apresentado no capítulo anterior, a *quadtree* foi a estrutura de dados em multi-resolução escolhida neste trabalho. Neste capítulo será apresentada essa estrutura, como ela pode ser usada para representar terrenos, assim como alguns aspectos de sua implementação e construção.

3.1
Definição da Estrutura

Conforme mencionado anteriormente, a estrutura de dados escolhida como estrutura hierárquica de multi-resolução foi a árvore quaternária ou *quadtree*. Neste tipo de árvore, o domínio de cada nó é recursivamente dividido em quatro quadrantes iguais, sendo cada um deles atribuído a um dos quatro filhos conforme ilustrado na Figura 3.1.

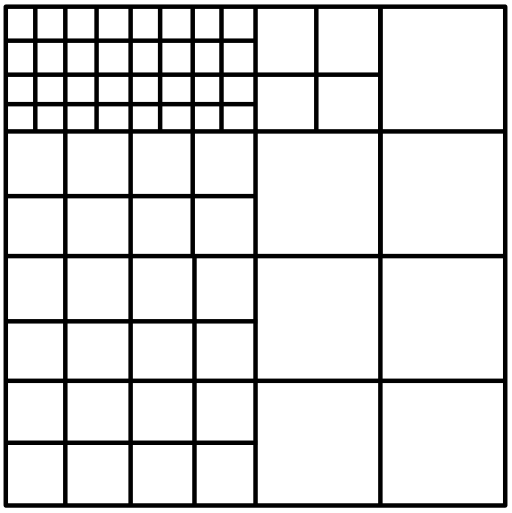


Figura 3.1: Exemplo de *quadtree*

Cada nó da árvore representa um ladrilho do terreno, sendo que cada ladrilho contém uma malha regular de triângulos de resolução fixa. Além disso, a resolução de todos os ladrilhos do terreno é sempre igual. O ladrilho

de um nó pai é formado por uma amostragem dos vértices dos filhos. Portanto, cada pai tem 25% da quantidade de vértices de seus quatro filhos juntos, formando uma decomposição hierárquica de multi-resolução.

Uma restrição da solução proposta é quanto à resolução do nível mais baixo (folhas) da *quadtree*. Ela deve ser de $2^m \times 2^m$ ladrilhos. Dessa forma, a estrutura de multi-resolução é sempre uma *quadtree* completa. Em cada nó, além dos dados de altura, estão presentes metadados que serão utilizados posteriormente para a visualização do terreno, tais como erro máximo no espaço do objeto de um ladrilho, caixa envolvente de um ladrilho, entre outros que serão apresentados posteriormente. Conforme será apresentado na Seção 4.1.4, para facilitar a eliminação de falhas entre ladrilhos existe uma restrição da resolução das malhas dos ladrilhos, que deverá ser $2^n + 1 \times 2^n + 1$.

Uma preocupação considerável em uma estrutura de multi-resolução é a quantidade adicional de memória necessária para armazenamento. No entanto, na presente proposta, a quantidade adicional de memória para armazenar esta estrutura em multi-resolução não é muito maior do que o dado original. Cada nível ocupa 1/4 do espaço em memória do nível imediatamente inferior. Dessa forma, a variação de tamanho entre níveis consecutivos ocorre conforme uma progressão geométrica de razão 1/4. Resolvendo a progressão geométrica $\left(1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \cdots = \frac{4}{3}\right)$, pode-se perceber que a estrutura em multi-resolução ocupa apenas 1/3 de espaço a mais que o dado original.

3.2 Árvore Quaternária em Vetor

Como será visto mais adiante, o algoritmo de visualização baseado em *quadtree* utiliza uma navegação complexa através dos nós. Um método eficiente de percurso desta árvore, assim como de armazenamento, é extremamente importante dado o tamanho que a árvore pode atingir em terrenos muito grandes. Isto ficará evidente, principalmente, no algoritmo descrito na Seção 4.1.3, no qual constantemente é necessário consultar nós vizinhos para garantir a ausência de falhas. Para lidar com tal problema, foi utilizado um esquema de indexação eficiente para uma *quadtree* linear (sem ponteiros), armazenada em um vetor unidimensional de nós. A implementação utilizada é bem parecida com a proposta por Balmelli *et al.* [1].

Uma *quadtree* de profundidade d contém $\sum_{i=0}^{d-1} 4^i$ nós, ou seja, o número total de nós é $\frac{1}{3}(4^d - 1)$. Dado um nó com índice $n > 0$ e uma

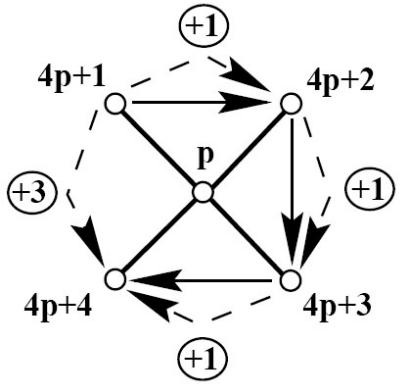
árvore com profundidade $d > 0$, pode-se estabelecer o seguinte:

$$\text{Índice de um nó pai:} \quad \left\lfloor \frac{n-1}{4} \right\rfloor \tag{3-1}$$

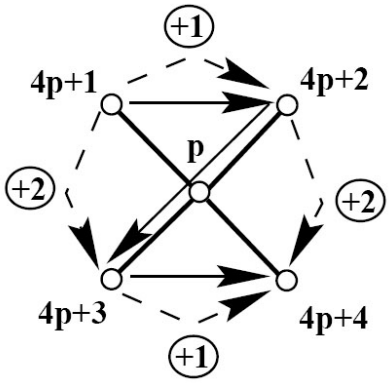
$$\text{Índice dos filhos:} \quad 4n + i, \qquad i = 1 \ldots 4 \tag{3-2}$$

$$\text{Nível de um nó na árvore:} \quad \lfloor \log_4(3n+1) \rfloor \tag{3-3}$$

Com as equações 3-1, 3-2 e 3-3, tem-se um esquema de indexação em que as relações entre pais e filhos ficam implícitas, ou seja, não é necessário armazenar ponteiros, referências ou índices para percorrer a estrutura de dados. Entretanto, a organização espacial dos nós não é revelada pelas equações acima. A Figura 3.2(a) ilustra uma possível organização espacial dos nós pais e filhos. Pode-se perceber pelas setas hachuradas que as distâncias em índices entre nós irmãos são variadas. Na organização da Figura 3.2(a), a maior distância entre nós irmãos é 3. A organização espacial escolhida foi a da Figura 3.2(b), cuja maior distância é 2. De acordo com Balmelli *et al.* [1] este tipo de organização espacial, conhecida como *z-ordering*, é melhor por garantir que a distância entre nós próximos espacialmente seja sempre a menor possível em memória.



3.2(a):



3.2(b):

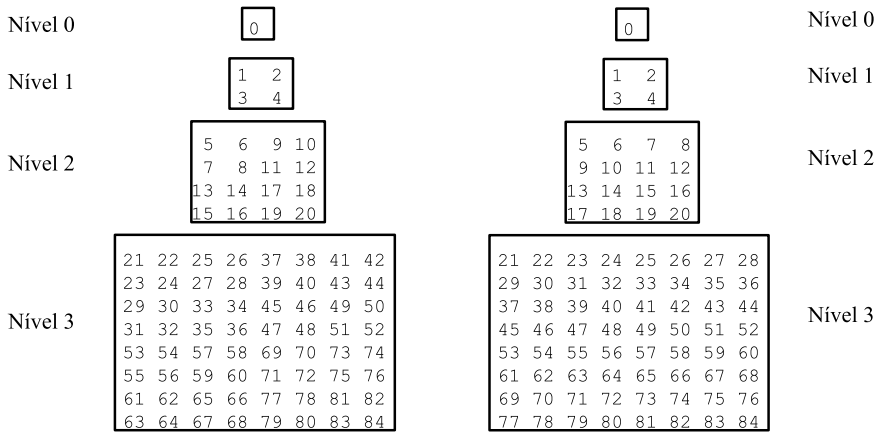
Figura 3.2: Índices dos filhos: (a) Exemplo de organização ruim; (b) Organização escolhida: z-ordering (figura extraída de Balmelli *et al.* [1])

O principal problema de usar este tipo de implementação de *quadtree* é que é sempre necessária a alocação da memória de todos os $\frac{1}{3}(4^d - 1)$ nós, mesmo que nem todos existam. No caso deste trabalho isso não ocorre, pois a árvore é sempre considerada completa.

3.2.1
Indexação de Vizinhança

O algoritmo tradicional para se obter a vizinhança de um nó em *quadtree* apresenta-se como um procedimento recursivo, cujo custo computacional é $O(d)$, sendo d a profundidade da árvore. Conforme mencionado anteriormente, o algoritmo de visualização faz consultas freqüentes às vizinhanças dos nós, portanto é necessária uma solução eficiente. Outra solução seria armazenar ponteiros ou índices explícitos para os nós vizinhos, mas isto não seria eficiente em termos de armazenamento pois seriam necessários quatro índices adicionais por nó.

O algoritmo proposto neste trabalho para resolução de vizinhança é um meio-termo dos algoritmos acima. Observando-se a Figura 3.3(a) podem-se ver os primeiros quatro níveis da *quadtree* de ladrilhos, utilizando a indexação que segue a organização espacial *z-ordering*. Os números apresentados em cada nível representam o índice no vetor unidimensional em que está localizado cada ladrilho. Pode-se observar que nesse tipo de indexação a obtenção dos vizinhos de um nó não é trivial. Já em uma indexação matricial como a apresentada na Figura 3.3(b), é possível obter os vizinhos de um nó de forma trivial. A idéia deste algoritmo é utilizar dois vetores de conversão, uma da indexação matricial para a *z-ordering* e outra que faz a conversão inversa.



3.3(a): 3.3(b):

Figura 3.3: Matrizes de índices: (a) Indexação no formato *z-ordering*; (b) Indexação no formato matricial (os números representam o índice no vetor unidimensional em que está localizado cada ladrilho)

Com os dois vetores de conversão, podem-se achar os seus vizinhos de uma forma simples. Inicialmente, dado um nó de índice n na indexação *z-ordering*, utiliza-se o vetor de conversão para obter o índice da indexação

matricial do nó em questão. Dado que esse nó pertence a um nível k, deve-se computar a sua posição local (i, j) no nível, que é de resolução $2^k \times 2^k$. Esta posição pode ser calculada fazendo $n_{local} = n - \frac{1}{3}(4^k - 1)$. Com essa posição local, calcula-se os vizinhos à esquerda e à direita, respectivamente, subtraindo ou somando 1 à posição local. Para os vizinhos superior e inferior, basta subtrair ou somar o tamanho de uma linha na matriz local, ou seja, 2^k. Casos de bordas são facilmente tratados utilizando a posição local. Por fim, descoberto o índice vizinho na indexação matricial, basta utilizar o vetor de conversão para obter o índice na indexação *z-ordering*.

O algoritmo, portanto, apresenta um custo de armazenamento de duas vezes o tamanho da árvore, contra quatro da indexação explícita de vizinhos, e ainda tem um custo computacional baixo. Em seguida é apresentado o pseudo-código para a construção das duas matrizes de conversão descritas acima:

```
ConstroiIndices(idx_matricial, idx_zorder, nivel)
{
  conversao_matricial_para_zorder[idx_matrix] = idx_zorder;
  conversao_zorder_para_matricial[idx_zorder] = idx_matrix;

  Se (nivel >= (profundidade-1))
    retorna;

  // Calculando coordenadas locais
  tamanho_nivel = pow(2, nivel);
  j = (idx_matrix - num_nos(nivel))/tamanho_nivel;
  i = (idx_matrix - num_nos(nivel))%tamanho_nivel;

  // Calculando filhos matriciais
  tamanho_proximo_nivel = 2*tamanho_nivel;
  num_nos_proximo_nivel = num_nos(tamanho_proximo_nivel);
  nw_m = num_nos_proximo_nivel + 2*(j*tamanho_proximo_nivel + i);
  ne_m = num_nos_proximo_nivel + 2*(j*tamanho_proximo_nivel + i) + 1;
  sw_m = num_nos_proximo_nivel + 2*(j*tamanho_proximo_nivel + i) +
         tamanho_proximo_nivel;
  se_m = num_nos_proximo_nivel + 2*(j*tamanho_proximo_nivel + i) +
         tamanho_proximo_nivel + 1;

  // Continua a construção recursivamente
  ConstroiIndices(nw_m, Filho_NW(idx_zorder), nivel+1);
  ConstroiIndices(ne_m, Filho_NE(idx_zorder), nivel+1);
  ConstroiIndices(sw_m, Filho_SW(idx_zorder), nivel+1);
  ConstroiIndices(se_m, Filho_SE(idx_zorder), nivel+1);
  retorna;
}
```

3.3 Pré-processamento

O objetivo da etapa de pré-processamento é fazer todo o processamento que independe dos parâmetros de câmera. Assim, pode-se obter um melhor desempenho durante a execução do algoritmo de visualização. Geralmente as etapas de pré-processamento são custosas e geram um grande volume de dados. Este trabalho teve como um dos objetivos principais criar uma etapa de pré-processamento oposta a essas características devido ao grande volume de dados que pode ser usado para representar um terreno.

Para o algoritmo de visualização proposto neste trabalho, que será apresentado no capítulo 4, a etapa de pré-processamento consiste basicamente na construção da estrutura de multi-resolução, fazendo cálculo do erro máximo no espaço do objeto e da caixa envolvente de cada ladrilho. Além disso, é necessária uma forma de persistência do dado pré-processado.

3.3.1 Construção da Quadtree

A estrutura de multi-resolução é construída através de um procedimento feito de baixo para cima. Inicialmente, divide-se o terreno em ladrilhos de resoluções iguais, sendo esta resolução na forma $2^n + 1 \times 2^n + 1$. Os vértices nas bordas dos ladrilhos são replicados em ladrilhos vizinhos. A quantidade de ladrilhos nas duas dimensões deve ser de $2^m \times 2^m$, conforme mencionado anteriormente, para que se possa formar uma *quadtree* completa.

Em seguida, cada ladrilho pai é construído fazendo uma amostragem dois a dois em linha e em coluna das alturas dos filhos, conforme ilustrado na Figura 3.4. Assim, quatro nós filhos dão origem a um nó pai e o procedimento segue de forma recursiva. Neste ponto, calcula-se também a caixa envolvente do pai como sendo a caixa que envolve as caixas envolventes de seus quatro filhos.

3.3.2 Erro no Espaço do Objeto

Muitos trabalhos utilizam a caixa envolvente como uma estimação máxima do erro no espaço do objeto. A decisão de fazer um cálculo mais preciso desse erro foi tomada pois tinha-se como objetivo obter uma malha

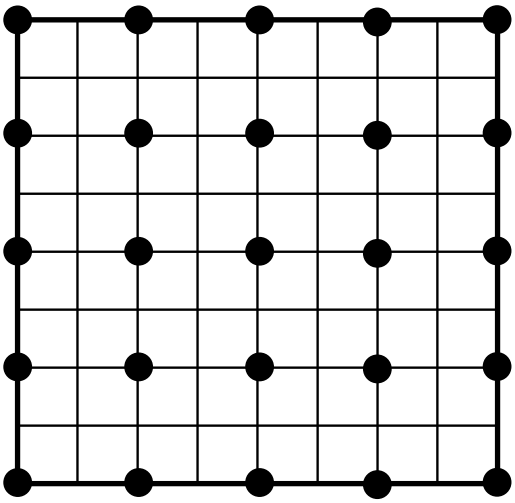


Figura 3.4: Amostragem dois a dois

melhor adaptada à rugosidade do terreno, evitando o uso excessivo de triângulos em regiões planas ou cujas diferenças de altura fossem pequenas.

Primeiramente, os nós folhas da *quadtree* possuem erro sempre zero. Após o processo de amostragem para a construção de um nó pai, é feito o cálculo do erro para este ladrilho. Este cálculo leva em consideração todas as possíveis triangulações que podem ser feitas para desenhar um ladrilho, desde que essa triangulação contenha todos os vértices do mesmo. Observando a Figura 3.4, pode-se classificar os vértices eliminados conforme a orientação da aresta na malha do novo nível que passa pelo vértice que foi eliminado. São três categorias: verticais, horizontais e diagonais.

Os vértices eliminados verticais são os vértices beges ilustrados na Figura 3.5(a). Conforme pode-se ver nesta figura, existe uma única aresta de qualquer possível triangulação que passará pelo vértice C. Esta aresta é a que liga os vértices A e B. Desta forma, calcula-se o erro cometido pela remoção do vértice C conforme mostrado pela equação 3-4. De forma análoga é ilustrado na Figura 3.5(b) o caso dos vértices eliminados horizontais, que estão representados na cor vermelha. No caso dos vértices eliminados diagonais, que aparecem na cor azul, existem duas possíveis triangulações. Pode-se observar nas Figuras 3.5(c) e 3.5(d) as duas triangulações ABC, sendo D o vértice eliminado em questão. O cálculo do erro do vértice D neste caso deve ser feito usando as arestas que ligam os vértices A e C nas Figuras 3.5(c) e 3.5(d), sendo que o valor máximo desses dois erros é atribuído ao vértice D.

$$\text{ErroObj} = \left|\text{altura(C)} - \frac{\text{altura(A)} + \text{altura(B)}}{2}\right| \qquad (3\text{-}4)$$

O cálculo de erro apresentado acima é feito para todos os vértices eliminados, sendo o maior desses erros atribuído como erro máximo do ladrilho no espaço do objeto.

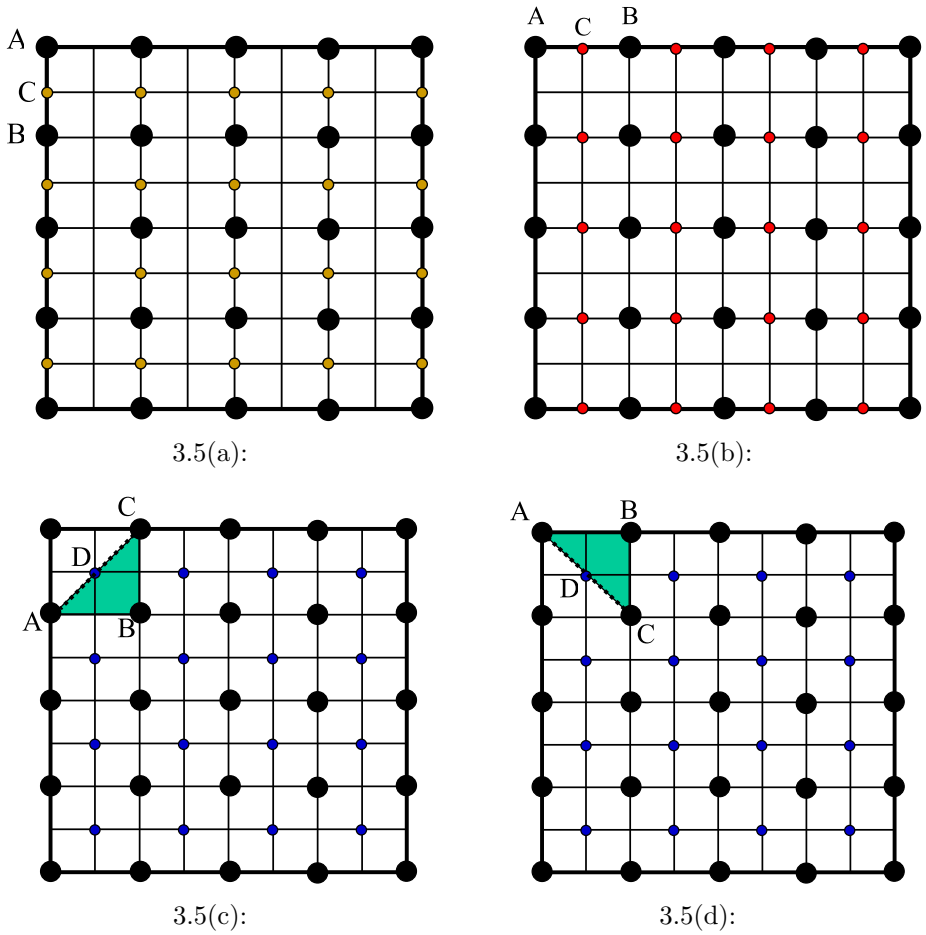


Figura 3.5: Cálculo do erro: (a) Vértice eliminado vertical; (b) Vértice eliminado horizontal; (c) Primeira possível triangulação para vértice eliminado diagonal; (d) Segunda possível triangulação para vértice eliminado diagonal

Por fim, é necessário garantir que os erros no espaço do objeto estejam aninhados entre os nós filhos e pais. Este aninhamento significa que os erros máximos dos nós filhos devem ser sempre menores ou iguais ao erro máximo de seus pais. Isto se justifica pela seguinte situação: dado um nó, suponha que algum dos vértices eliminados para a sua formação seja, por exemplo, o vértice que acarretava o maior erro. Como este vértice já foi eliminado, ele não afetará o cálculo do erro no pai deste nó, que pode ser portanto menor do que o erro do filho.

3.3.3 Persistência em Memória Secundária

Um processo de persistência dos dados de pré-processamento é necessário por dois motivos principais: evitar que o pré-processamento seja executado toda vez que se deseja visualizar o terreno e permitir uma construção escalável do dado pré-processado. Conforme o método de construção apresentado na Seção 3.3.1, quando um nó pai é criado com todos os seus dados e metadados, os nós filhos não precisam mais estar em memória principal para a continuação do processo de construção. Assim, os filhos podem ser salvos em disco e removidos da memória principal. A persistência em memória secundária é feita utilizando dois tipos de arquivos, um contendo metadados e outro, dados. Para um terreno pré-processado existe somente um arquivo de metadados e um ou mais arquivos de dados.

O arquivo de metadados contém todos os dados necessários para a reconstrução de cada nó da *quadtree*, exceto as alturas dos vértices dos ladrilhos. A posição dos nós no arquivo de metadados segue a organização espacial *z-ordering* apresentada na Seção 3.2. Os dados para cada nó são: o erro máximo no espaço do objeto; a caixa envolvente dos vértices; uma referência para o arquivo onde estão localizados os dados de altura; e a posição ou *offset* em *bytes* dos dados no arquivo de dados.

O arquivo de dados contém as alturas dos vértices dos ladrilhos. A ordenação dos ladrilhos é feita nível a nível, e também segue a organização espacial *z-ordering* mencionada anteriormente. É possível também fazer a compressão dos dados de altura. Sem compressão, como o tamanho dos dados de altura é fixo, nenhuma informação adicional sobre quanto deve ser lido a partir do *offset* é necessária. Quando a compressão é usada, é necessário indicar nos primeiros 4 *bytes* do *offset* o tamanho do dado comprimido. A biblioteca de compressão utilizada foi a *LZO* [15]. Ela foi escolhida devido ao seu alto desempenho na descompressão, característica ideal para algoritmos de interação em tempo real.