**Thiago Ramos dos Santos**

# *Representação e Visualização Tridimensional de Imagens Paralelamente Alinhadas usando* **Quadtrees** *e* **Octrees**

Florianópolis

2003

**Thiago Ramos dos Santos**

# *Representação e Visualização Tridimensional de Imagens Paralelamente Alinhadas usando* **Quadtrees** *e* **Octrees**

Trabalho de Conclusão de Curso do curso de Ciências da Computação, realizado no Laboratório de Integração de Software e Hardware (LISHA) do Departamento de Informática e Estatística (INE) da Universidade Federal de Santa Catarina (UFSC), sob orientação do Prof. Dr. rer. nat. Aldo von Wangenheim.

Orientador:
Aldo von Wangenheim, Prof. Dr. rer. nat.

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Florianópolis

2003

Trabalho de Conclusão de Curso do curso de Ciências da Computação da Universidade Federal de Santa Catarina, sob o título "Representação e Visualização Tridimensional de Imagens Paralelamente Alinhadas usando Quadtrees e Octrees", defendido por Thiago Ramos dos Santos e aprovado em fevereiro de 2004, em Florianópolis, Santa Catarina, Brasil, pela banca examinadora constituída pelos senhores:

Prof. Dr. rer. nat. Aldo von Wangenheim
Orientador
Universidade Federal de Santa Catarina

Prof. Dr. Antônio Augusto M. Fröhlich
Universidade Federal de Santa Catarina

MsC. Harley Miguel Wagner, Doutorando
Universidade Federal de Santa Catarina

MsC. Leonardo Andrade Ribeiro
*The Cyclops Project*

Daniel Duarte Abdala, Mestrando
Universidade Federal de Santa Catarina

# *Dedicatória*

Aos meus pais: Euclydes e Maristela.

Aos avós: Euclydes, Vilma, Reynaldo e Acedilte.

Aos grandes amigos de Florianópolis: Davis, Fedro e Pelotas.

Aos grandes amigos de Rio Grande: Alemão, Bambo, Biru, Bonito, Casão, Cueca, DC, Digo, Jonatas, Orc Jr. e Zé.

À família CCO992.

# *Agradecimentos*

Ao pessoal do LISHA, principalmente ao Vô,

Caju, Harley, Leo, Ester, Guto e Aldo.

# *Resumo*

Imagens paralelamente alinhadas têm aplicações em diversas áreas, principalmente na área médica, aonde visam mostrar detalhes internos de uma determinada área do corpo humano. Tais imagens, no entanto, apresentam certas limitações, como a falta de noção volumétrica da estrutura que representam e da impossibilidade de visualização desta por planos diferentes daqueles em que as imagens foram geradas. Uma representação tridimensional de tais imagens, através de estruturas de dados bem conhecidas, como as *octrees*, permitem que tais limitações sejam superadas e possibilitam que se realizem novas operações com tais imagens. Este trabalho demonstra métodos de criação de tais estruturas de forma a representar imagens paralelamente alinhadas e como renderizar estas de forma eficiente.

**Palavras-chave: Estruturas de Dados Espaciais, Imagens Paralelamente Alinhadas, Reconstrução Tridimensional.**

# *Abstract*

Parallel aligned images have many fields of applications, mainly in the medical field, where such images are used for visualization of internal details of some part of the human body. However, those images have certain limitations, like the leak of volumetric notion of structure being represented and the impossibility of visualization of the structure throw some plan which is not the one in which the images were taken. A tridimensional representation of those images, through well-know data structures, like the octrees, allow the surpassing of those limitations and make possible new operations to be performed. This work shows mothods for construction of such structures in order to represent parallel aligned images and how to render them efficiently.

**Key-words: Spatial Data Structures, Parallel Aligned Images, Tridimensional Reconstruction.**

# *Sumário*

**Lista de Figuras**

**1 Introdução** p. 11

1.1 Objetivo Geral . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 12

1.2 Objetivos Específicos . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 12

**2 Imagens Paralelamente Alinhadas** p. 13

**3 Estruturas de Dados para Representação de Imagens** p. 15

3.1 *Quadtrees* e *Octrees* . . . . . . . . . . . . . . . . . . . . . . . . . . p. 15

3.2 *Quadtrees Volumétricas* . . . . . . . . . . . . . . . . . . . . . . . . . p. 18

3.2.1 Adaptação de Métodos Aplicáveis à *Quadtrees* e *Octrees* às *Quadtrees Volumétricas* . . . . . . . . . . . . . . . . . . . . . . . . p. 19

**4 Construção de Estruturas para Representação Tridimensional a partir de *Quadtrees*** p. 21

4.1 Construção de *Quadtrees Volumétricas* . . . . . . . . . . . . . . . . . . p. 21

4.2 Construção de *Octrees* . . . . . . . . . . . . . . . . . . . . . . . . . . p. 23

4.3 Análise entre os Métodos de Criação das *Quadtrees Volumétricas* e *Octrees* . p. 29

**5 Renderização das Estruturas Tridimensionais** p. 31

5.1 Acelerando a Renderização . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 31
5.1.1 Renderização com Observador Externo . . . . . . . . . . . . . . . . . p. 32
5.1.2 Renderização com Observador Interno . . . . . . . . . . . . . . . . . . p. 36
5.2 Operações para Renderização . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 37

**6 Trabalhos Futuros** p. 39

**7 Conclusão** p. 40

**Referências** p. 41

**Anexo A – Representação por *Octree* de uma Superfície Não-plana** p. 43

**Anexo B – Artigo Submetido ao CBMS 2004** p. 45

**Anexo C – Diagrama de Classes** p. 49

**Anexo D – Código-fonte** p. 50

# *Lista de Figuras*


1 Processo de aquisição de imagens tomográficas, retirada de Sprawls (2004). . . p. 14

2 Conjunto de imagens paralelamente alinhadas adquiridas por tomografia, retirada de Sprawls (2004). . . . . . . . . . . . . . . . . . . . . . . . . . . p. 14

3 Representação de uma imagem binária por uma *quadtree*, retirada de Samet (1984). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 16

4 Processo de criação de uma *octree*, retirado de Wagner (2001). . . . . . . . . p. 17

5 Representação numérica de quadrantes e octantes. . . . . . . . . . . . . . . . p. 18

6 Conjunto de imagens paralelamente alinhadas com indicação do valor da coordenada z. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 22

7 Exemplos de atribuição de alturas às *quadtrees*. . . . . . . . . . . . . . . . . p. 22

8 Processo de criação de uma *octree*, visto lateralmente. . . . . . . . . . . . . . p. 24

9 Divisões das *quadtrees* usadas no exemplo da figura 8. . . . . . . . . . . . . p. 25

10 Métodos para construção da *octree*. . . . . . . . . . . . . . . . . . . . . . . p. 28

11 Representações por *octree* de *quadtrees* com apenas um nodo. . . . . . . . . p. 30

12 Renderização de uma *octree* representando o crânio humano. . . . . . . . . . p. 32

13 Métodos para eliminação de cubos e faces não visíveis. . . . . . . . . . . . . p. 35

14 Volumes de renderização em projeções paralelas e em perspectiva. . . . . . . p. 36

15 Testes de intersecção em árvore com observador interno. . . . . . . . . . . . p. 37

16 Uma superfície não-plana. . . . . . . . . . . . . . . . . . . . . . . . . . . p. 43

17 Renderização da representação por *octree* da superfície apresentada na figura 16. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 44

18 Artigo CBMS 2004 - Página 1 . . . . . . . . . . . . . . . . . . . . . . . . . p. 45

19 Artigo CBMS 2004 - Página 2 . . . . . . . . . . . . . . . . . . . . . . . . . p. 46

20 Artigo CBMS 2004 - Página 3 . . . . . . . . . . . . . . . . . . . . . . . . . p. 47

21 Artigo CBMS 2004 - Página 4 . . . . . . . . . . . . . . . . . . . . . . . . . p. 48

22 Diagrama de classes da implementação mostrada no anexo D. . . . . . . . . p. 49

# 1 Introdução

Imagens paralelamente alinhadas são, hoje em dia, utilizadas em diversas áreas, visando prover conhecimento sobre o conteúdo interno de uma determinada estrutura. Na medicina, exemplos de imagens paralelamente alinhadas são as tomografias computadorizadas e as ressonâncias magnéticas.

No entanto, imagens paralelamente alinhadas fazem com que se perca a noção de volume de uma estrutura, obrigando aquele que analisa as imagens a tentar encaixar, mentalmente, uma a outra. Além disto, a visualização da imagem por outros ângulos e planos, além daqueles na qual foi realizada a amostragem, é impossível.

Uma forma de representação tridimensional do conteúdo de um conjunto de imagens paralelamente alinhadas possibilitaria a visualização volumétricas destes, possibilitando, desta maneira, uma análise mais intuitiva dos dados, sem a necessidade de visualizações mentais. Dependendo da estrutura de dados selecionada para a representação, outras formas de visualização também podem ser empregadas, como a eliminação de certas partes do conteúdo das imagens.

Este trabalho visa obter a representação tridimensional de imagens paralelamente alinhadas utilizando de estruturas de dados bem conhecidas como as *quadtrees* e *octrees* e métodos eficientes de renderização destas. Nesta trabalho também é apresentada uma estrutura simples e rápida para se obter uma representação tridimensional, denominada aqui de *quadtrees volumétricas*, juntamente com algumas adaptações dos métodos de operações em *quadtrees* e *octrees* propostos por outros autores.

## 1.1 Objetivo Geral

Obter uma representação tridimensional de imagens paralelamente alinhadas utilizando *quadtrees* e *octrees* e métodos eficientes para representação destas.

## 1.2 Objetivos Específicos

Como objetivos específicos este trabalho assume:

- Mostrar a definição, utilizada neste trabalho, de imagens paralelamente alinhadas.
- Apresentar, brevemente, as estruturas de representação de imagens (*quadtree*, *octree* e *quadtree volumétrica*) e como operá-las.
- Demonstrar a criação de *octrees* e *quadtrees volumétricas* a partir de um conjunto de imagens representadas por *quadtrees* .
- Demonstrar métodos eficientes para renderização das estruturas de dados para representações tridimensionais.

# 2 Imagens Paralelamente Alinhadas

São consideradas imagens paralelamente alinhadas, no escopo deste trabalho, toda seqüencia de imagens bidimensionais que representa uma parte de uma determinado volume, sendo estas obrigatoriamente paralelas e possuem suas coordenadas alinhadas. Simplificando, em um determinada região tridimensional representada pelos eixos X, Y e Z, todas as imagens paralelamente alinhadas possuem as mesmas coordenadas nos eixos X e Y, porém cada uma delas está numa diferente coordenada no eixo Z.

Um exemplo de imagens paralelamente alinhadas são as imagens tomográficas, aonde um conjunto de imagens bidimensionais visa representar o volume de uma determinada região de uma determinada estrutura, mais comumente, do corpo humano.

A figura 1 mostra o processo de aquisição das imagens tomográficas. O resultado do processo de aquisição pode ser visto na figura 2. Pode-se observar que cada uma das imagens representa uma determinada área e são paralelas umas as outras.

Da união das imagens paralelamente alinhadas espera-se, então, a representação tridimensional da estrutura representada por uma seqüencia bidimensional. No entanto, nem sempre todas as áreas são representadas pelas imagens, havendo entre elas certos espaços. Em tomografias computadorizadas, estes espaços variam entre 3 e 8 milímetros. Uma representação tridimensional de imagens paralelamente alinhadas precisa preencher estes espaços de forma a representar de forma mais próxima possível as formas originais que, posteriormente, foram armazenadas de forma parcial nas imagens.

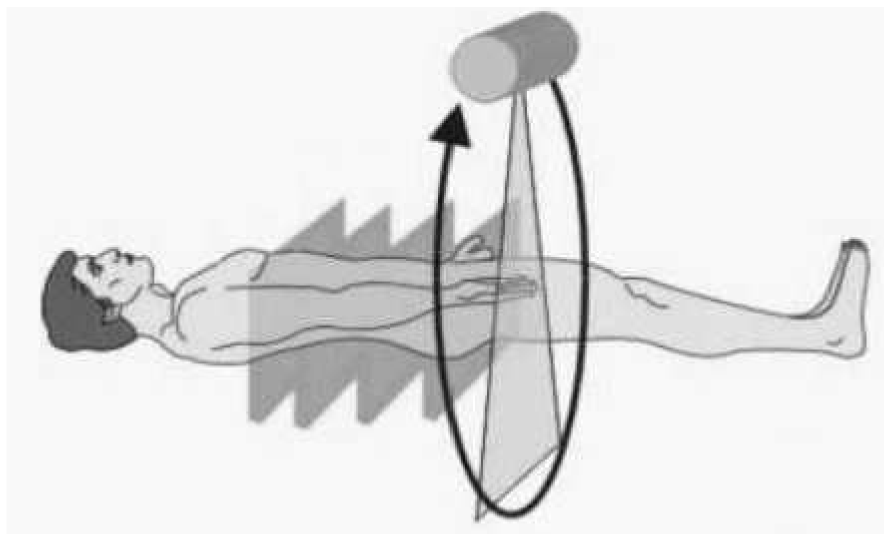

Figura 1: Processo de aquisição de imagens tomográficas, retirada de Sprawls (2004).

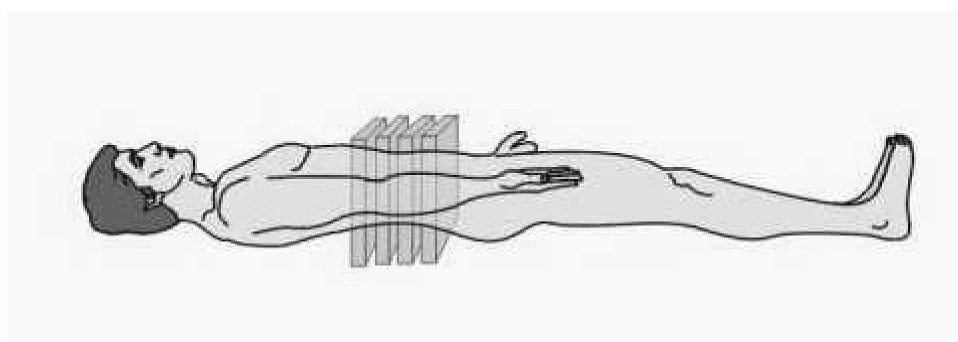

Figura 2: Conjunto de imagens paralelamente alinhadas adquiridas por tomografia, retirada de Sprawls (2004).

# 3 *Estruturas de Dados para Representação de Imagens*

Existem inúmeras estruturas de dados para representação de imagens, sendo as *quadtrees*, para representações bidimensionais, e as *octrees*, para representações tridimensionais, as mais comumente utilizadas. Neste trabalho serão abordadas apenas estas e as *quadtrees volumétricas*. Outras estruturas de dados são abordadas por Zachmann e Langetepe (2003) e Samet (1989a).

Estruturas de dados para representação de imagens apresentam algumas vantagens em relação a estas, sendo a mais importante a possibilidade de se fazer testes hierarquicamente nas regiões representadas, muitas vezes reduzindo a quantidade de tarefas necessárias para se realizar determinada atividade. Um exemplos de otimizações obtidas através do uso de árvores podem ser vistos na seção 5.1.

## 3.1 *Quadtrees* e *Octrees*

*Quadtrees* e *octrees* são estruturas de dados que subdividem recursivamente um certo espaço até que todos os elementos destas subdivisões espaciais tenham um determinado número de primitivas em comum (SAMET, 1989a). Neste trabalho, se utiliza como primitiva a ser verificada a variação entre os valores de *pixel* da imagem a ser representada.

No caso de uma imagem bidimensional, o processo de criação de uma representação utilizando uma *quadtree* começa por atribuir à raiz da árvore a largura e altura da imagem. Caso a variação entre os valores de *pixel* nesta região estejam estejam abaixo de um determinado valor de tolerância, o processo termina, caso contrário esta área é subdividida em quatro partes

iguais, sendo cada uma destas áreas representadas por um nodo filho. Este mesmo processo de verificação é aplicado aos nodos filhos, até que as áreas representadas por cada nodo possuam variações de valores de *pixel* abaixo da tolerância. Aos nodos folhas, então, serão atribuídos valores de *pixel*, e, através da renderização destes, a imagem é recomposta.

A figura 3 mostra uma *quadtree* criada a partir de uma imagem binária, aonde (a) mostra o formato da região, (b) é a representação binária da região, (c) demonstra a decomposição em blocos de (b) e (d) é a representação por *quadtree* dos blocos em (c).

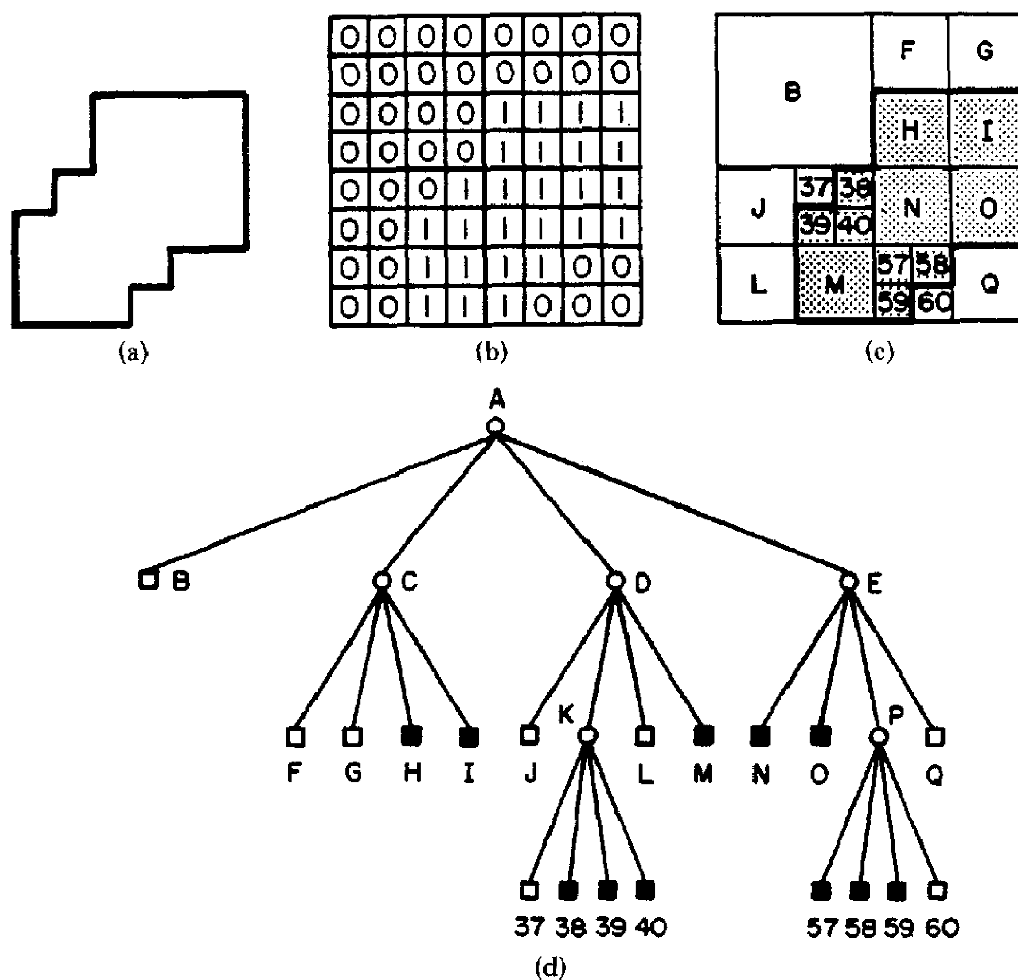


Figura 3: Representação de uma imagem binária por uma *quadtree*, retirada de Samet (1984).

O processo de criação de uma *octree* segue o mesmo processo que o da *quadtree*, no entanto, cada porção do volume é dividida em oito paralelepípedos. O processo de criação de uma representação por *octree* de um determinado volume é mostrado na figura 4.

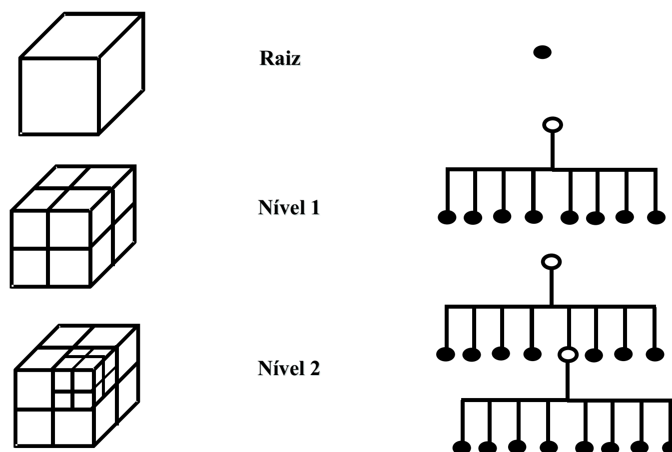


Figura 4: Processo de criação de uma *octree*, retirado de Wagner (2001).

Como nas *quadtrees*, a construção de octrees consiste representar uma determinada região pelo nodo da árvore e, recursivamente, ir subdividindo este nodo até que se satisfaçam as premissas. No entanto, a cada nível da recursão, os nodos são divididos em oito partes iguais. Com a renderização de todas as folhas da *octree*, assim como nas *quadtrees*, tem-se novamente a imagem.

No anexo A é mostrada uma superfície e sua representação por *octree*.

Métodos para buscar regiões vizinhas em representações por *quadtrees* e *octrees* são apresentados por Samet (1982) e Samet (1989b), respectivamente. Bhattacharya (2001) e Frisken e Perry (2002) propõem métodos mais eficientes para a busca de vizinhos através da indexação dos nodos.

Uma análise da aproximação de formas por *quadtrees* é mostrada por Ranade, Rosenfeld e Samet (1982). Lin e Wong (1996) demonstra como realizar operações morfológicas nas imagens representadas por estas.

Neste trabalho é utilizada a notação numérica para a indicação de quadrantes e octantes nos nodos das *quadtrees* e *octrees,* respectivamente. Esta notação segue conforme a indicada na figura 5. Esta decisão facilita o acesso aos filhos de um determinado nodo, podendo ser acessados via *n.FILHOS[i]*, aonde *n* representa um nodo de uma das árvores e *i* o quadrante ou

octante que se deseja acessar.

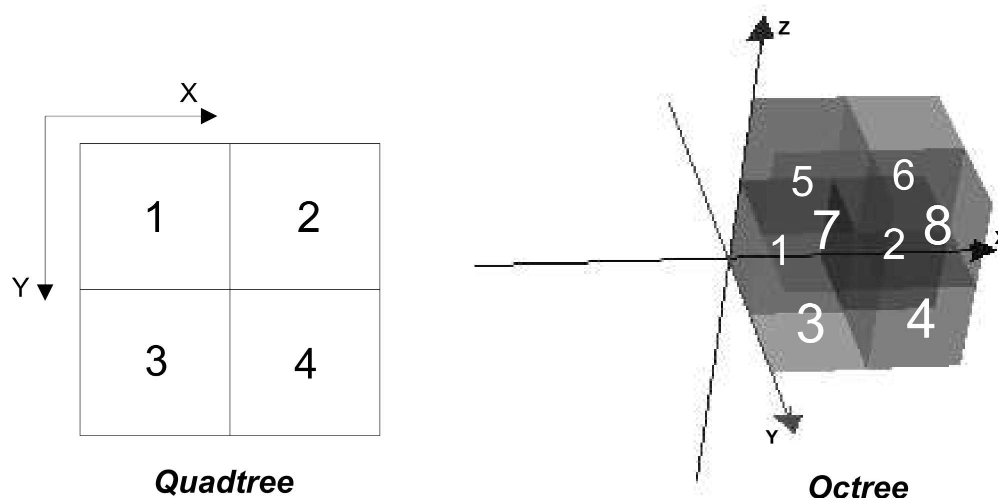


Figura 5: Representação numérica de quadrantes e octantes.

Caso o valor de *n.FILHOS[i]* seja *NULO*, não existe nada a ser representado no quadrante ou octante *i* do nodo *n*. A inclusão de filhos com valores *NULO* é indicada no caso de não ser necessário representar com valores de *pixel* todas as regiões de uma determinada imagem, ou seja, existem áreas que não precisam ser representadas. Como exemplo pode ser tomada as tomografias computadorizadas, aonde a região escura em volta da estrutura submetida ao exame não precisa, necessariamente, participar da reconstrução tridimensional.

## 3.2 *Quadtrees Volumétricas*

*Quadtrees volumétricas* são uma variação das *quadtrees* normais, mas que representam um espaço tridimensional. A única diferença entre elas é que os nodos da primeira são paralelepípedos, enquanto que os nodos da segunda são quadrados. A transformação de uma para outra é direta e está explicada na seção 4.1.

As *quadtrees volumétricas* foram criadas para possibilitar, rapidamente, uma visualização tridimensional de imagens paralelamente alinhadas, através de suas representações por *quadtrees* convencionais. Importante notar que, para uma representação tridimensional por este mé-

todo não está envolvida apenas uma árvore, mas sim tantas quantas forem as imagens. Todas as *quadtrees volumétricas* necessárias para representar um volume formam um conjunto ordenado por sua posição no eixo Z.

Algumas vezes menciona-se *quadtrees volumétricas* referindo-se ao conjunto ordenado destas. Para não haver confusões, pode-se supor a existência de uma raiz única, igual a de uma *octree*, com a única diferença que não tem apenas oito filhos, mas sim tantos quantos forem as árvores. Cada filho desta grande raiz aponta para a raiz de uma das *quadtrees volumétricas*.

### 3.2.1 Adaptação de Métodos Aplicáveis à *Quadtrees* e *Octrees* às *Quadtrees Volumétricas*

Em Samet (1982) e Samet (1989b) são definidos métodos para busca de vizinhos de um determinado nodo e alguns métodos utilizados nesta busca, sendo que alguns destes são utilizados na seção 5.1 (em Bhattacharya (2001) são definidos outros métodos, no entanto não há a preocupação de adaptá-los às *quadtrees volumétricas*, já que estes métodos requerem indexação de nodos, que não está sendo utilizada).

No entanto, não é possível que os métodos definidos para as *octrees* sejam utilizados diretamente, pois os nodos das *quadtrees volumétricas* não possuem oito filhos, mas sim quatro. Quanto aos métodos definidos às *quadtrees*, estes também não podem ser diretamente utilizados, já que um nodo de *quadtree* não possui vizinhos acima ou abaixo.

Serão adaptados os métodos utilizados na seção 5.1, sendo estes:

- OT_GTEQ_FACE_NEIGHBOR($p$, $i$): Este método é responsável por retornar o vizinho de face da face $i$ do nodo $p$ de igual ou maior tamanho, em uma octree. Para que seja usado em uma *quadtree volumétrica* pode-se utilizar o método GTEQUAL_ADJ_NEIGHBOR (SAMET, 1982) caso a face em questão seja da frente, de trás, da esquerda ou da direita (preocupando-se com a notação, já que Samet utiliza estas mencionadas para as *octrees*, mas os pontos cardeais para as *quadtrees*, ou seja, deve-se correlacionar frente com norte, trás com sul, etc...). Se desejar-se saber o vizinho de cima ou abaixo, basta buscar na

*quadtree volumétrica* acima ou abaixo o nodo na mesma posição e de nível menor ou igual ao do nodo em questão. Caso não exista tal nodo, o retorno é *NULO*.

- ADJ($i$, $o$): Retorna *VERDADE* se o octante $o$, no caso de uma *octree*, ou quadrante $o$, no caso de uma *quadtree*, é adjacente à face ou aresta $i$. Adaptando-o às *quadtrees volumétricas*, se a face em questão for da frente, de trás, da esquerda ou da direita, pode-se utilizar o mesmo método ADJ proposto por (SAMET, 1982) (preocupando-se com a notação, conforme anteriormente mencionado). Se a face em questão for a de cima ou de baixo, a resposta é sempre *VERDADE*, já que todos os quatro nodos compõem estas faces.

# 4 *Construção de Estruturas para Representação Tridimensional a partir de* Quadtrees

Uma representação tridimensional de imagens paralelamente alinhadas visa restituir o aspecto volumétrico que é perdido no armazenamento de dados visuais de um determinado objeto através de uma seqüencia de imagens bidimensionais. Grande parte dessa restituição depende da atribuição de valores aos espaços entre as imagens, cujos dados são ignorados.

Partindo de um conjunto de *quadtrees*, cada uma delas sendo uma representação de cada uma das imagens paralelamente alinhadas, será apresentado o processo de construção de *octrees* e de *quadtrees volumétricas*.

Supõe-se que, para cada uma das imagens, a coordenada z destas é conhecida, como mostra a figura 6.

## 4.1 Construção de *Quadtrees Volumétricas*

Para a construção de *quadtrees volumétricas* basta assumir que cada uma das regiões retangulares representadas pelos nodos das *quadtrees* é um paralelepípedo. Ou seja, será atribuída uma altura aos nodos das *quadtrees*, constante para todos os nodos de uma mesma árvore.

Cada altura deve ser atribuída de forma a eliminar totalmente o espaço entre uma *quadtree* e outra. Os nodos que representam os paralelepípedos criados herdam o valor de *pixel* dos nodos originais das *quadtrees*. A figura 7 exemplifica três maneiras de atribuição desta altura.

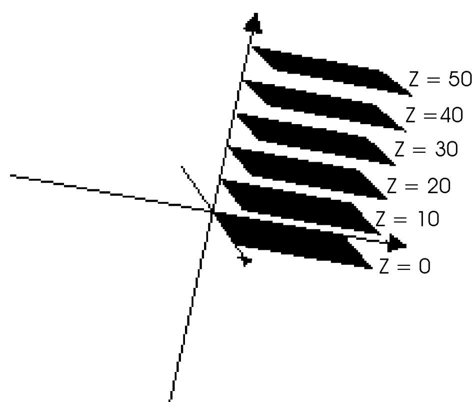


Figura 6: Conjunto de imagens paralelamente alinhadas com indicação do valor da coordenada z.

Importante notar que não foram adotadas regras fixas para atribuição das alturas, sendo que novas formas de atribuição podem ser adotadas conforme o caso.

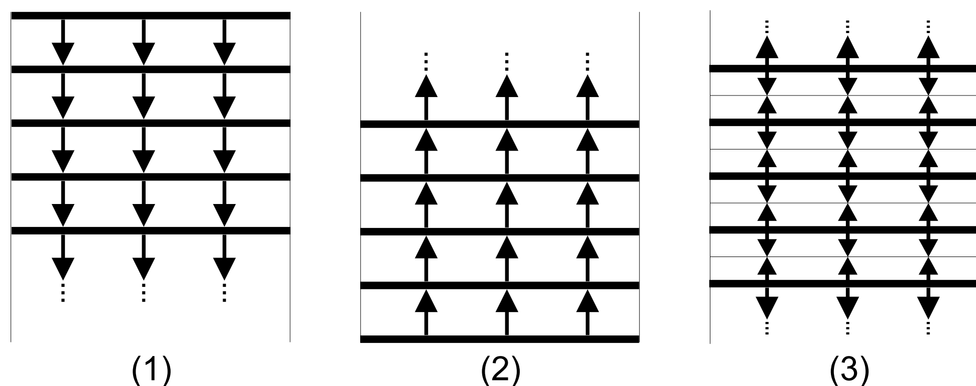


Figura 7: Exemplos de atribuição de alturas às *quadtrees*.

As duas primeiras maneiras mostram a adoção de uma das *quadtrees* como limite do volume, e a altura é atribuída até preencher o espaço entre a próxima *quadtree*. À última *quadtree* do volume pode-se atribuir uma altura arbitrária ou adotá-la, também, como um limite do volume. O terceiro exemplo mostra a atribuição de altura de forma a fazer com que os paralelepípedos gerados a partir de cada *quadtree* preencha metade do espaço entre uma imagem e outra. Novamente, a primeira e última *quadtrees* podem ser adotadas como limites do volume

ou pode-se atribuir uma altura arbitrária à elas.

Ao final do processo de atribuição de altura o resultado é um conjunto ordenado de *quadtrees volumétricas* sem espaços entre cada árvore subseqüente.

## 4.2 Construção de *Octrees*

Como as *octrees* consistem na subdivisão do espaço em paralelepípedos, ou seja, subdivisões do espaço nos três eixos (tridimensional), e as *quadtrees* fornecem a subdivisão do espaço bidimensional, para a construção da primeira só falta a subdivisão em um dos eixos (o eixo Z). Porém, como os dados entre uma *quadtree* e outra (uma imagem e outra) não são conhecidos, algumas regras devem ser adotadas para a subdivisão deste espaço.

O processo começa supondo que todas as *quadtrees* estão dentro de um mesmo paralelepípedo, de forma que a base e o topo deste paralelepípedo sejam a primeira e a última *quadtrees* respectivamente, enquanto as outras *quadtrees* seccionam o paralelepípedo em suas respectivas coordenadas do eixo Z. Este grande paralelepípedo é a raiz da *octree*.

Caso nenhuma das *quadtrees* seja subdividida (todas possuem apenas um nível, sendo este a raiz) e o valor de *pixel* dos nodos do nível são iguais (ou dentro de um valor de tolerância), a *octree* também não será subdividida, possuindo também apenas um nível. Caso uma das premissas anteriores não seja verdade, a subdivisão da *octree* irá ocorrer. Se a primeira premissa não for satisfeita a subdivisão ocorre por que, se há uma subdivisão nos eixos X e Y (o plano representado pela imagem), deve haver também uma subdivisão no eixo Z, conforme a regra de montagem da *octree*. Não satisfazendo a segunda premissa a subdivisão ocorre por que os valores de *pixel* diferentes devem ficar em paralelepípedos diferentes.

Havendo a subdivisão da raiz da *octree*, o processo continua até que as premissas apresentadas sejam satisfeitas para todos os nodos. No entanto, os sub-nodos da *octree* não irão depender mais das raízes das *quadtrees*, mas sim dos nodos no mesmo nível ou menor e que estejam na mesma região dos primeiros. A figura 8 mostra o processo de criação da *octree*, visto

lateralmente, com as imagens representadas pelas *quadtrees* na horizontal (linhas grossas). As linhas pontilhadas indicam as subdivisões da *octree*, as setas indicam o ponto de subdivisão das *quadtrees* e os números indicam o valor de pixel em cada região. As *quadtrees* usadas no exemplo da figura 8 são mostradas na figura 9, aonde as bolas pretas mostram o ponto para onde apontam as setas.

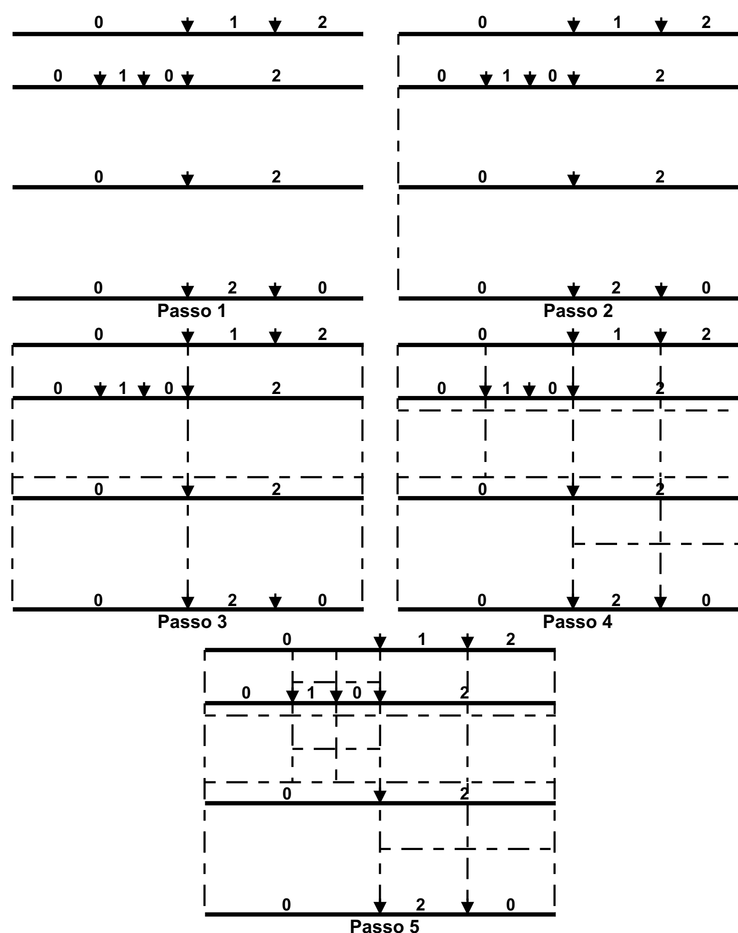


Figura 8: Processo de criação de uma *octree*, visto lateralmente.

Nota-se que, como mostra o exemplo da figura 8, alguns nodos da *octree* não são cortados por nenhuma *quadtree*. Para a atribuição dos valores de *pixel* destes nodos estão sendo utilizados os valores dos nodos de mesmo nível ou menor da *quadtree* mais próxima ao centro daquele. Caso não haja um nodo de mesmo nível na *quadtree* mais próxima, estes nodos serão subdivididos até que se encontre um.

Tanto no processo de subdivisão dos nodos que são seccionados por *quadtrees* como naque-

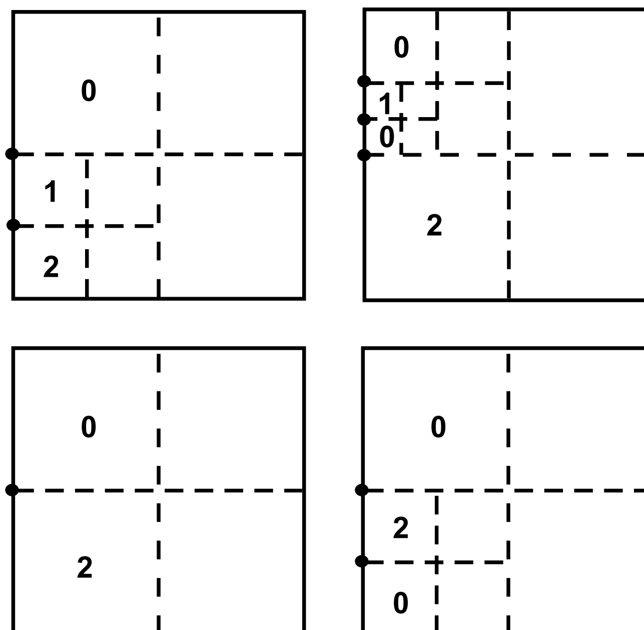


Figura 9: Divisões das *quadtrees* usadas no exemplo da figura 8.

les que não são seccionados são considerados também os nodos de *quadtree* de níveis menores do que o nodo da *octree* sendo processado no momento. Isso se deve ao fato de que, mesmo que não tenha havido divisão no plano da imagem, no momento da geração das *quadtrees*, pode ser necessário uma subdivisão deste plano devido a influência de uma outra *quadtree* que possui subdivisões na mesma região. Isto pode ser visto na figura 8, aonde no canto superior esquerdo existem quatro paralelepípedos, apesar de que uma das *quadtrees* (a segunda) não possua uma subdivisão naquele ponto (nodo de nível menor). No entanto, para a atribuição dos valores de *pixel* destes quatro paralelepípedos, este nodo de *quadtree* de nível inferior terá influência.

Os dois métodos usados para a criação da *octree* são mostrados na figura 10. O procedimento CONSTRÓI_OCTREE realiza as verificações acerca da necessidade de subdivisão ou não de um determinado nodo da *octree* e, caso este deva ser subdivido, o método SUBDIVIDE_NODO é chamado, caso contrário, é atribuído ao nodo um valor de pixel apropriado. O procedimento SUBDIVIDE_NODO é responsável por selecionar os nodos das *quadtrees* na mesma região, nos eixos X e Y, que as subdivisões do nodo de *octree* que estiver sendo processado, e encaminhar os nodos que representam essas subdivisões para que sejam verificados pelo

método CONSTRÓI_OCTREE. Este procedimento também é responsável por atribuir o valor *NULO* a um nodo que não represente uma região na imagem.

Ambos os métodos recebem os mesmos tipos de parâmetros, sendo estes:

- QN: Conjunto ordenado de nodos de *quadtree*.
- PN: Conjunto ordenado que contém as posições no eixo Z das *quadtrees* a que pertencem os nodos em QN.
- on: Nodo de *octree* a ser verificado (no método CONSTRÓI_OCTREE) ou subdividido (pelo método SUBDIVIDE_NODO).
- t: Valor de tolerância da variação dos valores de *pixel* dentro de um determinado nodo.

O processo de construção da *octree* é iniciado com o método CONSTRÓI_OCTREE, usando como parâmetros:

- QN: As raízes de todas as *quadtrees*.
- PN: As posições, no eixo Z, de todas as *quadtrees*.
- on: A raíz da *octree*.
- t: Um valor de tolerância arbitrário.

Para a criação da raiz da *octree*, atribuí-se ao ponto mínimo (minPto) os valores do ponto mínimo das *quadtrees*, para as coordenadas dos eixos X e Y, e o valor do menor elemento em PN, para o eixo Z, e ao ponto máximo (maxPto) os valores do ponto máximo das *quadtrees*, para as coordenadas dos eixos X e Y, e o valor do maior elemento em PN, para o eixo Z.

No decorrer dos procedimentos apresentados na figura 10, algumas funções são consideradas pré-existentes, sendo estas:

- CENTRO_Z($n$): Retorna o valor da coordenada Z do centro do nodo $n$ de uma *octree.*

- FOLHA(*n*): Retorna *VERDADE* caso o nodo *n* seja uma folha e *FALSO* caso contrário.
- CRIA_FILHO(*n*, *o*): Cria um filho no octante *o* em um nodo *n* de uma *octree*, seguindo a regra de subdivisão dos nodos em *octrees*, e retorna o nodo criado. Após a criação do filho, este pode ser acessado via *n*.FILHOS[o].

Após terminada a execução dos métodos de criação da *octree*, o retorno será *VERDADEIRO*, caso a criação tenha ocorrida com sucesso, e *FALSO*, caso não haja nenhum valor a ser representado por essa *octree*. Neste último caso, pode-se atribuir *NULO* à raiz.

De acordo com a linha 11, no método CONSTRÓI_OCTREE, só terão influência, neste primeiro momento, no processo de decisão quanto a subdivisão ou não do nodo *on* da *octree*, aqueles nodos de *quadtree* que estejam no intervalo do eixo Z representado por *on*. A preocupação de selecioná-los quanto a região dos eixos X e Y está no método SUBDIVIDE_NODO (linhas 36-39), aonde só são incluídos no conjunto de nodos de *quadtree* aqueles nodos de mesmo nível ou inferior, tendo estes o valor *NULO* ou não. Os nodos de *quadtree* de valor *NULO* são incluídos, pois influenciam na decisão de subdivisão ou não do nodo da *octree*.

Na linha 21 estão as condições para que seja feita a subdivisão de *on*, sendo estas a ultrapassagem do valor de tolerância para os valores de pixel, a existência de nodos vazios (com valor *NULO*) e de nodos folhas simultaneamente, que indica que nem todos os nodos naquela região tem o mesmo valor, e caso o nodo da *quadtree* não seja folha, que provoca a necessidade de subdivisões no nodo da *octree*, como explicado previamente.

O final do método CONSTRÓI_OCTREE (a partir da linha 23) só será atingido caso nenhuma subdivisão tenho sido determinada, até o momento, como necessária, depois de examinados todos os nodos das *quadtrees*. No entanto, na linha 23, o número de folhas e vazios iguais a zero indicam que as condições impostas na linha 11 nunca foram satisfeitas, ou seja, não existe nenhum nodo de *quadtree* no intervalo do eixo Z representado por *on*. Neste caso, atribuí-se o valor de *pixel* do nodo mais próximo (maisProx), caso este seja uma folha. Caso seja o nodo mais próximo seja *NULO*, o método retorna com o valor *FALSO*, para que este valor seja atribuído no método SUBDIVIDE_NODO. Caso o nodo mais próximo não seja nem folha

```
01. BOOLEANO CONSTRÓI_OCTREE(QN, PN, on, t) {
02.     vazios ← 0;
03.     folhas ← 0;
04.     maxValor ← 0;
05.     minValor ← ∞;
06.     menorDist ← ∞;
07.     PARA i ← 1 ATÉ |QN| {
08.       SE menorDist > |PN[i] - CENTRO_Z(on)| ENTÃO {
09.         menorDist ← |PN[i] - CENTRO_Z(on)|;
10.         maisProx ← QN[i]; }
11.       SE PN[i] >= on.minPto.z E PN[i] <= on.maxPto.z ENTÃO {
12.         SE QN[i] = NULO ENTÃO vazios ← vazios + 1;
13.         SE FOLHA(QN[i]) ENTÃO {
14.           folhas ← folhas + 1;
15.           éFolha ← VERDADE;
16.           SE QN[i].valorPixel > maxValor ENTÃO
17.             maxValor ← QN[i].valorPixel;
18.           SE QN[i].valorPixel < minValor ENTÃO
19.             minValor ← QN[i].valorPixel; }
20.         SENÃO éFolha ← FALSO;
21.         SE maxValor - minValor > t OU
              (vazios > 0 E folhas > 0) OU NÃO éFolha ENTÃO
22.             RETORNA(SUBDIVIDE_NODO(QN, PN, on, t)); }}
23.     SE vazios = 0 E folhas = 0 ENTÃO {
24.       SE maisProx = NULO ENTÃO RETORNA(FALSO);
25.       SE FOLHA(maisProx) ENTÃO {
26.         on.valorPixel ← maisProx.valorPixel;
27.         RETORNA(VERDADE); }
28.       RETORNA(SUBDIVIDE_NODO(QN, PN, on, t); }
29.     SE folhas > 0 ENTÃO {
30.       on.valorPixel ← maxValor;
31.       RETORNA(VERDADE); }
32.     RETORNA(FALSO); }

33. BOOLEANO SUBDIVIDE_NODO(QN, PN, on, t) {
34.     PARA j ← 1 ATÉ 4 {
35.       nQN ← ϕ;
36.       ∀n ∈ QN {
37.         SE n = VAZIO ENTÃO nQN ← nQN ∪ {VAZIO} SENÃO
38.           SE FOLHA(n) ENTÃO nQN ← nQN ∪ {n} SENÃO
39.             nQN ← nQN ∪ {n.FILHOS[j]}}
40.       SE NÃO CONSTRÓI_OCTREE(nQN, PN, CRIA_FILHO(on, j), t)
          ENTÃO
41.           on.FILHOS[j] ← NULO;
42.       SE NÃO CONSTRÓI_OCTREE
                (nQN, PN, CRIA_FILHO(on, j + 4), t) ENTÃO
43.         on.FILHOS[j + 4] ← NULO; }
44.     RETORNA(VERDADE); }
```

Figura 10: Métodos para construção da *octree*.

e nem *NULO*, *on* será subdividido até que se encontre um nodo folha ou *NULO* de mesmo nível.

Sendo todos folhas os nodos de *quadtree* na região representada por *on*, deve-se atribuir uma valor de pixel a este. Isto é verificado na linha 29 e a atribuição do valor de *pixel* é feita logo após, usando o maior valor encontrado. Neste ponto poderia-se atribuir a *on* o menor valor de *pixel* ou a média entre eles, sendo que, neste caso, pode haver problemas se estiver sendo utilizada uma paleta de cores.

Na última linha do método CONSTRÓI_OCTREE (linha 32) está o retorno que indica que todos os nodos na região representada por *on* possuem o valor *NULO*. Este valor vai ser atribuído ao nodo da *octree* pelo método SUBDIVIDE_NODO, que o criou.

Nos métodos propostos, em nenhum momento houve a preocupação com o tamanho mínimo que os paralelepípedos representados pelos nodos da *octree* podem assumir. Em uma implementação real destes métodos, este fato pode fazer com que o procedimento nunca termine devido à possível insuficiência de precisão da máquina que o executa durante as subdivisões dos nodos.

## 4.3 Análise entre os Métodos de Criação das *Quadtrees Volumétricas* e *Octrees*

O método de criação de *quadtrees volumétricas* partindo de um conjunto de *quadtrees* é rápido e o número de nodos permanece constante ao terminar o processo, assim como o nível das árvores.

O processo de geração das *octrees*, além de mais complexo e de necessitar mais recursos computacionais, pode gerar mais ou menos nodos na árvore do que a soma dos nodos das *quadtrees* e a profundidade da árvore atinge valores arbitrários.

Esta variabilidade da quantidade de nodos e da profundidade das *octrees* depende da disposição das *quadtrees* no eixo Z e das suas subdivisões nos eixos X e Y, como pode ser observado no exemplo da figura 11.

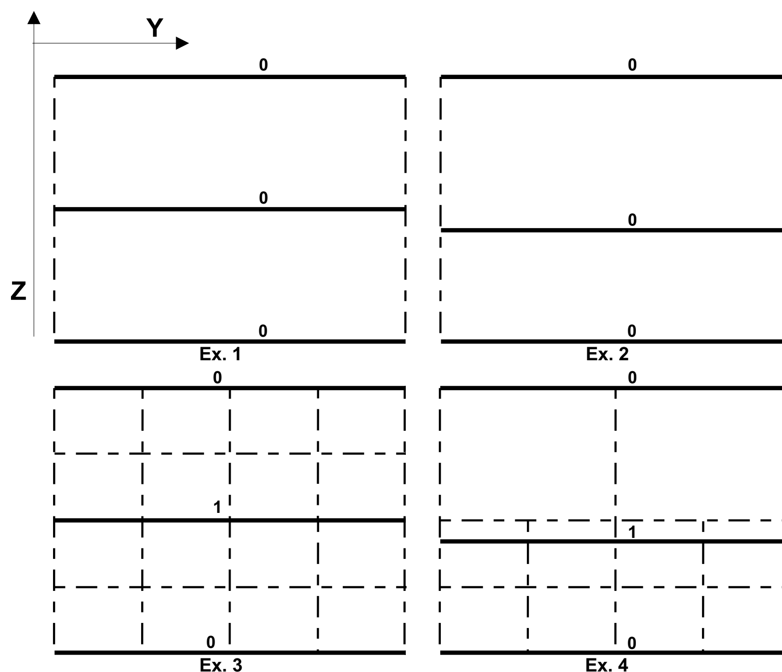


Figura 11: Representações por *octree* de *quadtrees* com apenas um nodo.

A figura nos mostra quatro representações de *octrees*, vistas lateralmente, da mesma maneira como foi apresentado na figura 8. Nos quatro exemplos são apresentadas três *quadtrees*, com apenas um nodo (a raiz), e o resultado da geração da representação por *octree* destas. No entanto cada exemplo apresenta uma variação em relação ao outro, seja na disposição das *quadtrees* ou no valor de *pixel* do nodo.

No primeiro e no segundo exemplo, aonde todas as *quadtrees* têm o mesmo valor de *pixel* para os seus nodos, será criada uma octree com apenas um nodo, não importando a disposição daquelas. O terceiro exemplo mostra a criação de uma *octree* com uma *quadtree* com valor de *pixel* distinto das outras duas. Neste caso a *octree* terá setenta e três nodos e profundidade três. Com um pequeno deslocamento da *quadtree* com valor de *pixel* distinto das outras duas, o número de nodos da *octree* resultante cai para quarenta e um, porém a profundidade da árvore permanece constante, como é demonstrado no quarto exemplo.

# 5 *Renderização das Estruturas Tridimensionais*

Tanto as *quadtrees volumétricas* quanto as *octrees* podem ser renderizadas diretamente, já que cada nodo representa um cubo. A renderização de todos os cubos representados pelos nodos folhas destas árvores resultaria, então, no objeto final.

Durante o processo de renderização é, no entanto, necessária a preocupação sobre quais dos cubos ficarão visíveis, já que não são visíveis os cubos que estão atrás de outros. Caso o *hardware* de renderização disponibilize de um *z-buffer* (MöLLER; HAINES, 1999), o problema de que um cubo atrás de outros seja renderizado na frente não ocorrerá. Caso contrário, Foley et al. (1990) sugerem que a renderização comece pelo cubo mais distante do observador, e continue, em ordem decrescente de suas distâncias, até o cubo mais próximo.

Como as estruturas apresentadas lidam com o valor de *pixel*, atribuídos diretamente das imagens, presume-se que uma paleta de cores, ou outro método de atribuição de cores as valores de uma imagem, seja utilizado para a renderização dos cubos.

A figura 12 mostra uma renderização de uma *octree* representando o crânio humano, criada a partir de *quadtrees* geradas de imagens tomográficas. Os nodos das *quadtrees* representavam áreas de no mínimo 15 *pixels*.

## 5.1 Acelerando a Renderização

Como foi citado anteriormente, grande parte dos cubos são ocultados pelos cubos que estão mais próximos do ponto de observação. Em média, o número de cubos que estão ocultos atinge

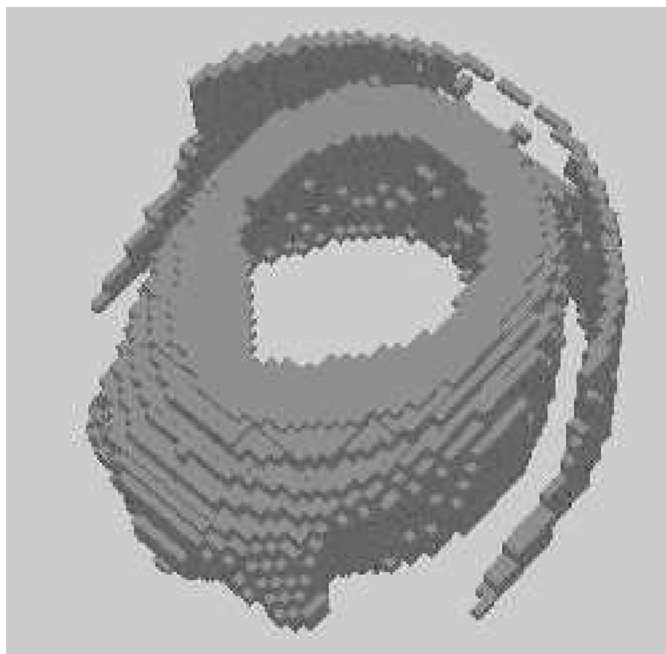

Figura 12: Renderização de uma *octree* representando o crânio humano.

taxas maiores que 90% do total. Como não são visíveis, não precisam ser renderizados. Deve-se, então, posteriormente ao processo de renderização, realizar a seleção daqueles cubos que irão influenciar na imagem final, que será visualizada.

Serão apresentadas propostas para a seleção de cubos a serem renderizados, sendo uma caso o observador seja externo à estrutura e outra caso este seja interno.

### 5.1.1 Renderização com Observador Externo

Se o observador é externo à estrutura pode-se , inicialmente, descartar todos os cubos internos, ou seja, todos aqueles cubos que não estão na periferia da estrutura.

Um método eficiente de selecionar os cubos que estão na periferia (mas não totalmente eficaz, pois não seleciona somente os da periferia) é escolher todos aqueles nodos folhas que possuem vizinhos de face, de mesmo tamanho ou menor, que sejam *NULOS*. Caso a estrutura possua alguma cavidade interna, cubos internos serão também selecionados. Neste caso deve-se continuar tendo o cuidado de não renderizá-los na frente de outros.

Esta pré-seleção de cubos reduz enormemente a quantidade de cubos a serem renderizados

e este número pode ser reduzido ainda mais, já que os cubos periféricos da parte da estrutura que não faz face com o observador também podem ser descartados (além dos cubos selecionados erroneamente como periféricos). Dos cubos visíveis, certas faces destes permanecem ocultas, podendo ser ignoradas juntamente com os outros cubos. Estes cubos e faces ocultos podem ser detectados através das técnicas conhecidas como *occlusion culling* (seleção de oclusão) (GREENE; KASS; MILLER, 1993; MöLLER; HAINES, 1999) e *backface culling* (seleção de face traseira) (CLARCK, 1976; MöLLER; HAINES, 1999).

A figura 13 mostra os métodos para realizar a seleção dos cubos visíveis. O procedimento RENDERIZAR_ÁRVORE tem como parâmetro a árvore de representação tridimensional a ser renderizada, seja ela uma *octree* ou uma *quadtree volumétrica*. O método VERIFICA_NULO verifica se o nodo, passado como parâmetro é *NULO* ou se também são algum de seus descendentes, adjacentes à face também passada como parâmetro.

Algumas funções são presumidas como existentes, sendo elas:

- FOLHA($n$): Retorna *VERDADE* caso o nodo $n$ seja uma folha e *FALSO* caso contrário.
- OT_GTEQ_FACE_NEIGHBOR($p$, $i$): Esta função localiza o vizinho de face da face $i$ do nodo $p$ com tamanho maior ou igual a $p$. Caso o nodo $p$ pertença a uma *octree*, esta função está descrita em Samet (1989b), caso pertença a uma *quadtree volumétrica*, ver a seção 3.2.1.
- PRÓXIMO(): Faz com que a estrutura de repetição passe para o próximo elemento sem executar o restante das instruções após esta função.
- APLICA_OCCLUSION_CULLING($N$): Aplica, no conjunto de nodos $N$, técnicas de seleção de oclusão e retorna os nodos que não estão oclusos. Presume-se que a localização do observador é conhecida.
- APLICA_BACKFACE_CULLING($N$): Aplica, no conjunto de nodos $N$, técnicas de seleção de face traseira e retorna as faces que são visíveis. Presume-se que a localização do observador é conhecida.

- RENDERIZAR(*f*): Faz com que a face *f* seja mostrada no dispositivo de saída escolhido.

- ADJ(*i*, *o*): Retorna *VERDADE* se o octante *o*, no caso de uma *octree*, ou quadrante *o*, no caso de uma *quadtree volumétrica*, é adjacente à face *i*. Caso o seja um octante, esta função está descrita em Samet (1989b), caso seja um quadrante, ver a seção 3.2.1.

- OCTANTE(*i*): Transforma o octante *i*, neste caso numérico, no padrão apresentado por Samet (1989b) (baseado em caracteres), para servir como parâmetro do método ADJ. Este método é simplesmente utilizado para que não seja necessária uma correlação explícita dos padrões de representação de octantes utilizados neste trabalho com os propostos por Samet, já que a utilização direta do método ADJ é mais simples. Em uma implementação real este método é totalmente desnecessário, já que uma representação única pode ser utilizada.

Importante notar que a implementação do método VERIFICA_NULO mostrada na figura considera que a árvore sendo renderizada é uma *octree*. Caso fosse uma *quadtree volumétrica*, a estrutura de repetição da linha 34 deveria variar de 1 a 4 e, na linha 35, a chamada à função OCTANTE deve ser substituída por uma chamada à função QUADRANTE, cujo o retorno seria a representação de quadrantes propostas por Samet (1982).

Nas linhas 5 a 26 ocorre as buscas dos vizinhos do nodo que está sendo verificado. Caso o vizinho encontrado tenha sido da face de frente ('F')[1], verifica-se se este ou algum de seus filhos adjacentes à sua face traseira ('B'), face esta comum à ambos os nodos, são *NULO*s. Caso isto ocorra, o nodo em questão entrará na lista de renderização, o que, como mencionado anteriormente, não significa que este será mantido nesta lista.

Pode ser feita uma otimização no tempo de execução do método caso uma lista de vizinhos seja criada para os nodos percorridos. Isto é, digamos que se esteja decidindo se o nodo *n* será posto na lista de renderização ou não. Para isso, será procurado seu vizinho da frente, sendo este *f*. Portanto, o vizinho de trás do nodo *f* é *n*. Isto podia ser armazenado numa lista e, no

[1]O padrão de nomenclatura de faces utilizado por Samet são: F (*front*, frente), B (*back*, atrás), R (*right*, direita), L (*left*, esquerda), U (*up*, cima) e D (*down*, baixo).

```
01. NADA RENDERIZAR_ÁRVORE(A) {
02.     R ← φ;
03.     ∀nodo ∈ A {
04.       SE FOLHA(nodo) ENTÃO {
05.         vizinho ← OT_GTEQ_FACE_NEIGHBOR(nodo, 'F');
06.         SE VERIFICA_NULO(vizinho, 'B') ENTÃO {
07.           R ← R ∪ {vizinho};
08.           PRÓXIMO();}
09.         vizinho ← OT_GTEQ_FACE_NEIGHBOR(nodo, 'B');
10.         SE VERIFICA_NULO(vizinho, 'F') ENTÃO {
11.           R ← R ∪ {vizinho};
12.           PRÓXIMO();}
13.         vizinho ← OT_GTEQ_FACE_NEIGHBOR(nodo, 'R');
14.         SE VERIFICA_NULO(vizinho, 'L') ENTÃO {
15.           R ← R ∪ {vizinho};
16.           PRÓXIMO();}
17.         vizinho ← OT_GTEQ_FACE_NEIGHBOR(nodo, 'L');
18.         SE VERIFICA_NULO(vizinho, 'R') ENTÃO {
19.           R ← R ∪ {vizinho};
20.           PRÓXIMO();}
21.         vizinho ← OT_GTEQ_FACE_NEIGHBOR(nodo, 'U');
22.         SE VERIFICA_NULO(vizinho, 'D') ENTÃO {
23.           R ← R ∪ {vizinho};
24.           PRÓXIMO();}
25.         vizinho ← OT_GTEQ_FACE_NEIGHBOR(nodo, 'D');
26.         SE VERIFICA_NULO(vizinho, 'U') ENTÃO R ← R ∪ {vizinho};}}
29.     R ← APLICA_OCCLUSION_CULLING(R);
30.     R ← APLICA_BACKFACE_CULLING(R);
31.     ∀face ∈ R (RENDERIZAR(face));}

32. BOOLEANO VERIFICA_NULO(nodo, face) {
33.     SE nodo = NULO ENTÃO RETORNA(VERDADE) SENÃO {
34.       SE FOLHA(nodo) ENTÃO RETORNA(FALSO) SENÃO {
34.         PARA i ← 1 ATÉ 8
35.           SE ADJ(face, OCTANTE(i)) ENTÃO {
36.             SE VERIFICA_NULO(nodo.FILHOS[i], face) ENTÃO
37.               RETORNA(VERDADE);}}}
38.     RETORNA(FALSO);}
```

Figura 13: Métodos para eliminação de cubos e faces não visíveis.

momento em que a decisão sobre a renderização ou não seja sobre o nodo *f*, seu vizinho de trás não precisa ser procurado. Aplicando esta técnica a todos os vizinhos, o número de buscas na árvore se reduzirá significativamente.

### 5.1.2 Renderização com Observador Interno

Todos os elementos de uma cena que devem ser renderizados estão dentro de um volume de renderização, sendo este volume um paralelepípedo, caso a projeção adotada seja ortográfica, ou uma seção de pirâmide, caso a projeção seja em perspectiva (FOLEY et al., 1990). Não importando o tipo de projeção adotada, existem no volume de renderização dois planos que indicam a distância máxima e mínima que devem se encontrar os objetos a serem renderizados, conforme a figura 14.

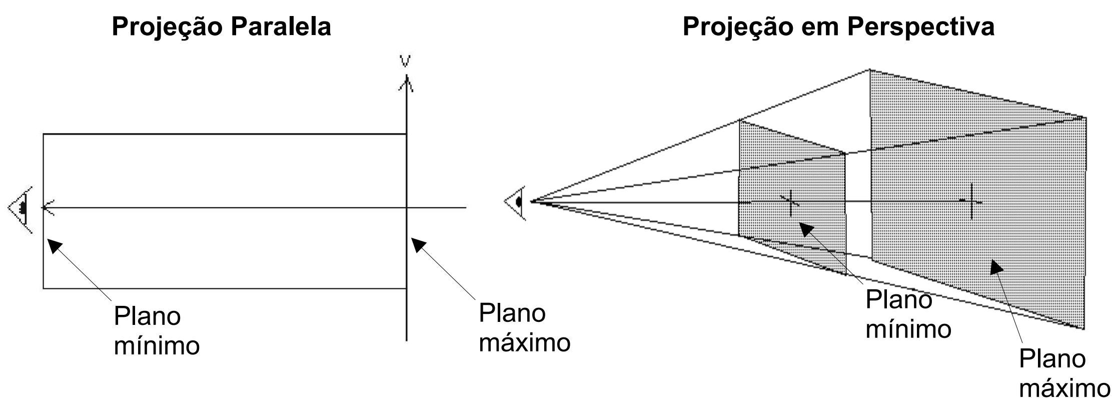


Figura 14: Volumes de renderização em projeções paralelas e em perspectiva.

Só devem ser renderizados os nodos que intersectam o volume de renderização delimitado pelos planos máximo e mínimo. Testes de intersecção podem ser encontrados em Möller e Haines (1999).

O fato de se estar renderizando árvores permite que os testes de intersecção sejam realizados hierarquicamente. Em uma *octree*, testa-se, primeiramente a intersecção com a raiz. Se esta existir presume-se que o observador é interno, e continua-se testando as intersecções com os filhos até que se chegue nas folhas. Caso a intersecção com um determinado nodo não exista, não será necessário que se teste seus filhos.

Em *quadtrees volumétricas* testa-se a intersecção com a raiz de cada uma das árvores e continua-se testando com os filhos das quais o teste foi verdadeiro, semelhante ao procedimento realizado com a *octree*. Para exemplificar, a figura 15 mostra um teste hierárquico de intersecções em uma *quadtree* simples.

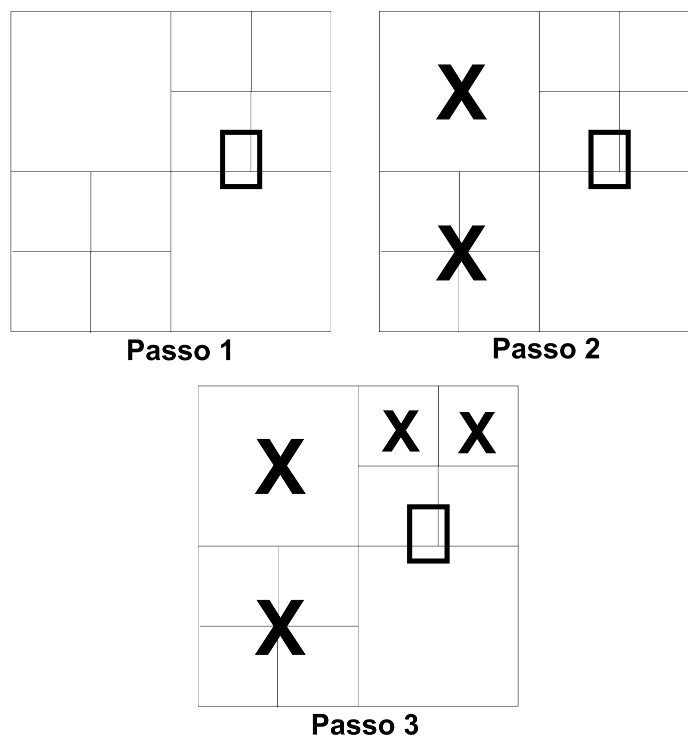


Figura 15: Testes de intersecção em árvore com observador interno.

Testes de oclusão (*occlusion culling* e *backface culling*) também podem ser utilizados, já que os nodos que possuem intersecção com o plano mínimo geralmente escondem os outros que estão presentes dentro do volume de renderização.

## 5.2 Operações para Renderização

Várias operações para renderização podem ser realizadas com as *octrees*. Será mencionado, brevemente, como operações de *peeling* (descascamento), eliminação de partes e visualização por planos arbitrários podem ser realizados.

O *peeling* consiste em eliminar certas camadas ao redor da estrutura, como se estivesse des-

cascando esta. Para realização do *peeling* é necessário que os nodos das bordas da árvore sejam encontrados e a renderização destes não será completa, mas se realizará de forma a eliminar parte ou todo ele.

Por exemplo, se é desejada a realização de um *peeling* de 10 *pixels* no topo da estrutura, deve-se buscar, primeiramente, os nodos no topo através da busca de vizinhos (em *octrees* ver Samet (1989b) e em *quadtrees volumétricas* ver 4.1). Dos nodos com mais de 10 *pixels* de altura, parte destes deve ser renderizada, ou seja, um paralelepípedo com sua altura reduzida em 10 *pixels*. Caso os nodos não tenham 10 *pixels* de altura, a operação elimina este nodo e retira o restante dos vizinhos abaixo, até ser atingida a profundidade desejada.

A eliminação de certas partes da estrutura pode ser feita pela remoção de um determinado número de nodos em determinada direção, o que pode ser realizado facilmente com a busca de vizinhos.

Deslocando o ponto de localização do volume de renderização por dentro da estrutura, esta será visualizada por planos arbitrários, representado pelo plano mínimo do volume, conforme discutido na seção 5.1.2.

Após a aplicação de alguma destas operações, a árvore resultante pode ser renderizada conforme mostrado nas seções anteriores.

# 6 Trabalhos Futuros

Muitas vezes, quando criando a representação tridimensional a partir de *quadtrees*, cujos nodos representam áreas mínimas maiores que 1 *pixel*, pode ocorrer o efeito de escada na estrutura, como pode ser observado na figura 12. Para que isto não ocorra, técnicas de suavização da superfície utilizando superfícies bicubícas com *splines* (FOLEY et al., 1990) estão sendo desenvolvidas. Outra alternativa para isso seria utilizar triangularização, o que será, também, tema para estudo.

Serão estudadas novas propostas para a atribuição de valores de *pixel* aos nodos não seccionados por *quadtrees*, tendo em vista que o método utilizado atualmente (valor de *pixel* do nodo de *quadtree* mais póximo) não é o mais indicado.

Métodos de exportação de parte ou toda a árvore também estão sendo planejadas, o que permite o compartilhamento de regiões de interesse entre usuários da técnica.

# 7 Conclusão

Imagens paralelamente alinhadas, segundo a definição apresentada, têm, atualmente, várias áreas de aplicação, com destaque na área médica. Certas limitações provindas do uso de tais imagens para a análise de aspectos internos de uma determinada estrutura podem ser superadas através de uma representação tridimensional destas.

Várias estruturas para representação de dados tridimensionais são conhecidas, tais como as *octree*, cujas propriedades são bastante conhecidas e documentadas.

A utilização de *octrees* como estrutura de dados para uma representação tridimensional permite que várias operações (como renderização e busca em árvore) sejam realizadas de forma eficiente. No entanto, o processo de criação de tal estrutura, através do método proposto, é lento e carece de muitos recursos (devido às recursões no método).

Apesar de lento, o processo de criação de *octrees* para representação de uma conjunto de imagens bidimensionais preenche de forma realista os espaços existentes entre aquelas. Importante notar que a precisão no preenchimento de tais espaços é maior quanto menor for a distância entre as imagens.

Uma representação tridimensional de imagens paralelamente alinhadas podem ser obtidas rapidamente com a utilização de *quadtrees volumétricas*, cujo método de criação é direto. Porém, esta forma de representação preenche de forma grosseira os espaços entre as imagens.

# *Referências*

BHATTACHARYA, Parthajit. *Efficient Neighbor Finding Algorithms in Quadtree and Octree*. Dissertação (Mestrado) — Department of Computer Science & Engineering, Indian Institute of Technology, Kanpur, India, 2001.

CLARCK, James H. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, v. 19, n. 10, p. 547–554, 1976.

FOLEY, James D.; VAN DAM, Andries; FEINER, Steven K.; HUGHES, John H. *Computer Graphics - Principles and Practice*. 2. ed. Reading, Massachusetts, EUA: Addison-Wesley, 1990.

FRISKEN, Sarah F.; PERRY, Ronald N. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, v. 7, n. 3, p. 1–11, 2002.

GREENE, Ned; KASS, Michael; MILLER, Gavin. Hierarchical z-buffer visibility. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. New York, New York, EUA: ACM Press, 1993. p. 231–238.

LIN, Reitseng; WONG, Edward. Morphological operations on images represented by quadtrees. In: *Proceedings of the IEEE International Conference on Acoustics, Speech & Signal Processing*. Atlanta, Georgia, EUA: [s.n.], 1996.

MöLLER, Tomas; HAINES, Eric. *Real-Time Rendering*. Natick, Massachusetts, EUA: A K Peters, 1999.

RANADE, Sanjay; ROSENFELD, Azriel; SAMET, Hanan. Shape approximation using quadtrees. *Pattern Recognition*, v. 15, n. 1, p. 31–40, 1982.

SAMET, Hanan. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, n. 18, p. 37–57, 1982.

SAMET, Hanan. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, ACM Press, v. 16, n. 2, p. 187–260, 1984.

SAMET, Hanan. *The Design and Analysis of Spatial Data Structures*. Reading, Massachusetts, EUA: Addison-Wesley, 1989a.

SAMET, Hanan. Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, n. 46, p. 367–386, 1989b.

SPRAWLS, Perry. Computed tomography image formation. In: ____. *The Physical Principles of Medical Imaging*. 2. ed. [s.n.], 2004. Disponível em: <http://www.sprawls.org/resources/CTIMG>. Acesso em: 15/01/2004.

WAGNER, Harley M. *Atlas Cerebral Digital: Desenvolvimento de uma Ferramenta Computacional para Mapeamento Funcional e Anatômico de Áreas Cerebrais, Baseado no Atlas de Talairach*. Dissertação (Mestrado) — Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis, Santa Catarina, 2001.

ZACHMANN, Gabriel; LANGETEPE, Elmar. Geometric data structures for computer graphics. In: *Proceedings of ACM SIGGRAPH*. [S.l.]: ACM Transactions of Graphics, 2003. p. 27–31.

# *ANEXO A – Representação por* **Octree** *de uma Superfície Não-plana*

As figuras 16 e 17 mostram uma determinada superfície não-plana e a renderização, em arame, de sua representação por *octree*, respectivamente.

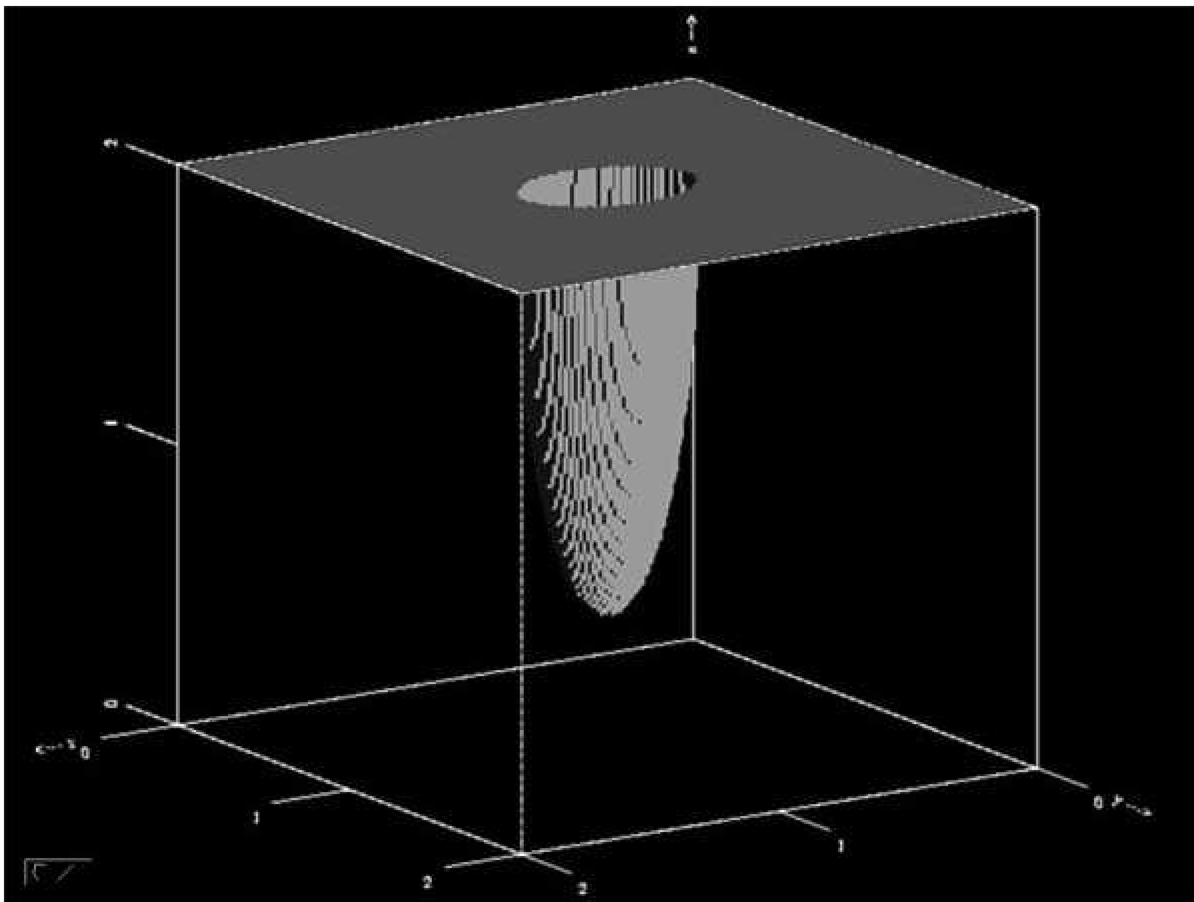


Figura 16: Uma superfície não-plana.

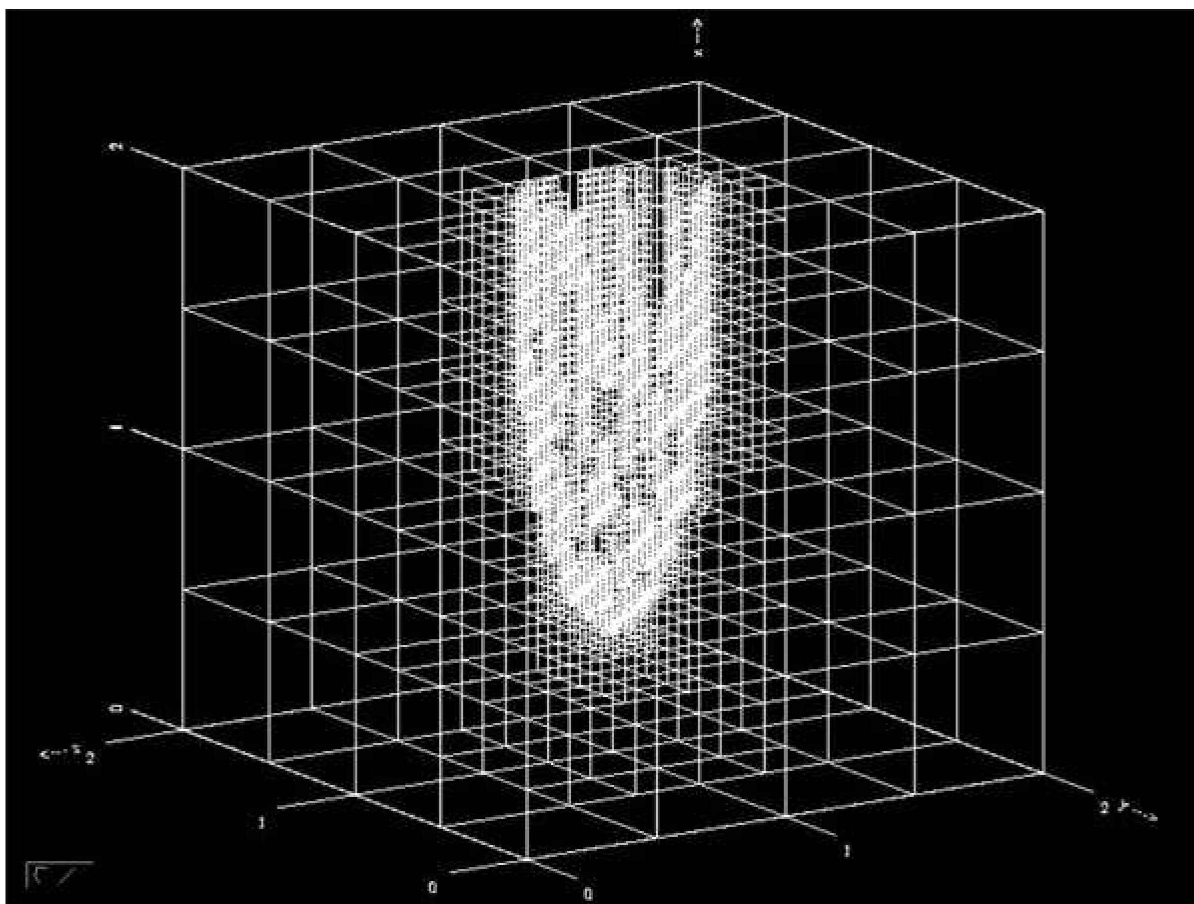

Figura 17: Renderização da representação por *octree* da superfície apresentada na figura 16.

# *ANEXO B – Artigo Submetido ao CBMS 2004*

Artigo submetido ao "The 17th IEEE SYMPOSIUM ON COMPUTER-BASED MEDICAL SYSTEMS" (CBMS 2004).

## Three-Dimensional Visualization of Radiological Images using Octrees Paper Summary

Thiago R. dos Santos, Daniel D. Abdala, Aldo von Wangenheim

January, 2004

**Abstract**

This paper presents a method for robust three-dimensional reconstruction of radiological image volumes, such as computer tomography and magnetic resonance, taking advantage of the use of a well know data structure like the octree.

## 1 Introduction

Three-Dimensional volume visualization of radiological images is a restricted task where the medical user faces several limitations. Commonly three situations occur: visualization through radiological films, radiological workstations with no support to of three-dimensional volume rendering and radiological workstations with support to three-dimensional volume rendering based on non-documented proprietary solutions.

When using radiological films and radiological workstations lacking 3D support, the volumetric notion of the structures in analysis depends on the imagination of the user. The method hereby presented is to be compared to the methods used in the workstations with 3D support, with the advantages provided by a well know data structure (octrees).

This kind of 3D visualization allows one to show the radiological data in any arbitrary plan, including the sagital, axial and coronal. Also, through the octree volume mapping, various effects can be reached, such as structures peeling and cutoffs.



Figura 18: Artigo CBMS 2004 - Página 1

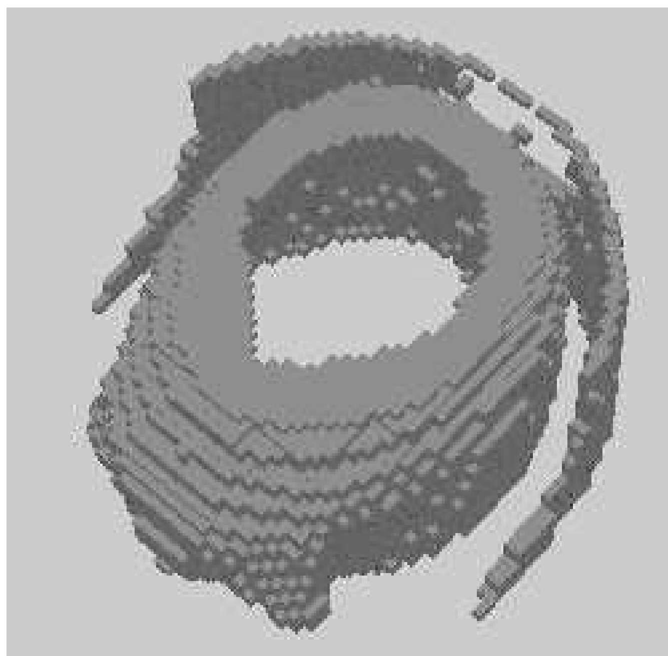

Figure 1: Three-dimensional reconstruction of a human skull using octree nodes with minimal base area of 15 pixels.

## 2 Methods

Initially, the octree mapping of the whole volume represented by the individual image slices must be performed. Samet [1] describes a method for octree representation applied to 3D images. In order to allow the octree representation of a sequence of 2D images, an adaptation of the method proposed by Samet was developed and implemented.

The modifications proposed in Samet´s methods are based on a quadtree mapping [2] of each image, followed by unification of each quadtree node into a common octree node. During this unification, any time when a quadtree node can not be unified with any other node of some other quadtree, it will be considered a octree node by itself, respecting some interpolation heuristics.

The figure 1 shows an example of a three-dimensional reconstruction.

One of the most important reasons to select the octree as our strategy to represent the radiological volume are the existence of well-developed methods based on the octree data structure properties, like neighbor finding [3, 4, 5].



Figura 19: Artigo CBMS 2004 - Página 2

Those methods allow some interesting visualization operations to be done. By instance, the peeling can be reached selecting all the external nodes from the octree and removing them from the structure to be rendered. Further peelings can be processed using the previous processed data structure. The external nodes can be easily found by searching the ones that do not have neighbours in some direction.

The cutoffs can be achieved by the removal of a given number of nodes in a given direction, and new visualization planes can be generated simply by moving the camera position through the structure. Hierarchical culling [6] helps to make this task very quick, taking advantage of the tree representation of the radiological volume.

# 3 Conclusion

The preliminary results of this work shows that the approach presented requires a considerable processing time for tree generation, however the tree data structure traversals is very quick, making the visualization of selected areas, as well as the whole volume, very quick.

Because of directly mapping of radiological images stereotaxy is achieved, been of great support to surgery planning.

# 4 Future works

We are concentrating at the moment on ways to speed-up the octree generation and to export the tree or part of it.

# References

[1] H. Samet, *The Design and Analysis of Spatial Data Structures.* Reading, Massachusetts, EUA: Addison-Wesley, 1989a.

[2] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[3] H. Samet, "Neighbor finding in images represented by octrees," *Computer Vision, Graphics, and Image Processing*, no. 46, pp. 367–386, 1989b.



Figura 20: Artigo CBMS 2004 - Página 3

[4] P. Bhattacharya, "Efficient neighbor finding algorithms in quadtree and octree," Master's thesis, Department of Computer Science & Engineering, Indian Institute of Technology, Kanpur, India, 2001.

[5] S. F. Frisken and R. N. Perry, "Simple and efficient traversal methods for quadtrees and octrees," *Journal of Graphics Tools*, vol. 7, no. 3, pp. 1–11, 2002.

[6] T. Möller and E. Haines, *Real-Time Rendering.* Natick, Massachusetts, EUA: A K Peters, 1999.

4

Figura 21: Artigo CBMS 2004 - Página 4

# *ANEXO C – Diagrama de Classes*

Diagrama de classes da implementação apresentada no anexo D. A classe *Rectangle* é padrão no ambiente utilizado para a implementação, enquanto a classe *Jun3dBoundingBox* é importada do pacote *Jun 532*.

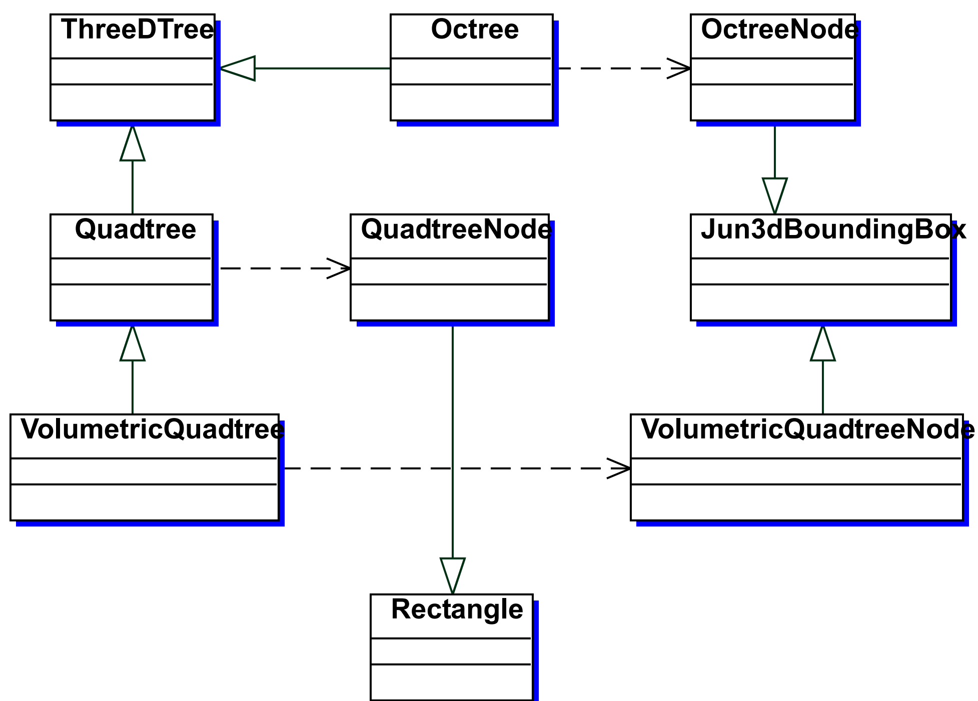


Figura 22: Diagrama de classes da implementação mostrada no anexo D.

# *ANEXO D – Código-fonte*

Código-fonte das classes implementadas para validação dos métodos de construção das árvores. A implementação foi feita utilizando a linguagem *Smalltalk* e o interpratador *VisualWorks 5i.4* com o pacote *Jun 532*.

```
Smalltalk defineClass: #VolumetricQuadtreeNode
    superclass: #{Jun.Jun3dBoundingBox}
    indexedType: #none
    private: false
    instanceVariableNames: 'childs pixelValue parent '
    classInstanceVariableNames: ''
    imports: ''
    category: '3D - TreeNodes'
VolumetricQuadtreeNode methodsFor: initialize
initialize
    childs := OrderedCollection withSize: self childCount
VolumetricQuadtreeNode methodsFor: testing
hasChilds
    childs do: [:child | child isNil ifFalse: [^true]].
    ^false
isLeaf
    ^pixelValue ~= nil
VolumetricQuadtreeNode methodsFor: accessing
childCount
    ^4
parent
    ^parent
```

```
parent: aVolumetricQuadtreeNode

        parent := aVolumetricQuadtreeNode

pixelValue

        ^pixelValue

pixelValue: aValue

        pixelValue := aValue

removeChildAt: aValue

        ^self removeChildAtQuadrant: aValue

VolumetricQuadtreeNode methodsFor: childs operations

addChildAtQuadrant: quadrant

        ^self addChildAtQuadrant: quadrant withPixelValue: nil

addChildAtQuadrant: quadrant withPixelValue: aValue

        | node |

        self verifyQuadrant: quadrant.

        self isLeaf

                ifFalse:

                        [childs at: quadrant put: (node := self perform: ('createNodeForQuadrant'
, quadrant printString , 'WithValue:') asSymbol with: aValue).

                                ^node]

                ifTrue: [self error: #errVolumetricQuadtreeNodeChildAddition « #dialogs »
'Impossible to create childs on leafs!']

childAt: index

        ^self childAtQuadrant: index
```

```
childAtQuadrant: quadrant
    self verifyQuadrant: quadrant.
    ^childs at: quadrant
removeChildAtQuadrant: quadrant
    self verifyQuadrant: quadrant.
    childs at: quadrant put: nil
VolumetricQuadtreeNode methodsFor: private
createNodeForQuadrant1WithValue: aValue
    ^self class
        origin: self origin
        corner: self origin x + self corner x // 2 , (self origin y + self corner y // 2) ,
self corner z
        pixelValue: aValue
        parent: self
createNodeForQuadrant2WithValue: aValue
    ^self class
        origin: self origin x , (self origin y + self corner y // 2) , self origin z
        corner: self origin x + self corner x // 2 , self corner y , self corner z
        pixelValue: aValue
        parent: self
createNodeForQuadrant3WithValue: aValue
    ^self class
```

```
origin: self origin x + self corner x // 2 , self origin y , self origin z
corner: self corner x , (self origin y + self corner y // 2) , self corner z
pixelValue: aValue
parent: self
createNodeForQuadrant4WithValue: aValue
ˆself class
origin: self origin x + self corner x // 2 , (self origin y + self corner y // 2) ,
self origin z
corner: self corner
pixelValue: aValue
parent: self
verifyQuadrant: quadrant
quadrant >= 1 & (quadrant <= self childCount) & (quadrant isKindOf: Integer)
ifFalse: [self error: #errVolumetricQuadtreeNodeChild « #dialogs » ’The quadrant must be a
Integer >= 1 and <= ’ , self childCount printString , ’!’]
Smalltalk.VolumetricQuadtreeNode class
instanceVariableNames: ”
VolumetricQuadtreeNode class methodsFor: instance creation
origin: originPoint corner: cornerPoint pixelValue: aValue
ˆ(self origin: originPoint corner: cornerPoint) initialize
pixelValue: aValue
origin: originPoint corner: cornerPoint pixelValue: aValue parent: aVolumetricQuadtree-
Node
```

```
^((self origin: originPoint corner: cornerPoint) initialize
        pixelValue: aValue) parent: aVolumetricQuadtreeNode
```

```
Smalltalk defineClass: #ThreeDTree
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'root pixelValueTolerance bitsPerPixel '
    classInstanceVariableNames: ''
    imports: ''
    category: '3D - Trees'
ThreeDTree methodsFor: private
countLeafsOf: aNode
    aNode isNil
        ifTrue: [^0]
        ifFalse: [aNode isLeaf
                ifTrue: [^1]
                ifFalse:
                    [| count |
                    count := 0.
                    1 to: aNode childCount do: [:i | count := count + (self
countLeafsOf: (aNode childAt: i))].
                    ^count]]
countNodesOf: aNode
    aNode isNil
```

```
                ifTrue: [^0]
                ifFalse: [aNode isLeaf
                        ifTrue: [^1]
                        ifFalse:
                            [| count |
                            count := 0.
                            1 to: aNode childCount do: [:i | count := count + (self
countNodesOf: (aNode childAt: i))].
                            ^count + 1]]
  free: aNode
        aNode isNil ifFalse: [aNode hasChilds
                    ifTrue:
                        [1 to: aNode childCount
                            do:
                                [:i |
                                self free: (aNode childAt: i).
                                aNode removeChildAt: i].
                        aNode parent: nil]]
  ThreeDTree methodsFor: accessing
  bitsPerPixel
        ^bitsPerPixel
  imageHeight
```

```
        root isNil
                ifTrue: [^0]
                ifFalse: [^root height + 1]
imageWidth
        root isNil
                ifTrue: [^0]
                ifFalse: [^root width + 1]
leafCount
        ^self countLeafsOf: root
nodeCount
        ^self countNodesOf: root
pixelValueTolerance
        ^pixelValueTolerance
root
        ^root
ThreeDTree methodsFor: destructor
free
        self free: root
ThreeDTree methodsFor: displaying, viewing
displayOn: aGraphicsContext withPalette: aPalette
        self
                displayNode: root
```

```
                    in: aGraphicsContext
                    withPalette: aPalette
    viewWithPalette: aPalette
            | compoundObject |
            compoundObject := JunOpenGL3dCompoundObject new.
            self
                    viewNode: root
                    inCompoundObject: compoundObject
                    withPalette: aPalette.
            compoundObject show
    ThreeDTree methodsFor: private - displaying, viewing
    displayNode: aNode in: aGraphicsContext withPalette: aPalette
            aNode isNil ifFalse: [aNode isLeaf
                            ifTrue: [aNode asFiller displayOn: (aGraphicsContext paint: (aPalette
at: aNode pixelValue))]
                              ifFalse: [1 to: aNode childCount do: [:i | self
                                                  displayNode: (aNode childAt: i)
                                                  in: aGraphicsContext
                                                  withPalette: aPalette]]]
    viewNode: aNode inCompoundObject: aCompoundObject withPalette: aPalette
            aNode isNil ifFalse: [aNode isLeaf
                           ifTrue: [aCompoundObject add: (aNode asJunOpenGL3dObjectColor:
(aPalette at: aNode pixelValue))]
```

```
                ifFalse: [1 to: aNode childCount do: [:i | self
                                 viewNode: (aNode childAt: i)
                                 inCompoundObject: aCompoundObject
                                 withPalette: aPalette]]]

Smalltalk.ThreeDTree class
    instanceVariableNames: ''

ThreeDTree class methodsFor: instance creation

new
    self error: #errThreeDTreeCreation « #dialogs » 'Can not create ThreeDTrees using
#new!'
```

```
Smalltalk defineClass: #Quadtree
    superclass: #{Smalltalk.ThreeDTree}
    indexedType: #none
    private: false
    instanceVariableNames: 'image minRectangleArea visibilityIntervals '
    classInstanceVariableNames: ''
    imports: ''
    category: '3D - Trees'
Quadtree methodsFor: accessing
minRectangleArea
    ^minRectangleArea
visibilityIntervals
    ^visibilityIntervals
Quadtree methodsFor: private - child generation
generateChildNodesFrom: aQuadtreeNode
    | maxValue total areaOk visibles minValue value |
    maxValue := 0.
    minValue := Infinity positive.
    visibles := 0.
    total := 0.
    value := aQuadtreeNode center.
   areaOk := aQuadtreeNode area / 4 >= minRectangleArea & (value ~= aQuadtreeNode
```

```
origin) & (value ˜= aQuadtreeNode corner).

        aQuadtreeNode origin y to: aQuadtreeNode corner y do: [:y | aQuadtreeNode origin
x to: aQuadtreeNode corner x

                do:

                    [:x |

                    maxValue < (value := image atPoint: x @ y) ifTrue: [maxValue
:= value].

                    minValue > value ifTrue: [minValue := value].

                    maxValue - minValue > pixelValueTolerance | (total > visibles)
& areaOk

                        ifTrue:

                            [| removed |

                            removed := 0.

                            1 to: 4 do: [:i | (self generateChildNodesFrom:
(aQuadtreeNode addChildAtQuadrant: i))

                                        ifFalse:

                                                [aQuadtreeNode
removeChildAtQuadrant: i.

                                                removed := removed + 1]].

                            ˆremoved < 4].

                    (self verifyIfVisible: value)

                        ifTrue: [visibles := visibles + 1].

                    total := total + 1]].
```

```
visibles > 0
        ifTrue:
                [aQuadtreeNode pixelValue: maxValue.
                ^true].
^false

verifyIfVisible: aPixelValue
        visibilityIntervals isNil ifTrue: [^true].
visibilityIntervals do: [:interval | (interval first <= aPixelValue) & (interval last >= aPixel-
Value)
                ifTrue: [^true]].
"(visibilityIntervals includes: aPixelValue)
        ifTrue: [
                ^true]."
        ^false

Quadtree methodsFor: private

minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue visibilityInter-
vals: anIntervalCollection image: aImage
        minRectangleArea := aAreaValue.
        pixelValueTolerance := aToleranceValue.
        visibilityIntervals := anIntervalCollection.
        image := aImage.
        (self generateChildNodesFrom: (root := QuadtreeNode
                                origin: 0 @ 0
```

```
                              corner: image width - 1 @ (image height - 1)
                              pixelValue: nil))
                ifFalse: [root := nil].
        bitsPerPixel := image bitsPerPixel.
        image := nil
    viewWithPalette: aPalette
        ^self shouldNotImplement
    Smalltalk.Quadtree class
        instanceVariableNames: ''
    Quadtree class methodsFor: instance creation
    minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue image: aImage
        ^self
            minimalRectangleArea: aAreaValue
            pixelValueTolerance: aToleranceValue
            visibilityIntervals: nil
            image: aImage
    minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue visibilityInter-
vals: anIntervalCollection image: aImage
        aAreaValue >= 0 & (aToleranceValue >= 0)
            ifTrue: [^self basicNew
                        minimalRectangleArea: aAreaValue
                        pixelValueTolerance: aToleranceValue
```

visibilityIntervals: anIntervalCollection

image: aImage]

ifFalse: [self error: #errQuadtreeCreation « #dialogs » 'Minimal rectangle area and pixel value tolerance must be greater then or equal to 0!']

```
Smalltalk defineClass: #QuadtreeNode
    superclass: #{Graphics.Rectangle}
    indexedType: #none
    private: false
    instanceVariableNames: 'childs pixelValue parent '
    classInstanceVariableNames: ''
    imports: ''
    category: '3D - TreeNodes'
QuadtreeNode methodsFor: initialize
initialize
    childs := OrderedCollection withSize: self childCount
QuadtreeNode methodsFor: testing
hasChilds
    childs do: [:child | child isNil ifFalse: [^true]].
    ^false
isLeaf
    ^pixelValue ~= nil
QuadtreeNode methodsFor: accessing
childCount
    ^4
parent
    ^parent
```

```
parent: aQuadtreeNode

        parent := aQuadtreeNode

pixelValue

        ^pixelValue

pixelValue: aValue

        pixelValue := aValue

removeChildAt: aValue

        ^self removeChildAtQuadrant: aValue

QuadtreeNode methodsFor: childs operations

addChildAtQuadrant: quadrant

        ^self addChildAtQuadrant: quadrant withPixelValue: nil

addChildAtQuadrant: quadrant withPixelValue: aValue

        | node |

        self verifyQuadrant: quadrant.

        self isLeaf

                ifFalse:

                  [childs at: quadrant put: (node := self perform: ('createNodeForQuadrant'
, quadrant printString , 'WithValue:') asSymbol with: aValue).

                        ^node]

               ifTrue: [self error: #errQuadtreeNodeChildAddition « #dialogs » 'Impossible
to create childs on leafs!']

childAt: index

        ^self childAtQuadrant: index
```

```
childAtQuadrant: quadrant
	self verifyQuadrant: quadrant.
	^childs at: quadrant
removeChildAtQuadrant: quadrant
	self verifyQuadrant: quadrant.
	childs at: quadrant put: nil
QuadtreeNode methodsFor: private
createNodeForQuadrant1WithValue: aValue
	^self class
		origin: self topLeft
		corner: self center
		pixelValue: aValue
		parent: self
createNodeForQuadrant2WithValue: aValue
	^self class
		origin: self topCenter
		corner: self rightCenter
		pixelValue: aValue
		parent: self
createNodeForQuadrant3WithValue: aValue
	^self class
		origin: self leftCenter
```

```
                corner: self bottomCenter
                pixelValue: aValue
                parent: self
createNodeForQuadrant4WithValue: aValue
        ^self class
                origin: self center
                corner: self bottomRight
                pixelValue: aValue
                parent: self
verifyQuadrant: quadrant
            quadrant >= 1 & (quadrant <= self childCount) & (quadrant isKindOf: Integer)
ifFalse: [self error: #errQuadtreeNodeChild « #dialogs » 'The quadrant must be a Integer >= 1
and <= ' , self childCount printString , '!']
Smalltalk.QuadtreeNode class
        instanceVariableNames: ''
QuadtreeNode class methodsFor: instance creation
origin: originPoint corner: cornerPoint pixelValue: aValue
        ^(self origin: originPoint corner: cornerPoint) initialize
                pixelValue: aValue
origin: originPoint corner: cornerPoint pixelValue: aValue parent: aQuadtreeNode
        ^((self origin: originPoint corner: cornerPoint) initialize
                pixelValue: aValue) parent: aQuadtreeNode
```

```
Smalltalk defineClass: #VolumetricQuadtree
    superclass: #{Smalltalk.Quadtree}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: '3D - Trees'

VolumetricQuadtree methodsFor: private

displayOn: aGraphicsContext withPalette: aPalette
    ^self shouldNotImplement

minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue visibilityIntervals: anIntervalCollection image: aImage depth: aDepthValue
    minRectangleArea := aAreaValue.
    pixelValueTolerance := aToleranceValue.
    visibilityIntervals := anIntervalCollection.
    image := aImage.
    (self generateChildNodesFrom: (root := VolumetricQuadtreeNode
                origin: 0 , 0 , 0
                    corner: image width - 1 , (image height - 1) , aDepthValue
                pixelValue: nil))
        ifFalse: [root := nil].
```

bitsPerPixel := image bitsPerPixel.

image := nil

setZPosition: z ofNode: aVolumetricQuadtreeNode

aVolumetricQuadtreeNode isNil

ifFalse:

[aVolumetricQuadtreeNode corner setZ: aVolumetricQuadtreeNode depth + z.

aVolumetricQuadtreeNode origin setZ: z.

aVolumetricQuadtreeNode hasChilds ifTrue: [1 to: aVolumetricQuadtreeNode childCount do: [:i | self setZPosition: z ofNode: (aVolumetricQuadtreeNode childAtQuadrant: i)]]]

VolumetricQuadtree methodsFor: viewing

viewWithPalette: aPalette

| compoundObject |

compoundObject := JunOpenGL3dCompoundObject new.

self

viewNode: root

inCompoundObject: compoundObject

withPalette: aPalette.

compoundObject show

VolumetricQuadtree methodsFor: private - viewing

viewNode: aVolumetricQuadtreeNode inCompoundObject: aCompoundObject withPalette: aPalette

```
aVolumetricQuadtreeNode isNil ifFalse: [aVolumetricQuadtreeNode isLeaf
                    ifTrue: [aCompoundObject add: (aVolumetricQuadtreeNode
asJunOpenGL3dObjectColor: (aPalette at: aVolumetricQuadtreeNode pixelValue))]
                ifFalse: [1 to: aVolumetricQuadtreeNode childCount do: [:i | self
                                viewNode: (aVolumetricQuadtreeNode
childAtQuadrant: i)
                            inCompoundObject: aCompoundObject
                            withPalette: aPalette]]]

Smalltalk.VolumetricQuadtree class
    instanceVariableNames: ''

VolumetricQuadtree class methodsFor: instance creation

minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue image: aImage
    self error: #errVolumetricQuadtreeCreation << #dialogs >> 'Use #minimalRectangle-
Area:pixelValueTolerance:image:depth: instead.'

minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue image: aImage
depth: aDepthValue
    ^self
        minimalRectangleArea: aAreaValue
        pixelValueTolerance: aToleranceValue
        visibilityIntervals: nil
        image: aImage
        depth: aDepthValue

minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue visibilityInter-
```

vals: anIntervalCollection image: aImage

self error: #errVolumetricQuadtreeCreation « #dialogs » 'Use #minimalRectangleArea:pixelValueTolerance:visibilityIntervals:image:depth: instead.'

minimalRectangleArea: aAreaValue pixelValueTolerance: aToleranceValue visibilityIntervals: anIntervalCollection image: aImage depth: aDepthValue

aAreaValue >= 0 & (aToleranceValue >= 0)

ifTrue: [^self basicNew

minimalRectangleArea: aAreaValue

pixelValueTolerance: aToleranceValue

visibilityIntervals: anIntervalCollection

image: aImage

depth: aDepthValue]

ifFalse: [self error: #errQuadtreeCreation « #dialogs » 'Minimal rectangle area and pixel value tolerance must be greater then or equal to 0!']

```
Smalltalk defineClass: #Octree
    superclass: #{Smalltalk.ThreeDTree}
    indexedType: #none
    private: false
    instanceVariableNames: 'positions '
    classInstanceVariableNames: ''
    imports: ''
    category: '3D - Trees'
Octree methodsFor: private - child generation
generateChildNodesFrom: aOctreeNode with: aQuadtreeNodeCollection
    | maxValue minValue invisibles leafs value areaOk v minDist mostNear |
    maxValue := 0.
    minValue := Infinity positive.
    invisibles := 0.
    leafs := 0.
    minDist := Infinity positive.
    value := aOctreeNode origin + aOctreeNode corner // 2.
    areaOk := aOctreeNode origin ~= value & (aOctreeNode corner ~= value).
    aQuadtreeNodeCollection
        keysAndValuesDo:
            [:key :node |
            value := positions at: key.
```

```
minDist > (v := (value - aOctreeNode center z) abs)
        ifTrue:
            [minDist := v.
            mostNear := node].
value >= aOctreeNode origin z & (value <= aOctreeNode corner z)
& areaOk
        ifTrue:
            [node isNil
                ifTrue: [invisibles := invisibles + 1]
                ifFalse: [node isLeaf
                        ifTrue:
                            [leafs := leafs + 1.
                            maxValue < (value :=
node pixelValue) ifTrue: [maxValue := value].
                            minValue > value ifTrue:
[minValue := value]]
                        ifFalse: [areaOk := false]].
                maxValue - minValue > pixelValueTolerance
| (invisibles > 0 & (leafs > 0)) | areaOk not ifTrue: [^self subdivideNode: aOctreeNode with:
aQuadtreeNodeCollection]]].
    invisibles = 0 & (leafs = 0) ifTrue: [mostNear isNil
            ifTrue: [^false]
            ifFalse: [mostNear isLeaf
```

```
                                        ifTrue:
                                            [aOctreeNode pixelValue: mostNear pixelValue.
                                              ^true]
                                          ifFalse: [^self subdivideNode: aOctreeNode with:
aQuadtreeNodeCollection]]].
        leafs > 0
            ifTrue:
                [aOctreeNode pixelValue: maxValue.
                ^true].
        ^false
    subdivideNode: aOctreeNode with: aQuadtreeNodeCollection
        | removed nQN v |
        removed := 0.
        1 to: 4
            do:
                [:i |
                nQN := OrderedCollection new.
                aQuadtreeNodeCollection do: [:n | n isNil ifFalse: [n isLeaf ifFalse:
[nQN add: (n childAtQuadrant: i)]
                                              ifTrue: [nQN add: n]]
                                        ifTrue: [nQN add: nil]].
                v := i - 4.
                  2 timesRepeat: [(self generateChildNodesFrom: (aOctreeNode
```

```
addChildAtOctant: (v := v + 4))
                                        with: nQN)
                                        ifFalse:
                                             [aOctreeNode removeChildAtOctant: v.
                                             removed := removed + 1]]].
          ^removed < 8
   Octree methodsFor: private
   displayOn: aGraphicsContext withPalette: aPalette
          ^self shouldNotImplement
   positionQuadtreeDictionary: aDictionary pixelValueTolerance: aToleranceValue
          | quadRoots offset firstQuadtree |
          positions := aDictionary keys asSortedCollection asOrderedCollection.
          pixelValueTolerance := aToleranceValue.
          quadRoots := OrderedCollection new.
          offset := positions first.
          firstQuadtree := aDictionary at: offset.
          positions
               keysAndValuesDo:
                    [:i :value |
                    quadRoots add: (aDictionary at: value) root.
                    positions at: i put: value - offset].
          (self generateChildNodesFrom: (root := OctreeNode
```

origin: 0 , 0 , 0

corner: firstQuadtree imageWidth - 1 , firstQuadtree imageHeight - 1 , positions last

pixelValue: nil) with: quadRoots)

ifFalse: [root := nil].

positions := nil

Octree methodsFor: accessing

imageDepth

root isNil

ifTrue: [^0]

ifFalse: [^root depth + 1]

Smalltalk.Octree class

instanceVariableNames: ''

Octree class methodsFor: instance creation

positionQuadtreeDictionary: aDictionary pixelValueTolerance: aToleranceValue

| trees height width |

aDictionary size < 2 ifTrue: [self error: #errOctreeCreation « #dialogs » 'At least 2 Quadtress must be suplied!'].

trees := aDictionary values.

height := trees first imageHeight.

width := trees first imageWidth.

2 to: trees size do: [:i | height = (trees at: i) imageHeight & (width = (trees at: i) imageWidth) ifFalse: [self error: #errOctreeCreation « #dialogs » 'The Quadtrees must have

the same size!']].

ˆself basicNew positionQuadtreeDictionary: aDictionary pixelValueTolerance: aToleranceValue

```
Smalltalk defineClass: #OctreeNode
        superclass: #{Jun.Jun3dBoundingBox}
        indexedType: #none
        private: false
        instanceVariableNames: 'childs pixelValue parent '
        classInstanceVariableNames: ''
        imports: ''
        category: '3D - TreeNodes'
OctreeNode methodsFor: initialize
initialize
        childs := OrderedCollection withSize: self childCount
OctreeNode methodsFor: testing
hasChilds
        childs do: [:child | child isNil ifFalse: [^true]].
        ^false
isLeaf
        ^pixelValue ~= nil
OctreeNode methodsFor: accessing
childCount
        ^8
parent
        ^parent
```

```
parent: aOctreeNode
        parent := aOctreeNode
pixelValue
        ^pixelValue
pixelValue: aValue
        pixelValue := aValue
removeChildAt: aValue
        ^self removeChildAtOctant: aValue
OctreeNode methodsFor: private
createNodeForOctant1WithValue: aValue
        ^self class
                origin: self origin x , self origin y , (self corner z + self origin z // 2)
                corner: self origin x + self corner x // 2 , (self origin y + self corner y // 2) ,
self corner z
                pixelValue: aValue
                parent: self
createNodeForOctant2WithValue: aValue
        ^self class
                origin: self origin x , (self origin y + self corner y // 2) , (self corner z + self
origin z // 2)
                corner: self origin x + self corner x // 2 , self corner y , self corner z
                pixelValue: aValue
                parent: self
```

```
createNodeForOctant3WithValue: aValue
    ^self class
        origin: self origin x + self corner x // 2 , self origin y , (self corner z + self origin z // 2)
        corner: self corner x , (self origin y + self corner y // 2) , self corner z
        pixelValue: aValue
        parent: self
createNodeForOctant4WithValue: aValue
    ^self class
        origin: self origin + self corner // 2
        corner: self corner
        pixelValue: aValue
        parent: self
createNodeForOctant5WithValue: aValue
    ^self class
        origin: self origin
        corner: self origin + self corner // 2
        pixelValue: aValue
        parent: self
createNodeForOctant6WithValue: aValue
    ^self class
        origin: self origin x , (self origin y + self corner y // 2) , self origin z
```

```
corner: self origin x + self corner x // 2 , self corner y , (self origin z + self corner z // 2)
pixelValue: aValue
parent: self
createNodeForOctant7WithValue: aValue
ˆself class
origin: self origin x + self corner x // 2 , self origin y , self origin z
corner: self corner x , (self origin y + self corner y // 2) , (self origin z + self corner z // 2)
pixelValue: aValue
parent: self
createNodeForOctant8WithValue: aValue
ˆself class
origin: self origin x + self corner x // 2 , (self origin y + self corner y // 2) , self origin z
corner: self corner x , self corner y , (self origin z + self corner z // 2)
pixelValue: aValue
parent: self
verifyOctant: octant
octant >= 1 & (octant <= self childCount) & (octant isKindOf: Integer) ifFalse: [self error: #errOctreeNodeChild « #dialogs » ’The octant must be a Integer >= 1 and <= ’ , self childCount printString , ’!’]
OctreeNode methodsFor: childs operations
```

```
addChildAtOctant: octant

    ^self addChildAtOctant: octant withPixelValue: nil

addChildAtOctant: octant withPixelValue: aValue

    | node |

    self verifyOctant: octant.

    self isLeaf

        ifFalse:

            [childs at: octant put: (node := self perform: ('createNodeForOctant'
, octant printString , 'WithValue:') asSymbol with: aValue).

            ^node]

        ifTrue: [self error: #errOctreeNodeChildAddition « #dialogs » 'Impossible
to create childs on leafs!']

childAt: index

    ^self childAtOctant: index

childAtOctant: octant

    self verifyOctant: octant.

    ^childs at: octant

removeChildAtOctant: octant

    self verifyOctant: octant.

    childs at: octant put: nil

Smalltalk.OctreeNode class

    instanceVariableNames: ''

OctreeNode class methodsFor: instance creation
```

```
origin: originPoint corner: cornerPoint pixelValue: aValue

    ^(self origin: originPoint corner: cornerPoint) initialize pixelValue: aValue

origin: originPoint corner: cornerPoint pixelValue: aValue parent: aOctreeNode

    ^((self origin: originPoint corner: cornerPoint) initialize pixelValue: aValue)

        parent: aOctreeNode
```