UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE INFORMÁTICA

Visualização de Terrenos em Tempo Real

Hugo Castelo Pires

Dissertação submetida à Universidade do Minho
para obtenção do grau de Mestre em Informática

Toda a actividade científica sob orientação do
Prof. António José Borba Ramires Fernandes

Janeiro 2003

Resumo

A visualização de terrenos virtuais em tempo real requer algoritmos capazes de simplificarem os dados iniciais por forma a permitirem uma navegação suave e precisa com *frame rates* elevadas.

Esta tem sido uma área de investigação na qual têm surgido várias publicações recentes que apresentam soluções distintas, nomeadamente os algoritmos de Lindstrom et al., de Röttger et al. e o ROAM.

Nesta dissertação realizou-se um estudo sobre a aplicabilidade do mecanismo de *display lists* existente no *OpenGL* nos algoritmos de visualização de terrenos em tempo real. Apesar da utilização de *display lists* não ser comum neste tipo de algoritmos devido à natureza dinâmica da geometria a visualizar, concluiu-se nesta dissertação que o recurso a este mecanismo produz resultados relevantes.

Foi desenvolvido um novo algoritmo, baseado no algoritmo de Röttger et al., que recorre às *display lists* para a visualização contínua, em tempo real, de terrenos de grandes dimensões. O algoritmo utiliza mapas regulares de alturas cuja complexidade é reduzida utilizando níveis de detalhes contínuos, os quais são mantidos em *display lists* que auxiliam e melhoram a velocidade da visualização.

Foram desenvolvidas três diferentes versões do algoritmo: a primeira versão utiliza *display lists* hierárquicas; a segunda utiliza apenas *display lists* simples ligadas por uma lista duplamente ligada; a terceira utiliza a mesma lista duplamente ligada de *display lists* simples e procura tirar partido da coerência entre *frames*.

Abstract

Real time terrain rendering requires specific view dependent continuous level of detail algorithms in order to achieve smooth and accurate navigation with high frames rates.

Many publications in this area have emerged recently, presenting different solutions, namely Lindstrom et al., Röttger et al. and ROAM.

In this thesis, a study on OpenGL's display list mechanism applied to real time terrain rendering algorithms is presented. Although the use of display lists is not common in this type of algorithms, due to the dynamic nature of the triangle mesh, a significant performance increase can be obtained with high frame-to-frame coherence.

A new algorithm is proposed, based on the work by Röttger et al.. It uses display lists for real time continuous terrain rendering. The algorithm simplifies the height-field data and stores the results in display lists.

Three different versions of the algorithm were implemented: the first one uses hierarchical display lists; the second one uses simple display lists stored in a double linked list; the last one takes advantage of frame-to-frame coherence.

Agradecimentos

Muitas pessoas apoiaram-me no desenvolvimento desta dissertação. Saliento o meu orientador, o Prof. Dr. António Ramires Fernandes, cuja ajuda e orientação foram essenciais para a conclusão deste trabalho, e todos os restantes elementos do Grupo SIM - Software, Interacção e Multimédia do Departamento de Informática da Universidade do Minho.

Agradeço ainda à Fundação para a Ciência e a Tecnologia pelo apoio financeiro que me deram com a atribuição de uma bolsa de mestrado PRAXIS XXI.

E, para terminar, sendo que os últimos são para mim os primeiros, toda a minha família, em especial, a minha esposa Sandra e os meus pais, cujo apoio constante permitiram levar a bom porto esta dissertação.

Índice

Lista de Figuras

1 Introdução

Com o aumento do desempenho nos computadores pessoais, a distância entre o virtual e o real estreita-se. As imagens geradas por computador apresentam-se cada vez com mais resolução e mais qualidade. As capacidades gráficas dos sistemas permitem, hoje em dia, interagir com os modelos[1] visualizados dentro de parâmetros inalcançáveis à apenas alguns anos atrás.

Durante a visualização interactiva, muitos são os tipos de modelos que competem pelos recursos limitados do sistema. Dentre eles, os terrenos são, provavelmente, um dos tipos que apresentam um maior desafio devido à sua natureza e extensão geométrica.

Há mais de duas décadas que os terrenos têm sido objecto de estudo na computação gráfica [23]. Com diversas aplicações em diferentes áreas, a utilização de terrenos virtuais é, sem dúvida, importante nas várias aplicações que deles dependem [15]. A título de exemplo podem-se considerar algumas áreas de aplicação:

- *Simuladores de Voo e Viação*, com aplicação na fase de instrução de pilotos e condutores, e, na área militar, no planeamento de missões, procurando reproduzir as condições do terreno de acção.
- *Visitas Virtuais*, possibilitando visitas a locais existentes outrora e actualmente destruídos, como o Motsuji Temple Paradise Garden destruído à cerca de 770 anos,

[1] Chama-se a um conjunto de entidades geométricas que detenham alguma relação geométrica *modelo* ou *objecto*. Essas entidades geométricas, também chamadas de primitivas geométricas, serão pontos, linhas ou triângulos dado serem estas as primitivas mais utilizadas pelo *hardware* gráfico disponível no mercado.

Ao conjunto de todos os modelos, incluindo tudo o que diz respeito ao ambiente a visualizar (descrição do material dos modelos, iluminação, parâmetros da câmara, ...) chama-se *mundo virtual* ou, simplesmente, *mundo*.

a locais inacessíveis, como o Stonehenge no qual não é permitido actualmente ao público circular por entre as pedras, a locais inóspitos, como o Vulcão Kilaues no Havai, para além de muitos outros de interesse turístico e cultural, como as cidades de Atenas e São Francisco.

- *Entretenimento*, onde cada vez mais surgem jogos cuja acção decorre no exterior – jogos de condução, estratégia e simuladores.
- *Atlas Virtuais*, para auxilio no estudo de ciências como a geografia, a geologia, a meteorologia, entre outras.
- *Sistemas de Informação Geográfica (SIG)*, permitindo a visualização no terreno de dados geo-referenciados.

A visualização de terrenos em tempo real tem sido um assunto aceso no decorrer dos últimos anos e apresenta-se ainda hoje como um desafio [15][20][23][33]. No entanto, a utilização de terrenos virtuais não se resume apenas à visualização de uma superfície. Muitas são as questões e situações a ter em conta antes e durante essa visualização que definem, por si só, áreas de investigação complementares. Salienta-se por exemplo:

- A origem dos dados (altitudes), que podem resultar de um levantamento no terreno utilizando, por exemplo, um GPS diferencial, de uma combinação de levantamentos quando a mesma área se encontra representada em formatos diferentes e com diferentes resoluções, ou da geração de terrenos artificial [5][6][10].
- A utilização de texturas de grandes dimensões, que levanta questões de memória, quando a textura excede a memória disponível, questões de *caching* e *paging* quando a mesma não está totalmente em memória, e questões de detalhe quando a textura tem uma resolução inferior à desejada.
- A utilização de oceanos, lagos, e cursos de água em simultâneo com o terreno, cujos dados são normalmente representados em formatos diferentes.
- A utilização, colocação e visualização de edifícios, vias de comunicação, vegetação e ecossistemas que acarretam uma panóplia de situações que são necessárias resolver.

Dada a quantidade de dados em causa, faltam ainda largos anos até que seja possível visualizar "à força bruta" terrenos virtuais de grandes dimensões e toda a informação a ele associada [8][13][22][23][29]. A abordagem a este problema passa, como em inúmeras outras áreas da computação gráfica, por estabelecer um compromisso entre a fidelidade de representação dos modelos e o desempenho da aplicação. Mais realismo implica menor velocidade, maior precisão

implica uma menor taxa de *frames* por segundo (FPS), maior detalhe implica uma deterioração na navegação e interacção com os modelos.

Na área dos terrenos virtuais, a procura de uma solução que satisfaça estes compromissos tem sido repartida, como mencionado anteriormente, em diferentes áreas de investigação complementares, que têm evoluído paralelamente.

Alcançada uma possível solução, é nossa condição humana querermos sempre mais e melhor. A vontade de alcançarmos o inatingível persegue-nos ao longo de toda a nossa história, e, na era da tecnologia, o desejo de se ter algo nunca visto impõe ao mercado toda uma pressão que exige novas soluções.

Em linhas gerais, o objectivo é o de, tendo grandes quantidades de informação, visualizar e interagir em tempo real com os modelos associados. Como exemplo, considere-se um terreno de $4km^2$ amostrado metro a metro, com altitudes de 32 bits. A quantidade de informação necessária para armazenar estas amostras de altitude é de cerca de 64MB, excluindo a informação relativa a textura e cor. Uma visualização à "força bruta" resultaria em cerca de 32 milhões de triângulos a processar em cada *frame*.

Tais condições implicam a implementação de um algoritmo que simplifique a triangulação a visualizar, isto é, que reduza o número de triângulos reduzindo o número de amostras a utilizar, pois não existe actualmente *hardware* gráfico capaz de processar/visualizar um tão elevado número de triângulos/vértices numa fracção de segundo.

O algoritmo deve ainda visar os seguintes objectivos gerais:

- Uma visualização o mais precisa possível, isto é, a triangulação simplificada deve ser o mais próxima do terreno original quanto possível, introduzindo um erro dentro de limites pré-estabelecidos.
- Esta visualização deve ser efectuada com uma taxa de FPS elevada e, tanto quanto possível, constante, permitindo desta forma uma interactividade real.
- Deve ser evitada a introdução de descontinuidades na triangulação a visualizar.

A simplificação das amostras do terreno por forma a permitir a visualização interactiva de uma representação da superfície do terreno é o tema central desta dissertação.

Os algoritmos existentes, apesar de produzirem resultados bastante positivos, baseiam-se no modo imediato. A principal justificação para tal é o facto da representação do terreno ser altamente dinâmica. A aceleração destes algoritmos tirando partido do *hardware* dos sistemas gráficos não tem sido portanto muito explorada.

É sobre este tópico que irá recair o trabalho de investigação desta dissertação. Para este efeito, desenvolveu-se um novo algoritmo para a visualização interactiva de terrenos de grandes dimensões, com uma taxa de FPS elevada, à custa da aceleração por *hardware* disponível nas placas gráficas actuais.

Recorre-se ao mecanismo de *display lists*[2], existente no *OpenGL*[3] [28], para acelerar o processo de visualização. Apesar das *display lists* serem normalmente utilizadas na visualização de modelos estáticos, os testes efectuados mostram que estas permitem uma visualização eficiente da representação dinâmica do terreno (ver capítulo *6 – Testes e Análise de Resultados*).

1.1 Organização da Dissertação

Esta dissertação encontra-se organizada nos seguintes capítulos.

Capítulo 2 - Conceitos Básicos de Computação Gráfica. Neste capítulo faz-se uma breve introdução a conceitos básicos de Computação Gráfica. Apresentam-se os diferentes sistemas de coordenadas envolvidos no *pipeline* gráfico e as diversas fases que o constituem.

Capítulo 3 - A Visualização de Terrenos. Neste capítulo aborda-se a origem e as estruturas de dados utilizadas para a representação dos terrenos. Apresenta-se uma classificação para os algoritmos de visualização de terrenos em tempo real e algumas questões relevantes para estes algoritmos.

Capítulo 4 - Algoritmos de Visualização de Terrenos em Tempo Real - (Estado da Arte). Neste capítulo descreve-se um sub conjunto do trabalho desenvolvido nesta área. Deu-se especial relevo aos algoritmos que tratam desta problemática construindo uma estrutura hierárquica para a simplificação das amostras do terreno.

Capítulo 5 - Contribuição Científica. Neste capítulo é apresentado o trabalho de investigação que resultou na contribuição científica desta dissertação para este campo de investigação, nomeadamente a apresentação de um novo algoritmo para a visualização em tempo real de terrenos de grandes dimensões.

Capítulo 6 - Testes e Análise de Resultados. Neste capítulo apresentam-se os testes efectuados e uma análise comparativa dos resultados obtidos por diferentes variações do algoritmo

---

[2] Uma *display list* é uma lista estática de comandos gráficos.

[3] O *OpenGL* (*Open Graphics Library*) é uma interface para o *hardware* gráfico. Esta interface consiste num conjunto de comandos (funções) que permitem especificar um ambiente virtual e produzir uma aplicação interactiva tridimensional [28][35].

desenvolvido no âmbito desta dissertação e pelos algoritmos discutidos nos capítulos anteriores para a mesma problemática.

Capítulo 7 - Conclusão. Neste capítulo faz-se um breve resumo de tudo o que foi abordado nesta dissertação, identificam-se alguns dos problemas que não foram resolvidos e apresentam-se possíveis soluções a serem desenvolvidas em trabalho futuro.

1.2 Taxinomia e Definições

Nesta secção são apresentadas as principais notações utilizadas nesta dissertação.

Sempre que possível traduziu-se para português os termos técnicos anglo-saxónicos, sendo o termo original apresentado em nota de rodapé aquando da primeira tradução. Para os termos em que não foi encontrada uma tradução consensualmente aceite pela comunidade científica Portuguesa decidiu manter-se a terminologia anglo-saxónica, sendo os termos, neste caso, apresentados em *itálico*.

Os principais termos técnicos que figuram nesta dissertação foram compilados num glossário (ver *Glossário*), onde se apresenta uma breve descrição dos mesmos.

Sempre que um texto é utilizado frequentemente utilizam-se acrónimos, que serão apresentados sempre em MAIÚSCULAS.

As linhas de código e pseudo-código incluídas são apresentadas dentro de um quadro conforme o exemplo que se segue:

```
Linhas de código são apresentadas assim...
```

2 Conceitos Básicos de Computação Gráfica

Neste capítulo faz-se uma breve introdução a conceitos básicos de Computação Gráfica. Apresenta-se o *pipeline* gráfico [9][12][25] e as diversas fases que o constituem, no sentido de contextualizar todos os conceitos descritos.

O *pipeline* gráfico compreende todo o processo desde a especificação de uma cena tridimensional até à sua visualização num ecrã bidimensional ou em qualquer outro periférico de saída.

Devido à natureza essencialmente sequencial da geração de imagens tridimensionais (3D) num computador, todo este processo é dividido em várias fases, que no seu conjunto formam um *pipeline* gráfico. A ordem segundo a qual estas etapas são executadas é habitualmente a que se apresenta nesta dissertação. No entanto, dado o número e a complexidade das etapas existentes, esta ordem pode variar ligeiramente de implementação para implementação.

Genericamente, um *pipeline* consiste numa sequência de diferentes fases que podem ser executadas em paralelo sobre dados mutuamente independentes. Num *pipeline* não é possível avançar os dados de uma fase `n` para a próxima fase `n+1` se os dados dessa fase seguinte `n+1` não tiverem, também eles, transitado para a fase `n+2`. Deste modo, a velocidade de execução do *pipeline* fica naturalmente condicionada pela execução da sua fase mais lenta.

Uma das principais vantagens da utilização de um *pipeline* na computação gráfica 3D é permitir que a computação das diferentes fases seja efectuada em paralelo, o que obviamente aumenta o desempenho do sistema. Esta vantagem advém do facto de algumas fases do *pipeline* serem mutuamente independentes, trabalhando umas sobre vértices, outras sobre triângulos e outras ainda sobre *pixels*.

Mais ainda, como nenhuma informação é passada de um vértice para outro ou de um *pixel* para o outro, a computação de dois ou mais vértices e de dois ou mais *pixels* é também mutuamente

independente e logo passível de ser efectuada em simultâneo. Consequentemente, os fabricantes de *hardware* gráfico implementam actualmente, em algumas das fases do *pipeline* gráfico, múltiplos *pipelines* para o processamento em paralelo de *pixels* e vértices.

Por outro lado, a utilização de um *pipeline* dificulta a obtenção de informação relativa a um vértice após se ter iniciado o seu processamento no *pipeline*, tornando o custo desta operação elevado em termos de desempenho.

Num *pipeline* gráfico, algumas fases são executadas exclusivamente em *software*, outras em *hardware*, e outras ainda dependem da arquitectura em que se está a trabalhar, sendo que, há actualmente uma tendência em passar a maior quantidade possível de fases do *pipeline* gráfico para *hardware* dedicado por forma a acelerar essas mesmas etapas.

Todas as fases são passíveis de serem optimizadas. As implementadas em *software* são aquelas sobre as quais o programador tem um maior controlo, pois é-lhe possível modificar directamente o código de implementação, o que facilita a sua optimização. As fases implementadas total ou parcialmente em *hardware*, são mais difíceis de serem optimizadas. A sua optimização passa, fundamentalmente, por fazer diminuir o tempo consumido por estas fases, reduzindo os dados que estas têm de processar, isto é, diminuindo a quantidade de informação a ser utilizada nestas fases. Um exemplo é a diminuição do número de triângulos que constituem uma cena, o que leva a redução do tempo de cálculo necessário à sua visualização.

Estruturalmente, o *pipeline* gráfico é composto por 3 fases conceptuais [25] (ver Figura 2-1): Aplicação, Geometria e *Rasterizer*. Algumas destas fases são ainda constituídas por várias sub-fases que formam, também elas, um novo *pipeline*.

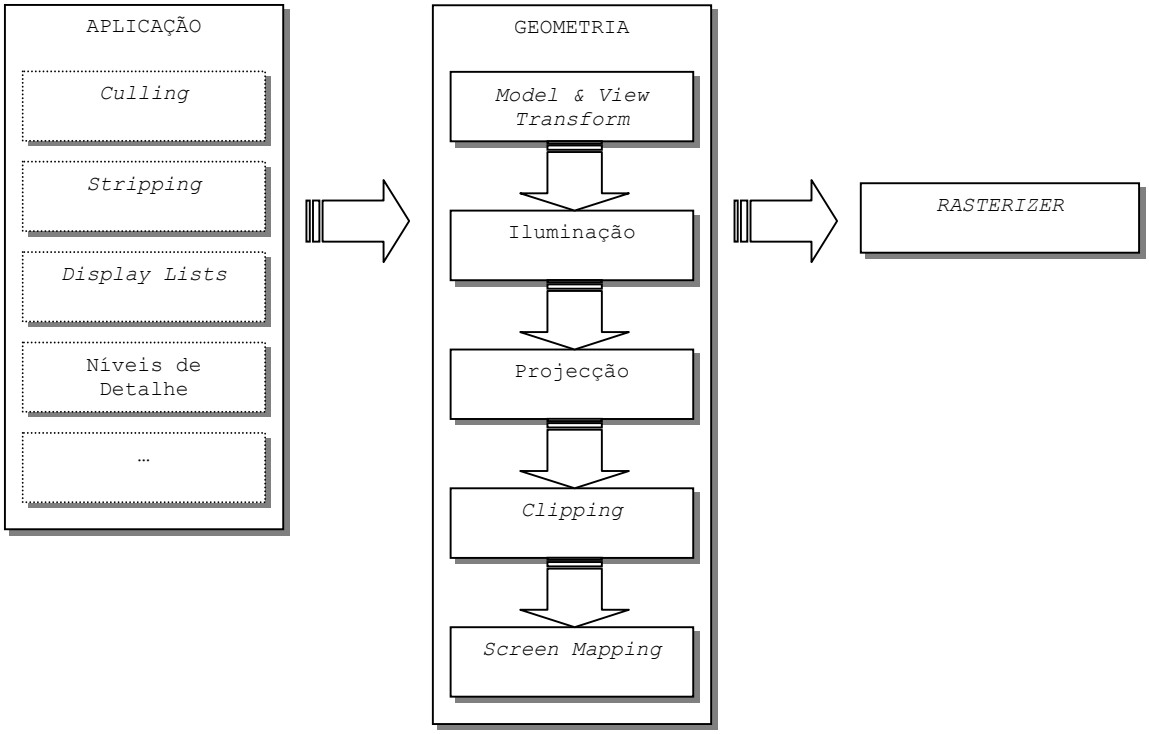


Figura 2-1: Fases do *pipeline* gráfico.

2.1.1 A Fase de Aplicação

Esta é a primeira fase do *pipeline* gráfico, e nela o programador tem total controlo sobre a implementação. É aqui que se definem e criam as especificidades de uma aplicação e tudo que a ela diz respeito.

É fundamentalmente na fase de Aplicação que as optimizações introduzidas nesta dissertação vão ocorrer. É aqui que normalmente se efectuam processos como a definição e selecção dos níveis de detalhe[4] a utilizar, detecção de colisões e oclusões, processamento de eventos de entrada para aplicações interactivas, animações, *morphing*, *culling*, geração de estruturas de dados optimizadas, nomeadamente a construção de tiras[5] e leques[6] de triângulos, de entre outras.

2.1.1.1 *Culling*

Do total de polígonos que constituem o mundo, apenas parte contribui para a imagem final. É neste contexto que surge uma técnica designada por *culling*. O seu objectivo é o de determinar

[4] Do anglo-saxónico, *level of detail*.

[5] Do anglo-saxónico, *strip*.

[6] Do anglo-saxónico, *fan*.

quais os polígonos que não interferem na imagem final, e evitar, logo que possível, o seu processamento e passagem para as restantes fases do *pipeline*. A aplicação de *culling*, em mundos com um elevado número de polígonos, permite aumentar o desempenho do sistema gráfico ainda que à custa de um aumento na utilização do CPU [25].

Existem vários tipos de *culling* efectuados durante a fase de Aplicação, dos quais se destacam:

- *View Frustum Culling*. De todos os polígonos do mundo, apenas uma parte se encontra dentro do volume de visualização[7] e fará, potencialmente, parte da imagem final. A realização de testes de visibilidade permite determinar quais os modelos ou polígonos que se encontram dentro desse volume, identificando-se assim quais os que serão potencialmente visíveis na imagem final. Este processo designa-se por *view frustum culling* (ver Figura 2-2).

  Quando aplicada aos terrenos, a realização destes testes de visibilidade é normalmente antecedida por uma partição espacial hierárquica do terreno em células com o objectivo de reduzir o número de testes a serem efectuados:

  - Se uma célula está totalmente fora do volume de visualização, todos os polígonos dentro desse volume são omitidos da imagem final, reduzindo-se desta forma o número total de polígonos a serem visualizados e, consequentemente, melhorando o desempenho.
  - Se, por outro lado, a célula está completamente dentro do volume de visualização, todos os polígonos dentro dessa célula serão incluídos na imagem final.
  - Finalmente, se a célula está parcialmente dentro, o teste é aplicado recursivamente ao nível seguinte da hierarquia.

[7] Do anglo-saxónico, *view volume* ou *view frustum*.

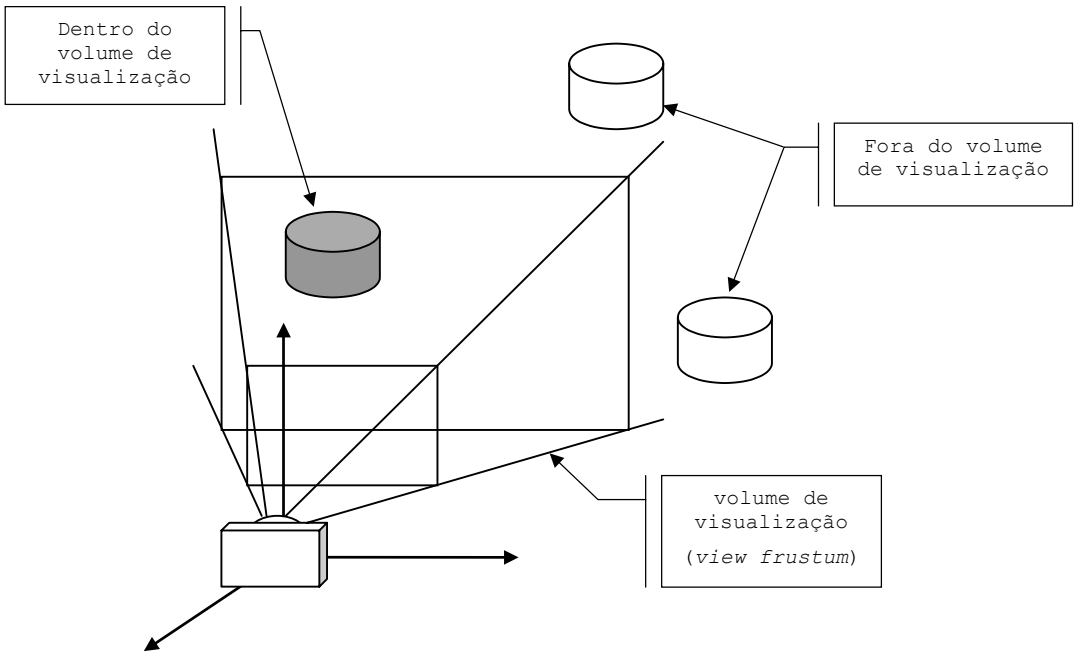


Figura 2-2: Exemplo de *view frustum culling*.

A simplicidade deste tipo de *culling*, aliada a uma eficácia reconhecida [2][8] [17][19][25][26], permite melhorar consideravelmente o desempenho da aplicação com um custo de processamento reduzido.

- *Occlusion Culling*. Nem todos os polígonos que se encontram dentro do volume de visualização são visíveis na imagem final. Em inúmeras situações, polígonos (ou parte deles) encontram-se ocultos por outros. O processamento gráfico destes polígonos ocultos é também desnecessário e deve ser evitado. A todo o processo que determina e permite não processar estes polígonos ocultos designa-se por *occlusion culling*.

  Comparativamente com o *view frustum culling*, este é computacionalmente mais dispendioso, pelo que são poucos os algoritmos que recorrem a este *culling*. Uma excepção é o algoritmo proposto por Mortensen descrito em [26].

2.1.1.2 Stripping

Num modelo, muitos dos vértices são comuns a vários polígonos que o constituem. Assim, a organização dos polígonos com vista à partilha dos vértices comuns traduz-se num envio e processamento únicos destes vértices [25].

O número de vértices a processar é um factor fundamental que determina o desempenho de uma aplicação [34]. Desta forma, a diminuição do número de vértices, para o mesmo número de polígonos, melhora o desempenho, pois ao reduzir o número de vértices reduz a quantidade de dados enviados pelo *pipeline* e, por consequência, reduz também os cálculos necessários ao seu processamento [34]. Este é o objectivo de uma técnica conhecida como *stripping*.

Os tipos de *stripping* mais comuns são as tiras e os leques.

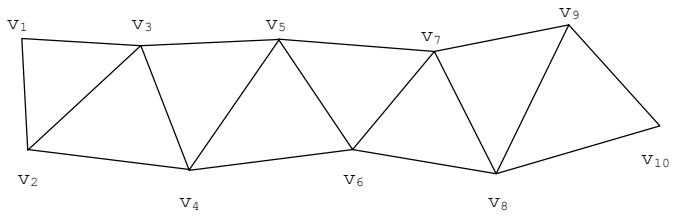


Figura 2-3: Uma tira de 8 triângulos.

Tanto as tiras como os leques são definidas como uma lista ordenada de vértices a partir dos quais se criam os polígonos de acordo com uma regra estabelecida [25].

Numa tira de triângulos são utilizados os três primeiros vértices, $\{v_0, v_1, v_2\}$, para definir o primeiro triângulo. Todos os triângulos seguintes necessitam apenas de um novo vértice, v_i, sendo os outros dois vértices, os dois últimos vértices do triângulo anterior, v_{i-1}, v_{i-2}..

A Figura 2-3 mostra uma tira de 8 triângulos. Para a sua construção foi utilizada a seguinte lista ordenada de 10 vértices: $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$. Se estes triângulos fossem enviados pelo *pipeline* individualmente, isto é, sem partilharem os vértices, seria necessário enviar e processar 24 vértices.

Uma tira de n triângulos, definida desta forma, necessita de $n+2$ vértices versus $3n$ caso se repetissem os vértices.

Um tipo especial de tira é o leque. Num leque o primeiro vértice utilizado define o centro a partir do qual se vão especificando os restantes triângulos. Esse primeiro vértice é comum a todos os triângulos (ver Figura 2-4).

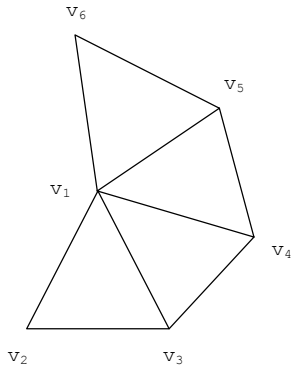


Figura 2-4: Um leque de 4 triângulos.

Tal como nas tiras, apenas são necessários *n+2* vértices para criar um leque de *n* triângulos. Quando aplicado a terrenos, os leques mais utilizados são definidos como uma lista ordenada de vértices obtidos de um conjunto máximo de 9 vértices da forma apresentada na Figura 2-5.

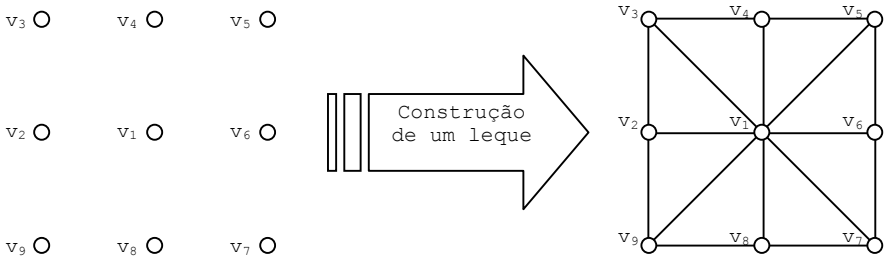


Figura 2-5: Um leque com 9 vértices.

Neste contexto, define-se como leque total a um leque fechado, isto é, a um leque cujo segundo e último vértice da lista ordenada de vértices que o define são o mesmo. Todos os leques não totais são designados por parciais (ver Figura 2-6).

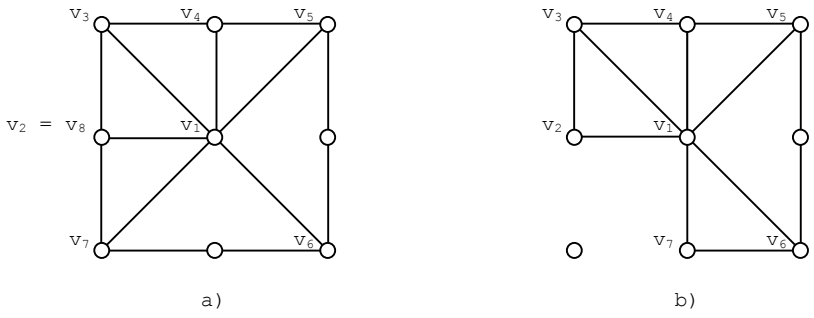


Figura 2-6: a) Exemplo de um leque total. b) Exemplo de um leque parcial.

Define-se por leque completo o leque, total ou parcial, que utiliza todos os vértices que pertencem à superfície por ele definida. Os leques não completos designam-se por incompletos (ver Figura 2-7).

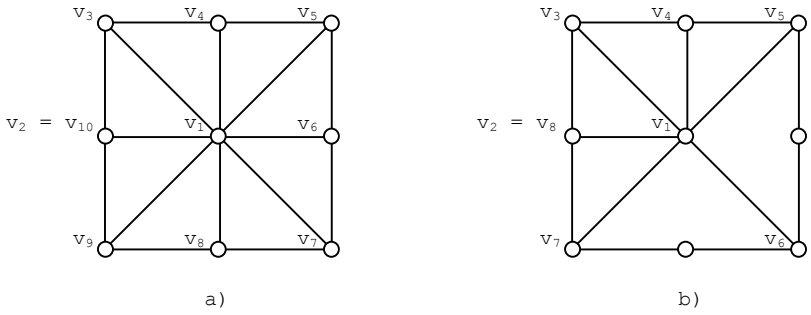


Figura 2-7: a) Exemplo de um leque completo. b) Exemplo de um leque incompleto.

2.1.1.3 Display Lists

O modo de funcionamento base de uma biblioteca gráfica é o modo imediato no qual os comandos gráficos são executados sequencialmente logo que são invocados.

Quando neste modo, a repetição de uma sequência de comandos gráficos implica a repetição do envio e processamento pelo *pipeline* gráfico da mesma informação. Ainda que este modo seja flexível e eficiente em situações dinâmicas, ele torna-se ineficiente quando grandes partes do modelo se mantêm inalteradas. Assim, torna-se necessário tirar partido de um mecanismo que permita armazenar o resultado do processamento do conjunto de comandos gráficos que originam as partes inalteradas de um modelo, evitando o constante reenvio do mesmo conjunto de informação pelas diferentes fases do *pipeline* gráfico e resumindo cada execução desses comandos a uma simples referência [25]. É neste contexto que surgem as *display lists* (ver Figura 2-8).

Uma *display list* é uma lista estática de comandos gráficos de uma biblioteca gráfica organizados de uma forma eficiente [35].

```
// Criação de uma display list com referência n
glNewList(n,GL_COMPILE);
     glColor3f(0.0f,0.0f,0.0f);
     glBegin(GL_TRIANGLE_FAN);
          glVertex3f(x2,0,z2);
          glVertex3f(x1,0,z1);
          glVertex3f(x1,0,z2);
```

```
            glVertex3f(x1,0,z3);
            glVertex3f(x2,0,z3);
            glVertex3f(x3,0,z3);
            glVertex3f(x3,0,z2);
            glVertex3f(x3,0,z1);
            glVertex3f(x2,0,z1);
            glVertex3f(x1,0,z1);
        glEnd();
glEndList();
```

Figura 2-8: Exemplo da definição em *OpenGL* de uma *display list* para armazenar um leque de triângulos.

Normalmente as *display lists* são armazenadas num segmento contínuo de memória, o que melhora os tempos de acesso à mesma [25]. Uma *display list* assemelha-se a uma sequência de comandos previamente compilados, que são normalmente optimizados de acordo com o *hardware* existente.

Não são permitidas quaisquer alterações a *display lists*, pois a sobrecarga resultante da procura de comandos, da gestão de memória e eventuais fragmentações da mesma causada por alocações e dealocações sucessivas, resultaria numa situação menos eficiente. Mais ainda, qualquer optimização realizada pela implementação da API gráfica no decurso da criação da *display list* teria, porventura, de ser novamente realizada. Todas estas situações conduziriam a uma diminuição no desempenho, o que seria contrário ao objectivo inicial de uma *display list*.

Sempre que é necessário efectuar uma alteração, é obrigatório destruir a *display list* e criar uma outra com a informação actualizada. É no entanto possível simular a edição/modificação de *display lists* com recurso a *display lists* hierárquicas. Uma *display list* hierárquica é uma *display list* que executa pelo menos uma outra *display list*, hierárquica ou não (ver Figura 2-9). Dado que a primeira armazena apenas uma referência para a segunda, qualquer redefinição da segunda resulta numa modificação, ainda que indirecta, da primeira.

```
// Criação de uma display list com referência p
glNewList(p,GL_COMPILE);
     glColor3f(0.0f,0.0f,0.0f);
     // Invocação à display list q
     glCallList(q);
     // Invocação à display list r
     glCallList(r);
glEndList();
```

Figura 2-9: Exemplo da criação de uma *display list* hierárquica.

As principais vantagens do uso de *display lists* são:

- *Pré-processamento dos comandos gráficos*. Os comandos gráficos que constituem uma *display list* são processados uma única vez, sendo o resultado optimizado e armazenado num segmento de memória contínuo por forma a acelerar a sua execução.

  De notar que as optimizações operadas sobre *display lists* dependem em grande parte dos controladores utilizados para o *hardware* gráfico disponível e do próprio *hardware* gráfico. Obviamente, quanto maior for o suporte dado por estes elementos para a utilização de *display lists*, melhor será o desempenho resultante.

- *Envio dos comandos em grupo*. A utilização de *display lists* melhora o desempenho do sistema, aumentando o número de FPS, pois permite à biblioteca gráfica o envio dos comandos em grupo para o *pipeline* gráfico, melhorando desta forma a utilização dos barramentos e de todo o *hardware* gráfico [17].

- *Armazenamento em memória dedicada*. Actualmente a maioria do *hardware* gráfico dispõe de memória dedicada, de acesso rápido, na qual é possível armazenar *display lists* [25]. Desta forma, é possível a utilização, em *frame*s sucessivas, do mesmo conjunto de comandos da API gráfica, sem que seja necessário repetir o envio das mesmas instruções da memória central para a memória dedicada do *hardware* gráfico.

As principais desvantagens da utilização de *display lists* são:

- A sobrecarga resultante da sua criação, remoção e manutenção, principalmente em *display lists* de tamanho reduzido.

- O aumento da utilização de memória, pois é necessário armazenar, para além dos comandos gráficos, informação acerca da própria *display list*.

- A imutabilidade do seu conteúdo e a impossibilidade da sua leitura.

2.1.1.4 Níveis de Detalhe

À medida que a distância da câmara a um modelo aumenta, o espaço por este ocupado no ecrã diminui e, consequentemente, o pormenor com que é visualizado também diminui.

Nestas situações, torna-se desnecessário e ineficiente defini-lo com todo o seu pormenor. É neste contexto que surgem os Níveis de Detalhe. O objectivo principal é o de utilizar diferentes representações de um modelo, normalmente de resoluções distintas, que serão seleccionadas de acordo com um critério de decisão pré-determinado. Um dos critérios de decisão mais utilizado é a distância do modelo à câmara.

Existem fundamentalmente dois tipos de Níveis de Detalhe:

- *Discreto* – quando utilizado, a construção das diferentes representações do modelo é realizada numa fase de pré-processamento, sendo associada a cada uma delas um intervalo de distâncias à câmara dentro do qual o nível de detalhe deve ser utilizado. Durante a execução, o algoritmo calcula a distância da câmara ao objecto e avalia qual dos diferentes níveis de detalhe deve ser utilizado.
- *Contínuo* – neste tipo, os níveis de detalhe são dinâmicos, sendo gerados em tempo de execução.

Este assunto será desenvolvido mais detalhadamente na secção *3.2.1 – Nível de Detalhe* para o caso particular dos terrenos.

2.1.2 A Fase de Geometria

A fase de Geometria é computacionalmente exigente, pois é nela que se efectuam a maioria das operações sobre polígonos e vértices [25]. Esta fase é normalmente composta pelas seguintes 5 sub-fases (ver Figura 2-10): *Model & View Transform*, Iluminação, Projecção, *Clipping* e *Screen Mapping*.

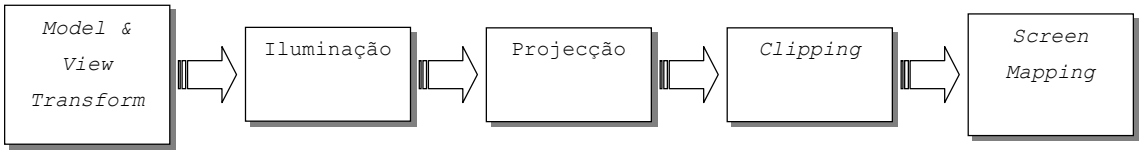


Figura 2-10: A fase de geometria do *pipeline* gráfico.

Durante estas sub-fases são utilizados diferentes sistemas de coordenadas, também designados por espaços. Os modelos tridimensionais do mundo são sequencialmente transformados para 5 espaços distintos ao longo de toda a fase de Geometria (ver Figura 2-11).

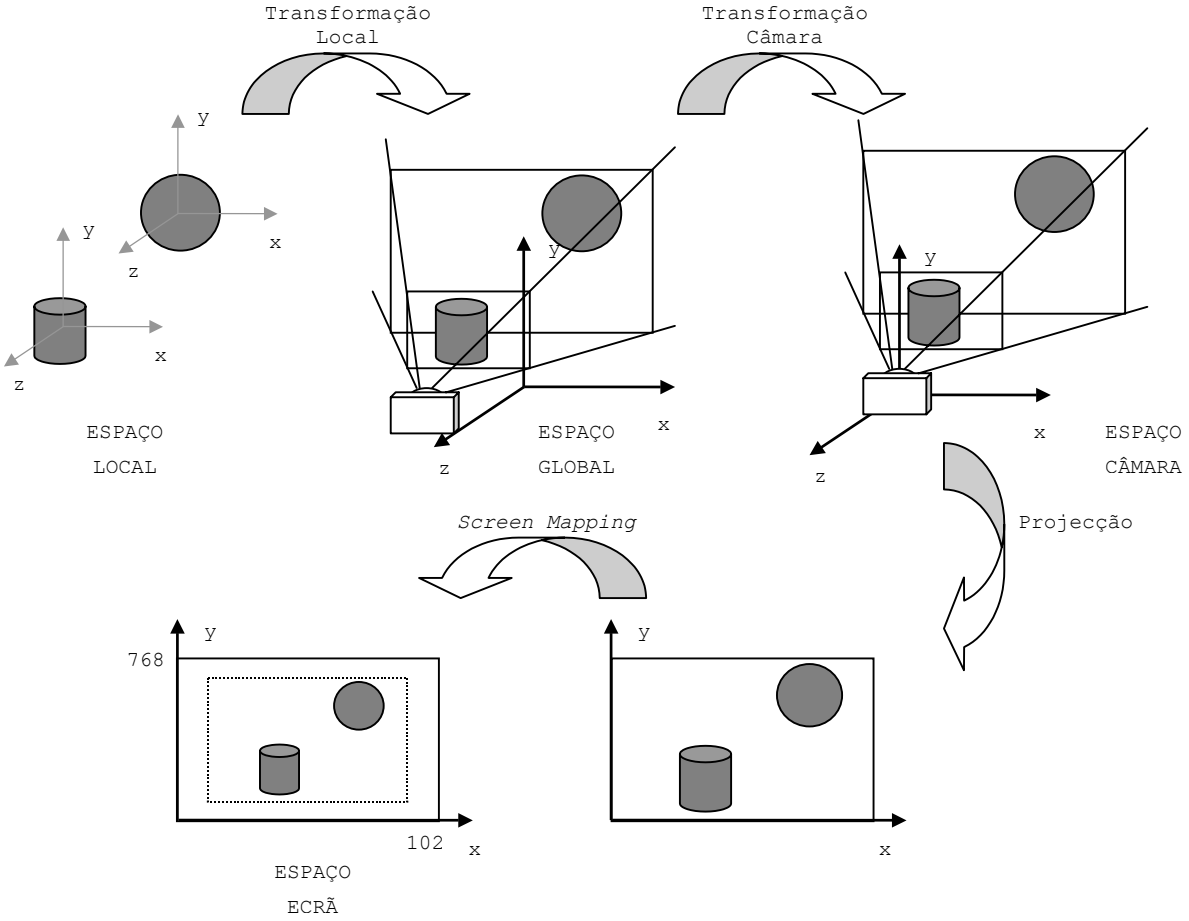


Figura 2-11: Os sistemas de coordenadas envolvidos na fase de geometria do *pipeline* gráfico.

2.1.2.1 Model & View Transform

Na sub-fase *Model & View Transform* transformam-se as coordenadas locais dos modelos em coordenadas câmara.

Inicialmente os modelos residem no espaço local[8]. Neste espaço, cada modelo está posicionado num sistema de coordenadas cartesiano próprio, com uma origem local, a partir da qual todos os pontos constituintes do modelo são definidos. As coordenadas dos pontos do modelo são

[8] Do anglo-saxónico, *model space*.

designadas por coordenadas locais[9]. Os modelos estão geralmente afectos de uma transformação local[10] que define a sua posição e orientação no mundo. Esta operação de transformação é efectuada sobre os vértices e as normais que constituem o modelo e, após a sua aplicação, o modelo passa para o espaço global[11], no qual todos os modelos são colocados num sistema de coordenadas unificado. Neste espaço, as coordenadas dos pontos do modelo são designadas por coordenadas globais[12].

Segue-se a esta uma outra transformação, designada por transformação câmara[13], que converte as coordenadas globais dos modelos em coordenadas câmara[14], passando os modelos a residir no espaço câmara[15] [16]. Neste espaço a câmara encontra-se na origem do sistema de coordenadas, orientada na direcção do eixo negativo `z`, com o eixo `y` a apontar para cima e o eixo `x` a apontar para a direita (ver Figura 2-12).

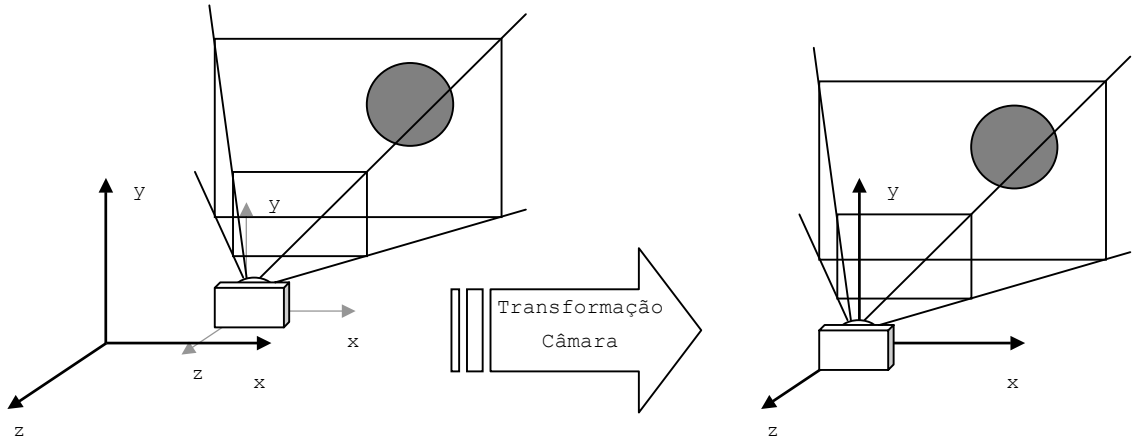


Figura 2-12: Transformação câmara.

[9] Do anglo-saxónico, *local coordinates* ou *model coordinates*.

[10] Do anglo-saxónico, *model transform*.

[11] Do anglo-saxónico, *world space*.

[12] Do anglo-saxónico, *world coordinates*.

[13] Do anglo-saxónico, *view transform*.

[14] Do anglo-saxónico, *camera coordinates*.

[15] Do anglo-saxónico, *view space*.

[16] Em *OpenGL*, as transformações local e câmara são implementadas como matrizes, que, por questões de eficiência, são concatenadas numa só antes de serem aplicadas aos modelos. Assim, na prática, passa-se directamente do espaço local do modelo para o espaço câmara sem se passar pelo espaço mundo.

O objectivo desta transformação é facilitar os cálculos nas sub-fases seguintes de Projecção e *Clipping* [25].

2.1.2.2 Iluminação

Na sub-fase de Iluminação efectuam-se os cálculos necessários à visualização das condições de iluminação definidas no mundo. Um mundo pode ter vários pontos de luz podendo estes afectar ou não os diferentes modelos existentes.

O modelo de iluminação utilizado na computação gráfica em tempo real é uma aproximação simples dos efeitos de luz reais. Quando a luz atinge um polígono, o efeito de luz produzido é determinado apenas pelas propriedades da fonte de luz, pelas propriedades da superfície atingida e pela relação geométrica entre ambas, isto é, os efeitos de luz são determinados localmente para cada polígono, sem considerar as interacções da luz com os restantes modelos/polígonos.

A utilização de técnicas de iluminação mais complexas, como a radiosidade[17] e o *ray-tracing*, permite obter uma iluminação global, onde já se pode observar efeitos como a refracção, reflexão e sombras. No entanto, estas técnicas não se apresentam actualmente como uma solução para sistemas interactivos em tempo real.

2.1.2.3 Projecção

Depois da sub-fase de Iluminação, é efectuada a projecção, que transforma o volume de visualização num cubo unitário chamado de volume de visualização canónico. Os dois tipos de projecção mais comuns são a projecção ortogonal e a projecção em perspectiva.

Quando se utiliza uma projecção ortogonal o volume de visualização é, antes da projecção, um prisma rectangular, cuja principal característica é manter paralelas, após a projecção, todas as linhas paralelas. Desta forma, todos os objectos são visualizados com a dimensão definida, independentemente da sua distância à câmara.

[17] Do anglo-saxónico, *radiosity*.

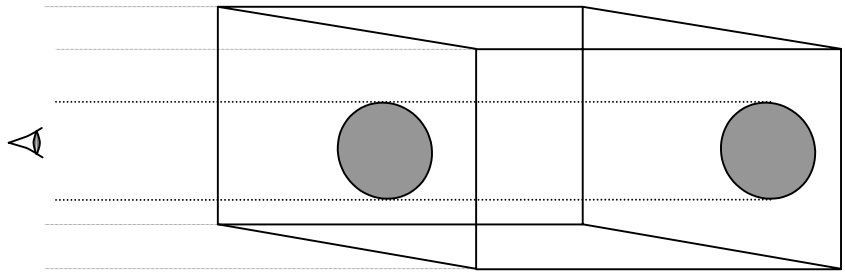


Figura 2-13: Exemplo da utilização de projecção ortogonal.

Ao contrário da projecção ortogonal, quando em perspectiva, as linhas paralelas convergem no horizonte, isto é, os objectos mais afastados da câmara aparecem mais pequenos na imagem final. Neste tipo de perspectiva o volume de visualização, antes da projecção, é uma pirâmide truncada de base rectangular a que se dá o nome de *frustum* (ver Figura 2-14).

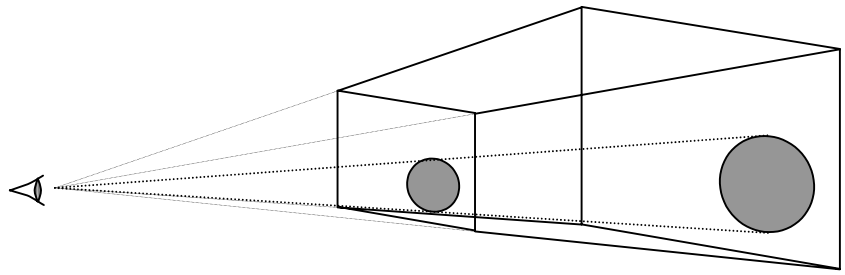


Figura 2-14: Exemplo da utilização de projecção em perspectiva.

Tal como as transformações local e câmara, a projecção, seja ela ortogonal ou em perspectiva, pode ser representada por uma matriz, e após a sua aplicação os modelos estão em coordenadas normalizadas[18].

Embora esta transformação converta um volume num outro de diferentes dimensões, esta operação é, na verdade, uma projecção pois os modelos são projectados de 3 para 2 dimensões. A informação relativa à profundidade (coordenada z) passa a ser guardada num *buffer* especial chamado *z-buffer* ou *buffer* de profundidade, que é utilizado posteriormente para, entre outras coisas, solucionar os problemas de visibilidade (ver secção *2.1.3 – A Fase de Rasterizer*).

2.1.2.4 Clipping

Apenas as primitivas que se encontram total ou parcialmente dentro do volume de visualização canónico necessitam de ser enviadas à fase de *Rasterizing* para serem desenhadas. As que se encontram totalmente fora deste podem ser ignoradas a partir desta fase.

[18] Do anglo-saxónico, *normalized device coordinates*.

As primitivas que intersectam o volume de visualização canónico necessitam de ser divididas em duas partes: a parte que se encontra dentro do volume de visualização canónico e a parte que se encontra fora. As primitivas nestas condições são modificadas, sendo-lhes removida a parte que se encontra fora do volume de visualização canónico. Esta operação passa pela criação de novos vértices na linha de intersecção com o volume de visualização canónico e pela remoção de todos os vértices que se encontram fora desse volume de visualização. As primitivas resultantes são, no final, trianguladas.

A esta operação de modificação das primitivas que se encontram parcialmente dentro do volume de visualização dá-se o nome de *clipping* (ver Figura 2-15).

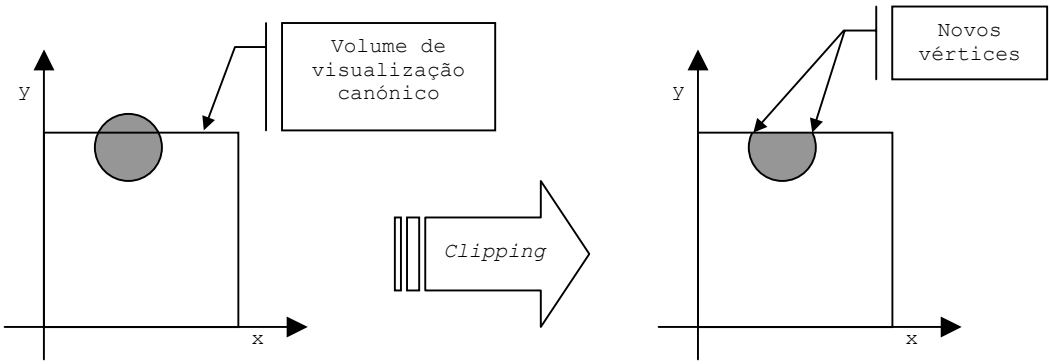


Figura 2-15: Operação de *clipping*.

2.1.2.5 Screen Mapping

Nesta sub-fase, as coordenadas das primitivas resultantes da sub-fase de *Clipping* são mapeadas no ecrã, de acordo com a dimensão deste, efectuando-se para tal a escala necessária. Esta sub-fase não é mais do que uma translação seguida de uma operação de escala. Esta operação converte os modelos para um sistema de coordenadas bidimensional x e y, o qual se designa por espaço ecrã[19], para serem apresentados no ecrã. Os valores de z são mantidos pelo *pipeline* gráfico para realizar testes de profundidade (*z-buffering*).

Neste espaço, as coordenadas dos pontos do modelo são designadas por coordenadas ecrã[20].

[19] Do anglo-saxónico, *screen space*.

[20] Do anglo-saxónico, *screen coordinates*.

2.1.3 A Fase de Rasterizer

No fim do *pipeline* gráfico, é realizada a fase de *Rasterizer* que é quase sempre implementada em *hardware*, ainda que existam raras implementações em *software* [25].

Nesta fase todas as primitivas são desenhadas, isto é, os vértices, representados em coordenadas ecrã, e toda a informação a eles associada (como o valor de profundidade, a cor e a coordenada da textura, quando existe) são convertidos para *pixels* do ecrã.

Esta fase é também responsável pela resolução das questões de visibilidade. Com recurso ao *z-buffer*, onde é armazenada toda a informação de profundidade, é determinado se um *pixel* de uma dada primitiva deve ou não aparecer na imagem final. Durante o desenho de uma primitiva o valor de profundidade de cada *pixel* que a compõe é comparado com o valor de profundidade existente no *z-buffer* (e que corresponde a um *pixel* de uma primitiva previamente desenhada na mesma posição). Se o valor for menor ao existente no *z-buffer*, então o *pixel* da nova primitiva está mais perto da câmara e o *pixel* resultante deve ser actualizado.

3 A Visualização de Terrenos

Nesta dissertação, os terrenos são a base do trabalho desenvolvido. Os dados que os representam são habitualmente obtidos pelo levantamento realizado num terreno real ou como resultado de um algoritmo de geração de terrenos artificiais [5][6][10]. Neste capítulo são apresentadas as estruturas de dados mais utilizadas na representação de terrenos.

De seguida classificam-se os algoritmos utilizados na visualização de terrenos com base no tipo de nível de detalhe que utilizam para obterem uma simplificação dos dados. Por fim, são apresentadas algumas das questões fundamentais que esses algoritmos necessitam tratar e/ou solucionar.

3.1 Estruturas de Dados Comuns para Representação de Terrenos

Os dados que representam terrenos num sistema informático são normalmente armazenados utilizando um dos seguintes tipos de estruturas:

- Os mapas regulares de alturas[21];
- Os grafos irregulares de triângulos[22].

3.1.1 Mapas Regulares de Altura

Os mapas regulares de alturas são uma grelha regular e uniforme de pontos que se adaptam facilmente à representação de terrenos (ver Figura 3-1). É uma estrutura a duas dimensões, x e z, cujas coordenadas distam equitativamente entre si em cada um dos dois eixos. A cada ponto (x,z) corresponde uma altitude.

[21] Do anglo-saxónico, *height-fields*.

[22] Do anglo-saxónico, *triangulated irregular networks*.

Esta característica permite reduzir a quantidade de informação armazenada, pois apenas é necessário conhecer o índice da altitude para se obter o seu valor. Os valores de x e z para uma dada altitude estão assim guardados implicitamente.

Tipicamente esta estrutura é representada por uma matriz bidimensional de altitudes (ver Figura 3-1 c)).

Esta simplicidade na organização dos dados permite, à partida, desenvolver algoritmos mais simples e computacionalmente menos exigentes, facilitando o recurso a estruturas hierárquicas de níveis de detalhe [22][23].

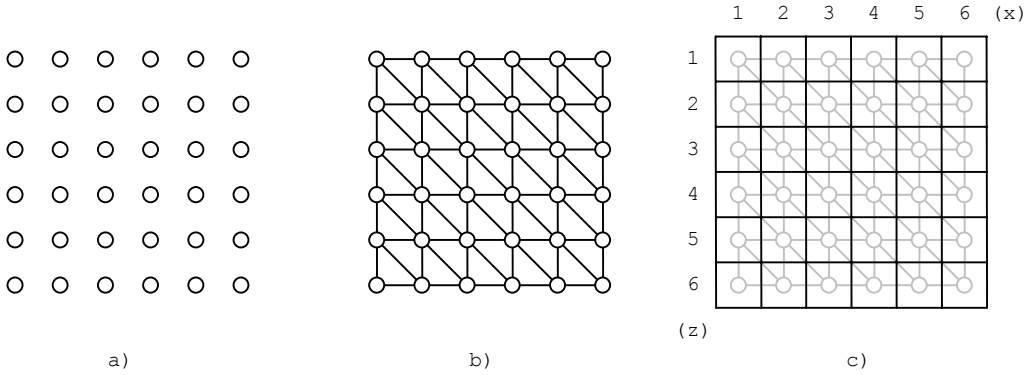


Figura 3-1: a) Pontos que constituem um mapa regular de alturas. b) Exemplo da triangulação regular completa com base no mapa regular de alturas de a). c) Matriz bidimensional para representação do mapa regular de alturas de a).

Por outro lado, os mapas regulares de alturas implicam, normalmente, um número de polígonos na triangulação superior ao necessário para representar adequadamente a superfície do terreno. Tal facto deve-se à obrigatoriedade de armazenar todos os pontos da grelha, mesmo quando não há variações de altitude entre um ponto e os seus vizinhos.

3.1.2 Grafos Irregulares de Triângulos

Os grafos irregulares de triângulos (GIT) são estruturas de dados que permitem uma representação mais precisa da superfície de um terreno.

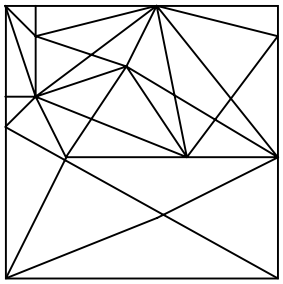


Figura 3-2: Exemplo de um GIT.

A utilização de GITs permite representar com melhor precisão as características específicas de um terreno, pois o processo de amostragem que origina um GIT pode ser orientado pelas variações nas altitudes do terreno e não pela obrigatoriedade de amostragem de todos, e só, os pontos de uma grelha, como é o caso dos mapas regulares de alturas.

Estas estruturas não regulares permitem o espaçamento variável entre as altitudes armazenadas, permitindo uma aproximação mais adequada às irregularidades do terreno [11]. No caso geral, é necessário um menor número de triângulos do que nos mapas regulares de alturas para representar a mesma superfície [22]. A densidade dos pontos está directamente relacionada com a irregularidade do terreno, ou seja, zonas menos acidentadas do terrenos podem ser representadas por menos pontos do que zonas mais montanhosas [11][22].

Apesar da eficiência da aproximação aos terrenos obtida por estas estruturas, a sua irregularidade implica, normalmente, algoritmos mais complexos e computacionalmente mais exigentes, o que dificulta a sua utilização em algoritmos dinâmicos e em tempo real [22]. As *Progressive Meshes* de Hoppe [20] são uma das poucas excepções, sendo um algoritmo de reconhecido sucesso. Este algoritmos será apresentado em detalhe no capítulo seguinte.

3.2 Classificação dos Algoritmos de Visualização de Terrenos em Tempo Real

Quando é necessário visualizar grandes quantidades de informação é imperativo alcançar um compromisso entre a complexidade da representação dos modelos e o desempenho de toda a aplicação. Quando se trata de terrenos de grandes dimensões, a quantidade de informação envolvida é tal que não é possível a utilização da totalidade dos dados num processo de visualização interactivo. É necessário reduzir as amostras do terreno para que uma visualização em tempo real seja possível.

É com base nesta simplificação que se podem classificar os algoritmos desenvolvidos para tratar desta problemática.

3.2.1 Nível de Detalhe

Os algoritmos de visualização de terrenos constroem triangulações que aproximam a superfície total do terreno, com base na sua topologia e numa métrica de erro, utilizando diferentes tipos de níveis de detalhe [9].

O tipo de nível de detalhe utilizado pelos algoritmos pode ser classificado como:

- *Único*. É realizada uma e uma só simplificação considerando os dados originais e é gerada uma única triangulação utilizada repetidamente durante a visualização. Neste caso, a aproximação calculada é realizada numa fase de pré-processamento de acordo com uma métrica independente dos parâmetros da câmara, isto é, os parâmetros da câmara não são considerados na simplificação.

  Normalmente, este tipo de aproximação resume-se ao cálculo de um GIT, pois é o tipo de estrutura de dados que permite obter um menor número de triângulos.

- *Multi-resolução Discreta*. Com este tipo é gerada, numa fase de pré-processamento, uma série de níveis de detalhe de resoluções diferentes. Em tempo de execução, o algoritmo selecciona, para uma dada *frame*, qual dessas aproximações pré-calculadas vai representar o terreno, de modo a minimizar o número de polígonos utilizados e com a menor perda de qualidade possível. Neste caso, o modelo multi-resolução do terreno consiste numa sequência de níveis de detalhe e num critério para selecção entre estes.

  Esta separação entre o processo de computação dos níveis de detalhe e a visualização é uma das vantagens deste tipo de nível de detalhe: o algoritmo de simplificação pode demorar o tempo necessário para produzir, numa fase de pré-processamento, níveis de detalhe que tirem partido de todas as vantagens dos *hardware* gráficos, nomeadamente na organização dos modelos em tiras ou leques de triângulos, *display lists* e *vertex arrays*. Este tempo de computação não condiciona o desempenho do sistema em tempo de execução.

  Algumas das principais desvantagens deste método são:

  - O limitado número de níveis de detalhe disponíveis. A operação de visualização é forçada a seleccionar um dos níveis de detalhe existentes ainda que a melhor aproximação possa estar num nível intermédio entre os existentes.
  - A utilização de métricas independentes dos parâmetros da câmara. No caso dos terrenos, que ocupam uma extensão geométrica considerável,

os níveis de detalhe gerados são obrigatoriamente constantes, isto é, a aproximação utilizada para partes do terreno que se encontram perto ou longe da câmara é a mesma, implicando uma aproximação demasiado densa ao longe e demasiado espaçada ao perto. Esta desvantagem pode, no entanto, ser atenuada, mas não totalmente resolvida, com a divisão do terreno em blocos, blocos esses que serão aproximados independentemente permitindo, durante a visualização, a utilização de níveis de detalhe com resolução distintas em cada um dos blocos.

- O espaço em memória necessário para armazenar todos os níveis de detalhe gerados é, normalmente, comprometedor dada as grandes dimensões dos terrenos utilizados.

- *Multi-resolução Contínua*. Este tipo suplanta as deficiências existentes no método anterior, pois permite variar continuamente o nível de detalhe ao longo da superfície do terreno. Ao contrário do tipo anterior, é construída uma estrutura hierárquica dinâmica, em tempo de execução, que permite abranger todo o espectro de detalhe possível sem a necessidade de armazenamento de todos os níveis intermédios. O nível de detalhe pretendido é extraído em tempo real, durante a fase de visualização, da estrutura criada anteriormente.

  As aproximações contínuas são efectuadas em tempo de execução e a métrica utilizada nestes métodos é normalmente dependente dos parâmetros da câmara, isto é, os parâmetros da câmara são utilizados no cálculo da métrica, o que permite variar o detalhe da aproximação em função da distância e orientação à câmara.

  No entanto, e dado que o nível de detalhe a utilizar é calculado em tempo de execução, é necessária uma maior utilização de CPU em tempo de execução.

3.2.2 Algoritmos Multi-Resolução

Os algoritmos de multi-resolução podem ser classificados de acordo com o tipo de abordagem e simplificação que utilizam. Heckbert e Garland [14][16] classificaram os algoritmos multi-resolução de aproximação a superfícies, por polígonos, baseadas em mapas regulares de alturas. Os dois primeiros tipos apresentados de seguida produzem triangulações regulares e todos os outros produzem GITs. A classificação foi apresentada da seguinte forma:

1. *Métodos de Grelha Regular*[23]. Nestes métodos é utilizada uma sub-grelha da grelha original de altitudes, igual e periodicamente espaçadas, considerando apenas os pontos de cada k-ésima linha e coluna. Este é um método simples e rápido, mas que produz terrenos de baixa qualidade.
2. *Métodos de Divisão Hierárquica*[24]. Nestes métodos a triangulação é construída pela divisão recursiva da superfície representada numa árvore.

   Nesta classe incluem-se todos os algoritmos que serão apresentados nesta dissertação à excepção de Hoppe [18][19][20].
3. *Métodos Orientados à Topologia*. Tendo um conhecimento sobre a natureza do terreno em causa, estes métodos permitem atribuir uma maior importância a um determinado conjunto de pontos do mapa regular de alturas. A estes pontos chamam-se pontos característicos ou pontos críticos, e a triangulação final é obtida a partir destes, utilizando um qualquer método de triangulação dependente dos dados. Estes pontos e os segmentos por eles definidos (conhecidos por *break lines*), representam normalmente características topográficas do terreno, tais como picos, vales e rios.
4. *Métodos de Refinamento*. Estes métodos utilizam uma abordagem *top-down*. Têm como ponto de partida uma triangulação aproximada (no caso de um mapa regular de alturas esta é normalmente constituída somente por dois triângulos) e em cada iteração são inseridos um ou mais pontos nessa triangulação, até que um determinado erro seja alcançado ou um determinado número de vértices seja utilizado. Normalmente os pontos a inserir numa dada iteração são os que minimizam uma determinada métrica de erro.
5. *Métodos de Dizimação*[25]. Estes métodos são o inverso dos métodos de Refinamento. Utilizam uma abordagem *bottom-up*, pois iniciam com o mapa de alturas completo e vão simplificando-o a cada iteração do algoritmo, removendo vértices, triângulos ou outros elementos geométricos.

   Nesta classe incluiu-se o algoritmo de Hoppe [18][19][20].

---

[23] Do anglo-saxónico, *Regular Grid*.

[24] Do anglo-saxónico, *Hierarchical Subdivision*.

[25] Do anglo-saxónico, *Decimation*.

6. *Métodos Óptimos*. Os métodos óptimos foram incluídos apenas pelas suas propriedades teóricas e não pela sua aplicação prática. O problema de determinar a triangulação óptima de um mapa regular de alturas é NP-completo, isto é, uma aproximação óptima não pode ser computada em menos do que tempo exponencial [17] o que seria incomportável.

Nesta dissertação apresenta-se um estudo sobre algoritmos Multi-resolução Contínua, classificados como algoritmos de Divisão Hierárquica em [14][16], e ainda sobre o algoritmo de Hoppe, classificado como Dizimação.

Os algoritmos de Divisão Hierárquica permitem, no decurso da visualização em tempo real, calcular uma nova aproximação ao terreno baseada numa estrutura hierárquica e de acordo com os parâmetros da câmara. Genericamente estes algoritmos seguem o mesmo processo de funcionamento:

1. Em cada *frame*, o primeiro passo é calcular uma nova simplificação das amostras do terreno, através de um processo iterativo. Cada iteração opera sobre uma estrutura hierárquica que representa todo o processo de simplificação.
2. Numa segunda fase, é construída a triangulação, com base na estrutura hierárquica calculada anteriormente, e enviada para o *pipeline* gráfico para visualização.

3.3 Questões

O que distingue os algoritmos de visualização de terrenos em tempo real é o modo como lidam com os problemas e como tiram partido de técnicas existentes para melhorar o seu desempenho.

Nesta secção apresentam-se algumas questões que se levantam na implementação de qualquer algoritmo de visualização de terrenos em tempo real.

3.3.1 Limitações de Memória

Os limitados recursos de memória RAM são obviamente um problema quando se trata de terrenos de grandes dimensões e da sua visualização em tempo real.

A maioria dos algoritmos procura atenuar este problema diminuindo a memória RAM utilizada pelas estruturas de dados. Recorrem às mais diversas estratégias para reduzir a utilização da memória, como por exemplo, a redução do número de bits para a representação dos valores e a utilização de árvores binárias e filas implícitas.

No entanto, nenhuma destas soluções é escalável. Basta considerarmos um terreno cujas necessidades em termos de armazenamento ultrapassem a memória RAM disponível.

A solução deste problema passa pelo carregamento e utilização da memória disponível com os dados estritamente necessários num dado instante. Dos algoritmos apresentados nesta dissertação apenas os de Ulrich (secção *4.3 – Continuous LOD Terrain Meshing Using Adaptive Quadtrees*), Turner (secção *4.6.2 – Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*) e Hoppe (secção *4.8 – View-Dependent Progressive Meshes*) abordam esta problemática.

Este problema não foi, no entanto, desenvolvido no âmbito desta dissertação pelo que o novo algoritmo aqui apresentado não lida com esta questão.

3.3.2 Métricas

A métrica quantifica o erro existente na aproximação utilizada pelo algoritmo relativamente aos dados exactos do terreno. Consoante o seu valor, o algoritmo opta pelo nível de detalhe adequado para representar uma dada região do terreno.

Nos algoritmos de visualização de terrenos em tempo real as métricas são, normalmente, dependentes dos parâmetros da câmara e devem considerar os seguintes factores de modo a torná-las mais eficientes:

- *Factor Variância* – zonas planas de um terreno devem ser aproximadas com menos triângulos do que zonas montanhosas.
- *Factor Distância* – zonas próximas da câmara devem ser aproximadas com mais triângulos, logo com maior detalhe, do que zonas afastadas.

Em muitos dos algoritmos existentes é possível a utilização de diferentes métricas. As métricas mais utilizadas são:

- *Erro Geométrico*. Esta métrica baseia-se exclusivamente no factor Variância, calculando o erro geométrico para cada ponto do terreno. O erro geométrico é a diferença entre o valor da altitude amostrada num ponto do terreno e o valor da altitude utilizada na triangulação para esse mesmo ponto. Esta métrica é utilizada por McNally em [24].
- *Erro Geométrico + Distância*. Esta métrica combina os factores Variância e Distância, dando um maior peso ao erros geométricos mais próximos da câmara. Esta métrica é utilizada por Ulrich em [33], por Turner em [32] e por Röttger et al. em [29].

- *Erro Geométrico em Pixels*[26]. Esta métrica é a mais precisa, e a computacionalmente mais exigente, de todas aqui apresentadas. Tal como a anterior, combina os dois factores Variância e Distância. No entanto, o erro é calculado projectando no ecrã o segmento de recta correspondente ao erro geométrico num ponto do terreno e medindo o número de *pixels* dessa projecção. Esta é a métrica utilizada no ROAM [8] e por Lindstrom et al. [22].

3.3.3 Coerência Entre Frames

A posição e orientação da câmara entre uma *frame* e a seguinte sofrem, normalmente, alterações muito pequenas [8][23]. Como consequência, as triangulações que representam o terreno nessas *frame*s são muito próximas, diferindo apenas em alguns triângulos.

Os algoritmos podem explorar esta característica, assumindo como ponto de partida na construção de uma nova triangulação a triangulação anterior, evitando assim refazer toda a triangulação de início. A esta característica dá-se o nome de coerência entre *frames*.

A maioria dos algoritmos apresentados nesta dissertação tira partido desta técnica.

3.3.4 Continuidade Temporal

A utilização de níveis de detalhe contínuos na visualização de terrenos implica a remoção ou inserção de novos vértices na triangulação. Estas operações, que representam a transição de um nível de detalhe para outro, potenciam um problema conhecido como *vertex popping*, pois os vértices introduzidos/removidos na triangulação não são, normalmente, co-planares com os triângulos que nela existem. Sendo assim, a mudança do nível de detalhe é perceptível, originando uma descontinuidade temporal.

Quando a mudança no nível de detalhe é imperceptível diz-se que existe continuidade temporal na triangulação.

[26] Do anglo-saxónico, *Geometric Screen Distortion.*

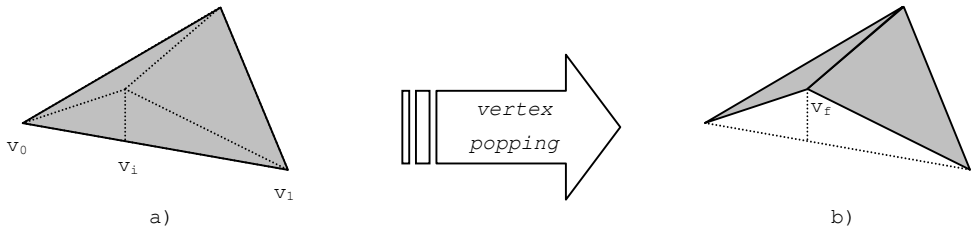


Figura 3-3: Exemplo de um *vertex popping*.

Considere-se a Figura 3-3. É criado um *vertex popping* se existir uma transição, em duas *frame*s consecutivas, da situação a) para a b).

Para solucionar este problema, podem ser aplicadas duas técnicas:

- *Alpha-Blending*.
- *Vertex Morphing* ou *Geomorph*.

Os algoritmos apresentados nesta dissertação apenas utilizam a técnica de *vertex morphing* dadas as vantagens enumeradas de seguida.

3.3.4.1 Alpha-Blending

Uma solução para o problema do *vertex popping* é uma técnica conhecida como *alpha-blending*. Esta técnica, ainda que não tendo sido concebida especificamente para estes casos, permite suavizar a transição de uma imagem para outra, não evitando contudo todos os artefactos visuais.

Na sua utilização mais comum, a técnica de *alpha-blending* permite criar diferentes efeitos numa imagem tais como transparência e *antialiasing*. A ideia por detrás desta técnica é simples: cada vértice a visualizar tem associado, para além da cor, um valor *alpha* que descreve o seu grau de opacidade (o valor *alpha* de 1 significa que o vértice é totalmente opaco e o valor 0 que é completamente transparente). A visualização de um vértice com um determinado grau de transparência tem de ser antecedida pelo desenho de todos os vértices que se encontram atrás e ocultados por ele. Isto porque a aplicação desta técnica utiliza os valores de cor dos *pixels* temporariamente a serem visualizados e combina-os com o valor do objecto transparente a ser desenhado, de acordo com o seu grau de transparência (valor *alpha*). Assim, a imagem final resulta do *blending* dos triângulos já existentes com os que vão ser desenhados, de acordo com a seguinte fórmula:

$$c_o = \alpha * c_s + (1-\alpha) * c_d$$

onde c_s é a cor do objecto transparente a ser desenhado, c_d é o valor da cor correntemente no *buffer*, e c_o é a cor resultante final.

Considere-se como exemplo a Figura 3-4, em que se pretende transitar de a), no instante de tempo 0, para d), no instante de tempo 1.

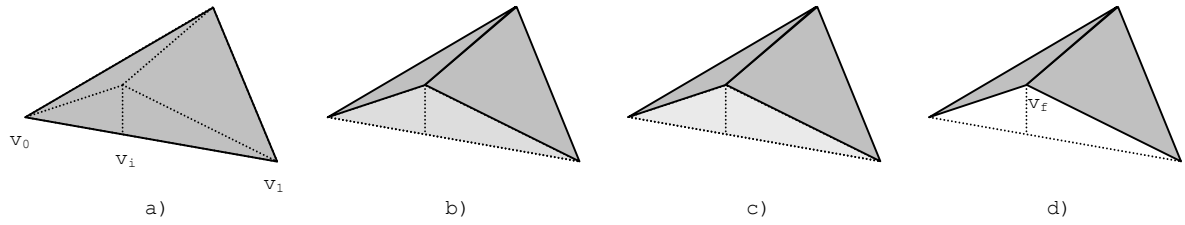


Figura 3-4: A técnica de *alpha-blending*.

Em cada *frame*, o cálculo da imagem a visualizar obtém-se da combinação das imagens inicial, i_i, (correspondente a a)) e final, i_f, (correspondente a d)), em proporções que variam linearmente, entre os 0% e 100% (ver Figura 3-5).

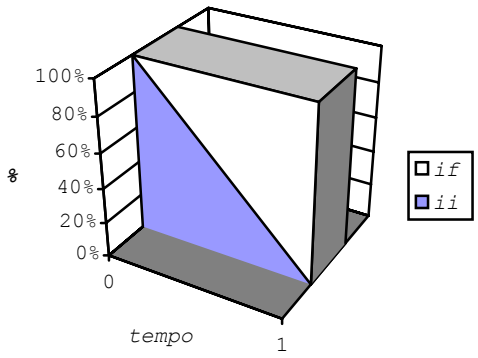


Figura 3-5: Gráfico de contribuição para a imagem final.

A contribuição de i_i para a imagem final inicia em 100% e decresce linearmente ao longo do tempo até 0%, enquanto que a contribuição de i_f cresce linearmente de 0% para 100%.

Formalmente, a imagem a utilizar, i_r, é dada por:

$$i_r(t) = (1-t)*i_i + t*i_f$$

A grande desvantagem desta técnica é necessitar do processamento dos dois níveis de detalhe em todas as *frame*s durante a sua aplicação. Tal aumenta o custo médio de computação, comprometendo a sua aplicação em sistemas de tempo real.

3.3.4.2 Vertex Morphing

A técnica mais utilizada para solucionar o problema do *vertex popping* tem o nome de *geomorphing* ou *vertex morphing* (ver Figura 3-6).

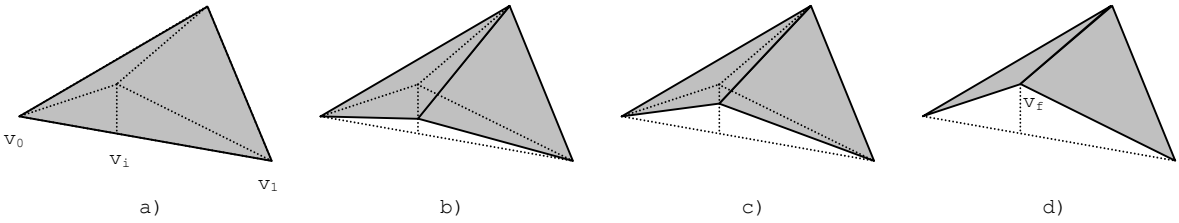


Figura 3-6: Utilização do *vertex morphing*.

Considere-se a Figura 3-6. Para evitar a transição brusca da situação a) para a d), é realizada uma animação suave nas *frame*s seguintes, com valores interpolados linearmente entre a posição inicial v_i do vértice em a) e a posição final v_f em d). Exemplo dessas posições interpoladas são as situações b) e c) da figura.

Formalmente, considere-se o intervalo de tempo $t \in [0,1]$. O vértice a animar, v, cuja posição inicial é dada por $v_i = (v_0 + v_1)/2$, é interpolado linearmente, até chegar a posição final v_f, no decurso do intervalo de tempo t, pela seguinte equação:

$$v_p(t) = (1-t) * v_i + t * v_f$$

Ao contrário da técnica de *alpha-blending*, a complexidade geométrica do modelo a ser visualizado não sofre alteração. Desta forma, o custo computacional de utilizar o *vertex-morphing* resume-se ao cálculo da interpolação [23].

3.3.5 Continuidade Espacial

Para além do problema da continuidade temporal, a inserção ou remoção de novos vértices pode trazer problemas de continuidade espacial. Isto é, estas operações podem criar falhas na triangulação tal como se apresenta na Figura 3-7.

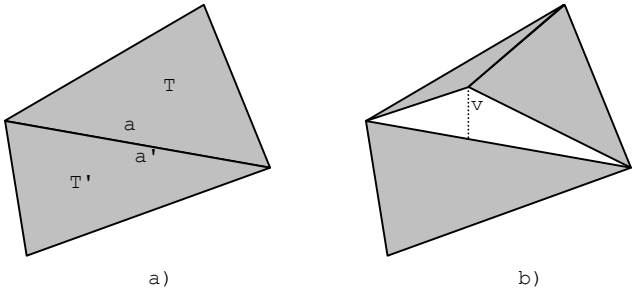


Figura 3-7: Exemplo de uma descontinuidade espacial[27].

Considere-se os triângulos vizinhos *T* e *T'* da Figura 3-7 a) e as suas arestas *a* e *a'*, respectivamente. As arestas *a* e *a'* coincidem no espaço e, do ponto de vista lógico, são a mesma aresta. Quando surge a necessidade de divisão do triângulo *T*, a aresta *a* é dividida em duas. Como *a'* se mantém inalterada, pois *T'* não é dividido, a triangulação resultante apresenta uma falha (Figura 3-7 b)).

A solução para este problema passa por forçar a divisão do triangulo *T'*, mantendo as novas arestas resultantes da divisão de *T* coincidentes com as arestas resultantes da divisão de *T'* (ver Figura 3-8 a) e b)). No caso da remoção de vértices teríamos a situação inversa.

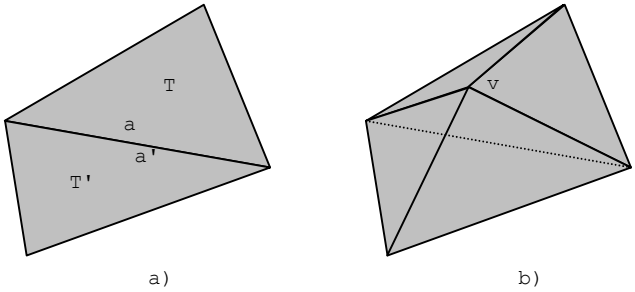


Figura 3-8: Solução para o problema de descontinuidade espacial.

Assim, para se evitar este problema é necessário garantir, à custa de divisões ou junções forçadas, que triângulos vizinhos partilhem sempre as suas arestas.

[27] Do anglo-saxónico, *T-junction crack*.

4 Algoritmos de Visualização de Terrenos em Tempo Real - (Estado da Arte)

Nesta secção apresentam-se as características gerais dos algoritmos baseados em multi-resolução contínua utilizados na visualização em tempo real de terrenos de grandes dimensões. Segue-se uma descrição dos algoritmos mais referenciados pela comunidade científica desta área, bem como algumas variantes mais significativas.

4.1 Características Gerais

Todos os algoritmos desenvolvidos nesta área são soluções para um mesmo problema: a navegação em tempo real em terrenos de grandes dimensões. Todos eles visam os mesmos objectivos que se enumeram de seguida.

4.1.1 Os Objectivos

Os objectivos principais deste tipo de algoritmos são:

- Redução considerável no número de polígonos visualizados através de uma geração dinâmica (em tempo real) dos níveis de detalhe utilizados, sem que esta tenha um impacto significativo no desempenho.
- As irregularidades localizadas no terreno, como vales e montanhas acidentados, não devem ter um efeito generalizado na complexidade do modelo.
- Variações suaves e contínuas entre os diferentes níveis de detalhe da superfície.
- Suporte a uma métrica dependente dos parâmetros da câmara, permitindo um fácil ajustamento entre a qualidade da imagem e o tempo de visualização.

4.1.2 Estrutura do Terreno

À excepção do algoritmo de *Progressive Meshes* de Hoppe [18][19][20], todos os algoritmos descritos nesta dissertação utilizam mapas regulares de alturas de dimensão $2^n+1 \times 2^n+1$, com $n \in$, como estrutura de dados principal para representação dos terrenos.

A utilização de mapas regulares de alturas de dimensão $2^n+1 \times 2^n+1$ permite a divisão recursiva do terreno em partes iguais, sendo as mais usuais a divisão em blocos e a divisão em triângulos rectângulos isósceles.

Esta divisão do terreno é a base do processo de simplificação utilizado pelos algoritmos e permite criar diferentes níveis de detalhe para a representação de regiões do terreno distintas.

Na divisão do terreno em blocos, os mapas regulares de alturas de dimensão $2^n+1 \times 2^n+1$ são divididos em 4 blocos[28] de igual dimensão ($2^{n-1}+1 \times 2^{n-1}+1$) que podem, por sua vez, ser divididos novamente em 4 sub-blocos de igual dimensão, num processo recursivo até à dimensão mínima de 3×3.

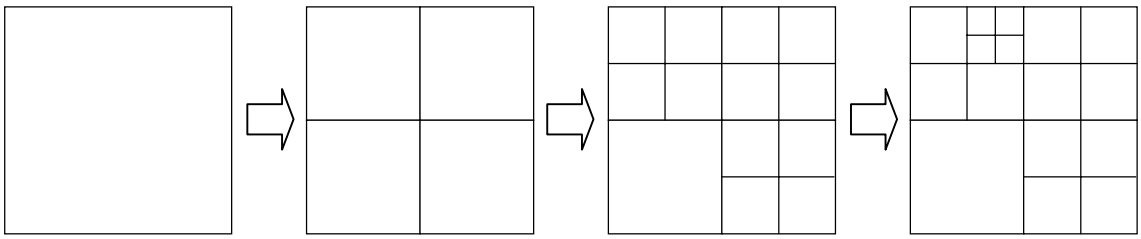


Figura 4-1: Exemplo da divisão sucessiva em blocos de um terreno de dimensão $2^n+1 \times 2^n+1$.

A estrutura de dados que suporta este processo de simplificação é uma árvore quaternária designada por *quadtree*.

Uma *quadtree* [21][30][31] é uma estrutura hierárquica na qual cada nodo tem 4 nodos filho distintos. A cada nodo da *quadtree* corresponde uma região do terreno e a cada nodo filho corresponde um quadrante distinto dessa região. À raiz da árvore corresponde o terreno completo (ver Figura 4-2).

[28] Também designados por quadrantes.

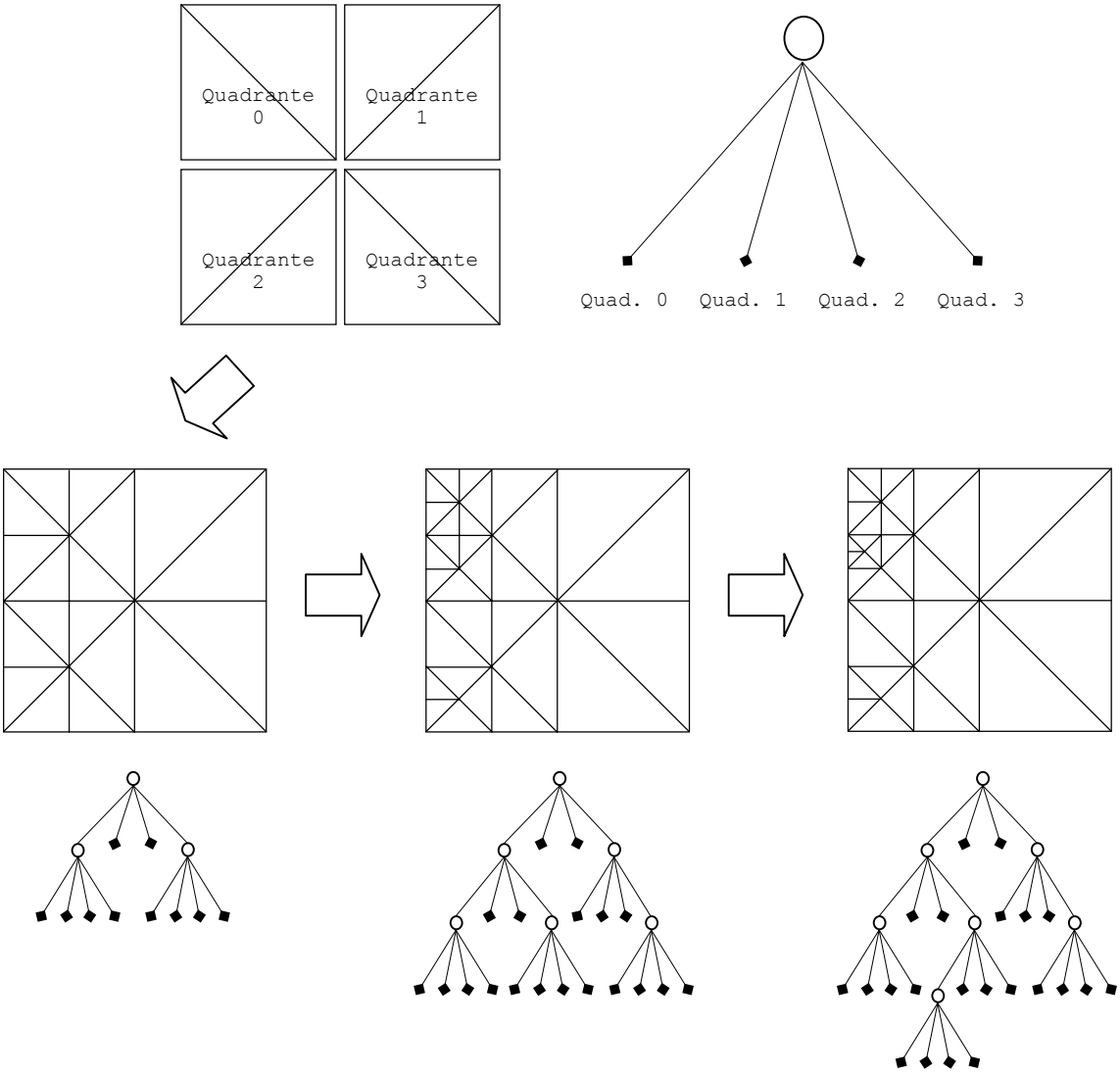


Figura 4-2: Representação de uma *quadtree*.

Esta estrutura é utilizado nos algoritmos de Lindstrom et al. [22], Ulrich [33], Röttger et al. [29], Castle et al. [3], Boer [2] e Chen et al [4].

Para além da divisão em blocos, um mapa regular de alturas de dimensão $2^n+1 \times 2^n+1$ possibilita a sucessiva divisão do terreno em triângulos rectângulos isósceles (ver Figura 4-3).

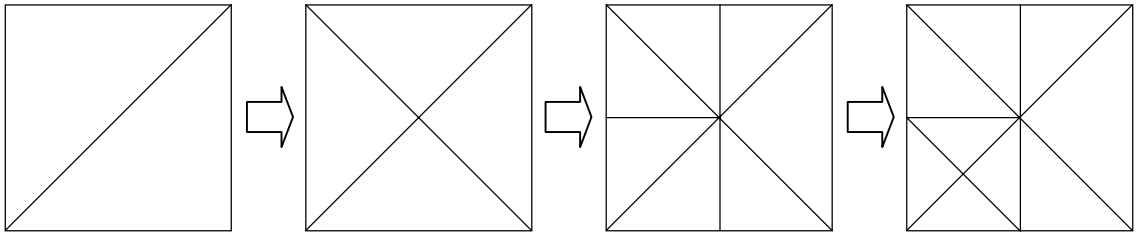


Figura 4-3: Divisão sucessiva em triângulos de um terreno de dimensão 2^n+1 x 2^n+1.

Esta divisão é suportada por uma estrutura de dados designada por árvore binária de triângulos (ABT). Uma ABT é uma árvore binária cujos nodos representam os triângulos da triangulação. Sendo triângulos, são necessárias duas ABTs para representar um terreno na totalidade (ver Figura 4-4).

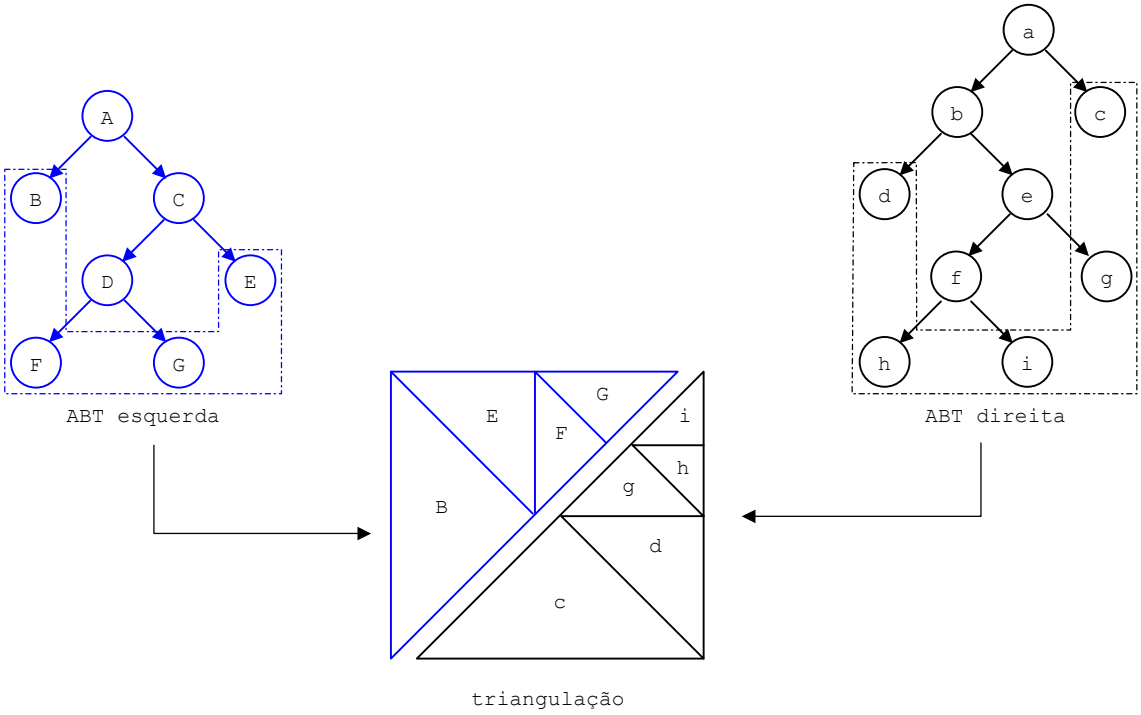


Figura 4-4: Árvore binária de triângulos.

Esta estrutura é utilizado nos algoritmos ROAM [8] e em todas as suas variantes apresentadas nesta dissertação [24][27][32].

4.2 O Algoritmo de Lindstrom et al.

Lindstrom et al. apresentam em [22] um algoritmo para visualização de terrenos em tempo real baseado na divisão recursiva do terreno em blocos. Esta divisão é representada por uma *quadtree*.

O algoritmo utiliza um processo de simplificação, executado em todas as *frames*, efectuado em dois passos consecutivos:

- Simplificação por blocos, a fim de determinar qual o nível de detalhe (bloco), isto é, qual o grau de simplificação, a utilizar para cada região do terreno.
- Simplificação por vértices, realizada ao nível do bloco, cujo objectivo é remover os vértices de menor importância, reduzindo assim o número de triângulos utilizados para representar a região do terreno correspondente ao bloco.

4.2.1 Simplificação por vértices

A simplificação por vértices é realizada independentemente para todo e qualquer bloco seleccionado na primeira fase.

Cada bloco é da forma *3x3*, sendo composto por 9 vértices conforme se apresenta na Figura 4-5. Na simplificação por vértices estes vértices são avaliados a fim de se determinar se devem ou não fazer parte da triangulação final do bloco em questão.

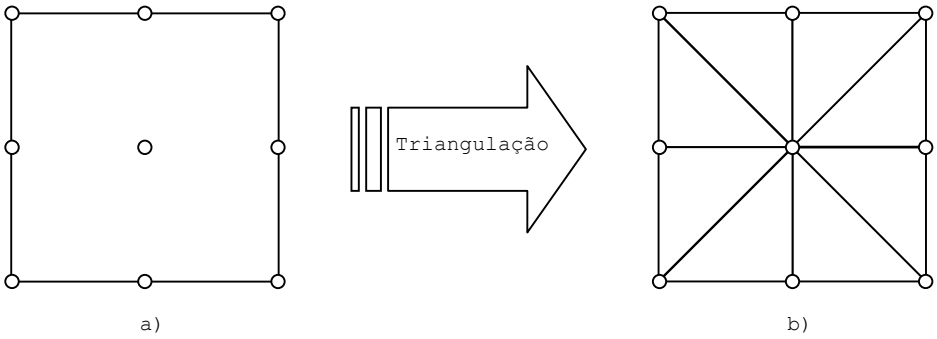


Figura 4-5: a) Bloco *3x3*. b) Exemplo da triangulação do bloco utilizando os seus 9 vértices.

Conceptualmente, cada uma destas avaliações representa uma tentativa de fusão de um par triângulo/co-triângulo num único triângulo maior (ver Figura 4-6) com base numa métrica de erro. Desta forma, o processo de simplificação por vértices reduz o número de triângulos a utilizar. Os triângulos resultantes da fusão formam, por sua vez, com outros triângulos de igual dimensão novos pares triângulo/co-triângulo e são considerados para simplificação de uma forma recursiva. Com este processo obtêm-se sucessivamente triângulos de maior dimensão e, consequentemente, uma triangulação mais simplificada e de menor resolução.

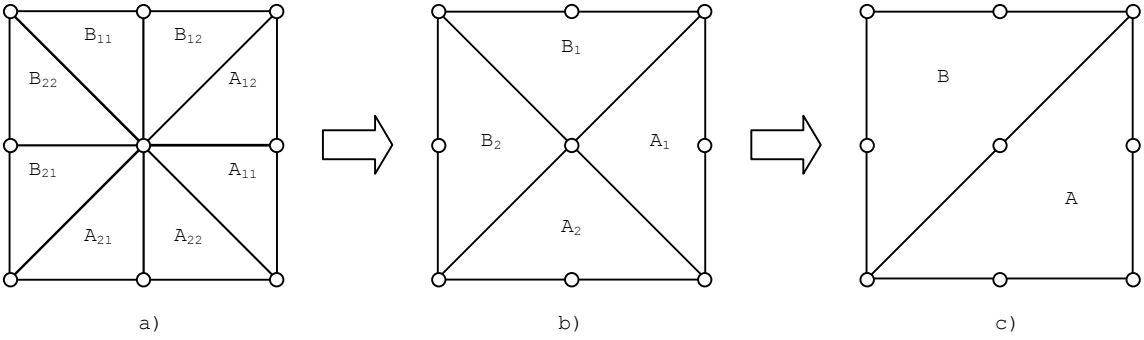


Figura 4-6: Pares triângulo/co-triângulo.

Considere-se a Figura 4-6. Os pares triângulo/co-triângulo em a) podem ser fundidos em novos triângulos resultantes em b). Os novos pares de b) podem, por sua vez, ser fundidos nos triângulos representados em c).

4.2.2 A Métrica

O critério de decisão para realizar a fusão dos triângulos é baseado no erro geométrico em *pixels*, isto é, no número de *pixels* utilizados pela projecção no ecrã do segmento de recta correspondente ao erro geométrico num ponto do terreno.

Considere-se a Figura 4-7.

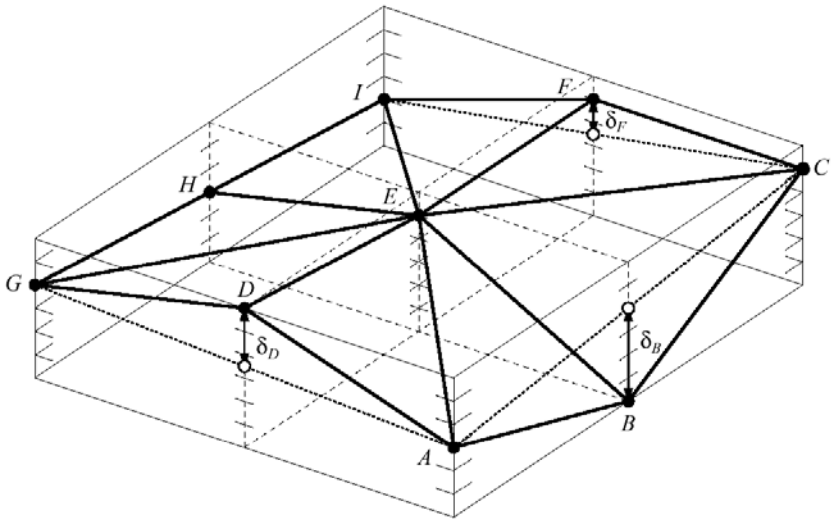


Figura 4-7: Representação geométrica do valor δ.

A fusão do par triângulo/co-triângulo (ΔABE, ΔBCE), cujo resultado seria o triângulo ΔACE, depende da distância vertical máxima, medida no espaço tridimensional, entre o triângulo resultante (ΔACE) e os triângulos que constituem o par (ΔABE e ΔBCE). A esta distância, que

é calculada no vértice a remover (neste caso, no vértice B), chama-se *valor delta*, δ, do vértice. O seu valor é dado pela seguinte equação:

$$\delta_B = \left| B_y - \frac{A_y + C_y}{2} \right|$$

Lindstrom et al. designaram por *segmento delta* de um vértice, o segmento de recta correspondente ao seu *valor delta*. No exemplo, o *segmento delta* de B é o segmento de recta definido por B e o ponto médio de AC.

Projectando o *segmento delta* no plano de projecção, isto é, projectando o segmento para o espaço ecrã, é possível determinar o máximo erro geométrico perceptível (em *pixels*) entre o par triângulo/co-triângulo e o triângulo resultante da sua fusão.

Se esse erro for menor que um determinado limite, τ, o par é alvo de uma fusão.

Este processo é aplicado recursivamente até que a triangulação do bloco não seja passível de sofrer mais junções.

A projecção do *segmento delta* pode ser calculado pela equação seguinte. (ver [22] para a derivação desta fórmula):

$$\delta_{proj} = \frac{d * \lambda * \delta * \sqrt{(e_x - v_x)^2 + (e_y - v_y)^2}}{(e_x - v_x)^2 + (e_y - v_y)^2 + (e_z - v_z)^2}$$

Nesta equação, (e_x, e_y, e_z) representam as coordenadas da posição da câmara, d a distância entre a câmara e o plano de projecção, λ o número de *pixels* por cada unidade (em coordenadas globais) no sistema de coordenadas x-y do ecrã e (v_x, v_y, v_z) as coordenadas do ponto médio do *segmento delta*.

Como foi mencionado anteriormente, um triângulo apenas forma um par com um outro do mesmo nível de subdivisão e só pares triângulo/co-triângulo podem ser simplificados. Para garantir estas condições, os vértices do bloco devem ser avaliados por uma determinada ordem durante o processo de simplificação por vértice.

Considere-se a 1ª situação da Figura 4-8. Na 1ª fase os pares triângulo/co-triângulo ((a_e, a_d), (b_e, b_d), (c_e, c_d) e (d_e, d_d)) sofrem uma fusão, isto é, os vértices apresentados a cinzento são avaliados e removidos, resultando na 2ª fase. Nesta fase, os novos pares triângulo/co-triângulo resultantes da fase anterior ((c,d) e (a,b)) são, também eles, alvos de uma fusão por avaliação do vértice a cinzento. Se este for removido passa-se à 3ª fase na qual não existem mais pares e, consequente, dá-se por terminada a avaliação de vértices para o bloco.

Considere-se agora a 2ª situação. Como a avaliação na 1ª fase dos vértices apresentados a cinzento não determinou a sua eliminação, apenas é possível a transição para uma 2ª fase. Não é possível transitar para a fase seguinte pois os vértices não removidos terão de fazer parte da triangulação final.

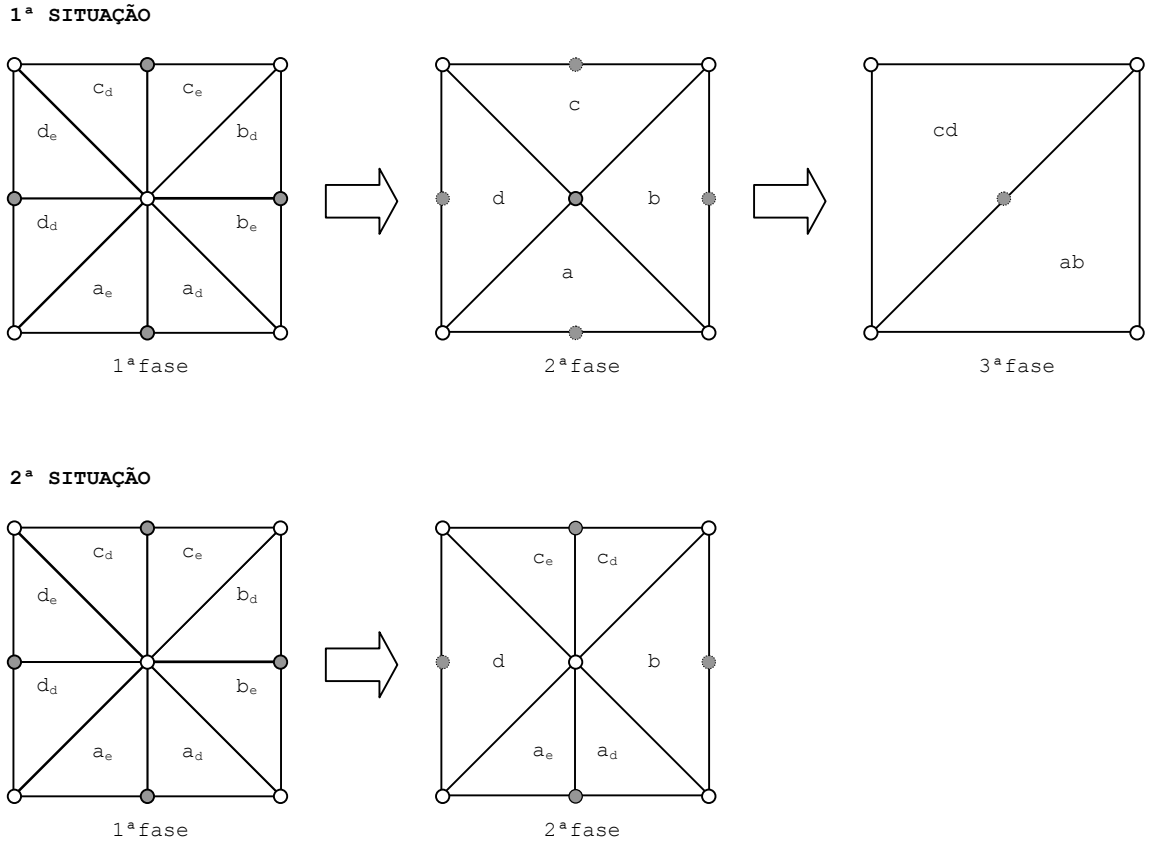


Figura 4-8: Ordem da remoção dos vértices.

Para garantir esta ordem de avaliação é criada uma árvore de dependências de vértices, que abrange o terreno na totalidade. O processo de simplificação por vértices é um processo *bottom-up*. Inicia nas folhas da árvore de dependências e, caso ocorram remoções de vértices, prossegue para o nível superior seguinte, até à raiz da árvore.

Cada vértice tem dois pais, um para cada par triângulo/co-triângulo do qual faz parte, e quatro filhos. Esta relação pai/filho representa uma dependência no processo de remoção. Se v_a é filho de v_b então v_a só pode ser removido após a remoção de v_b. Assim, cada vértice depende de 4 vértices e 2 dependem dele. Excepção feita para os vértices nas fronteiras do terreno, que apenas

dependem de 2 vértices e 1 depende dele[29], e para os vértices nas folhas da árvore (designados em [22] por *lowest level vértices*) que não dependem de nenhum vértice.

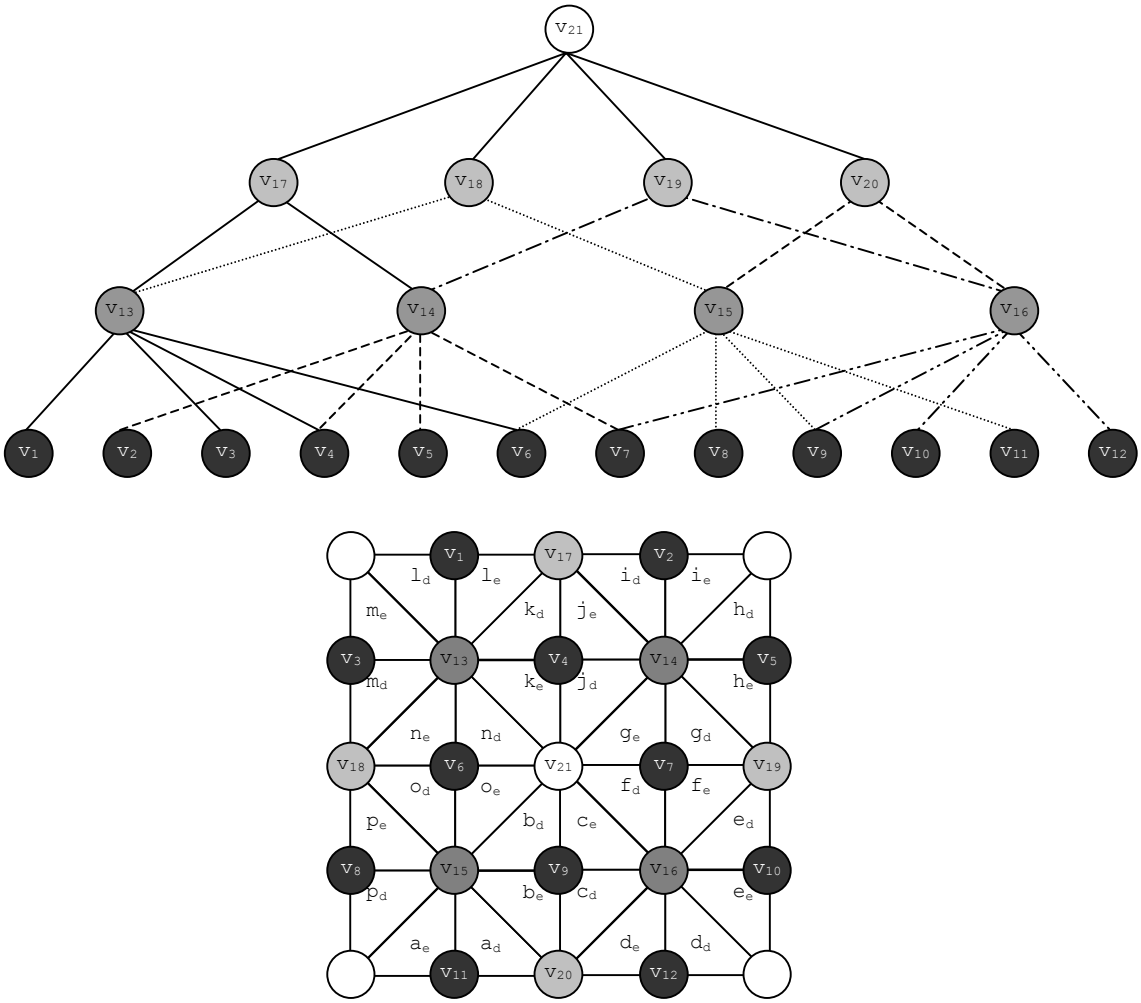


Figura 4-9: Árvore de dependências de vértices.

Considere-se a Figura 4-9 como exemplo de um terreno $5x5$. O vértice v_4 (e todos os apresentados a cinza escuro) é um vértice folha. Desta forma, v_4 não depende de nenhum vértice e pode ser avaliado imediatamente pelo processo de remoção. Por sua vez, o vértice v_{14} depende de v_4, bem como de v_2, v_5 e v_7 (os restantes filhos), pelo que só pode ser considerado para remoção se todos os seus filhos tiverem sido removidos em iterações anteriores. O processo de avaliação prossegue desta forma para os níveis superiores.

[29] Os vértices fronteira de bloco ignoram as relações de dependência com os vértices dos blocos adjacentes no sentido de tornar o processo completamente independente dos blocos contíguos.

Para manter a estrutura da árvore, os vértices são armazenados numa estrutura independente à *quadtree* que contém apenas referências para os vértices. Evita-se assim a repetição da informação de vértice comuns a mais do que um bloco e a inconsistência que daí poderia advir. Cada vértice tem o seguinte conjunto de atributos:

- *Presente*[30]: indica se o vértice faz parte da triangulação corrente. Um vértice está *presente* se pelo menos um dos seus filhos também está ou se está *activo*.
- *Activo*[31]: indica se o segmento delta projectado associado a este vértice excede o limite τ. Se um vértice está *activo* então também está *presente*.
- *Bloqueado*[32]: utilizado para forçar o valor de *presente* a `verdadeiro` ou `falso`. Alguns vértices comuns que não pertençam à triangulação final de ambos os blocos adjacentes devem ser eliminados/inseridos da triangulação da *frame* actual, a fim de evitar descontinuidades espaciais entre blocos adjacentes de diferentes níveis.
- *Dependentes*[33]: conjunto de 4 atributos que indicam se cada um dos filhos está ou não *presente*. Um vértice apenas pode ser removido da triangulação quando todos os seus dependentes não estão *presentes*.

Quando o segmento delta projectado associado a um vértice excede o limite τ, esse vértice é *activado* e *presente*. Consequentemente, todos os seus ascendentes na árvore de dependências são notificados desta alteração. Nessa notificação, o atributo *presente* de todos os vértices ascendentes é activado e, consequentemente, estes são incluídos na triangulação final.

Quando o segmento delta projectado for inferior a τ, o vértice pode ser removido da triangulação final, se não tiver filhos *presentes*, sem que tal crie descontinuidades espaciais ao nível do bloco. De seguida, todos os ascendentes são notificados desta alteração, sendo actualizado o atributo *dependentes* referente ao vértice removido. No caso de, ao ascendente, não restar nenhum *dependente* activado este pode ser também removido da triangulação, se não estiver ele próprio *activo*.

---

[30] Do anglo-saxónico, *enabled*.

[31] Do anglo-saxónico, *activated*.

[32] Do anglo-saxónico, *locked*.

[33] Do anglo-saxónico, *dependency*.

4.2.3 Simplificação por blocos

Se se considerar todos os blocos de maior resolução de um terreno e a ele aplicarmos o processo de simplificação por vértices anteriormente apresentado, este seria excessivamente exigente em termos computacionais para uma visualização em tempo real [22]. No entanto, se se determinar que grupos de vértices podem ser eliminados em bloco, reduz-se em várias ordens de grandeza a complexidade da triangulação com custo computacional reduzido [22].

Se o máximo segmento delta projectado, δ_{max}, para todos os vértices folha de um bloco for inferior a τ, todos estes vértices podem ser imediatamente removidos sem ser necessário efectuar a simplificação por vértice: o bloco pode ser substituído por um outro de menor resolução.

4.2.3.1 Intervalo de Confiança

Lindstrom et al. expandiram esta ideia por forma a obter um algoritmo mais eficiente, calculando um intervalo de confiança para valores de δ dentro do qual é necessário avaliar o vértice. Dado τ, os parâmetros da câmara e a *bounding box* do bloco, é possível calcular o menor valor delta, δ_l, que, quando projectado, nunca excede τ, bem como o maior valor delta, δ_h, a partir do qual a sua projecção excede τ. Ao intervalo $]\delta_l, \delta_h]$ dá-se o nome de I_u [22].

Os valores de δ_l e δ_h podem ser calculados pelas seguintes equações (ver [22] para a derivação destas fórmulas):

$$\delta_l = \frac{\tau}{d * \lambda * f_{max}}$$

$$\delta_h = \begin{cases} 0 & , \tau = 0 \\ \dfrac{\tau}{d * \lambda * f_{min}} & , \tau > 0 \wedge f_{min} > 0 \\ \infty & , caso\ contrário \end{cases}$$

Nas quais f_{max} e f_{min} representam, respectivamente, o máximo e mínimo da função f que é definida por:

$$f(r, h) = \frac{r}{r^2 + h^2}$$

restringida pela condição $r^2 + h^2 \geq d^2$, sendo $r = \sqrt{(e_x - v_x)^2 + (e_y - v_y)^2}$ e $h = e_z - v_z$.

Desta forma f_{max} pode ser encontrado quando $r = h$ e f_{min} quando $r = r_{min}$ ou $r = r_{max}$, sendo r_{min} e r_{max}, respectivamente, o mínimo e o máximo de r [22].

4.2.4 O Algoritmo

O processo de simplificação começa pela computação do intervalo I_u para todos os blocos activos[34], que é posteriormente comparado com δ_{max}. I_u é inicialmente $]0,\infty]$ para todos os blocos.

Em cada *frame*, o algoritmo efectua uma travessia à *quadtree* para encontrar blocos activos. Em cada bloco activo encontrado, o valor I_u referente à *frame* anterior é guardado e o novo valor de I_u é calculado de acordo com os parâmetros da câmara actual. De seguida, o valor de δ_{max} do bloco é comparado com os novos limites do intervalo I_u.

Se:

- $\delta_{max} > \delta_h$, o bloco não tem resolução suficiente de acordo com os parâmetros da câmara actual e da precisão escolhida, logo deve ser substituído pelos seus blocos filhos de maior resolução. O processo é repetido para os novos blocos.
- $\delta_{max} <= \delta_l$, o bloco tem uma resolução excessiva para os parâmetros da câmara actual e para a precisão escolhida, logo pode ser substituído pelo bloco pai de menor resolução. Neste caso, a substituição acontece apenas se todos os δ_{max} dos filhos do bloco pai forem inferiores a δ_l. O processo é repetido para o novo bloco.
- $\delta_{max} > \delta_l$ $\delta_{max} <= \delta_h$, o bloco encontra-se no intervalo de incerteza I_u, pelo que o bloco deve ser mantido para que lhe seja aplicada a simplificação por vértices.

  No processo de simplificação por vértices é calculado o segmento delta projectado para todos os vértices com valores delta pertencentes ao intervalo de incerteza I_u. O vértice é activado se o valor resultante desse cálculo é menor ou igual a τ, e é desactivado caso contrário.

  Vértices com valores delta inferior a δ_l devem ser desactivados, salvo se o seu valor delta é inferior a δ_l^{f-1}, sendo f a *frame* actual. Neste caso, os vértices já haviam sido desactivados em *frames* anteriores.

[34] Um bloco activo é um bloco seleccionado para representar uma a região de terreno. Inicialmente, os blocos activos são todas as folhas da *quadtree*, isto é, todos os blocos de maior resolução. A união de todos os blocos activos cobre o terreno na integra.

De igual forma, vértices com valores delta superiores a δ_h devem ser activados, salvo se o seu valor delta é superior a δ_h^{f-1}. Neste caso, os vértices já haviam sido activados em *frames* anteriores.

De notar que na primeira iteração o processo de simplificação por blocos apresentado é *bottom-up*, isto é, o processo parte dos blocos de maior detalhe e tenta substituí-los por níveis de detalhe mais simplificados.

No entanto, nas iterações seguintes a utilização de blocos activos permite tirar partido da coerência entre *frames* partindo-se dos blocos activos da *frame* anterior para a construção da *quadtree* seguinte.

4.2.5 A Construção e Visualização da Triangulação

Depois do processo de simplificação, os vértices *presentes* dos blocos activos são utilizados para criar a triangulação final durante uma nova travessia da *quadtree*. Em cada bloco activo os vértices *presentes* são encontrados por uma travessia pré-ordem das árvores de vértices e incluídos numa única tira de triângulos criada para cada bloco.

Cada bloco pode ser visto como um grupo de 4 triângulos (ver Figura 4-10) desenhados no sentido anti-horário, de q_0 a q_3.

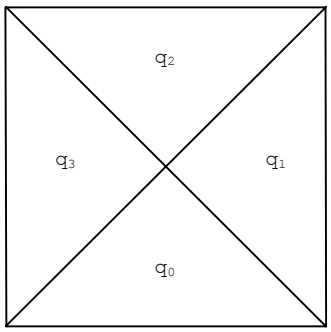


Figura 4-10: Os 4 quadrantes de um bloco.

Ainda que cada bloco tenha apenas 9 vértices explícitos, a sua triangulação poderá incluir mais vértices resultantes da propagação da activação de vértices de blocos adjacentes pela árvore de dependência de vértices.

Considere a Figura 4-11. Apesar do vértice assinalado em b) não fazer parte do bloco apresentado em a), a sua activação, e a propagação que daí resulta pela árvore de dependências, implica que a triangulação final do bloco seja a apresentada em c). A activação do vértice assinalado em b) resulta do processo de simplificação de um bloco adjacente de nível inferior (não apresentado na figura).

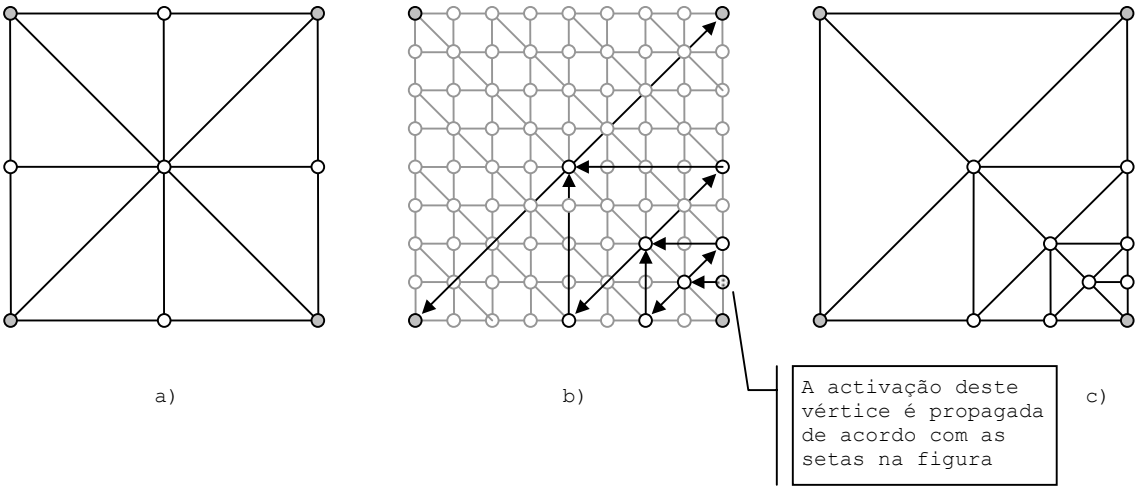


Figura 4-11: Construção e visualização da triangulação.

Durante esta fase os vértices são processados e incluídos na tira de triângulos pela ordem apresentada na Figura 4-12. Esta ordem garante a construção correcta da triangulação.

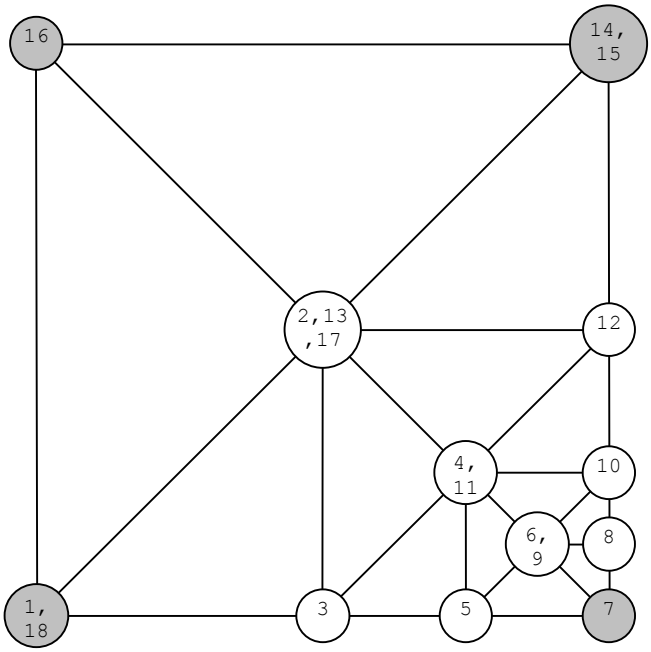


Figura 4-12: Ordem de inclusão dos vértices na tira de triângulos.

Conforme se constata, por vezes é necessário efectuar algumas operações de troca[35] de vértices para a correcta constituição da tira de triângulos. Note-se que é utilizada apenas uma tira para todo o bloco.

[35] Do anglo-saxónico, *swap*.

4.3 Continuous LOD Terrain Meshing Using Adaptive *Quadtree*s

Ulrich [33] implementou um algoritmo baseado no algoritmo de Lindstrom et al. [22] que lida com o problema criado pelo armazenamento em memória da totalidade do terreno.

Ulrich assenta o seu algoritmo numa *quadtree* adaptativa, que permite armazenar em disco a informação sobre terreno. Ao contrário do mapa regular de alturas utilizado por Lindstrom et al., a utilização duma *quadtree* adaptativa permite armazenar em disco informação de diferentes regiões do terreno com resoluções distintas. Lindstrom et al. constroem uma simplificação do terreno representada por uma *quadtree* a partir da informação armazenada em disco num mapa regular de alturas. Ulrich utiliza em memória uma *quadtree* adaptativa que lhe permite simplificar e carregar do disco apenas a informação necessária sobre o terreno.

Uma *quadtree* adaptativa é, tal como uma *quadtree*, uma árvore quaternária que permite dividir recursivamente o terreno em 4 quadrantes distintos.

No processo de construção da *quadtree* adaptativa inicia-se de um primeiro nível de amostragem, de menor resolução, no qual toda a extensão do terreno é amostrada com apenas alguns pontos (dentre eles, os pontos que correspondem aos cantos do terreno). A Figura 4-13 a) mostra o exemplo de um primeiro nível de amostragem num terreno de 17 x 17 amostras. Os pontos desenhados a preto representam as amostras incluídas nesse nível de amostragem. Todos os outros são ignorados. À medida que se aumenta a resolução do processo de amostragem (e se aumenta a profundidade da *quadtree* adaptativa) são incluídas apenas as regiões do terreno para as quais se pretende maior detalhe durante a visualização. A Figura 4-13 b) e c) são exemplos de regiões que foram amostradas com uma maior resolução.

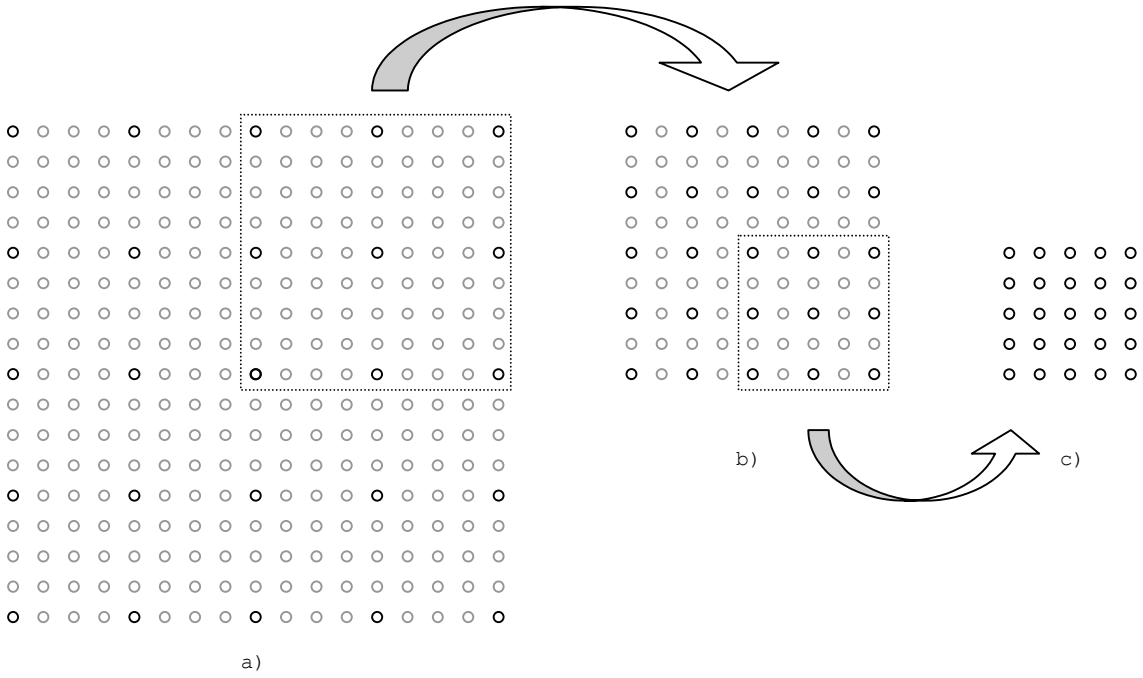


Figura 4-13: Uma *quadtree* adaptativa para armazenar informação sobre o terreno.

Estas diferentes resoluções para várias regiões do terreno permitem, durante a execução do algoritmo, carregar mais facilmente para memória apenas a informação relevante num dado instante de tempo [33]. Informação detalhada sobre uma região que é visualizada a grande distância ou mesmo que não é visualizada de todo, não necessita de ocupar espaço de memória.

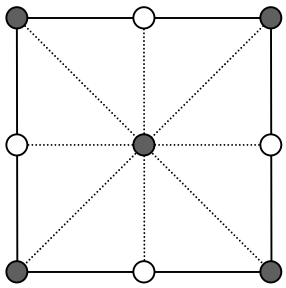


Figura 4-14: Triangulação de um bloco de 3 x 3 vértices.

No processo de construção da *quadtree* em memória, o algoritmo utiliza uma abordagem *top-down* que inicia carregando para memória um bloco 3×3 que representa o terreno total (ver Figura 4-14). Os quadrantes deste bloco são recursivamente avaliados e divididos até se atingir o nível de resolução adequado à métrica ou o nível máximo amostrado para a região em causa.

Num bloco os vértices dos cantos e do centro estão sempre *presentes* e, consequentemente, fazem parte da triangulação. Os restantes 4 vértices necessitam de ser avaliados de acordo com a métrica definida.

Quando um quadrante é dividido novos vértices são introduzidos pela criação de novos blocos `3 x 3` sobre os quais se aplica novamente o processo descrito. Os vértices central e dos cantos são assinalados como *presentes* em todos os novos quadrantes. Para manter a consistência da triangulação é necessário garantir que os vértices do bloco pai, comuns ao novo quadrante, são também assinalados como *presentes*. Estes são os vértices das arestas do bloco pai que correspondem a cantos no novo bloco (ver Figura 4-15).

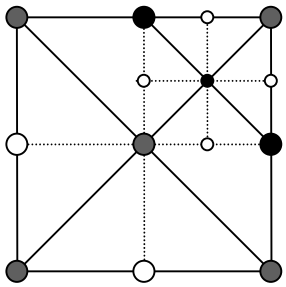


Figura 4-15: Criação de um novo quadrante. Os vértices pretos são assinalados como *presentes* e os dois vértices comuns ao bloco pai devem ser também assinalados como *presentes* nesse bloco.

Cada bloco partilha com cada um dos blocos vizinhos de igual nível um vértice que necessita de ser avaliado. Sempre que um desses vértices partilhado é assinalado como *presente* num bloco, também tem de o ser no bloco vizinho para evitar descontinuidades espaciais na triangulação (ver Figura 4-16).

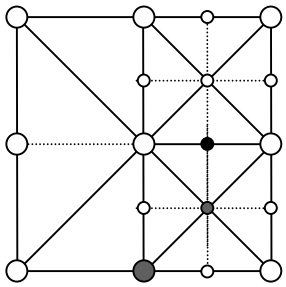


Figura 4-16: Partilha de vértices entre blocos adjacentes. Se o vértice preto estiver *presente* tem de o estar em ambos os blocos dos quais faz parte.

Depois do processo de construção descrito, é construída e visualizada a triangulação. Numa travessia pela *quadtree* criada durante o processo de divisão, são desenhados leques de

triângulos para cada bloco encontrado. Estes blocos encontram-se nas folhas da *quadtree* e os leques desenhados incluem apenas os vértices assinalados como *presentes* nesse bloco.

Posteriormente, as restantes partes do terreno representados por nodos da *quadtree* parcialmente divididos são também desenhados.

4.3.1 Coerência entre Frames

Para tirar partido da coerência entre *frames* parte-se da triangulação da *frame* anterior para construir uma nova. Desta forma, é necessário, para além do processo de divisão descrito, um processo de simplificação que remova a sinalização de *presente* dos vértices.

Como foi mencionado anteriormente, a sinalização de *presente* de um vértice é normalmente propagada para os blocos ascendentes da *quadtree* e para blocos vizinhos, por forma a evitar descontinuidades espaciais. Assim, a remoção desta sinalização não pode interromper esta cadeia de dependência, pois causaria descontinuidades espaciais na visualização de alguns sub-blocos.

Cada vértice que necessita de ser avaliado poderá ter quatro sub-blocos adjacentes entre si que o utilizam como vértice canto (ver Figura 4-17). A existência de um qualquer destes sub-blocos (preenchidos a cinza na Figura 4-17) implica que este vértice esteja *presente*. Assim, esse vértice apenas pode ser *desactivado* se nenhum destes sub-blocos existirem.

De igual forma, um bloco apenas pode ser *desactivado* se nenhum dos vértice das arestas estão *presentes* e se nenhum dos sub-blocos estiver *presente*.

Considere-se a Figura 4-17. O vértice assinalado a preto necessita de ser avaliado nos blocos `A` e `B` dos quais faz parte. Por outro lado, este vértice é um dos vértices canto dos quatro sub-blocos adjacentes entre si (preenchidos a cinza). Por este motivo, o vértice assinalado é automaticamente *presente* em todos os blocos e apenas pode ser *desactivado* se nenhum dos sub-blocos preenchidos a cinza existir. Caso contrário seriam criadas descontinuidades espaciais.

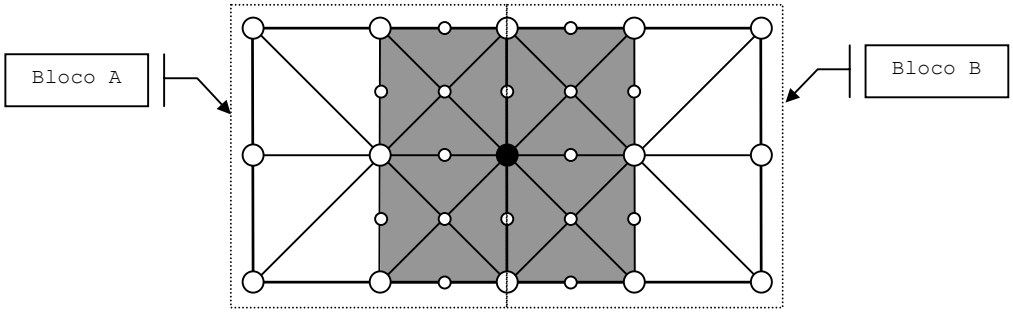


Figura 4-17: Tentativa de *desactivação* de um vértice com sub-blocos adjacentes.

4.3.2 A Métrica

Ao contrário de Lindstrom et al., Ulrich não utiliza a métrica do erro geométrico em *pixels* para determinar quando um vértice deve estar *presente*. A métrica definida considera apenas o erro geométrico, criado pela mudança de nível de detalhe, e a distância do vértice à câmara.

A métrica utilizada é computacionalmente mais rápida do que a de Lindstrom et al., embora seja menos precisa pois o erro geométrico utilizado não é medido em *pixels* [33]. A fórmula seguinte permite determinar quando um vértice deve estar *presente*:

$$Presente = \left(\frac{distância\ à\ câmara}{erro\ geométrico} < limite \right)$$

Ulrich utiliza a norma L1 para o cálculo da distância à câmara. O limite permite ajustar a qualidade da triangulação obtida e, consequentemente, da imagem final.

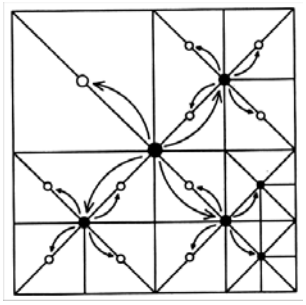
No processo de decisão de quando um bloco deve estar *presente* é calculado o máximo erro geométrico e a *bounding box* para o quadrante em questão. O teste efectuado permite determinar se os vértices contidos no quadrante estariam ou não *presentes*. Em caso afirmativo, o bloco deve ser dividido.

4.4 Real-Time Generation of Continuous Level of Detail for *Height-field*s

Röttger et al. apresenta em [29] uma versão simplificada do algoritmo de Lindstrom et al.. O algoritmo de Röttger et al. opera *top-down* na *quadtree* e utiliza apenas simplificação por blocos por forma a melhorar o desempenho do mesmo.

A estrutura de dados utilizada pelo algoritmo é uma *quadtree* implementada por uma matriz booleana de dimensão igual à do terreno, na qual cada entrada corresponde a uma amostra do terreno.

Uma entrada activa (valor `1` na matriz) corresponde à existência de um bloco na triangulação cujo vértice central é representado por essa posição na matriz. A todos os blocos de nível imediatamente inferior a um bloco activo, é que não estejam eles também activos, corresponde uma entrada inactiva (valor `0` na matriz) (ver Figura 4-18). Todos os restantes elementos da matriz não necessitam de nenhum valor específico, pois apenas os blocos com entradas activas/inactivas são utilizados pelo algoritmo.



$$\begin{bmatrix} ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & 0 & ? & ? & ? & 1 & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & ? & ? & 1 & ? & ? & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & 1 & ? & ? & ? & 1 & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? \end{bmatrix}$$

Figura 4-18: Exemplo de uma triangulação e a matriz correspondente.

A criação e actualização da *quadtree* é um processo *top-down* recursivo que não tira partido da coerência entre *frames* consecutivas. Em cada iteração é calculada uma condição que, no caso de ser verdadeira, e se o máximo nível de detalhe não tiver sido atingido, força à criação de um novo nível na *quadtree*. No caso de ser falsa, a recursividade nesse ramo termina e o bloco em questão é removido da *quadtree* no caso de existir[36].

A criação de um novo nível na *quadtree* implica a activação da entrada da matriz correspondente ao vértice central do bloco criado. É também necessário desactivar as entradas correspondentes aos vértices centrais dos blocos filhos de nível inferior pois estes poderão ser consultados durante a fase de construção e visualização da triangulação.

A remoção de um nível resume-se à desactivação das entradas correspondentes aos vértices centrais dos bloco a remover e também dos seus filhos de nível inferior.

4.4.1 A Métrica

À semelhança de Ulrich [33], os critérios de decisão utilizados para o refinamento da triangulação baseiam-se nos seguintes factores:

[36] Por questões de desempenho, o algoritmo de Röttger et al. apenas inicializa a matriz na primeira *frame*. Por este motivo é necessário remover da matriz os blocos para os quais o resultado da condição testada seja falso.

- A resolução deve diminuir se a distância à câmara aumentar. Esta condição é garantida pela seguinte inequação:

$$\frac{l}{d} < c$$

sendo l a distância à câmara (Röttger et al. utiliza a norma L1 para o cálculo desta distância), d o comprimento da aresta do bloco e c uma constante que define um parâmetro de qualidade (ver Figura 4-19).

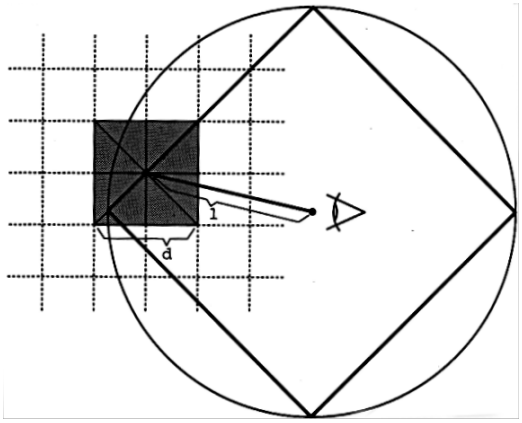


Figura 4-19: Resolução global mínima.

A constante c controla a resolução global mínima. À medida que c aumenta, o número total de vértices por *frame* cresce quadraticamente [29]. De notar que esta condição apenas necessita de ser avaliada uma vez para cada bloco, isto é, uma vez para cada 9 vértices.

- A resolução deve aumentar para regiões do terreno mais irregulares.

O objectivo deste critério é minimizar o erro geométrico em *pixels* e logo melhorar a qualidade da imagem. Quando se sobe um nível na *quadtree*, introduzem-se novos erros em exactamente 6 pontos: dois no vértice central do bloco anterior (um para cada diagonal que passa nesse vértice central, dado que num nível superior a região de terreno correspondente a esse bloco pode ser triangulada utilizando duas diagonais distintas, dependendo do quadrante do bloco no qual se encontra), $dh_{5/6}$, e um por cada ponto médio de cada aresta fronteira, $dh_{1..4}$ (ver Figura 4-20).

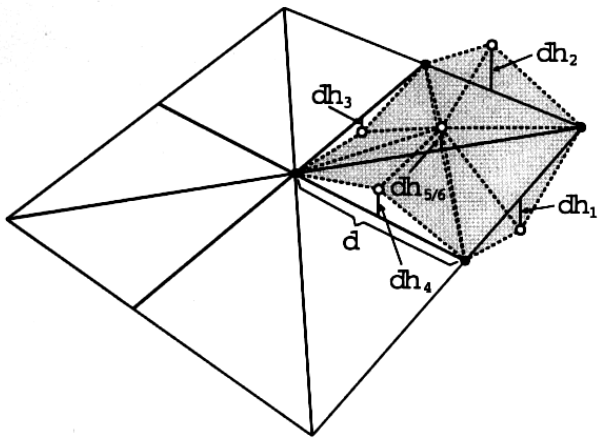


Figura 4-20: Erros introduzidos pela mudança no nível de detalhe.

O máximo desses erros pode ser utilizado para calcular um erro aproximado no espaço tridimensional, denominado d_2, de acordo com a seguinte fórmula:

$$d_2 = \frac{1}{d} * \max_{i=1..6} |dh_i|$$

Com estes dois critérios, define-se uma variável de decisão f nos termos seguintes:

$$f = \frac{1}{d * C * \max(c * d_2, 1)}$$

Se $f < 1$ deve utilizar-se um nível de detalhe de maior resolução para o bloco em questão.

A constante c representa a resolução global desejada e o seu valor condiciona directamente o número de triângulos a ser utilizados na *frame* corrente.

Para que a triangulação não contenha descontinuidades espaciais é necessário garantir que a diferença entre os níveis de detalhe nunca é superior a 1. Quando essa diferença é 1 (ver Figura 4-21 a)) é possível evitar a potencial descontinuidade pela remoção do vértice do meio da aresta comum aos blocos.

Se a diferença é superior a 1 (ver Figura 4-21 b)) não é possível evitar a existência de uma descontinuidade espacial. Neste caso, a remoção do vértice assinalado na figura impossibilita a construção e visualização correcta dos leques de triângulos.

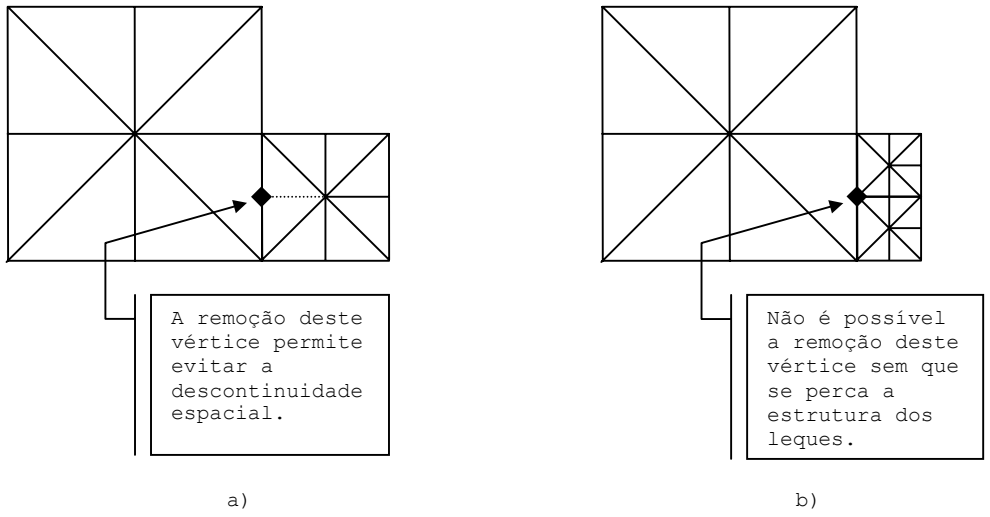


Figura 4-21: Descontinuidade espacial causada pela diferença de nível de blocos adjacentes.

Neste sentido, Röttger et al. redefine o cálculo de d_2 por forma a propagar os valores d_2 pelos sucessivos níveis da *quadtree*, começando pelos de maior resolução até à raiz da mesma.

Considere-se a constante $K = \frac{C}{2 * (C - 1)}$. O valor de d_2 para cada bloco é então dado pelo valor local de d_2, obtido pela fórmula apresentada, e K vezes o valor previamente calculado dos nodos adjacentes do nível imediatamente abaixo.

Este algoritmo para o cálculo de d_2 garante que a diferença dos níveis de detalhe de blocos adjacentes nunca é superior a 1 [29].

Uma descrição mais detalhada deste cálculo pode ser encontrada em [29].

4.4.2 Continuidade Espacial

Após a construção/actualização da *quadtree*, a visualização da triangulação é efectuada durante uma nova travessia à *quadtree*. Sempre que se atinge uma folha, que corresponde a um 0 na entrada da matriz, é desenhado um leque total de triângulos. A fim de se evitar descontinuidades espaciais entre blocos adjacentes de níveis (e logo resoluções) diferentes, este leque pode ser completo ou incompleto. O vértice do centro das arestas comuns a esses blocos não são utilizados na visualização (ver Figura 4-22). Isto apenas é possível porque o nível de resolução entre blocos adjacentes nunca difere mais do que 1.

Determinar se os blocos adjacentes são ou não do mesmo nível resume-se a uma consulta à entrada da matriz correspondente ao bloco vizinho. Caso seja `0`, o bloco adjacente é de um nível superior, logo o vértice não deve ser incluído na triangulação. Caso seja `1`, o bloco adjacente é do mesmo nível, logo o vértice deve ser incluído na triangulação.

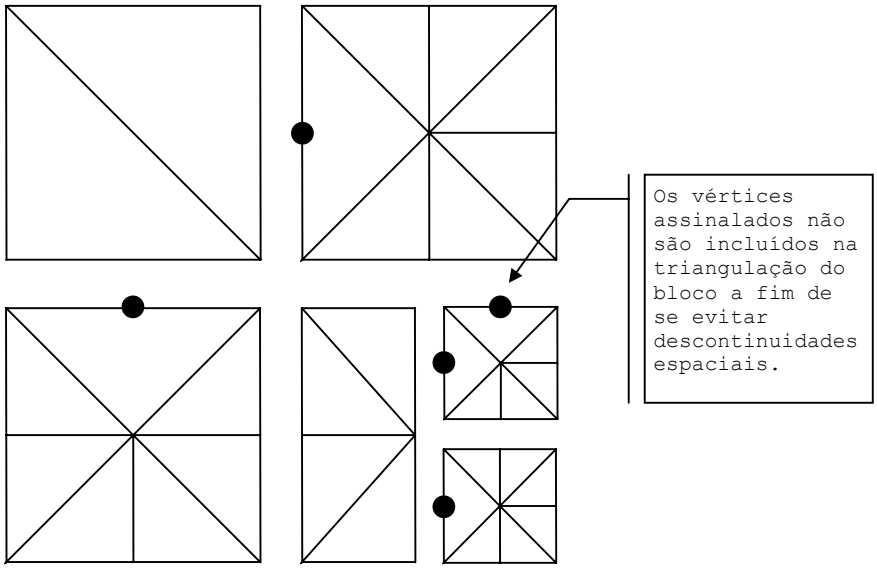


Figura 4-22: Eliminação de vértices para evitar a existência de descontinuidades espaciais.

4.5 ROAMing Terrain: Real-time Optimally Adapting Meshes

Real-time Optimally Adapting Meshes (ROAM) [8] foi apresentado por Duchaineau et al. e apresenta-se como um desenvolvimento ao algoritmo de Lindstrom et al..

Trata-se de um algoritmo que utiliza uma estratégia *top-down* no processo de construção/actualização da triangulação e que não divide o terreno em blocos quadrados. Tira partido da coerência entre *frames* e organiza os triângulos que compõe a triangulação por forma a facilitar as operações sobre eles.

4.5.1 A Árvore Binária de Triângulos

A árvore binária de triângulos (ABT) [8][24] é a estrutura de dados adoptada para representar a triangulação.

Como foi mencionado na secção *4.1.2 – Estrutura do Terreno*, uma ABT é uma árvore binária cujos nodos são triângulos rectângulos isósceles, pelo que são necessárias 2 ABTs para representar um mapa regular de alturas. Um triângulo rectângulo isósceles será denotado por $T = (v_a, v_0, v_1)$ (ver Figura 4-23).

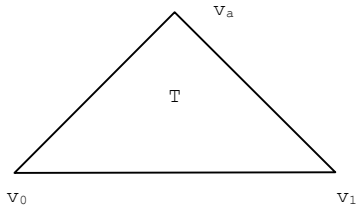


Figura 4-23: Triângulo rectângulo isósceles.

4.5.1.1 A Estrutura

A raiz de cada ABT é um triângulo rectângulo isósceles $T = (v_a, v_0, v_1)$ e corresponde ao nível menos refinado de divisão, $l = 0$. No nível seguinte, $l = 1$, os filhos deste triângulo (que são também eles triângulos rectângulo isósceles) são obtidos pela sua divisão em duas partes iguais ao longo da aresta $\{v_a, v_c\}$, sendo v_c o ponto médio da sua hipotenusa (ver Figura 4-24). Ao triângulo $T_0 = (v_c, v_a, v_0)$ chama-se filho esquerdo de T e ao triângulo $T_1 = (v_c, v_1, v_a)$ filho direito.

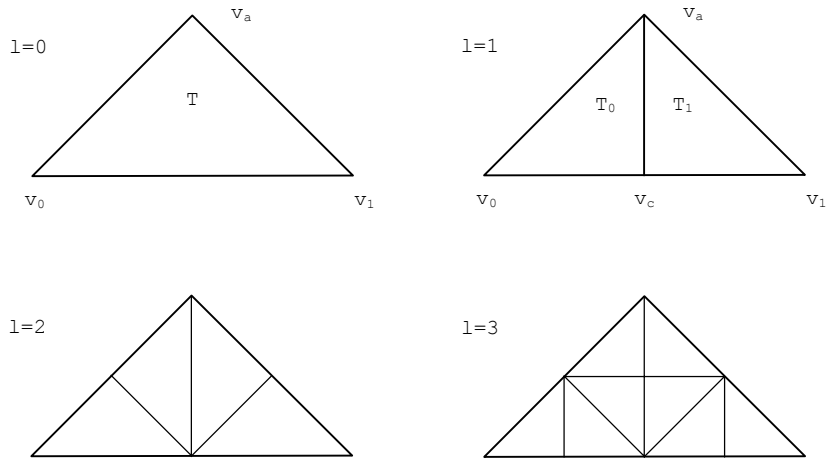


Figura 4-24: Os primeiros níveis de uma ABT.

O processo da divisão é repetido iterativamente para cada um dos triângulos (ver Figura 4-24), de acordo com a métrica definida.

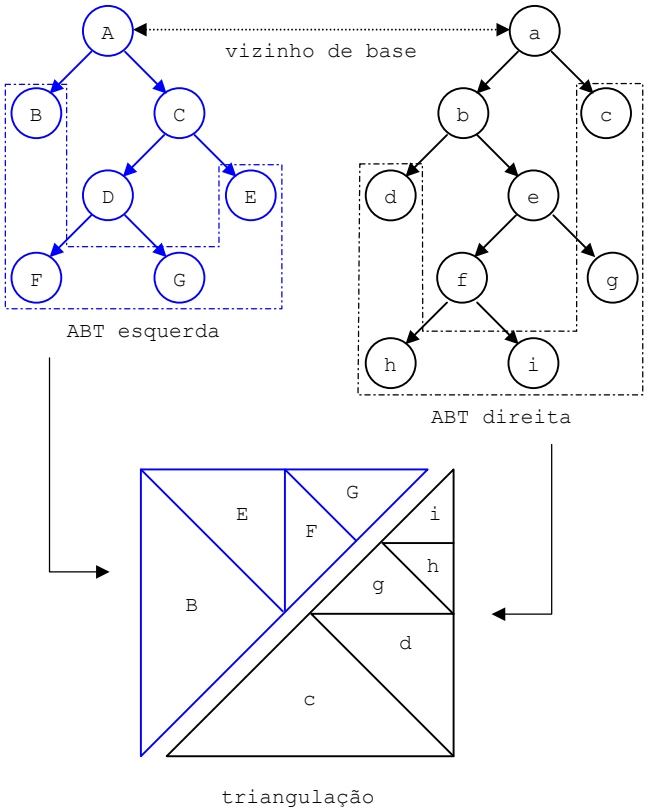


Figura 4-25: As ABTs correspondentes à triangulação apresentada.

A triangulação corrente pode ser construída utilizando todos os triângulos folha da ABT (ver Figura 4-25).

4.5.1.2 As Relações de Vizinhança

A construção da triangulação, tal como foi apresentada, permite que, com um pequeno esforço adicional, se obtenha uma triangulação sem descontinuidades espaciais. Para tal é necessário definir relações de vizinhança para os triângulos (ver Figura 4-26).

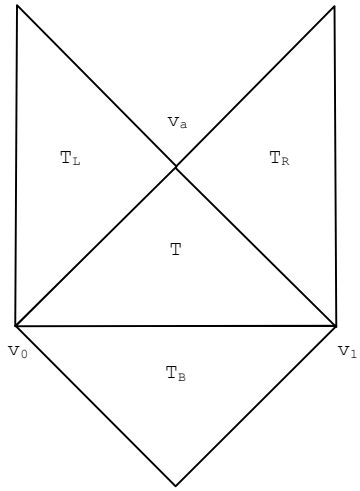


Figura 4-26: Relações de vizinhança entre triângulos de uma ABT.

Cada triângulo T da ABT tem 3 vizinhos (ver Figura 4-26):

- T_B é definido como sendo o vizinho de base, partilhando com T a aresta $\{v_0, v_1\}$.
- T_L é definido como sendo o vizinho da esquerda, partilhando a aresta $\{v_a, v_0\}$ com T.
- T_R é definido como sendo o vizinho da direita, partilhando a aresta $\{v_1, v_a\}$ com T.

Para evitar que durante as operações sobre os triângulos surjam descontinuidades espaciais na triangulação, basta garantir que:

- Os vizinhos da esquerda e da direita de em triângulo T numa ABT pertençam ao mesmo nível l ou ao nível seguinte $l+1$ da ABT.
- O vizinho de base seja do mesmo nível l ou do nível mais geral $l-1$ que T na ABT.

4.5.2 As Operações

Qualquer triangulação pode ser obtida a partir de outra à custa de uma sequência de duas operações: a divisão[37] e a fusão[38]. A primeira refina a triangulação, tornando-a mais definida, e a segunda generaliza-a, tornando mais simplificada.

[37] Do anglo-saxónico, *split*.

[38] Do anglo-saxónico, *merge*.

4.5.2.1 Divisão / Divisão Forçada

Quando um triângulo T e o seu vizinho de base T_B são ambos do mesmo nível, diz-se que o par (T, T_B) forma um diamante[39]. É apenas sobre este tipo de estrutura que se pode aplicar a operação de divisão. A única excepção a esta regra é quando T pertence à extremidade da triangulação, pois neste caso T_B não existe e a operação de divisão pode ser também realizada.

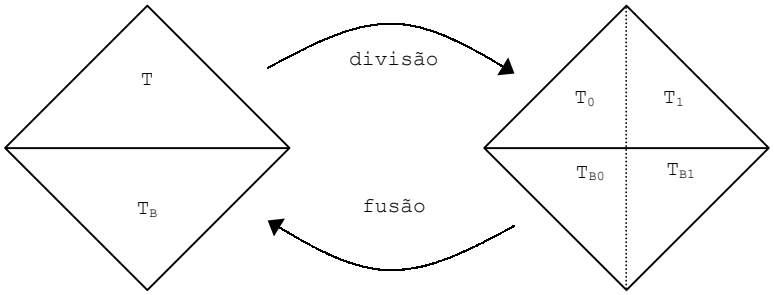


Figura 4-27: Operações de divisão e fusão.

Uma operação de divisão (ver Figura 4-27) substitui na triangulação um triângulo T pelos seus dois filhos (T_0, T_1) e, caso exista, o triângulo T_B pelos seus filhos (T_{B0}, T_{B1}). Esta operação introduz um novo vértice no centro do diamante, gerando desta forma uma triangulação contínua e mais refinada.

A aplicação desta operação apenas sobre diamantes permite garantir a continuidade espacial da triangulação.

Quando o par (T, T_B) não é um diamante[40], é necessário forçar a divisão de T_B para evitar descontinuidades espaciais. Esta operação, designada de divisão forçada[41], visa transformar o par (T, T_B) num diamante, antes da divisão de T.

A divisão forçada de T_B pode, por sua vez, despoletar mais divisões forçadas. Tal como T, T_B apenas pode ser alvo de uma operação de divisão se formar um diamante com o seu vizinho de base. Esta operação de divisão forçada é realizada de uma forma recursiva até que se encontre um triângulo que forme com o seu vizinho de base, na triangulação original, um diamante (ver Figura 4-28).

---

[39] Do anglo-saxónico, *diamant*.

[40] Neste caso, T_B pertence sempre ao nível acima, mais geral, de T.

[41] Do anglo-saxónico, *force split*.

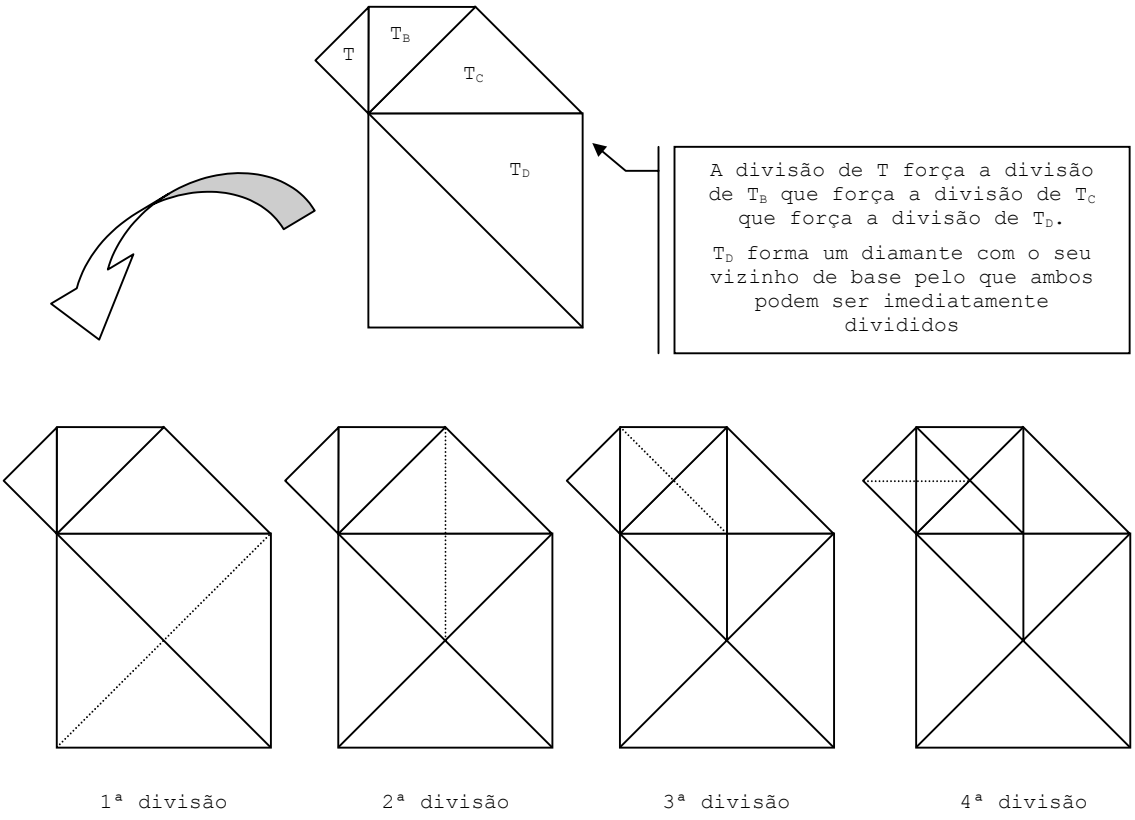


Figura 4-28: Divisão forçada.

4.5.2.2 Fusão

A operação inversa da divisão é a fusão (ver Figura 4-27). Esta operação remove da triangulação original os filhos T_0 e T_1 de T e os filhos T_{B0} e T_{B1} de T_B, substituindo-os, respectivamente, pelos seus pais T e T_B. Esta operação resulta no diamante (T, T_B).

Duchaineau et al. não especificaram a operação inversa da divisão forçada pelo que só é possível aplicar a operação de fusão se:

- Os 2 filhos de um triângulo T não tiverem filhos;
- Os 2 filhos do vizinho base de T, T_B, também não tiverem filhos (esta condição só é necessária se T tiver um vizinho base).

Quando nesta situação diz-se que estamos perante um diamante fundível[42].

[42] Do anglo-saxónico, *mergeable diamond*.

4.5.3 As Listas de Prioridades

A escolha do triângulo sobre o qual irá recair a próxima operação é baseada no conceito de prioridade.

A cada triângulo T está associada uma prioridade calculada com base no erro introduzido por se utilizar T em vez de uma triangulação mais refinada (isto é, uma triangulação com os filhos de T). Este cálculo é descrito nas secções seguintes.

Estas prioridades são armazenadas em duas listas, uma para cada uma das operações, com a finalidade de auxiliarem a selecção do próximo triângulo a operar.

Na lista de prioridades para divisão, Q_s, encontram-se armazenadas as prioridades de todos os triângulos existentes na triangulação corrente (isto é, todos os triângulos que são folhas da ABT). Sempre que um triângulo T é dividido, este é retirado de Q_s (pois T deixa de ser uma folha da ABT, e, consequentemente, de fazer parte da triangulação corrente) e os seus filhos inseridos em Q_s (pois passam a fazer parte da triangulação e folhas da ABT). Quando se efectua uma operação de divisão, é retirado desta lista o triângulo de maior prioridade, ou seja, o triângulo com o maior erro associado, e sobre ele é realizada a divisão.

Na lista de prioridades para fusão, Q_m, estão armazenadas as prioridades de todos os diamantes fundíveis existentes na triangulação corrente. Estas prioridades são definidas como sendo o máximo das prioridades dos triângulos que o constituem. Quando se efectua uma operação de fusão, é retirado desta lista o diamante de menor prioridade, que corresponde ao par de triângulos cuja fusão causará o menor aumento ao erro existente na triangulação corrente, e sobre ele é realizada a operação de fusão.

4.5.4 A Métrica

O ROAM utiliza no cálculo da métrica de erro a propagação dos erros geométricos[43] pelos níveis da ABT, desde o triângulo de maior resolução até à raiz da mesma.

Os erros geométricos são calculados *bottom-up*, começando pelo triângulo do nível mais refinado (no qual o erro é nulo) até à maior generalização possível do terreno (no qual o erro é máximo). O erro geométrico de um triângulo T é o máximo dos erros geométricos dos seus filhos acrescido do erro h introduzido por se utilizar T em vez dos seus filhos (ver Figura 4-29). A primeira parcela transporta o erro acumulado dos níveis mais refinados para os mais

[43] O volume associado ao erro geométrico é denominado em [8] por *wedgie*.

simplificados. Desta forma, o erro vai aumentando continuamente dos níveis mais refinados até à raiz da ABT.

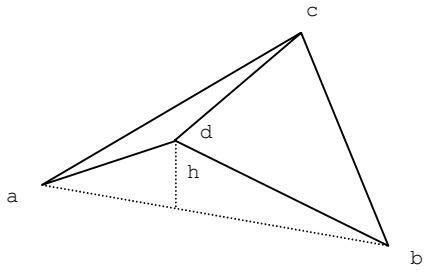


Figura 4-29: A altura h é o erro geométrico introduzido por se utilizar um triângulo abc em vez dos seus filhos acd e dbc.

Dada a natureza deste cálculo, que é independente dos parâmetros da câmara, ele é efectuado na componente de pré-processamento do algoritmo. Apenas quando o terreno sofre modificações em tempo real é necessário recalcular o erro geométrico durante a fase de execução.

4.5.4.1 O Cálculo da Prioridade de um Triângulo

As prioridades dos triângulos são calculadas utilizando o erro geométrico em *pixels*, isto é, a distância no ecrã, em *pixels,* entre a posição onde o ponto da superfície deveria estar e a posição onde a triangulação o coloca.

Seja $s(v)$ a posição correcta em coordenadas ecrã para o ponto v do terreno e $s_T(v)$ a posição desse ponto na triangulação T. Então, o erro geométrico em *pixels* em v é definida por

$$erro(v) = ||s(v) - s_T(v)||_2$$

O erro máximo em toda a imagem é dado por

$$erro_{max} = max\{erro(v) \mid v \in V\}$$

onde V é o conjunto de pontos do domínio cuja posição está dentro do volume de visualização.

O cálculo do erro geométrico em *pixels* para cada triângulo T é efectuado projectando o erro geométrico dos vértices de T no espaço ecrã. O valor máximo desta projecção é então definido como sendo a prioridade de T.

Note-se que os erros geométricos são monótonos por definição, isto é, o erro geométrico dos filhos nunca é superior à do pai, mas o erro geométrico em *pixels* pode não o ser, dado que resulta de uma projecção no ecrã. No entanto, numa nota [7] posterior à publicação do artigo, Duchaineau apresenta uma forma para garantir a monotonia dos erros geométricos em *pixels*, e por conseguinte das prioridades.

A monotonia das prioridades implica que o valor máximo da lista de divisão seja um limite para o erro geométrico em *pixels* máximo $erro_{max}$ existente na imagem [8].

4.5.5 O Algoritmo para a Triangulação

Duchaineau et al. [8] apresentam dois algoritmos para a criação de triangulações.

O primeiro, designado de Algoritmo de Divisão, tem como ponto de partida a triangulação base. Sobre esta são aplicadas operações de divisão/divisão forçada aos triângulos com prioridades mais altas, que são obtidos, a cada iteração, da lista de divisão, Q_s. Este algoritmo é apenas utilizado nas raras situações descritas na próxima secção e gera uma sequência de triangulações que minimiza a prioridade mais alta a cada iteração.

O segundo, designado de Algoritmo de Divisão/Fusão, tem como ponto de partida a triangulação anterior, pelo que tira partido da coerência entre *frames* por forma a melhorar o desempenho do algoritmo.

Considerando que as alterações dos parâmetros da câmara em duas *frame*s consecutivas são, geralmente, mínimas, as prioridades dos triângulos que fazem parte da triangulação sofrem poucas ou mesmo nenhumas alterações. Consequentemente, as triangulações de duas *frame*s consecutivas tendem a ser semelhantes. Para tirar partido desta coerência entre *frames*, este algoritmo assume a triangulação T_{f-1} como o ponto de partida para a construção da triangulação T_f.

A fila de fusão Q_m é utilizada neste contexto para auxiliar o algoritmo.

Nesta situação o algoritmo utilizado é:

```
Para todas frames f {
     Se f = 0 então {
          Seja T = triangulação base
          Limpar Qs e Qm
          Calcular prioridades para os triângulos e diamantes de T
          Inseri-las em Qs e Qm, respectivamente
     }
     senão {
          Seja T = Tf-1
          Actualizar prioridades para os triângulos de Qs
               e os diamantes de Qm
     }
```

```
     Enquanto o n° de triângulos de T é menor que um valor pré-definido
     ou o erro demasiado alto ou a máxima prioridade de divisão é maior
     que a mínima prioridade de fusão {
          Se o n° de triângulos de T é menor que um valor pré-definido
          ou o erro demasiado baixo {
               Identificar o (t,t_b) de menor prioridade em Q_m
               Fusão de (t,t_b)
               // Actualizar filas
               Remover de Q_s todos os filhos fundidos
               Inserir em Q_s os pais fundidos t e t_b
               Remover de Q_m o diamante (t,t_b)
               Inserir em Q_m todos os novos diamantes
          }
          senão {
               Identificar o t de maior prioridade em Q_s
               Dividir t
               // Actualizar filas
               Remover t e todos os triângulos divididos de Q_s
               Inserir novos triângulos de T em Q_s
               Remover de Q_m todos os diamantes cujos filhos foram
                    divididos
               Inserir em Q_m todos os novos diamantes
          }
     }
}
```

Figura 4-30: Algoritmo de divisão/fusão do ROAM.

Este algoritmo produz uma triangulação T_f, a mesma que seria produzida pelo Algoritmo de Divisão a partir da triangulação base, utilizando o menor número possível de operações de divisão/fusão sobre T_{f-1} [8]. Este número é proporcional ao número de triângulos que não são comuns a T_f e T_{f-1}, que no pior caso será igual ao número de triângulos de T_f mais o número de triângulos de T_{f-1}. Estas situações podem ser detectadas quando existe um grande número de triângulos e diamantes cujas prioridades estão entre a prioridade mínima da lista de fusão e a prioridade máxima da lista de divisão. Nestes casos é aconselhável optar por uma reinicialização da triangulação, aplicando o Algoritmo de Divisão [8].

4.5.6 Questões Relacionadas com o Desempenho

Duchaineau et al. [8] descrevem algumas optimizações que permitem melhorar o desempenho do algoritmo.

4.5.6.1 Construção Incremental de Tiras

A organização de triângulos em tiras é das optimizações que permite melhorar o desempenho do algoritmo. A breve descrição apresentada em [8] sobre o algoritmo de *stripping* classifica-o como simples, incremental e não óptimo, gerando tiras com um comprimento médio de 4 a 5 triângulos.

Basicamente, novos triângulos são inseridos como tiras de um só elemento, que são sucessivamente alvo de tentativas de junção com as extremidades de tiras vizinhas. Sempre que é necessário remover um triângulo, a tira à qual pertence ou é encurtada na extremidade, dividida em duas, ou removida.

4.5.6.2 Cálculo Parcial das Prioridades

Como foi mencionado anteriormente, as mudanças dos parâmetros da câmara entre duas *frame*s consecutivas são, normalmente, mínimas, pelo que as prioridades dos triângulos nas listas de divisão e fusão quase ou nada se alteram. Esta coerência entre *frames* permite evitar o cálculo destas prioridades em algumas *frame*s, o que possibilita melhorar o desempenho do algoritmo [8].

Assim, só é necessário efectuar este cálculo quando este afectar, potencialmente, as operações de divisão e fusão.

Para tal é necessário determinar um limite de velocidade para a câmara, o que permite calcular um limite para as prioridades dos triângulos em função do tempo. Por outro lado, a *crossover priority*, que é definida como sendo a máxima prioridade da lista de divisão quando todo o processo de simplificação termina, varia muito lentamente de *frame* para *frame* (cerca de 1%) [8]. Assim, o cálculo da nova prioridade de um triângulo pode ser adiada até que a sua prioridade ultrapasse a *crossover priority*.

É mantida uma lista de espera de triângulos para cada uma das próximas doze *frame*s, e, em cada *frame*, só os triângulos da lista respectiva necessitam de ter as suas prioridades recalculadas. Caso a fracção de tempo destinada a esta operação não se tenha esgotado, poder-se-á calcular as prioridades dos triângulos das listas de espera subsequentes.

Após o cálculo, os triângulos são colocados na lista de espera mais afastada, o que permite um escalonamento seguro destes cálculos.

4.5.6.3 Optimização Progressiva

Esta optimização assegura uma taxa de FPS constante, suspendendo as operações que melhoram a triangulação sempre que o tempo definido para cada *frame* expira.

A estrutura do ROAM facilita esta implementação pois as operações de divisão/fusão são aplicadas sequencialmente, num processo continuo de melhoramento da triangulação, executado por ordem crescente de importância. Desta forma é garantido que esta será a triangulação que produz o menor erro para o número de operações efectuadas.

Todas as fases do ROAM, incluindo o cálculo parcial das prioridades, são susceptíveis de serem limitadas em função do tempo da sua execução, à excepção do *view frustum culling*. Esta excepção não se apresenta como um problema grave pois esta fase requer apenas um fracção mínima do tempo de *frame* [8] e é completada antes das optimizações à triangulação e do cálculo parcial das prioridades.

4.6 Variações do Algoritmo ROAM

Após a apresentação do artigo de Duchaineau et al., muitos autores apresentaram novas versões do ROAM, na tentativa de melhorar o desempenho obtido pelo original.

Apresentam-se de seguida algumas dessas implementações.

4.6.1 Continuous Level of Detail In Real-Time Terrain Rendering

Ögren desenvolveu em [27] uma implementação do ROAM para o simulador de voo *Ericsson Saab Avionics AB*, cujas principais diferenças residem num novo processo de fusão e na ausência de listas de divisão e fusão.

O algoritmo de Ögren efectua, em cada iteração, uma travessia *top-down* à ABT criada nas iterações anteriores. Durante a travessia, os triângulos visitados são avaliados e a ABT é actualizada, se necessário, de acordo com a métrica definida. A avaliação de cada triângulo pode implicar uma operação de divisão ou fusão, resultando em novos nodos ou na remoção de nodos da ABT, respectivamente.

A métrica utilizada no processo de avaliação é baseada no erro geométrico e na distância. É uma métrica computacionalmente mais rápida mas menos precisa que a utilizada no ROAM [27].

Ögren utilizou árvores binárias implícitas e não implementou as optimizações sugeridas por Duchaineau et al.

4.6.1.1 As Operações

Das duas operações sobre triângulos apenas a de fusão difere do ROAM.

A operação de fusão proposta por Ögren permite juntar os filhos de qualquer triângulo mesmo que estes tenham sido divididos. Desta forma, a operação de fusão pode ser aplicada a triângulos que não sejam diamantes fundíveis, isto é, qualquer triângulo previamente dividido pode ser alvo de uma operação de fusão (ver Figura 4-31).

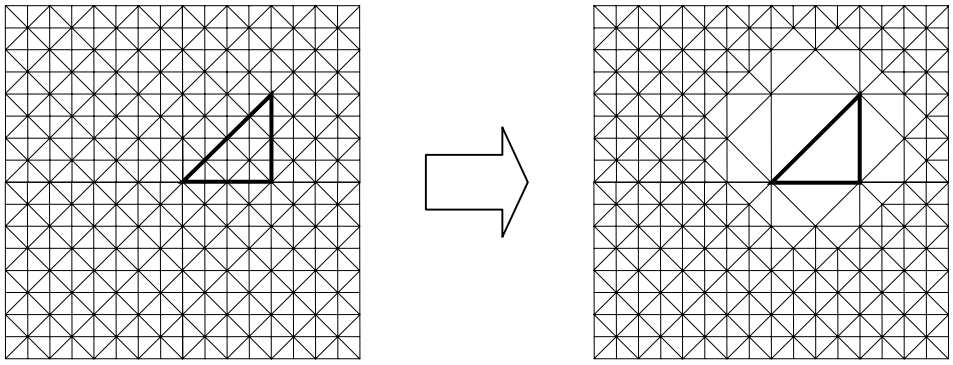


Figura 4-31: A operação de fusão forçada.

Nesta operação todos os descendentes do triângulo fundido são removidos e todos os triângulos vizinhos são também alvo da operação de fusão, para que seja mantida a continuidade espacial da triangulação. A este processo recursivo Ögren chamou de fusão forçada[44] em analogia com a divisão forçada de Duchaineau et al..

As principais vantagens da utilização da fusão forçada são, de acordo com Ögren [27]:

- Simplicidade da implementação e compreensão, não havendo a necessidade de introduzir conceitos como diamantes e diamantes fundíveis.
- Maior versatilidade da operação de fusão. A operação pode ser aplicada a qualquer triângulo da ABT.
- Melhoria do desempenho, pois não é necessário manter registo sobre os diamantes e diamantes fundíveis e uma só operação de fusão forçada equivale a uma sequência de operações de fusão original.
- Resolução implícita dos casos especiais referidos por Duchaineau et al., nos quais o número de triângulos diferentes entre uma triangulação e a seguinte é significativo.

[44] Do anglo-saxónico, *forced merge*.

Por outro lado, a utilização da operação de fusão forçada não é compatível com a optimização do cálculo parcial das prioridades utilizada no algoritmo original.

4.6.2 Real-Time Dynamic Level of Detail Terrain Rendering with ROAM

Turner apresenta em [32] um algoritmo que combina o ROAM com a divisão do terreno em blocos.

Antes da criação de qualquer ABT o terreno é dividido em vários blocos de igual dimensão. Para cada bloco são criadas duas ABTs como se do terreno na totalidade se tratasse. As vantagens desta abordagem são, de acordo com Turner [32]:

1. Melhoria na travessia das ABTs, pois apresentam menor profundidade.
2. Evitar o cálculo completo dos erros geométricos pré-calculados sempre que se realizarem actualizações do mapa regular de alturas em tempo de execução, pois o cálculo pode ser restrito ao bloco que representa a região de terreno modificada.
3. Facilitar o *page-in* de terrenos de grandes dimensões.

O algoritmo começa por carregar para memória o mapa regular de alturas e dividi-lo em vários blocos de igual dimensão. No contexto de cada bloco, são criadas duas ABTs que representam a triangulação da região do terreno a que estão associadas. Estas ABTs são actualizadas, em todas as *frames* seguintes, de acordo com o algoritmo Divisão/Fusão do ROAM. Por fim, efectuam-se as travessias às ABTs de todos os blocos e procede-se à visualização dos triângulos correspondentes às folhas da árvore.

4.6.2.1 A Métrica

A métrica utilizada é o erro geométrico ajustado pelo factor distância à câmara. Tal como no ROAM, o erro geométrico é pré-calculado e propagado desde o nível de maior resolução possível numa ABTs até à raiz da mesma. Dado que estes erros mantêm-se inalterados (excepto quando o mapa regular de alturas sofre alterações), Turner armazena-os em árvores implícitas de variância para auxiliarem os critérios de decisão utilizado nas operações de divisão e fusão.

Cada bloco tem duas árvores de variância, uma para cada ABT utilizada.

4.6.2.2 Optimizações

Turner propõe ainda algumas optimizações ao algoritmo:

- *Leques de Triângulos*. Turner propõe a utilização de leques de triângulos para acelerar a visualização da triangulação criada. A fim de se conseguir uma ordem correcta dos triângulos, a travessia da ABT deve ser modificada. Deve visitar-se

alternadamente os filhos esquerdo e direito, começando pelo filho esquerdo no primeiro nível.

Para além da ordem dos triângulos, é necessário garantir uma ordem correcta dos vértices. Para se determinar qual o vértice no centro de cada leque, é necessário passar para os níveis inferiores da ABT uma referência para o vértice que, até ao momento, é o melhor candidato a centro do leque. Em cada nível, o vértice candidato pode ser modificado.

Turner refere uma média de 3 a 4 triângulos por leque, num máximo possível de 8.

- *Terrenos de Grandes Dimensões*. A utilização da estruturação em blocos permite construir um terreno à custa de outros terrenos de menores dimensões. Para tal apenas é necessário garantir que as fronteiras das estruturas que mantêm os blocos estejam devidamente actualizadas para evitar inconsistências.

4.6.3 Terrain Rendering at High Levels of Detail

Blow [1] implementou uma variação do ROAM com o objectivo de visualizar interactivamente, a partir de qualquer ângulo, terrenos de grandes dimensões (2^{15} x 2^{15}) com uma textura de grande resolução (2^{36} *texels*) e com uma distância ao horizonte ilimitada.

Blow concluiu que o ROAM é um algoritmo ineficiente em situações de grande detalhe, dado que a sua métrica é incapaz de distinguir entre os triângulos que se aproximam e os que se afastam da câmara [1]. O facto da métrica se resumir a um valor escalar de prioridade, implica ainda que as listas de prioridades utilizadas no ROAM considerem que todos os triângulos se estão a aproximar da câmara à mesma velocidade relativa, pelo que o algoritmo passa muito tempo a reavaliar as prioridades dos triângulos [1].

4.6.3.1 A Métrica

Neste sentido, Blow utiliza uma métrica mais consciente da relação espacial entre os triângulos. Ao contrário do ROAM e de Lindstrom et al., que calculam uma métrica escalar p para cada triângulo a partir de coordenadas de um determinado ponto (x,y,z) e de um valor h relativo à altitude ($p = f(x,y,z,h)$), neste algoritmo a informação tridimensional não é comprimida num resultado unidimensional, mas são utilizadas as três dimensões da informação no processo de simplificação.

Considerando que h é constante para um determinado vértice e que o algoritmo procura fundamentalmente lidar com a deslocação da câmara em relação aos vértices do terreno, Blow em vez de ordenar os triângulos pela sua prioridade unidimensional, mantém uma estrutura de

dados de iso-superfícies. Estas iso-superfícies são da forma $q(a,b,c) = p$, onde q é uma função dependente do vértice em causa que incorpora o valor h relativo à altitude e (a,b,c) é a posição da câmara. A função q é tal que as iso-superfícies utilizadas são esferas.

Desta forma, os volumes associados aos erros geométricos dos triângulos são aqui representados por esferas. Sempre que a câmara entra no volume definido pela esfera, o triângulo é dividido. Sempre que a câmara sai da esfera, é efectuada uma operação de fusão.

A fim de evitar o teste exaustivo de todos os volumes, é criada uma hierarquia de esferas com vista à melhoria do desempenho do algoritmo [1]. Se uma esfera B está contida numa outra esfera A, a câmara não entra em B sem primeiro ter entrado em A. Ao criar uma hierarquia de esferas, na qual B é filha de A, a esfera B nunca é testada sem que A tenha sido dividida, isto é, penetrada (ver Figura 4-32).

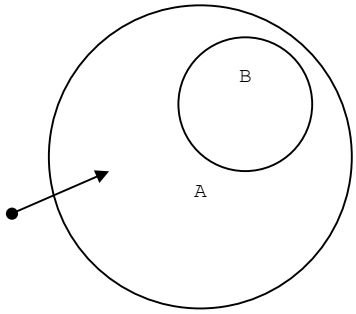


Figura 4-32: Hierarquia de esferas.

A simples utilização desta hierarquia não evita contudo que na raiz existam centenas a milhares (dependendo do nível de detalhe utilizado) de esferas [1]. Por conseguinte, a técnica de *clustering* é utilizada: são criadas esferas sem qualquer correspondência com os triângulos utilizados, com o único propósito de agrupar um conjunto de esferas da hierarquia. A utilização desta técnica permite aumentar significativamente o desempenho do algoritmo [1].

Utilizando esta métrica para construir a triangulação, Blow melhora os resultados obtidos pelo ROAM: para o mesmo máximo de erro geométrico em *pixels* são necessários menos triângulos e para o mesmo número de triângulos é alcançada uma melhor aproximação [1].

4.7 Geometrical MipMapping

Boer apresenta em [2] um algoritmo implementado no projecto *E-mersion*, a que deu o nome de *Geometrical MipMapping* (GeoMipMapping). Este algoritmo procura tirar partido das potencialidade do *hardware* gráfico disponível actualmente, capaz de processar e visualizar grandes quantidades de triângulos por *frame*. A motivação foi a de simplificar os

processamentos que geram esses triângulos, procurando obter não o conjunto perfeito destes mas o conjunto que conduz à menor utilização possível do CPU, implicando o envio pelo *pipeline* gráfico do máximo de triângulos que o *hardware* consiga processar.

Desta forma, não é realizada nenhuma simplificação a nível dos vértices ou triângulos, mas apenas a um nível mais alto de refinamento.

4.7.1 Os Dados

O algoritmo utiliza mapas regulares de alturas de dimensão $2^n+1 \times 2^n+1$ vértices, onde $n \in [1,\infty[$, para a representação do terreno. O terreno é dividido, durante a fase de pré-processamento, em blocos de tamanho fixo pré-definido com a forma 2^m+1, $m \in [1,n[$. Estes blocos são utilizados para realizar o *view frustum culling* e representam também o nível 0 de um GeoMipMap.

4.7.2 Os GeoMipmaps

Os blocos visíveis que se encontram afastados da câmara não devem ser visualizados com o mesmo detalhe daqueles que se encontram perto. Estes devem ser aproximados com uma triangulação de menor resolução, diminuindo-se assim o número de triângulos e, consequentemente, aumentando o desempenho do algoritmo.

Ao contrário da maioria dos algoritmos multi-resolução contínua, Boer não baseia o seu método de simplificação ao nível dos triângulos.

Fazendo uma analogia com a técnica de *mipmapping*[45] utilizada no contexto das texturas, este algoritmo aplica este conceito às triangulações, tratando os blocos de terreno como se de texturas se tratasse.

O objectivo é o de criar *mipmaps* geométricos com os blocos, obtendo cada nível desse *mipmap* a partir da redução para metade do bloco do nível anterior. Para designar o bloco original utiliza-se o termo GeoMipMap de nível 0. Os próximos blocos são designados de GeoMipMap de nível n, onde n assume valores inteiros de 1 a ∞. O GeoMipMap de nível $n+1$ tem metade da resolução do GeoMipMap de nível n.

---

[45] *Mipmaps* são uma sequência de texturas pré-filtradas de resoluções decrescentes. *Mip* é a abreviatura em Latim para *multum in parvo*, isto é, muitas coisas num espaço pequeno. Para se utilizar esta técnica, é necessário fornecer todas as dimensões da textura em potências de 2, desde a maior ($2^n x 2^m$, com n,m) à mais pequena ($1x1$). No decurso da visualização a dimensão mais adequada é seleccionada com base na dimensão em *pixels* do objecto a mapear com a textura.

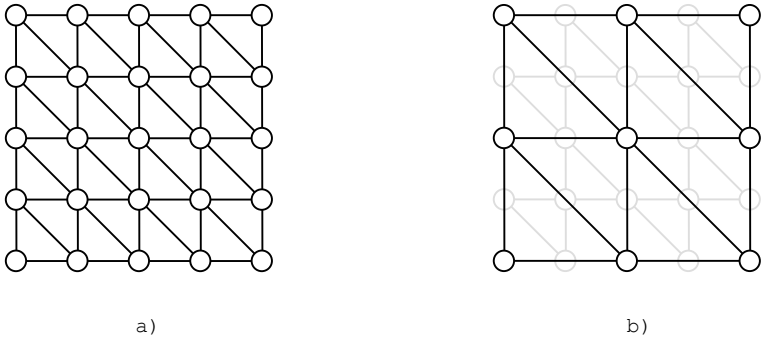


Figura 4-33: a) GeoMipMap de nível *n*. b) GeoMipMap de nível *n+1*.

Considere-se a Figura 4-33. a) representa um exemplo de um GeoMipMap de nível *n* de um bloco de dimensão *5x5* e em b) é apresentado o GeoMipMap de nível *n+1* resultante da redução para metade do GeoMipMap de a).

4.7.2.1 A Escolha do Nível de Detalhe Apropriado

Depois de criados os GeoMipmaps é necessário seleccionar, durante a visualização, o nível mais adequado a utilizar para representar uma determinada região do terreno.

Quando se transita de um nível *n* para um nível *n+1*, a remoção de vértices provoca um erro geométrico δ na altitude de cada vértice.

Dado que um bloco é constituído por vários vértices, o maior erro desses vértices representa a pior situação do bloco, pelo que é utilizado como factor de decisão. Este erro máximo, δ_B, é calculado e armazenado durante a fase de criação dos diferentes níveis do GeoMipMap, pois o seu valor é independente dos parâmetros da câmara.

Para determinar o erro perceptível num dado bloco, δ_B é projectado para o espaço ecrã durante a fase de decisão, obtendo-se assim um erro dependente dos parâmetros da câmara ε_B que será tanto menor quanto maior for a distância *d* do centro do bloco à câmara.

Se ε_B for maior que um limite τ pré-definido então deverão ser considerados recursivamente os níveis seguintes, até que o nível mais adequado seja atingido, isto é, até que o erro ε_B nesses níveis seja menor que o limite τ.

4.7.2.2 Acelerar a Escolha do Nível do GeoMipMap

Calcular ε_B para todos os GeoMipMaps visíveis numa *frame* e compará-lo com τ pode ser computacionalmente dispendioso [2]. É possível pré-calcular ε_B, ao considerar-se uma simplificação na qual o vector direcção da câmara se mantém permanentemente na horizontal.

Ainda que esta simplificação do critério conduza a blocos demasiado detalhados quando a inclinação da câmara em relação ao GeoMipMap é elevada, e consequentemente ao envio de uma maior quantidade de triângulos pelo *pipeline* gráfico, diminui-se consideravelmente a utilização do CPU, o que vai ao encontro do objectivo inicial do algoritmo.

Consegue-se assim acelerar todo o processo de selecção calculando-se na fase de carregamento do terreno a distância mínima, D, à qual se pode optar por este nível. Comparando a distância da câmara com D_n (sendo n o nível do GeoMipMap) determina-se o nível do GeoMipMap a utilizar na visualização de um dado bloco.

O valor dessa distância mínima é dado pela equação:

$$D_n = |\delta| * C$$

onde C é uma constante dada pela equação:

$$C = \frac{A}{T}, \text{ sendo } A = \frac{n}{|t|} \text{ e } T = \frac{2 * \tau}{v_{res}}$$

na qual n é o plano de corte do volume de visualização mais próximo da câmara, t é a coordenada de topo desse plano e v_{res} a resolução vertical do ecrã em *pixels*.

O valor de D_n é comparado, em cada *frame*, com a distância d da câmara ao centro de cada bloco[46].

4.7.2.3 Continuidade Espacial

Tal como em todas as soluções baseadas na divisão do terreno em blocos, criam-se potenciais situações de descontinuidade espacial nas fronteiras de blocos adjacentes aproximados com níveis de detalhe diferentes.

A solução para este problema passa pelo reorganização das conexões entre os vértices.

Boer propõe a alteração da conectividade dos vértices do bloco de maior detalhe, isto é, do bloco de nível inferior.

[46] Na prática, são comparados os valores de d^2 e D_n^2, para se evitar o uso da raiz quadrada no cálculo de d.

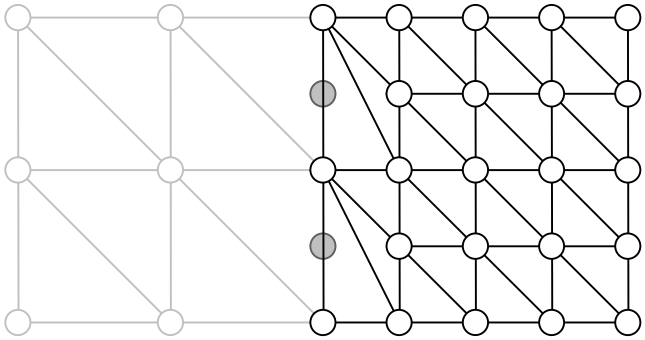


Figura 4-34: A solução para o problema das descontinuidades espaciais quando dois blocos adjacentes são aproximados com GeoMipMaps de níveis diferentes.

A Figura 4-34 representa essa alteração na conectividade. O bloco de esquerda (a cinza) é representado por um GeoMipMap de menor detalhe (nível $n+1$) e o da direita (a preto) por um de maior detalhe (nível n).

Os dois vértices cinzentos, causadores da descontinuidade espacial, não são incluídos na conectividade final, permitindo que a fronteira esquerda desse bloco coincida com a fronteira direita do bloco adjacente apresentado na figura.

Esta solução implica que todos os GeoMipMaps de nível 0 conheçam os seus quatro vizinhos, isto é, que tenham uma referência para cada um dos blocos adjacentes.

4.7.2.4 Trilinear GeoMipMapping

Para evitar descontinuidades temporais causadas pela mudança do nível de GeoMipMap para um dado bloco, num dado instante de tempo, Boer utiliza uma técnica que mais uma vez advém de uma analogia com o *mipmapping* de texturas: o *Trilinear Filtering*[47].

O GeoMipMap de nível n tem associado um determinado valor pré-calculado D_n que indica a que distância da câmara este nível deve ser substituído pelo nível antecessor. Da mesma forma, o GeoMipMap de nível $n+1$ tem um valor equivalente D_{n+1}. A fracção

$$t = \frac{d - D_n}{D_{n+1} - D_n}$$

[47] Quando se utiliza o *mipmapping* em texturas, é possível detectar a linha que une dois níveis diferentes, e que ocorre a distância fixas ao observador. Esta situação pode ser eliminada com o *Trilinear Filtering*, que não é mais do que uma interpolação entre um *mipmap* e o *mipmap* do nível superior utilizando uma distância fraccional dos dois valores pré-calculados d dos *mipmaps*.

pode ser utilizada como um multiplicador para os vértices do nível n que não fazem parte de do nível $n+1$[48]. Desta forma, as sucessivas posições de um vértice v podem ser dadas pela equação:

$$v' = v - t \cdot \delta_v$$

Esta operação permite a transição suavemente do nível n para o nível $n+1$, até que d atinja o valor de D_{n+1}, altura na qual o GeoMipMap de nível n é definitivamente substituído pelo GeoMipMap de nível $n+1$.

4.8 View-Dependent Progressive Meshes

Hoppe introduz em [18] uma estrutura para representar triangulações a que deu o nome de *Progressive Mesh* (PM).

Ao contrário dos restantes métodos apresentados nesta dissertação, este algoritmo gera uma triangulação irregular a partir de qualquer tipo de amostras (regulares ou irregulares). No contexto dos terrenos, são normalmente geradas triangulações irregulares a partir de um mapa regular de alturas, dado ser esta a estrutura mais comum para o armazenamento de informação relativa a terrenos.

Para além da simplificação do terreno, preservando a sua geometria e os seus atributos de acordo com uma determinada métrica, as PMs permitem também a compressão e a transmissão progressiva da triangulação utilizada.

A construção de uma PM apresenta-se assim como um algoritmo de simplificação de triangulação, inicialmente optimizado para níveis de detalhe independentes dos parâmetros da câmara. Posteriormente, Hoppe desenvolveu o refinamento incremental de PMs dependente dos parâmetros da câmara (VDPM) [19] que, mais tarde, foi especializado para o caso da visualização de terrenos [20].

Uma PM descreve uma triangulação arbitrária M^n como uma triangulação de base M^0 e uma sequência de n operações de refinamento $\{vsplit_0, \ldots, vsplit_{n-1}\}$. Esta sequência representa o processo de refinamento incremental de M^0 do qual resulta a triangulação M^n.

Assim, uma PM define uma sequência contínua de triangulações M^0, M^1, ..., M^n de crescente precisão, a partir da qual se pode obter uma aproximação ao terreno com a resolução pretendida.

[48] Exceptuam-se os vértices que fazem parte da fronteira com um GeoMipMap de menor detalhe, pois estes não contribuem para a triangulação final.

A sequência de refinamentos `vsplit` define univocamente uma hierarquia de vértices (ver Figura 4-35) na qual os nodos da raiz correspondem aos vértices da base M^0 e os vértices das folhas correspondem a triangulação detalhada M^n.

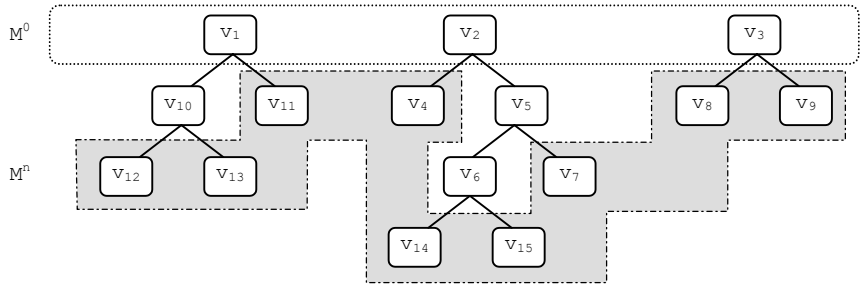


Figura 4-35: Representação de uma *progressive mesh*.

Duas operações podem ser aplicadas a uma triangulação: *vertex split* e *edge collapse* (ver Figura 4-36).

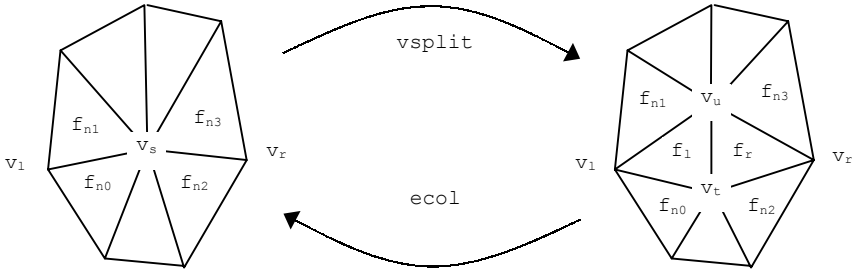


Figura 4-36: A operação de *vsplit* e a sua inversa, a operação de *ecol*.

4.8.1 Edge Collapse

A operação *edge collapse* (`ecol`) unifica 2 vértices adjacentes v_u e v_t num só vértice v_s (ver Figura 4-36).

Assim, a triangulação M^n pode ser simplificada para uma triangulação mais geral M^0 por aplicação sucessiva de n operações de *edge collapse*.

$$M^n \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0$$

Esta operação, parametrizada como $ecol(v_s, v_t, v_u, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$, modifica a triangulação, substituindo os dois filhos v_t e v_u pelo vértice pai v_s, e removendo as faces $f_l=\{v_s, v_t, v_l\}$ e $f_r=\{v_s, v_r, v_t\}$ entre dois pares de faces vizinhas (f_{n0}, f_{n1}) e (f_{n2}, f_{n3}) adjacentes a v_s.

4.8.2 Vertex Split

A operação inversa de *edge collapse* é designada por *vertex split* (`vsplit`) (ver Figura 4-36). Com esta operação pode representar-se uma triangulação M^n como uma triangulação de base M^0 e uma sequência de n operações de `vsplit`. É desta forma que a PM é armazenada, pois permite uma organização *top-down* da triangulação, facilitando a sua visualização e transmissão.

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} M^n$$

Esta operação, parametrizada como $vsplit(v_s, v_t, v_u, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$, substitui o vértice pai v_s por dois filhos v_t e v_u, e introduz duas novas faces $f_l=\{v_s, v_t, v_l\}$ e $f_r=\{v_s, v_r, v_t\}$ entre dois pares de faces vizinhas (f_{n0}, f_{n1}) e (f_{n2}, f_{n3}) adjacentes a v_s.

De notar que para suportar triangulações com fronteiras, as faces f_{n0}, f_{n1}, f_{n2}, f_{n3} podem serem nulas e operações de `vsplit` com $f_{n2} = f_{n3} = nil$ criam apenas uma face f_l.

4.8.3 Refinamento e Generalização Selectivos

A hierarquia de vértices, construída à custa das operações apresentadas, permite a criação de triangulações refinadas selectivas, isto é, triangulações que não estão necessariamente na sequência $M^0 \ldots M^n$, e que correspondem a uma travessia horizontal pela hierarquia de vértices.

Após a construção de uma PM, pode realizar-se, em tempo real, um refinamento selectivo sobre esta PM, isto é, pode seleccionar-se uma triangulação M^s obtida a partir da triangulação base M^0, aplicando uma sequência $S \subseteq \{0, \ldots, n-1\}$ de operações `vsplit` (ver Figura 4-37). A esta triangulação M^s dá-se o nome de triangulação activa[49].

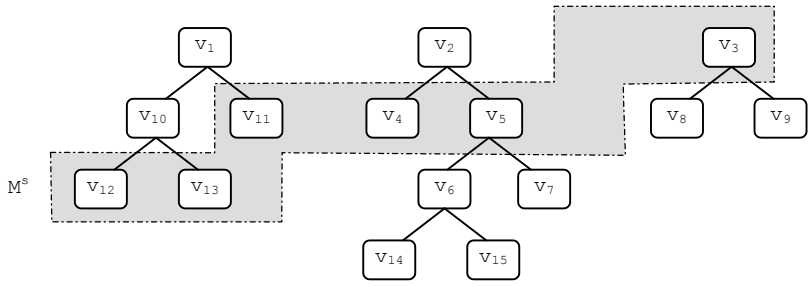


Figura 4-37: Uma triangulação refinada selectiva M^s.

[49] Do anglo-saxónico, *active mesh*.

Esta selecção pode, no entanto, levar a triangulações inconsistentes que é necessário evitar com um conjunto de pré-condições:

- A operação `vsplit` só é válida se:
    - v_s é um vértice activo, e
    - as faces $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$ são todas activas
- A operação `ecol` só é válida se:
    - v_t e v_u são os dois vértices activos, e
    - as faces adjacentes a f_l e f_r são $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$ (ver Figura 4-36)

Diz-se que um vértice ou face está activo se este existir na triangulação activa M^s.

4.8.3.1 A Métrica

No contexto dos terrenos, a métrica de erro utilizada é o erro geométrico em *pixels*.

O cálculo do erro geométrico é realizado na fase de pré-processamento e, em tempo de execução, é projectado no ecrã para tornar o critério de decisão dependente dos parâmetros da câmara.

Dada a natureza irregular da aproximação obtida por este algoritmo, o cálculo do erro geométrico compara a triangulação utilizada com a triangulação regular do mapa regular de alturas.

Assim, o erro geométrico associado a um vértice v_s é obtido pelo máximo dos erros geométricos existentes nos seguintes vértices[50] (ver Figura 4-38):

- Vértices do mapa regular de alturas que são internos às faces adjacentes a v_s (assinalados na figura com □).
- Vértices da intersecção das arestas adjacentes a v_s com as arestas dos triângulos da triangulação regular do mapa regular de alturas (assinalados na figura com ○).

[50] De notar que no vértice v_s o erro geométrico é zero, pois v_s faz parte da triangulação utilizada.

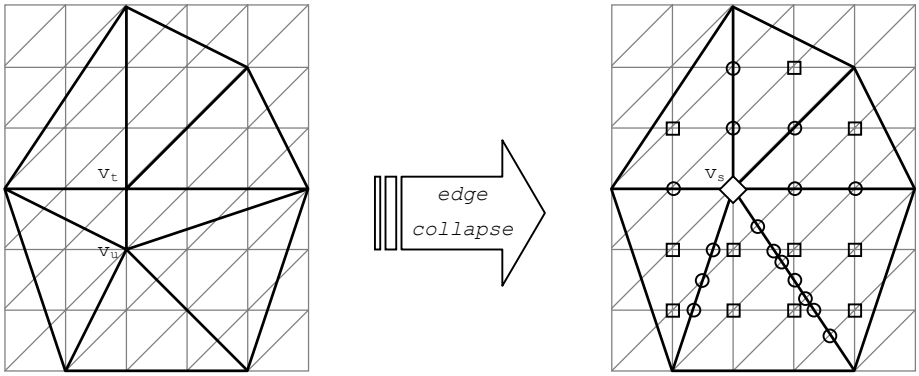


Figura 4-38: Cálculo do erro geométrico para um vértice.

O erro geométrico calculado é armazenado para ser utilizado, em tempo de execução, no critério de decisão. O critério de decisão projecta no ecrã o erro geométrico e compara-o com um determinado limite τ.

Antes do cálculo da métrica descrito, Hoppe utiliza o *view frustum culling* baseado em *bounding spheres* por forma a reduzir a carga gráfica. É calculada uma *bounding sphere* para cada vértice v, que cobre a região da triangulação que contém v e todos os seus descendentes. v não é refinado se essa *bounding sphere* se encontrar completamente fora do volume de visualização.

4.8.3.2 O Algoritmo de Refinamento e Generalização Selectivos

O algoritmo da Figura 4-39 resume o processo de refinamento e generalização selectivos descrito anteriormente.

Para todos os vértices da triangulação activa, determina-se se o vértice deve ser refinado. Em caso afirmativo, o vértice sofre um refinamento que poderá conduzir a refinamentos forçados sucessivos de outros vértices, no caso desta operação não ser válida.

Se o vértice não deve ser refinado, o seu vértice pai é generalizado se a operação de `ecol` for válida e se o vértice pai também não deve ser refinado.

```
Seja V = lista de vértice activos
Para todo o vértice v ∈ V {
     Se v tem filhos e deve refinar v
          vsplit forçado(v)
     senão {
          Se ecol(pai de v) é válida e pai de v não deve ser refinado
               ecol(pai de v)
     }
}
```

Figura 4-39: Algoritmo de refinamento e generalização selectivos.

Em resumo, pretende refinar-se sempre que necessário e generalizar sempre que possível. Depois de aplicada qualquer uma das operações de *vsplit* ou *ecol* a um vértice, alguns dos vértices adjacentes devem ser considerados para futuras transformações.

4.8.4 Construção de uma Progressive Mesh Hierárquica

Em [20], Hoppe desenvolve um esquema hierárquico para a construção de uma PM para a representação de terrenos de grandes dimensões, com o objectivo de:

- Aplicar o processo de simplificação descrito a terrenos cuja dimensão, aliada à própria estrutura necessária ao processo de simplificação, não permite tê-los completamente em memória.
- Utilizar um esquema que permita o *pre-fetch* dos dados necessários para memória.

O processo que aqui se descreve é efectuado durante uma fase de pré-processamento e o resultado é armazenado para posteriormente ser utilizado durante a visualização. Apesar deste processo gerar grandes quantidades de informação durante a fase de pré-processamento, requer apenas uma estrutura leve e dinâmica para auxiliar a sua visualização [20].

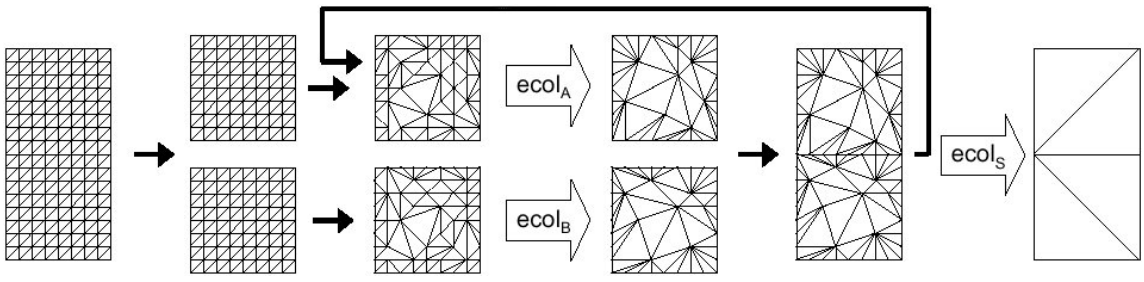


Figura 4-40: Construção recursiva de uma PM hierárquica.

Durante a fase de pré-processamento, o terreno é dividido em blocos e, de seguida, é executado o seguinte esquema recursivo (ver Figura 4-40):

- Cada bloco é simplificado por aplicação sucessiva de operações $ecol$, por forma a gerar a aproximação com o menor erro possível, de acordo com a métrica apresentada. Este processo termina quando esse erro ultrapassa um limite pré-definido.

  Estas simplificações são efectuadas por forma a deixar os vértices fronteira intactos e evitar assim dependências entre blocos.

- As sequências de simplificações utilizadas ($ecol_A$, $ecol_B$, ...) são guardadas.

- Os blocos simplificados são juntos 2 a 2 para que o processo de simplificação evolua para níveis de simplificação maiores.

- Todo o processo é repetido para a criação do próximo nível utilizando os blocos resultantes do ponto anterior.

O primeiro e o último níveis de simplificação são alvo de um processamento diferente. No primeiro nível, são ignoradas, em cada bloco, todas as operações $ecol$ iniciais até que uma tolerância de erro pré-definida seja ultrapassada. Apesar desta decisão truncar a hierarquia final, eliminando o conjunto inicial de simplificações em cada bloco, ela permite reduzir custos de armazenamento intermédio de informação. No caso do último nível, no qual existe apenas um último bloco, é permitida a simplificação das fronteiras da triangulação, pois deixam de existir dependências. Assim, a triangulação base final M^0 consiste em apenas 2 triângulos para cada bloco.

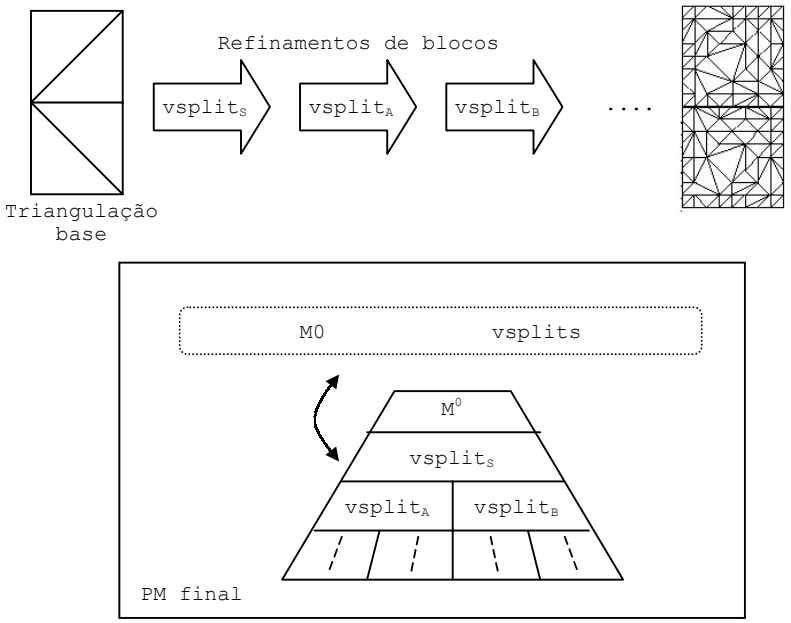


Figura 4-41: Resultado do processo de construção de uma PM hierárquica.

Como consequência de todo este processo, é criada uma PM hierárquica invertendo-se todas as operações `ecol` aplicadas. À sequência de registos `vsplit` resultante Hoppe designa por *block refinement*. Esta sequência é então guardada juntamente com a triangulação base[51] (ver Figura 4-41).

4.8.5 Vertex Morphing

Hoppe utilizou à técnica de *morphing* de vértices entre duas triangulações para eliminação de descontinuidades temporais durante a visualização.

Como foi descrito anteriormente, pretende-se mover gradualmente os vértices ao longo de várias *frame*s consecutivas, evitando a transição imediata (de uma *frame* para a seguinte) da posição inicial para a final. Esta técnica é aplicada apenas nos casos de refinamento e de generalização de vértices visíveis durante as *frame*s. No caso de vértices que deixam de ser visíveis ou de vértices que passam a ser visíveis a transição para a posição final é imediata.

Neste algoritmo, o *geomorphing* é aplicado não só à posição dos vértices em causa, mas também a outros atributos como a normal, a cor e as coordenadas da textura.

Formalmente, uma triangulação *geomorphed* $M^G(\alpha)$, onde $0 \leq \alpha \leq 1$, tal que $M^G(0)=M^i$ e $M^G(1)=M^{i+1}$, é a interpolação linear entre M^i e M^{i+1}.

[51] Esta operação envolve a re-enumeração dos vértices e dos parâmetros das faces em todos os registos `vsplit`.

5 Contribuição Científica

O estudo realizado nesta dissertação tem como objectivo analisar a aplicabilidade de *display lists* nos algoritmos de visualização em tempo real de terrenos de grandes dimensões.

Neste capítulo apresenta-se um novo algoritmo, desenvolvido no âmbito desta dissertação, que combina a utilização de multi-resolução contínua (MRC) com *display lists*.

5.1 Motivação

O crescente desenvolvimento das placas gráficas e o consequente melhoramento das suas capacidades não tem sido utilizada pelos algoritmos deste tipo [32]. Este foi o ponto de partida que delineou todo o trabalho desenvolvido nesta dissertação. A motivação foi tirar partido de algumas das características do *hardware* gráfico disponível no mercado, por forma a melhorar o desempenho dos algoritmos de visualização de terrenos de grandes dimensões. O facto de grande parte da triangulação se manter inalterada por várias *frames* consecutivas [8][23] permite evitar o processamento de toda e qualquer região do terreno cuja triangulação se mantenha igual à *frame* anterior. A solução encontrada foi a de guardar os blocos de terreno em *display lists* e, desta forma, optimizar o desempenho no processo de visualização dos mesmos. O algoritmo aqui proposto baseia-se no algoritmo de Röttger et al. [29] e combina a utilização de MRC com *display lists*.

À semelhança de Röttger et al. [29], a principal estrutura de dados sobre a qual assenta o algoritmo, e que representa a simplificação do terreno, é uma *quadtree*. No entanto, enquanto que Röttger et al. utiliza uma matriz booleana na sua implementação, neste algoritmo foi utilizada uma estrutura clássica com apontadores dada a necessidade de armazenamento de informação relativa aos blocos de terreno e à gestão das *display lists*.

Todo o processo é realizado em duas fases consecutivas, sendo primeiro calculada a simplificação do terreno com recurso a uma métrica dependente dos parâmetros da câmara, e de seguida efectuada a construção da triangulação resultante.

5.2 Construção da *Quadtree*

Tal como em Röttger et al. [29], o processo de construção da *quadtree* é *top-down* recursivo.

O algoritmo divide sucessivamente o terreno em vários blocos (regiões). Quanto maior o grau de divisão de um bloco, maior a resolução utilizada na região que ele representa, e logo melhor a aproximação efectuada para essa região. Por outro lado, o aumento de resolução conduz a um maior número de triângulos e logo a um menor desempenho do sistema gráfico.

Um bloco pode ser visto como um conjunto de 4 quadrantes de igual dimensão (ver Figura 5-1).

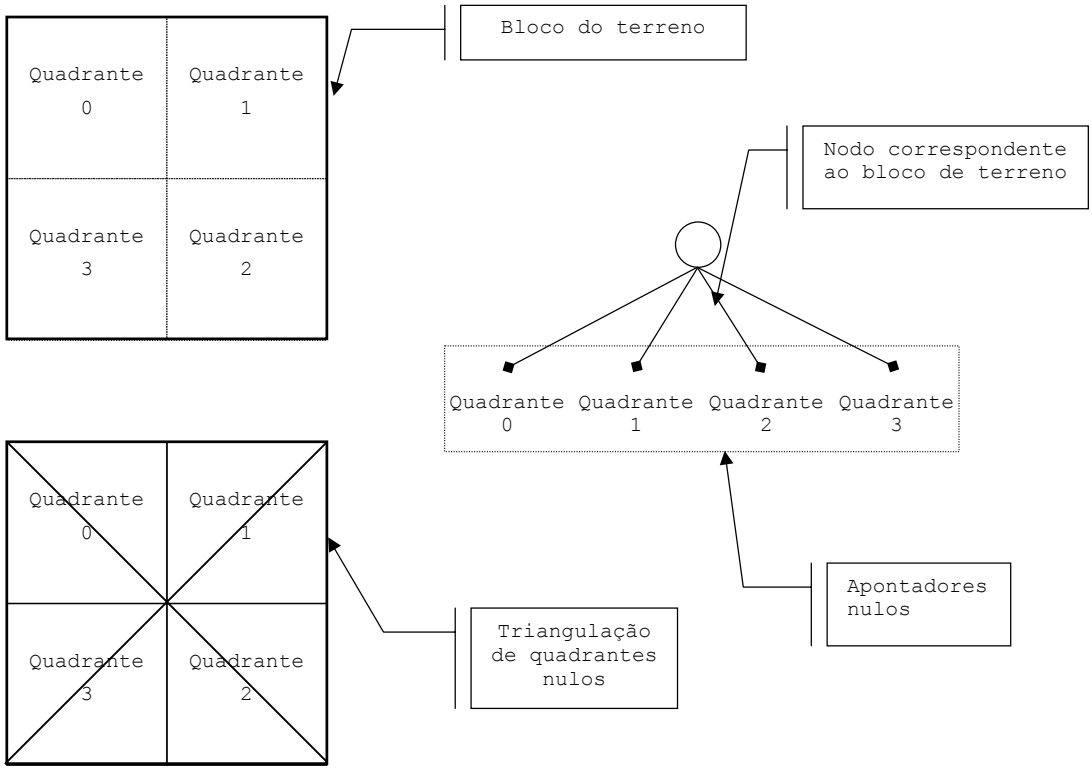


Figura 5-1: Representação de quadrantes que constituem um bloco de terreno e sua triangulação.

Cada um desses quadrantes é, por sua vez, um novo bloco que pode ser dividido em 4 novos quadrantes. Cada bloco (quadrante) é representado por um nodo da árvore quaternária.

No processo de construção da *quadtree*, cada um dos quatro quadrantes que constituem o bloco é avaliado no sentido de se determinar se deve ou não ser dividido. Em caso afirmativo, o

processo é recursivamente aplicado aos seus quadrantes (ver Figura 5-2). A recursividade termina quando o teste é negativo.

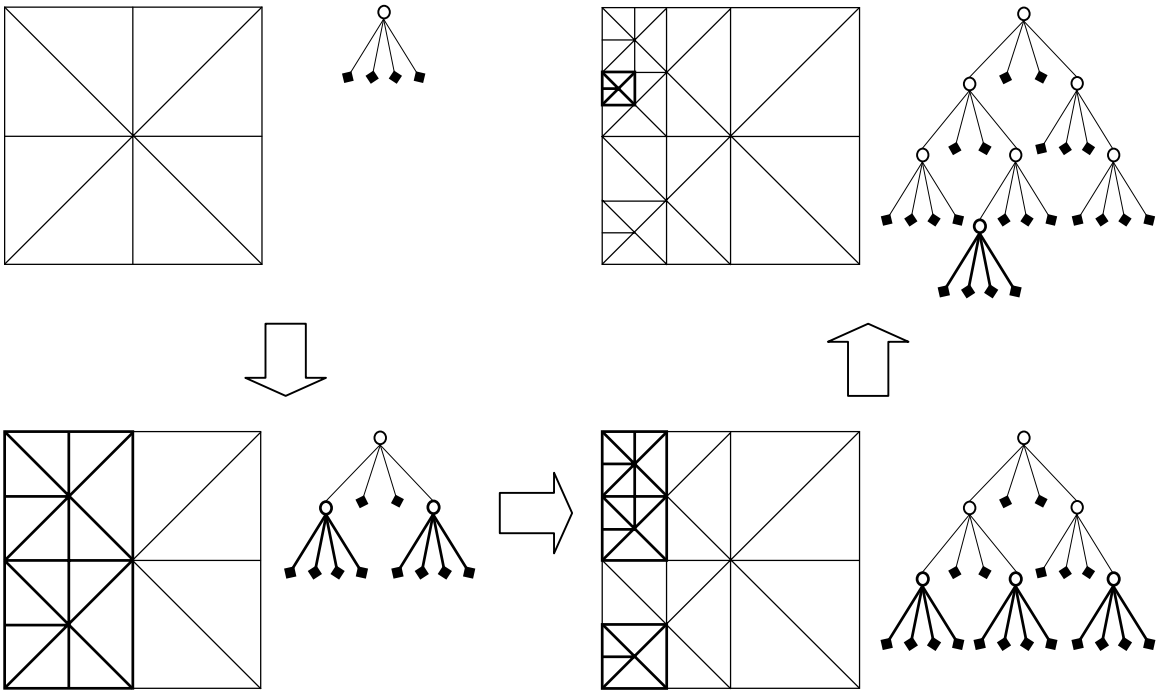


Figura 5-2: Exemplo de uma *quadtree* e os correspondentes blocos do terreno.

Em todas as *frames*, o processo recursivo inicia-se na raiz da *quadtree*, que representa o terreno na totalidade, e percorre a árvore no sentido de a actualizar de acordo com a métrica definida.

5.2.1 Display Lists Hierárquicas

No algoritmo aqui apresentado as *display lists* são utilizadas para auxiliar a visualização da triangulação que resulta do processo de simplificação do terreno.

Como foi mencionado na secção *2.1.1 – A Fase de Aplicação*, o objectivo inerente ao conceito de *display list* é a utilização de um mecanismo que guarde informação que se mantém inalterada pelo máximo de tempo possível, permitindo diluir no tempo o impacto criado pela sobrecarga de criar, gerir e remover as *display lists*. Desta forma, é possível tirar partido das vantagens, em termos de desempenho, da utilização das mesmas.

O algoritmo cria uma hierarquia de *display lists*, utilizando uma *display list* para cada nodo da *quadtree*, com excepção dos nodos folha (ver Figura 5-3). Cada *display list* representa a região de terreno correspondente ao nodo a que está associada e pode conter:

- Apenas informação relativa à triangulação;

- Apenas informação relativa à hierarquia, isto é, apenas referências para as *display lists* dos nodos descendentes;
- Ambas as anteriores, quando existem *display lists* nos nodos descendentes e quando pelo menos um dos quadrantes do nodo associado é nulo ou folha.

A razão para a excepção dos nodos folha é evitar a criação de *display lists* com pouca informação. Se os nodos folha tivessem uma *display list* associada teríamos uma *display list* para cada leque de triângulos.

Por outro lado, à medida que o nível da *quadtree* a partir do qual as *display list* fossem utilizadas fosse aumentando, a frequência de actualização das mesmas também aumentaria. Nestes casos, as *display lists* conteriam mais informação e representariam regiões do terreno de dimensão considerável, que seriam alvo de frequentes modificações nas várias sub-regiões que a constituem. Tal facto implicaria, na prática, actualizações constantes de toda a *display list*, ainda que um número significativo de partes desta não tivessem sofrido alteração.

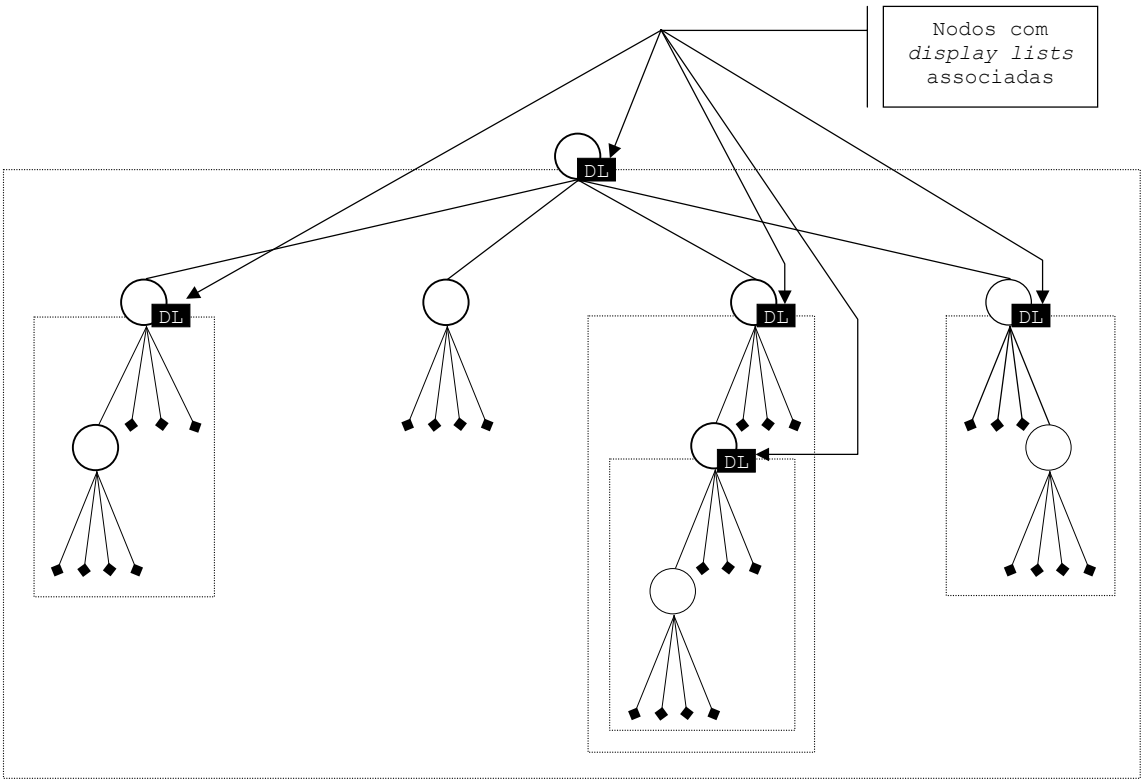


Figura 5-3: Hierarquia de *display lists*.

Neste sentido, o compromisso assumido foi o de não utilizar *display lists* para os nodos folha da *quadtree*, representando cada uma das *display lists* de nível inferior um mínimo de 8 vértices (8 triângulos) correspondente a 2 leques, e um máximo de 25 vértices (32 triângulos)

correspondente a 4 leques completos (ver Figura 5-4). Nos testes efectuados cada *display list* representa, em média, 15 triângulos.

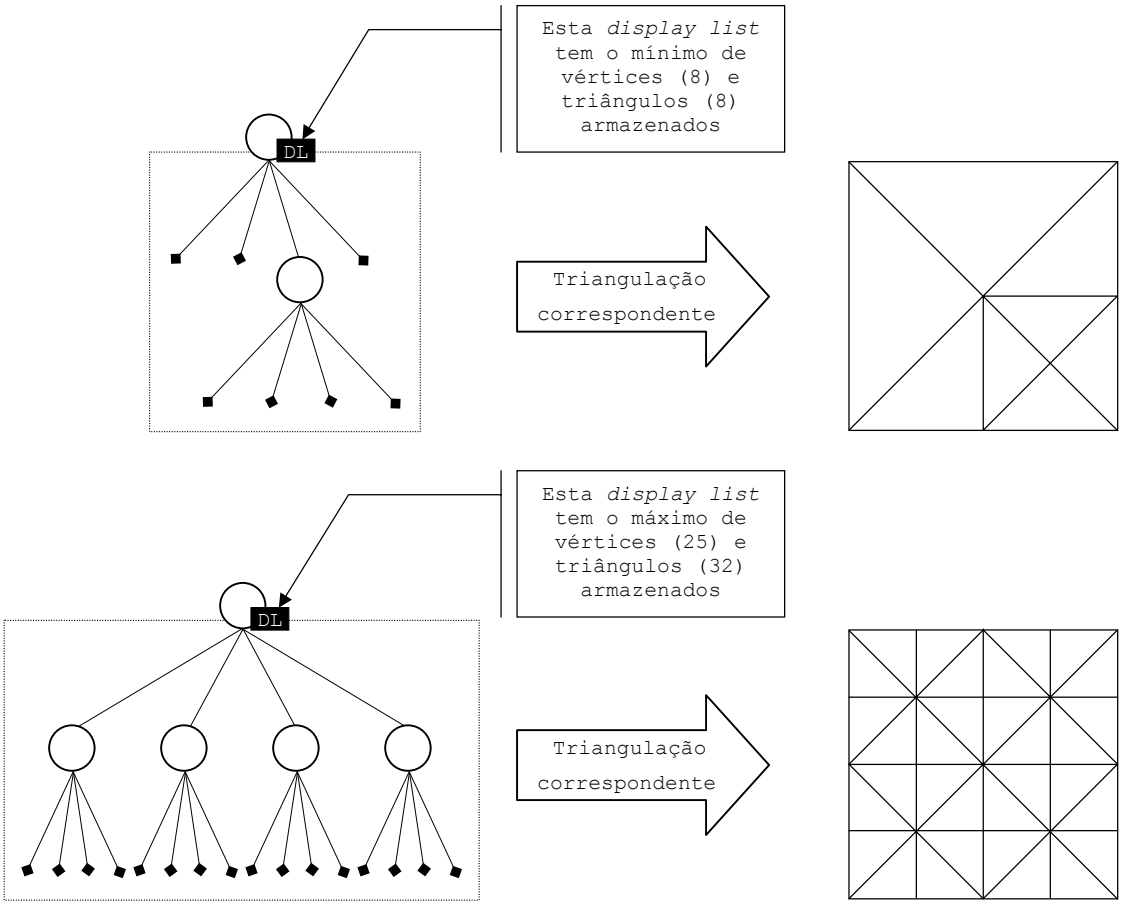


Figura 5-4: Mínimo e máximo de triângulos e vértices representados por uma *display list* de nível inferior.

A hierarquia de *display lists* é definida por todos os nodos da *quadtree* que têm filhos com descendência. Todas as *display lists* associadas a estes nodos invocam as *display lists* associadas aos seus nodos filho, caso existam.

Para além da hierarquia, estas *display lists* armazenam ainda as triangulações respeitantes às regiões do terreno correspondentes aos nodos filho que não têm *display lists* associadas. Estes são todos os filhos nulos e os filhos folha.

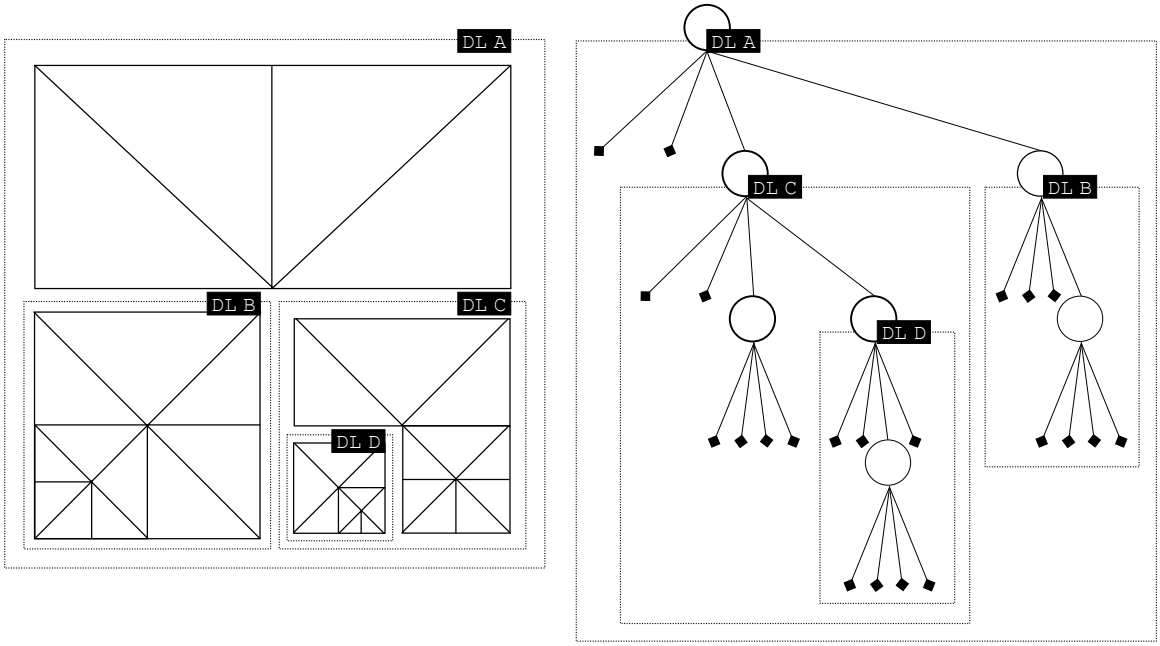


Figura 5-5: Hierarquia de *display lists*.

Considere-se a Figura 5-5. A *display list* A, associada à raiz da *quadtree*, invoca as *display lists* B e C correspondentes aos nodos filho da raiz que têm descendência, definindo assim parte da hierarquia. Esta *display list* define ainda as triangulações para os filhos nulos da raiz que, consequentemente, não são abrangidos por nenhuma outra *display list*.

Por sua vez, a *display list* D não invoca nenhuma *display list* pois todos os filhos do nodo da *quadtree* a que está associada são nulos ou nodos folha e logo não são representados por nenhuma *display list*. Desta forma, a *display list* D define apenas as triangulações para os todos esses filhos.

A utilização de uma hierarquia permite a modificação de diferentes sub-regiões do terreno, isto é, permite redefinir partes da hierarquia, sem implicar alterações nas restantes. Considere-se novamente a Figura 5-5. Qualquer alteração a um dos filhos do nodo a que está associada a *display list* C não implica uma actualização da *display list* A, no nível imediatamente superior na hierarquia, mas apenas a actualização da própria *display list* C.

5.2.2 As Operações

O processo de construção da *quadtree* é *top-down* recursivo e, conceptualmente, é constituído por uma sequência de operações de refinamento. Nesta operação são criados os nodos necessários de forma a obter um erro inferior a um determinado limite.

No entanto, por motivos de desempenho, o processo parte da *quadtree* da *frame* anterior, dado ser mais vantajoso alterar a *quadtree* existente do que remover essa *quadtree* na totalidade e

criar uma nova. Sendo assim, após o refinamento, é necessário remover da árvore os nodos inseridos previamente em *frames* anteriores.

Criação de um Nodo

Quando no processo recursivo se conclui que, para uma dada região, é necessário refinar, isto é, utilizar uma triangulação de maior resolução que produza um erro final menor, é criado na *quadtree* um novo bloco para representar a região em causa.

À criação deste novo bloco corresponde a criação[52] de um novo nodo na *quadtree*, filho do nodo correspondente à região cujos quadrantes estão a ser avaliados (ver Figura 5-6).

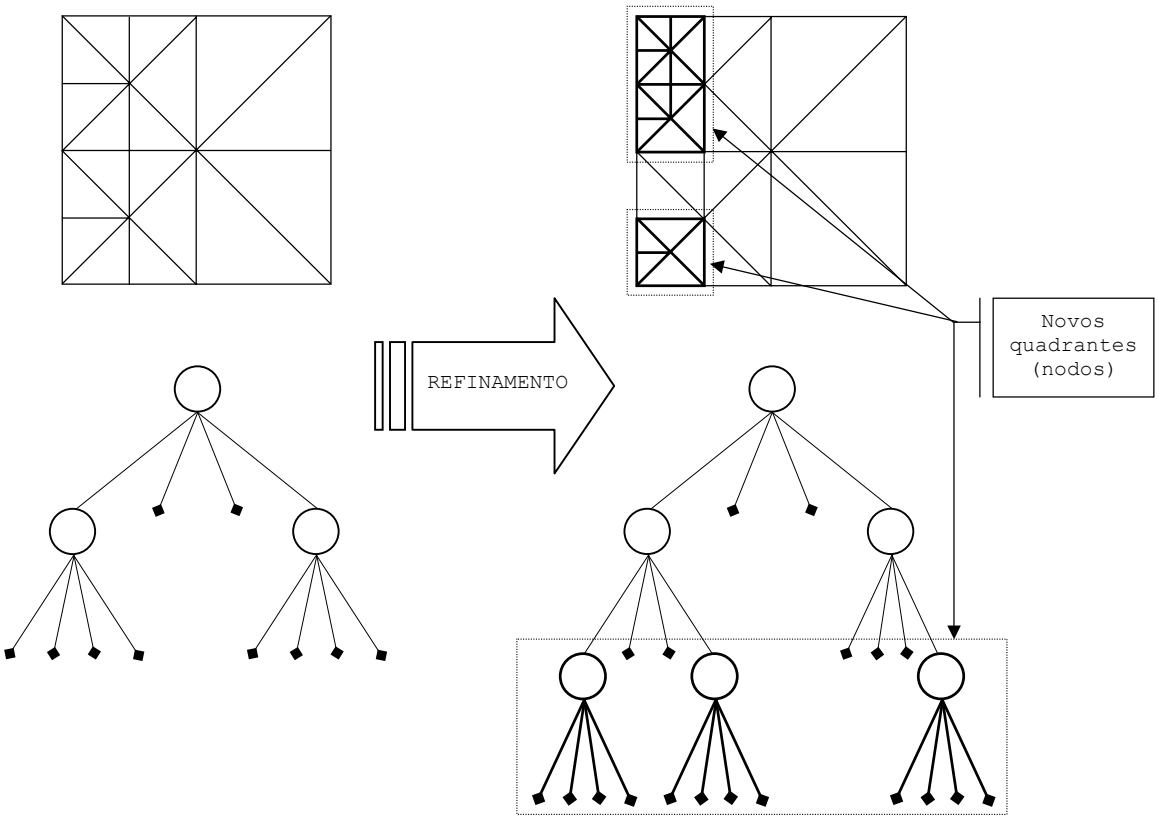


Figura 5-6: Refinamento da *quadtree*.

[52] Em termos da implementação, nem sempre é realizada a criação de um novo nodo pois é sempre utilizada a *quadtree* da *frame* anterior, na qual o nodo em causa já poderá existir.

Qualquer alteração à *quadtree* implica a actualização das *display lists* correspondentes para que estas representem a nova triangulação. Este processo de actualização das *display lists* é realizado em duas fases:

1. Sinalização dos nodos da *quadtree* directamente implicados na operação efectuada e que sofreram alterações.
2. Actualização propriamente dita de todas as *display lists* associadas aos nodos sinalizados.

Neste sentido, para manter consistente a hierarquia de *display lists*, sempre que um novo nodo é criado, são sinalizados:

- O bloco pai do nodo criado, pois a *display list* deste nodo deve ser criada, se ainda não existir, ou actualizada caso exista.

  Considere-se a Figura 5-7. A criação do nodo n_c implica a sinalização do nodo pai n_p para que a *display list* D a ele associada seja posteriormente criada.

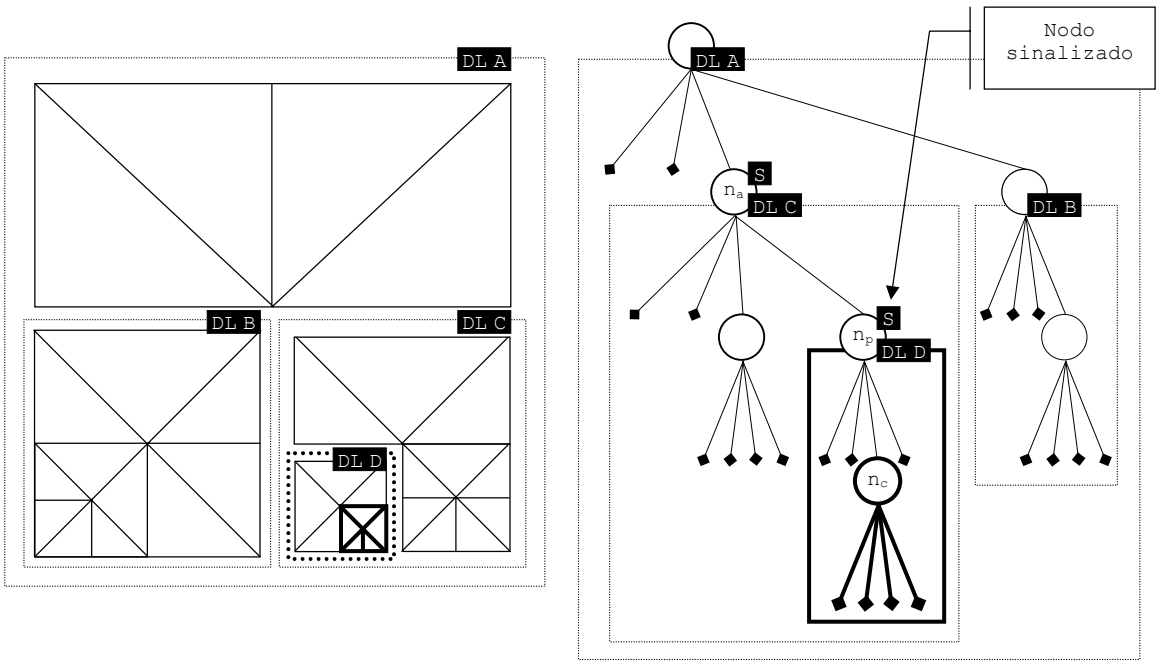


Figura 5-7: Sinalização dos nodos decorrente da operação de refinamento.

- O bloco avô do nodo criado, se e só se o bloco pai era, na *frame* anterior, uma folha da *quadtree*. Nesta situação, a *display list* do bloco pai deve ser criada e a do bloco avô deve ser modificada por forma a invocar a *display list* do bloco pai.

  Considere-se novamente a Figura 5-7. Antes do refinamento, o nodo pai n_p era um nodo folha, pelo que, para além da sinalização do nodo pai n_p referida no ponto

anterior, o nodo avô n_a deve ser sinalizado para que a *display list* C a ele associada seja posteriormente actualizada, por forma a invocar a recém criada *display list* D.

- Os blocos adjacentes ao quadrante criado, se e só se são de nível inferior e não são folhas da *quadtree*. Caso contrário são sinalizados os pais destes.

  Os blocos adjacentes podem sofrer alterações na sua triangulação causadas pela alteração da triangulação dos seus vizinhos. Estas alterações são necessárias para se evitarem descontinuidades espaciais.

  Para evitar uma travessia à *quadtree* e acelerar a procura dos blocos adjacentes, recorreu-se a uma matriz de apontadores, de dimensão igual à do terreno, que é actualizada sempre que se efectuam alterações à *quadtree*.

  Considere-se a Figura 5-8. A criação do nodo n_c implica a sinalização do nodo adjacente n_d para que a *display list* D a ele associada seja actualizada (desta forma evita-se uma descontinuidade espacial).

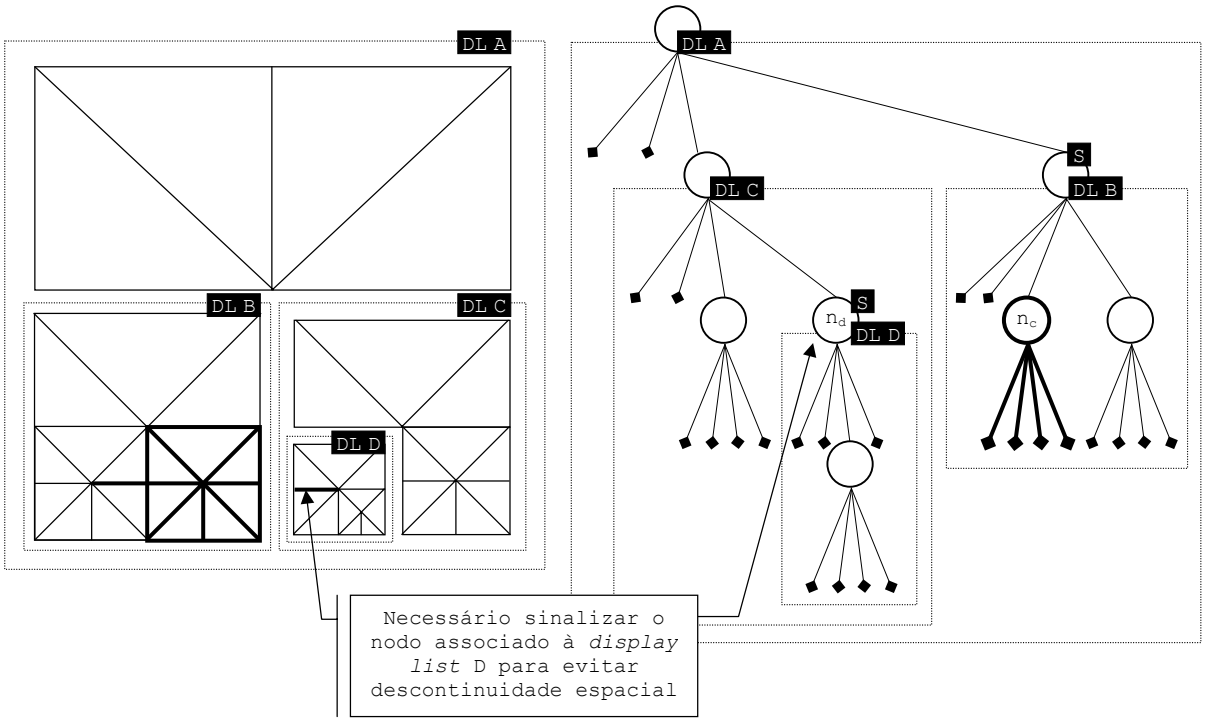


Figura 5-8: Sinalização dos blocos adjacentes no decorrer da operação de refinamento.

Remoção de nodos

Quando no processo recursivo se conclui que, para uma dada região, não é necessário refinar, o processo recursivo termina para essa região. Como foi mencionado anteriormente, na prática, o

processo de construção da *quadtree* parte da *quadtree* da *frame* anterior, dado ser mais vantajoso, em termos de desempenho, efectuar as alterações (criações e remoções de nodos) necessárias à *quadtree* existente do que remover essa *quadtree* na totalidade e criar uma nova. Por este motivo, quando o processo recursivo termina para uma região, é necessário remover da *quadtree*, caso existam, o nodo e todos os descendentes que a ela correspondem, bem como as *display lists* a eles associadas (ver Figura 5-9).

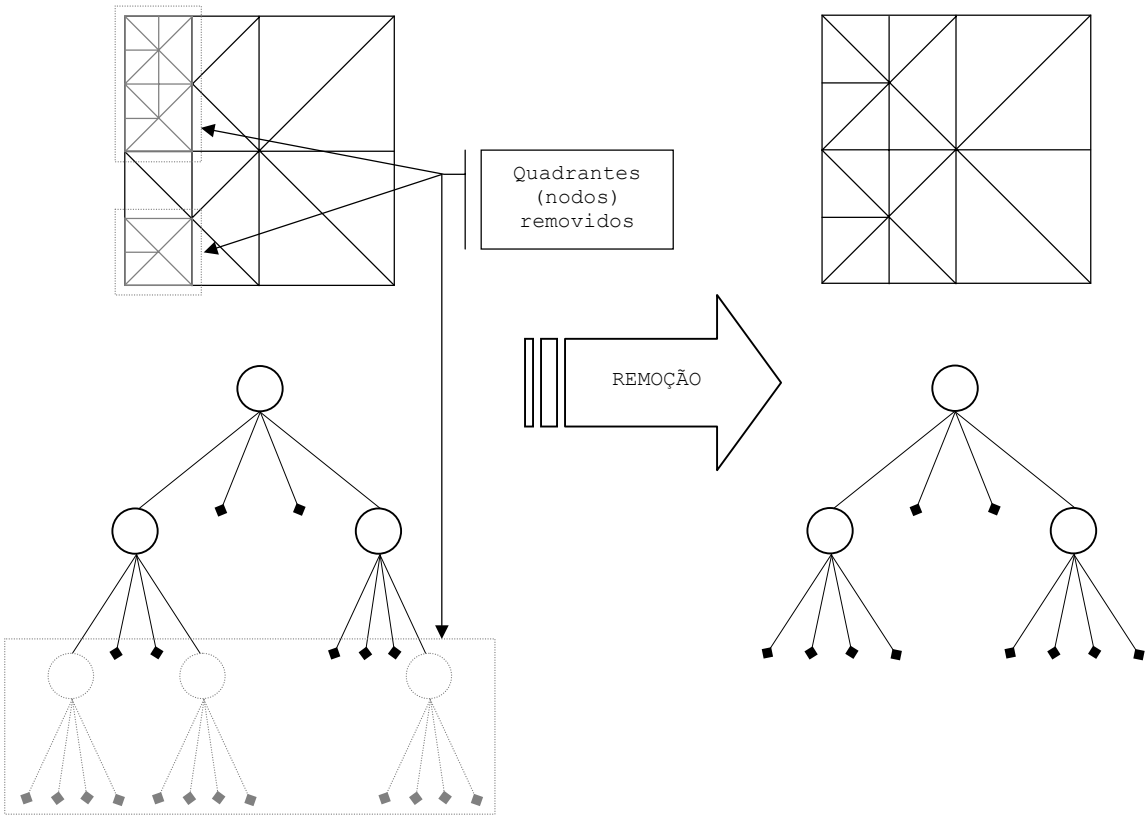


Figura 5-9: Remoção de nodos da *quadtree*.

Para manter consistente a hierarquia de *display lists*, é necessário sinalizar:

- O bloco pai do nodo removido, se e só se este não é um nodo folha da *quadtree* após esta operação. A *display list* associada ao bloco pai deve ser actualizada por forma a incluir a triangulação respeitante ao quadrante cujo nodo foi removido.

  Considere-se a Figura 5-10. A remoção do nodo n_r (e a consequente remoção dos nodos descendentes e *display lists* associadas) implica a sinalização do nodo pai n_p para que a *display list* C a ele associada seja actualizada.

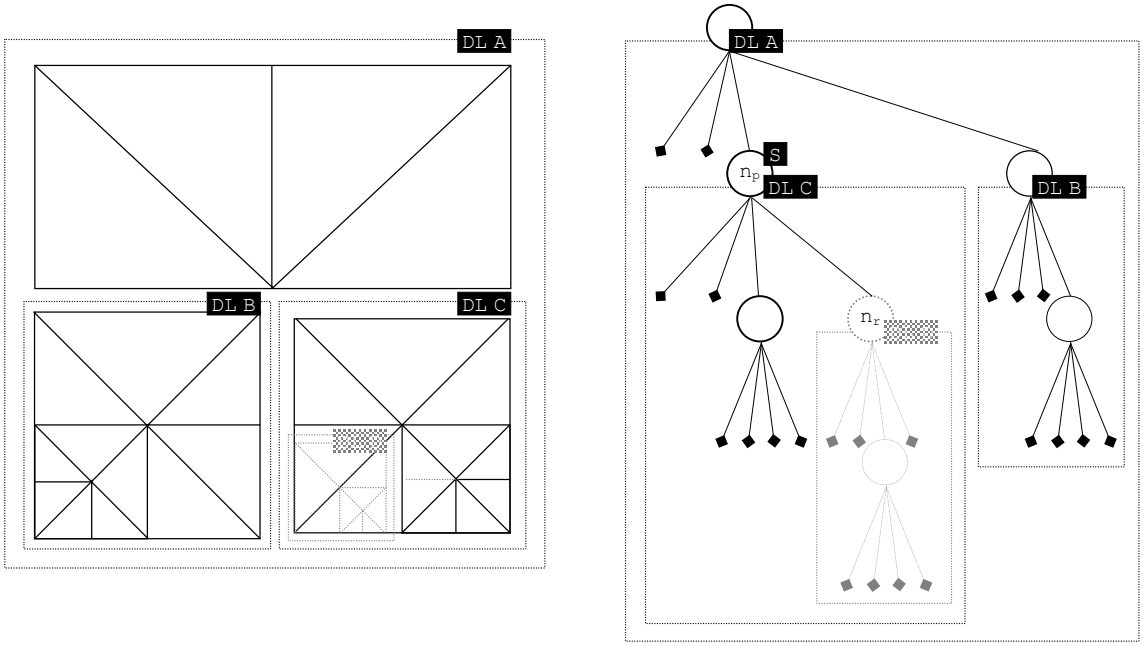


Figura 5-10: Sinalização dos nodos decorrente da remoção de nodos.

- O bloco avô do nodo removido, se e só se o bloco pai é um nodo folha da *quadtree* após esta operação. Nesta situação a *display list* associada ao bloco pai deve ser removida, se existir, e a *display list* associada ao nodo avô deve ser actualizada por forma a incluir a triangulação do bloco pai (que é, após a remoção, um nodo folha).

  Considere-se a Figura 5-11. A remoção do nodo n_r transforma o nodo pai n_p num nodo folha. Consequentemente a *display list* D associada a n_p é removida e o nodo avô n_a é sinalizado para que a *display list* C a ele associada seja posteriormente actualizada. Desta fará parte o leque correspondente a triangulação de n_p.

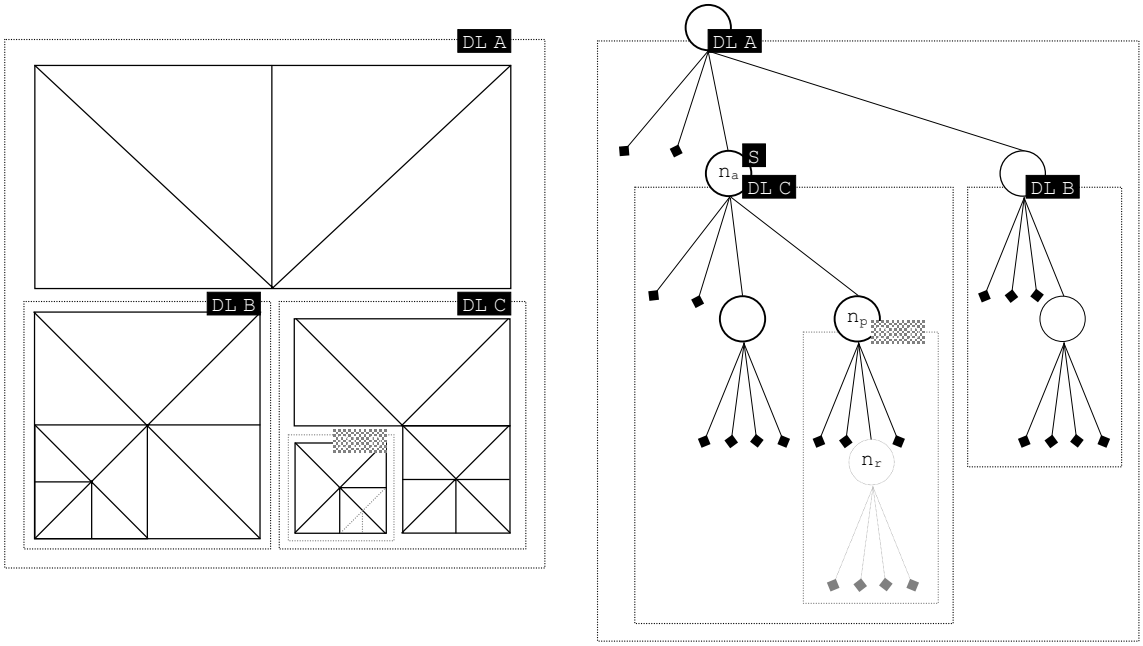


Figura 5-11: Sinalização dos nodos decorrente da operação de remoção de nodos (2ª situação).

- Os blocos adjacentes ao quadrante removido, se e só se não são folhas da *quadtree*. Caso contrário são sinalizados os pais destes. Como foi mencionado anteriormente, os blocos adjacentes podem sofrer alterações na sua triangulação causadas pela alteração da triangulação dos seus vizinhos. Estas alterações são necessárias para se evitarem descontinuidades espaciais.

  Considere-se a Figura 5-12. A remoção do nodo n_r implica a sinalização do nodo adjacente n_d para que a *display list* D a ele associada seja actualizada, evitando-se uma descontinuidade espacial.

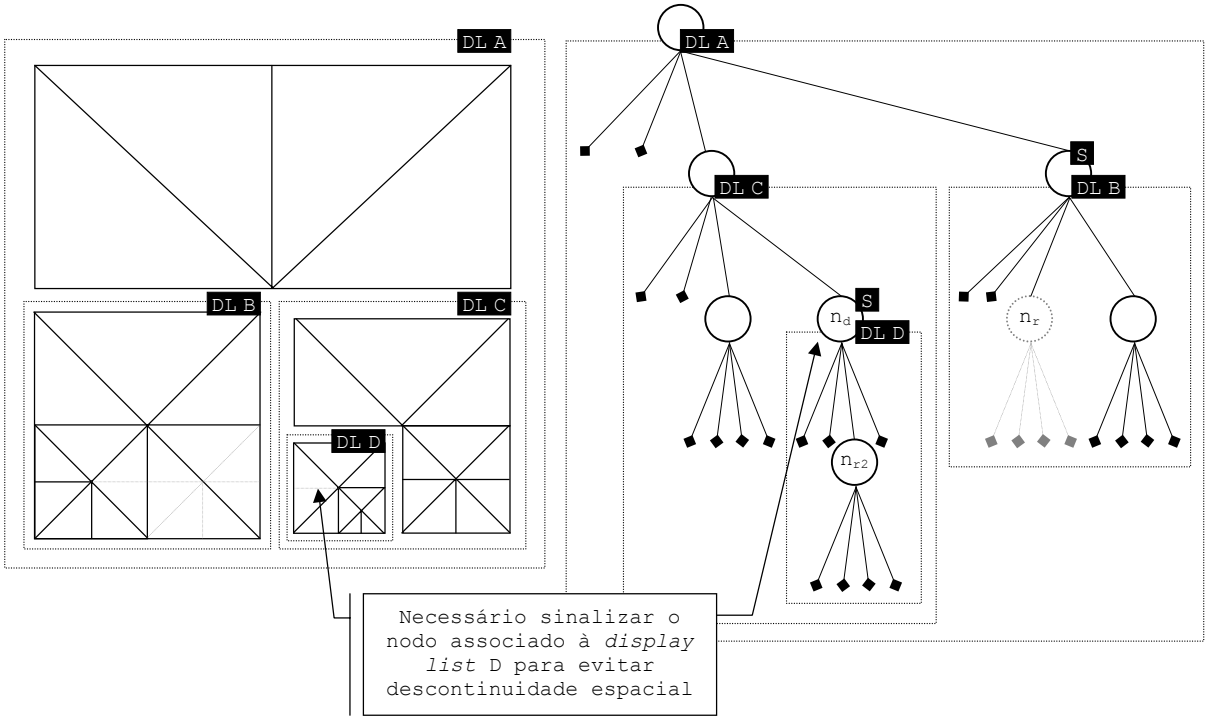


Figura 5-12: Sinalização dos blocos adjacentes no decorrer da operação de remoção de nodos.

5.3 A Métrica

O algoritmo proposto neste capítulo é independente da métrica utilizada. No entanto, tal como em [29], terá de se garantir que a diferença de nível entre blocos adjacentes nunca é superior a 1 por forma a se evitarem descontinuidades espaciais. Esta garantia pode ser uma consequência directa da métrica, como acontece em [29], ou de um outro mecanismo independente.

No sentido de estabelecer uma comparação objectiva entre o algoritmo desenvolvido nesta dissertação e o algoritmo de Röttger et al. [29], foi utilizada exactamente a mesma métrica no processo de decisão relativamente à operação a aplicar sobre cada nodo da *quadtree*.

Para uma descrição detalhada da mesma consultar secção *4.4 – Real-Time Generation of Continuous Level of Detail for Height-fields*.

5.4 Actualização das *Display Lists*

Após a construção/actualização da *quadtree* e a consequente sinalização dos blocos alterados, as *display lists* associadas a cada um destes blocos necessitam de ser actualizadas.

Neste sentido, é efectuada uma nova travessia da *quadtree* para encontrar os blocos sinalizados (ver Figura 5-13) e, para cada um deles, é actualizada a respectiva *display list* de acordo com a situação actual da *quadtree*.

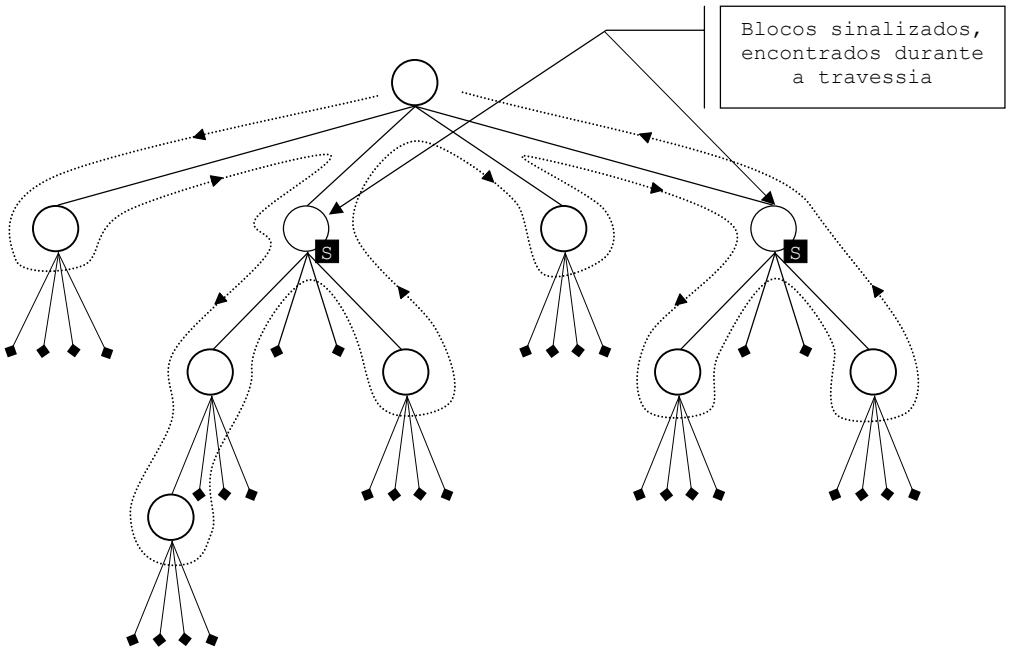


Figura 5-13: Travessia da *quadtree* para actualização das *display lists*.

Estas actualizações são efectuadas numa nova travessia devido à existência de sinalização de blocos vizinhos. Caso contrário diversas *display lists* seriam, eventualmente, actualizadas mais do que uma vez no processo de construção da triangulação de uma dada *frame*, e poderiam mesmo ser removidas após algumas dessas actualizações.

Considere-se novamente a Figura 5-12. Se as *display lists* fossem imediatamente actualizadas, a remoção do nodo n_r implicaria a actualização da *display list* D associada ao nodo adjacente n_d que não seria necessária, se o nodo n_{r2} fosse posteriormente removido.

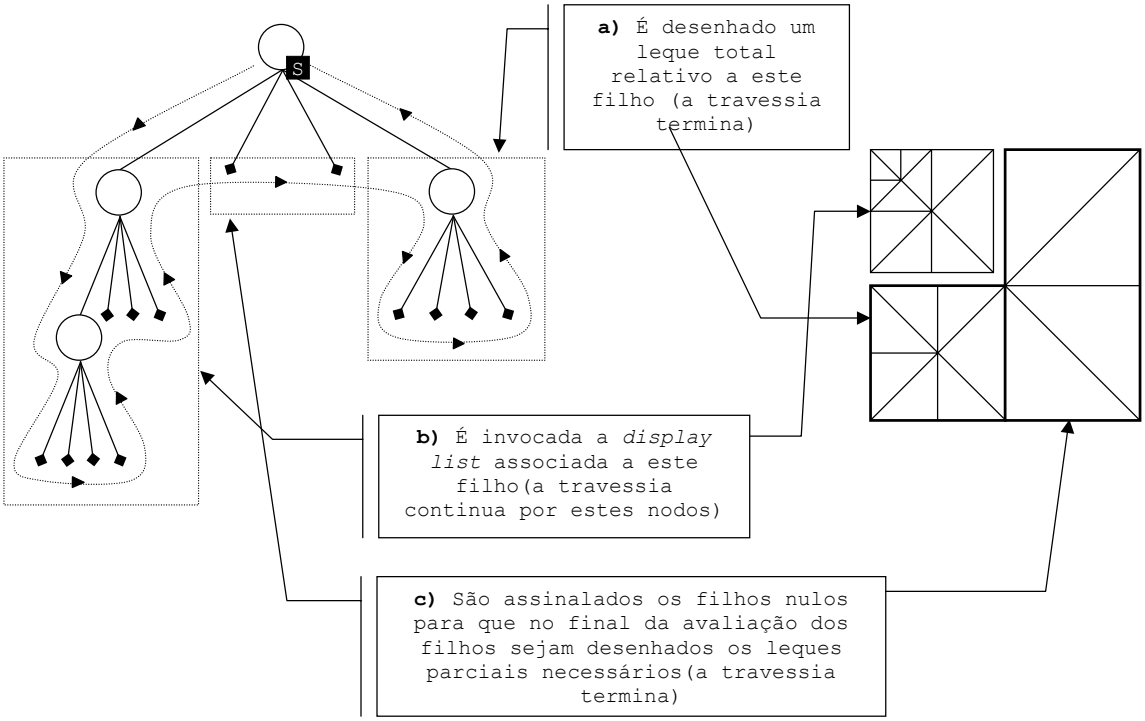


Figura 5-14: Actualização da *display list* de um nodo sinalizado.

O processo de construção e/ou actualização das *display lists* é o mesmo para todos os nodos sinalizados (ver Figura 5-14). A *display list* associada a cada um deles é criada (se ainda não existe) ou actualizada (se existe)[53] e cada um dos nodos filho é avaliado:

- Se não é nulo:
  - E é folha, é armazenado na *display list* um leque total correspondente a essa região do terreno (ver Figura 5-14 a)). Omitem-se os vértices nas arestas comuns a nodos adjacentes de nível superior a fim de se evitarem descontinuidades espaciais (ver Figura 5-12).
  - E não é folha, é invocada a *display list* correspondente a esse nodo, a qual é criada vazia, isto é, sem informação alguma, caso ainda não exista (ver Figura 5-14 b)).
- Se é nulo, a região do terreno correspondente a este filho é representada por um leque parcial. No final da avaliação de todos os nodos filho, os leques correspondentes aos nodos filho nulos são armazenados na *display list* após terem sido fundidos (caso sejam adjacentes) (ver Figura 5-14 c)). Desta forma o número

[53] O processo de actualização e criação de novas *display list* em *OpenGL* é exactamente o mesmo. O mesmo mecanismo que permite criar uma *display list* permite redefini-la no caso de esta já existir.

máximo de leques parciais utilizados por *display list* é de apenas 2. Também aqui são omitidos os vértices nas arestas comuns a nodos adjacentes de nível superior a fim de se evitarem descontinuidades espaciais.

De notar que nesta fase não é necessário remover qualquer *display list* pois tais operações foram realizadas aquando do processo de actualização da *quadtree*.

5.5 O Algoritmo

Após a actualização/construção da *quadtree* e a construção da triangulação com recurso a *display lists* hierárquicas, a fase de visualização da triangulação resume-se à chamada da função que visualiza a *display list* raiz da hierarquia. Esta *display list* corresponde ao nodo raiz da *quadtree* e logo ao terreno completo.

O algoritmo seguinte resume tudo o que foi mencionado anteriormente:

```
Actualiza Quadtree (q)
     Para cada quadrante c de q
          Se dentro do volume de visualização e se deve ser refinado
               Se existe c na quadtree
                    Se c não está refinado no máximo
                         Actualiza Quadtree(c)
               Senão
                    Se q não está refinado no máximo
                         Cria quadrante c
                         Sinaliza alterações no nodo pai de q se q era
                               folha antes da criação do quadrante c
                         Actualiza Quadtree(c)
          Senão
               Se existe c na quadtree
                    Remove c, todos os descendentes e DLs associadas
                    Sinaliza alterações cf. descrito na operação de
                         refinamento
     Fim do Para
Fim da Função
```

Figura 5-15: Função de actualização da triangulação.

```
Actualiza DLs
     Para todos os q sinalizados da quadtree
          Redefine DL de q
          Para todos os quadrantes c de q
               Se existe c
                    Se c é folha
                         Desenha o leque total
                    Senão
                         Chama DL associada a c
               Senão
                    Assinala quadrante para render
          Fim do para
          Desenho dos leques dos quadrantes assinalados
     Fim do Para
Fim da Função
```

Figura 5-16: Função de actualização das *display lists*.

```
Main
     // Inicialização
     Alocar recursos de memória
     Calcular Irregularidade do Terreno
     Criar raiz da quadtree r
     // Ciclo
     Enquanto não termina
          Actualiza Triangulação(r)
          Actualiza DLs
          Invoca DL de r
     Fim do Enquanto
Fim
```

Figura 5-17: Algoritmo final.

5.6 Variação com *Display Lists* Simples

Ainda que a utilização de *display lists* hierárquicas permita simplificar a fase de visualização do algoritmo, esta solução implica a utilização e manutenção desnecessárias de *display lists*. De facto, parte das *display lists* utilizadas na versão anterior tinham como único objectivo manter a estrutura hierárquica sem armazenar qualquer informação relativa à triangulação a visualizar.

Neste sentido, implementou-se uma variação do algoritmo inicial que utiliza apenas *display lists* simples e cujos nodos associados da *quadtree* se encontram ligados por uma lista duplamente ligada.

Esta segunda versão do algoritmo caracteriza-se fundamentalmente pela eliminação da hierarquia de *display lists*. Ao contrário da primeira versão, nesta versão as *display lists* são utilizadas exclusivamente para armazenamento da triangulação, não contendo nenhuma invocação a outra *display list*. Só nodos com pelo menos um filho nulo ou um filho folha têm uma *display list* associada. Mantém-se a condição de nodos folha não terem nenhuma *display list* associada.

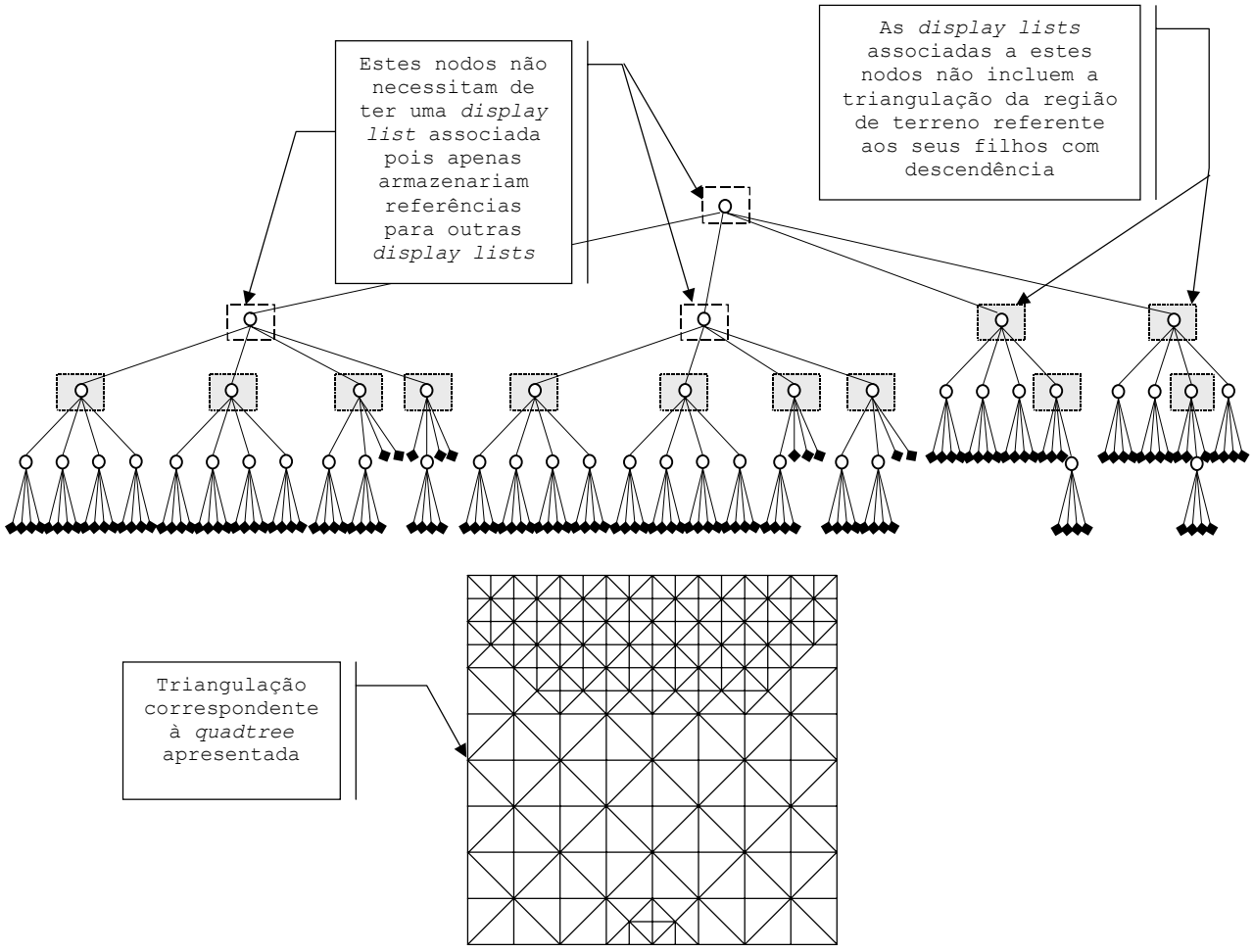


Figura 5-18: Eliminação da hierarquia de *display lists*.

Considere-se a Figura 5-18. Todos os nodos assinalados a cinzento têm uma *display list* associada na qual é armazenada exclusivamente a triangulação referente à região em causa. Esta triangulação diz respeito unicamente a filhos nulos e a filhos folha. A triangulação referente a

filhos com descendência não é incluída nesta *display list* mas em *display lists* associadas a estes filhos e/ou aos seus descendentes.

Os nodos assinalados a branco não têm nenhuma *display list* associada, pois não estão nas condições enunciadas anteriormente. Estas apenas armazenariam referências a outras *display lists*, dado que as áreas de terrenos que representam são cobertas na totalidade pelas *display lists* dos seus descendentes.

Todos os nodos com *display lists* associadas são inseridos numa lista duplamente ligada por forma a acelerar a fase de actualização das *display lists* e de visualização da triangulação (ver Figura 5-19). A segunda travessia à *quadtree*, que era efectuada nesta fase e na qual eram visitados a totalidade dos nodos da *quadtree* (excepto os nodos folha), é substituída pela travessia a esta lista duplamente ligada, durante a qual são actualizadas todas as *display lists* associadas a nodos sinalizados e são visualizadas todas as *display lists* de todos os nodos contidos na lista.

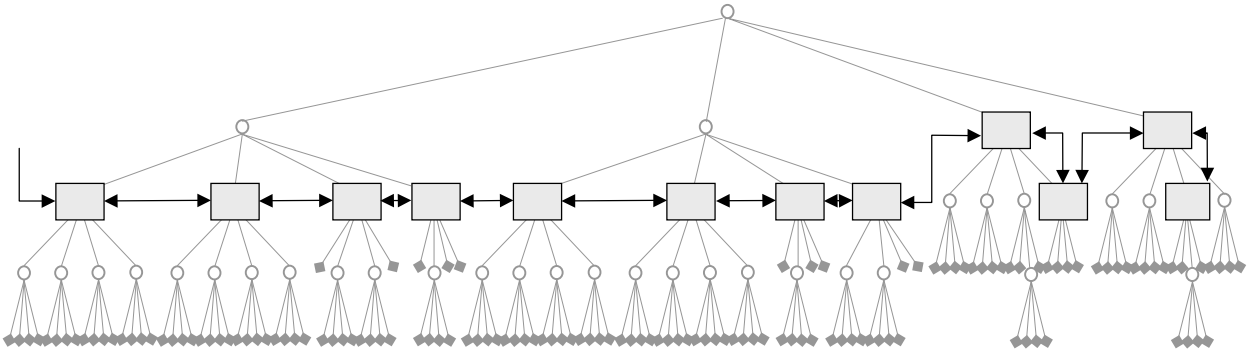


Figura 5-19: Lista duplamente ligada de nodos com *display lists*.

5.6.1 Actualização da Quadtree

Tal como no algoritmo original é efectuada, em todas as *frames*, uma travessia à *quadtree* com o objectivo de a actualizar.

Quando um nodo é visitado todos os seus filhos são avaliados e, recursivamente, todos os seus descendentes são actualizados. Sempre que necessário são criados novos nodos e removidos nodos existentes (Figura 5-20).

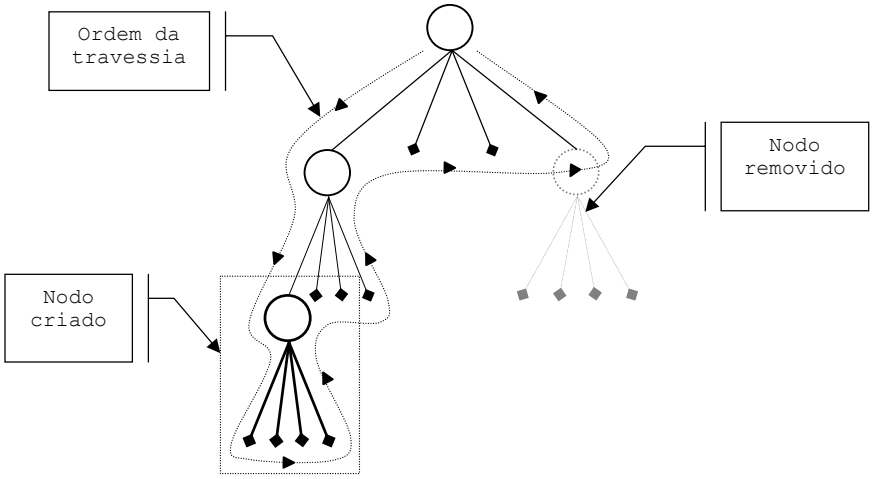


Figura 5-20: Travessia para actualização da *quadtree*.

Durante este processo de actualização, são associados a cada nodo quatro indicadores de filhos e quatro de actualização, um por cada filho, para auxiliar na sinalização de nodos e na inserção e remoção de nodos na lista duplamente ligada.

Durante a visita a cada nodo os indicadores a ele associados são actualizados:

- Os indicadores de filhos são actualizados de acordo com a situação final (após a actualização) do filho em causa, assumindo os valores 0 se o filho não existe, 1 se o filho é folha e 2 se o filho tem descendência.
- Os indicadores de actualização são activados se ocorrerem alterações no filho em causa, nomeadamente se este tiver sido criado, removido, se era folha e deixou de o ser, ou se não era folha e passou a ser (ver Figura 5-21).

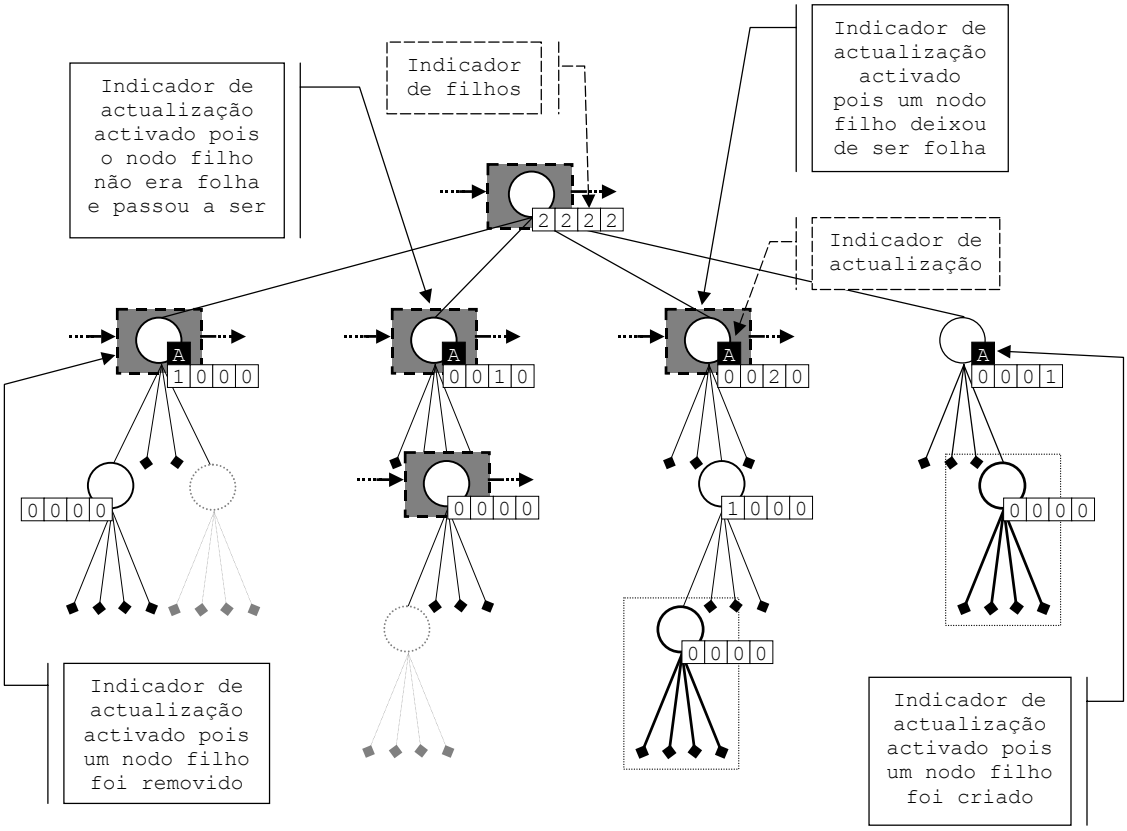


Figura 5-21: Indicadores associados a um nodo.

No final da avaliação de um nodo (de todos os filhos, e consequentemente de todos descendentes) a sua situação é reavaliada. São analisados os indicadores de filhos e de actualização a ele associados, que foram actualizados durante a travessia.

Se o nodo é, no final, um nodo folha (isto é, todos os indicadores de filhos têm o valor 0) ou se apenas tem filhos não folha (isto é, todos os indicadores de filhos têm o valor 2) este não deve ter uma *display list* associada e logo, caso o nodo pertença a lista duplamente ligada, este é removido da mesma e a *display list* a ele associada é removida (ver Figura 5-22 a) e b) respectivamente).

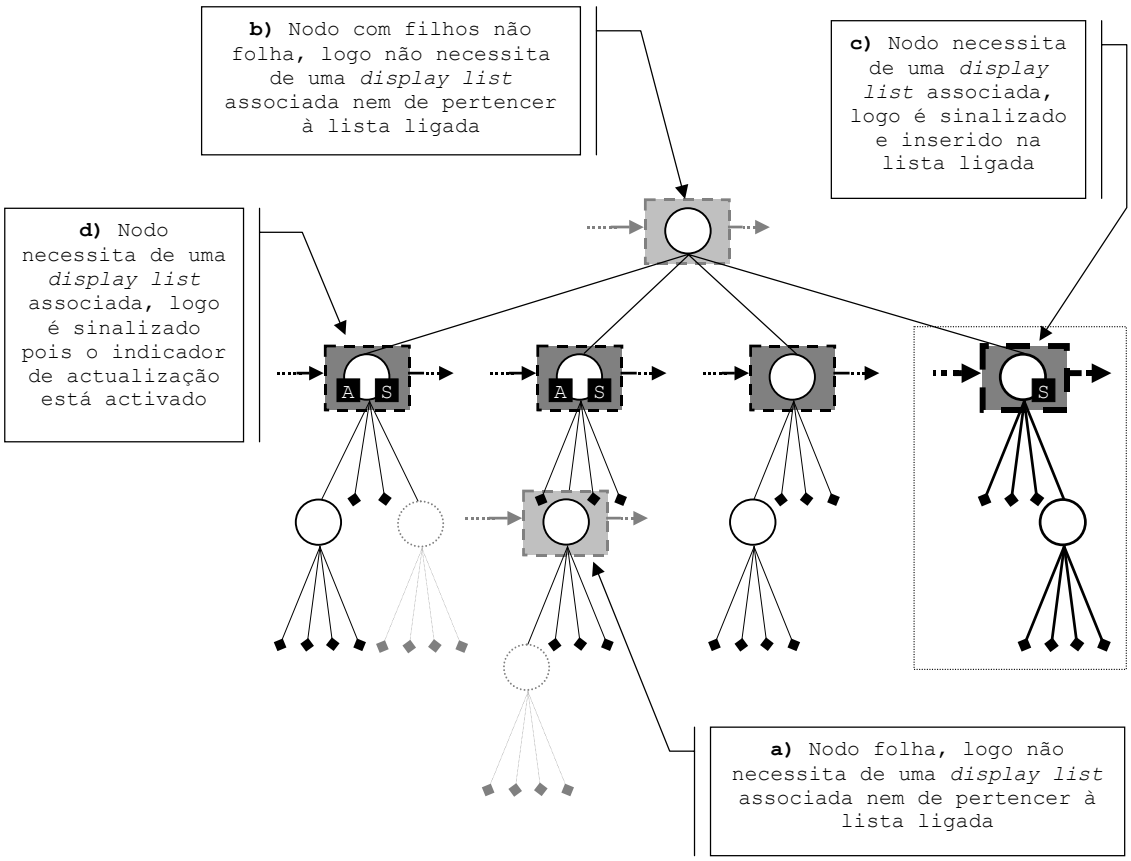


Figura 5-22: Nodo sem *display list* associada.

Caso contrário, o nodo deve ter uma *display list* associada e logo:

- Se o nodo não pertence à lista duplamente ligada este é inserido e é sinalizado para que a *display list* a ele associada seja actualizada ou criada durante a fase seguinte (ver Figura 5-22 c)).
- Se o nodo já pertence à lista duplamente ligada este é sinalizado se e só se pelo menos um dos indicadores de actualização estiver activado (o que significa que houve alterações relevantes nos seus descendentes) (ver Figura 5-22 d)).

Dada a natureza sequencial da visita aos nodos da lista duplamente ligada, toda e qualquer inserção de um nodo nesta é efectuada no início da lista afim de evitar duplas visitas a um nodo.

Todo o processo de construção/actualização da *quadtree* é descrito no algoritmo da Figura 5-23.

```
Actualiza Quadtree (q)
     Para cada quadrante c de q
          Se dentro do volume de visualização e se deve ser refinado
               Se não existe c na quadtree
                    Se o nível de máxima resolução não foi atingido
                         Cria c
```

```
                        Se c não está refinado no máximo
                              Actualiza Quadtree(c)
                        Regista alterações relevantes nos filhos de q
               Senão
                    Se q não está refinado no máximo
                        Actualiza Quadtree(c)
                        Se (c era folha e deixou de o ser)
                        ou (c não era folha e passou a sê-lo))
                              Regista alt. relevantes nos filhos de q
          Senão
               Se existe c na quadtree
                    Remove c, todos os descendentes e DLs associadas
                    Sinaliza nodos vizinhos
                    Regista alterações relevantes nos filhos de q
     Fim do Para
     Se (q é folha ou apenas tem filhos não folha)
          Se q pertence à lista ligada
               Remove q da lista ligada e DL associada
               Remove sinalização de q
     Senão
          Se q não pertence à lista ligada
               Insere q na lista ligada
               Sinaliza q
          Senão
               Sinaliza q se houve alt. relevantes nos seus filhos
Fim da Função
```

Figura 5-23: Algoritmo de actualização da *quadtree*.

5.6.2 Actualização das Display Lists *e Visualização da Triangulação*

Nesta versão do algoritmo a actualização das *display lists* e a visualização da triangulação são efectuadas durante uma travessia à lista duplamente ligada.

Para cada um dos nodos visitados durante essa travessia (ver Figura 5-24):

- Se o nodo está sinalizado a *display list* associada é recalculada de acordo com o algoritmo inicial e, logo de seguida[54], é executada.
- Se não está sinalizado, a *display list* associada é imediatamente executada pois não necessita de ser actualizada.

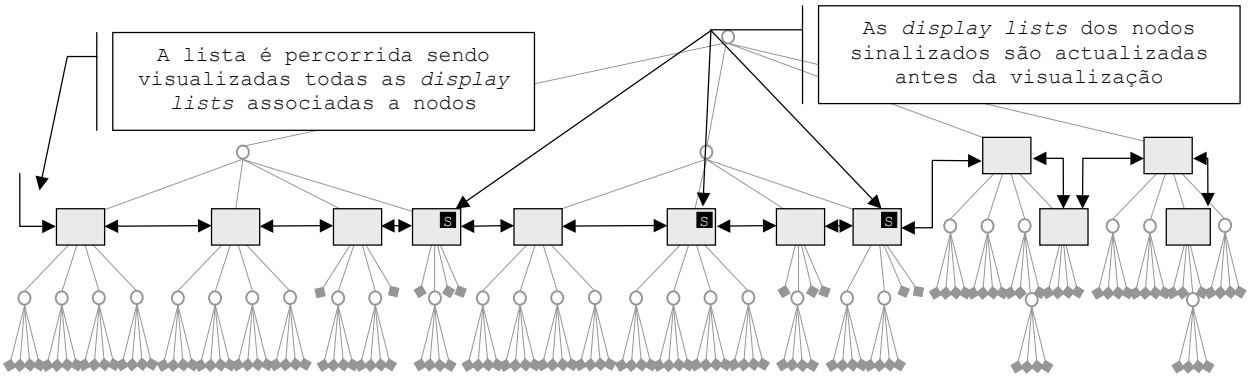


Figura 5-24: Actualização das *display lists* e visualização da triangulação.

O algoritmo que descreve este processo é apresentado de seguida na Figura 5-25.

```
Actualiza DLs
     Para todos os q da lista ligada
          Se q está sinalizado
               Remove sinalização de q
               Se não existe DL associada a q
                    Cria DL de q
               Redefine DL de q
                    Para todos os quadrantes c de q
                         Se existe c
                              Se c é folha
                                   Desenho do leque total
                         Senão
                              Assinala quadrante para render
                    Fim do para
                    Desenho dos leques dos quadrantes assinalados
               Fim da redefinição da DL de q
```

[54] Concluiu-se após alguns testes que, nos controladores de *OpenGL* testados, é mais rápido calcular a *display list* utilizando o parâmetro `GL_COMPILE` e, logo de seguida, executá-la, em vez de calcular e executar em simultâneo utilizando o parâmetro `GL_COMPILE_AND_EXECUTE`.

```
                Executa a DL associada a q
           Senão
                Executa a DL associada a q
      Fim do Para
Fim da Função
```

Figura 5-25: Algoritmo de actualização das *display lists* e visualização da triangulação.

5.7 Variação com Coerência entre Frames

O passo seguinte, o qual deu origem à terceira variação do algoritmo, visa tirar partido da coerência entre *frames*, utilizando a lista duplamente ligada para actualizar a *quadtree* da *frame* anterior. Desta forma toma-se como ponto de partida para a construção da *quadtree* seguinte apenas o conjunto de nodos cujas *display lists* associadas formam a triangulação da *frame* anterior. Evita-se assim a travessia à *quadtree* realizada durante esta fase nas versões anteriores.

Nesta versão, a actualização da *quadtree* é efectuada percorrendo a lista duplamente ligada e operando sobre cada um dos nodos que a constituem. Designa-se, nesta versão, por nodo activo o nodo da lista ligada no qual se inicia uma operação.

Cada nodo activo é testado com o volume de visualização e se estiver total ou parcialmente contido é avaliado, determinando-se qual a operação a efectuar a partir daquele nodo: refinamento ou generalização. Para tal recorre-se à mesma métrica de erro da versão inicial.

Os nodos que se encontram completamente fora do volume de visualização são alvo de uma operação de generalização pois o procedimento a seguir em ambos os casos deve ser o mesmo: a remoção do nodo e de todos os seus descendentes e a propagação da operação de generalização pelos seus ascendentes directos.

Quando se opta por iniciar o refinamento ou a generalização num dado nodo activo, esta operação é propagada, respectivamente, pelos seus descendentes válidos ou ascendentes directos, até que a métrica determine que esta operação deve ser terminada num dos nodos. Um descendente é designado de válido se é nulo ou folha (ver Figura 5-26).

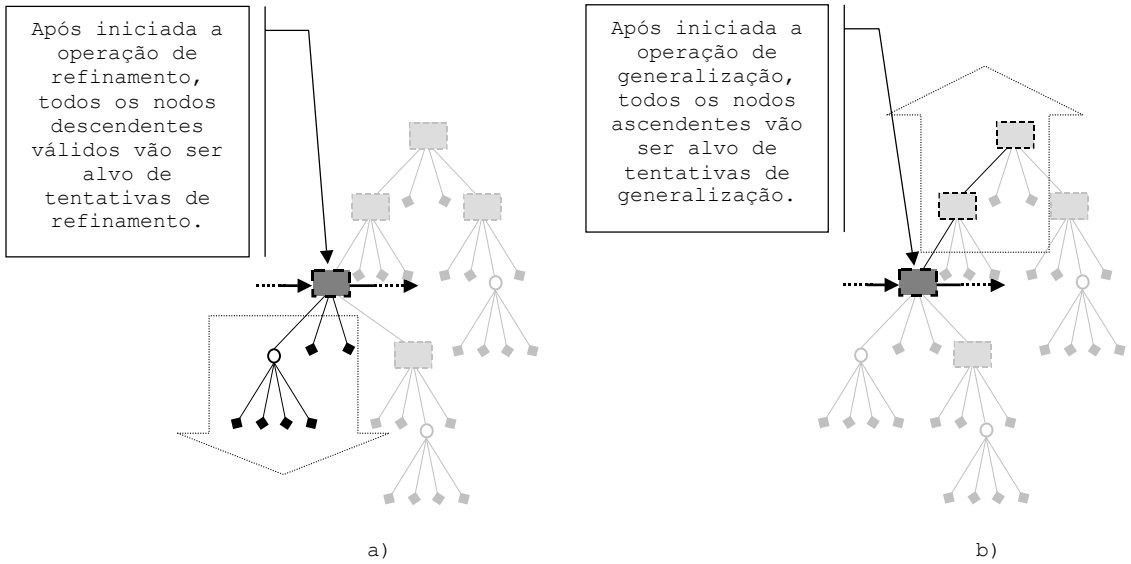


Figura 5-26: Propagação da operação escolhida num nodo da lista ligada. a) Refinamento. b) Generalização.

5.7.1 Operação de Refinamento

Após se determinar que o nodo activo deve ser refinado inicia-se um processo recursivo para o refinamento dos seus descendentes válidos.

Para auxiliar a operação de refinamento cada nodo contém cinco indicadores (ver Figura 5-27):

- Quatro indicadores, um por cada filho, que sinalizam se o filho deve ou não ser visitado/avaliado.

  Considere-se a Figura 5-27. Os indicadores de filhos são apresentados com um número que representa a ordem de avaliação. Esse número é apresentado com o fundo preto para o nodo que está a ser avaliado e com o fundo branco para os nodos a serem avaliados.

- Um indicador de alterações, que sinaliza se houve ou não alterações nos seus descendentes, para que sejam actualizadas as *display lists* associadas a nodos alterados.

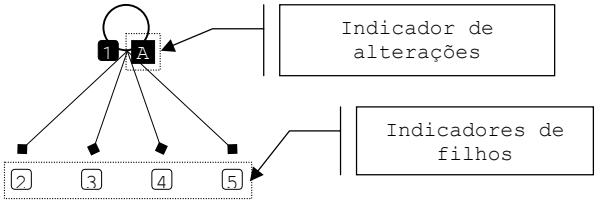


Figura 5-27: Indicadores associados a cada um dos nodos da *quadtree*.

O processo de refinamento começa por determinar quais os filhos válidos do nodo activo. Se um filho é válido, o indicador que lhe corresponde é activado, significando que este filho deve ser visitado e avaliado. Qualquer filho não válido não deve ser visitado durante o processo de avaliação a este nodo activo. Esta distinção pretende minimizar o número de nodos avaliados. Considere-se, por exemplo, um filho não válido com uma descendência completa, isto é, sem filhos nulos, por cinco níveis da *quadtree* e que após avaliados não sofreriam alteração. Neste caso, a avaliação de filhos não válidos implicaria a avaliação desnecessária de 84 nodos (4 + 16 + 64 correspondente aos 1º, 2º e 3º níveis de descendentes). Note-se que todos os nodos do 4º nível de descendência fariam parte da lista ligada, pois todos os seus filhos seriam nodos folha (correspondendo ao 5º nível de descendência), e seriam sempre avaliados na sua vez durante a travessia à lista ligada.

Mais ainda, no caso do nodo activo coincidir com a raiz da *quadtree*, o processo completo de actualização da *quadtree* seria realizado no processo de refinamento deste nodo activo – os descendentes deste nodo cobririam por completo todos os nodos da *quadtree* – o que seria idêntico ao processo de actualização da *quadtree* da versão anterior e contrário ao objectivo desta versão.

O processo segue analisando um a um todos os filhos sinalizados.

Filhos Nulos

Se ao avaliar um filho nulo se conclui que este deve ser refinado, é criado um novo nodo no seu lugar e o indicador de alterações do nodo pai é activado (ver Figura 5-28). Após a criação deste novo nodo, são sinalizados de imediato os indicadores de todos os filhos[55] para que todos eles sejam, de seguida, avaliados e, se necessário, refinados. Excepção feita se tiver sido atingido o ponto de máxima resolução.

[55] De notar que todos os filhos de um novo nodo são nulos e, consequentemente, válidos.

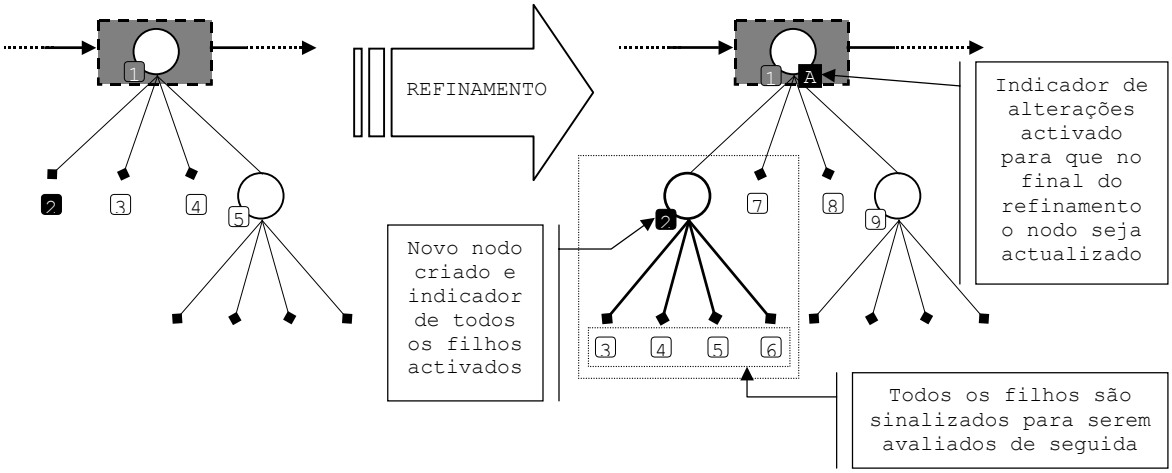


Figura 5-28: Criação de um novo nodo.

Para garantir uma correcta actualização das *display lists*, que é realizada na fase seguinte do algoritmo, o nodo avô do nodo criado deve ser sinalizado se se verificarem as seguintes condições (ver Figura 5-29. O número da ordem de avaliação dos nodos que já foram avaliados é apresentado com o fundo cinza):

1. Pertencer à lista duplamente ligada, isto é, tiver uma *display list* associada;
2. O nodo pai do nodo criado era um nodo folha antes da criação;
3. Se após a criação do novo nodo, o nodo avô ainda necessitar de uma *display list* associada.

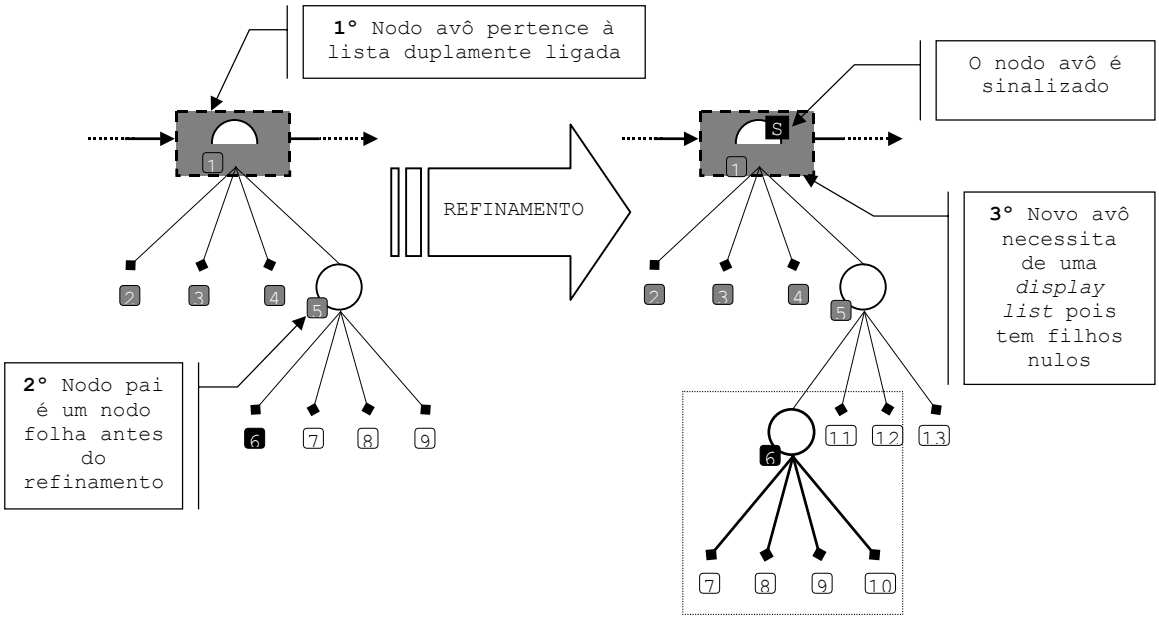


Figura 5-29: Sinalização do nodo avô do nodo criado.

Caso falhe apenas a terceira condição, o indicador de alterações do nodo avô deve ser activado para que este seja removido da lista duplamente ligada e para que a *display list* a ele associada seja também removida (ver Figura 5-30).

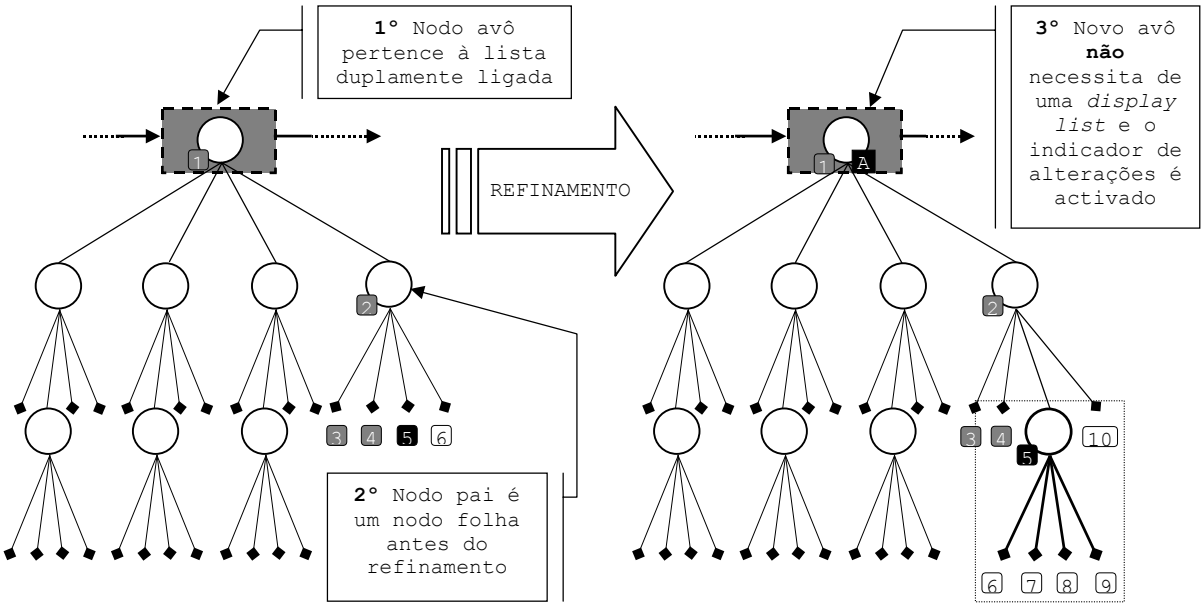


Figura 5-30: Activação do indicador de alterações do nodo avô do nodo criado.

O processo termina quando o processo de avaliação de um filho nulo determina que este não necessita de ser refinado.

Filhos Não Nulos

Na avaliação de um filho não nulo a operação de refinamento do nodo activo assume um significado mais lato: é possível que os seus filhos válidos não nulos sejam removidos.

O facto de um nodo activo necessitar de ser refinado não implica que todos os seus filhos necessitem também de refinamento. É inclusivamente possível que um nodo activo necessite de ser refinado mas que um dos seus filhos esteja demasiado refinado, de acordo com a métrica de erro em vigor. Nestes casos, o filho em questão deve ser simplificado, isto é, generalizado, e não refinado.

Entende-se portanto que a operação aqui descrita é uma operação de refinamento ao nível global, podendo, no entanto, localmente, assumir um carácter de generalização.

Nestes casos, para além da remoção do nodo correspondente a este filho, o indicador de alterações do nodo pai é activado e o processo de refinamento que iniciou no nodo activo termina aqui para este filho (ver Figura 5-31).

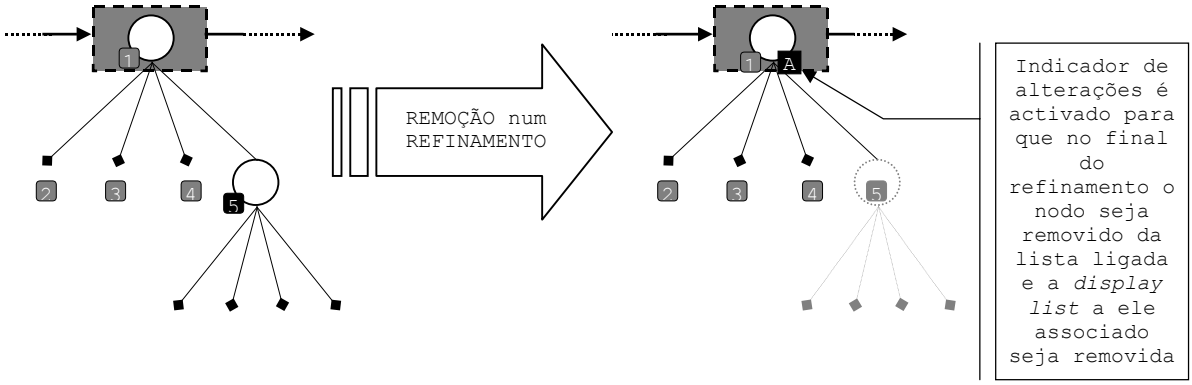


Figura 5-31: Remoção de um filho não nulo no processo de refinamento.

No caso da avaliação do filho não nulo resultar num verdadeiro refinamento, são sinalizados todos os filhos deste nodo (ver Figura 5-32), excepto se tiver sido atingido o ponto de máxima resolução. Estes filhos são avaliados, logo de seguida, e o processo de refinamento prossegue como descrito anteriormente para filhos nulos.

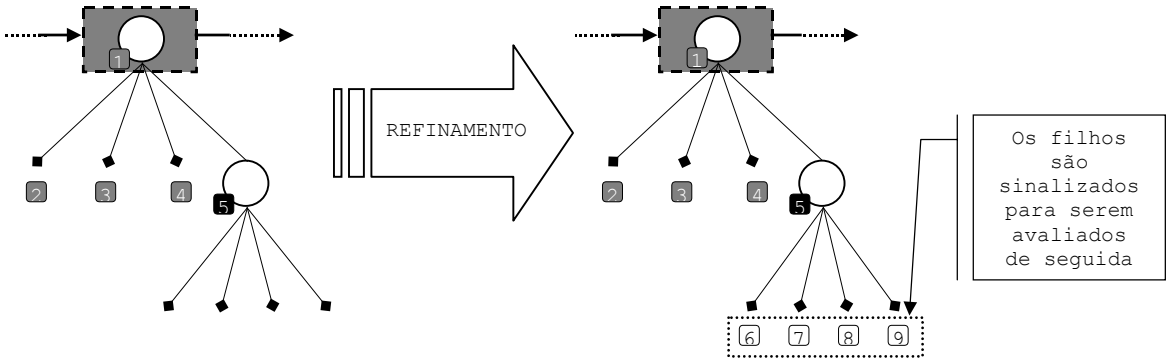


Figura 5-32: Refinamento de um filho não nulo.

Actualizações finais

Quando todos os filhos de um nodo que foi alvo de um processo de refinamento tiverem sido avaliados há que proceder a actualizações à lista duplamente ligada e/ou à sinalização do nodo, se o indicador de actualizações do nodo em causa tiver sido activado (ver Figura 5-33).

Caso contrário nada foi alterado na descendência deste nodo pelo que nenhuma actualização é necessária.

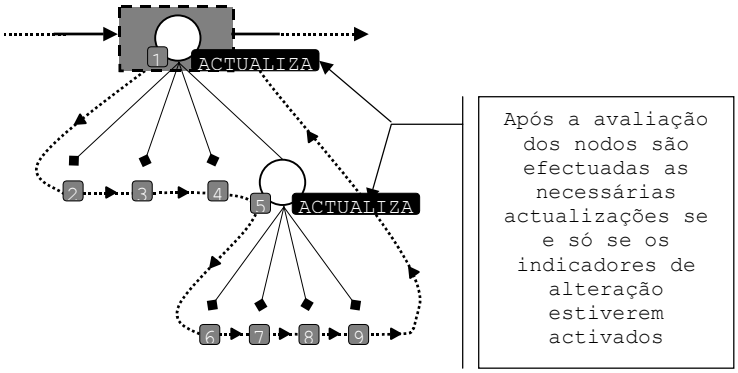


Figura 5-33: Actualizações finais.

As actualizações finais, se necessárias, efectuam-se do seguinte modo:

- Se o nodo necessitar de uma *display list* associada, ele é sinalizado e inserido na lista duplamente ligada (caso ainda não pertença) (ver Figura 5-34).

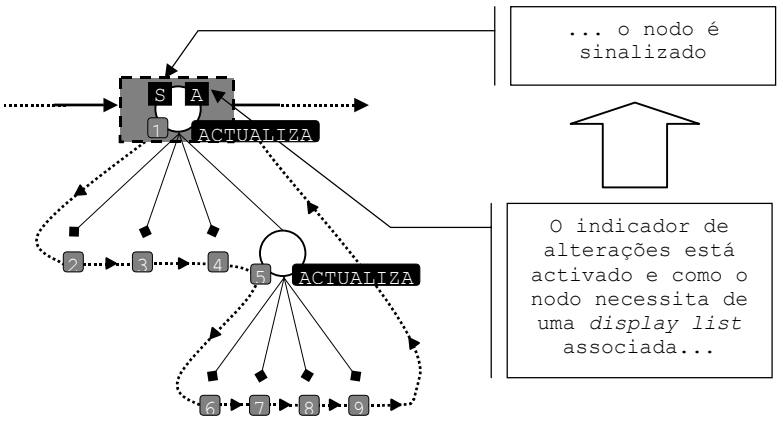


Figura 5-34: Actualização final num nodo que necessita de uma *display list* associada.

- Se não necessitar de uma *display list* associada...
    - ... mas pertencer à lista duplamente ligada, ele é removido da mesma e a di*splay list* a ele associada é removida (ver Figura 5-35).

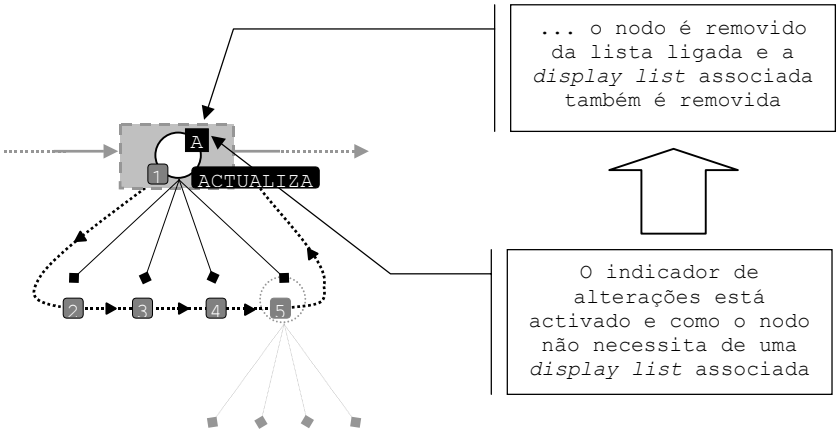


Figura 5-35: Actualização final num nodo que não necessita de uma *display list* associada.

- ... e for um nodo folha, o nodo pai é sinalizado e inserido na lista se ainda não pertencer (ver Figura 5-36).

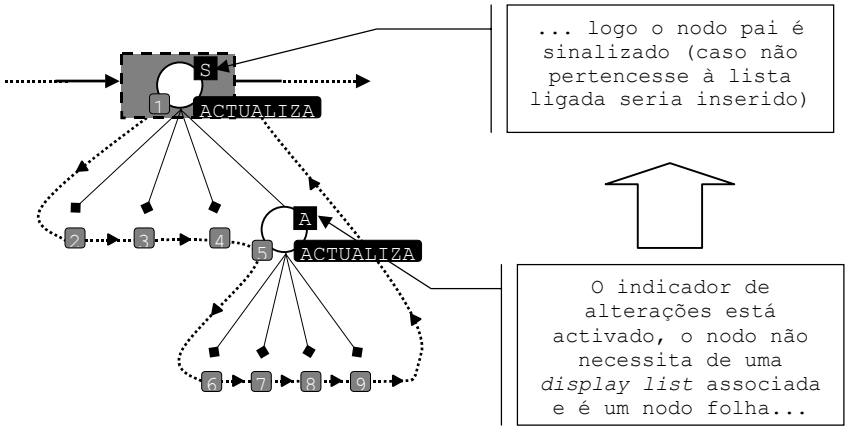


Figura 5-36: Actualização final num nodo folha com o indicador de alterações activado.

Quando o nodo activo for alcançado, o processo de refinamento termina e reinicia-se o processo de avaliação com o nodo seguinte da lista duplamente ligada, que passa a ser o novo nodo activo (ver Figura 5-37).

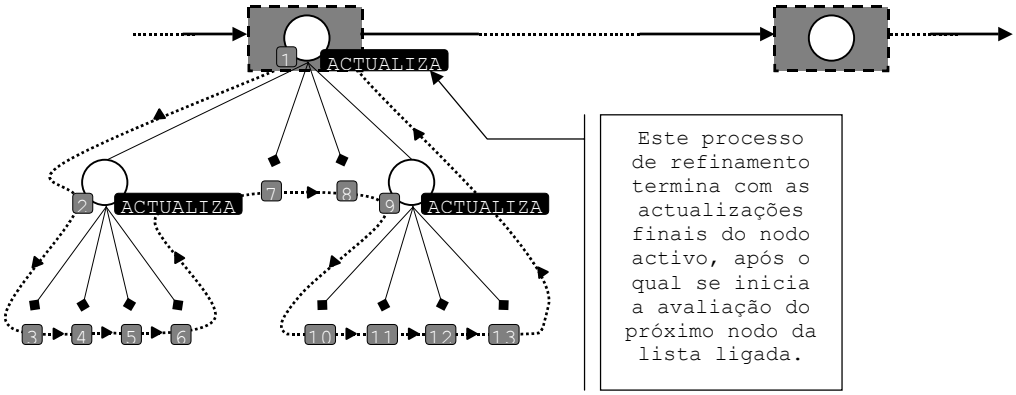


Figura 5-37: Fim do processo de refinamento para o nodo activo corrente.

5.7.2 Operação de Generalização

Após se determinar que o nodo activo pode ser generalizado inicia-se um processo que pretende generalizar ao máximo os seus ascendentes directos.

Na operação de generalização o nodo activo e todos os seus descendentes são removidos, bem como todas as *display lists* a eles associadas. Pelas razões apresentadas na versão inicial (com *display lists* hierárquicas) são sinalizados os nodos adjacentes necessários (ver secção *5.2.2 – As Operações*) no sentido de se evitarem descontinuidades espaciais.

O processo de generalização continua pelos nodos ascendentes do nodo activo removido, até que se conclua, num dos ascendentes, que não é possível generalizar mais aquela região de terreno (ver Figura 5-38).

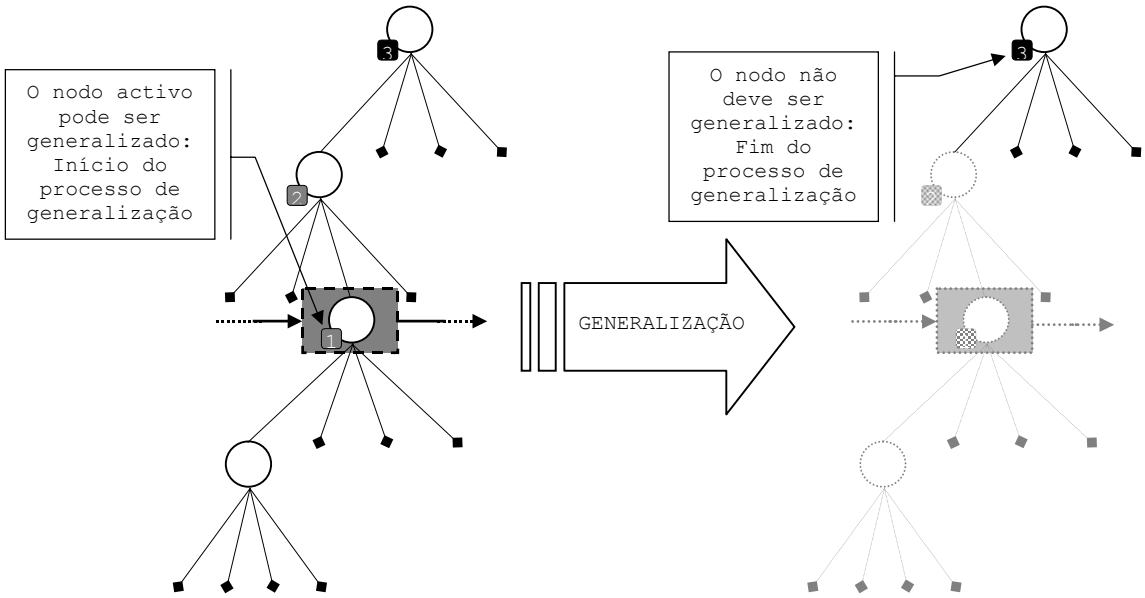


Figura 5-38: Processo de generalização.

A fim de evitar que a cadeia da lista duplamente ligada seja quebrada, é necessário algum cuidado sempre que se removem nodos da lista. Se o nodo que se segue ao nodo a remover na lista duplamente ligada fizer parte dos descendentes a remover, é necessário garantir que o processo de avaliação prossegue com o primeiro nodo não removido que se segue ao nodo a remover na lista duplamente ligada. Assim, são realizadas as seguintes operações (ver Figura 5-39):

1. Antes de qualquer remoção, o apontador para o nodo a remover passa a apontar o nodo anterior da lista ligada.
2. Segue-se a remoção do nodo a remover e de todos os seus descendentes, com a consequente actualização da lista duplamente ligada.
3. O apontador para o nodo a remover passa a apontar para o nodo seguinte, que é agora o primeiro nodo não removido que segue ao nodo a remover.

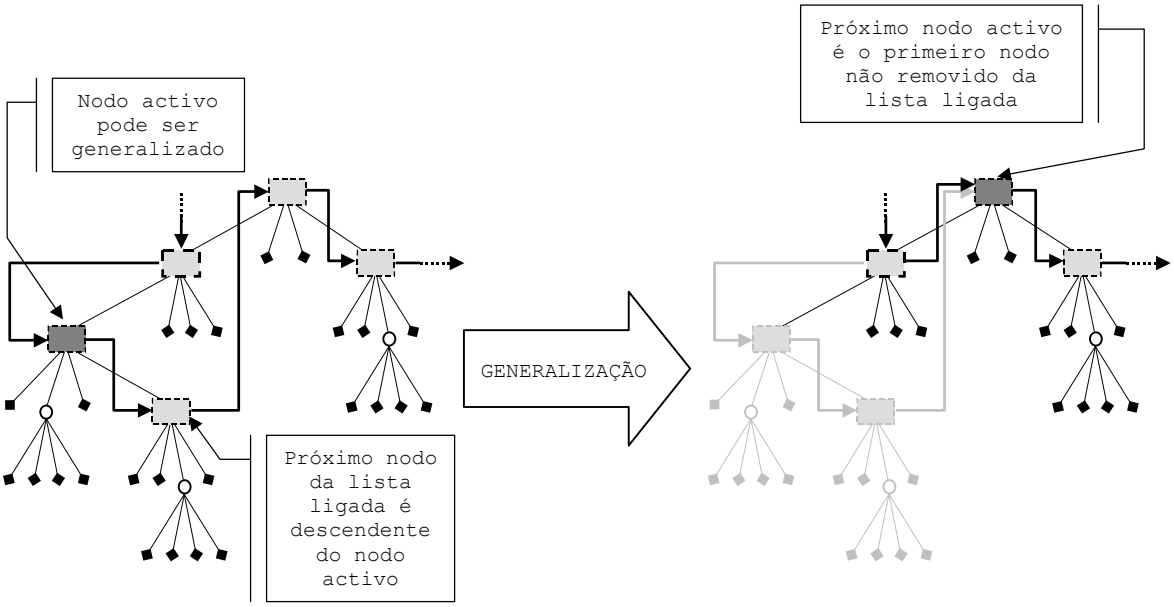


Figura 5-39: Remoção do nodo activo e seus descendentes da lista duplamente ligada.

No fim do processo de generalização, iniciado num nodo activo, é necessário proceder a algumas actualizações.

Se o último nodo ascendente não removido resultar num nodo folha (ver Figura 5-40):

- O seu pai é sinalizado e, se não pertencer à lista duplamente ligada, é inserido.
- Se o nodo pertencer à lista duplamente ligada, ele é removido da lista e a *display list* a ele associada é também removida.

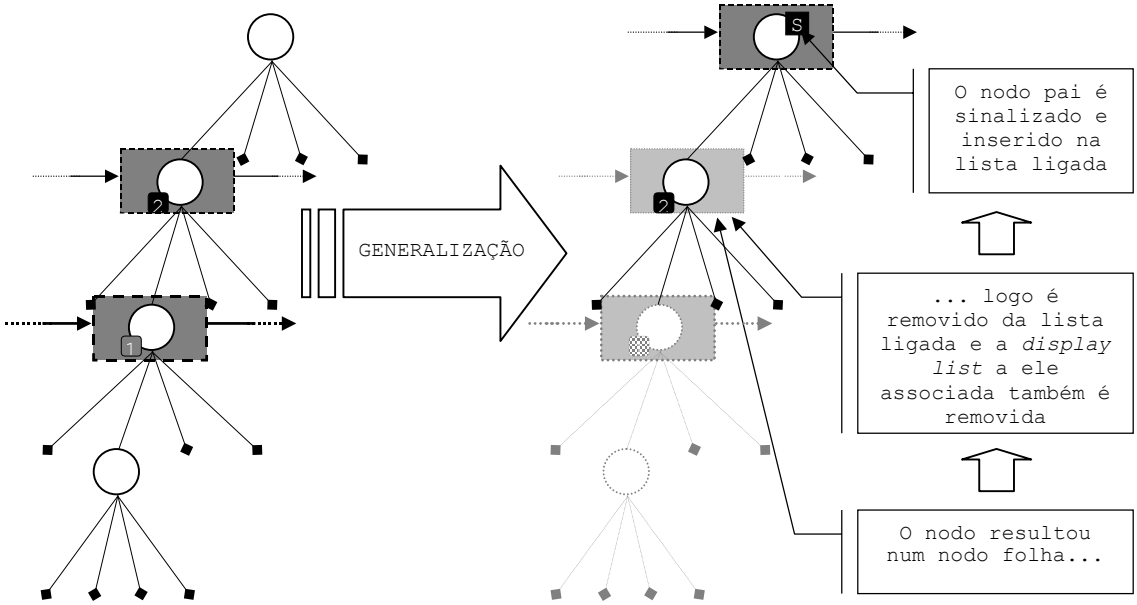


Figura 5-40: Actualizações finais no processo de generalização para um nodo folha.

Senão (ver Figura 5-41):

- O nodo é sinalizado, pois sofreu alterações, e, se não pertencer à lista duplamente ligada, é inserido. De notar que o nodo tem de ter sempre uma *display list* associada pois tem pelo menos um filho nulo (o que acabou de ser removido).

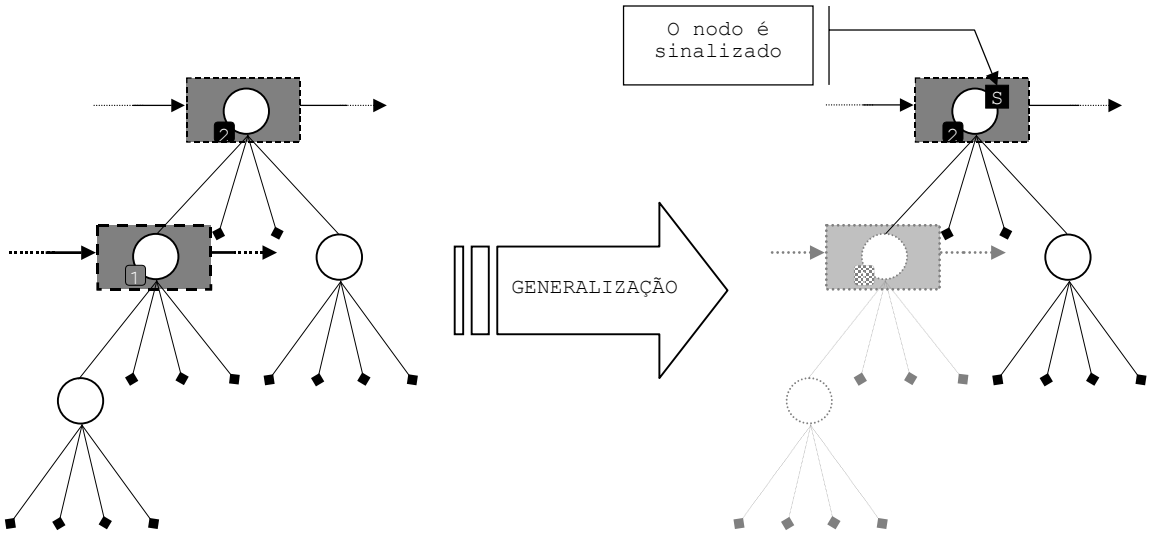


Figura 5-41: Actualizações finais no processo de generalização para um nodo não folha.

Após a avaliação e actualização de todos os nodos da lista duplamente ligada, a fase seguinte é a da actualização das *display lists* e a visualização da triangulação. Esta última fase é efectuada de acordo com o algoritmo da versão anterior (ver Figura 5-25, página 111).

Apesar desta versão tirar partido da coerência real entre *frames*, todo o processo descrito apresenta uma complexidade algorítmica acrescida quando comparada com as versões anteriores, que advém do número de procedimentos que são necessários realizar durante a actualização da *quadtree*.

5.8 Conclusão

Neste capítulo apresentaram-se três versões de um novo algoritmo que utilizam *display lists* para a visualização de terrenos em tempo real. As três versões desenvolvidas apresentam características e complexidades algorítmicas diferentes que se traduzem em desempenhos distintos, conforme se comprova no capítulo seguinte.

A versão inicial do algoritmo utiliza *display lists* hierárquicas. Os processos de construção e actualização da *quadtree* são *top-down* recursivos, e neles é realizada a sinalização dos nodos cujas *display lists* associadas necessitam de actualização. Segue-se uma nova travessia à *quadtree*, no processo de actualização de *display lists*, que procura e actualiza todas as *display lists* associadas a nodos sinalizados.

Todos estes processos mantêm a hierarquia de *display lists* o que permite reduzir a fase de visualização da triangulação a uma simples invocação à *display list* associada à raiz da *quadtree*. A utilização desta hierarquia permite transferir para a API gráfica a responsabilidade pela sua travessia.

A segunda versão do algoritmo é uma variação que utiliza *display lists* simples e que gere uma lista duplamente ligada com todos os nodos a elas associados.

A eliminação da hierarquia de *display lists* evita a utilização desnecessária de *display lists*, pela eliminação de *display lists* que não armazenam directamente parte da triangulação utilizada.

A utilização de uma lista duplamente ligada permite tornar mais eficientes os processos de actualização e visualização de *display lists*, apesar dos processos de construção e actualização da *quadtree* necessitarem de realizar a sua construção e manutenção. Esta lista duplamente ligada permite evitar uma nova travessia da *quadtree* nos processos de actualização e visualização das *display lists*, substituindo-a por uma travessia à lista duplamente ligada. Desta forma, apenas são visitados os nodos cujas *display lists* associadas necessitam de actualização e/ou visualização. No decurso desta travessia são actualizadas todas as *display lists* associadas a

nodos sinalizados e todas elas são visualizadas. Dado que não é mantida uma hierarquia de *display lists* é necessário efectuar a visualização explícita de todas as *display lists*.

A terceira versão do algoritmo tira partido da coerência entre *frames*, recorrendo à lista duplamente ligada para efectuar a actualização da *quadtree* da *frame* seguinte.

Ao contrário das versões anteriores, esta versão não realiza uma travessia pela *quadtree* no processo da sua actualização. Esta travessia é substituída por uma travessia à lista duplamente ligada que contêm exclusivamente nodos com *display lists* associadas, isto é, nodos que contribuem directamente para a triangulação utilizada na *frame* anterior.

O processo de actualização e visualização de *display lists* é o mesmo da versão anterior.

Apesar das vantagens enumeradas, a complexidade algorítmica desta última versão é maior do que a das anteriores, como já foi mencionado anteriormente.

No quadro seguinte enumeram-se os pontos fundamentais que distinguem as diferentes versões descritas neste capitulo (ver Figura 5-42).

<table>
<tr><th></th><th>Versão com Display Lists Hierárquicas</th><th>Versão com Display Lists Simples</th><th>Versão com Coerência entre Frames</th></tr>
<tr><td>Construção da Quadtree</td><td rowspan="2">Top-Down Recursivo, com sinalização das alterações</td><td rowspan="2">Top-Down Recursivo, com sinalização das alterações e manutenção duma lista duplamente ligada</td><td>Top-Down Recursivo, com sinalização das alterações e manutenção duma lista duplamente ligada</td></tr>
<tr><td>Actualização da Quadtree</td><td>Actualização com recurso à lista duplamente ligada</td></tr>
<tr><td>Actualização das DLs</td><td>Top-Down Recursivo. Actualização das DLs de nodos sinalizados</td><td colspan="2" rowspan="2">Travessia à lista duplamente ligada para:
Actualização das DLs de nodos sinalizados.
Visualização de todas as DLs.</td></tr>
<tr><td>Visualização da Triangulação</td><td>Invocação da DL associada à raiz da quadtree.</td></tr>
</table>

***Display Lists* Hierárquicas**	Sim	Não	Não
Coerência entre Frames	Não	Não	Sim

Figura 5-42: Resumo das principais características das três versões do novo algoritmo.

6 Testes e Análise de Resultados

Para avaliar o desempenho das diferentes versões do algoritmo desenvolvido no âmbito desta dissertação, foram efectuados diversos testes sendo aqui analisados os resultados obtidos.

Foram implementadas as três versões do algoritmo desenvolvido no âmbito desta dissertação. Para efeitos de comparação foram seleccionados os algoritmos de Lindstrom et al. [22], de Röttger et al. [29] e o ROAM [8], apresentados no capítulo *4 - Algoritmos de Visualização de Terrenos em Tempo Real - (Estado da Arte)*, uma vez que dentro dos algoritmos que operam sobre grelhas regulares são os mais referenciados pela comunidade científica. Todos eles foram testados em condições tão semelhantes quanto possível.

Os testes efectuados passaram pela utilização de dois terrenos diferentes, que foram percorridos segundo dois percursos, em três máquinas com *hardware* gráfico distinto.

Neste capítulo descrevem-se as condições em que todos os testes foram realizados, apresentam-se os testes e uma análise aos resultados obtidos. Inicialmente comparam-se os algoritmos seleccionados do capítulo *4*, no sentido de determinar o de melhor desempenho. Segue-se a comparação deste com as diferentes versões do novo algoritmo desenvolvido nesta dissertação.

6.1 Descrição dos Testes

Para que a comparação entre os diferentes algoritmos implementados fosse o mais isenta possível, foram criadas condições semelhantes para a execução de todos os testes.

6.1.1 Ambiente de Teste

Todos os algoritmos foram implementados na linguagem C e compilados no *Microsoft Visual Studio 6.0 Enterprise Edition*. A biblioteca gráfica utilizada foi o *OpenGL* versão 1.2, auxiliada pelo GLUT que, de entre outras funções, é um sistema de gestão de janelas independente da plataforma. A versão do GLUT utilizada foi a 3.7.5.

O ambiente de testes utilizado foi o mesmo para todos os algoritmos, para garantir as mesmas condições na execução dos testes efectuados. As decisões tomadas durante a implementação dos algoritmos foram, sempre que possível, semelhantes em todos os algoritmos e privilegiaram sempre o desempenho.

Eliminou-se a sobrecarga do CPU criada pela alocação de memória em tempo de execução [24][27], alocando-se durante a fase de inicialização toda a memória RAM necessária as estruturas de dados utilizadas.

Utilizou-se a técnica de *view frustum culling* em todas as implementações, mas em nenhuma delas foi utilizada a técnica de *vertex morphing*.

6.1.2 Terrenos

Os terrenos utilizados nos testes permitem analisar o comportamento dos algoritmos em duas situações geográficas distintas.

O primeiro, terreno `A`, representa uma região com uma montanha central (ver Figura 6-1). O segundo, terreno `B`, representa uma região montanhosa mais dispersa (ver Figura 6-2).

A origem dos dados são imagens em tons de cinza, sendo que a altitude de cada ponto do terreno é dada pela sua intensidade, que pertence a um gradiente entre o branco e o preto. Quanto mais clara for a cor de um ponto, maior será a sua altitude. A altitude máxima do terreno `A` foi escalada para 60 unidades e a do terreno `B` para 120.

As dimensões dos terrenos utilizados foram de $2^n+1 \times 2^n+1$ com $n=10$, 11 e 12, o que corresponde às dimensões 1025×1025, 2049×2049 e 4097×4097 unidades, respectivamente.

Note-se que o espaçamento utilizado entre os pontos de um terreno foi de 1 unidade em todas as dimensões. Desta forma, uma maior dimensão corresponde a um terreno maior e não a uma maior resolução para a mesma dimensão.

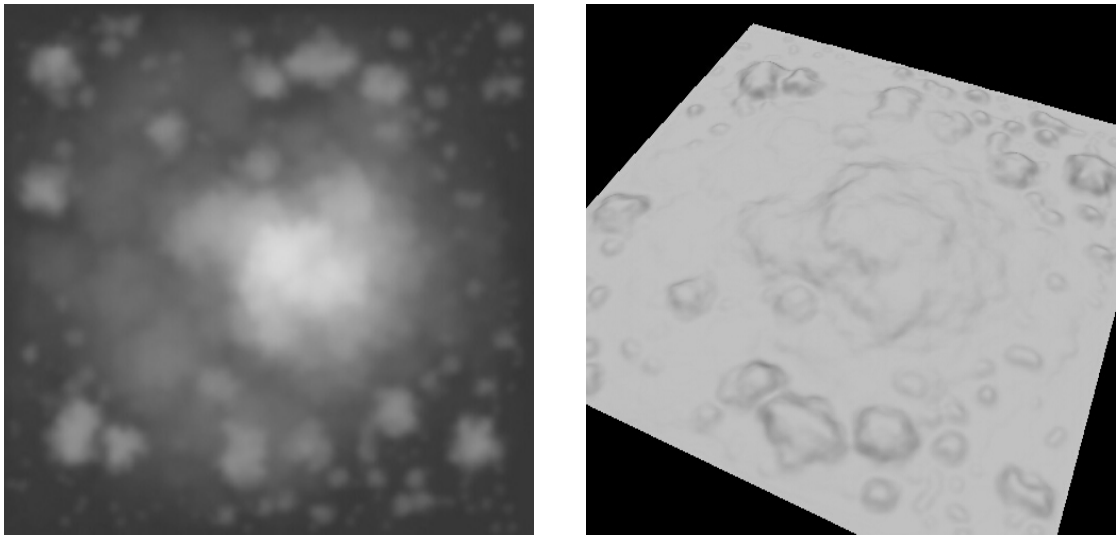


Figura 6-1: Terreno `A` utilizado nos testes realizados.

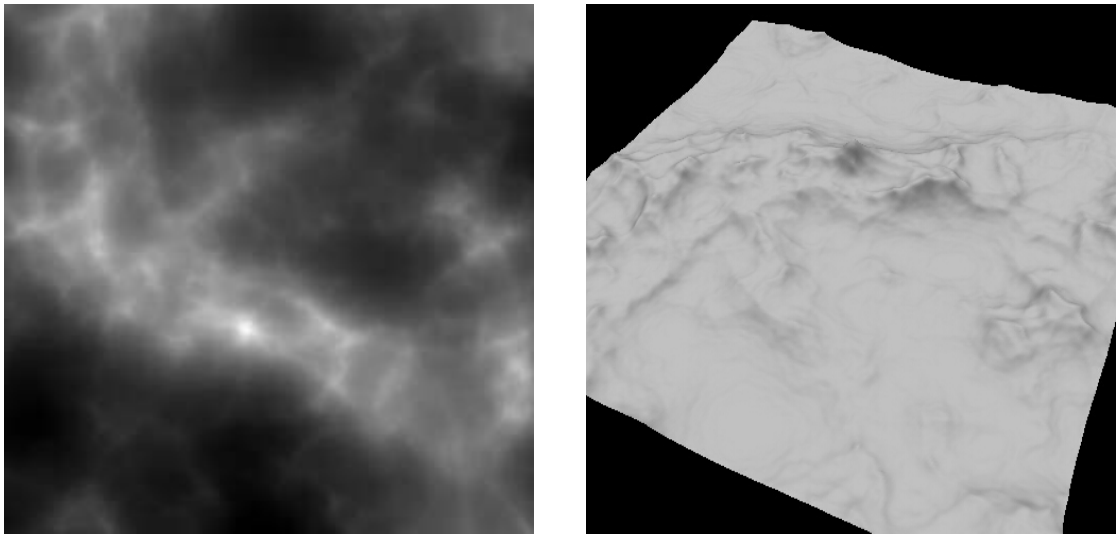


Figura 6-2: Terreno `B` utilizado nos testes realizados.

6.1.3 Percursos

No sentido de testar os algoritmos em diferentes situações, foram efectuados dois percursos distintos, a duas velocidades distintas, totalizando quatro situações de teste. O teste a duas velocidades permite analisar o comportamento dos diferentes algoritmos em situações de maior ou menor variação das *frames* geradas.

Os percursos utilizados foram:

- *Percurso Circular*: O terreno é percorrido por um trajecto circular com a câmara direccionada para o centro (ver Figura 6-3).

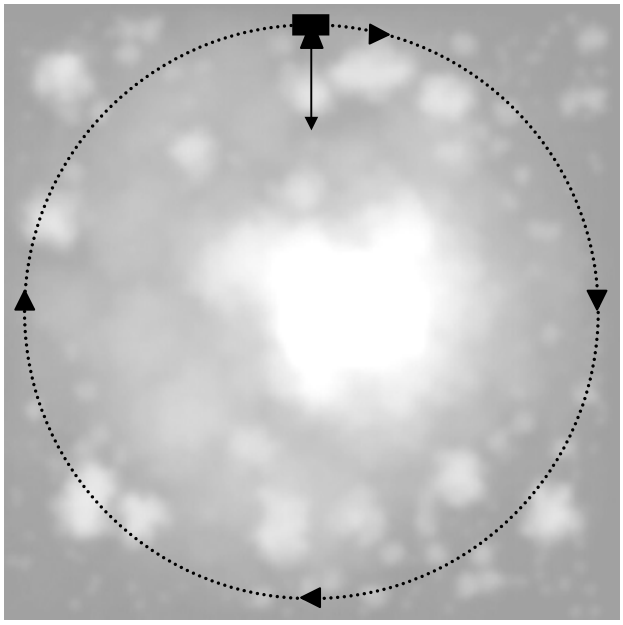


Figura 6-3: Percurso circular efectuado nos testes.

A velocidade lenta neste percurso, considerando o terreno com dimensões `1025 x 1025`, corresponde a uma deslocação de 0,1 unidades por *frame*, sendo um ciclo completado em 37000 *frames*.

A velocidade rápida corresponde a uma deslocação de 1,0 unidades por *frame*, sendo um ciclo completado em 3700 *frames*.

- *Percurso Lemniscata de Bernoulli (LB)*: O terreno é percorrido por um trajecto em forma de lemniscata de Bernoulli com a câmara direccionada para a frente (ver Figura 6-4).

  A velocidade lenta neste percurso corresponde, em média, a 0,22 unidades por *frame* no terreno com dimensões `1025 x 1025`, sendo um ciclo completado em 37000 *frames*.

  A velocidade rápida corresponde, em média, a uma deslocação de 2,2 unidades por *frame*, sendo um ciclo completado em 3700 *frames*.

  Note-se que neste percurso a velocidade da câmara não é constante. Existe uma aceleração à medida que a câmara se aproxima do centro do terreno, facto que também contribui para uma diminuição da coerência real entre *frames* nesta zona do percurso.

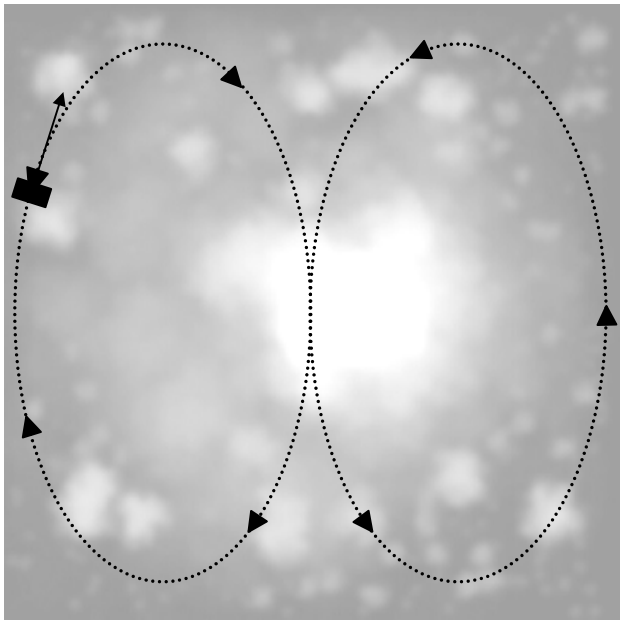


Figura 6-4: Percurso em forma de lemniscata de Bernoulli efectuado nos testes.

Em todos os percursos a altitude da câmara é actualiza em cada *frame* de acordo com o valor da altitude do terreno no ponto onde se encontra. A este valor foi adicionada 1 unidade.

6.1.4 Máquinas

Diferentes sistemas produzem resultados distintos. Neste sentido os algoritmos implementados foram testados em várias arquitecturas com diferentes *hardwares* gráficos. Estes organizam e processam toda informação gráfica de modo diverso, influenciando os resultados obtidos.

Os algoritmos apresentados foram testados nas seguintes máquinas[56]:

- *Máquina Obelix*: Intel Pentium II 266MHZ, 128MB de RAM, placa gráfica ELSA Erazor III Pro com GPU nVidia Riva TNT2 PRO com 32MB SyncRAM, em Windows 98. Resolução do monitor: 1024x768, 32 bits de cor.
- *Máquina Jerry*: Intel Pentium III 500MHZ com SSE, 512MB de RAM, placa gráfica ASUS com GPU nVidia GeForce256 com 32MB DDR SDRAM, em Windows XP. Resolução do monitor: 1280x1024, 32 bits de cor.

[56] Os nomes atribuídos às máquinas surgem da sua identificação no ambiente de rede no qual se encontram.

- *Máquina Calvin*: Intel Pentium IV 2.0GHZ com SSE2, 1GB de RAM, placa gráfica ASUS com GPU nVidia GeForce4 TI 4600 com 128MB DDR SDRAM, em Windows XP. Resolução do monitor: 1600x1200, 32 bits de cor.

Note-se que a resolução do monitor tem apenas um carácter informativo dado que a dimensão da janela utilizada nos testes foi a mesma para todas as máquinas enumeradas.

Dada a disparidade existente entre estas máquinas, no que se refere a capacidade computacional e gráfica, foram utilizados parâmetros das métricas distintos. Assim sendo, apresenta-se na tabela da Figura 6-5 os valores desses parâmetros para as diferentes máquinas.

		MÁQUINAS		
ALGORITMOS	**Parâmetros da Métrica**	**Obelix**	**Jerry**	**Calvin**
Lindstrom et al.	Valor do Limite	3,2	3,2	3,2
ROAM	Número de Triângulos	10000	15000	15000
Röttger et al.	Resolução Global Mínima	10,0	20,0	20,0
	Resolução Global Desejada	30,0	30,0	30,0
Novo Algoritmo	Resolução Global Mínima	10,0	20,0	20,0
	Resolução Global Desejada	30,0	30,0	30,0

Figura 6-5: Valores dos parâmetros das métricas utilizados nos algoritmos implementados.

De salientar que os parâmetros utilizados para o algoritmo de Röttger et al. são, por questões de isenção nas comparações realizadas, iguais aos utilizados pelas diferentes versões do novo algoritmo desenvolvido no âmbito desta dissertação.

6.2 Testes Preliminares

Nesta secção apresentam-se os resultados dos testes efectuados aos algoritmos de Lindstrom et al. [22], de Röttger et al. [29] e o ROAM [8] com o objectivo de se determinar qual o mais rápido nas diversas situações testadas.

Os testes realizados procuraram avaliar o comportamento destes algoritmos em situações de maior ou menor coerência real entre *frames* e em diferentes máquinas.

A situação de teste que se apresenta de seguida refere-se a dois percursos distintos no terreno `A` de `1025 x 1025` realizados na máquina `Jerry`. O primeiro gráfico mostra a taxa de FPS num

percurso `LB` realizado a uma velocidade rápida (ver Figura 6-6), que representa uma situação com grandes variações de imagem.

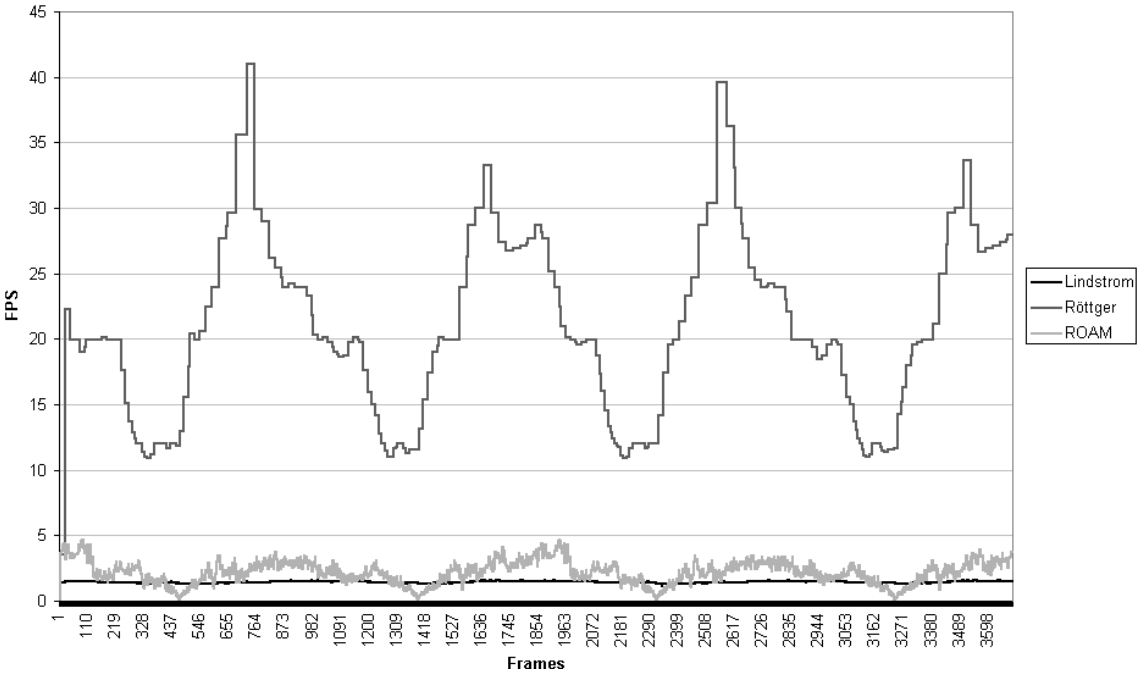


Figura 6-6: Taxa de FPS no terreno A de 1025x1025 num ciclo do percurso LB, velocidade rápida, realizado na máquina Jerry.

O gráfico seguinte mostra a taxa de FPS num percurso `circular` efectuado a uma velocidade lenta (ver Figura 6-7) no qual existe uma maior coerência real entre *frames*.

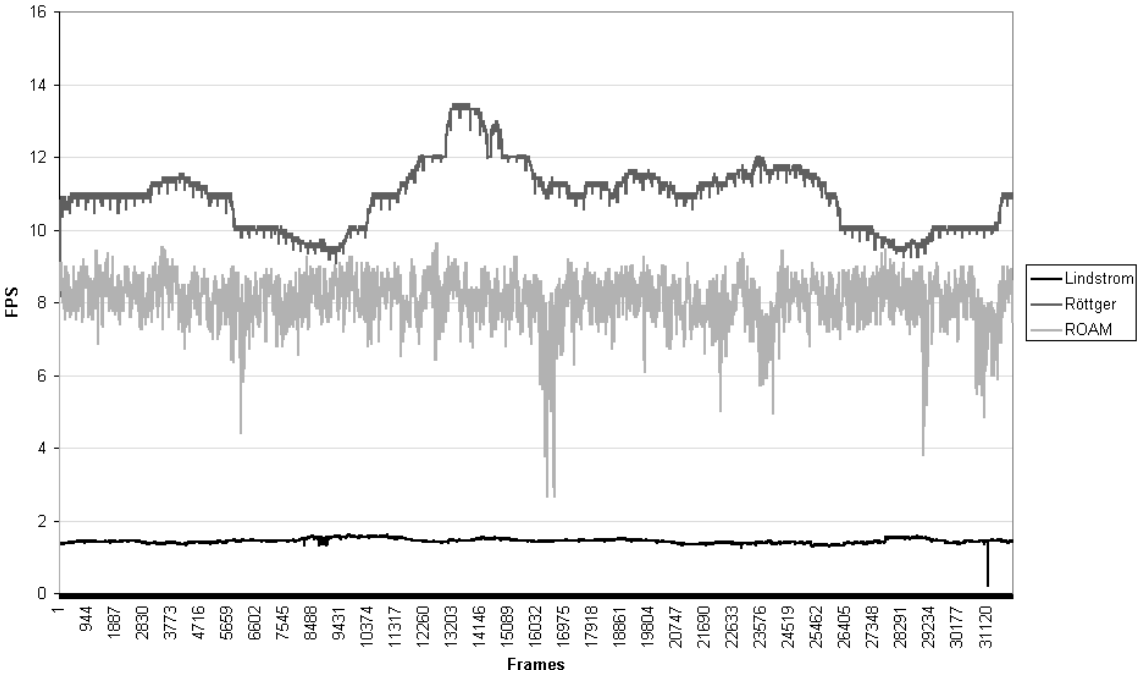


Figura 6-7: Taxa de FPS no terreno A de 1025x1025 num percurso circular, velocidade lenta, realizado na máquina Jerry.

Em ambas as situações o algoritmo de Röttger et al. é claramente mais rápido do que o ROAM ou o de Lindstrom et al.. Mais ainda, o número de triângulos utilizados é também claramente superior, em média, (ver Figura 6-8 e Figura 6-9) o que permite obter uma imagem de melhor qualidade.

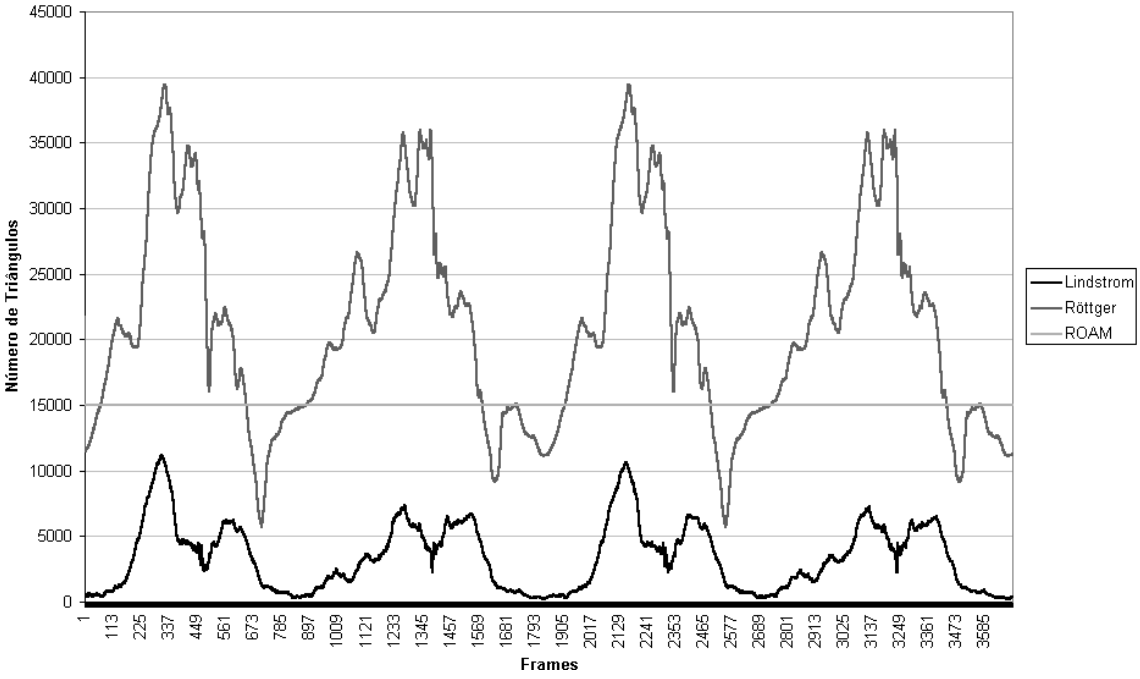


Figura 6-8: Número de Triângulos utilizados no terreno A de 1025x1025 num ciclo do percurso LB, velocidade rápida, realizado na máquina Jerry.

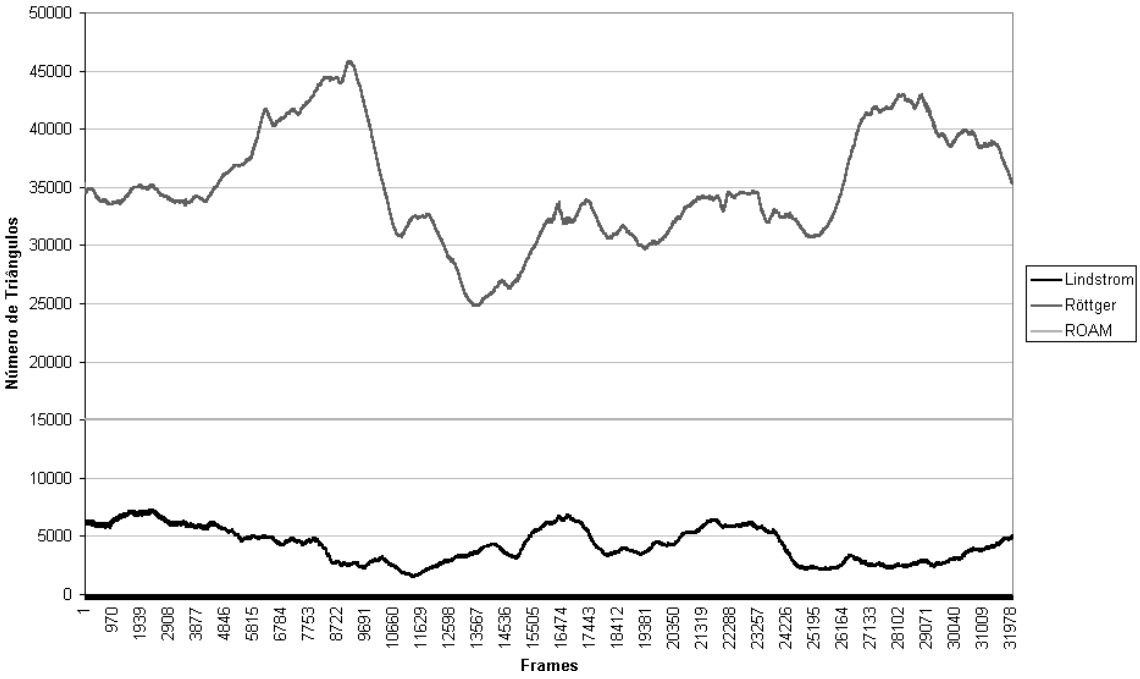


Figura 6-9: Número de Triângulos utilizados no terreno A de 1025x1025 num percurso circular, velocidade lenta, realizado na máquina Jerry.

As diferenças notadas nas situações enumeradas mantiveram-se em todas as máquinas, terrenos e situações testadas.

Da análise efectuada concluiu-se que, nas situações testadas, o algoritmo de Röttger et al. é o mais rápido de entre os algoritmos mencionados, facto que contribuiu para a escolha deste algoritmo como base de teste para o novo algoritmo desenvolvido nesta dissertação.

6.3 Testes à Aplicabilidade de *Display Lists*

Nesta secção apresentam-se os testes realizados às diferentes versões do algoritmo desenvolvido nesta dissertação. A primeira versão utiliza *display lists* hierárquicas para armazenar e visualizar a triangulação do terreno; a segunda versão utiliza apenas *display lists* simples e o processo de visualização é auxiliado por uma lista duplamente ligada contendo os nodos com *display lists* associadas; a terceira versão tira partido da coerência entre *frames*, recorrendo à lista duplamente ligada para actualizar a *quadtree* existente e assim obter a *quadtree* para a *frame* seguinte.

As diferentes versões do algoritmo apresentam comportamentos diferentes em situações distintas e são aqui comparadas com o algoritmo mais rápido da secção anterior, o algoritmo de Röttger et al..

No sentido de se criar uma comparação isenta, isto é, independente da métrica, que avalie apenas o desempenho do algoritmo, foi utilizada em todas as versões do algoritmo desenvolvido nesta dissertação a métrica do algoritmo de Röttger et al..

Os algoritmos comportam-se de maneira diferente consoante o *hardware* gráfico utilizado. Dada a utilização de diferentes parâmetros de métrica para a máquina `Obelix`, apenas se pode comparar o desempenho dos algoritmos nas máquinas `Jerry` e `Calvin`. Nestas últimas pode comprovar-se que o desempenho das diferentes versões do novo algoritmo varia distintamente com o *hardware* gráfico utilizado. Para as mesmas situações de teste (terreno `A` de `1025 x 1025`, percurso `circular`, velocidade rápida) a ordem pela qual se classificaram os algoritmos, em termos de velocidade, não foi a mesma nestas duas máquinas, como se pode observar na tabela seguinte (ver Figura 6-10):

	Máquinas	
Classificação	**Jerry**	**Calvin**
1º lugar	Versão com *display lists* hierárquicas	Versão com *display lists* simples
2º lugar	Versão com *display lists* simples	Versão com *display lists* hierárquicas
3º lugar	Algoritmo de Röttger et al.	Versão com coerência entre *frames*
4 lugar	Versão com coerência entre *frames*	Algoritmo de Röttger et al.

Figura 6-10: Classificação dos algoritmos no percurso circular, velocidade rápida, realizado no terreno A de 1025 x 1025, nas máquinas Jerry e Calvin.

Os gráficos seguintes mostram a taxa de FPS nas situações de teste referidas (Figura 6-11 e Figura 6-12).

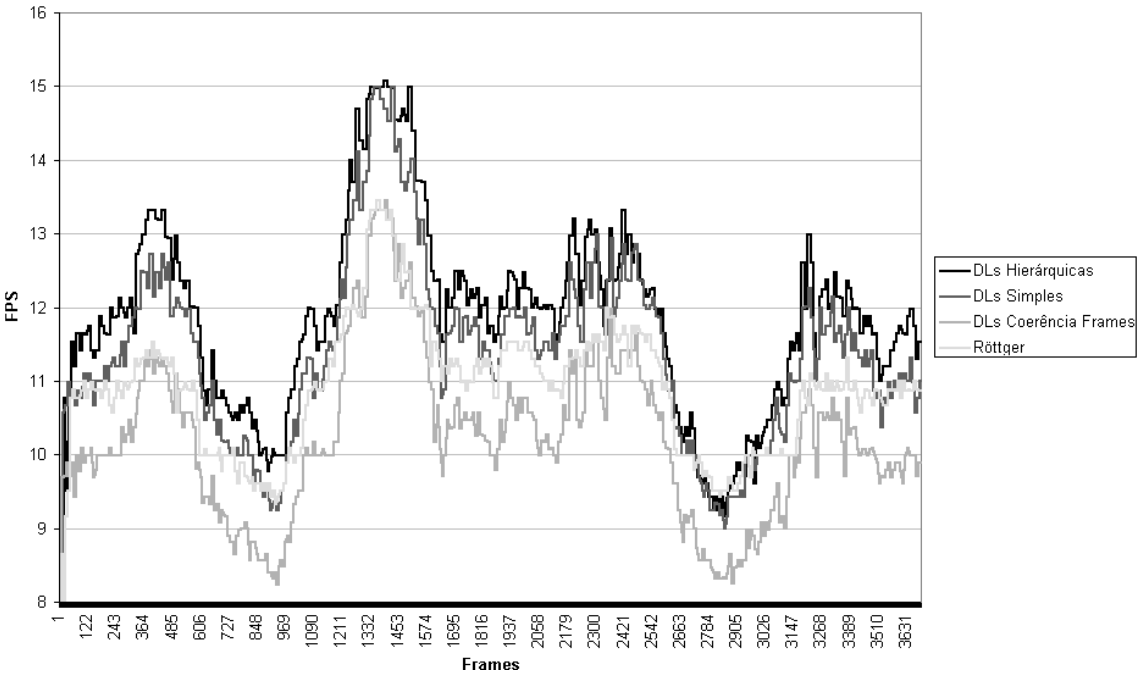


Figura 6-11: Taxa de FPS no terreno A de 1025x1025 num ciclo do percurso circular, velocidade rápida, realizado na máquina Jerry.

Figura 6-12: Taxa de FPS no terreno A de 1025x1025 num ciclo do percurso circular, velocidade rápida, realizado na máquina Calvin.

Das três versões do novo algoritmo criado no âmbito desta dissertação, a que apresentou um desempenho mais fraco em todas as situações testadas, foi a versão que tira partido da coerência entre *frames*. Este facto sugere que a maior complexidade algorítmica desta versão penaliza excessivamente o seu desempenho.

Ainda assim, esta versão revelou-se mais rápida que o algoritmo de Röttger et al. na maioria dos testes efectuados nas máquinas `Obelix` e `Calvin`, com excepção para os percursos `LB` em velocidade rápida nos terrenos de dimensão `1025 x 1025` e `2049 x 2049`, nos quais existe uma menor coerência real entre *frames*. Na máquina `Jerry` esta versão apenas superou o algoritmo de Röttger et al. nos percursos `circulares` a baixa velocidade, isto é, nos percursos com maior coerência real entre *frames*.

O desempenho das diferentes versões do novo algoritmo desenvolvido nesta dissertação está fundamentalmente condicionado pela coerência real entre *frames*, isto é, pela variação registada na triangulação de uma *frame* para a seguinte. A coerência real entre *frames* influi directamente no processo de gestão/manutenção das *display lists*, que é tanto menor quanto maior for essa coerência.

Em percursos com maior coerência entre *frames*, e logo menor variação entre imagens consecutivas, os algoritmos com *display lists* revelam-se consideravelmente mais rápidos que o de Röttger et al.. Considere-se o percurso `circular` a uma velocidade lenta pelo terreno `A` de

`4097 x 4097` na máquina `Calvin` apresentado na Figura 6-13. Nesta situação de elevada coerência real entre *frames*:

- o algoritmo com *display lists* simples é, em média, 47,4% mais rápido do que o de Röttger et al.;
- o algoritmo com *display lists* hierárquicas é, em média, 38,1% mais rápido do que o de Röttger et al.;
- o algoritmo com coerência entre *frames* é, em média, 34,1% mais rápido do que o de Röttger et al..

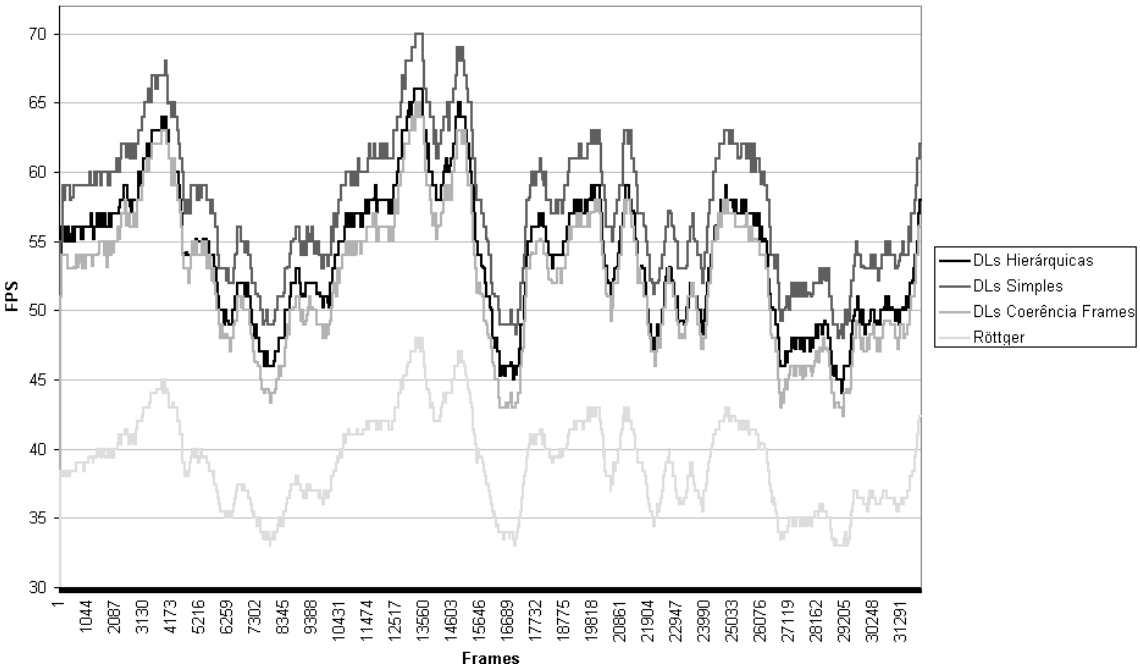


Figura 6-13: Taxa de FPS no terreno A de 4097x4097 num percurso circular, velocidade lenta, realizado na máquina Calvin.

No entanto, quando o mesmo percurso é efectuado à velocidade rápida essas diferenças médias diminuem para 23,4%, 16,3% e 13,9% respectivamente (ver Figura 6-14).

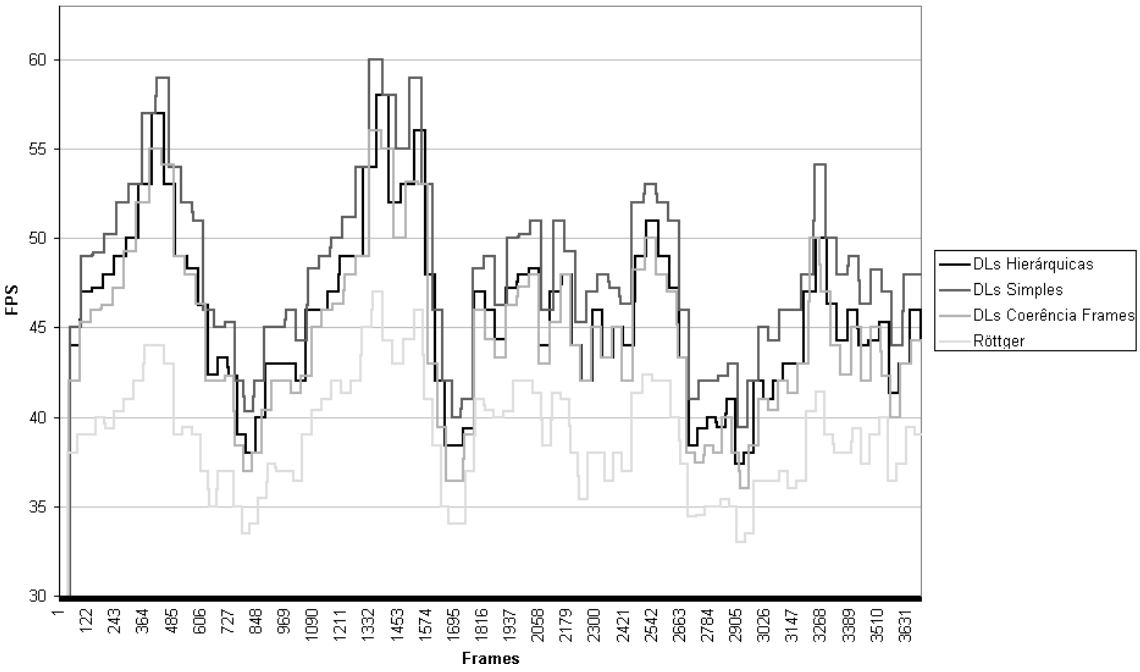


Figura 6-14: Taxa de FPS no terreno A de 4097x4097 num ciclo do percurso circular, velocidade rápida, realizado na máquina Calvin.

A variação no desempenho relativo dos algoritmos com *display lists* pode ser justificada pela menor coerência de *frames* existente no teste à velocidade rápida, donde resulta um maior número de alterações, inserções e remoções de *display lists*. Os gráficos seguintes mostram o número de alterações[57], inserções e remoções de *display lists* no teste à velocidade lenta e o aumento sofrido quando realizados à velocidade rápida (por questões de legibilidade dos gráficos apenas é apresentado um excerto de 100 *frames* do total de *frames* registado – ver Figura 6-16 a Figura 6-21).

Os valores médio e máximo registados nestas situações para o número de alterações, inserções e remoções de *display lists*[58] nas três versões do algoritmo podem ser consultados na tabela seguinte (ver Figura 6-15):

[57] O número de alterações registado inclui também as inserções de *display lists*.

[58] O número de inserções realizado na primeira *frame*, e que resulta da criação da *quadtree* inicial, não foi incluído nos valores apresentados para que não adultere os valores máximo e médio.

		Número de Alterações		Número de Inserções		Número de Remoções	
		Vel. Lenta	**Vel. Rápida**	**Vel. Lenta**	**Vel. Rápida**	**Vel. Lenta**	**Vel. Rápida**
Valor Máximo	**Versão com DLs Hierárquicas**	181	282	77	86	74	111
	Versão com DLs Simples	150	275	77	89	67	96
	Versão com Coerência entre *Frames*	150	277	77	89	67	96
Valor Médio	**Versão com DLs Hierárquicas**	26,9	161,2	2,98	29,8	2,97	29,8
	Versão com DLs Simples	25,9	152,8	3,59	35,6	3,60	35,5
	Versão com Coerência entre *Frames*	26,0	153,2	3,59	35,6	3,59	35,5

Figura 6-15: Valores máximo e médio do número de alterações, inserções e remoções de *display lists*, no percurso circular sobre o terreno A de 4097x4097, realizado na máquina Calvin.

Como se pode comprovar da análise da tabela, o número de alterações, inserções e remoções é, em média, sensivelmente igual para as versões com *display lists* simples e com coerência entre *frames*. Para a versão com *display lists* hierárquicas o número médio de alterações é superior, dada a necessidade de actualização da hierarquia. O número médio de inserções e remoções é inferior às outras versões pois a existência da hierarquia apenas obriga à criação de *display lists* quando se refina (nestes casos não há lugar a remoções) e à remoção quando se remove nodos (nestes casos não há lugar a inserções).

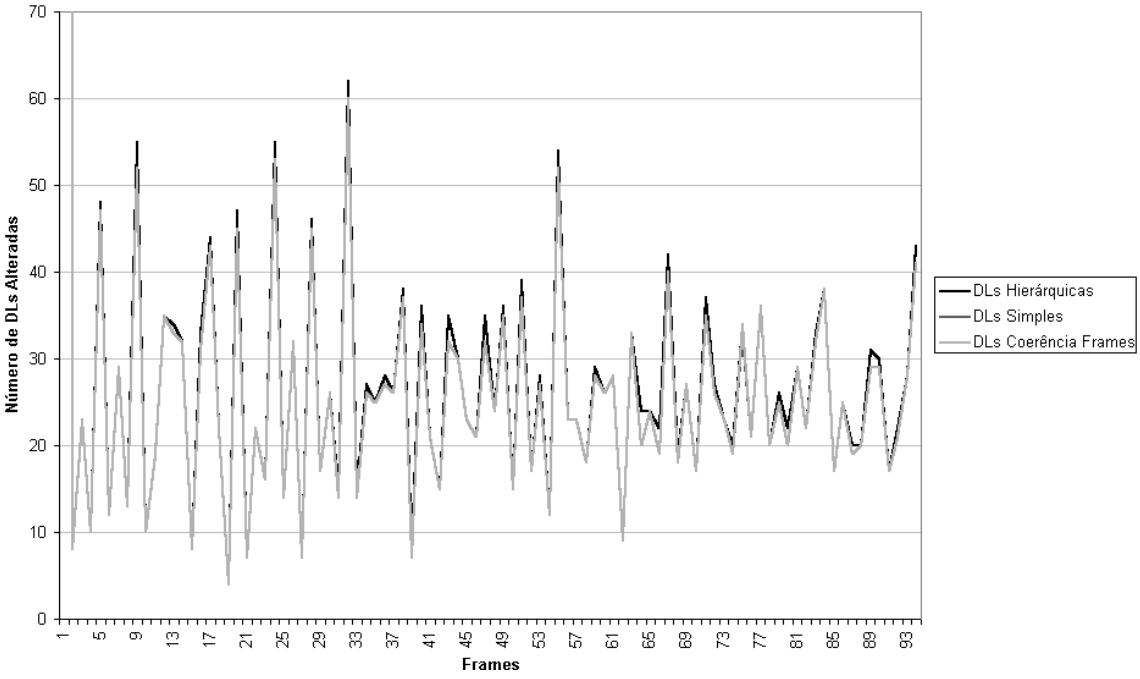


Figura 6-16: Número de *display lists* alteradas nos algoritmos, quando realizados percursos circular em velocidade lenta, no terreno A de 4097x4097 na máquina Calvin.

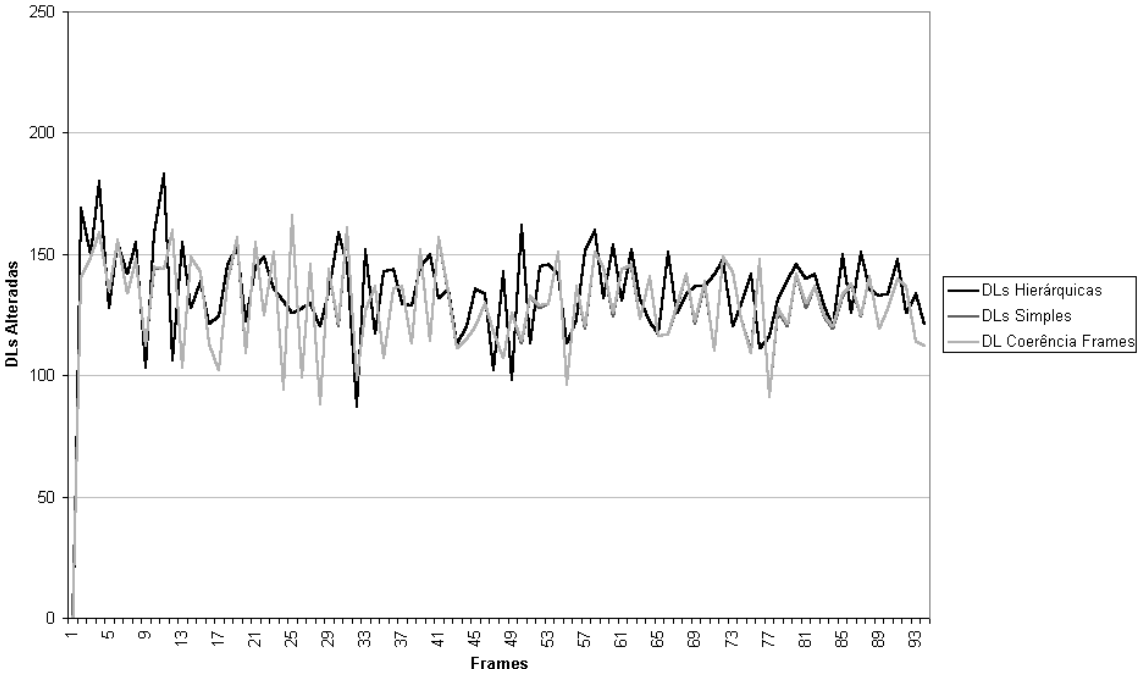


Figura 6-17: Diferença no número de *display lists* alteradas nos algoritmos, entre as velocidade rápida e lenta, para o percurso circular no terreno A de 4097x4097 na máquina Calvin.

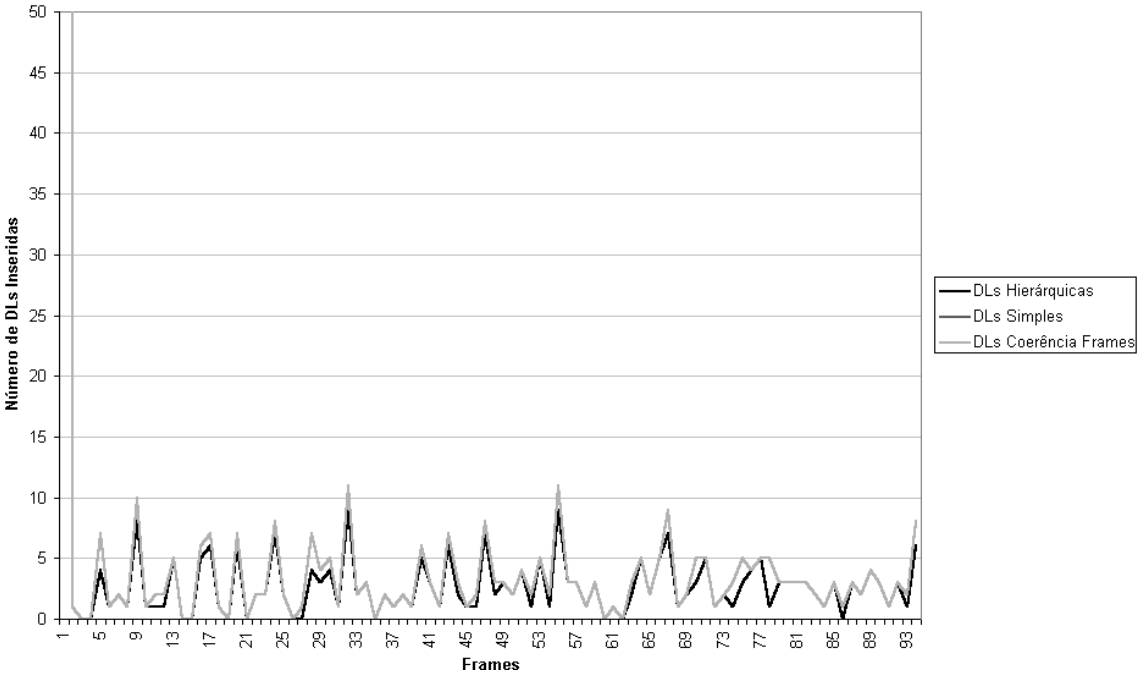


Figura 6-18: Número de *display lists* inseridas nos algoritmos, quando realizados percursos circular em velocidade lenta, no terreno A de 4097x4097 na máquina Calvin.

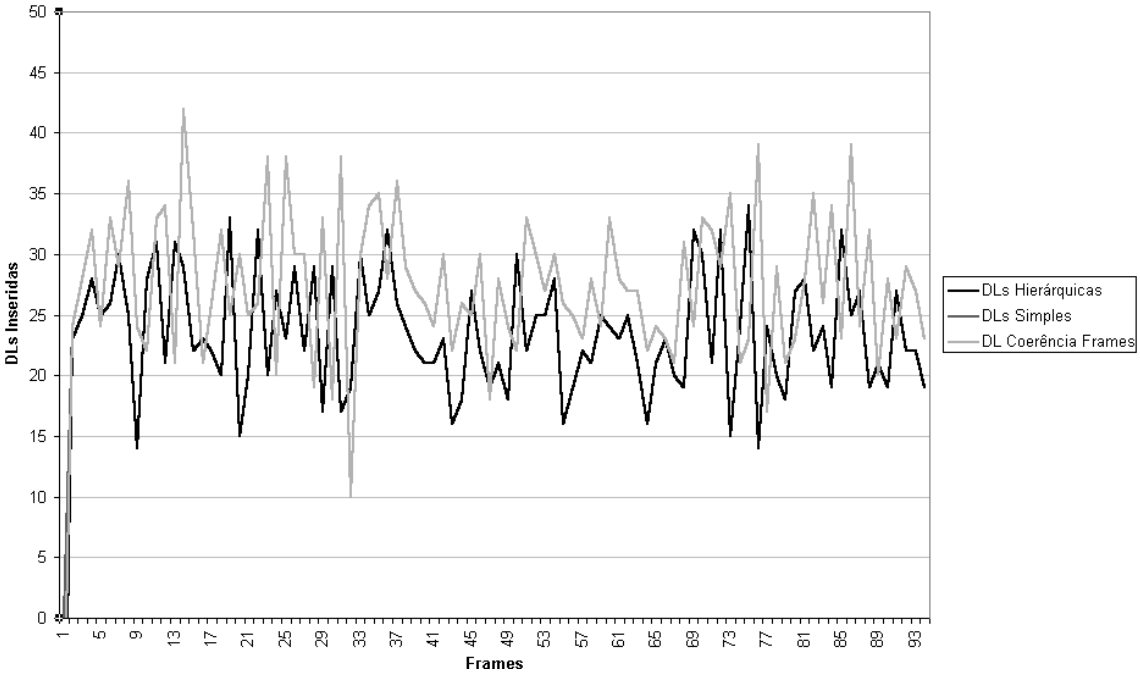


Figura 6-19: Diferença no número de *display lists* inseridas nos algoritmos, entre as velocidades rápida e lenta, para o percurso circular no terreno A de 4097x4097 na máquina Calvin.

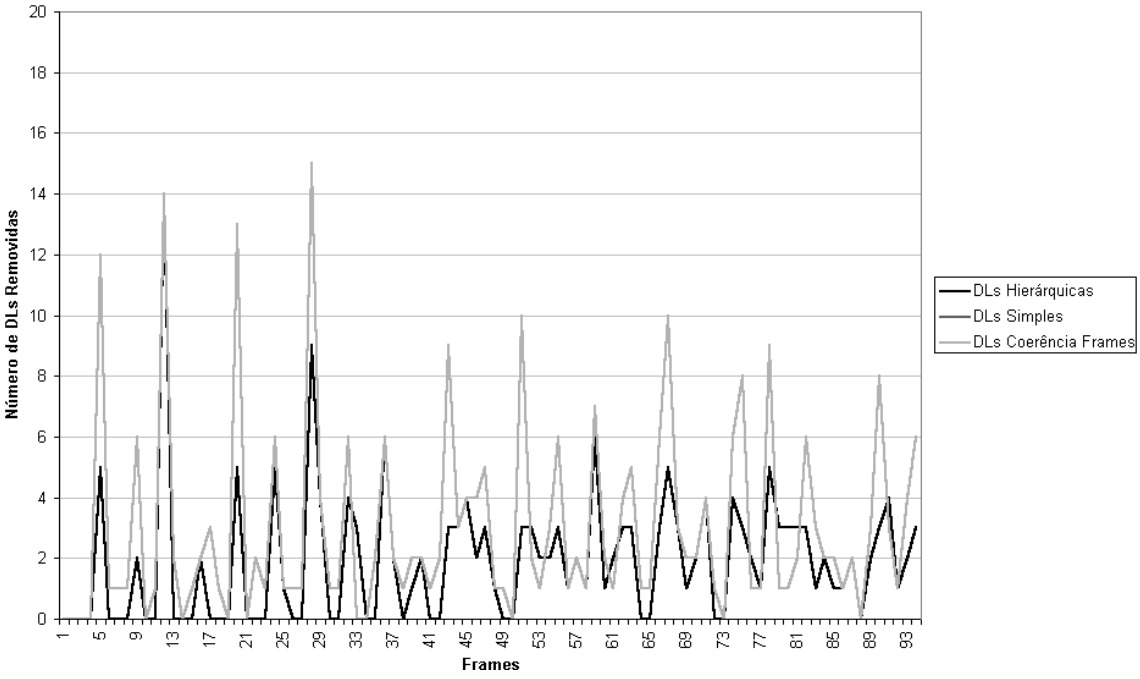


Figura 6-20: Número de *display lists* removidas nos algoritmos, quando realizados percursos circular em velocidade lenta, no terreno A de 4097x4097 na máquina Calvin.

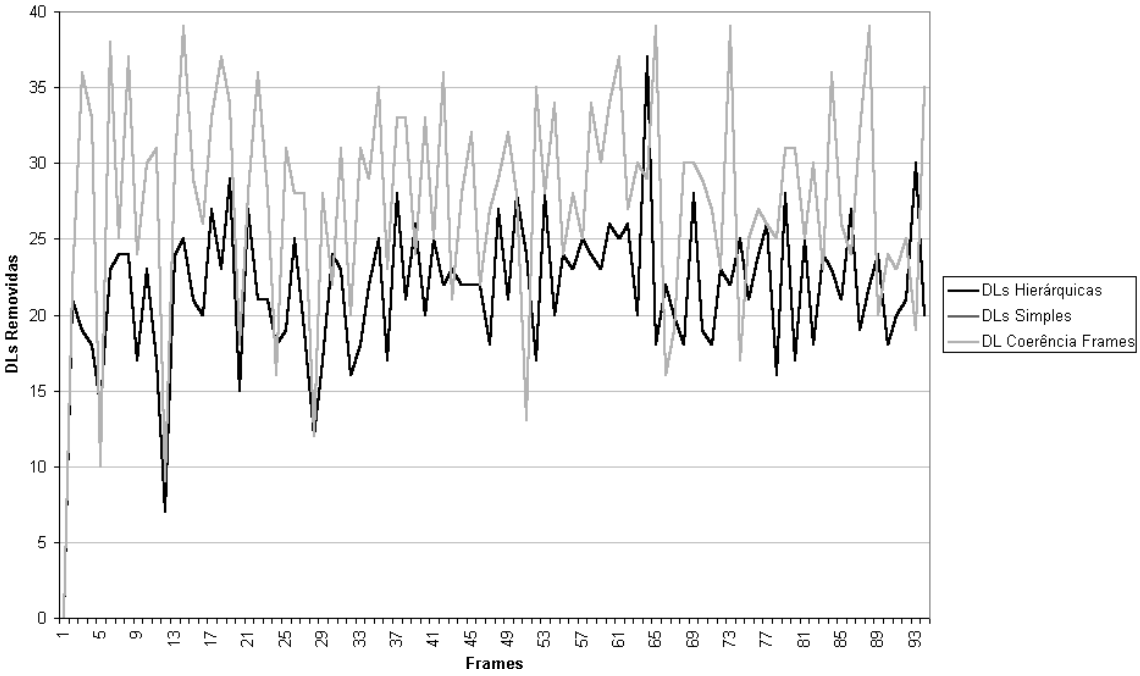


Figura 6-21: Diferença no número de *display lists* removidas nos algoritmos, entre as velocidades rápida e lenta, para o percurso circular no terreno A de 4097x4097 na máquina Calvin.

Nesta situação (e em todas as situações testadas) o número de triângulos é igual para as três versões com *display lists* e ligeiramente inferior, cerca de 1%, para o algoritmo de Röttger et al.. Esta semelhança era esperada pois foi utilizada a mesma métrica em todos estes algoritmos. A ligeira diferença registada relativamente ao algoritmo de Röttger et al. deve-se ao protelar da actualização das *display lists* cujas regiões associadas acabaram de entrar ou sair do volume de visualização.

A Figura 6-22 apresenta o número de triângulos para o percurso circular em velocidade lenta, no terreno `A` de `4097 x 4097`, realizado na máquina `Calvin`.

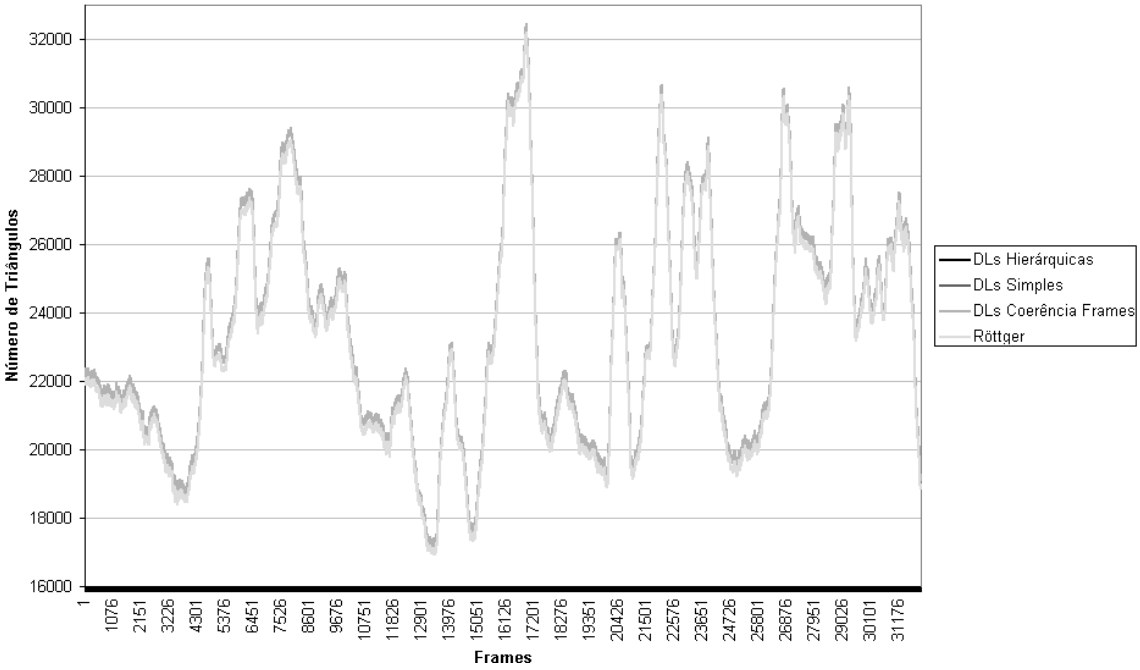


Figura 6-22: Número de Triângulos utilizados no terreno A de 4097x4097
num percurso circular, velocidade lenta, realizado na máquina Calvin.

A mesma conclusão pode ser retirada da análise ao percurso `LB` (neste caso, a própria natureza deste percurso também diminui a coerência entre *frames*) efectuado à velocidade lenta (ver Figura 6-23):

- a versão com *display lists* simples é, em média, 30,2% mais rápida do que o algoritmo de Röttger et al.;
- a versão com *display lists* hierárquicas é, em média, 22,2% mais rápida do que o algoritmo de Röttger et al.;
- a versão com coerência entre *frames* é, em média, 16,8%, mais rápida do que o algoritmo de Röttger et al..

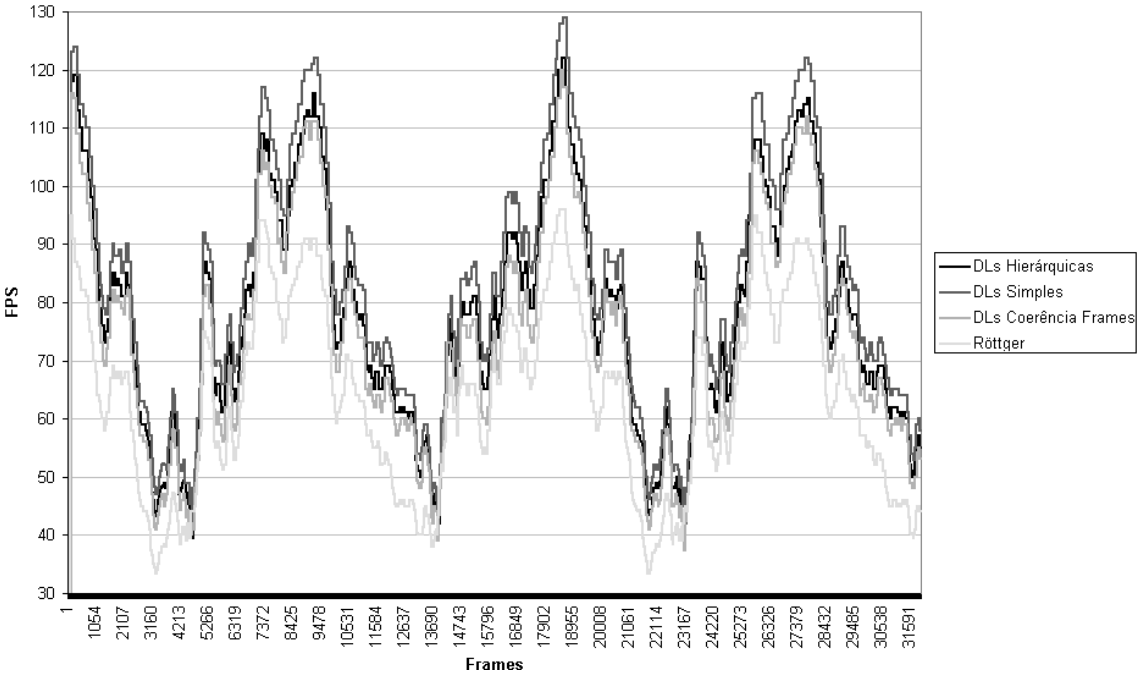


Figura 6-23: Taxa de FPS no terreno A de 4097x4097 num percurso LB, velocidade lenta, realizado na máquina Calvin.

Na velocidade rápida, esta relação inverte-se obtendo-se com o algoritmo de Röttger et al. o melhor desempenho em média. (ver Figura 6-24 – por questões de legibilidade apresenta-se apenas um excerto de 5000 *frames* do total de *frames* registada).

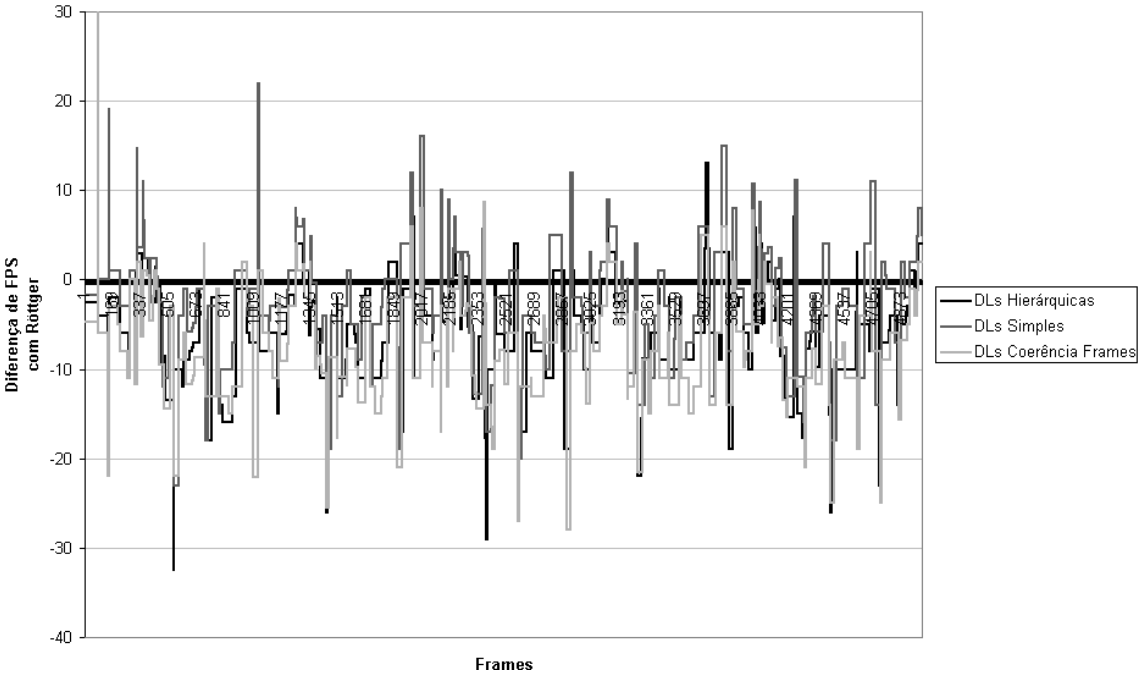


Figura 6-24: Diferença na taxa de FPS das versões do novo algoritmo para o algoritmo de Röttger et al. no terreno A de 4097x4097, no percurso LB, velocidade rápida, realizado na máquina Calvin.

Neste caso, as diferenças de desempenho são as seguintes:

- a versão com *display lists* simples é, em média, 4,8% mais lenta do que o algoritmo de Röttger et al.;
- a versão com *display lists* hierárquicas é, em média, 9,7% mais lenta do que o algoritmo de Röttger et al.;
- a versão com coerência entre *frames* é, em média, 12,6%, mais lenta do que o algoritmo de Röttger et al..

A inferioridade do desempenho dos algoritmos com *display lists* pode ser justificada pela análise ao número de alterações[59] de *display lists* apresentado nas Figura 6-25 e 6-22 (por questões de legibilidade dos gráficos apenas se apresenta um excerto de 5000 *frames* do total de *frames* registado).

[59] Inclui também o número de inserções de *display lists*.

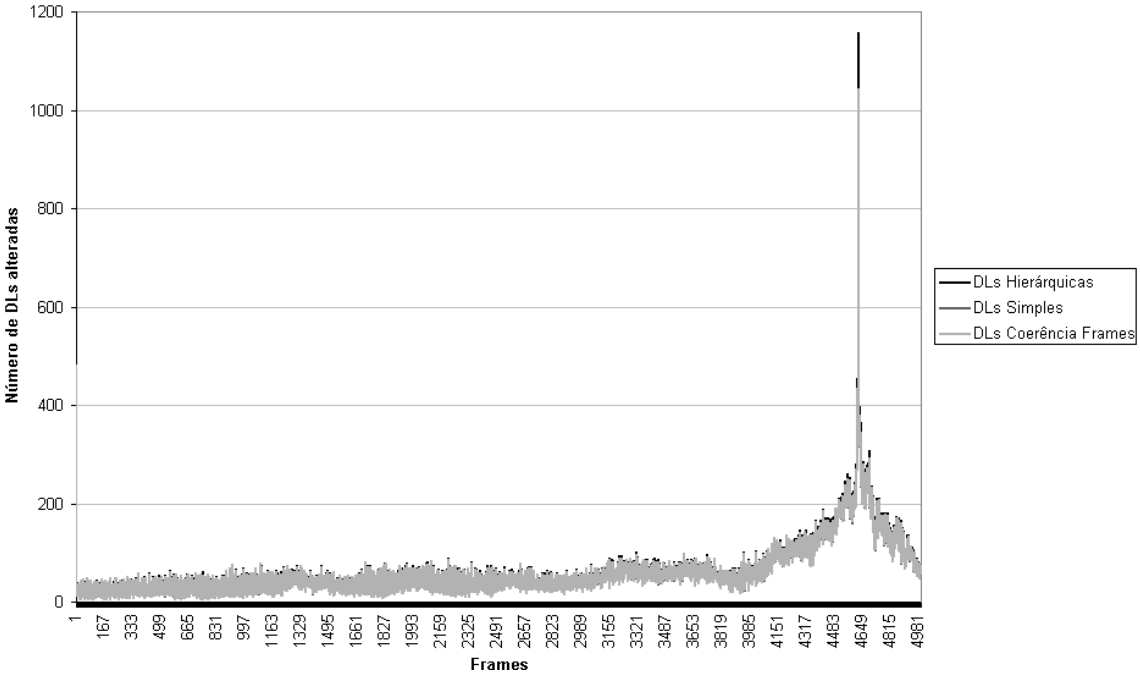


Figura 6-25: Número de *display lists* alteradas nos algoritmos, quando realizados percursos LB em velocidade lenta, no terreno A de 4097x4097 na máquina Calvin.

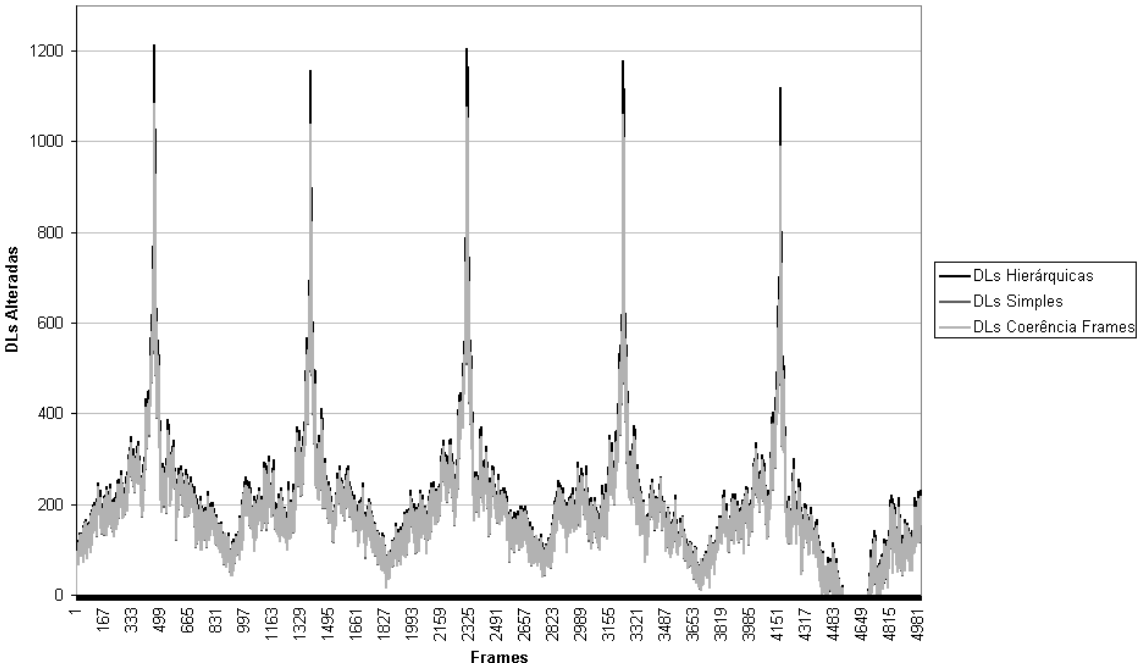


Figura 6-26: Diferença no número de *display lists* alteradas nos algoritmos, entre as velocidades rápida e lenta, para o percurso LB no terreno A de 4097x4097 na máquina Calvin.

Os valores médio e máximo registados neste percurso para o número de alterações, inserções e remoções de *display lists* nas três versões do algoritmo pode ser consultado na tabela seguinte (ver Figura 6-27):

		Número de Alterações		**Número de Inserções**		**Número de Remoções**	
		Vel. Lenta	**Vel. Rápida**	**Vel. Lenta**	**Vel. Rápida**	**Vel. Lenta**	**Vel. Rápida**
Valor Máximo	**Versão com DLs Hierárquicas**	1155	1238	848	948	889	1022
	Versão com DLs Simples	1039	1104	855	947	900	999
	Versão com Coerência entre *Frames*	1041	1108	857	952	949	1067
Valor Médio	**Versão com DLs Hierárquicas**	50,2	250,2	7,2	69,4	7,1	69,3
	Versão com DLs Simples	48,9	238,2	8,5	79,2	8,4	79,2
	Versão com Coerência entre *Frames*	49,2	240,0	8,5	79,4	8,4	80,4

Figura 6-27: Valores máximo e médio do número de alterações, inserções e remoções de *display lists*, no percurso LB sobre o terreno A de 4097x4097, realizado na máquina Calvin.

A análise às tabelas apresentadas nas Figura 6-15 e Figura 6-27, permite comprovar que o número de alterações de *display lists* é significativamente superior no percurso `LB` percorrido a velocidade rápida. Quanto maior o número de alterações às *display lists* menor é o desempenho dos algoritmos desenvolvidos nesta dissertação.

Uma avaliação às velocidades médias em cada percurso permite ter uma visão complementar sobre os testes aqui apresentados.

Como exemplo, considere-se uma taxa média de 40 FPS e 1m por cada unidade de terreno (no terreno de `4097 x 4097`). Neste caso, as velocidades médias para as diferentes situações testadas são:

- 57,6 Km/h para o percurso circular à velocidade lenta;
- 576 Km/h para o percurso circular à velocidade rápida;
- 115,2 Km/h para o percurso LB à velocidade lenta;
- 1152 Km/h para o percurso LB à velocidade rápida;

Por análise a estas velocidades, pode concluir-se que é vantajosa a utilização de algoritmos com *display lists* em situações de menor velocidade ou menor variação de imagem, como é o caso de visitas virtuais, simuladores de viação ou mesmo de voo, quando realizado a alta altitude. Já para situações de grande velocidade e, simultaneamente, grande variação de imagem, como em simuladores de voo realizado a baixa altitude, o algoritmo de Röttger et al. seria mais apropriado.

Assumindo os valores apresentados, o algoritmo de Röttger et al. só em situações extremas, na ordem dos 1000 Km/h, ultrapassa os algoritmos com *display lists*.

As conclusões retiradas dos testes realizados com o terreno `A` não se alteram com a utilização do terreno `B`. Em todos os percursos efectuados mantém-se a ordem da classificação dos algoritmos quanto ao desempenho relativo.

6.4 Conclusão

Neste capítulo foram testados três algoritmos descritos no capítulo 4 e o novo algoritmo para utilização de *display list* nas suas três variantes.

Os algoritmos com *display lists* apresentam um maior desempenho quanto maior for a coerência real entre *frames*. Nomeadamente nos percursos `circulares` a velocidade lenta testados, de elevada coerência real entre *frames*, o desempenho destes algoritmos é notoriamente superior ao algoritmo de Röttger et al.. No outro extremo estão os percursos `LB` percorridos a uma velocidade rápida. Nestes casos a coerência entre *frames* é menor e, por conseguinte, a sobrecarga da gestão das *display lists* penaliza os novos algoritmos. As situações extremas a que correspondem estas situações, permitem concluir que, na generalidade dos casos, o desempenho obtido pelos algoritmos com *display lists* é superior ao do algoritmo de Röttger et al.

Diferentes sistemas produzem resultados distintos. Na máquina `Jerry` o melhor desempenho é obtido pela versão com *display lists* hierárquicas enquanto que nas máquinas `Obelix` e `Calvin` é a versão com *display lists* simples que regista o melhor desempenho.

7 Conclusão

O estudo realizado nesta dissertação abordou a aplicabilidade do mecanismo de *display lists* nos algoritmos de visualização de terrenos em tempo real.

Tradicionalmente, os algoritmos desenvolvidos nesta área utilizam estruturas hierárquicas, construídas com base nas amostras do terrenos, com o intuito de produzirem uma triangulação simplificada que represente a superfície do terreno.

Habitualmente, estes algoritmos exploram apenas um número limitado das capacidades existentes no *hardware* gráfico, não sendo as *display lists* referenciadas em nenhuma das publicações recentes no contexto da visualização de terrenos em tempo real.

Nesta dissertação foi apresentado um novo algoritmo que utiliza *display lists* no contexto da visualização interactiva de terrenos com níveis de detalhe contínuos dependentes dos parâmetros da câmara.

A base para o novo algoritmo desenvolvido foi o algoritmo de Röttger et al. [29]que se mostrou, à partida, o algoritmo com melhor desempenho (ver capítulo *6 - Testes e Análise de Resultados*) e com uma estrutura de dados apropriada a aplicação de *display lists*.

A superfície do terreno é representada por uma triangulação sendo esta armazenada na integra em diversas *display lists*. Em cada *frame*, partes da triangulação sofrem alterações de acordo com uma métrica implicando a actualização das *display lists* que as armazenam, sem que as restantes sofram qualquer alteração.

Nesta dissertação, foram desenvolvidas e testadas três versões do algoritmo:

- A primeira utiliza *display lists* hierárquicas;
- A segunda utiliza apenas *display lists* simples ligadas por uma lista, eliminando a utilização desnecessária de uma hierarquia de *display lists*;

- A última tira partido da coerência entre *frames*, construindo a nova triangulação a partir da anterior.

Da análise dos resultados obtidos nos diversos testes efectuados, pode concluir-se que as soluções apresentadas com *display lists* apresentam melhores desempenhos em situações de elevada coerência real entre *frames* (ver capítulo *6 - Testes e Análise de Resultados*). Os resultados obtidos justificam a utilização deste algoritmo quando comparado com outros algoritmos testados. A opção por algoritmos com *display lists* é ainda mais relevante quando a aplicação corre em ambiente munidos de *hardware* gráfico mais recente.

7.1 Trabalho Futuro

O processo de desenvolvimento do novo algoritmo apresentado nesta dissertação levantou algumas questões. Nesta secção descrevem-se as que poderão conduzir a trabalho futuro realizado nesta área de investigação:

- Dimensão dinâmica das *display lists*, de acordo com uma previsão do tempo de alteração. Ao contrário de manter uma *display list* para cada nodo com filhos nulos ou folha, a profundidade da descendência associada a cada *display list* poderia variar de acordo com uma previsão de alteração para a região de terreno que esta representa. Desta forma, diminuir-se-ia a taxa de actualização das *display lists* utilizadas o que iria reflectir-se num aumento de desempenho. Esta solução apresentaria ainda um menor número de *display lists*, que conduziria a um aumento do número de triângulos (e vértices) por *display list*.

  Poder-se-ia ainda recorrer a essa previsão de alteração das regiões do terreno para evitar a utilização de *display lists* em regiões que previsivelmente fossem sofrer grandes alterações, durante as próximas *frames*. A utilização de uma *display list* para armazenar a triangulação referente a essa região poderia ser adiada enquanto fosse justificado.

- Aplicação da técnica de *morphing* para evitar descontinuidades temporais. A utilização desta técnica prevê alterações, por várias *frames* consecutivas, nas regiões de terreno modificadas até que os vértices que a representam atinjam a sua situação (posição) final. Nestes casos a utilização de *display lists* para armazenar a triangulação referente a estas regiões poderá ser protelada até que se atinja essa situação final, e evitar-se desta forma a constante actualização dessas *display lists*.

Glossário

Alpha Blending	Técnica utilizada normalmente para criar transparências. No contexto dos terrenos permite suavizar a transição entre dois níveis de detalhe contínuos.
Árvore Binária de Triângulos	Estrutura de dados que suporta o processo de divisão de um terreno em triângulos rectângulos isósceles.
Bounding Volume	Volume que contêm todos os elementos geométricos de um modelo.
***Buffer* de Profundidade**	Bloco de memória que regista os valores de profundidade para os *pixels* desenhados no ecrã.
Clipping	Processo que elimina partes das primitivas que se encontram fora do volume de visualização canónico.
Coerência entre *Frames*	Semelhança entre a triangulação de duas *frames* consecutivas.
Continuidade Espacial	Existência de uma superfície contínua, sem "buracos", na representação do terreno.
Continuidade Temporal	Existência de uma transição suave, a nível da triangulação, entre dois nível de detalhe contínuos.

Culling	Técnica que permite determinar quais os polígonos que não interferem na imagem final e evitar o seu processamento. Os tipos de *culling* mais comuns são o *backface culling*, o *view frustum culling* e o *occlusion culling*.
Display List	Uma sequência de comandos gráficos, normalmente pré-processados, que permite acelerar o processamento gráfico.
Divisão	No contexto dos algoritmos debatidos nesta dissertação, refere-se a uma operação que divide um triângulo em duas partes iguais.
Erro geométrico	Diferença entre o valor da altitude amostrada num ponto do terreno e o valor da altitude utilizada na triangulação para esse mesmo ponto.
Erro geométrico em *pixels*	Número de *pixels* correspondente à projecção no ecrã do segmento correspondente ao erro geométrico.
Frame	Imagem utilizada num dado instante de tempo para representar no ecrã o mundo tridimensional.
Fusão	Operação que funde dois triângulos no triângulo maior que lhes deu origem.
Generalização	Operação de simplificação sobre uma *quadtree* que permite diminuir a profundidade de um dado ramo. Operação inversa do refinamento.
Grafo Irregular de Triângulos	Estrutura de dados irregular que permite uma representação mais próxima da superfície de um terreno.
Leque de Triângulos	Lista ordenada de vértices a partir dos quais se criam os triângulos de acordo com uma regra pré-estabelecida. O primeiro vértice da lista é o centro do leque criado.
Mapa Regular de Alturas	Estrutura de dados regular e uniforme que se adapta facilmente à representação de terrenos.

Mipmapping	Uma sequência de texturas pré-filtradas de resoluções decrescentes que permite adequar a dimensão da textura utilizada ao número de *pixels* utilizado pelo modelo.
Níveis de Detalhe	Diferentes representações de um modelo, normalmente de resoluções distintas, que serão criadas/seleccionadas de acordo com um critério de decisão pré-determinado.
***Pipeline* Gráfico**	Todo o processo desde a especificação de uma cena tridimensional até à sua visualização num ecrã bidimensional ou em qualquer outro periférico de saída.
Quadtree	Estrutura de dados que suporta o processo de divisão de um terreno em quadrantes.
Rasterizer	Fase do *pipeline* gráfico na qual são transformadas todas as primitivas para *pixels* do ecrã.
Refinamento	Operação de divisão sobre uma *quadtree* que permite aumentar a profundidade de um dado ramo. Operação inversa da generalização.
Tira de Triângulos	Lista ordenada de vértices a partir dos quais se criam os triângulos de acordo com uma regra pré-estabelecida.
Triangulação	Conjunto de triângulos que representa um modelo.
Vertex Morphing	Técnica utilizada para evitar descontinuidades temporais na transição de níveis de detalhe, que consiste na interpolação, ao longo de um conjunto de *frames*, dos vértices da posição inicial até à posição final.
Visualização	Processo de construção da *frame* a utilizar.
Volume de Visualização	Volume que define o conjunto de polígonos projectados no ecrã.

Bibliografia

[1] J. Blow, *Terrain Rendering at High Levels of Detail*, GDC 2000.

[2] W. de Boer, *Fast Terrain Rendering Using Geometrical MipMapping*, 2000, *http://www.connectii.net/emersion*.

[3] L. Castle, J. Lanier, J. McNeill, *Real-time Continuous Level of Detail (LOD) for PCs and Consoles*, GDC 2000.

[4] B. Chen, J. Swan II, E. Kuo, A. Kaufman, *LOD-Sprite Technique for Accelerated Terrain Rendering*, In Visualization ’99 Proceeding, IEEE, pp. 291-298, 1999.

[5] M. DeLoura, *Game Programming Gems*, Charles River Media, Inc, Agosto 2000, ISBN 1-58450-049-2.

[6] M. DeLoura, *Game Programming Gems 2*, Charles River Media, Inc, Outubro 2001, ISBN 1-58450-054-9.

[7] M. Duchaineau, *Implementation Notes*, *http://www.conigraph.com/ROAM_homepage/*.

[8] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, M. Mineev-Weinstein, *ROAMing Terrain: Real-time Optimally Adapting Meshes*, In Visualization '97 Proceedings, IEEE, pp. 81-88, Outubro 1997, *http://www.llnl.gov/graphics/ROAM*.

[9] D. Eberly, *3D Game Engine Design*, Morgan Kaufmann Publishers, 2001, ISBN 1-55860-593-2.

[10] D. Ebert, F. Musgrave, D. Peachey, K. Perlin, S. Worley, *Texturing and Modeling, Second Edition*, AP Professional, 1998.

[11] W. Evans, D. Kirkpatrick, G. Townsend, *Right Triangular Irregular Networks*, Technical Report 97-09, Universidade do Arizona, 1997.

[12] T. Foley, A. Van Dam, S. Feiner, J. Hugues, *Computer Graphics – Principles and Practice, Segunda Edição*, Addison-Wesley, Reading, 1990.

[13] M. Garland, P. Heckbert, *Fast Polygonal Approximation of Terrains and Height-fields*, Technical Report CMU-CS-95-181, Universidade de Carnegie Mellon, 1995, *http://www.cs.cmu.edu/~garland/scape*.

[14] M. Garland, *Multiresolution Modeling: Survey & Future Opportunities*, EUROGRAPHICS' 99, State of the Art Reports, pp. 111-131, 1999.

[15] Y. He, *Real-Time Visualization of Dynamic Terrain for Ground Vehicle Simulation*, Ph.D Thesis, Universidade de Iowa, Dezembro 2000.

[16] P. Heckbert, M. Garland, *Survey of Polygonal Surface Simplification Algorithms*, SIGGRAPH 97 Multiresolution Surface Modeling Course, 1997, *http://www.cs.cmu.edu/~ph*.

[17] J. Hofmann, *A Survey of Real-Time Rendering Algorithms*, CSci 361, Outono 2000.

[18] H. Hoppe, *Progressive Meshes*, Proceedings de SIGGRAPH 96. In Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 99-108, Agosto 1996.

[19] H. Hoppe, *View-Dependent Refinement of Progressive Meshes*, Proceedings de SIGGRAPH 97. In Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, pp 189-198, Agosto 1997.

[20] H. Hoppe, *Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering*, IEEE Visualization 98, pp. 35-42, Outubro 1998.

[21] M. Lee, H. Samet, *Navigating through Triangle Meshes Implemented as Linear Quadtrees*, University of Maryland.

[22] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, G. Turner, *Real-Time, Continuous Level of Detail Rendering of Height-fields*, Proceedings of SIGGRAPH 96, In Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 109-118, 1996.

[23] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, R. Huebner, *Level of Detail for 3D Graphics*, Morgan Kaufmann Publishers, 2002, ISBN 1-55860-838-9.

[24] S. McNally, *Binary Triangle Trees and Terrain Tessellation*, GameDev.net, 1999, *http://www.gamedev.net/reference/articles/article806.asp*.

[25] T. Möller, E. Haines, *Real-Time Rendering*, A K Peters, Ltd, 1999, ISBN 1-56881-101-2.

[26] J. Mortensen, *Real-time rendering of height-fields using LOD and occlusion culling*, Master's Thesis, Setembro 2000, University College London.

[27] A. Ögren, *Continuous Level of Detail In Real-Time Terrain Rendering*, Master Thesis, Universidade de Umea, Suécia.

[28] OpenGL Architecture Review Board, *OpenGL Reference Manual Third Edition*, Addison-Wesley Developers Press, 1999, ISBN 0-201-65765-1.

[29] S. Röttger, W. Heidrich, P. Slusallek, H. Seidel, *Real-Time Generation of Continuous Level of Detail for Height-fields*, Proceedings of 6th International Conference on Computer Graphics and Visualization, pp. 315-322, 1998.

[30] H. Samet, *The Quadtree and Related Hierarchical Data Structures*, ACM Computing Surveys vol. 16, nº 2, pp. 187-260, Junho 1984.

[31] H. Samet, *The Design and Analysis of Spatial Data Structure*, Addison Wesley Developers Press, 1990.

[32] B. Turner, *Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*, Gamasutra, 2000, *http://www.gamasutra.com/features/20000403/turner_01.htm*.

[33] T. Ulrich, *Continuous LOD Terrain Meshing Using Adaptive Quadtrees*, Gamasutra, 2000, *http://www.gamasutra.com/features/20000228/ulrich_pfv.htm*.

[34] A. Varshney, *A Hierarchy of Techniques for Simplifying Polygonal Models*, ACM SIGGRAPH '97 Course Notes, Course 25, Agosto 1997.

[35] M. Woo, J. Neider, Tom Davis, *OpenGL Programming Guide Second Edition*, Addison-Wesley Developers Press, 1997, ISBN 0-201-46138-2.