UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÀTICA

Geração de Terrenos em Tempo Real

Ivo Alexandre Marta Leitão

MESTRADO EM INFORMÁTICA

2008

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÀTICA

Geração de Terrenos em Tempo Real

Ivo Alexandre Marta Leitão

MESTRADO EM INFORMÁTICA

2008

Dissertação orientada pela Prof. Doutora Maria Beatriz Duarte Pereira do Carmo

Declaração

Ivo Alexandre Marta Leitão, aluno nº 24021 da Faculdade de Ciências da Universidade de Lisboa, declara ceder os seus direitos de cópia sobre a sua Dissertação de Mestrado em Informática, intitulada "Geração de Terrenos em Tempo Real", realizada no ano lectivo 2007 / 2008 na Faculdade de Ciências da Universidade de Lisboa para o efeito de arquivo e consulta nas suas bibliotecas e publicação do mesmo em formato electrónico na Internet.

FCUL, 29 de Setembro de 2008

Maria Beatriz Duarte Pereira do Carmo, supervisora da dissertação de *Ivo Alexandre Marta Leitão* aluno da Faculdade de Ciências da Universidade de Lisboa, declara concordar com a divulgação da dissertação de Mestrado em Informática, intitulada "Geração de Terrenos em Tempo Real".

Lisboa, 29 de Setembro de 2008

Resumo

A geração de terrenos em tempo real é um problema complexo. Efectivamente, as necessidades de armazenamento e de processamento resultantes da quantidade de dados envolvida levantam um conjunto de problemas que tornaram esta área um tópico de investigação muito activo no domínio da computação gráfica. A maioria do trabalho efectuado concentra-se num conjunto de técnicas que procuram colmatar as dificuldades que surgem na representação de terrenos. Estas técnicas consistem sobretudo na aplicação de estratégias de *culling* e de nível de detalhe, com o intuito de reduzir o impacto que a representação de um terreno, especialmente os de grandes dimensões, tem ao nível do desempenho. Isto não obstante a grande evolução ao nível das placas gráficas que se tem verificado ao longo dos anos, mas que não tem sido, no entanto, suficiente para lidar com a tensão constante entre realismo e velocidade, entre fidelidade e número de *frames* por segundo que encontramos nesta área em particular e de uma forma geral na computação gráfica.

Nesta dissertação apresentam-se conceitos fundamentais relacionados com a geração de terrenos em tempo real, tais como a representação do terreno, o particionamento espacial, o *culling*, o *vertex caching*, a coerência espacial e temporal e a utilização de *vertex textures* no envio dos valores elevação para o GPU. Inclui-se também a descrição dos algoritmos de geração de terrenos considerados mais relevantes, seleccionando-se dois da classe *Tiled Blocks*, segundo a classificação proposta por Losasso e Hoppe, para comparar o seu desempenho. Estes algoritmos aplicam duas técnicas diferentes de nível de detalhe, bastante comuns na maioria dos algoritmos desta classe e são, respectivamente, o Geomipmapping, descrito por De Boer e o GPU Terrain Rendering, descrito por Vistnes. Avaliou-se ainda a integração da técnica de *occlusion culling* empregue no algoritmo de Terrain Occlusion Culling With Horizons, descrito por Fiedler, e a utilização das *vertex textures* como alternativa no envio dos valores de elevação para o GPU. Como ponto de referência e com o intuito de avaliar a diferença a nível de desempenho, bem como verificar a necessidade de utilizar técnicas de nível de detalhe, concretizou-se também uma aproximação de "força bruta" que não utiliza nenhuma técnica de nível de detalhe.

PALAVRAS-CHAVE: Algoritmos de Geração de Terrenos, Nível de Detalhe, Algoritmos de Oclusão, Coerência Espacial e Temporal.

Abstract

Real time terrain rendering is a complex topic. The main reason is cheer amount of geometry involved, which raises a number of problems that made this area an active topic of research in the field of computer graphics. Most of the work is centered on a group of techniques especially developed to give an answer to the problems faced when representing a terrain in real time. Culling and level of detail techniques are therefore essential tools to face the performance problems that a representation of a terrain in real time, specially a large one, brings. This despite the great technological evolution of the graphic cards over the years, which has not been, however, sufficient to deal with the constant tension between realism and speed, between fidelity and number of frames per second present in this area in particular and generally in computer graphics.

In this dissertation real time terrain rendering concepts are presented, such as terrain representation models, spatial partitioning, culling, vertex caching, spatial and temporal coherence and vertex textures. Some of the most relevant real time terrain rendering algorithms are also discussed. From these, two belonging to the Tiled Blocks class, following the classification proposed by Losasso and Hope, are compared in terms of performance. These algorithms apply two different of level of detail techniques, quite common in most algorithms of this class. They are, respectively, the Geomipmapping, described by De Boer, and the GPU Terrain Rendering, described by Vistnes. Additionally the integration of the occlusion culling technique described by Fiedler on the Terrain Occlusion Culling With Horizons algorithm is evaluated as well as the use of vertex textures as way of sending the elevation values to the GPU. As a reference point and to evaluate the impact in the performance, of level of detail techniques a brute force approach that does not apply any of those techniques was also developed.

KEY WORDS: Terrain Rendering Algorithms, Level-of-Detail, Occlusion Culling Algorithms, Spatial and Temporal Coherence.

Agradecimentos

Primeiro à Prof. Doutora Maria Beatriz Duarte Pereira do Carmo, pela disponibilidade e apoio que sempre demonstrou na orientação desta dissertação, conduzindo-me, de forma rigorosa, durante todo este trabalho. Segundo aos meus pais e à minha mulher que sempre me apoiaram, contribuindo de forma decisiva para que fosse possível chegar ao fim de mais um objectivo na minha carreira.

Aos meus: Mãe, Pai, Susana e Matilde

Índice

Lista de Figuras

Lista de Equações

Lista de Listagens

Lista de Tabelas

1. Introdução

A geração de terrenos é, hoje em dia, um importante campo de estudo no âmbito da computação gráfica. É de particular importância na implementação de simuladores de voo ou de jogos de computador bem como de sistemas de informação geográfica (SIG) e a sua contribuição para a criação de ambientes virtuais credíveis é essencial. No entanto, e independentemente do objectivo, em quase todas as situações em que é utilizada pretende-se obter um compromisso entre detalhe e desempenho, que se traduza numa experiência que seja o mais realista possível. Este equilíbrio é difícil de manter e não está ainda, infelizmente, ao alcance dos computadores existentes (pelo menos ao nível do foto realismo das imagens) muito embora, e assumindo que a lei de Moore[1] se mantém por mais uns anos, este equilíbrio deixe de ser uma questão de cedência "forçada" de cada uma das partes envolvidas. O ideal seria um sistema que permitisse navegar por terrenos extremamente detalhados em tempo-real, no qual o utilizador fosse completamente livre de se deslocar para onde deseja sem que a sua experiência fosse limitada por constrangimentos tecnológicos. De facto, ter acesso a uma experiência imersiva sem que sejam impostas barreiras artificiais, como paredes invisíveis ou mesmo artifícios elaborados como os que são utilizados em alguns jogos em que a acção decorre convenientemente em ilhas rodeadas por um mar interminável (o *Black & White* [113] e o *Far Cry* [37] são disso exemplo), é algo que não está ainda ao alcance da tecnologia actual. Assumindo então esta cedência imposta por constrangimentos tecnológicos como um facto com o qual se tem de conviver, a única solução para se chegar a um compromisso entre qualidade e fluidez que nos permita atingir os nossos objectivos é melhorar os algoritmos responsáveis pela geração de terrenos em tempo real. Deve-se para isso aproveitar o crescente poder computacional das placas gráficas, que têm passado rapidamente nos últimos anos de componente acessório a algo essencial no futuro dos computadores pessoais. São por isso, também, cada vez mais um factor diferenciador nas arquitecturas propostas exercendo desde logo uma grande influência em todos os algoritmos desenvolvidos neste âmbito. Com o intuito de ultrapassar os constrangimentos nesta área, esses algoritmos optam por sua vez, na sua grande maioria, por técnicas de *culling* (ver **3.3**) e de nível de detalhe (ver **3.4**) que tentam determinar que triângulos são necessários para representar o ângulo de visão do utilizador em relação à cena, tendo em conta para isso o efeito de perspectiva, que torna os detalhes tão menos importantes quanto mais longe se encontrar o objecto a representar. Por outro lado, como o ângulo de visão tem tendência a mudar de *frame* para *frame* e na maioria dos casos é controlado pelo utilizador existe uma variação inerente em cada uma dessas *frames* do número de triângulos a enviar para processamento que deve ser calculada. Partindo destes conceitos, o problema resume-se, assim, em achar a melhor forma de reduzir o número de triângulos a enviar para *rendering*, optando-se para isso por atribuir mais detalhe (entenda-se neste contexto triângulos) às zonas que dele mais necessitam. Essas zonas

[1] Na edição de 1965 da *Electronics Magazine* [137], Gordon Moore, um dos fundadores da Intel, previu que o número de transístores por circuito integrado duplicaria de ano para ano. Em 1970 a lei foi modificada para 18 meses tendo sido desde então quase que uma regra que rege a indústria de hardware, e uma forma de prever o aumento do poder de processamento.

são normalmente as mais próximas do ponto vista e/ou as que pelas suas características necessitam de mais detalhe para poderem ser representadas com a precisão desejada.

1.1. Áreas de Aplicação

A geração de terrenos em tempo real tem um conjunto de aplicações muito vastas, sendo por isso mesmo, tal como já foi referido, uma área de estudo muito importante em que qualquer melhoria terá bastante impacto uma vez que dela estão dependentes uma série de processos com bastante influência em diversas áreas. De acordo com [47] podemos então considerar os seguintes tipos de aplicações:

- Turismo virtual e planeamento de viagens: numa tentativa de cativar possíveis turistas, vários locais foram digitalizados e disponibilizados em *websites* ([4] e [93] por exemplo) e aplicações interactivas ([44] e [96] por exemplo) recorrendo-se para isso a dados topológicos reais. Tal permite ao utilizador obter uma visão muito mais detalhada de cada um dos destinos bem como do que poderá encontrar em cada um deles. É por isso, um meio único de publicidade que tem tendência a ser cada vez mais explorado.
- Educação: por exemplo, o *Virtual Field Course* [205] foi desenvolvido especificamente com intuitos educativos, servindo de suporte ao trabalho de campo dos estudantes. Para isso, fornece um ambiente visual de exploração de informação espacial bem como um conjunto de exercícios e notas para professores.
- Planeamento rural e urbano: a tomada de decisão pode ser claramente influenciada por uma visão mais abrangente dos dados nomeadamente quando a análise não é possível sem uma representação visual convincente. O planeamento de colheitas e a gestão de florestas são exemplos de análises de dados que podem ser efectuadas muito mais precisamente se para isso nos socorrermos de uma representação detalhada da área abrangida, principalmente se for possível efectuar alterações e de uma maneira geral interagir com os dados.
- Engenharia civil e de telecomunicações e outras infra-estruturas: a colocação de estruturas pode ser devidamente planeada se for possível recorrer a modelos precisos que permitam uma visualização das zonas afectadas.
- Meteorologia: os fenómenos atmosféricos podem ser mais facilmente estudados se pudermos simular o seu efeito; por exemplo, uma das aplicações poderá ser o estudo do impacto de taxas de pluviosidade excessivas no leito dos rios para obter informações sobre as zonas circundantes e determinar se estas podem ser afectadas em caso de inundação.
- Comunicações: por exemplo, na instalação de transmissores (rádio, televisão ou celulares).
- Arquitectura virtual: por exemplo, na contextualização de um modelo CAD de uma habitação na zona onde vai ser construída, permitindo-se a navegação por essa zona para visualização de todo o ambiente envolvente.
- Jogos: um grande conjunto de novas ideias introduzidas na geração de terrenos teve a sua origem em algoritmos utilizados em jogos (por exemplo, as alterações [198] introduzidas por Seumas McNally no algoritmo ROAM (ver **4.2**) para o jogo Tread Marks [116]). Hoje em dia, assistimos a um mercado extremamente

competitivo que já atingiu um estatuto que o eleva acima da indústria cinematográfica (nomeadamente em termos de ganhos). Cada jogo é uma superprodução que custa muitos milhões de euros e, como é sabido, a procura do lucro esteve muitas vezes ao longo da história na origem de inovações.

- Diplomacia: por exemplo, como auxiliar nas negociações de paz na Bósnia [79] o Pentágono desenvolveu uma representação de um terreno em realidade virtual que permitiu aos negociadores obterem uma vista aérea sobre as fronteiras propostas. Esta representação permitiu ao presidente Sérvio tomar uma decisão, aceitando criar um corredor mais largo entre Sarajevo e o conclave islâmico em Gorazde depois de verificar que as montanhas tornavam um corredor mais estreito impraticável.
- Indústria de defesa: a indústria da defesa teve sempre uma grande influência e, principalmente, meios para criar inovações. Muito embora, os objectivos finais não sejam os mais nobres, vários projectos nesta área são hoje em dia financiados pelo exército permitindo o treino de soldados e pilotos e utilizando para isso, terrenos virtuais baseados em dados topológicos recolhidos por satélite.

1.2. Objectivos

O principal objectivo desta dissertação foi o estudo de dois algoritmos de geração de terrenos em tempo real: o Geomipmapping [39] e o GPU Terrain Rendering [207]. Estes pertencem, de acordo com a classificação descrita em [120] e aqui adoptada, à classe *Tiled Blocks*. Foram escolhidos por aplicarem duas das técnicas mais comuns no tratamento do nível de detalhe, partilhadas por diversos algoritmos dessa classe, e por tirarem partido das capacidades das placas gráficas mais recentes. No entanto, o nível de detalhe foi apenas uma vertente do trabalho realizado, na medida em que se pretendeu analisar outros factores relacionados com a geração de terrenos em tempo real.

Para melhor se contextualizar este tema, efectuou-se um resumo do estado da arte com a descrição de todas as outras classes consideradas em [120] bem como dos algoritmos que representaram algumas das abordagens mais bem sucedidas em cada uma delas, realçando-se a grande diferença que existe entre a forma como se encarou este tema antes e depois de as placas gráficas começarem a assumir um papel mais relevante.

Desta forma, para os dois algoritmos considerados, pretendeu-se, por um lado, avaliar o seu desempenho face às placas gráficas mais recentes e, por outro, abordar outros factores directa ou indirectamente relacionados com estes algoritmos e de um modo geral com a geração de terrenos em tempo real, tais como a representação do terreno, o particionamento espacial, o *culling*, a gestão do nível de detalhe, a coerência espacial e temporal. Para isso, todos estes temas são apresentados no decurso desta dissertação com o intuito de fundamentar algumas das opções tomadas em cada um deles.

Para se efectuar este estudo desenvolveu-se uma aplicação em XNA que concretiza os algoritmos mencionados. A utilização do XNA permitiu acima de tudo avaliar o desempenho numa plataforma de desenvolvimento em que a libertação da memória é feita por um *garbage collector*, representando consequentemente uma abordagem totalmente diferente da tradicional implementação numa linguagem nativa. Por outro lado,

como o XNA permite a utilização da consola XBOX 360 da Microsoft representou uma oportunidade para se fazer nessa máquina um estudo do desempenho obtido.

Em relação aos outros factores, um ao qual se deu especial destaque foi ao *culling*, isto é, a remoção de blocos de geometria não visível numa perspectiva de impacto no desempenho. A motivação esteve no facto de na maior parte dos algoritmos se ignorar o problema da oclusão, isto é, da detecção de blocos ocultos por outros blocos, o *occlusion culling*. Para colmatar esse problema, aplicou-se a técnica de oclusão do algoritmo de Terrain Occlusion Culling With Horizons [63]. Esta foi integrada em cada um dos outros algoritmos de modo a se poder avaliar o impacto deste tipo de *culling* no desempenho e mais uma vez aferir a utilidade de algoritmos como este face às placas gráficas mais recentes.

Atendendo ainda à grande evolução das placas gráficas este trabalho foi também uma oportunidade para incorporar algumas das novas funcionalidades, nomeadamente as *vertex textures*, enquanto forma alternativa de enviar os valores de elevação para a placa gráfica, funcionalidade que está disponível apenas no *shader model* 3.0. Nesta perspectiva, pretendeu-se avaliar a aplicabilidade desta técnica em algoritmos da classe *Tiled Blocks* e também a sua influência ao nível do desempenho nos resultados obtidos.

1.3. Convenções

Nesta dissertação adoptou-se um conjunto de convenções que importa referir. Assim, e começando pelo texto propriamente dito, o tipo de letra utilizado foi Times New Roman para o texto em geral, Cambria Math para equações e `Courier New` para listagens de código. Todos os termos estrangeiros serão apresentados em *itálico* excepto se forem nomes de algoritmos ou técnicas. Sempre que existir uma referência a uma secção de texto, a uma figura, equação, listagem de código ou tabela estas vão ser apresentadas a **negrito**, com o seguinte formato: (ver **1.3**), **Figura 2-1**, **Equação 3-1**, **Listagem 3-1**, **Tabela 2-1**. Por outro lado a **negrito**, é também colocada a primeira letra de cada secção para marcar o início da mesma.

Estabelece-se ainda a nível geométrico que sempre que for feita uma referência ao valor de elevação ou a uma altura, este será associado ao eixo dos yy de um sistema de coordenadas tridimensional com eixos (x, y, z).

1.4. Organização do Documento

Esta dissertação está organizada em 7 capítulos e um anexo que abordam os temas mais relevantes na geração de terrenos em tempo real. Assim, e descrevendo cada um deles em mais detalhe:

- No capítulo 1, ao qual pertence esta secção é feita uma introdução ao tema da geração de terrenos em tempo real, realçando-se as múltiplas aplicações que o trabalho nesta área tem e também os objectivos que esta dissertação se propõe a atingir. São também estabelecidas as convenções adoptadas no texto.
- No capítulo 2, é abordada a representação do terreno, dando-se especial destaque às estruturas de dados utilizadas para esse efeito, bem como aos diferentes formatos de dados utilizados no armazenamento. São também discutidas as diferentes formas de obter os valores de elevação para construir o terreno.

- No capítulo 3, descreve-se, em detalhe, o processo de geração do terreno e os factores mais importantes aí envolvidos. Assim, fala-se da construção da malha triangular especialmente das diferentes primitivas gráficas que podem ser utilizadas para esse efeito. Depois, introduzem-se uma série de conceitos importantes, nomeadamente o particionamento espacial, o *culling* e o tratamento do nível de detalhe, todos eles especialmente relevantes neste contexto. Destaca-se também a *quadtree* enquanto estrutura mais utilizada para particionamento espacial de terrenos, o *frustum culling* e o *occlusion culling* como as técnicas de *culling* mais relevantes e, descrevem-se diferentes abordagens utilizadas na gestão do nível de detalhe em terrenos.
- No capítulo 4, é feito um resumo do estado da arte dos algoritmos de geração de terrenos em tempo real. Para isso apresenta-se uma classificação dos mesmos e descrevem-se alguns deles, sobretudo os algoritmos da classe *Tiled Blocks* mas também outros pertencentes às restantes classes e que se considerou serem especialmente relevantes. Os algoritmos descritos são: ROAM, Real-Time generation of Continuous Levels of Detail, Geomipmapping, Chunked LOD, Terrain Occlusion Culling with Horizons, Rendering Very Large, Very Detailed Terrains, GPU Terrain Rendering, Geometry Clipmaps e finalmente o GPU Based Geometry Clipmaps.
- No capítulo 5, descreve-se em detalhe todo o trabalho desenvolvido no âmbito desta dissertação com intuito de se atingir os objectivos estipulados em **1.2**. Este inclui uma descrição detalhada de todos os passos efectuados em cada uma das fases consideradas para se gerar um terreno em tempo real com cada um dos algoritmos considerados, nomeadamente o Geomipmapping, o GPU Terrain Rendering e uma aproximação sem qualquer tipo de redução de nível de detalhe que por uma questão de coerência de designações se denominou de Brute Force.
- No capítulo 6, os resultados obtidos nos testes realizados com a aplicação desenvolvida em XNA são discutidos em detalhe e apresentadas as conclusões a que se chegou.
- No capítulo 7, é feita a conclusão deste trabalho apresentando-se de forma resumida os resultados mais importantes obtidos no contexto do trabalho realizado, e também algum do trabalho futuro a realizar no âmbito deste tema.
- No anexo A1, é abordada a síntese de terrenos nomeadamente a criação por intermédio de abordagens procedimentais dos valores de elevação, discutindo-se nesse contexto os algoritmos mais relevantes e sobretudo a importância do conceito de fractal nesta área.

2. Representação do Terreno

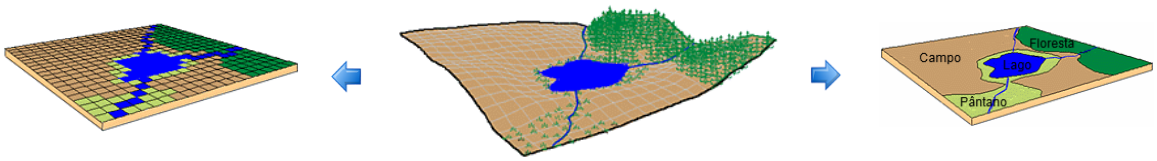
Antes de se construir o terreno há que perceber primeiro como representá-lo, que estruturas de dados utilizar, quais os formatos mais comuns de armazenamento da informação e sobretudo como obter os dados que definem a estrutura do terreno. Esse tema é abordado neste capitulo, começando-se em **2.1** com uma aproximação mais generalista em que se descreve as estratégias adoptadas nos sistemas de informação geográfica (SIG) para modelar a realidade e caracterizar as entidades que a povoam. É a partir dos modelos aí definidos que se define em **2.2** as estruturas de dados mais utilizadas na modelação de terrenos, mais especificamente a grelha regular (ver **2.2.1**) e a *triangulated irregular network* ou TIN (ver **2.2.2**) e se comparam as duas aproximações (ver **2.2.3**).

Descrevem-se de seguida em **2.3** os formatos de dados mais comuns no armazenamento das características do terreno, nomeadamente as imagens (ver **2.3.1**), mas também formatos específicos (ver **2.3.2**) que armazenam informações adicionais de carácter geográfico.

Finalmente em **2.4** foca-se os métodos utilizados para obter os dados de elevação com especial destaque para as abordagens procedimentais que permitem a criação dos valores de elevação por intermédio de um algoritmo.

2.1. Modelos de Dados

Para modelar a realidade temos de caracterizar as entidades que a povoam. Consideram-se para isso dois tipos de abstracções: entidades discretas e entidades contínuas. Entidades discretas (por exemplo casas ou torres) são entidades com fronteiras bem definidas sendo por isso fáceis de representar. As entidades contínuas (por exemplo terreno ou mar) variam continuadamente no espaço, pelo que é mais difícil capturar a informação necessária para as representar. No mundo dos sistemas de informação geográfica (SIG) adoptaram-se dois tipos de abordagens [5][143]: o modelo *raster*, em que a realidade é representada como uma grelha de células com um valor em cada ponto da grelha e o modelo vectorial, em que a realidade é representada com um conjunto de primitivas, nomeadamente: pontos, linhas e polígonos. Na representação dos terrenos, qualquer um dos modelos é válido, muito embora, o modelo *raster* seja definitivamente o mais comum. Na **Figura 2-1** é visível a diferença que existe entre os dois modelos. O modelo *raster* "dispõe" uma grelha virtual sobre todo o terreno, permitindo a recolha de um valor por cada célula e o modelo vectorial identifica as diferentes regiões que caracterizam o terreno descrevendo-as com recurso ao conjunto de primitivas de que dispõe.



Raster Terreno Vector

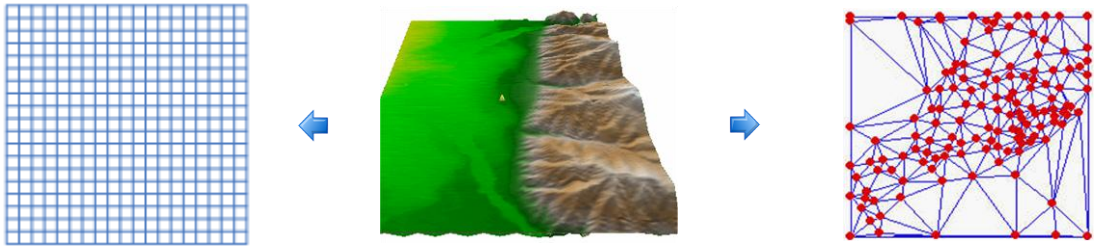
Figura 2-1: Os dois modelos de representação [77].

Comparando de uma forma mais elaborada cada um dos modelos de dados:

- O foco do modelo vectorial é a característica geográfica, no modelo *raster* é a localização.
- O modelo vectorial procura responder á questão "O que sabemos acerca desta característica geográfica?", ao passo que o modelo *raster* responde a questão "Que fenómeno geográfico ocorre nesta localização?".
- O modelo vectorial define fronteiras, o modelo *raster* não.
- O modelo vectorial representa as características de uma forma mais precisa, o modelo *raster* utiliza células para representar zonas, pelo que é mais generalista e consequentemente menos preciso.

2.2. Modelação de Terrenos

Partindo dos modelos de dados discutidos em **2.1** aborda-se nesta secção as estruturas de dados mais comuns na modelação de terrenos (ver **Figura 2-2**): a grelha regular, também designada de DEM (*Digital Elevation Model*), no contexto do modelo *raster*, e a TIN (*Triangulated Irregular Network*) no contexto do modelo vectorial. Nas secções seguintes (ver **2.2.1** e **2.2.2**) vão ser discutidas em detalhe cada uma delas, bem como as vantagens e desvantagens (ver **2.2.3**) que advêm da sua utilização.



Grelha regular Terreno *Triangulated Irregular Network*

Figura 2-2: As duas estruturas de dados [192].

2.2.1. Grelha Regular

No modelo *raster* (ver **2**) o espaço é dividido numa grelha em que a área abrangida por cada uma das células é representada por um valor de elevação. O conjunto dos

valores de elevação obtidos em cada uma das células pode ser representado por uma grelha regular. Nas grelhas regulares, cada um dos valores pode ser acedido por um índice (i, j) em 2D ou (i, j, k) em 3D e os vértices têm coordenadas $(i.dx, j.dy)$ em 2D e $(i.dx, j.dy, z.dz)$ em 3D, representando dx, dy e dz o espaçamento da grelha em cada uma das dimensões. As grelhas regulares são topológica e geometricamente regulares e, consequentemente, cada um dos vértices está definido de forma implícita, não sendo necessário definir explicitamente as ligações entre eles, nem armazenar as coordenadas dos mesmos. Este conceito nos terrenos assume a forma de uma grelha regular em 2D, também designada por grelha regular de elevações ou campo de elevações (*height field*) no qual, a cada vértice corresponde um valor obtido na fonte elevação, tal como está representado na **Figura 2-3**. Nesta figura é disposta uma grelha virtual sobre o terreno, de acordo com o modelo *raster*, e recolhido um valor por célula (normalmente o mais representativo da zona de terreno abrangida) que vai constituir com os outros valores, a grelha regular que representa o terreno.

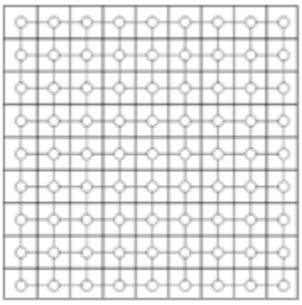


Figura 2-3: Grelha regular utilizada na representação de uma fonte de elevação.

Este valor é depois utilizado na construção do terreno, constituindo efectivamente a componente y das coordenadas. Para espaçamento são normalmente considerados valores iguais no eixos dos xx e dos zz. Estas características contribuem para que este seja definitivamente o formato mais utilizado devido às inúmeras vantagens que oferece nomeadamente, ao nível da facilidade de utilização e de armazenamento. Uma das suas principais características está precisamente relacionada com a facilidade de armazenamento, já que é necessário guardar apenas o valor de elevação correspondente à coordenada y e não o conjunto de coordenadas, (x, y, z), isto porque as coordenadas x e z são implicitamente obtidas a partir do número de pontos e dos espaçamento segundo cada uma das dimensões. Por outro lado, a sequência de valores de que é composta pode ser directamente representada por estruturas de dados nativas nas linguagens de programação (pode ser armazenada num *array*, por exemplo), sendo facilmente convertida para outras representações. Outro factor importante tem a ver com a quantidade de técnicas de nível de detalhe (ver **3.4**), *culling* (ver **3.3**), detecção de colisões e de deformações dinâmicas já desenvolvidas para este tipo de estruturas, em grande parte devido às facilidades de utilização já referidas. Como se pode observar na **Figura 2-4**, para representar o terreno é criada uma grelha regular 20×20 que é "disposta" sobre este, correspondendo a cada vértice um valor de elevação. É fácil

verificar também que a precisão depende da resolução da grelha, ou seja, a quantidade de vértices representa o número de valores de elevação recolhidos.

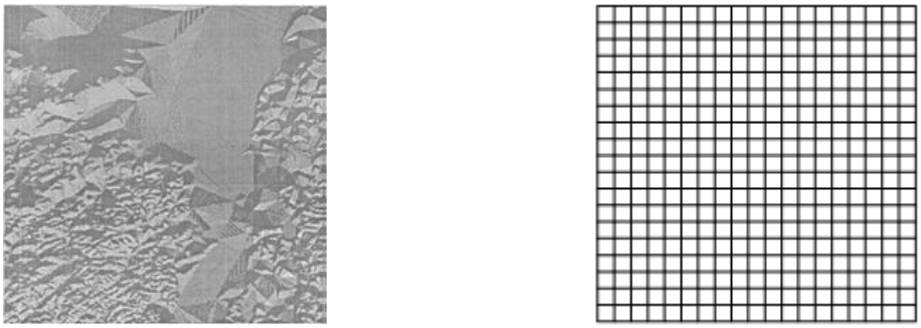


Figura 2-4: Uma textura de um terreno e a grelha regular correspondente [154].

Esta estrutura tem, no entanto, um conjunto de limitações, nomeadamente não permite mais do que um valor de y na mesma posição (x, z), não sendo por isso possível representar, por exemplo, cavernas, túneis subterrâneos ou outras formações como a ilustrada na **Figura 2-5**. Isto é, não é possível a sobreposição de nenhum valor y, ou dito de outra forma no contexto dos terrenos estas não permitem a sobreposição de nenhum valor de elevação para uma determinada posição espacial. Outra característica negativa está relacionada com o facto de todas as zonas do terreno terem a mesma densidade de vértices, mesmo que os vértices vizinhos tenham a mesma elevação. Este facto significa que é desperdiçado espaço na representação de, por exemplo, vastas zonas planas, que nas *triangulated irregular networks* são representadas com muito menos triângulos (comparar a **Figura 2-4** e a **Figura 2-7**).

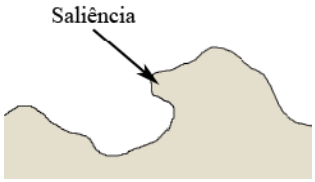


Figura 2-5: Exemplo de uma saliência não representável numa grelha regular [212].

2.2.2. Triangulated Irregular Network

A *triangulated irregular network* (TIN) foi desenvolvida para armazenar de uma forma inteligente dados contínuos num modelo vectorial. Pode ser definida como um conjunto de triângulos adjacentes que são obtidos a partir de pontos irregularmente espaçados com coordenadas x e z horizontais e y elevações verticais, permitindo um espaçamento variável entre vértices, ao contrário das grelhas regulares. Por isso, são topológica e geometricamente irregulares, o que significa que é necessário armazenar as coordenadas de cada vértice e as ligações entre eles (triângulos). Os pontos que as compõem correspondem ao melhor conjunto de elevações numa perspectiva de se obter os dados mais representativos de variações no terreno, como, por exemplo, picos ou

depressões [143]. O próximo passo consiste numa triangulação do conjunto de pontos seleccionados, algo que é normalmente efectuado utilizando o método de Delaunay [43]. Neste método três pontos formam um triângulo de Delaunay, se o círculo que passa por eles não contém nenhum outro ponto (ver **Figura 2-6**). A implementação em si pode ser efectuada de diversas formas, existindo um grande conjunto de algoritmos para esse efeito. Uma comparação detalhada de algoritmos de triangulação de Delaunay pode ser encontrada em [189].

Figura 2-6: Triangulação de Delaunay.

Na **Figura 2-7** é bem visível que este tipo de estrutura pode aproximar uma superfície com menos triângulos, especialmente nas zonas do terreno que necessitam de menos detalhe por serem inerentemente planas. Pelo contrário, nas outras zonas onde existe mais detalhe, são usados mais triângulos. Esta permite ainda a representação de saliências (ver **Figura 2-5**), ao contrário das grelhas regulares, isto é a sobreposição de valores em y. Intuitivamente, e tendo em conta as vantagens que apresenta, este seria o modelo mais correcto já que além do detalhe ser atribuído onde é mais necessário, permite ainda representar determinados tipos de terreno que não são possíveis de representar através de uma grelha regular. No entanto, a dificuldade adicional que traz ao nível da implementação de outras técnicas em fases posteriores anula em grande parte esta aparente vantagem. Por exemplo, é mais complicado calcular a intersecção para efeitos de detecção de colisões com este tipo de estruturas. Sem dados adicionais, pode implicar a pesquisa de triângulos em estruturas de dados complexas para se efectuar os testes de intersecção. O mais preocupante é o facto de a construção e actualização das *triangulated irregular networks* implicar um esforço computacional significativo. Trabalhos de investigação prévios [110][209] sugerem *inclusive* que não é possível gerar diferentes níveis de detalhe para estas estruturas em tempo real. São por isso construídas normalmente numa fase de pré-processamento.

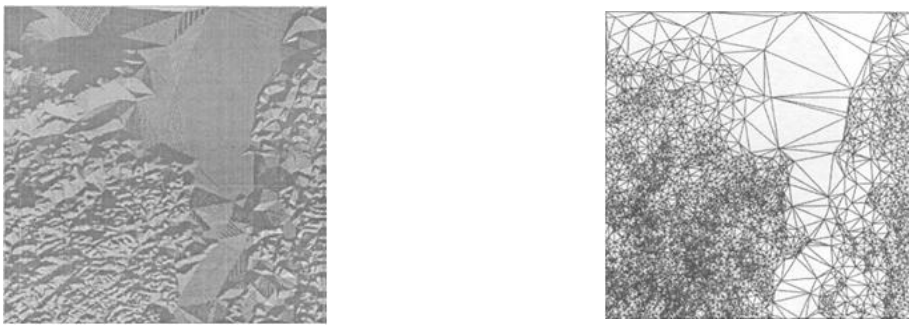


Figura 2-7: Uma textura de um terreno e a TIN correspondente [154].

2.2.3. Grelha Regular Vs TIN

Na **Tabela 2-1** é feita uma comparação entre as duas estruturas de dados utilizadas para a representação e armazenamento de terrenos: as grelhas regulares e as *triangulated irregular networks* enunciando-se para isso as vantagens e desvantagens de cada uma delas.

	Grelha Regular	**TIN**
Vantagens	• Fáceis de armazenar e de manipular. • A integração com a grande maioria dos formatos de dados existentes é mais fácil. • Facilita a implementação da detecção de colisões porque a intersecção entre um raio e a grelha pode ser calculada em $O(1)$. • A estrutura regular torna a utilização de uma *quadtree* (ver **3.2.3.3**) mais fácil para implementação de *culling* (ver **3.3**).	• Permite espaçamento variável entre os vértices. • Permite a sobreposição de valores de elevação.
Desvantagens	• Não permite mais do que um valor de y na mesma posição (x, z). • Não permite a utilização de diversos espaçamentos para representar zonas com diferentes níveis de complexidade.	• É preciso armazenar o conjunto de coordenadas (x, y, z) para cada ponto. • Torna mais complexa a implementação de técnicas de nível de detalhe (ver **3.4**), *culling* (ver **3.3**) e de detecção de colisões. • Não é compatível com a maioria dos formatos de dados.

Tabela 2-1: Comparação entre uma grelha regular e uma TIN

2.3. Armazenamento de Dados

No armazenamento dos dados de terreno podemos utilizar formatos que codificam as elevações em imagens, e que são tipicamente designados por mapas de elevação (*height maps*), e formatos específicos, ASCII ou binários, com características geoespaciais específicas que permitem obter um conjunto de informações extra de carácter geográfico.

2.3.1. Formatos Baseados em Imagens

Um *height map* é a forma mais simples e comum de armazenar um *height field*, não sendo mais do que uma imagem que armazena as elevações de uma determinada porção de um terreno codificada nas cores de cada um dos *pixels* que a compõem. Mais precisamente é um *array* de duas dimensões que representa uma grelha regular (ver **2.2.1**). Um exemplo muito comum de *height map* é o mapa a preto e branco, em que a intensidade do *pixel* representa a elevação do terreno na coordenada correspondente. Um *pixel* branco representa neste caso o ponto mais alto do *height map*, pelo contrário um *pixel* preto representa o caso oposto. Este mapa pode ser armazenado nos mais variados tipos de ficheiros de imagem, por exemplo, BMP, JPG ou TIFF. Nestes casos retira-se de um determinado elemento da componente RGB de cada *pixel* a informação de elevação necessária (normalmente a partir da componente R). Grande parte dos algoritmos de geração de terrenos em tempo real utiliza este formato de dados, existindo *inclusive* várias ferramentas que permitem a conversão para esta representação. Tal acontece porque permite uma correspondência quase directa entre o armazenamento e a representação em memória.

Formalmente, um *height map* pode ser descrito por uma função F tal que $F(x,z) = [x, y(x,z), z]$, ou seja, o valor de y pode ser obtido a partir dos valores de x e z que funcionam como índices. Por exemplo, na **Figura 2-8**, está representada uma grelha regular e o *height map* correspondente. Neste exemplo, para obter o valor da coordenada y em que $(x,z) = (4,3)$ basta considerar x e z como índices na grelha (utilizando a função F). Por outro lado, para obter o valor das coordenadas (x,z) propriamente ditas basta multiplicá-las pelo espaçamento considerado que tem que ser fornecido como parâmetro e permite estabelecer a distância entre os diferentes valores de elevação.

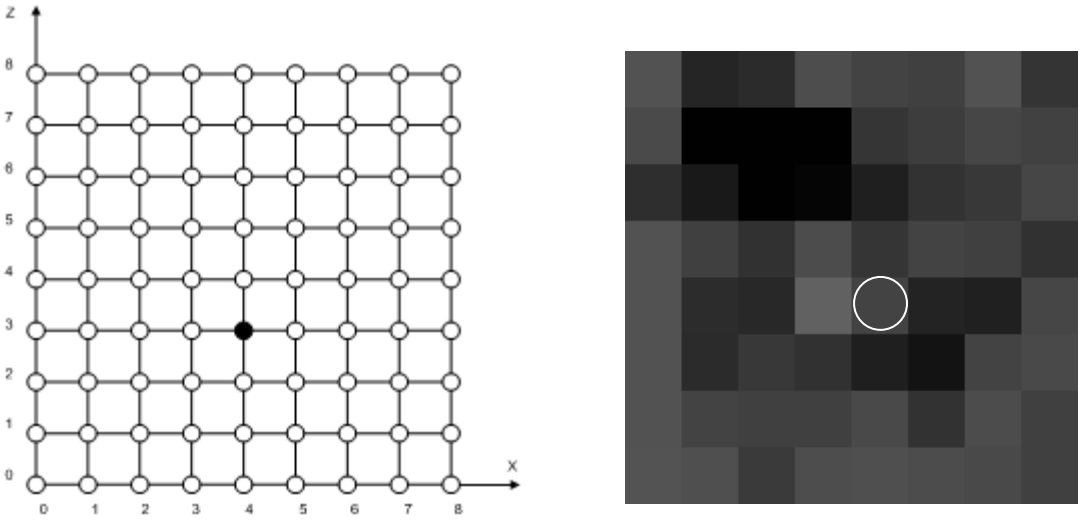


Figura 2-8: Um *height map* de 9 × 9.

2.3.2. Formatos Específicos

Os dados de terreno são normalmente armazenados em *height maps* tal como foi referido em **2.3.1**. Estes resultam muitas vezes de conversões feitas a partir de formatos geoespaciais específicos utilizados por instituições como a USGS (*US Geological Survey*) [202] e a NIMA (*National Geospatial-Intelligence Agency*) [141] ou outras organizações que recolhem dados de elevação. Estes dados resultam sobretudo de trabalho de campo, cartas, fotografia aérea, detecção remota, GPS e de sistemas de informação geográfica [5] e requerem normalmente um conjunto de capacidades acima das que são disponibilizadas por um simples *height map*, nomeadamente uma precisão acima dos 8 bits disponíveis numa imagem e a capacidade de armazenar metadados (por exemplo, o sistema de coordenadas utilizado). São apresentados de seguida alguns dos formatos geoespaciais mais comuns, bem como uma pequena descrição de cada um deles:

- Formatos *raster* [119].
    - USGS DEM [142]: Este formato de dados é proveniente do Instituto Geológico dos Estados Unidos e é uma simples grelha regular com três níveis de precisão disponíveis (respectivamente 90, 30 e 10 metros).
    - DTED (*Digital Terrain Elevation Data*) [156]: Semelhante ao formato USGS foi concebido pela agência americana de imagem e mapas (a NIMA [141]) para o exército dos Estados Unidos e têm seis níveis de precisão com as seguintes resoluções:
        - Nível 0: 1 km.
        - Nível 1: 100 metros.
        - Nível 2: 30 metros.
        - Nível 3: 10 metros.
        - Nível 4: 3 metros.
        - Nível 5: 1 metro.
    - BT (*Binary Terrain*) [47]: Criado no âmbito do projecto vterrain.org [47], teve como objectivo responder às limitações do formato USGS DEM,

nomeadamente o facto de este consumir muito espaço e de ser bastante inflexível na representação dos dados.
    - GTOP030 [203]: Um formato mais antigo e com menos precisão (aproximadamente de 1 km).
- Formatos vectoriais [221].
    - AutoCAD DXF [12]. É um formato vectorial criado pela Autodesk e suportado por um grande número de aplicações CAD. É também um *standard* na transferência de informação entre sistemas CAD e GIS.
    - ESRI Shape File [59]. O formato nativo utilizado por produtos GIS como o ArcINFO. Existe uma grande quantidade de dados neste formato que é suportado pela maioria das aplicações GIS.

2.4. Fontes de Elevação

A fonte de elevação utilizada na construção do terreno pode ser obtida de duas formas distintas: a primeira corresponde a utilização de dados reais obtidos a partir de medições efectuadas por satélites, trabalho de campo e cartas topográficas por exemplo. A segunda corresponde à criação desses dados de uma forma manual ou algorítmica/procedimental Para isso, recorre-se muitas vezes a ferramentas que podem ir desde programas de desenho como o Adobe Photoshop [6], até a pacotes específicos de criação de terrenos, como o Wilbur [184], Grome [82], World Machine [219], Terragen [164], L3DT [103] e Leveller [196]. Muitos destes permitem a importação de dados reais e a sua alteração de acordo com as necessidades específicas da aplicação em que vão ser inseridos. A alternativa é a geração dos dados de elevação em tempo real por intermédio de um algoritmo. A principal vantagem nesse caso é não ser necessário armazenar directamente o *height field* pois o terreno pode ser totalmente representado pelo conjunto de parâmetros que lhe é fornecido. Há que considerar, no entanto o impacto do tempo neste processo, uma vez que sempre que se correr o programa tem de se voltar a criar o terreno. Existem diversos algoritmos para este efeito por exemplo os descritos em [180], [61], [67], [182] e [139]. A sua discussão está fora do âmbito desta dissertação, no entanto devido a sua importância disponibiliza-se no **Anexo A** um resumo dos mais importantes com ênfase nos fractais (ver **A.1**) enquanto estratégia de referência.

A geração de conteúdos por intermédio de algoritmos está a ganhar cada vez mais importância, algo que é evidenciado em jogos como o Spore [126] que representam bem a sua potencialidade não só na construção terrenos mas também, como é o caso específico deste jogo, de mundos. O MojoWorld [152], uma aplicação de criação de planetas procedimentais e o ProFx [168] um pacote de criação de texturas, são outros exemplos, bem como o KKrieger [1], um jogo em apenas 96 Kbytes mas com gráficos equivalentes aos jogos mais exigentes, tudo isto graças a um sistema de sintetização de todos os elementos que o constituem.

2.5. Sumário

Este capítulo começa por abordar em detalhe as duas estruturas de dados mais comuns na representação de terrenos: as grelhas regulares (ver **2.2.1**) e as *triangulated irregular networks* (ver **2.2.2**). Foram realçadas as suas vantagens/desvantagens (ver **2.2.3**) e

concluiu-se que muito embora as *triangulated irregular networks* sejam mais flexíveis tornam mais complexa a implementação de técnicas essenciais para a representação de terrenos em tempo real, tais como o *culling*, a gestão do nível de detalhe e a detecção de colisões e necessitam além disso de mais espaço de armazenamento. Em contrapartida as grelhas regulares permitem a utilização de todas essas técnicas, necessitam de menos espaço e são mais fáceis de manipular pelo que são a forma mais comum de representar terrenos. Os seus problemas mais relevantes são não permitirem a sobreposição de valores de elevação, não podendo consequentemente representar formações como saliências e cavernas, e o facto de utilizarem sempre o mesmo número de polígonos para todas as regiões do terreno mesmo quando estas necessitam de menos detalhe. No entanto, apesar das limitações referidas, as grelhas regulares são, na maior parte dos casos, a estrutura mais apropriada para representar terrenos.

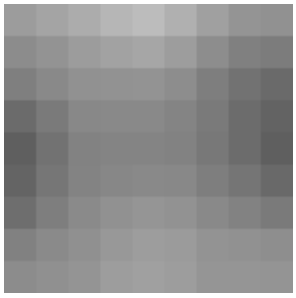
De seguida (ver **2.3**) dá-se especial ênfase ao armazenamento dos dados de elevação do terreno, nomeadamente às duas formas mais comuns de armazenar essa informação: através de formatos que codificam as elevações em imagens (ver **2.3.1**), e que são tipicamente designados por mapas de elevação (*height maps*), e através de formatos específicos (ver **2.3.2**). No caso dos *height maps* os valores de elevação são codificados nas cores de cada um dos *pixels* que compõem a imagem. Isto é, o valor de elevação é armazenado num determinado elemento da componente RGB de cada *pixel*. No caso dos formatos específicos estes armazenam para além dos valores de elevação também as características geográficas do terreno que representam agrupando-se em duas categorias: formatos *raster* (por exemplo USGS DEM, DTED) e formatos vectoriais (por exemplo DXF, ESRI Shape File).

Finalmente discutem-se em **2.4** as duas aproximações mais comuns para obter os dados de elevação**:** dados reais e métodos manuais e/ou algorítmicos/procedimentais. No primeiro caso pressupõe-se a obtenção dos valores de elevação através de um conjunto de técnicas que permitem efectuar medições numa determinada região do terreno obtendo um conjunto de dados que são depois processados e armazenados num determinado formato. Em alternativa é possível utilizar técnicas artificiais na criação do terreno como métodos manuais e/ou algorítmicos/procedimentais. No caso dos métodos manuais, pressupõe-se a utilização de um editor para manualmente se criar as elevações. No caso dos métodos algorítmicos/procedimentais, implica a utilização de algoritmos matemáticos que mediante a configuração de um conjunto de parâmetros permitem gerar os valores de elevação do terreno

3. Geração do Terreno

Para desenhar um terreno é necessário definir a sua representação tridimensional. Para isso são normalmente utilizadas malhas triangulares, pois estas têm diversas vantagens em comparação com outras formas de representação. Nomeadamente, é de realçar a sua simplicidade matemática do ponto de vista teórico, e o suporte em diversas API e placas gráficas, o que tem contribuído para a larga utilização e consolidação desta forma de representação. Os triângulos que as compõem são os componentes básicos na construção não só de terrenos, mas de qualquer objecto que se pretenda representar, sendo nesta perspectiva possível aproximar qualquer um por intermédio de uma malha triangular. Neste sentido há que considerar então novamente as estruturas de dados para representação de um terreno referidas em **2**, as grelhas regulares de elevações e as *triangulated irregular networks*. As grelhas regulares, como foi referido em **2.2.3**, são consideravelmente mais simples de utilizar e são, por conseguinte, a estrutura mais comum nos algoritmos de geração de terrenos em tempo real mais recentes. Por isso, dá-se especial ênfase a esta representação neste capítulo, nomeadamente às técnicas de nível de detalhe (ver **3.4**), de *culling* (ver **3.3**) e de partição de espaço (ver **3.2**) que lhe servem de suporte.

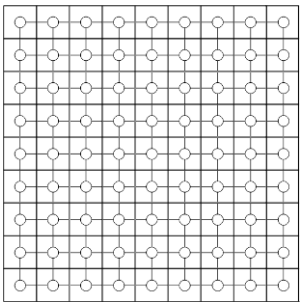
Partindo então de uma grelha regular, o próximo passo é construir a malha triangular correspondente, recorrendo para isso às diferentes primitivas que as API gráficas nos disponibilizam (ver **3.1**). Utilizando uma aproximação "ingénua" ao problema, ou tal como é referida na literatura, de força bruta, vamos partir, para efeitos de exemplo, de um *height map* do qual obtemos o *height field* correspondente. De seguida, utilizando a API gráfica seleccionada (por exemplo, o XNA [220], o DirectX [45] ou OpenGL [150]) constrói-se a malha triangular e enviam-se todos os triângulos que a compõem para *rendering*. Este processo está representado na **Figura 3-1** onde, o resultado final é uma malha triangular que representa os valores de elevação de um *height field* obtido a partir de um *height map*.



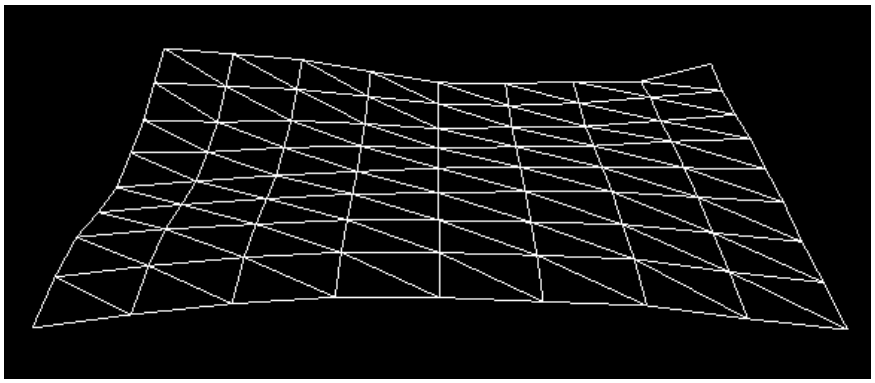
Height map

118	108	99	79	58	55	61	75	94
107	100	97	83	70	68	72	82	95
94	92	93	88	81	82	87	97	106
93	96	103	105	104	106	109	116	121
96	107	117	125	128	132	133	132	130
119	133	143	151	153	156	154	146	140
147	159	167	175	176	174	171	157	144
158	175	187	196	196	192	188	169	155
161	181	197	206	206	200	197	177	162

Height field



Construção da malha a partir do *height field*



Malha Triangular

Figura 3-1: Uma malha triangular de um terreno a partir de um *height map*.

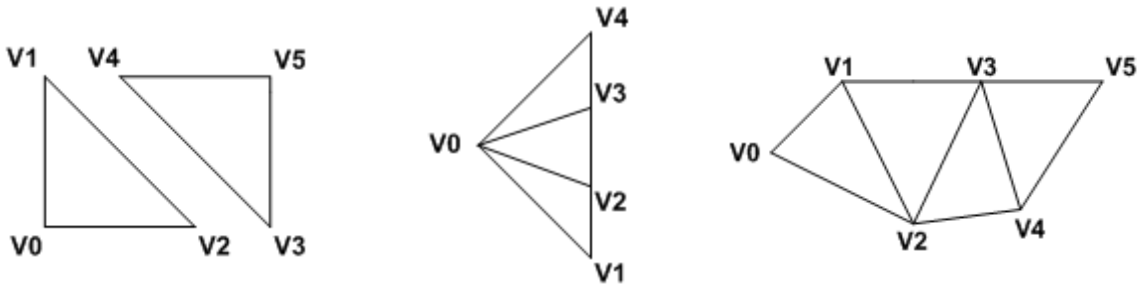
Na construção da malha, para cada um dos vértices, as coordenadas x e z são atribuídas tendo em conta o número de valores de elevação do *height field* de origem e o espaçamento pretendido entre vértices. Para representar a elevação, ou seja a coordenada y, é obtido o valor correspondente no *height field* nas coordenadas x e z. Ou seja, basicamente o número de valores de elevação no *height field* contribui para a dimensão do terreno (o x e z) e o valor na coordenada respectiva para a elevação. Uma vez definidas as coordenadas de cada um dos vértices na malha, há que construir então os triângulos. Estes são obtidos a partir de sequências de três vértices numa lista ou a partir de sequências de índices relativos à posição dos vértices na lista de vértices. O envio para *rendering* de todos os triângulos que a compõem a malha triangular é uma estratégia que pode ser utilizada em *height maps* de dimensões reduzidas, como o da **Figura 3-1** (9×9). No entanto, na representação de terrenos de maiores dimensões, muitas vezes com requisitos de memória acima das capacidades do computador, esta aproximação é na verdade impraticável, pois levanta um conjunto de problemas que impedem a sua concretização. Um dos principais é a grande quantidade de triângulos envolvidos que tornam o processamento demasiado pesado impedindo consequentemente a sua concretização em tempo real. Desta forma, a única hipótese é utilizar um conjunto de técnicas que permitem a redução do número de triângulos enviados. Uma delas é o *culling*, que permite remover os triângulos que não são visíveis, a outra é a diminuição do

nível de detalhe em determinadas áreas do terreno. Nesta última tira-se partido do efeito de perspectiva para diminuir o detalhe (entenda-se neste contexto triângulos) de blocos de terreno, que pela distância a que se encontram da câmara não necessitam de ser representados com a mesma precisão e cujo processamento tem apenas como efeito aumentar a carga na placa gráfica. Para concretizar qualquer uma destas técnicas temos, no entanto, de utilizar estruturas de dados que permitam efectuar a partição do espaço, isto para que seja possível avaliar separadamente tanto para efeitos de *culling* como de redução de detalhe blocos separados do terreno. Neste contexto, as técnicas de redução de detalhe são as que têm mais impacto e as que geram paralelamente um conjunto de outros problemas cuja resolução tem sido um dos objectivos dos algoritmos publicados nesta área (ver **4**). Um factor importante em relação a estes algoritmos está relacionado com a dimensão das fontes de elevação utilizadas que são praticamente em todos os casos da forma $2^n + 1$, por exemplo, 2×2, 3×3, 5×5, 9×9, 17×17 e 33×33, em grande parte porque esta dimensão permite uma subdivisão contínua em 2^n blocos, facto que é muito útil na partição espacial do terreno. Apresentam-se de seguida as diferentes formas de construir a malha triangular em função do tipo de primitiva gráfica utilizada (ver **3.1**), as estruturas de dados espaciais utilizadas na organização dos objectos em espaços 2D e 3D (ver **3.2**), as técnicas de *culling* para remoção de triângulos não visíveis (ver **3.3**) e as técnicas para variação do nível de detalhe (ver **3.4**).

3.1. Construção da Malha Triangular

Para construir os triângulos que constituem os objectos é necessário especificar a lista dos vértices que os compõem. Por sua vez, a forma como os triângulos são construídos depende do tipo de primitiva utilizada, que estabelece como os vértices na lista são interpretados. Assim, podemos considerar neste contexto três tipos de primitivas [135]:

- Lista de triângulos, na **Figura 3-2 a)**, em que cada grupo de três vértices formam um triângulo. Por exemplo, os vértices $(V0, V1, V2)$ formam um e $(V4, V5, V3)$ formam o próximo.
- Leques (*fans*) de triângulos, na **Figura 3-2 b)**, em que o primeiro vértice forma um triângulo com cada par sucessivo de vértices. Por exemplo, $(V0, V1, V2)$, $(V0, V2, V3)$ e $(V0, V3, V4)$ em que $V0$ é o vértice comum a todos eles.
- Tiras (*strips*) de triângulos, na **Figura 3-2 c)**, em que cada grupo de três vértices contíguos forma um triângulo. Por exemplo, $(V0, V1, V2)$, $(V1, V2, V3)$, $(V2, V3, V4)$ e $(V3, V4, V5)$. Note-se que os dois últimos vértices de cada triângulo são partilhados pelo triângulo seguinte.



a) Lista de triângulos c) Leques de Triângulos b) Tiras de Triângulos

Figura 3-2: Tipos de primitivas.

Qualquer uma destas primitivas pode ser utilizada na triangulação de um *height field*, muito embora existam diferenças bastantes significativas entre cada uma delas. Comum a todas é, no entanto, a possibilidade de associar uma lista de índices à lista de vértices. Estas listas não são mais do que sequências de três índices correspondentes às posições ocupadas pelos vértices na lista de vértices, permitindo que cada um deles seja armazenado apenas uma vez, como é exemplificado na **Figura 3-3** em que é utilizada uma lista de quatro vértices para construir um quadrado.

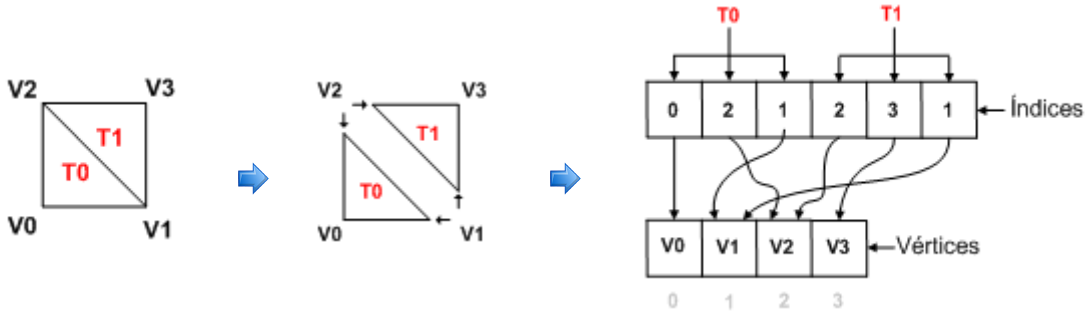


Figura 3-3: Listas de índices na triangulação de um quadrado.

A utilização de índices permite ainda tirar proveito da *cache* de vértices que existe nas placas gráficas. A ideia desta *cache* é garantir que a placa não processa o mesmo vértice mais do que uma vez no *rendering* de uma primitiva, algo que está muito dependente da ordem dos vértices fornecidos e da flexibilidade da primitiva considerada, existindo diversas técnicas que procuram tirar o máximo partido desta optimização (por exemplo, as descritas em [65], [20] e [91]). Normalmente, esta *cache* não suporta mais do 12 a 24 vértices [65] e em determinadas situações pode aumentar significativamente o desempenho, sobretudo quando a transformação dos vértices ou a utilização da memória são factores limitativos na aplicação em causa. Existem, no entanto, dois factores que podem tornar esta optimização difícil de aplicar. Em primeiro lugar não existe ainda uma forma estandardizada de obter o número de vértices que compõem a *cache* da placa gráfica, sendo, por vezes difícil, detectar *inclusive* se a placa tem essa *cache* ou não. Por outro lado, a menos que a organização dos vértices possa ser facilmente modificada, a optimização a ser aplicada corre o risco de ser demasiadamente especifica ou dito de outra forma demasiadamente orientada a uma determinada placa gráfica o que não é

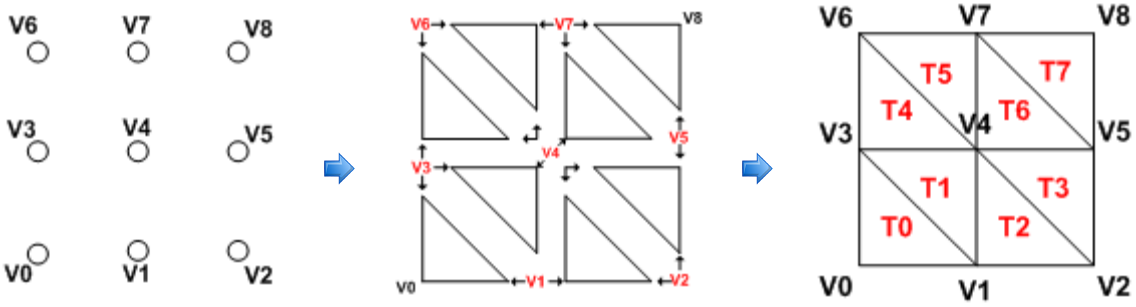
minimamente desejável na maior parte dos casos. Por estas razões as aproximações mais recentes procuram construir a melhor sequência de triângulos que possa tirar proveito da *cache* independentemente do seu tamanho [65] [20].

Ainda no que diz respeito ao desempenho, podemos definir três pontos fundamentais a considerar na escolha da primitiva que devem ser analisados mediante o tipo de objecto a representar. Assim do mais importante para o menos importante [66]:

- O número de vezes que a primitiva tem de ser "chamada" para efectuar o *rendering* de um objecto;
- O número de vértices processados;
- A quantidade total de dados enviados para *rendering*.

3.1.1. Listas de Triângulos

Considere-se, a **Figura 3-4** na qual se representa a construção de uma malha triangular de um *height field* com uma lista de triângulos. Neste caso, se não fossem utilizadas listas de índices, era necessário especificar individualmente todos os vértices que constituem cada um triângulos, ou seja, tinha de se definir uma lista com 24 vértices em que $(V0, V3, V1)$ formam um triângulo, $(V3, V4, V1)$ outro e assim sucessivamente. Como se pode verificar, existe uma partilha de vértices em triângulos adjacentes e consequentemente uma repetição dos mesmos na lista de vértices, algo que é bastante ineficiente e implica um maior consumo de memória, dado que se armazena os mesmos dados de um vértice duas vezes. Tem ainda como consequência um maior processamento por parte do *hardware* gráfico, porque implica processar os mesmos vértices mais do que uma vez. Nesta primitiva são assim de especial importância as listas de índices, já que, permitem a construção de listas de vértices distintas, sendo os índices utilizados na definição de cada um dos triângulos. Deste modo, no exemplo em consideração, poderíamos construir uma lista com apenas 8 vértices utilizando a respectiva lista de índices em que $(0,3,1)$ define o primeiro triângulo e $(3,4,1)$ o segundo, e assim sucessivamente. Esta primitiva é desta forma a mais flexível das mencionadas, pois permite a definição triângulo a triângulo, algo que não acontece nos leques de triângulos (ver **3.1.2**) e nas tiras de triângulos (ver **3.1.3**). Em contrapartida são necessárias listas de índices (no caso de ser indexada) maiores em comparação com outras primitivas. Por outro lado, com esta primitiva é possível concretizar o *rendering* de um *height field* com uma única chamada, pois neste caso o único constrangimento é, no limite, a memória já que os triângulos são independentes entre si. Ou seja, recorrendo a esta primitiva no limite, todos os triângulos que compõem um *height field* podem ser enviados de uma só vez para o API, independentemente do tamanho.

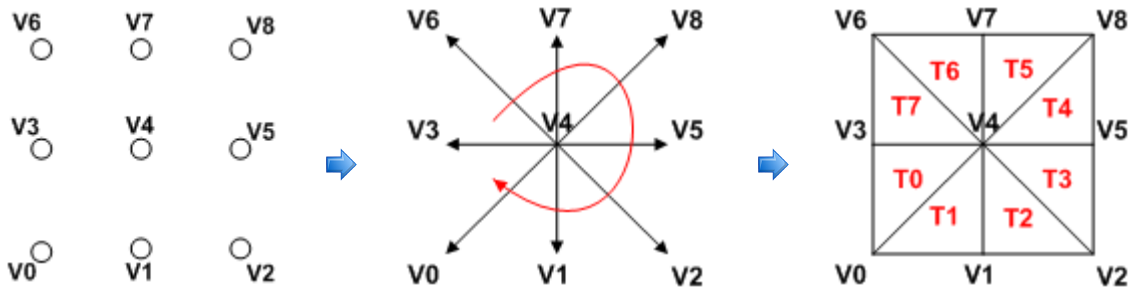


Não Indexada	Vértices	$(V0, V3, V1, V3, V4, V1, V1, V4, V2, V4, V5, V2, V3, V6, V4, V6, V7, V4, V4, V7, V5, V7, V8, V5)$
Indexada	Vértices	$(V0, V1, V2, V3, V4, V5, V6, V7, V8)$
	Índices	(0,3,1,3,4,1,1,4,2,4,5,2,3,6,4,6,7,4,4,7,5,7,8,5)

Figura 3-4: Triangulação de um *height field* com uma lista de triângulos.

3.1.2. Leques de Triângulos

Os leques de triângulos, tal como se pode verificar na **Figura 3-5**, são construídos a partir de um vértice central que é partilhado por todos os triângulos que o compõem. Esse vértice, no caso da figura, o $V4$, tem de ser o primeiro da lista e constitui com os dois vértices seguintes ($V3$ e $V6$) o primeiro triângulo. Todos os triângulos seguintes partilham com o anterior um dos seus vértices e o vértice central. É assim mais eficiente a nível de espaço, pois por cada vértice adicionado após os três primeiros, obtém-se um novo triângulo, o que significa na prática que os índices não têm um impacto tão grande ao nível do número de vértices submetidos para *rendering* como nas listas de triângulos. Paralelamente, o facto de existir um vértice central que serve de base a todos os triângulos do leque torna fácil ignorar um determinado vértice na sua construção. Isto é particularmente útil na correcção de falhas na malha triangular causadas por diferentes níveis de detalhe adjacentes (ver **3.4.4.1**). Por exemplo, o vértice $V5$ na figura podia ser ignorado dando origem a um leque composto pelos vértices $V4$, $V8$ e $V2$, o que tornaria possível um leque adjacente de menos detalhe, sem causar as descontinuidades no terreno.



Não Indexada	Vértices	$(V4, V3, V6, V7, V8, V5, V2, V1, V0, V3)$
Indexada	Vértices	$(V0, V1, V2, V3, V4, V5, V6, V7, V8)$
	Índices	(4,3,6,7,8,5,2,1,0,3)

Figura 3-5: Triangulação de um *height field* com um leque de triângulos.

É de realçar, no entanto, que a forma como é construída, ou seja, a partir de um único vértice partilhado por todos os triângulos, condiciona a área abrangida por um único leque. Assim, e ao contrário das outras primitivas, implica várias chamadas para representar um bloco de dimensão superior a 3×3, sendo necessário utilizar mais do que um leque. Tal tem um grande impacto no desempenho nos processadores gráficos actuais optimizados para grandes sequências de triângulos. Por exemplo, para representar o *height field* 9×9 da **Figura 3-6** com o máximo detalhe são necessários 16 leques, isto é, 16 chamadas com 8 triângulos cada.

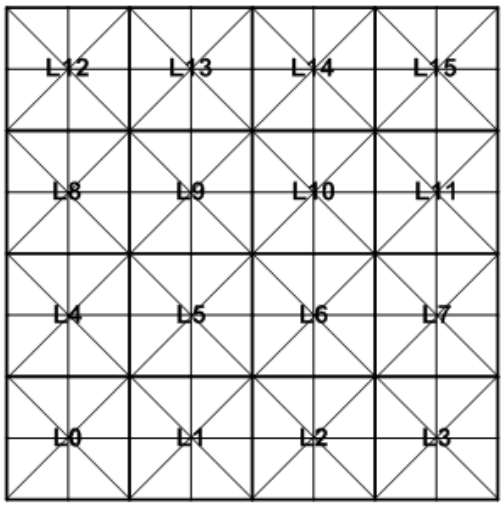


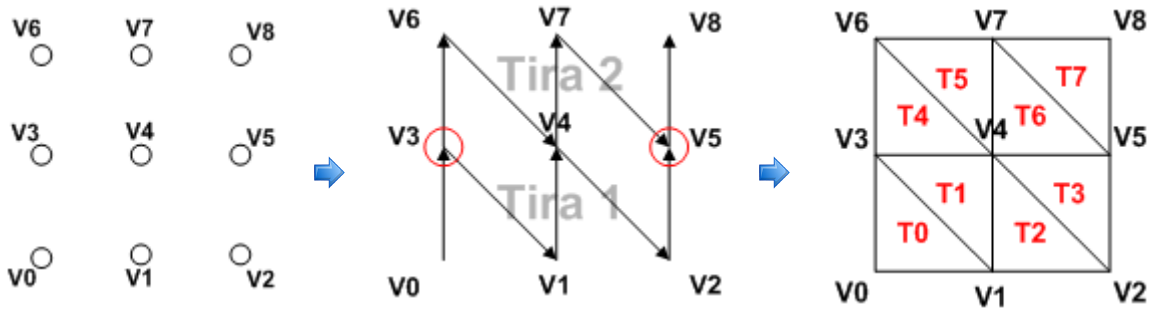
Figura 3-6: Triangulação de um *height field* 9×9 com um leque de triângulos.

A única solução possível para este problema é uma capacidade ainda não muito suportada pelas API: o *primitive restart*. Inicialmente disponibilizada via uma extensão de OpenGL [146] que permite, mediante a especificação de um determinado valor na lista de índices, recomeçar a primitiva noutra posição e a única solução que permite utilizar

uma única chamada. Alguns algoritmos já tiram partido desta capacidade nomeadamente os descritos em [177] e em [199].

3.1.3. Tiras de Triângulos

Numa tira de triângulos, cada grupo de três vértices contíguos forma um triângulo. É assim, tal como os leques, bastante eficiente a nível de espaço, com a vantagem adicional de não existir um vértice central partilhado. Tome-se como exemplo a triangulação do *height field* na **Figura 3-7**. De maneira a criar uma tira, ou seja, uma sequência de triângulos interligados entre si, os vértices são dispostos na lista de vértices ou referenciados na lista de índices segundo uma determinada ordem que permite formar cada uma das duas tiras que compõem o *height field* representado. Tal como está evidenciado na figura, torna-se necessário utilizar duas tiras, o que significa na prática duas chamadas à primitiva, um dos problemas dos leques. Uma solução seria utilizar, tal como foi descrito para os leques (ver **3.1.2**) *primitive restarts* [146], uma capacidade que não é ainda muito suportada. No entanto, ao contrário dos leques existe uma outra solução que passa pela criação de triângulos invisíveis para ligar cada uma das tiras. Para construir estes triângulos podemos considerar duas alternativas. A primeira corresponde à utilização de triângulos com a face virada para trás, ou seja, com as normais invertidas. Mesmo tendo em consideração que o *rendering* destes triângulos é efectuado, como têm as normais invertidas apenas podem ser detectados quando observados do lado oposto isto, assumindo que o *back-face culling* (ver **3.3.1**) está activo. Este método pode no entanto dar origem a um aparecimento súbito de triângulos se o terreno rodar ou mudar de posição não se tratando portanto de uma boa solução. A segunda envolve a utilização de triângulos degenerados para ligar as tiras de triângulos, sendo a forma mais comum de resolver este problema. Estes triângulos não têm área matemática e não são por isso considerados no processo de *rendering*, sendo normalmente construídos pela repetição do vértice final na primeira tira $(\dots, 5, 5, \dots)$ e do vértice inicial $(\dots, 3, 3, \dots)$ na segunda, tal com está representado na figura.



Não Indexada	Vértices	$(V0, V3, V1, V4, V2, V5, V5, V3, V3, V6, V4, V7, V5, V8)$
Indexada	Vértices	$(V0, V1, V2, V3, V4, V5, V6, V7, V8)$
	Índices	(0,3,1,4,2,5,5,3,3,6,4,7,5,8)

Figura 3-7: Triangulação de um *height field* com uma tira de triângulos.

3.1.4. Listas vs Leques vs Tiras

Na **Tabela 3-1** é feita uma comparação entre as diferentes primitivas descritas. No caso dos terrenos, se levarmos em consideração o *hardware* actual, a escolha recai tipicamente nas listas e nas tiras de triângulos indexadas. Os leques sem *primitive restarts* não são uma opção válida ao nível do desempenho, algo que está directamente relacionado com o número de chamadas necessárias para efectuar o *rendering* de um *height field* $n \times m$ o que tem, como já foi referido, um impacto adverso no desempenho dos processadores gráficos mais recentes, optimizados para grandes conjuntos de triângulos.

	Nº Índices	**Nº Chamadas**	**Vantagens**	**Desvantagens**
Lista	$6 \cdot (n-1) \cdot (m-1)$	1	• Flexibilidade • Uma única chamada	• Espaço ocupado
Leque	$\frac{5}{2} \cdot (n-1) \cdot (m-1)$	$\frac{1}{4} \cdot (n-1) \cdot (m-1)$	• Correcção de falhas • Espaço ocupado	• Vértice central • Mais do que uma chamada.
Tiras	$n \cdot m + (n+2) \cdot (m-2)$	1	• Espaço ocupado • Uma única chamada (com triângulos degenerados)	• Implica a utilização de triângulos degenerados para ligar as tiras. • Mais complexa.

Tabela 3-1: Comparação entre primitivas (adaptado de [206]).

3.2. Estruturas de Dados Espaciais

As estruturas de dados espaciais permitem a organização dos objectos em espaços n-dimensionais, sobretudo de duas e três dimensões, muito embora os conceitos que as caracterizam possam ser facilmente adaptados para dimensões mais elevadas [135]. Na maior parte das vezes são hierárquicas, tendo por princípio a divisão recursiva do espaço, o que significa, que o nível mais elevado abrange todo o espaço ocupado pelo nível inferior, que por sua vez abrange o espaço ocupado pelo nível imediatamente abaixo, e assim sucessivamente. Nesta perspectiva, a grande vantagem da utilização de uma abordagem hierárquica está no facto de permitir pesquisas significativamente mais rápidas, reduzindo a complexidade tipicamente de $O(n)$ para $O(\log n)$[135], em grande parte, pelo facto de a rejeição de um nó na hierarquia implicar o descartar de todos os que dele descendem. Estas estruturas servem por outro lado de suporte a um conjunto de operações, nomeadamente de *culling* (ver **3.3**), triangulação, testes de intersecções e de detecção de colisões e são portanto essenciais, na representação de cenas tridimensionais. Podem ser classificadas em *bounding volume hierarchies* (ver **3.2.2**) e *spatial partitioning hierarchies* (ver **3.2.3**). As primeiras não recorrem a subdivisão do espaço na sua totalidade mas sim ao agrupamento de um ou mais objectos dentro de um mesmo volume, o *bounding volume* (ver **3.2.1**). As segundas dividem o espaço na sua totalidade

num conjunto de regiões regulares ou não regulares categorizando cada uma delas de acordo com o seu conteúdo. Consequentemente, uma *bounding volume hierarchy* classifica regiões do espaço à volta de objectos e as *spatial partitioning hierarchies* classificam objectos à volta de regiões do espaço [163].

3.2.1. Bounding Volumes

O termo *bounding volume* é muito genérico e está associado a qualquer objecto que contenha um conjunto de objectos. Não contribuem para a imagem final e têm como principal objectivo aumentar a eficiência de operações geométricas pela utilização de volumes simples em substituição dos objectos complexos que contêm. Permitem assim tornar as operações geométricas mais fáceis de implementar e também computacionalmente menos pesadas. Os *bounding volumes* podem assumir as mais variadas formas, algumas delas representadas na **Figura 3-8**, nomeadamente:

- *Bounding Spheres* (BS) que como o nome indica, são esferas que envolvem o objecto.
- *Axis Aligned Bounding Boxes* (AABB), volume com faces rectangulares alinhadas com os eixos do sistema de coordenadas.
- *Oriented Bounding Boxes* (OBB), semelhantes às *axis aligned bounding boxes* mas arbitrariamente "rodadas" no sentido de melhor se adaptarem à estrutura do objecto.
- *K-dimensional discrete oriented polytopes* (K-DOP), construídos a partir de um conjunto *k* de planos que delimitam o objecto.
- *Convex Hulls* que representam o mais pequeno volume convexo que contêm o objecto.

É de referir que o cálculo destes volumes está muito dependente do grau de precisão pretendido, tendo este de ser avaliado mediante as condicionantes de desempenho existentes, tal como está representado na figura.

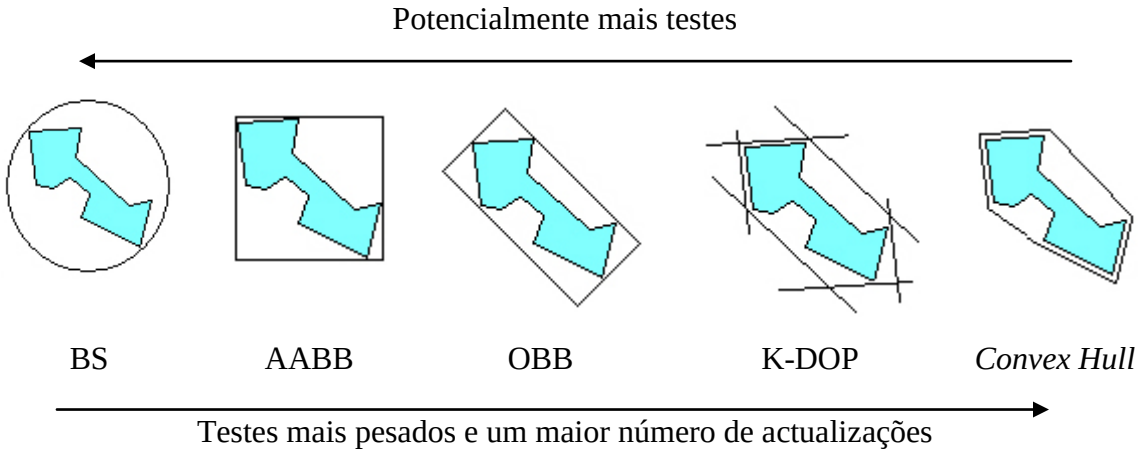


Figura 3-8: Tipos mais comuns de *bounding volumes* [163].

3.2.2. Bounding Volume Hierarchies

As *bounding volume hierarchies* permitem a organização hierárquica de um conjunto de objectos. Cada objecto é armazenado numa árvore composta por um nó raiz, o mais alto da árvore, e o único sem antecessor, por nós folha, que armazenam os objectos e não têm filhos, e por nós internos, que armazenam referências para outros nós (o nó raiz é também um nó interno). A cada um dos nós está associado um *bounding volume* (ver **3.2.1**) que engloba o espaço ocupado por todos os objectos na sua sub-árvore. Ou seja, o *bounding volume* do nó raiz contém toda a cena. Estes conceitos estão representados na **Figura 3-9**, na qual são utilizadas *axis aligned bounding boxes* para cada um dos objectos na estrutura hierárquica.

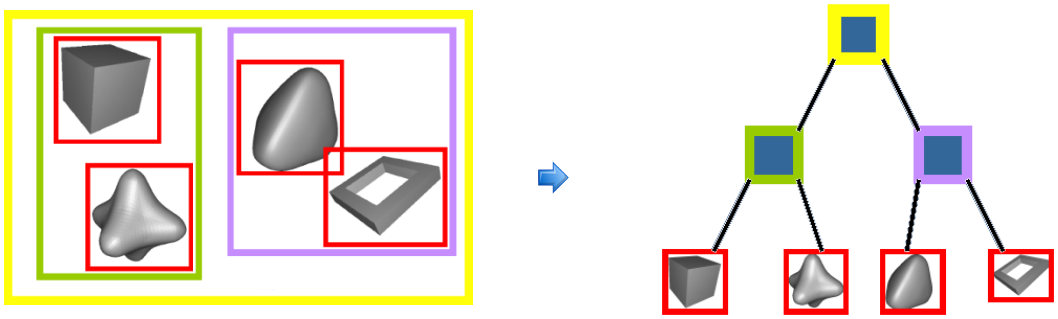


Figura 3-9: *Bounding Volume Hierarchy* [136].

Para criar uma *bounding volume hierarchy* é necessário executar um processo recursivo de subdivisão do espaço ocupado pelo conjunto de todos os objectos na cena. Para isso calcula-se uma série de *bounding volumes* cujo objectivo é agrupar a cada nível da árvore progressivamente menos objectos. Para as *axis aligned bounding boxes* o processo inicia-se com o cálculo da *bounding box* mais pequena que engloba todos os objectos na cena. O próximo passo passa por dividir a *bounding box* segundo o eixo de maior dimensão criando assim dois novos *bounding volumes* que englobam todos os objectos nas partições respectivas, isto recursivamente até se satisfazer uma determinada condição de paragem, por exemplo [136] :

- Quando um *bounding volume* está vazio
- Quando uma única primitiva está dentro do *bounding volume*.
- Quando um número de primitivas inferior a n esta dentro do *bounding volume.*
- Quando um nível de recursão l for atingido.

Este processo é ilustrado na **Figura 3-10** onde são aplicadas um conjunto de subdivisões até todos os objectos estarem englobados por uma única *axis aligned bounding box*.

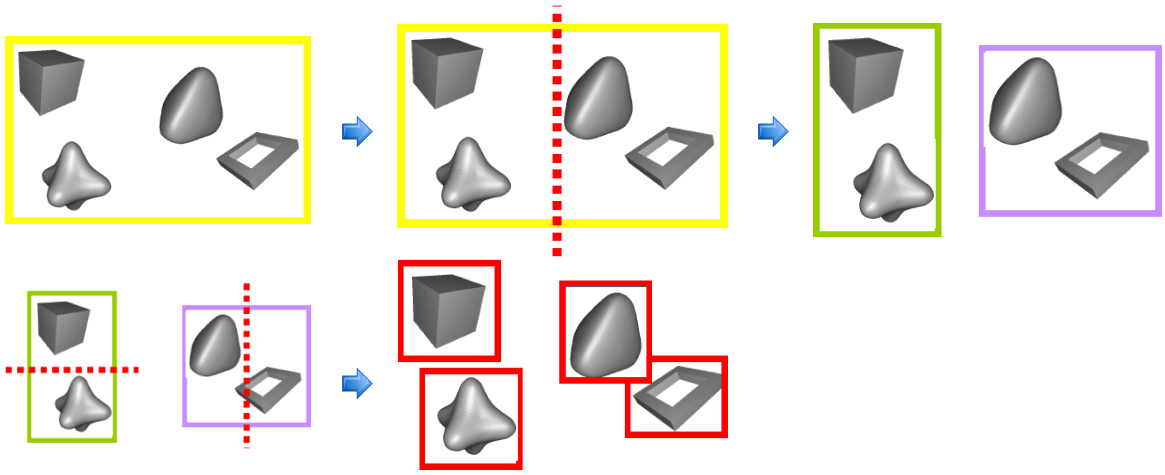


Figura 3-10: Criação de uma *bounding volume hierarchy* [136].

Esta estrutura é importante na medida em que define um conceito de alto nível na representação da cena permitindo o estabelecimento de relações espaciais entre objectos. Muito embora os terrenos sejam normalmente representados por *spatial partitioning hierarchies* (ver **3.2.3**) mais especificamente por *quadtrees* (ver **3.2.3.1**) as *bounding volume hierarchies* são um conceito mais geral permitindo o englobar de outras estruturas nos nós folha. Por exemplo um dos nós podia ser uma *quadtree* e o terreno que representa. Nesta perspectiva estão associadas a um conceito ainda mais geral: os *scene graphs* que não são mais do que *bounding volume hierarchies* enriquecidas com um conjunto de características adicionais [136].

3.2.3. Spatial Partitioning Hierarchies

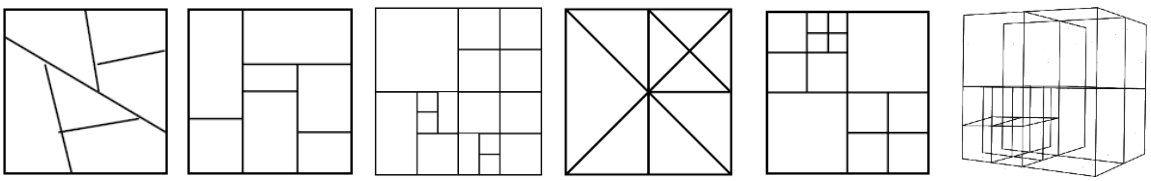
Se as *bounding volume hierarchies* procuram agrupar objectos de acordo com o seu relacionamento espacial, as *spatial partitioning hierarchies* dividem o espaço em diversas regiões, categorizando-o de acordo com o conteúdo de cada uma delas. Assim, cada região armazena ou tem uma referência para zero ou mais objectos e um objecto por sua vez pode ser referenciado de mais do que uma região [163]. De um modo geral, pode-se dizer que neste caso a união do espaço ocupado por todos os nós que constituem a hierarquia é igual ao espaço ocupado por toda a cena. Existem diversas estruturas utilizadas na partição do espaço, das quais se destacam as seguintes, tidas como as mais relevantes neste domínio (ver **Figura 3-11**):

- *Binary Space Partitioning Tree* (*BSP tree*) ([70] e [71]) que permite a subdivisão recursiva de um espaço k-dimensional em duas regiões **convexas** por intermédio de hiperplanos $k-1$ dimensionais **arbitrários**.
- *Kd-tree* ([13], [14] e [15]) é uma variante da *BSP Tree*, por vezes designada de *axis aligned BSP tree* que permite a subdivisão recursiva de um espaço k-dimensional em duas regiões por intermédio hiperplanos $k-1$ dimensionais **paralelos aos eixos principais**. Nesta estrutura a direcção do hiperplano, isto é, a dimensão na qual foi efectuada a divisão, alterna entre as k possibilidades de um nível da árvore para outro. Por exemplo no primeiro nível a divisão é efectuada

no eixo dos xx, no segundo no eixo dos yy, no terceiro no eixo dos zz, no quarto no eixo dos xx e assim sucessivamente.

- *Bintree* ([99][174] e [190]) é uma variante da *Kd-tree* que permite a subdivisão recursiva de um espaço k-dimensional em duas regiões **iguais** por intermédio de hiperplanos $k-1$ dimensionais paralelos aos eixos principais. Tal como as *Kd-trees* o eixo em que é efectuada a divisão, alterna entre as k possibilidades de um nível da árvore para outro.
- *Triangle Bintree* (ver **3.2.3.3**) é uma variante da *bintree* que permite a subdivisão recursiva de um espaço k-dimensional triangular em **dois triângulos** por intermédio de hiperplanos $k-1$ dimensionais **perpendiculares à hipotenusa** do triângulo. Desta forma cada um dos triângulos é dividido a partir do vértice situado no ângulo de 90° até ao ponto intermédio oposto.
- *Quadtree* (ver **3.2.3.1**) que divide o espaço em quatro quadrantes regulares.
- *Octree* (ver **3.2.3.2)**, uma variante da *quadtree* a três dimensões que divide o espaço em oito octantes regulares.

Tanto as *BSP trees*, como as *Kd-Trees*, e as *bintrees* subdividem o espaço em duas regiões por intermédio de um hiperplano. No caso das *quadtrees* e das *octrees* a subdivisão do espaço é feita em quatro e oito regiões respectivamente, por intermédio de dois hiperplanos no caso das *quadtrees* e três no caso das *octrees*. No domínio da geração de terrenos, as *quadtrees*, as *octrees* e as *triangle bintrees* são as estruturas mais utilizadas e algumas delas, as *quadtrees* e as *triangle bintrees*, mais especificamente, têm ainda em alguns algoritmos um papel a assumir na triangulação propriamente dita. Nessa perspectiva dá-se especial ênfase a essas estruturas pela qual se descreve de seguida cada uma delas em maior detalhe.



BSP Tree *Kd-Tree* *Bintree* *Triangle Bintree* *Quadtree* *Octree*

Figura 3-11: *Spatial Partitioning Hierarchies* (adaptado de [117], [135] e [72]).

3.2.3.1. Quadtrees

O termo *quadtree* é usado para descrever uma classe de estruturas hierárquicas que tem como característica comum o facto de se basearem no princípio da divisão recursiva do espaço. Podem ser classificadas de acordo com o tipo de dados que representam incluindo pontos, áreas curvas, superfícies e volumes. Neste sentido, podemos considerar dois tipos de *quadtree*: a *region quadtree* [92] e a *point quadtree* [64]. O tipo mais comum é a *region quadtree* que permite a organização espacial a duas dimensões. Para isso divide o espaço em quatro quadrantes ou regiões num processo recursivo, que

termina, por exemplo, quando um número máximo de níveis de profundidade na árvore for atingido. As *point quadtrees*, por outro lado, têm todas as características de uma *region quadtree* e variam apenas no centro da subdivisão que é neste caso sempre num ponto, sendo por isso especialmente úteis na representação de dados de pontos multidimensionais. Desta forma, as *region quadtrees* são as mais relevantes no domínio desta dissertação pelo que são as únicas consideradas daqui em diante. Assume-se que a partir deste ponto qualquer referência a uma *quadtree* deve ser entendida como uma *region quadtree*. Para uma descrição mais detalhadas de todos os tipos de *quadtree*, consultar [176].

A construção de uma *quadtree* implica uma série de passos. Numa primeira fase é encontrado o menor *bounding volume* (ver **3.2.1**) que engloba todos os objectos da cena. A partir daí, a inserção de um objecto consiste em encontrar o quadrante mais adaptado à sua posição e tamanho, para isso executam-se testes sobre cada um dos quadrantes da *quadtree* com base no *bounding volume* que representa o objecto. Utilizando a **Figura 3-12** como guia, podemos descrever este processo que se inicia no nó mais alto da árvore, a raiz, e pode ser concretizado em três passos:

1. Verificar a qual dos nós filhos pertence o objecto. Se não for possível efectuar uma subdivisão no nó corrente que possa conter o objecto, saltar para 3.
2. Se o objecto estiver totalmente contido dentro do nó filho, selecciona-lo como o corrente e saltar para 1.
3. Se o objecto é membro do nó corrente, adicioná-lo à lista dos objectos para desse nó e sair.

Na **Figura 3-12** este processo é repetido três vezes até se encontrar o quadrante mais adaptado. Sempre que é encontrado um nó filho que possa conter o objecto, o processo é repetido até que não seja possível "encaixar" o objecto em mais nenhum nó filho. Tal como se pode verificar na figura, na terceira iteração, as linhas a tracejado indicam que na próxima subdivisão o objecto não faz parte de nenhum dos quadrantes obtidos, passando por conseguinte a fazer parte do nó corrente, o mais pequeno que o pode conter.

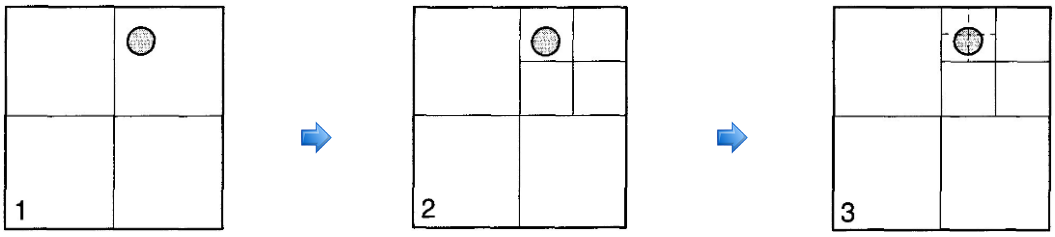


Figura 3-12: Atribuição de um objecto a um nó na *quadtree* [185].

Na **Figura 3-13** é possível verificar a distribuição de um conjunto de objectos por cada um dos nós de uma *quadtree*. É de realçar que alguns, pela sua posição ou tamanho têm de ser colocados em níveis mais elevados. Por exemplo a cruz está no segundo nível, devido à sua posição, pois intersecta a linha vertical que subdivide um dos quadrantes do segundo nível. No entanto se este objecto estivesse posicionado mais à direita ou mais à esquerda, poderia ser incluído num dos nós do terceiro nível.

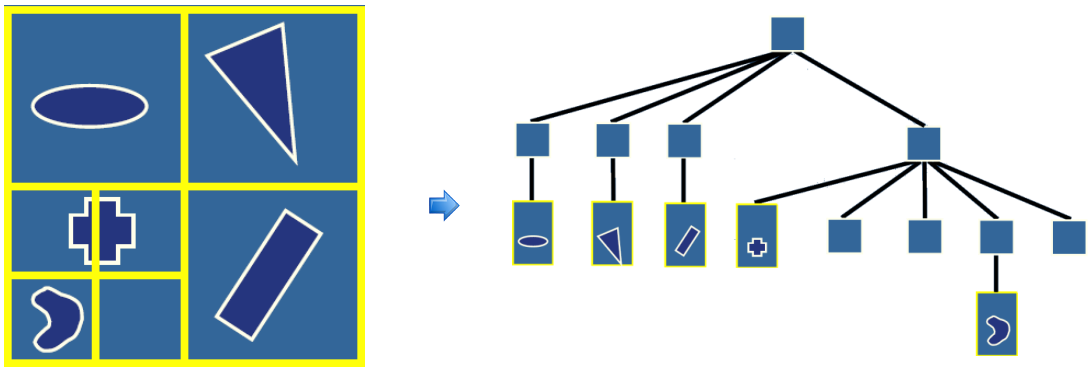


Figura 3-13: Organização de objectos numa *quadtree* [136].

As *quadtrees* são especialmente úteis no domínio da geração de terrenos em tempo real para particionamento do terreno em blocos permitindo a sua organização a nível espacial para efeitos de *culling* (ver **3.3**). No entanto, têm, tal como a *triangle bintree* descrita em **3.2.3.3**, uma outra característica: podem servir também de estrutura de suporte a toda a triangulação do terreno, permitindo *inclusive* o determinar do nível de detalhe mais adequado para cada um dos sectores que define, de acordo com a métrica de erro definida. Por exemplo, a *quadtree* construída na **Figura 3-14** suporta uma determinada triangulação do terreno que neste caso atribui mais detalhe à zona superior direita, ou seja, nessa zona vão ser considerados mais pontos de elevação no *height field* que lhe está subjacente.

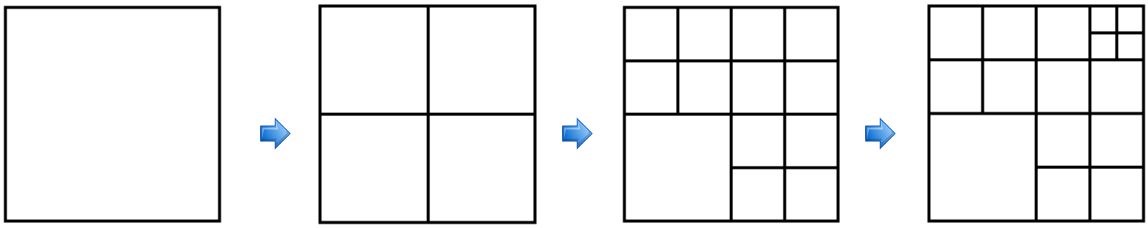


Figura 3-14: Refinamento recursivo de uma *quadtree* [119].

Para a triangulação propriamente dita, cada um dos quadrantes é subdividido em dois triângulos sendo possível recorrer a qualquer uma das primitivas discutidas em **3.1** para a concretizar. Todo o processo que controla o número de subdivisões de cada um dos quadrantes será discutido em detalhe em **3.4** e no contexto dos algoritmos que recorrem à *quadtree* como estrutura de suporte à triangulação do terreno, nomeadamente o algoritmo de Real Time Generation of Continuous Level of Detail (ver **4.3**), o algoritmo de Chunked LOD (ver **4.5**), o algoritmo de Rendering Very Large, Very Detailed Terrains (ver [107], o algoritmo de Terrain Occlusion Culling With Horizons (ver **4.6**) e o algoritmo de GPU Terrain Rendering (ver **4.8**).

3.2.3.2. Octrees

A extensão da *quadtree* a uma *octree* para representar objectos tridimensionais foi proposta independentemente por vários investigadores [92][95][134][170]. Uma *octree* não é mais do que uma versão a três dimensões de uma *quadtree* (ver **3.2.3.1**) que utiliza três planos paralelos a cada um dos eixos principais para subdividir o espaço. Assim a raiz de uma *octree* contém um cubo que envolve todos os objectos da cena. Os filhos de cada nó são oito cubos de igual tamanho que subdividem o nó pai em oito octantes, tal como está representado na **Figura 3-15**. Em relação às *quadtrees* tem como principal vantagem o facto de permitirem o teste a mais uma dimensão, o que para cenas muito povoadas de objectos se torna importante, justificando-se nesse caso a sua utilização.

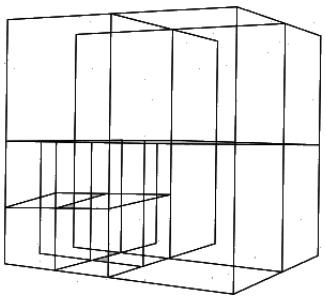


Figura 3-15: Uma *octree* [72].

No âmbito da geração de terrenos não é a estrutura mais utilizada. No entanto, se o objectivo for a representação de mais objectos para além do terreno, pode ser usada para tirar proveito da subdivisão espacial a três dimensões que proporciona.

3.2.3.3. Triangle Bintree

Uma *triangle bintree*, também designada de *binary triangle tree*, *bintritree*, *right triangular irregular network* ou simplesmente BTT, funciona da mesma forma que uma *quadtree*, só que em vez de segmentar um rectângulo em quatro partes, segmenta um triângulo em duas [119], combinando assim a simplicidade de uma árvore binária (em que cada nó tem dois descendentes) com a cobertura bidimensional de áreas que caracteriza uma *quadtree* [127]. Esta estrutura é especialmente útil na triangulação de *height fields* já que o método de subdivisão que utiliza aplica-se directamente em triângulos pelo que está, especialmente adaptada para descrever o processo de triangulação. Assim, numa *binary triangle tree* a raiz é um triângulo rectângulo (dois lados iguais e um ângulo recto de 90°) em que a subdivisão do espaço é feita pela divisão do triângulo a partir do vértice situado no ângulo de 90° até ao ponto intermédio oposto, dando origem a dois novos triângulos rectângulo num processo recursivo de divisão representado na **Figura 3-16**.

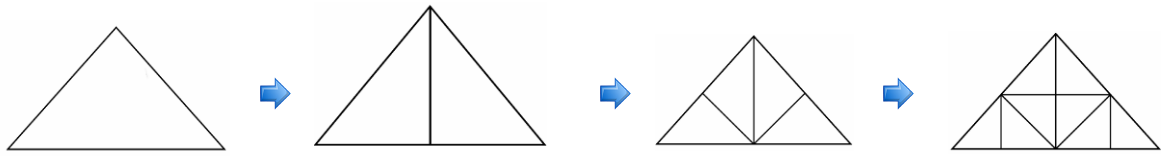


Figura 3-16: Subdivisões sucessivas de um triângulo rectângulo isósceles [165].

Desta forma a triangulação resulta, tal como nas *quadtrees*, de um processo recursivo que permite atribuir detalhe a cada uma das partes que compõem o terreno e que é controlado por uma métrica que estabelece o número de subdivisões para cada um dos dois triângulos em que o terreno é segmentado. Este processo é ilustrado na **Figura 3-17**.

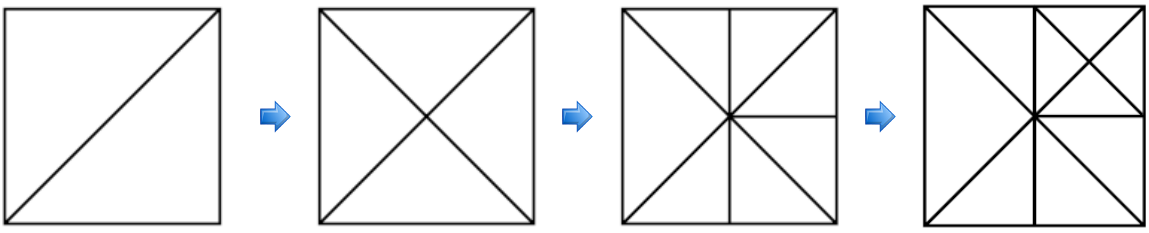
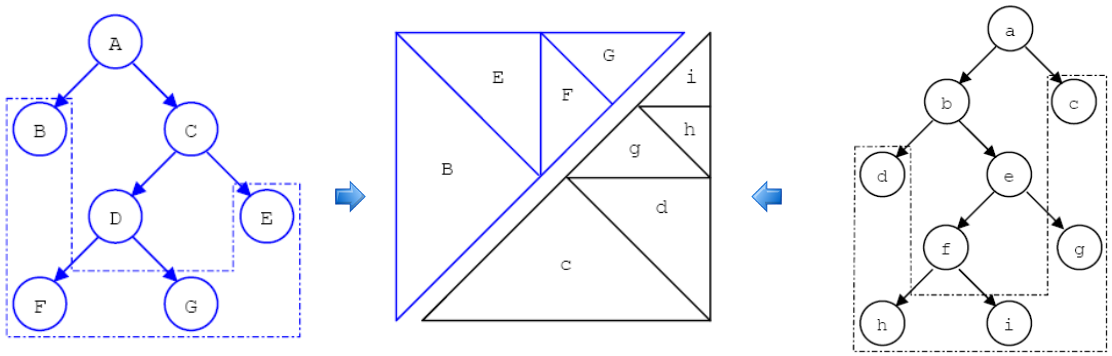


Figura 3-17: Refinamento recursivo de uma *binary triangle tree* [119].

Como é necessário que dois triângulos partilhem a hipotenusa para formar o quadrado que representa o terreno, têm de se utilizar duas *binary triangle trees* ([127] e [162]), tal como se pode verificar na **Figura 3-18**.



BTT esquerda Triangulação BTT direita

Figura 3-18: As duas *binary triangle trees* que constituem o terreno [162].

Esta estrutura foi pela primeira vez utilizada no algoritmo de Lindstrom [110] mas foi graças ao algoritmo de ROAM (ver **4.2)** que se tornou uma das estruturas mais populares na representação de terrenos.

3.3. Culling

***C**ulling* significa "remover do grupo" [135] e, no contexto da computação gráfica, isso é exactamente o que as técnicas de *culling* fazem, removem objectos que não contribuem para a imagem final, evitando assim o seu processamento por parte do *hardware* gráfico. Nesta perspectiva, um objecto ou parte do objecto pode não estar visível por diversas razões. Por exemplo, pode não estar dentro do volume de visualização do observador, o *frustum* ou, pode estar oculto por outro objecto para um determinado ângulo de visão. Qualquer uma destas situações, se não for corrigida, implica o envio para o *hardware* gráfico de objectos que não têm nenhuma contribuição para a imagem final, desperdiçando-se assim poder de processamento, o que, em determinadas situações, pode ter como consequência uma diminuição do desempenho da aplicação, especialmente em cenas com um grande número de objectos. Nesta perspectiva, o *hardware* gráfico deve, numa situação ideal, ser utilizado apenas para processar as partes da cena que são efectivamente visíveis. Para isso, podemos utilizar cinco técnicas principais de *culling* (representadas na **Figura 3-19**):

- *Back-Face Culling* (ver **3.3.1**), que tem como objectivo eliminar os polígonos, cujas faces apontam no sentido oposto ao do observador.
- *View-Frustum Culling* (ver **3.3.2**), que permite eliminar os objectos que não estão dentro do campo de visão do observador definido por um *frustum*.
- *Occlusion Culling*(ver **3.3.3**) que elimina os objectos ocultos por outros objectos.
- *Portal Culling* [7][8][194][195][198], que é especialmente útil no *culling* de cenas interiores, onde se tira partido das delimitações naturais (como paredes) estabelecendo um conjunto de células (por exemplo, quartos e salas) que estão conectadas por portais (que são normalmente portas ou janelas) através dos quais se projecta um *frustum* de modo a identificar toda a geometria visível através de cada uma delas. Este tipo de *culling* é uma espécie de *occlusion culling* sendo tratado separadamente devido à sua importância.
- *Detail Culling* [135], no qual se parte do princípio que os detalhes na cena, por exemplo objectos pequenos, contribuem pouco ou nada para a imagem final quando o observador está em movimento, podendo nesse caso ser removidos.

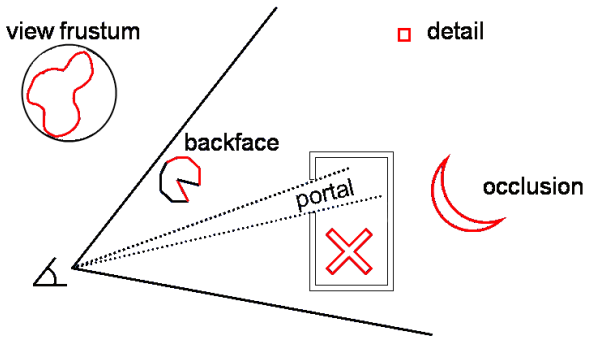


Figura 3-19: Os cinco tipos de *culling* [136].

Na figura estão representadas todas as primitivas que não são visíveis para os objectos representados e que devem ser numa situação ideal eliminadas. Num algoritmo ideal seria possível eliminá-las enviando apenas o *Exact Visible Set* (EVS) para *rendering*. Este é composto pelo conjunto de todas as primitivas que estão total ou parcialmente visíveis a partir do ponto de observação corrente. Como a obtenção desse conjunto, embora seja possível na teoria, não se torna na prática implementável, é utilizada uma aproximação designada de *Potentially Visible Set* (PVS) que não é mais que uma previsão do EVS e que pode ser classificado de duas formas: aproximada, em que contém a maioria das primitivas, mas não necessariamente todas os visíveis e ainda algumas invisíveis, e conservadora, que contém todos as primitivas visíveis e ainda algumas invisíveis. A segunda hipótese é assim na maior parte dos casos a melhor escolha já que garante que todas as primitivas visíveis são enviados para *rendering*.

No que diz respeito à geração de terrenos, são ainda de especial importância, as técnicas de *back-face culling*, *view frustum culling* e de *occlusion culling*, já que o *portal culling* é mais utilizado em modelos arquitecturais, ou seja, espaços fechados. O *detail culling* por seu lado, pode ser implementado no contexto das técnicas de nível de detalhe descritas em **3.4,** sendo necessário assumir para isso que um dos níveis implica não enviar o objecto para *rendering* [135].

3.3.1. Back-Face Culling

No *back-face culling* procura-se resolver um caso especial de oclusão denominado de *convex self occlusion*. Basicamente, se um objecto é fechado, ou seja, se tem uma parte interior e exterior bem definida, então algumas partes da superfície exterior são bloqueadas por outras da mesma superfície. Para isso consideram-se as normais de cada uma das faces que compõem o polígono (ver **Figura 3-20**). A ideia é comparar a normal de cada face com o vector de visualização, o vector dirigido do objecto para o observador. Se o ângulo deste vector com a normal for superior a 90º a face é descartada. Note-se que esta informação pode ser facilmente obtida fazendo o produto interno entre esses dois vectores.

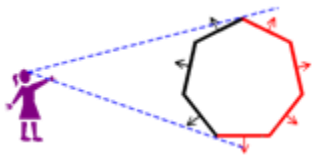


Figura 3-20: *Back-face culling* [223].

Utilizando uma API como o XNA [220] o DirectX [45] ou o OpenGL [150] o *back-face culling* é normalmente controlado com um pequeno conjunto de parâmetros que basicamente o activam ou desactivam. Para isso é necessário apenas que os polígonos sejam fechados [135].

3.3.2. View Frustum Culling

Esta é talvez a forma mais óbvia de *culling*, pois exclui todos os objectos que não se encontrem total ou parcialmente dentro do volume de visualização, e cujo processamento é desnecessário por não contribuírem em nada para a imagem final. Este volume de visualização é tipicamente denominado de *frustum* e não é mais do que uma pirâmide truncada, constituída por seis planos (ver **Figura 3-21**): frente (*near*), trás (*far*), esquerda (*left*), direita (*right*), cima (*top*) e baixo (*bottom*).

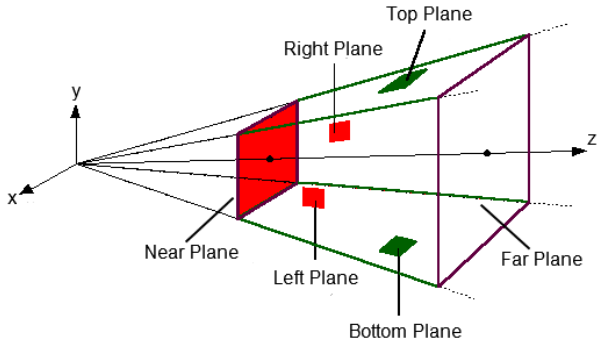
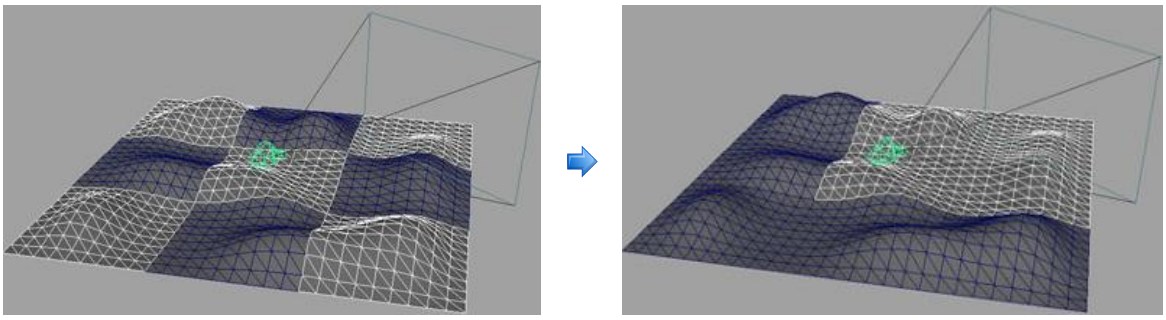


Figura 3-21: *View Frustum* [171].

O plano da frente e de trás são definidos, respectivamente, no ponto de vista do observador e no infinito. Na prática, porém, para o *far plane* é especificado um valor que representa a maior distância de visualização considerada. Uma implementação "ingénua" desta técnica testa todos os objectos contra o *frustum*, o que como é óbvio torna a sua aplicação em cenas com um grande número de objectos muito ineficiente. Felizmente, pelo particionamento do espaço num conjunto de blocos que contêm os objectos, é possível implementar esta técnica de *culling* hierarquicamente. Significa isto que podem ser utilizadas para esse fim algumas das estruturas hierárquicas referidas em **3.2**, como, por exemplo, as *quadtrees*. Ou seja, a ideia é percorrer a árvore testando cada um dos nós contra o *frustum*. Desta forma, utilizando como exemplo uma *quadtree* (o processo é semelhante para as outras estruturas hierárquicas) cada um dos nós é testado contra o *frustum*. Se algum deles estiver totalmente fora, então não é mais processado, isto é, toda a sua sub-árvore é descartada, bem como todos os objectos aí contidos, não sendo consequentemente enviados para processamento. Se, por outro, lado um nó intersecta o *frustum*, então os objectos desse nó são testados também eles contra o *frustum* para decidir se vão ser ou não incluídos na lista de objectos a enviar para *rendering*. Esta técnica de *culling* é essencial para um bom desempenho na geração de terrenos em tempo real, sendo normalmente utilizada em conjunto com uma técnica hierárquica de partição do espaço que no caso dos terrenos é na grande maioria dos casos uma *quadtree*. Como se pode verificar na **Figura 3-22** onde está representada uma triangulação de um terreno dividido em nove sectores, na posição da câmara, o *frustum* correspondente não intersecta todos os nove blocos definidos para o terreno. Consequentemente, após o *frustum culling*, são enviados apenas quatro sectores para *rendering*. Este facto seria ainda mais relevante se a câmara estivesse mais próxima de um dos cantos do terreno.



Divisão em blocos *Frustum culling*

Figura 3-22: A eficiência obtida pela partição do terreno em sectores [167].

3.3.3. Occlusion Culling

As técnicas de *occlusion culling* procuram evitar que objectos ocultos por outros objectos para um determinado ponto de vista sejam enviados para *rendering*. Este problema aparentemente estaria solucionado pelo facto de a grande maioria do *hardware* gráfico existente suportar o *z-buffering*[2]. No entanto, quando o número de objectos é muito grande uma aproximação ingénua que implique enviar para *rendering* todos esses objectos, é aqui e noutros contextos simplesmente impraticável, pois acaba por não ser suficiente, na medida em que o algoritmo tem de verificar todas as primitivas na cena para resolver o problema da visibilidade [84]. Este problema é, no entanto, bastante mais difícil de resolver do que, por exemplo, o *back-face* e o *frustum culling* (ver **3.3.1** e **3.3.2**), pois envolve relações entre objectos. Por exemplo, na **Figura 3-23** à esquerda, a menos que as esferas a vermelho sejam removidas da lista de objectos a enviar para *rendering*, cada uma delas vai ser processada e eventualmente recusada pelo algoritmo de *z-buffering*. Neste caso, o importante não é analisar a cena em si, pois não é a mais típica, mas constatar a dimensão deste problema quando o que se tem de representar é, por exemplo uma cidade (à direita na figura), ou no contexto desta dissertação, um terreno com um grande número de montanhas que se bloqueiam umas às outras para determinados ângulos de visão.

[2] O *z-buffer* ou *depth buffer* [26] armazena a profundidade dos objectos representados por cada pixel do ecrã. Quando é efectuado o *rendering* de um objecto a profundidade de cada pixel é comparada pela placa gráfica com a profundidade do pixel correspondente no *z-buffer*. Se for inferior significa que o novo *pixel* está mais próximo do observador ocultando por isso o anterior, pelo que, nesse caso o pixel é desenhado e o novo valor de z armazenado na posição correspondente. Caso contrário está oculto por outro *pixel* já existente no *z-buffer*.

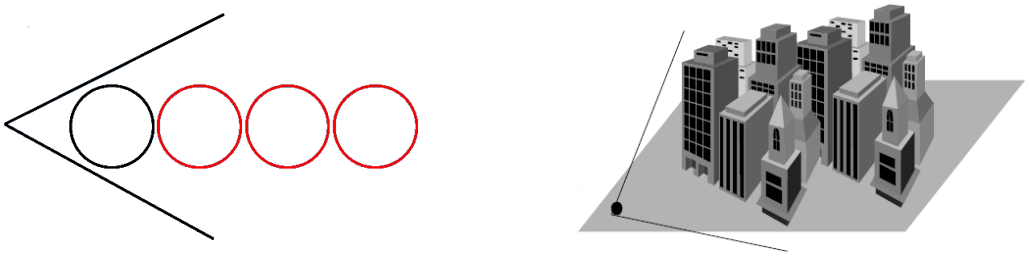
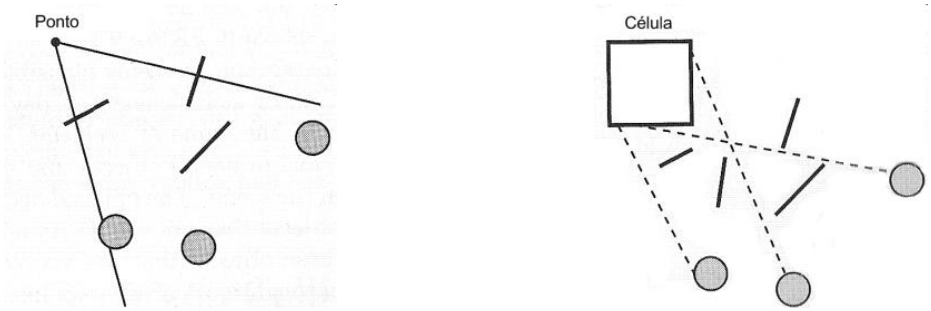


Figura 3-23: A importância do *occlusion culling* [136].

Existem dois tipos principais de algoritmos de oclusão: *point based* e *cell based*, também designados de *from point* e de *from region*, respectivamente. Os algoritmos *point based* são normalmente executados por *frame* e calculam o que está visível em cada posição na cena. Os algoritmos *cell based*, por outro lado, calculam a visibilidade para regiões no espaço normalmente definidas por caixas ou esferas. Desta forma, têm a vantagem de permitir a utilização da mesma informação de oclusão num conjunto de *frames* pelo menos, enquanto o ponto de observação estiver dentro de uma determinada célula. Em contrapartida, o cálculo da visibilidade nestes algoritmos é normalmente um processo pesado pelo que é executado numa fase de pré-processamento. Na **Figura 3-24** podemos verificar a diferença que existe entre estes dois tipos de abordagens. À esquerda, na figura, está representada a visibilidade a partir de um ponto ou seja corresponde a um algoritmo *point based*. Neste caso, partindo do ponto considerado nenhuma das esferas está visível pois o ângulo de visão é bloqueado pelas barreiras. Note-se que basta o ponto de observação mudar de posição para a situação mudar por completo, ou seja em cada *frame* a visibilidade tem de ser obrigatoriamente calculada. No caso representado à direita, o correspondente a um algoritmo *cell based*, as esferas seriam sempre marcadas como visíveis, pelo menos enquanto o ponto de observação se mantivesse dentro da célula isto porque o cálculo da visibilidade é efectuado para cada uma das regiões no seu todo. Isto é, se de alguma posição dentro dessa célula um determinado objecto é visível, assume-se que também o é para o resto da célula mesmo que não o seja para alguma posição específica. Por exemplo, no caso apresentado é sempre possível traçar pelo menos uma linha a partir de um ponto de observação dentro da célula que intersecte cada uma das esferas, ou seja, enquanto o ponto de visão se mantiver aí a informação utilizada sobre que objectos estão visíveis é sempre a mesma. De realçar ainda que na maioria dos casos esta informação é estática, ou seja calculada previamente e como tal basta obter a informação de oclusão respectiva não sendo necessário efectuar cálculos nenhuns em tempo de execução.



Point based *cell based*

Figura 3-24: *Point based occlusion culling* vs *cell based occlusion culling* [135].

Outra forma de categorizar os algoritmos de oclusão está relacionada com o espaço onde estes operam, por exemplo, existem algoritmos que operam em *image space*, isto é num espaço a duas dimensões depois de efectuada a projecção e algoritmos que operam em *object space* isto é, num espaço tridimensional. Não sendo o foco desta dissertação a classificação dos algoritmos de oclusão, pode-se consultar [135], por exemplo, para obter mais detalhes.

3.3.4. Algoritmos de Occlusion Culling

Em [225] Zhang definiu um algoritmo geral de oclusão que é de especial relevância na análise de algoritmos concretos. Este algoritmo representado na **Listagem 3-1** aborda a grande maioria dos princípios subjacentes em algoritmos de oclusão. Nesta listagem, a função `IsOccluded` é um teste de visibilidade que verifica se um objecto `g` do conjunto de objectos em `G` a serem enviados para *rendering* está ou não oculto. Para isso, têm em consideração no teste um conjunto `Or`, que representa a informação de oclusão considerada, ou seja, este conjunto não é mais do que os dados de oclusão que vão permitir identificar se o objecto `g` está ou não oculto. O `P`, por sua vez, é o conjunto de objectos que potencialmente podem ocultar outros objectos. O objectivo deste conjunto intermédio é armazenar objectos até a informação de oclusão conjunta ser suficientemente significativa para ser adicionada a representação de oclusão `Or`. Nessa situação é efectuada uma fusão com os objectos existentes em `Or`. Sendo assim, o algoritmo começa por testar o objecto `g` contra a informação de oclusão em `Or`. Se `g` estiver oculto face a essa representação já não é processado pois não vai contribuir para a imagem final. Se o objecto passar este teste, é enviado para *rendering* e adicionado ao conjunto `P`. O adicionar de objectos a este conjunto pode no passo seguinte fazer com que a informação de oclusão armazenada em `P` seja fundida com a representação de oclusão em `Or`. Para isso é necessário verificar se essa informação de oclusão nesse conjunto é suficientemente significativa, neste caso por intermédio da função `LargeEnough`. O adicionar dos objectos em `P` ao conjunto `Or` está normalmente dependente do algoritmo seleccionado e engloba mecanismos que permitem fundir os dois conjuntos. Esta operação que permite a fusão das representações dos objectos em `P` com o conjunto `Or` é muito importante na medida em que permite avaliar a contribuição de conjuntos de

objectos para a oclusão de outros objectos. Dito de outra forma, um objecto pode não ocultar por si só um outro objecto, mas em conjunto com outros isso já se pode verificar.

```
01 OcclusionCullingAlgorithm(G)
02  Or:= empty
03  P:= empty
04  For Each object g ∈ G
05   If(IsOccluded(g,Or)) Then
06    Skip(g)
07   Else
08    Render(g)
09    Add(g,P)
10    If (LargeEnough(P)) Then
11     Update(Or,P)
12     P:= empty
13    End If
14   End If
15 End OcclusionCullingAlgorithm
```

Parâmetros

G: Conjunto dos objectos a enviar para *rendering*

Variáveis

Or: Informação de oclusão.
P: Conjunto de objectos que potencialmente podem ocultar outros objectos.
g: Um objecto da lista de objectos a enviar para *rendering*.

Funções

IsOccluded(): Verifica se um objecto **g** está oculto face a representação de oclusão **Or**
Skip(): Não considera o **g** para *rendering*.
Render(): Efectua o *rendering* do objecto **g**.
Add(): Adiciona o objecto **g** ao conjunto de objectos **P** que potencialmente podem ocultar outros objectos.
LargeEnough(): Verifica se o conjunto **P** de objectos é significativo o suficiente para se actualizar a informação de oclusão **Or**.
Update(): Efectua a fusão da informação de oclusão em **P** com **Or**

Listagem 3-1: Pseudo-código do algoritmo geral de *occlusion culling* [225].

Um factor muito importante nestes algoritmos está relacionado com a ordem segundo a qual o processo de *rendering* é efectuado, ou melhor, a ordem pela qual os objectos são adicionados ao conjunto `G`. Se considerarmos, por exemplo, uma casa vista do exterior, não faz sentido efectuarmos o *rendering* de todos os objectos no seu interior e só depois a parte exterior. Significa isto que é possível ter um melhor desempenho, se ordenarmos os objectos em função da sua distância ao ponto de observação. Assim, nesta perspectiva, os mais próximos são os primeiros a ser enviados para *rendering* e também os primeiros a ser adicionados ao conjunto `Or`, o que permite construir uma representação de oclusão permitindo, neste exemplo, eliminar logo a partida todos os objectos que se encontrem dentro da casa, pelo que a ordenação é essencial nestes casos.

Os algoritmos de oclusão em particular e de visibilidade em geral são tópicos que têm sido alvo de uma ampla investigação pelo que existem diversas aproximações que se encaixam de um modo geral na caracterização feita no início desta secção. É nesse sentido que se referem de seguida alguns dos algoritmos mais comuns, numa listagem que não pretende ser de modo algum exaustiva nem detalhada. Para uma abordagem mais completa, consultar [32] onde é feita uma descrição mais detalhada de cada uma das aproximações existentes. As aproximações mais comuns são:

- *Occlusion Horizons*: É um algoritmo *point based* que opera no espaço da imagem e que permite a fusão do poder de oclusão dos objectos. Descrito pela primeira vez em [216] e mais tarde em [48] é especialmente útil no *culling* de cenas urbanas e de terrenos, já que assume que é possível descrever a geometria envolvida por meio de um *height field*. O conceito em si é muito simples: o objectivo é construir uma linha de horizonte que vai sendo redefinida em função

da altura dos objectos que vão sendo enviados para *rendering*. Tal como já foi referido, neste tipo de algoritmos a ordem interessa, pelo que, os objectos têm de ser ordenados do mais próximo para o mais distante em relação ao ponto de vista corrente. Dito de outra forma, este processo pode ser descrito como um plano que "varre" todos os objectos na direcção de visualização acumulando numa linha de horizonte a contribuição de cada um desses objectos, isto é, cada um deles é testado contra o horizonte. Se estiverem abaixo da linha por ele definida até àquele momento, significa que estão ocultos, não sendo dessa forma enviados para *rendering*. Este conceito está representado na **Figura 3-25**, onde a linha a amarelo representa o horizonte. Supondo que o prisma vermelho está atrás dos outros objectos, este é rejeitado de acordo com o algoritmo, já que está abaixo da linha de horizonte e, consequentemente, atrás dos outros objectos que formam esse horizonte.

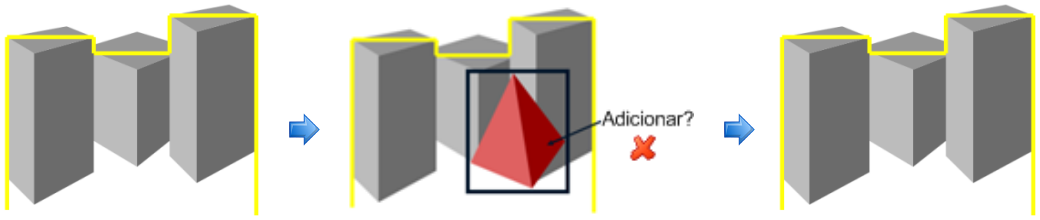


Figura 3-25: Teste de oclusão num algoritmo baseado em horizontes [135].

Na **Figura 3-26** é ilustrado o caso oposto, no qual o objecto a testar (representado a verde) está visível face à linha de horizonte corrente. Nessa situação o poder de oclusão desse objecto é adicionado e o objecto enviado para *rendering*.

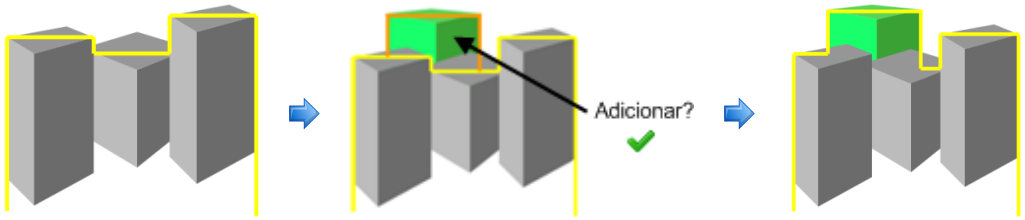


Figura 3-26: O adicionar de um objecto ao horizonte [135].

- *Hardware Occlusion Queries*: Este tipo de oclusão é efectuado no espaço da imagem e baseia-se numa funcionalidade adicionada às placas gráficas que permite efectuar o teste de oclusão em *hardware*. Esta funcionalidade surgiu pela primeira vez no sistema VISUALIZE da Hewlett-Packard [179] tendo sido posteriormente disponibilizada em *hardware* de consumo. Simplesmente permite efectuar uma pergunta ao *hardware* gráfico determinando se um polígono ou conjunto de polígonos, normalmente *bounding volumes* (ver **3.2.1**), esta visível em relação ao conteúdo do *z-buffer*. As versões mais recentes devolvem o número de *pixels*, n, que passaram o teste. Assim se $n = 0$ então o *bounding volume* está completamente oculto e os objectos que contêm podem ser descartados. Por outro lado, se $n > 0$, então uma fracção dos *pixels* falhou o teste pelo que o objecto tem de ser enviado para *rendering*. Muito embora esta solução pareça ser à

primeira vista óptima, existem dois problemas que condicionaram a sua adopção, nomeadamente o peso associado à chamada ao *hardware* propriamente dita e a latência causada pela espera do resultado. Para colmatar este problema, surgiram algumas aproximações, por exemplo as descritas em [16] e em [214].

- *Hierarchical Z-Buffering (HZB)*: Este algoritmo desenvolvido por Greene et al. e descrito em [80] e [81] teve uma influencia significativa nesta área de investigação [135]. O princípio base consiste em manter o modelo da cena numa *octree* (ver **3.2.3.2**) e o *z-buffer* de uma *frame* numa pirâmide de imagens, a *z-pyramid* (ver **Figura 3-27**). A *octree* permite o *culling* hierárquico de regiões na cena e a *z-pyramid* permite o *z-buffering* hierárquico de primitivas individuais e de *bounding volumes*, constituindo a representação de oclusão utilizada neste algoritmo. Tal como se pode verificar na figura, o nível mais detalhado desta pirâmide é um *z-buffer* standard. Em cada um dos níveis menos detalhados, a profundidade de um *pixel* representa a máxima profundidade dos quatro *pixels* correspondentes no nível anterior de maior detalhe. Com o intuito de manter esta pirâmide actualizada, sempre que um valor é escrito para o *z-buffer*, este é propagado para os níveis de menor detalhe.

Figura 3-27: *Z-buffer* hierárquico (*z-pyramid*) [183].

Basicamente o algoritmo consiste em percorrer os nós da *octree* da frente para trás, testando cada um deles contra a *z-pyramid*, teste que é efectuado com uma *occlusion query* como a descrita no ponto anterior. Na ausência de *hardware* especializado que implemente uma *z-pyramid*, é sugerida a utilização de um *z-buffer* convencional e a exploração da coerência entre *frames*. A ideia é que os nós da *octree* visíveis na *frame* anterior tendem a ser visíveis na próxima *frame*. Assim, uma vez terminada a geração da primeira *frame*, é criada uma lista dos objectos visíveis nessa *frame*. As *frames* subsequentes são geradas em dois passos: no primeiro é feito o *rendering* dos nós visíveis na *frame* anterior para o *z-buffer* construindo uma *z-pyramid* a partir do *z-buffer* obtido; no segundo passo, o algoritmo corre em *software* percorrendo a *octree* da frente para trás e ignorando os nós para os quais já foi efectuado o *rendering*. Finalmente, é actualizada a lista dos nós visíveis reiniciando-se de novo o processo.

- *Hierarchical Occlusion Map (HOM)*: Este algoritmo descrito por Zhang em [225] e [226] é similar em conceito ao algoritmo de *hierarchical z-buffering* descrito no ponto anterior, só que não necessita de nenhum suporte especial em *hardware*. A ideia é escolher explicitamente os objectos que vão ser utilizados na representação da oclusão construindo para estes um mapa hierárquico de oclusão (o HOM). O teste em si está dividido em duas partes: um teste de profundidade na direcção de visualização e um teste de sobreposição no espaço da imagem. É no teste de

sobreposição que é necessário construir o tal mapa hierárquico de oclusão. Assim, o primeiro passo consiste no *rendering* dos objectos seleccionados a branco sobre um *buffer* com fundo preto. O segundo passo consiste na construção de uma pirâmide como a representada na **Figura 3-28**. Nesta, os níveis menos detalhados são obtidos efectuando a média de quadrados de 2×2 *pixels* com o intuito de formar um mapa com metade da resolução em cada uma das dimensões, o que permite a fusão dos diferentes objectos numa única representação de oclusão. Na pirâmide, à medida que o detalhe diminui os *pixels* deixam de ser exclusivamente pretos ou brancos, passando também a assumir tons de cinzento. Deste modo a intensidade do *pixel* passa a representar a opacidade da região representada.

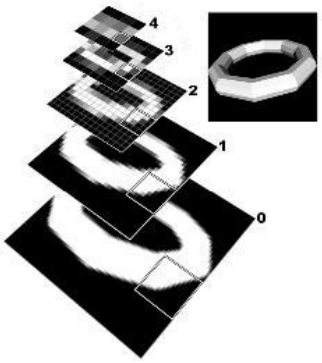


Figura 3-28: *Hierarchical Occlusion Map* [32].

O teste propriamente dito implica a projecção da *bounding box* do objecto alvo e o seu teste contra o mapa hierárquico de oclusão. Se o objecto sobrepuser *pixels* que não são opacos (isto é, diferentes de branco), significa que não pode ser descartado, pois contribui para a imagem pelo que o teste tem de continuar recursivamente até ao fundo da pirâmide. Se, pelo contrário, sobrepuser apenas *pixels* brancos, significa que o objecto está oculto. Nesse caso é necessário um teste de profundidade para verificar se o objecto está à frente ou atrás da representação de oclusão. Em [225] são descritas várias formas de efectuar esse teste, por exemplo, a mais simples, utiliza um plano colocado atrás da representação de oclusão contra o qual todos os objectos que passam o teste anterior são testados.

Os algoritmos de oclusão descritos abarcam a grande maioria das aproximações existentes. No que diz respeito a algoritmos especificamente concebidos para o tratamento da oclusão em terrenos, a maior parte deles utiliza o conceito de horizonte descrito no primeiro ponto como base, algo que se justifica em parte pela simplicidade dessa abordagem. São disso exemplo os algoritmos descritos em **4.6**, [48], [115]. Existem também aproximações que optam por um cálculo da visibilidade apenas numa fase de pré-processamento, tal como as descritas em [188], [208] e [227], mas que impedem no entanto alterações dinâmicas no terreno, pois os cálculos efectuados nessa fase são demasiado pesados para serem executados em tempo real. Em todas estas aproximações, os conceitos em si, não são muito diferentes dos descritos anteriormente para algoritmos mais gerais. No entanto, há que realçar que o terreno pelas suas características justifica

alguma especificidade: em primeiro lugar, pela grande quantidade de polígonos que lhe está associada e, em segundo, pela sua importância em cenas exteriores sendo em muitas situações o objecto que mais bloqueia outros objectos e como tal o que pode contribuir potencialmente para reduzir mais a quantidade de geometria enviada para *rendering*.

3.4. Nível de Detalhe

No domínio da computação gráfica sempre existiu uma necessidade de se chegar a um compromisso entre desempenho e complexidade. Efectivamente ocorre um confronto permanente entre um conjunto de características que são desejáveis em qualquer representação em tempo real. Deparamo-nos, com uma tensão constante entre realismo e velocidade, entre fidelidade e número de *frames* por segundo, entre mundos extremamente detalhados e ricos e uma animação fluida [119]. Tudo isto porque a complexidade dos objectos que pretendemos representar em tempo real parece constantemente crescer mais depressa que a capacidade do *hardware* para os representar. Técnicas como o *culling* (ver **3.3),** por si só já permitem grandes melhorias no desempenho de aplicações interactivas no entanto, e infelizmente, não é o suficiente. Isto porque, mesmo com o *exact visible set*, ou seja, o conjunto de todos os objectos que estão visíveis ou parcialmente visíveis para um determinado ponto de vista, a quantidade de dados enviados contínua a ser ainda para determinados tipos de cenas impossível de representar em tempo real. Desta forma é necessário recorrer a outras técnicas. Uma solução para este problema é simplificar a geometria de objectos pequenos ou distantes em relação ao ponto de vista do observador. Esta abordagem foi proposta pela primeira vez por Clark em [30] no que é considerado um dos trabalhos mais importantes desta área e onde pela primeira vez se reconheceu a redundância de utilizar muitos polígonos para um objecto que depois de transformado ocupa no ecrã poucos *pixels*. Técnicas baseadas neste princípio são denominadas de técnicas de nível de detalhe e têm como objectivo estabelecer uma ponte entre a complexidade e o desempenho regulando o nível de detalhe utilizado na representação da cena [119]. Para cada um destes objectos há que gerar, então, um determinado conjunto de representações progressivamente menos detalhadas: os níveis de detalhe. Na **Figura 3-29** um objecto com 69491 polígonos na sua versão original é simplificado progressivamente num conjunto de versões menos detalhadas tendo a última cerca de 76 polígonos.

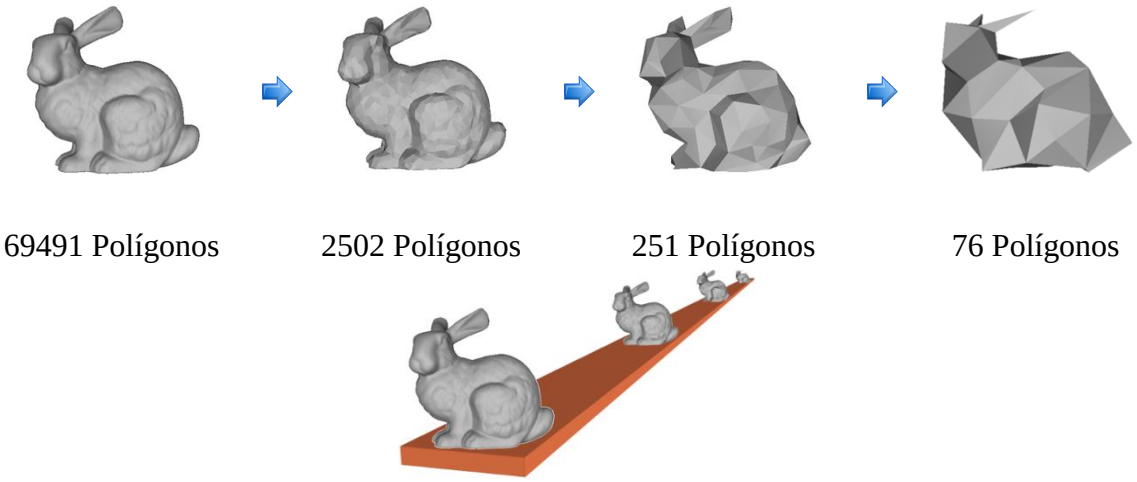


Figura 3-29: Diferentes níveis de detalhe dependentes da distância [118].

Partindo deste conceito, há que estabelecer agora formas de o concretizar, por isso, surgem nessa perspectiva quatro abordagens: a discreta, a contínua, a dependente do ponto de vista e a hierárquica. Em **3.4.1** é feita uma descrição mais detalhada de cada uma delas, no entanto, de um modo geral, permitem estabelecer, por um lado, o momento em que os diferentes níveis de detalhe são gerados, isto é, se numa fase inicial ou dinamicamente e, por outro, o modo como cada um deles é seleccionado. Paralelamente, para cada uma destas abordagens, surgem três grandes questões relacionadas com a aplicabilidade desta técnica, nomeadamente [118]:

- Como representar e gerar versões mais simples de um modelo complexo? (ver **3.4.2**).
- Quando se deve utilizar um determinado nível de detalhe para um objecto? (ver **3.4.3**).
- Como avaliar a fidelidade dos modelos simplificados?

A primeira está relacionada com as diferentes formas que existem de simplificar objectos para construir níveis de detalhe e a segunda com os métodos utilizados em tempo de execução para seleccionar cada um deles. Por fim, a última está relacionada com factores perceptuais, nomeadamente, se após a simplificação, o objecto mantém a forma original. Para isso, recorre-se normalmente a modelos perceptuais para medir a simplificação e prever o impacto desta no utilizador, o que pode ser particularmente importante em determinadas situações em que um determinado grau de precisão tem de ser atingido.

Em relação ao modo como se aplicam os níveis de detalhe na geração de terrenos em tempo real, a grande maioria dos algoritmos opta por particionar o terreno usando como suporte uma estrutura de subdivisão espacial (ver **3.2**) que permite determinar ou ajudar na selecção do nível de detalhe mais adequado para cada um dos blocos de terreno assim obtidos. Paralelamente são seleccionados apenas os blocos relevantes para o ponto de vista corrente recorrendo-se para isso a técnicas de *culling* (ver **3.3**). Comparando as técnicas de nível de detalhe aplicadas a objectos, como o representado na **Figura 3-29**,

com as aplicadas a terrenos cedo nos apercebemos que é de certa forma mais fácil lidar com representações de terrenos do que com modelos tridimensionais arbitrários porque a geometria está muito mais constrangida sendo constituída por grelhas uniformes de valores de elevação, dando por isso origem a algoritmos mais simples e especializados. Por outro lado, é mais difícil, em grande parte devido à natureza contínua dos terrenos e ao seu tamanho, o que implica paralelamente uma grande quantidade visível de geometria. Consequentemente, os algoritmos tendem a optar por técnicas em que a quantidade de detalhe fica dependente do ponto de vista do utilizador numa lógica em que se atribui mais detalhe às zonas mais próximas e menos às zonas mais distantes (ver **3.4.1.3**). Outra dificuldade também relevante na representação de terrenos, nomeadamente os de grandes dimensões, é a memória propriamente dita não ser suficiente para armazenar a representação na sua totalidade, sendo por isso necessário recorrer a técnicas de *paging* para permitir que os algoritmos operem *out-of-core*, ou seja capazes de representar terrenos de dimensão arbitrária (em **4.5**, **4.7**, e **4.10** são discutidos algoritmos de geração de terrenos em tempo real que endereçam esse problema).

Por fim, em **3.4.4** vão ser discutidos os tipos de problemas mais comuns que surgem pela aplicação de técnicas de nível de detalhe no contexto dos terrenos, nomeadamente as falhas, *t-junctions* e o *popping* e ainda algumas formas de os tratar.

3.4.1. Tipos

Existem quatro tipos principais de nível de detalhe que são utilizados na representação de objectos:

- Discreto (ver **3.4.1.1**), que implica a criação de várias versões do mesmo objecto em diferentes níveis de detalhe numa fase de pré-processamento.
- Contínuo (ver **3.4.1.2**), que permite a variação do nível de detalhe em tempo de execução.
- Dependente do ponto de vista (ver **3.4.1.3**), no qual o detalhe varia dinamicamente de acordo com o ponto de vista do utilizador.
- Hierárquico (ver **3.4.1.4**), que é uma generalização do conceito de nível de detalhe para representações hierárquicas de objectos.

3.4.1.1. Discreto

Este é o tipo de nível de detalhe mais tradicional e corresponde ao método proposto por Clark em [30]. Basicamente implica a criação, numa fase de pré-processamento, de múltiplas versões do mesmo objecto cada uma num nível de detalhe diferente (ver **Figura 3-30**). Posteriormente, em tempo de execução, é seleccionado o nível mais apropriado para representar o objecto mediante um critério específico que varia de acordo com a abordagem adoptada. A principal vantagem desta aproximação está precisamente na separação entre o processo de simplificação e o *rendering* propriamente dito. Ou seja, esse processo pode demorar todo o tempo que for necessário pois não existem constrangimentos nem de desempenho nem de tempo. Isto permite que os níveis de detalhe possam ser optimizados ao máximo e *inclusive* construídos "à mão", numa perspectiva de obter a melhor relação qualidade/número de polígonos possível. Este processo de simplificação pode também incluir alterações no modelo com o intuito de

seleccionar a primitiva (ver **3.1**) mais adequada em termos de desempenho para a API seleccionada. As desvantagens desta técnica tornam-se no entanto bastante evidentes quando é necessário efectuar alterações drásticas à geometria, nomeadamente em modelos de grande complexidade, para os quais, pode haver uma necessidade de subdividir grandes objectos ou combinar pequenos objectos, o que se torna praticamente impossível a utilização de uma abordagem como esta. Paralelamente, o limitado número de níveis de detalhe disponíveis é igualmente um factor limitador uma vez que a escolha fica restringida a um conjunto de níveis que podem não ser os mais adequados para representar uma determinada situação. Por outro lado, como os níveis de detalhe são pré-fabricados, o processo de simplificação não tem em conta a direcção de visualização do objecto, pelo que a simplificação efectuada tem a mesma aparência independentemente da posição do observador.

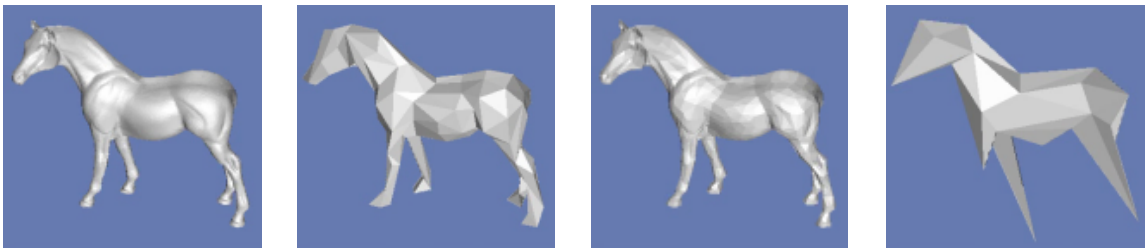


Figura 3-30: Nível de detalhe discreto.

3.4.1.2. Contínuo

Nesta aproximação, em vez de se criar um conjunto individual de níveis de detalhe numa fase de pré-processamento, o sistema de simplificação cria uma estrutura de dados que codifica um espectro contínuo de detalhe [119]. Esse conceito é representado na **Figura 3-31**, onde a processo de simplificação é visto como algo contínuo com uma granularidade muito maior do que a considerada no nível de detalhe discreto. O nível de detalhe mais apropriado é assim extraído desta estrutura em tempo de execução. De facto, a granularidade deste processo é uma das principais vantagens desta abordagem, até porque o nível de detalhe é especificado exactamente ao invés de ser seleccionado de um conjunto pré-estabelecido de opções. Desta forma, não se corre o risco de se utilizar menos ou mais polígonos para representar um determinado objecto numa situação em que o nível de detalhe intermédio, que era o mais apropriado, não exista. Ganha-se por isso, por um lado, na fidelidade e, por outro, no número de triângulos enviados para *rendering*. Outra das vantagens desta aproximação é suportar o *streaming* de objectos no qual um modelo base é seguido de uma série de refinamentos progressivos que são integrados dinamicamente. Esta propriedade permite que largos modelos sejam carregados do disco ou mesmo da rede, pelo *rendering* progressivo de versões cada vez mais detalhadas à medida, que os dados vão chegando. Um dos melhores exemplos desta técnica é o já famoso *Google Earth* [78] que utiliza um *streaming* progressivo de dados para representar as diferentes partes do planeta e que é integrado progressivamente à medida que o utilizador se movimenta.

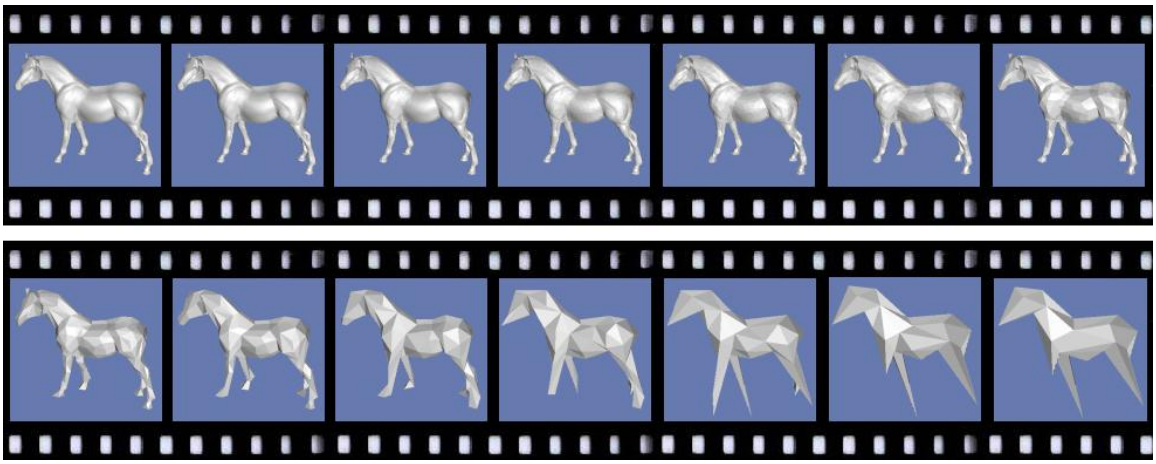


Figura 3-31: Nível de detalhe contínuo.

3.4.1.3. Dependente do Ponto de Vista

Neste método existe uma dependência entre o critério de simplificação e o ponto vista, o que torna possível a selecção dinâmica do nível de detalhe mais apropriado para a visão corrente do utilizador [119]. Procura-se, desta forma, atribuir detalhe apenas às zonas que mais necessitam, numa perspectiva de se produzir a melhor imagem possível, mantendo paralelamente os critérios de desempenho assumidos e obtendo ainda, por outro lado, uma melhor granularidade e consequentemente uma melhor fidelidade. Este tipo de nível de detalhe é especialmente importante na representação de terrenos em tempo real (ver **Figura 3-32**) nomeadamente terrenos de grandes dimensões em que existe claramente uma necessidade de representar as partes mais próximas com mais detalhe (como aliás é bem visível na figura). Por isso, nesses casos, são utilizadas métricas de erro dependentes da câmara o que permite variar o detalhe em função da sua distância e orientação.

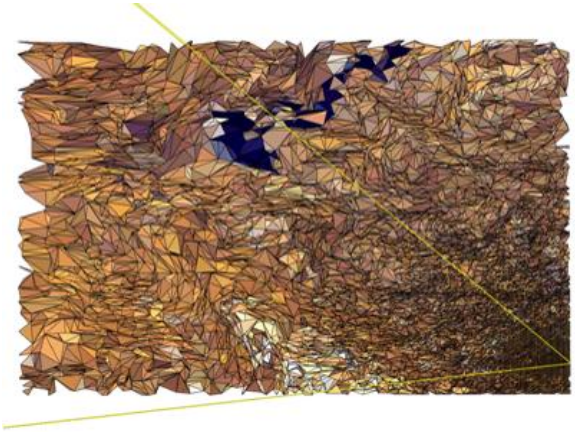


Figura 3-32: Nível de detalhe dependente do ponto de vista [119].

3.4.1.4. Hierárquico

Este tipo de nível de detalhe baseia-se no conceito de *scene graph*, mas aplicado à simplificação de objectos [58]. O objectivo é, em contraste com as técnicas de nível de detalhe tradicionais, representar a simplificação de conjuntos de objectos. Assim, assumindo que cada nó folha é a representação mais detalhada de cada um dos objectos que compõem uma cena (ver **Figura 3-33**), e também que cada um desses objectos armazena um determinado número de níveis de detalhe discretos, o que ocorre no nó superior é um agrupamento em um ou mais níveis de detalhe dos níveis de detalhe discretos de cada um dos seus descendentes. À medida que se vai subindo na hierarquia, os nós agrupam desta forma cada vez mais elementos da cena num conjunto de níveis de detalhe. É, assim, particularmente útil na representação de cenas muito complexas, compostas por diversos objectos. Uma das suas vantagens é permitir que numa travessia da hierarquia se possa parar muito mais cedo, pois os nós que agrupam os objectos têm por si só já uma representação dos seus descendentes.

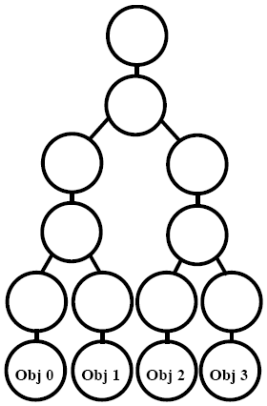


Figura 3-33: Nível de detalhe hierárquico [34].

3.4.2. Simplificação

Para obter os diferentes níveis de detalhe é necessário utilizar técnicas de simplificação. Estas têm como principal objectivo a diminuição do número de triângulos do objecto alvo, permitindo a criação das diferentes versões a serem utilizadas na sua representação. Para isso existe uma grande quantidade de algoritmos cuja aplicabilidade depende muito do tipo de objecto a simplificar e das concessões que se está disposto fazer, nomeadamente ao nível da fidelidade. Assim, no que diz respeito à simplificação propriamente dita, esta incide principalmente na geometria e/ou topologia do objecto. Antes de descrever como é efectuada essa simplificação, há que definir alguns dos conceitos associados. Assim, começando pela topologia, esta descreve a conectividade do objecto e é representada pelas arestas ou faces que ligam os vértices. Os vértices correspondem por sua vez à parte geométrica do objecto. Por fim, e relacionado com a topologia, o *genus* é o número de buracos num objecto (por exemplo, uma esfera e um

doughnut têm *genus* de 0 e 1 respectivamente). Neste contexto, uma alteração topológica é, por exemplo, o fecho de um ou mais buracos num objecto, por outro lado, uma alteração geométrica corresponde, por exemplo, à redução do número de triângulos de que é composto. Nesse sentido, a principal distinção está entre simplificações que alteram a forma do objecto, actuando ao nível da topologia, e simplificações que mantêm a forma do objecto reduzindo apenas o número de primitivas de que é composto mantendo, por isso, uma fidelidade relativamente boa em relação à versão original. Por exemplo, removendo o buraco num *doughnut* dá origem a algo não tem nada a ver com a representação anterior. Como os terrenos são estruturas regulares representadas por *height fields*, são, neste sentido, de especial importância os algoritmos de simplificação geométrica. Neste contexto, Heckbert e Garland descrevem em [88] uma classificação que é de especialmente relevante na simplificação de *height fields* e que está dividida em seis categorias:

- Métodos de grelha regular: o objectivo nestes métodos é criar representações progressivamente menos detalhadas do terreno, considerando-se para isso apenas os pontos de cada k-ésima linha e coluna. Significa isto que não são tidos em consideração todos valores de elevação, mas apenas um subconjunto destes em cada representação, o que diminui naturalmente a resolução mas tem também como consequência uma perda importante de detalhes.
- Métodos de subdivisão hierárquica: estes métodos constroem a triangulação pela subdivisão recursiva do terreno. Esta subdivisão apoia-se tipicamente em estruturas de partição do espaço, como as descritas em **3.2** (a *quadtree* por exemplo).
- Métodos orientados às características dos terrenos: esta aproximação baseia-se numa análise do *height field*, na qual se classificam cada um dos valores de elevação de acordo com a sua importância. É com base na contribuição de cada um destes pontos que são seleccionados um conjunto deles e construída a triangulação. Para isso utiliza-se normalmente o método de Delaunay (ver **2.2.2**). Por sua vez, a relevância de cada um dos pontos seleccionados, os *critical points* e das arestas que os ligam, as *break lines*, está normalmente associada a características topográficas do terreno, tais como picos, vales e rios.
- Métodos de refinamento: estes métodos operam *top-down* tendo por isso como ponto de partida uma representação simplificada do modelo, que é normalmente constituída por um número mínimo de triângulos. Por exemplo, no caso de um *height field* a primeira versão pode ser constituída por apenas dois triângulos. Esta representação inicial é depois recursivamente refinada pela inclusão de um ou mais pontos até o erro desejado ser atingido ou um número de vértices preestabelecido for alcançado. Neste método, a escolha dos pontos a serem introduzidos está relacionada com uma determinada métrica de erro que é normalmente uma medida do erro associado a inclusão do ponto vs a utilização de uma aproximação. Esta aproximação, que corresponde à não inclusão do ponto, pode ser mais ou menos precisa estando naturalmente condicionada pelos pontos adjacentes já incluídos na triangulação
- Métodos de dizimação: estes são o inverso dos métodos de refinamento operando *bottom-up* no modelo a simplificar, ou seja começam com o modelo no detalhe máximo e simplificam-no progressivamente num conjunto de passos que

removem vértices, triângulos ou outros elementos geométricos a cada passo do algoritmo.

- Métodos óptimos: estes métodos são importantes apenas a nível teórico pois a sua concretização muito embora possível está condicionada pelo tempo exponencial que leva a obter uma aproximação óptima à triangulação de um *height field*, portanto, é um problema NP-completo[3], não sendo assim prática a sua aplicação.

Dos métodos referidos, apenas os dois primeiros dão origem a triangulações regulares (ver **2.2.1**), todos os outros pela forma como executam a simplificação dão sempre origem a *triangulated irregular networks* (ver **2.2.2**).

3.4.3. A Escolha do Nível de Detalhe

Claramente, a questão mais importante na gestão do nível de detalhe é quando mudar para uma versão mais ou menos detalhada de um determinado objecto [119]. Para isso são considerados um conjunto de factores que contribuem para métrica utilizada. Esta métrica permite quantificar a diferença que existe entre dois níveis de detalhe distintos, devendo o valor obtido estar abaixo de um determinado limiar para que a mudança se possa concretizar. Assim, antes de descrever as métricas mais comuns é importante caracterizar os principais factores a ter em consideração, nomeadamente [119]:

- A distância do objecto ao ponto de vista (ver **3.4.3.1**). Esta é utilizada na selecção do nível de detalhe pela atribuição a cada um dos níveis de um valor de distância a partir do qual cada um deles poderá ser seleccionado para representar o objecto.
- O tamanho do objecto (ver **3.4.3.2**), onde se tem em consideração o tamanho associado para construir as variações entre níveis de detalhe.
- O erro (ver **3.4.3.3**), para o qual se tem em consideração um determinado valor de erro associado à variação do nível de detalhe entre níveis sucessivos.
- A prioridade, que tem em consideração a importância de determinadas características do objecto que por serem de especial relevância para compreensão da cena não podem ser simplificadas. Assim, para determinados objectos a prioridade pode ditar a selecção de um nível de detalhe mais elevado no caso de percepção da cena poder vir a ser afectada pela selecção de um nível inferior.
- Histerese, que é basicamente um atraso introduzido na transição entre níveis de detalhe com o objectivo de reduzir o efeito de *flickering* que ocorre quando um objecto muda constantemente de nível de detalhe no limiar da distância de transição.

[3] Um problema está em NP se pode ser resolvido não deterministicamente em tempo polinomial. Um processo de resolução não determinístico é feito "adivinhando" a solução e verificando de seguida se essa solução está correcta. Intuitivamente o problema está em NP se existe um polinómio $P(n)$, tal que se a solução for encontrada pode ser verificado se está correcta em $O(P(n))$.
Um problema é NP-completo se é pelo menos "tão difícil" como qualquer outro problema em NP. De uma forma mais formal um problema P_1 em NP é NP-completo se a seguinte propriedade se verifica: para todos os outros problemas P em NP se P_1 pode ser resolvido deterministicamente em $O(f(n))$ então P_i pode ser resolvido em $O\left(P(f(n))\right)$ para um polinómio P. A teoria dos problemas NP-completos é discutida em mais detalhe em [73].

- Condições ambientais. A utilização de efeitos ambientais como o nevoeiro ou o fumo, quando aplicáveis ao ambiente que envolve o objecto, permite níveis de detalhe mais baixos. Não é, no entanto, claramente apropriada para todas as situações.
- Perceptuais, que leva em consideração o modo como o sistema visual humano funciona, nomeadamente o conjunto de características que lhe estão associadas, e que podem ser utilizadas para seleccionar o nível de detalhe mais apropriado. Por exemplo, o nosso sistema consegue detectar menos detalhes em objectos na visão periférica ou que se movem rapidamente. Para objectos nestas circunstâncias é possível reduzir o nível de detalhe, sem que seja detectada uma degradação perceptível de qualidade.

No que diz respeito aos algoritmos utilizados na geração de terrenos em tempo real os três factores mais importantes são a distância, o tamanho e o erro. Desta forma, as métricas mais comuns são [162]:

- Erro geométrico, que se baseia exclusivamente no valor do erro.
- Erro geométrico + distância, que tem em consideração os dois factores, isto é a distância à câmara e o valor do erro.
- Erro geométrico em *pixels*, que tem em consideração tal como a anterior a distância à câmara e o valor do erro, só que neste caso o valor do erro é projectado para coordenadas de ecrã.

3.4.3.1. Distância

Baseia-se na associação de um valor de distância a cada um dos níveis de detalhe do objecto, dependendo a utilização de cada um deles da distância ao ponto de vista. Na **Figura 3-34** está representado este conceito para blocos de terreno.

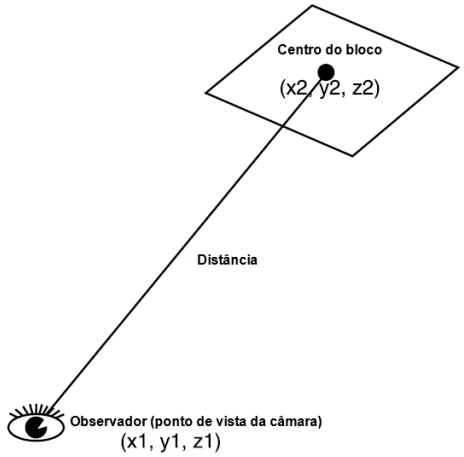


Figura 3-34: Cálculo da distância entre o centro do bloco de terreno e a câmara [165].

Para o seu cálculo é tipicamente utilizada a **Equação 3-1** também designada de distância euclidiana ou norma $L2$ ou então a **Equação 3-2**, a distância de *Manhattan* ou

norma $L1$. A norma $L1$ é uma optimização que evita o cálculo da raiz quadrada na norma $L2$ sendo por vezes utilizada em sua substituição. Outra forma comum de optimizar este cálculo é utilizando a norma $L2$ mas aplicando uma potência de dois a cada um dos seus membros, o que na **Equação 3-1** corresponde ao cálculo de d^2.

$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$	x_n: Valor da coordenada x.
	y_n: Valor da coordenada y.
	z_n: Valor da coordenada z.

Equação 3-1: Distância euclidiana (norma $L2$).

$d = \lvert x_1 - x_0 \rvert + \lvert y_1 - y_0 \rvert + \lvert z_1 - z_0 \rvert$	x_n: Valor da coordenada x.
	y_n: Valor da coordenada y.
	z_n: Valor da coordenada z.

Equação 3-2: Distância de *Manhattan* (norma $L1$).

A utilização da distância por si só não é suficiente pois não tem em consideração alguns factores importantes como o impacto do efeito de perspectiva na representação do objecto, podendo causar *inclusive* um efeito de *popping* (ver **3.4.4.2**) mais pronunciado.

3.4.3.2. Tamanho

Este factor implica medir o espaço ocupado pelo objecto em coordenadas de ecrã para determinar o nível de detalhe mais apropriado para o representar. Para isso o *bounding volume* (ver **3.2.1**) respectivo é projectado para o espaço do ecrã, ou seja, é tido em consideração o efeito de perspectiva que torna os objectos mais pequenos quanto maior for a sua distância ao ponto de vista. É então com base no tamanho do *bounding volume* em *pixels* que a selecção é efectuada. Isto é, a contribuição do objecto para a imagem final determina qual o nível de detalhe seleccionado para o representar. Assim nesta perspectiva, se o objecto estiver distante não é necessário utilizar um nível de detalhe muito elevado, pelo que este factor está relacionado com o factor de distância referido anteriormente (ver **3.4.3.1**).

3.4.3.3. Erro

Uma aproximação bastante comum nos algoritmos de geração de terrenos em tempo real está relacionada com o valor do erro resultante de uma mudança de detalhe. Neste contexto, o erro está relacionado com a diferença de elevações entre dois níveis de detalhe distintos, ou seja a quantidade de informação de elevação que se perde de um nível para outro. Consequentemente, este erro é medido então como a maior distância entre os valores de elevação dos dois níveis de detalhe considerados: o nível corrente e o nível alvo. O resultado é depois na maior parte dos casos projectado para o espaço de ecrã, permitindo medir o erro em *pixels* que resulta da mudança de detalhe considerada.

3.4.4. Problemas

A utilização de níveis de detalhe, embora importante na medida em que permite reduzir drasticamente o número de triângulos a enviar para *rendering*, causa em contrapartida um conjunto de problemas que têm de ser resolvidos ou, no limite, minimizados para que o utilizador não se aperceba deles em tempo de execução. No caso dos terrenos, a sua partição (ver **3.2**) num conjunto de blocos simplificados individualmente de acordo com um ponto de vista dinâmico (ver **3.4.1.3**) dá origem a blocos adjacentes com diferentes níveis de detalhe, pelo que não é possível garantir nem a continuidade espacial, nem a continuidade temporal entre esses blocos. A continuidade espacial diz respeito a transição geométrica entre os blocos e está relacionada com problemas como as falhas e as *t-junctions*. As falhas são causadas por um vértice num nível de detalhe mais elevado que não existe no nível inferior que lhe é adjacente. Este novo vértice introduz por vezes, uma variação na elevação em relação a um nível de detalhe inferior que lhe é adjacente, dando por isso origem a pequenos "buracos" na geometria. As *t-junctions* ocorrem quando um vértice de um nível de detalhe mais elevado não é partilhado por um nível de detalhe mais baixo, que lhe é adjacente. Ou seja, é a mesma situação que ocorre nas falhas, mas sem a variação na elevação. A continuidade temporal diz respeito aos efeitos visuais em tempo de execução resultantes de diferentes níveis de detalhe adjacentes. O principal problema é o efeito de *popping* que resulta do impacto visual de uma mudança de nível de detalhe, que causa um súbito aparecimento/desaparecimento de geometria, um efeito que pode constituir um factor de distracção para o utilizador quebrando assim a experiência visual. Em **3.4.4.1** e **3.4.4.2** são discutidas em detalhe as técnicas normalmente utilizadas para eliminar ou minimizar o impacto visual destes problemas.

3.4.4.1. Falhas & T-junctions

A Figura 3-35 ilustra os dois tipos de problemas mais comuns entre blocos adjacentes com níveis de detalhe distintos: as falhas e as *t-junctions*. As falhas são as que têm o efeito mais pronunciado (ver **Figura 3-36**) sendo a sua presença claramente incomportável na representação de um modelo tridimensional. As *t-junctions*, por seu lado, causam pequenas descontinuidades devido a erros de arredondamento e diferenças no *shading* das arestas afectadas [119]. Por isso mesmo são, por vezes, ignoradas pois o seu efeito não é tão pronunciado como o das falhas.

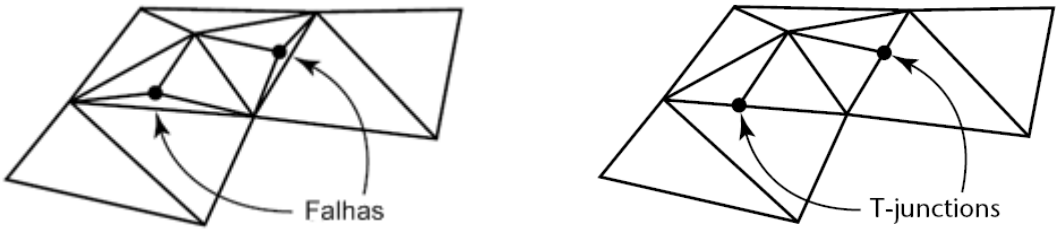


Figura 3-35: Falhas & *t-junctions* [222].

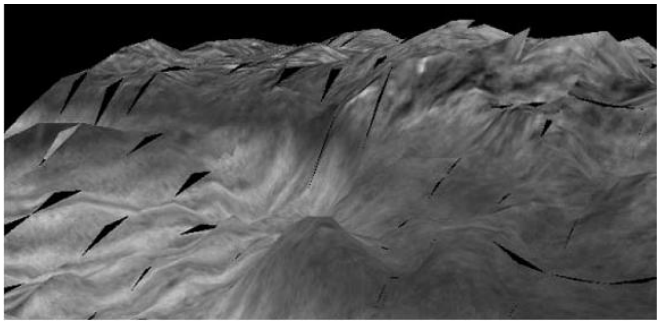
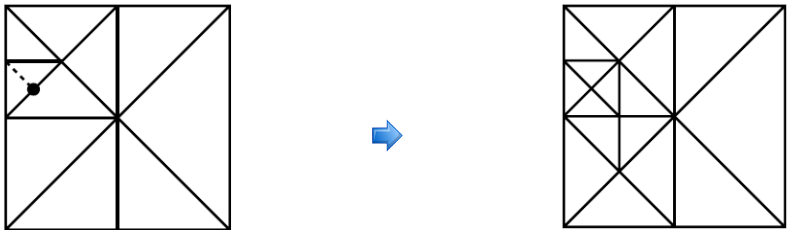


Figura 3-36: Falhas no terreno.

Nesta perspectiva, descrevem-se de seguida algumas das soluções mais comuns [119][200][201] para estes problemas, nomeadamente:

- A divisão recursiva dos triângulos que envolvem a falha. O objectivo é que o nível de detalhe nas regiões "fronteira" seja igual (ver **Figura 3-37**). Esta abordagem implica a introdução de mais triângulos mas permite obter resultados bastante convincentes, sendo utilizada principalmente nos algoritmos que recorrem a *binary triangle tree* (ver **3.2.3.3**) como estrutura de suporte à triangulação, tais como o ROAM (ver **4.2**) ou o algoritmo de Lindstrom [110].



A linha a tracejado vai introduzir uma falha/*t-junction*.

O resultado da divisão recursiva efectuada para evitar essa situação

Figura 3-37: Eliminação de falhas e *t-junctions* pela divisão recursiva [119].

- A alteração dos vértices no nível de detalhe mais elevado. O objectivo é que estes tenham a mesmo elevação da aresta correspondente no nível de detalhe inferior. Esta alteração pode ser feita no momento em que existe a mudança de nível de detalhe ou ser interpolada de um nível de detalhe para outro, o que corresponde ao método de Geomorphing descrito em **3.4.4.2**, utilizado para minimizar o efeito de *popping*. No entanto, independentemente da forma como é alterado o valor de elevação, este método dá origem a *t-junctions* não sendo por isso por si só suficiente para resolver estes dois problemas.
- O ignorar dos vértices no nível de detalhe mais elevado. Esta solução é implementada, por exemplo, no algoritmo de Real Time Generation of Continuous Level of Detail (ver **4.3)** e em algumas versões do Geomipmapping (ver **4.4**), estando a sua concretização dependente da primitiva utilizada na triangulação. No primeiro caso, são utilizados leques de triângulos (ver **3.1.2**), para os quais é muito fácil ignorar um determinado vértice, bastando para isso

omiti-lo da lista de vértices ou de índices associada. No segundo caso, são utilizadas tiras de triângulos (ver **3.1.3**), pelo que é necessária a modificação dos índices em todos os triângulos dependentes desse vértice, de modo a que o vértice "a mais" no nível de detalhe mais elevado seja ignorado. A **Figura 3-38** ilustra para cada uma das primitivas referidas as modificações necessárias para que seja possível eliminar as falhas entre dois blocos adjacentes de 5×5 e 3×3. Tanto num caso como noutro, basta modificar a lista de índices associada para que os vértices sejam ignorados e os dois níveis de detalhe possam "encaixar" perfeitamente.

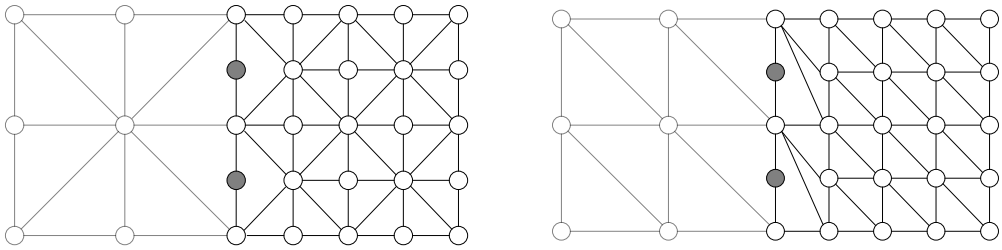
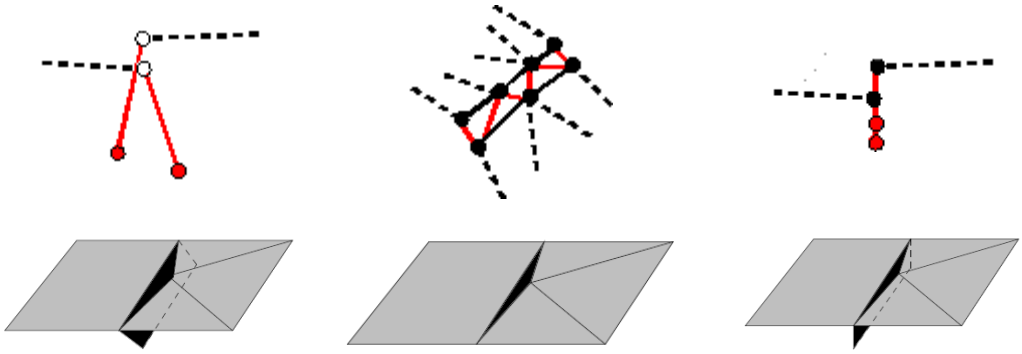


Figura 3-38: Correcção de falhas dependente da primitiva.

- A introdução de um vértice extra na aresta do nível de detalhe inferior igual ao vértice no nível de detalhe mais elevado. Embora esta abordagem produza bons resultados implica mais uma vez a modificação da geometria.
- Impedir a simplificação dos vértices que se encontram na fronteira de um determinado bloco de detalhe. Este método foi utilizado por Hoppe em [90] mas o algoritmo aí descrito utiliza um processo de simplificação que dá origem a *triangulated irregular networks* (ver **2.2.2**).
- A introdução de novos triângulos. Esta solução pode ser aplicada de diversas formas, estando três das mais comuns representadas na **Figura 3-39**. A primeira envolve a utilização de *flanges*, uma malha triangular que forma um ângulo superior a 90° com a aresta de cada um dos blocos ao qual é adicionada. O objectivo é colocar a *flange* na aresta do bloco de maior detalhe seleccionando o ângulo e o tamanho mais adaptado de modo a cobrir a falha entre os dois blocos. O problema reside precisamente na escolha do ângulo e do tamanho mais apropriado pois é algo que varia de caso para caso. A segunda utiliza *ribbons*, ou seja, um triângulo como " ponte" entre as duas arestas adjacentes. O problema mais óbvio aqui é como descobrir as coordenadas exactas dos vértices que compõem esse triângulo. Finalmente, uma das melhores aproximações a este problema foi a proposta no âmbito do algoritmo de Chunked LOD (ver **4.5**): as *skirts*. Estas combinam as virtudes das *flanges* e dos *ribbons*. A ideia é criar uma espécie de "saia" vertical à volta de cada um dos blocos. A principal vantagem deste método é não ser necessário alterar a geometria, sendo por isso bastante rápido, muito embora implique a utilização de mais vértices.



a) *Flange* b) *Ribbon* c) *Skirt*

Figura 3-39: Formas de corrigir as falhas num terreno [201].

3.4.4.2. Popping

O efeito de *popping* é uma consequência directa da mudança de nível de detalhe que ocorre na representação de um objecto. Essa mudança tem um impacto visual bastante perceptível que não contribui em nada para o realismo da cena. Assim, com o intuito de minimizar este efeito, recorre-se normalmente a um de dois métodos [119]: o *alpha blending* ou o Geomorphing. O *alpha blending* procura tornar a transição entre dois níveis de detalhe menos perceptível, associando um valor de opacidade ou valor *alpha* a cada um deles. Este valor vai de 1.0, para o qual o objecto é opaco, a 0.0, no qual o objecto é invisível. O objectivo é estabelecer para cada objecto uma zona de transição que tem como centro o ponto de mudança do nível de detalhe. Por exemplo, se existe uma mudança aos 100 metros e a zona de transição tem 10 metros, o *alpha blending* ocorre entre os 95 e os 105 metros. Nesse intervalo é feito o *rendering* simultâneo dos dois níveis de detalhe. Assim, no caso de uma mudança de um nível de detalhe mais alto para um nível mais baixo, que para efeitos de exemplo são aqui referidos como LOD1 e LOD2 respectivamente, aos 95 metros ocorre o *rendering* do LOD1 com 1.0 de opacidade e do LOD2 com 0.0 de opacidade. À medida que o ponto de vista se vai afastando a opacidade do LOD1, diminui progressivamente e a do LOD2 aumenta, até que o LOD1 desaparece sendo substituído pelo LOD2. Este conceito é exemplificado na **Figura 3-40**, onde é efectuada uma transição entre um nível de detalhe representado por um bloco de terreno com 3 vértices para um nível de detalhe com 2 vértices. A principal desvantagem deste método é o facto de durante o período de transição ser necessário efectuar o *rendering* dos dois objectos, o que tem como consequência um aumento dos triângulos, podendo isso ser problemático na medida em que a mudança do nível de detalhe deve-se precisamente à necessidade de reduzir o número de triângulos. Assim convém que a zona de transição neste método seja a mais curta possível para que se possa tirar partido da mudança de nível de detalhe em relação ao número de triângulos enviados para *rendering*.

Figura 3-40: *Alpha Blending*.

O *alpha blending* actua no espaço da imagem enquanto que o Geomorphing actua a nível geométrico executando o *morphing* dos vértices de um nível de detalhe para outro. É por isso muitas vezes designado de *vertex morphing*. O objectivo é interpolar a posição dos vértices a modificar de um nível de detalhe para outro de modo a que se torne praticamente imperceptível a mudança e, consequentemente, seja possível eliminar o efeito de *popping*. Esta estratégia não implica ao contrário do *alpha blending* a introdução de mais geometria, é no entanto necessário modificar a posição dos vértices. Este processo é ilustrado na **Figura 3-41** onde a posição do vértice vai sendo modificada de *frame* para *frame* com o seu valor interpolado linearmente entre a sua posição inicial e final, ou seja, da esquerda para a direita na figura. A nível dos terrenos a forma como esta técnica é aplicada depende muito da implementação adoptada em cada um dos algoritmos pelo que as diferentes aproximações são descritas no contexto dos algoritmos correspondentes (ver **4**). No entanto, vale a pena referir que na grande maioria dos casos é efectuada ao nível da componente programável da placa gráfica, mais precisamente ao nível do *vertex shader* para a geometria e ao nível do *pixel shader* para as texturas associadas. Outro factor relevante é se no momento em que ocorre o *morphing* este é feito de forma independente para cada um dos blocos de terreno ou se pelo contrário é feito tendo em consideração os blocos vizinhos. Esta última forma de aplicar o Geomorphing é abordada em detalhe no contexto do algoritmo de Geomipmapping (ver **4.4**)

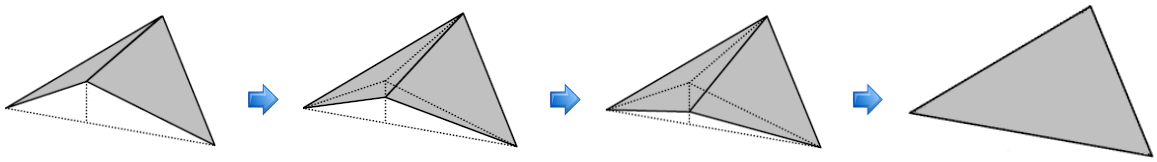


Figura 3-41: Geomorphing [162].

3.5. Sumário

O objectivo deste capítulo foi abordar em detalhe o processo de geração do terreno, nomeadamente, a construção da geometria e os diferentes métodos utilizados para reduzir o número de triângulos a enviar para *rendering*. No que diz respeito à construção da geometria (ver **3.1**), discutiu-se em detalhe os diferentes tipos de primitivas que podem ser utilizados, mais especificamente as listas de triângulos (ver **3.1.1**), os leques de triângulos (ver **3.1.2**) e as tiras de triângulos (ver **3.1.3**), tendo sido posteriormente expostas as diferenças que existem entre eles, bem como quais os mais utilizados na geração de terrenos em tempo real. Desta análise conclui-se que as melhores são as listas

e as tiras de triângulos em grande parte por permitirem efectuar o menor número de chamadas tendo consequentemente um melhor desempenho.

Em relação aos métodos utilizados na redução da geometria, a maioria deles está dependente de estruturas de dados espaciais (ver **3.2**). Estas permitem efectuar o particionamento dos diferentes objectos que constituem a cena e no caso especifico dos terrenos permitem a partição do terreno em múltiplos blocos e consequentemente o remover ou o diminuir do número de triângulos necessários por bloco. Para que a partição espacial seja menos complexa, cada um dos objectos é representado por um *bounding volume* (ver **3.2.1**) que delimita a área que ocupa e que permite que se actue não ao nível do objecto propriamente dito, mas sobre uma representação deste inerentemente mais simples, por exemplo sobre uma *axis aligned bounding box* (AABB) ou uma *bounding sphere* (BS). Estes *bounding volumes* estão organizados nas estruturas de dados espaciais numa hierarquia que estabelece as relações entre eles, tendo-se abordado dois tipos de hierarquias: as *bounding volume hierarchies* (ver **3.2.2**) e as *spatial partitioning hierarchies* (ver **3.2.3**). As *spatial partitioning hierarchies* são especialmente relevantes dado que são as estruturas de dados espaciais de particionamento hierárquico mais utilizadas na geração de terrenos em tempo real. Algumas das estruturas mais importantes deste grupo são as *quadtrees* (ver **3.2.3.1**), as *octrees* (ver **3.2.3.2**) e as *triangle bintrees* (ver **3.2.3.3**), algumas delas com um papel importante não só na partição espacial dos terrenos, mas também no processo de triangulação propriamente dito, como é o caso das *quadtrees* e das *triangle bintrees*.

Utilizando como base as estruturas espaciais de particionamento abordaram-se de seguida as duas principais formas de reduzir o número de triângulos enviados para *rendering*: o *culling* (ver **3.3**) e a redução do nível de detalhe (ver **3.4**). No que diz respeito ao *culling*, o objectivo é remover os objectos que não contribuem para a cena isto é, aqueles que não vão ser visualizados e que não necessitam por isso de serem processados. O *back-face culling* (ver **3.3.1**), o *view frustum culling* (ver **3.3.2**) e o *occlusion culling* (ver **3.3.3**) foram os tipos de *culling* discutidos destacando-se o *view frustum culling* e o *occlusion culling*. O primeiro porque de uma forma simples permite remover uma grande quantidade de geometria, apoiando-se para isso nos métodos de partição espacial já aqui discutidos, o segundo porque permite a remoção de objectos ocultos por outros objectos, existindo *inclusive* um conjunto de algoritmos (ver **3.3.4**) especialmente adaptados para resolver esse problema.

Finalmente, discutiram-se as técnicas de nível de detalhe (ver **3.4**), nomeadamente os diferentes tipos de nível de detalhe (ver **3.4.1**): o discreto (ver **3.4.1.1**), o contínuo (ver **3.4.1.2**), o dependente do ponto de vista (ver **3.4.1.3**) e o hierárquico (ver **3.4.1.4**). Destes o mais relevante para gestão do nível de detalhe em terrenos é o dependente do ponto de vista dado que implica a selecção dinâmica do nível de detalhe mais apropriado para a visão corrente do utilizador, bem como uma simplificação diferente em partes distintas do terreno, o que mais uma vez é conseguido graças aos métodos de partição espacial aqui discutidos. Foram ainda abordados tópicos como os métodos de simplificação (ver **3.4.2**) e a forma como o nível de detalhe é seleccionado (ver **3.4.3**) para cada objecto. Por fim foram discutidos os problemas resultantes de técnicas de nível de detalhe (ver **3.4.4**) com especial ênfase nos terrenos e em dois problemas principais: as falhas/*t-junctions* (ver **3.4.4.1**) e o *popping* (ver **3.4.4.2**).

4. Algoritmos de Geração de Terrenos em Tempo Real

Numa perspectiva histórica, existiu uma grande evolução dos algoritmos de geração de terrenos em tempo real. As primeiras técnicas datam dos anos setenta, oitenta [68][40][177][35], mas é possível encontrar referências desde essa altura até aos dias de hoje. No entanto, foram nos anos noventa que surgiram alguns dos algoritmos mais importantes neste campo nomeadamente o *Progressive Meshes* [90], o algoritmo de Lindstrom [110], o algoritmo de Real Time Generation of Continuous Level of Detail (ver **4.3**) e o algoritmo de ROAM (ver **4.2**). O ROAM em particular gozou de uma popularidade considerável durante este período, sendo considerado nessa altura como uma das melhores soluções para a geração de terrenos em tempo real. No entanto, e independentemente da aproximação, os algoritmos concebidos nessa época, continuavam a estar muito dependentes do CPU porque o *hardware* gráfico disponível era ainda muito limitado e, muitas vezes, o ponto de estrangulamento. Consequentemente o objectivo era fornecer à placa gráfica o conjunto "perfeito" de triângulos de modo a reduzir o processamento ao máximo desse componente. Entretanto, a partir de 2001 até aos dias de hoje, tem-se assistido a um progresso imenso no *hardware* gráfico. Mais especificamente, tornou-se possível efectuar o processamento de uma série de fases do *pipeline* gráfico, o que, aliado às facilidades de programação das placas gráficas actuais, veio modificar completamente os métodos utilizados, tendo como principal consequência uma mudança no paradigma dominante. Hoje em dia, o GPU pode processar grandes quantidades de dados em paralelo. Nesta perspectiva, o objectivo passa agora por executar o máximo possível de operações ao nível do GPU deixando o CPU responsável por executar outras tarefas, por exemplo, no domínio dos jogos ficaria responsável pela inteligência artificial, pelas comunicações, entre outros. Esta revolução implicou assim a adopção de novas estratégias e veio tornar os algoritmos mais importantes até aí, completamente inadaptados face a esta nova perspectiva. A característica fundamental destas novas estratégias é não tentarem construir um conjunto óptimo de dados a fornecer, mas procurarem por outro lado obter dados "suficientemente bons" o mais rapidamente possível [9]. Algoritmos como o Geomipmapping (ver **4.4**), o Chunked LOD (ver **4.5**), o GPU Terrain Rendering (ver **4.8**) e o Geoclipmapping (ver **4.9** e **4.10**) representam neste novo paradigma algumas das aproximações mais comuns. Actualmente o expoente máximo desta nova visão, é a segunda versão do algoritmo de Geoclipmapping (ver **4.10**) que consegue passar a maior parte do processamento para o GPU e se enquadra quase por completo nesta nova tendência.

Neste capítulo, pretende-se, por um lado, classificar os diferentes tipos de algoritmos, e por outro descrever alguns dos mais relevantes neste domínio, dando-se especial ênfase às técnicas mais recentes orientadas ao GPU como o Geomipmapping (ver **4.4**), o Chunked LOD (ver **4.5**), o Rendering Very Large, Very Detailed Terrains (ver **4.7**), o GPU Terrain Rendering (ver **4.8**), e o Geoclipmapping (ver **4.9**). Procurou-se também focar, com a descrição do algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**), a integração de conceitos como o *occlusion culling* (ver **3.3.3**) nos algoritmos mais recentes. As técnicas mais antigas também foram discutidas porque embora ultrapassadas face à evolução dos GPU, os conhecimentos aí obtidos não deixam de ser um fundamento sólido para compreender a origem e o porquê de muitos dos conceitos presentes em

algoritmos mais recentes. Neste contexto, e com o intuito de criar uma linha temporal representativa da evolução dos algoritmos de geração de terrenos em tempo real, descrevem-se ainda o algoritmo de ROAM (ver **4.2**) e o algoritmo de Real Time Generation of Continuous Level of Detail (ver **4.3**).

4.1. Classificação

A classificação dos algoritmos de geração de terrenos em tempo real adoptada nesta dissertação segue a proposta por Losasso e Hoppe em [120]. Essa classificação comporta quatro classes: *Irregular Meshes*, *Bin-Tree Hierarchies*, *Bin-Tree Regions* e *Tiled Blocks*. No entanto para acomodar as especificidades do algoritmo de Geoclipmapping (ver **4.9** e **4.10**), considera-se aqui uma nova classe introduzida pela primeira vez em [9]: as *Concentric Regions*. Cada uma das classes tem características muito específicas:

- *Irregular Meshes*. Os algoritmos desta classe utilizam *triangulated irregular networks* que permitem, tal como já foi referido em **2.2.2,** a representação do terreno da maneira mais optimizada possível usando para isso apenas os pontos mais relevantes da fonte de elevação. Em contrapartida esta estrutura implica armazenar as coordenadas de cada vértice ou então dos triângulos pelo que não é a forma mais eficiente de armazenar os dados de um terreno. Alguns destes algoritmos constroem o terreno com triangulações de Delaunay (ver **Figura 2-6**) (por exemplo [33], [29] e [169]) e outros com triangulações arbitrárias (por exemplo [42], [90] e [56]).
- *Bin-Tree Hierarchies*. Os métodos nesta classe de algoritmos usam a divisão recursiva de triângulos rectângulos por meio de estruturas hierárquicas de partição do espaço, nomeadamente *triangle bintrees* (ver **3.2.3.3**) e *quadtrees* (ver **3.2.3.1**). Assim o objectivo é aproximar o terreno com o menor número de triângulos de acordo com os parâmetros de visualização correntes. São exemplo desses algoritmos [110], [112], [151], [17] e [111] e os descritos em **4.2** e **4.3**.
- *Bin-Tree Regions*. Os métodos nesta classe podem ser vistos como abordagens híbridas que tentam adaptar muitos dos conceitos associados aos algoritmos na classe de *bin-tree hierarchies*, referida no ponto anterior, ao modo de funcionamento dos processadores gráficos mais recentes. De um modo geral não preconizam abordagens orientadas ao triângulo e à obtenção de conjuntos perfeitos de triângulos a enviar para o GPU, procurando pelo contrário agrupá-los, pelo que recorrem a conjuntos de triângulos pré-assemblados e residentes em *caches* na memória da placa gráfica. No entanto este *caching* de geometria torna estes algoritmos especialmente sensíveis a problemas de falhas e a *t-junctions* (ver **3.4.4.1**) que são de difícil resolução neste modo de representação. Além disso torna também muito complicado garantir a continuidade temporal pois o *caching* impede o acesso à representação dos dados do qual o Geomorphing (ver **3.4.4.2**) está dependente. Alguns dos métodos nesta classe são, por exemplo, os algoritmos descritos em [27], [28], [108] e [166].
- *Tiled Blocks*. Nesta classe de algoritmos, o terreno é dividido num conjunto de blocos aos quais é atribuído um determinado nível de detalhe que pode variar de bloco para bloco, ou seja, as decisões de nível de detalhe baseiam-se em grupos de triângulos em vez de serem efectuadas triângulo a triângulo.

Consequentemente, o objectivo deste tipo de aproximações é o envio do maior número de triângulos para a placa gráfica, diferenciando-se por isso das abordagens anteriores que tentavam obter o conjunto "perfeito" triângulos. Nesta perspectiva, foram especialmente concebidas tendo em consideração as modernas arquitecturas de GPU com uma grande capacidade de processamento. Em contrapartida os diferentes níveis de detalhe tornaram o problema das falhas especialmente relevante pelo que foi necessário desenvolver um conjunto de técnicas para o colmatar. Alguns dos algoritmos mais relevantes nesta classe são, por exemplo, os descritos em **4.4**, **4.5**, **4.6**, **4.7** e **4.8**, [186], [177] e em [207].

- *Concentric Regions*. Esta classe teve origem no algoritmo de Geoclipmapping descrito em **4.9** e em **4.10**. No entanto, rapidamente surgiram uma série de variações que utilizam o principio base tais como [31], [153] e [197]. Esta aproximação define uma hierarquia de regiões centradas no utilizador composta por um conjunto de anéis concêntricos organizados em níveis de progressivamente maior resolução, em que a resolução aumenta progressivamente do exterior para o centro. Assim, ao passo que os *Tiled Blocks* dividiam o terreno num conjunto de blocos, esta aproximação utiliza uma janela centrada no ponto de vista composta de múltiplos níveis que é incrementalmente actualizada à medida que o utilizador se movimenta no terreno.

4.2. ROAM

O algoritmo de ROAM (*Real Time Optimally Adapting Meshes*) publicado por Mark Duchaineau et al. em [49] foi um dos algoritmos mais importantes nesta área. Inicialmente desenvolvido para simuladores de voo, foi no domínio dos jogos que teve mais relevância sendo inclusivamente implementado numa série de motores gráficos como o *Tread Marks* [116], *Genesis3D* [53], *Crystal Space* [36], *Ogre* [147] entre outros [119]. No ROAM a triangulação utiliza como estrutura de suporte uma *triangle bintree* (ver **4.2.1**) construída a partir de um conjunto de operações de divisão/fusão aplicadas a pares de triângulos que partilham a hipotenusa. Para suportar estas operações são utilizadas duas listas ordenadas por prioridade: uma de triângulos divididos e outra de triângulos fundidos. Estas listas permitem tirar partido da coerência entre *frames*, ou seja torna-se possível em cada *frame* partir da triangulação efectuada para o *frame* anterior. De realçar ainda o princípio base, isto é, a geração do conjunto "perfeito" de triângulos para o ponto de vista corrente, tendo em conta a distância à câmara e as irregularidades do terreno, o que, de um modo geral, corresponde à atribuição de mais triângulos (detalhe) a zonas mais irregulares e/ou próximas do observador, tal como está representado na **Figura 4-1** para uma triangulação típica deste algoritmo.

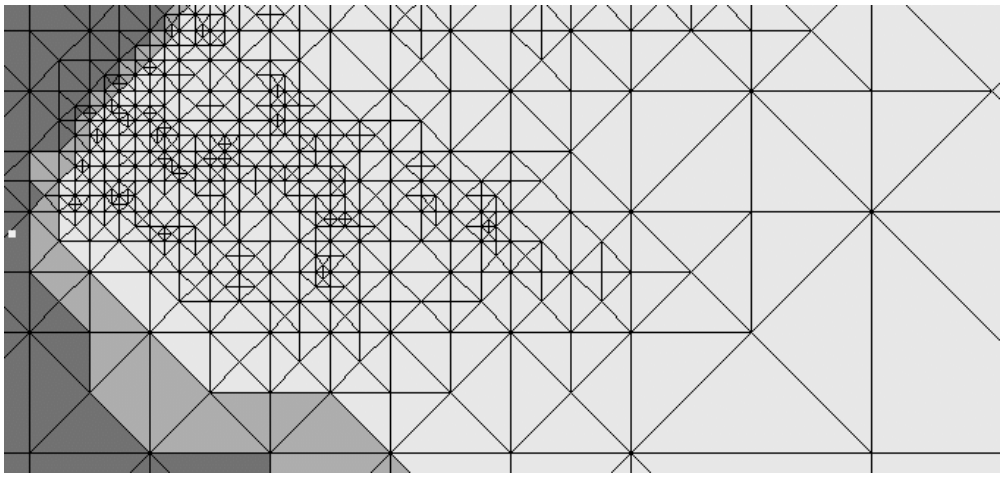


Figura 4-1: Exemplo de uma triangulação no algoritmo de ROAM [49].

Outro factor importante está relacionado com a facilidade com que o ROAM lida com problemas como falhas e *t-junctions* (ver **3.4.4.1**) típicos nestes algoritmos. Aqui esse problema é contemplado pelo próprio processo de triangulação, pelo que se efectua apenas o Geomorphing de forma a garantir a continuidade temporal (ver **3.4.4.2**).

O algoritmo de ROAM em si consiste num conjunto de passos. O primeiro é executado numa fase de pré-processamento e consiste no cálculo de erros geométricos que são depois utilizados para determinar as prioridades dos triângulos em tempo de execução. Quanto à execução do algoritmo propriamente dita, este passa em cada *frame* por quatro fases distintas:

1. Determinação do *view frustum culling* (ver **4.2.4**) que permite logo à partida reduzir o número de triângulos considerados.
2. Actualização das prioridades para os triângulos que podem ser potencialmente divididos/fundidos na fase 3 (ver **4.2.3**).
3. Actualização da triangulação através de operações de divisão e fusão controladas por duas listas de prioridade associadas a cada uma dessas operações (ver **4.2.2**).
4. Actualização das tiras de triângulos afectadas pelas alterações introduzidas no processo de *culling* na fase 1 e pelas divisões/fusões na fase 3.

Muito embora esta abordagem fosse para a altura revolucionária sentiu-se a necessidade de efectuar alterações ao algoritmo original de modo a aumentar o seu desempenho e/ou diminuir a complexidade associada (ver **4.2.6**). De todas as alterações propostas a que teve mais impacto foi a que Seumas McNally introduziu no jogo *Tread Marks*, que permitiu tornar o algoritmo original mais rápido e simples, o que aliado às características já por si inovadoras tornou o ROAM no método de referência pelo menos, até as placas gráficas começarem a assumir definitivamente um papel mais relevante.

4.2.1. Representação do terreno

O factor mais relevante no ROAM está precisamente na estrutura de dados que utiliza para representar o terreno, a *triangle bintree*, já referida em **3.2.3.3**. Nesta, cada um dos

níveis é obtido pela divisão de triângulos rectângulos isósceles (dois lados iguais e um ângulo recto de 90°), divisão que ocorre no vértice situado no ângulo de 90° até ao ponto intermédio oposto situado na hipotenusa, dando origem desta forma a dois novos triângulos (também eles triângulos rectângulos isósceles) que podem ser subdivididos da mesma forma num processo recursivo que permite triangular o *height field* subjacente. A **Figura 4-2** representa os primeiros níveis de subdivisão. Nesta, o nível 0 da árvore (o de menor detalhe) é um simples triângulo rectângulo isósceles denotado por $T = (v_a, v_0, v_1)$. No nível seguinte, os filhos, T_0 e T_1, são obtidos pela divisão do triângulo no nível anterior. Esta divisão é efectuada a partir do vértice v_a até ao vértice intermédio na aresta oposta (v_0, v_1), a hipotenusa. Assim, o filho esquerdo de T é $T_0 = (v_c, v_a, v_0)$ e o filho direito $T_1 = (v_c, v_1, v_a)$. Tal como já foi mencionado em **3.2.3.3**, é necessário que dois triângulos partilhem a hipotenusa para formar o quadrado que representa o terreno, por isso temos de utilizar duas *triangle bintrees*, uma para cada lado da diagonal, para concretizar a triangulação (ver **Figura 3-18**).

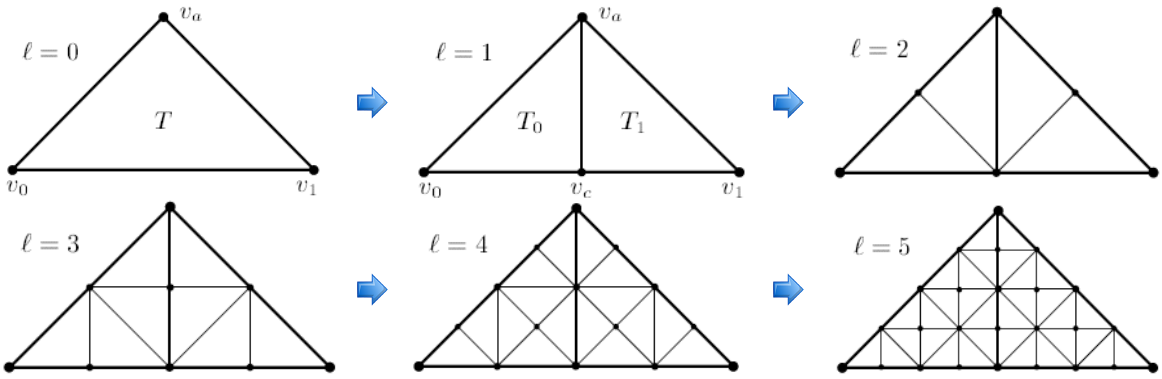


Figura 4-2: Níveis 0 a 5 de uma *triangle bintree* [49].

A triangulação propriamente dita é concretizada por duas operações simples: divisão (*split*) e fusão (*merge*). Mas, antes de descrever cada uma delas, há que definir as relações de vizinhança entre triângulos. Assim, tomando como referência a **Figura 4-3** temos de considerar para cada triângulo, T, três vizinhos: T_B, o vizinho base que partilha com T a aresta (v_0, v_1), T_L, o vizinho esquerdo que partilha com T a aresta (v_a, v_0), e, finalmente, T_R, o vizinho direito que partilha com T a aresta (v_1, v_a) . Usando como ponto de partida estas relações de vizinhança, é possível definir um conjunto de regras a aplicar na triangulação que permitem a divisão de um triângulo sem causar descontinuidades espaciais, nomeadamente falhas e *t-junctions* (ver **3.4.4.1**). Desta forma, temos de garantir que [162][49]:

- O triângulo à esquerda, T_L, e à direita, T_R, de um triângulo T pertence ao mesmo nível l, ou ao nível de maior detalhe seguinte, $l + 1$, na *triangle bintree*.
- O vizinho base está no mesmo nível, l, ou num nível menos detalhado, $l - 1$, que T na *triangle bintree*.

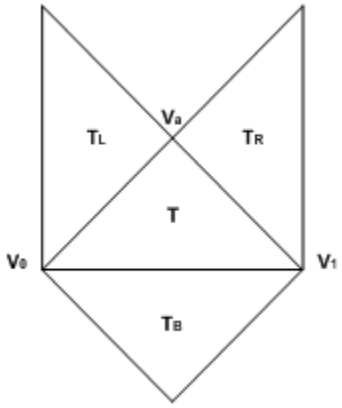


Figura 4-3: Relações de vizinhança entre triângulos [162].

Com as relações de vizinhança estabelecidas estamos agora em condições de abordar as operações que permitem triangular o terreno: a divisão e a fusão. Para isso há que definir no contexto da *triangle bintree* mais um conceito, o de diamante. Assim, diz-se que o par de triângulos (T, T_B) forma um diamante quando pertencem ao mesmo nível l. Esta estrutura é particularmente importante uma vez que é a única à qual podem ser aplicadas as operações de divisão e de fusão, com excepção da situação em que não existe triângulo base, sendo nesse caso considerado apenas o triângulo T correspondente. Na **Figura 4-4** estão representadas as duas operações que servem de suporte ao processo de triangulação. No caso da divisão de um diamante, são obtidos quatro novos triângulos pela subdivisão do triângulo T em T_0 e T_1 e do triângulo T_B em T_{B0} e T_{B1}. Esta divisão cria um novo vértice, v_c, e pode ser repetida recursivamente enquanto existirem dados de elevação no *height field* subjacente que possam ser usados na introdução de novos vértices aumentando-se dessa forma a resolução do terreno. Em relação à fusão o processo é inverso ao descrito para a divisão e só pode ser aplicado a um diamante (T, T_B) quando T e T_B (se T_B existir) estão na triangulação, isto é, se nem T nem T_B tiverem filhos.

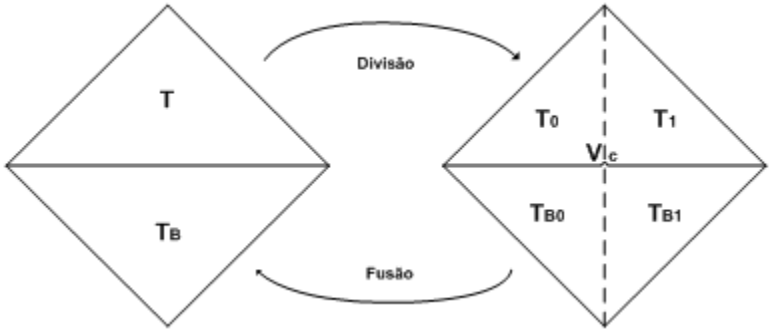


Figura 4-4: As operações de divisão e de fusão numa *triangle bintree* [162].

Quando o par (T, T_B) não forma um diamante, é necessário forçar a divisão de T_B para evitar descontinuidades espaciais. Este processo designa-se de divisão forçada (*forced split*) e visa transformar o par (T, T_B) num diamante, antes de ser efectuada a divisão de T. Para isso é necessário que os triângulos envolvidos formem também diamantes pelo que se desencadeia um processo de divisão recursiva como o ilustrado na **Figura 4-5** que pára apenas quando se encontra um triângulo que forme com o seu vizinho base um diamante

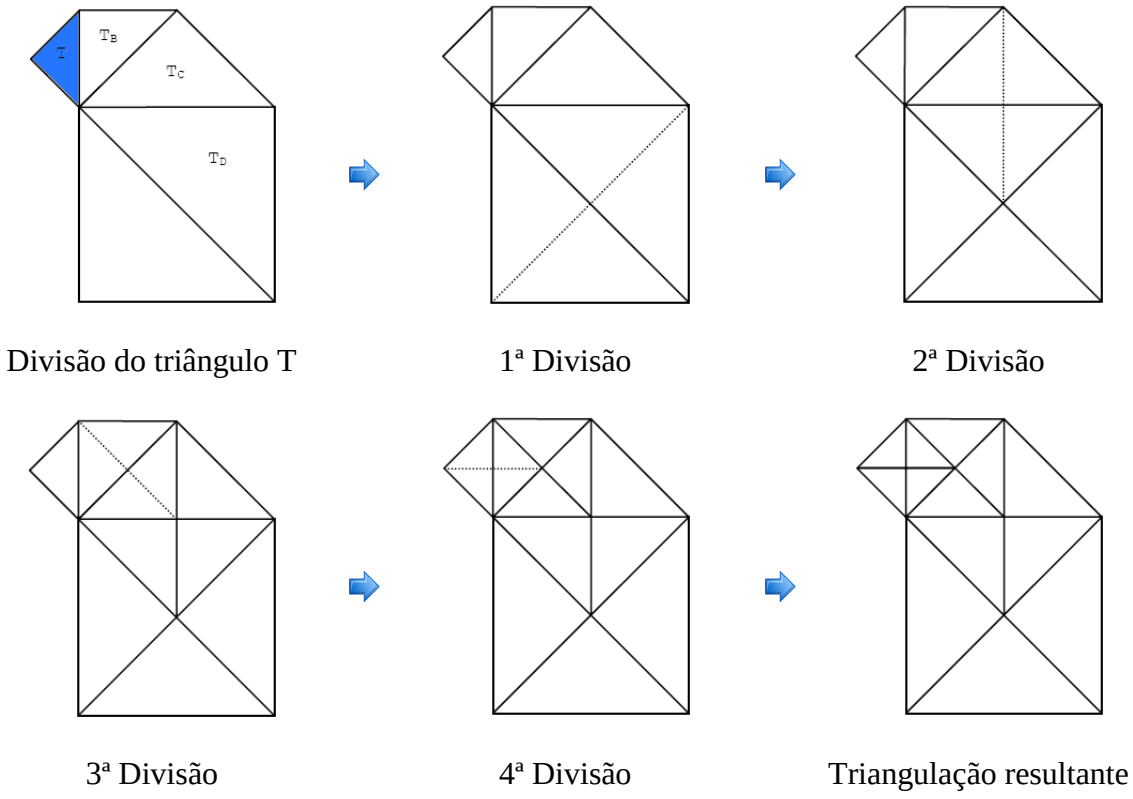


Figura 4-5: Divisão forçada de um triângulo T [162].

A triangulação no algoritmo de ROAM pode ser construída das mais diversas formas. A forma mais fácil é com listas de triângulos (ver **3.1.1**), no entanto também é possível utilizar leques (ver **3.1.2**) e tiras de triângulos (ver **3.1.3**). Esta última aproximação é descrita no artigo original [49] como uma optimização de desempenho do algoritmo. O método empregue é classificado como simples, incremental e sub-óptimo, construindo tiras com um comprimento médio de 4 a 5 triângulos. Desta forma, à medida que os triângulos são divididos, fundidos ou o seu estado de *culling* muda, é alterada a ligação entre as tiras. O eliminar de um triângulo pode por sua vez ter como efeito o encurtar da tira, a sua eliminação ou divisão.

4.2.2. O Algoritmo de Triangulação

O processo de triangulação baseia-se 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 com os filhos de T [162]. Esse erro é obtido numa fase de pré-processamento (ver **4.2.3**) sendo utilizado em cada *frame* na actualização das prioridades. A ideia base é utilizar listas ordenadas em função de um valor de prioridade para controlar todas as operações de divisão/fusão envolvidas no processo de triangulação. Para isso é necessário que as prioridades sejam monotónicas, isto é a prioridade de um determinado triângulo tem de ser superior à dos seus filhos [49]. Assim para construir a triangulação são consideradas duas aproximações: uma utiliza apenas uma lista de prioridades associada à operação de divisão armazenando para isso as prioridades de todos os triângulos na triangulação corrente, a outra considera uma lista adicional com as prioridades associadas à operação de fusão pelo que armazena as prioridades de todos os diamantes passíveis de serem fundidos. Nessa lista, a prioridade de cada diamante é definida como o máximo das prioridades dos triângulos que o constituem, isto é $\max\big(p(T), p(T_B)\big)$, e tem como principal objectivo permitir a cada *frame* partir de uma triangulação anterior e consequentemente tirar partido da coerência entre *frames*. Na **Listagem 4-1** está representado o algoritmo de divisão que corresponde à primeira abordagem e no qual se considera que a cada triângulo T é atribuída uma prioridade monotónica $p(t) \in [0,1]$ armazenada numa lista Q_s, que contêm todos os triângulos de uma determinada triangulação R. Como se pode verificar na listagem apresentada, em cada *frame* constrói-se por completo a triangulação R, dividindo-se para isso cada um dos triângulos na lista ordenada por prioridade até R estar de acordo com o nível de detalhe pretendido, o que, visto de outra perspectiva, permite efectivamente a diminuição do erro associado a triangulação. Sempre que é aplicada a operação de divisão a um triângulo, cada um dos triângulos afectados é retirado da lista e os seus filhos inseridos. Este algoritmo permite construir uma triangulação óptima, R', com uma prioridade máxima inferior à da triangulação base R e, muito embora possa ser utilizado por si só, o algoritmo sugerido por Duchaineau et al. em [49] é o de divisão/fusão, isto por questões de desempenho já que permite tirar proveito da coerência entre *frames*.

```
01 Para cada uma das frames f {
02   Seja R =triangulação base
03   Para cada triângulo T pertencente a R inserir a prioridade de T em Q_s
04     Enquanto R for demasiado pequeno ou impreciso {
05       Identificar o triângulo T de maior prioridade em R
06       Efectuar a divisão forçada de T
07       Actualizar Q_s da seguinte forma {
08         Remover T e outros triângulos divididos de Q_s
09         Adicionar os novos triângulos em R a Q_s
10       }
11   }
12 }
```

Listagem 4-1: Algoritmo de divisão [49].

Considere-se agora a prioridade a variar no tempo $p_f(t) \in [0,1]$ para um conjunto de *frames* $f \in (0,1,\dots)$ e o problema de obter um conjunto de triangulações óptimas $(R_0, R_1, \dots)$. Assumindo que de *frame* para *frame* as alterações no ponto de vista são mínimas podemos dessa forma considerar que as prioridades dos triângulos sofrem também elas poucas alterações. Nessa perspectiva, podemos assumir que a triangulação em duas *frames* consecutivas é muito semelhante, pelo que faz sentido partir de uma triangulação anterior R_{f-1} para construir a triangulação correspondente à *frame* corrente,

R_f. A fila de fusão Q_m é utilizada para este propósito contendo, tal como já foi referido, um conjunto de diamantes ordenados por prioridade. A operação de fusão de um diamante dessa lista é feita sobre o diamante de menor prioridade numa perspectiva de minimizar o erro introduzido por essa fusão, na triangulação. O algoritmo correspondente, o de divisão/fusão, é apresentado em pseudo-código na **Listagem 4-2**. A triangulação produzida por este algoritmo é, tal como no algoritmo de divisão, óptima, sendo obtida por um conjunto de operações de divisão/fusão aplicadas aos triângulos na triangulação base, R_{f-1}. O número total de divisões e de fusões aplicadas a R_{f-1} é proporcional ao número de triângulos de R_f e de R_{f-1} que não são comuns [49], o que no pior caso corresponde à soma dos dois. Para esse caso, o autor recomenda a inicialização de R, Q_s e de Q_m tal como é feito na listagem para $f = 0$.

```
Para cada uma das frames f {
 Se f = 0 então {
   Seja R =triangulação base
   Limpar Q_s e Q_m
   Calcular prioridades para os triângulos e diamantes de R
   Inseri-las em Q_s e Q_m respectivamente
 }
 Senão {
   Seja R = R_{f-1}
   Actualizar as propriedades para os triângulos de Q_s e os diamantes de Q_m
 }
 Enquanto a triangulação R não tiver o tamanho/precisão desejada ou a máxima
 prioridade de divisão for maior que a mínima prioridade de fusão {
   Se a triangulação R for demasiado grande ou precisa {
     Identificar o diamante (T,T_B) de menor prioridade em Q_m
     Fundir (T,T_B)
     Actualizar Q_s e Q_m da seguinte forma {
       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 Q_s e Q_m da seguinte forma {
       Remover T e todos os triângulos divididos de Q_s
       Inserir novos triângulos de R em Q_s
       Remover de Q_m todos os diamantes cujos filhos foram divididos
       Inserir em Q_m todos os novos diamantes
     }
   }
 }
 R_f = R
}
```

Listagem 4-2: Algoritmo de divisão/fusão [49][162].

4.2.3. Prioridades

O ROAM realiza numa fase de pré-processamento o cálculo dos erros geométricos associados a cada triângulo, erros que são depois utilizados para obter o valor de cada uma das prioridades em tempo de execução. O cálculo desses erros é efectuado

bottom-up na *triangle bintree* a partir do triângulo mais refinado (no qual o erro é nulo) até à maior generalização possível (na qual o erro é máximo)[162]. Para medir esse erro é utilizada uma hierarquia de *bounding volumes* (ver **3.2.1**) que são denominados em [49] de *wedgies*. A *wedgie* é uma abstracção utilizada pelo ROAM para medir o erro geométrico associado a cada um dos triângulos que constituem o terreno e tem a forma de um prisma triangular. Como sabemos as coordenadas de cada um dos três vértices que formam o triângulo associado à *wedgie* a única informação que falta obter é a altura ou a distância entre a face superior e inferior da *wedgie* e o triângulo. Neste caso simplifica-se e assume-se um valor de altura igual para cada uma delas. Esse valor é obtido como a diferença de altura entre as coordenadas de elevação máxima e as coordenadas de elevação mínima. No ROAM considera-se que os triângulos que pertencem ao nível de detalhe mais elevado têm um erro geométrico igual a zero, pois representam o maior nível de detalhe que se consegue obter a partir do *height field* subjacente. Isto significa que as *wedgies* associadas a esses triângulos têm uma altura igual a zero, pelo que são o ponto de partida para a avaliação do erro geométrico. Para um triângulo que não pertence ao nível de detalhe mais preciso, o erro geométrico resultante da utilização desse triângulo em vez dos seus filhos que cobrem a mesma área será certamente superior a zero. Nesse caso, o método utilizado na avaliação da altura da *wedgie* é o seguinte (ver **Figura 4-6**): primeiro obtém-se a altura no ponto intermédio do segmento entre o vértice esquerdo e direito do triângulo (o a e o b respectivamente), pois no próximo nível de detalhe directamente acima ou abaixo desse ponto vai ser adicionado um novo vértice; essa altura é por sua vez obtida efectuando a média das alturas dos vértices adjacentes; depois calcula-se a diferença (o h) entre a altura do novo vértice (o d) no triângulo mais refinado e a altura do vértice calculado (o m); finalmente, adiciona-se o máximo dos valores calculados para cada uma das *wedgies* definidas para os triângulos filhos de modo a assegurar que toda a geometria coberta pelo triângulo está incluída na nova *wedgie*.

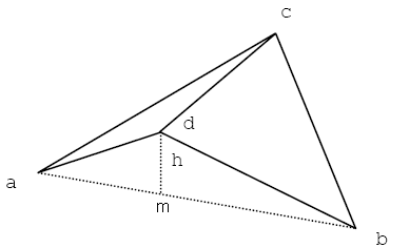


Figura 4-6: Erro introduzido por se utilizar abc em vez de acd e dbc [162].

Dito de outra forma, uma *wedgie* pode ser vista como um prisma triangular com uma altura definida por um segmento, o *thickness segment* que vai de $y - e_T$ até $y + e_T$ no qual o y é a altura do triângulo em cada ponto e e_T a *wedgie thickness*. Esta é definida como o máximo das *wedgie thicknesses* dos seus filhos e pode ser calculada por intermédio da **Equação 4-1**. Este cálculo é efectuado *bottom-up* tendo início no nível de maior detalhe para o qual se assume que $e_T = 0$, pelo que serve desta forma de ponto de partida para o cálculo efectuado.

$e_T = \max\{e_{T0}, e_{T1}\} + \lvert y(v_c) - y_T(v_c)\rvert$	$\{e_{T0}, e_{T1}\}$: *Wedgie thickness* dos filhos.
	$y(v_c)$: Altura no triângulo de maior detalhe.
	$y_T(v_c)$: Altura calculada como $\big(y(v_0) + y(v_1)\big)/2$ para o triângulo de menor detalhe.

Equação 4-1: Cálculo da *wedgie thickness*.

Em tempo de execução cada *wedgie* é projectada no espaço do ecrã tal como está representado na **Figura 4-7**. É a partir do *thickness segment* de maior dimensão de cada uma das *wedgies* que é então obtida a prioridade do triângulo respectivo.

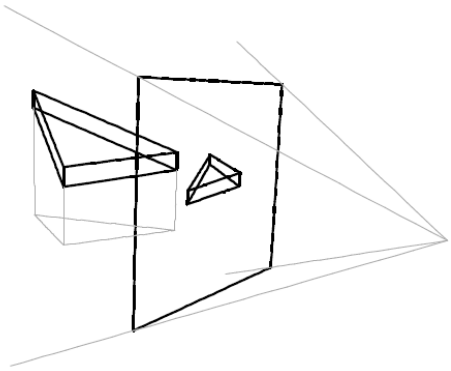


Figura 4-7: Projecção de uma wedgie [49].

4.2.4. Culling

O processo de *culling* no algoritmo de ROAM centra-se no teste de cada um dos triângulos na *triangle bintree* contra o *frustum* da câmara (ver **3.3.2**), processo que tira partido da estrutura da árvore. Mais precisamente a cada triângulo são atribuídas seis IN *flags*, uma para cada um dos seis planos que compõem o *frustum* (ver **Figura 3-21**) e também um estado geral de OUT, ALL-IN ou DONT-KNOW. Podemos definir cada um deles da seguinte forma: a *flag* IN de um plano é atribuída se a *wedgie* (ver **4.2.2**) que representa o triângulo estiver totalmente dentro desse plano; o estado OUT é dado quando pelo menos uma das *flags* IN não está activa, pelo contrário o estado ALL-IN é atribuído no caso oposto, isto é quando todas as *flags* IN estão activas; finalmente o estado, DONT-KNOW é atribuído para os casos restantes (isto é, quando não é nem OUT nem ALL-IN). O desempenho deste teste é garantido pelo processo recursivo efectuado na *triangle bintree* que pára quando a um triângulo T é atribuído o estado ALL-IN ou OUT numa determinada *frame* T_{f-1} e na *frame* corrente, T_f, o estado é o mesmo. Nesse caso, a sub-árvore não necessita de ser actualizada e a recursão pára. Pode-se assim assumir com segurança que se um triângulo for visível (ALL-IN) os seus filhos também serão necessariamente visíveis e o mesmo para o caso oposto (OUT).

4.2.5. Optimizações

Duchaineau et al. descrevem ainda em [49] um conjunto de optimizações ao algoritmo original. A primeira está relacionada com a construção incremental de tiras de triângulos (ver **3.1.3**) já descrita em **4.2.1**. As outras duas são, respectivamente, o cálculo parcial de prioridades e a optimização progressiva da triangulação. O cálculo parcial de prioridades foi uma optimização motivada pelo facto de as prioridades mudarem muito pouco de *frame* para *frame*, e também pelo custo de as recalcular em cada uma das *frames* para todos os triângulos. O objectivo é calcular as prioridades apenas quando estas podem potencialmente afectar a decisão de efectuar uma operação de divisão/fusão [49]. Para isso são estabelecidos limites de velocidade para a câmara, o que permite o cálculo de limites para as prioridades dos triângulos em função do tempo. Paralelamente temos ainda de ter em consideração a *crossover priority* (definida como a máxima prioridade na lista de divisão após efectuadas todas as operações de divisão/fusão) que varia muito pouco de *frame* para *frame* (segundo [49] cerca de 1%). Dessa forma, o cálculo da prioridade de um triângulo pode ser adiado até o seu limite de prioridade ser igual ou superior à *crossover priority* definida anteriormente. Para isso, é mantida uma lista de espera para cada uma das próximas doze *frames* que armazena os triângulos que têm de ter a sua prioridade recalculada. Se o tempo permitir triângulos adicionais podem ser recalculados em listas subsequentes. Finalmente, após ser efectuado o cálculo o triângulo é colocado na lista de espera mais afastada.

A optimização progressiva tem como objectivo garantir uma *frame rate* constante controlando para isso o processo de triangulação. Mais especificamente estabelece-se que a triangulação deve parar quando o tempo execução atribuído a cada uma das *frames* estiver por expirar. Como o algoritmo de ROAM executa um processo incremental de triangulação baseado num conjunto de operações de divisão/fusão que ocorrem em ordem decrescente de importância a triangulação efectuada até ao momento de paragem, continua a ser óptima no sentido em que representa a melhor triangulação possível dentro do tempo disponível. A grande maioria das operações no ROAM pode aderir a este conceito. A única excepção é o *frustum culling* (ver **4.2.4**), mas como este consome uma fracção muito pequena do tempo em cada *frame* [49], não tem um impacto muito significativo.

4.2.6. Variações do Algoritmo

O algoritmo de ROAM [49] foi modificado por uma série de investigadores e de programadores de jogos. Uma das implementações que teve maior sucesso foi a de Seumas McNally no jogo TreadMarks [116]. Para este jogo, McNally fez um conjunto de alterações ao algoritmo de ROAM original que foram documentadas em [165] e [198]. Nesta implementação não é utilizado o conceito de listas de prioridade pelo que foram efectuadas um conjunto de alterações, nomeadamente [165]:

- A informação sobre os triângulos enviados para *rendering* não é armazenada.
- É utilizada uma métrica de erro mais simples.
- Não se tira partido da coerência entre *frames*.

Em relação ao primeiro ponto, a única informação armazenada por cada triângulo é um conjunto de ligações a cada um dos triângulos filhos e vizinhos, tal como está representado na **Figura 4-8**.

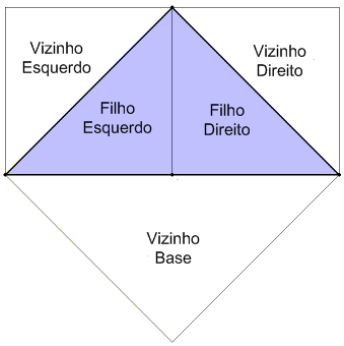


Figura 4-8: Filhos e vizinhos de um triângulo numa *triangle bintree* [198].

Como se pode verificar, são armazenadas apenas cinco ligações, duas para os triângulos filhos (esquerdo e direito) e três para os triângulos vizinhos (base, esquerdo e direito). No que diz respeito ao segundo ponto, McNally utilizou uma métrica de erro que denominou de variância. Esta não é mais do que a diferença de altura entre um vértice calculado no meio hipotenusa do triângulo (cálculo efectuado como a média da altura dos dois pontos adjacentes) e a altura real desse vértice no *height field* subjacente. A fórmula de cálculo da variância está representada na **Equação 4-2**

$variancia = \left\lvert c_y - \frac{a_y + b_y}{2} \right\rvert$	c_y: Valor real da altura no vértice.
	a_y: Valor da altura num dos vértices adjacentes.
	b_y: Valor da altura no outro vértice adjacente.

Equação 4-2: Cálculo da variância.

Finalmente em relação ao último ponto, esta implementação não tira partido da coerência entre *frames*, pelo que é normalmente designada de *split only* ROAM, já que em cada *frame* a triangulação é completamente refeita.

Para além desta implementação do algoritmo de ROAM, é necessário referir ainda outras implementações de especial relevância, nomeadamente:

- *RUSTiC (ROAM Using Surface Triangle Clusters)*: Descrito em [166] é uma extensão ao algoritmo original desenvolvida numa perspectiva de aumentar o seu desempenho. A diferença entre o ROAM e o RUSTIC está no facto do RUSTIC agrupar conjuntos de triângulos, formando o que o autor designa de um *cluster*, isto ao contrário do ROAM que trabalha com triângulos individuais.
- DEXTER (*Dynamic EXTEnsion of Resolution*): Este algoritmo descrito em [86] e [87] é uma concretização do ROAM que suporta terreno dinâmico nomeadamente deformações em tempo real. Dito de outra forma, não assume uma geometria estática, pelo que esta pode ser modificada em tempo real, por exemplo, na representação das marcas deixadas pelos pneus de veículos todo-o-terreno.

- *Terrain Rendering at High Levels of Detail*. Em [17] é descrita uma nova métrica de erro que tenta melhorar o desempenho do ROAM para níveis elevados de resolução. Segundo [17], o ROAM não funciona bem nessas situações devido ao cálculo prematuro das prioridades das *wedgies*. Esta nova métrica não se baseia, ao contrário do ROAM, no cálculo do erro geométrico no espaço do ecrã, cálculo do qual resulta um escalar a 1D, mas sim na utilização do erro geométrico a 3D. Assim, em vez de se ordenar os triângulos de acordo com um valor de prioridade unidimensional, constrói-se uma hierarquia de isosuperficies que contêm todos os vértices dentro de um determinado limite de erro [119]. Para simplificar utiliza-se uma esfera, que permite a representação das *wedgies* num espaço tridimensional. Sempre que a câmara entra no volume definido pela esfera, o triângulo é dividido, e, sempre que esta sai da esfera o triângulo é fundido. Para optimizar este processo é definida uma hierarquia de esferas o que não implica desta forma o teste a todas elas em tempo de execução. Paralelamente, as esferas são também agrupadas em *clusters*, isto é, são criadas esferas sem qualquer correspondência com os triângulos utilizados com o único propósito de agrupar conjuntos de outras esferas na hierarquia. Estes *clusters* têm como objectivo aumentar ainda mais a capacidade deste algoritmo de lidar com terrenos de grandes dimensões.
- *Continuous Level of Detail in Real-Time Terrain Rendering*: Em [148] é apresentada uma variação do algoritmo de ROAM cuja principal diferença em relação ao algoritmo original está na ausência de listas de divisão/fusão, num novo processo de fusão e numa nova métrica de erro que se baseia no erro geométrico e na distância. A operação de fusão proposta permite juntar filhos de qualquer triângulo mesmo que estes tenham sido divididos, isto é, pode ser aplicada a triângulos sem ser necessário existir um diamante como no algoritmo original (ver **4.2.1**).

4.3. Real-Time Generation of Continuous Levels of Detail

Este algoritmo descrito em [172] por Stefan Röttger utiliza uma aproximação *top-down* na triangulação do *height field* de acordo com uma métrica que tem em consideração dois factores principais: a distância à câmara e as irregularidades do terreno. Esta aproximação permite que em cada *frame* seja necessário visitar apenas uma fracção do *height field*, tendo consequentemente ganhos substanciais no desempenho, mesmo em *height fields* de grandes dimensões. Por outro lado, é de especial relevância o facto de ter em consideração a estrutura do terreno, o que lhe permite no decorrer da triangulação atribuir mais detalhe, por exemplo, a zonas montanhosas, e menos detalhe a vastas zonas planas. Outra das suas características é ainda o suporte para Geomorphing (ver **3.4.4.2**) que lhe permite tornar as transições entre os diferentes níveis de detalhe quase imperceptíveis, eliminando assim o efeito de *popping* (ver **3.4.4.2**) com a vantagem adicional de esse processo estar integrado de uma forma natural no algoritmo. Como estrutura utiliza uma *quadtree* (ver **4.3.1**) que serve de suporte a toda a triangulação efectuada bem como ao *culling* (ver **3.3**) de cada um dos blocos de terreno não visíveis.

4.3.1. Representação do terreno

A estrutura de dados utilizada é uma *quadtree* sobre a forma de uma matriz booleana com o mesmo tamanho que o *height field* subjacente, que serve de suporte ao processo de triangulação. Nesta, cada entrada corresponde a um valor de elevação no *height field* subjacente, como aliás se pode verificar na **Figura 4-9** para um *height field* de 9×9, onde cada um dos caracteres representados é o nó central de uma possível subdivisão na *quadtree*. Esta subdivisão depende apenas do valor em cada um dos nós, que neste caso pode ser 0, 1 ou ?. Estes caracteres representam, respectivamente, se o centro está activo (1), inactivo (0), ou se não foi considerado (?). O centro estar activo significa que para esse nó da *quadtree* existe mais uma subdivisão, ou seja, mais um nível de detalhe, na medida em que são utilizados mais valores de elevação para representar essa porção do terreno. Por outro lado, se não existir mais nenhuma subdivisão, diz-se que o centro não está activo. Restam assim todos os outros pontos que são representados na figura com o carácter ?. Estes representam pontos não considerados no processo de subdivisão.

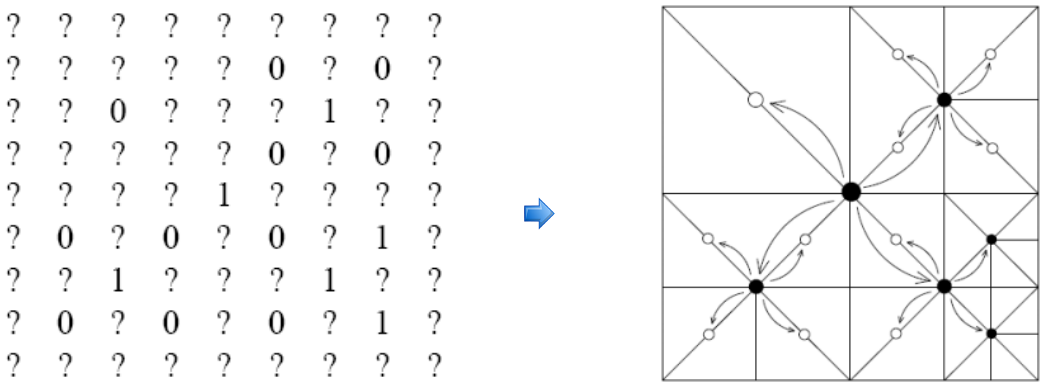


Figura 4-9: Matriz booleana da *quadtree* e a triangulação correspondente [172].

A triangulação, por seu lado, é construída com leques de triângulos em parte porque estes permitem, tal como já foi referido em **3.1.2,** a correcção de falhas causadas por diferentes níveis de detalhe em blocos adjacentes de uma forma simples, que passa por ignorar um determinado vértice na construção do leque. Na **Figura 4-10** é possível verificar precisamente essa situação. Neste caso, os vértices marcados com uma cruz são ignorados na construção do leque correspondente para que um bloco vizinho de menos detalhe possa "encaixar" sem que surjam falhas no terreno. Para isso, é necessário verificar o valor do vértice central correspondente no bloco vizinho, que nesse caso teria de ser 0. É de referir, no entanto, que, para que esta técnica funcione, é necessário que a diferença entre os níveis de detalhe adjacentes nunca seja superior a um, o que é garantido pela métrica de erro utilizada pelo algoritmo na construção da *quadtree* (ver **4.3.2**).

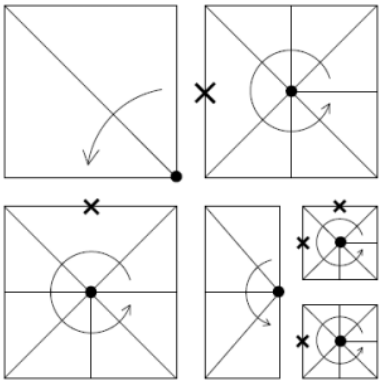


Figura 4-10: Construção dos leques de triângulos da **Figura 4-9** [172].

4.3.2. Construção da Quadtree.

A construção da matriz *booleana* que representa a *quadtree* é feita recursivamente pela avaliação em cada um dos seus nós de um critério de subdivisão. Sempre que o resultado da avaliação do critério para um determinado nó indicar uma subdivisão é armazenado 1 no nó correspondente e avalia-se cada um dos seus quatro sub-nós. Isto, recursivamente, até se chegar ao máximo detalhe possível ou até o critério indicar que a subdivisão não deve continuar, armazenando-se nesse caso um zero na posição correspondente. Este processo permite a construção da matriz *booleana*. Todas as outras subdivisões possíveis para esse nó, não consideradas pelo critério, são os pontos de interrogação visíveis na **Figura 4-9**. Por outro, lado o critério referido pelo qual a construção da matriz se rege tem em conta dois factores: a resolução deve diminuir à medida que a distância à câmara aumenta e deve aumentar para regiões irregulares do terreno. Para satisfazer o primeiro factor é utilizada a **Equação 4-3** que tem como principal objectivo controlar a quantidade total de detalhe visível em todo o terreno. Os principais componentes desta equação estão representados na **Figura 4-11**, nomeadamente a distância à câmara, medida a partir do centro de cada um dos leques e o comprimento da aresta no bloco correspondente. Faz ainda parte da equação a constante C, cujo valor recomendado pelo autor é 8, que permite controlar o processo de subdivisão, pelo que é necessário ter em consideração que aumentar o seu valor significa aumentar o número total de vértices por *frame* quadraticamente [172]. Ainda nesta equação é de referir que no cálculo do valor de l o autor utiliza a norma $L1$ (ver **Equação 3-2**).

$\frac{l}{d} < C$	l: Distância medida a partir do centro do nó corrente até à câmara.
	d: Comprimento da aresta do bloco.
	C: Constante que indica a resolução global mínima.

Equação 4-3: Critério de resolução global.

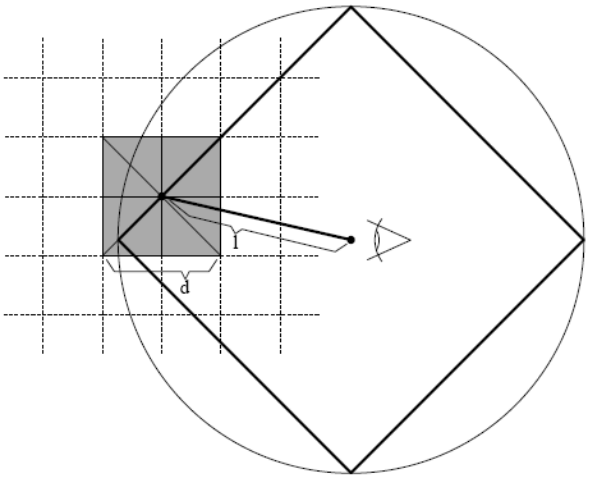


Figura 4-11: Critério de resolução global: distância vs tamanho das células [172].

Para que seja possível ter em conta o segundo factor, isto é a que a resolução deve aumentar para regiões irregulares do terreno, é necessário avaliar o erro introduzido numa região do terreno com uma mudança de nível de detalhe. Assim, para começar é preciso ter em consideração que a diminuição de detalhe implica a introdução de erros em exactamente cinco pontos: no centro do nó e nos quatro pontos intermédios em cada uma das suas arestas. Este erro é calculado como o máximo dos valores absolutos da diferença entre os valores de elevação (os dh_i na **Figura 4-12**) entre dois níveis de detalhe. As diferenças de elevação por sua vez são calculadas em cada uma das arestas e também nas diagonais, totalizando seis diferenças, uma vez que para o nó central existe uma contribuição de duas diagonais, ou seja, dois pares de vértices adjacentes (o dh_5 e o dh_6 na figura). No cálculo dessa diferença, em cada aresta/diagonal temos de considerar duas elevações: uma obtida do vértice no nível de detalhe mais elevado, e a outra calculada como a média de elevações entre os vértices adjacentes no nível de detalhe mais baixo.

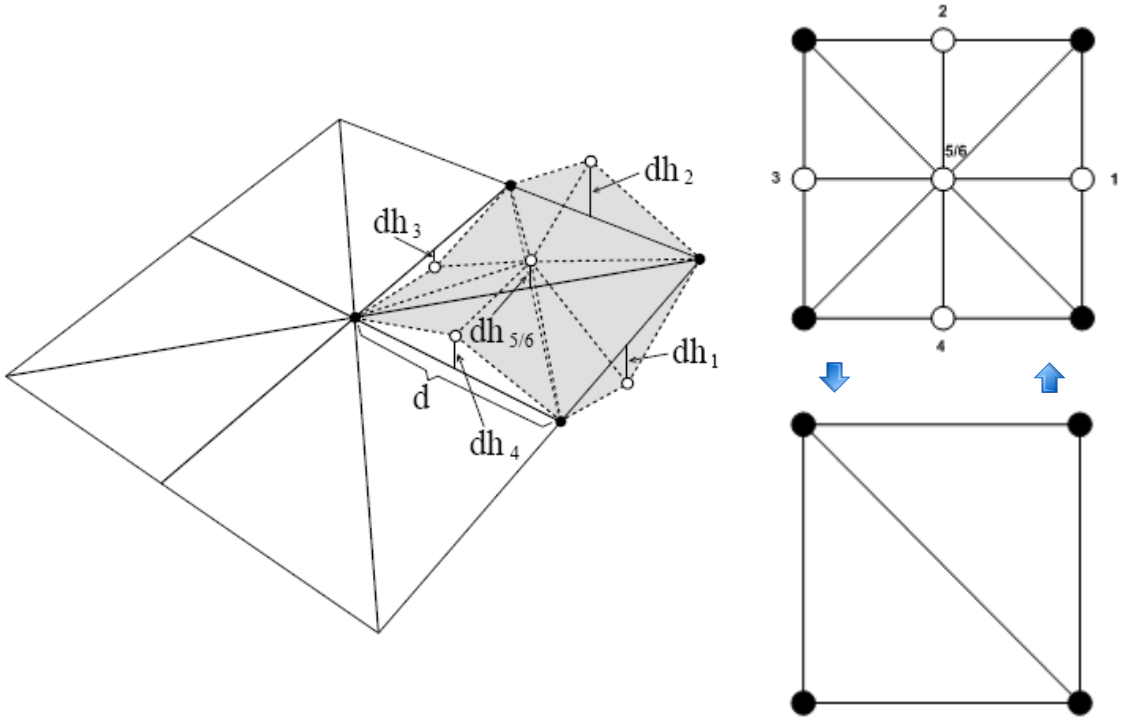


Figura 4-12: Medindo o erro resultante de um diminuição de detalhe [172].

Como o cálculo destes erros está apenas dependente da estrutura do terreno é pré-processado através da **Equação 4-4** sendo o resultado obtido denominado de $d2$.

$d2 = \frac{1}{d} \cdot \max_{i=1\ldots6} \lvert dh_i \rvert$	dh_i: Diferença do valor de elevação em i.
	d: Comprimento da aresta do bloco.

Equação 4-4: Cálculo do erro na mudança de detalhe.

A partir destes dois factores define-se então uma variável de decisão f que é calculada pela **Equação 4-5** que constitui a métrica de erro deste algoritmo, permitindo o resultado obtido decidir a criação ou não de mais uma subdivisão num determinado nó da *quadtree*.

$f = \frac{l}{d \cdot C \cdot \max(c \cdot d2, 1)}$	l: Distância medida a partir do centro do nó corrente até à câmara.
	d: Comprimento da aresta do bloco.
	C: Constante que indica a resolução global mínima.
Subdividir se $f < 1$	c: Constante que indica a resolução global desejada.
	$d2$: Calculado na **Equação 4-4**.

Equação 4-5: Condição de subdivisão do nó.

4.3.3. Culling

O *culling* é implementado utilizando como suporte a *quadtree*, mais especificamente pela intersecção do volume de visualização, o *frustum*, com cada um dos blocos tal com está descrito em **3.3.2** e representado na **Figura 4-13**. Para isso, para cada bloco de detalhe é calculada uma *bounding box* que é verificada contra o *frustum* em duas fases distintas: durante o processo de triangulação e no *rendering* propriamente dito. Durante o processo de triangulação permite que não seja necessário avaliar a métrica de erro em cada uma das subdivisões possíveis. No *rendering* permite reduzir o número de triângulos enviados para processamento, ignorando os blocos que não intersectam o *frustum*.

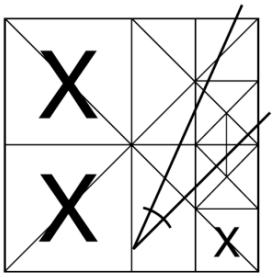


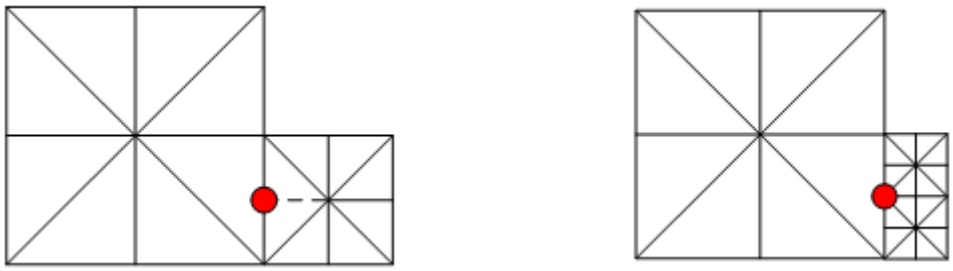
Figura 4-13: *Frustum culling* num bloco de terreno [165].

4.3.4. Rendering

Uma vez construída a *quadtree*, tal como foi descrito em **4.3.2** o *rendering* do *height field* subjacente é feito percorrendo-o recursivamente com base nos diferentes valores de 0 e 1, atribuídos a cada um dos vértices centrais, na matriz *booleana* que o representa. Quando é atingido um nó folha, ou seja, quando não existem mais subdivisões, é desenhado um leque de triângulos que representa a porção do terreno corrente.

4.3.5. Falhas

Tal como foi referido, para que seja possível evitar falhas entre blocos adjacentes com diferentes níveis de detalhe é necessário garantir que a diferença de detalhe entre eles é menor ou igual a um, já que, tal como está exemplificado na **Figura 4-14**, para diferenças maiores não é possível corrigir as falhas sem que se perca a estrutura dos leques. Assim, assumindo uma diferença máxima de um nível de detalhe nessas situações basta ignorar um vértice para que seja possível encaixar os dois blocos adjacentes sem causar falhas visíveis no terreno.



Diferença de detalhe =1 Diferença de detalhe =2

Figura 4-14: Resolução de falhas em blocos adjacentes de diferente detalhe [162].

Para garantir então que a diferença entre os níveis de detalhe nunca é superior a um, Röttger redefiniu o cálculo de $d2$, executando para isso numa fase de pré-processamento uma propagação dos valores de $d2$ pelos sucessivos níveis da *quadtree*, começando pelos níveis de maior resolução até à raiz da mesma. Para isso, em cada nível de detalhe é calculada uma constante K (**Equação 4-6**) que influencia o cálculo do $d2$. Mais precisamente, o $d2$ correspondente passa a ser obtido como o máximo entre o valor local e K vezes os valores calculados previamente para os nós adjacentes do nível imediatamente abaixo. Esta alteração permite garantir que a diferença entre níveis de detalhe adjacentes nunca é superior a um.

$K = \frac{C}{2 \cdot (C-1)}$	C : Constante que indica a resolução global mínima.

Equação 4-6: Cálculo do factor de propagação.

4.3.6. Geomorphing

Uma das principais características deste algoritmo é integrar de uma forma natural o Geomorphing (ver **3.4.4.2**), permitindo que o efeito de *popping* (ver **3.4.4.2**), muito comum em algoritmos de nível de detalhe, possa ser resolvido, tornando a mudança de detalhe quase imperceptível ao utilizador. Para suportar esta característica a matriz descrita em **4.3.1** tem de passar a armazenar *blending factors* ao invés de valores booleanos. Estes são calculados pela **Equação 4-7** sendo utilizados para efectuar a transição entre os dois níveis de detalhe. Esta transição pode ter de ser feita para um máximo de cinco vértices por bloco e consiste em efectuar uma interpolação linear com o factor *b* calculado pela **Equação 4-7** entre a elevação do nível de detalhe mais baixo (calculada pela média de elevação dos vértices adjacentes) e a elevação do nível mais alto obtida directamente do *height field*. Paralelamente, foram tomadas precauções especiais para que as falhas não surjam durante o Geomorphing por dois blocos adjacentes terem *blending factors* diferentes. Para o evitar, é utilizado o valor mínimo dos *blending factors*

em blocos adjacentes. Neste caso, um valor de 0 indica que não existe subdivisões, ao passo que outros valores representam directamente *blending factors*.

$b = \text{clamp}(2 \cdot (1 - \text{f}), 0{,}1)$	clamp: Constrange o resultado ao intervalo [0,1].
	f: Condição de subdivisão do nó calculada pela **Equação 4-5**.

Equação 4-7: *Blending function*.

4.4. Geomipmapping

Este algoritmo descrito por Willem H. de Boer em [39] e desenvolvido no âmbito do projecto E-mersion tem como principais características a sua simplicidade e o facto de estar orientado ao *hardware* mais recente ao contrário de outros algoritmos (como os descritos em **4.2** e **4.3**), que foram concebidos antes de os GPU assumirem um papel tão determinante como o que têm hoje ao nível da computação gráfica. O objectivo é então o envio do maior número de triângulos visíveis por *frame* para o GPU, isto é, não se pretende obter um conjunto perfeito de triângulos. O foco agora é num conjunto de blocos fruto da subdivisão do terreno via uma *quadtree* (ver **3.2.3.1**) para os quais se aplica para efeitos de *culling*, o *view frustum culling* (ver **3.3.2**), seleccionando-se posteriormente para cada um deles o nível de detalhe adequado ao ponto de vista corrente. Este algoritmo goza de uma grande popularidade sobretudo devido à sua simplicidade tendo sido utilizado, com algumas modificações, em alguns jogos como, por exemplo, o *Black & White* 1 e 2 [113]. Estes têm como principal característica vastos terrenos o que implica necessariamente uma gestão de nível de detalhe, já que fica grande parte do terreno visível no ecrã.

4.4.1. Representação do terreno

O Geomipmapping utiliza uma estrutura que assenta numa grelha regular em que o número de vértices em cada linha e em cada coluna tem de ser da forma $2^n + 1$ em que $n \in [1, \rightarrow)$, ou seja, são 2^n células com quatro vértices e dois triângulos por célula, tal como está representado na **Figura 4-15** para uma grelha com $n = 3$ (9×9 vértices por bloco). A triangulação descrita pode por sua vez ser efectuada com qualquer uma das primitivas descritas em **3.1** nomeadamente listas de triângulos (ver **3.1.1**), tiras de triângulos (ver **3.1.3**) e leques de triângulos (ver **3.1.2**). Os leques de triângulos utilizados, por exemplo, em [165] facilitam o processo de tratamento de falhas (ver **3.4.4.1**), mas são a aproximação menos eficiente de todas, não sendo por isso a melhor opção.

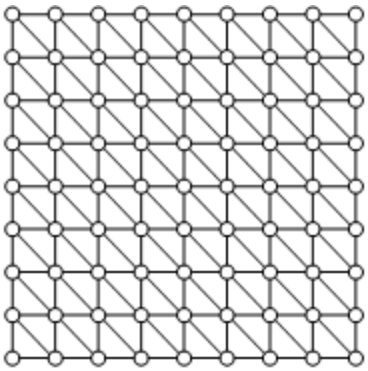
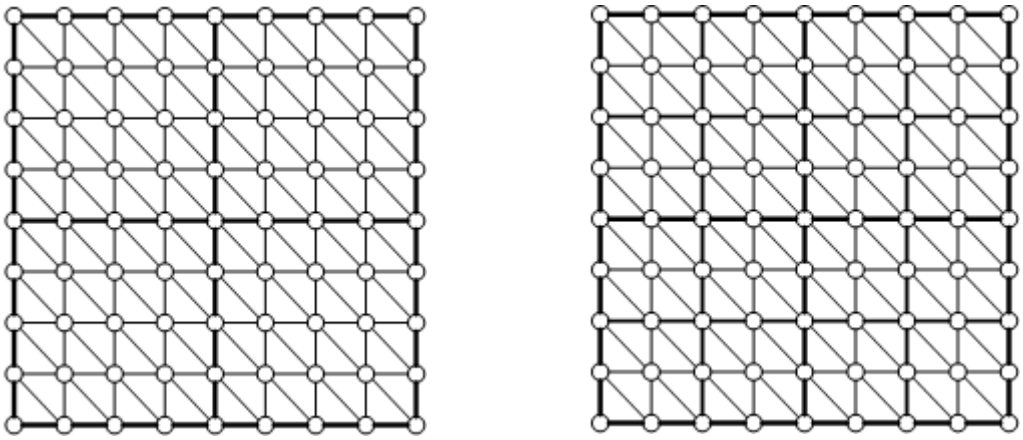


Figura 4-15: Representação de um terreno 9×9.

Utilizando a *quadtree* como estrutura de particionamento espacial o terreno é subdividido num conjunto de blocos que têm de ser da forma $2^n + 1$ em que $n \in [1, \rightarrow)$. Por exemplo, na **Figura 4-16**, são consideradas duas hipóteses de subdivisão para um terreno 9×9, uma com $n = 2$ (5×5 vértices por bloco) e outra com $n = 1$ (3×3 vértices por bloco). Esta subdivisão é essencial para mais tarde ser possível efectuar o *culling* de todos os blocos visíveis de modo a que o *rendering* se processe apenas para os que intersectam o *frustum* e também para efectuar separadamente a variação do detalhe em cada um dos blocos.



a) $n = 2$ b) $n = 1$

Figura 4-16: Duas opções na subdivisão do terreno em blocos**.**

4.4.2. Conceito de Geomipmapping

O *culling* por si só não é suficiente para reduzir o número de triângulos enviados para processamento especialmente em terrenos de grande dimensão. Muito embora permita uma redução substancial, ainda continua a existir um grande peso associado. Se considerarmos que os blocos que se encontram mais distantes da câmara não necessitam de serem representados com o mesmo detalhe, já que não são visíveis na totalidade

devido ao efeito de perspectiva, é possível ir um pouco mais além. Nesses casos não é necessário enviar todos os triângulos que constituem o bloco, basta enviar uma versão menos detalhada com menos triângulos. Para se atingir esse objectivo é necessário aplicar uma técnica de nível de detalhe (ver **3.4**) que permita a diminuição do número de vértices para cada um dos blocos que constitui o terreno. Para implementar níveis de detalhe De Bóer baseou-se num conceito que já existia na computação gráfica: o *mipmapping* [213]. O *mipmapping* não é uma ideia nova, muito pelo contrário, há muito que é aplicado na representação de texturas com o intuito de simular a noção de perspectiva. "*Mip*" de *mipmapping* vem do latim *multum in parvo*, que significa "muitas coisas num pequeno espaço", um bom nome para um processo em que a textura original é filtrada progressivamente no sentido de obter uma cadeia de *mipmaps* de menor resolução (ver **Figura 4-17**). Cada *mipmap* é assim duas vezes mais pequeno que o anterior (i.e. se a textura original for de 64×64 então os *mipmaps* dessa textura serão 32×32, 16×16, 8×8, 4×4, 2×2 e 1×1 respectivamente). Estes *mipmaps* são seleccionados em tempo de *rendering* de acordo com a distância a que se encontram da câmara. Para um objecto a pouca distância a versão seleccionada será muito provavelmente a original, para os outros casos à medida que aumenta a distância vão sendo seleccionadas versões menos detalhadas até ao nível de detalhe mais baixo gerado. Esta forma de representação é muito mais eficiente pois, ao invés de se efectuar o *rendering* da mesma textura em diferentes resoluções permite a utilização de múltiplas texturas de diferentes resoluções. Este conceito é de certo modo análogo ao conceito de nível de detalhe discreto descrito em **3.4.1.1**, já que no *mipmapping* também existe um pré-processamento, neste caso de uma imagem, com o intuito de criar diferentes níveis de detalhe (*mipmaps*) para a textura resultante.

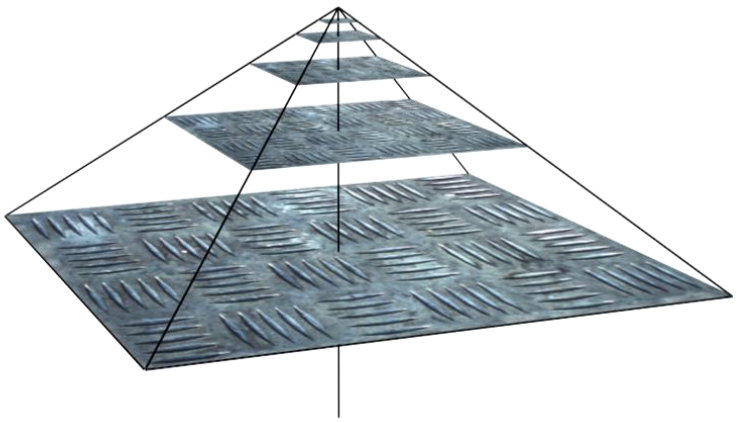


Figura 4-17: Uma pirâmide de *mipmaps*.

O Geomipmapping baseia-se num conceito semelhante mas aplicado a grelhas poligonais para as quais são criados diferentes níveis de detalhe que de uma forma análoga são designados de *geomipmaps*. Estes são uma cadeia de blocos com sucessivamente menos detalhe que o bloco original. Considera-se, assim, nesta analogia o bloco original como o nível de detalhe 0 (ou seja o máximo), o nível 1 como uma versão com menos detalhe (entenda-se vértices) e assim sucessivamente. De acordo com a

distância a que se encontra a câmara, é atribuído então um nível de detalhe ao bloco de terreno tal como está ilustrado na **Figura 4-18**.

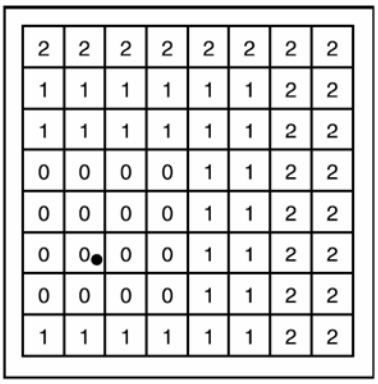
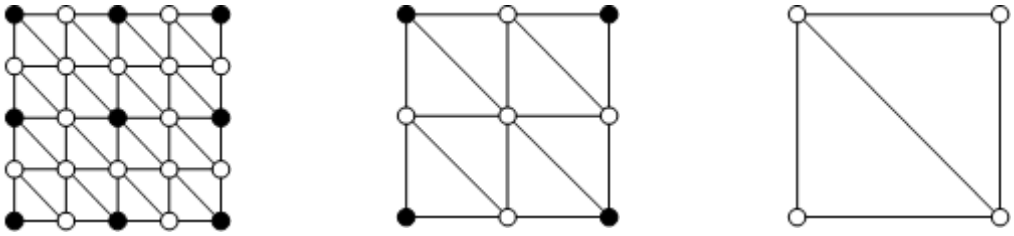


Figura 4-18: Nível de detalhe de blocos de terreno em relação ao ponto de vista [165].

A transição do nível 0 para o nível 2 pode ser observada na **Figura 4-19**. O que ocorre neste exemplo é uma eliminação dos vértices, passando-se de um bloco 5×5 para um bloco 3×3 e finalmente para um bloco de 2×2. Esta diminuição de detalhe, como se pode verificar, traduz-se numa diminuição no número de vértices, e consequentemente na perda do valor de elevação nesses pontos. A semelhança entre as duas técnicas, o *mipmapping* e Geomipmapping, torna-se portanto evidente, basta comparar a **Figura 4-19** com a **Figura 4-17**.



a) Nível 0 b) Nível 1 c) Nível 2

Figura 4-19: Diferentes níveis de detalhe.

4.4.3. A escolha do Geomipmap

A primeira decisão a tomar no âmbito deste algoritmo é que nível de *geomipmap* se irá utilizar para que distância. Este terá de ser escolhido de modo a que o impacto visual da mudança de nível (de mais vértices para menos vértices ou vice-versa) seja o menor possível, não causando um efeito de *popping* muito pronunciado. Este efeito de *popping* resulta, tal como foi referido em **3.4.4.2**, da mudança de altura dos vértices afectados (ver **Figura 3-34**). Isto é, com a eliminação ou adição de vértices, perde-se ou ganha-se informação de elevação, provocando-se uma súbita mudança no terreno que pode ser mais ou menos aparente e que só está dependente da distância a que esta transição se

realiza. Recorrendo de novo à analogia com os *mipmaps*, nesse caso a transição só deve ocorrer quando o rácio *pixel-texel*[4] já não é $1:1$, o que acontece a uma determinada distância. No âmbito dos *geomipmaps* utiliza-se um conceito semelhante com vista a reduzir o efeito de *popping*. Para isso, calcula-se para cada bloco, sujeito a uma mudança de nível a maior diferença de altura de todos os vértices afectados, ou seja, $max\{\delta_0, \dots, \delta_n\}$ em que δ_n representa a diferença de altura num determinado vértice (na **Figura 4-19** corresponde a um cálculo efectuado sobre todos os vértices que desaparecem numa transição de um nível para outro) projectando-se esse máximo obtido para coordenadas de ecrã, ε. Se o número de *pixels* respectivo estiver acima de um determinado valor τ (que pode ser por exemplo quatro e que é fornecido como parâmetro ao algoritmo), estipula-se nesse caso que a mudança para outro nível não deve ser efectuada.

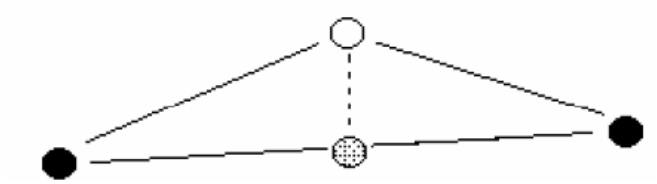


Figura 4-20: Mudança de altura que ocorre quando se remove o vértice branco [39].

O cálculo da maior diferença de altura e a sua comparação com o τ para cada *geomipmap* visível em cada *frame* seria extremamente ineficiente, por isso de modo a evitar esse peso considera-se uma simplificação baseada no facto do vector de direcção da câmara ser horizontal. Com esta nova possibilidade em tempo de execução, a única operação a executar por *frame* para cada bloco de terreno é a comparação da distância mínima calculada a que deve ser seleccionado cada *geomipmap* associado com a distância corrente da câmara ao centro do bloco. Este cálculo é feito utilizando a distância euclidiana (ver **Equação 3-1**). Para pré-calcular a distância mínima a partir da qual um *geomipmap* é visível, temos de recorrer às seguintes fórmulas:

$D_n = \lvert\delta\rvert \cdot C$	δ: A maior diferença de altura de todos os vértices alterados no bloco, ou seja $max\{\delta_0, \dots, \delta_n\}$.
	C: Calculado na **Equação 4-9**.

Equação 4-8: Cálculo da distância mínima para cada *geomipmap*.

[4] O rácio *pixel-texel* pode-se definir como a relação entre o tamanho do pixel de uma textura (também designado de *texel*) e o tamanho do pixel no ecrã. Por exemplo se um determinado objecto ocupar no ecrã 100 *pixels* e for utilizada uma textura de 50 *pixels* temos um rácio de 2 *pixels* para 1 *texel* (2:1). No caso de o objecto ocupar 100 *pixels* e a textura tiver 100 *pixels* nesse caso temos um rácio de 1:1.

$C = \frac{A}{T}$	A: Calculado na **Equação 4-10**.
	T: Calculado na **Equação 4-11**.

Equação 4-9: Permite calcular o valor de C na **Equação 4-8**.

$A = \frac{n}{\|t\|}$	n: Representa o plano de corte da frente (*near clipping plane*).
	t: A coordenada superior do plano de corte.

Equação 4-10: Permite calcular o valor de A na **Equação 4-9**.

$T = \frac{2 \cdot \tau}{V_{res}}$	τ: Representa o número máximo de *pixels* a partir do qual já não é permitida a mudança de nível.
	V_{res}: A resolução vertical em *pixels***.**

Equação 4-11: Permite calcular o valor de T na **Equação 4-9**.

4.4.4. Culling

O *culling* tal como já foi descrito em **3.3** é utilizado para remover da lista de objectos a enviar para *rendering* todos os não visíveis para um determinado ponto de vista. A remoção desses objectos, que são neste caso blocos de terreno, é essencial para um bom desempenho do algoritmo sendo utilizada para isso como estrutura de particionamento espacial a *quadtree* (ver **3.2.3.1**) cujo objectivo aqui é permitir a pesquisa de todos os blocos visíveis de uma forma eficiente. Mais precisamente, permite organizar espacialmente todos os blocos de terreno facilitando a pesquisa dos que se encontram visíveis em tempo de *rendering* pela intersecção de cada um deles com o *frustum*. Os que passam este teste são depois enviados finalmente para processamento no GPU.

4.4.5. Falhas

Quando dois *geomipmaps* vizinhos têm diferentes níveis de detalhe podem vir a aparecer falhas na representação do terreno se não forem tomadas medidas para as corrigir, problema, aliás, bem comum nestes algoritmos (ver **3.4.4.1**). Estas falhas ocorrem muito simplesmente porque existe uma diferença de vértices entre eles, ou seja, os vértices adjacentes no *geomipmap* de maior detalhe não tem correspondência no *geomipmap* vizinho de menor detalhe. Isto é algo que faz com que exista uma diferença de altura na coordenada *y* em relação ao *geomipmap* vizinho no qual não existe esse mesmo vértice mas apenas a aresta que liga os dois vértices vizinhos. Nota-se consequentemente um "buraco" no terreno algo que não é minimamente desejável e tem um grande impacto visual. A maneira mais fácil de resolver este problema, e ao mesmo tempo a mais eficiente, é alterar a ligação dos vértices, omitindo os que estão a mais nos *geomipmaps* de maior detalhe tal como está representado na **Figura 4-21** em que o *geomipmap* de maior detalhe à direita omite os vértices a cinzento de modo a que possa

"encaixar" com *geomipmap* de menor detalhe à esquerda, não causando os artefactos que surgem nas outras soluções. Esta é a solução proposta pelo autor e, de um modo geral, uma das melhores para resolver o problema (ver **3.4.4.1**).

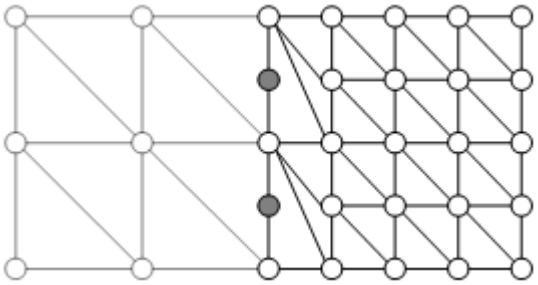


Figura 4-21: Omissão de vértices na resolução de falhas [162].

4.4.6. Trilinear Geomipmapping

De maneira a minimizar o efeito de *popping* o autor descreve uma técnica que utiliza mais uma vez a analogia com os *mipmap*: o *trilinear filtering*. Nesta, o objectivo é eliminar os artefactos visuais que ocorrem a determinadas distâncias da câmara entre dois níveis adjacentes de diferente resolução. O conceito é simples, em vez de se utilizar sempre o mesmo *mipmap* durante um determinado intervalo de distâncias é feita uma interpolação entre o *mipmap* corrente e o *mipmap* correspondente ao próximo nível. O mesmo princípio pode ser aplicado aos *geomipmaps*, só que neste caso especifico o objectivo é fazer o *morphing* entre cada um dos níveis de detalhe, ou seja mover os vértices de modo a que seja possível transitar suavemente de um nível para outro. Este conceito é basicamente o mesmo que o conceito de Geomorphing descrito em **3.4.4.2**. Neste caso para concretizar o *morphing* entre níveis de detalhe é utilizado o valor D_n calculado com a **Equação 4-8** para cada um dos n *geomipmaps* considerados. O D_n indica a distância a partir da qual o *geomipmap* n pode substituir o antecessor. Considere-se então para efeitos de exemplo dois *geomipmaps* consecutivos, D_n e D_{n+1}. Se calcularmos uma fracção entre eles (**Equação 4-12**), é possível obter o factor multiplicativo t utilizado na transição dos vértices entre os dois níveis (**Equação 4-13**).

$t = \frac{(d - D_n)}{(D_{n+1} - D_n)}$	d : Distância da câmara ao *geomipmap*.
	D_n: Distância mínima pré-calculada para o *geomipmap* n.
	D_{n+1}: Distância mínima pré-calculada para o *geomipmap* seguinte $n + 1$.

Equação 4-12: Cálculo da fracção de interpolação t.

$v' = v - t \cdot \delta_v$	v: O vértice a alterar.
	t: O factor de interpolação calculado na **Equação 4-12**.
	δ_v: Mudança de altura do vértice v (ou seja o erro geométrico).

Equação 4-13: Cálculo da nova posição do vértice.

Para ilustrar este conceito na **Figura 4-22** está representada a transição entre dois níveis de detalhe: do mais elevado (5×5) para o mais baixo (3×3). Neste caso, os vértices 1, 2 e 3 não necessitam de ser modificados pois mantêm-se no nível seguinte. Existem três casos específicos em que os vértices têm de ser modificados [208]:

- Caso *A*: o vértice está numa posição x ímpar e numa posição z par. O vértice tem de se mover para uma posição entre o próximo vértice à esquerda (o 1) e o próximo à direita (o 2).
- Caso B: o vértice está numa posição x e z ímpar. O vértice tem de se mover para uma posição entre o próximo vértice em cima e à esquerda (o 1) e o próximo em baixo à direita (o 3).
- Caso C: o vértice está numa posição x par e numa posição z ímpar. O vértice tem de se mover para uma posição entre o próximo vértice em cima (o 2) e o próximo em baixo (o 3).

Note-se que este movimento ocorre apenas ao longo da coordenada vertical, o y de cada um dos vértices considerados.

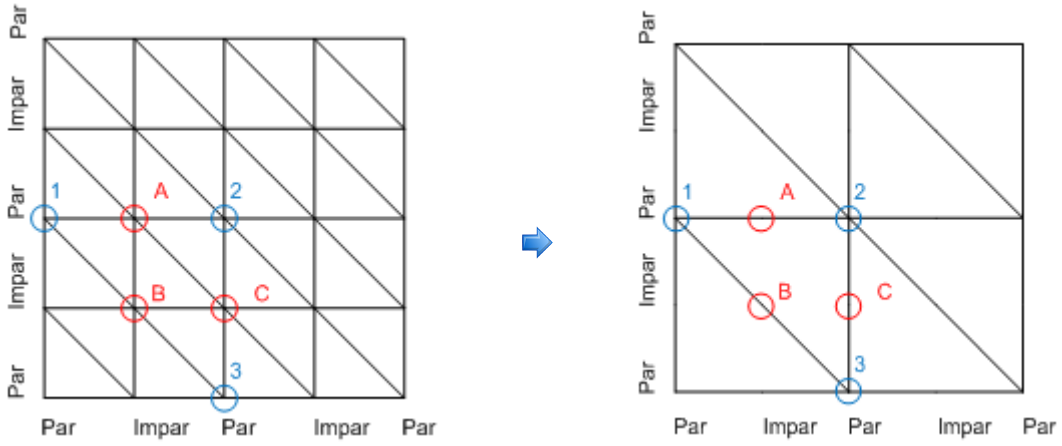


Figura 4-22: Vértices afectados pelo Geomorphing entre dois níveis de detalhe [208].

4.4.7. Variações do Algoritmo

Uma das variações mais importantes é a aproximação descrita por Daniel Wagner em [208] que aplica o Geomorphing de uma forma diferente da descrita por Boer (ver **4.4.6**). O que distingue esta aproximação da original, merecendo por isso especial destaque, é o facto de na concretização do Geomorphing se ter levado em consideração os blocos vizinhos e a forma como foi efectuada permitir também o tratamento de falhas. Numa

implementação típica de Geomorphing o aplicar do factor de *morphing* a cada um dos vértices é feita normalmente por bloco, o que dependendo da forma como é efectuado o tratamento de falhas pode ou não dar origem a mais falhas. Basta para isso pensar que a movimentação dos vértices em cada um dos blocos é independente, logo, tendo em conta que o factor de *morphing* aplicado está dependente da distância à câmara é fácil concluir que essa movimentação não é feita ao mesmo ritmo, isto é, o movimento dos vértices num bloco não está coordenado com o movimento dos vértices noutro bloco, pelo que podem ocorrer falhas na geometria se estas forem tratadas como é sugerido por Boer em **4.4.5**. Na abordagem sugerida por Wagner, a forma como é feito o processo de Geomorphing acaba por resolver o problema das falhas já que o movimento dos vértices tem em consideração o factor de Geomorphing dos blocos vizinhos.

4.5. Chunked LOD

O algoritmo de Chunked LOD [200][201][185] concebido por Thatcher Ulrich é uma técnica de geração de terrenos em tempo real simples mas efectiva, que se caracteriza por um pré-processamento da fonte de elevação e da textura que a representa com o intuito de criar uma *quadtree* (ver **3.2.3.1**) de *chunks* optimizados para *rendering*. Um *chunk* é constituído pela malha poligonal e pela textura associada à porção do terreno que representa. Desta forma, na *quadtree*, o nó raiz contém uma representação do terreno em baixo detalhe e cada um dos nós descendentes representa o nó pai em níveis de detalhe progressivamente mais elevados. Neste algoritmo, o principal objectivo não é obter uma triangulação óptima, mas maximizar a quantidade de triângulos enviados para processamento no GPU minimizando a intervenção do CPU. Para isso, a gestão do nível de detalhe é efectuada ao nível de blocos de terreno e não de vértices numa perspectiva muito semelhante à do algoritmo de Geomipmapping (ver **4.4)**, com o qual partilha algumas ideias, mas do qual difere também num conjunto de pontos importantes. Assim de acordo com Ulrich os principais objectivos deste algoritmo são [201]:

- Correr em *hardware* comum;
- Permitir o carregamento em tempo de execução de dados a partir do disco;
- Suportar texturas de alta resolução;
- Ser fácil de implementar;
- Minimizar quanto possível o processamento ao nível do CPU;
- Eliminar o *popping* (ver **3.4.4.2**) dos vértices e texturas;
- Permitir a atribuição de mais detalhe as zonas que dele mais necessitem.

Neste conjunto de características é de realçar o suporte para o Geomorphing na eliminação do *popping*, o carregamento de *chunks* a partir do disco (que lhe permite suportar terrenos de dimensão arbitrária) e a gestão integrada do nível de detalhe do terreno e da textura que o representa. Outra característica relevante é a técnica que emprega na correcção das falhas entre *chunks* de diferentes níveis de detalhe: o *skirting* (ver **4.5.2**). O *skirting* implica o adicionar de triângulos extra em torno do *chunk*, que se estendem à sua volta de uma forma análoga aos lençóis à volta de uma cama o que permite que o problema das falhas seja resolvido de uma forma estática durante a fase de pré-processamento. De acordo com Ulrich este algoritmo tem como principais vantagens:

- Uma utilização eficiente de triângulos

- Nível de detalhe ao nível das texturas e da geometria integrado numa *quadtree*.
- Permitir o armazenamento de partes do terreno em disco, nomeadamente quando a dimensão do terreno não permite colocar tudo em memória.
- *Popping* eliminado com a implementação do Geomorphing.
- Carga reduzida no CPU, mesmo quando o ponto de vista se move rapidamente.

As características negativas incluem:

- Pré-processamento dos dados não é trivial.
- Os dados têm de ser estáticos.
- Usa mais primitivas para a mesma medida de erro.

4.5.1. Representação do terreno

Tal como o Geomipmapping também o Chunked LOD utiliza uma grelha regular em que o número de vértices horizontais e verticais tem de ser da forma $2^n + 1$ em que $n \in [1, \rightarrow)$. A grelha também é subdividida num conjunto de blocos que são armazenados numa *quadtree*. No entanto, ao contrário do Geomipmapping, a *quadtree* é construída numa fase de pré-processamento e não serve apenas de suporte ao *frustum culling*, neste caso permite também a gestão do nível de detalhe para cada um dos blocos de terreno. Assim cada nó da *quadtree* cobre uma parte do terreno em níveis sucessivos de detalhe armazenando em cada um, porções independentes do terreno: os *chunks*. Destes fazem parte a estrutura geométrica e a textura correspondente a porção que representam num determinado nível de detalhe, que está associado a profundidade a que se encontram na *quadtree*. É devido a esta independência entre blocos que o seu carregamento a partir do disco se torna fácil de implementar sendo assim possível representar terrenos de dimensões muito superiores à memória existente. Assim, na raiz da *quadtree* está um *chunk* de baixo detalhe que representa todo o terreno. Os quatro filhos são por sua vez uma subdivisão deste em quatro *chunks* de menor dimensão e de maior detalhe, isto de uma forma recursiva até atingirmos uma profundidade predefinida na qual estão os *chunks* de maior detalhe possível para a secção de terreno delimitada pela subdivisão corrente. Significa isto que o detalhe aumenta de uma forma proporcional ao nível de profundidade em que nos encontramos na *quadtree* e, paralelamente, a área coberta por cada uma das subdivisões diminui. Na implementação disponibilizada por Ulrich, são construídas, tal como está representado na **Figura 4-23**, duas *quadtrees* na fase de pré-processamento: uma a partir do *height field* e outra a partir da textura que o representa. Cada uma delas é armazenada num ficheiro funcionando como parâmetros de entrada para o algoritmo. Em tempo de execução para cada uma das porções de terreno que constituem a *quadtree* é então seleccionado o nível de detalhe mais apropriado de acordo com o ponto de vista corrente (ver **4.5.2**). Para isso, a informação relevante é extraída de cada um dos ficheiros, mais precisamente a estrutura geométrica e a textura associada.

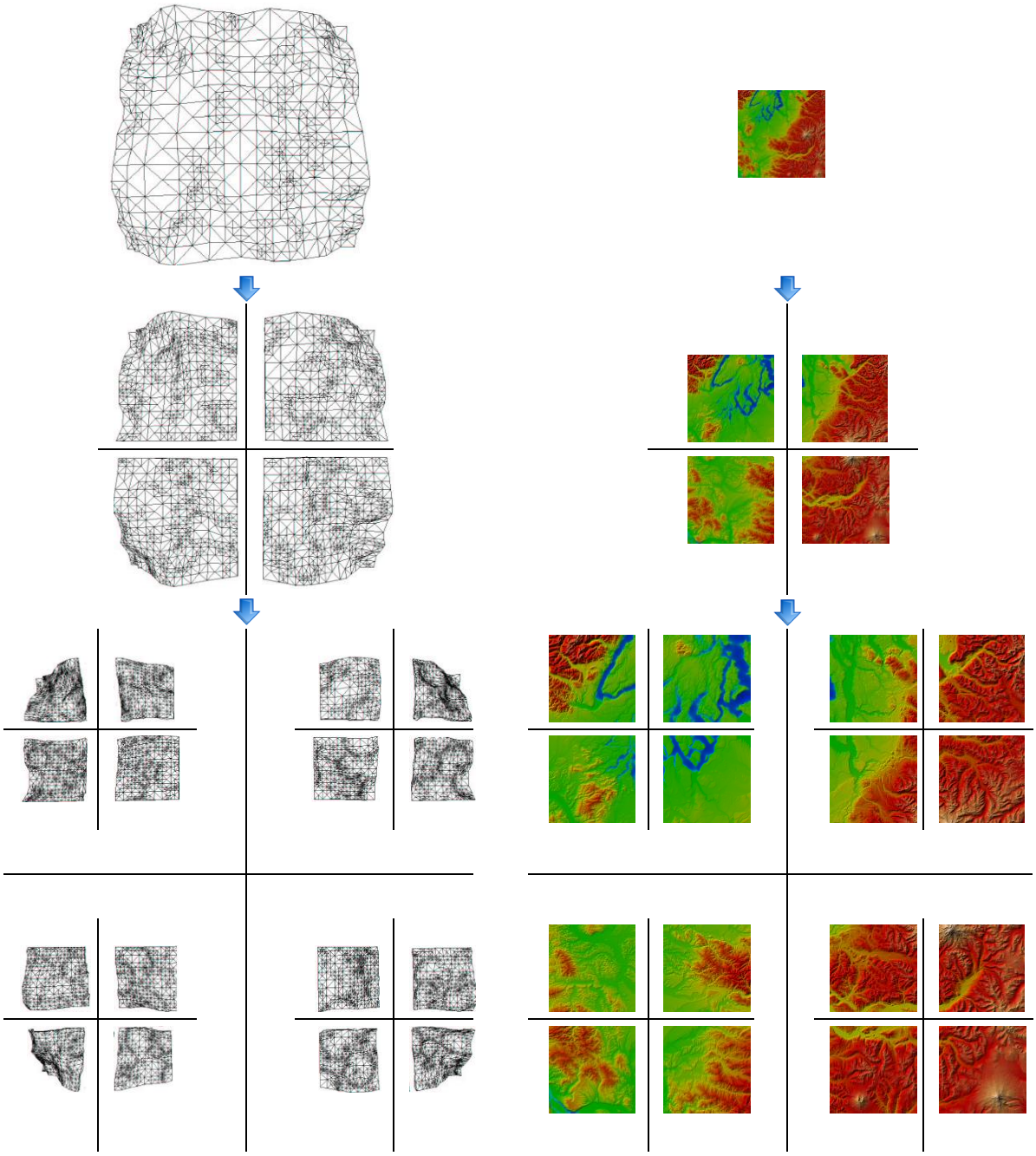


Figura 4-23: Os três primeiros níveis de uma *chunked quadtree* [25].

No que diz respeito à primitiva (ver **3.1**) utilizada na triangulação do terreno, este algoritmo não está, ao contrário do algoritmo de Röttger (ver **4.3**), associado a uma determinada primitiva, isto porque a triangulação é feita numa fase de pré-processamento e na correcção de falhas é utilizado o *skirting*. Significa isto, que é possível utilizar qualquer uma das primitivas descritas em **3.1** devendo o *rendering* de cada um dos *chunks* ser, por questões de desempenho, concretizado com uma única chamada.

4.5.2. A Escolha do Nível de Detalhe

A cada um dos nós é atribuído um erro geométrico δ que representa o desvio máximo do *chunk* associado em relação à versão original da zona do terreno que representa. Ulrich atribuiu o mesmo δ a todas as porções do terreno num mesmo nível de detalhe na *quadtree* diminuindo o δ de nível para nível em metade do seu valor anterior. O δ num determinado nível é assim calculado pela **Equação 4-14**, ou seja, em função do nível de detalhe anterior. Por exemplo se na raiz da árvore o terreno tem um desvio de 16 unidades em relação a versão original ou seja $\delta(0) = 16$ então no segundo nível cada um dos quatro *chunks* tem um desvio de 8 unidades em relação a zona do terreno que representa, $\delta(1) = 8$, e assim sucessivamente até atingirmos o quinto nível de detalhe em que $\delta(4) = 1$ e no qual a porção original do terreno é representada.

$\delta(L+1) = \frac{\delta(L)}{2}$	L: Nível na *quadtree*.

Equação 4-14: Cálculo do erro geométrico máximo para um *chunk*.

Em tempo de execução para efectuar o *rendering* do terreno basta escolher os *chunks* que melhor representam cada uma das porções consideradas, de acordo com um parâmetro fornecido ao algoritmo, τ, que representa o erro máximo em *pixels*, ou seja, a fidelidade desejada [200]. Para isso, e à semelhança de outros algoritmos, é calculado o erro geométrico máximo em coordenadas de ecrã, ρ, que é determinado pela **Equação 4-15** tendo-se também em consideração o factor de escala da perspectiva calculado na **Equação 4-14**.

$\rho = \frac{\delta}{D} \cdot K$	δ: Calculado na **Equação 4-14**.
	D: Distância da câmara ao ponto mais próximo do *chunk*.
	K: Calculado na **Equação 4-16**.

Equação 4-15: Cálculo do erro do *chunk* em *pixels* no espaço do ecrã.

$K = \frac{W}{2 \cdot \left(\tan\left(\frac{FOV}{2}\right)\right)}$	W: Largura do ecrã em *pixels*.
	FOV: Campo de visão.

Equação 4-16: Cálculo do factor de escala da perspectiva.

A escolha do nível de detalhe mais apropriado é assim feita recursivamente iniciando-se na raiz da *quadtree* pelo cálculo para cada *chunk* da **Equação 4-15**. O objectivo é determinar se o erro em coordenadas de ecrã é ou não aceitável para o valor de τ seleccionado. Se não for, é necessário descer mais na hierarquia até encontrar o *chunk* mais adequado ao ponto de vista corrente, um procedimento representado em pseudo-código na **Listagem 4-3**.

```
01 Render_lod(node)
02  If rho(node, viewpoint) <= tau Then
03   draw(node.mesh)
04  Else
05   For Each c In node.children
06    render_lod(c)
07   End For
08  End If
09 End RederLod
```

Listagem 4-3: Selecção do nível de detalhe [200].

4.5.3. Culling

Tal como nos outros algoritmos, o *culling* é feito via *view frustum culling* (ver **3.3.2**) utilizando como suporte a *quadtree* construída na fase de pré-processamento. Funciona assim pela intersecção do *bounding volume* (ver **3.2.1**) de cada um dos *chunks* com o *frustum* ou seja são rejeitados todos *chunks* que não passem esse teste. Nesse sentido são, como seria de esperar, enviados para *rendering* apenas os *chunks* que passem por este teste inicial.

4.5.4. Falhas

Como cada um dos *chunks* é seleccionado independentemente, não existe garantia de que as arestas de dois *chunks* vizinhos se encaixem perfeitamente sem dar origem a falhas (ver **3.4.4.1**). Para corrigir este problema Ulrich usou uma técnica que designou de *skirting*. As *skirts* ou saias poligonais que a caracterizam são colocadas em redor de cada um dos *chunks* (ver **Figura 4-24**) e tem uma altura suficiente para compensar o desvio na altura em relação ao *chunk* adjacente.

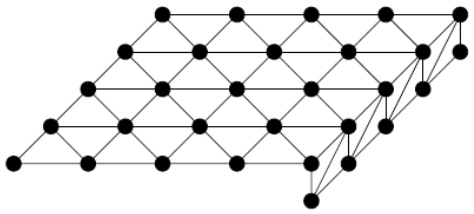


Figura 4-24: Uma *skirt* em torno de um *chunk* (apenas para o lado direito) [107].

A resolução do problema das falhas desta forma tem no entanto o inconveniente de adicionar mais triângulos ao terreno original na construção das *skirts*. Por outro lado dá origem a pequenas imprecisões cujo efeito é dificilmente detectável em tempo de execução, principalmente se tivermos em consideração o efeito de perspectiva e um valor de τ suficientemente baixo (ver **4.5.2**). Nesse caso a diferença de altura pode nessas condições corresponder a um *pixel* ou menos sendo deste modo praticamente indetectável [185]. Esta diferença na altura resulta do desnível entre dois *chunks* adjacentes de diferente detalhe e está representada na **Figura 4-25**.

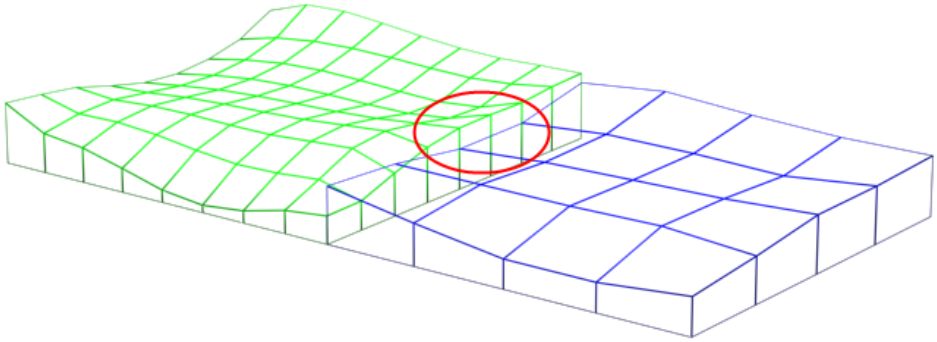


Figura 4-25: Diferença na altura entre dois níveis de detalhe adjacentes.

4.5.5. Geomorphing

O Geomorphing é, tal como a correcção de falhas, essencial na representação de terrenos com diferentes níveis de detalhe uma vez que permite que o efeito de *popping* (ver **3.4.4.2**) seja eliminado aumentando a qualidade da experiência do utilizador. No contexto deste algoritmo este efeito ocorre na transição entre *chunks* de diferente detalhe, mais precisamente, no momento em um *chunk* é dividido em quatro *chunks* de maior detalhe ou no caso inverso no qual os quatro *chunks* se transformam num único com menor detalhe. A concretização do Geomorphing no algoritmo de Chunked LOD envolve o aplicar de um pequeno efeito (possivelmente 0) de *morphing* à coordenada vertical de cada vértice que compõe o *chunk*, utilizando-se para isso um parâmetro de *morph* igual para cada um dos vértices que o constituem [200]. Nessa perspectiva, e uma vez que a mudança de um nível de detalhe mais alto para um nível de detalhe mais baixo envolve a eliminação de um conjunto de vértices, é necessário estimar a sua posição no *chunk* antecessor para o qual se vai transitar. Esta posição é também designada de *morph target* sendo este valor calculado para cada vértice de um *chunk* na fase de pré-processamento. Em tempo de execução, o parâmetro de *morph* varia entre 0 e 1, na transição de um nível de detalhe mais baixo para um nível de detalhe mais alto, ou seja de 4 *chunks* para 1, e entre 1 e 0 no caso contrário. Daí podemos concluir que um *chunk* se divide em 4 *chunks* de maior detalhe quando o parâmetro é 0 e que nessa situação cada um dos seus filhos tem um valor de 1 para esse parâmetro. Para o cálculo do parâmetro de *morph* é utilizada a **Equação 4-17** que permite efectuar uma interpolação linear entre os dois valores.

$t_{morp\ h} = \text{clamp}\left(\frac{2 \cdot \rho}{\tau} - 1{,}0{,}1\right)$	clamp: Constrange o resultado ao intervalo [0,1].
	ρ: Calculado na **Equação 4-17**.
	τ: Erro máximo em *pixels*.

Equação 4-17: Cálculo do parâmetro de *morph*.

4.6. Terrain Occlusion Culling with Horizons

Este algoritmo concebido por Glen Fiedler e descrito pela primeira vez em [63] utiliza uma abordagem ao problema da oclusão em terrenos que se baseia na construção de um horizonte. Muito embora a utilização de horizontes na determinação da oclusão não seja uma novidade (ver **3.3.4**), é especialmente útil no domínio da geração de terrenos em tempo real em grande parte devido a sua simplicidade. Mais precisamente, a forma como este algoritmo foi desenhado torna-o especialmente adaptado para situações em que é efectuado um particionamento espacial do terreno via uma *quadtree* (ver **3.2.3.1**) sendo por isso de fácil integração com outros algoritmos como o Geomipmapping (ver **4.4**), o Chunked LOD (ver **4.5**), o GPU Terrain Rendering (ver **4.8**) e o Rendering Very Large, Very Detailed Terrains (ver **4.7**). Por ser um algoritmo que utiliza uma linha de horizonte, é essencial que todos os blocos que compõem o terreno sejam ordenados de acordo com a sua proximidade à câmara e enviados por essa ordem para a placa gráfica de modo a que seja possível ir incorporando a contribuição de cada um deles. O objectivo é identificar os que estão abaixo da linha de horizonte até aí definida para que possam ser rejeitados. Segundo Fiedler, uma das principais vantagens desta abordagem é não implicar uma fase de pré-processamento demasiado pesada e de se adaptar dinamicamente a mudanças no terreno. Mais especificamente, a aproximação descrita destaca-se sobretudo pela técnica que utiliza na construção do horizonte permitindo a sua geração de uma forma eficiente e em tempo de execução.

4.6.1. Culling com Horizontes

Para implementar um algoritmo de oclusão com horizontes é necessário [63]:

- Que a cena seja percorrida por ordem de distância dos objectos à câmara.
- Alguma forma de ir armazenando a linha de horizonte à medida que se vai percorrendo a cena.
- Um teste que determine se um determinado objecto está abaixo da linha de horizonte.
- Um método que permita incorporar a contribuição desse objecto para a linha de horizonte.

Para armazenar a linha do horizonte é utilizado um simples *array* com uma dimensão igual ao número de *pixels* da resolução horizontal do ecrã (por exemplo 800, 1024…). Cada entrada nesse *array* representa a altura do horizonte para uma coluna de *pixels* no ecrã em que um valor de 0 representa o fundo do ecrã e os valores positivos correspondem directamente à coordenada *y* do horizonte [63]. Para testar um objecto é necessário comparar os *pixels* que ocupa contra os valores correspondentes no *array*. Basicamente, se a coordenada *y* de qualquer um dos *pixels* for maior ou igual que o valor correspondente no *array* então o objecto tem de estar visível. Caso contrário, o objecto está abaixo do horizonte sendo por isso rejeitado. A actualização do *array* é feita de uma forma semelhante, isto é, sempre que o espaço ocupado pelo objecto incorporar valores em *y* superiores aos valores no *array* este é actualizado de maneira a incorporar esses valores. O objectivo é ter em conta esse objecto para a definição da linha de horizonte de

modo a que mais tarde possa contribuir para determinar se outros objectos estão ou não ocultos.

4.6.2. Aproximação

A forma mais simples de construir o horizonte é através de uma abordagem de força bruta que envolve os seguintes passos [63]:

1. Percorrer a cena da frente para trás.
2. Para cada bloco verificar se está abaixo do horizonte. Tal implica testar cada um dos triângulos que compõem esse bloco.
3. Se todos os triângulos que compõem o bloco estiverem abaixo do horizonte então o bloco não está visível.
4. Caso contrário enviar o bloco para *rendering* e actualizar o horizonte de modo a que possa incluir a contribuição desse bloco (ver **Figura 3-26**).

Esta abordagem, muito embora seja simples não é, como seria de esperar, a mais eficiente pois implica o teste de todos os triângulos. Por exemplo, para um *height field* de 1024×1024 seriam necessários 6291456 testes considerando 2 triângulos por célula e três arestas por triângulo. Ora este número de testes é claramente incomportável, pelo que se justifica a procura de soluções mais eficientes que permitam de alguma forma aproximar a precisão do teste de força bruta mas que possam em contrapartida produzir resultados não muito distantes do ideal. A palavra-chave é então "aproximação", isto é, há que procurar uma forma de construir o horizonte sem usar directamente os triângulos do *height field*, mas sim uma aproximação do mesmo. Tal como é sublinhado pelo autor, é crítico que para essa aproximação se armazene o valor máximo e mínimo do erro associado [63] já que, o objectivo é efectuar um teste conservativo de oclusão, ou seja garantir que um objecto visível nunca é marcado como oculto, assumindo por outro lado a possibilidade de objectos invisíveis serem enviados para *rendering*. Para garantir esta importante propriedade é necessário que se teste o valor do erro máximo de cada um dos blocos de terreno relativamente ao horizonte.

Assim, este algoritmo envolve a construção de uma aproximação ao terreno numa fase de pré-processamento e a sua reutilização sempre que é necessário efectuar o *culling* com o horizonte.

4.6.3. Construção do Horizonte

Para construir a linha do horizonte, uma das formas mais simples é utilizar uma linha horizontal que passe pela média da altura de todos os pontos do terreno. Tal como foi referido antes, é necessário registar o valor máximo e mínimo dessa aproximação, por isso, são necessárias linhas que passem por esses valores tal como está representado na secção superior esquerda da **Figura 4-26**. Tal como se pode visualizar na figura, a construção do horizonte implica o refinar sucessivo de diferentes aproximações, sendo necessário em cada uma delas dividir o horizonte e calcular uma nova linha média bem como novas linhas mínimas e máximas. Estas divisões são recursivas e só terminam quando se atinge um determinado limiar de erro ou quando se chega a resolução máxima do *height field*, construindo-se efectivamente uma árvore binária que permite a

construção do horizonte em vários de níveis precisão. O erro é definido como a distância vertical entre a linha mínima e máxima, sendo que, a construção do horizonte é feita percorrendo a árvore binária da raiz para baixo parando quando o erro excede um determinado limiar ou se chega à resolução máxima. Os vários níveis envolvidos na construção de uma linha de horizonte são visíveis na **Figura 4-26** onde é simulada a sua construção.

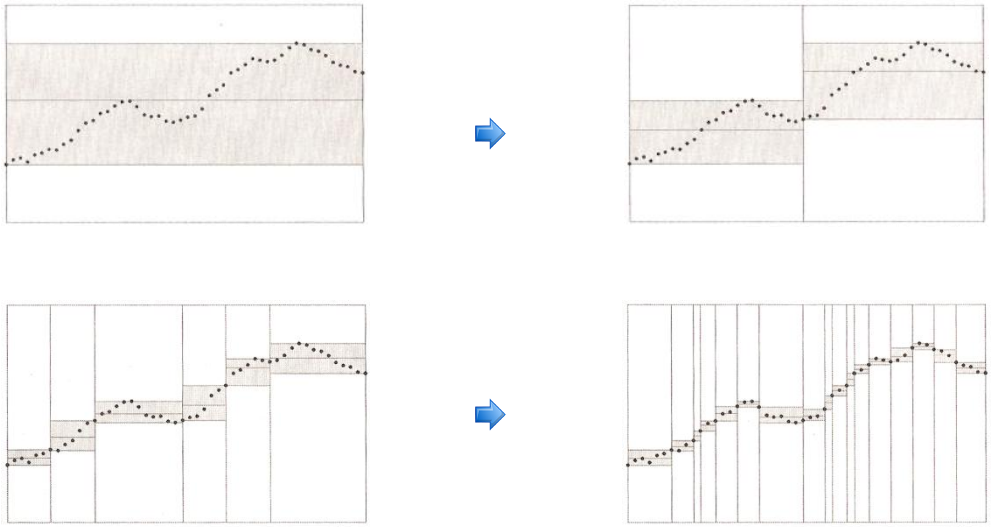


Figura 4-26: Aproximação do horizonte com linhas horizontais [63].

As linhas horizontais não são a melhor solução para obter boas aproximações do horizonte, implicando muitas chamadas recursivas para se chegar a bons resultados. Uma solução que permite produzir aproximações melhores passa pela utilização de linhas de inclinação arbitrária. Esta nova abordagem está representada na **Figura 4-27** e tal como se pode observar permite atingir um grau de precisão elevado com muito menos chamadas recursivas, ou seja permite atingir o erro desejado muito mais rapidamente [63].

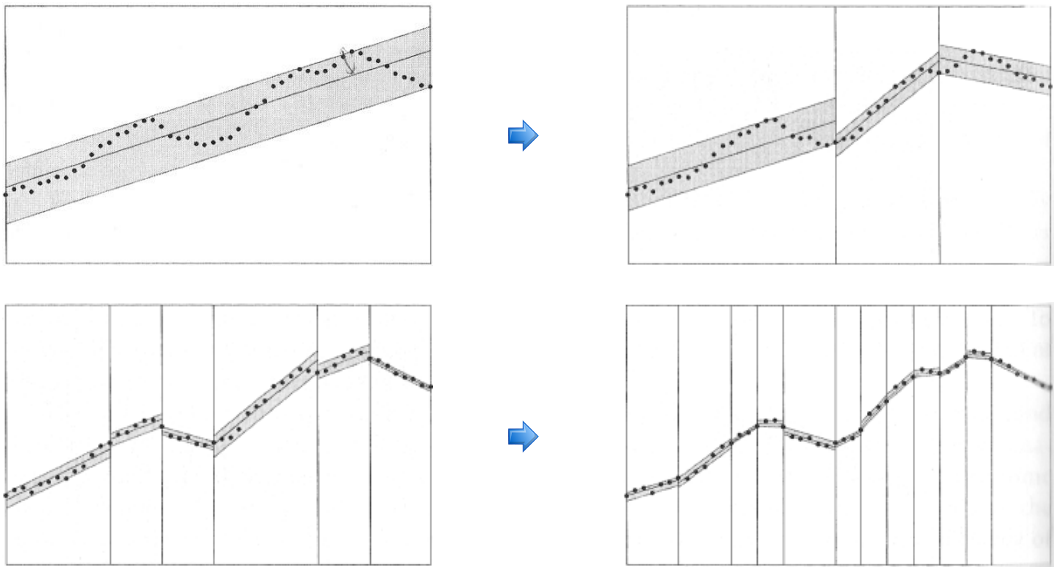


Figura 4-27: Aproximação do horizonte com linhas inclinadas [63].

Para calcular a linha de inclinação arbitrária que passa pela média de todos os pontos a solução mais comum é efectuar o cálculo da linha que minimiza a soma dos quadrados das distâncias verticais entre ela própria e cada ponto, também designada de linha dos mínimos quadrados. Assim, dado um conjunto n de pontos na linha do horizonte: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ a linha dos mínimos quadrados é definida como $y = ax + b$ em que o valor de a é calculado pela **Equação 4-18** e o valor de b é calculado pela **Equação 4-19**.

$a = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$	x_i: O valor da coordenada x_i.
	y_i: O valor da coordenada y_i.

Equação 4-18: Cálculo de a.

$b = \frac{(\sum_{i=1}^{n} y_i)(\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} x_i y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$	x_i: O valor da coordenada x_i.
	y_i: O valor da coordenada y_i.

Equação 4-19: Cálculo de b.

A aproximação do horizonte apresentada é uma solução a duas dimensões para o problema, mas o que realmente se pretende é uma solução a três dimensões. Para tal, o principio a aplicar é o mesmo só que desta vez o objectivo é aproximar o conjunto de pontos que constituem o terreno não por uma linha mas por um plano. Para isso ao invés de se utilizar uma árvore binária utiliza-se uma *quadtree* (ver **3.2.3.1**). Assim, na solução a duas dimensões as linhas que aproximavam secções de uma linha horizontal transformam-se na solução a três dimensões em planos que aproximam regiões quadradas do terreno. O princípio efectivamente é o mesmo: primeiro é calculado o plano que

melhor minimiza a distância entre si e todos os pontos que representa. Depois são obtidos a partir desse plano dois outros planos que representam o valor máximo e mínimo do erro. Cada um destes passos é efectuado de forma recursiva pela subdivisão sucessiva do *height field* com base na *quadtree*. Ou seja, é calculado para cada bloco os planos correspondentes e o mesmo processo se aplica de forma recursiva a cada um dos quatro nós filhos de um nó na *quadtree*. A recursividade pára quando o erro é aceitável ou quando um nó folha é atingido (ou seja a resolução do *height field* é atingida). A métrica de erro utilizada é o erro geométrico em *pixels* derivado a partir do valor máximo da distância vertical entre os planos máximos e mínimos utilizando-se para isso a **Equação 4-20**.

$(1-f)\cdot\frac{e}{D}\cdot K$	f: A coordenada *y* do vector de direcção da câmara.
	e: O valor máximo de distancia vertical entre os planos mínimo e máximo e o plano central.
	D: Distância da câmara ao centro do bloco.
	K: Calculado na **Equação 4-21**.

Equação 4-20: Cálculo da métrica de erro que controla a recursão.

$K=\frac{W}{2\cdot\left(\tan\left(\frac{FOV}{2}\right)\right)}$	W: Largura do ecrã em *pixels*.
	FOV: Campo de visão.
	tan: Função de cálculo da tangente.

Equação 4-21: Cálculo do factor de escala da perspectiva.

A *quadtree* que representa a subdivisão de um *height field* é apresentada na **Figura 4-28** onde para cada bloco se estabelece três planos: o plano intermédio, o plano máximo e o plano mínimo.

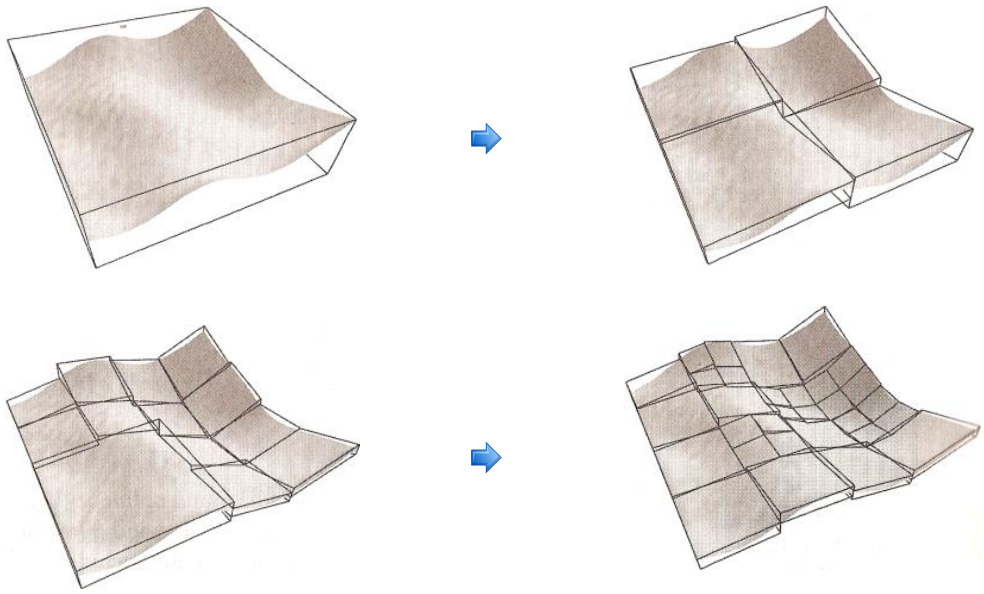


Figura 4-28: Aproximação do terreno para *culling* com horizontes [63].

O plano intermédio é obtido calculando uma versão da linha dos mínimos quadrados a três dimensões denominada de plano dos mínimos quadrados, que permite obter o plano que minimiza a distância entre todos os pontos do bloco de terreno que aproxima. O cálculo do plano dos mínimos quadrados baseia-se no mesmo princípio que o cálculo da linha dos mínimos quadrados: dado um conjunto n de pontos do *height field* $(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n)$ o objectivo é determinar A, B e C de modo a que o plano $z = Ax + by + C$ seja o plano que melhor representa o conjunto de pontos, isto é, pretende-se que a soma do quadrado dos erros entre z_i e os valores do plano $Ax_i + By_i + C$ sejam minimizados [51]. Isto leva a um sistema de três equações lineares em A, B e C representado na **Equação 4-22** que permite calcular $z = Ax + by + C$.

$$\begin{bmatrix} \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i y_i & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i y_i & \sum_{i=1}^{n} y_i^2 & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} x_i z_i \\ \sum_{i=1}^{n} y_i z_i \\ \sum_{i=1}^{n} z_i \end{bmatrix}$$

x_i: O valor da coordenada x_i.
y_i: O valor da coordenada y_i.
z_i: O valor da coordenada z_i.

Equação 4-22: Cálculo do plano dos mínimos quadrados [51].

4.6.4. Culling com Aproximação

A partir da representação construída é agora possível efectuar o *culling* de blocos que não estão visíveis por estarem ocultos por outros blocos. Assim, para construir o

horizonte, o algoritmo envolve o percorrer de uma *quadtree* até serem garantidos *n pixels* de erro. Esta *quadtree* armazena em cada um dos nós três planos: o plano máximo, o plano mínimo e o plano intermédio sendo este ultimo não mais do que o plano dos mínimos quadrados para a porção de terreno representada. Desta forma, em vez de se utilizar directamente os triângulos na actualização do *array* do horizonte, são utilizados os planos máximos e mínimos. Mais precisamente como o nó cobre um quadrado do terreno sabemos que os planos máximos e mínimos intersectam os limites deste quadrado formando arestas verticais que são rasterizadas para o *array* que armazena o horizonte (ver **Figura 4-28**), tal como acontecia com arestas dos triângulos na aproximação de força bruta (ver **4.6.2**). Para testar se o nó está acima do horizonte é utilizado o plano máximo mas para adicionar o nó ao horizonte é utilizado o plano mínimo. Isto garante que qualquer erro na aproximação é correctamente tratado e que o teste é conservativo.

4.6.5. Construção Dinâmica do Horizonte.

A construção da representação do *height field* na fase de pré-processamento envolve o percorrer da *quadtree* do nó raiz para os nós filhos. Por exemplo, para um *height field* de 1024×1024 primeiro é calculado o plano dos mínimos quadrados (ver **4.6.3**) para todo o terreno. Depois para cada um dos quatro nós filhos com uma dimensão de 512×512, é calculado novamente o plano dos mínimos quadrados e assim sucessivamente até se atingir a resolução do *height field*. Se o terreno for estático, esta aproximação é perfeitamente viável, uma vez que envolve percorrer o *height field* apenas uma vez e não é necessária uma actualização em tempo de execução. No entanto, para actualizações dinâmicas do terreno, esta operação é demasiado pesada para correr em tempo de execução, pois ainda demora alguns segundos dependendo do tamanho do terreno. Assim é necessário actualizar a representação construída na fase de pré-processamento de outra forma. A solução proposta pelo autor envolve o percorrer da *quadtree* dos nós folhas para o nó raiz sempre que seja necessário actualizar uma porção do terreno. O objectivo é reutilizar os cálculos efectuados para o plano dos mínimos quadrados dos nós filhos sendo tal possível porque como se pode verificar na **Equação 4-22** estes cálculos envolvem apenas uma matriz de somas, ou seja a matriz não necessita de ser calculada explicitamente, pode ser criada somando as matrizes dos nós filhos. Da mesma forma, no cálculo dos planos máximos e mínimos podemos utilizar os planos calculados para os nós filhos pelo que não é necessário voltar a calcular todos os valores. Para aumentar ainda mais o desempenho, a geração dos novos planos é feita apenas para as zonas afectadas do terreno. Por exemplo, se a alteração incidir apenas sobre uma zona de 64×64 será apenas a partir dos nós folha dessa zona que se vão construir os novos planos.

4.7. Rendering Very Large Very, Detailed Terrains

O algoritmo de Rendering Very Large, Very Detailed Terrains descrito por Thomas Lauritsen e Steen Lund Nielsen em [107] é uma aproximação híbrida que combina os algoritmos de Geomipmapping (ver **4.4**) e de Chunked LOD (ver **4.5**) numa única abordagem que utiliza o método de simplificação do Geomipmapping e a *chunked quadtree* do Chunked LOD. Algumas das suas características mais relevantes são a correcção de falhas (ver **3.4.4.1**) por intermédio de *skirts* e a diminuição do efeito de

popping (ver **3.4.4.2**) através do Geomorphing. Destaca-se, no entanto, por suportar terrenos de grande dimensão carregando para isso os dados progressivamente a partir do disco e também por permitir o adicionar de detalhe em tempo de execução pela extensão da *quadtree* (ver **3.2.3.1**) que lhe serve de suporte. Combina assim algumas das características mais importantes dos algoritmos em que se baseia mantendo de acordo com os autores o desempenho do Chunked LOD e a simplicidade do Geomipmapping.

4.7.1. Representação do Terreno

Tal como no Chunked LOD (ver **4.5**), também aqui a *quadtree* (ver **3.2.3.1**) tem um papel preponderante, servindo de base para a representação do terreno. É construída de uma forma tradicional, isto é, cada nó cobre um quarto da área do nó pai e os quatro nós filhos cobrem exactamente a mesma área do nó pai, de tal forma que o nó raiz cobre toda a superfície do terreno e os seus nós filhos vão cobrindo uma área progressivamente menor à medida que o número de níveis aumenta. Ainda à semelhança do Chunked LOD, é armazenado em cada nó da *quadtree* uma *mesh* que representa a zona do terreno coberta por esse nó cujo detalhe aumenta à medida que o número de níveis aumenta. Isto é, no nó raiz é armazenada uma *mesh* que representa todo o terreno com o menor detalhe possível ao passo que os quatro nós filhos armazenam cada um, uma *mesh* que representa um quarto do terreno mas com um maior nível de detalhe. Daí resulta que o nível de detalhe da *mesh* vai aumentando e a área coberta diminuindo até se chegar aos nós folha que representam o terreno com o maior detalhe possível. Esta *mesh* é designada de *chunk* por armazenar um bloco estático de geometria completamente independente dos outros, o que é uma boa propriedade uma vez que simplifica a implementação e permite suportar *chunks out-of-core* isto é não residentes em memória [107]. Além da geometria, o nó da *quadtree* armazena também uma textura com a resolução mais adequada para a região que representa, o que permite tornar cada um dos nós independentes não só a nível geométrico mas também ao nível da textura que lhe esta associada. A geração da textura e dos *chunks* é feita numa fase de pré-processamento, no entanto na construção dos *chunks* o processo de simplificação utilizado não é tão complexo como o do algoritmo de Chunked LOD adoptando-se aqui o método utilizado no algoritmo de Geomipmapping (ver **4.4**), isto é, em cada passo de simplificação são removidos um conjunto de linhas e de colunas mantendo-se no entanto uma estrutura da forma $2^n + 1$ em que $n \in [1, \rightarrow)$. Por exemplo, tal como está exemplificado na **Figura 4-29** podemos passar de um bloco 9×9 em que $n = 3$ para um bloco 5×5 em que $n = 2$.

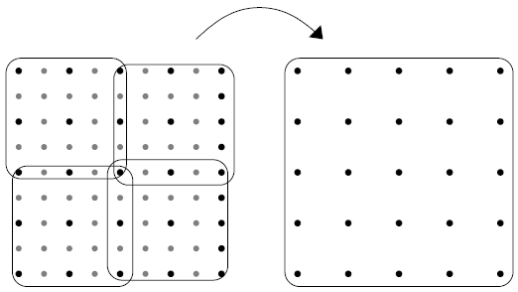


Figura 4-29: Método de simplificação adoptado [107].

4.7.2. A Escolha do Nível de Detalhe

A escolha do nível de detalhe é feita por intermédio da *quadtree* e envolve um processo recursivo que começa no nó raiz e só termina quando se atinge para cada porção de terreno o nível de detalhe mais apropriado. Este pode ser representado por mais ou menos blocos. Para isso, e com o intuito de determinar se o nível de detalhe de um *chunk*, em cada um dos nós da *quadtree* é aceitável ou não é utilizada uma métrica de erro para medir o erro associado a cada um deles. A métrica de erro utilizada é igual à do Geomipmapping (ver **4.4**) e envolve, tal como é referido em **4.4.3**, o cálculo da diferença de altura de todos os vértices que desaparecem na transição de um nível de detalhe para outro. Basicamente, envolve o cálculo da diferença entre a altura de um vértice num determinado nível e a altura que esse mesmo vértice teria se existisse no nível seguinte. Este cálculo é efectuado por *chunk* para cada um dos vértices que desaparecem. A partir dos valores obtidos, calcula-se depois o máximo que passa a ser utilizado como o erro do *chunk* propriamente dito. Formalmente, é calculado um erro δ_m como $max\{\delta_1, \delta_2, \dots, \delta_n\}$ em que n é o número de vértices que seriam removidos no próximo nível de detalhe. Para garantir que um *chunk* tem um δ_m maior do que um *chunk* com um nível de detalhe superior e que o erro cresce à medida que se sobe de nível na *quadtree*, o erro atribuído a cada *chunk*, δ_b, resulta do erro máximo δ_{bi} entre cada um dos seus i filhos mais o erro do próprio bloco δ_m, tal como está representado na **Equação 4-23**.

$\delta_m = \max(\delta_{v1}, \dots, \delta_{vn})$	δ_m: Máximo dos erros geométricos dos vértices que desaparecem na transição do nível l para o nível $l-1$ da *quadtree*.
$\delta_b = \begin{cases} 0, & \text{se nó folha} \\ \max(\delta_{b0}, \delta_{b1}, \delta_{b2}, \delta_{b3}) + \delta_m, & \text{caso contrário} \end{cases}$	$\max(\delta_{b0}, \delta_{b1}, \delta_{b2}, \delta_{b3})$: Máximo dos erros geométricos dos filhos.
	δ_m: Erro geométrico do bloco.

Equação 4-23: Cálculo do erro geométrico δ_b.

É a partir de δ_b que se determina em tempo de execução se um determinado nível de detalhe é o mais apropriado. Para isso, o erro δ_b é projectado para o espaço do ecrã obtendo-se um valor, ε, em *pixels*, que é posteriormente comparado com um limiar de

erro, τ, estabelecido pelo utilizador. Se ε for maior que τ, então é necessário um nível de detalhe maior, ou seja temos de descer um nível na *quadtree* e repetir o processo.

Na projecção do erro para coordenadas de ecrã, foram feitas neste algoritmo as mesmas simplificações que no algoritmo Geomipmapping (ver **4.4**), mais precisamente assume-se que a direcção de visualização é paralela ao plano horizontal, o que permite simplificar o cálculo da projecção. Por outro lado, e com o intuito de tornar o cálculo mais rápido este é orientado à distancia entre o *chunk* e o ponto de vista, pelo que se calcula um valor d_m, que não é mais do que a distância mínima a partir da qual um *chunk* deve ser utilizado dado um limiar de erro τ. Desta forma calcula-se apenas a distância, d, de um *chunk* ao ponto de vista de modo a comparar esse valor com d_m. Se for menor, então é necessário um nível de detalhe maior. Para isso, armazena-se em cada *chunk* o δ_b e calcula-se previamente um valor C através da **Equação 4-24** que é depois multiplicado em tempo de execução, por δ_b obtendo-se o d_m, tal como está representado na **Equação 4-25**. A principal vantagem deste método é o facto de se tornar possível a modificação do valor de τ em tempo execução sendo para isso necessário apenas recalcular a constante C.

$C = \frac{S}{2 \cdot \tau \cdot \left\lvert \tan\left(\frac{fov}{2}\right) \right\rvert}$	S: A altura do ecrã em *pixels*.
	τ: Representa o número máximo em *pixels* a partir do qual já não é permitida a mudança de nível.
	fov: Campo de visão.

Equação 4-24: Cálculo da constante C.

$d_m = \delta_b \cdot C$	δ_b: Calculado na **Equação 4-23**.
	C: Calculado na **Equação 4-24**.

Equação 4-25: Cálculo da mínima distância d_m.

4.7.3. Culling

Como a *quadtree* (ver **3.2.3.1**) é na prática uma estrutura de dados hierárquica de partição espacial (ver **3.2.3**) o *culling* neste algoritmo baseia-se na execução do *view frustum culling* (ver **3.3.2**) de forma hierárquica, isto é, à medida que se percorre de forma recursiva a *quadtree*. Basicamente se um nó não intersectar o *frustum*, então todos os nós filhos podem ser descartados o que permite obter ganhos bastante significativos ao nível do desempenho. O teste propriamente dito de cada um dos *chunks* de terreno é efectuado por intermédio de um *bounding volume* (ver **3.2.1**) que o representa o que torna os testes mais simples e sobretudo mais rápidos de executar. Desta forma o processo de *culling* ocorre durante a descida recursiva na *quadtree* o que permite executá-lo ao mesmo tempo que a selecção do nível de detalhe.

4.7.4. Falhas

Um dos principais problemas dos algoritmos de nível detalhe é, tal como foi referido em **3.4.4**, o aparecimento de falhas entre blocos adjacentes. Obviamente, esse problema

tem de ser corrigido e com esse intuito utiliza-se neste algoritmo a mesma estratégia empregue no algoritmo de Chunked LOD (ver **4.5**). Esta envolve o adicionar de uma malha geométrica vertical a volta de todo o *chunk* tal como está exemplificado na **Figura 4-30**. Esta saia poligonal ou *skirt* como costuma ser designada permite que as falhas entre dois *chunks* adjacentes com diferentes níveis de detalhe possam ser mascaradas resultando em artefactos menos visíveis ou na maioria dos casos praticamente indetectáveis [107].

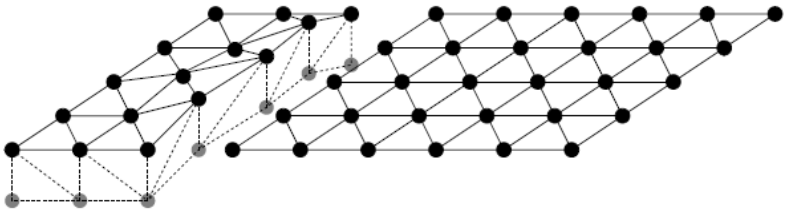


Figura 4-30: A utilização de *skirts* na correcção de falhas [107].

O principal problema na utilização de *skirts* está relacionado com a altura das mesmas. Estas devem ter a mínima altura necessária para que não se detectem falhas entre blocos adjacentes, independentemente do nível de detalhe desses blocos. Para resolver esse problema adoptou-se o valor do erro δ_b (ver **4.7.2**), já que este representa a maior diferença de altura entre todos os vértices que desaparecem de um nível para outro acumulando com o maior valor de erro dos seus filhos. Em contrapartida, a grande vantagem das *skirts* é serem mais simples de implementar e os resultados obtidos a nível visual serem bastante bons, nomeadamente quando o valor de erro considerado é baixo. Nessas situações são praticamente indetectáveis, o que as torna uma opção bastante efectiva.

4.7.5. Morphing

À medida que o ponto de vista se movimenta, o algoritmo de nível de detalhe empregue vai atribuindo mais ou menos detalhe a cada uma das regiões do terreno de acordo com a sua proximidade à câmara, o que se traduz na substituição de quatro blocos por um bloco, quando se pretende diminuir o nível de detalhe, ou o inverso, a substituição de um bloco por quatro blocos, quando se pretende aumentar o nível de detalhe. A nível visual causa um súbito aparecer ou desaparecer de geometria cujo efeito se denomina de *popping* (ver **3.4.4.2**). Os autores distinguem aqui dois tipos de *popping*: geométrico e de textura. O *popping* ao nível da textura ocorre porque o nível de detalhe da textura está neste algoritmo ligado ao nível de detalhe do *chunk*, uma vez que se armazena em cada nó da *quadtree* o bloco de geometria e a textura associada. A solução para os dois problemas é a mesma: *morphing*. A nível geométrico este método é tipicamente designado de Geomorphing (ver **3.4.4.2**) e implica mover gradualmente os vértices que são afectados por uma mudança de nível de detalhe da posição em que se encontram para a posição que teriam no próximo nível de detalhe de modo a que esta transição entre níveis se torne menos perceptível. Note-se que em cada transição apenas uma parte dos vértices são afectados, por exemplo, se considerarmos um *chunk*, os seus filhos, vão ter o

dobro dos vértices em cada direcção, no entanto os vértices que têm em comum ocupam a mesma posição [107]. Deste modo é necessário efectuar apenas o *morphing* desses vértices na transição entre níveis de detalhe ou seja mover lentamente os vértices extra da posição projectada no nível de detalhe seguinte para a sua posição original. Para efectuar o *morphing* propriamente dito é necessário armazenar por vértice um valor que representa o deslocamento a efectuar para se atingir a posição no próximo nível de detalhe. Por outro lado é também utilizado um factor f com um valor entre 0 e 1 que permite efectuar a interpolação entre a posição original e a nova posição de cada um dos vértices. O cálculo do factor f é efectuado por intermédio da **Equação 4-26** sendo utilizado no cálculo na nova posição do vértice tal como está representado na **Equação 4-27**.

$f = 1.0 - \frac{d - d_m}{d_{mp} - d_m}$	d: Distância do *chunk* ao ponto de vista.
	d_m: Distância mínima, calculada por intermédio da **Equação 4-25**.
	d_{mp}: Distância mínima do *chunk* pai, calculada por intermédio da **Equação 4-25**.

Equação 4-26: Cálculo do factor de *morph* f.

$\{x', y', z'\} = \{x, y, z\} + f \cdot \{0, d, 0\}$	$\{x, y, z\}$: Coordenadas do vértice original.
	f: Valor entre 0 e 1 calculado na **Equação 4-27**.
	d: Distância a percorrer até ao nível de detalhe seguinte.

Equação 4-27: Cálculo da posição do vértice de acordo com factor de *morphing*.

No que diz respeito ao *morphing* ao nível das texturas, este envolve a utilização de duas texturas com diferentes níveis de detalhe e o *blending* das mesmas por intermédio de uma interpolação linear e de um factor f. O cálculo do valor final da cor de cada um dos *pixels* assim obtido é apresentado na **Equação 4-28**.

$c = t_0 \cdot f + t_1.(1 - f)$	t_0: Primeira textura.
	t_1: Segunda textura.
	f: Valor entre 0 e 1 calculado na **Equação 4-27**.

Equação 4-28: Cálculo da cor do *pixel* por interpolação linear.

4.7.6. Geração de Detalhe

Uma das características mais notórias deste algoritmo é o facto de permitir a adição de detalhe em tempo de execução. Para que isso seja possível é necessário aumentar a densidade do *height field* adicionando novos valores entre cada linha e coluna tal como está representado na **Figura 4-31**.

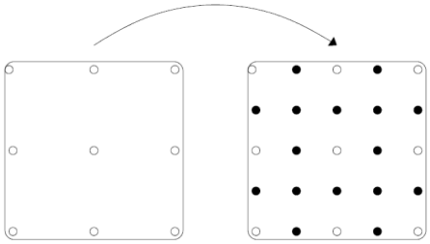


Figura 4-31: O adicionar de detalhe a um *height field* [107].

O principal objectivo é, por um lado, adicionar detalhe ao terreno e, por outro, fazê-lo de tal modo que cada um dos valores adicionados esteja relacionado com os valores de elevação que o rodeiam pelo que é necessário efectuar alguma forma de interpolação. O processo tal como é descrito pelos autores está representado na **Figura 4-32** envolve uma interpolação de segunda ordem[5] combinada com uma perturbação de cada um dos valores de elevação de modo a criar mais detalhe. O processo de interpolação e de perturbação é separado de modo a que o detalhe adicionado por perturbação de cada um dos valores de elevação não seja também ele interpolado. Assim, numa primeira fase efectua-se de forma recursiva a interpolação adicionando-se depois detalhe adicional pela perturbação dos valores de elevação assim adicionados.

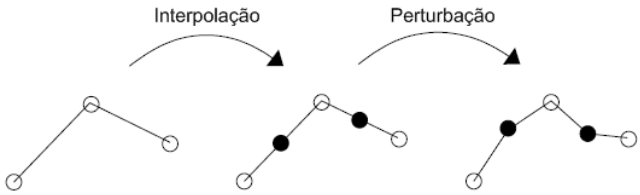


Figura 4-32: Interpolação seguida de uma perturbação dos valores de elevação [107].

A interpolação propriamente dita é realizada por intermédio do método de subdivisão de superfícies descrito em [100] por Kobbelt. Este método permite que as zonas do terreno onde existem transições abrutas dos valores de elevação se tornem mais suaves e, por conseguinte, mais naturais, algo que não seria possível se fosse utilizada uma simples interpolação linear.

A perturbação dos valores de elevação adicionados por interpolação é feita por intermédio de fractais (ver **A.1**). Estes permitem gerar detalhe pela adição sucessiva de valores obtidos a partir de uma função base, que no caso deste algoritmo foi o Perlin Noise (ver **A.4.5.1**).

Para que seja possível adicionar mais detalhe em tempo de execução é necessário adicionar também mais nós à *quadtree*. Basicamente, para cada nó são adicionados quatro nós filhos tal como está representado na **Figura 4-33** pela cópia dos vértices do

[5] Um processo de interpolação implica encontrar uma função que intersecte um conjunto de pontos. Considere-se m pontos (x_k, y_k) em que $x_k \in \mathbb{R}^n, y_k \in \mathbb{R}^n$ e os x_k são distintos. O objectivo é construir uma função $f: \mathbb{R}^n \to \mathbb{R}$ tal que $y_k = f(x_k)$. Existem diversas formas de interpolação sendo a mais comum a interpolação linear que utiliza como função de interpolação um polinómio linear. Em contrapartida uma interpolação de segunda ordem ou quadrática utiliza um polinómio quadrático.

pai (a preto) e pelo adicionar de novos vértices (a branco). No entanto esses nós terão de ser necessariamente dinâmicos, isto é, devem ser adicionados apenas quando a proximidade do ponto de vista ao terreno justificar mais detalhe. Assim, tem-se logo à partida de distinguir dois tipos de nós: estáticos e dinâmicos. Os nós estáticos correspondem aqueles que foram carregados directamente a partir da fonte de elevação e os dinâmicos aos que são adicionados apenas para satisfazer a necessidade de mais detalhe numa determinada zona do terreno face à proximidade do ponto vista. Desta forma, para manter os requisitos de memória baixos, estes terão de ser removidos a partir do momento em que se determine não serem mais necessários.

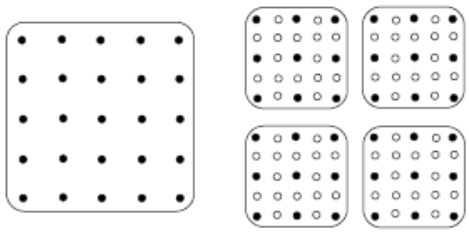


Figura 4-33: Criação de quatro novos nós a partir do nó pai [107].

É importante referir, no entanto, que o adicionar de nós levanta um problema relacionado com o erro associado a cada um deles. De acordo com a **Equação 4-23**, o erro atribuído a cada um dos nós folha é zero, algo que tem de ser alterado já que se pretende adicionar mais nós e o erro está sempre dependente dos filhos. Como em teoria o número de filhos a adicionar pode ser infinito, foi necessário estimar numa fase de pré-processamento o erro de cada um dos nós folha de modo a que este não seja zero. Para isso, adicionou-se de forma aleatória detalhe e utilizou-se o erro máximo encontrado como o valor aproximado do erro nos nós folha. A partir daí cada um dos nós dinâmicos adicionados tem muito simplesmente metade do erro do nó pai o que permite adicionar recursivamente todo o detalhe que se pretenda.

4.8. GPU Terrain Rendering

O algoritmo de GPU Terrain Rendering é descrito por Harald Vistnes em [207] e tem como principal característica o facto de tirar proveito de uma das novas funcionalidades do *Shader Model* 3.0. Mais precisamente utiliza o *vertex texture fetch* que permite aceder a texturas no *vertex shader* pelo que não implica enviar para a placa gráfica os valores de elevação de forma explícita pois podem ser armazenados em texturas. Opta, no entanto, por uma abordagem tradicional de divisão do terreno em blocos tal como é feito no algoritmo de Geomipmapping (ver **4.4**) e no algoritmo de Chunked LOD (ver **4.5**). Aliás, é do algoritmo de Chunked LOD que retira muitos dos conceitos principalmente o método de tratamento de falhas (ver **4.8.4**) e o modo como é utilizada a estrutura da *quadtree* na construção dos diferentes níveis de detalhe (ver **4.8.2**).

4.8.1. Representação do Terreno

O terreno é neste algoritmo divido num conjunto de blocos em que os vértices estão organizados numa grelha regular de$(2^n + 1) \times (2^n + 1)$ em que $n \in [1, \rightarrow)$ tal como no Geomipmapping (ver **4.4**). Por outro lado, de uma maneira muito semelhante ao que é feito no Chunked LOD (ver **4.5**) também aqui para efectuar a partição espacial do terreno se utiliza uma *quadtree* (ver **3.2.3.1**) que influencia directamente a forma como é feita a selecção do nível de detalhe mais apropriado. Neste caso, o número de vértices dos blocos não é sempre o mesmo, muito embora o número de vértices se mantenha, isto ao contrário do Geomipmapping em que a diminuição do nível de detalhe é feita pela diminuição do número de vértices em cada bloco. Neste algoritmo a diminuição do nível de detalhe implica a substituição de quatro blocos por um único que corresponde à mesma área desses quatro blocos mantendo o mesmo número de vértices já que o espaçamento entre eles aumenta. Efectivamente, isto significa que a diminuição/aumento do nível de detalhe corresponde na prática ao modo como é particionado o espaço numa *quadtree*. Por exemplo, se o número de vértices de cada bloco for 5 a diminuição do nível de detalhe implica a substituição de 4 blocos 5×5 por 1 bloco 5×5 que ocupa o espaço correspondente aos quatro blocos filhos. Esta abordagem é algo semelhante ao que é feito no algoritmo de Chunked LOD que também aumenta/diminui o nível de detalhe da mesma forma, só que nesse caso cada um dos nós contém uma *mesh* pré-assemblada fruto de uma simplificação numa fase de pré-processamento pelo que, o processo de simplificação utilizado não é o mesmo. Assim, esta perspectiva é claramente diferente e mais importante traz algumas vantagens pois permite logo à partida diminuir o número de blocos enviados para *rendering*, já que, e de acordo com a métrica de erro utilizada (ver **4.8.2**), os blocos mais distantes passam a representar uma porção muito maior do terreno e como tal reduz-se para esses casos o número de chamadas que é necessário efectuar à primitiva gráfica (ver **3.1**). Nesta situação, no caso do algoritmo de Geomipmapping, seria necessário efectuar 4 chamadas e, muito embora o número de vértices de cada um dos blocos possa ser inferior ao original (pela diminuição do nível de detalhe do próprio bloco que é característico desse algoritmo), as 4 chamadas teriam sempre de ser executadas. Pelo contrário nesta abordagem é necessário efectuar apenas uma chamada em que o número de vértices definidos para o bloco vai ocupar o espaço correspondente ao nó pai na *quadtree*. Na **Figura 4-34** é ilustrado este conceito pela representação de um terreno de 33×33 vértices e de dois blocos correspondentes a diferentes níveis da *quadtree*. Se observarmos a figura em detalhe, é possível chegar a uma conclusão muito importante: qualquer bloco de terreno pode ser descrito por um factor de escala e uma translação. A principal implicação deste facto é que não é necessário armazenar explicitamente todos os vértices que constituem o terreno. Muito pelo contrário, basta armazenar apenas os vértices que constituem um bloco (neste caso os 25 vértices correspondentes a um bloco 5×5) com valores em x e z definidos na forma canónica (entre 0 e 1). Isto só é possível porque os valores de elevação (os únicos que efectivamente diferenciavam cada um dos vértices) podem ser agora obtidos directamente a partir de uma textura pelo que o valor da coordenada y passa a ser obtido directamente no *vertex shader* associado. Da mesma forma também no *vertex shader* se passa a efectuar a translação e a aplicar o factor de escala correspondente a cada um dos blocos que se pretende representar. Formalmente, e usando a figura como ponto de referência,

todos os elementos do *height field* são indexados pelos parâmetros u e v em que $0 \leq u \leq 1$ e $0 \leq v \leq 1$. Da mesma forma, a posição horizontal dos vértices no bloco é determinada pelos parâmetros s e t em que $0 \leq s \leq 1$ e $0 \leq t \leq 1$. Assim para um determinado bloco o factor de escala e a translação correspondem a $\{s,t\} \Rightarrow \{u,v\}$, isto é à transformação do par $\{s,t\}$ em $\{u,v\}$ em que o novo par $\{u,v\}$ é usado para obter o valor de elevação h em que $0 \leq h \leq 1$. Como é óbvio, esta abordagem implica multiplicar cada um dos vértices resultantes por uma matriz de transformação de modo a obter os valores de cada vértice em coordenadas globais.

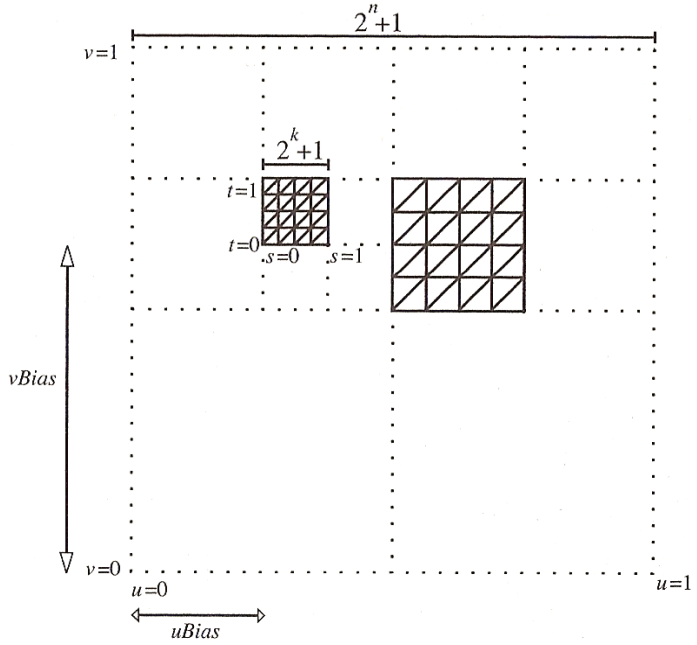


Figura 4-34: O factor de escala e de translação na representação dos blocos [207].

4.8.2. A Escolha do Nível de Detalhe

Tal como foi referido em **4.8.1**, o nível de detalhe é controlado neste algoritmo pela substituição de quatro blocos por um único bloco ou, no caso oposto, o transformar de um único bloco em quatro blocos, o que equivale à partição espacial recursiva que a *quadtree* (ver **3.2.3.1**) proporciona. Assim, se começarmos com um único bloco que cobre todo o terreno e recursivamente efectuarmos a sua divisão em quatro blocos filhos até atingir a resolução do *height field* (ver **3**), estamos a criar efectivamente uma *quadtree* de blocos [207]. Este é basicamente o princípio subjacente ao algoritmo de nível de detalhe utilizado, isto é, é feita uma avaliação recursiva de cada bloco para decidir se vamos enviá-lo para *rendering* ou dividi-lo. O autor utiliza neste caso um critério de avaliação simples baseado na distância, isto é, quanto mais perto da câmara, maior o número de subdivisões efectuadas para cada um dos blocos, ou seja mais detalhe. O critério de avaliação baseado em distância é dado pelo teste representado na **Equação 4-29**. Se este teste for bem sucedido, o bloco é enviado para *rendering*, caso contrário a

recursão continua pelo que o aumento da constante C significa neste caso mais blocos a serem subdivididos e consequentemente o *rendering* de mais triângulos.

$\frac{l}{d} < C$	l: Distância do centro do bloco à câmara.
	d: O espaço ocupado por um triângulo.
	C: Constante que controla a qualidade do terreno.

Equação 4-29: O critério de avaliação do nível de detalhe.

Na **Figura 4-35** está representado o critério de avaliação do nível de detalhe sendo visível a sua dependência dos dois factores que entram para a equação, isto é o espaço ocupado por um triângulo e a distância do centro do bloco à câmara.

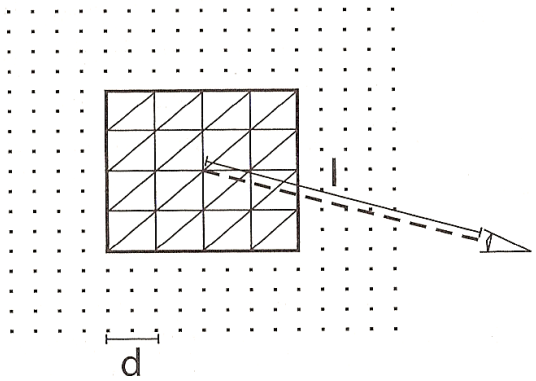


Figura 4-35: Representação do critério de avaliação do nível de detalhe [207].

4.8.3. Culling

À semelhança da grande maioria dos algoritmos descritos também aqui se efectua o *view frustum culling* (ver **3.3.2**) utilizando como estrutura de suporte a *quadtree* (ver **3.2.3.1**). Mais precisamente é feita a intersecção de uma *axis aligned bounding box* (ver **3.2.1**) construída para cada um dos blocos de terreno com o *view frustum* de modo a que seja possível enviar para *rendering* apenas os blocos que estão visíveis ou parcialmente visíveis.

4.8.4. Falhas

O aparecimento de falhas (ver **3.4.4.1**) entre blocos de diferentes níveis de detalhe é um problema que ocorre em todos os algoritmos que utilizam técnicas de nível de detalhe e este algoritmo não é excepção. Neste caso o problema é um pouco mais grave pois os blocos não têm a mesma dimensão tornando mais complicado efectuar adaptações geométricas. Deste modo, para simplificar o autor adoptou a mesma técnica de tratamento de falhas utilizada no Chunked LOD (ver **4.5**), ou seja, as *skirts* verticais onde, tal como está representado na **Figura 4-36**, cada bloco é estendido com uma *skirt* ou saia

poligonal. Assim, quando se juntam dois blocos com diferentes níveis de detalhe, qualquer falha potencial que possa ocorrer entre os dois é preenchida pela geometria da *skirt* resolvendo-se deste modo este problema. A altura da *skirt* é igual á elevação do vértice multiplicada por -1, o que torna a sua altura negativa e por isso com um valor inferior à altura de qualquer vértice de um bloco adjacente. Neste caso pressupõe-se que o terreno não tem pontos com elevação negativa.

Figura 4-36: Cada bloco é estendido com uma *skirt* vertical [207].

4.9. Geometry Clipmaps

Os Geometry Clipmaps introduzidos por Frank Losasso e Huges Hoppe em 2004 [120] são uma estrutura de nível de detalhe inovadora que deu origem a uma nova classe de algoritmos. O algoritmo, também designado de Geoclipmapping, procura enviar, tal como os algoritmos pertencentes à classe *Tiled Blocks* (ver **4.1**), o maior número possível de triângulos para a placa gráfica numa perspectiva de obter o melhor desempenho possível tendo em conta as capacidades do *hardware* actual. No entanto, ao contrário dos algoritmos pertencentes a essa classe não trabalha sobre secções do terreno encarando o terreno com um todo. Para isso recorre a um conceito já existente na gestão do nível de detalhe para texturas: o *texture clipmap* [191]. Os *texture clipmaps* permitem o *caching* de um subconjunto de uma pirâmide de *mipmaps* (ver **Figura 4-17**) de acordo com o ponto de vista corrente. Em cada nível da pirâmide o subconjunto da textura correspondente, o *clipmap*, funciona assim como que uma janela sobre a totalidade da textura que esse nível representa. Esta estratégia em conjunto com uma rápida actualização de cada um dos *clipmaps* permite uma exploração de imagens de grandes dimensões. Analogamente os *geometry clipmaps* permitem o *caching* de um terreno num conjunto de grelhas regulares concêntricas centradas no ponto de observação (ver **Figura 4-37**).

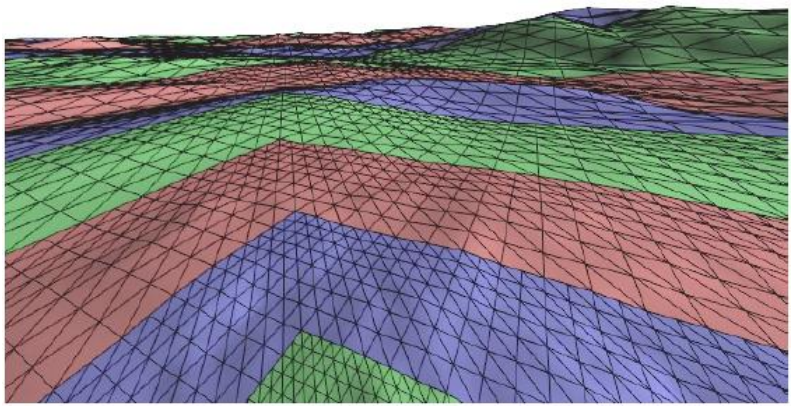


Figura 4-37: *Rendering* de um terreno com *Geometry Clipmaps* [120].

Mais especificamente, o terreno é representado como uma pirâmide de grelhas regulares em que o *clipmap* é um subconjunto dessa grelha em cada um dos níveis. Cada uma das grelhas é por sua vez uma versão filtrada do terreno em sucessivas potências de dois. Um dos principais benefícios desta aproximação é permitir englobar sobre uma mesma *framework* não só a gestão do nível de detalhe geométrico mas também o tratamento pelo mesmo princípio das texturas. Assim, segundo [120] os *Geometry Clipmaps* têm uma série de vantagens em relação às outras aproximações, nomeadamente:

- Simplicidade, por não implicarem percorrer a cada *frame* uma estrutura para encontrar o nível detalhe mais apropriado para cada uma das zonas do terreno.
- Eficiência no *rendering*, pela utilização de tiras de triângulos indexadas (ver **3.1.3**).
- Continuidade visual entre *clipmaps* de diferente resolução, pela adição de uma região de transição (ver **4.9.4**) que resolve o problema da continuidade espacial e temporal ao nível da geometria e da textura.
- *Rendering* constante, pois o nível de detalhe não está dependente da complexidade de cada zona do terreno.
- Regulação de complexidade, já que mesmo com um *clipmap* de tamanho fixo é possível reduzir o tamanho das regiões enviadas para *rendering* para regular a taxa de actualização.
- Degradação controlada, por exemplo, em situações em que o utilizador se move rapidamente pode não ser possível actualizar todos os níveis. Nesse caso o objectivo é actualizar o maior número possível dentro de um determinado limiar. Como o *rendering* se processa do nível de detalhe mais baixo para o mais alto perde-se nesse caso alguma fidelidade na representação das zonas mais próximas do ponto de observação, mas mantém-se a fluidez.
- *Shading* da superfície, que é realizado com mapas de normais calculadas directamente a partir da geometria.
- Compressão, como apenas o *clipmap* necessita de ser concretizado em vértices, o resto da pirâmide que representa o terreno pode ser comprimida.
- Síntese, pois permite a especificação de detalhe para além do definido no mapa de elevações subjacente.

4.9.1. Representação do terreno

Os *geoclipmaps* tiram proveito do facto de um *height field* poder ser armazenado sobre a forma de uma imagem, o *height map* (ver **2.3.1**). Nesta perspectiva a teoria por detrás dos *geoclipmaps* extrapola a teoria dos *texture clipmaps* [191] para o domínio da geometria. Um *texture clipmap* permite que uma textura de grandes dimensões seja usada numa cena sem que para isso seja necessário armazená-la totalmente na memória. Para isso, é utilizada uma janela sobre os dados da textura que é incrementalmente actualizada de acordo com a área visível e cuja posição depende apenas do movimento efectuado pelo utilizador [9]. Seguindo a mesma perspectiva, mas a nível geométrico, no *geoclipmapping* o terreno é representado por uma pirâmide de grelhas regulares concêntricas em m níveis de progressivamente maior resolução. A hierarquia completa corresponde a uma sequência de grelhas previamente filtradas em sucessivas potências de dois em que o nível m representa o terreno na máxima resolução. Assim, em cada nível, desta pirâmide é definida uma "janela" de $n \times n$ de vértices, o *clipmap*, que é efectivamente um subconjunto da grelha regular associada ao nível corrente (ver **Figura 4-38**). Desta forma, para cada um dos níveis, o número de amostras obtidas na fonte de elevação subjacente é exactamente o mesmo, por exemplo $n = 31$ em todos os níveis. Varia no entanto a frequência de amostragem que aumenta do nível menos detalhado, 0 para o mais detalhado $l - 1$. Por sua vez, a distância entre valores na grelha, g_l é definida em função do nível corrente tal que $g_l = 2^{-l}$. Consequentemente, o tamanho da grelha é definido como $ng_l \times ng_l$. O principal objectivo deste algoritmo é garantir que as zonas mais detalhadas são as que envolvem o observador. Assim, os níveis menos detalhados são anéis que envolvem o próximo nível dando desta forma origem a uma pirâmide de grelhas regulares concêntricas como a representada na figura. Ainda em relação ao tamanho do *clipmap*, este deve ser um número de vértices ímpar para que o perímetro possa encaixar com a grelha do nível menos detalhado que o envolve, pelo que é normalmente da forma $2^k - 1$ (255 por exemplo).

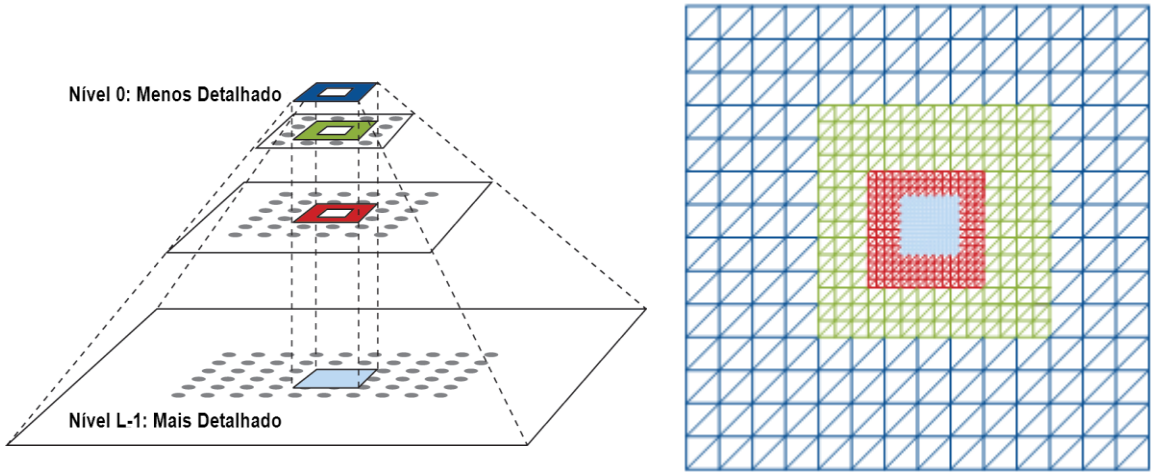


Figura 4-38: Terreno como uma pirâmide de *mipmaps* [11].

Em cada nível l, o *clipmap* é dividido num conjunto de regiões rectangulares representadas na **Figura 4-39**. Começando pela *clip region*, esta corresponde à grelha $n \times n$ armazenada em cada um dos níveis. A *active region* é por sua vez uma grelha $n \times n$ centrada no observador. Assim, sempre que existir movimento a *clip region* de cada nível tem de ser actualizada de modo a corresponder à *active region* desejada. Se por questões de carga não for possível, a *active region* pode ser cortada de acordo com a *clip region* disponível tal como está representado na figura. Finalmente o "anel" (a verde na figura) estabelece a zona de *rendering* do nível, a $render\_region(l)$, estando delimitado por $active\_region(l)$ e $active\_region(l+1)$ respectivamente. No nível mais detalhado, m, é efectuado o *rendering* de uma grelha regular completa estando por isso por definição vazia a $active\_region(m+1)$. Losasso e Hoppe especificaram quatro regras em relação a estas regiões [120][22]:

1. $clip\_region(l+1) \subseteq clip\_region(l) \ominus 1$, em que $\ominus$ representa erosão por uma distância escalar, isto é, na $clip\_region(l)$ a frequência de amostragem diminui pelo que é necessária pelo menos uma unidade de distância, g_l em todos os lados da grelha entre as *clip regions*.
2. $active\_region(l) \subseteq clip\_region(l)$, já que os dados enviados para *rendering* são um subconjunto dos dados disponíveis em cada *clipmap*. Garante-se desta forma que só é efectuado o *rendering* dos dados disponíveis.
3. O perímetro da $active\_region(l)$ deve estar definido sobre um número par de vértices, pois estes vão ser utilizados no nível seguinte de menor detalhe, $l-1$, que têm duas vezes mais espaço entre os vértices.
4. $active\_region(l+1) \subseteq active\_region(l) \ominus 2$, já que a *render region* tem de ter pelo menos duas unidades de distância, $2g_l$ na grelha para permitir uma transição continua entre níveis evitando as falhas (ver **4.9.4**).

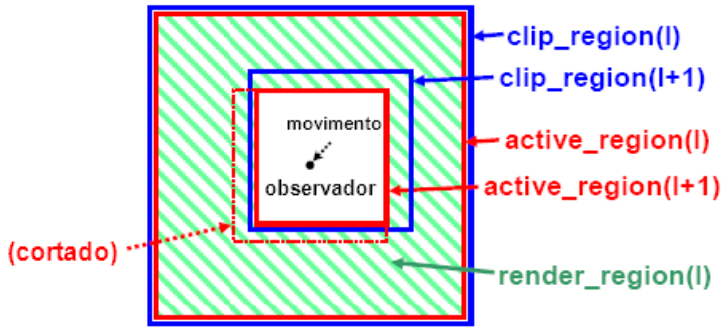


Figura 4-39: Regiões definidas dentro de cada *clipmap* [120].

A $render\_region(l)$ é dividida em quatro blocos rectangulares, tal como está representado na **Figura 4-40** (note-se que é apenas para efeitos da ilustração, na realidade são necessárias mais tiras de triângulos do que as representadas). Em cada uma dessas regiões são utilizadas tiras de triângulos indexadas (ver **3.1.3**), representadas a verde na figura para efectuar a triangulação, algo que vai implicar no máximo quatro chamadas por *clipmap* à primitiva correspondente. Isto porque esta divisão em regiões vai ser utilizada no *view frustum culling* (ver **3.3.2**) tal como é descrito em **4.9.3** podendo por isso serem removidas no decorrer desse processo uma ou mais regiões.

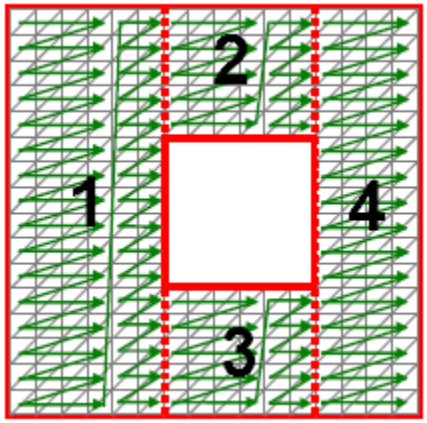


Figura 4-40: As regiões de um *clipmap* trianguladas com tiras de triângulos [120].

4.9.2. Actualização dos Clipmaps

À medida que o observador se movimenta no terreno é necessário actualizar em cada nível a *active region* correspondente, centrando-a na posição (x, z) corrente. Para isso as *clip regions* têm também elas de ser actualizadas de modo a corresponderem às *active regions* pretendidas. Neste algoritmo em cada um dos níveis os dados de elevação são estimados a partir do nível anterior de menor detalhe, pelo que é necessário armazenar apenas o nível de menor resolução. Como é óbvio isto é algo que implica um método adicional para aproximar em cada nível a resolução efectiva no *height map* original. Para isso é adicionado um sinal residual (*residual*) que pode ser proveniente de duas fontes de dados distintas: descompressão de dados reais e síntese. Os dados comprimidos são construídos numa fase de pré-processamento, num processo descrito em detalhe em **4.9.5.** Em contrapartida, para níveis de maior detalhe, possivelmente acima da resolução do *height map* original, os dados são sintetizados. Tal como já foi referido em **4.9.1**, esta actualização das *clip regions* pode não vir a ser efectuada na sua totalidade pelo que nessa situação a *active region* é cortada pela *clip region* obtida. Por outro lado, tendo em conta que o *rendering* se realiza do nível menos detalhado para o mais detalhado, dependendo da velocidade com que o utilizador se move no terreno alguns dos níveis mais detalhados podem vir a ser ignorados se for ultrapassado um determinado limiar de processamento. Outro factor que também influencia o número de níveis considerados é a altura do ponto de vista. A partir de uma determinada altura os níveis mais detalhados deixam de contribuir de modo relevante para a cena podendo inclusivamente causar *aliasing*, por isso, nessa situação não se consideram esses níveis. Mais precisamente, segundo [120], para cada nível l a *active region* não é considerada se a altura do ponto de vista for superior a $(0.4)ng_l$. Independentemente do número de níveis considerados, é necessário actualizar para cada nível a lista de vértices associada. Assim, de forma a permitir uma actualização incremental, a lista é acedida de forma toroidal, tal como está ilustrado na **Figura 4-41**. Desta forma não é necessário actualizar a lista de vértices completa, mas apenas a nova região em forma de "L" resultante do movimento efectuado no terreno. Na figura apenas o topo ("T"), o canto ("C") e o lado direito ("D") necessitam

de ser actualizados (os três formam a região em forma de "L" resultante do movimento). Substituem por isso os dados da região oposta também em forma de "L".

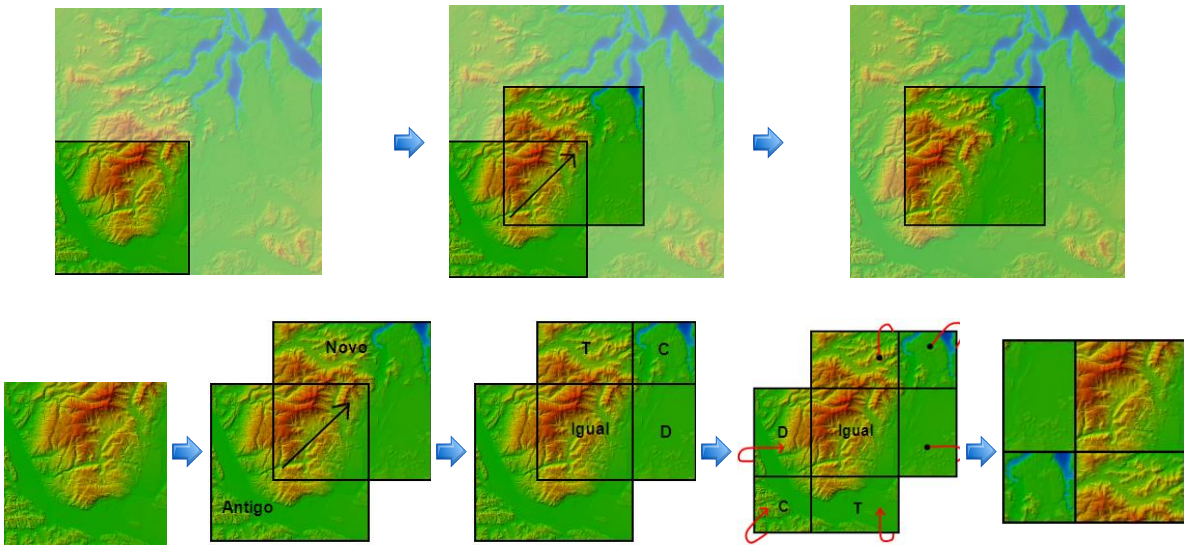


Figura 4-41: Actualização toroidal do *clipmap*.

Especificando mais precisamente todo processo de actualização do *clipmap*, são executados por *frame* os seguintes passos [120][9]:

1. Determinar as *active regions*. Em cada nível l com um espaçamento entre valores na grelha $g_l = 2^{-l}$ a *active region* é um quadrado de tamanho $ng_l \times ng_l$ centrado na posição (x, z) corrente do observador.
2. Actualizar o *geometry clipmap*. Tal como já foi referido não é necessário actualizar para cada nível toda a lista de vértices da *active region* correspondente. Pelo contrário, graças ao acesso toroidal da lista de vértices respectiva basta actualizar a região em forma de "L" resultante do movimento efectuado pelo utilizador. Os dados utilizados nesta actualização podem resultar da descompressão de dados reais ou da sintetização dos mesmos para níveis de maior detalhe (ver **4.9.5**).
3. Cortar as *active regions* em função dos dados disponíveis nas *clip regions* respectivas. Este corte pode ser efectuado para aumentar a velocidade de processamento ou porque o ponto de vista está demasiado alto não se justificando a utilização dos níveis de maior detalhe.
4. Efectuar o *rendering* propriamente dito. Para cada $active\_region(l)$ é obtida a respectiva $render\_region(l)$ que é depois dividida em quatro regiões (ver **Figura 4-40**) para efectuar o *view frustum culling* (ver **4.9.3**). Após o *culling* as regiões resultantes são enviadas para a placa gráfica. A $render\_region(l)$ resulta de $active\_region(l) - active\_region(l+1)$, ou seja, da região entre o nível de detalhe mais baixo e o nível de detalhe mais alto.

4.9.3. Culling

O *view frustum culling* é efectuado neste algoritmo pela divisão das *render regions* em quatro blocos rectangulares tal como está representado na **Figura 4-40**. Para cada um destes blocos é constituída uma *axis aligned bounding box* (ver **3.2.1**) que é depois intersectada com o *frustum* de modo a identificar se o bloco é ou não visível para o ponto de vista corrente. Tal como nos outros algoritmos, esta técnica é essencial para reduzir o número de triângulos enviados para *rendering* podendo reduzir a carga em cerca de três vezes para um ângulo de visão de 90° [120]. O efeito do *view frustum culling* na estrutura concêntrica de uma pirâmide de *clipmaps* está representado na **Figura 4-42** onde é bem visível a diminuição no número de blocos enviados para *rendering* em cada *clipmap*.

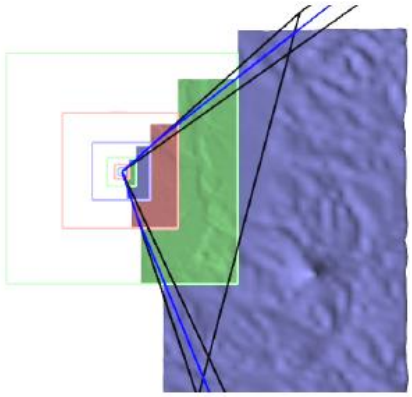


Figura 4-42: *View Frustum Culling* com *Geometry Clipmaps* [120].

4.9.4. Falhas

Tal como se pode verificar na **Figura 4-38** existe uma diferença de vértices entre *render regions* adjacentes, mais especificamente nos vértices ímpares o que pode dar origem a falhas. Para resolver esse problema é utilizada uma aproximação equivalente à do Geomorphing descrita em **3.4.4.2**, definindo-se para isso em cada *render region* uma região de transição (a verde na **Figura 4-43**) onde é efectuado o *morphing* da geometria, ou seja nessa região os vértices transitam do valor de elevação corrente para o próximo, evitando desta forma também o efeito de *popping*. Os autores sugerem para essa região um tamanho w de $n/10$ que, no caso da $active\_region(l+1)$ estar demasiado próxima, é calculado como $w = \min(n/10\,, \text{min\_width}(l))$ em que o $\text{min\_width}(l)$ é pelo menos 2 tal como está representado na figura. O *morphing* por usa vez é feito em função da posição do observador, ou seja, relacionando as coordenadas (x, z) de cada vértice na grelha com a posição do observador (v_x, v_z). Este método, muito embora elimine efectivamente as falhas, dá origem a *t-junctions* (ver **3.4.4.1**) pelo que são utilizados triângulos degenerados no perímetro de cada *render region* para ligar os diferentes níveis.

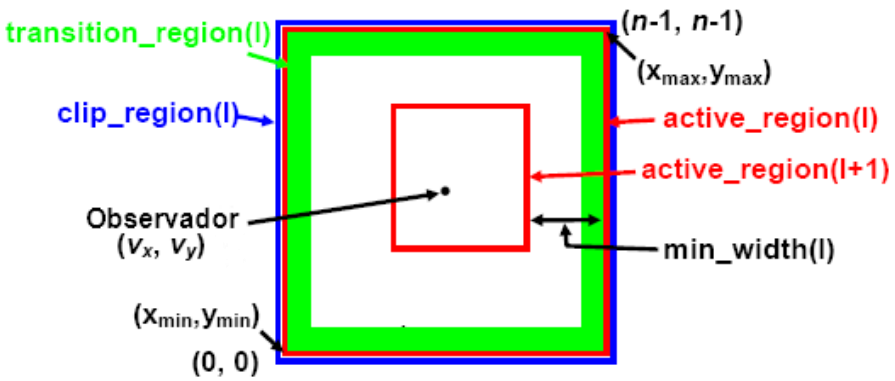


Figura 4-43: Regiões de transição [120].

Na concretização do Geomorphing, os autores sugerem que o *morphing* dos vértices no perímetro da *render region* seja efectuado no *vertex shader*. Assim, para um vértice (x, y, z) em que a altura no próximo nível menos detalhado é y_c, w o tamanho da região de transição, $(v_x^{'}, v_z^{'})$ a posição do ponto de vista no nível e (x_{min}, z_{min}) e (x_{max}, z_{max}) o tamanho da *active region* o valor de elevação $y^{'}$ que resulta do *morphing* efectuado na região de transição é calculado pela **Equação 4-30**.

$y^{'} = (1-\alpha)y + \alpha y_c$	α: Resulta de $\max(\alpha_x, \alpha_z)$ em que α_x e α_z são calculados pela **Equação 4-31** e **Equação 4-32** respectivamente.
	y: Elevação no nível corrente.
	y_c: Elevação no nível seguinte de menor detalhe $(l-1)$.

Equação 4-30: Cálculo do valor de elevação na zona de transição.

$\alpha_x = \text{clamp}\left(\frac{\lvert x - v_x^{'}\rvert - \left(\frac{x_{max} - x_{min}}{2} - w - 1\right)}{w}, 0{,}1\right)$	clamp: Constrange o resultado ao intervalo [0,1].
	x: Coordenada x do vértice na grelha.
	$v_x^{'}$: Coordenada x da posição do ponto de vista no *clipmap*.
	x_{min} : Mínimo em x na *active region*.
	x_{max} : Máximo em x na *active region*.
	w: Tamanho da região de transição.

Equação 4-31: Cálculo do parâmetro de *blend* α_x.

$\alpha_z = \text{clamp}\left(\frac{\left\|z - v_z'\right\| - \left(\frac{z_{max} - z_{min}}{2} - w - 1\right)}{w}, 0,1\right)$	clamp: Constrange o resultado ao intervalo [0,1].
	z: Coordenada z do vértice na grelha.
	v_z': Coordenada z da posição do ponto de vista no *clipmap*.
	z_{min} : Mínimo em z na *active region*.
	z_{max} : Mínimo em z na *active region*.
	w: Tamanho da região de transição.

Equação 4-32: Cálculo do parâmetro de *blend* α_z.

4.9.5. Compressão e Síntese

Tal como já foi referido, os dados de elevação em cada um dos níveis podem ser obtidos a partir de dados comprimidos ou sintetizados. Os dados comprimidos são produzidos numa fase de pré-processamento onde é construída uma pirâmide de mipmaps, $T_1 \ldots T_m$, que representa o terreno em sucessivos níveis de progressivamente maior resolução (ver **Figura 4-44**). Esta pirâmide é construída pela aplicação sucessiva de um filtro linear $T_{l-1} = D(T_l)$ aos dados de elevação de maior detalhe, T_m. O próximo passo consiste em estimar cada nível da pirâmide T_l a partir do nível anterior de menor detalhe, T_{l-1}, através de um processo de subdivisão por interpolação $U(T_{l-1})$ descrito em [100]. É a diferença entre os dados originais de elevação no nível T_l corrente e o estimado a partir do nível anterior T_{l-1} para esse nível, o *residual*, que é calculado tal que $R_l = T_l - U(T_{l-1})$. Por fim a diferença entre os dois é comprimida, $\widetilde{R_l} = \text{compress}(R_l)$ com o *codec* PTC de compressão de imagem descrito em [123]. Desta forma, a grande vantagem deste método é não ser necessário manter em memória todo o *height map* mas apenas os *residuals*, o que segundo Lossaso e Hoppe [120] permitiu a compressão de um *height map* dos Estados Unidos da América com 30 metros de resolução horizontal, 1 metro de resolução vertical e com uma dimensão de 216.000 por 93.600 vértices que ocupava cerca de 40 Gbytes de espaço em apenas 350 Mbytes, o que significa uma factor de compressão de 115:1.

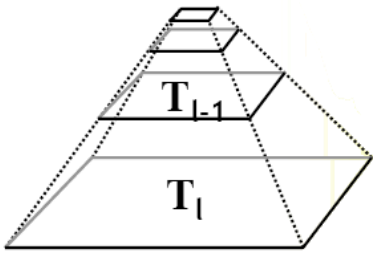


Figura 4-44: Pirâmide que representa o terreno [120].

Em tempo de execução como o *rendering* é efectuado do nível de menor detalhe para o de maior detalhe os dados de elevação são novamente estimados a partir do nível

anterior adicionando um *residual*$\widetilde{R_l}$que pode ser constituído por dados comprimidos ou sintetizados, ou seja, $\widetilde{T_l} = U(\widetilde{T}_{l-1}) + \tilde{R}_l$ num processo ilustrado na **Figura 4-45** que representa a transição entre dois níveis sucessivos de detalhe.

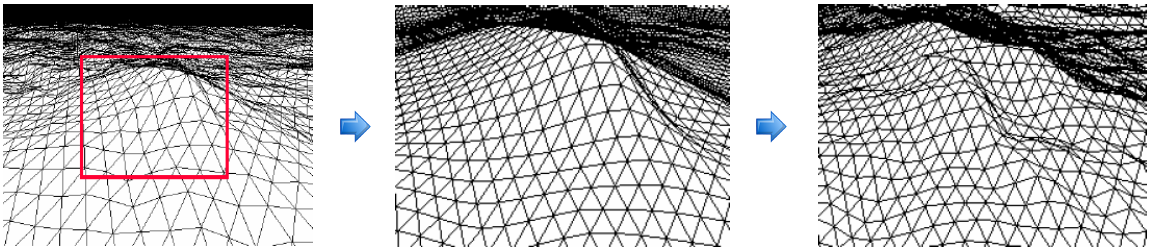


Figura 4-45: Refinamento de um nível de detalhe [10].

Em relação aos dados sintetizados, estes fazem sentido sobretudo para níveis de resolução acima da resolução do *height map* de origem, e são necessários sempre que o utilizador se aproxima do terreno, permitindo efectivamente o *rendering* de um terreno com um número potencialmente infinito de níveis de detalhe. Neste caso, o *residual* consiste em valores de elevação gerados com *Gaussian Noise* [67].

4.10. GPU Based Geometry Clipmaps

No artigo onde foi descrito pela primeira vez o algoritmo de geoclipmapping [120], os autores já referiam a possibilidade de concretizar esta técnica ao nível do GPU. No entanto, como nessa altura o *hardware* disponível não suportava ainda o *shader model* 3.0 mais especificamente uma das suas funcionalidades, o *vertex texturing* a implementação original recorria ao CPU em quase todos os passos do algoritmo com excepção do Geomorphing (ver **4.9.4**). De facto o algoritmo original implicava a alteração a cada *frame* de uma lista de vértices e de uma lista de índices o que muito embora tivesse algum peso tinha de ser efectuado desta forma por não existir nenhuma alternativa viável. Em [11] Asirvatham e Hoppe concretizaram a visão original descrevendo uma implementação deste algoritmo ao nível do GPU via *vertex textures*. Esta nova funcionalidade no *shader model* 3.0 permite o tratamento da geometria como um conjunto de imagens, algo que foi intensamente explorado nesta implementação, conseguindo-se desta forma pela primeira vez no domínio dos algoritmos de geração de terrenos efectuar quase todo o processamento ao nível da componente programável do *pipeline* gráfico (**Tabela 4-1)**. Assim, no caso de um terreno sintetizado, todas as operações anteriormente efectuadas ao nível do CPU como o *upsampling*, a síntese o cálculo do mapa de normais e o *rendering* passaram a ser executadas ao nível do GPU. Para terrenos comprimidos o CPU participa apenas ao nível da operação que permite o descomprimir dos dados [11].

	Implementação Original	GPU Based Geometry Clipmaps
Dados de Elevação	Numa lista de vértices	*Vertex Texture*
Lista de Vértices	Actualizado incrementalmente pelo CPU	Constante!
Lista de Índices	Gerado em cada *frame* pelo CPU	Constante!
Upsampling	CPU	GPU
Descompressão	CPU	CPU
Síntese	CPU	GPU
Adicionar residuais	CPU	GPU
Actualização do mapa de normais	CPU	GPU
Zonas de transição	CPU	GPU

Tabela 4-1: Comparação entre o algoritmo de *geoclipmapping* original e o novo.

Para adaptar o algoritmo original foram, como é óbvio, necessárias alterações, muito embora os princípios básicos se mantenham nomeadamente a representação do terreno como um conjunto de grelhas regulares concêntricas centradas no ponto de observação. Nesta perspectiva descrevem-se nas próximas secções as alterações mais relevantes à implementação original, nomeadamente ao nível da estrutura (ver **4.10.1**), da actualização dos *clipmaps* (ver **4.10.2**) já que ao nível do *culling* (ver **4.9.3**) e do tratamento de falhas (ver **4.9.4**) o princípio manteve-se inalterado ou foi adaptado tendo em conta as novas funcionalidades do *shader model* 3.0 exploradas por este algoritmo.

4.10.1. Representação do terreno

Nesta nova implementação do algoritmo de geoclipmapping apenas os valores (x, z) da grelha são armazenados numa lista de vértices, já que, os valores y de elevação são armazenados numa textura, o *height map*. Mais precisamente em cada nível l da pirâmide de *mipmaps* (ver **Figura 4-38**) é definida uma "janela" $n \times n$, o *clipmap*, que representa uma porção do terreno com uma determinada resolução. Assim, à medida que o utilizador se movimenta no terreno a textura é actualizada de acordo com a janela sobre os dados de elevação definida pelo *clipmap* corrente. Na versão anterior deste algoritmo de acordo com o movimento do utilizador era necessário modificar as listas de vértices e de índices associadas pelo que se dividia cada um dos *clipmaps* em 4 regiões (ver **Figura 4-40**) que tinham de ser recalculadas com o movimento do utilizador. No entanto, com esta nova possibilidade de alterar os valores de elevação no *vertex shader* podemos dividir o *clipmap* de modo a que os blocos se mantenham constantes e não seja necessário recalcular as listas de vértices e índices associadas. Estes blocos, também designados de 2D *footprints*, são reutilizados em todos os níveis e não sofrem alterações de *frame* para *frame*, o que é possível apenas porque a cada um deles, é aplicado no *vertex shader* uma translação e uma mudança de escala para que possam, por um lado, representar a posição actual do terreno e, por outro, a resolução de cada um dos *clipmaps* em cada um dos níveis. Na **Figura 4-46** estão representados todos os blocos utilizados no *rendering* de um *clipmap*. Estes são construídos com tiras de triângulos (ver **3.1.3**) tal como na versão

anterior. Tal como se pode verificar grande parte do *clipmap* é constituído por 12 blocos (a cinzento) de tamanho $m \times m$ em que $m = (n+1)/4$, o que significa que para um *clipmap* com $n = 255$ cada um destes blocos tem 64 vértices. No entanto, tal como se pode igualmente verificar, a união de todos estes blocos não cobre por completo o *clipmap*. Consequentemente, são necessárias *footprints* adicionais. A primeira zona a considerar é uma falha de $(n-1) - \big((m-1) \times 4\big)$ no meio de cada um dos lados do *clipmap* preenchida com um bloco $m \times 3$ (a verde na figura). A segunda zona está relacionada com o facto de um *clipmap* de menor detalhe não estar centrado em relação ao *clipmap* de maior detalhe que o "envolve". A zona que sobra em forma de "L" pode ocorrer em 4 posições distintas (topo-direita, topo-esquerda, baixo-direita e baixo-esquerda) dependentes da localização do *clipmap* de maior detalhe em relação ao de menor detalhe. Finalmente, as zonas a laranja na figura representam triângulos degenerados necessários para evitar *t-junctions* entre blocos de diferente detalhe (ver **3.4.4.1**). Resta apenas o nível de maior detalhe onde são utilizados 4 blocos $m \times m$ e mais uma região em forma de "L".

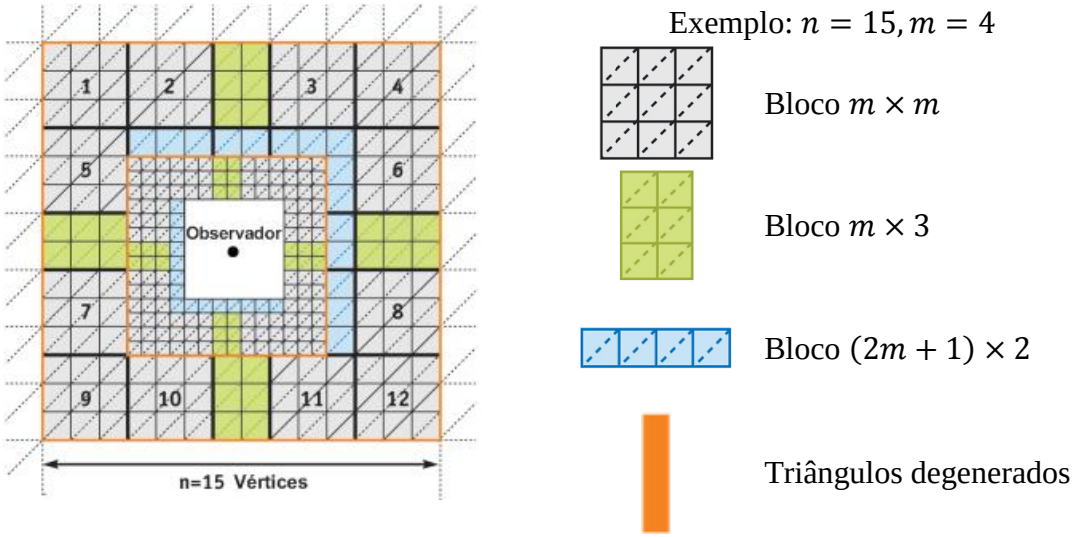


Figura 4-46: Divisão de um *clipmap* em regiões [11].

4.10.2. Actualização dos Clipmaps

À medida que o utilizador se movimenta no terreno em cada nível da pirâmide o *clipmap* respectivo tem de ser actualizado para que se mantenha centrado na posição do utilizador. Como o movimento do utilizador é coerente [120] de uma maneira geral apenas uma região em forma de "L" necessita de ser actualizada o que é efectuado acedendo toroidalmente aos dados de elevação tal como já foi descrito em **4.9.2**. À semelhança do algoritmo original, também aqui não se consideram por vezes todos os níveis da pirâmide de *clipmaps*, mas apenas aqueles em que o espaçamento entre os vértices é superior a $2.5h$, onde h corresponde à altura do observador em relação ao terreno. Tal está relacionado com o facto de a partir de uma determinada altura os níveis do *clipmap* serem demasiado densos, podendo *inclusive* causar problemas de *aliasing*. Por outro lado para simplificar a implementação do algoritmo o corte das *active regions*

em função da disponibilidade das *clip regions* (ver **Figura 4-39**) não foi implementado assumindo-se que o nível é actualizado na sua totalidade ou declarado inactivo.

4.11. Sumário

Neste capítulo abordaram-se em detalhe alguns dos algoritmos mais relevantes de geração de terrenos em tempo real, partindo-se para isso de uma classificação estabelecida em **4.1** que os divide em cinco classes: *Irregular Meshes*, *Bin-Tree Hierarchies*, *Bin-Tree Regions*, *Tiled Blocks* e *Concentric Regions*. Deu-se especial ênfase aos algoritmos da classe *Tiled Blocks* tendo-se descrito em particular o algoritmo de Geomipmapping (ver **4.4**), o algoritmo de Chunked LOD (ver **4.5**), o algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**), o algoritmo de Rendering Very Large, Very Detailed Terrains (ver **4.7**) e o algoritmo de GPU Terrain Rendering (ver **4.8**). Adicionalmente, apresentam-se dois algoritmos da classe *Bin-Tree Hierarchies* mais especificamente o ROAM (ver **4.2**) e o Real Time Generation of Continuous Level of Detail (ver **4.3**) e também dois algoritmos da classe *Concentric Regions*: o Geometry Clipmaps (ver **4.9**) e o GPU Based Geometry Clipmaps (ver **4.10**). De um modo geral, os algoritmos da classe *Irregular Meshes* e *BinTree Hierarchies* representam abordagens que se focam demasiado no CPU traduzindo as necessidades da época em que surgiram, nomeadamente a geração de uma triangulação quase perfeita de modo a que se processem o mínimo de triângulos. Entretanto abordagens baseadas em *BinTree Regions* surgiram para colmatar esse problema baseando-se na criação de blocos de geometria que eram no entanto simplificados da mesma forma. Foi só com os algoritmos da classe *Tiled Blocks* que se começou a ter uma abordagem mais orientada ao imenso poder de computação dos GPU actuais tendo-se vindo a manter essa tendência. Efectivamente, na classe de algoritmos mais recente, as *Concentric Regions*, foi possível em alguns algoritmos passar a maior parte do trabalho para o GPU tirando partido de algumas das novas funcionalidades do *shader model* 3.0 nomeadamente o *vertex texture fetch* que permite obter os valores de elevação ao nível do *vertex shader*.

De especial relevância foram os métodos empregues por cada um destes algoritmos para diminuir o nível de detalhe e resolver problemas como falhas (ver **3.4.4.1**) e *popping* (ver **3.4.4.2**) e também as diferentes estratégias adoptadas nomeadamente ao nível do *culling* (ver **3.3**) e do suporte para terrenos de grande dimensão (*out-of-core*).

Para finalizar, um nota especial para o algoritmo Terrain Occlusion Culling with Horizons já que de todos os algoritmos apresentados é único que utiliza uma técnica específica para remover os blocos ocultos por outros blocos diminuindo assim a quantidade de geometria enviada para *rendering* pelo que representa nesse domínio uma abordagem mais completa.

5. Trabalho Desenvolvido

Numa perspectiva de se concretizar alguns dos conceitos descritos nesta dissertação, procurou-se realizar um trabalho que pudesse servir de base para se efectuar uma análise detalhada dos mesmos. O foco foram as técnicas de nível de detalhe (ver **3.4**) aplicadas à geração de terrenos em tempo real complementadas com a exploração de outras técnicas ao nível da representação geométrica (ver **3.1**) e do *culling* (ver **3.3**). Para restringir a gama de algoritmos a analisar, este trabalho incidiu sobre uma classe específica de algoritmos: os *Tiled Blocks* (ver **4.1**). A escolha desta classe foi motivada pelo facto de a maior parte das abordagens existentes utilizarem uma aproximação semelhante e por os algoritmos pertencentes a esta classe tirarem partido do poder computacional das placas gráficas mais recentes, o que neste caso significa que não procuram encontrar o conjunto perfeito de triângulos, mas sim efectuar o processamento da maior quantidade de geometria possível. Pretendeu-se, deste modo, efectuar a análise mais completa possível dos diferentes elementos a ter em consideração para se concretizar um algoritmo de geração de terrenos em tempo real que garanta não só um uma boa qualidade em termos visuais, mas também um bom desempenho. Para isso, e numa perspectiva de encontrar a melhor abordagem possível, concretizaram-se dois dos algoritmos descritos em **4**: o Geomipmapping (ver **4.4)** e o GPU Terrain Rendering (ver **4.8**) e também uma abordagem de força bruta que não se baseando em nenhuma técnica de diminuição do nível de detalhe, é utilizada com o intuito de servir de termo de comparação com os outros dois algoritmos, para avaliar a eficácia das técnicas de redução de nível de detalhe.

Os algoritmos considerados foram seleccionados por concretizarem aproximações diferentes no tratamento do nível de detalhe nomeadamente a simplificação por blocos no caso do Geomipmapping e a simplificação por intermédio da *quadtree* no caso do GPU Terrain Rendering e por estas serem duas das técnicas mais utilizadas no tratamento do nível de detalhe.

Globalmente, o objectivo foi, por um lado, comparar os algoritmos, numa perspectiva de se encontrar qual o melhor não só a nível de desempenho como a nível de qualidade visual, e por outro, descrever todos os passos necessários para concretizar um algoritmo de geração de terrenos em tempo real. Neste contexto é de realçar a relevância dada aos métodos utilizados para resolver alguns dos problemas mais comuns em algoritmos de nível de detalhe, tais como as falhas (ver **3.4.4.1**) e o *popping* (ver **3.4.4.2**) descrevendo-se para estes as soluções adoptadas. Igualmente relevante foi a integração da técnica de *occlusion culling* (ver **3.3.3**) do algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**), que permitiu a detecção de blocos ocultos e consequentemente a diminuição do número de blocos enviados para *rendering*. Finalmente, e com o intuito de avaliar o impacto das novas funcionalidades do *shader model* 3.0, nomeadamente das *vertex textures* em algoritmos da classe *Tiled Blocks*, descrevem-se as estratégias adoptadas para a inclusão desta funcionalidade. A abordagem descrita constitui por si só uma perspectiva nova por integrar um conjunto de conceitos como o *occlusion culling* e as *vertex textures* numa implementação única.

No que diz respeito à implementação propriamente dita, adoptou-se o XNA, uma *framework* em .NET que assenta no DirectX [45] da Microsoft e na linguagem C#. O .NET e todas as linguagens que suporta baseia-se numa *virtual machine* comum, o

CLR ou *Common Language Runtime*, que efectua o *garbage collection* e oferece de um modo geral um conjunto serviços que tornam o desenvolvimento muito mais fácil e seguro num claro contraste com abordagens nativas como o C++. Por outro lado o XNA permite também utilização da consola XBOX 360 pelo que permite obter uma perspectiva interessante nomeadamente ao nível do desempenho de uma solução nesse ambiente.

Na organização deste capítulo procurou-se introduzir progressivamente as diferentes partes que constituem os algoritmos implementados. Para isso, é feito um resumo de todas as funcionalidades concretizadas em **5.1**. De seguida, dá-se uma visão base do que vai ser discutido com a apresentação de um algoritmo geral em **5.2** que estabelece a sequência de passos comuns aos algoritmos concretizados. Depois, em **5.3** descreve-se a estrutura base do terreno nomeadamente a sua organização nos diferentes algoritmos. Na secções seguintes, segue-se o fluxo do algoritmo pelo que se começa com uma descrição do processo de construção da *quadtree* (ver **5.4**), dando-se especial destaque ao conjunto de dados necessário para nas fases seguintes se conseguir concretizar as funcionalidades referidas. Continua-se com uma descrição do modo como é construída a malha triangular (ver **5.5**) utilizada na representação do terreno, para de seguida se apresentar as estratégias utilizadas para efectuar o *rendering* do terreno discutindo em maior detalhe as diferentes fases associadas a cada um dos algoritmos de nível de detalhe concretizados (ver **5.6**) bem como o modo como em tempo de *rendering* se consegue a cada *frame* determinar os blocos de terreno ocultos por outros blocos. Finalmente em **5.7** e **5.8** abordam-se de uma forma resumida as opções tomadas na aplicação da textura ao terreno e na iluminação do mesmo, enquanto que em **5.9** se descrevem algumas das características mais relevantes do XNA enquanto plataforma de desenvolvimento.

5.1. Funcionalidades

Tal como já foi referido, o objectivo principal foi comparar algoritmos baseados na classe *Tiled Blocks* (ver **4.1**), incluindo um conjunto de funcionalidades directamente relacionadas com esta classe e de um modo geral presentes nos algoritmos de geração de terrenos em tempo real. Na avaliação efectuada de cada uma dessas funcionalidades o intuito foi realizar uma análise detalhada de cada uma delas tendo sido consideradas em alguns casos diferentes aproximações numa perspectiva de se seleccionar a que apresenta melhor desempenho. Assim, e de uma forma mais especifica, foram concretizadas as seguintes funcionalidades:

- Estrutura Geométrica (ver **3**)
    - Suporte para listas e tiras indexadas de triângulos (ver **3.1.1** e **3.1.3**)
    - Envio dos valores de elevação em vértices ou em alternativa por intermédio de *vertex textures*.
    - Optimização para a *cache* de vértices existente nas placas gráficas (ver **3.1**).
    - Partição espacial (ver **3.2**) por intermédio de uma *quadtree* (ver **3.2.3.1**)
- *Culling* (ver **3.3**)
    - *View Frustum Culling* (ver **3.3.2**)
    - *Occlusion Culling* (ver **3.3.3**), por intermédio da técnica de oclusão descrita no algoritmo de Terrain Occlusion Culling with Horizons

- Tratamento do nível de detalhe (**3.4**)
    - Brute Force
    - Algoritmo de Geomipmapping (ver **4.4**).
    - Algoritmo de GPU Terrain Rendering (ver **4.8**).
- Tratamento de problemas causados pela variação do nível de detalhe (ver **3.4.4**)
    - Actualização de índices no algoritmo de Geomipmapping.
    - Skirts (ver **3.4.4.1**) no algoritmo de Geomipmapping e no algoritmo de GPU Terrain Rendering.
    - Geomorphing (ver **3.4.4.1**) no algoritmo de Geomipmapping e no algoritmo de GPU Terrain Rendering.
- Iluminação (ver **5.8**)
    - Per Pixel Lightning com luzes direccionais.

5.2. Algoritmo Geral

Antes de se iniciar a discussão propriamente dita do trabalho realizado é necessário descrever a alto nível os diferentes passos envolvidos na construção do terreno, e no *rendering* do mesmo. Estes são apresentados em traços muito gerais na **Listagem 5-1** que de uma forma resumida aborda todas as fases implementadas. Este resumo é importante na medida em que permite observar o sistema como um todo e paralelamente enquadrar os três algoritmos numa *framework* comum que permite identificar as fases mais importantes de todo o processo.

Como se pode verificar são consideradas duas fases: a inicialização e o *rendering*. Na inicialização constrói-se a *quadtree* (ver **5.4**) que armazena em cada nó o bloco correspondente à porção de terreno que representa. Após a construção da *quadtree*, é a vez da estrutura geométrica do terreno (ver **5.5**), mais precisamente das listas de vértices e de índices associadas. Terminada esta fase é efectuado o *rendering* propriamente dito da cena. A primeira fase desse processo implica a determinação do conjunto de blocos a enviar para *rendering.* Para isso, é necessário percorrer a *quadtree* recursivamente a partir do nó raiz, isto é, a partir do bloco que representa o terreno na sua totalidade, verificando-se a visibilidade de cada um dos blocos face ao ponto de vista corrente. Nesse sentido, testa-se em primeiro lugar a *bounding box* (ver **3.2.1**) associada a cada um desses blocos contra o *frustum* de modo a determinar a sua visibilidade. Se se verificar que o bloco está visível, o próximo passo tem como objectivo determinar se o bloco deve ou não ser adicionado à lista de blocos a enviar para *rendering* passo que vai variar de acordo com o algoritmo de nível de detalhe adoptado. Este processo continua recursivamente até se atingir um nó folha ou então até um nó interno ser aceite pelo algoritmo de nível de detalhe como válido para o ponto de vista corrente. Nesse momento, o bloco correspondente é adicionado a uma lista de blocos por ordem da distância à câmara, ou seja, os blocos mais próximos do ponto de vista são introduzidos em primeiro lugar. Esta ordenação é essencial para que seja possível aplicar o algoritmo de oclusão no passo seguinte. Desta forma, tal como está representado na listagem, numa fase posterior percorre-se a lista de blocos obtida sendo efectuado o *rendering* de todos os blocos visíveis face à linha de horizonte. Para isso, à medida que se percorre a lista de blocos vai sendo adicionada ao horizonte a contribuição de cada um dos blocos visíveis, o que

permite efectuar o *culling* de blocos posteriores na lista que se verifique estarem abaixo da linha horizonte até aí definida.

```
01 Inicializar()
02   ContruirQuadtree()
03   ContruirGeomeriaTerreno()
04 Fim Inicializar

01 ObterListaBlocos(listaBlocos, bloco)
02   Se o bloco não intersecta o frustum
03    Sair
04   Fim Se
04
05   Se Aceitar(bloco) ou bloco interno então
06    Adicionar bloco à listaBlocos por ordem
07    da distância à camâra
08   Senão
09    Para Cada bloco filho
10      ObterListaBlocos (listaBlocos, filho)
11    Fim Para Cada
12   Fim Se
13 Fim ObterListaBlocos

01 Render()
02   listaBlocos:=Vazia
03   ObterListaBlocos(listaBlocos, blocoRaiz)
04   Para Cada bloco em listaBlocos
05     Se Visivel(bloco)
06       AdicionarHorizonte(bloco)
07       Desenhar(bloco)
08     Fim Se
09   Fim Para Cada
10 Fim Render

01 Inicializar()
02 Enquanto Não Sair()
03   Render()
04 Fim Enquanto
```

Listagem 5-1: Pseudo-código dos principais passos do algoritmo base.

5.3. Estrutura Base

A característica mais marcante nos algoritmos baseados na classe *Tiled Blocks* (ver **4.1**) é a divisão do terreno em blocos o que, tal como já foi referido, facilita o *culling* por um lado e por outro permite controlar de uma forma mais eficaz o nível de detalhe. O terreno em si, tal como os blocos que o compõem, está organizado numa grelha regular de $(2^n + 1) \times (2^n + 1)$ em que $n \in \left[1,\rightarrow\right]$ devendo o número de vértices do bloco ser inferior ou igual ao o número de vértices do terreno. Assim, por exemplo, para um terreno de 9×9 podemos ter um único bloco de 9×9, 4 blocos de 5×5, 16 blocos de 3×3 ou 64 blocos de 2×2. É de realçar ainda que uma das principais características destes algoritmos é facto de o bloco ser a unidade de *rendering*. Isto é, usando-se uma determinada primitiva (ver **3.1**), neste caso, listas de triângulos ou tiras de triângulos, para criar a triangulação de um terreno é possível definir um bloco com uma única chamada a uma destas primitivas. Desta forma, o número de blocos visíveis num determinado momento corresponde ao número máximo de chamadas à primitiva gráfica que se terá de efectuar para representar a porção visível do terreno. Por exemplo, partindo

do princípio que toda a extensão do terreno está visível se fizermos a divisão de um terreno 9×9 em blocos 5×5 vamos ter no máximo quatro chamadas à primitiva de *rendering* utilizada.

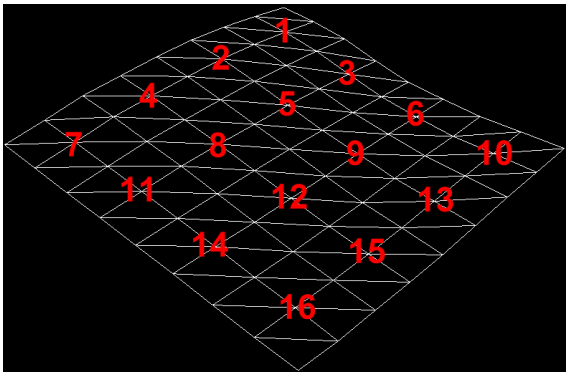
No que diz respeito aos diferentes algoritmos de nível de detalhe e às diferentes formas de representar a geometria, procurou-se encapsular numa implementação base a maioria dos conceitos relevantes no domínio da classe *Tiled Blocks*, sendo o objectivo a unificação desses conceitos, obtendo-se dessa forma um ponto de partida para a implementação de algoritmos de nível de detalhe semelhantes (ver **5.2**). A primeira constatação foi que seria necessária uma estrutura para efectuar a partição espacial do terreno. Nesse domínio a *quadtree* (ver **3.2.3.1**) foi uma escolha natural pois permite descrever facilmente a estrutura espacial do terreno tendo nesta implementação um papel determinante no *culling* e na selecção do nível de detalhe pelo que é efectivamente a base de toda a estrutura.

Utilizando a *quadtree* como base, é importante agora unificar os conceitos inerentes aos dois algoritmos de nível de detalhe implementados. Nesse sentido, e comparando mais especificamente o algoritmo de Geomipmapping e o algoritmo de GPU Terrain Rendering, podemos verificar que os dois se baseiam na *quadtree* como estrutura de suporte. No entanto, existe um conjunto de diferenças entre eles no que diz respeito ao modo como é feito o tratamento do nível de detalhe que importa realçar. No Geomipmapping a dimensão dos blocos enviados para *rendering* mantêm-se e o número de vértices do bloco varia com o nível de detalhe. No GPU Terrain Rendering a dimensão dos blocos varia com o nível de detalhe e o número de vértices mantêm-se. A questão que surge imediatamente é como unificar estes dois conceitos. Se observarmos o problema do ponto de vista da estrutura espacial que é comum aos dois, a *quadtree*, depressa verificamos que no caso do Geomipmapping se percorre a *quadtree* até ao nó folha correspondente ao tamanho de bloco pretendido sendo o nível de detalhe controlado ao nível do próprio bloco, isto é aumenta-se ou diminui-se o número de vértices a esse nível. No caso do GPU Terrain Rendering a mesma porção de terreno que é representada por vários blocos de diferentes níveis de detalhe no Geomipmapping é neste algoritmo representada por um único bloco. A variação neste caso ocorre no espaçamento entre vértices. Isto é, no GPU Terrain Rendering o nível de detalhe corresponde directamente aos níveis da *quadtree* e ao princípio de partição espacial que lhe está associado. Efectivamente o Geomipmapping controla o nível de detalhe via uma variação ao nível da lista de índices de cada um dos blocos e o GPU Terrain Rendering pela dimensão dos blocos que envia para *rendering*, ou seja, pela área que ocupam, e consequentemente actua mais ao nível do número de chamadas efectuadas à primitiva.

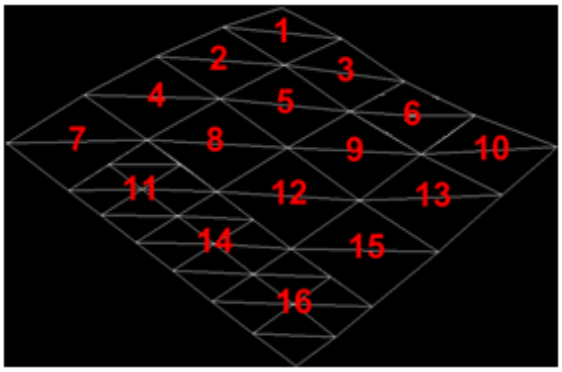
As diferenças entre os dois algoritmos estão ilustradas na **Figura 5-1** com a representação da malha poligonal gerada por cada um deles para um terreno 9×9 com um bloco 3×3. Como se pode verificar, no Geomipmapping o espaço ocupado pelos blocos é sempre o mesmo, variando o número de vértices considerados e consequentemente o número de triângulos de cada um dos blocos (por exemplo no bloco 1 são considerados apenas 4 vértices). Para que isso seja possível, são geradas diferentes descrições do bloco na lista de índices que correspondem a diferentes níveis de detalhe. Adicionalmente, o número de blocos enviados para *rendering* depende neste caso exclusivamente do *frustum culling*, pelo que no exemplo como o terreno está todo visível são 16 os blocos enviados para *rendering*. No caso do GPU Terrain Rendering para este

ponto de vista específico e para a métrica de erro considerada são enviados para *rendering* 10 blocos. É de notar que o bloco 1, por exemplo, ocupa uma dimensão superior ao bloco 3, ou seja, este algoritmo actua mais ao nível do número de chamadas efectuadas à primitiva mantendo-se, no entanto, o número de triângulos por bloco independentemente do seu tamanho.

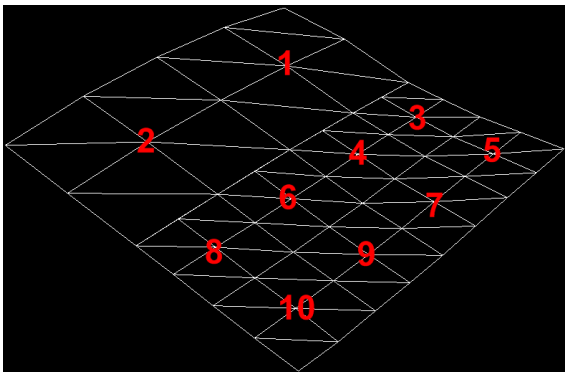
No que diz respeito ao método de força bruta, o princípio utilizado corresponde à selecção do número de vértices para o bloco, à semelhança do que é feito para o Geomipmapping, mas neste caso tal como está representado não é efectuada uma diminuição do nível de detalhe. Pura e simplesmente são enviados para *rendering* todos os blocos visíveis, pelo que esta abordagem integra-se facilmente no esquema existente.



Brute Force



Geomipmapping



GPU Terrain Rendering

Figura 5-1: A redução do nível de detalhe em cada um dos algoritmos.

5.4. Construção da Quadtree

A *quadtree* (ver **3.2.3.1**) é fundamental, pois permite por um lado acelerar o processo de identificação dos blocos visíveis para um determinado ponto de vista e por outro no caso do algoritmo de GPU Terrain Rendering determinar o nível de detalhe mais adequado para representar uma determinada porção do terreno. É construída directamente

a partir do terreno que se pretende representar, mais precisamente o número de níveis é determinado a partir do número de vértices do bloco e do número de vértices do *height field* pelo que a área representada por cada um dos nós está directamente relacionada com uma porção do terreno. Por exemplo, tal como está representado na **Figura 5-2** para um terreno 9×9 e um bloco 3×3 são necessários 3 níveis. O primeiro nível representa o terreno na sua totalidade e corresponde ao nó raiz da *quadtree*. O segundo nível representa a divisão em 4 blocos 5×5 de terreno e o terceiro e último nível à divisão dos 4 blocos 5×5 em 16 blocos 3×3, o número de vértices considerado para o bloco.

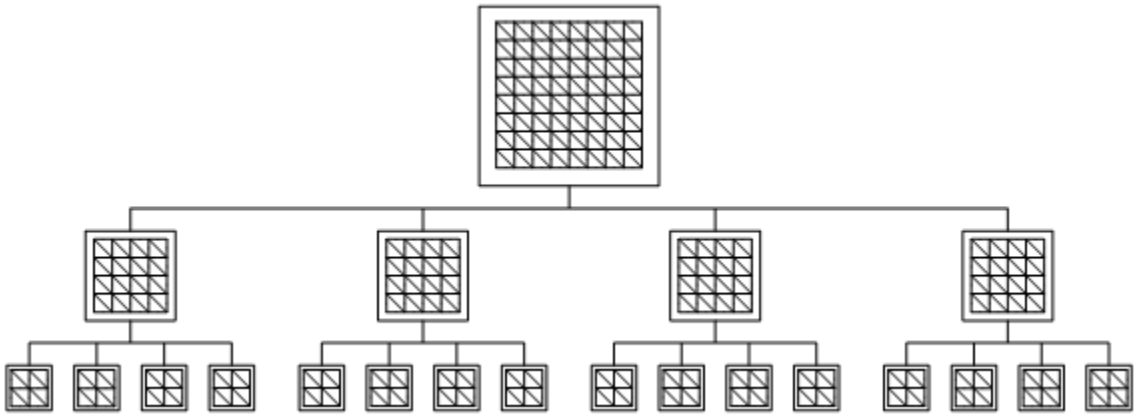


Figura 5-2: Partição espacial de um terreno 9×9 para um bloco 3×3.

É de realçar no entanto que nesta implementação são considerados efectivamente dois valores para o número de vértices do bloco: o número de vértices do bloco de terreno e o número de vértices dos blocos nos nós folha da *quadtree*. Este último determina efectivamente o número de níveis da *quadtree*, devendo ser obrigatoriamente inferior ou igual ao número de vértices do bloco de terreno. O porquê desta diferença está relacionado directamente com o método de oclusão utilizado, neste caso, o descrito no algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**). Neste algoritmo, tal como é referido em **4.6.4**, é construído um horizonte percorrendo a *quadtree*, pelo que está dependente do número de níveis considerados a resolução desse horizonte. Assim para obter taxas de oclusão mais elevadas é por vezes necessário considerar mais níveis do que os resultantes do número de vértices pretendido para o bloco de terreno. Deste modo em terrenos de grandes dimensões, por exemplo, 1025×1025 ou 2049×2049 é típico considerar blocos de 17×17, 33×33 ou mesmo de 65×65. Nesses casos pode-se considerar mais níveis de modo a aumentar a eficiência do algoritmo de oclusão. Por exemplo para um terreno de 1025×1025 se considerássemos blocos de 17×17 teriamos 7 níveis. No entanto, utilizando um valor diferente para o número de vértices do bloco, por exemplo 5×5, já vamos ter 9 níveis. Esta necessidade de gerar mais níveis está directamente relacionada com a informação que é armazenada em cada bloco que é necessária para a execução do algoritmo de oclusão, nomeadamente, o plano dos mínimos quadrados e os planos máximos e mínimos que lhe estão associados. Assim, em cada nó e em cada nível da *quadtree* são armazenados um conjunto de dados que permitem concretizar algumas das funcionalidades referidas em **5.1**, das quais se destacam as seguintes:

- A lista de erros geométricos (ver **5.4.1**) para a determinação do nível de detalhe no algoritmo de Geomipmapping e no algoritmo de GPU Terrain Rendering. Esta lista tem apenas um elemento excepto no nível da *quadtree* que corresponde ao número de vértices adoptado para o qual, no caso do Geomipmapping, é necessário armazenar a lista de erros geométricos correspondentes aos diferentes níveis de detalhe possíveis para esse bloco. É neste nível que no Geomipmapping se diminui o número de triângulos por selecção do nível de detalhe mais adaptado para o bloco face ao ponto de vista.
- A altura das *skirts* (ver **5.4.2**), no caso de estas serem usadas na correcção de falhas entre blocos adjacentes de diferentes níveis de detalhe.
- Dados de oclusão (ver **5.4.3**), nomeadamente o plano dos mínimos quadrados, os planos mínimos e máximos que lhe estão associados e o valor do erro da aproximação, todos para o algoritmo de Terrain Occlusion Culling with Horizons.
- As relações de vizinhança entre cada um dos blocos que são utilizadas no algoritmo de Terrain Occlusion Culling with Horizons e na correcção de falhas por actualização de índices.

5.4.1. Erro Geométrico

O erro geométrico é muito importante na medida em que permite determinar o impacto resultante de uma diminuição do nível de detalhe. É calculado por vértice e resulta da diferença entre o valor de elevação real e o valor calculado por interpolação dos valores de elevação dos vértices vizinhos do qual está directamente dependente. O conceito é ilustrado em 2D na **Figura 5-3** onde as setas a tracejado representam o erro geométrico resultante de um diminuição de nível de detalhe para cada um dos vértices que desaparecem na transição de um bloco 9×9 para um bloco 5×5.

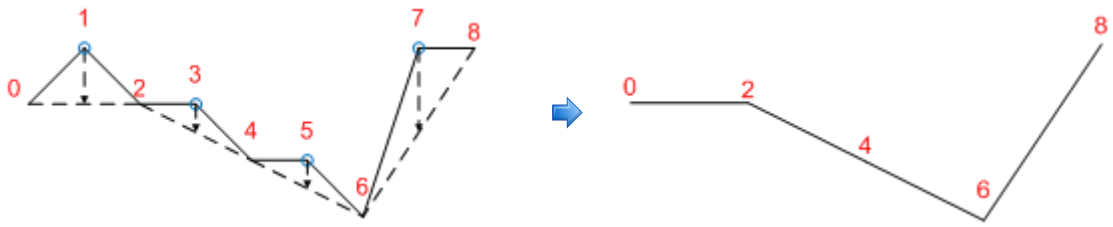


Figura 5-3: Erro geométrico na transição de um bloco 9×9 para um bloco 5×5.

Na **Figura 5-4** podemos observar os vértices vizinhos que têm de ser considerados no cálculo do erro geométrico para os vértices que desaparecem na transição entre níveis tal como está ilustrado na figura. De particular relevância é o cálculo do erro geométrico do vértice 10. Neste caso foram considerados os vértices 4 e 16 como vizinhos e não o 6 e o 14, o que está directamente relacionado com a forma como é efectuada a triangulação. Em cada um dos casos o valor de elevação calculado corresponde ao valor médio dos valores de elevação vizinhos e não é mais do que o valor de elevação desse vértice no próximo nível de detalhe.

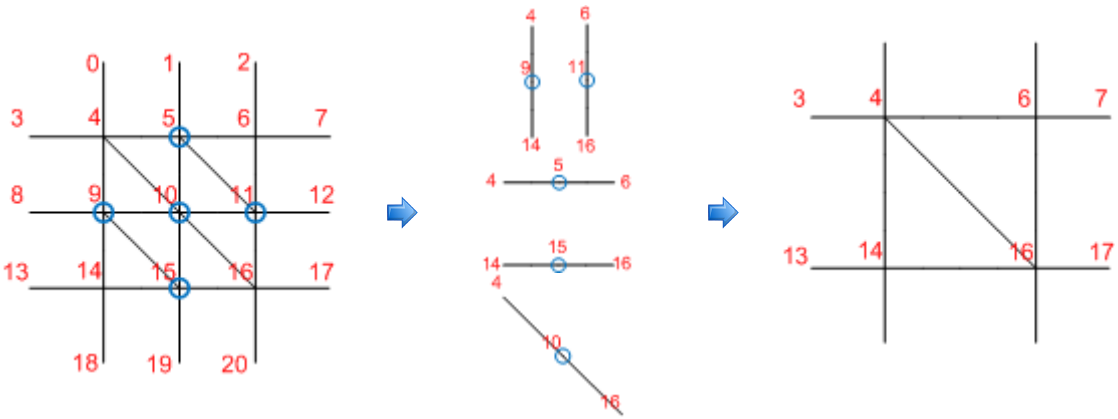
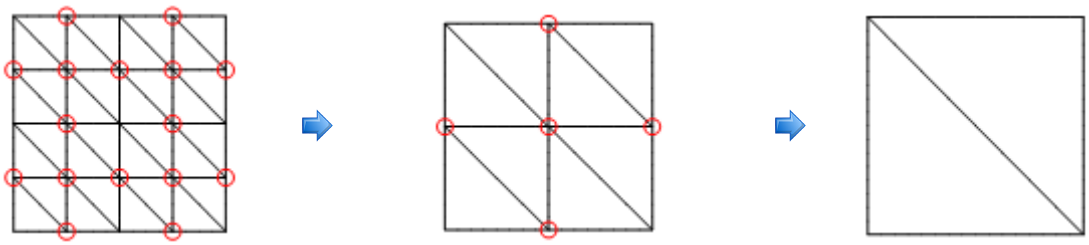


Figura 5-4: Vértices vizinhos considerados na interpolação dos valores de elevação.

É de notar, no entanto, que erro o geométrico é normalmente visto por bloco, sendo por isso interpretado a esse nível como o erro geométrico máximo de todos os vértices que desaparecem numa transição entre níveis de detalhe.

Falando agora especificamente na forma como o erro geométrico é utilizado, temos de considerar os dois algoritmos de nível de detalhe em estudo: o Geomipmapping e o GPU Terrain Rendering. Estes são de particular importância neste contexto já que aplicam técnicas de redução do nível de detalhe recorrendo a métricas de erro para determinar o nível mais adaptado. Essas métricas de erro podem basear-se pura e simplesmente na distância, tal como a descrita no algoritmo original de GPU Terrain Rendering, ou de uma forma mais comum no erro geométrico em *pixels* (ver **3.4.3**). O erro geométrico em *pixels* é a melhor opção, já que, para cada um dos blocos, tem em consideração não só a distância mas também o erro resultante de uma mudança de nível de detalhe. Tal permite diminuir o efeito de *popping* pois as mudanças de nível de detalhe com mais impacto são realizadas a maiores distâncias. Por isso, e também com o intuito de uniformizar e de facilitar a comparação, adoptou-se o erro geométrico em *pixels* como métrica de erro nos dois algoritmos. A mudança ocorre no algoritmo de GPU Terrain Rendering onde se utilizava apenas a distância. No entanto, tal como é referido pelo autor [207], para se obter melhores resultado é preferível uma métrica de erro deste género. Desta forma é necessário calcular o erro geométrico nos dois casos. No entanto, a forma como este é utilizado na determinação do nível de detalhe em cada um dos casos é diferente. No caso do Geomipmapping, o nível de detalhe diminui ou aumenta pela alteração do número de triângulos considerados para cada bloco. Desta forma, para o número de vértices do bloco seleccionado para *rendering*, é necessária uma lista de erros geométricos que representam o erro resultante da transição entre os diferentes níveis de detalhe associados a esse bloco. Assim, tomando como exemplo a *quadtree* representada na **Figura 5-2**, que representa um terreno 9×9, mas assumindo desta vez um bloco de dimensão 5×5, isto é, a divisão do terreno em 4 blocos 5×5, temos de armazenar no segundo nível da árvore em cada um desses blocos uma lista de erros geométricos que representam a transição do nível de detalhe mais elevado, que é neste caso 5×5, para o nível de detalhe mais baixo, 2×2. Isto corresponde a ter uma lista com três níveis de detalhe em cada um dos blocos: 0 (5×5), 1 (3×3) e 2 (2×2), tal como está representado na **Figura 5-5** onde estão marcados com um círculo os vértices que desaparecem em cada uma das transições. No nível de detalhe 0, ou seja, o de maior detalhe, o erro geométrico é 0, já que não existe

nenhum desvio em relação ao bloco original. No nível de detalhe 1, existe uma diminuição no número de vértices passando-se de 5×5 vértices para 3×3 vértices, pelo que é necessário calcular o erro geométrico resultante dessa transição. O mesmo se passa na transição do nível de detalhe 1 (3×3) para o nível de detalhe 2 (2×2) onde se calcula novamente o erro geométrico, mas desta vez apenas sobre os vértices afectados durante essa transição. O erro geométrico de cada um dos blocos vai resultar neste caso do erro geométrico máximo calculado a partir de cada um dos vértices afectados na transição do nível corrente para um nível inferior mais o erro geométrico obtido da transição anterior, caso exista. Por exemplo, no caso de uma transição do nível 1 (3×3) para o nível 2 (2×2) o erro geometrico é obtido como o máximo dos erros geométricos calculados para cada um dos 5 vértices que desaparecem nesta transição (passando-se de 9 para 4 vértices) mais o erro geométrico calculado previamente que resulta da transição do nível 0 para o nível 1 na qual se "perderam" 16 vértices (de 25 para 9 vértices). Este modo de calcular o erro geométrico garante que cada nível de detalhe na lista de erros geométricos tem um valor progressivamente mais elevado, dado que se soma sempre ao valor do anterior sendo este método de particular importância para se efectuar uma transição suave entre níveis de detalhe por intermédio de Geomorphing (ver **5.6.3**).



Nível 0 (5×5) Nível 1 (3×3) Nível 2 (2×2)

Figura 5-5: Vértices que desaparecem na transição de um nível para outro.

Na **Equação 5-1** é formalizado este conceito correspondendo esta ao método adoptado na implementação do Geomipmapping para calcular o impacto da diminuição do nível de detalhe.

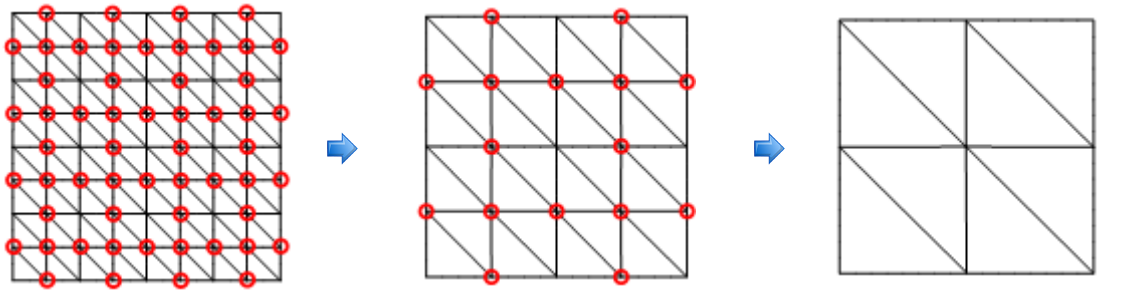
$\delta_m = \max(\delta_{v1}, \dots, \delta_{vn})$	δ_m: Máximo dos erros geométricos dos vértices que desaparecem na transição do nível l para o nível $l+1$
$\delta_{bl} = \begin{cases} 0, \text{ se nível de maior detalhe} \\ \delta_m + \delta_{b(l-1)}, \text{ caso contrário} \end{cases}$	$\delta_{b(l-1)}$: Erro geométrico do nível anterior.

Equação 5-1: Erro geométrico δ_{bl} de um bloco no Geomipmapping.

É ainda importante realçar que este exemplo considera propositadamente uma *quadtree* de profundidade superior ao número de vértices do bloco, pois tal como foi referido é feita uma distinção entre o número de níveis da *quadtree* propriamente dita e o número de níveis resultantes do número de vértices pretendido para o bloco, neste caso 5×5. Esta distinção resulta da necessidade de se armazenar mais níveis de modo a

aumentar a precisão do algoritmo de oclusão empregue. Concluindo, no algoritmo de Geomipmapping para cada bloco com o número de vértices pré-estabelecido para *rendering* armazena-se uma lista erros geométricos que correspondem aos níveis de detalhe considerados para esse bloco.

Analisando agora o algoritmo de GPU Terrain Rendering, o erro geométrico deve ser calculado de outra forma. Aqui, a diminuição do nível de detalhe é efectuada pela diminuição do número de blocos enviados para *rendering*, isto é, pela substituição de 4 blocos por um único bloco que ocupa o mesmo espaço. Desta forma, neste algoritmo, há, ao contrário do Geomipmapping, uma ligação ao princípio de partição espacial da *quadtree*. Assim, e tal como está representado na **Figura 5-6** o erro geométrico que resulta de uma mudança de nível de detalhe está directamente relacionado com os erros geométricos dos quatro blocos que vão ser substituídos no nível de detalhe seguinte por um único bloco. Deste modo e tendo em conta a figura, se considerarmos um bloco 3 × 3 num terreno 9 × 9 vamos ter no nível 2 da *quadtree* 16 blocos 3 × 3, no nível 1, 4 blocos 3 × 3 e no nível 0, 1 bloco 3 × 3, isto é, o número de vértices por bloco mantém-se em cada nível, o que varia é o espaço ocupado pelos blocos e o número de blocos. Sendo assim, é necessário armazenar o erro geométrico em cada um dos blocos de cada um dos níveis. Este erro geométrico propaga-se do nível de maior detalhe, onde é zero, para o nível de menor detalhe onde resulta do erro geométrico do próprio bloco e dos erros dos blocos filhos.



Nível: 2
Número Vértices: 9 × 9
Blocos: 16

Nível: 1
Número Vértices: 5 × 5
Blocos: 4

Nível: 0
Número Vértices: 3 × 3
Blocos: 1

Figura 5-6: Vértices que desaparecem num terreno 9 × 9 com um bloco 3 × 3.

Resta agora identificar a equação utilizada no cálculo do erro geométrico no algoritmo de GPU Terrain Rendering, que é neste caso a **Equação 5-2**, a mesma utilizada no algoritmo de Rendering Very Large, Very Detailed Terrains (ver **4.7**) e que se repete aqui por questões de referência. Tal como foi referido anteriormente a métrica original do algoritmo de GPU Terrain Rendering considerava apenas a distância o que impede a atribuição de detalhe às áreas que mais dele necessitam. Isto é, se considerarmos apenas a distância, o efeito de *popping* pode ser mais visível já que numa transição entre níveis de detalhe a perda de informação pode ser maior nuns casos do que noutros. Assim, se tivermos em consideração esse facto é possível tornar o *rendering* do terreno fluido e detalhado. Esta métrica é ideal nessa perspectiva uma vez que não só tem em consideração a distância, mas também o erro resultante de uma mudança de nível de

detalhe. Paralelamente garante que o erro aumenta dos nós filhos para o nó pai pois o erro geométrico do bloco δ_b é calculado como o máximo dos erros geométricos dos filhos desse bloco, $\delta_{b0}, \delta_{b1}, \delta_{b2}, \delta_{b3}$, mais o erro geométrico do próprio bloco δ_m.

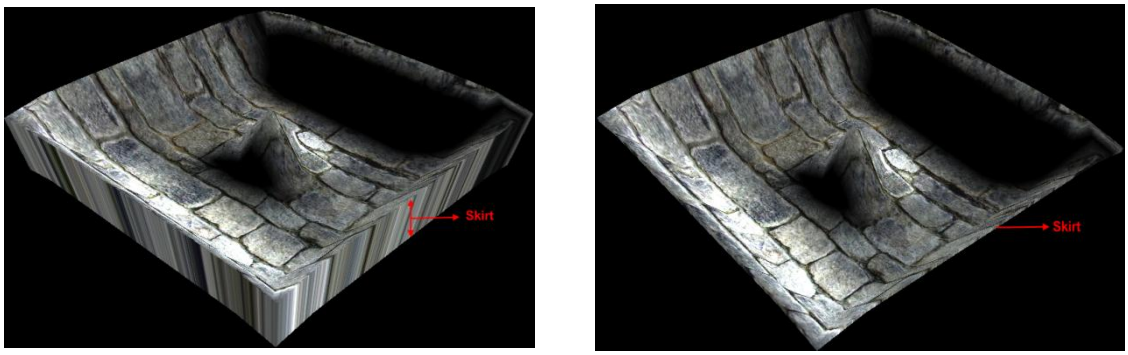
$\delta_m = \max(\delta_{v1}, \dots, \delta_{vn})$	δ_m: Máximo dos erros geométricos dos vértices que desaparecem na transição do nível l para o nível $l-1$ da *quadtree*.
$\delta_b = \begin{cases} 0, & \text{se nó de maior detalhe} \\ \max(\delta_{b0}, \delta_{b1}, \delta_{b2}, \delta_{b3}) + \delta_m, & \text{caso contrário} \end{cases}$	$\max(\delta_{b0}, \delta_{b1}, \delta_{b2}, \delta_{b3})$: Máximo dos erros geométricos dos filhos.
	δ_m: Erro geométrico do bloco.

Equação 5-2: Erro geométrico δ_b de um bloco no GPU Terrain Rendering.

5.4.2. Altura das Skirts

As *skirts* (ver **3.4.4.1**) são uma técnica utilizada para evitar o aparecimento de falhas entre blocos adjacentes de diferentes níveis de detalhe. Pelo facto de não implicarem uma mudança da geometria são possivelmente uma das formas mais eficientes de tratar esse problema. No entanto, para que o desempenho seja efectivamente uma vantagem, é muito importante que a altura das *skirts* que envolvem o bloco seja estritamente a necessária. Isto porque, a altura tem um impacto significativo no *fill rate* podendo, se demasiado elevada, causar uma diminuição no desempenho o que é precisamente o contrário do que se pretende. Assim para que as *skirts* possam ser uma alternativa a considerar a nível de desempenho e paralelamente possam da mesma forma corrigir todas as falhas que possam surgir, é necessário utilizar um valor que esteja directamente relacionado com o valor de altura resultante de uma diminuição do nível de detalhe. O erro geométrico (ver **5.4.1**) é à partida um excelente candidato já que representa efectivamente a máxima variação de altura associada a uma mudança de nível de detalhe. De facto, esta é a aproximação utilizada no algoritmo de Rendering Very Large, Very Detailed Terrains descrito em **4.7** e permite obter resultados bastantes satisfatórios. No entanto, é possível optimizar um pouco mais. Basta pensar que se utilizarmos o erro geométrico do bloco, este corresponde à máxima variação de altura entre dois níveis de detalhe, sendo esta variação calculada com base em todos os pontos de elevação desse bloco. Mas na realidade o que interessa são apenas os valores de elevação que delimitam o bloco. Na realidade são apenas esses que são considerados no método desenvolvido no cálculo da altura das *skirts*. Esse cálculo é efectuado da mesma forma que o descrito em **5.4.1** para o erro geométrico só que neste caso envolve apenas os valores que delimitam o bloco. Obtendo a altura das *skirts* desta forma, está-se a optimizar alguns casos. Basta pensar, por exemplo, num bloco que tenha um declive de elevação no centro, ou seja, os valores de elevação mais baixos estão no centro do bloco. Nessa situação, utilizar o erro geométrico do bloco numa mudança de nível de detalhe seria considerar todos os valores de elevação, o que daria origem a *skirts* demasiado altas quando não eram nesse caso necessárias. Esta situação é representada na **Figura 5-7** onde se pode observar a diferença entre as duas abordagens: à esquerda são bem visíveis as *skirts* resultantes da utilização do erro geométrico do bloco para a altura; à direita, as *skirts*, muito embora presentes, são praticamente invisíveis pois

o erro geométrico resultante de uma transição de nível de detalhe, se levarmos em consideração apenas os valores de elevação que delimitam o bloco, é muito pequeno.



Valores de elevação do bloco Valores de elevação que delimitam o bloco

Figura 5-7: Diferença de altura das *skirts* dependente dos valores considerados.

5.4.3. Dados de Oclusão

Os dados de oclusão armazenados em cada nó permitem, em tempo de execução, construir o horizonte utilizado no algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**). Estes dados são essencialmente o plano dos mínimos quadrados, o plano máximo e o plano mínimo bem como um valor de erro associado à aproximação. O plano dos mínimos quadrados é o plano que melhor minimiza a distância entre si e todos os pontos que representa, o que neste caso corresponde a todos os pontos de elevação de cada um dos blocos da *quadtree*. Este é obtido por intermédio da **Equação 5-3** que se repete aqui por questões de referência dado que discutida em maior detalhe em **4.6.3**.

$$\begin{bmatrix} \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i y_i & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i y_i & \sum_{i=1}^{n} y_i^2 & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} x_i z_i \\ \sum_{i=1}^{n} y_i z_i \\ \sum_{i=1}^{n} z_i \end{bmatrix}$$

x_i: O valor da coordenada x_i.

y_i: O valor da coordenada y_i.

z_i: O valor da coordenada z_i.

Equação 5-3: Cálculo do plano dos mínimos quadrados [51].

Como este plano não é mais do que uma aproximação, é necessário determinar o valor do erro associado a essa aproximação de modo a que se possa obter o limite máximo e o limite mínimo associados a esse erro. É a partir do valor desse erro que se obtêm então o plano máximo e o plano mínimo que delimitam a aproximação. Para se obter o valor do erro percorrem-se todos os pontos de elevação do bloco e calcula-se o valor máximo e mínimo de elevação de acordo com a normal do plano dos mínimos quadrados. O valor

máximo e mínimo são depois subtraídos ao valor da distância, medida ao longo da normal do plano, até à sua origem. O máximo absoluto dessas duas diferenças vai constituir então o valor do erro utilizado na construção dos planos mínimos e máximos. Significa isto que a distância entre o plano dos mínimos quadrados e o plano mínimo é igual à distância entre o plano dos mínimos quadrados e o plano máximo, tal como é possível observar na **Figura 5-8** onde estão representados os três planos que delimitam, neste caso, um bloco de terreno de 33×33. Em tempo de execução, o plano mínimo vai ser utilizado para adicionar a contribuição deste bloco ao horizonte e o plano máximo para verificar se o bloco está abaixo ou acima do horizonte, o que permite determinar respectivamente se está oculto por outro bloco ou não. No que diz respeito ao valor do erro este vai ser utilizado na métrica de erro que determina neste algoritmo a condição de paragem da recursão a efectuar na *quadtree* tal como é descrito em **5.6.4**.

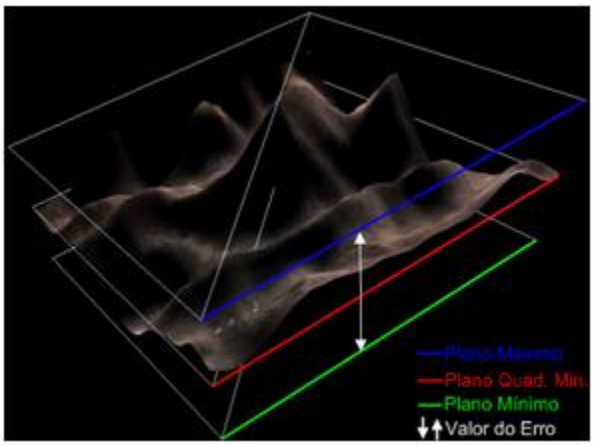


Figura 5-8: Plano dos mínimos quadrados, plano mínimo e plano máximo.

5.5. Construção da Malha Triangular

A construção da malha triangular tem de ter em consideração vários factores de modo a que seja possível concretizar todas as funcionalidades previstas (ver **5.1**). Destas funcionalidades as que mais influenciam este processo são:

- A utilização de *vertex textures*.
- Os dois tipos de primitivas suportadas: listas e tiras de triângulos indexados (ver **3.1.1** e **3.1.3**).
- A utilização de *skirts* (ver **3.4.4.1**) para o tratamento de falhas entre blocos adjacentes de diferentes níveis de detalhe.
- A optimização de sequências de triângulos para maximizar a utilização da *vertex cache*.
- O algoritmo de geração de terrenos em tempo real considerado, que pode ser neste caso o Geomipmapping (ver **4.4**), o GPU Terrain Rendering (ver **4.8**) ou uma aproximação puramente Brute Force.

Estas opções têm impacto sobretudo no modo como é construída a lista ou a tira de triângulos e a lista de índices associadas à geometria. Assim, a primeira funcionalidade a ter em consideração é a utilização, ou não, de *vertex textures*. O objectivo é suportar de uma forma transparente uma abordagem ou outra, isto é, com e sem *vertex textures*. A solução para este problema foi puramente tecnológica, pois baseou-se na possibilidade de se enviar múltiplas *streams* de vértices para a placa gráfica funcionalidade que está disponível no XNA [220] e também no DirectX [45] e no OpenGL [150]. A possibilidade de se enviar várias *streams* de vértices não é mais do que uma forma de separar o *input* de várias listas de vértices que são depois unificadas ao nível do *vertex shader*. Como o objectivo era unificar de alguma forma as duas abordagens, esta foi a solução perfeita para o problema já que permite enviar a estrutura geométrica de cada bloco numa *stream* e os valores de elevação numa segunda *stream*. Ora, esta capacidade é precisamente o que se pretende pois, quando os valores de elevação forem fornecidos numa textura, basta não enviar a segunda *stream*, pelo que a especialização ocorre apenas ao nível do *vertex shader*, isto é, diferentes métodos a esse nível para cada um dos casos. A principal consequência desta abordagem é assim a separação do *input* em valores que representam a geometria do terreno e em valores que representam a elevação numa abordagem equivalente à utilizada no algoritmo de GPU Terrain Rendering (ver **4.8**). Para isso utilizam-se valores canónicos, isto é, entre 0 e 1 para a posição dos vértices tirando-se partido do facto de qualquer bloco de terreno poder ser descrito por um factor de escala e por uma translação tal como é referido em **4.8.1**. Isto significa que é apenas ao nível do *vertex shader* que é aplicado o factor de translação e escala, responsáveis pela posição e pela dimensão do bloco respectivamente (ver **Figura 4-34**). Esse factor de escala é constante no caso do Geomipmapping e varia no caso do GPU Terrain Rendering. Consegue-se assim através da alteração de parâmetros do *vertex shader* a unificação dos dois algoritmos pelo menos ao nível da representação geométrica.

Uma consequência directa desta forma de representar a geometria é ficar-se independente do número de vértices do terreno, ou seja, torna-se possível mudar a dimensão do terreno sem ser necessário reconstruir a lista de vértices o que permite a modificação desta propriedade em tempo de execução. Efectivamente, mantém-se o número de vértices e o que muda é apenas a distância entre cada um deles na malha triangular. A principal diferença entre a abordagem descrita e a proposta no GPU Terrain Rendering é precisamente as *streams*, pois nesse algoritmo considera-se apenas o uso de *vertex textures*, pelo que é necessária apenas uma *stream* de vértices com a estrutura geométrica. Neste caso, como se pretende uma implementação que funcione em placas gráficas com e sem suporte para o *shader model* 3.0, esta foi a solução encontrada para suportar as duas variantes. Este processo está representado na **Figura 5-9** onde é identificado o conteúdo das duas *streams* utilizadas nesta implementação: na primeira *stream* são enviados três valores: o valor da coordenada x, o valor da coordenada y e o valor da coordenada z. As coordenadas x e z são canónicas, isto é podem assumir valores entre 0 e 1. A coordenada y não especifica ao contrário do que seria de esperar a altura, pois tal como já foi dito os valores de elevação são enviados numa segunda *stream*. O y pode tomar apenas os valores -1 ou 1 e indica se o valor de elevação enviado na segunda *stream* faz parte do terreno (1) ou é uma *skirt* (-1) utilizada na correcção de falhas entre blocos adjacentes de diferentes níveis de detalhe. Em relação à segunda *stream*, o primeiro valor corresponde à elevação propriamente dita e o segundo permite efectuar o

vertex morphing (ver **3.4.4.2**) entre diferentes níveis de detalhe, sendo descrito em detalhe em **5.5.2.1**. Note-se que, a lista de valores de elevação terá de ter no mínimo tantos valores quantos os valores de elevação do terreno e a lista que descreve a estrutura geométrica tantos valores quanto o número de vértices do bloco. Por exemplo, num terreno de 1024×1024 com 33×33 vértices por bloco temos de utilizar $33 \times 33 = 1089$ vértices para descrever a estrutura geométrica do bloco e $1024 \times 1024 = 1048576$ valores de elevação.

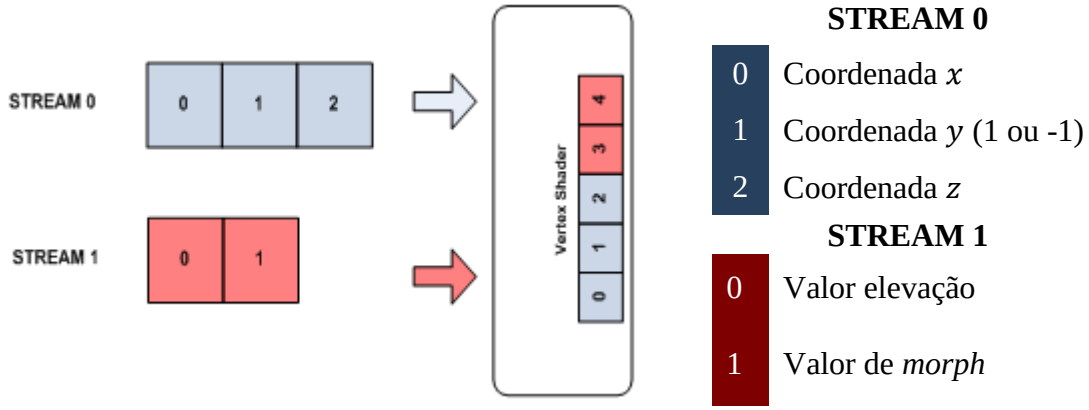


Figura 5-9: As duas *vertex streams* e a sua unificação ao nível do *vertex shader*.

5.5.1. Estrutura Geométrica

Como já foi referido, a primeira *stream* descreve a estrutura geométrica de um bloco de terreno. Para construir essa estrutura é gerada uma lista de vértices. Para exemplificar este processo considere-se um bloco de dimensão 9×9 num terreno de dimensão igual ou superior a 9×9. Para construir a representação geométrica de um bloco desta dimensão são necessários 81 vértices que são armazenados de forma sequencial numa lista de vértices da esquerda para a direita e de baixo para cima, tal como está representado na **Figura 5-10** onde se ilustra a título de exemplo o conteúdo de 5 dos 81 elementos da lista de vértices. Como se pode verificar, os valores das coordenadas x e z são canónicos e o valor da coordenada y é igual a 1 o que significa, como já foi referido, valores de elevação do terreno pelo que a figura representa um bloco sem *skirts*.

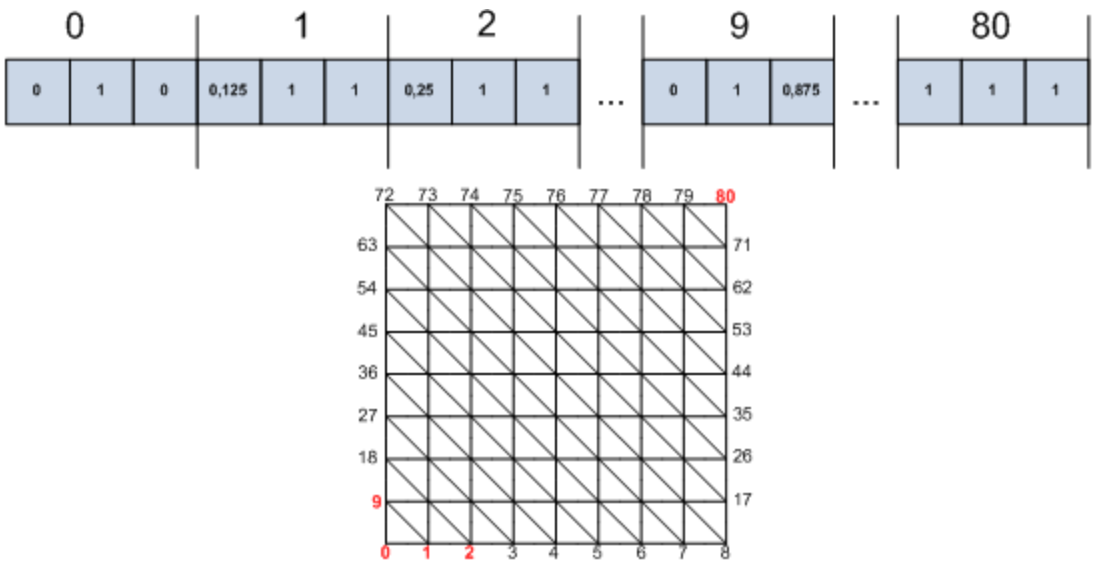


Figura 5-10: Ordem considerada na lista de vértices.

Para interligar os vértices é necessária uma lista de índices, cujo objectivo é definir cada um dos triângulos que vão constituir a malha triangular. A ordem pela qual cada um desses triângulos é construído é ilustrada na **Figura 5-11** para um bloco 9×9.

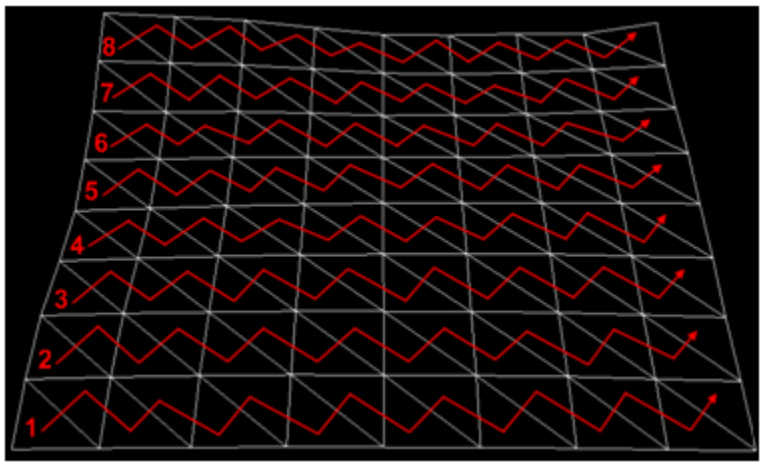


Figura 5-11: Sequência de triângulos utilizada na construção de um bloco.

No caso de a primitiva ser uma lista de triângulos (ver **3.1.1**) cada um dos triângulos é especificado na lista de índices de forma individual pelo que nesse caso o processo é simples. No caso das tiras de triângulos (ver **3.1.3**) o processo é um pouco mais complicado dado que é necessário utilizar triângulos degenerados (ver **3.1.3**) para ligar cada uma das tiras. A **Figura 5-12** ilustra através de números a ligação entre as tiras, que ocorre do lado direito para o lado esquerdo da figura entre triângulos identificados com o mesmo número. Cada um deles pertence a uma tira distinta, pelo que são necessários dois triângulos degenerados entre estes triângulos. Desta forma torna-se possível continuar a tira e paralelamente garantir que o bloco pode ser enviado para *rendering* apenas numa única chamada à primitiva.

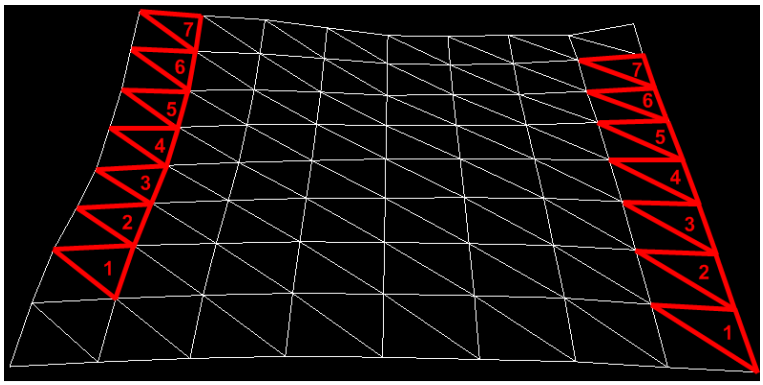


Figura 5-12: Ligação entre tiras de triângulos por meio de triângulos degenerados.

Na construção da malha triangular existem dois factores adicionais que têm de ser levados em consideração: as *skirts* e a optimização da estrutura geométrica para a *vertex cache* da placa gráfica. Cada um deles vai ser abordado em detalhe em **5.5.1.1** e **5.5.1.2**, pois tem um impacto profundo na lista de vértices e/ou na lista de índices gerada.

5.5.1.1. Skirts

A possibilidade de colmatar as falhas resultantes de diferentes níveis de detalhe adjacentes por intermédio de *skirts*, isto é, saias poligonais que envolvem cada um dos blocos, implica um conjunto de triângulos adicionais. Assim, é necessário ter isso em consideração na lista de vértices e na lista de índices resultantes. No caso da lista de vértices, na **Figura 5-9** já tinha sido possível verificar que um dos valores enviado na primeira *stream* tem precisamente como objectivo definir se o vértice se refere a um valor da grelha propriamente dita ou a um valor da *skirt* poligonal. Efectivamente, se o valor for negativo, indica uma *skirt*, o que permite em tempo de execução distingui-la de um valor de elevação do terreno ao nível do *vertex shader*. Desta forma, no caso de optarmos por este método para colmatar as falhas entre blocos adjacentes, o número de vértices terá de ser superior. Mais precisamente, são necessários 113 vértices para um bloco 9×9, isto é, 81 da estrutura propriamente dita sendo os restantes para construir a saia poligonal que delimita o bloco (note-se que os vértices dos cantos são partilhados). A ordem pela qual são definidos os triângulos na lista de índices é representada na **Figura 5-13** onde como se pode verificar a especificação dos triângulos da *skirt* começa no último vértice do bloco prosseguindo em torno deste.

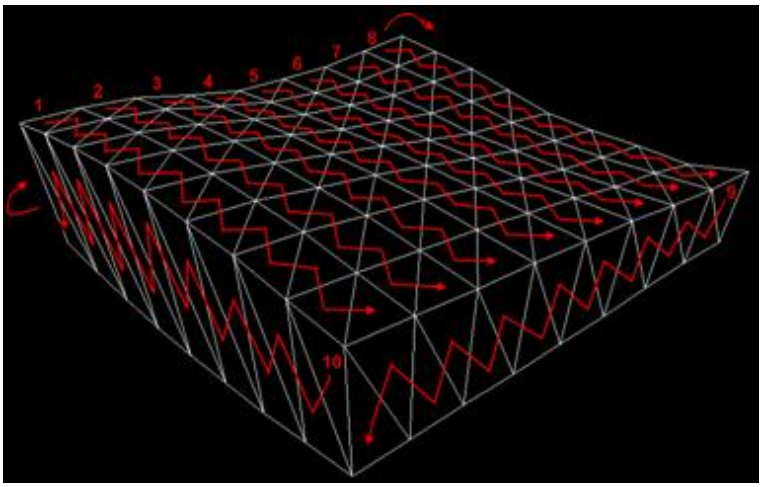


Figura 5-13: Sequência de triângulos com *skirts* utilizada na construção de um bloco.

Independentemente da primitiva utilizada, o objectivo é utilizar uma única chamada para o bloco e para as *skirts* associadas a esse bloco. Mais uma vez esse processo é muito mais simples se a primitiva for uma lista de triângulos, pois os triângulos são especificados individualmente. O mesmo já não ocorre nas tiras de triângulos onde os triângulos estão dependentes uns dos outros. Para isso é necessário utilizar novamente triângulos degenerados, neste caso apenas um que liga o último triângulo da última tira do bloco ao primeiro triângulo da saia poligonal, o que de acordo com a **Figura 5-14**, corresponde aos triângulos numerados com 8.

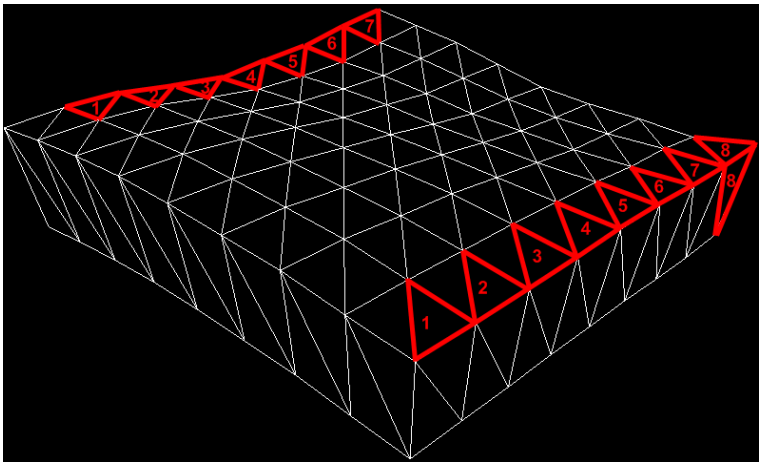


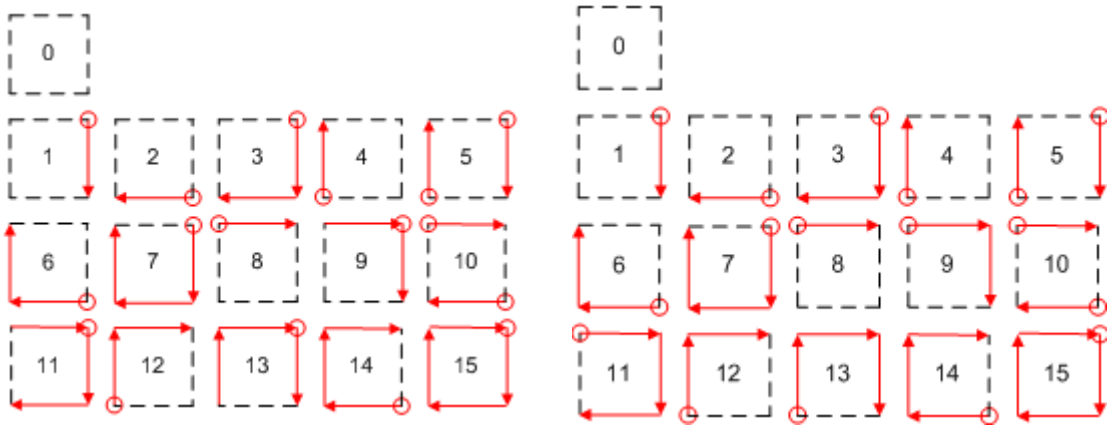
Figura 5-14: Ligação com a *skirt* por meio de triângulos degenerados.

Apesar de as *skirts* permitirem corrigir de uma forma simples o problema das falhas, têm algum impacto no desempenho que se manifesta em primeiro lugar ao nível do *fill rate* e em segundo lugar no número de polígonos adicionais que implicam. Para minimizar o impacto das *skirts* a esse nível teve-se em consideração que:

- As *skirts* devem ter a altura mínima necessária para cobrir qualquer falha que possa aparecer, mas não mais.
- Nem todas as *skirts* que envolvem o bloco são visíveis para um determinado ponto de vista pelo que apenas as necessárias devem ser enviadas para *rendering*.

Desta forma, foi necessário actuar a dois níveis: ao nível da altura e ao nível da visibilidade da *skirt* para um determinado ponto de vista. No que diz respeito à altura, o valor utilizado vai corresponder, tal como já tinha sido referido em **5.4.2**, ao valor do erro geométrico de um determinado bloco de terreno calculado com base nos valores de elevação que delimitam esse bloco. Isto tanto no algoritmo de Geomipmapping como no algoritmo de GPU Terrain Rendering. Em relação à visibilidade das *skirts* cedo se chega à conclusão que para se conseguir efectuar algo a esse nível é necessário em primeiro lugar ter a capacidade de se seleccionar as *skirts* que se pretende por bloco. Para isso é preciso actuar ao nível da lista de índices gerando-se todas as configurações possíveis. Isto é, para cada uma das configurações gera-se os índices associados ao bloco propriamente dito e especificamente os índices associados a essa configuração. Na **Figura 5-15** estão representadas as 16 configurações consideradas. As linhas a cheio indicam a presença de uma *skirt* e as linhas a tracejado o contrário. Cada um dos círculos indica onde a *skirt* se inicia em cada uma das primitivas consideradas. Por exemplo, a configuração 15 representa uma situação em que o bloco tem todas as *skirts*. Nesta configuração a primeira *skirt* tem início no canto superior direito, o que está de acordo com a representação dessa mesma situação na **Figura 5-14**. No caso das listas de triângulos, a configuração é em tudo semelhante, a única coisa que varia é a posição no bloco onde se começam a adicionar os triângulos das *skirts*. Nesse caso, para cada configuração navega-se em torno do bloco no sentido dos ponteiros do relógio, adicionando-se a *skirt* respectiva em cada um dos lados. Comparando as duas situações representadas na figura, distinguem-se três casos, o 9 o 11 e o 13. Como se pode verificar, existe uma diferença no ponto onde se inicia a *skirt*. Efectivamente quando a primitiva é uma lista de triângulos a construção da *skirt* começa sempre no canto superior direito, o mesmo já não ocorre com as tiras de triângulos. A razão para esta diferença está relacionada com os triângulos degenerados que uma tira de triângulos implica e com o facto de as configurações 9, 11 e 13 permitirem reduzir o número desses triângulos. Tomando como exemplo o caso do bloco 9, se a tira tivesse início no canto superior direito, seria necessário um triângulo degenerado a ligar o último triângulo do bloco ao primeiro triângulo da *skirt* e dois triângulos degenerados para ligar a tira vertical direita à tira horizontal superior. Começando a tira no canto superior esquerdo, basta um triângulo degenerado a ligar o último triângulo do bloco ao primeiro triângulo da *skirt* evitando-se desta forma uma interrupção na *skirt* que implicaria a utilização de mais dois triângulos degenerados.

Com esta estratégia é possível seleccionar apenas as *skirts* que intersectam o *frustum*, enviando-se apenas essas para *rendering*, o que permite diminuir o número de polígonos, por um lado, e o *fill rate*, por outro. Tal será discutido em detalhe em **5.6.2.2**.



Listas de Triângulos Tiras de Triângulos

Figura 5-15: Diferentes configurações das *skirts*.

Ainda no que diz respeito à lista de índices, há que considerar agora a influência dos algoritmos de nível de detalhe na sua construção. No caso do GPU Terrain Rendering como o número de vértices é sempre o mesmo e o que varia efectivamente é o espaçamento entre eles basta construir uma lista de índices que representa um bloco de um determinado número de vértices. O mesmo já não se passa no algoritmo de Geomipmapping já que este efectua redução do nível de detalhe pela diminuição do número de triângulos enviados para *rendering* em cada um dos blocos. Essa diminuição é feita pela redução do número de vértices abrangidos pela lista de índices. Por exemplo, de um bloco 3×3 para um bloco 2×2, tal como acontece com alguns dos blocos na **Figura 5-1**. Nesse caso armazena-se sequencialmente na lista de índices os índices correspondentes aos vários tipos de blocos, por exemplo, para um bloco 5×5 armazena-se a sequência de índices necessária para construir blocos 5×5, seguida da sequência para blocos 3×3 e, finalmente, para blocos 2×2. Basicamente do maior detalhe para o menor detalhe sendo a selecção do nível de detalhe controlada por uma métrica de erro. Obviamente em cada um destes níveis ter-se-á em conta cada uma das configurações alternativas resultantes da utilização de *skirts* no tratamento de falhas entre blocos adjacentes de diferentes níveis de detalhe.

5.5.1.2. Vertex Cache

Na construção da estrutura geométrica existe ainda um outro factor a ter em consideração que corresponde à taxa de utilização da *cache* da placa gráfica que lhe está associada. Esta *cache* tal como já foi referido em **3.1** existe na maioria das placas gráficas mais recentes tendo como objectivo evitar o processamento repetido de vértices. Para tirar proveito dessa *cache* é possível alterar um parâmetro que indica basicamente o comprimento máximo de uma sequência de vértices ao longo da coordenada x. O objectivo é maximizar a reutilização dos vértices. Isto é, se o número de vértices tiver um comprimento igual ou inferior a metade do tamanho da *cache* quando a segunda linha de triângulos se iniciar muitos dos vértices vão estar ainda em *cache* não sendo por isso necessário efectuar de novo a sua transformação. Assim, e assumindo que se tem de

alguma forma a informação do tamanho da *cache*, pode-se fornecer um valor para esse parâmetro que maximize o desempenho pela reutilização do maior número de vértices possível. Para concretizar esta funcionalidade é necessário ordenar os triângulos de uma forma diferente. Neste caso basta fazer linhas de triângulos mais curtas, o que no caso de a primitiva ser uma lista de triângulos não apresenta nenhuma dificuldade adicional, basta apenas mudar a ordem dos índices. No caso das tiras de triângulos a ordem é igual mas é necessário utilizar mais triângulos degenerados já que são também necessárias mais tiras tal como está bem patente na **Figura 5-16** que ilustra a construção de uma malha poligonal optimizada para uma *cache* de 6 vértices dado que o número de vértices máximo numa linha é 3 ou seja este será o valor do parâmetro a configurar para a dimensão da *cache*.

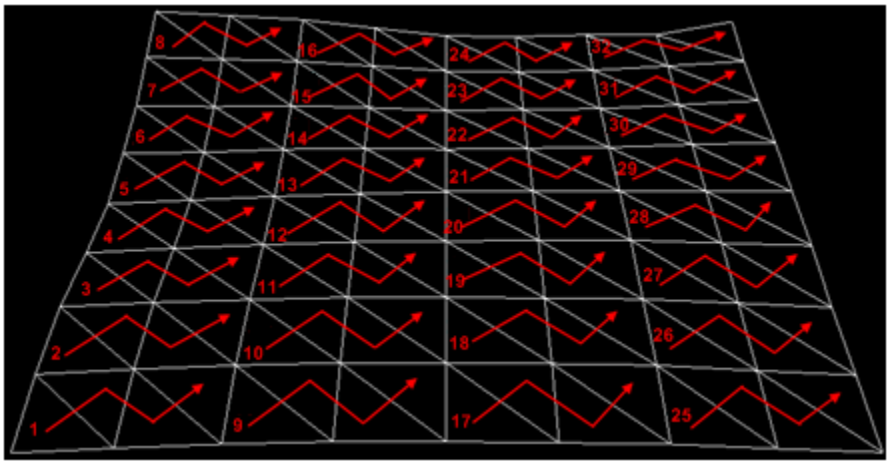


Figura 5-16: Organização de triângulos orientada à *vertex cache*.

5.5.2. Dados de Elevação

Esta implementação permite o envio dos valores de elevação de duas formas distintas. Se as *vertex textures* forem suportadas, podemos construir apenas a lista de vértices e a lista de índices que descrevem a geometria do terreno enviando os valores de elevação numa textura que será endereçada no *vertex shader*. Se não existir suporte para *vertex textures* é possível de acordo com a abordagem adoptada, construir uma nova lista de valores de elevação que é depois associada à segunda *stream* e unificada ao nível do *vertex shader* com a *stream* que contêm a estrutura geométrica. Ambas as abordagens têm vantagens e desvantagens: se optarmos por enviar uma segunda *stream* o volume de dados enviados para a placa gráfica é muito maior, no entanto, tem-se mais flexibilidade, já que se torna possível adicionar facilmente outro valor à *stream* que esteja directamente relacionado com o valor de elevação. Se se optar por *vertex textures*, basta enviar apenas uma *stream* com a estrutura geométrica, o que diminui a quantidade de dados enviados mas por outro lado torna mais complicado adicionar outro valor, pois para isso ou este é armazenado na mesma textura ou numa textura adicional. Este problema das *vertex textures* é especialmente relevante pois é mesmo necessário um valor adicional. Este

valor é utilizado (ver **Figura 5-9**) para suportar o *vertex morphing* (ver **3.4.4.1**), uma funcionalidade importante na medida em que permite eliminar o efeito de *popping*. Este valor está directamente relacionado com os valores de elevação, já que é o resultado de uma interpolação dos valores de elevação dos vértices vizinhos do qual está directamente dependente, sendo por isso calculado para cada um dos vértices. O objectivo é determinar a elevação que estaria associada a um determinado vértice se este desaparecesse fruto de uma diminuição do nível de detalhe, isto de modo a que seja possível transitar para esse valor de elevação progressivamente diminuindo desta forma o efeito de *popping*. A forma como é construído este valor de *morph* é discutida em detalhe em **5.5.2.1**. As duas formas de enviar o valor de elevação e o valor de *morph* são discutidas em detalhe em **5.5.2.2** e **5.5.2.3**, respectivamente.

5.5.2.1. Valores de Morph

Para eliminar o indesejado efeito de *popping* (ver **3.4.4.2**) que ocorre numa mudança de nível de detalhe uma das hipóteses é efectuar o Geomorphing. Este permite a transição entre dois níveis de detalhe de uma forma praticamente indetectável, pelo que foi uma das funcionalidades contemplada. No entanto, para que o Geomorphing seja possível, é necessário, tal como foi referido em **5.5**, enviar mais um valor por cada um dos vértices: o valor de *morph*. Na construção deste valor há que entender primeiro o que é necessário para efectuar o *morphing* de cada um dos vértices. Em primeiro lugar, há que determinar o último nível de detalhe em que cada um dos vértices aparece. Em segundo lugar, calcular o valor de elevação por interpolação dos vértices vizinhos, isto de modo a determinar a posição desse vértice no próximo nível de detalhe. O primeiro é essencial na medida em que apenas alguns vértices desaparecem na transição entre níveis pelo que o *morphing* terá de ser aplicado apenas a esses vértices. O segundo define qual a posição para onde o vértice terá de transitar antes de desaparecer. Em tempo de execução, o objectivo será efectuar uma interpolação linear entre o valor de elevação de cada um dos vértices considerados e o valor de elevação desses mesmos vértices no próximo nível de detalhe (que é, como foi referido, obtido para cada um deles por interpolação dos vértices vizinhos). Deste modo, aparentemente, são necessários dois valores por vértice: o último nível de detalhe em que o vértice aparece e o valor de elevação desse vértice no nível de detalhe seguinte. No entanto basta um, já que o valor de elevação obtido por interpolação é expresso na forma canónica, podendo por isso ser armazenado na parte decimal do valor de *morphing*. Resta a parte inteira onde é armazenado o último nível em que o vértice aparece. Por exemplo, 2.5 é um valor de *morphing* em que 2 é o último nível em que o vértice aparece e 0.5 é o valor de elevação obtido por interpolação linear dos vértices vizinhos expresso na forma canónica. Considerando neste exemplo que o valor de elevação real é 0.1, o objectivo será fazer em tempo de execução uma interpolação linear entre 0.1 e 0.5 entre o nível de detalhe 2 e o nível de detalhe 3.

Temos agora de considerar o impacto dos dois algoritmos de nível de detalhe, o Geomipmapping (ver **4.4**) e o GPU Terrain Rendering (ver **4.8**) na construção do valor de *morph*. Para isso considere-se a título de exemplo um terreno 9×9 e um bloco de dimensão 5×5. Tal como já foi referido, no Geomipmapping a diminuição do nível de detalhe ocorre ao nível do bloco pelo que para um bloco 5×5 temos de considerar 3 níveis de detalhe, respectivamente, 5×5, 3×3 e 2×2, e quatro blocos enviados para

rendering em qualquer um dos casos. No caso do GPU Terrain Rendering a diminuição do nível de detalhe processa-se pela transformação de 4 blocos num único bloco que ocupa o mesmo espaço, pelo que se actua ao nível do número de chamadas à primitiva. Por isso, nesse caso, para um bloco 5×5 temos apenas 2 níveis, o primeiro em que temos 4 blocos 5×5 e o segundo em que temos 1 bloco 5×5. As duas abordagens são representadas na **Figura 5-17** onde para cada vértice se indica o valor do último nível em que aparece, ou dito de outra forma, o nível em que o vértice fica sujeito ao efeito de *morphing*.

Como se pode verificar no caso do Geomipmapping temos sempre quatro blocos sendo a diminuição de nível de detalhe feita por bloco, pelo que os três níveis se aplicam individualmente a cada um dos quatro blocos. No caso do GPU Terrain Rendering são possíveis apenas dois níveis de detalhe já que no segundo nível o terreno tem já a dimensão do bloco. Tanto num caso como no outro os vértices que desaparecem são precisamente os afectados pelo *morphing*, assim, por exemplo, na transição do nível 0 de detalhe para o nível 1 desaparecem todos os vértices marcados com 0, logo entre os dois níveis de detalhe são apenas esses que se movem progressivamente para a posição que teriam se existissem no nível seguinte. O mesmo se aplica tanto num algoritmo como noutro entre outros níveis de detalhe.

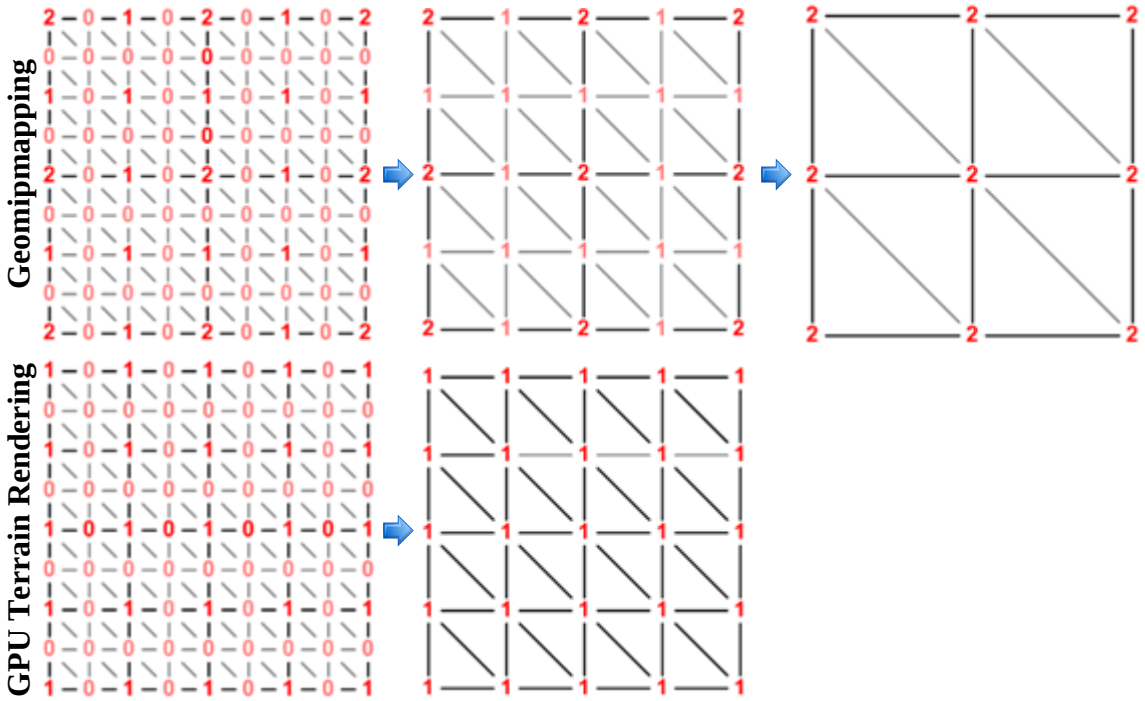


Figura 5-17: Níveis em que os vértices desaparecem nos dois algoritmos.

5.5.2.2. Multi-Stream

Esta opção é suportada pelo maior número de placas gráficas, no entanto, tal como foi referido, é também a que implica o envio do maior número de dados. Por um lado, da lista de vértices com a estrutura geométrica do bloco e, por outro, da lista com os valores

de elevação. Assim, ao contrário do que acontece com as *vertex textures* (ver **5.5.2.3**) onde se envia apenas a lista de vértices com a estrutura geométrica, aqui é necessário enviar uma lista adicional com os valores de elevação. Basta pensar num terreno de 1025×1025 considerando 8 bytes para cada ponto (o valor de elevação e o valor de *morph* assumindo 4 bytes para cada um) para se chegar a uma lista com pelo menos 8 Mbytes. Por outro lado, tendo em consideração que em tempo de execução as duas *streams* vão ser unificadas (ver **Figura 5-9**) e que o *rendering* é feito por bloco, terá de existir uma correspondência entre os dados de elevação e os dados geométricos de cada porção do terreno. Além disso, como todos os valores de elevação estão armazenados numa única lista é necessário identificar a sequência de valores de elevação que correspondem a cada bloco. Para esse efeito determina-se qual o índice da lista onde tem início a descrição dos valores de elevação do bloco a desenhar.

Há ainda a salientar que a quantidade de valores de elevação a tratar varia de acordo com o algoritmo. Tal como foi referido em **5.3** os dois algoritmos de nível de detalhe considerados, o Geomipmapping e o GPU Terrain Rendering diferem essencialmente no tamanho dos blocos enviados para *rendering*. O Geomipmapping implica blocos sempre do mesmo tamanho diminuindo-se o nível de detalhe pela alteração da conectividade dos vértices de forma a se considerar menos triângulos. No GPU Terrain Rendering os blocos têm sempre o mesmo número de vértices mas o tamanho varia, tal como se pode verificar na **Figura 5-1**. Isto significa que, para existir neste caso uma correspondência entre os valores de elevação e a porção do terreno a enviar para *rendering* é necessário armazenar na lista de valores de elevação a descrição de todos os blocos em cada um dos níveis de detalhe. Mais precisamente são armazenados na lista de valores de elevação os blocos correspondentes ao nível 0, isto é os que têm mais detalhe, depois os do nível 1, ou seja aqueles que combinam quatro blocos do nível de detalhe anterior, e assim sucessivamente. Este conceito é ilustrado na **Figura 5-18** onde é feita uma comparação entre as listas necessárias para armazenar os dados de elevação no algoritmo de GPU Terrain Rendering, no algoritmo de Geomipmapping e na aproximação Brute Force. Em todos os casos considera-se um terreno de dimensão 9×9 e um bloco de dimensão 3×3. Tal como se pode verificar para o GPU Terrain Rendering são necessários 3 níveis para armazenar os dados de elevação. Note-se que cada um dos blocos tem sempre um número de vértices igual (3×3) o que muda efectivamente é a área de terreno correspondente. Como se pode facilmente concluir o GPU Terrain Rendering implica uma lista de valores de elevação de maior dimensão, pelo que ocupa consequentemente mais memória do que o Geomipmapping e o Brute Force.

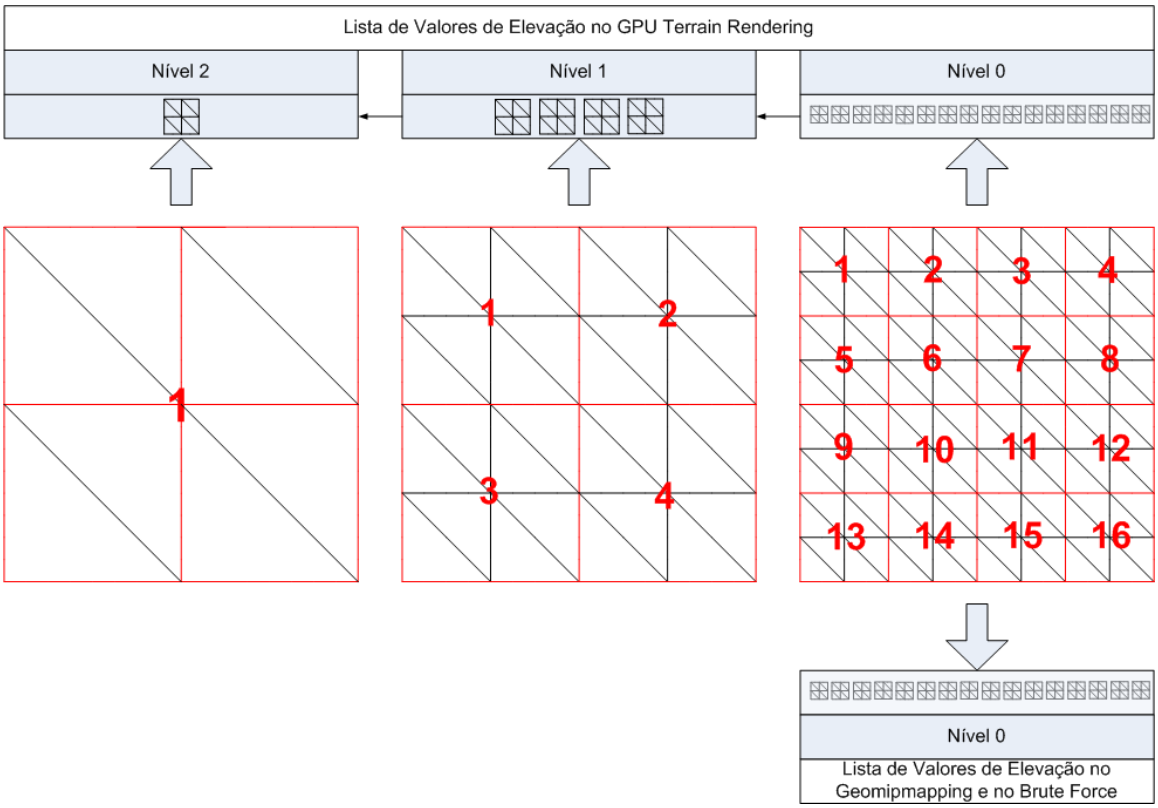


Figura 5-18: Comparação entre as listas de valores de elevação em cada algoritmo.

5.5.2.3. Vertex Textures

A possibilidade de se obter o valor de elevação a partir de uma textura ao nível do *vertex shader* faz com que deixe de ser necessário enviar os dados de elevação numa segunda *stream*, permitindo que os valores processados por cada vértice sejam muito menores. Infelizmente, nenhum dos formatos de texturas que suportam o *vertex texture fetch* nas placas gráficas baseadas em *shader model* 3.0 pode ser utilizado para os dados que têm neste caso de ser enviados: o valor de elevação e o valor de *morph*, ambos valores decimais. Na verdade só dois formatos de texturas é que podem ser utilizados ao nível do *vertex shader* numa placa gráfica com o *shader model* 3.0 [76]:

- Uma textura com um canal decimal de 32 bits (Single em XNA, D3DFMT_R32F em DirectX e GL_LUMINANCE_FLOAT32_ATI em OpenGL).
- Uma textura com quatro canais decimais de 32 bits (Vector4 em XNA, D3DFMT_A32B32G32R32F em DirectX e GL_RGBA_FLOAT32_ATI em OpenGL).

Estes são os formatos tipicamente suportados e, como facilmente se verifica, nenhum pode ser usado directamente para armazenar os dois valores decimais na textura. O primeiro formato não tem espaço suficiente, o segundo formato está fora de questão pois implicaria desperdiçar 50% do espaço da textura, utilizando-se apenas dois dos quatro

decimais. A solução foi utilizar duas texturas com o primeiro formato, uma com o valor de elevação e outra com o valor de *morphing*.

5.6. Rendering

Em tempo de execução, após a construção da *quadtree* (ver **5.4**) e da malha triangular do terreno (ver **5.5**) é iniciada a fase de *rendering* propriamente dita. O objectivo é a construção em cada *frame* da lista de blocos que se encontra visível para o ponto de vista corrente e, finalmente, o *rendering* de cada um dos blocos resultantes. Usando como ponto de referência a **Listagem 5-1** onde se descrevem as principais fases do método implementado verifica-se que na fase de *rendering* o primeiro passo é a obtenção da lista de blocos visíveis para um determinado ponto de vista, pelo que é necessário efectuar o *view frustum culling* (ver **3.3.2**) utilizando a *quadtree* como estrutura de suporte. É também neste passo que é determinado o tamanho e o nível de detalhe de cada um dos blocos, algo que vai depender directamente do algoritmo adoptado:

- Brute Force, o tamanho e o nível de detalhe são fixos.
- Geomipmapping, o tamanho é fixo e o nível de detalhe varia por bloco (isto é, o número de vértices diminui por bloco).
- GPU Terrain Rendering, o tamanho é variável e o nível de detalhe corresponde ao nível da *quadtree* em que o bloco se encontra.

A lista de blocos é construída com base num processo recursivo que percorre a *quadtree* a partir do nó raiz determinando os blocos que intersectam o *frustum* e a dimensão mais adequada de cada um deles. O resultado é uma lista de blocos ordenados segundo a proximidade à câmara, que permite no passo seguinte efectuar o *occlusion culling* (ver **3.3.3**) por intermédio do algoritmo de Terrain Occlusion Culling With Horizons (ver **4.6**). Nesse passo constrói-se progressivamente a linha de horizonte com base na informação armazenada por bloco, o que permite determinar à medida que essa linha vai sendo refinada se um determinado bloco está ou não oculto por outros blocos que foram entretanto adicionados ao horizonte e enviados para *rendering*.

5.6.1. Selecção do Bloco

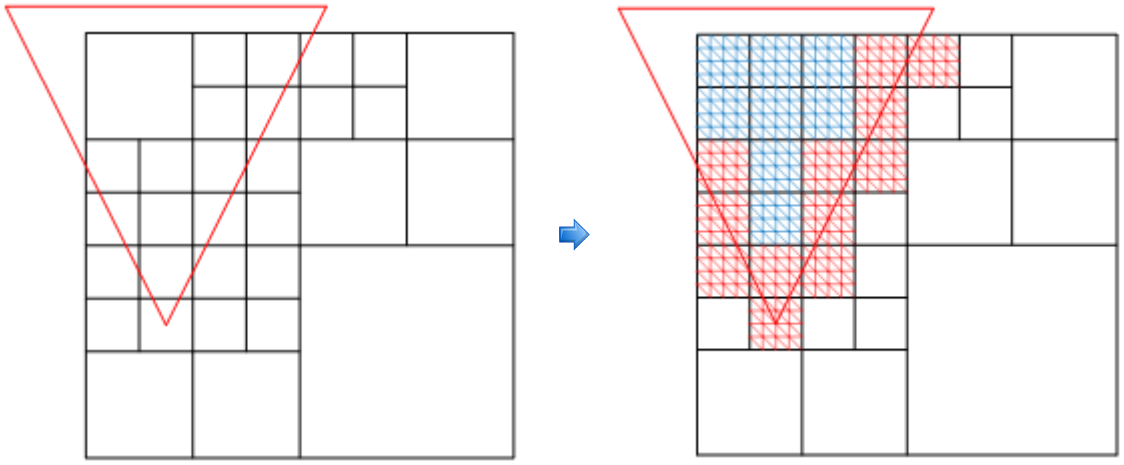
Como foi referido, o primeiro passo do processo de *rendering* corresponde à selecção dos blocos visíveis para o ponto de vista corrente. Para testar se o bloco pode ser adicionado é necessário verificar em primeiro lugar se este intersecta ou não o *frustum*. Se não intersectar, a recursão pára, o que significa que o bloco e todos os blocos que dele descendam são imediatamente rejeitados. Se pelo contrário, o bloco intersectar o *frustum*, então o adicionar ou não desse bloco à lista vai passar a depender das especificidades do algoritmo empregue. A forma como essa decisão é tomada é discutida em detalhe no contexto de cada um dos algoritmos considerados: o Brute Force em **5.6.1.1**, o Geomipmapping em **5.6.1.2** e o GPU Terrain Rendering em **5.6.1.3**. Se no contexto do algoritmo empregue o bloco não foi adicionado à lista de blocos, então analisa-se recursivamente cada um dos quatro blocos descendentes em que este se subdivide voltando a verificar-se a intersecção de cada um deles com o *frustum* e também se no contexto do algoritmo em causa devem ser adicionados à lista de blocos. Efectivamente,

a recursividade só é interrompida se o bloco não for visível ou se de acordo com o algoritmo empregue não for necessário analisar recursivamente os descendentes. O resultado final é uma lista de todos os blocos visíveis para o ponto de vista corrente ordenada pela distância à câmara.

5.6.1.1. Brute Force

Esta abordagem é a mais simples e foi desenvolvida com o intuito de servir de termo de comparação com técnicas mais complexas que efectuam a diminuição do nível de detalhe. Neste método, a dimensão do bloco fornecida como parâmetro determina o número máximo de chamadas à primitiva seleccionada (ver **5.5.1**) como aliás se pode verificar na **Figura 5-19** onde estão representados todos os nós da *quadtree* percorridos para o *view frustum*, considerado bem como a triangulação resultante num terreno de dimensão 33×33 com um bloco de dimensão 5×5. Aqui, para cada nó da *quadtree*, o processo recursivo para determinação dos blocos visíveis pode terminar de três formas diferentes: quando um bloco não intersecta o *frustum*, quando está completamente visível ou quando se chegou a um nó folha. Caso contrário, são analisados os nós do nível seguinte.

Se compararmos **a)** e **b)** da **Figura 5-19**, verificamos que existe uma diferença entre eles. Mais especificamente, na *quadtree* de pesquisa o nó superior esquerdo não está refinado em comparação com o mesmo nó na triangulação da *quadtree*. Isto porque, na verdade, a recursão termina nesse nó. No entanto, como se sabe à partida que o nó está totalmente contido dentro do *frustum*, mesmo tendo um número de vértices superior ao considerado (5×5) para cada um dos blocos, é possível adicionar imediatamente todos os blocos que contêm com esse número de vértices (quatro neste caso). Isto é possível porque estando os blocos totalmente contidos dentro do *frustum* têm de ser obrigatoriamente enviados para *rendering*. Numa situação em que o bloco considerado intersecta o *frustum* rejeita-se esse bloco o que implica mais um nível de recursividade. Esta optimização permite reduzir consideravelmente a profundidade da recursividade necessária para cada um dos blocos considerados e tem um impacto maior ainda se um bloco de um nível mais alto estiver totalmente contido no *frustum*, pois nesse caso todos os blocos da dimensão considerada são adicionados sem ser necessário efectuar os passos de recursividade que lhes estariam associados.



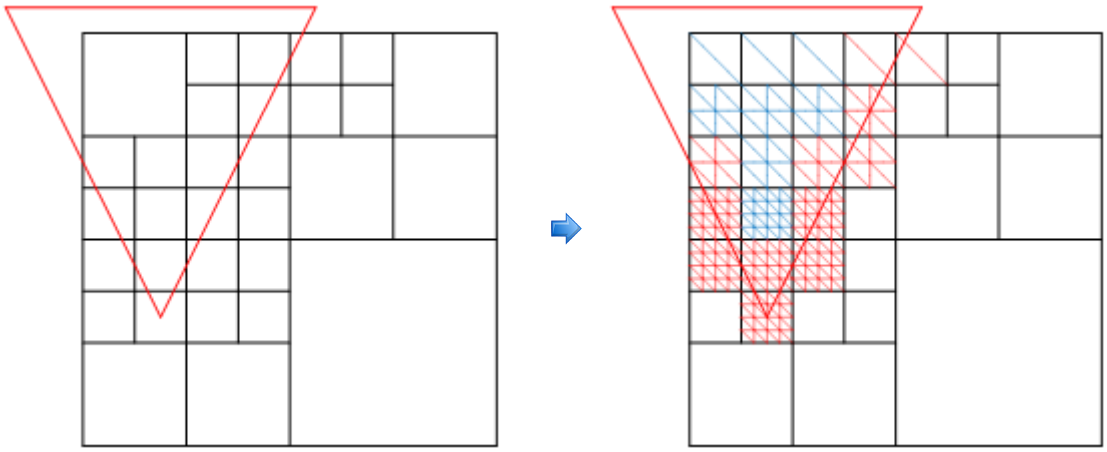
a) Pesquisa na *quadtree* b) Triangulação da *quadtree*

Figura 5-19: Pesquisa e triangulação no método de Brute Force.

5.6.1.2. Geomipmapping

No algoritmo de Geomipmapping, a aproximação seguida na pesquisa efectuada na *quadtree* é em tudo semelhante à utilizada no método de Brute Force (ver **5.6.1.1**), mas neste caso, para cada um dos blocos resultantes, o nível de detalhe vária de acordo com a métrica de erro utilizada. Tal como no Brute Force, o número de vértices do bloco determina o número máximo de chamadas à primitiva. Por outro lado, também aqui, numa situação em que um nó esta totalmente contido no *frustum* a recursão termina e os blocos da dimensão considerada contidos dentro desse nó são adicionados à lista de blocos a enviar para *rendering*. Assim, recorrendo à **Figura 5-20** onde estão representados em **a)** todos os nós da *quadtree* percorridos para o *view frustum* considerado, bem como em **b)**, a triangulação resultante, num terreno de dimensão 33×33 com um bloco de dimensão 5×5, podemos verificar que em comparação com a **Figura 5-19** a pesquisa efectuada na *quadtree* é exactamente a mesma, o que varia efectivamente é o nível de detalhe da triangulação resultante. Como se pode observar, à medida que a distância aumenta o nível de detalhe de cada bloco vai diminuindo. Para que tal seja possível é obtido para cada um dos blocos um nível de detalhe do conjunto de níveis de detalhe possíveis, estando o número de níveis dependente da dimensão do bloco. Por exemplo para um bloco 5×5, como o representado na figura, temos apenas três níveis de detalhe: 5×5, 3×3 e 2×2. Para determinar o nível de detalhe mais apropriado calcula-se numa fase de pré-processamento o erro geométrico de cada um dos níveis de detalhe considerados. O modo como esse cálculo é efectuado foi descrito em detalhe em **5.4.1** pelo que falta esclarecer o modo como o valor resultante é utilizado na selecção do nível de detalhe mais apropriado. Neste caso seguiu-se o algoritmo original pelo que se calcula uma constante C numa fase de pré-processamento, que tem em consideração o erro máximo em *pixels*, τ, desejado. É a partir dessa constante e do erro geométrico máximo armazenado no bloco que se obtém a distância mínima a partir da qual esse bloco deve ser seleccionado. A fórmula utilizada no cálculo da constante C está representada na **Equação 5-4** e a do cálculo da distância mínima na **Equação 5-5**. Estas

são efectivamente as mesmas equações apresentadas em **4.4.3**, só que desta vez de uma forma mais sucinta, voltando-se a repeti-las por uma questão de referência.



a) Pesquisa na *quadtree* b) Triangulação da *quadtree*

Figura 5-20: Pesquisa e triangulação no algoritmo de Geomipmapping.

$C = \frac{n}{\frac{\lvert t \rvert}{2 \cdot \tau}}{V_{res}}$	n: Representa o primeiro plano de corte (*near clipping plane*).
	t: A coordenada superior do plano de corte.
	τ: Representa o número máximo de *pixels* a partir do qual já não é permitida a mudança de nível.
	V_{res}: A resolução vertical em *pixels*.

Equação 5-4: Cálculo da constante C.

$D_n = \lvert \delta \rvert \cdot C$	δ: A maior diferença de altura de todos os vértices alterados no bloco, ou seja $max\{\delta_0, \dots, \delta_n\}$.
	C: Calculado na **Equação 5-4**.

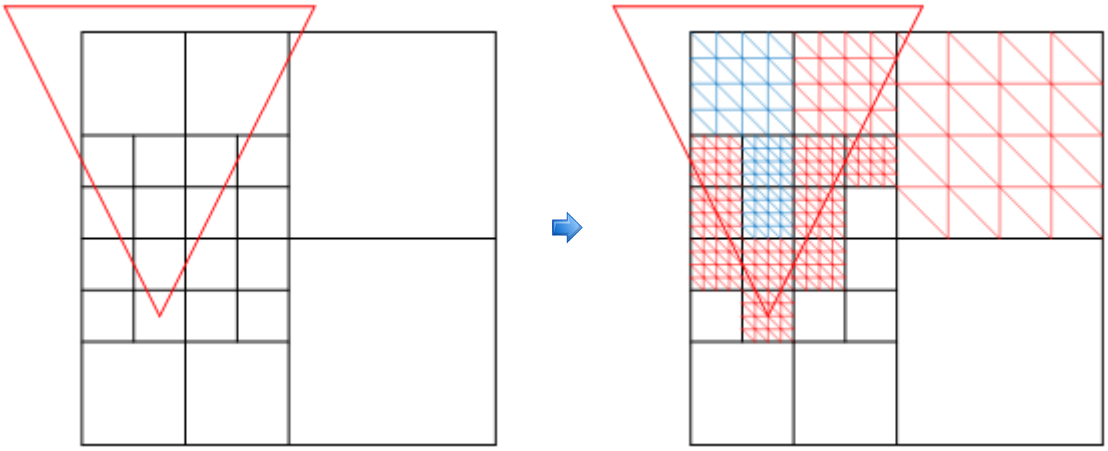
Equação 5-5: Cálculo da distância mínima para cada nível de detalhe considerado.

5.6.1.3. GPU Terrain Rendering

O algoritmo de GPU Terrain Rendering utiliza uma aproximação bastante diferente efectuando a diminuição do nível de detalhe pela utilização de blocos de maiores dimensões que são seleccionados de acordo com a métrica de erro considerada e que estão directamente relacionados com os blocos existentes nos diferentes níveis da *quadtree*. Esta técnica permite assim actuar ao nível do número de chamadas efectuadas à primitiva que passam deste modo a estar dependentes do ponto de vista. Note-se que, apesar da dimensão do bloco aumentar, o número de vértices mantém-se. O que acontece efectivamente é que a distância entre eles aumenta, uma consequência do factor de escala aplicado. Assim, mal um bloco de um determinado nível da *quadtree* satisfaça a métrica

de erro considerada, pode ser adicionado à lista de blocos a enviar para *rendering* parando-se aí a recursividade. Se compararmos com o Brute Force (ver **5.6.1.1**) e o Geomipmapping (ver **5.6.1.2**), também aí a recursividade pode parar antes de se atingir a dimensão do bloco obtendo-se nessa situação vários blocos da dimensão considerada. A diferença é que no GPU Terrain Rendering, para a mesma situação de acordo com o ponto de vista, é possível obter apenas um bloco. Estas diferenças estão bem patentes na **Figura 5-21** onde mais uma vez em **a)** se representam todos os nós da *quadtree* percorridos para o *view frustum* considerado bem como, em **b)**, a triangulação resultante num terreno de dimensão 33×33 com um bloco de dimensão 5×5. Em comparação com as figuras equivalentes no Brute Force (ver **Figura 5-19**) e no Geomipmapping (ver **Figura 5-20**), conclui-se que a pesquisa na *quadtree* é diferente. Isto porque os blocos mais distantes podem ser aceites em níveis mais altos da *quadtree* resultando na triangulação num bloco de maior dimensão mas com o mesmo número de vértices (o bloco superior direito na figura é um exemplo disso). A principal consequência desta aproximação é a redução do número de chamadas efectuadas, neste caso de 20 para 14 no GPU Terrain Rendering.

A métrica de erro adoptada nesta implementação é diferente da utilizada no algoritmo original. No algoritmo original descrito em **4.8.2** considera-se apenas a distância, aqui no entanto, e seguindo uma recomendação do próprio autor [207], utiliza-se uma métrica mais eficiente que tem também em consideração o erro geométrico. Aproveitando o facto de uma métrica deste género já ser usado no contexto do Geomipmapping optou-se por uma estratégia semelhante à usada no algoritmo de Rendering Very Large Very, Detailed Terrains (ver **4.7**) na qual se emprega exactamente a mesma fórmula do Geomipmapping, mas utilizando o erro geométrico associado a cada bloco em cada um dos níveis da *quadtree*. Este erro geométrico do bloco resulta, tal como foi descrito em **5.4.1,** do erro geométrico máximo dos blocos filhos mais o erro geométrico do próprio bloco o que garante que o erro cresce obrigatoriamente de nível para nível. Assim, tal como no Geomipmapping, é calculada uma constante C numa fase de pré-processamento por intermédio da **Equação 5-4** e efectuado o cálculo da distância mínima a partir da qual um determinado bloco num determinado nível é seleccionado através da **Equação 5-5.** Na posse desta informação a condição de recusa de um determinado nó da *quadtree* passa a estar directamente relacionada com a distância a que este se encontra do ponto de vista e a distância mínima a partir da qual o bloco associado pode ser utilizado de modo a respeitar o erro máximo em *pixels*, τ, estabelecido. Se a distância a que se encontra o bloco do ponto de vista for superior ou igual à distância mínima, o bloco é aceite e a recursão termina nesse nó. Se, pelo contrário, a distância for inferior, então é necessário subdividir mais, entenda-se aumentar detalhe e consequentemente aumentar o número de chamadas à primitiva, pelo que se desce mais um nível na *quadtree* para o nó em consideração.



a) Pesquisa na *quadtree* b) Triangulação da *quadtree*

Figura 5-21: Pesquisa e triangulação no algoritmo de GPU Terrain Rendering.

5.6.2. Coerência Espacial

A utilização de técnicas de nível de detalhe, embora reduza substancialmente a quantidade de geometria enviada para *rendering*, dá origem a uma série de problemas (ver **3.4.4**). Causa nomeadamente, o aparecimento de falhas (ver **3.4.4.1**) entre blocos de terreno de diferentes níveis de detalhe e o *popping* (ver **3.4.4.2**). Destes dois o aparecimento de falhas é aquele que tem maior impacto a nível visual uma vez que estas descontinuidades são visíveis em tempo de execução. É por isso extremamente importante corrigir este problema de modo a não afectar de modo determinante a qualidade visual do terreno. Nesse sentido recorreu-se a duas estratégias:

- Actualização de índices, aplicável ao algoritmo de Geomipmapping e que envolve a alteração em tempo de execução da lista de índices utilizada por cada um dos blocos de terreno enviados para *rendering*. Esta técnica é discutida em detalhe em **5.6.2.1**.
- *Skirts*, aplicável aos algoritmos de Geomipmapping e de GPU Terrain Rendering e que envolve o adicionar de uma malha poligonal em torno do bloco de terreno de modo a mascarar qualquer falha que possa eventualmente surgir entre blocos adjacentes de diferentes níveis de detalhe. Esta técnica já referida em **5.5.1.1** é discutida em detalhe **5.6.2.2**.

5.6.2.1. Actualização de Índices

O objectivo desta técnica é a adaptação dinâmica da geometria de cada um dos blocos de terreno em função do nível de detalhe dos blocos vizinhos pelo que dada a sua natureza pode ser aplicada apenas no algoritmo de Geomipmapping (ver **4.4**). Assim, implica uma alteração da lista de índices quando um dos lados é partilhado com um bloco de um nível de detalhe inferior. Essa adaptação ocorre pelo colapso dos vértices na fronteira desse bloco quando não existe correspondência com os vértices do bloco vizinho estando por isso dependente da diferença entre ambos os níveis de detalhe. Na

Figura 5-22 esse colapso dos vértices é bem visível e, embora ocorra, neste caso apenas no lado superior de cada um dos blocos afectados, pode surgir em qualquer um dos lados e em mais do que um ao mesmo tempo. Dada a natureza da métrica de erro empregue no Geomipmapping (ver **5.6.1.2**), que tem em consideração não só a distância mas também o erro geométrico podem surgir um grande número de situações distintas. Na figura é ainda exemplificado o efeito deste método nos dois tipos de triangulações possíveis, a lista de triângulos (ver **3.1.1**) e a tira de triângulos (ver **3.1.3**), isto para um bloco 3×3 que partilha o lado superior com um bloco 2×2, ambos níveis de detalhe de um bloco 5×5 num terreno 33×33.

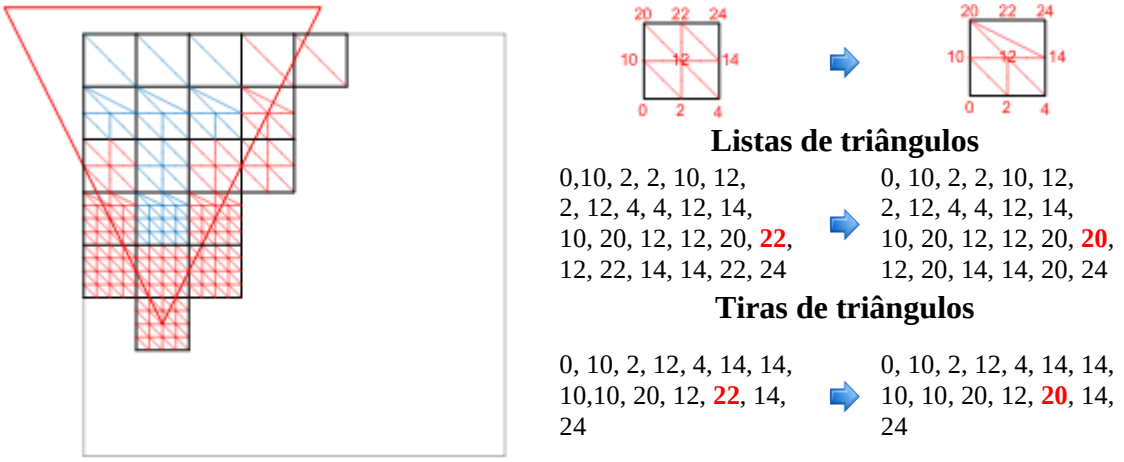


Figura 5-22: Actualização de índices no Geomipmapping.

Logo no início desta secção referiu-se que este método pode ser aplicado apenas no algoritmo de Geomipmapping. Com o intuito de fundamentar um pouco mais essa observação, apresenta-se na **Figura 5-23** uma possível triangulação face ao ponto de vista representado num terreno de 33×33 com um bloco de 3×3 no algoritmo de GPU Terrain Rendering. Nesta figura os círculos indicam vértices que não podem desaparecer nem convergir para outros vértices pelo que nestas situações podem sempre ocorrer falhas. Esta é a razão pela qual não se utiliza esta estratégia no algoritmo de GPU Terrain Rendering. Uma possível solução para este problema seria estabelecer um máximo para o número de níveis de detalhe de diferença entre blocos numa abordagem muito semelhante à tomada no algoritmo de Real-time Generation of Continuous Levels of Detail (ver **4.3**). No entanto, dada a natureza do GPU Terrain Rendering, entendeu-se que essa alternativa poderia limitar o desempenho, pelo que o único método utilizado no tratamento de falhas é a criação de *skirts* (ver **5.6.2.2**).

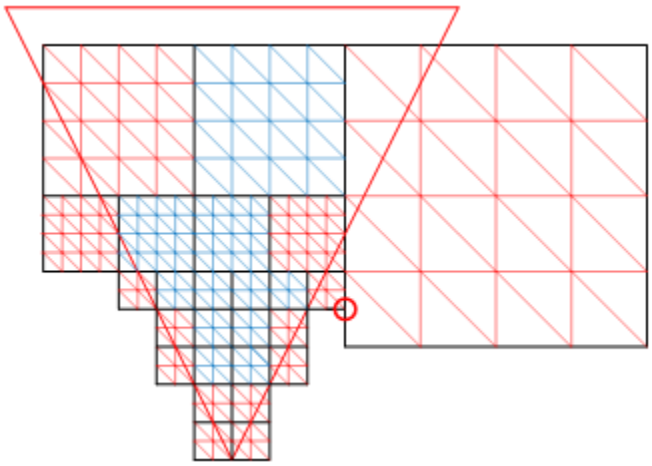


Figura 5-23: Problemas da actualização de índices no GPU Terrain Rendering.

5.6.2.2. Skirts

Ao contrário do método anterior (ver **5.6.2.1**), este método pode ser usado com as duas técnicas de redução do nível de detalhe implementadas no contexto dos algoritmos de Geomipmapping (ver **4.4**) e de GPU Terrain Rendering (ver **4.8**). É, no entanto, uma abordagem completamente diferente ao problema das falhas entre blocos adjacentes com diferentes níveis de detalhe, já que não procura corrigir a geometria mas sim mascarar as falhas por intermédio de uma malha poligonal que envolve o bloco. Em contrapartida, não implica em tempo de execução a alteração nem da lista de índices nem da lista de vértices, o que devidamente explorado pode funcionar como uma vantagem em termos de desempenho. Nesta perspectiva, a altura das *skirts* é determinante, pelo que estas devem ser suficientemente altas para cobrir qualquer falha que possa surgir, mas não mais. Assim, tal como é descrito em **5.4.2**, o valor utilizado é o erro geométrico máximo dos vértices que envolvem o bloco, valor que permite mascarar qualquer falha que possa surgir, tanto no Geomipmapping como no GPU Terrain Rendering, garantindo paralelamente a menor altura possível. Deste modo, antes de se efectuar o *rendering* é necessário obter a altura das *skirts* que melhor se adapta ao bloco. Esta vai depender no Geomipmapping do nível de detalhe do bloco, já que nesse caso os blocos têm todos a mesma dimensão. No caso do GPU Terrain Rendering, vai depender do tamanho do bloco, já que aqui a variação do nível de detalhe ocorre pela selecção de blocos de maior dimensão, pelo que nesse caso basta obter a altura das *skirts* armazenada em cada um deles. Isto é, no Geomipmapping por cada um dos níveis de detalhe possíveis para o bloco armazena-se a altura das *skirts* que o envolvem. No GPU Terrain Rendering isso acontece por cada bloco nos diferentes níveis da *quadtree*.

Uma vez garantida a menor altura possível para as *skirts*, resta actuar ao nível da visibilidade das mesmas. Para isso é necessário que as diferentes combinações das quatro *skirts* que envolvem o bloco estejam disponíveis de modo a que de acordo com o ponto de vista seja possível seleccionar a mais apropriada. Este ponto foi discutido em **5.5.1.1**, onde se descreve o modo como essas combinações foram geradas e armazenadas, pelo que resta aqui descrever como se poderá tirar proveito dessa flexibilidade para

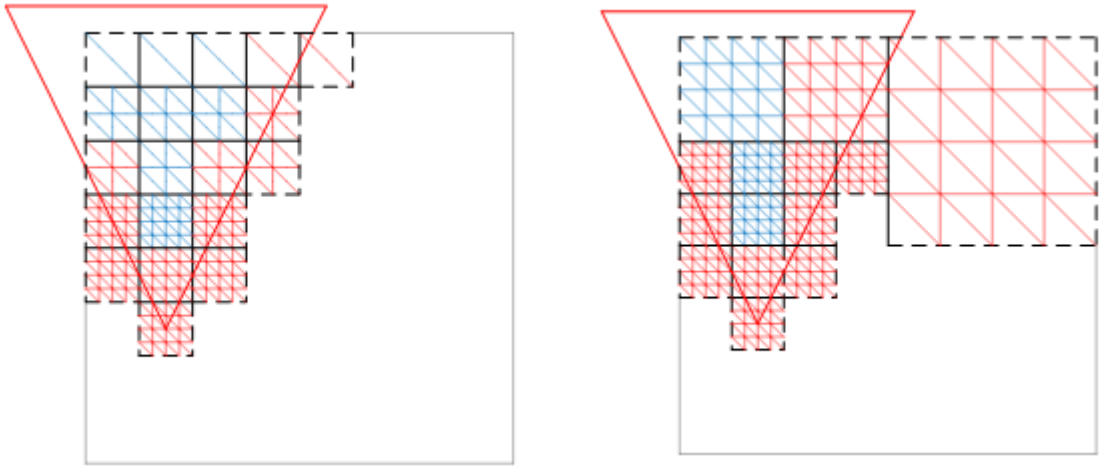
seleccionar apenas as *skirts* que estão visíveis para o ponto de vista corrente. Assim, procurou-se remover *skirts* nas seguintes situações:

- Quando o bloco está nos limites do terreno, já que nesse caso não são necessárias *skirts* nas partes do bloco viradas para fora pois não existem nessa situação blocos vizinhos.
- Quando uma ou mais *skirts* do bloco não está contida no *view frustum*.

Na **Figura 5-24**, para um terreno de 33 × 33 e um bloco de 5 × 5 podemos observar as *skirts* visíveis em cada um dos algoritmos de geração de terrenos em tempo real considerados. Nesta figura, a tracejado, estão representados todos os lados para os quais não foi utilizada uma *skirt*. Para obter esta distribuição de *skirts* por bloco a estratégia a adoptar envolve apenas 3 passos:

1. Começa-se assumindo que o bloco vai ter as quatro *skirts*.
2. Se o bloco estiver na fronteira do terreno, removem-se as *skirts* que estão no limite do terreno.
3. Se o bloco estiver contido dentro do *frustum*, então não é possível retirar mais *skirts*. Caso contrário, se o bloco intersectar o *frustum*, verifica-se para as *skirts* que sobram (algumas podem ter sido removidas no passo 2), se estas intersectam o *frustum*. Se não intersectarem, então são removidas.

Deste modo é garantido que apenas as *skirts* visíveis são enviadas para *rendering*, obtendo-se no final deste processo a combinação de *skirts* (ver **Figura 5-15**) a ser utilizada para o bloco em análise. Isto permite diminuir ainda mais a quantidade de geometria enviada para *rendering* e consequentemente aumentar o desempenho.



Geomipmapping GPU Terrain Rendering

Figura 5-24: As *skirts* visíveis nas triangulações dos dois algoritmos.

5.6.3. Coerência Temporal

Para além das falhas (ver **3.4.4.1**) entre blocos adjacentes de diferentes níveis de detalhe também o *popping* (ver **3.4.4.2**) tem um impacto determinante na qualidade

visual do terreno. Para eliminar esse efeito, utiliza-se o Geomorphing tanto no algoritmo de Geomipmapping como no algoritmo de GPU Terrain Rendering o que implica em qualquer um dos casos um movimento progressivo dos vértices ao longo da coordenada y, tendo como consequência uma transição entre a posição actual e a posição que esses vértices teriam no próximo nível de detalhe, diminuindo assim o impacto dessa mudança. Essas duas posições, que podem ser vistas como a posição original e a posição alvo, são enviadas tal como se descreve em **5.5.2** numa lista de valores de elevação (ver **5.5.2.2**) ou em duas texturas (ver **5.5.2.3**), dependendo da forma como os valores de elevação são tratados. Na posse desses dados é necessário efectuar a transição propriamente dita. Para isso, além dos valores de elevação que são enviados para *rendering* da forma descrita, é necessário fornecer para cada bloco um factor entre 0 e 1. Esse factor permite determinar ao nível do *vertex shader* a posição que cada um dos vértices deverá ter em cada instante. Para que possa actuar apenas sobre os vértices do bloco que vão desaparecer no próximo nível de detalhe, envia-se à semelhança do que é feito com o valor de *morph* na lista de valores de elevação (ver **5.5.2.1**), informação distinta na parte inteira e na parte decimal. Na parte inteira o nível de detalhe, que é no caso do Geomipmapping o do próprio bloco e no caso do GPU Terrain Rendering o nível em que o bloco se encontra na *quadtree* (ver **3.2.3.1**). Na parte decimal envia-se o factor propriamente dito. Por exemplo, 2.2 significa que se pretende afectar apenas os vértices que desaparecem na transição do nível de detalhe 2 para o nível de detalhe 3. A posição dos vértices afectados é obtida a partir da parte decimal, 0.2 no exemplo, permitindo determinar por interpolação linear o valor da coordenada y de cada um deles.

Para se calcular o valor do factor foi adoptada a fórmula descrita no algoritmo de Geomipmapping (ver **4.4.6**) para os dois algoritmos considerados o que só foi possível porque na determinação do nível de detalhe utiliza-se a métrica de erro do Geomipmapping tal como é descrito em **5.6.1.2** e **5.6.1.3**. Essa fórmula repete-se aqui na **Equação 5-6** por questões de referência.

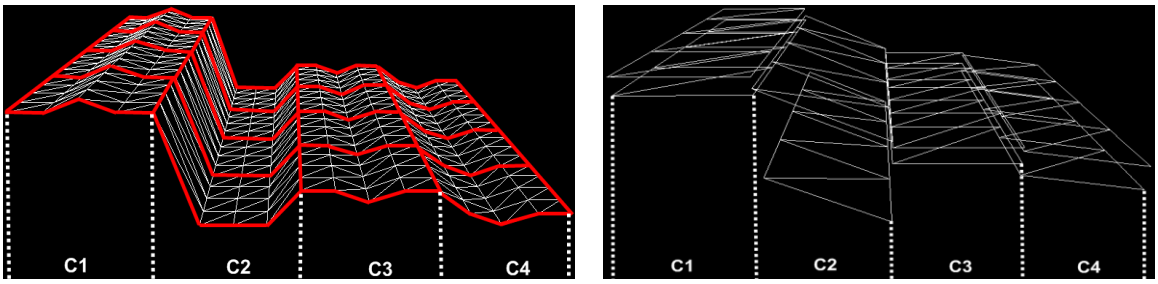
$t = \frac{(d - D_n)}{(D_{n+1} - D_n)}$	d : Distância da câmara ao bloco.
	D_n: Distância mínima pré-calculada para o nível de detalhe.
	D_{n+1}: Distância mínima pré-calculada para o nível de detalhe seguinte $n + 1$.

Equação 5-6: Cálculo do factor de *morphing* nos dois algoritmos.

5.6.4. Occlusion Culling

A técnica de oclusão adoptada baseia-se na empregue no algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**). Esta técnica tem como ponto de partida uma lista de blocos ordenados pela distância à câmara construída no decorrer dos passos anteriores (ver **5.6.1**). O objectivo é percorrer essa lista testando a visibilidade de cada um dos blocos face a um horizonte que vai sendo construído em função dos blocos que entretanto se determinem visíveis. Assim, em cada *frame*, a linha de horizonte é limpa e começa a ser percorrida a lista de blocos. O primeiro bloco é imediatamente adicionado ao horizonte e enviado para *rendering*, uma vez que está de certeza visível. A partir daí,

cada um dos blocos seguintes tem de ser testado pois já existe a possibilidade de estarem ocultos face a outros que passaram entretanto o teste de oclusão. Para que tal seja possível, é necessário utilizar os dados de oclusão que foram previamente construídos e armazenados por bloco (ver **5.4.3**). Esses dados, nomeadamente o plano máximo, o plano mínimo e o valor do erro são utilizados respectivamente no teste de oclusão, no adicionar do bloco ao horizonte e na métrica de erro. Desta forma, como primeiro passo no sentido de se esclarecer o modo como esses dados são utilizados e a forma como o algoritmo foi concretizado, apresenta-se na **Figura 5-25** para um terreno 17 × 17 com 16 blocos 5 × 5 os 16 planos mínimos e máximos associados a esse terreno. São também identificadas cada uma das colunas de 4 blocos que constituem o terreno. Estas serão usadas mais tarde no decorrer deste exemplo para identificar os blocos que vão ser removidos pelo algoritmo de oclusão.



Divisão do terreno em 16 blocos Planos mínimos e máximos dos 16 blocos

Figura 5-25: Planos mínimos e máximos dos 16 blocos 5 × 5 de um terreno 17 × 17.

Tal como é abordado em detalhe em **4.6.4**, os planos mínimos são utilizados na construção do horizonte e os planos máximos para determinar relativamente ao horizonte se o bloco está ou não oculto. Assim, a primeira condição para um bloco ser rejeitado é o seu plano máximo estar abaixo de todos os planos mínimos dos blocos entretanto adicionados ao horizonte. Obviamente, isso é algo que está directamente dependente da ordem dos blocos na lista o que por sua vez depende do ponto de observação. Nesse sentido considere-se agora a **Figura 5-26** onde para dois pontos de observação do terreno se demonstra a criação da linha de horizonte a partir do plano mínimo. No primeiro caso, em que temos uma vista lateral do terreno, não existe nenhuma espécie de *culling*. Uma vez que a ordenação dos blocos em função da distância à câmara garante que o plano máximo de cada um dos blocos nunca vai estar abaixo do horizonte construído com os planos mínimos dos blocos anteriormente processados. Essa situação já não ocorre na vista frontal representada na figura, pois existem blocos cujo plano máximo está abaixo do horizonte obtido até aí. Esses blocos são os pertencentes às colunas C3 e C4, identificadas na **Figura 5-25**, e estão efectivamente ocultos face a outros blocos de terreno sendo correctamente detectados pelo algoritmo de oclusão empregue.

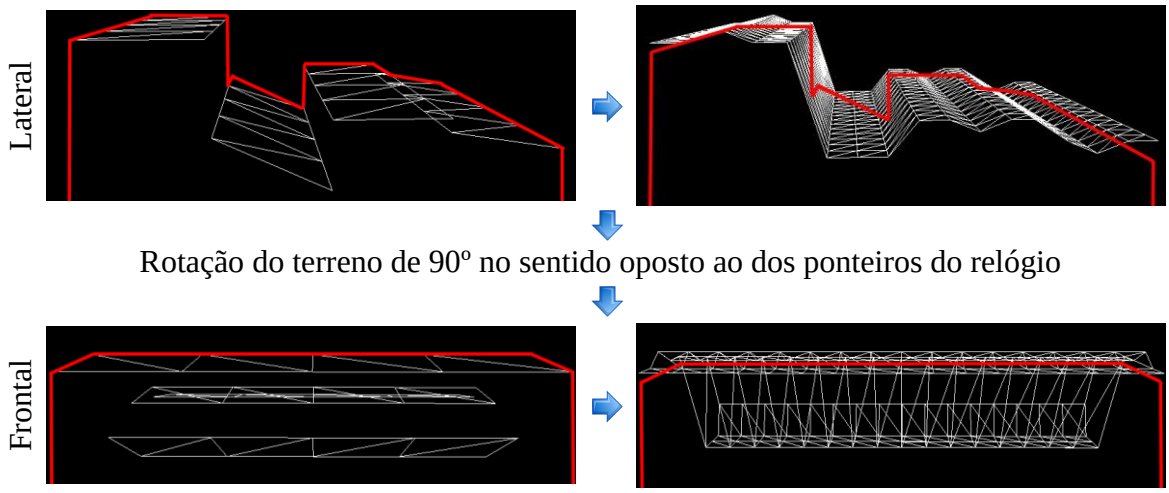


Figura 5-26: Construção do horizonte a partir do plano mínimo.

No que diz respeito ao adicionar do plano mínimo de cada um dos blocos ao horizonte, este implica a transformação para coordenadas de ecrã das quatro linhas que compõem cada um dos planos tal como é exemplificado na **Figura 5-27** para o plano P1. Nesta figura, que representa um terreno com 4 blocos cada uma das linhas é transformada individualmente sendo possível verificar o seu efeito no horizonte e o modo como este vai sendo aos poucos construído. Note-se que o primeiro plano avaliado, o P1, é o que se encontra mais perto do ponto de vista.

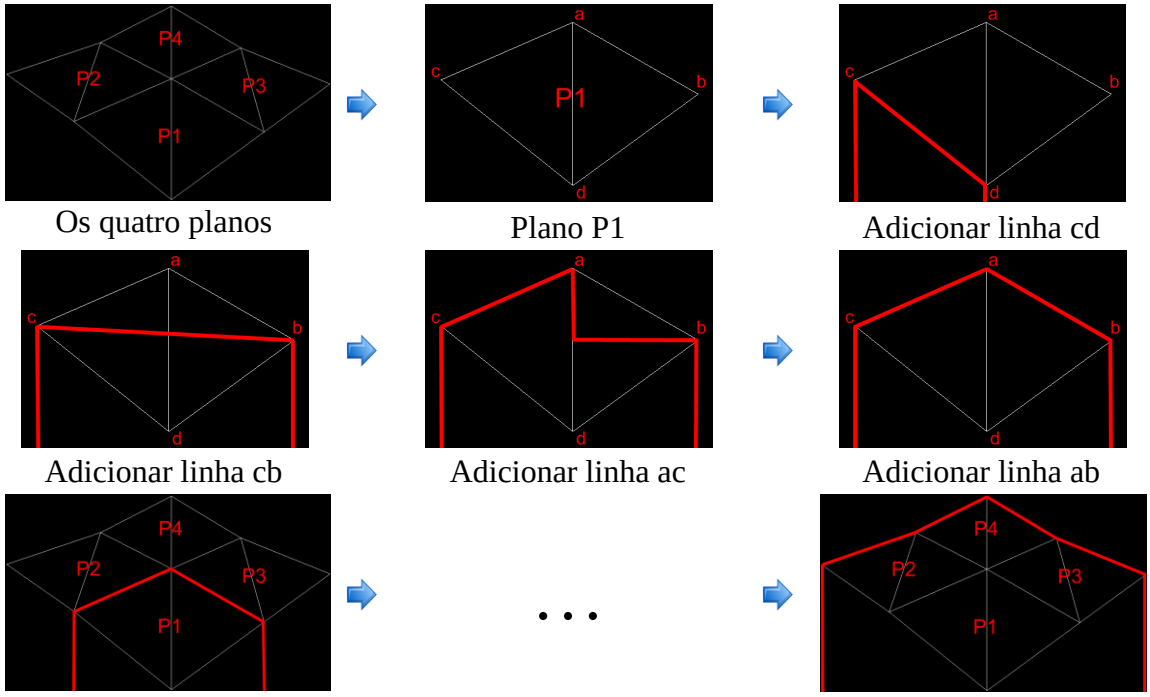


Figura 5-27: Adição de cada uma das linhas do plano mínimo ao horizonte.

Passando agora ao teste de oclusão, este envolve, tal como a adição do bloco, a transformação de cada uma das linhas do plano máximo, mas desta vez para verificar se estão abaixo do horizonte até aí construído. Por outro lado, o plano utilizado também é diferente. Neste caso utiliza-se o plano máximo. Isso garante, tal como foi referido em **4.6.4**, que qualquer erro na aproximação é correctamente tratado e que o teste é conservativo, ou seja, quanto muito vão ser enviados para *rendering* blocos que não estão visíveis, mas o contrário, ou seja, o rejeitar de blocos visíveis, nunca irá ocorrer.

Resta agora perceber o papel da métrica de erro neste algoritmo. O objectivo é muito simples: controlar a qualidade do horizonte obtido. É neste contexto que o número de vértices do bloco e a profundidade da *quadtree* (ver **3.2.3.1**) entram. Tal como foi referido em **5.4**, a profundidade da *quadtree* não é determinada pelo número de vértices do bloco, mas por um segundo valor fornecido como parâmetro que pode ser igual ou inferior ao número de vértices do bloco considerado. Dessa forma torna-se possível utilizar um maior espaço de pesquisa em consequência do maior número de níveis disponibilizados. No entanto, essa transição entre níveis só é efectuada se o erro em *pixels* estiver acima de um limiar fornecido como parâmetro. É aqui que é utilizado o valor do erro armazenado no bloco, e, que não é mais do que o valor máximo de distância vertical entre os planos mínimo e máximo e o plano central. Este é convertido num erro em *pixels* por intermédio da **Equação 5-7** e da **Equação 5-8** que se reptem aqui por questões de referência.

$(1-f)\cdot\frac{e}{D}\cdot K$	f: A coordenada *y* do vector de direcção da câmara.
	e: O valor máximo de distancia vertical entre os planos mínimo e máximo e o plano central.
	D: Distância da câmara ao centro do bloco.
	K: Calculado na **Equação 5-8**.

Equação 5-7: Cálculo da métrica de erro que controla a recursão.

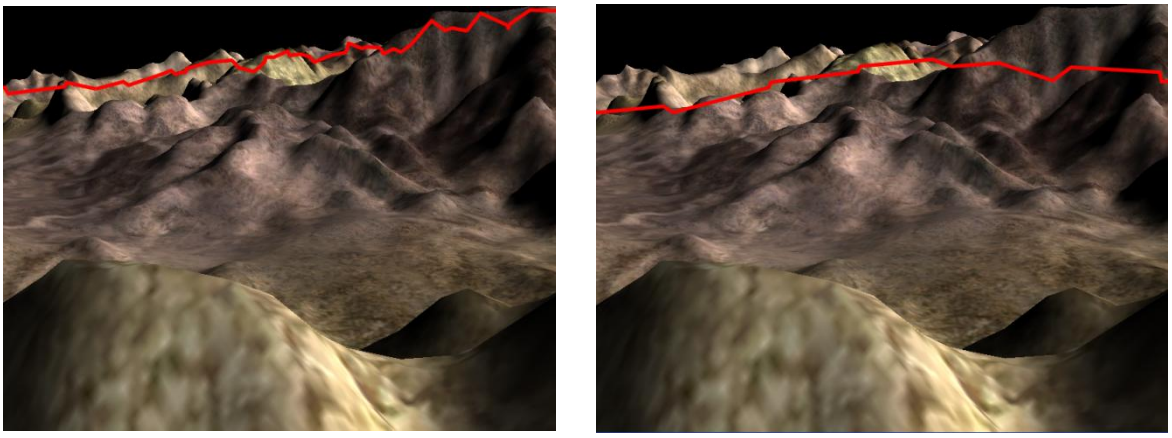
$K=\frac{W}{2\cdot\left(\tan\left(\frac{FOV}{2}\right)\right)}$	W: Largura do ecrã em *pixels*.
	FOV: Campo de visão.
	tan: Função de cálculo da tangente.

Equação 5-8: Cálculo do factor de escala da perspectiva.

É de realçar que a recursão na *quadtree* no caso do Geomipmapping (ver **4.4**) e do Brute Force, não ocorre se o número de vértices do bloco for igual ao número de vértices que determina a profundidade da *quadtree*. Isto muito simplesmente porque nesses dois algoritmos o tamanho do bloco não varia o que é o mesmo que dizer que todos os blocos estão no mesmo nível da *quadtree* que, nesse caso, seria o último. O mesmo não acontece no GPU Terrain Rendering (ver **4.8**) onde, os blocos podem variar de tamanho pelo que é ainda possível descer um pouco mais até ao tamanho do bloco. Basicamente, na determinação do nível de detalhe, a recursividade pára no nível da *quadtree* correspondente ao número de vértices do bloco fornecido como parâmetro. Quando é aplicado o algoritmo de oclusão para cada um dos blocos é possível efectuar-se ainda um

ou mais passos de recursão de acordo com o número de níveis disponíveis e a métrica erro.

De maneira a se visualizar de uma forma mais efectiva o impacto que tem a utilização de mais blocos na determinação da linha de horizonte, é apresentada na **Figura 5-28** para um terreno com 257 × 257 a diferença que existe entre a linha de horizonte obtida para um bloco de 33 × 33 e para um bloco de 5 × 5. Como se pode verificar, à direita da figura a linha de horizonte no segundo caso está muito mais refinada e na prática permite o *culling* de mais blocos uma vez que detecta de uma forma mais precisa a silhueta do terreno.



Bloco: 33 × 33, Quadtree: 33 × 33 Bloco: 33 × 33, Quadtree: 5 × 5

Figura 5-28: Relação entre a profundidade da *quadtree* e a qualidade do horizonte.

5.7. Material

A aproximação adoptada corresponde à utilização de uma única textura para todo o terreno, o que é coerente com os objectivos estabelecidos, já que não se teve em consideração nem terrenos de dimensões arbitrárias, nem um nível muito elevado de detalhe da textura aplicada a cada um dos blocos de terreno. Para que terrenos de dimensão arbitrária fossem possíveis seria necessário efectuar o carregamento da geometria e da textura em tempo de execução, isto é, utilizar técnicas *out-of-core* como as aplicadas no algoritmo de Chunked LOD (ver **4.5**) e no algoritmo de Rendering Very Large, Very Detailed Terrains (ver **4.7**). Paralelamente para se suportar uma maior detalhe de textura, ao nível de cada um dos blocos de terreno seria necessário utilizar estratégias como o *detail texturing* [47] ou o *texture splatting* [18]. Estas técnicas não foram implementadas dado esta dissertação se concentrar na análise da variação do nível de detalhe. No entanto, tanto num caso como no outro são facilmente integráveis.

5.8. Iluminação

No que diz respeito à iluminação, foram utilizadas apenas luzes direccionais e o tradicional modelo difuso [57] que segue a lei desenvolvida por Lambert em [106]. O cálculo em si é efectuado de uma forma dinâmica ao nível do *shader*, pelo que foi

necessário considerar duas alternativas: o *per-vertex lighting* e o *per-pixel lighting*. O primeiro calcula a iluminação por vértice e o segundo por *pixel*, sendo necessário em ambos os casos normais. A diferença é que no *per-vertex lighting* estas são enviadas juntamente com a geometria, ao passo que no *per-pixel lighting* são obtidas a partir de uma textura (o *normal map*, ou mapa de normais). Relevante neste contexto é, o facto de no *per-vertex lighting* ser necessário passar as normais do *vertex shader* para o *pixel shader*, o que resulta numa interpolação das mesmas e uma consequente variação em função do número de vértices considerados. Nessa perspectiva, a decisão pelo *per-pixel lighting* foi quase obrigatória dado que a redução do nível de detalhe iria causar um impacto visual considerável com uma mudança constante da iluminação já que o número de vértices varia constantemente como consequência da mudança de nível de detalhe. Para que seja mais fácil perceber a diferença, apresenta-se na **Figura 5-29** uma comparação entre estas duas abordagens. Como se pode verificar existe uma variação bem patente da iluminação no *per-vertex lighting* que está directamente relacionada com a variação da geometria. Ao nível do *per-pixel lighting* essa variação já não ocorre já que o mapa de normais utilizado no *pixel shader* permite obter em qualquer ponto do terreno a normal correcta. Tal é possível porque o mapa é obtido a partir da geometria no detalhe máximo pelo que é independente do número de vértices.

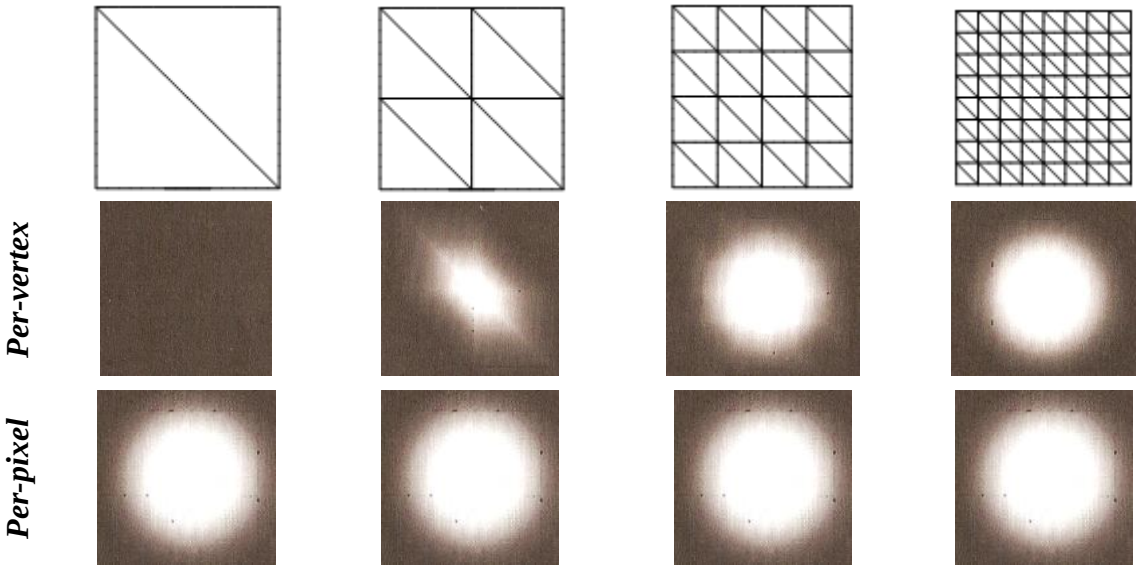
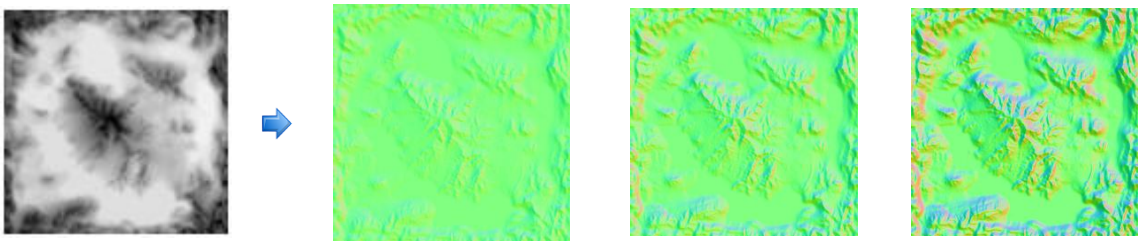


Figura 5-29: *Per-vertex lighting* vs *per-pixel lighting* (adaptado de [57]).

Uma vez que as normais são armazenadas numa textura, torna-se possível aplicar outra técnica que permite aumentar significativamente a qualidade de imagem: o *normal mapping* [97]. Esta técnica muito comum hoje em dia permite, mediante a utilização de um mapa de normais extraído a partir de um modelo de elevado detalhe, utilizar versões menos detalhadas desse modelo mantendo-se a mesma aparência. Em termos técnicos implica perturbar as normais armazenadas em cada um dos *pixels* do mapa de normais acentuando desse modo os detalhes em cada uma das zonas do modelo. Existem diversas ferramentas que permitem gerar os mapas de normais a partir de modelos ([130], por exemplo) e também a partir de mapas de elevações ([103], por exemplo). Neste caso foi implementado um simples filtro de Sobel [187] para gerar o mapa de normais a partir de

um mapa de elevações numa fase de pré-processamento. Este filtro perturba as normais geradas acentuando as zonas onde existe uma transição. Em termos práticos a sua utilização permite acentuar a nível visual as sombras no terreno permitindo uma aparência muito mais realista do mesmo. Na **Figura 5-30** apresenta-se um mapa de elevação e um conjunto de mapas de normais obtidos a partir de diferentes valores de perturbação. O primeiro é um mapa que armazena as normais sem nenhuma perturbação, em todos os outros a delimitação das diferentes zonas é como se pode verificar mais bem definida tendo como consequência o intensificar das sombras no terreno.



Height Map *Normal Map* *Normal Map* (× 2) *Normal Map* (× 4)

Figura 5-30: O *normal map* obtido a partir de diferentes valores de perturbação.

Em relação aos mapas de normais propriamente ditos, estes têm uma aparência esverdeada por que é na componente G (Green) de uma textura RGB que é armazenado o valor y, da normal. Como este valor é o que mais afecta a geometria do terreno, é naturalmente o mais proeminente na figura.

5.9. Implementação

A plataforma adoptada na concretização deste trabalho foi o XNA Framework. Muito embora recente (saiu em Dezembro de 2006) é já uma API madura que se encontra em franca expansão. É especialmente apelativa por permitir o desenvolvimento em C#, uma das linguagens da plataforma .NET da Microsoft. À semelhança das outras linguagens, do .NET corre numa máquina virtual que oferece serviços de *garbage collection* o que permite desenvolver sem que seja necessário remover explicitamente a memória alocada, dado que o *garbage collector* encarrega-se dessa tarefa. Essa é de facto uma das grandes vantagens de linguagens *managed* como o C# ou o Java, pois permitem diminuir a complexidade e sobretudo aumentar a segurança das aplicações. É de referir no entanto que a base da XNA Framework é o DirectX 9.0, mas que, todas as chamadas nativas são abstraídas pela *framework* o que permite utilizar o C# em todas as situações sem que nunca seja necessário utilizar código nativo.

Um dos grandes extras desta plataforma é permitir a execução na consola XBOX 360, tendo sido *inclusive* a primeira plataforma a disponibilizar essa funcionalidade. Tal foi possível graças ao ambiente constrangido e seguro em que corre a *virtual machine* do .NET o que permitiu o acesso a consola sem a comprometer ao nível da segurança.

Existem no entanto diferenças entre o XNA num *desktop* e numa consola. A principal diferença é a *framework* de .NET em si. Num sistema Windows como o que corre nos *desktops* é utilizada a *framework* na sua totalidade. Na XBOX 360 é utilizada uma versão reduzida, a .NET Compact Framework. Esta não foi desenvolvida à medida para a XBOX

360 mas sim adaptada, já que é utilizada em alguns dispositivos Windows CE como o Pocket PC, o Pocket PC Phone Edition e o Smartphone. Como o nome implica é compacta não apenas porque os dispositivos onde costuma ser utilizada são compactos mas também porque a *framework* propriamente dita é um subconjunto da utilizada no desktop. Especialmente relevantes são as diferenças na forma como é feito o *garbage collection*. Na *framework* .NET para *desktops* o *garbage collector* é muito mais sofisticado já que utiliza um mecanismo de gerações, três neste caso, para marcar os objectos. Este mecanismo vai promovendo os objectos de geração em geração à medida que o seu tempo em memória aumenta. As passagens do *garbage collector* por todos os objectos ocorrem assim por geração sendo essas passagens mais frequentes na primeira onde estão todos os objectos que foram alocados recentemente. Tal permite a execução do *garbage collector* de uma forma mais rápida já que se percorrem menos objectos. Na .NET Compact Framework não existe um mecanismo de gerações apenas uma única lista de objectos que tem de ser percorridos, devendo por isso adoptar-se como estratégia a alocação do menor número de objectos em tempo de execução. O melhor é pré-alocar todos os que se necessite numa fase inicial. Além das diferenças na *framework* em si nem todas as capacidades da XBOX 360 são aproveitadas, nomeadamente ao nível das operações matemáticas esperando-se em versões posteriores alterações nesse comportamento. Estas diferenças e outras não referidas aqui mas que podem ser consultadas em maior detalhe em [1] e [2], fazem com que o desempenho obtido seja previsivelmente diferente da versão *desktop*.

5.10. Sumário

Este capítulo descreve em detalhe o trabalho realizado no contexto desta dissertação, focando-se na implementação efectuada em XNA de dois dos algoritmos de geração de terrenos em tempo real discutidos em **4**: o Geomipmapping (ver **4.4**) e o GPU Terrain Rendering (ver **4.8**) bem como na concretização de uma aproximação designada de Brute Force que não emprega nenhuma técnica de nível de detalhe.

Apresenta-se um algoritmo geral (ver **5.2**) que incorpora de forma modular as várias funcionalidades comuns aos algoritmos de geração de terrenos pertencentes à classe Tiled Blocks, como sejam o particionamento espacial, o *culling*, a gestão do nível de detalhe, o *vertex caching*, o tratamento de falhas, o *popping* e a utilização de *vertex textures* no envio dos valores elevação para o GPU. Este algoritmo geral permitiu comparar separadamente o comportamento dos algoritmos em estudo relativamente a cada uma destas funcionalidades, bem como estender esses algoritmos integrando a técnica de oclusão do algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**).

Dá-se especial ênfase à *quadtree* (ver **5.3** e **5.4**) enquanto estrutura de particionamento espacial adoptada, já que vai desempenhar um papel muito importante no tratamento do nível de detalhe pois é utilizada na determinação do nível mais adequado para cada um dos blocos de terreno só que de uma forma diferente em cada um dos algoritmos.

Explica-se também o modo como são construídas as listas de vértices e as listas de índices (ver **5.5**), nomeadamente o suporte para dois tipos de primitivas, as listas e as tiras triângulos e a integração de conceitos como o *vertex caching* (ver **5.5.1.2**) e as *skirts* (ver **5.5.1.1**) neste processo. Descreve-se ainda a estratégia adoptada no envio dos valores de elevação para a placa gráfica. Baseada em *streams* permite a integração de duas

abordagens distintas: listas de valores de elevação (ver **5.5.2.2**) e *vertex textures* (ver **5.5.2.3**). Tal permitiu separar a estrutura geométrica dos valores de elevação e a reutilização dessa mesma estrutura em cada um dos blocos.

São também abordados problemas como a coerência espacial e temporal característicos de algoritmos de nível de detalhe como os descritos. No caso da coerência espacial utilizaram-se duas estratégias: as *skirts* (ver **5.6.2.2**), no Geomipmapping e no GPU Terrain Rendering e a actualização de índices (ver **5.6.2.1**), no Geomipmapping. Para manter a coerência temporal efectuou-se o *vertex morphing* (ver **5.6.3**) em cada um dos algoritmos. Ainda em relação às *skirts* foram também descritas algumas optimizações (ver **5.4.2** e **5.5.1.1**) com o intuito de reduzir o impacto ao nível do desempenho.

A selecção dos blocos a enviar para *rendering* (ver **5.6.1**) e do nível de detalhe em cada um dos algoritmos (ver **5.6.1.2** e **5.6.1.3**) foi outros dos tópicos em discussão nomeadamente a métrica de erro adoptada, neste caso o erro geométrico em *pixels* (ver **5.4.1**). Esta foi utilizada nos dois algoritmos, mas no GPU Terrain Rendering, o seu cálculo é efectuado tal como no algoritmo de Rendering Very Large, Very Detailed Terrains (ver **4.7**).

Em **5.6.4** é discutido em detalhe o processo de construção do horizonte que permite a detecção dos blocos de terreno ocultos por outros blocos de acordo com a técnica de oclusão empregue no algoritmo de Terrain Occlusion Culling with Horizons.

Finalmente aborda-se o material utilizado na representação do terreno (ver **5.7**), a técnica de iluminação adoptada (ver **5.8**) e por fim a alguns dados sobre o XNA enquanto plataforma de implementação (ver **5.9**).

6. Resultados

Neste capítulo vão ser apresentados os resultados obtidos nos testes efectuados a cada um dos algoritmos, respectivamente o Geomipmapping, o GPU Terrain Rendering e a aproximação Brute Force. Na concretização dos testes utilizaram-se máquinas (ver **6.1**) com capacidades gráficas distintas, bem como um conjunto de terrenos (ver **6.2**) de diferentes dimensões/características. O principal objectivo foi obter uma amostra minimamente representativa a partir de diferentes configurações de *hardware* pela aplicação de uma metodologia de testes (ver **6.4**) que procurou garantir que as condições e o modo como os testes se efectuaram foram as melhores possíveis. Em relação aos testes propriamente ditos (ver **6.5**), a ideia foi investigar o desempenho face aos diferentes valores possíveis para os itens de configuração considerados (ver **6.3**), com o intuito de se determinar a melhor configuração face às máquinas utilizadas. Adicionalmente, e de uma forma mais global, procurou-se avaliar o desempenho dos algoritmos de geração de terrenos em tempo real, especialmente as características descritas em **5**, aplicadas ao Geomipmapping e ao GPU Terrain Rendering. No final deste capítulo em **6.6** é feita uma análise detalhada dos resultados mais importantes e expostas as conclusões a que se chegou.

6.1. Ambiente de Testes

Na **Tabela 6-1** são descritas as diferentes máquinas utilizadas nos testes. Como se pode verificar, consideram-se três tipos diferentes de configurações: dois *notebooks*, um *desktop* e uma XBOX 360. O objectivo foi obter uma amostra alargada que permitisse tirar conclusões relevantes para cada um dos itens de configuração considerados (ver **6.3**). Neste tipo de aplicações, a placa gráfica desempenha quase sempre o papel mais importante tendo uma influência directa no desempenho, pelo que se realçam aqui os diferentes tipos de placas utilizadas em cada uma das máquinas. Destas máquinas, a D1 e a X1, são as que têm as placas gráficas mais rápidas. Por outro lado são as únicas que suportam o *shader model* 3.0, pelo que só nestas é que foram possíveis abordagens diferentes no tratamento dos dados de elevação dos terrenos. Nestes casos pode-se enviar os dados de elevação em listas valores de elevação ou utilizar *vertex textures*, isto é, enviar dos dados de elevação numa textura.

N1	*Tipo*	Notebook
	Sistema Operativo	Microsoft Windows XP Professional Service Pack 2
	Processador	Intel Core Duo
	Memória	2.0 Gbytes RAM
	Placa Gráfica	Mobile Intel 945 GM
	Shader Model	2.0
	.NET Framework	3.5
	XNA Framework	2.0

N2	*Tipo*	Notebook
	Sistema Operativo	Microsoft Windows XP Professional Service Pack 2
	Processador	Intel Pentium M 755
	Memória	1,28 Gbytes RAM
	Placa Gráfica	ATI Mobility Radeon 9700
	Shader Model	2.0
	.NET Framework	3.5
	XNA Framework	2.0

D1	*Tipo*	Desktop
	Sistema Operativo	Microsoft Windows Vista Ultimate
	Processador	Intel Core 2 6600
	Memória	2.0 Gbytes RAM
	Placa Gráfica	NVIDIA GeForce 7950 GX2
	Shader Model	3.0
	.NET Framework	3.5
	XNA Framework	2.0

X1	*Tipo*	Xbox 360 [132]
	Sistema Operativo	Proprietário
	Processador	Power PC Triple Core
	Memória	512 Mbytes RAM partilhados com o GPU
	Placa Gráfica	ATI Xenos (desenvolvida exclusivamente para a Xbox 360)
	Shader Model	Variação do 3.0 com algumas capacidades adicionais
	.NET Framework	Compact Framework
	XNA Framework	2.0

Tabela 6-1: Máquinas de teste.

Os códigos atribuídos na tabela a cada um das máquinas identificadas, N1, N2, D1, X1 serão utilizados a partir deste ponto para as identificar em cada um dos testes (ver **6.5**) e de um modo geral no decorrer deste capítulo sempre que seja necessário referenciá-las.

6.2. Terrenos

Na **Figura 6-1** são apresentados os terrenos utilizados nos testes, bem como as respectivas texturas a partir das quais foram construídos. Cada terreno utiliza quatro fontes de dados:

- *Height map* (ver **2.3.1**), que armazena os valores de elevação numa imagem a preto e branco em que os *pixels* mais claros representam os maiores valores de elevação e os mais escuros valores de elevação mais baixos.
- *Color map* (ver **5.7**), a textura utilizada para dar cor ao terreno.
- *Normal map* (ver **5.8**), a textura que armazena as normais do terreno e que é utilizada para calcular a iluminação. Esta é gerada a partir do *height map* numa fase de pré-processamento com um filtro de Sobel [187] com uma perturbação de $2\times$.
- *Sky map*, um *cube map*[6] que é utilizado no ambiente que envolve o terreno para simular o céu.

Cada um dos terrenos tem características específicas sendo a mais significativa a dimensão dos mesmos. Assim e descrevendo mais especificamente cada um deles:

- Valley com uma dimensão de 257×257 é o terreno utilizado no jogo XNA Racing Game [60] .
- Desert com uma dimensão de 513×513 é um dos terrenos disponibilizados no motor gráfico Ogre [147].
- Terrace com uma dimensão de 1025×1025 foi gerado pelo autor no programa L3DT [103].
- Highland com uma dimensão 2049×2049 foi à semelhança do anterior gerado pelo autor no programa L3DT [103].

Quanto aos *cube maps* utilizados no *rendering* do *sky map* estes foram obtidos a partir de um pacote de mapas disponíveis gratuitamente em [217].

[6] Um *cube map* é uma textura com 6 lados diferentes. É normalmente utilizado para representar o ambiente que envolve a cena ou então para simular reflexões num objecto através de uma técnica designada de *environment mapping*.

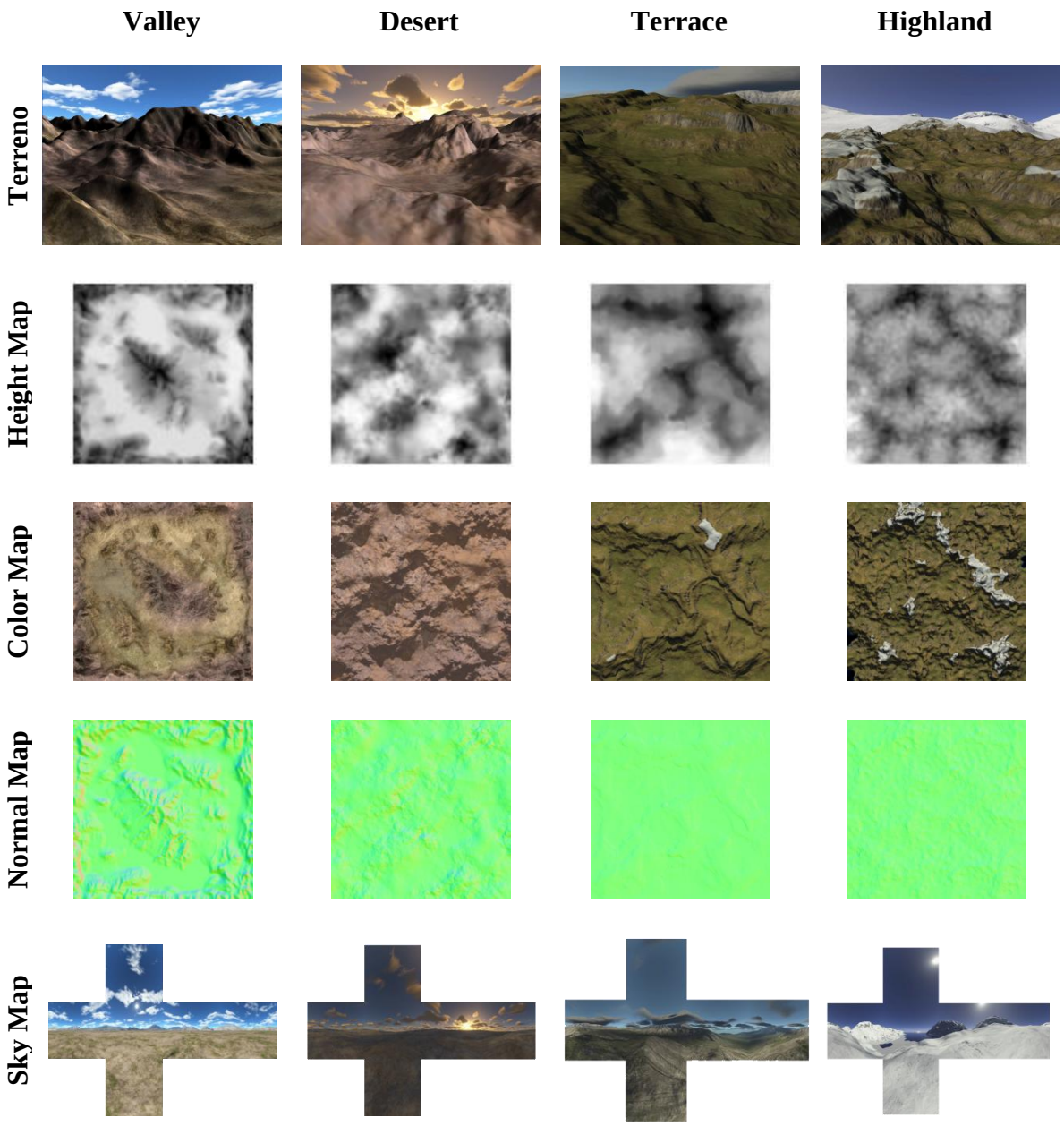


Figura 6-1: Terrenos utilizados nos testes e respectivos mapas.

6.3. Configuração

Na **Tabela 6-2** apresentam-se os parâmetros de configuração utilizados por omissão nos testes efectuados bem como uma descrição de cada um deles. Sempre que nos testes (ver **6.5**) não for mencionado o valor utilizado para um determinado parâmetro, então deve-se assumir o apresentado nesta tabela. Estes são efectivamente os nomes dos parâmetros utilizados a nível aplicacional para configurar a aplicação daí a nomenclatura

apresentada. O objectivo é variar nos testes alguns destes valores de modo a que seja possível apurar o seu impacto ao nível do desempenho. Os valores de cada um deles foram escolhidos por constituírem uma configuração a partir da qual se obtiveram bons resultados, não devendo no entanto assumir-se que são os melhores para todas as situações. Constituem no conjunto, uma configuração razoável que se adoptou para efectuar os testes já que seria impossível apresentar aqui todas as combinações possíveis.

Parâmetro	Valor por Omissão	Valores Possíveis	Descrição
***Terrain* (ver 6.2)**	Terrace	Valley, Desert, Terrace, Highland	O nome do terreno utilizado nos testes.
ScreenWidth	1024	Todas as resoluções suportadas	A largura do ecrã em *pixels*
ScreenHeight	768	Todas as resoluções suportadas	A altura do ecrã em *pixels*.
FullScreen	True	True, False	Indica se os testes são executados em ecrã inteiro.
NearPlaneDistance	1	>0	Define o início do *frustum*.
FarPlaneDistance	10000	> NearPlaneDistance	Define o fim do *frustum*.
***LodMaxError* (ver 5.6.1)**	8	≥ 0	O erro máximo em *pixels* do algoritmo de nível de detalhe do terreno.
***SpatialCoherence* (ver 5.6.2)**	SkirtsOptimized	Nenhum, Skirts, SkirtsOptimized, IndexUpdate	O método utilizado na correcção das falhas. No SkirtsOptimized só são enviadas para *rendering* as *skirts* que intersectam o *frustum* e a altura das mesmas é calculada com base no erro geométrico dos valores de elevação que envolvem o bloco. O IndexUpdate é suportado apenas no Geomipmapping.
***TemporalCoherence* (ver 5.6.3)**	Geomorphing	Nenhum, Geomorphing	O método utilizado na correcção do efeito de *popping*.
***TerrainTileSize* (ver 5.4)**	33	Da forma $2^n + 1$	O número de vértices do bloco de terreno.
***Primitive* (ver 5.5.1)**	IndexTriangleStrips	IndexedTriangleLists, IndexedTriangleStrips	A primitiva utilizada.
***PrimitiveBlockSize* (ver 5.5.1.2)**	33	$\leq$ TerrainTileSize e da forma $2^n + 1$	Número máximo de vértices por linha.

Parâmetro	Valor por Omissão	Valores Possíveis	Descrição
QuadTreeTileSize **(ver 5.4)**	5	≤ TerrainTileSize e da forma $2^n + 1$	O número de vértices do bloco na *quadtree* ou seja o valor que estabelece a profundidade da *quadtree*.
MaxVBSizeInBytes **(ver 5.5.2.2)**	33554432 (32 MB)	> 0	O tamanho máximo do *vertex buffer* em bytes.
MaxIBSizeInBytes **(ver 5.5.2.2)**	1048576 (1 MB)	> 0	O tamanho máximo do *index buffer* em bytes.
CullMode **(ver 5.6.4)**	FrustumHorizon	Nenhum, Frustum, FrustumHorizon	O método de *culling* utilizado.
HorizonMaxError **(ver 5.6.4)**	8	> 0	O erro máximo em *pixels* do horizonte.
HeightProcessing **(ver 5.5.2)**	MultiStream	MultiStream, VertexTexture	O método utilizado no tratamento dos valores de elevação.

Tabela 6-2: Valores de configuração.

6.4. Metodologia

Na concretização dos testes procurou-se seguir uma abordagem que garantisse a obtenção de resultados fiáveis em todas as máquinas com o intuito de se chegar a conclusões válidas. Para isso, a recolha de dados de desempenho foi em todos eles feita de uma forma automática, através de um percurso de um minuto previamente construído e que está representado na **Figura 6-2** para cada um dos terrenos considerados. No decorrer deste percurso são recolhidos dados de desempenho para a memória com um intervalo de um segundo, mais especificamente o número de *frames* por segundo. No final de cada um dos testes, os resultados são gravados no disco num ficheiro com o formato CSV (*Comma Separated Values*) para depois poderem ser tratados no Excel.

A amostra em si é constituída por 60 valores, os *frames* por segundo obtidos ao longo do percurso. Estes valores vão ser utilizados na construção de um conjunto de gráficos, para cada um dos testes, permitindo observar a oscilação do número de *frames* por segundo ao longo do percurso para cada uma das variações dos itens de configuração (ver **6.3**) considerados. É ainda calculada a média de todos os valores com o intuito de se apresentar no final de cada teste uma tabela que permite comparar todos os resultados obtidos para esse teste em todas as máquinas/algoritmos/variação do item de configuração considerado. É de referir ainda que a recolha desses valores só começa 5 segundos após o início do teste de modo a colmatar possíveis oscilações resultantes de todos os acessos a disco necessários para se carregar os dados utilizados para construir um terreno, mais especificamente, o *height map*, o *color map*, o *normal map* e o *sky map*. (ver **6.2**).

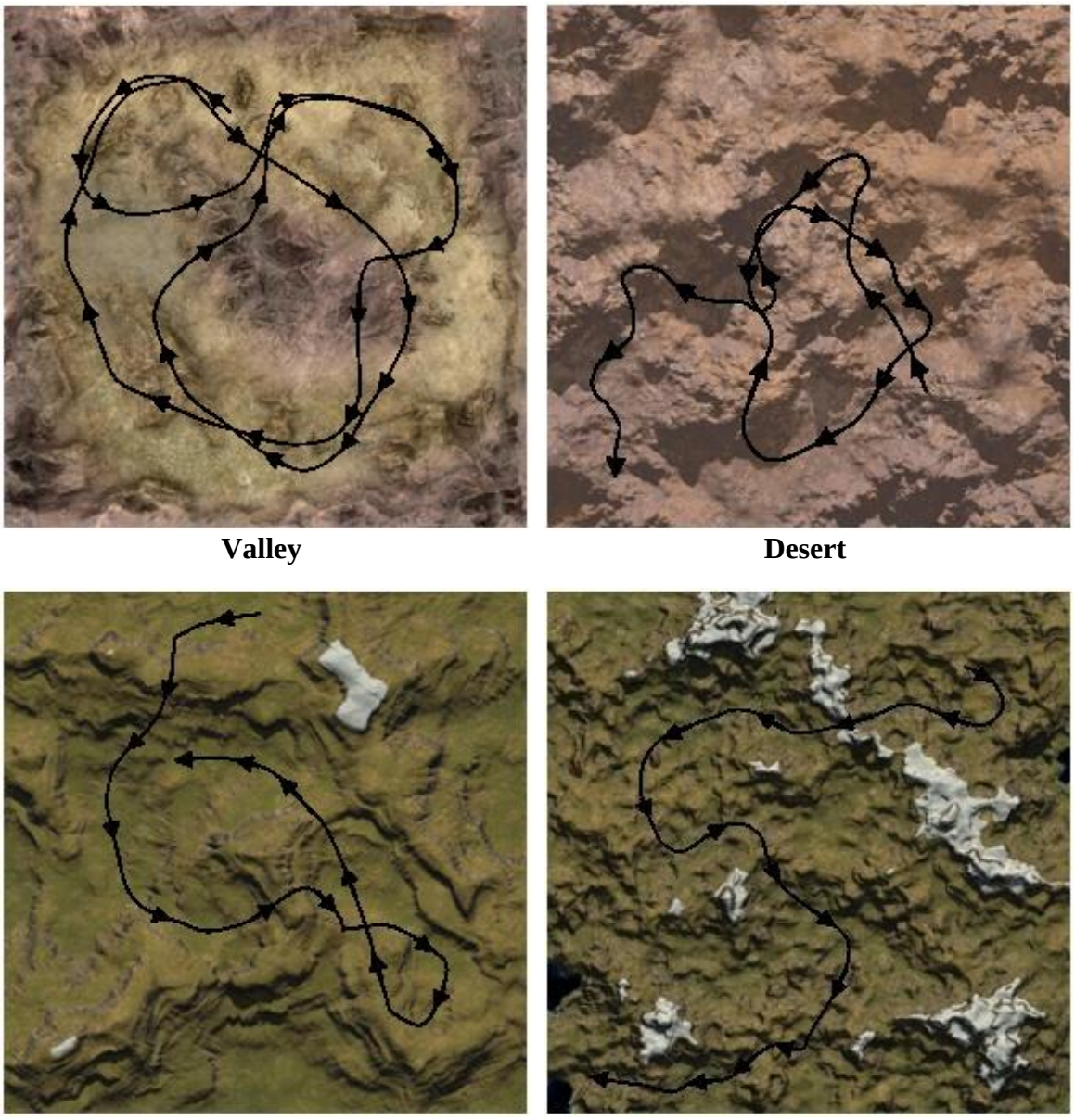


Figura 6-2: Percurso de um minuto efectuado em cada um dos terrenos.

6.5. Testes

Em cada uma das subsecções apresentam-se os resultados dos testes efectuados e discutem-se em detalhe esses resultados. Os algoritmos considerados foram o Brute Force, o Geomipmapping e o GPU Terrain Rendering, as máquinas, a N1, a N2, a D1 e a X1, os terrenos, o Valley, o Desert, o Terrace e o Highland e os parâmetros de configuração são os descritos na **Tabela 6-2**. Nos testes assumem-se os valores por omissão identificados para esses parâmetros na tabela quando não se atribui explicitamente um valor a um deles.

No que diz respeito à configuração de cada um dos testes, na maioria varia-se apenas o algoritmo e um parâmetro de configuração específico, utilizando-se para isso um terreno fixo, neste caso o Terrace. No entanto, e como é importante em alguns casos avaliar a variação de um determinado item de configuração face à dimensão do terreno, apresentam-se nessas situações os resultados para cada um dos terrenos e para cada um dos algoritmos. Os resultados em si são apresentados num gráfico onde por razões de espaço se considerou apenas uma máquina que se seleccionou de acordo com a relevância dos resultados aí obtidos para o teste. Utilizaram-se também uma ou mais tabelas onde se pode obter uma visão mais detalhada dos resultados e onde se varia o item de configuração considerado face ao algoritmo e à máquina ou face ao terreno e à máquina. De seguida, descrevem-se em detalhe cada um dos 11 testes realizados:

1. Número de Vértices do Bloco (ver **6.5.1**), em que se apresentam os resultados obtidos para blocos com um número de vértices distinto, neste caso 17, 33, 65 e 129 variando-se para isso o parâmetro `TerrainTileSize`. Este teste foi efectuado em cada uma das máquinas para os três algoritmos considerados e em todos os terrenos.
2. *Culling* (ver **6.5.2**), em que se apresentam os resultados obtidos para diferentes técnicas de *culling* (ver **3.3**) nomeadamente para o *frustum culling* (ver **3.3.2**) e para ao *frustum culling* mais a técnica de oclusão do algoritmo de Terrain Occlusion Culling with Horizons (ver **4.6**). Para isso variou-se o parâmetro `CullMode` tendo sido efectuado o teste em cada uma das máquinas para os três algoritmos considerados e em todos os terrenos.
3. Envio de Dados de Elevação (ver **6.5.3**), em que se apresentam os resultados obtidos com *multi-stream* (ver **5.5.2.2**) e *vertex textures* (ver **5.5.2.3**). Para isso variou-se o parâmetro `HeightProcessing`, tendo sido efectuado o teste nas duas máquinas que suportam *vertex textures*, a D1 e a X1 para os três algoritmos considerados e em todos os terrenos.
4. Nível de Detalhe do Horizonte (ver **6.5.4**), em que se apresentam os resultados obtidos para diferentes valores de nível de detalhe do horizonte, neste caso 4, 8 16 e 32 *pixels*. Para isso variou-se o parâmetro `HorizonMaxError`, tendo sido efectuado o teste no terreno Terrace (ver **6.2**) em cada uma das máquinas para cada um dos três algoritmos considerados.
5. Nível de Detalhe do Terreno (ver **6.5.5**), em que se apresentam os resultados obtidos para os diferentes valores de nível de detalhe do terreno considerados, neste caso 4, 8, 16 e 32 *pixels*. Para isso variou-se o parâmetro `LodMaxError`, tendo sido efectuado o teste no terreno Terrace (ver **6.2**) em cada uma das máquinas e nos dois algoritmos que controlam o nível de detalhe.
6. Coerência Espacial (ver **6.5.6**), em que se apresentam os resultados obtidos com as diferentes técnicas de correcção de falhas. Neste caso consideraram-se dois métodos: as *skirts* que podem ser utilizadas no Geomipmapping e no GPU Terrain Rendering e a actualização de índices (ver **5.6.2.1**) que pode ser utilizada apenas no algoritmo de Geomipmapping. No que diz respeito às *skirts*, testaram-se duas formas de as implementar enquanto método de correcção de falhas. A primeira utiliza o erro geométrico do bloco para determinar a altura de todas as *skirts* que o envolvem enviando-se todas elas para *rendering*. O segundo, uma optimização ao método inicial, utiliza apenas os valores do erro geométrico que envolvem o bloco

para determinar a altura das *skirts* e só envia para *rendering* as que intersectam o *frustum*. O parâmetro que variou neste caso foi o `SpatialCoherence` tendo sido efectuado o teste em cada uma das máquinas e nos dois algoritmos que controlam o nível de detalhe bem como em cada um dos terrenos.

7. Coerência Temporal (ver **6.5.7**), em que se apresentam os resultados obtidos pelo método utilizado na correcção do efeito de *popping* (ver **3.4.4.2**), o Geomorphing (ver **5.6.3**), comparando-se o impacto desta solução no desempenho. O parâmetro que variou neste caso foi o `TemporalCoherence` tendo sido efectuado o teste no terreno Terrace (ver **6.2**) em cada uma das máquinas e nos dois algoritmos que controlam o nível de detalhe.
8. Primitiva (ver **6.5.8**), em que se apresentam os resultados obtidos nos testes efectuados com os dois tipos de primitivas suportados: as listas de triângulos (ver **3.1.1**) e as tiras de triângulos (ver **3.1.3**). Para isso variou-se o parâmetro `Primitive`, tendo sido efectuado o teste no terreno Terrace (ver **6.2**) em cada uma das máquinas e para cada um dos três algoritmos, considerados.
9. *Vertex Cache* (ver **6.5.9**), em que se apresentam os resultados obtidos nos testes efectuados com diferentes valores para o número máximo de vértices por linha. Cada um desses valores tem como alvo um tamanho específico de *cache*. Mais especificamente, testaram-se valores de 4, 6, 8, 12 e de 33 vértices por linha. Como o último tem um tamanho igual ao do bloco equivale a uma situação normal em que não se tenta tirar proveito da *cache*. Na execução deste teste variou-se o parâmetro `PrimitiveBlockSize` tendo sido efectuado o teste no terreno Terrace (ver **6.2**) em cada uma das máquinas e para cada um dos três algoritmos.
10. *Fill rate* (ver **6.5.10**), em que se apresentam os resultados obtidos nos testes efectuados com diferentes valores de resolução, neste caso 640×480, 800×600, 1024×768 e 1280×768. Para isso variaram-se os parâmetros `ScreenWidth` e `ScreenHeight`, tendo sido efectuado o teste no Terreno Terrace (ver **6.2**) em cada uma das máquinas e para cada um dos três algoritmos considerados.
11. Algoritmo (ver **6.5.11**), em que se apresentam os resultados obtidos nos testes efectuados com os diferentes algoritmos em cada um dos terrenos considerados (ver **6.2**). Na execução deste teste variou-se o parâmetro `Algorithm`, tendo sido efectuado o teste em todos os terrenos (ver **6.2**) e em cada uma das máquinas.

6.5.1. Número de Vértices do Bloco

Como se pode verificar pela análise da **Figura 6-3**, da **Tabela 6-3**, **Tabela 6-4** e da **Tabela 6-5** o número de vértices do bloco tem um impacto significativo no número de *frames* por segundo obtidas. A primeira conclusão é que o número de vértices ideal para o bloco ao nível de desempenho varia com o algoritmo, mas depende também da máquina e da dimensão do terreno. No geral e, excluindo a máquina X1, os blocos 17×17 e 33×33 são de acordo com esta amostra as opções que apresentam melhores resultados ao nível do desempenho. Verifica-se também que o bloco 17×17 apresenta melhores resultados no algoritmo de GPU Terrain Rendering do que no Geomipmapping e no Brute Force. Nesta perspectiva há que realçar a diferença de abordagem entre o Brute Force, o Geomipmapping e o algoritmo de GPU Terrain Rendering. Mais precisamente, no Brute Force e no Geomipmapping o tamanho do bloco é fixo e o envio dos blocos para *rendering* depende apenas de intersectarem ou não o *frustum*. No caso do GPU Terrain Rendering, o tamanho do bloco pode aumentar e consequentemente o número de blocos enviados para *rendering* depende também da métrica de erro empregue. Salienta-se também que em todos os algoritmos se verifica que na máquina mais rápida e nos terrenos de maior dimensão se obtêm melhores resultados com blocos de dimensão superior a 17×17. No entanto, nas máquinas mais lentas, apenas no algoritmo de GPU Terrain Rendering o bloco de dimensão 17×17 continua a ter os melhores resultados em terrenos de dimensão superior.

Abordando agora os resultados obtidos na máquina X1, o comportamento é de certo modo extremo uma vez que na grande maioria dos testes o valor máximo considerado para o tamanho do bloco, 129×129, é o mais rápido. Este comportamento parece indicar que existe alguma contenção ao nível do CPU, já que à medida que o tamanho do bloco aumenta, a pesquisa na *quadtree* (ver **3.2.3.1**) passa a ser muito mais rápida, pois a recursividade efectuada não necessita de ser tão profunda para blocos de maior dimensão. Desta forma, a conclusão a tirar deste teste é que o número de vértices do bloco tem como se pode verificar um impacto diferente em função do algoritmo adoptado. Por outro lado atendendo aos resultados da máquina D1, quanto maior o desempenho da placa gráfica, maior pode ser a quantidade de geometria enviada em cada chamada à primitiva, o que é coerente de um modo geral com o principio subjacente aos algoritmos da classe *Tiled Blocks* (ver **4.1**).

TerrainTileSize	17	33	65	129

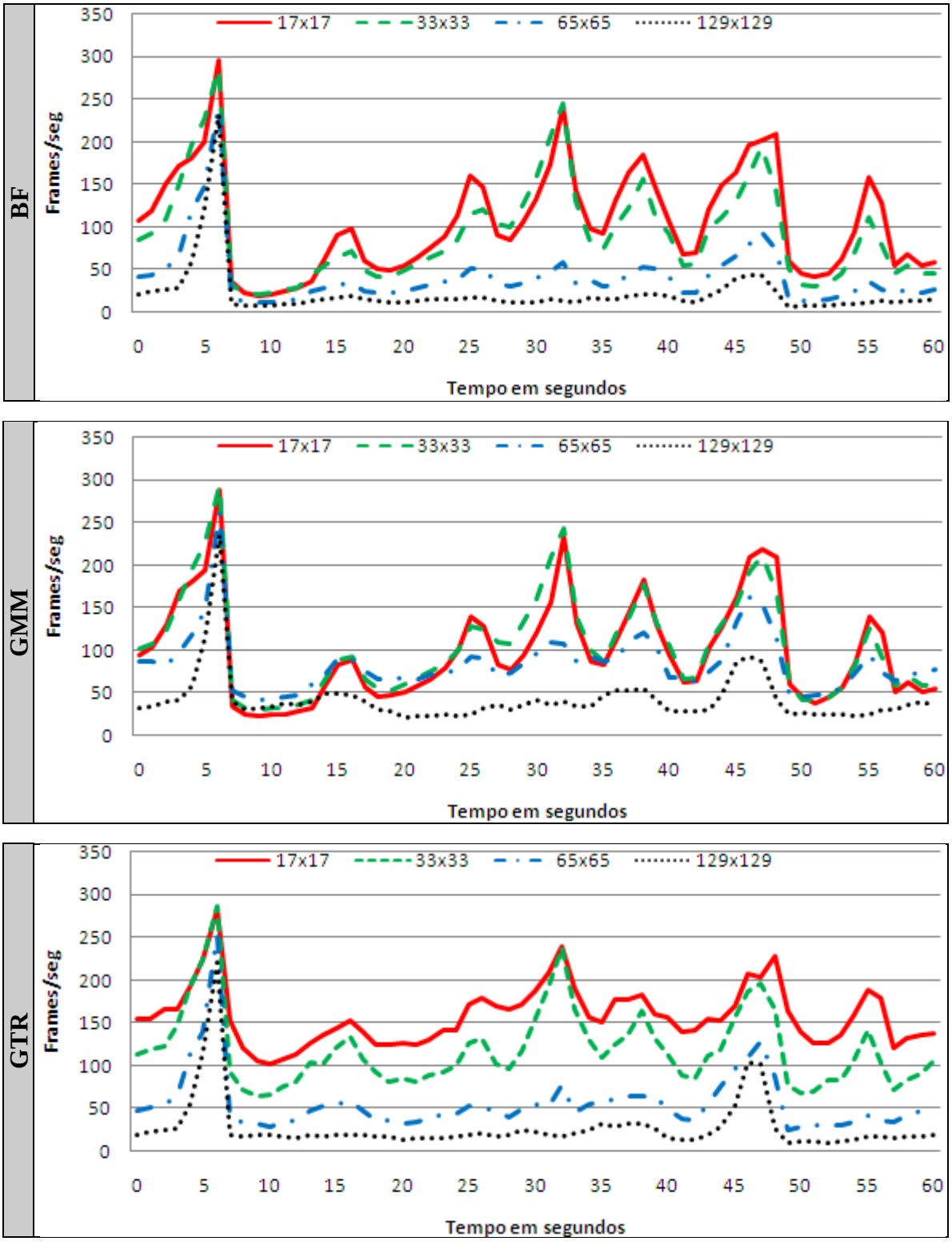


Figura 6-3: Número de vértices em cada um dos algoritmos na máquina N1.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
Valley	**17**	**202**	**309**	**1348**	339
	33	201	296	1304	632
	65	176	270	1234	1032
	129	158	225	1091	**1205**
Desert	**17**	**166**	**253**	837	113
	33	137	214	**1017**	259
	65	72	171	854	512
	129	43	133	705	**822**
Terrace	**17**	**103**	**191**	477	41
	33	90	171	**946**	126
	65	41	128	727	299
	129	20	88	497	**532**
Highland	**17**	33	65	99	7
	33	**50**	**115**	332	33
	65	22	75	**387**	113
	129	7	44	247	**235**

Tabela 6-3: M/FPS por máquina/terreno/número vértices bloco no algoritmo BF.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
Valley	**17**	**204**	**312**	1281	299
	33	196	293	**1301**	571
	65	172	267	1223	**975**
	129	155	223	1079	835
Desert	**17**	**171**	**290**	776	103
	33	156	274	**1207**	233
	65	139	235	1121	465
	129	91	190	918	**876**
Terrace	**17**	96	218	438	38
	33	**102**	**254**	1099	115
	65	83	128	**1210**	268
	129	40	88	732	**533**
Highland	**17**	31	67	92	6
	33	**52**	**199**	327	30
	65	50	160	**719**	121
	129	26	103	497	**242**

Tabela 6-4: M/FPS por máquina/terreno/número vértices bloco no algoritmo GMM.

		Média de frames/seg							
		N1		N2		D1		X1	
Valley	**17**	**193**		**300**		**1295**		310	
	33	**193**		289		1277		571	
	65	170		266		1218		983	
	129	155		223		1075		**1188**	
Desert	**17**	**178**		**273**		**1195**		210	
	33	165		250		1173		390	
	65	119		214		1055		659	
	129	99		179		913		**1046**	
Terrace	**17**	**157**		**249**		1166		130	
	33	116		210		**1275**		235	
	65	54		157		912		404	
	129	27		107		603		**632**	
Highland	**17**	**122**		**215**		579		63	
	33	81		168		**784**		128	
	65	32		111		563		252	
	129	12		66		354		**379**	

Tabela 6-5: M/FPS por máquina/terreno/número vértices bloco no algoritmo GTR.

6.5.2. Culling

Tendo em consideração a **Figura 6-4**, a **Tabela 6-6**, a **Tabela 6-7** e a **Tabela 6-8** a importância do *frustum culling* (ver **3.3.2**) parece incontestável em qualquer uma das máquinas em que foi efectuado o teste, bem como em qualquer um dos terrenos. Da mesma forma o adicionar de um algoritmo de oclusão tem também um efeito significativo ao nível do desempenho principalmente nas máquinas mais lentas, neste caso a N1 e a N2 e nos terrenos mais pesados, como o Highland. No caso do Geomipmapping, em terrenos de maior dimensão e na máquina mais rápida, a D1, o algoritmo de oclusão empregue não se revela tão eficiente como nas outras situações. Tal pode indicar que o tempo gasto na construção do horizonte e no *culling* dos blocos ocultos é superior ao tempo que o GPU levaria a processar os blocos não sendo por isso a solução mais eficiente. Por outro lado, no Geomipmapping o tamanho do bloco é fixo, pelo que a quantidade de blocos enviada é muito maior e consequentemente existe mais trabalho a efectuar ao nível do CPU.

Note-se também que a utilização de um algoritmo de oclusão tem um impacto muito maior no Brute Force do que nos algoritmos que reduzem o nível de detalhe. Tal era esperado uma vez que nesse caso a redução do número blocos, permite a eliminação de mais geometria do que nas outras situações.

A máquina X1 tem mais uma vez um comportamento diferente pois, os melhores desempenhos foram obtidos apenas com o *frustum culling* o que parece indicar mais uma vez a necessidade de se fornecer rapidamente a geometria à placa gráfica para se alcançar as melhores taxas desempenho, enviando-se logo que possível os blocos para o GPU.

CullMode	Nenhum	Frustum	Frustum,Horizon

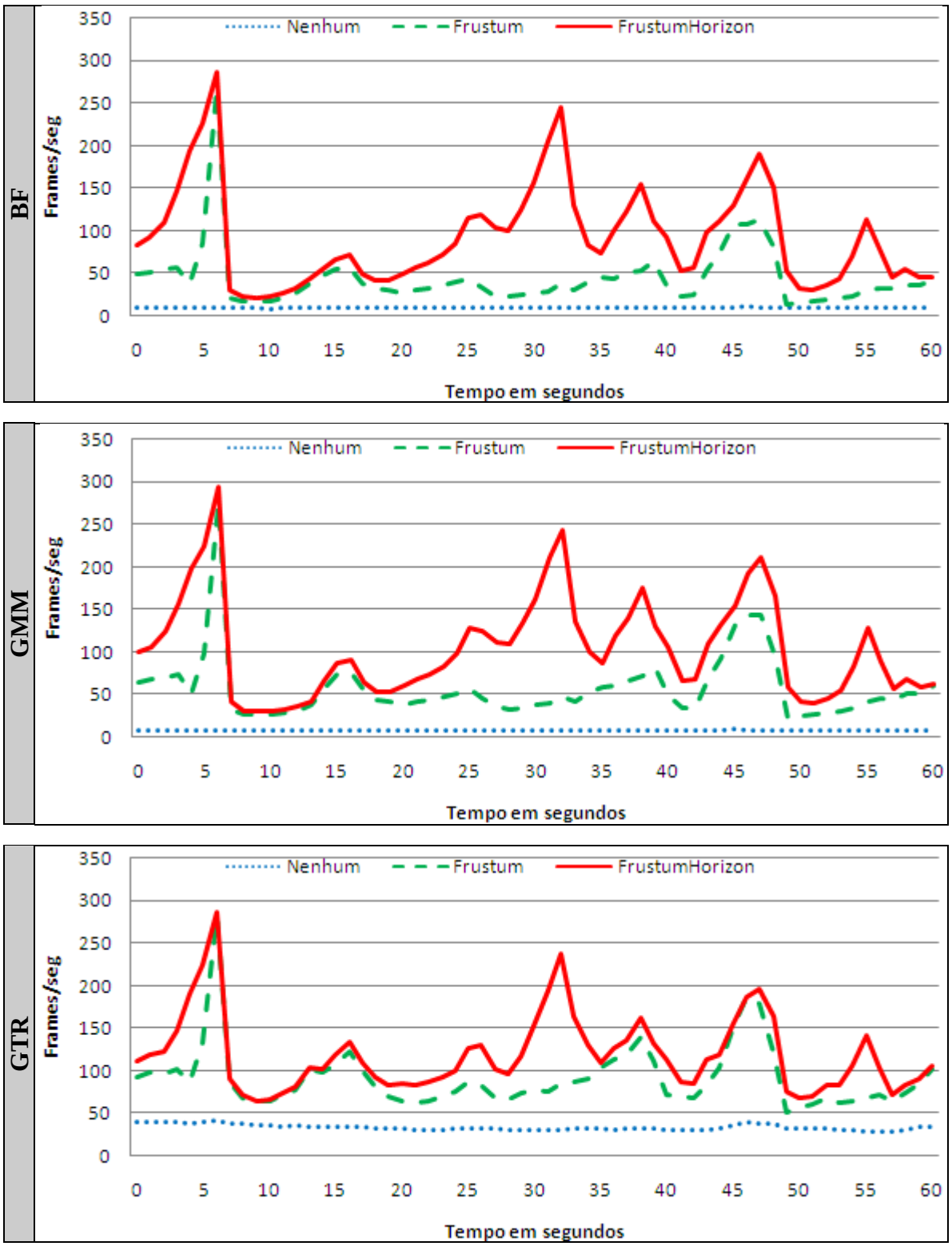


Figura 6-4: Método de *culling* em cada um dos algoritmos na máquina N1.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
Valley	***Nenhum***	136	157	802	657
	Frustum	199	294	1294	**1432**
	FrustumHorizon	**200**	**296**	**1304**	631
Desert	***Nenhum***	37	55	321	141
	Frustum	105	180	866	**746**
	FrustumHorizon	**137**	**214**	**1016**	259
Terrace	***Nenhum***	9	17	92	15
	Frustum	43	103	566	**289**
	FrustumHorizon	**90**	**171**	**946**	126
Highland	***Nenhum***	2	4	22	2
	Frustum	13	34	180	**64**
	FrustumHorizon	**50**	**115**	**329**	33

Tabela 6-6: M/FPS por máquina/terreno/método *culling* no algoritmo BF.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
Valley	***Nenhum***	118	164	818	462
	Frustum	**196**	291	1278	**1256**
	FrustumHorizon	**196**	**293**	**1299**	571
Desert	***Nenhum***	32	148	729	102
	Frustum	126	261	1201	**515**
	FrustumHorizon	**155**	**274**	**1203**	233
Terrace	***Nenhum***	8	94	215	13
	Frustum	55	228	**1299**	**202**
	FrustumHorizon	**103**	**254**	1111	115
Highland	***Nenhum***	2	26	31	2
	Frustum	17	172	**385**	**46**
	FrustumHorizon	**52**	**199**	330	30

Tabela 6-7: M/FPS por máquina/terreno/método *culling* no algoritmo GMM.

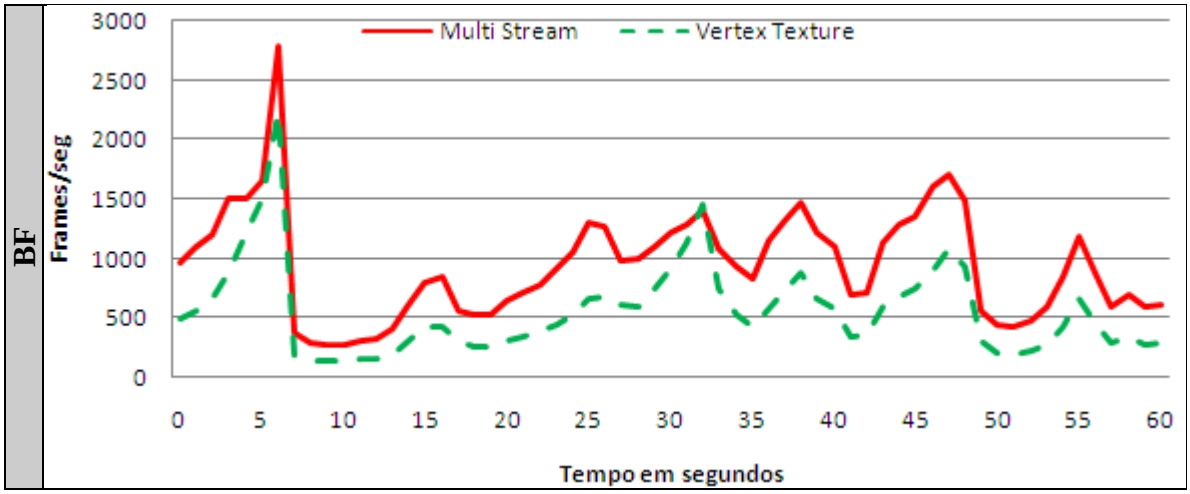
		Média de frames/seg							
		N1		**N2**		**D1**		**X1**	
Valley	***Nenhum***	120		149		760		502	
	Frustum	191		287		1266		**1281**	
	FrustumHorizon	**193**		**289**		**1278**		579	
Desert	***Nenhum***	94		126		661		414	
	Frustum	158		241		1135		**966**	
	FrustumHorizon	**165**		**251**		**1171**		390	
Terrace	***Nenhum***	33		55		302		112	
	Frustum	90		180		1048		**506**	
	FrustumHorizon	**117**		**210**		**1271**		235	
Highland	***Nenhum***	17		29		170		46	
	Frustum	50		123		595		**271**	
	FrustumHorizon	**81**		**168**		**780**		128	

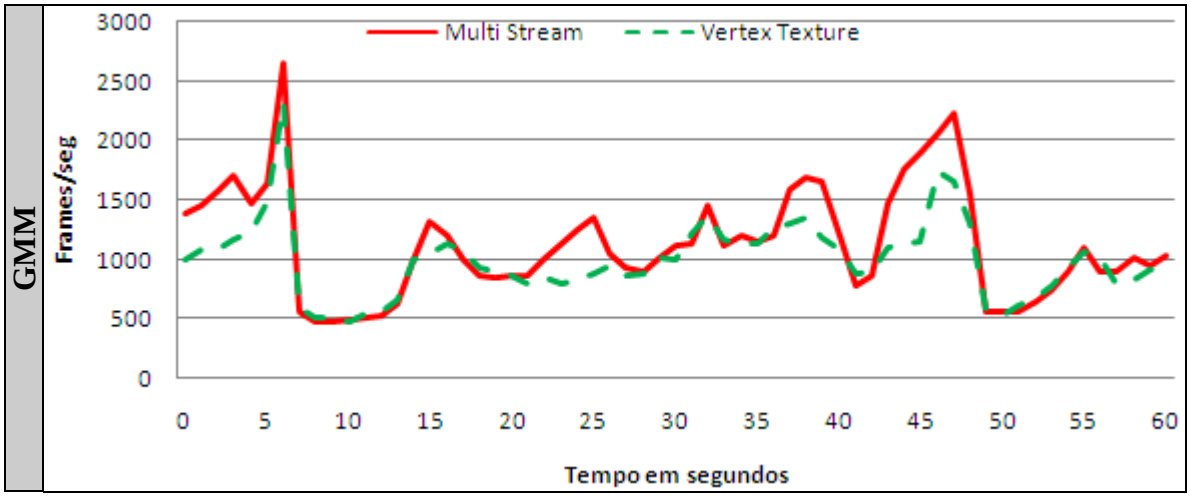
Tabela 6-8: M/FPS por máquina/terreno/método *culling* no algoritmo GTR.

6.5.3. Envio de Dados de Elevação

Este teste, ao contrário dos anteriores, foi efectuado apenas na máquina D1 e na máquina X1 já, que são as únicas do conjunto de máquinas utilizadas nos testes que suportam o *shader model* 3.0, e, consequentemente, a utilização de texturas ao nível do *vertex shader* (ver **5.5.2.3**). Assim, o objectivo foi comparar o desempenho das duas abordagens, isto é a utilização de duas *streams*, uma com os dados geométricos e outra com os dados de elevação (aqui designada de *multi-stream*) e a utilização de apenas uma *stream* com os dados geométricos e de uma textura para a obtenção dos dados de elevação no *vertex shader* (a *vertex texture*). Pela observação da **Figura 6-5** da **Tabela 6-10** e da **Tabela 6-11**, facilmente se verifica que na máquina D1 em todos os testes efectuados a utilização de múltiplas *streams* é claramente mais rápida, o que está de acordo com os resultados obtidos por outros autores (tal como é referido em [10] e [207], por exemplo) em que se verificou que o *vertex texture fetch* efectuado ao nível do *vertex shader*, não obstante a flexibilidade que oferece, continua a ser ainda o ponto de estrangulamento. Na máquina X1, muito embora a diferença entre as duas abordagens seja praticamente negligenciável, as *vertex textures* são em alguns casos ligeiramente mais rápidas no Geomipmapping e no GPU Terrain Rendering. Mais uma vez se levanta a possibilidade de o CPU ser o factor limitador nesta máquina, já que também aqui a abordagem que implica menos trabalho por parte do CPU é a vencedora em termos de desempenho. Deste modo conclui-se que apesar das vantagens oferecidas pelas *vertex textures*, não são ainda em termos de desempenho a opção mais rápida pelo menos nas máquinas de testes consideradas.

HeightProcessing	MultiStream	VertexTexture





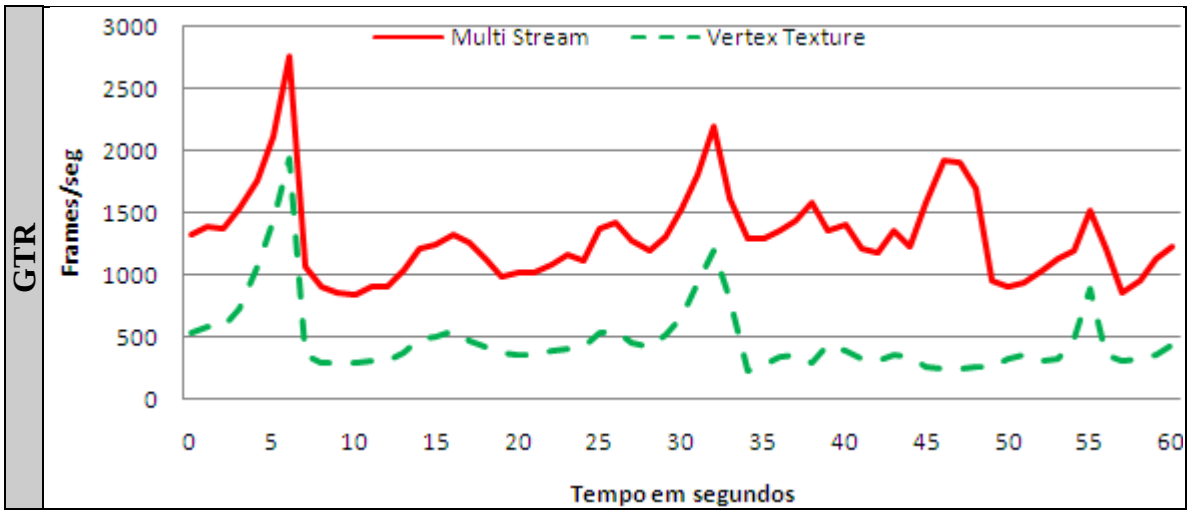


Figura 6-5: Envio dos dados de elevação em cada um dos algoritmos na máquina D1.

		Média de frames/seg							
		N1		**N2**		**D1**		**X1**	
Valley	***MultiStream***	N/D		N/D		**1307**		630	
	VertexTexture	N/D		N/D		1199		**644**	
Desert	***MultiStream***	N/D		N/D		**1019**		259	
	VertexTexture	N/D		N/D		784		**265**	
Terrace	***MultiStream***	N/D		N/D		**932**		**126**	
	VertexTexture	N/D		N/D		549		123	
Highland	***MultiStream***	N/D		N/D		**333**		**33**	
	VertexTexture	N/D		N/D		249		**33**	

Tabela 6-9: M/FPS por Máquina/terreno/envio dados elevação no algoritmo BF.

		Média de frames/seg							
		N1		**N2**		**D1**		**X1**	
Valley	***MultiStream***	N/D		N/D		**1254**		571	
	VertexTexture	N/D		N/D		1025		**579**	
Desert	***MultiStream***	N/D		N/D		**1193**		233	
	VertexTexture	N/D		N/D		850		**237**	
Terrace	***MultiStream***	N/D		N/D		**1131**		114	
	VertexTexture	N/D		N/D		1001		**115**	
Highland	***MultiStream***	N/D		N/D		**337**		**30**	
	VertexTexture	N/D		N/D		308		**30**	

Tabela 6-10: M/FPS por Máquina/terreno/envio dados elevação no algoritmo GMM.

		Média de frames/seg							
		N1		**N2**		**D1**		**X1**	
Valley	***MultiStream***	N/D		N/D		**1278**		579	
	VertexTexture	N/D		N/D		1042		**588**	
Desert	***MultiStream***	N/D		N/D		**1177**		390	
	VertexTexture	N/D		N/D		710		**398**	
Terrace	***MultiStream***	N/D		N/D		**1299**		235	
	VertexTexture	N/D		N/D		556		**240**	
Highland	***MultiStream***	N/D		N/D		**782**		128	
	VertexTexture	N/D		N/D		298		**131**	

Tabela 6-11: M/FPS por Máquina/terreno/envio dados elevação no algoritmo GTR.

6.5.4. Nível de Detalhe do Horizonte

O nível de detalhe do horizonte determina na execução do algoritmo de oclusão (ver **4.6**) a qualidade do horizonte produzido pela projecção do erro resultante da diferença máxima entre o plano dos mínimos quadrados e os planos máximos e mínimos em *pixels* e, consequentemente, tem influência no número de blocos detectados como ocultos pelo algoritmo (ver **5.6.4**). Assim, é expectável que para valores mais baixos do erro sejam detectados mais blocos ocultos. Em contrapartida implica uma maior recursividade, pois para refinar o horizonte é necessário descer um pouco mais na *quadtree* (ver **3.2.3.1**), o que pode, se o factor limitador for o CPU, contribuir negativamente para o desempenho. Pela observação da **Figura 6-6** e da **Tabela 6-12** chega-se à conclusão que aos valores mais baixos de erro correspondem as taxas de *frame rate* mais elevadas, principalmente nas máquinas mais lentas. Muito embora a diferença no desempenho não seja significativa entre 4 e 8 *pixels* de erro começa a sê-lo para valores mais elevados. Desta vez o comportamento esperado foi observado em todas as máquinas oscilando entre 4 e 8 *pixels*, o melhor valor em termos de desempenho.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
BF	***4***	**96**	**179**	672	67
	8	90	171	**943**	**126**
	16	83	147	289	32
	32	83	147	289	32
GMM	***4***	100	**257**	649	64
	8	**102**	254	**1132**	**126**
	16	79	154	297	32
	32	79	154	293	32
GTR	***4***	**123**	**218**	855	88
	8	117	210	**1271**	**235**
	16	95	168	374	40
	32	96	167	362	40

Tabela 6-12: M/FPS por máquina/algoritmo/detalhe horizonte no terreno Terrace.

HorizonMaxError	4	8	16	32

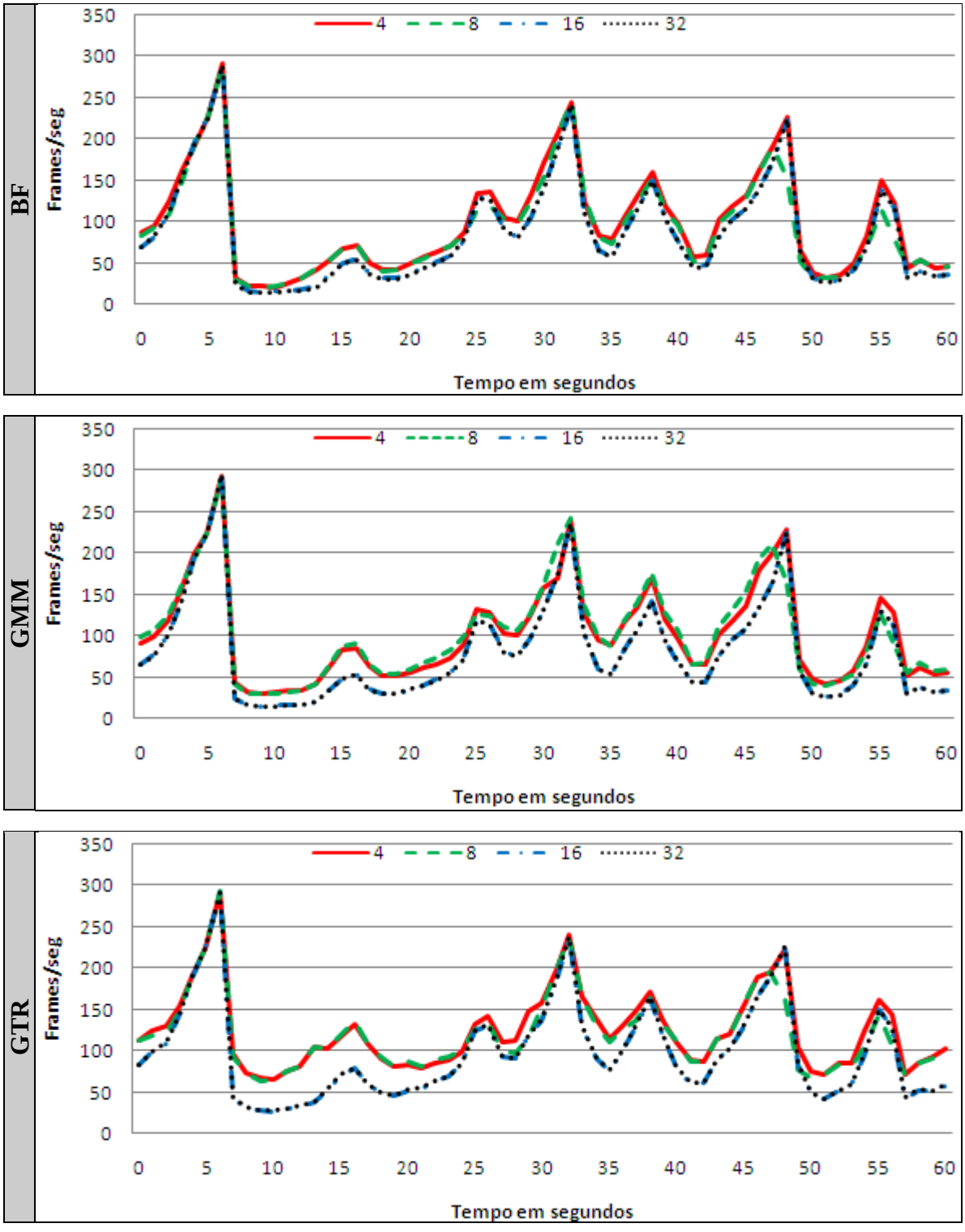


Figura 6-6: Nível de detalhe do horizonte em cada um dos algoritmos na máquina N1.

6.5.5. Nível de Detalhe do Terreno

Este teste abrange apenas os algoritmos de Geomipmapping e de GPU Terrain Rendering por serem os únicos em que existe uma variação do nível de detalhe do terreno. O objectivo é observar o comportamento em termos de desempenho mediante a variação de um parâmetro que estabelece o erro máximo em *pixels*. Este parâmetro permite controlar a qualidade, mais especificamente a quantidade de triângulos utilizada para representar cada um dos blocos de terreno, algo que é feito tendo em consideração a distância e o erro geométrico que resulta da diminuição do nível de detalhe em cada um dos blocos. Como foi utilizada a mesma métrica de erro nos dois algoritmos implementados, a comparação entre os dois torna-se mais fácil muito embora a forma como o detalhe diminui em cada um deles seja concretizada de modo completamente diferente. Isto é, no Geomipmapping ocorre pela diminuição do número de vértices em cada bloco, no GPU Terrain Rendering pelo aumento do tamanho do bloco e consequentemente pela diminuição não só do número de vértices como do número de chamadas à primitiva gráfica (ver **5.3**). Assim, neste teste o comportamento expectável é o aumento do desempenho à medida que o erro aumenta e paralelamente um decréscimo na qualidade do terreno. De facto esse foi o comportamento observado nos testes efectuados, tal como se pode verificar na **Figura 6-7** e na **Tabela 6-13**. A única excepção parece ser o algoritmo de Geomipmapping na máquina X1 onde, independentemente do nível de detalhe, o desempenho foi sempre o mesmo.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
GMM	***4***	92	211	1071	**115**
	8	103	202	1098	**115**
	16	115	287	1071	**115**
	32	**123**	**299**	**1099**	**115**
GTR	***4***	94	185	1059	169
	8	117	174	1272	148
	16	158	243	1564	351
	32	**189**	**273**	**1843**	**553**

Tabela 6-13: M/FPS por Máquina/algoritmo/nível detalhe no terreno Terrace.

LodMaxError	4	8	16	32

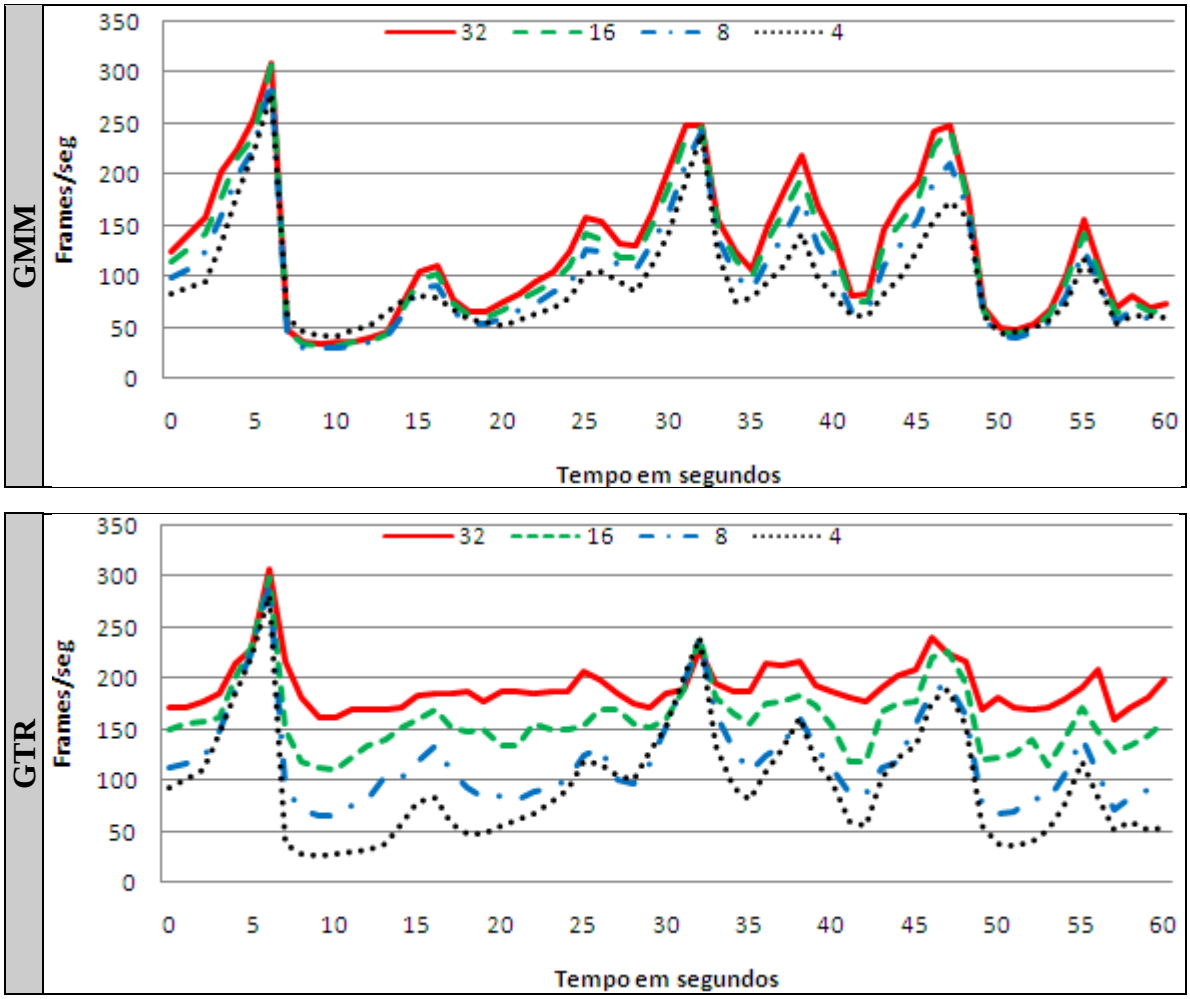


Figura 6-7: Nível de detalhe em cada um dos algoritmos na máquina N1.

Ao nível da qualidade da imagem, é apresentada na **Figura 6-8** uma comparação entre o efeito de um erro de 8 e de 32 *pixels* no algoritmo de Geomipmapping e no algoritmo de GPU Terrain Rendering. Como se pode verificar as zonas mais distantes tem menos detalhe, principalmente com 32 *pixels* de erro, muito embora esse efeito seja mais fácil de distinguir no caso do algoritmo de Geomipmapping. Para isso basta observar em detalhe as zonas marcadas com um círculo preto. Independentemente dos resultados pela análise do terreno produzido por cada um dos valores de erro, conclui-se que o valor preferencial situa-se entre 4 e 8 *pixels*, tendo-se dado mais importância na análise deste parâmetro a questões de qualidade do que a questões de desempenho já que o objectivo foi maximizar o realismo da cena.

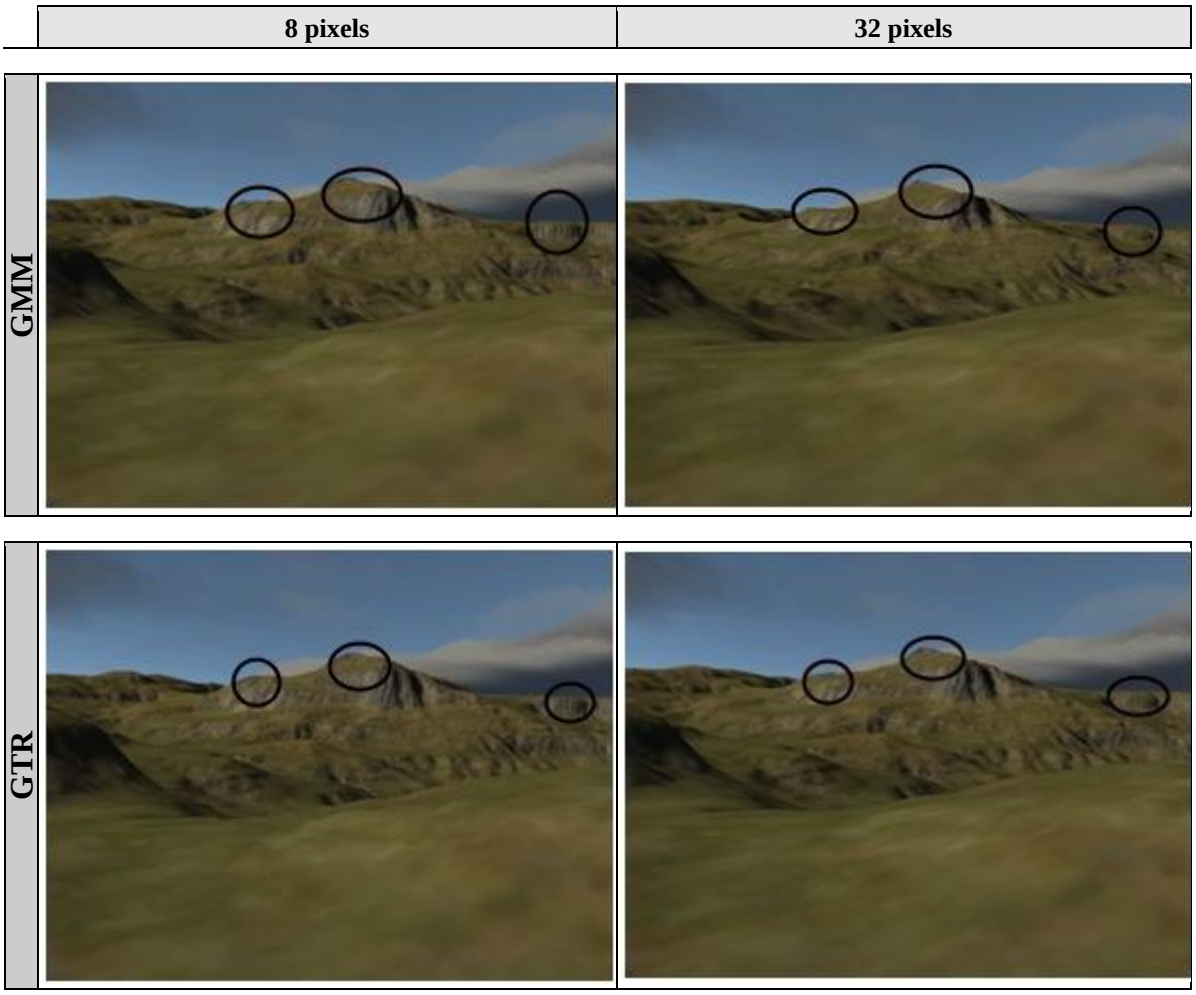


Figura 6-8: Qualidade de imagem com um erro de 8 e 32 *pixels* nos dois algoritmos.

6.5.6. Coerência Espacial

Tal como no teste anterior também aqui foram considerados apenas o algoritmo de Geomipmapping e o algoritmo de GPU Terrain Rendering já que o objectivo é avaliar o desempenho das diferentes técnicas utilizadas na correcção de falhas (ver **3.4.4.1**) causadas por blocos adjacentes de diferentes níveis de detalhe empregues nos algoritmos considerados. Mais especificamente, estes algoritmos controlam o nível de detalhe por bloco independentemente do nível de detalhe dos blocos vizinhos, o que significa que blocos adjacentes podem ter níveis de detalhe diferentes dando assim origem a falhas que tem um impacto muito significativo a nível visual. No teste em si foram considerados dois métodos, um que se aplica aos dois algoritmos, as *skirts* (ver **5.6.2.2**) e um que pode ser aplicado apenas ao algoritmo de Geomipmapping, a actualização de índices (ver **5.6.2.1**). Para as *skirts* consideraram-se duas variações (ver **5.4.2** e **5.5.1.1**): a primeira, utiliza o erro geométrico do bloco para determinar a altura de todas as *skirts* que o envolvem enviando-se todas elas para *rendering*; a segunda, uma optimização ao método inicial, utiliza apenas os valores do erro geométrico que envolvem o bloco para determinar a altura das *skirts*, e só envia para *rendering* as que intersectam o *frustum*. Para comparar o impacto de cada um dos métodos apresenta-se também a média de *frames* por segundo obtida sem nenhuma das técnicas de correcção de falhas.

Como seria de prever e se pode verificar pela análise da **Figura 6-9** da **Tabela 6-14** e da **Tabela 6-15**, não utilizar nenhuma técnica de correcção de falhas é a opção que tem melhor desempenho em todas as máquinas, no entanto, a diferença não é significativa. Resta agora avaliar o impacto das duas variações das *skirts* e da actualização de índices. No caso do Geomipmapping as *skirts* foram, nas máquinas mais rápidas, o método de correcção de falhas com melhor desempenho. A versão optimizada das *skirts* teve mais impacto no algoritmo de GPU Terrain Rendering onde se justifica de alguma forma a sua aplicação muito embora os ganhos ao nível de desempenho tenham sido muito reduzidos.

De realçar o facto de a diferença entre não utilizar nenhuma técnica de correcção de falhas e utilizar as *skirts* não é na maioria dos casos muito significativo, o que indica que estas estão a ter um impacto reduzido no desempenho. No caso da actualização de índices o método esteve muito aquém dos resultados esperados tendo um pior desempenho principalmente nas máquinas mais rápidas, a D1 e a X1. Assim em qualquer um dos casos as *skirts* são o método recomendado até porque a nível visual não se detectaram nenhuns artefactos, o que está em grande parte relacionado com o valor de erro em *pixels* considerado que, sendo suficientemente baixo, torna este método uma opção efectiva na correcção das falhas entre blocos adjacentes com diferentes níveis de detalhe.

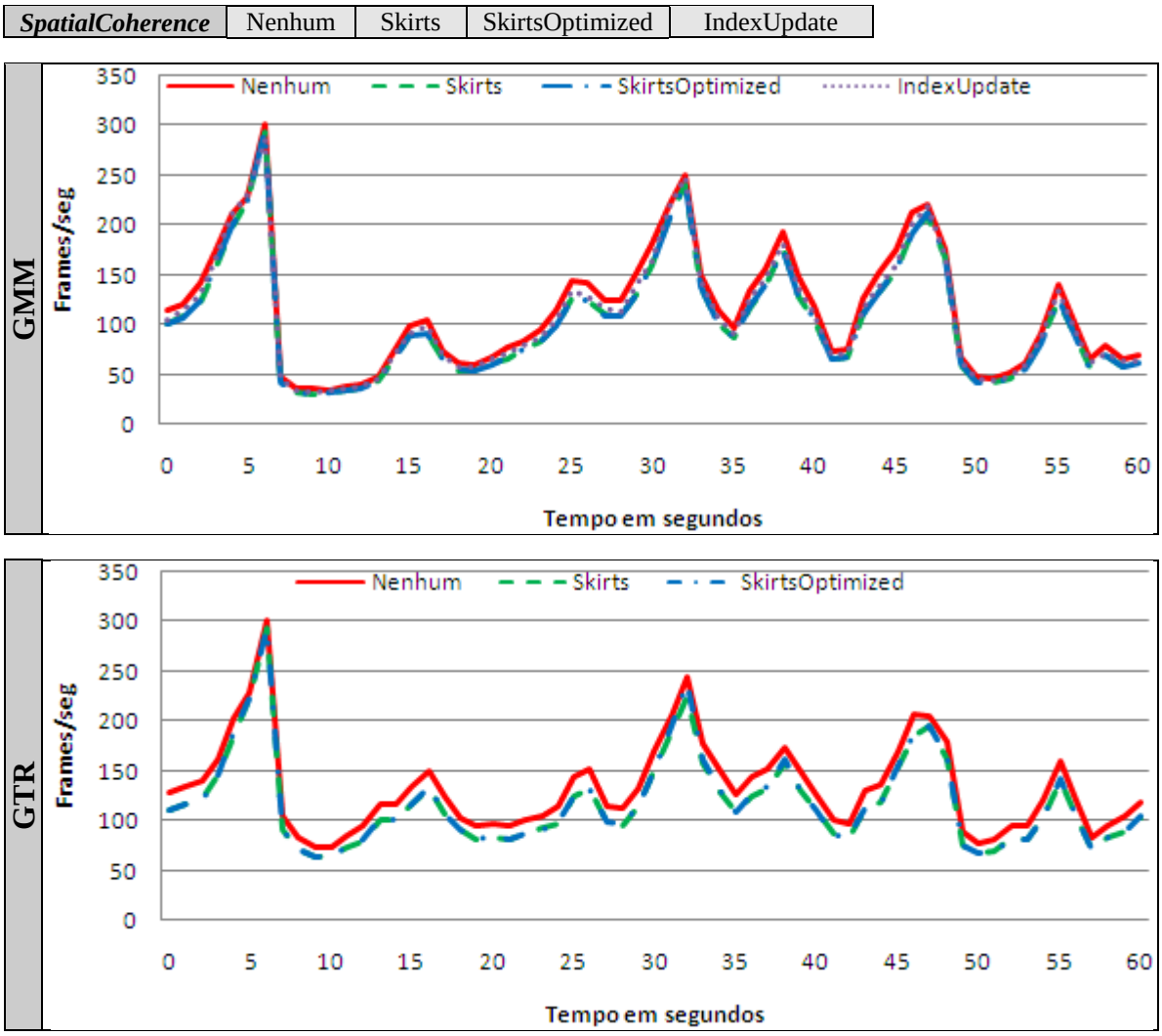


Figura 6-9: Coerência espacial em cada um dos algoritmos na máquina N1.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
Valley	***Nenhum***	**205**	**299**	**1327**	**614**
	Skirts	195	290	1278	599
	SkirtsOptimized	197	292	1274	571
	IndexUpdate	203	278	1148	281
Desert	***Nenhum***	**167**	**281**	**1247**	**251**
	Skirts	156	270	1198	244
	SkirtsOptimized	156	272	1185	233
	IndexUpdate	163	241	763	145
Terrace	***Nenhum***	**113**	**266**	**844**	**121**
	Skirts	102	251	808	118
	SkirtsOptimized	102	252	774	113
	IndexUpdate	107	203	455	71
Highland	***Nenhum***	**58**	**210**	**357**	**32**
	Skirts	52	198	347	31
	SkirtsOptimized	52	195	335	30
	IndexUpdate	55	133	194	22

Tabela 6-14: M/FPS por máquina/terreno/coerência espacial no algoritmo GMM.

		Média de frames/seg			
		N1	**N2**	**D1**	**X1**
Valley	***Nenhum***	**201**	**296**	**1317**	**614**
	Skirts	190	284	1264	608
	SkirtsOptimized	194	288	1288	579
Desert	***Nenhum***	**173**	**256**	**1203**	**416**
	Skirts	162	245	1159	410
	SkirtsOptimized	165	249	1174	389
Terrace	***Nenhum***	**131**	**214**	**996**	**240**
	Skirts	115	202	945	237
	SkirtsOptimized	117	205	954	224
Highland	***Nenhum***	**95**	**177**	**830**	**137**
	Skirts	81	165	777	135
	SkirtsOptimized	82	168	782	128

Tabela 6-15: M/FPS por máquina/terreno/coerência espacial no algoritmo GTR.

6.5.7. Coerência Temporal

Neste teste procurou-se investigar o impacto no desempenho da técnica de Geomorphing utilizada para tratamento do efeito de *popping* (ver **3.4.4.2**) nas diferentes máquinas e nos diferentes algoritmos que empregam técnicas de nível de detalhe. Como se pode verificar pela análise da **Figura 6-10** e da **Tabela 6-16**, é praticamente negligenciável o seu impacto, muito embora não seja como seria de esperar a solução mais rápida pois existe uma diferença ao nível do desempenho em relação à versão sem tratamento do *popping*, que não é, no entanto, significativa.

Se tivermos em consideração os benefícios que uma técnica como o Geomorphing traz nomeadamente ao nível do realismo, este pequeno impacto no desempenho é praticamente negligenciável.

A nível visual, muito embora seja difícil de representar, o movimento dos vértices é praticamente indetectável, principalmente se o nível de detalhe do terreno tiver em consideração um erro em *pixels* relativamente baixo como é o caso. Ou seja, existe uma transição suave dos vértices tanto numa situação de diminuição como de aumento do nível de detalhe, sem que tal seja perceptível para o utilizador.

Assim, tendo em conta o impacto observado, e os benefícios que advêm da utilização desta técnica, conclui-se que é uma boa opção em qualquer um dos algoritmos de nível de detalhe implementados.

		Média de frames/seg							
		N1		**N2**		**D1**		**X1**	
GMM	***Nenhum***	**105**		**261**		**1172**		**118**	
	Geomorphing	103		254		1131		115	
GTR	***Nenhum***	**127**		**217**		**1322**		**247**	
	Geomorphing	117		210		1274		235	

Tabela 6-16: M/FPS por máquina/algoritmo/coerência temporal no terreno Terrace.

TemporalCoherence	Nenhum	Geomorphing

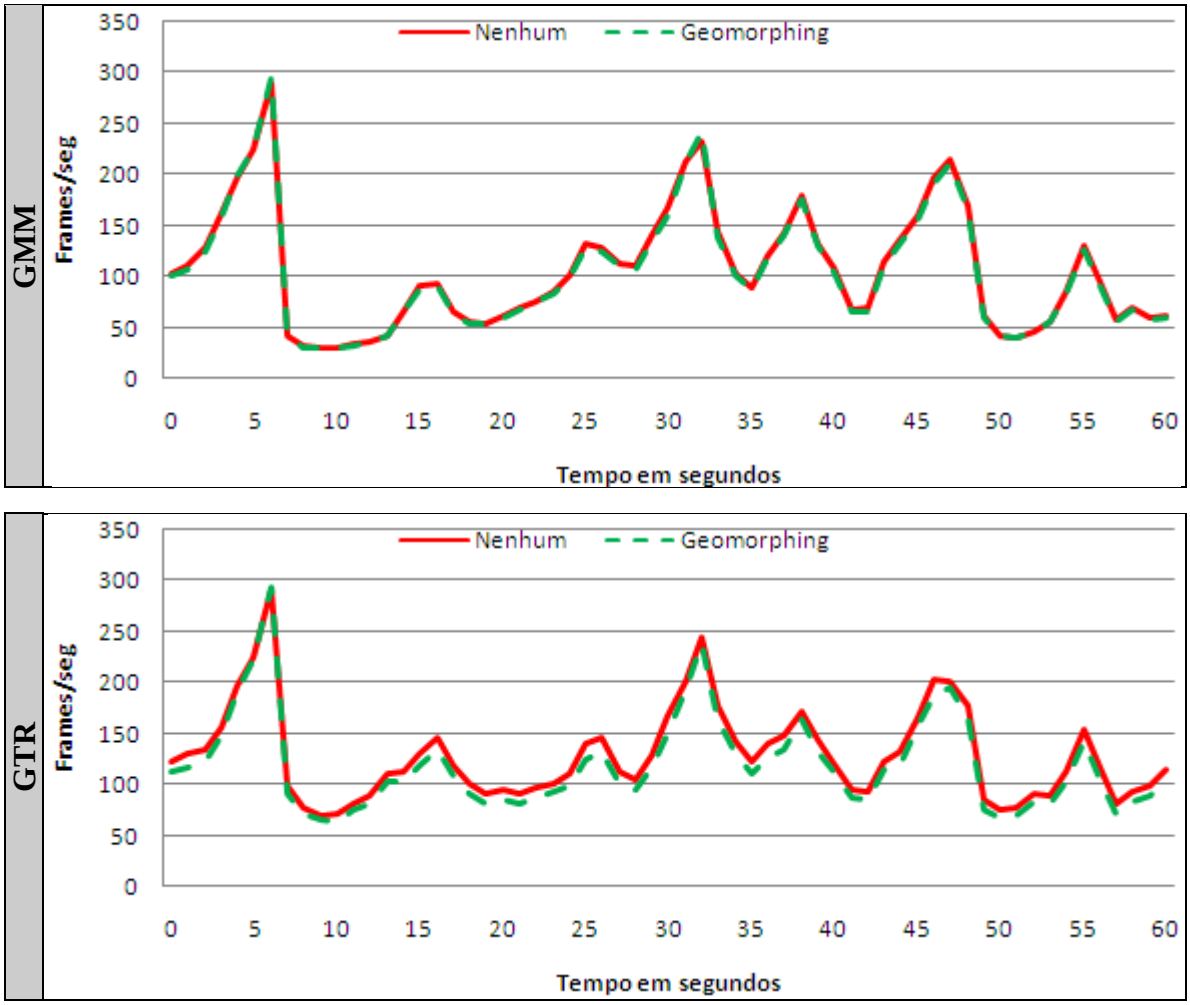


Figura 6-10: Coerência temporal em cada um dos algoritmos na máquina N1.

6.5.8. Primitiva

O objectivo deste teste foi verificar se existe alguma diferença ao nível do desempenho entre os dois tipos de primitivas suportadas, isto é, listas de triângulos (ver **3.1.1**) e tiras de triângulos (ver **3.1.3**). Como se pode verificar pela análise da **Figura 6-11** e da **Tabela 6-17** o desempenho é praticamente igual em qualquer uma das máquinas/algoritmos e quando existe diferença esta é demasiado pequena para ser considerada. Assim, deste teste retira-se apenas que a selecção de uma ou outra pode acabar por depender apenas do espaço ocupado na lista de índices e, nesse caso, as tiras de triângulos são claramente a abordagem que implica menos espaço tal como se refere em **3.1.4**.

Primitive	IndexedTriangleStrips	IndexedTriangleLists

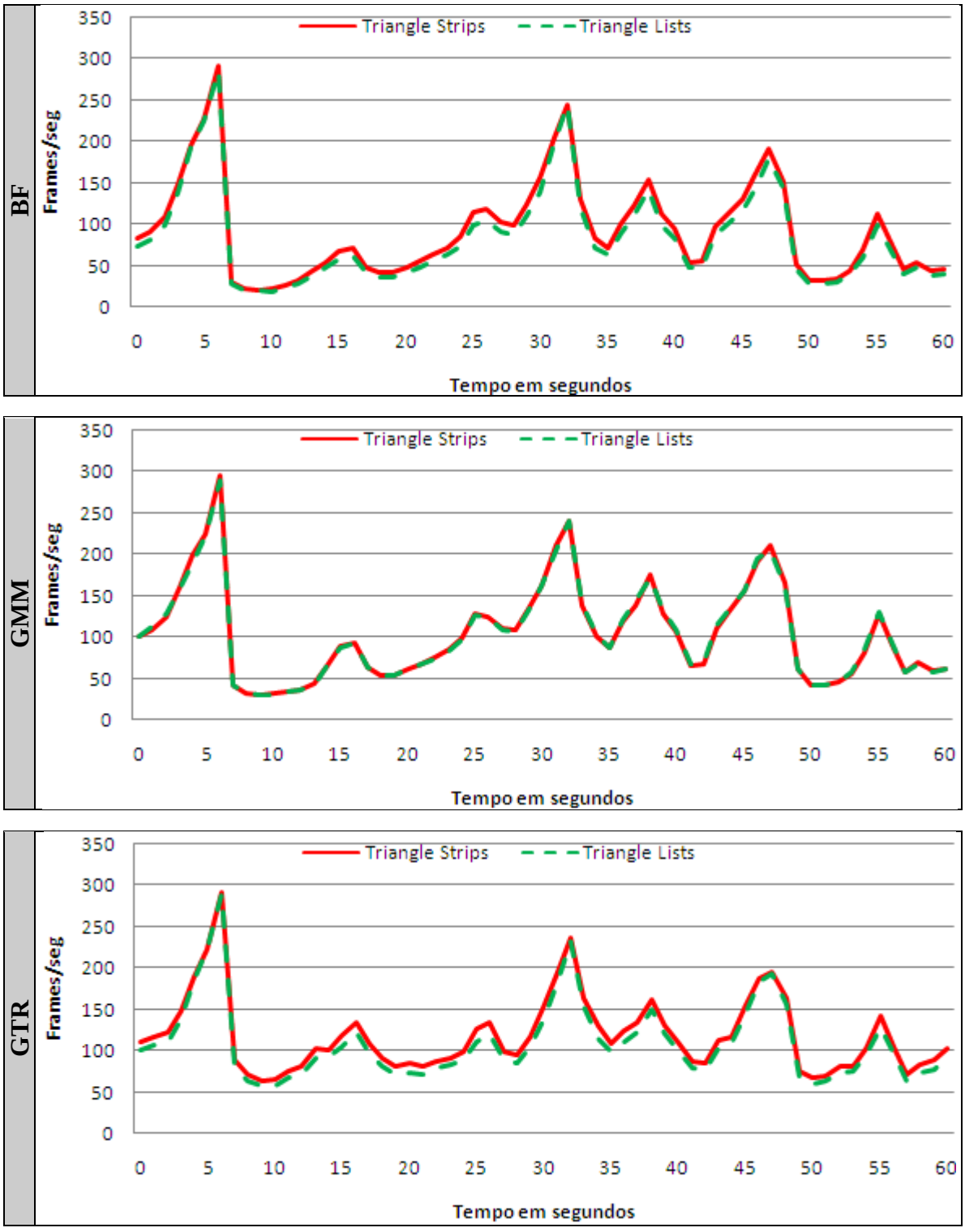


Figura 6-11: Primitiva em cada um dos algoritmos na máquina N1.

		Média de frames/seg							
		N1		**N2**		**D1**		**X1**	
BF	***IndexedTriangleLists***	81		**171**		935		**126**	
	IndexedTriangleStrips	**90**		**171**		**939**		**126**	
GMM	***IndexedTriangleLists***	**102**		**254**		**1145**		107	
	IndexedTriangleStrips	**102**		**254**		1130		**114**	
GTR	***IndexedTriangleLists***	107		**210**		**1277**		233	
	IndexedTriangleStrips	**117**		**210**		1270		**234**	

Tabela 6-17: M/FPS por máquina/algoritmo/primitiva no terreno Terrace.

6.5.9. Vertex Cache

A *vertex cache* é tal como foi referido em **3.1**, uma pequena *cache* que armazena os vértices recentemente transformados evitando dessa forma o seu processamento mais do que uma vez no *rendering* de uma primitiva. Para tirar proveito desta capacidade é necessário que na construção da malha triangular a sequência de vértices maximize a reutilização da *cache*. Tendo em conta que as linhas de triângulos partilham vértices, é necessário que o número de vértices em cada linha tenha um comprimento igual ou inferior a metade do tamanho da *cache* (ver **5.5.1.2**). Desta forma, quando a segunda linha de triângulos se iniciar muitos dos vértices vão estar ainda em *cache* não sendo por isso necessário efectuar de novo a sua transformação. Como a maioria das placas gráficas não permite obter o tamanho da *cache* e essa informação também não é tipicamente disponibilizada pelos fabricantes, a abordagem seguida foi o teste de diferentes máximos de vértices por linha que representassem os valores de *cache* mais comuns [65], com o intuito de se detectar uma possível alteração no desempenho. Assim, testaram-se valores de 4, 6, 8 e 12 vértices por linha que representam uma *vertex cache* alvo de 8, 12, 16 e 24 vértices respectivamente. Para se comparar com uma situação em que não te tenta tirar proveito da *vertex cache*, testou-se um valor adicional igual ao tamanho do bloco considerado no teste, isto é, 33. Dos testes efectuados apenas se obtiveram resultados significativos na máquina N2, mais especificamente foi observado nesta máquina uma variação não desprezável da média de *frames* por segundo com uma *cache* de 8 e 12 vértices (4 e 6 vértices por linha respectivamente) sobretudo no Brute Force e no GPU Terrain Rendering (ver **4.8**). Esse comportamento pode ser observado na **Figura 6-12** onde se apresentam os gráficos de desempenho por algoritmo relativos à máquina N2 e na **Tabela 6-18**, onde é efectuado um resumo da média de *frames* por segundo obtida em cada máquina/algoritmo para cada um dos valores considerados. Em relação às máquinas mais rápidas D1 e X1, coloca-se a hipótese de o tamanho da *cache* ser superior ao tamanho do bloco o que faria com que esta optimização para o tamanho de bloco considerado nestes testes (33 vértices) não tivesse nenhum efeito. Na máquina N1 por sua vez não foi observada uma diferença significativa no número de *frames* por segundo pelo que se coloca a hipótese de não existir uma *cache* de vértices ou a *driver* da placa gráfica não estar a tirar partido da *cache* existente. Não foi, no entanto, possível confirmar estas hipóteses já que não existe nenhuma informação do fabricante que permita chegar a alguma conclusão relativamente ao tamanho da *vertex cache*.

PrimitiveBlockSize	4	6	8	12	33

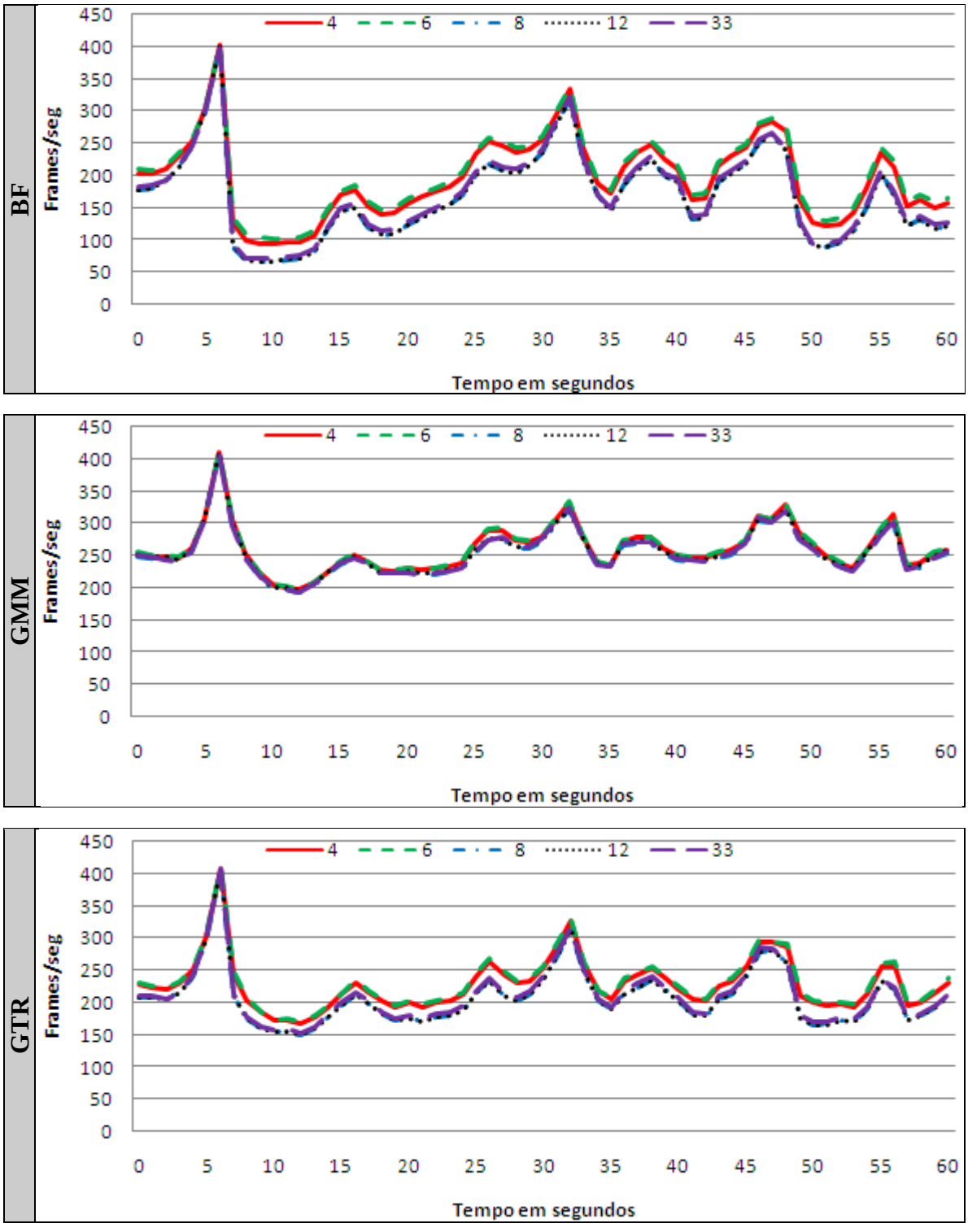


Figura 6-12: Diferentes vértices por linha em cada um dos algoritmos na máquina N2.

		Média de frames/seg			
		N1	N2	D1	X1
BF	*4*	91	195	921	125
	6	88	200	946	126
	8	92	168	949	126
	12	91	167	1000	126
	33	90	171	946	126
GMM	*4*	98	260	1117	106
	6	99	261	1113	114
	8	100	253	1067	114
	12	101	254	1146	114
	33	102	254	1099	115
GTR	*4*	129	227	1274	224
	6	118	230	1302	224
	8	121	208	1315	224
	12	116	207	1338	224
	33	116	210	1275	235

Tabela 6-18: M/FPS por Máquina/algoritmo/max. vértices/linha no terreno Terrace.

6.5.10. Fill Rate

O objectivo deste teste foi determinar o impacto da mudança de resolução no método desenvolvido, e mais especificamente verificar se existe uma limitação ao nível do *pixel shader* já que quanto maior for a resolução, maior é o número de *pixels* que a placa gráfica tem de processar, ou seja mais vezes tem de ser executado código ao nível do *pixel shader*. Como se pode verificar pela análise da **Figura 6-13** e da **Tabela 6-19**, a média de *frames* por segundo diminui em todas as máquinas/algoritmos à medida que a resolução aumenta, o que era de esperar. No entanto, não existe uma variação muito significativa, ou seja, os algoritmos considerados escalam bem para maiores resoluções. É de realçar, no entanto, que esta implementação utiliza apenas duas texturas ao nível do *pixel shader*, o *color map* responsável pela cor do terreno e o *normal map* utilizado no cálculo da iluminação.

ScreenWidth/ScreenHeight	640 × 480	800 × 600	1024 × 768	1280 × 768

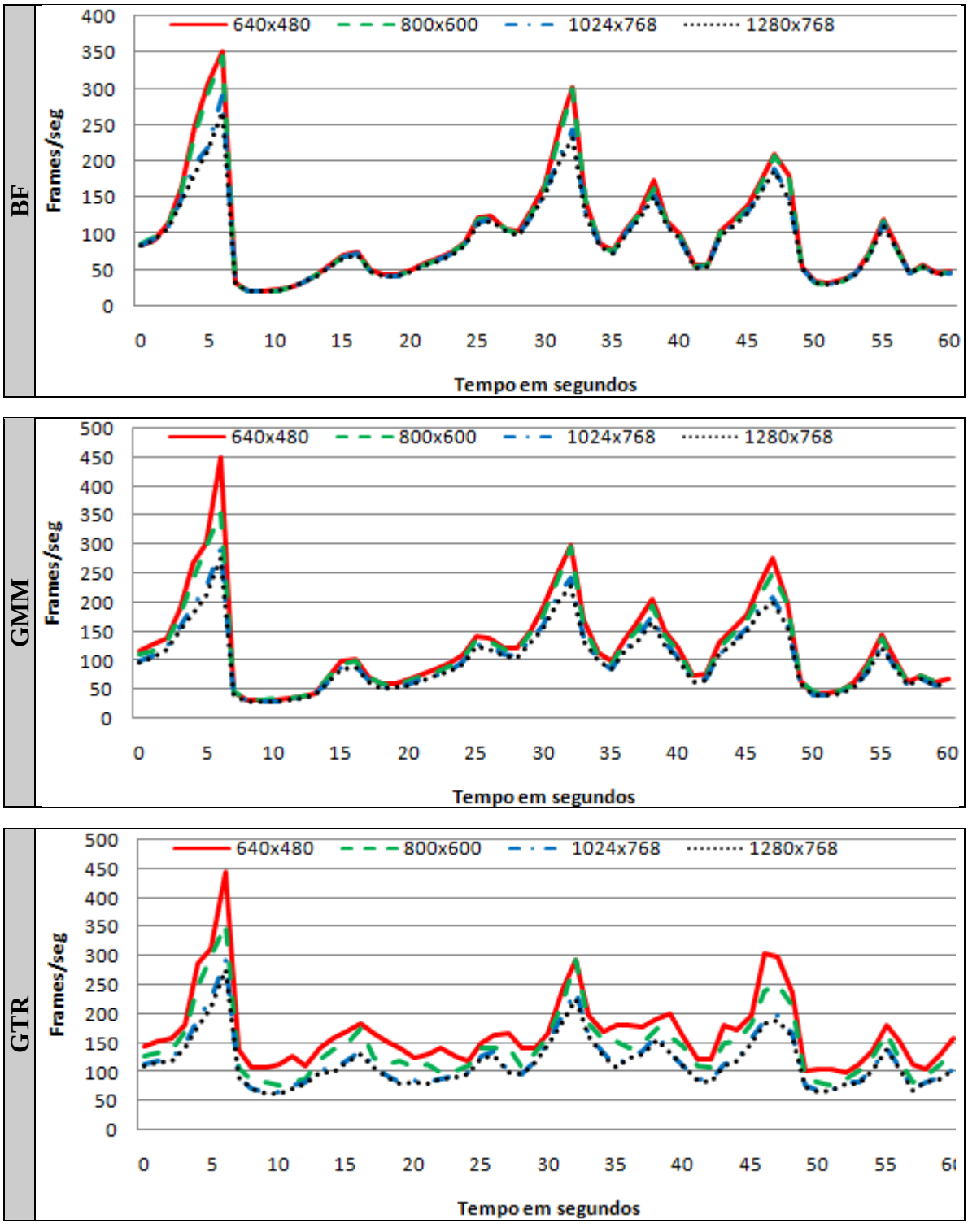


Figura 6-13: Resolução de ecrã em cada um dos algoritmos na máquina N1.

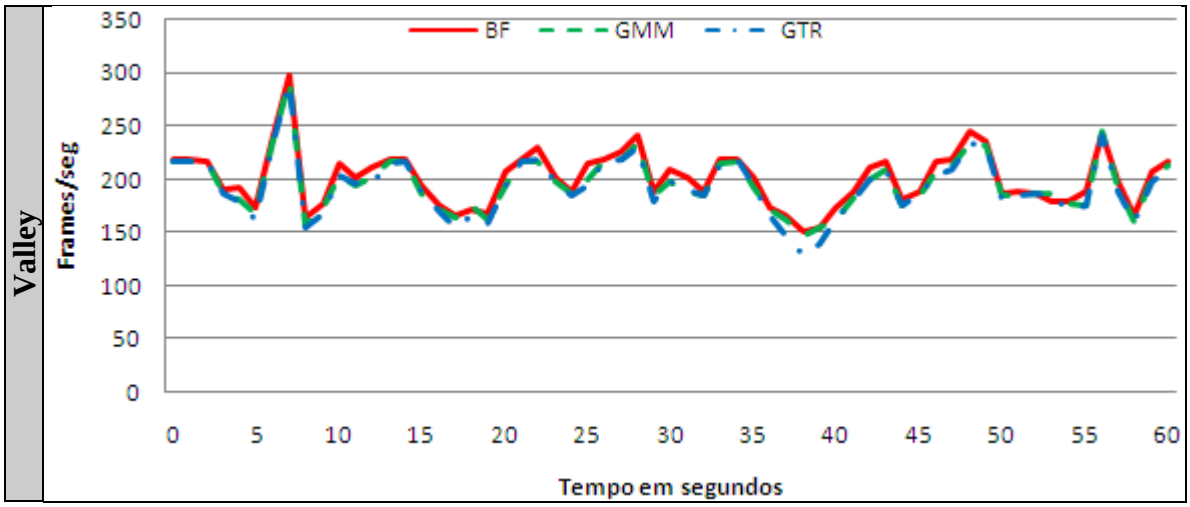
		Média de frames/seg							
		N1		N2		D1		X1	
BF	640 × 480	100		271		1030		144	
	800 × 600	98		209		1031		136	
	1024 × 768	90		171		939		126	
	1280 × 768	88		150		885		116	
GMM	640 × 480	121		576		1209		129	
	800 × 600	113		363		1125		123	
	1024 × 768	102		254		1082		115	
	1280 × 768	99		212		1075		106	
GTR	640 × 480	165		421		1729		362	
	800 × 600	140		282		1578		289	
	1024 × 768	117		210		1274		235	
	1280 × 768	114		180		1163		214	

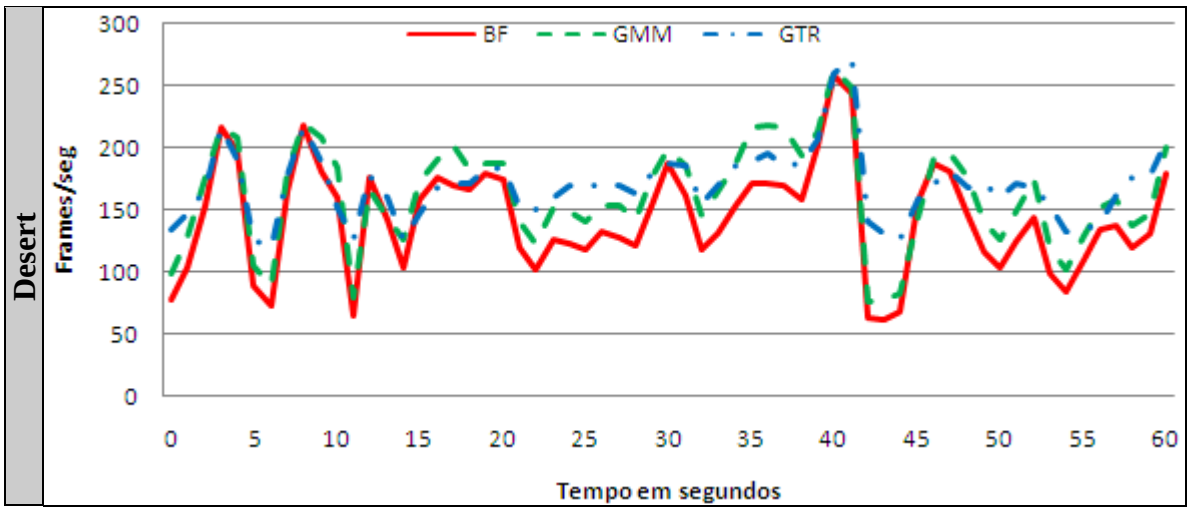
Tabela 6-19: M/FPS por máquina/algoritmo/resolução ecrã no terreno Terrace.

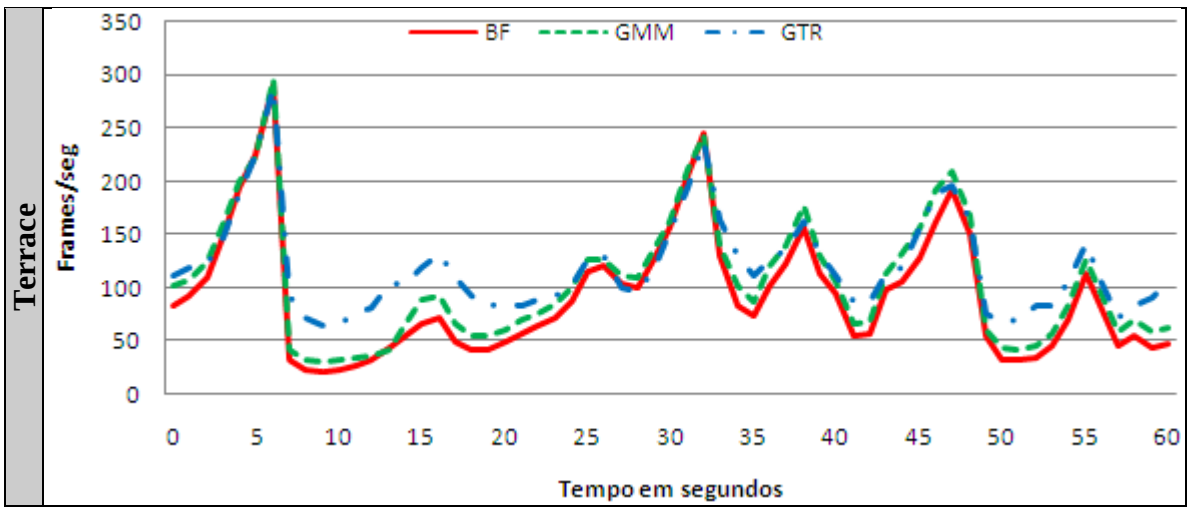
6.5.11. Algoritmo

O objectivo deste teste foi determinar o algoritmo com melhor desempenho em cada um dos terrenos (ver **6.2**). Como podemos concluir a partir da **Figura 6-14** e da **Tabela 6-20**, nos terrenos de menor dimensão como o Valley, por exemplo com um tamanho de 257×257, os ganhos obtidos pela utilização de algoritmos que controlam o nível de detalhe como o Geomipmapping e o GPU Terrain Rendering não são tão significativos como nos terrenos de maior dimensão como o Terrace com um tamanho de 1025×1025 ou o Highland com um tamanho de 2049×2049. Esperava-se um comportamento deste género uma vez que à medida que aumenta o tamanho do terreno, aumenta também o número de triângulos que se tem de processar o que tem como consequência um aumento da carga. Nos terrenos de maior dimensão, o algoritmo de Geomipmapping só foi mais rápido na máquina N2, em todas as outras máquinas na maior parte dos casos obteve-se sempre um melhor desempenho com o algoritmo de GPU Terrain Rendering acentuando-se esse facto à medida que o tamanho do terreno aumenta tal como se pode verificar no gráfico.

Algorithm	BF	GMM	GTR







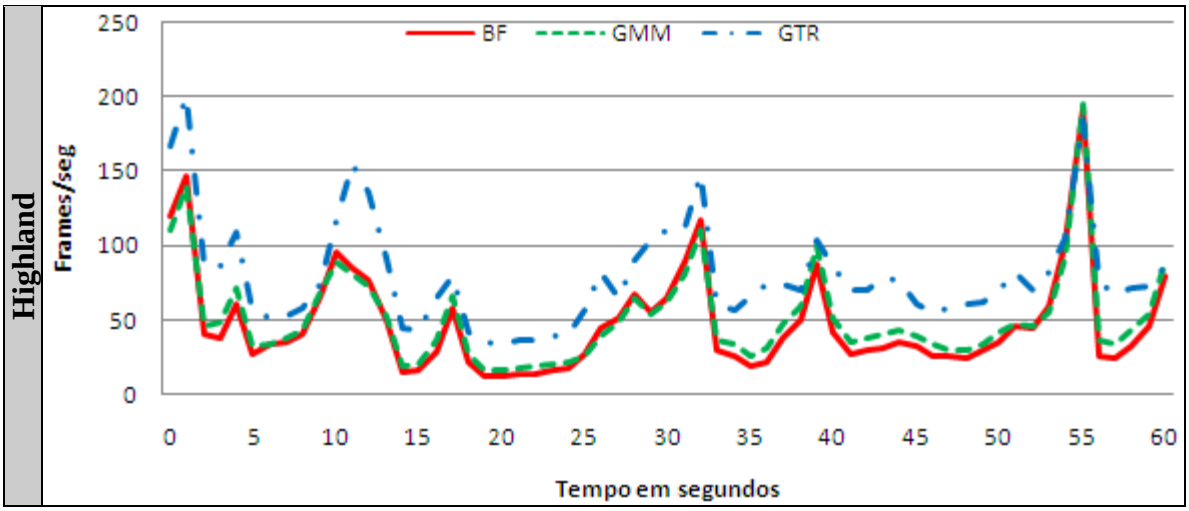


Figura 6-14: Tamanho do terreno em cada um dos algoritmos na máquina N1.

		Média de frames/seg							
		N1		N2		D1		X1	
Valley	**BF**	200		294		1310		626	
	GMM	197		291		1302		569	
	GTR	194		287		1277		574	
Desert	**BF**	137		217		1020		258	
	GMM	156		267		1206		233	
	GTR	165		250		1170		389	
Terrace	**BF**	90		169		941		126	
	GMM	103		251		1133		114	
	GTR	117		208		1227		224	
Highland	**BF**	50		120		379		33	
	GMM	53		195		409		30	
	GTR	82		173		987		127	

Tabela 6-20: M/FPS por máquina/terreno/algoritmo.

6.6. Análise

Analisando de uma forma global os resultados de cada um dos testes realizados em **6.5**, sobretudo aqueles em que se obtiveram os resultados mais interessantes do ponto de vista do desempenho é possível chegar a uma série de conclusões. Antes de se iniciar essa análise é de referir que na máquina X1 os resultados foram na maioria dos casos diferentes do esperado, pelo que se analisa essa máquina em particular em **6.6.1** não abrangendo a discussão que se segue os resultados aí obtidos.

Em relação aos testes propriamente ditos, pode-se começar por destacar o impacto que o número de vértices do bloco tem ao nível do desempenho (ver **6.5.1**). De facto, nos

algoritmos pertencentes à classe *Tiled Blocks* (ver **4.1**) este é normalmente um factor determinante tendo-se observado grandes oscilações no número de *frames* por segundo para cada um dos valores testados. Verificou-se também que o número de vértices do bloco para o qual se obtêm os melhores desempenhos é superior no Geomipmapping ao do GPU Terrain Rendering. Efectivamente os melhores resultados foram obtidos com um número de vértices que oscila entre os 17×17 para o GPU Terrain Rendering e 33×33 para o Geomipmapping, reflectindo a diferença no princípio utilizado no tratamento do nível de detalhe.

No que diz respeito ao *culling* (ver **3.3**) conclui-se que o *frustum culling* (ver **3.3.2**) foi em todos os testes realizados (ver **6.5.2**) crucial para a obtenção de um bom desempenho. Em relação ao algoritmo de oclusão este teve impacto sobretudo nas máquinas mais lentas e nos terrenos de maior dimensão, constituindo efectivamente uma mais-valia nesses casos. A melhoria no desempenho nota-se mais no Brute Force, pois aí a eliminação de um bloco oculto tem um impacto muito maior do que nos algoritmos que reduzem o nível de detalhe. Há, no entanto, que realçar que o algoritmo oclusão faz sentido sobretudo quando o ponto de vista é baixo e/ou o terreno tem um conjunto de elevações que causam a oclusão, uma vez que nestes casos haverá um maior número de blocos eliminados pelo algoritmo.

Ao nível dos dados de elevação e da forma como são tratados, outra das conclusões interessantes foi o baixo desempenho das *vertex textures* (ver **5.5.2.3**) nos algoritmos implementados. A avaliar pelos testes executados (ver **6.5.3**), são mais lentas do que a técnica de *multi-stream* (ver **5.5.2.2**) pelo que, se o desempenho pesar mais do que alguma da flexibilidade que permitem, ao nível da manipulação dos valores de elevação do terreno, esta não deve ser a técnica adoptada. Estes resultados eram esperados e são consistentes com os obtidos por outros autores em algoritmos que utilizam *vertex textures* (ver [10] e [207], por exemplo).

No que diz respeito ao nível de detalhe do horizonte (ver **6.5.4**), verificou-se que com valores de erro entre 4 e 8 *pixels* se obtêm a melhor relação qualidade/desempenho.

Para o nível de detalhe do terreno (ver **6.5.5**), verificou-se que à medida que se aumenta o erro máximo em *pixels* e consequentemente se diminui o nível de detalhe, o desempenho aumenta, mas em contrapartida a qualidade do terreno diminui. O mais relevante foi que a degradação causada por valores de erro mais elevados afectou sobretudo o algoritmo de Geomipmapping, isto é, para o mesmo valor de erro a qualidade de imagem é superior no GPU Terrain Rendering. Isto poderá significar que o nível de detalhe no GPU Terrain Rendering diminui mais lentamente que no Geomipmapping ou pelo menos essa diminuição tem menos impacto a nível visual.

Em relação ao tratamento das falhas a utilização de *skirts* (ver **5.5.1.1**) é a nível de desempenho a melhor alternativa na maior parte das máquinas testadas, mesmo no Geomipmapping onde se considerou também a técnica de actualização de índices (ver **5.6.2.1**). Especialmente relevante foi a pequena diferença observada nos resultados obtidos (ver **6.5.6**) com *skirts* e sem nenhuma técnica de tratamento de falhas o que mostra o impacto reduzido que esta abordagem tem no desempenho. Em relação às optimizações implementadas, observou-se uma melhoria pequena ao nível do desempenho que se manifestou sobretudo no algoritmo de GPU Terrain Rendering.

Relativamente à utilização do algoritmo de Geomorphing para tratamento do efeito de *popping*, verifica-se (ver **6.5.7**) que o seu impacto a nível de desempenho não é muito

significativo nos algoritmos estudados, em comparação com o ganho ao nível do realismo que proporciona..

Finalmente, no teste efectuado aos algoritmos (ver **6.5.11**) sobretudo nos terrenos de maior dimensão o GPU Terrain Rendering foi na maior parte das máquinas o algoritmo mais rápido, com excepção da máquina N2. No entanto, mesmo nessa máquina a diferença entre os dois métodos não foi muito grande, pelo que de um modo geral o GPU Terrain Rendering é o método com melhores resultados.

Abordando agora os resultados obtidos a nível geral, notou-se que nos testes que têm um efeito directo na quantidade de geometria enviada para *rendering* as máquinas mais rápidas, em comparação com as máquinas mais lentas, obtêm melhores resultados para valores que implicam o envio de mais geometria. É o caso do número de vértices do bloco e do nível de detalhe do horizonte na máquina D1. Aqui, o melhor desempenho foi obtido com valores que implicam envio de mais geometria pelo que se está dependente da rapidez de processamento da placa gráfica.

Os testes realizados permitiram também avaliar de um modo geral a importância das técnicas de gestão de nível de detalhe e de oclusão e sobretudo a aferir a sua relevância em placas gráficas cada vez mais rápidas e com maior capacidade de processamento. Face aos resultados obtidos, conclui-se que, para terrenos de menor dimensão como o Valley (ver **6.2**) com um tamanho de 257×257, utilizar algoritmos de nível de detalhe não é determinante já que a quantidade de geometria envolvida não tem um impacto muito profundo no desempenho. Por outro lado, para terrenos de maior dimensão a verdade é que a capacidade de processamento das placas gráficas continua a não ser suficiente, pelo que as técnicas de nível de detalhe se justificam plenamente, bem como as de oclusão sobretudo para terrenos montanhosos.

6.6.1. Resultados na XBOX 360

Como já foi referido, os resultados obtidos nesta máquina são algo contraditórios em comparação com os resultados dos mesmos testes nas outras máquinas. A hipótese mais provável é o ponto de estrangulamento ser o CPU, pois todos os testes parecem evidenciar que os dados não estão a ser fornecidos à placa gráfica tão rápido quanto o desejável. Basta considerar o tamanho do bloco mais rápido para esta máquina, 129×129. Este tamanho de bloco implica menos pesquisas na *quadtree*. Por outro lado, é uma quantidade de geometria demasiado grande em comparação com tamanho do bloco mais rápido na máquina D1, 33×33. O mesmo se passa com o *culling* pois em todos os casos, o *frustum culling* sem o algoritmo de oclusão foi a solução mais eficiente, e esta é, mais uma vez, a que implica menos trabalho ao nível do CPU. Mesmo no tratamento de dados de elevação, as *vertex textures* foram na máquina X1 mais rápidas do que a aproximação que implica a utilização de múltiplas *streams*, mais uma vez um comportamento diferente. Muito embora não existam muitas informações sobre a placa gráfica da XBOX 360, tudo aponta para uma placa extremamente rápida e com grandes capacidades. O próprio CPU constituído por 3 *cores* não parece ser o problema. Então, como se justificam estas discrepâncias? Tudo leva a crer que o problema esteja na .NET Compact Framework a versão da .NET Framework utilizada na XBOX 360. Essa versão da Framework de .NET é muito mais limitada do que a versão normal para *desktops*, a avaliar por algumas das poucas referências existentes [1][2]. Assim, muito embora o código tenha sido de certo

modo optimizado, não teve como principal objectivo colmatar as limitações da .NET Compact Framework na XBOX 360 pelo que se encontrou alguns constrangimentos a esse nível no que diz respeito ao desempenho. Um dos principais indicadores que não está reflectido nos testes foi o tempo de inicialização de terrenos como o Highland na XBOX 360 vs o tempo de inicialização numa máquina como a N1. Nesse caso, são necessários vários minutos para que a *quadtree* (ver **3.2.3.1**) se inicialize por completo na máquina X1 ao contrário da máquina N1 onde essa inicialização não demora mais do que meio minuto. Este foi logo à partida um factor indicador de contenção ao nível da .NET Compact Framework, o que pode indicar necessidades específicas de optimização que não foram abordadas directamente nesta dissertação mas que teriam de envolver necessariamente a utilização de conceitos de *multi-threading* para se tirar partidos dos múltiplos processadores disponíveis na XBOX 360.

6.7. Sumário

Neste capítulo descrevem-se em detalhe os testes efectuados para se avaliar o desempenho dos três métodos de geração de terrenos em tempo real concretizados, respectivamente o Geomipmapping (ver **4.4**), o GPU Terrain Rendering (ver **4.8**) e o Brute Force. Cada um dos testes foca um parâmetro de configuração (ver **6.3**), tendo-se avaliado individualmente o desempenho e/ou a qualidade da imagem de acordo com o tipo de teste. Para melhor se avaliar os algoritmos desenvolvidos, utilizou-se um conjunto de terrenos (ver **6.2**) com dimensões diferentes (257×257, 513×513, 1025×1025 e 2049×2049) e também máquinas diferentes (dois *notebooks*, um *desktop* e uma consola). Foi ainda adoptada uma metodologia de testes (ver **6.4**) que estabelece o modo como cada um dos testes foi executado para que os resultados obtidos possam ser considerados relevantes. Cada um dos testes é descrito em detalhe em **6.5**, tendo-se concretizado os seguintes:

1. Número de Vértices do Bloco (ver **6.5.1**).
2. *Culling* (ver **6.5.2**).
3. Envio de Dados de Elevação (ver **6.5.3**).
4. Nível de Detalhe do Horizonte (ver **6.5.4**).
5. Nível de Detalhe do Terreno (ver **6.5.5**).
6. Coerência Espacial (ver **6.5.6**).
7. Coerência Temporal (ver **6.5.7**).
8. Primitiva (ver **6.5.8**).
9. *Vertex Cache* (ver **6.5.9**).
10. *Fill rate* (ver **6.5.10**).
11. Algoritmo (ver **6.5.11**).

A partir deste conjunto de testes foi possível determinar quais os valores mais adaptados para cada um dos itens de configuração considerados em função do nível de desempenho. Destaca-se a influência da dimensão do bloco de terreno, o impacto dos diferentes métodos de *culling*, nomeadamente do algoritmo de oclusão utilizado, os resultados obtidos para cada um dos métodos de coerência espacial, o impacto causado pelo envio dos valores de elevação por intermédio de *vertex textures* e o teste aos algoritmos propriamente ditos. A análise dos resultados obtidos é feita em **6.6** tendo-se

concluído que o algoritmo de GPU Terrain Rendering é o que apresenta na maior parte dos casos a melhor relação qualidade/desempenho e também que as optimizações concretizadas ao nível das *skirts* enquanto método de coerência espacial tornaram esse método ligeiramente mais eficiente. Por outro lado, verificou-se que as *vertex textures*, não obstante a flexibilidade que oferecem, não se equiparam ainda em termos de desempenho aos métodos mais tradicionais. Por fim, no que diz respeito ao algoritmo de oclusão, verificou-se que em situações de grande carga, mais especificamente em terrenos de maior dimensão este tem um efeito determinante ao nível do desempenho.

A análise dos resultados obtidos para a XBOX 360 (ver **6.6.1**) é feita separadamente neste capitulo uma vez que estes entram, em alguns casos, em contradição com todas as outras maquinas tendo-se concluído que as diferenças se devem ao .NET Compact Framework e ao facto de este ser bastante mais lento que a *framework* disponível nos *desktops* e nos *notebooks*.

7. Conclusão

Nesta dissertação, compararam-se dois algoritmos de geração de terrenos em tempo real pertencentes à classe *Tiled Blocks* [120], respectivamente o Geomipmapping [39] e o GPU Terrain Rendering [207], e aprofundaram-se conceitos directa ou indirectamente, relacionados com estes dois algoritmos. Destes, destaca-se sobretudo o *culling*, nomeadamente o *occlusion culling* com a implementação da técnica de oclusão do algoritmo de Terrain Occlusion Culling With Horizons [63] e a utilização das *vertex textures* para enviar os valores de elevação para a placa gráfica.

Ao nível do trabalho realizado, contribui-se sobretudo com a descrição de uma implementação na qual se define uma estrutura base que abstrai os conceitos inerentes a algoritmos da classe *Tiled Blocks*, integrando de uma forma simples os algoritmos implementados. Ainda nesta perspectiva, realça-se também a integração nessa estrutura de um algoritmo de *occlusion culling* e de duas formas de enviar os valores de elevação, nomeadamente a utilização de *vertex textures* para esse efeito. Por fim, também merecem destaque os métodos utilizados para garantir a coerência espacial e temporal, nomeadamente a utilização das *skirts* para corrigir as falhas e do Geomorphing para corrigir o efeito de *popping*. A abordagem concretizada permitiu assim obter uma visão sobre a geração de terrenos em tempo real na plataforma XNA, na qual este trabalho foi implementado, conseguindo-se obter um conjunto de conclusões interessantes ao nível do desempenho.

Uma das conclusões a que se pretendia chegar estava relacionada com a aplicabilidade destes algoritmos face à grande evolução das placas gráficas. O que se verificou foi que para terrenos de menor dimensão, 257×257, por exemplo, a utilização de técnicas de nível de detalhe não foi determinante. O mesmo se verificou, muito embora, não de uma forma tão significativa, com a técnica de oclusão empregue. Também esta só começa a ter maior impacto ao nível do desempenho em terrenos de maior dimensão e sobretudo em máquinas mais lentas. Isto significa que nesses terrenos é perfeitamente possível obter resultados bastante bons apenas com *view frustum culling*, sem se recorrer a técnicas de *occlusion culling* ou de nível de detalhe.

No que diz respeito às *vertex textures*, os resultados obtidos nas máquinas consideradas em cada um dos testes efectuados, levam a concluir que não são ainda a melhor alternativa em algoritmos da classe *Tiled Blocks*, a atender pelos resultados obtidos nas máquinas consideradas em cada um dos testes efectuados.

Em relação à comparação efectuada ao nível dos algoritmos de nível de detalhe, o objectivo era perceber qual a estratégia mais eficiente sobretudo aquela que garantia os padrões de qualidade necessários para se conseguir um representação credível do terreno em tempo real. Os dois algoritmos implementados, o Geomipmapping e o GPU Terrain Rendering utilizam duas abordagens distintas. No Geomipmapping a dimensão dos blocos enviados para *rendering* mantém-se e o número de vértices do bloco varia com o nível de detalhe. No GPU Terrain Rendering a dimensão dos blocos varia com o nível de detalhe e o número de vértices mantêm-se. Efectivamente, o Geomipmapping controla o nível de detalhe via uma variação ao nível da lista de índices de cada um dos blocos e o GPU Terrain Rendering pela dimensão dos blocos que envia para *rendering*, ou seja, pela área que ocupam, e consequentemente actua mais ao nível da redução do número de

chamadas efectuadas à primitiva gráfica. O que se verificou foi que na maior parte dos casos o GPU Terrain Rendering foi a melhor opção não só ao nível do desempenho como a nível de qualidade da imagem pelo que a estratégia que adopta para gerir o nível de detalhe é a recomendada.

Outro factor importante deste trabalho foi a opção tomada em relação à plataforma de desenvolvimento, o XNA. A característica mais relevante neste aspecto foi esta assentar sobre a tecnologia .NET que utiliza uma *virtual machine* que faz a gestão de toda a memória.

Ao nível de desempenho, a excepção foi a XBOX 360, mas esses resultados já eram de alguma forma esperados dadas as limitações já conhecidas da versão da *framework* de .NET, a Compact Framework, utilizada nessa consola, que se tornaram evidentes nos testes efectuados.

De um modo geral tem vindo a tornar-se cada vez mais fácil passar a maior parte do processamento para a placa gráfica, em grande parte fruto das grandes evoluções ao nível da componente programável da mesma. Esta tendência tem-se vindo a intensificar sobretudo na área da computação gráfica. Em contrapartida, a importância do CPU tem vindo a diminuir consideravelmente. Consequentemente, abordagens baseadas em *managed code* começam a ser relevantes nesta área até porque cada vez mais o código mais crítico ao nível de desempenho pode ser executado na placa gráfica. Este facto, aliado à rapidez de desenvolvimento que plataformas deste género proporcionam, promete tornar esta opção cada vez mais viável, até mesmo numa área tão exigente como é a da geração de terrenos em tempo real.

7.1. Trabalho Futuro

Sendo a geração de terrenos em tempo real um tópico tão vasto é natural que qualquer trabalho realizado neste domínio deixe sempre muito por dizer e fazer, especialmente com a evolução constante dos GPU que permitem a actualização das técnicas já existentes e o aparecimento de novas abordagens de uma forma tão rápida que nem sempre se consegue acompanhar. Procura-se por isso identificar nesta secção possíveis caminhos a seguir, especialmente aqueles que face ao que foi concretizado parecem ser os mais relevantes.

No que diz respeito à comparação efectuada entre algoritmos da classe *Tiled Blocks*, uma extensão natural seria incluir um ou mais algoritmos da classe *Concentric Regions* discutida em **4.1** nomeadamente o algoritmo de GPU Based Geometry Clipmaps (ver **4.10**) para o qual seria interessante avaliar a qualidade visual e o desempenho face aos outros algoritmos.

Igualmente relevante seria a extensão do trabalho realizado no sentido de se suportar *out-of-core*, isto é terrenos de dimensão arbitrária seguindo-se, por exemplo, o trabalho já efectuado em alguns dos algoritmos descritos em **4**, nomeadamente no Chunked LOD (ver **4.5)**, no Rendering Very Large, Very Detailed Terrains (ver **4.7**) e mesmo no Geometry Clipmaps. Neste último é descrita uma estratégia baseada na utilização de uma pirâmide de *clipmaps* que permite terrenos de dimensão arbitrária e que pode ser aplicada mesmo em algoritmos da classe *Tiled Blocks* tal com o é feito em [23] e em [74], pelo que é um rumo interessante a seguir.

Outra área importante abordada nesta dissertação foi a detecção de porções do terreno ocultas, o *occlusion culling*. Neste domínio concretizou-se o algoritmo de Terrain Occlusion Culling With Horizons (ver **4.6**), no entanto, seria interessante avaliar o impacto de uma abordagem baseada em *occlusion queries* (ver **3.3.4**), uma nova funcionalidade dos GPU que surgiu com o *shader model* 3.0.

Outro caminho a seguir no futuro seria o de tirar partido de múltiplos processadores (*multi-core*) com técnicas de *multi-threading* algo em que não se apostou neste trabalho. Algumas das máquinas utilizadas nos testes efectuados (ver **6**) nomeadamente as maquinas D1 e X1 já disponibilizavam múltiplos processadores e esse caminho é *inclusive* o recomendado para fazer face a alguns dos problemas de desempenho identificados na máquina X1. Nesse sentido seria também um tópico importante a explorar já que pode trazer inúmeras vantagens nomeadamente ao nível do desempenho.

Faz também sentido explorar formas de aumentar o realismo do terreno pois, muito embora a abordagem concretizada tenha já uma qualidade visual razoável nota-se sempre que se aproxima a câmara a falta de detalhe o que resulta do facto de a textura não ter a resolução suficiente. Nesta perspectiva a aplicação de estratégias como o *splatting* [18] com recurso a atlas de texturas [215] tal como é feito em [38] é uma hipótese a considerar. Ainda em relação ao tema do realismo é de realçar a possibilidade de representar outros elementos como massas de água (por exemplo, oceanos, rios e lagos), fenómenos meteorológicos (por exemplo, nuvens, nevoeiro, chuva, neve), objectos celestiais (por exemplo sol, lua, planetas, estrelas) e objectos à superfície do terreno (por exemplo, rochas, árvores, casas). Todos estes objectos contribuiriam para criar uma representação mais realista e são por conseguinte uma vasta área não explorada nesta dissertação.

Finalmente com o aparecimento do *shader model* 4.0 e do novo estágio programável do *pipeline*, o *geometry shader*, passou a ser possível a geração de nova geometria a partir de um conjunto de vértices transformados e da informação sobre a conectividade dos mesmos, o que pode permitir aumentar ou diminuir o detalhe directamente ao nível do GPU. Muito embora ainda não seja o mais indicado ao nível do desempenho [105] pode representar no futuro, especialmente com o aparecimento do *shader model* 5.0 [75][98] e de placas gráficas que o suportem uma alternativa viável sendo por isso um possível rumo a seguir neste trabalho.

Anexo A. Síntese de Terrenos

A sintetização de terrenos permite criar os *height fields* que os representam por meio de um algoritmo. Estes algoritmos dados os mesmos parâmetros produzem sempre o mesmo *height field*, o que significa que podemos armazenar apenas os parâmetros fornecidos para construir o terreno, conseguindo-se assim um ganho de espaço notável. No entanto, em contrapartida temos de garantir que o terreno criado desta forma corresponde a uma representação suficientemente realista, para que possa ser utilizado como se de um terreno real se tratasse. Essa representação não é obtida certamente pela atribuição de um valor aleatório de altura a cada um dos pontos do *height field*. É assim necessário gerar complexidade, mas uma complexidade controlada e não completamente caótica. Uma das ferramentas mais utilizadas para atingir esse objectivo são os fractais, considerados como a forma mais simples de gerar complexidade. Estes são abordados neste anexo com o intuito de realçar a sua importância especialmente como ferramenta de síntese de terrenos. Para isso em **A.1** é explicado o conceito de fractal e em **A.2** os factores envolvidos no processo iterativo que permite a construção do mesmo. Em **A.3** aborda-se a função de *fractional brownian motion* que permite definir uma relação entre os diversos factores envolvidos na construção do fractal. Finalmente são abordados em **A.4** um conjunto de algoritmos que permitem a geração de terrenos por intermédio de fractais que podem ser divididos em dois tipos: métodos procedimentais e métodos não procedimentais. Ao nível dos métodos procedimentais aborda-se em detalhe o método de *noise synthesis* (ver **A.4.5**). No que diz respeito aos métodos não procedimentais descreve-se o Fault Formation (ver **A.4.1**), o Circles (ver **A.4.2**), o Mid Point Displacement (ver **A.4.3**) e o Particle Deposition (ver **A.4.4**).

A.1. O que é um Fractal[7]?

Os fractais foram descobertos no início do século XX, mas como era impossível nessa altura representá-los, foi só em 1977 com a publicação do livro *The Fractal Geometry of Nature* [124] que Benoit Mandelbrot, um investigador da IBM, considerado o pai dos fractais, os revelou ao mundo. É da necessidade de representar esse mundo extremamente complexo que habitamos que surgiram os fractais. De facto, reproduzir fielmente em imagens geradas por computador os objectos que povoam o mundo que habitamos está hoje em dia muito para além das capacidades actuais de processamento. Qualquer objecto, por mais pequeno que seja, tem um conjunto quase infinito de detalhes que não podem ser reproduzidos na totalidade. É verdade que podemos produzir aproximações muito realistas desses objectos, muito provavelmente suficientemente detalhadas para convencerem o observador mais exigente, no entanto, a realidade é que estas estão muito aquém do objecto que pretendem representar. Neste contexto, a palavra-chave é complexidade, porque é efectivamente o detalhe que compõe a cena, o que a torna tão rica em termos de informação visual. A geometria fractal é a primeira linguagem de complexidade visual que nos oferece um potente vocabulário de construção de formas

[7] Mandelbrot derivou o termo fractal do latim *fractus* que significa fragmentado ou irregular. *Frangere* o verbo latim correspondente significa partir ou criar fragmentos irregulares.

complexas, particularmente das formas existentes na natureza. Para além disso, a forma como os fractais encapsulam a complexidade está especialmente adaptada às capacidades dos computadores, que podem traduzir as abstracções matemáticas que os representam na forma mais adaptada para a cognição humana: as imagens [52]. Através da utilização de interfaces gráficas, um utilizador menos familiarizado com os conceitos matemáticos pode gerar fractais visualizando o resultado da selecção de diferentes valores dos parâmetros que os definem. A base matemática por detrás do conceito de fractal que pode ser consultada em detalhe em [155]. Segundo [52], um fractal é um objecto geometricamente complexo, cuja complexidade advêm da repetição de uma determinada forma num intervalo de escalas. A repetição de uma forma num intervalo de escalas é denominada de *self similarity*, ou seja, um fractal é muito parecido consigo mesmo numa variedade de escalas. A **Figura A-1** ilustra o conceito de *self similarity* num feto. Este, como se pode verificar, é muito semelhante a si mesmo em cada uma das escalas em que é representado. Esta figura, para além de ser um óptimo exemplo da natureza fractal de objectos, indica-nos que o intervalo de escalas em que um objecto é *self similar* é limitado, limite que no exemplo do feto está situado na 3ª escala. Assim, diz-se que existe um tamanho superior e inferior a partir do qual a propriedade de *self similarity* deixa de se manifestar. Estes são designados, respectivamente, de *upper crossover scale* e de *lower crossover scale*.

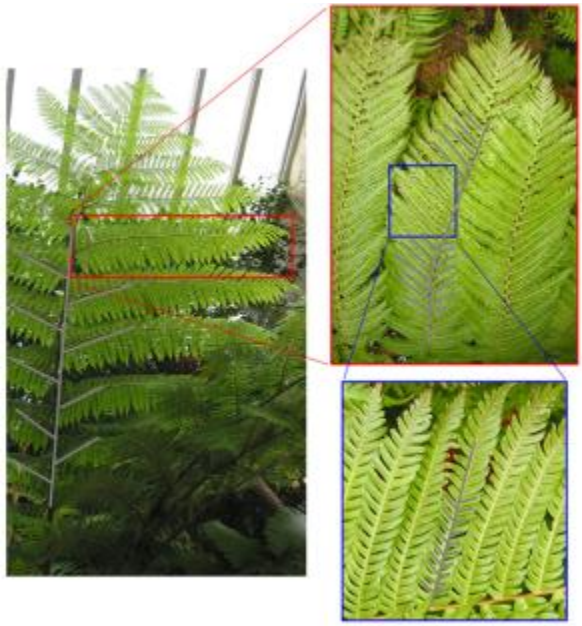


Figura A-1: Ilustração do conceito de *self similarity* [21].

Outra forma de definir um fractal é usar como base uma nova forma de simetria: *dilation symmetry*, que caracteriza os objectos que não se modificam independentemente do factor de escala (ampliação ou redução) aplicado sobre eles, ou seja, independentemente da escala não existem alterações relevantes na forma. Por exemplo, as redes de rios são fractais na medida em que os pequenos rios e as suas redes são muito semelhantes às redes de rios maiores que os incluem. Finalmente, para completar a definição de fractal, é necessário referir ainda uma propriedade muito importante que os caracteriza: a dimensão

fractal. No caso da dimensão euclidiana, esta é facilmente compreendida, zero dimensões correspondem a um ponto, uma a uma linha, duas a um plano e três a um espaço. A dimensão fractal estende este conceito adicionando a possibilidade de a dimensão ser representada por um valor real, por exemplo, 2.3. O 2 representa a dimensão euclidiana subjacente e o 3 é designado de incremento fractal. À medida que o incremento fractal aumenta, o fractal passa literalmente de uma dimensão euclidiana, por exemplo de um plano para uma nova dimensão, neste caso um espaço a três dimensões. A forma mais fácil de abordar esta propriedade, que é de difícil compreensão, é encará-la como uma medida da complexidade visual do fractal resultante.

Estabelecida a fundação teórica, se analisarmos agora mais alguns dos fenómenos existentes na natureza, verificamos que muitos são fractais. Especialmente relevantes neste contexto, as montanhas são talvez o exemplo mais conhecido. Nestas, uma parte mais pequena é muito parecida com uma parte maior. As rochas, outro exemplo, parecem-se muito umas com as outras independentemente do tamanho, pelo que, nos livros de geologia é costume colocar um martelo ou uma régua nas fotografias de formações rochosas. Isto resulta do facto de estas não terem uma escala definida, não se conseguindo por isso distinguir o quão grande é uma formação rochosa se não for especificado um contexto. É de referir que nestes exemplos as formas não são exactamente iguais nas diferentes escalas, mas aproximadamente iguais. É nesse sentido que se faz a distinção entre *statistical self similarity* e *exact self similarity*. Quase todos os fractais na natureza exibem *statistical self similarity*, ou seja, apenas as estatísticas de uma geometria aleatória são similares em diferentes escalas. No caso da *exact self similarity*, os componentes mais pequenos são exactamente iguais aos de maior dimensão. Um exemplo clássico de *exact self similarity* é o famoso floco de neve de von Koch [101] representado na **Figura A-2**.

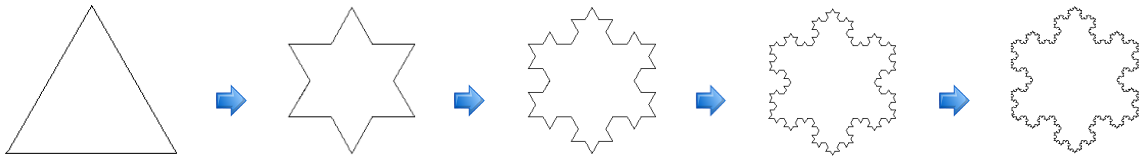
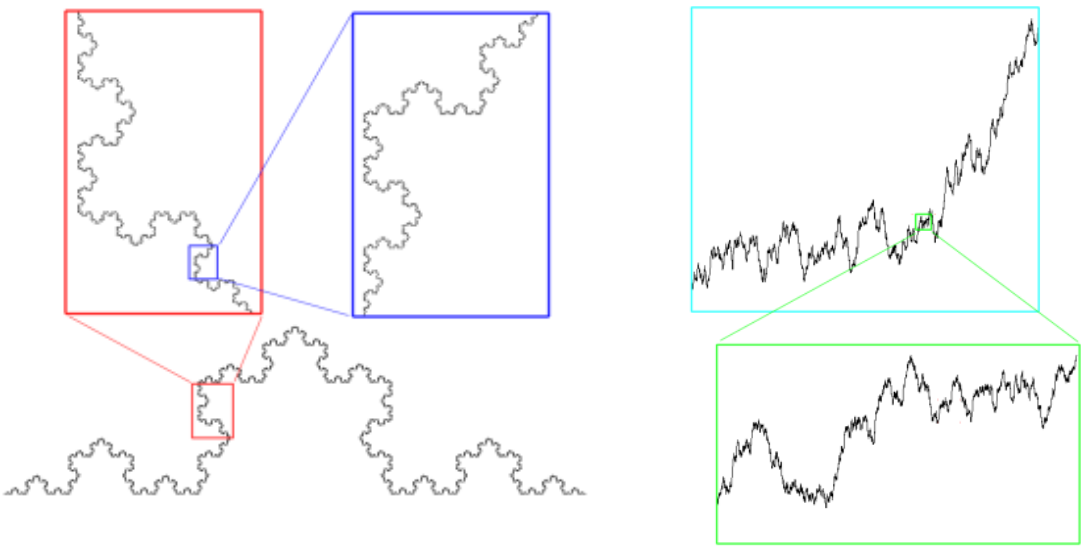


Figura A-2: O floco de neve de von Koch [52].

A comparação entre as duas formas de *self similarity* é feita na **Figura A-3**. Note-se que no caso da *exact self similarity* representada à esquerda da figura, independentemente da ampliação aplicada, a forma obtida é exactamente a mesma. No caso da *statistical self similarity* representada à direita, essa semelhança pode não ser tão óbvia, mas existem de facto medidas estatísticas ou numéricas que são preservadas entre escalas.



exact self similarity *statistical self similarity*

Figura A-3: Comparação entre *exact self similarity* e *statistical self similarity* [21].

Aos fractais que exibem *statistical self similarity* atribui-se também a designação de *random fractals*, definidos dessa forma como fractais que incorporam variações aleatórias na sua construção [52]. Estas variações não são aleatórias no sentido exacto da palavra uma vez que os computadores são deterministas por natureza. Dependem pelo contrário dos parâmetros que lhes são fornecidos que determinam o resultado obtido.

A.2. Construção do Fractal

A construção de um fractal é de uma maneira geral bastante simples e traduz-se normalmente num processo iterativo que envolve quatro factores [52]:

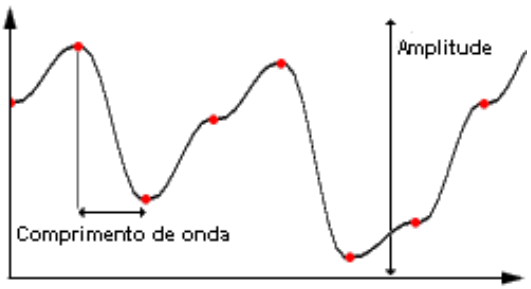
- A função base.
- O parâmetro de dimensão fractal.
- A lacunaridade.
- O número de oitavas.

A origem da complexidade num fractal está simplesmente na repetição de uma forma num intervalo de escalas. Esta forma é obtida por intermédio de uma função base que e é uma das escolhas mais importantes a efectuar, pois determina o aspecto e a qualidade visual do fractal resultante. Estas funções, por convenção, devem retornar valores entre -1.0 e 1.0 e não devem ser demasiado complexas, pelo que existe um conjunto delas que pelas suas características são frequentemente utilizadas, nomeadamente o Perlin Noise (ver **A.4.5.1**), as células de *Voronoi* [217] e a *sparse convolution* [109]. O parâmetro de dimensão fractal, também designado em muitos textos de expoente de Hurst ou simplesmente de H, permite especificar a complexidade do fractal podendo ser relacionado com a dimensão fractal por intermédio da **Equação A-1.**

$D_f = D_E + 1 - H$	D_E: Dimensão euclidiana.
	H: Parâmetro de dimensão fractal em que $H \in [0,1]$.

Equação A-1: Cálculo da dimensão fractal D_f.

Este parâmetro é utilizado para modular a amplitude ou tamanho vertical da função base em cada uma das iterações. O terceiro factor, a lacunaridade[8], é a quantidade pela qual modificamos a frequência ou tamanho horizontal da função base e pode ser encarado como um intervalo (*gap*) entre frequências, tendo normalmente um valor de dois. Na música sempre que se dobra a frequência diz-se que se incrementa uma oitava. É nesse sentido que se fala do número de oitavas, o último factor a considerar, que corresponde ao número de iterações efectuadas, ou seja o número de vezes que se repete a forma geométrica da função base em diferentes escalas. A dimensão fractal propriamente dita (ver **A.1**) vai resultar em termos práticos da modulação da amplitude e da frequência do traço geométrico da função base a cada iteração, conceito que é ilustrado na **Figura A-4**.



Comprimento de onda: Distância entre duas variações.

Amplitude: Diferença entre os valores máximos e mínimos da função

Frequência: Definida como $\frac{1}{ComprimentoOnda}$

Figura A-4: Amplitude e frequência numa função base [55].

Resumindo, a construção do fractal envolve a cada iteração (oitava) a repetição de uma função base, cuja amplitude é modificada pelo valor do parâmetro de dimensão fractal e a frequência pelo valor da lacunaridade, existindo uma relação bem definida entre as duas. É este relacionamento que, como já foi referido, determina a dimensão fractal do resultado. Em **A.3** é definida uma função, que estabelece essa relação chamada de *fractional brownian motion*.

A.3. Fractional Brownian Motion

Os terrenos fractais derivam na sua grande maioria da função matemática de *fractional brownian motion* (fBm) uma generalização de *browning motion* que se define como o movimento aleatório de partículas suspensas num fluido[9]. O *fractional browning motion* é também designado de *random walk process* e consiste basicamente num conjunto de passos numa direcção aleatória com um determinado tamanho, dai a

[8] Lacunaridade vem do latim "lacuna" que significa buracos ou intervalos e está relacionada neste caso com o intervalo entre frequências.

[9] Por volta de 1827 Robert Brown (daí o nome de *browning motion*) observou ao microscópio partículas de pólen na água num aparente movimento aleatório. Este "fenómeno" só foi explicado em 1905 por Albert Einstein [54] como o efeito causado pelas moléculas de água a atingirem o pólen.

designação de *random walk* [210]. A principal característica desta função é o facto de uma ampliação em qualquer parte dar origem a um *random walk* semelhante na parte ampliada. Basicamente, à medida que se progride na função, o valor obtido aumenta ou diminui em uma determinada quantidade que está dependente da escala utilizada. Dito de outra forma, é uma linha que sobe ou desce de acordo com um valor aleatório e que quando vista noutra escala revela padrões semelhantes aos obtidos numa escala menor. Utilizando uma definição mais precisa [139] é uma função não diferenciada do tempo, com um espectro (i.e. o gráfico da amplitude do sinal pela frequência) característico que corresponde à função $1/f^{\beta}$ em que f é a frequência, β o expoente espectral e em que $\beta \epsilon [1,3]$. Para mais detalhes, *inclusive* uma descrição matemática mais precisa consultar [155], onde é feita uma excelente descrição desta função. Resumindo, *fractional brownian motion* é uma função fractal caracterizada pelo seu comportamento estatístico que funciona como um modelo matemático do movimento aleatório [139]. A sua característica mais importante é a uniformidade: está concebida para parecer o mesmo em todos os lugares e em todas as direcções. Na prática, esta função que parece extremamente complexa é na verdade muito simples de implementar (ver em **A.4.5.2** a implementação para *noise synthesis*) podendo ser definida em função dos quatro factores que caracterizam um fractal enunciados em **A.2** (a função base, o parâmetro de dimensão fractal, a lacunaridade e o número de oitavas) e interpretada como o somatório de um número discreto de frequências de uma função base no qual se aplica a cada uma das frequências um factor de escala à amplitude associada [139]. Por outro lado, é importante ainda referir que o expoente espectral β da função $1/f^{\beta}$ que caracteriza o espectro da função de *fractional brownian motion* pode ser relacionado com a dimensão fractal D_f, a dimensão euclideana, D_E e o parâmetro de dimensão fractal, H**,** por intermédio da **Equação A-2**.

$D_f = D_E + 1 - H = D_E + \frac{3-\beta}{2}$	D_E: Dimensão euclidiana.
	H: Parâmetro de dimensão fractal em que $H \in [0,1]$.
	β: Expoente espectral em que $\beta \in [1,3]$.

Equação A-2: Relação entre os componentes do fractal e o espectro.

A ideia de utilizar a *fractional brownian motion* na síntese de terrenos fractais surgiu quando Mandelbrot trabalhava com esta função a uma dimensão. Foi então que notou que o traço geométrico da função com uma dimensão fractal de 1.2 (ver **Figura A-5**) era muito parecido com montanhas, pelo que chegou a conclusão que se esta função fosse estendida a duas dimensões seria possível obter terrenos. E de facto era. Esta é, tal como foi referido em **A.1**, a perspectiva dominante nos fractais ou seja não existe uma razão exacta para muitas das decisões, resultando algumas de puras observações visuais como neste caso.

Figura A-5: Traço geométrico de uma fBm com dimensão fractal de 1.2 [52].

Esta função serve hoje em dia de base a muitos dos métodos de criação de terrenos sendo uma das melhores formas de criar terrenos realistas e paralelamente um dos melhores métodos fractais de geração de terrenos.

A.4. Algoritmos de Síntese de Terrenos

Os algoritmos de síntese de terrenos, e de uma forma mais genérica de síntese de texturas, foram desde sempre uma área de estudo bastante activa no campo da computação gráfica, existindo uma serie de aproximações ao problema todas com um conjunto de vantagens e de desvantagens. Estes métodos podem ser classificados genericamente de duas formas [139]: métodos procedimentais e não procedimentais. Os métodos procedimentais são avaliados onde e quando são necessários, normalmente em tempo de *rendering*, muito embora possam também ser utilizados numa fase de pré-processamento. Os métodos não procedimentais por seu lado actuam de uma forma global e de uma só vez numa fase de pré-processamento. Os métodos procedimentais, mais especificamente o de *noise synthesis* (ver **A.4.5**) que concretiza a função de *fractional brownian motion* descrita anteriormente (ver **A.3**), são flexíveis, eficientes ao nível da memória (pois apenas partes visíveis necessitam de ser calculadas) e de uma forma geral simples dada a implementação da função base. Por outro lado os métodos não procedimentais como o Fault Formation (ver **A.4.1**), o Circles (ver **A.4.2**), o Mid Point Displacement (ver **A.4.3**) e o Particle Deposition (ver **A.4.4**) são normalmente mais rápidos e simples de implementar, mas têm de ser executados numa fase de pré-processamento. Dito de outra forma, todos os métodos não procedimentais geram o terreno de uma só vez através de processos iterativos, não sendo os mais adaptados para uma utilização em tempo real. Os métodos procedimentais geram os valores a pedido, sendo avaliados a cada ponto e não estão desta forma associados a um processo iterativo fechado. Na verdade grande parte dos métodos não procedimentais, especialmente os descritos em **A.4.1**, **A.4.2**, **A.4.3** e **A.4.4** são de certa forma aproximações *ad hoc* para gerar fractais, em contrapartida, os métodos procedimentais especialmente os que se baseiam na função de *fractional brownian motion* tem uma base matemática rigorosa e propriedades que os tornam bastante simples para trabalhar [104].

A.4.1. Fault Formation

O algoritmo de Fault Formation [101][165][180][61][94] é uma técnica de criação de terrenos utilizada para simular o efeito de tremores de terra ao longo de linhas de "falhas" seleccionadas aleatoriamente. O princípio que lhe está subjacente é muito simples muito embora seja uma técnica lenta, ou seja, não é a mais indicada para processamentos em

tempo real. Baseia-se num processo iterativo em que se começa com um *height field* vazio, com uma altura igual a zero para todos os pontos que o compõem e para o qual, se seleccionam dois pontos de forma aleatória entre os quais se traça uma linha (ver **Figura A-6**). É com base nesta linha que se subdivide o *height field* e que se selecciona um dos lados para o qual se aumenta a altura de todos os pontos que lhe estão associados, ao passo que para o outro se diminui de forma equivalente, isto de acordo com um determinado valor calculado para cada iteração e que deve ser limitado por um máximo e um mínimo fornecidos como parâmetros ao algoritmo. Para diferenciar os lados é construído um vector v_1 tal que $v_1 = (x_2 - x_1, z_2 - z_1)$ em que (x_1, z_1) e (x_2, z_2) correspondem aos pontos aleatórios que definem a linha. O próximo passo é calcular para cada ponto no terreno, (t_x, t_z), outro vector v_t tal que $v_t = (t_x - x_1, t_z - z_1)$. Finalmente, calcula-se o produto cruzado[10] dos dois vectores, $v_r = v_1 \times v_t$ sendo o sinal do componente y utilizado para distinguir o lado em que se vai aumentar ou diminuir a altura. Por exemplo, pode-se estabelecer que a altura aumenta para todos os pontos em que $v_r \cdot y > 0$.

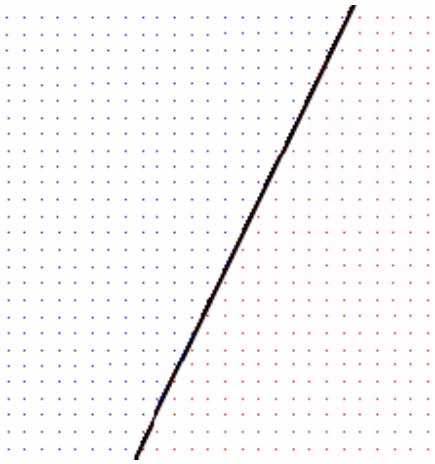


Figura A-6: *Height field* subdividido pela primeira iteração do algoritmo Fault [61].

Na próxima iteração repete-se este processo, seleccionando-se mais dois pontos e traçando-se uma nova linha entre eles, só que desta vez o incremento/decremento da altura em cada um dos lados é menor, decaindo progressivamente à medida que o número de iterações do algoritmo aumenta. Para isso, recorre-se a uma interpolação linear (**Equação A-3**) entre os valores máximos e mínimos de elevação fornecidos, isto numa perspectiva de adicionar detalhe às "falhas" produzidas pelas iterações anteriores, causando-se assim a cada iteração alterações progressivamente menos marcantes no terreno. Como se pode verificar na **Figura A-7**, a partir de uma determinada iteração começa a ser possível obter representações bastantes realistas de terrenos o que é de alguma forma surpreendente dada a simplicidade deste algoritmo.

[10] O produto cruzado entre dois vectores a e b define-se como: $a \times b = (a_y \cdot b_z - b_y \cdot a_z, a_z \cdot b_x - b_z \cdot$ *ax,ax·by−bx·ay*

$y = y_0 + \alpha(y_1 - y_0)$	y_0: O valor máximo de incremento**.**
	y_1: O valor mínimo de incremento.
	α: O rácio da iteração corrente pelo número de iterações.

Equação A-3: Interpolação linear no algoritmo Fault.

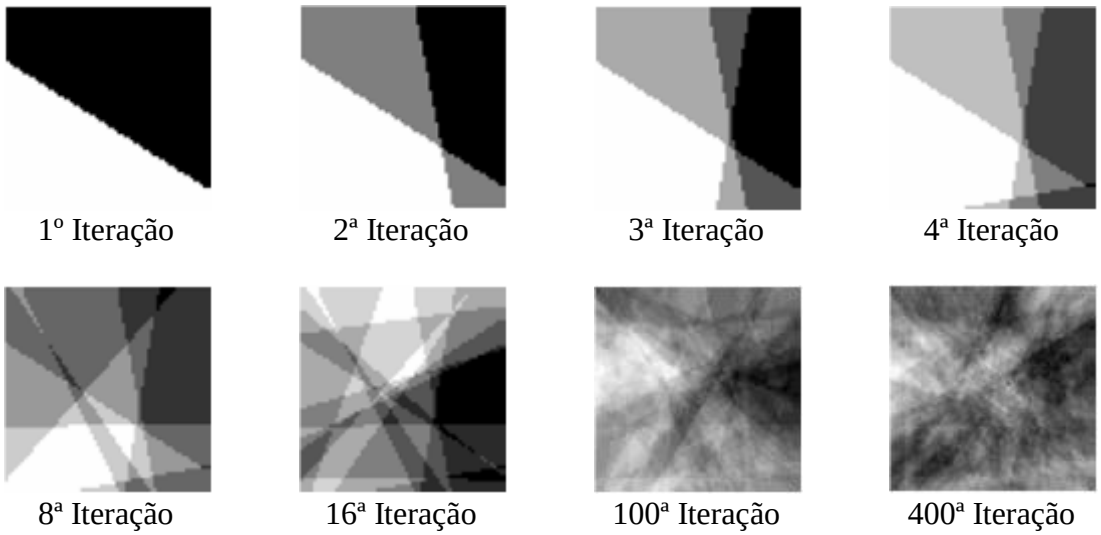


Figura A-7: Sequência de iterações associadas ao algoritmo Fault [61].

Diferentes variações foram propostas para este método [61][180] que permitem obter, em alguns casos, superfícies com características bastante diferentes das do algoritmo original. Todas envolvem uma alteração na forma como são atribuídos os valores para cada um dos lados da linha que divide o *height field*. Este processo no algoritmo *Fault* original [101] corresponde, tal como foi descrito, a um incremento da altura igual em todos os vértices de cada um dos lados e a um decremento equivalente no outro, ou seja, a função de degraus na **Figura A-8**. É precisamente a substituição desta função por outras que é efectuada. Como exemplo podemos considerar a função de seno e a função de co-seno, cujos comportamentos estão descritos na **Figura A-8**. Na **Figura A-9,** por sua vez, estão representados terrenos resultantes da utilização das três funções descritas.

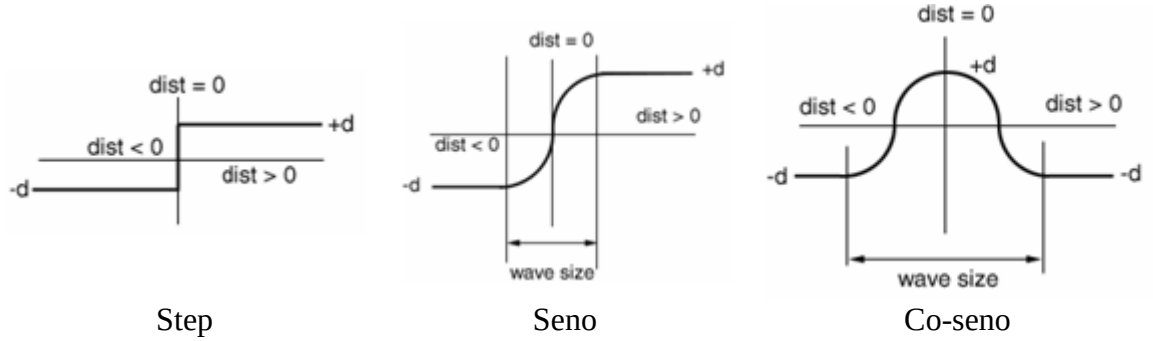
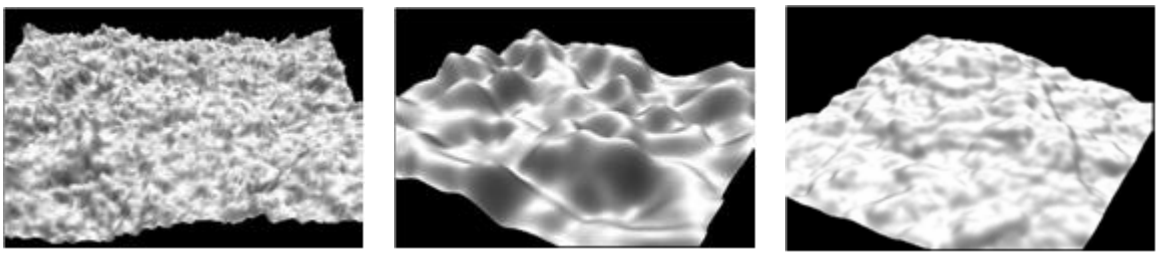


Figura A-8: Diferentes formas de atribuir valores de elevação no Fault [61].



Step Seno Co-seno

Figura A-9: Três variações do algoritmo Fault com 1000 iterações cada uma [61].

A.4.2. Circles

O algoritmo de Circles [61][94], também designado de algoritmo de Hill em [145], é muito semelhante ao algoritmo de Fault Formation (ver **A.4)**, já que também aqui se constrói um *height field* via iterações sucessivas, existindo em cada uma delas um processo de selecção de um conjunto de pontos para os quais se calcula um valor d com o qual se incrementa/decrementa a altura de cada um dos pontos. A diferença reside na forma como os pontos são seleccionados e também no modo como é efectuada a alteração da altura. Ao passo que, no algoritmo de Fault Formation original [101] a altura é incrementada num dos lados da linha aleatoriamente traçada, no *height field* e decrementada no outro (ver **Figura A-6**), neste é utilizado um círculo de centro aleatório para delimitar o conjunto de pontos seleccionados. Para os pontos dentro do círculo, a altura é alterada, ao passo que para os outros a altura corrente é mantida. Também aqui, à semelhança do Fault Formation, é típico reduzir-se progressivamente o valor do incremento a ser feito em cada um dos pontos seleccionados com base na iteração corrente, recorrendo-se para isso a uma interpolação linear (**Equação A-3**) entre o valor máximo de altura e o valor mínimo fornecidos como parâmetro ao algoritmo. No entanto, é de realçar que existem inúmeras hipóteses na forma como é efectuado o cálculo de d e no modo como este é aplicado a cada um dos pontos no terreno, por exemplo, na implementação descrita em [61] para todos os pontos dentro do círculo é aumentada ou diminuída aleatoriamente a altura. O diâmetro do círculo por sua vez também é fornecido como parâmetro, permitindo controlar de um modo geral a aparência das montanhas formadas. Mais especificamente, se o diâmetro for grande em relação ao tamanho do *height field*, as áreas afectadas serão muito maiores e as elevações mais bem definidas, uma vez que tem o contributo de várias iterações. Se, pelo contrário, o diâmetro for pequeno em comparação com o tamanho do *height field*, as elevações serão muito menos pronunciadas. Para calcular o incremento final aplicado a cada ponto dentro do círculo, pode ser utilizada a função de co-seno tal como está descrito no gráfico da **Figura A-10** isto com o intuito de modular o valor do incremento corrente. Mas mais uma vez é de realçar que é perfeitamente possível recorrer a outro género de funções com as quais se obtêm certamente resultados diferentes.

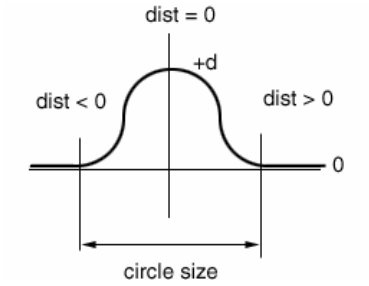


Figura A-10: Função de co-seno utilizada no algoritmo de Circles [61].

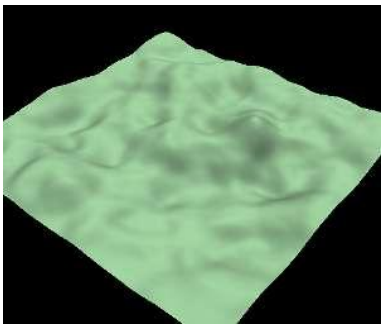
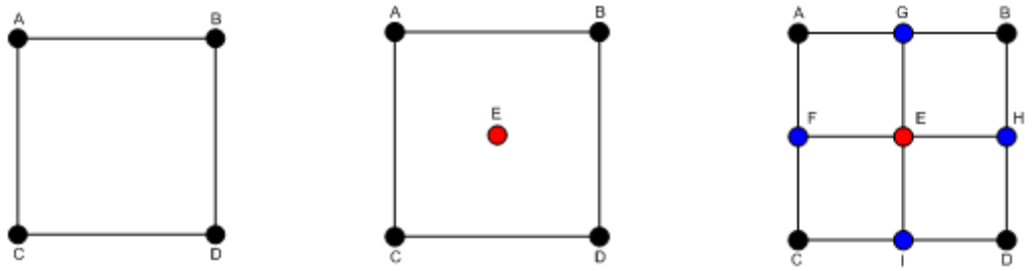


Figura A-11: Circles/Hill com a função de co-seno e mil iterações [61].

A.4.3. Mid Point Displacement

Este algoritmo descrito pela primeira vez em [67] e também conhecido por Plasma baseia-se num processo iterativo de subdivisão que provoca um aumento de detalhe a cada iteração (e consequentemente um aumento das dimensões do *height field* durante o processo). Do ponto de vista conceptual, baseia-se na forma como as montanhas se formam na natureza, mais especificamente simula o movimento tectónico das placas que ao chocarem entre si provocam o aparecimento de elevações. Uma das suas principais características é o facto de gerar apenas *height fields* quadrados de dimensões $2^n + 1 \times 2^n + 1$ [61][124], o que não traz grandes problemas uma vez que grande parte dos algoritmos de geração de terrenos em tempo real utilizam *height fields* de dimensões semelhantes. Na implementação da subdivisão existem diversos métodos [24], sendo o *Diamond Square* o mais comum. Para este, a criação do terreno inicia-se com uma *height field* vazio de dimensões 2×2, atribuindo-se a cada um dos vértices um valor igual a zero, aleatório ou predefinido. De seguida, obtém-se a altura para o ponto no centro do quadrado que irá corresponder a média das alturas dos vértices associados a este (na **Figura A-12** serão os pontos A, B, C, D), adicionando-lhe de seguida um valor aleatório entre $-d/2$ e $d/2$ em que d está na maioria dos casos associado às dimensões do terreno a criar, mas que pode ser na prática qualquer valor aleatório desde que esteja constrangido pelas dimensões máximas do *height field* a gerar. Este passo denomina-se de *Diamond Step* e utiliza a **Equação A-4** no cálculo da elevação a atribuir.



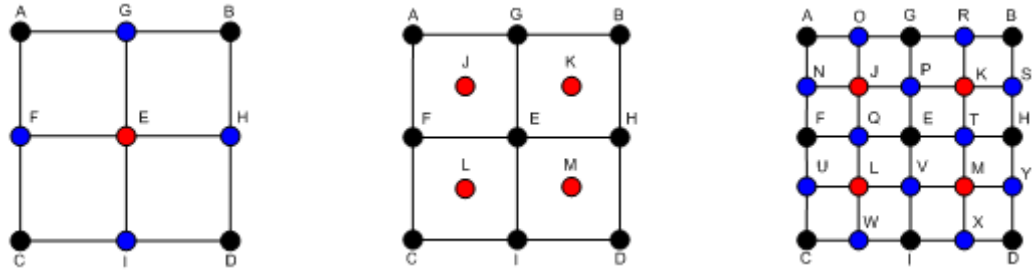
Estado Inicial | 1º Passo – *Diamond Step* | 2º Passo – *Square Step*

Figura A-12: Passos do algoritmo de Mid Point Displacement na sua 1ª iteração [61].

$E = \frac{(A+B+C+D)}{4} + \text{rand}\left(-\frac{d}{2}, \frac{d}{2}\right)$	A, B, C, D: Valores em cada um dos vértices.
	d: Valor a considerar no cálculo da elevação

Equação A-4: Cálculo do valor de elevação a atribuir no *Diamond Step*.

O 2º passo (ver **Figura A-12**) é denominado de *Square Step* e consiste numa subdivisão de cada uma das arestas do quadrado de modo a obter quatro pontos intermédios, formando-se assim quatro novos quadrados e concretizando-se a subdivisão do *height field*. Para cada um destes pontos (na **Figura A-12** o F, o G, o H e o I) é calculada a elevação fazendo a média dos valores de elevação previamente atribuídos de cada um dos pontos que lhe estão ligados, por exemplo, no caso do F são o E, o A, o C e outro valor num ponto adjacente caso exista (**Equação A-4**). No caso de não existir, podemos considerar duas formas de efectuar a média: ou se utiliza outro ponto do outro lado admitindo que o quadrado se "dobra" sobre si mesmo, ou se consideram apenas três pontos e é efectuada a média apenas com esses três. De seguida adiciona-se um valor aleatório entre $-d/2$ e $d/2$ tal como no *Diamond Step*. Terminada a primeira iteração do algoritmo, as próximas são uma repetição do processo para cada uma das subdivisões criadas (ver **Figura A-13**).

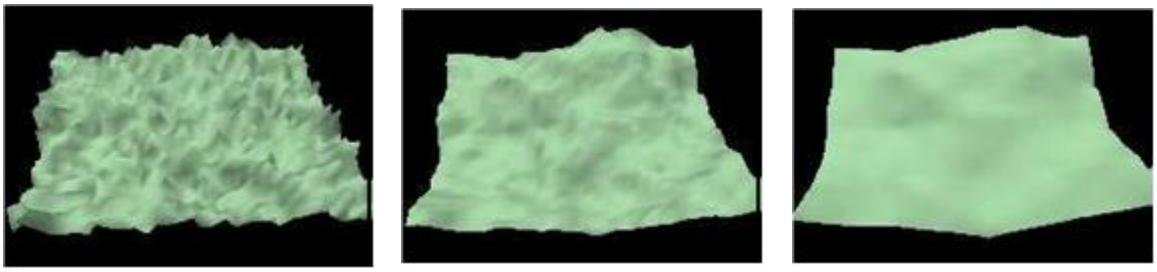


Estado Inicial | 1º Passo – *Diamond Step* | 2º Passo – *Square Step*

Figura A-13: 2ª iteração do algoritmo de *Mid Point Displacement* [61].

Antes de passar para a próxima iteração, é necessário alterar o valor da elevação atribuída a cada um dos pontos nos dois passos do algoritmo. Para isso multiplica-se o d

corrente por 2^{-r} em que r é um parâmetro fornecido ao algoritmo e que tem como função controlar o efeito de cada subdivisão do *height field*. Como se pode verificar na **Figura A-14**, este tem grande influência na superfície final gerada, conseguindo-se ter assim algum controle no resultado final, ao contrário do que sucedia com o algoritmo de Fault Formation (ver **A.4.1**). Mais precisamente quando $r > 1$, as iterações iniciais têm um efeito bastante grande dando origem a terrenos com poucas elevações. Por outro lado, para $r < 1$ as iterações finais têm um efeito maior dando origem a terrenos mais caóticos.



$r = 0.5$ $r = 1.0$ $r = 2.0$

Figura A-14: O efeito de diferentes valores de r utilizados no algoritmo [61].

A.4.4. Particle Deposition

O conceito por detrás deste algoritmo, descrito em [67], [182], [94] e em [85], está relacionado com a simulação do fluxo de lava que leva à formação de ilhas e de montanhas vulcânicas na natureza. Partindo de um *height field* vazio, a ideia é depositar sequências de partículas simulando o seu movimento e interacção com outras à medida que se vão acumulando. O processo ilustrado na **Figura A-15** começa com a escolha aleatória de um ponto no *height field* no qual se deposita a primeira partícula, ou seja, incrementa-se a altura no ponto seleccionado de acordo com algum valor predefinido (por exemplo, uma unidade) ou parametrizável. Cada uma das partículas subsequentes são "agitadas" numa direcção aleatória de modo a que nenhuma das vizinhas esteja a uma altitude menor. Isto é, sempre que possível estas partículas transitam para o nível inferior. Por exemplo, na **Figura A-15 d)** a partícula depositada (a vermelho) não se pode mover em nenhuma direcção pois existe uma partícula mais baixa em ambos os lados, não existindo neste caso específico movimento. Porém, na próxima iteração, como se pode verificar, já é possível o movimento para outra posição. Este algoritmo contempla ainda uma possível mudança periódica do ponto onde as partículas são depositadas. Esta mudança pode ser efectuada de diferentes formas, sendo possível *inclusive* manter o ponto de depósito sempre no mesmo sítio, o que daria origem a uma grande montanha, ou movê-lo ao longo de uma linha, formando-se assim uma cadeia de montanhas.

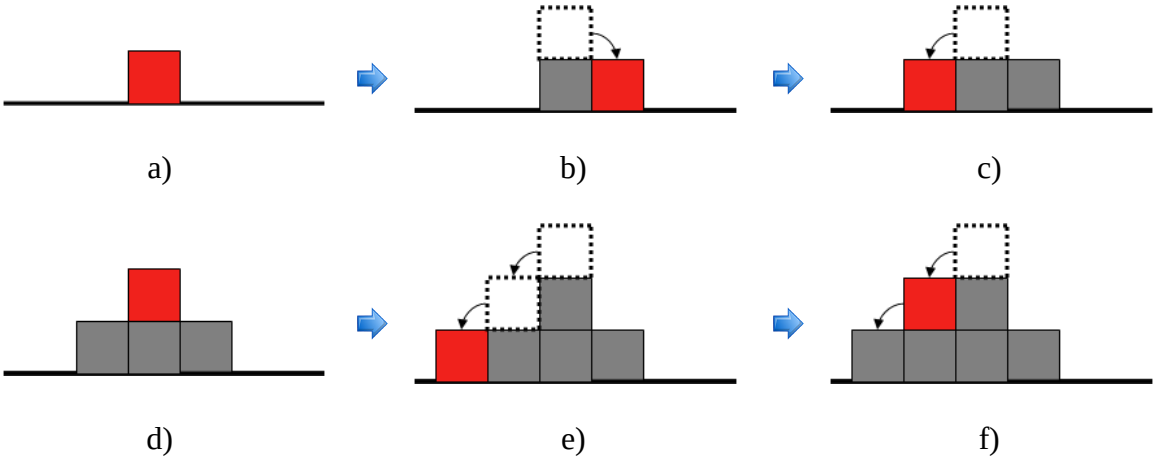


Figura A-15: Depósito de uma sequência de partículas num *height field* [182].

Em [182] é descrito ainda um processo que leva a formação de crateras nas montanhas criadas que envolve a definição de um plano de corte acima do qual todos os pontos de elevação são invertidos, tal como está representado na **Figura A-16**.

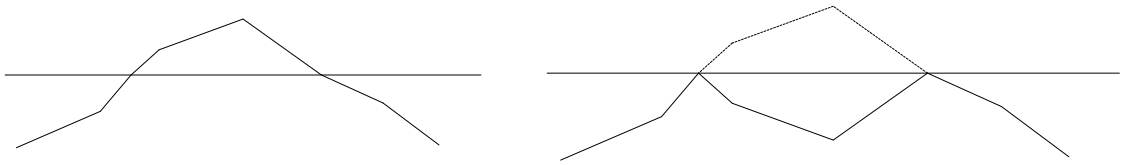


Figura A-16: Definição do plano de corte utilizado na formação da cratera [182].

Na **Figura A-17** é possível verificar que este método tende a gerar *height fields* com uma grande concentração de partículas, o mesmo será dizer com um conjunto de zonas onde estão concentradas as maiores elevações, ao passo que, nas outras zonas a altura é zero. Está por isso especialmente vocacionado para a criação de ilhas e/ou vulcões rodeados de água.

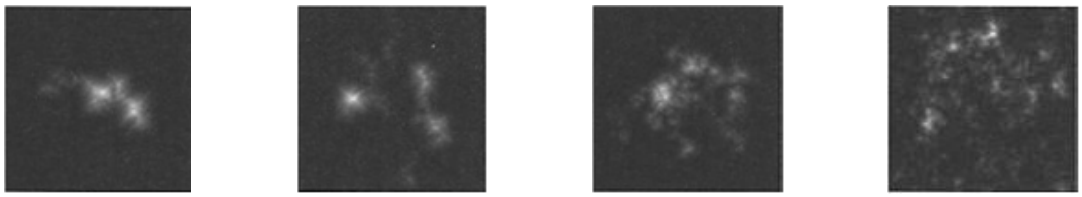


Figura A-17: Alguns *height maps* criados por Particle Deposition [182].

A.4.5. Noise Synthesis

O algoritmo de Noise Synthesis descrito por Musgrave em [139] e [140] é, ao contrário dos anteriores, um algoritmo procedimental, que permite o controlo local da dimensão fractal, da lacunaridade e do número de oitavas (ver **A.2**) e que utiliza como

função base o Perlin Noise (ver **A.4.5.1)**. Também designado de *rescale and add* [173], este algoritmo é extremamente flexível em virtude dos valores serem obtidos ponto a ponto pelo que é independente do contexto [139], o que significa que podemos obter o valor de uma determinada posição sem ser para isso necessário calcular as outras. A ideia é utilizar a função de *fractional brownian motion* (ver **A.3**) na construção do fractal o que significa combinar em diversas iterações o valor obtido na execução da função base para um ponto, alterando a frequência e a amplitude em cada uma dessas iterações. Esta ideia está representada na **Figura A-18** onde são combinadas seis oitavas de Perlin Noise de modo a obter a textura final.

Figura A-18: 6 oitavas combinadas de Perlin Noise.

Esta forma de construir o fractal já existia antes de este algoritmo surgir. A principal inovação está relacionada com um conjunto de alterações efectuadas no ciclo de construção fractal que permitem a variação da dimensão, alterações essas que Musgrave introduziu numa série de variantes da função de *fractional brownian motion* descritas em **A.4.5.2** e que são designadas de uma maneira geral de multifractais. A principal motivação está na criação de terrenos mais realistas, que permitam representar determinadas características naturais como, por exemplo, o facto de nas montanhas as zonas mais altas serem mais irregulares que as zonas mais baixas.

A.4.5.1. Perlin Noise

Perlin Noise, "a função que lançou um milhão de texturas", foi criada por Ken Perlin em 1983 [160] na sequência do trabalho que realizou no TRON [114], um filme revolucionário para a época por ter sido o primeiro a usar extensivamente a computação gráfica na geração de animações e de imagens de fundo. Esta função, publicada pela primeira vez em 1985 no artigo *An Image Synthesizer* [158], está hoje na base da maior parte dos algoritmos de criação de texturas existentes nos pacotes comercias de 3D e de tratamento de imagem. Ken Perlin chegou *inclusive* a ser galardoado com um Óscar em 1997 pelo seu trabalho no campo da geração procedimental de texturas em virtude do grande impacto que este teve na indústria dos efeitos especiais para cinema. O objectivo inicial de Perlin era criar uma primitiva que permitisse preencher o espaço com um conjunto de valores que parecessem aleatórios, mas que pudessem ser de alguma forma controlável. É importante definir neste contexto a noção de "aleatório". Normalmente este conceito significa aparentemente aleatório ou mais precisamente, pseudo-aleatório. Uma primitiva verdadeiramente aleatória é uma fonte de *white noise*, ou seja, um conjunto uniformemente distribuído de valores sem nenhuma correlação entre valores sucessivos de que é exemplo a imagem de uma televisão sem um canal sintonizado. Uma aproximação a este tipo *noise* pode ser obtida por intermédio das funções pseudo-aleatórias tradicionais que existem na grande maioria das API. Estas funções utilizam um valor, designado de semente, que permite inicializar o algoritmo de geração

de aleatória. Este número é muitas vezes derivado a partir de fontes dinâmicas como seja, por exemplo, o relógio interno do computador. Claramente isso significa que para momentos diferentes se obtém valores diferentes. No entanto, não é este o comportamento desejado, pois isso significa que o *white noise* nunca é o mesmo duas vezes, o que nos impede de obter um determinado padrão mais do que uma vez.Na realidade, o que pretendemos é uma função aparentemente aleatória, mas cujo padrão produzido possa ser repetido em função de algum género de parâmetros de entrada [52], normalmente um conjunto de coordenadas, o que contrasta com as funções verdadeiramente aleatórias que por definição não tem parâmetros de entrada. Em relação á utilização de um conjunto de coordenadas, como parâmetros de entrada de um gerador de valores pseudo-aleatórios, isso é algo que pode ser concretizado facilmente (por exemplo, utilizando as coordenadas como índice num *array* de valores pseudo-aleatórios). O problema é a descontinuidade que existe entre dois valores sucessivos nestas funções, descontinuidade que leva a que os padrões gerados sejam muitas vezes designados de *non-coherent noise*. Assim, está visto que esses valores têm de ter características específicas, mais precisamente para quaisquer dois pontos no espaço o valor da função deve mudar "suavemente" à medida que se transita de um ponto para outro, não causando desta forma descontinuidades [228]. Por isso, os padrões produzidos por funções com estas características são normalmente designados de *coherent noise*. Os dois tipos de *noise* referidos estão representados na **Figura A-19** onde são claras as diferenças e onde se percebe facilmente que a textura à direita é a mais adaptada para ser usada como *height field*.

Non-coherent noise *Coherent noise*

Figura A-19: *Non-coherent noise* vs *coherent noise* [228].

De uma forma mais precisa a função de *noise* ideal tem de ter as seguintes características [52]:

- Deve ser uma função pseudo-aleatória em função dos seus *inputs*.
- Tem de ter um intervalo de valores bem definido, normalmente entre -1.0 e 1.0.
- Deve ser limitada na banda, com uma frequência máxima de aproximadamente 1
- Não deve exibir periodicidades óbvias ou padrões regulares.
- Deve ser estacionária, ou seja o carácter estatístico deve ser translacionalmente invariante (invariante ao factor de escala aplicado).
- Deve ser isotrópica, ou seja o carácter estatístico deve ser rotacionalmente invariante (invariante às rotações aplicadas).

Existem várias formas de gerar *noise* com estas características. No caso do Perlin Noise, são definidos numa grelha (também designada de *lattice*) num espaço de n dimensões, um conjunto de gradientes (que correspondem a vectores que apontam numa determinada direcção) que são depois interpolados por uma função específica. Vamos ver então para $n = 2$ como se processa a geração de Perlin Noise, começando por definir o formato da função. Assim, temos $noise2D(x, y) = z$ com x e y como coordenas e z como resultado, todos números reais. O primeiro passo consiste em encontrar as quatro coordenadas de cada um dos pontos da grelha que envolvem o ponto (x, y), recebido como parâmetro da função. Nesta grelha cada um dos pontos é um número inteiro pelo que as coordenadas recebidas como parâmetro, sendo números reais, vão situar-se algures no meio de uma das células. Por sua vez cada uma das células é definida por quatro pontos que vamos chamar de (x_0, y_0), (x_0, y_1), (x_1, y_0) e de (x_1, y_1) tal como está representado à esquerda da **Figura A-20** onde o quadrado a negro representa o ponto (x, y) enviado por parâmetro. Por exemplo, para $(x, y) = (2.4,4.3)$os valores de cada um dos pontos envolventes seriam respectivamente $(x_0, y_0) = (2,4)$, $(x_0, y_1) = (2,5)$, $(x_1, y_0) = (3,4)$ e $(x_1, y_1) = (3,5)$. O próximo passo, representado à direita da figura, corresponde à atribuição de um gradiente pseudo-aleatório a cada um dos pontos que envolvem o ponto (x, y), operação definida pela função $g(x_n, y_n) = (g_x, g_y)$, na qual como já foi referido anteriormente é importante que para as mesmas coordenadas seja devolvido sempre o mesmo gradiente.

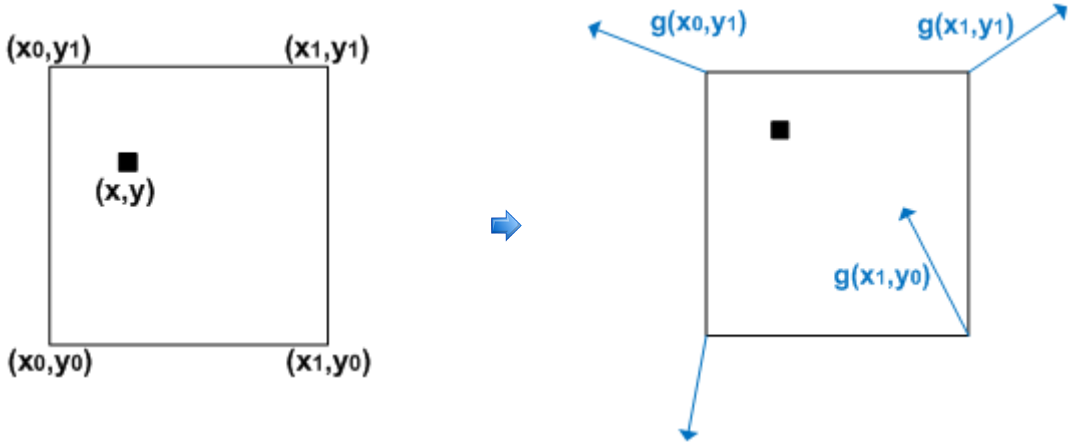


Figura A-20: Definição dos quatro pontos envolventes e dos gradientes [228].

De seguida, para cada um dos pontos da grelha é definido um vector (à esquerda na **Figura A-21**). Estes vectores vão de cada um dos pontos envolventes para o ponto (x, y) em consideração e podem ser obtidos facilmente subtraindo ao ponto (x, y) cada um deles. Resta agora obter a influência de cada um dos gradientes pseudo-aleatórios no resultado final. Para isso, em cada um dos pontos é calculado o produto interno[11] do gradiente com o vector que vai do ponto correspondente ao ponto (x, y). Estes estão representados à direita na figura pelos vectores que partem de cada um dos pontos envolventes e pelos vectores que apontam para o ponto (x, y). Aos quatro valores obtidos vamos chamar de s, t, u e v:

[11] O produto interno entre dois vectores a e b é definido por $a \cdot b = \sum_{i=1}^{n} a_i b_i$

- $s = g(x_0, y_0) \cdot \big((x, y) - (x_0, y_0)\big)$.
- $t = g(x_1, y_0) \cdot \big((x, y) - (x_1, y_0)\big)$.
- $u = g\big(x_{0,} y_1\big) \cdot \big((x, y) - (x_0, y_1)\big)$.
- $v = g\big(x_{1,} y_1\big) \cdot \big((x, y) - (x_1, y_1)\big)$.

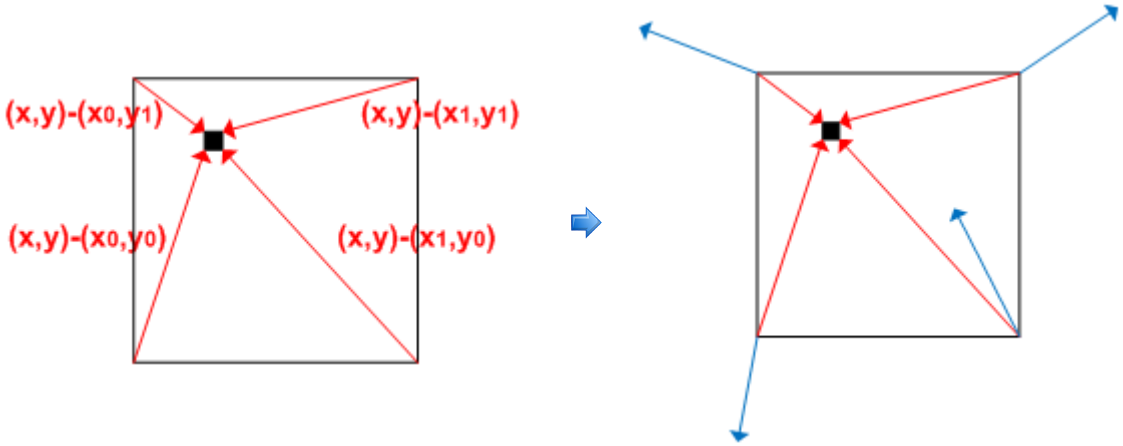


Figura A-21: Definição do vector e do gradiente utilizados no produto interno [228].

Obtidos estes valores, é agora necessário combiná-los de modo a chegar ao resultado final, para isso temos de fazer uma média ponderada. Para quatro números, a forma mais fácil é encontrar a média do primeiro e do segundo par, efectuando depois a média dos resultados de cada um deles. Para isso vamos considerar os pares (s, t) e (u, v), selecção propositada na medida em que tanto num como noutro existe variação apenas na coordenada x pelo que podemos utilizar como factor ponderador na média de cada um deles um peso calculado em função de uma única dimensão. O objectivo é calcular a média de (s, t) e de (u, v) com um factor ponderador s_x (visto que a variação nestes pares é na coordenada x) obtendo um novo par, para o qual executamos também a média mas desta vez com um factor s_y que representa a variação na coordenada y. O objectivo destes pesos é influenciar o valor final de maneira a criar uma transição suave entre valores pseudo-aleatórios sucessivos. A ideia base traduz-se no seguinte: se o x do par (x, y) fornecido como parâmetro estiver mais perto de x_0 do que de x_1 o resultado da função deve ser mais influenciado por s do que por t e da mesma forma, mais por u do que por v, aplicando-se também esse mesmo conceito para a coordenada y. Para calcular os pesos é utilizada **Equação A-5** ou a **Equação A-6**, substituindo t por $x - x_0$ e $y - y_0$ para calcular s_x e s_y respectivamente.

$s = 3t^2 - 2t^3$	t: Variável onde se substitui o valor das coordenadas.

Equação A-5: Cálculo dos pesos no Perlin Noise (1ª versão).

$s = 6t^5 - 15t^4 + 10t^3$	t: Variável onde se substitui o valor das coordenadas.

Equação A-6: Cálculo dos pesos no Perlin Noise (2ª versão).

Perlin, utilizou no método original a 1ª equação, no entanto embora esta seja mais rápida, pode originar artefactos no resultado final. Num artigo posterior [159] Perlin descreve as razões por detrás desses artefactos e introduz a 2ª equação como uma solução para esses problemas. Finalmente, para calcular as médias utilizamos a **Equação A-7**, uma interpolação linear.

$m = a + s(b - a)$	a: Valor inicial da interpolação.
	b: Valor final da interpolação.
	s: Peso calculado com a **Equação A-5** ou a **Equação A-6**.

Equação A-7: Cálculo das médias no Perlin Noise.

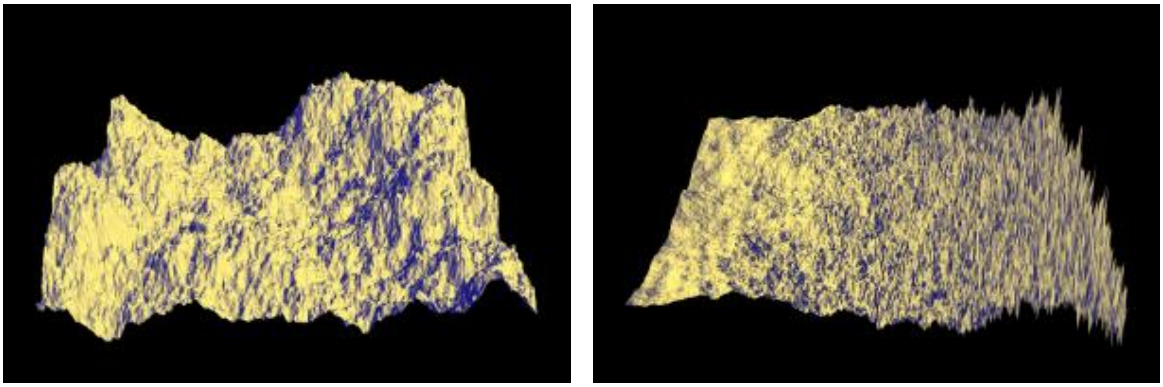
Resumindo temos de executar o seguinte conjunto de passos:

1. Calcular o peso s_x, com a **Equação A-5** ou a **Equação A-6**, substituindo t por $x - x_0$.
2. Calcular o peso s_y, com a **Equação A-5** ou a **Equação A-6**, substituindo t por $y - y_0$.
3. Efectuar uma interpolação linear (**Equação A-7**) entre s e t com o peso s_x calculado em 1.
4. Efectuar uma interpolação linear (**Equação A-7**) entre u e v com o peso s_x calculado em 1.
5. Efectuar uma interpolação linear (**Equação A-7**) entre os resultados obtidos em 3 e 4 com o peso s_y calculado em 2 obtendo-se finalmente o valor de z.

A.4.5.2. Multifractal

A função de *fractional brownian motion* descrita em **A.3** que serve de modelo na construção de terrenos fractais é estatisticamente homogénea (estacionaria) e isotrópica, o que significa, respectivamente, "o mesmo em todo o lado" e o "mesmo em todas as direcções" [52]. Muito embora os terrenos criados desta forma tenham já um nível de detalhe razoável não o distribuem da melhor forma em comparação com os terrenos reais em que a distribuição do detalhe não é uniforme. As montanhas, por exemplo, não têm o mesmo detalhe em todo o lado, normalmente as zonas mais altas têm mais irregularidades que as zonas mais baixas pois a maior parte delas erguem-se de zonas planas. Tal como já foi referido em **A.2** a propriedade nos fractais que controla a complexidade é a dimensão. No modelo inicial de *fractional brownian motion* esta é constante em todas as iterações (oitavas) consideradas pelo que essa versão da função é designada de *monofractal*. Foi numa tentativa de colmatar os problemas do modelo inicial que em [52], [139] e [140] se descreve um conjunto de variações da função de *fractional brownian motion*, designadas de um modo geral de *multifractais*. Um *multifractal* pode-se definir como um fractal heterogéneo, cuja heterogeneidade é invariante com a escala (a heterogeneidade pode significar, por exemplo, que os ponto mais altos do terreno tem mais detalhe, comportamento que se mantêm em todas as escalas). Mais precisamente, este tipo de fractais, passa também a ser definido por uma dimensão que varia entre escalas (mas que varia sempre da mesma forma, pelo que mantem as características do modelo inicial).

Este conceito de *multifractal* é ilustrado na **Figura A-22**, onde à esquerda está representado um terreno com um dimensão fractal constante e à direita um terreno com uma dimensão fractal variável.



Dimensão fractal homogénea (aproximadamente 2.2)

A dimensão fractal varia da esquerda para a direita (de 2.0 para 3.0)

Figura A-22: Comparação entre um *monofractal* e um *multifractal* [52].

Para melhor esclarecer as diferenças entre um monofractal e um multifractal apresentam-se de seguida algumas implementações da função de *fractional brownian motion* descritas por Musgrave, a primeira com uma dimensão fractal constante (a implementação mais comum) e as restantes com uma dimensão fractal variável. Assim começando pela versão homogénea, designada simplesmente de fBm, é apresentada na **Listagem A-1** o pseudo-código correspondente. Como podemos verificar, a implementação em si é muito simples, pois o fractal propriamente dito é construído num ciclo de apenas quatro linhas (da linha 7 a 11 excluindo os espaços). Em cada iteração é modificada a amplitude e a frequência da função base (o Perlin Noise) o que está de acordo com a definição de *fractional brownian motion* dada em **A.3**. Esta função vai servir de base a todas as construções multifractais descritas de seguida, pelo que em cada uma delas são evidenciadas as diferenças.

```
01 fBm(point,H,lacunarity,octaves)
02  frequency:= 1.0
03
04  result:= noise(point)
05
06  For octave:= 2 To octaves
07   frequency:= frequency * lacunarity
08   amplitude:= pow(frequency, -H)
09   point:= point * lacunarity
10
11   result:= result + noise(point) * amplitude
12  End For
13
14  Return result
15 End fBm
```

Parâmetros
point: Ponto a avaliar. **H**: Parâmetro de incremento fractal. **lacunarity**: O intervalo entre frequências. **octaves**: O número de frequências consideradas.
Funções
noise(): função base, neste caso Perlin Noise **pow()**: potência

Listagem A-1: Implementação de *fractional brownian motion* [52][139].

Como já foi referido, os modelos de terreno gerados por esta função não representam muitas das características dos verdadeiros terrenos. Daí a necessidade de criar novos

modelos que se baseiam na modificação da dimensão fractal ao longo das iterações consideradas. O primeiro multifractal descrito por Musgrave tenta concretizar um facto muito simples: as áreas mais altas, por exemplo, as montanhas são mais irregulares que as áreas mais baixas. Uma variação da função de *fractional brownian motion* que permite concretizar esse facto é apresentada na **Listagem A-2**. O objectivo é alcançado pela multiplicação de cada oitava pelo valor corrente da função. Assim, em áreas com uma elevação próxima de zero as frequências mais altas são reduzidas mantendo o terreno com poucas irregularidades. Para frequências mais altas essa redução já não é tão proeminente pelo que essas zonas se tornam mais irregulares o que permite atingir o objectivo pretendido.

```
01 heterofBm(point,H,lacunarity,octaves,offset)
02  frequency:= 1.0
03
04  result:= noise(point) + offset
05
06  For octave:= 2 To octaves
07   frequency:= frequency * lacunarity
08   amplitude:= pow(frequency, -H)
09   point:= point * lacunarity
10
11   increment:= (noise(point) + offset) * amplitude
12   increment:= increment * result
13   result:= result + increment
14  End For
15
16  Return result
17 End hfBm
```

Parâmetros

point: Ponto a avaliar.

H: Parâmetro de incremento fractal.

lacunarity: O intervalo entre frequências.

octaves: O número de frequências consideradas.

offset: Elevação a partir da qual a dimensão fractal começa a mudar.

Funções

noise(): função base, neste caso Perlin Noise.

pow(): potência

Listagem A-2: *Hetero multifractal* [52][139].

Musgrave descreve ainda outras duas variações, respectivamente, o *hybrid* e o *ridged multifractal*. O *hybrid multifractal* resulta da observação de uma outra característica dos terrenos que não era contemplada pela aproximação anterior e que está relacionada com os vales. Nestes, as zonas relativamente planas com poucas irregularidades podem existir a todas as alturas não apenas nas zonas mais baixas. Para contemplar essa situação Musgrave propõe um aumento de escala das frequências mais altas, na soma efectuada, pelo valor local da frequência anterior [52], o que resulta numa mistura entre a soma e o produto das oitavas, razão pela qual se atribui o nome *hybrid*. O pseudo-código desta função está representado na **Listagem A-3**.

```
01 hybridfBm(point,H,lacunarity,octaves,offset,gain)
02  frequency:= 1.0
03  weight:= 1.0
04
05  signal:= noise(point) + offset
06  result:= signal
07
08  For octave:= 2 To octaves
09   frequency:= frequency * lacunarity
10   amplitude:= pow(frequency, -H)
11   point:= point * lacunarity
12   weight:= weight * gain * signal
13
14   If weight > 1.0 Then weight:= 1.0
15   signal:= (noise(point) + offset) * amplitude
16   result:= result + (signal * weight)
17  End For
18
19  Return result
20 End hfBm
```

Parâmetros
point: Ponto a avaliar.
H: Parâmetro de incremento fractal.
lacunarity: O intervalo entre frequências.
octaves: O número de frequências consideradas.
offset: Elevação a partir da qual a dimensão fractal começa a mudar.
gain: Determina a gama de valores criados pela função.
Funções
noise(): função base, neste caso Perlin Noise.
pow(): potência

Listagem A-3: *Hybrid multifractal* [52][139].

O *ridged multifractal* cujo pseudo-código é apresentado na **Listagem A-4** baseia-se na alteração do valor de Perlin Noise obtido em cada iteração, mais especificamente no cálculo de $1 - \text{abs}(noise)$. Este método é um dos melhores para criar montanhas, especialmente cordilheiras de montanhas de picos aguçados, exibindo as mesmas características do método anterior.

```
01 ridgedfBm(point,H,lacunarity,octaves,offset,gain)
02  frequency:=  1.0
03
04  signal:= offset - abs(noise(point))
05  signal:= signal * signal
06  result:= signal
07
08  For octave:= 2 To octaves
09   frequency:= frequency * lacunarity
10   amplitude:= pow(frequency, -H)
11   point:= point * lacunarity
12   weight:= signal * gain
13
14   If weight > 1.0 Then weight:= 1.0
15   If weight < 0.0 Then weight:= 0.0
16   signal:= offset - abs(noise(point))
17   signal:= signal * signal * amplitude
18   result:= result + (signal * weight)
19  End For
20
21  Return result
22 End ridgedfBm
```

Parâmetros
point: Ponto a avaliar.
H: Parâmetro de incremento fractal.
lacunarity: O intervalo entre frequências.
octaves: O número de frequências consideradas.
offset: Elevação a partir da qual a dimensão fractal começa a mudar.
gain: Determina a gama de valores criados pela função.
Funções
noise(): função base, neste caso Perlin Noise.
abs(): valor absoluto
pow(): potência

Listagem A-4: *Ridged multifractal* [52][139].

Um factor importante que não está contemplado no código de nenhuma das listagens apresentadas, mas que tem de ser levado em consideração numa implementação, é a possibilidade de as oitavas serem números reais. Nesse caso temos de efectuar o cálculo do resto. Este facto é muito importante, pois se não fosse levado em consideração, dava origem a falhas na criação de texturas adjacentes. Por exemplo, se este método fosse utilizado para gerar o *height field* para um bloco de terreno, quando se pretendesse fazer o

mesmo para um bloco adjacente (naturalmente em coordenadas também adjacentes) a transição entre as duas texturas poderia originar mudanças abruptas no terreno.

A.5. Sumário

Neste anexo abordou-se a síntese de terrenos dando-se especial relevância aos métodos baseados em fractais. O objectivo foi divulgar esta forma de criar os terrenos numa perspectiva de complementar o capítulo **2** em que este tema não foi abordado em detalhe por não ser o foco da dissertação, pelo que é discutido neste anexo.

A principal vantagem deste método é permitir a criação das fontes de elevação mediante o fornecimento de apenas um conjunto de parâmetros. Tal permite consequentemente uma compressão imensa dos dados pelo que as necessidades de armazenamento passam nesse caso a ser praticamente negligenciáveis. Além disso, os fractais permitem a criação de terrenos realistas sendo por isso uma das melhores ferramentas à nossa disposição na síntese de terrenos. Em contrapartida, o controlo sobre o terreno gerado é muito menor, já que se está dependente da configuração de um conjunto parâmetro.

Neste anexo começa-se por abordar o conceito de fractal propriamente dito (ver **A.1**). Assim um fractal é definido como um objecto geometricamente complexo, cuja complexidade advêm da repetição de uma determinada forma num intervalo de escalas. Essa repetição é denominada de *self similarity* sendo possível distinguir dois tipos: *statistical self similarity* e *exact self similarity*. A segunda é particularmente relevante neste contexto já que incorpora variações aleatórias na sua construção e é por isso especialmente útil na criação de terrenos.

No que diz respeito á construção do fractal são considerados quatro parâmetros (ver **A.2**):

- A função base.
- O parâmetro de dimensão fractal.
- A lacunaridade.
- O número de oitavas.

Estes parâmetros controlam todo o processo de criação que resulta efectivamente de um conjunto de iterações (oitavas) em que se repete a função base modificando a amplitude dessa função com o valor do parâmetro de dimensão fractal e a frequência com o valor da lacunaridade. Para estabelecer a relação entre a amplitude e a frequência da função base é utilizada a função de *fractional brownian motion* (ver **A.3**).

Uma vez estabelecida a fundação teórica necessária para se entender o conceito de fractal abordaram-se os algoritmos propriamente ditos, distinguindo-se dois tipos: procedimentais e não procedimentais. Os métodos procedimentais são avaliados onde e quando são necessários, normalmente em tempo de *rendering*, isto é, geram os valores a pedido, que são avaliados a cada ponto e não estão desta forma associados a um processo iterativo fechado. Os métodos não procedimentais por seu lado, actuam de uma forma global e de uma só vez numa fase de pré-processamento não sendo por isso os mais adaptados para uma utilização em tempo real.

Descreveram-se de seguida um conjunto de algoritmos procedimentais e não procedimentais. Os algoritmos não procedimentais abordados em detalhe foram o Fault

Formation (ver **A.4.1**), o Circles (ver **A.4.2**), o Mid Point Displacement (ver **A.4.3**) e o Particle Deposition (ver **A.4.4**). Discute-se apenas um método procedimental, o *noise synthesis* (ver **A.4.5**), ao qual é dado especial destaque por ser um dos mais utilizados. Este tem como principal vantagem permitir o controlo local da dimensão fractal, da lacunaridade e do número de oitavas utilizando como função base o Perlin Noise discutido em detalhe em **A.4.5.1**. Por outro lado implementa a função de *fractional brownian motion* pelo que tem uma base matemática rigorosa ao contrário dos métodos não procedimentais.

É descrito ainda em **A.4.5.2** o conceito de *multifractal* que procura adaptar a função de *fractional brownian motion* com o intuito de produzir valores de elevação que representam melhor os terrenos que encontramos na natureza. A função original é estatisticamente homogénea (estacionaria) e isotrópica, ou dito de outra forma produz *monofractais*. Os *multifractais* pelo contrário caracterizam-se pela sua heterogeneidade sendo invariantes com a escala. Definiram-se assim neste domínio, um conjunto de alterações à função de *fractional brownian motion* designadas de *hetero*, *hybrid* e *ridged multifractal* respectivamente. Estas permitem obter terrenos muito mais realistas e são por isso as mais utilizadas para esse efeito.

Bibliografia

[1] .NET Compact Framework Tem Blog. *Managed Code Performance on Xbox 360 for XNA: Part 1 - Intro and CPU*. http://blogs.msdn.com/netcfteam/archive/2006/12/22/managed-code-performance-on-xbox-360-for-the-xna-*framework*-1-0.aspx

[2] .NET Compact Framework Tem Blog. *Managed Code Performance on Xbox 360 for XNA: Part 2 - GC and Tools*. http://blogs.msdn.com/netcfteam/archive/2006/12/22/managed-code-performance-on-xbox-360-for-xna-part-2-gc-and-tools.aspx

[3] .THEPRODUKKT. *KKrieger*. http://theprodukkt.com/kkrieger

[4] *A Virtual Stroll in 3-D through Lucerne – Switzerland*. http://www.3dcity.info/luzern/e/

[5] ABRANTES Graça. *Sistemas de Informação Geográfica – Conceitos*. 1998. http://www.isa.utl.pt/dm/sig/sig/SIGconceitos.html.

[6] ADOBE. *Adobe Photoshop*. http://www.adobe.com/products/photoshop/

[7] AIREY John. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, University of North Carolina, Chappel Hill, 1991.

[8] AIREY John M., ROHLF John H. e BROOKS Frederick P. Jr. *Towards image realism with interactive update rates in complex virtual building environments*. Computer Graphics (1990 Symposium on Interactive 3D Graphics), 24(2):41–50, March 1990.

[9] AQUILIO Anthony S. *A Framework For Dynamic Terrain With Application In Off-Road Ground Vehicle Simulations*. A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in the College of Arts and Sciences of Georgia State University. Under the Direction of Ying Zhu and G. Scott Owen. 2006. http://etd.gsu.edu/theses/available/etd-11162006-155442/unrestricted/aquilio_anthony_s_200612_phd.pdf

[10] ASIRVATHAM Arul, HOPPE Hugues. *Terrain Rendering Using GPU-Based Geometry Clipmaps*. Game Developers Conference. Março 2005. http://download.nvidia.com/developer/presentations/2005/GDC/Sponsored_Day/GPUTerrain.pdf

[11] ASIRVATHAM Arul, HOPPE Hugues. *Terrain Rendering Using GPU-Based Geometry Clipmaps*. GPU Gems 2. Publicado por Addison-Wesley. ISBN: 0321335597. 2005. http://research.microsoft.com/~hoppe/gpugcm.pdf

[12] AUTODESK. *Data Exchange File (DXF)*. http://www.autodesk.com/techpubs/autocad/acad2000/dxf/

[13] BENTLY J.H. e FRIEDMA J. H. *Datastructures for range searching*. ACM Computing Surveys, 11(4):397–409, 1979.

[14] BENTLY, J. L. *Multidimensional binary search in database appliacations*. IEEE Transactions on Software Engineering, 4(5):333–340, 1979.

[15] BENTLY, J. L. *Multidimensional binary search trees used for associative searching*. Communications of the ACM, 18(9):509–517, 1975.

[16] BITTNER Jirí, WIMMER Michael, PIRINGER Harald e PURGATHOFER Werner. *Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful*. Setembro 2004. http://www.cg.tuwien.ac.at/research/vr/chcull/bittner-eg04-chcull.pdf

[17] BLOW Jonathan. *Terrain Rendering at High Levels of Detail*. Proceedings Game Developers Conference (GDC). 2000. http://number-none.com/blow/papers/terrain_rendering.pdf

[18] BLOOM Charles Terrain Texture Compositing by Blending in the Frame-Buffer (aka "Splatting" Textures). November 2000. http://cbloom.com/3d/techdocs/splatting.txt

[19] BLOW Jonathan. *Terrain Rendering Research for Games*. Course Notes for SIGGRAPH 2000 Course #39. 2000. http://www.red3d.com/siggraph/2000/course39/S2000Course39SlidesBlow.pdf

[20] BOGOMJAKOV Alexander, GOTSMAN Craig. *Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes*. http://www.cs.technion.ac.il/~gotsman/caching/downloads/caching.pdf.

[21] BOURKE Paul. *Self Similarity*. http://local.wasp.uwa.edu.au/~pbourke/fractals/selfsimilar/

[22] BRETTELL Nick. Terrain Rendering Using Geometry Clipmaps. Honours report submitted for the University of Canterbury under the supervision of Dr. R. Mukundan. Novembro 2005. http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/2005/hons_0502.pdf

[23] BROGGER Nicolai de Haan. Real-time Rendering of Large Terrains with Support for Runtime Modifications. Bachelor Projekt. Datalogisk Institut (DIKU). Københavns Universitet. 2008.

[24] BURKE C. *Generating Terrain*. http://www.geocities.com/Area51/6902/terrain.html.

[25] CHEVALIER Fanny, CAURANT Guillaume. *Rendering Massive Terrains using Chunked Level of Detail Control*. http://www.labri.fr/perso/preuter/imageSynthesis/0304/presentations/fannyguillaum.ppt

[26] CATMULL Edwin E. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.d. thesis, University of Utah, Dezembro 1974.

[27] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F. e SCOPIGNO, R. *BDAM – Batched dynamic adaptive meshes for high performance terrain visualization*. Computer Graphics Forum 22(3).2003. http://www.crs4.it/vic/data/papers/eg2003-bdam.pdf

[28] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F. e SCOPIGNO R. *Planet-sized batched dynamic adaptive meshes (P-BDAM)*. IEEE Visualization 2003. http://www.crs4.it/vic/data/papers/ieeeviz03-pbdam.pdf

[29] CIGNONI P., PUPPO E., e SCOPIGNO R. *Representation and Visualization of Terrain Surfaces at Variable Resolution*. The Visual Computer, vol. 13, pp. 199-217. 1997. http://dienst.isti.cnr.it/Dienst/Repository/2.0/Body/ercim.cnr.iei/1996-B4-27-10/pdf?tiposearch=ercim&langver=

[30] CLARK, J. H.*Hierarchical Geometric Models for Visible Surface Algorithms*. Communications of the ACM. vol 19(10).pp. 547-554. 1976.
[31] CLASEN Malte e HEGE Hans-Christian. *Terrain Rendering using Spherical Clipmaps*. Zuse Institute Berlin, Germany. 2006. http://www.zib.de/clasen/download/SphericalClipmaps_Electronic.pdf
[32] COHEN-OR Daniel, CHRYSANTHOU Yiorgos, SILVA Cláudio T., DURAND Frédo. *A Survey of Visibility for Walkthrough Applications*. http://people.csail.mit.edu/fredo/PUBLI/surveyTVCG.pdf.
[33] COHEN-OR D. e LEVANONI Y. *Temporal Continuity of Levels of Detail in Delaunay Triangulated Terrain*. Proceedings of the 7th conference on Visualization, pp. 37-42, 1996.
[34] COHEN Jonathan D., MANOCHA Dinesh. *Model Simplification*. http://www.cse.iitb.ac.in/~lakulish/seminar/papers/Manocha-Cohen-Simplification.pdf
[35] COSMAN, M. A.,MATHISEN A. E.,ROBINSON, J. A.*A New Visual System to Support Advanced Requirements*. In Proceedings, IMAGE V Conference. 1990.
[36] CRISTAL SPACE TEAM. *Crystal Space*. http://www.crystalspace3d.org/main/Main_Page
[37] CRYTEK. *Far Cry*. http://www.farcry-thegame.com/uk/
[38] DACHSBACHER Carsten. *Cached Procedural Textures for Terrain Rendering*. ShaderX 4. Publicado por Charles River Média Inc. ISBN: 1584504250.
[39] DE BOER, Willem H. *Fast Terrain Rendering Using Geometrical Mipmapping*. Outubro 2000. http://www.flipcode.com/archives/article_geomipmaps.pdf.
[40] DE FLORIANI L.,FALCIDIENO B.,PIEN-OVI C. *A Delaunay-Based Method for Surface Approximation*. Proceedings of Eurographics' 83. Pp. 333-350. 1983
[41] DE FLORIANI L., MAGILLO P e PUPPO E. *VARIANT: A System for Terrain Modeling at Variable Resolution*. GeoInformatica. Vol. 4(3). Pp. 287-315. 2000.
[42] DE FLORIANI L., MAGILLO P. e PUPPO E. *Building and Traversing a Surface at Variable Resolution*. IEEE Visualization 1997, 103-110.
[43] DELAUNAY B. *Sur la sphère vide*. Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk. 1934.
[44] DELEON Victor J., BERRY Robert H. *Virtual Florida Everglades*. http://digitalo.com/deleon/vrglades/VDeleon-vrglades.pdf
[45] DirectX. http://msdn.microsoft.com/directx/
[46] DITCHBURN Keith.Toymaker. http://www.toymaker.info
[47] DISCOE Ben. Virtual Terrain Project. http://vterrain.org.
[48] DOWNS Laura, MÖLLER Tomas, SÉQUIN Carlo H. *Occlusion Horizons for Driving through Urban Scenery*. Proceedings 2001 Symposium on Interactive 3D Graphics, pp. 121-124. Março 2001. http://www.cs.lth.se/home/Tomas_Akenine_Moller/pubs/i3d2001.pdf
[49] DUCHAINEAU Mark, WOLINKSY Murray, SIGETI David E, MILLER M C, ALDRICH Charles e MINEEV-WEINSTEIN Mark B. *ROAMing Terrain: Real-Time Optimimally Adapting Meshes.* Proceedings IEEE Visualization' 97. pp. 81-88. 1997. http://www.llnl.gov/graphics/ROAM/.
[50] DURAND Frédo. *3D Visibility: Analytical Study and Applications*. Dissertation presented in partial fulfillement of the requirements for the degree of Docteur de

l'Université Joseph Fourier, discipline informatique. 1999. http://people.csail.mit.edu/fredo/THESE/vo.pdf

[51] EBERLY, David H. Series In Interactive 3d Technology. *3D Game Engine Design: A pratical Approach to Real-Time Computer Graphics*. Second Edition. ISBN: 0122290631.

[52] EBERT David S, MUSGRAVE F. Kenton, PEACHEY Darwyn, Perlin Ken, Worley Steven. *Texture and Modelling: A Procedural Approach*. Third Edition. ISBN: 1558608486. http://www.texturingandmodeling.com/

[53] ECLIPSE ENTERTAINMENT. *Genesis3D*. http://www.genesis3d.com/

[54] EINSTEIN Albert. *Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen*. Ann. Phys. 17, 549, 1905. http://lorentz.phl.jhu.edu/AnnusMirabilis/AeReserveArticles/eins_brownian.pdf

[55] ELIAS Hugo. *Perlin Noise*. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

[56] EL-SANA J. e VARSHNEY A. *Generalized View-Dependent Simplification*. Proceedings of Eurographics, 1999. http://www.cs.umd.edu/projects/gvil/papers/elsana_eg99.pdf

[57] ENGEL, Wolfgang F. Programming Vertex and Pixel Shaders. Publicado por Charles River Média Inc. ISBN:1584503491.

[58] ERIKSON Carl, MANOCHA Dinesh, BAXTER III William V. *HLODs for Faster Display of Large Static and Dynamic Environments*. http://www.cs.unc.edu/~walk/hlod/paper/Erikson_HLODs_I3D01.pdf.

[59] ESRI. *ESRI Shapefile Technical Description*. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[60] ExDream Entertainment. *XNA Racing Game*. http://www.xnaracinggame.com/.

[61] FERNANDES António Ramires. *Terrain Tutorial*. http://www.lighthouse3d.com/opengl/terrain/

[62] FERNANDO Randima (Randy). *Shader Model 3.0 Unleashed*. NVIDIA Developer Technology Group. ftp://download.nvidia.com/developer/presentations/2004/SIGGRAPH/Shader_Model_3_Unleashed.pdf

[63] FIEDLER Glenn. *Terrain Occlusion Culling with Horizons*. Game Programming Gems 4. Publicado por Charles River Média Inc. 2004. ISBN: 1584502959.

[64] FINKEL R. A. e BENTLEY J. L. *Quad Trees: A Data Structure for retrieval on composite keys*. Acta Informatica 4. 1(1974), 1-9.

[65] FORSYTH Tom. *Linear-Speed Vertex Cache Optimisation*. http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html

[66] FORSYTH Tom. *My own little DirectX FAQ*. http://tomsdxfaq.blogspot.com/

[67] FOURNIER A, FUSSEL D, CARPENTER L. *Computer Rendering of Stochastic Models*. Comunications of the ACM. Junho 1982. http://accad.osu.edu/~waynec/history/PDFs/p371-fournier.pdf

[68] FOWLER, R. J., LITTLE J. J. *Automatic Extraction of Irregular Network Digital Terrain Models*. Proceedings of SIGGRAPH 79. Pp. 199-207. 1979

[69] FRAME Michael, MANDELBROT Benoit, NEGER Nial. *Fractal Geometry*. Yale University. http://classes.yale.edu/fractals/

[70] FUCHS H., ABRAM G. D., e GRANT E. D. *Near real-time shaded display of rigid objects*. Computer Graphics (Proceedings of SIGGRAPH 83), 17(3):65–72, July 1983.
[71] FUCHS H., KEDEM Z. M. e NAYLOR Bruce F. *On visible surface generation by a priori tree structures*. Computer Graphics (Proceedings of SIGGRAPH 80), 14(3):124–133, July 1980.
[72] GINSBURG Dan. *Octree Construction*. Game Programming Gems. Publicado por Charles River Média Inc. 2000. ISBN: 1584500492.
[73] GAREY M. R. e JOHNSON D. S. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman e Co., San Francisco. 1979.
[74] GARNEY Ben. *Clipmapping on SM1.1 and Higher*. Game Programming Gems 7. Publicado por Charles River Média Inc. 2008. ISBN: 1584505273.
[75] GEE Kev. *Direct3D 11 Tessellation*. Gamefest. 2008.
[76] GERASIMOV Philipp, FERNANDO Randima (Randy), GREEN Simon. *Shader Model 3.0 - Using Vertex Textures*. NVIDIA Corporation. ftp://download.nvidia.com/developer/Papers/2004/Vertex_Textures/Vertex_Textures.pdf
[77] GOKHALE Bhushan, DUNSFORD Ted. Principles of GIS. GEOL 403/505. 2006. http://giscenter.isu.edu/training/ppt/Principles/Lecture1_BG.ppt
[78] GOOGLE. *Google Earth*. http://earth.google.com/.
[79] GORE Al. The Digital Earth: Understanding our planet in the 21st Century. Vice-presidente dos E.U.A. http://www.digitalearth.gov/VP19980131.html.
[80] GREENE Ned. *Hierarchical Rendering of Complex Environments*. Ph.D Thesis, Universidade da California. Junho 1995.
[81] GREENE Ned, KASS Michael e MILLER Gavin. *Hierarchical Z-Buffer Visibility*. Computer Graphics (Proceedings of SIGGRAPH 93), pp. 231–238. Agosto 1993. http://www.cs.princeton.edu/courses/archive/fall02/cs526/papers/greene93.pdf
[82] GROME. http://www.quadsoftware.com
[83] GUSTAVSON Stefan. *Simplex Noise Demystified*. Linköping University, Sweden. 2005. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
[84] HADWIGER Markus, VARGA Andreas. *Visibility Culling*. http://www.cg.tuwien.ac.at/~msh/viscull.pdf.
[85] HAYES Jeremy. *Advanced Particle Deposition*. Game Programming Gems 7. Publicado por Charles River Média Inc. 2008. ISBN: 1584505273.
[86] HE Yefei. *Real-Time Visualization of Dynamic Terrain for Ground Vehicle Simulation*. Ph.D. Thesis, Department of Computer Science, University of Iowa. 2000.
[87] HE Yefei, CREMER James, PAPELIS Yiannis. *Real-time Extendible-resolution Display of On-line Dynamic Terrain*. http://www.graphicsinterface.org/cgi-bin/DownloadPaper?name=2002/139/paper139.pdf
[88] HECKBERT Paul S. e GARLAND Michael. *Survey of Polygonal Surface Simplification Algorithms*. Multiresolution Surface Modeling Course, SIGGRAPH 1997. http://graphics.cs.uiuc.edu/~garland/papers/simp.pdf
[89] HILL Dave. *An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains*. Dissertation submitted for the Graduate Department

of Computer Science of the University of Toronto, supervised by Alejo Hausner. 2002.
[90] HOPPE Hugues. *Smooth View-Dependent Level-of-Detail Control and its application to Terrain Rendering*. Proceedings IEEE Visualization 98. pp. 99-108. 1998. http://research.microsoft.com/~hoppe/svdlod.pdf.
[91] HOPPE Hugues. *Optimization of mesh locality for transparent vertex caching*. Proceedings ACM SIGGRAPH 1999, 296-276.
[92] HUNTER G. M. *Efficient Computation and Data Structures for Graphics*. Ph. D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.
[93] INFMÉDIA. *In Volo Sull' Isola d'Elba*. http://www.rete.toscana.it/sett/pta/cartografia_sit/sit/elba3d/it/start.htm
[94] INVERSE. *Focus on Terrain Generation*. http://www.ziggyware.com/readarticle.php?article_id=132
[95] JACKINS C. e TANIMOTO S. L. *Oct-trees and their use in representing three-dimensional objects*. Comput. Gr. Image Process. 14, 3 (Nov.), 249-270.1980
[96] JFP INC. *Virtual Garden Walkthrough*. http://www.jfp.co.jp/walk/index_e.htm
[97] KILGARD, M. J. *A Practical and Robust Bump-mapping Technique for Today's GPUs*. Game Developers Conference, Advanced OpenGL Game Development. 2000.
[98] KLEIN Allison. *Introduction to the Direct3D 11 Graphics Pipeline*. Gamefest. 2008.
[99] KNOWLTON, K. 1980. *Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes*. Proc. IEEE 68, 7 (July), 885 896.
[100] KOBBELT Leif. *Interpolatory Subdivision on Open Quadrilateral Nets with Arbitrary Topology*. Department of Computer Sciences University of Wisconsin. 1995. http://kucg.korea.ac.kr/education/2003_2/vip618/paper/interpolatory_quad.pdf
[101] KOCH H. Von. *Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire*. 1904.
[102] KRTEN, R. (1994). *Generating Realistic Terrain*. Dr Dobbs Journal: Software Tools for the Professional Programmer, 19(7), 26-28. http://www.parse.com/~rk/articles/fractal.html
[103] L3DT. Large 3D Terrain Generator. http://www.bundysoft.com/L3DT/
[104] LAEUCHLI Jesse. *Programming Fractals*. Game Programming Gems 2. Publicado por Charles River Média Inc. 2001. ISBN: 1584500549.
[105] LLAMAS I., THIBIEROZ N.. DirectX 10 Performance. NVIDIA/AMD. Presentation at GDC 2008. http://developer.download.nvidia.com/presentations/2008/GDC/GDC08-D3DDay-Performance.pdf
[106] LAMBERT, J. H. *Photometria sive de mensure de gratibus luminis colorum umbrae*. 1760.
[107] LAURITSEN Thomas, NIELSEN Steen Lund. *Rendering Very Large, Very Detailed Terrains*. Abril 2005. http://www.terrain.dk/terrain.pdf.

[108] LEVENBERG, J. *Fast view-dependent level-of-detail rendering using cached geometry*. IEEE Visualization 2002, 259-266. http://www.technomagi.com/josh/vis2002/dlodtalk.pdf
[109] LEWIS J. P. *Algorithms for solid noise synthesis*. Proceeedings SIGGRAPH' 89. 1989. http://www.idiom.com/~zilla/Work/Crunge/crunge.pdf
[110] LINDSTROM Peter, KOLLER David, RIBARSKY William, HODGES Larry F., FAUST Nick e TURNER Gregory A. *Real-Time, Contínuous Level of Detail Rendering of Height Fields*. Proceedings ACM SIGGRAPH' 96. pp. 109-118. 1996. http://www.cc.gatech.edu/gvu/people/peter.lindstrom/.
[111] LINDSTROM Peter, PASCUCCI V. *Terrain simplification simplified: A general framework for view-dependent out-of-core visualization*. IEEE TVCG 8(3), 239-254. 2002. http://www.pascucci.org/pdf-papers/IEEE-TVG-2002.pdf
[112] LINDSTROM Peter, PASCUCCI V. *Visualization of Large Terrains Made Easy*. Proceedings IEEE Visualization 2001. pp. 363-370 e 574. 2001. http://www-static.cc.gatech.edu/~lindstro/papers/vis2001a/paper.pdf
[113] LIONHEAD. *Black & White*. http://www.lionhead.com/bw2/
[114] LISBERGER Steven. *Tron*. 1982. http://www.imdb.com/title/tt0084827/
[115] LLOYD Brandon EGBERT Parris. *Horizon Occlusion Cullingfor Real-time Rendering of Hierarchical Terrains*. Proceedings of the conference on Visualization '02. 2002. http://rivit.cs.byu.edu/a3dg/publications/horizonCullingVISFinal.pdf
[116] LONGBOW DIGITAL ARTS. *Tread Marks*. http://www.ldagames.com/treadmarks/
[117] LUEBKE David. *A Developer's Survey of Polygonal Simplification Algorithms*. University of Virginia. 2001. http://www.cs.virginia.edu/~luebke/publications/pdf/cg+a.2001.pdf
[118] LUEBKE David. *Level of Detail & Visibility: A Brief Overview*. Course 446: Real-Time Rendering. University of Virginia. 2004. http://www.cs.virginia.edu/~gfx/Courses/2004/RealTime/lecture06.LOD1.ppt
[119] LUEBKE David, REDDY Martin, COHEN Jonathan D, VARSHNEY Amitabh, WATSON Benjamin, HUEBNER Robert. Level of Detail for 3D Graphics. ISBN: 1558608389. http://lodbook.com/
[120] LOSASSO Frank, HOPPE Hugues. *Geometry clipmaps: Terrain rendering using nested regular grids*. Proceedings ACM SIGGRAPH' 2004. pp. 769-776. 2004. http://research.microsoft.com/~hoppe/gpugcm.pdf, http://research.microsoft.com/~hoppe/geomclipmap.ppt.
[121] MACRI Dean, PALLISTER Kim. *Procedural 3D Content Generator - Part 1 of 2*. http://www.intel.com/cd/ids/developer/asmo-na/eng/segments/games/resources/graphics/20247.htm
[122] MACRI Dean, PALLISTER Kim. *Procedural 3D Content Generator - Part 2 of 2*. http://www.intel.com/cd/ids/developer/asmo-na/eng/segments/games/resources/graphics/20246.htm
[123] MALVAR, Henrique. *Fast Progressive Image Coding without Wavelets*. Data Compression Conference (DCC '00), pp. 243–252. 2000. http://research.microsoft.com/~malvar/papers/ptc_dcc00.pdf

[124] MANDELBROT Benoit B. *The Fractal Geometry of Nature*. New York: W.H. Feeman. 1977.
[125] MARTZ Paul. *Generating Random Fractal Terrain*. http://www.gameprogrammer.com/fractal.html.
[126] MAXIS. *Spore*. http://www.spore.com/
[127] MCNALLY Seumas. *Binary Triangle Trees and Terrain Tessellation*. Seumas's Programming Page. http://www.ldagames.com/seumas/progbintri.html.
[128] MCNALLY Seumas. *Class: Two Advanced Terrain Rendering Systems - The Tread Marks Engine*. Longbow Digital Arts Incorporated. http://www.gamasutra.com/features/gdcarchive/2000/mcnally.doc
[129] MCNALLY Seumas. *The Tread Marks Engine - Real Time View-dependent LOD Terrain With Dynamic Modifiability*. Longbow Digital Arts Incorporated. http://www.gamasutra.com/features/gdcarchive/2000/mcnally.ppt
[130] MELODY. Nvidia Melody. http://developer.nvidia.com/object/melody_home.html
[131] METZE Joscha. *Optimising the visual appearance of a virtual reality simulation by using modern graphics cards features*. Dissertation submitted for the Institut für Software- und Multimédiatechnik Fakultät Informatik Technische Universität Dresden, California supervised by Markus Wacker. Setembro 2005. http://www.inf.tu-dresden.de/content/institutes/smt/cg/results/majorthesis/jmetze/files/MasterThesis_VisualAppearance.pdf
[132] MICROSOFT. *Xbox 360 Technical Specifications*. http://www.xbox.com/en-AU/support/xbox360/manuals/xbox360specs.htm
[133] MILLER G. *The definition and rendering of terrain maps*. Proceedings SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986) em Computer Graphics, 20, 4 (Agosto 1986), páginas 39–48, ACM SIGGRAPH, New York, 1986.
[134] MEAGHER D. *Geometric modeling using octree encoding*. Comput. Gr. Image Process. 19, 2(June), 129-147. 1982
[135] MÖLLER Thomas Akenine, HAINES Eric. *Real-Time Rendering*. Publicado por AK Peters. ISBN: 2002025151.
[136] MÖLLER Thomas Akenine. *Spatial Data Structures and Speed-Up Techniques*. Department of Computer Engineering, Chalmers University of Technology. http://www.ce.chalmers.se/edu/year/2004/course/EDA425/course2003/lectures/spatial.pdf
[137] MOORE Gordon E. *Cramming more components onto integrated circuits*.ftp://download.intel.com/research/silicon/moorespaper.pdf.
[138] MORTENSEN Jesper. *Real-time rendering of height fields using LOD and occlusion culling*. http://www.cs.ucl.ac.uk/teaching/vive/PastProjects/Mortensen2000.pdf
[139] MUSGRAVE Forest Kenton. *Methods for Realistic Landscape Imaging*. Yale University. In Candidacy for the Degree of. Doctor of Philosophy. 1993. http://www.kenmusgrave.com/dissertation.pdf
[140] MUSGRAVE Forest Kenton, KOLB Craig E. e MACE Robert S. *The Synthesis and Rendering of Eroded Fractal Terrains*. Proceedings of SIGGRAPH '89.1989. http://ftp.funet.fi/pub/graphics/papers/papers/musg89b.ps.Z
[141] *National Geospatial-Intelligence Agency (NIMA)*. http://www.nima.mil

[142] *National Mapping Program Standarts*. http://rockyweb.cr.usgs.gov/nmpstds/demstds.html
[143] *NCGIA Core Curriculum*. Versão de 1990. http://www.geog.ubc.ca/courses/klink/gis.notes/ncgia/toc.html
[144] NIMA. http://www.nima.mil
[145] NYSTROM Bob. *Hill Algorithm*. http://www.robot-frog.com/3d/hills/index.html.
[146] NVIDIA. *NV_primitive_restart*. http://www.opengl.org/registry/specs/NV/primitive_restart.txt.
[147] OGRE. *Object-Oriented Graphics Rendering Engine* (OGRE). http://www.ogre3d.org
[148] ÖGREN Andreas. C*ontínuous Level of Detail in Real-Time Rendering*. Master Thesis. 2000.
[149] O'NEIL Sean. *A Real Time Procedural Universe, Part 1: Generating Planetary Bodies*. http://www.gamasutra.com/features/20010302/oneil_01.htm
[150] OpenGL. http://www.opengl.org/
[151] PAJAROLA, R. *Large scale terrain visualization using the restricted quadtree triangulation*. IEEE Visualization 1998, 19-26. http://www.ifi.unizh.ch/vmml/admin/upload/Vis98.pdf
[152] PANDROMEDA. *MojoWorld*. http://www.pandromeda.com/products/
[153] PANGERL David. *Quantized Ring Clipping*. ShaderX 6. Publicado por Charles River Média Inc. ISBN: 1584505443.
[154] PEDRINI Hélio. *Multiresolution Terrain Modeling Based On Triangulated Irregular Networks*. http://www.sbgeo.org.br/rgb/vol31_down/3102/3102117.pdf
[155] PEITGEN Heinz-Otto, SAUPE Dietmar. The Science of Fractal Images. Springer-Verlag New York, Inc. ISBN: 0387966080. 1988.
[156] *Performance Specification Digital Terrain Elevation Data*. http://earth-info.nga.mil/publications/specs/printed/89020B/89020B.pdf
[157] *Perlin Noise and it's fractal nature*. http://www.animeimaging.com/asp/PerlinNoise.aspx
[158] PERLIN Ken. *An Image Synthesizer*. In Proceedings of SIGGRAPH 85, pp. 287–296. 1985.
[159] PERLIN Ken. *Improving Noise*. Computer Graphics Vol. 35 No. 3. http://mrl.nyu.edu/~perlin/paper445.pdf
[160] PERLIN Ken. *Making Noise*. Talk presented at GDCHardCore. 1999. http://www.noisemachine.com/talk1/index.html
[161] PERLIN Ken. *Noise and Turbulence*. http://mrl.nyu.edu/~perlin/doc/oscar.html
[162] PIRES Hugo Castelo. *Visualização de Terrenos em Tempo Real*. Dissertação submetida à Universidade do Minho para obtenção do grau de Mestre em Informática sob orientação do Prof. António José Borba Ramires Fernandes. Janeiro 2004. http://sim.di.uminho.pt/mestrados/hugopires/Dissertacao.pdf.
[163] PIRES Pedro. *Dynamic Algorithm Binding for Virtual Walkthroughs*. Instituto Superior Técnico. Novembro 2001. http://paginas.fe.up.pt/~aas/pub/Aulas/RVA/MaterialConsulta/AcelRendering.pdf.
[164] PLANETSIDE SOFTWARE. *Terragen*. http://www.planetside.co.uk/terragen/
[165] POLACK Trent, LAMOTHE André. *Focus on 3D Terrain Programming*. Publicado por Muska & Lipman/Premier-Trade. ISBN: 1592000282.

[166] POMERANZ Alex A. *ROAM Using Surface Triangle Clusters (RUSTiC)*. M.S. Thesis, Department of Computer Science: University of California (Davis). 1998. http://www.cognigraph.com/ROAM_homepage/PomeranzThesis.pdf
[167] PRICE Alan. *Frustum culling implications*. http://accad.osu.edu/~aprice/courses/BVE/frustum/frustum.html
[168] PRO FX. http://www.profxengine.com/
[169] RABINOVICH B. e GOTSAMN C. *Visualization of Large Terrains in Resource-Limited Environments*. IEEE Visualization. 1997.
[170] REDDY D. R. e RUBIN S. *Representation of three-dimensional objects.* CMU-CS-78-113, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Apr. 1978.
[171] RHEINGANS Penny. *Viewing*. CMSC 435 Introduction to Computer Graphics. http://www.cs.umbc.edu/~rheingan/435/pages/res/gen-8.Viewing-single-page-0.html
[172] RÖTTGER Stefan, HEIDRICH Wolfgang, SLUSALLEK Philipp, e SEIDEL Hans- Peter. *Real-Time Generation of Contínuous Levels of Detail for Height Fields*. Proceedings WSCG '98, Skala (editor), pp. 315–322. 1998. http://wwwvis.informatik.uni-stuttgart.de/~roettger/data/Papers/TERRAIN.PDF.
[173] SAUPE D. *Point Evaluation of Multi-Variable Random Fractals*. Visualisierung in Mathematik und Naturissenschaft - Bremer Computergraphik Tage. Springer-Verlag, Heidelberg. 1989.
[174] SAMET, Hanan e TAMMINEN M. *Efficient image component labeling*. TR-1420, Computer Science Dept., University of Maryland, College Park, Md., July. 1984.
[175] SAMET Hanan. Quadtree Background. Computer Science Department and Center for Automation Research and Institute for Advanced Computer Studies University of Maryland College Park, Maryland 20742. 1996.
[176] SAMET Hanan. *The quadtree and related hierarchical data structure*. ACM Computing Surveys,16(2):187–260, 1984. http://www.cs.umd.edu/class/spring2005/cmsc828s/slides/quad.anim.pdf
[177] SCARLATOS, L., PAVLIDIS, T. *Hierarchical Triangulation Using Cartographic Coherence*. CVGIP: Graphical Models and Image Processing. Vol. 54(2). Pp. 147-161. 1992.
[178] SCHNEIDER Jens e WESTERMANN Rüdiger. *GPU-Friendly High-Quality Terrain Rendering*. 2006. http://wwwcg.in.tum.de/Research/data/Publications/wscg06.pdf
[179] SCOTT Noel D., OLSEN Daniel M. e GANNETT Ethan W. *An Overview of the VISUALIZE fx Graphics Accelerator Hardware*. Hewlett-Packard Journal, pp. 28-34. Maio 1998. http://www.hpl.hp.com/hpjournal/98may/may98a4.pdf
[180] SHANKEL Jason. *Fractal Terrain Generation – Fault Formation*. Game Programming Gems. Publicado por Charles River Média Inc. 2000. ISBN: 1584500492.
[181] SHANKEL Jason. *Fractal Terrain Generation – Midpoint Displacement*. Game Programming Gems. Publicado por Charles River Média Inc. 2000.ISBN: 1584500492.

[182] SHANKEL Jason. *Fractal Terrain Generation – Particle Deposition*. Game Programming Gems. Publicado por Charles River Média Inc. 2000. ISBN: 1584500492.
[183] SHEN Han-Wei. *Visibility Culling*. Department of Computer Science and Engineering, The Ohio State University. 2001. http://www.cse.ohio-state.edu/~hwshen/788/sp01/Culling.ppt
[184] SLAYTON Joe. Wilbur. http://www.ridgecrest.ca.us/~jslayton/*software*.html
[185] SNOOK Greg. *Real Time 3D Terrain Engines using C++ and DirectX 9.0.* Publicado por Charles River Média. ISBN: 1584502045.
[186] SNOOK Greg. *Simplified Terrain Using Interlocking Terrain Tiles*. Game Programming Gems 2. Publicado por Charles River Média Inc. 2001.ISBN: 1584500549.
[187] SOBEL, I., Feldman,G. *A 3x3 Isotropic Gradient Operator for Image Processing*, presented at a talk at the Stanford Artificial Project in 1968. Unpublished.
[188] STEWART James A.*Hierarchical Visibility In Terrains*. Eurographics Rendering Workshop. Junho 1997. http://www.cs.queensu.ca/home/jstewart/papers/egwr97.pdf
[189] SU P. e DRYSDALE L. S. *A Comparison of Sequential Delaunay Triangulation Algorithms*. Proceedings 11th Annual Symposium on Computational Geometry, páginas 61-70, Vancouver, CANADA, 1995. http://www.cs.berkeley.edu/~jrs/meshpapers/SuDrysdale.ps.gz
[190] TAMMINEN, M. *Comment on Quad- and Oct- trees*. Commun. ACM 27, 3 (Mar.) 248-249. 1984.
[191] TANNER, Christopher C., MIGDAL Christopher J. e JONES Michael T. 1998. *The Clipmap: A Virtual Mipmap*. In Computer Graphics (Proceedings of SIGGRAPH 98), pp. 151–158. http://www.cs.virginia.edu/~gfx/Courses/2002/BigData/papers/Texturing/Clipmap.pdf
[192] TARBOTON David G. *Spatial Fields in GIS and Hydrology*. Utah State University. http://www.ce.utexas.edu/prof/maidment/giswr2003/visual/spatial.ppt.
[193] TELLER Seth J e HANRAHAN Pat.*Global Visibility Algorithms for Illumination Computations*. Proceedings SIGGGRAPH ’94 pp. 443-450. Julho 1994.
[194] TELLER Seth J. *Visibility Computations in Densely Occluded Polyhedral Environments*. Ph.D Thesis. Department of Computer Science, University of Berkeley. 1992.
[195] TELLER Seth J., SÉQUIN Carlos H.*Visibility Preprocessing for Interactive Walkthroughs*. Proceedings SIGGGRAPH ’91 pp. 61-69. 1991.
[196] THE DAYLON LEVELLER. *http://www.daylongraphics.com/products/leveller/*
[197] TORCHELSEN Rafael P.,COMBA João L. D., BASTOS Rui. *Practical Geometry Clipmaps for Rendering Terrains in Computer Games*. ShaderX 6. Publicado por Charles River Média Inc. ISBN: 1584505443.
[198] TURNER Brian. *Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*. http://www.gamasutra.com/features/20000403/turner_01.htm.
[199] TYLER Marcus Craig. A thesis submitted to the faculty of The University of North Carolina at Charlotte in partial fulfillment of the requirements for the degree

of Master of Science in the Department of Computer Science. 2007. http://www.cs.uncc.edu/~krs/theses/07/craig_tyler_ms.pdf
[200] ULRICH Thatcher. *Rendering Massive Terrains Using Chunked Level of Detail Control*. Abril 2002. http://tulrich.com/geekstuff/sig-notes.pdf.
[201] ULRICH Thatcher. *Super-Size It! Rendering Massive Terrains*. SIGGRAPH'02 Course.
[202] USGS. http://www.usgs.gov/
[203] USGS. *GTOP30*. http://edc.usgs.gov/products/elevation/gtopo30/gtopo30.html.
[204] USGS. *Shuttle Radar Topography Mission*. http://srtm.usgs.gov/.
[205] UNIVERSITY OF LEICESTER. *Virtual Field Course*. http://www.geog.le.ac.uk/vfc/index.html
[206] VANEK Jan, Jezek Bruno. Real-Time Terrain Visualization on PC. Proceedings the 12th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2004. http://wscg.zcu.cz/wscg2004/Papers_2004_Poster/P89.pdf
[207] VISTNES Harald. *GPU Terrain Rendering*. Game Programming Gems 6. Publicado por Charles River Média Inc. 2006. ISBN: 1584504501.
[208] WAGNER Daniel. *Terrain* Geomorphing *in the Vertex Shader*. ShaderX2 - Shader Programming Tips and Tricks. Publicado por Wordware Publishing, Inc. ISBN: 1556229887. http://www.ims.tuwien.ac.at/média/documents/publications/Terrain_Geomorphing_in_the_Vertex_Shader.pdf
[209] WAHL R. et al. *Scalable Compression and Rendering of Textured Terrain Data*. In Journal of WSCG vol. 12, 2004. http://www.cg.cs.uni-bonn.de/docs/publications/2004/wahl-2004-scalable.pdf
[210] WARD Matthew. *Fractional brownian motion (fBm)*. http://davis.wpi.edu/~matt/courses/fractals/brownian.html
[211] WEIBEL R, e JONES C B. *Special Issue on Automated Map Generalization. GeoInformatica*. vol. 2(4). 1998.
[212] WELLS William David. *Generating Enhanced Natural Environments and Terrain for Interactive Combat Simulations (GENETICS)*.Dissertation submitted for the Naval Postgraduate School of Monterey, California supervised by Rudolph Darken. September 2005. http://www.movesinstitute.org/~wdwells/GENETICS%20Dissertation.pdf
[213] WILLIAMS, Lance. *Pyramidal Parametrics*. Computer Graphics, vol. 7, no 3, pp.1.11, Julho 1983.
[214] WIMMER Michael e BITTNER Jiri. *Hardware Occlusion Queries Made Useful*. GPU Gems 2. Publicado por Addison-Wesley. 2005. ISBN: 0321335597.
[215] WLOKA Matthias. *Improved Batching via Texture Atlases*. ShaderX 3. Publicado por Charles River Média Inc. ISBN: 1584503572.
[216] WONKA Peter, SHMALSTIEG Dieter. *Occluder Shadows for Fast Walkthroughs of Urban Environments*.Computer Graphics Forum, vol18, no. 3, pp. 51-60. 1999. http://www.cg.tuwien.ac.at/research/vr/occshadow/occshadows_eg99.pdf
[217] WHORLEY Hazel. *Skybox Texture Pack 1, 2 e 3*. http://www.hazelwhorley.com/textures.html

[218] WORLEY S. *A Cellular Texture Basis Function*. Proceedings of SIGGRAPH '96. 1996.
[219] WORLD MACHINE. http://www.world-machine.com/
[220] XNA. http://creators.xna.com/
[221] YANG T., WÜNSCHE B. C., LOBB R. J. *Game Engine Support for Terrain Rendering in Architectural Design*. Graphics Group, Dept. of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand. http://www.cs.auckland.ac.nz/~burkhard/Publications/IVCNZ04_YangWuenscheLobb.pdf
[222] YOUBING Z, JI Z., JIAOYING S.,ZHIGENG P. A Fast Algorithm for Large- Scale Terrain Walkthrough. Proceedings of CAD/Graphics 2001. http://www.cad.zju.edu.cn/home/zhaoyb/paper/2001terranwalk_release.pdf
[223] YU Jingyi. *Clipping & Culling*. Lecture 12 CISC440/640 2006. http://www.cis.udel.edu/~yu/Teaching/CISC440_06F/handouts/Lecture12.pdf
[224] ZACHMANN Gabriel, LANGETEPE Elmar. *Geometric Data Structures for Computer Graphics*. http://www.cg.cs.unibonn.de/docs/publications/2003/zachmann-2003-geometric.pdf
[225] ZHANG Hansong. *Effective Occlusion Cullingfor Interactive Display of Arbitrary Models*. Ph.D Thesis, Department of Computer Science, University of North Carolina at Chapel Hill. Julho 1998. http://www.cs.unc.edu/~zhangh/dissertation.pdf
[226] ZHANG Hansong, MANOCHA Dinesh, HUDSON T., HOFF III K.E. *Visibility Culling using Hierarchical Occlusion Maps*. Computer Graphics (SIGGRAPH 97 Proceedings), pp. 77-88, Agosto 1997.
[227] ZAUGG Brian, Egbert Parris K. *Voxel Column Culling: Occlusion CullingFor Large Terrain Models*. http://rivit.cs.byu.edu/a3dg/publications/Voxel_Column_Culling.pdf
[228] ZUCKER Matt. *The Perlin Noise Math FAQ*. http://www.cs.cmu.edu/~mzucker/code/perlin-noise-math-faq.html