PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Diogo Strube de Lima

Henry Braun

EXIBIÇÃO DE TERRENOS EM TEMPO REAL

UMA ABORDAGEM A TERRENOS COM LARGA ESCALA GEOMÉTRICA

Porto Alegre

2008

Diogo Strube de Lima

Henry Braun

EXIBIÇÃO DE TERRENOS EM TEMPO REAL

UMA ABORDAGEM A TERRENOS COM LARGA ESCALA GEOMÉTRICA

Trabalho de Conclusão dos cursos de graduação, Ciência da Computação e Sistemas de Informação, apresentado à Faculdade de Informática da Pontifica Universidade Católica do Rio Grande do Sul, como requisito parcial na obtenção do grau de Bacharelado em ambos os cursos.

Orientador: Márcio Sarroglia Pinho

Porto Alegre

2008

Dedicamos esta pesquisa à comunidade de desenvolvedores de jogos pelo empenho constante na busca de melhores soluções.

AGRADECIMENTOS

Agradecemos ao Professor Doutor Marcio Sarroglia Pinho por ter nos orientado ao longo deste ano de forma atenciosa e dedicada. Agradecemos também à Professora Doutora Soraia Raupp Musse pela oportunidade de estágio, necessária para a realização do Estágio Profissional, e no suporte de Marcelo Paravisi e Rafael Rodrigues.

Gostaríamos de mencionar a Professora Doutora Vera Lúcia Strube de Lima, o Engenheiro Harold Braun e Norma Braun, pois nos auxiliaram em momentos cruciais do desenvolvimento deste documento. Por fim, gostaríamos de agradecer à Cristiane Bellenzier Piaia e Gabriela Cerveira pela compreensão que tiveram até a conclusão deste trabalho.

There must be a beginning of any great matter, but the continuing unto the end until it be thoroughly finished yields the true glory.

Sir Francis Drake

RESUMO

O crescimento na indústria de jogos de computador ocasionou no aumento da complexidade dos mesmos. Estes jogos podem ser divididos em diversos módulos que realizam, por exemplo, os cálculos de física, a inteligência artificial de personagens e a exibição dos terrenos. Destaca-se no desenvolvimento de jogos a necessidade de exibição do terreno em tempo real e com boa qualidade gráfica. O presente trabalho consiste na análise de soluções comuns para a exibição de terrenos de larga escala em tempo real. Esta análise apresenta como resultado uma proposta que visa viabilizar esta visualização além de proporcionar uma qualidade gráfica adequada.

Palavras-chave: Exibição de Terreno de Larga Escala em Tempo Real.

ABSTRACT

The growth of the game industry led to the ever increasing complexity of the computer games. These games can be divided into different modules, like physics, artificial intelligence and terrain rendering. Of relevance is the need of real-time terrain rendering with good graphics quality. This paperwork analyses the current solutions for the real-time large scale terrain rendering. The analysis presents a proposal that allows terrain visualizations with appropriate graphic quality.

Keywords: Real-Time Large Scale Terrain Rendering.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

2D	Bidimensional
3D	Tridimensional
AGP	*Accelerated Graphics Port*
API	*Application Program Interface*
CG	Computação Gráfica
CLOD	*Continuous Level-Of-Detail*
DC	*Drawcalls*
EOM	Espaço Ocupado de Memória
ESA	*Entertainment Software Association*
FPS	*Frames per Second*
GIS	*Geographic Information System*
GLSL	*GL Shading Language*
GPU	*Graphic Processing Units*
HLSL	*High Level Shading Language*
IFPI	*International Federation of the Phonographic Industry*
IGN	*Imagine Games Network*
KUP	*Keep Under Dev*
LOD	*Level-of-Detail*
LST	*Large Scale Terrains*
MPAA	*Motion Picture Association of America*
NTE	Número de Triângulos Exibidos
NZE	Número de Zonas Exibidas
NZM	Número de Zonas em Memória
PCIe	*Peripheral Component Interconnect Express*
PWC	Pricewaterhouse Coopers

PGF *Pipeline* Gráfico Fixo

PGP *Pipeline* Gráfico Programável

RIAA *Recording Industry Association of America*

QPS Quadros Por Segundo

SUMÁRIO

1. Introdução

O mercado de jogos encontra-se em alto crescimento, confirmando-se como foco para novas pesquisas e investimentos, conforme citado no relatório *Global Entertainment and Media Outlook:* 2007-2011 da empresa de consultoria *Pricewaterhouse Coopers* (PWC) [1] e destacado em outras fontes de comunicação [2] [3].

Segundo a mesma consultoria, a indústria de *videogames* está pronta para ultrapassar a indústria de música. O relatório indica que a indústria de jogos terá um crescimento de 9,1% entre 2007 e 2011, resultando num mercado de US$ 48,9 bilhões em 2011, contra US$ 37,5 bilhões em 2007.

Os jogos também estão movimentando mais dinheiro que a indústria de cinema. De acordo com a **Figura 1**, obtida de Bangeman [4] e gerada a partir de dados provenientes das fontes: *NPD Group*, *Entertainment Software Association* (ESA), *Motion Picture Association of America* (MPAA), *Recording Industry Association of America* (RIAA) e *Internacional Federation of the Phonographic Industry* (IFPI) é possível observar o crescimento da arrecadação, ao longo de cinco anos (2002-2007), das três indústrias de entretenimento: cinema, música e jogos.

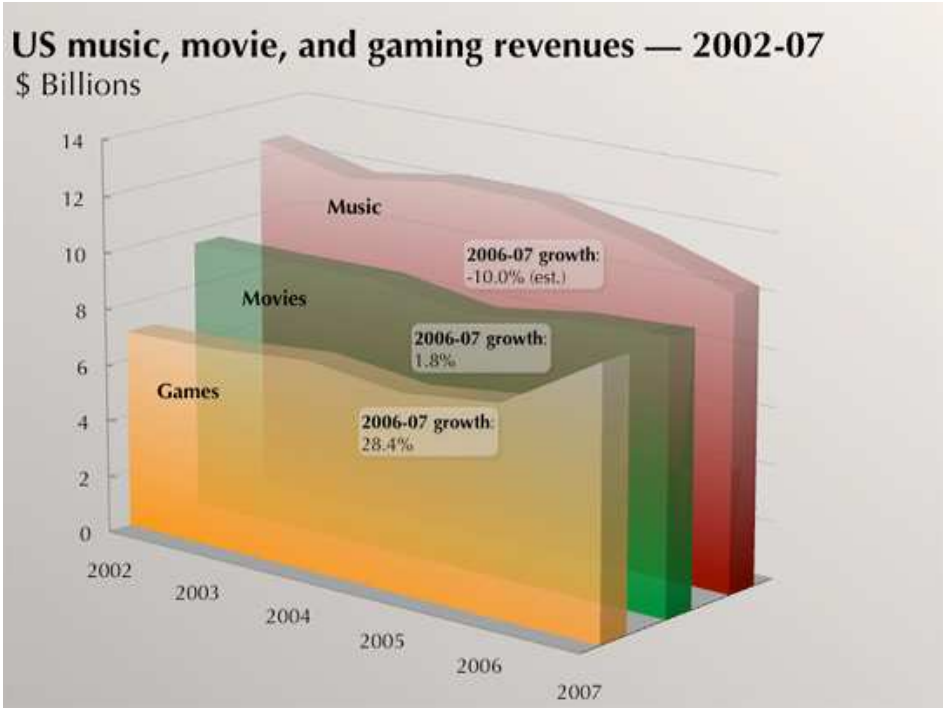


Figura 1 – Arrecadação das indústrias de entretenimento.

Um jogo de computador pode ser avaliado de diferentes maneiras, tais como o lucro obtido no mercado, ou mesmo, a classificação realizada por comunidades de jogadores. Estas comunidades, como a *Imagine Games Network* (IGN)[1], GameSpot[2] e Outer Space[3], possuem credibilidade no mercado, contando com até 31 milhões de usuários conectados mensalmente em busca de análises (*reviews*) disponíveis sobre os jogos.

As análises realizadas pelas comunidades possuem quesitos comuns, como qualidade gráfica e jogabilidade (*gameplay*). Conforme a metodologia de avaliação da IGN, por exemplo, o quesito gráfico consiste na qualidade com que os componentes do jogo são representados visualmente, como a qualidade de texturas e animações. As técnicas utilizadas e o desempenho apresentado em máquinas de teste também fazem parte desta categoria. A jogabilidade, segundo a mesma metodologia, envolve o nível de interação e diversão que o usuário cria com o jogo.

As avaliações realizadas pelas comunidades e as expectativas dos consumidores, auxiliam os estúdios de jogos de computador na criação de produtos que visam alcançar um grande público [5]. A busca pelo público influenciou na melhoria de aspectos como a qualidade gráfica e a jogabilidade, obtidas a partir de inovações, por exemplo, nas técnicas de exibição e no *hardware* dos computadores. Estas inovações, visando à jogabilidade, permitiram mudanças como o aumento de tamanho dos cenários e a maior liberdade de movimentação que o jogador possui nos mesmos [6].

Através das inovações nas técnicas de exibição tornou-se possível a melhoria dos ambientes gráficos como, por exemplo, a dos cenários ao ar livre (*outdoor environments*). Nestes casos os terrenos são um componente fundamental, pois estão presentes na maioria dos momentos de interação do usuário com o jogo. Nos casos em que estes terrenos têm um nível de

---

[1] http://www.ign.com

[2] http://www.gamespot.com

[3] http://www.outerspace.com.br

detalhamento (número de polígonos) muito elevado para serem exibidos em tempo real, em uma dada configuração de máquina, os mesmos são conhecidos como *large scale terrains* (LST). Para exibir os LSTs em tempo-real, existem técnicas específicas, conhecidas como *real-time large scale terrain rendering*. A exibição deste tipo de terreno é o objetivo principal deste trabalho.

1.1 Motivação

Os LSTs têm aplicação em áreas como a indústria cinematográfica, simulação, realidade virtual e cartografia, além de possuírem relevância nos quesitos que demonstram a qualidade gráfica, o realismo e a jogabilidade dos jogos de computador. A **Figura 2**, por exemplo, apresenta imagens dos jogos Guild Wars[4] (a), Test Drive Unlimited[5] (b) e Flight Simulator X[6] (c), nas quais é possível visualizar cenários ao ar livre, e perceber que o terreno constitui uma grande parte do campo visual do usuário.

(a) **(b)** **(c)**

Figura 2 - Exemplo de jogos de computador.

As áreas que utilizam os LSTs seguidamente requerem um método de visualização em tempo real, como é o caso dos jogos de computador e da simulação em realidade virtual. Para que tal requisito seja alcançado, técnicas

[4] http://www.guildwars.com/

[5] http://www.testdriveunlimited.com/

[6] http://www.microsoft.com/games/pc/flightsimulatorx.aspx

de exibição de terrenos em tempo real devem solucionar problemas derivados, em sua maioria, da grande quantidade de polígonos representando a geometria do terreno [7].

As técnicas de exibição de LSTs envolvem diferentes tópicos da computação como: otimização de algoritmos, gerência de memória, processamento paralelo, computação gráfica (CG) entre outros. Estes diferentes tópicos tornam a implementação destes terrenos em tempo real um desafio instigante.

1.2 Objetivos

O principal objetivo deste trabalho consiste na criação de um algoritmo de exibição de LSTs em tempo real. Este deve permitir a livre movimentação do observador mantendo uma taxa de exibição de 30 quadros por segundo (*frames per second*, FPS).

Outros objetivos são a visualização do terreno com uma boa qualidade gráfica e o aproveitamento do conhecimento adquirido para melhorar a capacitação profissional na área de computação gráfica, além de futuras realizações pessoais.

1.3 Organização do Texto

Até este ponto do documento foi apresentada uma introdução sobre o mercado de jogos de computador comparado a outros setores do entretenimento. Além disto, as motivações para a realização deste trabalho, junto aos seus objetivos, foram detalhadas citando a exibição dos LSTs em tempo real, as dificuldades na área e mencionando exemplos de aplicações que fazem uso destes tipos de terreno.

O Capítulo 2 aborda conceitos básicos sobre o processo de exibição de objetos tridimensionais. Esta abordagem inclui o *pipeline* gráfico fixo, que é exemplificado neste Capítulo e o *pipeline* gráfico programável que é citado junto a exemplos de efeitos e benefícios do mesmo.

O Capítulo 3 aprofunda a definição de um LST, além de trazer

informações consideradas relevantes sobre a exibição deste tipo de terreno. Este Capítulo destaca os problemas que devem ser solucionados para permitir esta exibição em tempo real além de explicar alguns dos algoritmos que serão utilizados neste trabalho.

Com base na análise das informações mencionadas no Capítulo 3, o Capítulo 4 apresenta a abordagem desenvolvida neste trabalho para a exibição de LSTs em tempo real. São descritas a estrutura do algoritmo desenvolvido, a implementação do mesmo e os objetivos específicos desta abordagem.

O Capítulo 5 descreve os testes realizados no algoritmo e o compara com outros motores gráficos de código aberto. No Capitulo 6 possui uma conclusão junto a futuros trabalhos na abordagem desenvolvida. Por fim, são listadas as referências bibliográficas utilizadas neste documento e apêndices contendo imagens e códigos.

2. Processo de Exibição Gráfica Tridimensional

Para se obter a visualização de um objeto tridimensional (3D) em um computador deve ser realizado um processo composto por diversas etapas. Este processo, conhecido como processo de exibição 3D, ou *pipeline* gráfico, realiza uma seqüência de operações que geram a imagem a ser exibida.

Uma aplicação fornece a descrição geométrica que é enviada à placa de vídeo, onde são realizadas as operações do *pipeline* gráfico. Estas operações podem ser realizadas de maneira fixa (*pipeline* gráfico fixo ou tradicional, PGF) ou programável (*pipeline* gráfico programável, PGP).

A trajetória da descrição geométrica, junto a um simples fluxograma [18] das etapas e operações do PGF, é apresentada na **Figura 3**. As descrições destas etapas e operações [23] seguem nos parágrafos seguintes.

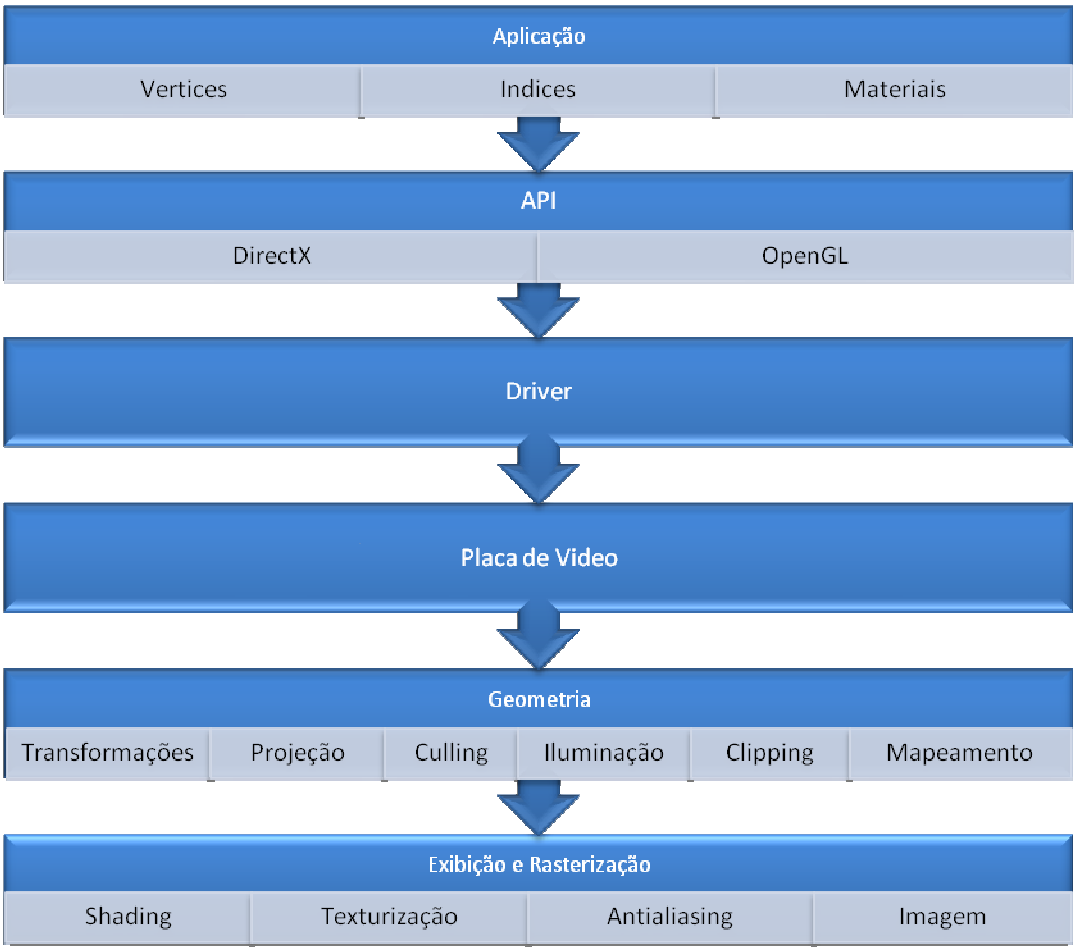


Figura 3 – Representação do *Pipeline* Gráfico Fixo.

Uma **Aplicação** é responsável por gerar a descrição geométrica que representa computacionalmente um objeto 3D. Para a **Aplicação**, esta descrição pode ser representada por diferentes formatos de arquivo (como 3DS, X e OBJ) e segue uma trajetória, para a **Placa de Vídeo**, formatada conforme a **API** utilizada. As coordenadas utilizadas nesta descrição são dadas no sistema de referência do próprio objeto.

A descrição geométrica contém um conjunto de vértices, índices e materiais. Os vértices representam pontos em um universo 3D, os índices identificam a ordem da união destes vértices e os materiais são os atributos, como a cor e a textura. A partir deste conjunto de dados é possível obter informações como as faces, as cores e outros atributos do objeto.

A *Application Program Interface* (**API**) é uma interface utilizada pela **Aplicação** que contém funcionalidades necessárias para se comunicar com os dispositivos de vídeo. Estas interfaces utilizam ***Drivers*** para realizar operações na **Placa de Vídeo** e acessar recursos disponíveis na mesma. Exemplos muito utilizados de **API** são o *DirectX*[7] e o *OpenGL*[8].

O ***Driver*** é a interface, de baixo nível, utilizada para a comunicação com os dispositivos de vídeo. Esta interface permite a comunicação do sistema operacional com a **Placa de Vídeo** e é distribuída, normalmente, pelos fabricantes destes dispositivos. Exemplos de fabricantes são a NVIDIA[9] e a ATI[10].

A **Placa de Vídeo** é um dispositivo físico responsável, por exemplo, pelo envio de sinais para o monitor do computador. Também é neste dispositivo onde se encontra a *Graphic Processing Units* (GPU) e, portanto, onde são realizadas as etapas do PGF e do PGP.

---

[7] http://msdn.microsoft.com/en-us/directx/default.aspx

[8] http://www.opengl.org/

[9] http://www.nvidia.com.br/page/home.html

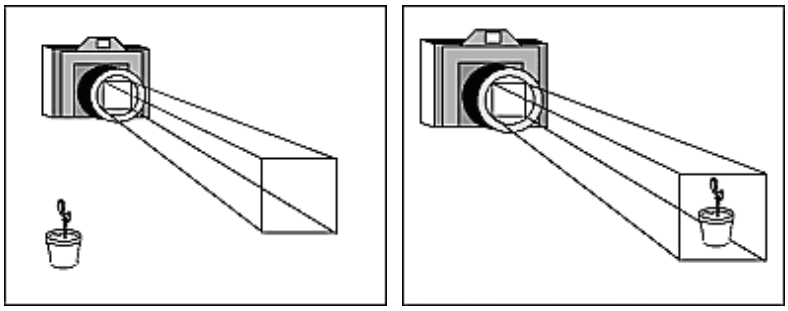
[10] http://www.amd.com/us-en/

2.1 Pipeline Gráfico Fixo

O PGF é composto por três etapas, cada uma destas realiza diversas operações a fim de obter uma **Imagem** a partir de uma descrição geométrica. Quando se utiliza este tipo de *pipeline* às operações descritas são sempre realizadas, mas a ordem e maneira de execução destas variam conforme a arquitetura da **Placa de Vídeo**.

A primeira etapa é a de **Geometria,** nela são realizadas operações nos primitivos [11] para a futura **Exibição e Rasterização** dos mesmos. Estas geometrias são provenientes da entrada do *pipeline*, ou seja, da descrição geométrica.

A primeira operação são as **Transformações** que realizam a escala, a translação e a rotação dos objetos. Além disto, as coordenadas existentes são transformadas do sistema de referência do objeto (a) para o sistema de referência da câmera (b). A **Figura 4** exemplifica o resultado destas transformações de coordenadas.



(a) **(b)**

Figura 4 – Transformação de coordenadas.

A **Projeção** é a segunda operação desta etapa e projeta o conjunto de primitivos existentes dentro do campo de visão (*frustum*) da câmera, determinando o formato (corpo de pirâmide) e o volume da visualização. Esta **Projeção** é realizada a partir de informações como a distância mínima (dMin) e

[11] Primitivos são os objetos (pontos, linhas ou triângulos) utilizados pela **Placa de Vídeo** para gerar a Geometria.

máxima (dMax) do *frustum,* além do ângulo de abertura (θ). A **Figura 5** ilustra a **Projeção** do campo de visão com estas informações da distância mínima e máxima.

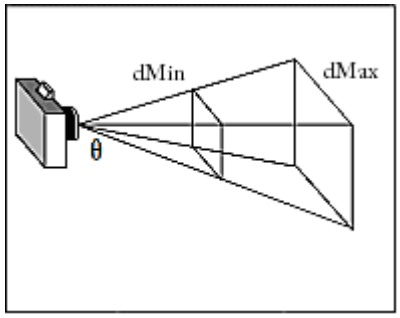


Figura 5 – Projeção do campo de visão.

A operação de ***Culling*** remove os primitivos não visíveis pelo observador, como por exemplo, aqueles fora do campo de visão. O ***Clipping*** é responsável pelo recorte dos primitivos que se encontram parcialmente presentes neste campo. Como resultado, é obtido um conjunto de primitivos que ocupa por completo o espaço do *frustum*.

A **Iluminação** é uma operação que determina a cor de um vértice da geometria a partir de um cálculo da interação entre os materiais e as fontes de luz.

O **Mapeamento** é responsável por realizar a conversão da geometria restante para as coordenadas de vídeo. Esta operação é realizada através de um conjunto de transformações para que os primitivos se enquadrem no espaço de dispositivo (*viewport*). A **Figura 6** representa o *viewport* relacionado ao campo de visão.

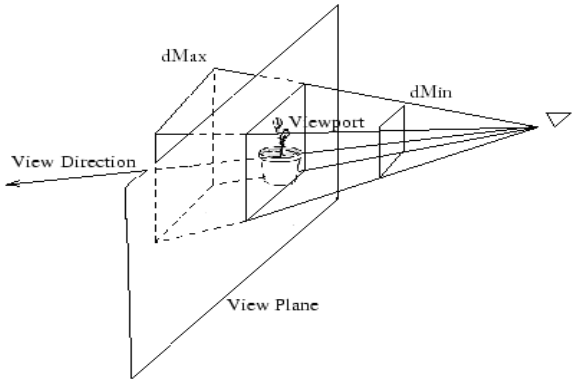
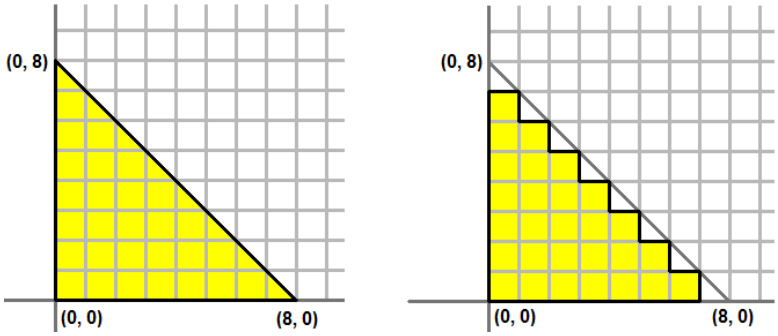


Figura 6 – Espaço de dispositivo (*viewport*).

As etapas de **Exibição e Rasterização** têm por objetivo gerar a saída do *pipeline*, ou seja, a **Imagem**. A **Rasterização** realiza um processo de amostragem onde cada polígono pode gerar um grande número de *pixels*[12]. Esta amostragem é a conversão entre representações vetorial e matricial, passando de um domínio continuo (a) para um domínio discreto (b). A **Figura 7** exemplifica o processo descrito.



(a) **(b)**

Figura 7 – Exemplo da etapa de Rasterização.

Shading é a operação que realiza a colorização dos *pixels*, provenientes da etapa de **Rasterização**, com base nos cálculos realizados pela **Iluminação**. Exemplos de tipos de ***Shading*** são o *Flat (a)*, *Gouraud (b)* e *Phong (c)*. Estes estão ilustrados na **Figura 8** e correspondem resumidamente a um ***Shading*** por polígono, por vértice e por *pixel*.

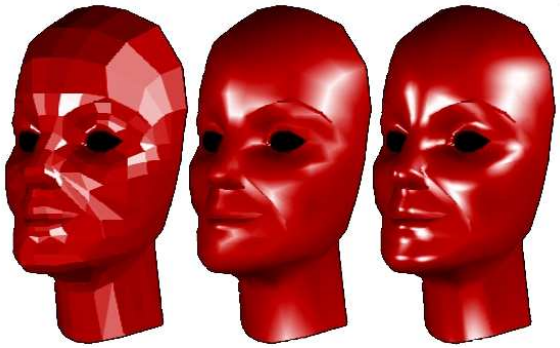


Figura 8 – Tipos de *shading*.

[12] O *pixel* é um ponto único de uma imagem gráfica. Aumentando a quantidade de *pixels* em uma mesma área, diminui-se o tamanho dos mesmos, elevando a resolução gráfica.

Texturização é uma forma de definir os aspectos das superfícies de uma geometria e isto pode ser realizado com o uso de uma imagem. **Antialiasing** é uma etapa opcional que suaviza as bordas das geometrias na visualização. Estas operações, para organização deste trabalho, estão descritas no Capítulo 3, nas seções 3.3.2 e 3.4.1.1 respectivamente.

Como resultado final da etapa de **Exibição e Rasterização**, e de todo o PGF, têm-se a **Imagem**. Esta **Imagem** pode ser exibida no monitor, armazenada de alguma forma ou mesmo, utilizada novamente no *pipeline*.

2.2 Pipeline Gráfico Programável

O PGP substitui partes das operações do PGF com o objetivo de realizar alguns efeitos impossíveis anteriormente. Esta liberdade, junto à busca pela realização de melhores efeitos artísticos, tornou o PGP atrativo para a comunidade de desenvolvedores e artistas.

Segundo Villar [24], existem diversas vantagens no uso do PGP. Uma destas é a possibilidade de desabilitar algumas funções do *PGF*, obtendo um ganho de performance. Além disto, o *PGP* pode fazer uso da GPU para processamento de dados, como por exemplo, a simulação de fluídos ou a análise de sinais e imagens.

As **APIs** *OpenGL* e *DirectX* utilizam linguagens de alto nível para expor as funcionalidades do *pipeline* programável para os desenvolvedores. O *DirectX* nomeou sua linguagem como *High Level Shading Language*[13] (HLSL) e o *OpenGL* optou por se referir a sua linguagem como *GL Shading Language*[14] (GLSL). Estas linguagens de *shading* utilizam funções gráficas, denotadas *vertex shader*, *geometry shader e pixel shader* (HLSL) ou *fragment shader* (GLSL), para substituir funções realizadas nas operações do PGF [25].

---

[13] http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx

[14] http://www.opengl.org/documentation/glsl/

2.2.1 *Vertex Shader*

O vertex shader é uma função que possui um vértice como entrada e saída. Esta função *é* executada para cada vértice que passa pelo processo de exibição, nunca operando sobre mais de um vértice simultaneamente. Além disto, esta função não pode adicionar ou remover vértices de um primitivo [24].

O vértice, neste contexto, é uma estrutura composta por atributos, sendo que um deles é obrigatoriamente a posição do vértice. Outros atributos podem ser o vetor de normal, a cor, as coordenadas da textura ou qualquer outro valor, definido pelo usuário, necessário para a execução do *vertex shader* desenvolvido.

Quando um *vertex shader* é utilizado certas funções das operações de **Transformações**, **Iluminação** e **Texturização** não são realizadas. As demais operações dos processos de **Geometria**, **Exibição e Rasterização** permanecem inalteradas.

O *vertex shader* pode ser utilizado em simulações de ondas através da função senoidal. A posição e a coloração dos vértices são modificadas de acordo com a função mencionada, ocasionando em um comportamento semelhante à de ondas coloridas [8], conforme ilustra a **Figura 9**.

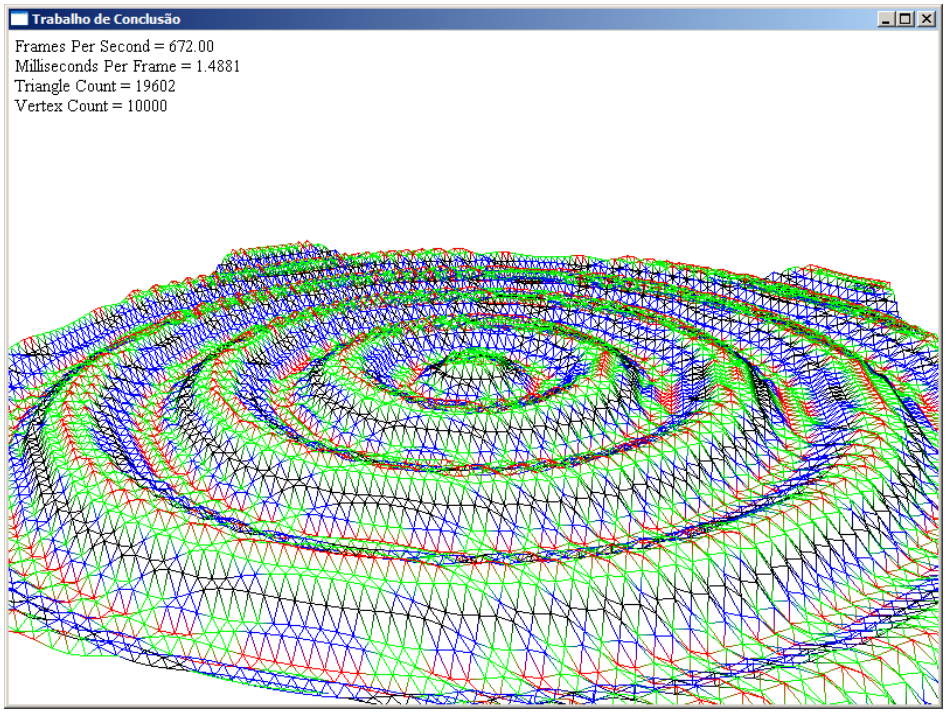


Figura 9 - Exemplo de *vertex shader*.

2.2.2 *Geometry Shader*

O *geometry shader* é uma função que recebe como entrada um vértice, um segmento de reta (dois vértices) ou um triângulo (três vértices). Com estes primitivos a função manipula a geometria de objetos 3D, podendo gerar, em tempo real, outros primitivos a partir do conjunto de entrada.

O *geometry shader* é executado entre as funções de *vertex shader* e de *pixel shader*. Isto ocorre pela necessidade de que as **Transformações** sejam realizadas da maneira adequada antes do envio da nova geometria ao *pixel shader*. A operação pode ser ajustada pelo *vertex shader* ou pelo próprio *geometry shader*.

Esta função pode ser utilizada para criar um efeito de explosão em um objeto 3D a partir do cálculo da normal de cada triângulo da geometria do objeto. Com estes vetores é possível determinar quais as direções que serão tomadas na extrusão de cada vértice. Este efeito é ilustrado na **Figura 10**, retirada do *Microsoft DirectX SDK*.

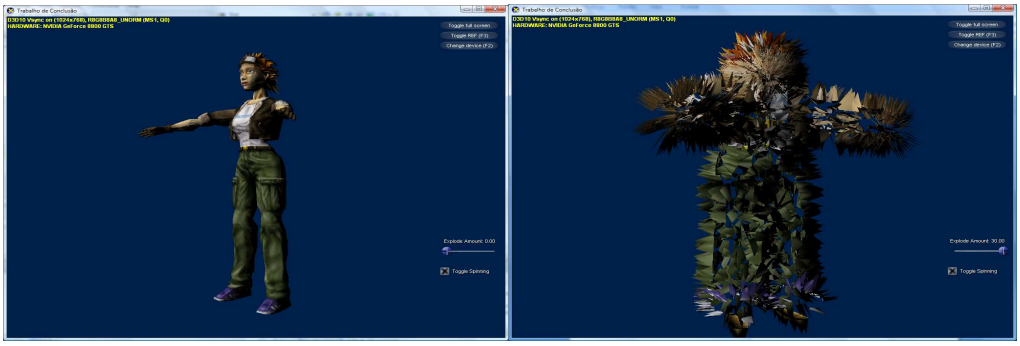


Figura 10 - Exemplo de *geometry shader*.

2.2.3 *Pixel Shader*

O pixel shader é uma função que possui como entrada *pixels* representados pelos valores da cor, da profundidade (Z-Buffer[15]) e dados da textura. Outras informações utilizadas anteriormente no *pipeline*, como os valores das normais, também podem ser utilizadas como entrada nesta função.

[15] Vetor responsavel em armazenar o valor de profundidade para cada *píxel*.

A função de *pixel shader* possui como saída padrão um valor de cor, podendo adicionalmente ter outros valores, como a profundidade. Os *pixels,* utilizados por esta função, são obtidos através da etapa de **Rasterização** e da operação de **Antialiasing**, quando utilizada.

O *pixel shader* substitui certas funções das operações de **Shading** e **Texturização**. As demais operações dos processos de **Geometria**, **Exibição e Rasterização** não são alteradas.

Um exemplo de *pixel shader* é sua utilização na técnica de *Bump Mapping*. Está técnica consiste em adquirir os valores das normais de uma superfície e a partir destes, alterar o comportamento da luminosidade em relação a cada *pixel*. A **Figura 11** ilustra a técnica de *Bump Mapping* no jogo Quake IV[16].

Figura 11 – Exemplo de *pixel shader*.

2.2.4 Outros Efeitos

O efeito de *depth of field* representa um fenômeno existente em fotografias onde a porção da imagem que não esta em foco perde nitidez. Com o pipeline gráfico programável e uma placa de vídeo moderna este efeito pode ser representado em tempo real. A **Figura 12**, proveniente do *plugin Depth of Field*

[16] http://www.quake4game.com/

Generator PRO[17] para a ferramenta *Adobe Photoshop*[18], ilustra a diferença de uma exibição com (b) e sem (a) este efeito.

(a) **(b)**

Figura 12 – Ilustração de *depth of field*.

Outro efeito é o *relief mapping,* este realiza a texturização usufruindo de parâmetros adicionais como o mapa de normal, explicado na seção 3.3.2. Com estes dados é possível determinar profundidade nas texturas o que gera uma perspectiva 3D em uma imagem bidimensional (2D). A **Figura 13**, obtida através da página[19] do professor Manuel Oliveira, ilustra um exemplo de r*elief mapping.*

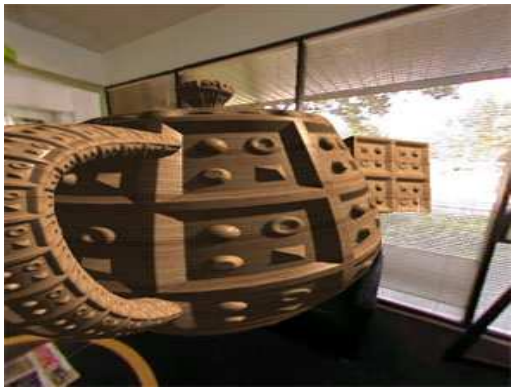


Figura 13 – Ilustração de *relief mapping*.

[17] http://www.dofpro.com/

[18] http://www.adobe.com/br/products/photoshop/photoshop/

[19] http://www.inf.ufrgs.br/~oliveira/RTM.html

3. Visualização de Terrenos

Para este trabalho, uma visualização de terrenos ideal é definida como aquela que garanta a resolução máxima de detalhes e um grande campo de visão, proporcionando ao observador a exibição de terrenos distantes, semelhante à visão humana. A visualização ideal também garante liberdade para navegar no terreno em qualquer direção, não percebendo qualquer deformação na geometria ou lentidão na exibição.

Infelizmente, esta visualização ideal de um LST em tempo real não pode ser obtida com os recursos existentes na maioria dos computadores [13]. Além disso, existem aplicações que exibem LSTs onde é necessário realizar diversas outras tarefas, como a animação de modelos e cálculos de física. Um exemplo destas aplicações são os jogos de computador, que realizam tarefas como estas em tempo real e com suporte a um amplo número de máquinas.

Enquanto a visualização de LSTs na forma ideal não pode ser realizada, são desenvolvidos algoritmos para permitir uma exibição mais rápida do terreno e poupar recursos computacionais necessários para as demais tarefas a serem realizadas. Este Capítulo apresenta os conceitos e problemas envolvidos nos algoritmos de visualização de terrenos que são importantes para o entendimento deste trabalho.

3.1 Mapas de Altura

A modelagem computacional de um terreno pode ser realizada através de estruturas de dados denominadas mapas de altura, estas têm por objetivo definir os valores de elevação da superfície ao longo do terreno. Estes mapas armazenam a definição das alturas, que é realizada através de uma matriz onde cada ponto representa a altura de um vértice específico em uma grade de triângulos [8].

Os mapas de altura são amplamente utilizados em aplicações que exibem terrenos, como sistemas de informação geográfica (*geographic information system, GIS*), e podem ser armazenados de diferentes formas, como RAW ou como os formatos de imagem *bitmap* (BMP) e *portable network*

graphics (PNG). O RAW não é considerado um formato, não esta vinculado a uma imagem e não possui nenhuma forma de compactação, as elevações são simplesmente armazenadas como uma cadeia de *bits* em um arquivo binário.

Os formatos de imagem BMP e PNG são alguns dos formatos possíveis para representar o mapa de altura como uma figura. A utilização de imagens para armazenar um mapa de altura é vantajosa pelos algoritmos de compactação existentes para as mesmas, além de facilitar a visualização para o usuário. A **Figura 14** é um exemplo de um mapa de altura em uma escala de cinza, sendo que o branco representa as zonas de maior elevação do terreno.

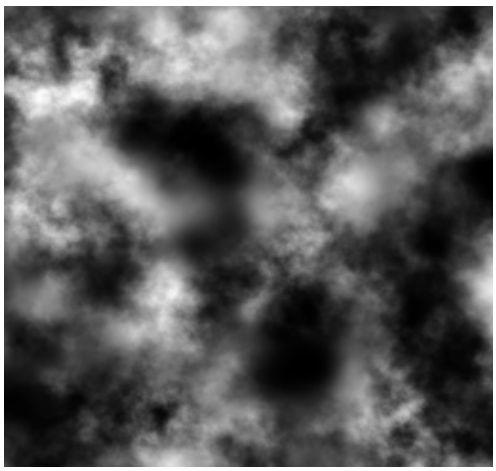


Figura 14 – Exemplo de mapa de altura em formato BMP.

Gerar um terreno a partir de mapas de altura requer informações como a altura mínima e máxima que o mapa representa e a distância entre os pontos do mesmo. Estas informações implicam na forma do terreno, influenciando a área que este irá representar e a sua qualidade de detalhes.

Mapas de altura com dimensões de 16384 por 16384 (16384^2), por exemplo, podem representar um terreno de pequena área. Se a distância entre os pontos deste mapa for de um metro, o mesmo representaria aproximadamente dezesseis quilômetros quadrados. Esta distância entre os vértices pode levar a baixa qualidade na geometria do relevo.

Diminuindo a distância entre os pontos do mapa de altura para meio metro aumentaria a qualidade visual do terreno, mas como conseqüência, o número de triângulos necessários para representar o mesmo mapa agora exibiria um terreno com um quarto da área (quatro quilômetros quadrados). A

Figura 15, gerada pelo protótipo deste trabalho, ilustra as diferenças causadas pela distância dos pontos de elevação quando utilizado o valor de um metro (a, c) e dois metros (b, d).

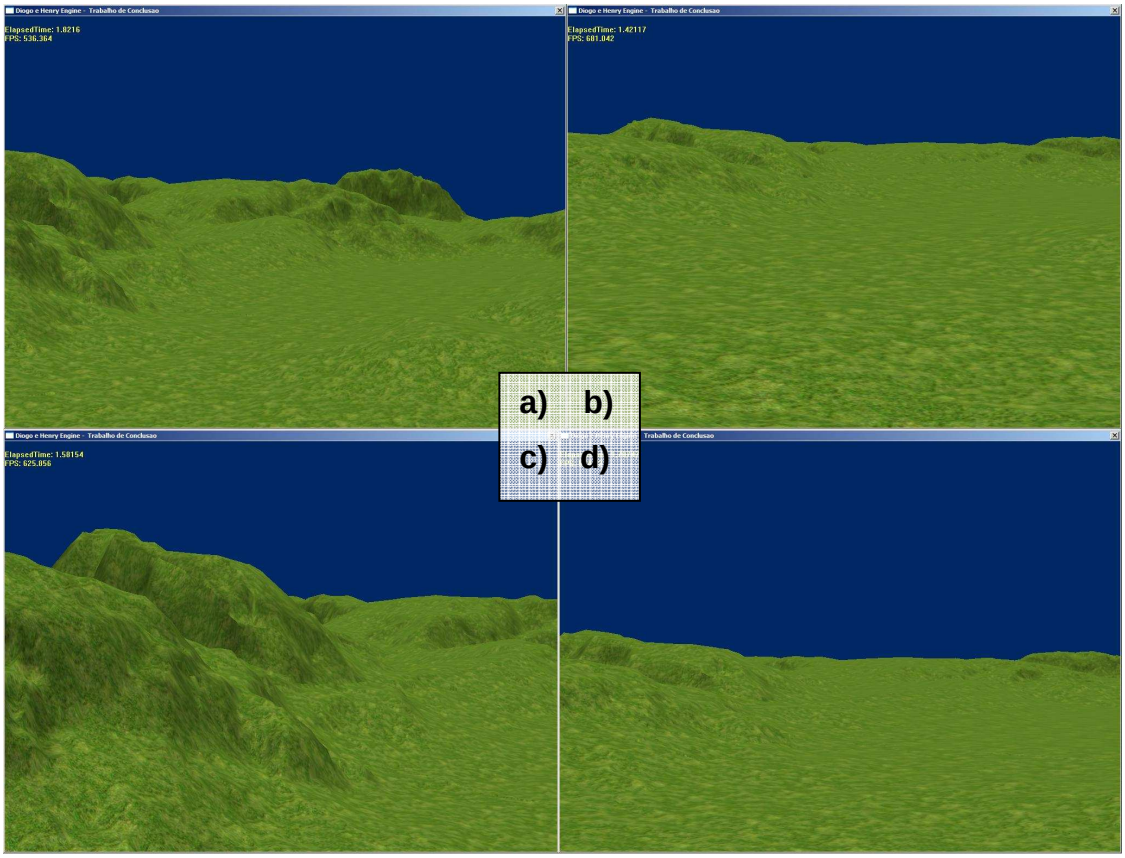


Figura 15 – Diferenças causadas pela distância dos pontos de elevações.

3.2 Terrenos de Larga Escala

É importante prosseguir detalhando a definição de LST apresentada no Capítulo de Introdução deste documento. No contexto do presente trabalho, para que um terreno seja considerado de larga escala, deve ter uma dimensão tal que as técnicas comuns de exibição não atinjam uma taxa menor que 30 quadros por segundo.

São consideradas técnicas comuns de exibição aquelas que simplesmente exibem todos os triângulos que compõe o mapa de altura, também conhecidas como técnicas de força bruta. A taxa de quadros por segundo obtida é considerada em relação à placa de vídeo escolhida para apresentar à aplicação, conseqüentemente, a exibição de um terreno varia conforme o dispositivo utilizado.

Utilizando a placa de vídeo *Quadro Plex*[20] da *NVIDIA* como exemplo, com capacidade de exibir até um bilhão de triângulos por segundo, pode-se observar que um terreno derivado de um mapa de altura com dimensões de 8192^2 (representado por uma grade de 134.184.962 triângulos) pode ser considerado um LST, já que a placa de vídeo não possui capacidade para exibir este terreno em tempo real.

Este trabalho considera a importância da qualidade com que o relevo é representado, além disto, busca trabalhar com LSTs que representem uma grande área, como oito quilômetros quadrados. Esta combinação de necessidades pode ser encontrada, por exemplo, em sistemas de informação geográfica utilizados no acompanhamento de obras, logística e controle ambiental. Os jogos de computador também compartilham deste conjunto de necessidades.

3.3 Etapas da Visualização

Como a exibição de qualquer objeto 3D, a visualização de LSTs necessita das etapas do processo de exibição e do *pipeline* gráfico, entretanto, a complexidade destes terrenos demanda tarefas adicionais. No contexto deste trabalho, estas tarefas estão organizadas em três grupos: **Processamento de Polígonos**, **Formas de Texturização** e **Gerência de Memória**.

O **Processamento de Polígonos** e a **Gerência de Memória** são essenciais para armazenar e melhor descrever a geometria que será entregue para a placa de vídeo. As **Formas de Texturização** são abordadas como um grupo pela sua importância na qualidade visual do terreno e relação com as funções de *shaders*. Segue na próxima seção detalhes sobre estes grupos, junto a problemas relacionados aos mesmos.

---

[20] http://www.nvidia.com/page/quadroplex.html

3.3.1 Processamento de Polígonos

O processo de exibição dos grandes blocos de geometria que compõem um LST não pode ser atribuído às GPU sem que antes a geometria seja tratada. Estes blocos, mesmo com o ótimo e crescente desempenho das placas de vídeo, possuem um número muito elevado de polígonos para uma exibição com FPS apropriados para tempo real [10].

As placas de vídeo podem obter melhor desempenho no **Processamento de Polígonos** conforme a organização da descrição geométrica entregue as mesmas. Desta maneira, técnicas de triangularização são utilizadas para desenvolver a geometria de diferentes formas, como por exemplo, *triangle strips, triangle lists, triangle fan* e *quad strips*, ilustradas na **Figura 16**.

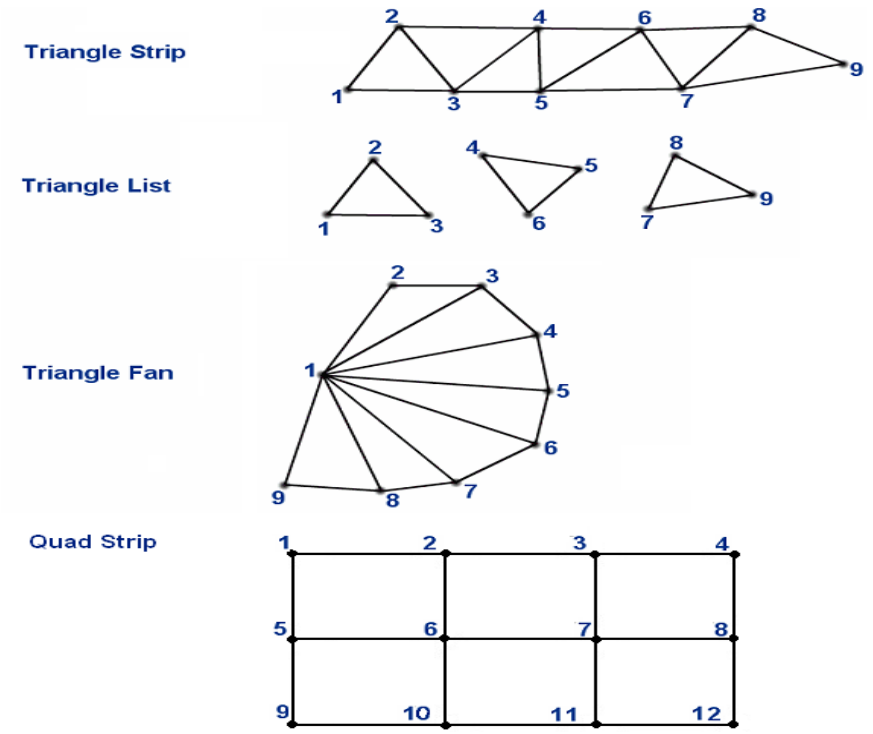


Figura 16 – Formas de triangularização.

A descrição geométrica, ao ser encaminhada à placa de vídeo, percorre um barramento existente no computador, como por exemplo, o *Peripheral Component Interconnect Express* (PCIe) ou o *Accelerated Graphics Port* (AGP). Este barramento influencia na velocidade com que a geometria será encaminhada para o dispositivo e, portanto, influencia na taxa de quadros por segundo de exibição da mesma.

Para que um objeto seja exibido pela placa de vídeo deve ser realizado um *drawcall*[21]. Esta chamada de sistema faz com que o dispositivo realize as etapas do *pipeline* gráfico para uma determinada descrição geométrica. O *drawcall* é realizado através do *driver* da placa de vídeo e pode ser acompanhado de um tempo de atraso (*delay*). No sistema operacional *Windows XP*, por exemplo, este atraso é resultado da arquitetura implementada para a utilização dos *drivers* e limita o número de chamadas possíveis para a exibição em tempo real.

3.3.2 Formas de Texturização

É comum a necessidade de colorir os *pixels* existentes nos polígonos para alcançar a visualização esperada de um objeto 3D. Uma maneira de realizar esta tarefa é o processo de texturização, que mapeia uma imagem sobre os polígonos deste objeto ou, como mostra a **Figura 17**, projeta a geometria do mesmo em uma imagem que será pintada da forma desejada.

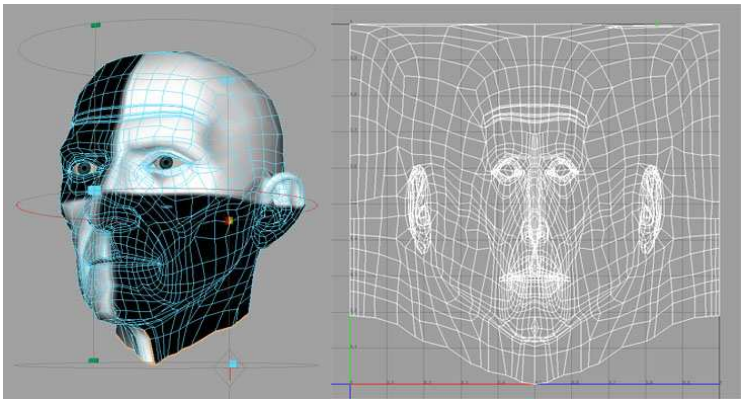


Figura 17 – Exemplo do processo de texturização.

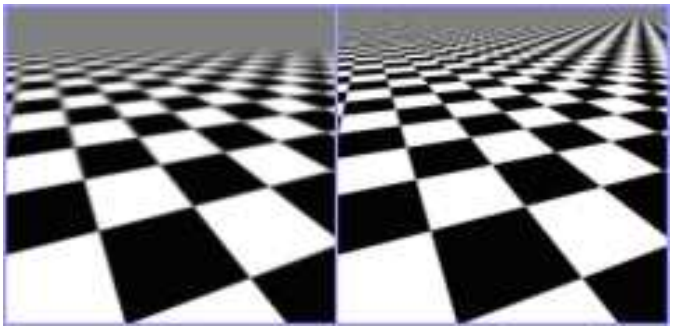
A texturização de uma geometria pode acarretar problemas, como em casos que o polígono está sendo exibido em uma área pequena da tela, e uma textura grande deve ser projetada na mesma. A fim de solucionar este problema existe a técnica de *mipmaps* que consiste na criação de um número

[21] Chamada realizada através da API e do *driver* para que a placa de vídeo realize o processo de exibição em uma determinada descrição geométrica.

determinado de réplicas da textura. Cada réplica é de tamanho menor que a textura original e simplificada com algum algoritmo [11].

As versões reduzidas da textura permitem opções para as diferentes áreas em que o polígono pode se encontrar no processo de exibição. A técnica de *mipmaps*, além de solucionar o problema descrito, economiza tempo computacional em troca do maior consumo de memória pelas texturas. A **Figura 18** é um exemplo de uma textura com os *mipmaps* e a **Figura 19** mostra a melhoria na exibição utilizando a técnica explicada (b).

Figura 18 – Exemplo de *mipmap*.



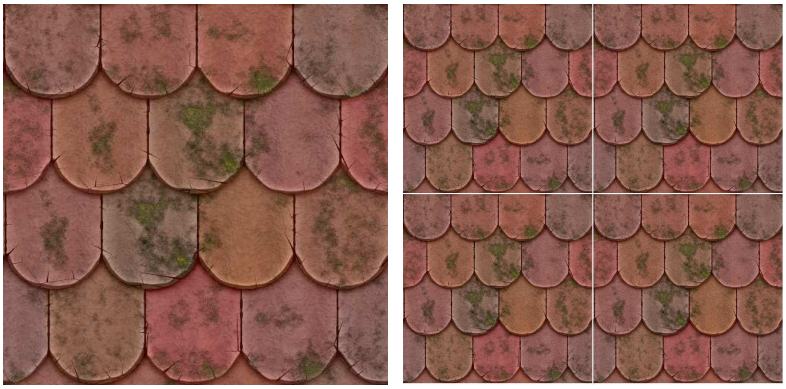
(a) **(b)**

Figura 19 – Utilização de *mipmaps.*

O método de texturização utilizado para LSTs deve considerar a quantidade de polígonos existentes. Texturas únicas, em alta resolução, raramente podem ser utilizadas, já que o tamanho das mesmas superaria o espaço de memória da maioria das placas de vídeo [14].

Para contornar o problema de texturas únicas em alta definição existem as chamadas texturas em ladrilhos (*tiled textures*). Estas são repetidas ao longo do objeto e podem ser de menor tamanho. Para permitir isto, as

extremidades da textura são organizadas de forma que exista um vínculo quando uma réplica é colocada ao seu lado. A **Figura 20** ilustra uma textura (a) colocada em ladrilho (b).



(a) **(b)**

Figura 20 - Ilustração de uma textura em ladrilho.

Com o objetivo de melhorar a qualidade visual de uma superfície, é possível fazer uso de uma textura de detalhe *(detail texture*). Esta textura normalmente é exibida quando o observador encontra-se próximo do objeto 3D, pois sua exibição constantemente pode ser custosa computacionalmente.

Quando se utiliza texturas em ladrilho, a sobreposição de texturas de detalhe aumenta o realismo removendo os aspectos de repetição ao longo da superfície de um objeto. A **Figura 21**, retirada do jogo *Urban Empires*[22] ilustra as diferenças quando é feito o uso de uma textura de detalhe.

Figura 21 – Ilustração de uma textura de detalhe.

[22] http://www.radioactive-software.com/gangwar/DetailTextureCompare.jpg

O mapa de alfa (a*lpha maps)* é uma textura utilizada para definir áreas de transparência e translucência de cada objeto 3D. Estas texturas podem ser em preto e branco, com a cor branca definindo quais as partes do objeto que devem ser exibidas e a cor preta quais as partes que devem ser transparentes ou translúcidas. A **Figura 22** mostra um exemplo de mapa de alfa (a) utilizado na exibição de uma imagem em um cenário 3D (b), permitindo o efeito ilustrado com a imagem da árvore.

(a) **(b)**

Figura 22 – Efeito utilizando um mapa de alfa.

A idéia de transparência representada pelos mapas de alfa também pode ser utilizada nos terrenos. Neste caso, o mapa identifica a translucência de uma textura, identificando que regiões do LST possuem a textura. Esta utilização dos mapas permite, junto a texturas em ladrilhos, que os mesmos sejam suficientes para aplicar um conjunto de texturas na superfície do terreno.

Outro tipo de textura que, para o contexto deste trabalho, deve ser ressaltada são os mapas de normal (n*ormal maps)*. Estes reúnem informação sobre as direções dos vetores normais dos vértices ou das faces, possibilitando melhor detalhar um objeto 3D.

Com o uso dos mapas de normal a representação da superfície e da luminosidade é melhorada, sem o uso de polígonos adicionais. A **Figura 23** ilustra a diferença de um automóvel exibido sem utilizar (a) e utilizando (b) um mapa de normal.

(a) **(b)**

Figura 23 – Exemplo de utilização do mapa de normal.

Os mapas de normal utilizam dos três canais de cores (RGB) para representar os vetores normais. Isto ocorre pela forma de representar direções no plano 3D (x, y e z) e faz com que estes mapas tenham uma coloração peculiar quando visualizados, como ilustrado na **Figura 24**.

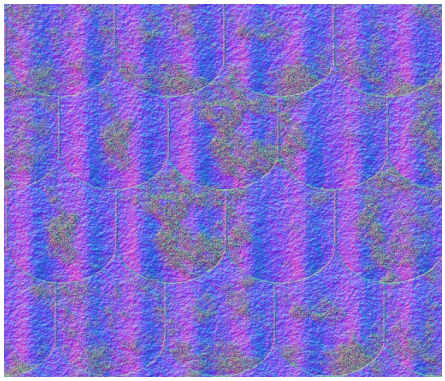


Figura 24 – Visualização de um mapa de normal.

3.3.3 Gerência de Memória

Uma questão relevante quando se esta lidando com LSTs é a utilização da memória, a grande dimensão destes terrenos torna volumosa a quantidade de informação a ser utilizada e armazenada. Utilizando-se da texturização como exemplo, uma textura com dimensões laterais de 8192, em formato *PNG* e com resolução de 96*dpi* ocupa 92*MB* em memória. Considerando a quantidade de memória nas placas de vídeo, este valor é alto, além disso, consiste em uma única textura entre as várias que podem ser utilizadas pela aplicação.

Nos mapas de altura de grandes dimensões é elevada a quantidade de memória necessária para armazenamento. Devido ao seu tamanho estes mapas não podem ser carregados inteiramente em memória. Algoritmos de compactação ou *swap* entre as memórias e o disco rígido devem ser utilizados para permitir a exibição dos LSTs [13].

Um mapa de altura em formato *RAW*, por exemplo, com dimensões laterais de 16384 e com as alturas do tipo ponto flutuante possui 268.435.456 elevações armazenadas. Cada ponto flutuante ocupa quatro *bytes*, portanto o mapa de altura possui 1*GB* no total.

3.4 Algoritmos

Os algoritmos de exibição de LST em tempo real utilizam técnicas de *Level of Detail* (LOD) [15]. Estas técnicas baseiam-se no decremento da complexidade de representação de um objeto 3D conforme um conjunto de métricas. Uma destas métricas é verificar onde se encontra o observador em relação à porção do terreno a ser exibida e utilizar um menor número de polígonos naquelas regiões distantes deste observador.

As técnicas de LOD têm por objetivo reduzir o número de polígonos a ser exibido, acelerando o **Processamento dos Polígonos**. Além destas técnicas outras devem ser utilizadas para auxiliar a **Gerência de Memória** e as **Formas de Texturização**. Segue alguns dos algoritmos de LOD existentes na literatura junto a problemas comuns entre estes.

3.4.1 Problemas Comuns

Os algoritmos de exibição de LSTs em tempo real devem solucionar ou amenizar problemas provenientes da execução de certas técnicas. Estas técnicas, como por exemplo, o LOD, são realizadas nas etapas da visualização de terrenos. Alguns destes problemas, relevantes a este trabalho, estão descritos a seguir.

3.4.1.1 Aliasing

Um número elevado de vértices nos blocos existente em um LST, com uma triangulação uniformemente densa, pode levar também a problemas de *aliasing*, como é exemplificado na **Figura 25** proveniente da *Ogre 3D*[23]. Isto ocorre por causa de um mapeamento de *samples*[24] para *pixels* de "muitos para um", como acontece em texturizações sem o uso de *mipmaps* [12].

Figura 25 - Exemplo de problema de *aliasing*.

Para solucionar estes problemas de *aliasing* as placas de vídeo modernas possuem recursos conhecidos como *antialiasing.* Estes consistem em algoritmos que trabalham com a imagem a ser exibida pela GPU a fim de diminuir o *aliasing* nas mesmas. A **Figura 26** mostra um exemplo do recurso de *antialiasing* existente nas placas de vídeo.

(a) Sem AA **(b) 4x MSAA**

Figura 26 - *Antialiasing* nas placas de vídeo.

[23] http://www.ogre3d.org/

[24] *Samples* representam uma porção ou segmento de um todo.

Outra solução para este problema pode ser realizada pelos algoritmos de exibição de LSTs. É possível evitar o *aliasing* controlando o tamanho dos triângulos em tela fazendo que os mesmos ocupem um número determinado de *pixels*, desta maneira, evitando a formação de uma triangulação uniformemente densa.

3.4.1.2 Brechas

Uma brecha é composta pela falta de um triângulo em uma geometria, que pode ser originada, por exemplo, pela falha nos índices que representam os triângulos da mesma. Este problema é comum nas técnicas de LOD, devido às mudanças na resolução de partes da geometria, que atualizam os índices para que os mesmos formem triângulos coerentes. A **Figura 27** abaixo mostra um exemplo de brecha em um terreno.

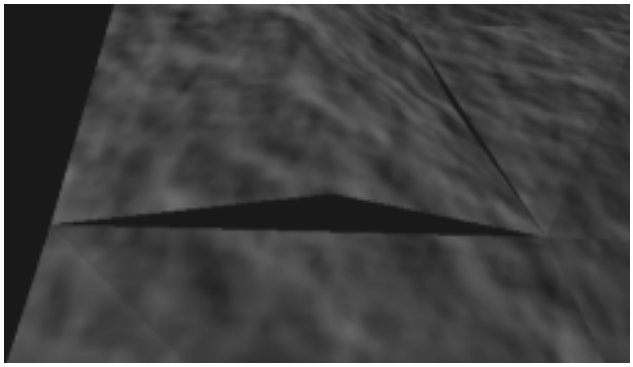


Figura 27 – Exemplo de brecha em terreno.

3.4.1.3 Deformações

As modificações na geometria do terreno realizadas pelas técnicas de LOD, em alguns casos, podem ser percebidas pelo usuário. Isto ocorre quando é realizada uma transformação no nível de detalhe em uma parte do terreno que se encontra em foco pelo usuário.

Conforme a definição da visualização ideal de um terreno, apresentada no início deste Capítulo, este é um problema que deve ser abordado. As deformações devem ser amenizadas, ou até mesmo evitadas, garantindo a qualidade visual na exibição. A **Figura 28** mostra a deformação na geometria do terreno visualizada pelo observador em dois quadros subseqüentes (a e b).

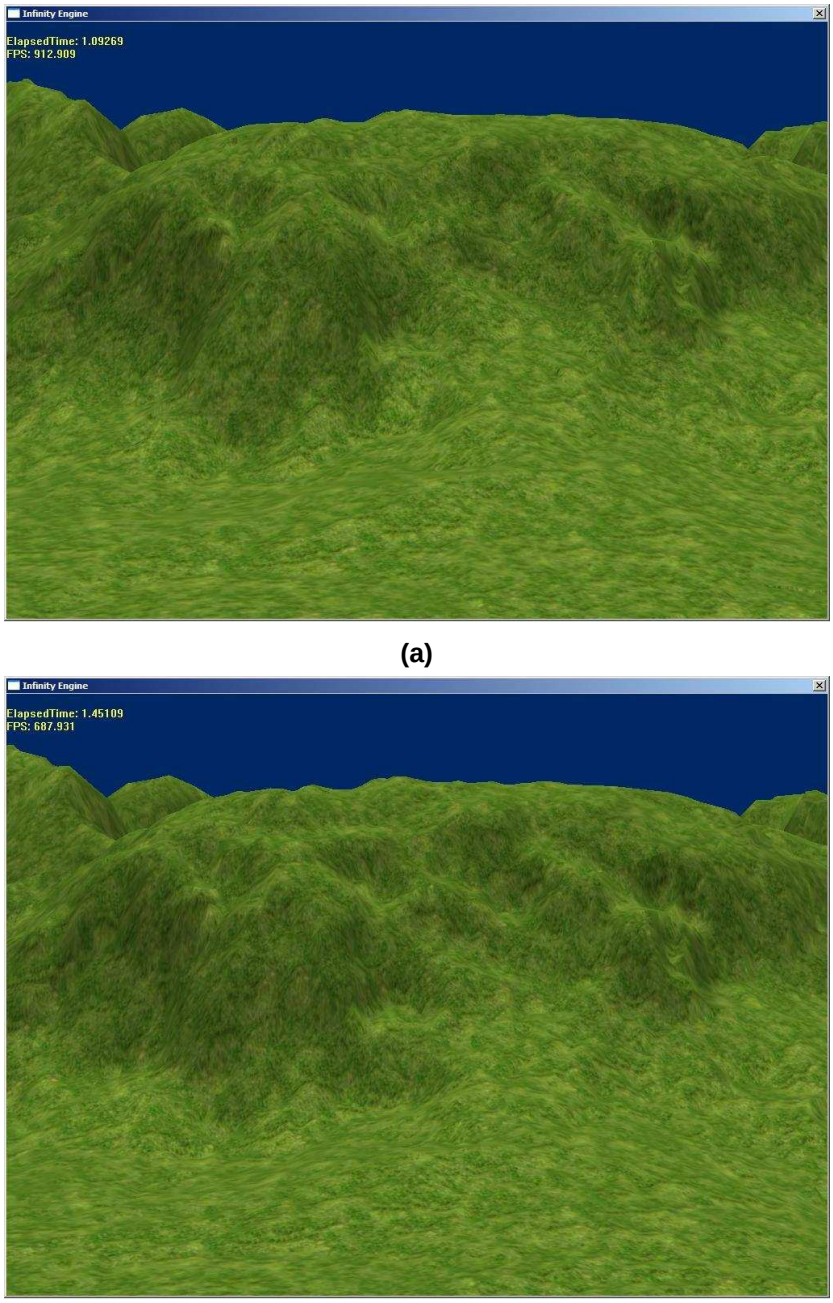


Figura 28 – Exemplo de deformações no terreno entre dois quadros.

3.4.2 *Quadtree*

A *quadtree* é uma estrutura de dados em árvore na qual cada nodo possui até quatro filhos. Estes filhos podem possuir formas distintas como, por exemplo, um quadrado ou outra forma qualquer. Estas árvores geralmente são utilizadas para particionar um espaço bidimensional dividindo-o recursivamente em quatro partes (quadrantes).

Para partição de terrenos é comum a utilização do tipo de árvore estruturada por regiões. Neste tipo de *quadtree* os quadrantes de um nodo possuem formas e tamanhos idênticos, além disto, cada um é limitado a ter quatro filhos ou nenhum. A **Figura 29** ilustra a estrutura de dados.

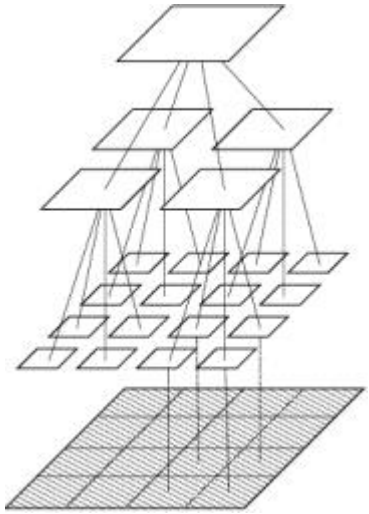


Figura 29 – Estrutura de dados *quadtree*.

3.4.3 *Continuous Level of Detail*

O *Continuous Level of Detail* (CLOD) foi elaborado por Lindstrom e Koller no ano de 1997 [17], representa o terreno com uma *quadtree* e pode ser descrito em dois passos. O primeiro é uma simplificação superficial da geometria do terreno que é realizada para determinar quais modelos de LOD serão necessários. Na segunda etapa, estes modelos têm sua geometria modificada verificando cada vértice individualmente.

Na simplificação superficial é realizada uma estimativa para a remoção de um conjunto de vértices de um bloco. Esta estimativa se baseia nas diferenças de alturas entre os vértices do grupo. Se esta diferença for menor do que um valor pré-estabelecido então os vértices podem ser descartadas e o bloco pode ser substituído por outro de menor LOD.

A simplificação baseada em vértices, diferentemente da simplificação superficial, é mais custosa computacionalmente. Esta é responsável por remover vértices considerados desnecessários, substituindo um grupo de triângulos por outro que ocupe o mesmo espaço.

O algoritmo garante que nenhum problema ocorre na primeira etapa de simplificação além daqueles que ocorreriam se a segunda etapa fosse aplicada. Este processo é executado constantemente (quadro a quadro) a fim de gerar um nível de detalhe apropriado para as áreas da superfície do terreno, auxiliando o **Processamento de Polígonos**, conforme exemplificado na **Figura 30**.

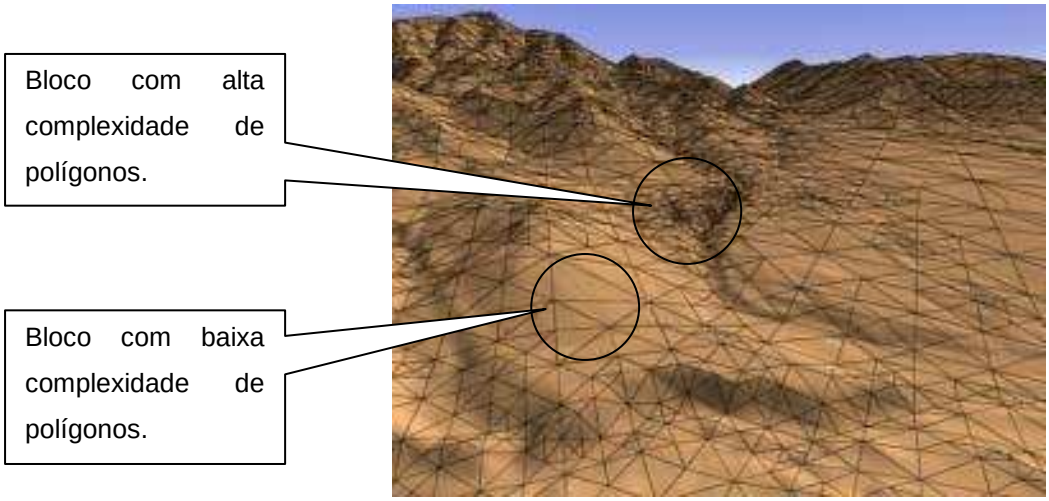
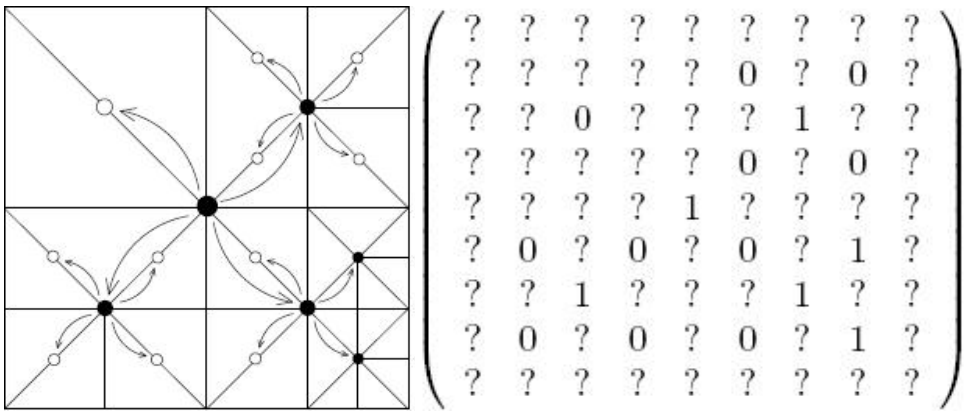


Figura 30 - Exemplo do algoritmo CLOD.

Na implementação de CLOD de Röttger [7] a estrutura dos dados do terreno também é organizada em uma *quadtree*, entretanto, esta árvore, é representada por uma matriz booleana. Esta árvore é estruturada por regiões, possuindo nodos com formas e tamanhos idênticos. A *quadtree* (a) e a matriz booleana (b) que a representa é organizada conforme a **Figura 31**.



(a) **(b)**

Figura 31 – Representação de uma *quadtree* em uma matriz booleana.

As entradas da matriz correspondem aos centros das regiões da *quadtree* e quando estas entradas estiverem denotadas por um valor verdadeiro (1), é possível aprofundar-se mais um nível na região. Caso este valor seja falso (0) a região é um nodo folha da árvore, ou seja, um nodo que não possui filhos. As entradas da matriz também podem não possuir um valor (?), indicando que tal região não será acessada pelo algoritmo neste momento.

Esta matriz é utilizada na geração dos índices da geometria quando é verificado o LOD dos nodos e de seus vizinhos. Esta rotina possibilita que os índices sejam criados corretamente, evitando o problema de brechas.

3.4.3.1 *Chunked* LOD

Chunked LOD [13] é outra implementação do CLOD,elaborada por Ulrich, que agrega **Formas de Texturização**, o reduzido **Processamento de Polígonos** e a busca de ser amigável à arquitetura da placa de vídeo. A principal modificação é a utilização da *quadtree* para armazenar descrições geométricas estáticas representando os diferentes LODs, estas representações são denominadas *chunks*. O uso de *chunks* na exibição de um terreno pode ser vista na **Figura 32**.

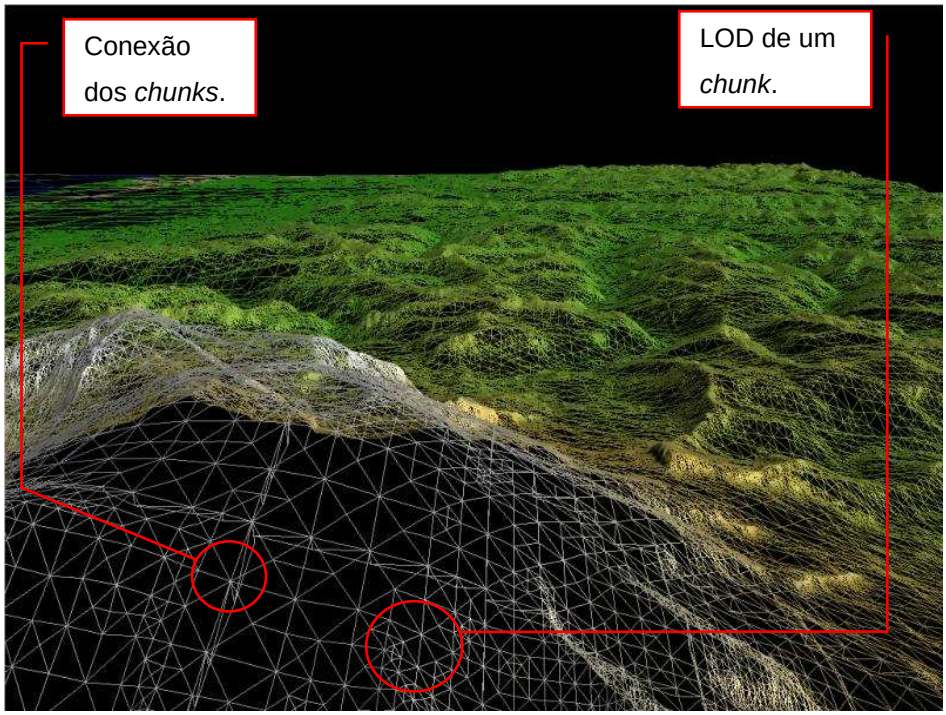
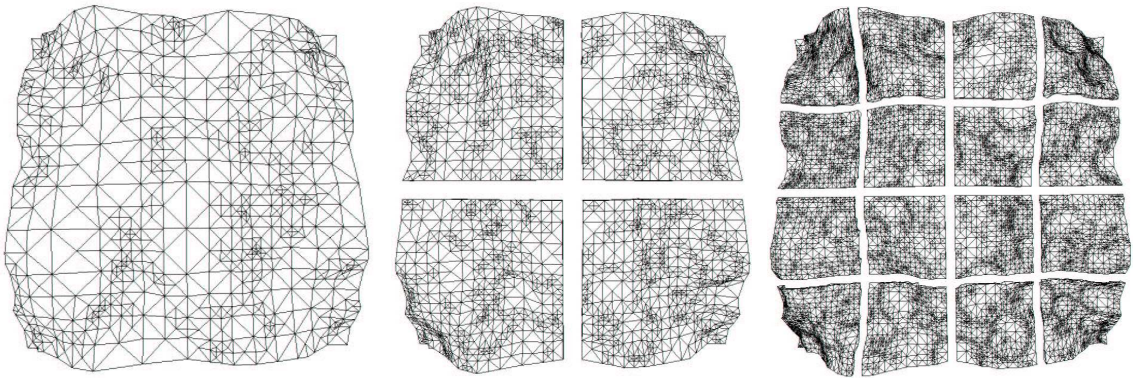


Figura 32 - Exibição de um terreno com *Chunked* LOD.

A árvore é estruturada por regiões e os nodos filhos (c) correspondem à representação, em maior detalhamento, da geometria descrita por seu pai (b). Desta forma, o nodo raiz (a) da árvore corresponde a uma representação com pouco detalhe de todo o terreno. Um exemplo desta organização de *chunks* é ilustrado na **Figura 33**.



(a) **(b)** **(c)**

Figura 33 – *Chunks* organizados em árvore.

O uso de *chunks* possui a desvantagem, ao utilizar uma estrutura de dados estática, de inviabilizar alterações na geometria do terreno em tempo real e de aumentar o consumo de memória. Em contrapartida, esta implementação diminui o uso da CPU, poupando recursos computacionais, além de permitir uma associação de uma textura para cada *chunk,* simplificando o processo de texturização para o LST.

3.4.4 *Geometry ClipMaps*

Este algoritmo foi desenvolvido por Hugues Hoppe e Frank Losasso [12] e consiste em agrupar o terreno em um conjunto de redes posicionadas ao redor do observador. Estas redes representam zonas com diferentes resoluções do terreno que são armazenadas em um *vertex buffer*[25] localizado na memória da placa de vídeo.

[25] Objeto para armazenar e representar a estrutura (vértices) de um objeto 3D.

As resoluções variam em uma potência de dois e, à medida que o observador se movimenta, as zonas são deslocadas e os valores das redes são recalculados [16]. A **Figura 34** ilustra estas diferentes zonas e suas diferentes resoluções.

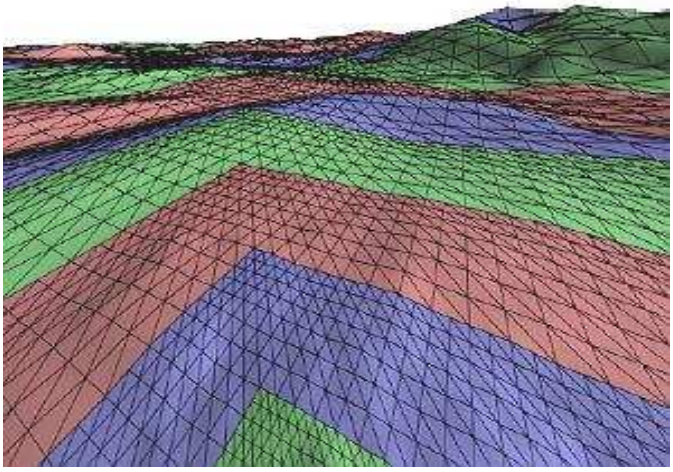


Figura 34 - Exemplo do algoritmo *Geometry ClipMaps*.

O algoritmo se diferencia da maioria das técnicas de LOD existentes por possuir como base o uso de *mipmaps*. O terreno é tratado como um mapa de altura pré-filtrado em uma pirâmide com um determinado número L de níveis, onde cada nível representa uma resolução diferente, como ilustrado na **Figura 35**.

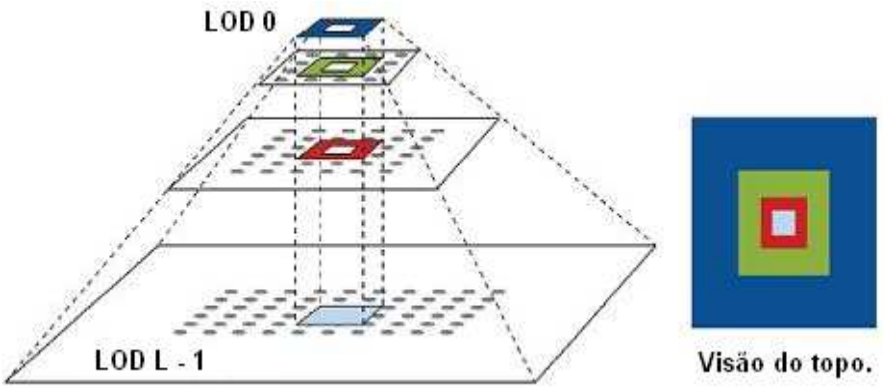


Figura 35 - Pirâmide do terreno no *Geometry ClipMaps*.

Para cada um dos níveis de resolução desta pirâmide o algoritmo armazena uma zona que é centralizada em relação ao observador, resultando na aparência de “anéis” circulares [16]. Estas zonas, com um nível (L) igual a

dois, estão exemplificadas na **Figura 36**, onde o ponto central simboliza o observador.

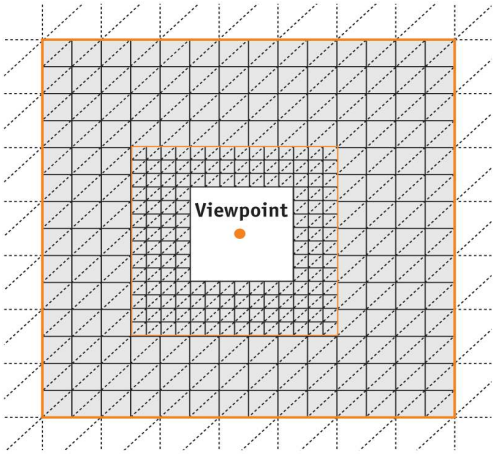


Figura 36 – Conjunto de grades regulares e aninhadas.

Esta abordagem não realiza um pré-processamento refinando a geometria e o mapa de altura, efetuando os processamentos somente durante a exibição do terreno. Isto faz com que uma representação de um LST possa ser mantida na memória principal [19], facilitando a **Gerência de Memória**.

Além disto, pode ser utilizado o modelo de *shaders* 3.0 para realizar o algoritmo inteiramente na GPU, possibilitando que os cálculos da posição dos vértices de cada zona, das normais e da texturização sejam realizados pela placa de vídeo.

3.4.5 *View-Dependent Refinement Progressive Meshes*

Este algoritmo foi desenvolvido por Hugues Hoppe no ano de 1997 [21] e realiza modificações na descrição da geometria de um objeto 3D conforme o campo de visão do observador. O algoritmo baseia-se na utilização das *Progressive Meshes* [20], desenvolvidas por Hoppe um ano antes. Estas estruturas consistem em uma maneira de descrever a geometria de forma que a mesma possa ser simplificada com pouca alteração em sua aparência.

As *Progressive Meshes* envolvem um estudo anterior de Hoppe sobre a otimização de geometrias [22], este estudo consiste em simplificar as mesmas removendo as bordas de maneira inteligente. Além disto, estas

estruturas armazenam modificações de forma que seja possível se recuperar a geometria original, como mostra a **Figura 37**.

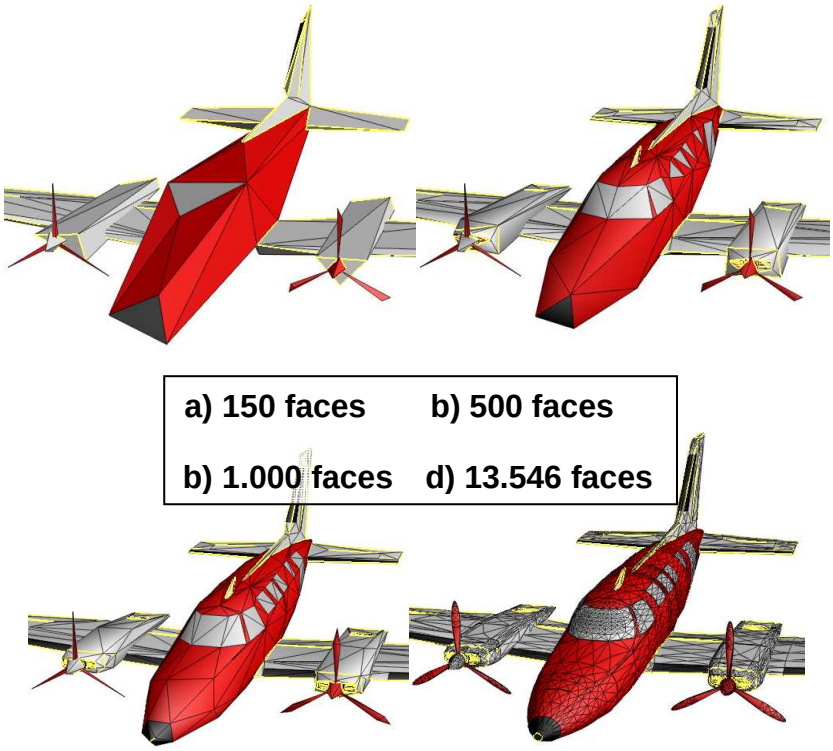


Figura 37 – Exemplo de *Progressive Mesh*.

Neste algoritmo de exibição de LSTs em tempo real, os estudos sobre a simplificação da geometria são utilizados para realizar o LOD de forma dependente ao observador. Um dos critérios de simplificação consiste em selecionar a geometria que se encontra fora do campo de visão e simplificá-la a fim de reduzir o custo de exibição, conforme a **Figura 38**.

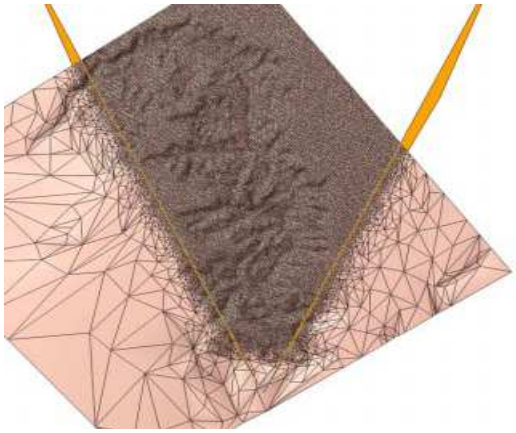


Figura 38 – Critério de refinamento com base no campo de visão.

Outro critério é a simplificação conforme a orientação da superfície, ou seja, identificar as faces que não estão voltadas para o observador (não visíveis) e então simplificá-las. Com isto, o terreno pode ser exibido mantendo detalhes somente dentro do campo de visão, como pode ser visualizado na **Figura 39** abaixo.

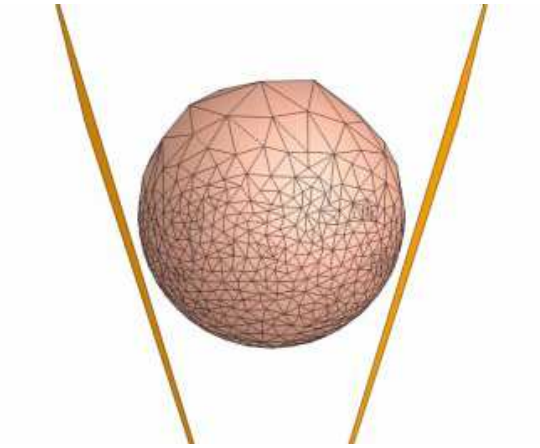


Figura 39 – Critério de refinamento com base na orientação da superfície.

4. Large Scale Terrain Renderer (LSTR)

Avaliando os problemas encontrados para a exibição de LSTs em tempo real, e incluindo a livre movimentação do observador no cenário, foram exploradas as técnicas e estruturas de dados dos algoritmos da seção 3.4. A partir desta avaliação foi desenvolvida uma abordagem para a exibição de LST que contempla as etapas de visualização da seção 3.3 (**Processamento de Polígonos**, **Formas de Texturização** e **Gerência de Memória**). Esta abordagem é um algoritmo de exibição de LSTs em tempo real, dividido em duas fases: a **Organização da Estrutura de Dados** e a **Exibição do LST em Tempo Real** conforme o diagrama na **Figura 40**.

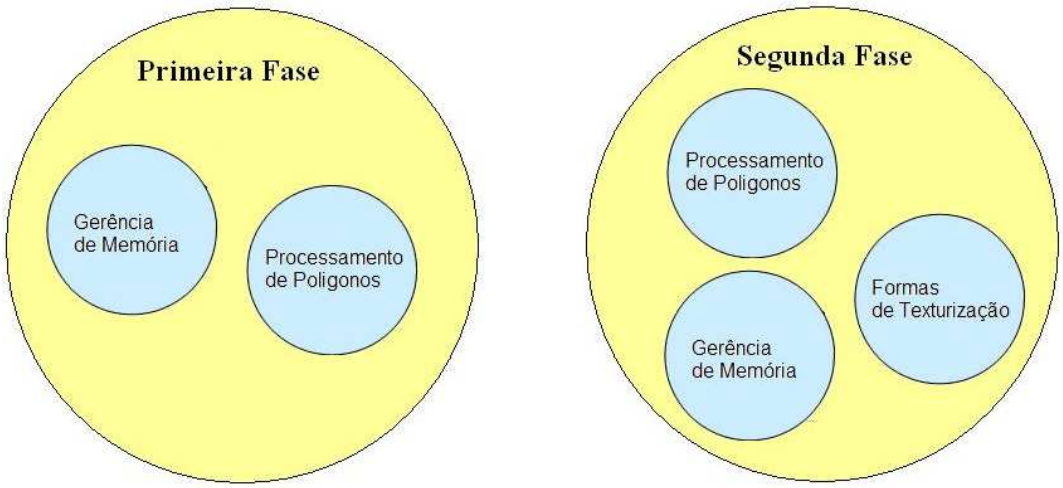


Figura 40 – Diagrama das fases do algoritmo e suas etapas de visualização.

A primeira fase tem por objetivo gerar um arquivo que contém, de forma organizada, todas as informações que descrevem o terreno. As geometrias, além dos diferentes LODs, mapas e texturas, são processadas formando uma *quadtree* estruturada por regiões, conforme citado na seção 3.4.2.

A segunda fase utiliza o arquivo de dados proveniente da primeira fase e o organiza nas diferentes memórias do computador (disco rígido, CPU e GPU) onde será realizada a visualização. Esta fase é responsável por gerenciar processo de exibição das regiões que se encontram no campo de visão do observador e realizar as trocas de LODs.

Para a implementação do algoritmo apresentado a seguir foi utilizada a linguagem C++ e o código foi compilado utilizando o *Microsoft Visual Studio Express 2008*[26]. A biblioteca gráfica adotada para a exibição dos terrenos foi a *DirectX* versão 9.0c.

4.1 Organização da Estrutura de Dados

Esta primeira fase consiste em um pré-processamento dos dados a serem exibidos e recebe como entrada os seguintes dados de descrição do terreno: mapa de altura, mapas de alfa, mapas de normal e texturas em ladrilho. Os formatos de mapas de altura suportados atualmente são BMP (8 bits) e RAW (16 bits) e as descrições do terreno utilizadas foram geradas com a ferramenta L3DT[27]. Escolheu-se esta ferramenta, pois a mesma é de fácil utilização e capaz de gerar todos os dados de entrada que precisamos (os mapas de altura, alfa e a textura em ladrilho). Além disto, a licença da ferramenta estava disponível para os autores do projeto desde antes do inicio deste trabalho.

Resumidamente o algoritmo de exibição de terrenos desenvolvido neste trabalho inicia filtrando o mapa de altura em busca de vértices removíveis, criando-se um mapa de booleanos que descreve a possível simplificação a ser realizada. Uma *quadtree* armazenando todo o terreno é criada, este terreno é gerado utilizando o mapa de booleanos e executando a simplificação dos vértices. A árvore armazena estes vértices junto aos índices, texturas e aos diferentes volumes de LODs da geometria do terreno em um arquivo no final desta fase.

Esta fase, pela inviabilidade de ser executado em tempo real, é realizado somente na criação do terreno, entretanto futuras modificações no relevo e visualizações do mesmo não necessitam repetir esta parte por completo. Aplicações como jogos MMORPG (*Massively Multiplayer Online*

---

[26] http://www.microsoft.com/Express/

[27] http://www.bundysoft.com/L3DT/

Role-Playing Game) que utilizam LSTs podem gerar o terreno e distribuí-lo, na necessidade de futuras alterações no terreno podem ser feitos pacotes contendo somente alguns nodos com novas geometrias.

Esta abordagem de substituição de nodos é possível porque o algoritmo utiliza a estrutura de árvore, que armazena a geometria por blocos em cada um dos seus nodos. Desta maneira existe a possibilidade de alterar somente um nodo, sem modificar os outros nodos da árvore.

É importante salientar que é possível, inclusive, que um nodo seja substituído por uma geometria que não tenha sido criada a partir do algoritmo aqui apresentado. Neste caso, porém, este nodo não terá as otimizações detalhadas nas operações de **Filtragem do Mapa de Altura**, **Criação da *Quadtree*** e **Armazenamento em Memória** que estão descritas nas seções a seguir.

4.1.1 Filtragem do Mapa de Altura

Nesta operação o objetivo é reduzir o número de vértices e índices utilizados para representar a geometria do terreno, buscando manter sua forma e aparência. Esta redução facilitará o **Processamento de Polígonos** e a **Gerência de Memória**.

O mapa de altura é analisado de acordo com uma margem de erro a fim de identificar vértices com possibilidade de simplificação, passíveis de remoção posteriormente. Esta analise é realizada em blocos de nove vértices (conjunto de 3 x 3 alturas) onde o vértice central (E) é candidato a simplificação, conforme a **Figura 41** ilustra.

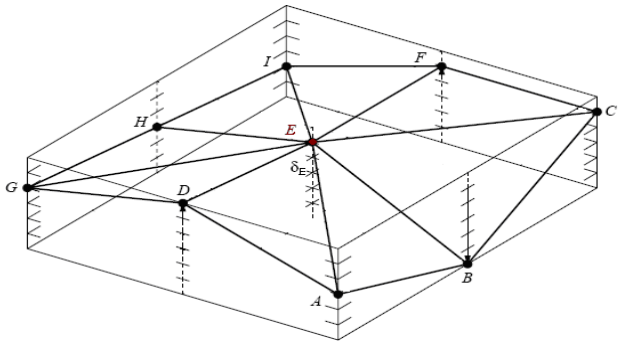


Figura 41 – Comparação de alturas de um setor.

Para proceder à simplificação, a altura do vértice central é comparada com a altura dos vértices que o cercam, de forma semelhante ao algoritmo de CLOD [17]. A altura média dos quatro conjuntos de vértices opostos (por exemplo, A e I ou B e H) é calculada. Cada valor é comparado com a altura do vértice central (E), verificando se a diferença obtida é inferior à margem de erro previamente especificada. Caso todas as diferenças alturas obtidas sejam inferiores, o vértice central (E) é marcado com o valor verdadeiro em uma matriz de booleanos, desta maneira sabemos que é um vértice passível de remoção, como ilustra o pseudocódigo da **Figura 42**.

```
SE( Modulo( (A + I) / 2 - E ) <= margemErro )
|    SE( Modulo( (B + H) / 2 - E ) <= margemErro )
|    |     SE(Modulo ( (C + G) / 2 - E ) <= margemErro )
|    |     |     SE(Modulo ( (F + D) / 2 - E ) <= margemErro )
|    |     |     |     matrizBooleana[E] = VERDADEIRO;
```

Figura 42 – Algoritmo para identificação de vértices passíveis de remoção.

Este algoritmo é executado para todos os vértices que não fazem parte das bordas do mapa de alturas, como ilustrado na **Figura 43** onde os vértices centrais encontram-se coloridos. Esta abordagem é importante em casos onde vários terrenos estão interligados, pois uma simplificação da borda poderia acarretar em brechas na geometria, como as mencionadas na seção 433.4.1.2.

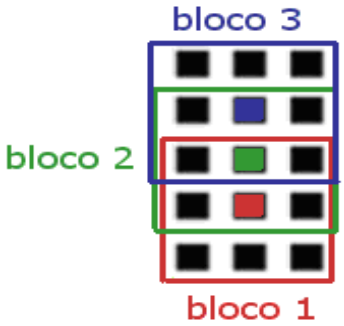


Figura 43 – Exemplo de blocos com vértices compartilhados.

Este trabalho utiliza terrenos onde as diferenças de altura estão distribuídas de forma equilibrada, não existindo grandes montanhas ou declives em regiões específicas do mapa de altura. Pela utilização destes mapas,

optou-se por fazer uso de uma margem de erro proveniente da diferença das alturas mínima e máxima do terreno. Utilizando, por exemplo, um terreno com a altura mínima de 3 metros e a máxima de 53 metros como exemplo, junto a uma variação de cinco por cento, resultaria uma margem de erro igual a 2.5 metros.

Caso o mapa de altura não possua seu relevo distribuído de forma equilibrada, ou seja, com uma maior concentração de relevos em uma parte do terreno, esta opção de margem de erro pode não ser adequada. Utilizando, por exemplo, as alturas mínima e máxima do terreno de 12 e 13.587 respectivamente, a margem de erro resultante seria 678,75 metros. Portanto, terrenos cujo relevo não é distribuído de forma equilibrada, acabam por sofrer uma simplificação de pior qualidade com esta margem e deveriam utilizar métrica. A **Figura 44** exemplifica este cenário.

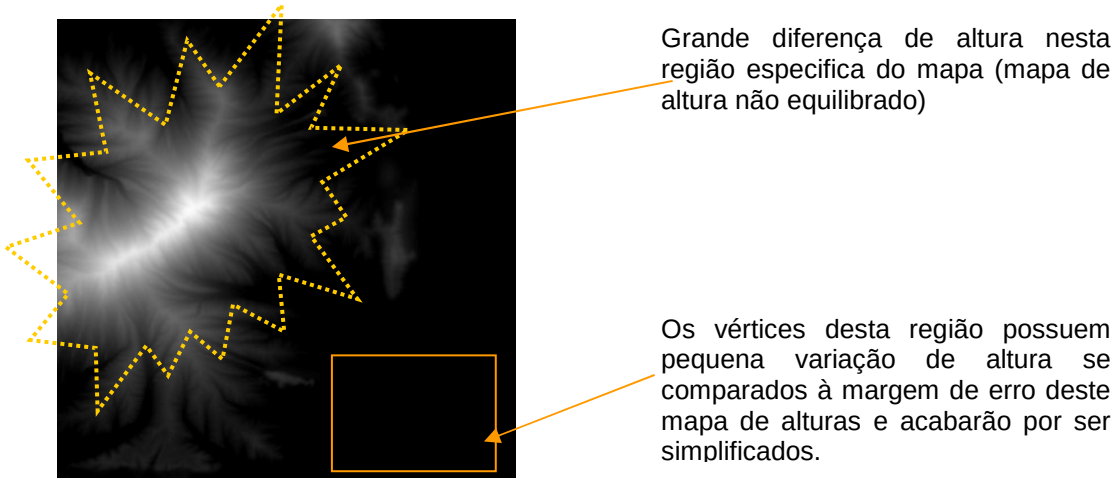


Figura 44 – Mapa de altura não equilibrado e a sua margem de erro.

Percorrer o mapa de altura em blocos e identificar os vértices a serem removidos é uma operação custosa computacionalmente. Por exemplo, em um mapa de altura com dimensões laterais de 8193 é realizada a verificação em mais de 67 milhões de vértices. Tais verificações utilizam os dados presentes no mapa de altura, requerendo que o mesmo já tenha sido carregado na memória principal do computador.

Visando acelerar esta etapa o algoritmo implementa a leitura do mapa

de altura de forma direta (lendo todo seu conteúdo em uma única chamada de arquivo). A filtragem do mapa é feita a partir da divisão do seu conteúdo em quatros partes que são processadas em paralelo. Além disto, um segundo método para a identificação dos vértices passíveis de remoção foi desenvolvido, realizando a operação de forma mais rápida através da diminuição do número de divisões e cálculos de módulo em cada bloco. O método simplifica a operação calculando uma média aproximada das alturas e comparando-as à margem de erro. Na **Figura 45** apresenta-se o pseudocódigo do método mencionado, seguido da **Tabela 1**, que compara o tempo de execução das duas soluções.

```
SE( Modulo( (A + I + B + H + C + G + F + D) / 8 - E )
|   <= margemErro )
|     matrizBooleana[E] = VERDADEIRO;
```

Figura 45 – Algoritmo de vértices passíveis de remoção.

Tabela 1 – Comparação de tempo de execução entre as soluções de filtragem em um mapa de altura com dimensões de 8193 x 8193.

Método utilizado	Tempo de execução (segundos)
Método de identificação original	10.31
Método de identificação otimizado e processamento paralelo	4.70

Na primeira solução (1) desenvolvida para este algoritmo cada valor do mapa de altura armazenado ocupava 4 bytes, totalizando um total de 256MB para um mapa de 8193 x 8193. Em face disto, decidiu-se pela troca do tipo ponto flutuante (*float*) para *unsigned short*, que ocupa 2 bytes, reduzindo pela metade o espaço ocupado pelas alturas. Esta medida, entretanto, não foi suficiente para que este trabalho executasse em um ambiente com 2GB de memória RAM, por isto foram necessárias outras otimizações que estão explicadas neste Capítulo.

Em razão do endereçamento de memória da maioria das plataformas e

compiladores não permitir gerenciar os ponteiros de memória com representação de um *bit*, um booleano ocupa o equivalente a um *char* (1 *byte*) em memória. Por isto, uma otimização foi realizada no mapa de booleanos utilizando à classe *dynamic_bitset*, proveniente da biblioteca ***Boost***[28]. Com esta biblioteca, o espaço ocupado por um booleano passou a ser de fato um *bit*. Com isto, foi diminuído em oito vezes o espaço ocupado pelo mapa de booleanos em memória. A **Figura 46** ilustra as diferenças da memória ocupada pelos mapas após as modificações mencionadas (2).

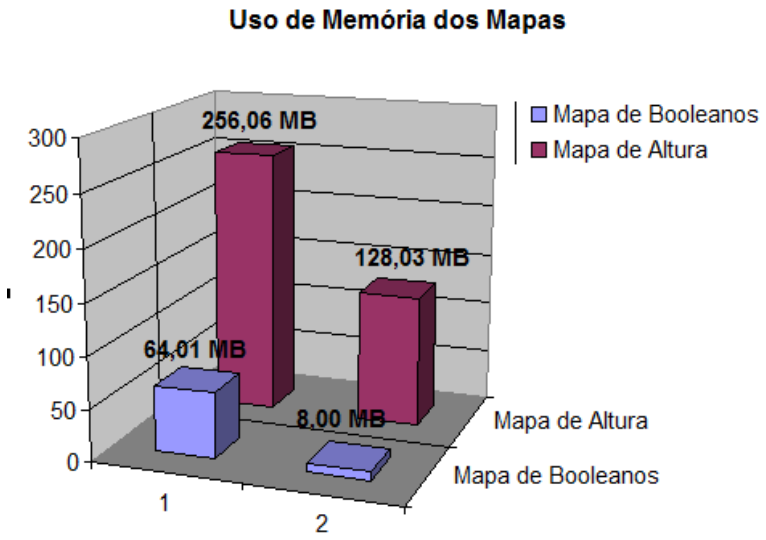


Figura 46 – Comparação do uso de memória dos mapas.

4.1.2 Criação da *Quadtree*

Nesta operação o mapa de altura e a matriz de booleanos, gerada anteriormente, são utilizados para criar uma árvore que represente o terreno e que seja armazenada em disco. Esta árvore é uma *quadtree* de forma que os nodos do nível inferior (N) compartilham os vértices de suas bordas. Estes nodos possuem dimensões de 33x33 e áreas iguais, como mostra a **Figura 47**.

[28] http://www.boost.org/

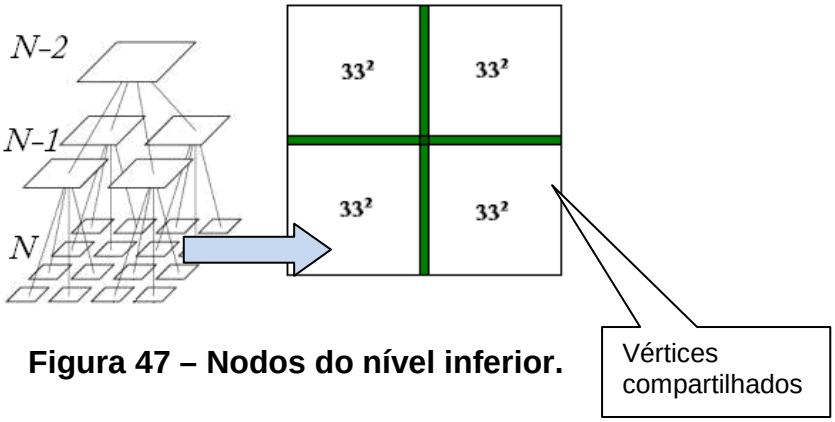


Figura 47 – Nodos do nível inferior.

O compartilhamento de vértices das bordas de cada nodo evita que a geometria sofra alterações nos processos de simplificações, além de garantir que os nodos se liguem corretamente evitando problemas de brechas. As bordas inalteradas também eliminam a necessidade de realizar a triangularização em tempo real entre os diferentes nodos e as diferentes resoluções da geometria.

Em uma primeira solução, acreditou-se que seriam criados todos os nodos do nível N da árvore, possuindo as descrições geométricas das regiões do terreno, e então os mesmos seriam armazenados em um arquivo. Foi descoberto que tal solução é inviável já que a quantidade de informações geradas somava 3GB de memória, não podendo ser armazenadas na memória da máquina utilizada neste trabalho.

Na solução adotada a criação da *quadtree* utiliza os mapas de altura e de booleanos para gerar, inicialmente, uma estrutura denominada “pseudo-árvore”. A pseudo-árvore é uma quadtree semelhante à árvore recém descrita, mas com diferença nos nodos inferiores, que não possuem descrições geométricas e também não armazenam as caixas envolventes[29] (*bounding boxes*). Estes nodos da pseudo-árvore armazenam apenas as informações da posição dos vértices (x, y e z) e de um dado que define o peso que cada vértice possui na estrutura do terreno.

[29] Cubo envolvendo uma geometria utilizada para representar de forma simplificada o volume da mesma.

O peso associado a cada vértice está diretamente relacionado com sua posição nos nodos e seu valor no mapa de booleanos. Quanto mais próximo da borda do nodo, maior é o peso do vértice. Os vértices marcados com valor verdadeiro no mapa são associados a um peso zero. Desta maneira sabem-se quais são os vértices que possuem maior relevância no momento de se realizar a simplificação com o algoritmo de *progressive meshes*. Uma ilustração das geometrias de um grupo de nodos, seus respectivos LODs, com destaque para as bordas não simplificadas é apresentada na **Figura 48**.

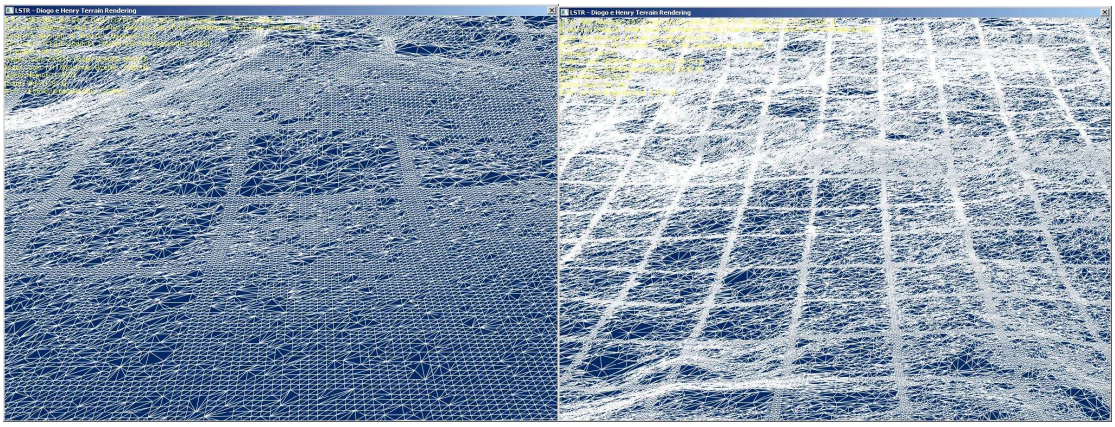


Figura 48 – Geometria dos nodos com bordas compartilhadas.

Examinando as dimensões dos nodos da árvore, um nodo do nível *N*-1 da árvore, e da pseudo-árvore, corresponde a uma área equivalente a quatro nodos do nível *N*. Entretanto, por compartilharem suas bordas, a dimensão de seus pais não segue a mesma proporção. Já que um nodo no nível N possui dimensões de 33 x 33, um nodo localizado no nível *N*-1 possui dimensões de 65 x 65 e o nodo do nível *N*-2 tem a dimensão de 129 x 129.

A **Figura 49** é uma representação dos três últimos níveis de uma árvore cujos nodos do nível N possuem dimensões de 3x3, demonstrando que os pais dos nodos de 33 x 33 possuem a dimensão de 65 x 65. O nodo pai possui esta dimensão, e não à de 66 x 66, pois as bordas de seus filhos representam o mesmo espaço. O mesmo comportamento ocorre para os nodos de 129 x 129 em relação aos seus filhos, nodos de 65 x 65.

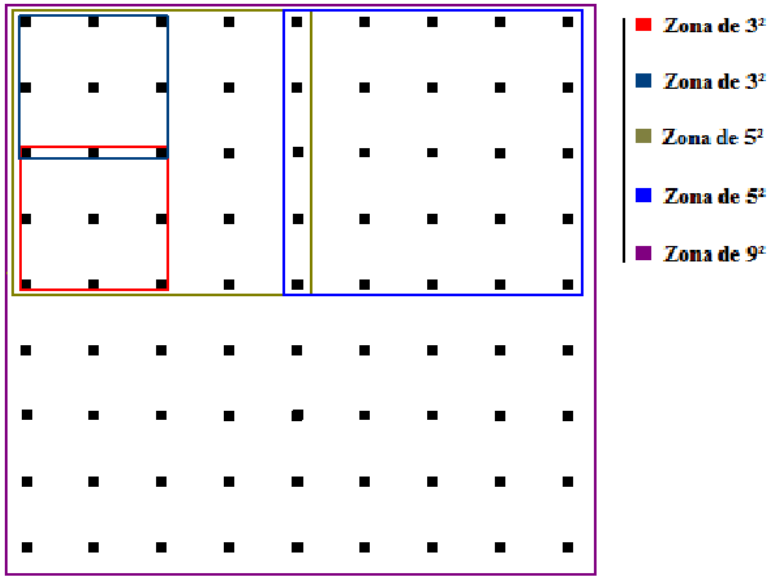


Figura 49 – Representação dos nodos e suas proporções.

No que se refere ao armazenamento de dados na árvore, os dois últimos níveis (*N e N-1*) armazenam informações sobre os índices da geometria. Os nodos do nível anterior (*N-2*), além dos índices, armazenam os vértices da geometria, o mapa de alfa e o mapa de normal. Os mapas de norma e de alfa incrementam a qualidade de exibição do terreno como é abordado na seção 3.3.2.

O nodo do nível inferior, *N*-1, representa a geometria de seus filhos e identifica um novo conjunto de bordas. Esta nova geometria, exceto pelas bordas, é simplificada pelo algoritmo apresentado na seção 3.4.5. Esta simplificação busca representar a geometria dos filhos do nodo com um número de polígonos inferior aos originais. A partir desta simplificação é gerado o conjunto de índices que representa a geometria com este LOD inferior.

Nos nodos do nível *N*-2 a simplificação segue a mesma lógica, também resultando em um LOD que represente seus filhos com um número reduzido de polígonos. Todos os nodos da árvore armazenam cubos que representam a *bounding box*, informação utilizada pela fase de **Exibição do LST em Tempo Real** do algoritmo na realização do *culling*.

Os vértices e índices que compõe a geometria são armazenados em *vertex buffers*[30] e *index buffers*[31]. A solução utiliza um número menor de buffers para representar a geometria, já que somente os nodos no nível *N*-2 possuem as informações dos vértices, reduzindo o número de *drawcalls*,.

Os índices dos nodos do nível *N* e *N*-1 são armazenados pelos nodos no nível *N*-2, reduzindo o número de *index buffers* necessários. Além disto, nodos do nível *N* e *N-1* contem a localização de seus índices (*offsets)* no *index buffer*. Esta estrutura esta exemplificada na **Figura 50**. A raiz da *quadtree*, além de possuir a informação da *bounding box* de todo o terreno, armazena informações sobre as texturas que o terreno utiliza.

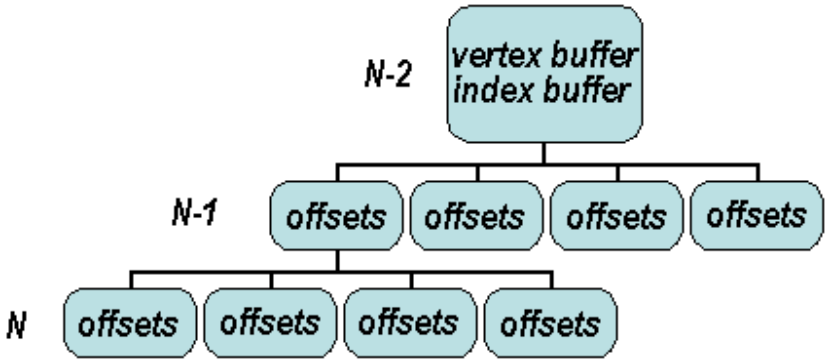


Figura 50 – Estrutura do conteúdo dos nodos por nível.

À medida que a árvore é construída também é armazenada em um arquivo. No final desta operação o terreno de dimensões 8193 x 8193 gerado neste trabalho alcançou 2GB de tamanho, seguem detalhes sobre o **Armazenamento em Memória**.

---

[30] Estrutura que armazena um conjunto de índices na memória de forma compartilhada, permitindo a utilização dos mesmos pela placa de vídeo e pela aplicação. Mais informações estão disponíveis na documentação do *DirectX:* http://msdn.microsoft.com/en-us/library/bb205133(VS.85).aspx#Index_Buffer

[31] Estrutura responsável pelo armazenamento de vértices com comportamento semelhante ao index buffer. Segue maiores informações na documentação do *DirectX:* http://msdn.microsoft.com/en-us/library/bb205133(VS.85).aspx#Vertex_Buffer

4.1.3 Armazenamento em Memória

A árvore obtida a partir das operações descritas na seção 4.1.2 possui todas as informações necessárias para exibir o LST. Além disto, ao armazená-la em arquivo, outras informações são adicionadas para que a mesma possa ser carregada com menos custo computacional. Estas informações são referentes, em sua maioria, à posição dos nodos no arquivo.

Cada nodo armazenado possui, além da localização de seus respectivos pais, a localização de seus filhos no arquivo. Com isto, a *quadtree* pode ser carregada progressivamente de diferentes pontos de partidas (posição da câmera), conforme os nodos necessários na exibição, facilitando o *swap* dos dados entre as diferentes memórias na fase de **Exibição do LST em Tempo Real**.

As texturas e os mapas utilizados pelo LST são armazenados em arquivos diferentes ao da árvore. Esta abordagem permite que estes dados não estejam presos ao arquivo do terreno, facilitando a manipulação dos mesmos. Além disto, com o objetivo de reduzir o tamanho final do arquivo e aproveitar melhor a memória disponível na placa de vídeo os índices foram ordenados em *triangle strips* [26].

A solução (A) descrita nesta fase, envolvendo processamento paralelo junto à pseudo-árvore, não suportou a geração do arquivo. Mesmo utilizando a biblioteca **Boost** o consumo da memória RAM ultrapassava o limite de 2GB RAM e o tempo estimado para o término do arquivo era superior a 47 horas.

A solução adotada (B) foi gerar um esqueleto da árvore no arquivo. Os nodos do nível N-2 eram armazenados no arquivo logo após terem sido gerados, liberando espaço em memória. Estes nodos foram escritos em conjuntos de 4 nodos de forma recursiva, ou seja, o esqueleto da árvore existente no arquivo é preenchido dos níveis N até a raiz. A **Tabela 2** mostra a memória ocupada, junto ao tempo de execução, nesta fase pelas duas soluções.

Tabela 2 – Comparação de memória e tempo entre as soluções realizadas.

Solução	Memória ocupada (MB)	Tempo gasto (horas)
Solução A	2602.26	13
Solução B	656.06	3

4.2 Exibição do LST em Tempo Real

Esta fase inicia com a realização de um diagnóstico no computador em que a aplicação está executando. São analisadas informações necessárias para o algoritmo, como a disponibilidade das memórias da CPU, do disco rígido e da GPU. Com as informações diagnosticadas, o arquivo contendo a estrutura do terreno é carregado.

A partir da localização do observador no terreno, da estrutura do arquivo e do campo de visão do observador uma esfera de *swap* é criada. Esta esfera possui como centro a posição do observador e como raio um comprimento 20% maior do que o campo de visão.

Todos os nodos dentro de esfera de *swap* são armazenados na memória da CPU e aqueles dentro do campo de visão são armazenados na memória da GPU. Em relação aos demais nodos é guardada em memória apenas sua localização no arquivo. A esfera, o campo de visão e os nodos estão exemplificados na **Figura 51**.

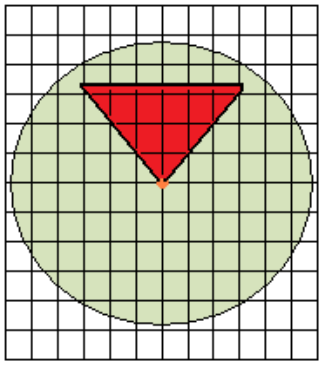


Figura 51 – Esfera de *swap* centralizada no observador envolvendo o campo de visão.

A esfera criada está sempre centralizada no observador, acompanhando o mesmo quadro a quadro na execução da aplicação. A esfera também é proporcional ao comprimento do campo de visão, portanto, se este tiver a sua distância mínima ou máxima modificada, a mesma é redimensionada.

Quando um nodo entra na esfera, sua localização no arquivo é utilizada para realizar a leitura de suas informações: os *buffers* e texturas. Com as memórias armazenadas da forma adequada, procedimento realizado paralelamente à exibição, o algoritmo percorre a árvore identificando os nodos que serão exibidos. A seguir, são apresentados detalhes sobre a operação de varredura e exibição dos nodos.

4.2.1 Percorrendo e Exibindo os Nodos

Através das informações do campo de visão do observador, a árvore é percorrida da raiz em direção aos nodos-folha em busca de nodos neste campo. A *bounding box* armazenada em cada nodo é testada, verificando se o mesmo está contido, ou possui uma interseção com o campo de visão.

Todos os nodos do nível N-2, cuja verificação foi verdadeira, tem sua posição comparada com a do observador para obter uma distância entre os mesmos. Esta distância é utilizada para definir o LOD apropriado para cada nodo, aqueles que ficarem com um LOD inferior representam uma área maior de terreno, pois quanto maior a distância do observador, maior a área visualizada.

O LOD selecionado determina se o nodo exibido será o do nível N-2, um ou mais dos quatros nodos do nível N-1 ou um ou mais dos 16 nodos do nível N. Estes nodos, quando selecionados, são armazenados em um vetor que representa todas as geometrias a serem exibidas. Este vetor é organizado utilizando o algoritmo de *quicksort* para garantir que as geometrias sejam exibidas de perto para longe, aproveitando o recurso do *Z-Buffer* presente na placa de vídeo. A **Tabela 3** abaixo mostra uma comparação do algoritmo habilitando e desabilitando o *quicksort*.

Tabela 3 – Comparação do algoritmo com e sem o uso do *quicksort*.

Quicksort	**Quadros por segundo**
Habilitado	2602.26
Desabilitado	656.06

A exibição da geometria é realizada utilizando o *pipeline* gráfico programável através de um código *shader*, disponível no **Apêndice I**, desenvolvido em HLSL. Este código, através das funções de *pixel shader* e *vertex shader*, projeta a geometria no campo de visão do observador. Este código também é responsável pela texturização das quatro imagens utilizadas, conforme o mapa alfa que cada nodo possui.

O código *shader,* além da texturização, realiza a iluminação da geometria de cada nodo utilizando os dados do mapa de normal e a posição de uma fonte de luz direcional. A posição da luz pode ser alterada, em tempo real, de maneira que simule ciclos de dias e noites. A **Figura 52** ilustra o *shader* descrito, outras imagens estão disponíveis no **Apêndice II**.

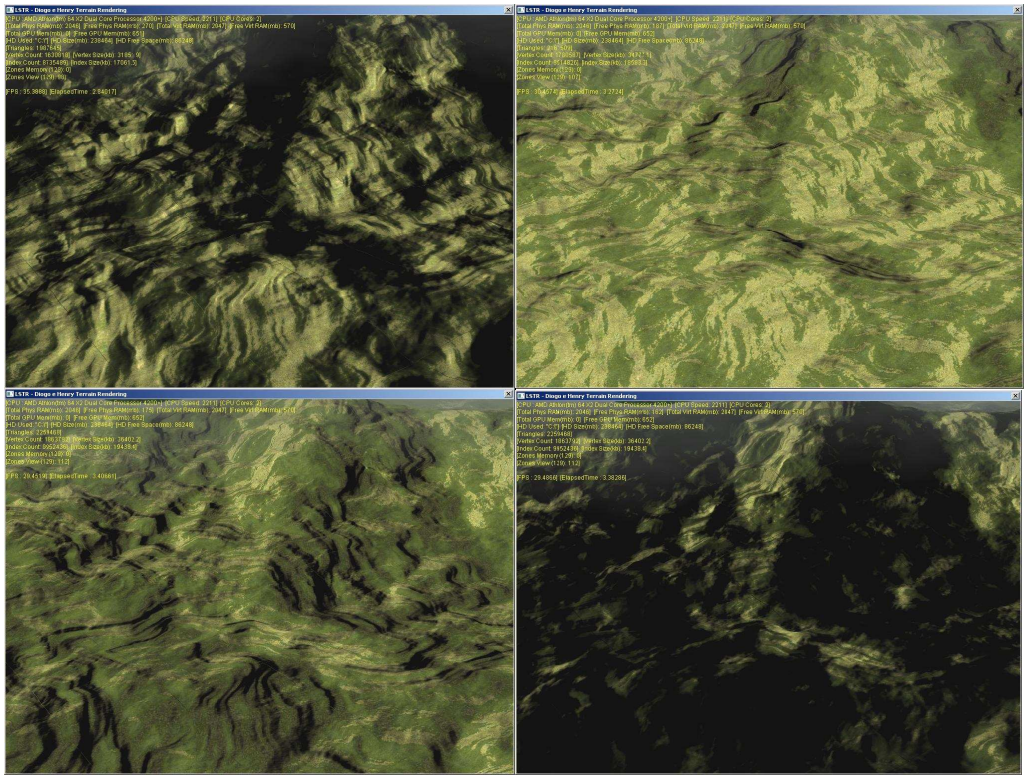


Figura 52 – Resultado obtido através do código *shader*.

5. Testes de Desempenho

Com o algoritmo proposto implementado, foi possível realizar certas provas para medir seu desempenho. Cada prova utilizou um critério de avaliação diferente, como por exemplo, o tamanho do campo de visão. As provas de desempenho possuem dois objetivos: comparar o algoritmo com outros existentes e testar o comportamento do mesmo de acordo com diferentes critérios.

Para o primeiro objetivo foram utilizados os exibidores de terrenos existentes em dois motores gráficos (*graphic engines*): Irrlicht[32] e Ogre3d. Para o segundo objetivo foram utilizados diferentes critérios de avaliação procurando atingir o limite de capacidade do algoritmo, de forma que este ainda executasse em tempo real. Seguem a metodologia de teste empregada, resultados obtidos e comparações realizadas.

5.1 Metodologia de Teste

Para o primeiro objetivo foram selecionados como provas o tamanho do mapa de altura e a distância máxima do campo de visão do observador. Estas provas avaliam os seguintes critérios: número de quadros por segundo (QPS), número de triângulos exibidos (NTE), espaço ocupado em memória em KB (EOM) e d*rawcalls* (DC).

O segundo objetivo utiliza como prova o tamanho da esfera de *swap* e adiciona dois critérios: número de zonas em memória (NZM) e numero de zonas exibidas (NZE). Para a apresentação mais simplificada, os critérios estão organizados em tabelas para cada uma das provas.

A prova referente ao tamanho do mapa de altura foi realizada com um mapa de altura de dimensões de 8193 x 8193 no LSTR. Este mapa teve sua exibição cortada para formar os mapas de dimensões menores devido à restrição de mapas grandes nos motores gráficos utilizados. Estes motores

[32] http://irrlicht.sourceforge.net/

utilizaram outros mapas para representar os tamanhos de 1024 e 2048.

A prova do tamanho da esfera de *swap* utiliza um mapa de altura com dimensões de 8193 x 8193. Este tamanho do mapa é importante para testar o número de zonas dentro do raio da esfera de *swap*. É importante ressaltar que a esfera não é sempre preenchida por completo, já que o observador pode estar posicionado próximo aos limites do terreno.

Para a realização das provas o código foi proveniente da API versão 1.0, gerada neste trabalho e disponível no **Apêndice III**. O compilador utilizado é o padrão encontrado no *Visual Studio* 2008 *Express*[33]. A máquina para execução das provas possui a seguinte configuração: processador Intel Core 2 Duo 6600, 2GB de RAM, placa de vídeo GeForce 8800 320MB RAM, sistema operacional Windows XP Home.

5.2 Resultados e Comparações

Na **Tabela 4** são apresentados os resultados da prova referente ao tamanho do mapa de altura. Os motores gráficos utilizados realizaram a exibição contendo duas texturas e não conseguiram exibir o mapa de tamanho 8193. O LSTR realizou a exibição utilizando seis texturas e, de acordo com os critérios adotados, superou os dois motores nos mapas selecionados para a prova.

É importante ressaltar a diferença nos NTEs que representam o terreno, o valor alto percebido no LSTR consome mais processamento na GPU, mas tende a melhorar a qualidade visual do terreno. Outra diferença percebida entre os exibidores é número de DCs realizados. O valor alto de DCs encontrado na Ogre3d é o reflexo do número de zonas que a mesma utiliza para representar o terreno. Este número, porem, facilita o controle de LOD e *culling*, diminuindo o NTEs.

---

[33] http://www.microsoft.com/Express/

Tabela 4 – Prova do tamanho do mapa de altura.

Mapa de Altura	LSTR				Irrlicht				Ogre3d			
	QPS	NTE	EOM	DC	QPS	NTE	EOM	DC	QPS	NTE	EOM	DC
1024	354	499.414	119.220	31	71	148.448	263.232	65	166	13.006	170.368	741
2048	186	2.012.288	198.616	97	30	329.312	508.768	170	42	384.614	386.556	2534
8193	162	2.130.093	652.900	105	x	x	x	x	x	x	x	x

A segunda prova consistiu em comparar o LSTR com os demais motores gráficos analisando a distância do campo de visão. Cabe ressaltar que esta prova foi aplicada para medir a relação entre a distância do campo de visão com o número de DCs e NTE. Em face de que os motores gráficos não suportaram mapas com dimensões acima de 2048 x 2048, não foi possível replicar o mesmo mapa. A **Tabela 5** mostra os resultados obtidos neste teste.

Tabela 5 – Prova da distância do campo de visão.

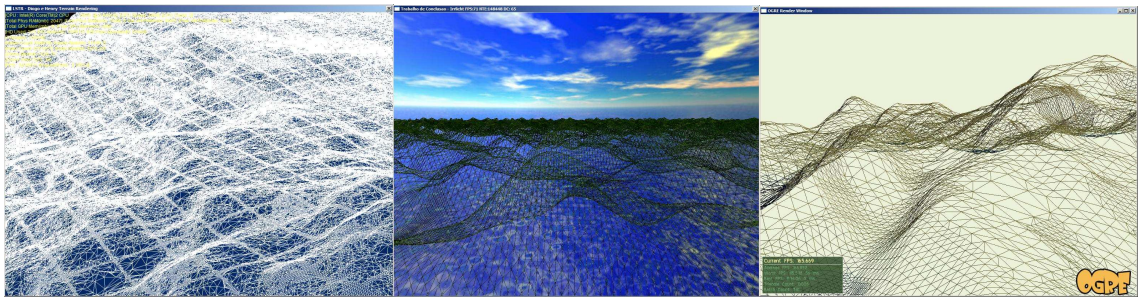
Campo de Visão	LSTR				Irrlicht				Ogre3d			
	QPS	NTE	EOM	DC	QPS	NTE	EOM	DC	QPS	NTE	EOM	DC
500	625	240.756	42.368	15	176	46.048	263.232	17	574	16.432	170.368	175
1500	354	499.414	119.220	31	70	149.984	263.232	65	166	13.006	170.368	741

Por fim, foi realizado um teste de desempenho do LSTR, onde foi alterado raio da esfera de *swap*. Quando maior o raio, maior o número de zonas que o LSTR deve carregar em memória, a **Tabela 6** mostra resultados desta prova.

Tabela 6 – Prova do raio da esfera de *swap*.

Esfera de *Swap*	LSTR					
	NZE	NZM	QPS	NTE	EOM	DC
1150	31	151	363	633.094	131.716	31
2230	105	913	162	2.130.093	652.900	105
3450	218	1075	100	4.487.210	757.284	218

Para ilustrar as provas aplicadas, segue na **Figura 53** imagens dos exibidores em execução. Estas imagens foram retiradas da exibição de um mapa de altura representado com dimensões de 1024 x 1024, com o objetivo de mostrar os triângulos e os LODs formados por cada exibidor. As imagens, por serem estáticas, não ilustram as deformações que ocorrem na Irrlicht (b) e na Ogre3d (c). O LSTR (a), não apresentou tais deformações.



(a) **(b)** **(c)**

Figura 53 – Imagens dos exibidores em execução.

6. Conclusões e Futuros Trabalhos

O mercado de jogos está em constante expansão e cada vez mais, novas tecnologias são empregadas para atrair e aumentar as comunidades de jogadores. Na linha destas tecnologias, este trabalho mostrou uma nova abordagem sobre a exibição de terrenos em tempo real, cujo objetivo principal é processar e exibir, em tempo real, um terreno com grande quantidade de geometria sem perder sua qualidade visual.

Para este trabalho foram estudados algoritmos de exibição de terrenos que contemplavam técnicas de **Processamento de Polígonos**, **Gerência de Memória** e **Formas de Texturização**. No decorrer do desenvolvimento da solução, realizada com o *SDK* do *DirectX,* ocorreram inúmeros problemas, como brechas e deformações na geometria, dificultando chegar no objetivo proposto. Mesmo com estes problemas solucionados, permaneceu a possibilidade de elaboração de um algoritmo que não acarretasse tais problemas na geração de LODs muito baixos.

A implementação deste trabalho foi realizada em dois programas separados. O primeiro realiza a fase de **Organização da Estrutura de Dados** e o segundo é responsável pela fase de **Exibição do LST em Tempo Real**.

O primeiro programa cria os LODs e a triangularização em *stripes*, a partir de recursos existentes no SDK do *DirectX*. Estes recursos poderiam ser substituídos por uma função mais rápida que permitisse LODs de menor detalhe. Na solução atual, os LODs com detalhes muito reduzidos não são utilizados, pois podem acarretar brechas ou deformações na geometria do terreno.

O arquivo gerado pelo primeiro programa, para um mapa de altura de dimensões de 8193 x 8193, ocupa 2GB de espaço no disco rígido. Futuramente pretende-se reduzir este espaço através de uma forma de compactação. A compactação também teria o objetivo de aproveitar melhor o espaço da memória da CPU, já que a mesma poderia armazenar um número

maior de nodos. Para efeito de ilustração, um dos arquivos gerados, quando compactado como o software WinRar[34] ocupou 19% de seu tamanho original.

O segundo programa, que realiza *swap* dos nodos entre as memórias e exibe de fato o terreno, pode ter a primeira tarefa (swap) otimizada, diminuindo-se o número de trocas de nodos em memória através de algum algoritmo para predição dos movimentos do observador. A troca de texturas em memória, dos mapas de alfa e de normal, também poderia ser otimizada utilizando os nodos do nível *N*-3 para agrupar estas texturas.

A estrutura em árvore empregada também poderia servir para o *culling* dos objetos presentes no terreno, utilizando-se os nodos para armazenar outras geometrias. Os nodos também poderiam ser utilizados para armazenar informações adicionais para a execução de algoritmos de inteligência artificial ou de física, por exemplo.

Para trabalhos futuros também se pretende adicionar pacotes de física junto ao terreno, desta maneira deformações em tempo real e testes de colisões poderiam ser feitos. Testes de física inseridos no terreno viabilizariam outras abordagens, valorizando o trabalho realizado até então.

Após as melhorias supramencionadas o algoritmo desenvolvido será comparado com os algoritmos citados na seção 3.4 utilizando os dados de tamanho do terreno, a taxa de quadros por segundo alcançada, a memória total utilizada e memória ocupada por cada vértice, encontrados no artigo *Geometry ClipMaps* [12]. Os dados atualizados sobre novas implementações e testes serão disponíveis pelo *website* em desenvolvimento *Keep Under Dev*[35] (KUP).

---

[34] http://www.win-rar.com/

[35] http://www.keepunderdev.com

7. Referencias Bibliográficas

1. PriceWaterHouse Coopers. Disponível em: http://www.pwc.com. Consultado em 3 de abril de 2008.

2. STEINER, Thomas. *Advertising in Online Games and EC Audiovisual Media Regulation.* NCCR Trade Working Paper. Janeiro 2008.

3. Hickling Arthurs Low, *Entertainment Software - The Industry in Canada*, 2007.

4. BANGEMAN, Eric., *Growth of gaming in 2007 far outpaces movies, music*, 2008, http://arstechnica.com/news.ars/post/20080124-growth-of-gaming-in-2007-far-outpaces-movies-music.html, consultado em 3 de abril de 2008.

5. Discovery Channel, *Rise of the Video Game*, episódio quatro, 2007. http://dsc.discovery.com/tv/video-game/video-game.html.

6. MCGREGOR, L. Georgia, *Architecture, Space and Gameplay in World of Warcraft and Battle for Middle Earth 2*, 2006, ACM International Conference Proceeding Series, Vol. 223, p. 69-76.

7. RÖTTGER Stefan., HEIDRICH, Wolfgan., SLUSSALLEK, Philipp., SEIDEL, Hans-Peter. P., *Real-Time Generation of Continuous Levels of Detail for Height Fields*, 1999, Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization, p. 315-322.

8. LUNA D. Frank., *Introduction to 3D Game Programming With DirectX 9.0c A Shader Approach*, 2006 Wordware Publishing, Inc.

9. SCHNEIDER, Jens., WESTERMANN, Rüdiger., *GPU-friendly high-quality terrain rendering*, 2006, Journal of WSCG, vol. 14.

10. LARSEN, D.Bent, CHRISTENSEN, J. Niels, 2003. *Real-time terrain rendering using smooth hardware optimized level of detail*. Journal of WSCG 11, 2, 282--9. WSCG'2003: 11th International Conference in Central Europe on Computer Graphics, Visualization and Digital Interactive Media.

11. GUTHE, Stefan., HECKBERT, Paul., *Non-Power-Of-Two Mipmap Creation*, 2003.

12. LOSASSO, Frank. HOPPE, Hugues. 2004. *Geometry clipmaps: terrain rendering using nested regular grids*. ACM Transactions on Graphics 23, p. 769-776.

13. ULRICH, Thatcher. *Rendering Massive Terrains using Chunked Level of Detail Control*. SIGGRAPH 2002 Course Notes, SIGGRAPH-ACM publication, San Antonio, Texas.

14. BLOW, Jonathan. *Terrain Rendering at High Levels of Detail*. Proceedings of the 2000 Game Developers Conference. Mar. 2000.

15. CLARK H. James. *Hierarchical Geometric Models for Visible Surface Algorithms*. 1976. Communications of the ACM, Volume 19 Number 10. Pages 547-554.

16. ASIRVATHAM, Arul. HOPPE, Hugues. 2005. *Terrain rendering using gpu-based geometry clipmaps*. In GPU Gems 2, Addison-Wesley, M. Pharr and R. Fernando, p. 27-45.

17. LINDSTROM, Peter., KOLLER, David., RIBARSKY, William., HODGES, F. Larry, FAUST, Nick., TURNER, A. Gregory. 1996. *Real-Time, continuous level of detail rendering of height fields*. In SIGGRAPH 96 Conference Proceedings, Addison Wesley, H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, p.109-118.

18. SALVATOR, Dave. 2001. *ExtremeTech 3D Pipeline Tutorial: A Deep Dive into the Pipeline Stages.*

19. PAJAROLA, Renato., GOBBETII, Enrico., 2007, *Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering*. In The Visual Computer, p. 583-605.

20. HOPPE, Hugues. *Progressive Meshes. SIGGRAPH* 1996*, 99-108.*

21. HOPPE, Hugues. *View-Dependent Refinement Progressive Meshes*. SIGGRAPH 1997, 189-198.

22. HOPPE, Hugues., DEROSE, Tony., DUCHAMP, Tom., MCDONALD, John., STUETZLE, Wemer. *Mesh Optimization*. *University of Washington*, Seattle.

23. FOLEY, James. et. al; Computer graphics: principles and pactise. 2nd ed. New York: Addison Wesley, 1990.

24. KESSENICH, John., *The OpenGL Shading Language*. 2007.

25. COHEN, Marcelo., MANSSOUR, H. Isabel. OpenGL: uma abordagem prática e objetiva. 2006. NOVATEC.

26. EVANS, Francine., SKIENA, Steven., VARSHNEY, Amitabh. *Optimizing Triangle Strips for Fast Rendering*. IEEE Visualization. 1996.

8. Apêndice I – Códigos *shader*

Para esclarecimento este apêndice apresenta os códigos *shader* utilizados pelo programa responsável pela exibição do terreno. O código para exibição do céu foi retirou dos exemplos de Frank Luna [8]. Todos os códigos estarão disponíveis no *website* do KUP.

```
//=============================================================================
// sky.fx by Frank Luna (C) 2004 All Rights Reserved.
//=============================================================================

uniform extern float4x4 gWVP;
uniform extern texture  gEnvMap;

sampler EnvMapS = sampler_state
{
    Texture   = <gEnvMap>;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
    MipFilter = LINEAR;
    AddressU  = WRAP;
    AddressV  = WRAP;
};


void SkyVS(float3 posL : POSITION0,
           out float4 oPosH : POSITION0,
           out float3 oEnvTex : TEXCOORD0)
{
    // Set z = w so that z/w = 1 (i.e., skydome always on far plane).
    oPosH = mul(float4(posL, 1.0f), gWVP).xyww;

    // Use skymesh vertex position, in local space, as index into cubemap.
    oEnvTex = posL;
}

float4 SkyPS(float3 envTex : TEXCOORD0) : COLOR
{
    return texCUBE(EnvMapS, envTex);
}

technique SkyTech
{
    pass P0
    {
        vertexShader = compile vs_2_0 SkyVS();
        pixelShader  = compile ps_2_0 SkyPS();

        CullMode = None;
        ZFunc = Always; // Always write sky to depth buffer
        StencilEnable = true;
        StencilFunc   = Always;
        StencilPass   = Replace;
        StencilRef    = 0; // clear to zero
    }
}
```

Figura 54 – Código *shader* para exibição do céu.

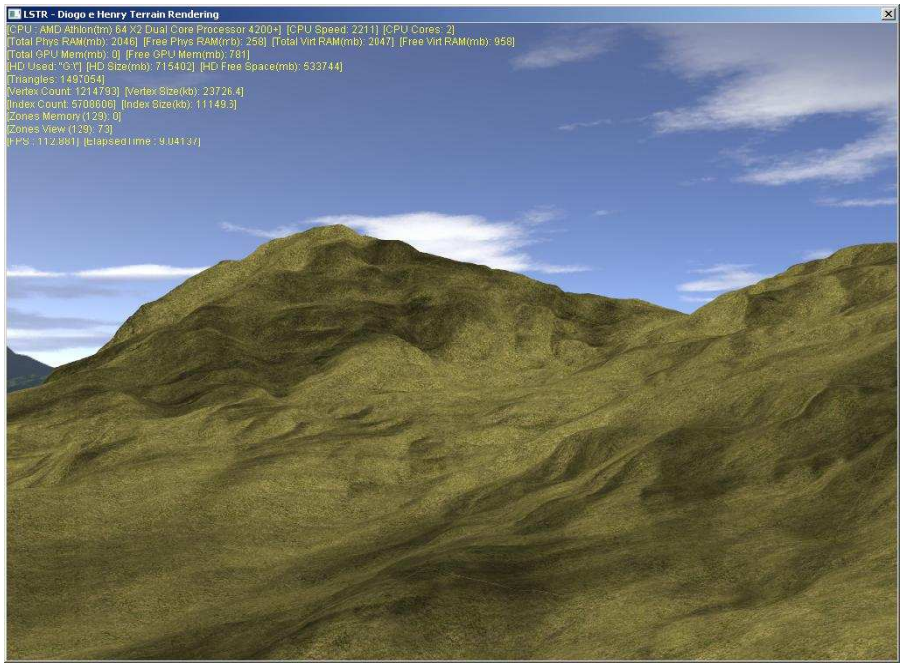
```
//=============================================================================================|
// TerrainEffect.fx by Diogo Strube de Lima & Henry Braun
//=============================================================================================
// Obrigado Pieter Germishuys - http://dotnet.org.za/pieterg/archive/2005/07/29/40407.aspx
//=============================================================================================

uniform extern float4x4 gViewProj;
uniform extern float3  gDirToSunW;
uniform extern float3  cameraPos;
uniform extern bool drawTextures;
texture texture0;
texture texture1;
texture texture2;
texture texture3;
texture texNormal;
texture texAlpha;
texture texDetail;

sampler2D texSampler0 : TEXUNIT0 = sampler_state
{
    Texture = (texture0);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};

sampler2D texSampler1 : TEXUNIT1 = sampler_state
{
    Texture = (texture1);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};

sampler2D texSampler2 : TEXUNIT2 = sampler_state
{
    Texture = (texture2);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};

sampler2D texSampler3 : TEXUNIT3 = sampler_state
{
    Texture = (texture3);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};


sampler2D texSampler3 : TEXUNIT3 = sampler_state
{
    Texture = (texture3);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};

sampler2D texSamplerNormal : TEXUNIT5 = sampler_state
{
    Texture = (texNormal);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};

sampler2D texSamplerAlpha : TEXUNIT4 = sampler_state
{
    Texture = (texAlpha);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};

sampler2D texSamplerDetail : TEXUNIT4 = sampler_state
{
    Texture = (texDetail);
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
};

struct OutputVS
{
    float4 posH         : POSITION0;
    float2 text  : TEXCOORD0;
};

OutputVS TerrainVS( float3 posW : POSITION0,
                            float2 tex0: TEXCOORD0 )
{
    OutputVS outVS = (OutputVS)0;

    outVS.posH = mul(float4(posW, 1.0f), gViewProj);
    outVS.text = tex0;

    return outVS;
}
```

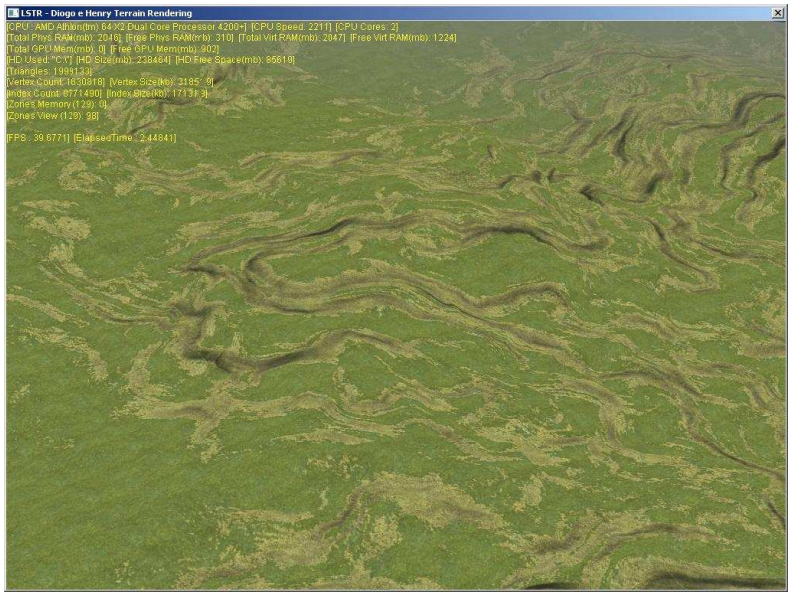
```
float4 TerrainPS( float2 text : TEXCOORD0 ) : COLOR
{
  if( drawTextures )
  {
    float2 textNormal;
    textNormal.x = text.x;
    textNormal.y = text.y;
    float4 alpha = tex2D(texSamplerAlpha, text);
    float4 color = tex2D(texSampler0, text * 2.0f) * 0.35f;
    color += tex2D(texSampler0, text * 2.0f) * alpha.r * 0.45f;
    color += tex2D(texSampler1, text * 2.0f)  * alpha.g * 0.45f;
    color += tex2D(texSampler2, text * 1.0f)  * alpha.b * 0.6f ;
    color += tex2D(texSampler3, text * 1.0f)  * alpha.w * 0.6f;
    color += tex2D(texSamplerDetail, text * 4.0f) * 0.1f;

    //float3 normal = tex2D(texSamplerNormal, textNormal).rgb;
    float3 normal = 2.0f * tex2D(texSamplerNormal, textNormal).rgb - 1.0f;
    float tmp = normal.z;
    normal.z = normal.y;
    normal.y = tmp;

    //set the output color
    float diffuse = saturate(dot(normal, gDirToSunW));

    //multiply the attenuation with the color
    return ( (color * diffuse) + 0.05f ) * 1.60f;
  }
  else
  {
    return 255;
  }
}

technique TerrainTech
{
    pass P0
    {
        vertexShader = compile vs_1_1 TerrainVS();
        pixelShader  = compile ps_2_0 TerrainPS();
    }
}
```

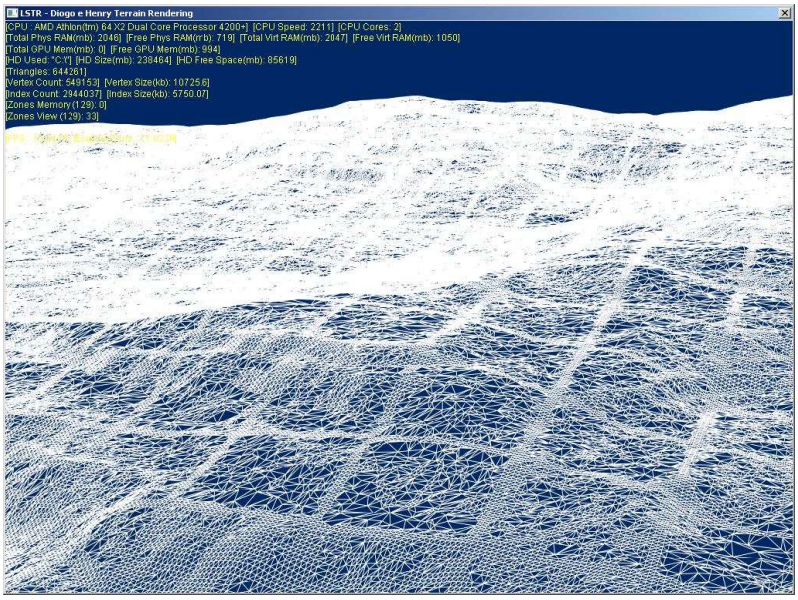
Figura 55 – Código *shader* para exibição do terreno.

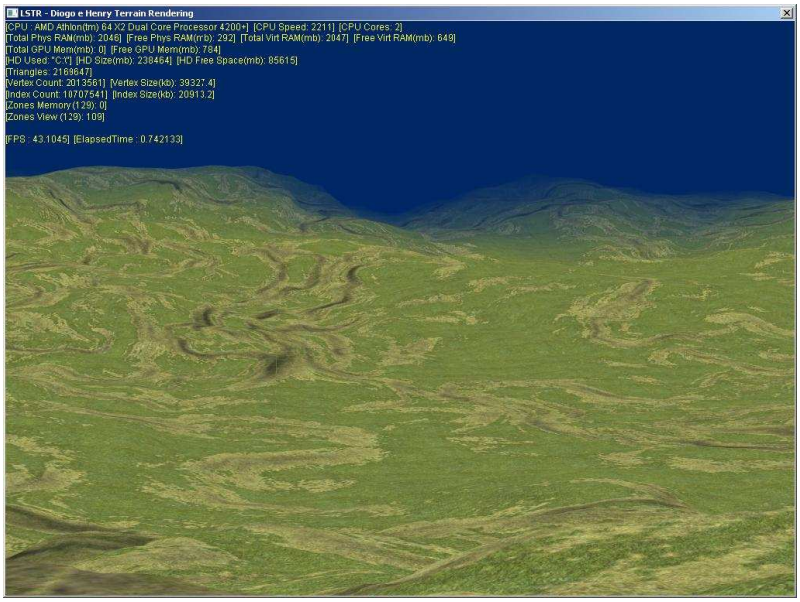
9. Apêndice II – Imagens do LSTR

Neste apêndice encontram-se imagens obtidas ao longo do desenvolvimento deste trabalho. As imagens se diferenciam de acordo com os efeitos utilizados.




LSTR - Diogo e Henry Terrain Rendering
[CPU : AMD Athlon(tm) 64 X2 Dual Core Processor 4200+] [CPU Speed: 2211] [CPU Cores: 2]
[Total Phys RAM(mb): 2046] [Free Phys RAM(mb): 310] [Total Virt RAM(mb): 2047] [Free Virt RAM(mb): 1224]
[Total GPU Mem(mb): 0] [Free GPU Mem(mb): 902]
[HD Used: "C:\"] [HD Size(mb): 238464] [HD Free Space(mb): 85619]
[Triangles: 1999133]
[Vertex Count: 1630818] [Vertex Size(kb): 3195.9]
[Index Count: 6771490] [Index Size(kb): 17131.3]
[Zones Memory (129): 0]
[Zones View (129): 98]
[FPS : 39.6771] [ElapsedTime : 2.44841]


LSTR - Diogo e Henry Terrain Rendering
[CPU : AMD Athlon(tm) 64 X2 Dual Core Processor 4200+] [CPU Speed: 2211] [CPU Cores: 2]
[Total Phys RAM(mb): 2046] [Free Phys RAM(mb): 719] [Total Virt RAM(mb): 2047] [Free Virt RAM(mb): 1050]
[Total GPU Mem(mb): 0] [Free GPU Mem(mb): 994]
[HD Used: "C:\"] [HD Size(mb): 238464] [HD Free Space(mb): 85619]
[Triangles: 644261]
[Vertex Count: 549153] [Vertex Size(kb): 10725.6]
[Index Count: 2944037] [Index Size(kb): 5750.07]
[Zones Memory (129): 0]
[Zones View (129): 33]


LSTR - Diogo e Henry Terrain Rendering
[CPU : AMD Athlon(tm) 64 X2 Dual Core Processor 4200+] [CPU Speed: 2211] [CPU Cores: 2]
[Total Phys RAM(mb): 2046] [Free Phys RAM(mb): 292] [Total Virt RAM(mb): 2047] [Free Virt RAM(mb): 649]
[Total GPU Mem(mb): 0] [Free GPU Mem(mb): 784]
[HD Used: "C:\"] [HD Size(mb): 238464] [HD Free Space(mb): 85615]
[Triangles: 2169647]
[Vertex Count: 2013561] [Vertex Size(kb): 39327.4]
[Index Count: 10707541] [Index Size(kb): 20913.2]
[Zones Memory (129): 0]
[Zones View (129): 109]
[FPS : 43.1045] [ElapsedTime : 0.742133]

10. Apêndice III – API 1.0

Tabela 7 – Nome e descrição dos métodos da API.

Classe LSTRGen	
Nome do Método	**Descrição**
LoadHeightMap	Carrega um arquivo contendo um mapa de altura no formato RAW 16 bits.
Simplify	Simplifica um mapa de altura, conforme o tipo de filtragem especificado, e retorna o mapa de booleanos gerado na simplificação.
Generate	Gera o arquivo necessário para a visualização.
Classe LSTRVis	
Nome do Método	**Descrição**
LoadLSTRFile	Carrega um arquivo contendo os dados do terreno (no formato do LSTR).
Update	Realiza o laço de atualização dos dados a partir de dois parâmetros de entrada: a posição da câmera e o tempo decorrido desde o último quadro.
Draw	Realiza a exibição do terreno conforme os dados atualizados pelo Update e o parâmetro de entrada do tempo decorrido desde o ultimo quadro.