FILIPE COSTEIRA VARELA

DESENHO DE SUPERFÍCIES PLANETÁRIAS EM 3D

Orientador: Prof. Dr. José Rogado

Universidade Lusófona de Humanidades e Tecnologias

Escola de Comunicação, Artes e Tecnologias de Informação

Departamento de Engª Informática e Sistemas de Informação

Lisboa 2012

2011

FILIPE COSTEIRA VARELA

DESENHO DE SUPERFÍCIES PLANETÁRIAS EM 3D

Dissertação apresentada para a obtenção do Grau de Mestre em Engenharia Informática e Sistemas de Informação no Curso de Mestrado em Engenharia Informática e Sistemas de Informação, conferido pela Universidade Lusófona de Humanidades e Tecnologias

Orientador: Prof. Dr. José Rogado

Universidade Lusófona de Humanidades e Tecnologias

Escola de Comunicação, Artes e Tecnologias de Informação
Departamento de Engª Informática e Sistemas de Informação

Lisboa
2011

Resumo

Esta dissertação pretende cobrir o desenvolvimento de um motor gráfico baseado em OpenGL para desenho de superfícies de dimensão planetária com topografia real. A quantidade de informação necessária para cobrir toda a superfície de um planeta torna necessária a utilização de mecanismos de representação multiresolução, compressão e organização da topografia para permitir a recuperação eficaz, possibilitando o seu desenho sem pausas perceptíveis para o utilizador.

Neste contexto, esta dissertação descreve a concepção e desenvolvimento de um motor gráfico para desenho de superfícies contínuas, de dimensão planetária, utilizando topografia real. Os objectivos deste motor são:

- Desenho de superfícies planetárias com topografia real
- Apresentar baixa complexidade tecnológica para adaptação simples a cada cenário de utilização
- Garantir performance satisfatória para vários cenários de utilização sem atrasos perceptíveis
- Ser disponibilizado em modo *Open Source* e de forma gratuita
- Utilizar apenas fontes de informação gratuitas e publicamente acessíveis

Apresentarei os métodos desenvolvidos para este efeito nas diferentes fases de pré-processamento, desenho e processamento vectorial no processador gráfico, assim como os resultados obtidos e comparação com métodos alternativos.

Posteriormente, para ilustrar a possibilidade de extensão de funcionalidades do motor desenvolvido, apresentarei métodos e implementações de codificação de regras de física e visualização de dados reais dentro do motor.

O título do motor gráfico desenvolvido para este projecto é Terrain Rendering Engine, doravante referenciado pelo acrónimo T.R.E.

Conteúdo

Lista de Tabelas

Lista de Figuras

1

Introdução

Nesta dissertação apresenta-se uma estratégia para a construção de um motor de desenho 3D baseado em OpenGL, cujo objectivo principal é a simulação de um sistema planetário à escala, com suporte à representação de planetas de dimensão real, com topografia real, assim como interacções físicas elementares, como gravitação ou fenómenos de dispersão atmosférica.

As aplicações para um motor deste género são várias e vão desde, numa ponta do espectro, jogos, até sistemas de informação geográfica, passando pela simulação e treino ou visualização científica em áreas como a meteorologia ou a oceanografia. Existem vários motores deste tipo disponíveis, como por exemplo o *Google Earth* ou o *NASA WorldWind*, cada um implementando técnicas distintas e com maior ou menor grau de abertura quanto à divulgação das mesmas. Este projecto pretende produzir um componente de software *opensource* e gratuito.

A quantidade de informação topográfica necessária para representar um planeta inteiro, independentemente da sua dimensão real, depende apenas da resolução com que o terreno fôr amostrado. A título de exemplo, o espaço de armazenamento necessário para guardar toda a topografia do planeta Terra incluindo superfícies de água e sugestão de côr em cada amostra, a uma resolução de $250m/amostra$ é de aproximadamente 73.5GB.

Os avanços computacionais em tecnologias de processamento gráfico nos anos recentes aumentaram drasticamente o espectro de possibilidades de tratamento e desenho de informação geográfica. Esta evolução põe constantemente à prova as formas de desenhar mais informação sem perda de desempenho comparativamente a implementações de geração anterior. Ainda assim, um dos objectivos deste motor é cumprir os requisitos previamente enumerados utilizando *hardware* modesto, existente em computadores de uso pessoal com capacidades de processamento gráfico limitadas.

1.1 Motivação

Mencionei anteriormente a existência de vários motores para desenho de planetas com topografia real. Estes motores são, no entanto, vocacionados para fins muito específicos. Independentemente de terem uma arquitectura modular, que permita a reutilização de componentes para contextos distintos, essa modularidade está, geralmente, escondida dos utilizadores e exposta apenas ao fabricante do motor. Não é possível a um utilizador reutilizar apenas os componentes responsáveis pelo desenho da superfície planetária num contexto para o qual não foi concebido.

Assim, a principal motivação da construção deste motor é distribuir um componente básico sobre o qual possam ser desenvolvidos um qualquer conjunto de métodos para utilização em contextos diferentes, como por exemplo:

- Representação fiel de um sistema solar com regras de gravitação
- Simulador para cálculo de trajectórias de vôo de sondas de exploração espacial
- Simulador de vôo de aeronaves com interacção atmosférica
- Visualização de dados em tempo real

O facto de o motor ter código aberto e ser gratuito e modular, permitirá a quem assim o desejar, substituir qualquer um dos componentes de pré-processamento, desenho ou cálculo vectorial no *GPU* por uma implementação melhor ou mais eficiente

dentro do contexto para o qual o motor esteja a ser adaptado. Aproveitando a filosofia vital inerente a algumas formas de licenciamento *opensource*, todos os utilizadores beneficiarão destas melhorias ou particularizações, podendo concentrar-se no desenvolvimento de futuras melhorias incrementais.

1.2 Organização

Esta dissertação está organizada com o seguinte formato:

Capítulo 2 - Introdução ao OpenGL Os mecanismos de desenho e processamento gráficos que serão abordados carecem de alguma contextualização para o leitor não familiarizado. Neste capítulo será feita uma breve introdução à API OpenGL, os seus componentes, tipos de dados que são manipulados e entidades responsáveis pelo seu processamento, assim como a motivação para a selecção desta API 3D face às alternativas.

Capítulo 3 - Desenho de terrenos Neste capítulo serão enumeradas as diferentes técnicas para desenho de terrenos em 3D.

Capítulo 4 - Modelo Físico Neste capítulo serão apresentados os elementos que são simulados para a reconstrução fiel de um sistema planetário à escala, a estrutura dos mesmos e as interacções a que são sujeitos.

Capítulo 5 - Arquitectura do motor Neste capítulo será apresentada, de forma esquemática e alto-nível, a arquitectura do motor. Serão enumerados os seus componentes e funcionalidades por componente, assim como a *pipeline* de processamento gráfico que é adoptada pelo motor.

Capítulo 6 - Caracterização dos dados Neste capítulo serão caracterizados os *datasets* topográficos que foram utilizados para o motor, que informação contêm e como foram gerados.

Capítulo 7 - Pré-Processamento Neste capítulo serão apresentadas as formas de pré-processar a informação topográfica para ser utilizada na implementação definida em **Desenho de terrenos 3D** e relação entre fidelidade aos dados originais e espaço de armazenamento utilizado por cada resolução possível.

Capítulo 8 - Manipulação CPU Neste capítulo será apresentada a forma como o motor trabalha os dados resultantes do pré-processamento no contexto do binário a ser executado no CPU. Serão abordadas as formas de leitura, *caching* e gestão de memória.

Capítulo 9 - Render GPU Neste capítulo serão apresentadas as técnicas para utilizar a informação topográfica armazenada em texturas para manipular, no GPU, os vértices que serão desenhados. Serão explicadas as formas de translacção de vértices, aplicação de informação para iluminação e texturas com sugestão de côr em cada ponto do terreno.

Capítulo 10 - Conclusão Este capítulo consiste numa análise dos perfis de execução da aplicação desenvolvida e interpretação destes resultados para aferir grau de cumprimento dos objectivos iniciais.

Capítulo 11 - Trabalho Futuro Neste capítulo serão enumerados alguns melhoramentos e desenvolvimentos planeados para o futuro do projecto.

2

Introdução ao OpenGL

O OpenGL - *Open Graphics Library* é uma API de processamento gráfico, independente de hardware ou sistemas operativos que não inclui funcionalidade para desenho de interfaces gráficos. O principal objectivo é que os diversos interfaces gráficos dos diversos sistemas operativos tenham pontes para delegar a responsabilidade do desenho de alguns dos seus componentes à API OpenGL.

Esta API constitui, conjuntamente com o Direct 3D, da Microsoft, o *standard de facto* para desenho 3D com suporte a aceleração por *hardware*. Há diferenças entre ambas que se traduzem numa maior ou menor facilidade de utilização, resultado gráfico final, suporte variado para extensões que permitem utilização de mecanismos avançados para manipulação de vértices no processador gráfico, entre várias outras. No entanto, a única diferença que serviu de base para a selecção da API a utilizar para a construção do motor gráfico aqui descrito, é a portabilidade. O Direct3D é proprietário da Microsoft, e está apenas disponível para sistemas Windows enquanto que o OpenGL é uma norma aberta e a sua portabilidade traduz-se na sua disponibilidade para praticamente todas as plataformas de computação e sistemas operativos existentes, incluindo Linux, Mac OS X, Windows, Solaris e BSD.

A título de curiosidade, o Direct3D é a API utilizada pelas consolas do mesmo fabricante, XBox 360. O OpenGL ES, versão para sistemas embebidos do OpenGL,

é a API utilizada nas consolas PlayStation. Para melhor ilustrar a consequência da portabilidade e do facto de se tratar de uma norma aberta, o suporte a desenho 3D em ambientes WEB, presente na norma WebGL suportada pelo HTML5, é uma versão de OpenGL ES.[1]

Apesar de se tratar de uma API de desenho, não inclui funcionalidade para o desenho de objectos complexos. Tudo o que é desenhado é baseado num conjunto de primitivas e instruções elementares como, por exemplo, translacções e rotações sobre pontos e linhas.

2.1 Arquitectura

O OpenGL consiste num conjunto de algumas centenas de funções que permitem, ao programador, a utilização das funções que o *hardware* gráfico implementa. Define-se como uma máquina de estados cujas transições são levadas a cabo através de chamadas a funções.

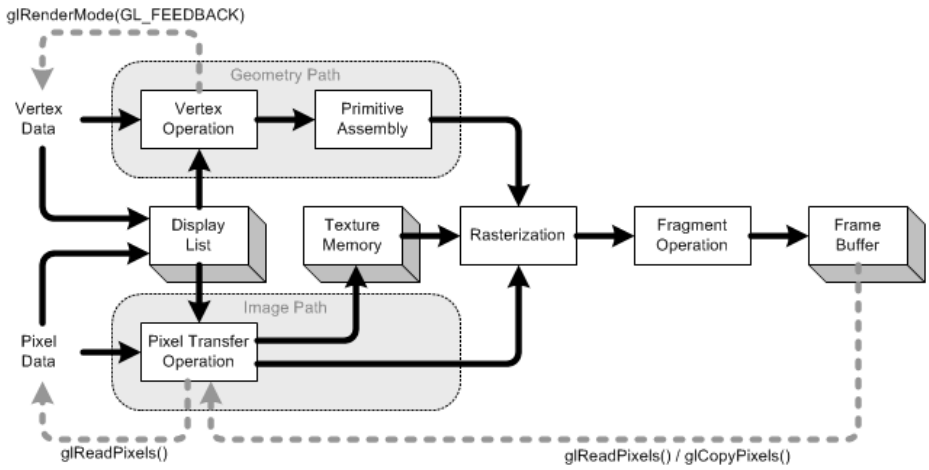

Figura 2.1: Arquitectura de processamento do OpenGL.

[1]Microsoft, Direct3D, Playstation são marcas registadas dos respectivos proprietários

2.2 Primitivas

A forma de desenhar objectos é adicioná-los ao contexto de execução do OpenGL. No entanto, a API não tem qualquer noção da natureza dos objectos a desenhar, visto que estes são constituídos apenas um conjunto de vértices que são inseridos na *pipeline* de processamento. Para que seja possível desenhar a superfície deste conjunto de vértices e não apenas um conjunto de pontos, é necessário indicar à API qual é a natureza das superfícies que os vértices representam. Para este efeito existem algumas pistas que identificam primitivas e que são utilizadas no processo de envio de vértices para processamento. Sendo o OpenGL uma máquina de estados, a forma de enviar esta informação é chamar uma função *glBegin* cujo argumento é um identificador da primitiva que deve ser desenhada com os vértices que se seguirão. Um exemplo de uma sequência para desenhar um triângulo é:

Listagem 2.1: Código para desenhar um triângulo.

```
glBegin(GL_TRIANGLES);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glVertex3f(−1.0f,−1.0f, 0.0f);
        glVertex3f( 1.0f,−1.0f, 0.0f);
glEnd();
```

Poder-se-ia continuar a enviar vértices em grupos de três antes de enviar a declaração de final de uso da primitiva de triângulos. A API saberia que cada três vértices seriam unidos por uma linha para formar um triângulo. A figura 2.2 ilustra quais as primitivas que estão disponíveis para desenho.

2.3 Mecanismo de Projecção

A API assenta num conceito fundamental de matrizes que representam todo o espaço a ser desenhado sobre pontos de vista diferentes. A construção de uma cena é um processo análogo ao de tirar uma fotografia e consiste no posicionamento da câmara,

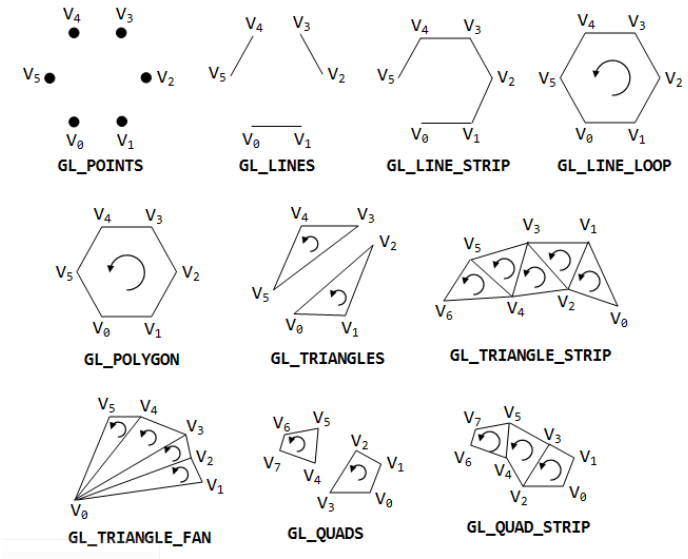

Figura 2.2: Primitivas do OpenGL.

na rotação da câmara e subsequente tranformação de coordenadas $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ para a construção da imagem bidimensional num écran. A sequência de transformações é ilustrada na figura seguinte.

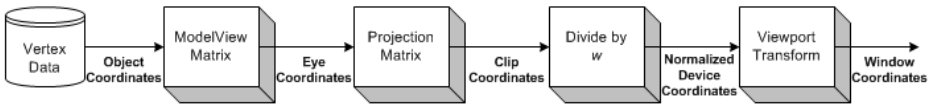

Figura 2.3: Processo de transformação de coordenadas até à fase de desenho no ecran.

Após o envio de coordenadas para a API através da chamada de uma função com valores, e depois de estabelecida a primitiva a utilizar, as coordenadas são transformadas do espaço objecto para o espaço do modelo. Neste espaço, todos os objectos têm coordenadas num referencial comum. As coordenadas são depois transformadas, projectando-se para um espaço cuja origem de referencial é a posição do observador. Este passo é o de projecção verdadeiramente dita. Segue-se uma fase de normalização das dimensões da imagem de acordo com as dimensões da zona em que vai ser mostrada.

2.4 Modos de Operação

Existem várias formas de enviar vértices para a API do OpenGL. Uma das formas, chamada de modo directo, é exemplificada na listagem de código 9.2. Esta forma consiste na invocação de uma função da API para o envio de cada vértice. No caso de primitivas compostas, como filas de triângulos, é feita uma optimização trivial, ou seja, os primeiros 3 vértices constituem um triângulo e todo o vértice adicional fecha mais um triângulo. Neste caso, o número de funções a invocar para desenhar n triângulos é igual a $n + 2$. Como há uma equivalência entre número de funções a invocar e de vértices em uso para a definição de cada triângulo, a memória necessária para armazenar n triângulos também desce de $3m$ para $m + 2$ com m correpondendo à memória ocupada por cada vértice.

No entanto, é facilmente compreensível que no processo de desenho de geometrias muito complexas, haverá um número de chamadas a funções n a partir do qual, o processador gráfico não consegue desempenhar de forma a manter a coerência visual. Isto é, a partir desses ponto, o tempo envolvido nas mudanças de contexto e chamadas a funções da API excede o tempo máximo entre duas imagens consecutivas que permitiria que o número de imagens produzidas por segundo transmitiria a sensação de fluidez ao olho humano. Para resolver este problema, foram desenvolvidos métodos de envio de geometria alternativos.

2.4.1 *Display Lists*

Uma das formas de minimizar as instruções enviadas à API é a utilização de *display lists*. Este grupo de comandos permite copiar para memória um conjunto de dados e instruções para futura referência. Desta forma, em vez de em cada imagem enviar toda a geometria de um objecto para o processador gráfico, envia-se uma única vez o conjunto de dados e instruções de desenho do objecto. Em imagens subsequentes, basta instruir o processador gráfico para desenhar a *display list* com um determinado

identificador. Este método permite que qualquer objecto passe a requerer apenas 1 chamada a funções para o seu desenho, independentemente da sua complexidade.

Listagem 2.2: Código para desenhar um triângulo utilizanto *display lists*.

```
// criar uma lista
GLuint index = glGenLists(1);

// compilar instrucoes e dados para desenho da lista
glNewList(index, GL_COMPILE);
    glBegin(GL_TRIANGLES);
            glVertex3f( 0.0f, 1.0f, 0.0f);
            glVertex3f(−1.0f,−1.0f, 0.0f);
            glVertex3f( 1.0f,−1.0f, 0.0f);
    glEnd();
glEndList();

// desenhar a lista
glCallList(index);

// apagar lista quando ja nao se pretende desenhar os seus objectos
glDeleteLists(index, 1);
```

A utilização de *display lists* permite várias outras optimizações. É possível incluír dentro da lista translações e rotações, operações com iluminação, entre outras.

2.4.2 *Vertex Arrays*

Os *vertex arrays* são uma forma de aglomerar informação de vértices num *buffer* em memória para, tal como com *display lists*, executar operações de desenho complexas com poucas chamadas a funções. O seu modo de funcionamento passa por alocar o espaço para a informação geométrica, indicar alguma informação sobre a forma como o conteúdo está organizado, e executar a operação de desenho. É possível indicar que tipo de primitiva deve ser utilizada para aglomerar os vértices ou o espaçamento entre

cada conjunto de coordenadas de vértices.

Permitem também uma optimização em relação ao modo de funcionamento de *display lists* que consiste na não duplicação de vértices partilhados entre superfícies. A título de exemplo, ao desenhar um cubo com *display lists*, seria necessário, para cada face, indicar os quatro vértices que a compoem. No entanto, cada um destes vértices é partilhado por 3 faces, pelo que seria especificado, em modo directo, em triplicado.

A metodologia de desenho de *vertex arrays* permite aglomerar todos os vértices únicos sequencialmente em memória, e utilizar um segundo *buffer* para armazenar índices dos vértices que compoem cada face. Desta forma, o espaço necessário para informação redundante desce do número de ocorrências redundantes de cada vértice multiplicado pelo espaço necessário a cada, para apenas o espaço necessário para armazenar uma posição ou índice do vértice redundante.

Ao contrário das *display lists*, não é possível executar chamadas a funções de manipulação de vértices ou iluminação com a utilização de *vertex arrays*.

3

Desenho de terrenos

O desenho de terrenos em 3D é uma área particularmente complexa da disciplina de computação gráfica. Isto deve-se ao facto de que uma representação pormenorizada de um terreno real ou sintetizado requer informação cuja complexidade espacial é suficientemente grande para serem necessários mecanismos de restrição à informação a ser desenhada de forma a poder ser trabalhada pelos recursos limitados disponíveis em máquinas para computação doméstica.

Por esta ser uma disciplina suficientemente complexa para merecer descrição mais pormenorizada e por este não ser o foco, mas uma generalização do problema tratado nesta dissertação, procurarei apresentar os mecanismos mais comuns de organização e desenho de informação topográfica, assim como as técnicas de restrição aos dados desenhados para que possam ser trabalhados por recursos modestos.

3.1 Conceitos Elementares

A topografia de um pedaço de terreno é, regra geral, representada por um mapa de elevações *heightmap*. Esta estrutura não é mais do que uma imagem comum cujos canais de cores representam não amplitude de cada côr, mas altitude de um ponto

do terreno. Estas imagens são tratadas de formas distintas consoante o método de desenho. Podem ser lidas no contexto do programa de desenho e interpretadas para atribuir coordenadas em 3D aos vértices que vão ser desenhados, ou podem ser interpretadas como imagens e passadas ao GPU como texturas para atribuição, numa fase posterior, das altitudes correspondentes.

Visto que uma estrutura de dados que contenha todos os vértices necessários para representar a topografia completa de um corpo planetário ou de dimensão equivalente possa conter uma quantidade de informação várias ordens de grandeza superior aos recursos habitualmente disponíveis para a processar, torna-se necessário aplicar algumas técnicas de particionamento de espaço e selecção de informação a desenhar que são transversais a todos os métodos de desenho. Algumas destas técnicas combinam com um método de desenho em particular, outras aplicam-se a qualquer método de desenho de qualquer objecto em 3D.

3.1.1 Depth Sorting

A técnica de *depth sorting* ou ordenação por profundidade consiste em ordenar uma lista de objectos a desenhar por distância crescente para os objectos completamente opacos, e pela ordem inversa para os translúcidos.

Desta forma, os objectos mais perto do ponto de vista do observador, ao serem desenhados primeiro, pintam uma mancha num *buffer* de profundidade da API 3D. Com este *buffer* pintado em algumas áreas, qualquer outro objecto mais distante que vá ocupar a mesma área é testado pela API 3D e alguns ou todos os seus vértices são imediatamente descartados, nunca chegando às fases de processamento mais intensivas do GPU e minimizando assim o tempo total de processamento dedicado ao desenho.

O motor desenvolvido para esta dissertação utiliza este método para filtragem de informação a desenhar pelo GPU em cada ciclo, para além de outras formas de ordenação temporal para determinação de elementos que precisam de saír de memória *cache*.

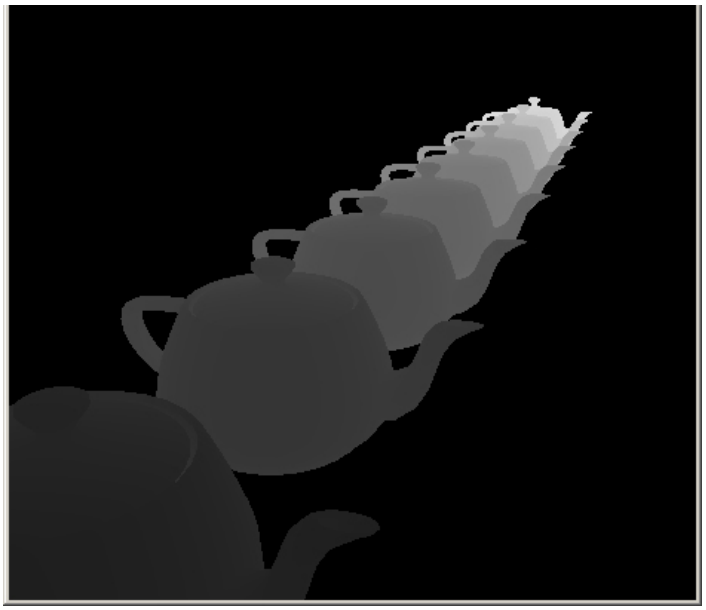

Figura 3.1: *buffer* de profundidade de uma cena. As zonas ocultas de cada objecto não chegam às fases avançadas de processamento.

3.1.2 Frustum Culling

Esta técnica consiste em determinar, para todos os objectos em memória, quais estão completamente fora do cone de espaço que é visível ao observador. Uma vez identificados, o desenho destes objectos é abortado, minimizando a informação que é desenhada em cada ciclo.

Para identificar quais os objectos que estão completamente ocultos, é necessário testar os seus vértices quanto à visibilidade. No entanto, alguns objectos são suficientemente complexos para que o tempo de execução destes testes seja superior ao tempo que demoraria o desenho dos mesmos. Por este motivo utilizam-se técnicas de optimização destes testes que variam consoante o tipo de geometria codificada pelos vértices em questão.

De todas estas técnicas, salientam-se duas particularmente comuns. O conceito fundamental é o mesmo e consiste em encapsular o objecto numa estrutura geométrica simplificada e testar apenas os vértices dessa estrutura. A principal variação tem que

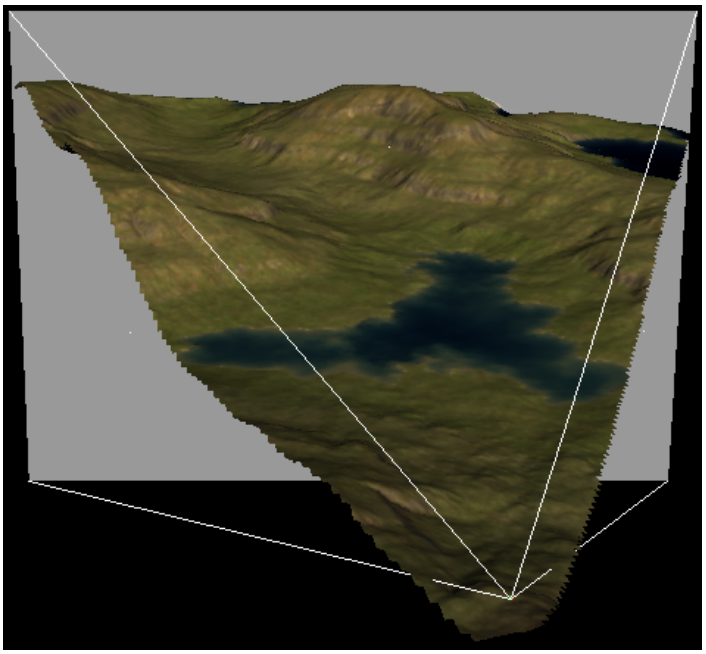

Figura 3.2: Detecção de visibilidade por *frustum culling*.

ver com a geometria da estrutura em si e não com o processo de teste. As duas estruturas mais comuns são paralelipípedos e esferas. Com estas técnicas, para um objecto com um qualquer número de vértices, apenas é preciso testar a visibilidade dos vértices da estrutura que o engloba e, em alguns casos, as arestas que unem estes vértices.

O motor desenvolvido para esta dissertação utiliza ambas, e alterna entre as duas dependendo do quão apta é cada uma para a detecção de visibilidade do objecto encapsulado.

Após estas fases, é necessário calcular informação que vai ser utilizada para cálculos de iluminação do terreno e, posteriormente, aplicar texturas à geometria.

3.1.3 Iluminação

A iluminação do terreno é feita através de cálculos que determinam, para cada vértice, coeficientes de luminosidade ambiente, *specular* e difusa. Estes cálculos têm em conta

o vector normal a cada vértice. Isto implica que seja calculado um novo conjunto de vectores de dimensão igual à dos vértices. Este novo conjunto é designado por um mapa de normais. Cada um dos três parâmetros de iluminação tem em conta o quão orientado para a fonte de iluminação está a superfície de cada triângulo de terreno.

Os três parâmetros calculados são ponderados para a produção de uma métrica escalar final que é multiplicada pela côr na textura de cada vértice, permitindo ajustar a sua intensidade e, assim, simular iluminação.

3.2 Particionamento Espacial e simplificação geométrica

Como referido anteriormente, não é possível desenhar em todos os ciclos, toda a informação que codifica a geometria do terreno. Por este motivo, é necessário particionar o terreno em blocos de vértices contíguos e aplicar os testes referidos acima a cada um destes blocos. Torna-se evidente, após referência à técnica de particionamento espacial, que estes testes podem ser aplicados de forma recursiva a cada bloco. Desta forma, sempre que há apenas visibilidade parcial de um bloco, é possível particionar o bloco em blocos de dimensão espacial inferior e testá-los individualmente.

Para além das técnicas de particionamento espacial que permitem análise de complexidade geométrica em blocos, existem também técnicas de simplificação geométrica, ou mais precisamente, de remoção de informação supérflua para o desenho de superfícies. Isto deve-se ao facto de as amostras topográficas serem feitas em malhas regulares, independentemente da complexidade do terreno em questão.

Existem muitas técnicas de particionamento espacial para a geometria de terrenos em 3D. Estas técnicas podem ser visualizadas como uma árvore evolutiva de algoritmos que foram sendo refinados ao longo do tempo, sendo que as mais recentes e com maior investigação e aplicação no presente são as que apresentarei abaixo.

3.2.1 Triangulated Irregular Networks - TIN

Este método de particionamento do terreno consiste em triangular o terreno de forma irregular. O objectivo é representar com poucos triângulos as áreas com pouca variação topográfica como, por exemplo, planícies, e com maior densidade de triângulos as áreas com muita variação de elevações como, por exemplo, cordilheiras.

Se se imaginar a representação de elevações de um terreno como uma imagem em que as elevações são codificadas como intensidade de um qualquer canal de côr, então o número de triângulos necessário para a representação de cada zona da imagem é directamente proporcional ao mapa que codifique as amplitudes da derivada de primeira ordem - o declive - do mapa original.

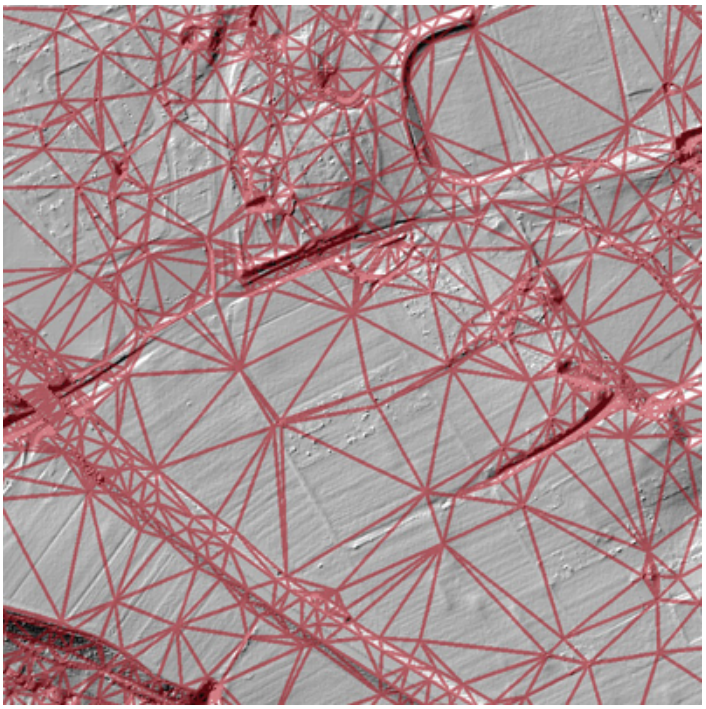

Figura 3.3: Terreno triangulado com TIN.

Embora esta técnica permita manter a densidade de amostras original do terreno nas zonas que precisem de maior pormenor e, simultaneamente, simplificar a geometria das zonas menos acidentadas, não é um método de particionamento espacial. Esta técnica pode, no entanto, ser aplicada aos blocos resultantes do particionamento

conseguido com outras técnicas.

3.2.2 Realtime Optimally Adapting Meshes - ROAM

Este método tornou-se muito popular em anos recentes. Embora seja substancialmente mais complexo que as alternativas contemporâneas, oferece melhor gestão de recursos e performance.

O conceito fundamental consiste na utilização de uma árvore binária cujos elementos contêm um triângulo e apontadores para elementos filhos em resolução crescente. A forma de popular a geometria dos níveis inferiores da árvore é segmentar o elemento imediatamente acima ao meio e copiar metade da sua geometria para o nível inferior.

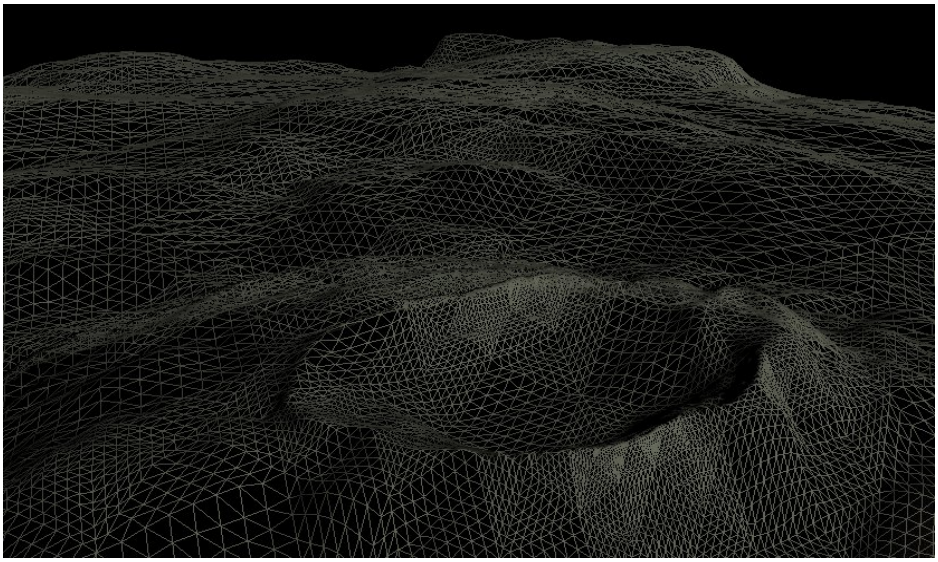

Figura 3.4: Bloco de terreno triangulado com ROAM.

Paralelamente a esta estrutura, são utilizadas duas filas. Uma de divisão de elementos e uma de fusão de elementos. Estas filas são utilizadas para seleccionar o nível de detalhe com que a geometria deve ser desenhada de acordo com uma métrica de erro. Esta métrica é uma combinação de elementos que determinam quão acidentada é a superfície a ser desenhada e, portanto, quão precisa deve ser a sua representação geométrica, e quão afastada está do observador. Existem vários parâmetros de ajuste

a estes dois componentes que variam de caso a caso consoante os dados topográficos, o mecanismo de projecção utilizado no motor gráfico, entre outros.

3.2.3 Chunked Level of Detail - CLOD

Este método consiste também, na utilização de uma árvore binária para armazenamento de geometria. No entanto, o conteúdo de cada elemento da árvore é um bloco de $n.n$ vértices em forma quadrangular que representa a geometria de um bloco de terreno. Cada um destes elementos contém apontadores para 4 outros elementos seus filhos. Cada um destes elementos filhos contém a mesma quantidade de informação que o seu pai, $n.n$ vértices que representam 1/4 da sua informação. Assim, a resolução duplica a cada nível da árvore mas mantém-se um número de vértices por bloco constante. Esta característica permite optimizações em estágios posteriores da linha de processamento, como por exemplo, a partilha de um único bloco de vértices não geo-referenciado e posterior transformação de altitudes através da leitura da informação topográfica como se de uma textura se tratasse.

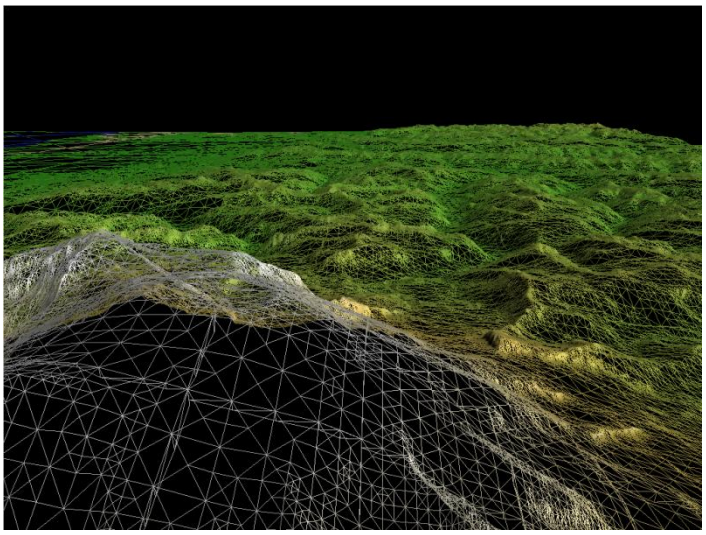

Figura 3.5: Bloco de terreno particionado com CLOD.

O processo de desenho destes elementos consiste na determinação, também, de uma métrica de erro relacionada com a distância de cada elemento ao observador e do quão

importante é o bloco para a área de foco da cena. Sempre que se determine que um elemento precisa de ser desenhado com maior ou menor resolução, retira-se o mesmo da lista de blocos que serão desenhados e repete-se a análise para, respectivamente, os seus filhos ou pai.

4

Modelo Físico

O motor desenvolvido no âmbito desta dissertação, como descrito na secção anterior, simula alguns corpos do nosso sistema solar. Os corpos simulados, neste momento, são o Sol e a Terra. Estes corpos foram seleccionados pela existência de informação topográfica completa para a Terra. Na altura da escrita deste documento, existem já dados topográficos para vários outros planetas. O formato em que esses dados são armazenados é praticamente igual em cada caso, pelo que seria possível, com um esforço de algumas horas, prepará-los para integração no motor.

Não existem simulações completas e integralmente fiéis à realidade. É sempre necessário traçar uma fronteira entre a informação que é representada e a que fica fora do modelo, de acordo com o fim para o qual a simulação se destina. Este motor simula o movimento de translacção da Terra em roda do Sol e a rotação sobre o seu próprio eixo Norte/Sul. As posições relativas do Sol e da Terra são calculadas de acordo com a data configurada no sistema e correspondem às localizações reais de ambos os corpos na mesma data.

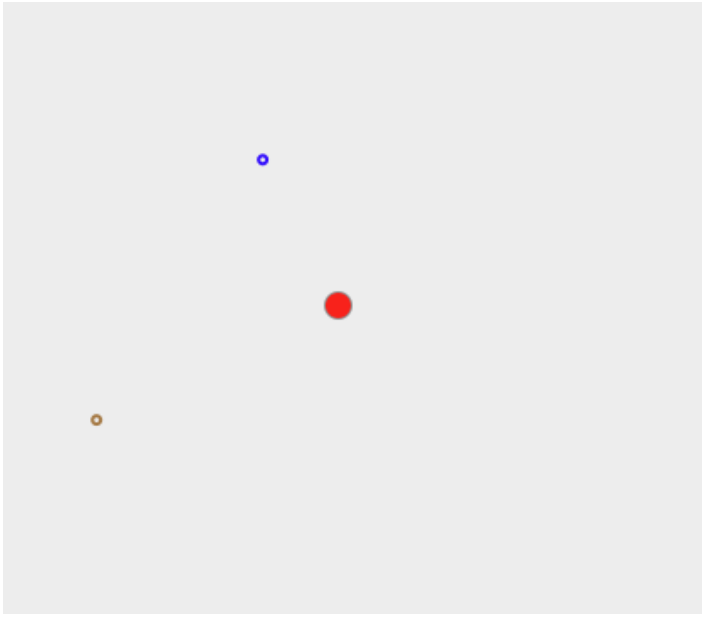

Figura 4.1: Protótipo para cálculo de posições relativas do Sol, Terra e Marte.

4.1 Posicionamento

Para o cálculo da posição da Terra em relação ao Sol, é utilizado um modelo publicado em [3] e é baseado numa combinação de *fitting* de observações. Este modelo utiliza, como parâmetros, uma data e um par de coordenadas latitude e longitude na superfície da Terra. O resultado do algoritmo é outro par de coordenadas que representa o ponto, na superfície terrestre, em relação ao ponto do observador, pelo qual passa uma recta que una o centro da Terra ao centro do Sol. A distância entre estes dois corpos é tida como constante e igual a $1 a.u.$[1].

O Sol é representado por uma esfera cuja posição corresponde ao versor do vector Terra-Sol multiplicada por $1 a.u.$.

[1] *a.u.*: Unidade Astronómica. Aproximadamente igual à distância média entre Terra e Sol, 149 597 870,7 km

4.2 Iluminação

Uma vez estabelecido o vector Sol-Terra, considera-se o seu versor como a direcção da fonte de iluminação da superfície do planeta. Para que os ângulos de incidência ao plano tangente de cada ponto na superfície terrestre sejam o mais próximos à realidade quanto possível, este versor é multiplicado por $1 a.u.$. Assim, a fonte de iluminação de modelo omnidireccional está posicionada no mesmo local que a sua representação geométrica.

Apesar de ser desejável representar estes números com a maior precisão possível, há um limite prático inultrapassável e definido pelo mínimo múltiplo comum de funcionalidades do próprio processador gráfico. O motivo pelo qual isto acontece é que, apesar de haver suporte para vírgula flutuante e dupla precisão na camada API de *OpenGL*, o mesmo não acontece na camada de *hardware*. Caso se utilize o tipo de dupla precisão e vírgula flutuante, todos os valores são convertidos para *fixed-point* de $32 bit$ pelo *hardware*. Esse é, então, o limite para o qual se deve apontar para representar o valor máximo, sem *overhead* de conversões de tipos, nem perda de precisão. Esse valor é, por consequência da norma IEEE 754-2008, igual a $3,4 \times 10^{38}$.

4.3 Gravitação

O modelo de gravitação utilizado tira vantagem da forma de organização, em memória, dos vários objectos que constituem a cena. Foi criada uma árvore que contém todos estes elementos. Cada elemento, para além da informação geométrica e algumas propriedades auxiliares, contém informação de massa e energia. A cada iteração do ciclo de desenho são calculadas as forças resultantes da interacção entre os vários objectos e aplicadas as transformações resultantes da acção dessas forças.

No entanto, e devido ao facto de se tratar de uma simulação em tempo discreto, acontecem alguns arredondamentos e imprecisões nos cáculos. Estas imprecisões le-

vam, ao fim de algum tempo, à acumulação de erros de posicionamento que desequilibram o sistema. Estes problemas manifestam-se pela perturbação das órbitas dos planetas simulados e perda de estabilidade de satélites em órbita dos mesmos.

Para contornar este problema, poder-se-ia presumir um sistema isolado e de estabilidade inerente. Partindo deste princípio, seria possível alterar as formas de cálculo para terem em conta a lei da conservação de energia e assim, corrigir os erros em cada iteração do ciclo de cálculo.

A forma de cálculo passa por, para cada elemento, calcular a força resultante da interacção com todos os outros. Para evitar que o resultado do processamento de cada elemento alimente os cálculos dos restantes na mesma iteração, é utilizado um duplo *buffer* tradicional.

A equação utilizada para determinar a força resultante entre cada elemento e cada um dos restantes é

$$\vec{F} = G\frac{m_1 \times m_2}{r^2} * \vec{u_{2-1}} \tag{4.1}$$

Em que r^2 é o quadrado da distância entre elementos e $\vec{u_{2-1}}$ é o versor do vector que une $\vec{p_1}$ a $\vec{p_2}$. Uma vez calculadas todas as resultantes, a posição de cada elemento é ajustada pela seguinte equação

$$\vec{p_t} = \vec{p_{t-1}} + \vec{v} \times dt + \vec{F} \times dt^2 \tag{4.2}$$

Em que dt é o tempo decorrido entre cada iteração do ciclo de cálculo.

5

Arquitectura do motor

O método de desenho desenvolvido para o motor gráfico baseia-se numa conjugação de técnicas aos vários níveis da pipeline de desenho. A figura representa esquematicamente os diversos componentes do motor.

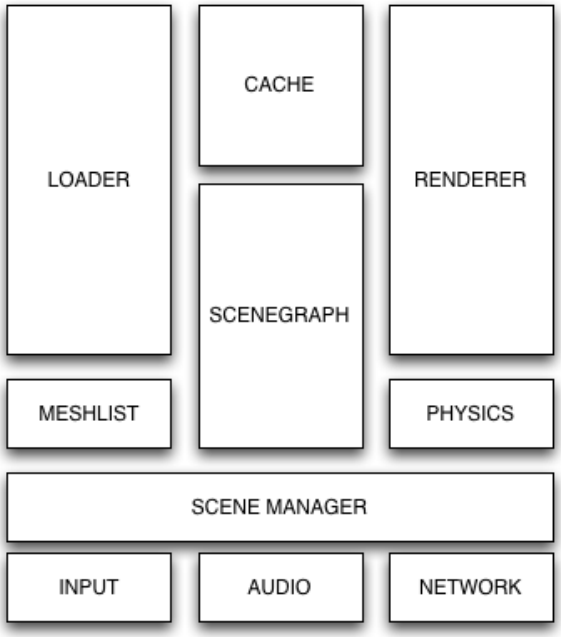

Figura 5.1: Componentes do motor T.R.E.

5.1 Loader

O componente Loader é apenas uma classe abstracta cujas implementações interpretam formatos diferentes de topografia, convertendo-os para a representação em memória utilizada pelo motor. A implementação de loader utilizado para esta dissertação lê os formatos *Blue Marble Next Generation*[1]. Este componente é utilizado por outros componentes sempre que se verificar a necessidade de ler informação topográfica out-of-core.

5.2 Cache

O componente de cache é responsável pela determinação de importância de manter um bloco de terreno em cache e pela organização e classificação de blocos na lista em cache. A estratégia de cache assenta num threshold temporal a partir do qual um bloco é descartado caso seja necessário ler um novo bloco e o tamanho da lista após esta leitura exceda um threshold de memória configurado. Os melhores resultados foram conseguidos com threshold temporal de aprox 60 segundos e alocação de 200 bloco de terreno. Este componente apenas marca os bloco com informação de caching. Os dados em si estão organizados numa lista gerida pelo componente MeshList.

5.3 SceneGraph

Para desenhar os vários blocos em memória é necessário determinar a posição e orientação da camara e, com base nessa informação, calcular que bloco estão visíveis na cena que vai ser desenhada. Desta forma garante-se que não é desenhada geometria não visível. Este componente é também responsável pela determinação, através de uma métrica combinada de distância e erro no ecran, de qual o nível de resolução a que um bloco deve ser desenhado. Sempre que fôr determinado que é necessário aumentar

[1] http://visibleearth.nasa.gov/, NASA

a resolução de um bloco, o bloco resultante é pedido à cache. Caso o retorno seja negativo, o nó é pedido ao Loader e inserido na cache.

5.4 Renderer

Devido a optimizações comuns nas APIs de 3D, os componentes opacos devem ser desenhados de frente para trás. Desta forma, o buffer de profundidade pode ser utilizado para determinar se um objecto está à frente do outro, ocultando-o completamente, e impedir o desenho do objecto ocultado. Para este efeito, o Renderer reordena a lista de elementos a desenhar por distância crescente à camara e desenha-os nessa ordem. Todos os elementos desenhados são implementações de uma subclasse de interface de desenho única.

5.5 Physics

Uma vez que o motor simula planetas à escala real, foi desenvolvido um módulo de física que é responsável pelo tracking de posição de todos os objectos. Este módulo é crítico para a determinação da posição da estrela do sistema e, consequentemente, cálculo dos vectores de iluminação a utilizar no render final.

O tracking de posições de planetas e estrelas é feito com base nos algoritmos descritos em [3]. As posições de todos os restantes objectos são calculadas através do modelo gravitacional de Newton.

6

Caracterização dos Dados

A topografia planetária utilizada é proveniente de vários *datasets*. Em anos recentes têm sido desenvolvidos esforços por diversas organizações para compilar informação topográfica do planeta Terra e corrigir erros existentes na informação já existente. Destes esforços salientam-se dois *datasets* que são a fonte *de facto* para software de informação geográfica que exija grande pormenor e fidelidade. Esses são o *Blue Marble Next Generation* - BMNG - e o *Shuttle Radar Topography Mission*[1] - SRTM.

6.1 Resolução da Informação Geográfica

Enquanto que o SRTM contém apenas informação de elevação a uma resolução de 90 metros por amostra, o BMNG utiliza uma resolução inferior (de 500 metros por amostra) mas contém informação topográfica, batimétrica, sinalização de presença de água e sugestão de côr média em cada amostra. Devido a este facto, é comum utilizar simultaneamente ambos os *datasets* representando a topografia com elevado grau de fidelidade.

O motor T.R.E. apenas utiliza o BMNG para toda a informação. Uma vez estabe-

[1] http://www.cgiar-csi.org/, Versão 4.1 do dataset produzido pela NASA

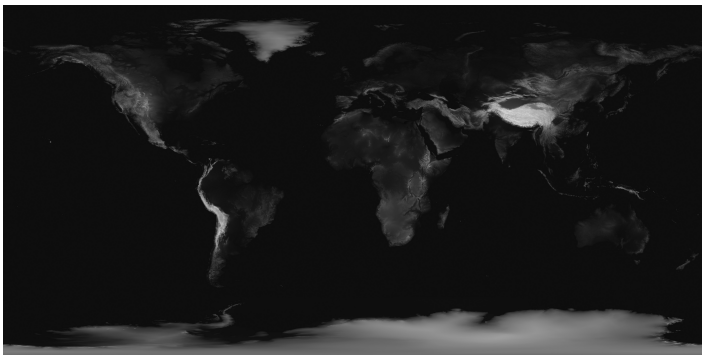

Figura 6.1: Mapa de elevação do planeta BMNG.

lecido o mecanismo mais favorável ao desenho de topografia sem atrasos perceptíveis, é trivial escrever um módulo Loader para suportar o formato SRTM.

O motor T.R.E. utiliza, para o desenho de superfícies com informação topográfica, vários subconjuntos de informação do BMBG. Todos estes subconjuntos são constituídos por amostras numa projecção cilíndrica equidistante (Plate-Carre) no sistema WGS-84. Os vários componentes utilizados são:

***Subdatasets* utilizados no pré-processamento.** 21.5**GB**

- **Raw Topography** Conjunto de 86400x43200 amostras de 16*bit* com elevação em metros em relação do nível médio do mar (aprox 7GB)
- **Raw Bathymetry** Conjunto de 21601x10801 amostras de 16*bit* com profundidade em metros em relação do nível médio do mar (aprox 445MB)
- **Raw Color Data** Conjunto de 86400x43200 amostras de 24*bit* com informação de côr média. Contém 3 canais de 8*bit* cada (aprox 10.5GB)
- **Raw Water Mask** Conjunto de 86400x43200 amostras de 8*bit* com informação de presença de água[2] (aprox 3.5GB)

Estes dados têm de ser transformados para um formato intermédio. É importante evidenciar a necessidade de transformar esta informação para que possa ser desenhada por um motor gráfico.

[2]Não se pode presumir que um vértice com altitude abaixo do nível médio do mar esteja submerso, nem que um vértice acima desta não o esteja. Existem, por exemplo, rios acima do nível do mar.

6.2 Tratamento dos Dados

Num contexto de 3D, as superfícies são constituídas por vértices. Para que uma superfície seja desenhada com iluminação realista, é necessário codificar informação de iluminação no ambiente 3D e é necessário especificar, para cada vértice, qual a sua orientação face à luz. Este método de funcionamento é transversal a todas as APIs de 3D e traduz um balanço entre memória e desempenho, já que a extracção de informação de declive de uma superfície é suficientemente complexa para ser preferível fazê-lo numa fase de pré-processamento e apenas ler essa informação na fase de desenho. Esta informação designa-se por informação de *normais* visto ser composta por um vector normal a cada vértice. Por esta razão, a partir da informação topográfica, gera-se um novo conjunto de amostras, mas contendo um vector em $\mathbb{R}^3$ por cada amostra de elevação.

O formato escolhido para conter a informação, ignorando a forma como a informação deve ser segmentada, é o seguinte:

- **Topography** Conjunto de 86400x43200 amostras de $24bit$ contendo $16bit$ de elevação em relação ao raio médio do planeta (fusão entre *topo* e *bathy*) e $8bit$ de máscara de presença de água para efeitos de reflexos (*specular highlight*) (aprox 10.5GB)
- **Normals** Conjunto de 86400x43200 amostras de $24bit$. Cada amostra é um vector normal à superfície em $\mathbb{R}^3$ (aprox 10.5GB)
- **Raw Color Data** Conjunto de 86400x43200 amostras de $24bit$ com informação de côr média. Contém 3 canais de $8bit$ cada (aprox 10.5GB)

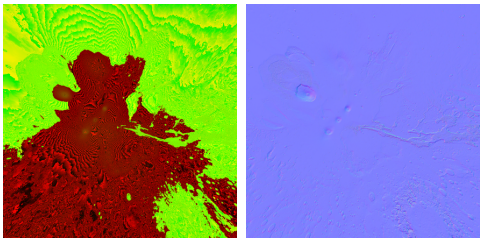

Figura 6.2: Altitudes codificadas em dois canais. Normais codificadas em 3 canais.

7

Pré-Processamento

Uma vez estabelecida a informação que é necessário ter para se poder desenhar a geometria com informação de iluminação, reflexos e côr, é possível definir a forma como o pré-processamento vai gerar a mesma. Da análise de alto nível dos dados, resultam duas observações importantes para definir o processo que se segue:

- A projecção cilíndrica equidistante contém uma densidade de informação não linear entre amostras perto do equador e nos pólos.
- A informação contida nos *datasets* não pode ser carregada em memória em sistemas de utilização genérica ou doméstica. E mesmo nos sistemas com maior quantidade de recursos de memória, continuaria a ser impossível solicitar a um GPU o desenho da informação e ainda manter uma quantidade de *frames* por segundo aceitável.

Daqui resultam novos problemas a resolver antes de codificar o processamento da informação:

- É necessário escolher um método de particionamento da informação para ser possível carregar selectivamente apenas a informação que vai ser desenhada e implementar mecanismos de cache para minimizar tempos de leitura de cada partição.
- É desejável escolher uma projecção que minimize a distorção resultante da não-linearidade na densidade de informação em cada partição.

- É desejável escolher uma projecção que permita à partida uma forma de pré-particionamento do espaço.

Assim sendo, a fase de pré-processamento consiste nos seguintes passos:

- **Projecção** dos vários *datasets* de cilíndrica equidistante para gnomónica resultando em seis partições/projecções.
- **Particionamento** de cada projecção para leitura selectiva de dados e caching de cada conjunto de dados.

7.1 Projecção

Dos vários processos de particionamento, foi utilizado um mecanismo de conversão de projecção cilíndrica para gnomónica [1], aproveitando a baixa distorção resultante desta projecção. Os seis pontos de projecção correspondem aos pólos e a pontos inscritos num circulo assente no mesmo plano do equador de forma a terem um afastamento de $\pi/2$ e a função de transferência de um par latitude/longitude para coordenadas na zona reprojectada é

$$x = \frac{cos\theta \times sin(\lambda - \lambda_0)}{cos\psi} \tag{7.1}$$

$$y = \frac{cos\theta_1 \times sin\theta - sin\theta_1 \times cos\theta \times cos(\lambda - \lambda_0)}{cos\psi} \tag{7.2}$$

A técnica de *cubemapping* é muito apropriada para objectos esféricos. Isto tem que ver com o facto de o mapa de elevações[1] serem, tipicamente, estruturas rectangulares, ou uma projecção planisférica do objecto original. Assim, a técnica consiste em mapear essa estrutura rectangular em várias faces de um cubo e, após isto, normalizar as alturas base de cada vértice resultante, gerando assim uma esfera. Desta forma, cada vértice passará a ter comprimento igual ao raio desejado da esfera - o planeta em si - mais a altura do terreno no ponto por ele representado.

[1] *Heightmaps* na literatura de referência

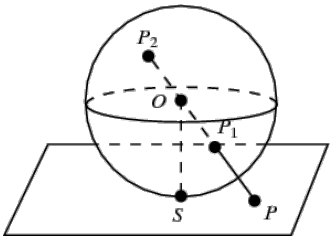

Figura 7.1: Exemplo de projecção gnomónica.

O processo passa por segmentar o mapa de elevações original em seis zonas. Como o processo de projecção planisférica que gera o mapa de elevações contém informação redundante em quantidade crescente do equador aos pólos, é necessário reduzir a quantidade de informação, sem perda, de forma a extraír apenas a informação que originou o planisfério.

Para este efeito foi escrito um programa[2] que converte um ficheiro com a topografia em projecção cilíndrica em seis ficheiros, cada um com a topografia da zona projectada à volta de cada um dos seis pontos de referência.

Cada uma das superfícies resultantes será desenhada normalizando cada um dos seus vectores em $\mathbb{R}^3$ e multiplicando o versor resultante pelo raio do planeta, conseguindo assim uma conversão de um cubo numa esfera.

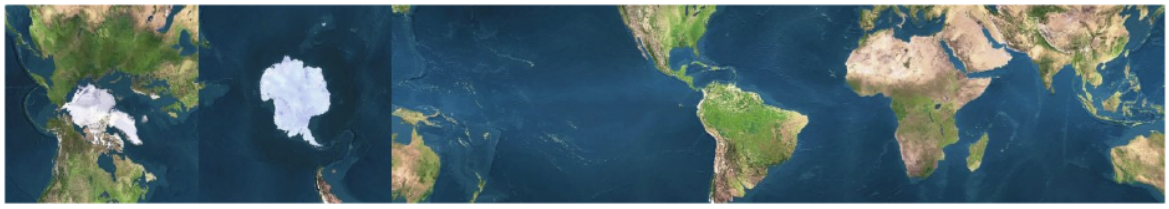

Figura 7.2: Seis projecções gnomónicas finais.

[2]project.c

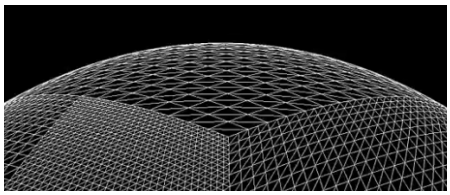

Figura 7.3: Várias zonas do cubo projectado em esférica com normalização dos vectores.

7.2 Particionamento

Foi estabelecido previamente que para conseguir um desempenho aceitável é necessário desenhar as várias zonas com graus de pormenor variáveis de acordo com alguma métrica que determine qual o grau de pormenor a utilizar, assim como não desenhar secções não visíveis do terreno. Também foi pré-estabelecido que para o conseguir é necessário usar técnicas de particionamento do espaço de cada face do cubo. E é necessário fazê-lo em unidades de dimensão estudada para que:

- Tenham dimensão suficientemente grande para que os vértices possam ser enviados numa operação para o GPU e assim conseguir acelerar o processo de render
- Tenham dimensão suficientemente grande para que seja útil tê-las numa cache de geometria e, portanto, reduzir consultas e leituras à base de dados principal
- Tenham dimensão suficientemente pequena para que seja possível mostrar simultaneamente zonas de alto e baixo pormenor de acordo com uma métrica de distância ao centro da zona

Para responder a estas necessidades, foi utilizada uma *quadtree*[5]. Uma *quadtree* é uma estrutura de dados utilizada para particionar espaços bidimensionais. Cada elemento da árvore contem 4 elementos, repetindo-se esta organização até ao último nível. O número de níveis corresponde ao número de níveis de detalhe. Cada nível contem um quarto da informação do nível seguinte. Desta forma, o primeiro nível é o menos detalhado e o último o mais detalhado.

A técnica de desenho consiste em determinar a distância do observador ao centro de uma zona e determinar quantos pixeis são necessários para desenhar a zona utilizando

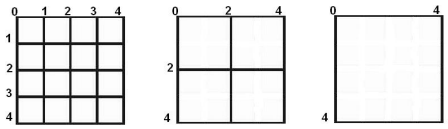

Figura 7.4: Mesma topografia em multiresolução, níveis *LOD* de LOD0 (esquerda, mais detalhado) até LOD2 (direita, menos detalhado).

uma projecção inversa. A partir de um determinado limite consegue-se saber que há mais pixeis do que vértices e, portanto, baixa fidelidade. Nesta fase, itera-se pelos vários níveis de detalhe até chegar ao primeiro nível cuja diferença de vértices para pixeis seja a mínima e escolhe-se esse nível para desenhar a zona, enviando os vértices contidos nos blocos de terreno para o GPU após actualizar o grafo que contém apontadores para todos os elementos a desenhar.

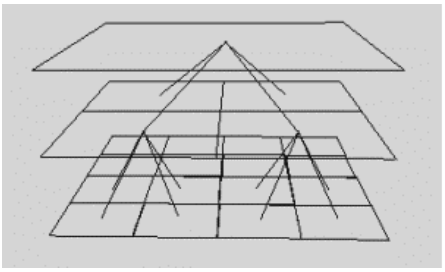

Figura 7.5: Vários níveis de pormenor. Cada bloco de terreno divide-se em quatro mais detalhados.

Os elementos com pedaços da geometria chamam-se blocos de terreno. Esta forma de agregação de vértices em blocos - *chunks* e de organização por níveis de detalhe na hierarquia de uma *quadtree* é aquilo a que se chama *Chunked LOD*[4].

Existe ainda uma restrição quanto ao número de amostras em cada bloco de terreno. Essa restrição implica que o número de vértices na aresta de cada bloco seja igual a uma potência de base dois mais uma unidade - $2^n + 1$. Como se pode ver pela figura 3.4, ao aumentar o nível de detalhe de um bloco de terreno, duplica-se a resolução, quadriplicando o número de vértices:

$$Verts_{LODn+1} = 4 \times Verts_{LODn} \tag{7.3}$$

Para que dois blocos de terreno possam ser desenhados adjacentes, têm que partilhar os vértices nas arestas que se tocam. Assim, é necessário ter 2^n vértices em cada aresta para ser possível duplicar a resolução a cada nivel LOD e é preciso acrescentar uma coluna e uma linha extra a cada bloco, num total de $2^n + 1$ vértices por aresta para que não haja singularidades entre os mesmos.

A escolha de n foi feita com base num compromisso entre, por um lado, ter pouca informação em cada bloco a fim de minimizar o consumo de memória e tempos de leitura do disco, e ter uma quantidade suficientemente grande que represente um melhoramento de performance quando o bloco tiver que ser cacheado. No motor T.R.E., o n é constante e igual a 32.

Como os *datasets* originais não têm, como aresta, um número de vértices que obedeça a estas condições, é necessário fazer *resample* para uma potência de base 2. As arestas originais têm, no entanto, uma relação de 2 : 1 em largura por altura. A escolha da dimensão final deve ser um compromisso entre tamanho do *dataset* resultante, e perda de informação. Neste caso, e porque o importante para este âmbito é desenvolver mecanismos de processamento de informação escaláveis, a dimensão final do *dataset* não é importante.

O objectivo de desenvolver algoritmos escaláveis para o pré-processamento pode ser ensaiado com uma quantidade de informação facilmente manejável e aplicado a um *dataset* maior no final. Por este motivo, foi feita uma sub-amostragem de 86400 amostras de largura para a potência de base dois imediatamente abaixo, 65536 amostras ou 2^{16}. Para demonstar o volume de informação que seria gerado com um *resample* para 2^{17}, o quadro de caracterização do *dataset* na secção anterior seria:

Informação resultante do pré-processamento. Aproximadamente 73.5GB

- **Topography** Conjunto de 131072x65536 amostras de $24bit$ contendo $16bit$ de elevação em

relação ao raio médio do planeta (fusão entre *topo* e *bathy*) e 8*bit* de máscara de presença de água para efeitos de reflexos (*specular highlight*) (aprox 24.5GB)

- **Normals** Conjunto de 131072x65536 amostras de 24*bit*. Cada amostra é um vector normal à superfície em $\mathbb{R}^3$ (aprox 24.5GB)
- **Raw Color Data** Conjunto de 131072x65536 amostras de 24*bit* com informação de côr média. Contém 3 canais de 8*bit* cada (aprox 24.5GB)

E com sub-amostragem para dimensão 2^{16}:

Informação resultante do pré-processamento. Aproximadamente 18.4GB

- **Topography** Conjunto de 65536x32768 amostras de 24*bit* contendo 16*bit* de elevação em relação ao raio médio do planeta (fusão entre *topo* e *bathy*) e 8*bit* de máscara de presença de água para efeitos de reflexos (*specular highlight*) (aprox 6.1GB)
- **Normals** Conjunto de 65536x32768 amostras de 24*bit*. Cada amostra é um vector normal à superfície em $\mathbb{R}^3$ (aprox 6.1GB)
- **Raw Color Data** Conjunto de 65536x32768 amostras de 24*bit* com informação de côr média. Contém 3 canais de 8*bit* cada (aprox 6.1GB)

Confirmando que o total de informação a um nível LOD_{n+1} é o quádruplo da informação no nível LOD_n. Desta forma é fácil estimar a quantidade de informação necessária para cada nível de resolução em $m/pixel$:

$m/pixel$	Largura	Altura	$n/bloco$	LOD	Espaço Ocupado
1318	32768 (2^{15})	16384	32	10	4.6GB
660	65536 (2^{16})	32768	32	11	18.4GB
500	86400	43200	32	12	21.5GB (BMNG original)
250	131072 (2^{17})	65536	32	13	73.5GB
164	262144 (2^{18})	131072	32	14	294GB

Tabela 7.1: Relação entre resolução de um *dataset* e espaço de armazenamento necessário.

Dado que o número de amostras por bloco é constante, o número de blocos varia consoante a dimensão do *dataset*. Assim, o número de blocos de largura em cada configuração de *dataset*, com a dimensão 2^n é dado por:

$$p = 2^n / v_{bloco} \tag{7.4}$$

E o número de níveis *LOD* em cada configuração é dado por:

$$l = log_2(p) \tag{7.5}$$

7.3 *Pipeline* de transformação

Uma vez identificados os diferentes processos de pré-processamento, é possível especificar uma *pipeline* com a sequência de operações que produzirá os dados para uso no motor gráfico. Os passos identificados são:

- **Projecção** dos vários *datasets* de cilíndrica equidistante para gnomónica resultando em seis partições/projecções.
- **Particionamento** de cada projecção para leitura selectiva de dados e caching de cada conjunto de dados.

8

Manipulação CPU

No contexto de aplicação residente em CPU são criadas várias estruturas de dados para guardar a informação topográfica, de texturas e mapas de normais às superfícies. Estas estruturas são referenciadas por listas de objectos a desenhar e são geridas por um módulo de *cache* que tenta acelerar o desenho de objectos que persistam temporalmente na zona de foco visível ao observador. O método de leitura e manipulação destes dados passa por uma cadeia de fases.

Para facilitar a compreensão do fluxo de execução, torna-se essencial explicar o *loop* principal do motor gráfico. Este *loop* consiste num conjunto de fases que enumerarei e pormenorizarei de seguida:

- Processamento de eventos do utilizador
- Cálculo de intervalo de tempo desde última execução do ciclo
- Reposicionamento e reorientação de objectos
- Desenho de objectos

8.1 Processamento de eventos

Esta fase consiste no processamento de eventos gerados pelo utilizador com o teclado e rato e subsequente ajustamento da posição e orientação no espaço do observador. O processo passa por analisar que acções o utilizador desempenhou através da extracção de dados de uma tabela de estados associada a cada evento gerado. Esses estados podem ser alteração de teclas pressionadas ou de alteração de valores nos eixos ou botões do rato e são tratados em duas funções distintas: $processClicks()$ e $processKeys()$.

Estas funções permitem adicionar às variáveis que contêm posição e orientação do observador. O mecanismo é, para qualquer uma destas variáveis, o seguinte:

$$state_n = state_{n-1} + fixed\_val * dt \tag{8.1}$$

8.2 Cálculo de intervalo temporal entre *frames*

Para ser possível processar os cálculos que permitem a simulação de posições de fontes de iluminação e gravitação, é necessário calcular o tempo passado entre cada iteração dos mesmos. A forma utilizada para o fazer é o armazenamento, numa estrutura de dados, da data da última execução destes cálculos, com precisão de micro-segundos. A cada iteração é calculada a diferença entre o momento de execução e o valor armazenado anteriormente e é armazenado o delta temporal - *dt* - que será utilizado nos cálculos subsequentes.

8.3 Reposicionamento e reorientação de objectos

Sempre que o utilizador interage com o sistema, alterando a sua posição ou orientação, o modelo deve reflectir estas mudanças. Mesmo que o utilizador não interaja com o sistema, a natureza dinâmica do modelo físico no que diz respeito ao posicionamento

dos corpos planetários e fonte de luz, provoca alterações nas posições destes corpos relativamente ao observador.

Desta forma, e num referencial relativo ao observador, é irrelevante qual o objecto que sofreu alterações para o objectivo de reflectir estas modificações. Assim, não é utilizada nenhuma forma de validar se é necessário reprocessar as posições e orientações dos corpos. Estes parâmetros são recalculados em todas as iterações.

As posições de todos os objectos e do observador são relativas a um referencial centrado no corpo planetário de maior interesse. Isto permite que os vectores de posição tenham a menor norma possível, minimizando problemas de precisão que resultariam de eventuais *overflows* às variáveis que o OpenGL utiliza para as guardar.

Para a orientação do observador, é necessário guardar a rotação do mesmo nos 3 eixos correspondentes às coordenadas espaciais. É possível guardar apenas os valores escalares de ângulos Euler em cada eixo. No entanto, e porque dessa forma, a ordem de aplicação de rotações seria relevante e sujeita a problemas de singularidades em funções trigonométricas como, por exemplo, no cálculo de $tan\frac{\pi}{2}$, optei pela utilização de uma estrutura mais apropriada - um quaternião[2].

Os quaterniões são um sistema numérico que extende os números complexos. São representados internamente como vectores de 3 dimensões imaginárias e uma real.

A vantagem na utilização desta estrutura é que é possível aplicar os deltas de rotação ao quaternião em cada iteração sem qualquer preocupação com o seu estado interno, singularidades, ou ordem de rotações. Uma vez processados estes pequenos ajustes de rotação, o quaternião é normalizado e é extraída uma matriz que representa a rotação do objecto. Esta matriz pode ser passada directamente ao OpenGL e representa a orientação do observador. Todos estes cálculos estão implementados num *template* C++ - *Quaternion.h*.

Para além dessa vantagem, há uma outra importante. Com a utilização de quaterniões, a rotação de um objecto em relação a outro - muito útil para cálculos de

órbitas - é tão trivial quanto multiplicar o quaternião de rotação do primeiro pelo do segundo.

Sendo que todos os objectos neste motor são subclasses de uma classe genérica de elemento numa árvore de objectos, todo o código que diz respeito ao posicionamento e orientação pertence à classe mãe: *Node*. A classe *Camera* partilha toda esta lógica, mas contém mais algum código para o posicionamento e orientação do observador. Este código é utilizado exclusivamente para a construção dos referenciais de objecto, observador e projecção do OpenGL.

8.4 Desenho de objectos

O desenho dos objectos que estão presentes numa cena obedece a uma lógica hierárquica. Todos estes objectos estão inseridos numa árvore que contém a lista de geometria a desenhar em cada iteração do ciclo. Esta lista é alimentada por um módulo de detecção de visibilidade. Assim, em cada iteração é analizada a visibilidade de todos os objectos dentro da zona visível. Os objectos que não estão em memória são instanciados e inseridos numa fila de elementos que precisam de ser carregados em memória.

Existem objectos que são sempre desenhados e outros que são desenhados dependendo da sua visibilidade. O único objecto sempre desenhado, e que tem uma geometria muito simples, é o Sol. A decisão de o reposicionar e desenhar sempre tem que ver com a importância da sua presença para os cálculos de gravitação e iluminação de toda a cena.

Em linhas gerais, o processo de desenho dos objectos obedece à seguinte lógica:

- Calcular tempo passado deste última frame e gravar hora actual
- Processar cálculos de física para observador e aplicar transformações
- Reposicionar luzes e desenhar sol
- Para cada planeta - Executar função de desenho de cena planetária

- Para cada objecto adicional - Aplicar cálculos de física e desenhar

A função *drawPlanetScene* é responsável por analizar a àrvore de elementos de terreno que devem ser desenhados de acordo com a posição e orientação do observador em cada planeta. O processo passa por determinar o cone visível ao observador, armazená-lo numa lista de valores visível a todas as entidades e, de seguida, iniciar o ciclo de desenho da classe *Planet*. Por motivos de performance e ordem de desenho de superfícies com transparência, o desenho é partido em duas fases: o desenho da atmosfera do planeta e o desenho do terreno.

O desenho da atmosfera passa por desenhar a superfície de uma semi-esfera com a concavidade orientada para o observador. Isto permite a visibilidade de um halo à volta do terreno. A função que desenha a atmosfera apenas desenha os vértices e recalcula os parâmetros de iluminação da mesma. Estes parâmetros são depois passados ao programa que será executado no GPU para o desenho propriamente dito dos *pixels* que constituem a atmosfera.

Quanto ao desenho do terreno, divide-se na fase de construção de uma árvore de elementos visíveis, ordenação dos mesmos por distância, reorganização da *cache* de elementos desenhados com frequência e posterior desenho da lista final. A sequência de operações é a seguinte:

- Activar shader
- Enviar posição de observador, luzes, parâmetros atmosféricos para shader
- Para cada face do cubo do planeta, executar função de análise de visibilidade

8.5 Gestão das faces do cubo

A estrutura que implementa a *quadtree* que constitui cada face do cubo de um planeta é implementada numa classe $TerrainNode$ e utiliza uma outra instância estática para solicitar a leitura de blocos de terreno de disco para memória: a *TerrainManager*.

Quando um planeta é instanciado, as suas 6 faces também o são. Uma vez instanciadas, é-lhes atribuído o nível de pormenor mais baixo e menos complexo. Este nível está sempre residente em memória e constitui a raíz da árvore. Assim que a função de desenho deste nível é executada, através da chamada referida acima *analyse*(), é iniciado o processo recursivo de determinação de visibilidade e necessidade de leitura de níveis de pormenor maior.

O processo de inicialização de qualquer bloco de terreno - incluindo a raíz da árvore - é o seguinte:

- Copiar informação sobre posição e orientação deste bloco recebida no construtor
- Determinar se este bloco tem endereço entre [1-4] ou se é um bloco raíz da árvore
- Marcar este bloco como não registado na lista de blocos a desenhar
- Marcar data de última vez desenhado como NULL
- Marcar os blocos filhos como não alocados
- Marcar este bloco como não pronto para ser utilizado pelo OpenGL. Este processo corre numa *thread* solta. O OpenGL não tem suporte para executar aqui. Assim que terminarmos, marcamos o bloco como pronto e será processado pelo OpenGL na *thread* principal
- Calcular posição do centro deste bloco e escalar somando o seu versor com o raio do planeta
- Inserir pedido de leitura de dados topográficos para este bloco numa lista

Após esta inicialização, inicia-se a leitura da geometria do terreno a partir do disco. A classe *TerrainManager* tem uma *pool* de *threads* para leitura simultânea de blocos de terreno. O número de *threads* é configurável num ficheiro. Nos sistemas testados, o nível máximo de *threads* a partir do qual a concorrência no acesso ao disco satura o máximo desempenho de *IOops* é 4.

Cada uma destas *threads* tem um ciclo de processamento intensivo para verificação de elementos $TerrainNode$ numa lista de pedidos de leitura. Estes elementos adicionam-se a eles próprios a esta lista com a chamada anterior $requestLoad$. Esta função executa a seguinte lógica:

Listagem 8.1: Código executado por cada bloco de terreno para se adicionar a uma lista.

```
TerrainManager::requestLoad(TerrainNode *node) {
    // add node to list
    pthread_mutex_lock(&_mutex);
    _nodeQueue.push_back(node);
    pthread_mutex_unlock(&_mutex);
}
```

E o ciclo de execução de cada uma das *threads* de leitura executa a seguinte lógica:

Listagem 8.2: Pseudo-código do ciclo de *thread* de leitura de geometria do disco.

```
TerrainManager::spawnThread(void *_NULL) {
    while(1) {
        // check if list has items
        pthread_mutex_lock(&_mutex);
        size_t count = _nodeQueue.size();
        if (count > 0) {
            // sort them by distance to make sure closer nodes load first
            _nodeQueue.sort(TerrainNode::compareByDistance);

            // load
            std::list<TerrainNode *>::iterator i = _nodeQueue.begin();
            TerrainNode *node = *i;
            _nodeQueue.pop_front();

            // release ASAP
            pthread_mutex_unlock(&_mutex);
            node->asynch_init();
        } else {
            pthread_mutex_unlock(&_mutex);
            usleep(1000);
        }
    }
}
```

É de destacar que como resultado desta lógica, o bloco de terreno na árvore vai

executar uma função - *asynch_init()* - numa *thread* separada. O OpenGL não suporta *multithreading* e qualquer chamada a uma das suas funções fora da principal causa um *crash* imediato. No entanto, a inicialização do bloco precisa de utilizar estas chamadas. Por esta razão, a inicialização é feita em duas fases. A primeira fase copia a geometria do disco num fluxo de execução secundário. A segunda fase corre no fluxo principal e trabalha esta informação para poder ser utilizada pelo OpenGL. Como a segunda fase é muito leve em termos de complexidade temporal, não há atrasos nem paragens consideráveis no fluxo principal que causem perturbações à ilusão de movimento e fluidez de animação.

A função de inicialização de um bloco fora do ciclo principal obedece à seguinte lógica:

Listagem 8.3: Código de inicialização de um bloco em *thread* separada.

```
TerrainNode::asynch_init() {
    // load topography from disk
    uint16_t tex_size = min(32*pow(2, _level), 512);
    size_t bytes_per_sample = temp_bps = (_planet->_maxlod == 8) ? 3 : 4;

    _rawTopo = (uint8_t *)malloc(sizeof(uint8_t)*tex_size*tex_size*2);
    _rawTexture = (uint8_t *)malloc(sizeof(uint8_t)*tex_size*tex_size*bytes_per_sample);
    _rawNormals = (uint8_t *)malloc(sizeof(uint8_t)*tex_size*tex_size*3);

    BlueMarbleTopo::readTopography(_planet, _cubeface, _qtindex, _level, _rawTopo);
    BlueMarbleTopo::readTextures(_planet, _cubeface, _qtindex, _level, _rawTexture);
    BlueMarbleTopo::readNormals(_planet, _cubeface, _qtindex, _level, _rawNormals);

    // mark us as initialized. the render queue will call the secondary init
    // which uses OpenGL calls inthe main thread to make us ready to draw
    initialized = true;
}
```

Esta função poderá demorar algum tempo a executar visto que vai utilizar uma classe estática auxiliar para ler os dados topográficos, texturas e mapa de normais do

disco. Estas operações têm uma complexidade espacial consideravelmente grande. Um bloco de terreno de 32x32 vértices a um nível de detalhe diferente do natural dos dados originais precisa de saltar a cada n amostras para criar uma representação de resolução inferior. Estes saltos são executados com operações $fseek$ e $fread$ intercaladas.

O número total de operações para qualquer nível de detalhe diferente do natural é igual a $2 + 32 \times 32$ e corresponde à operação $fopen$, respectivas procuras em disco e $fclose$. Estas operações são repetidas para a topografia, texturas e normais.

Após esta fase, a inicialização do bloco de terreno é finalizada criando as estruturas OpenGL que serão desenhadas e inicia-se a fase de análise de visibilidade e desenho. A análise de visibilidade do seu pai determinou que este bloco era visível, pelo que não há instanciação de blocos não visíveis num determinado momento. A excepção é o bloco que constitui a raíz da árvore.

Esta análise processa-se da seguinte forma:

- Abortar se bloco não estiver inicializado
- Se inicialização OpenGL não estiver feita, executá-la agora em thread princiapl
- Abortar se bloco não estiver dentro do cone visível ao observador
- Determinar métrica de erro para saber se este bloco tem detalhe suficiente para ser desenhado ou se se deve desenhar os seus filhos
- Se for necessário desenhar os filhos, pedir incialização dos mesmos caso necessári
- Se estiverem printos a desenhar, executar função *analyse* de cada um e abortar imediatamente. Esta função é, obviamente, recursiva.
- Se execução chegar aqui, este bloco tem resolução suficiente. Determinar se está oculto pela curvatura do planeta e desenhá-lo caso seja visível.

É importante destacar a última verificação feita neste pseudo-código. É possível que um bloco de terreno esteja dentro da zona visível ao observador, mas que esteja oculto pela curvatura do planeta. Nestes casos, é feita uma optimização adicional que consiste em descartar o bloco e não invocar a função *draw*. Para todos os efeitos, é

tratado como se não estivesse visível e, portanto, a data de última inserção da *queue* de desenho não é actualizada.

9

Render GPU

Na capítulo anterior foi pormenorizada toda a lógica de leitura de topografia, texturas e mapas de normais de disco, gestão destes elementos em memória e preparação dos dados para injecção em OpenGL nas variáveis que são utilizadas no GPU.

Nesta secção será pormenorizada a forma como a *pipeline* de função-fixa do OpenGL é substituída por programas novos, especialmente desenhados para o desenho de terrenos planetários e atmosfera. Estes programas - *shaders* - são escritos na linguagem GLSL - *GL Shading Language*, que é muito semelhante a C com suporte SIMD inerente à natureza vectorial das unidades de execução de *shaders* dos GPUs.

9.1 Funcionamento dos *Shaders*

Estes programas dividem-se em dois tipos: *vertex shaders* e *fragment shaders*. Os primeiros são responsáveis pela transformação dos vértices e aplicar informação topográfica aos mesmos a partir das texturas que a codificam. O resultado é, posteriormente, enviado para os segundos. O método passa por interpolar todos os *pixels* que compoem as superfícies entre vértices e, para cada um deles, aplicar a lógica de *render* de um fragmento.

Um fragmento é um candidato a *pixel*. Sempre que se determina que dois fragmentos vão ocupar a mesma zona do ecran, descarta-se o menos importante para a cena e só outro é realmente processado pelo programa de colorização final - *fragment shader*.

9.2 *Shader* de transformação de vértices

Tal como foi descrito anteriormente, a geometria é completamente plana até chegar ao GPU. A informação topográfica é enviada para uma unidade de texturas da placa gráfica como se de uma imagem se tratasse e é lida pelo programa de desenho de terreno. A informação de altitude é carregada para um vector e somada a cada vértice com a seguinte lógica:

Listagem 9.1: Codigo GLSL para transformacao de vértices.

```
void main(void) {
    // obter coordenadas de textura passadas pelo CPU para a unidade
    // de texturas 0
    gl_TexCoord[0] = gl_MultiTexCoord0;

    // recalibrar altitude de 0-65535 para -32768,32767
    // descartar arredondamentos anteriores para vector nulo [0-1000]
    if (length(gl_Vertex.xyz) > 1000.0) {
        vec4 heightmap = texture2D(heightTexture, gl_TexCoord[0].st);
        float height = heightmap.a * 65536.0 - 32768.0;
        height = floor(height);
        orig_vertex = vec4(normalize(gl_Vertex.xyz) * (height + fInnerRadius), 1.0);
    } else {
        orig_vertex = gl_Vertex;
    }

    // Aplicar transformacao de espaco objecto para referencial mundo
    v_vertex = gl_ModelViewMatrix * orig_vertex;
    gl_Position = gl_ModelViewProjectionMatrix * orig_vertex;
}
```

No final desta fase, todos os vértices estão prontos para o cálculo da superfície que os une. Todos os pontos desta superfície - os fragmentos - são posteriormente enviados para o próximo estágio - *fragment shader*.

9.3 *Shader* de produção de fragmentos

A lógica de transformação é repetida, já que o cálculo anterior apenas serve para determinar a altitude de cada ponto para o cálculo de declive que é utilizado para conferir algum pormenor adicional à superfície, como por exemplo, se deve ser mais rochosa ou mais arenosa.

A lógica de processamento vectorial na unidade de fragmentos é a seguinte:

Listagem 9.2: Codigo GLSL para transformacaoo e desenho de vertices.

```
void main(void) {
    // recalibrar altitude de unsigned para signed
    vec4 heightmap = texture2D(heightTexture, gl_TexCoord[0].st);
    float real_height = heightmap.a * 65536.0 - 32768.0;
    float height = floor(real_height);

    // aplicar textura de Blue Marble Next Generation
    vec4 terrain = texture2D(terrainTexture, gl_TexCoord[0].st);

    // se nivel de detalhe for o maior, adicionar pormenor gerado com gaussian noise
    vec4 detail = vec4(1.0);
    if (lod < 1)
        detail = texture2D(terrainDetailTexture, gl_TexCoord[0].st*16.0) * 2.0);

    // obter normais com Z comum apontado para cima de textura
    // escalar de 0-1 para -1,1
    vec3 normal = texture2D(normalTexture, gl_TexCoord[0].st).xyz;

    // calcular declive da superficie para pormenores
    float slope = abs(dot(vec3(0.0, 0.0, 1.0), normal.rbg));
```

```
    // normais sao orientadas todas com base no mesmo referencial
    // rodar cada uma com base no referencial do espaco tangente
    // a superficie esferica do planeta
    normal *= vec3(1.0, 1.0, 1.0);
    mat3 rot_matrix = fromToRotation(vec3(0.0, 0.0, 1.0), normalize(orig_vertex.xyz));
    normal = normalize(normal * rot_matrix);
    normal = normalize(gl_NormalMatrix * normal);
}
```

A rotação de normais executada neste passo é extremamente importante para o resultado final. As normais que chegam a esta fase estão orientadas para o mesmo referencial. Este referencial corresponde à projecção cilíndrica do planisfério com o eixo x orientado para longitude crescente a este de Greenwich, o eixo y orientado para norte e o eixo z perpendicular à superfície do planisfério. Assim, o eixo Z não só simboliza a componente maior, visto os vectores de altitude serem geralmente perpendiculares à superfície, como é o mesmo eixo para todas. No entanto, os vértices inscritos na superfície esférica têm sistemas de coordenadas próprios. Para estes vértices, o eixo Z é sempre perpendicular à superfície esférica e não à projecção cilíndrica da mesma superfície.

Como este eixo é partilhado por todos e a maior componente de todas as normais é a Z, e estão representadas como cores, a mesma coordenada equivale ao canal azul - $a.xyz = a.rgb$. Por esta razão, no mapa de normais não transformado para o espaço tangente, a côr predominante é o azul. Uma vez transformadas para o referencial geocêntrico, as cores representam a orientação face aos 3 eixos ortogonais com origem no núcleo do planeta. As figuras 9.1 e 9.2 ilustram as normais antes e depois da reorientação ao espaço tangente à superfície para o referencial geocêntrico do mundo desenhado.

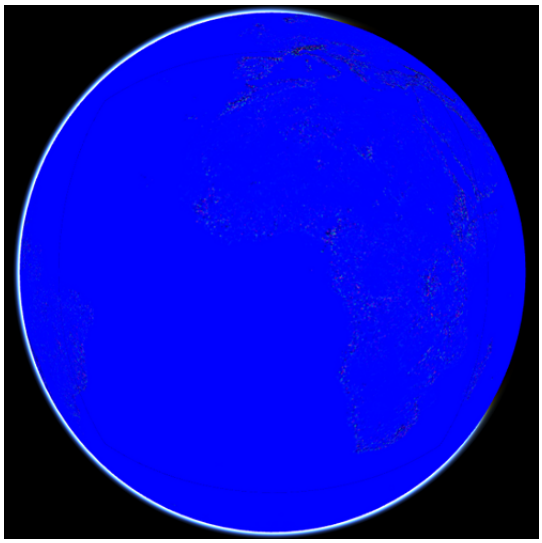

Figura 9.1: Normais em referencial espaço tangente.

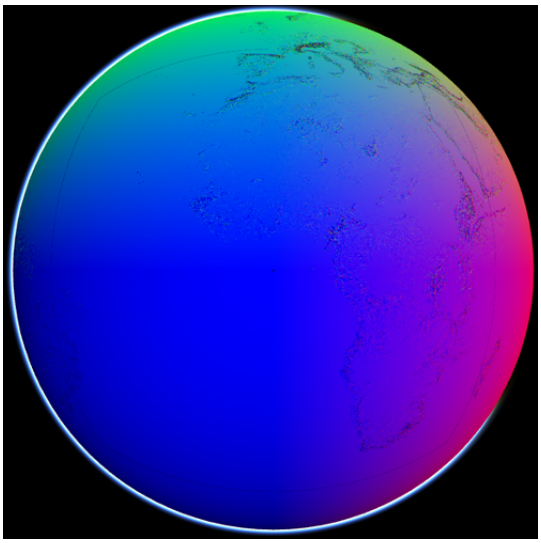

Figura 9.2: Normais em referencial geocêntrico.

9.4 Método para rotação de normais

A cada normal que precisa de ser rodada, corresponde um vértice já transformado. O processo de rotação passa por determinar a rotação necessária para passar alinhar o vértice transformado para o mesmo referêncial das normais. Assim, é calculada uma matriz de rotação que transforma o vértice do seu referencial actual - geocêntrico - para o referencial XYZ alinhado ao planisfério. Esta matriz é, posteriormente invertida e multiplicada pelo vector da normal, passando-a do referencial XYZ planisférico para o geocêntrico. A álgebra do processo é algo complexa e é implementada exclusivamente em cálculo vectorial pela função fomToRotation directamente no GPU, no *shader planet.glsl*.

10

Conclusão

O trabalho descrito nos capítulos anteriores desta dissertação constitui uma estratégia para atingir um conjunto de objectivos que, de forma resumida, são:

- Desenho de superfícies planetárias com topografia real
- Baixa complexidade tecnológica para adaptação simples a cada cenário de utilização
- Performance satisfatória para vários cenários de utilização sem atrasos perceptíveis

Serão apresentados resultados que clarificam a forma como todos os objectivos foram atingidos. Esta apresentação será feita com base em resultados gráficos e dados de execução da aplicação e programas no GPU. Os dados de execução permitem inferir o quão fluída é a produção de imagens e quão intensiva é a utilização de recursos de hardware para o conseguir.

10.1 Desenho de superfícies planetárias com topografia real

Independentemente de se ter ou não atingido o objectivo de manter fluidez suficiente na geração de imagens para conseguir uma cadência mínima que permita a percepção

de animação pelo olho humano, a qualidade das imagens geradas cumpre com os objectivos.

Foi possível, com o conjunto de algoritmos e técnicas desenvolvidas, criar superfícies com topografia real, texturas reais e iluminação correcta. O pormenor destas imagens poderia, no entanto, ser substancialmente melhorado no que diz respeito à precisão topográfica com a utilização de *datasets* mais recentes - como proposto em trabalho futuro.

É importante salientar que não existe qualquer representação de intervenção humana no terreno. As texturas representam uma tentativa de colorização da geometria de acordo com parâmetros definidos pela própria NASA que são, entre outros, a reflecção provocada por vegetação, declive e altitude do terreno, cobertura de neve baseada na latitude e estação do ano, entre outros.

As figuras ilustram a qualidade dos resultados obtidos. Os executáveis e dados em anexo permitem a reprodução dos mesmos.

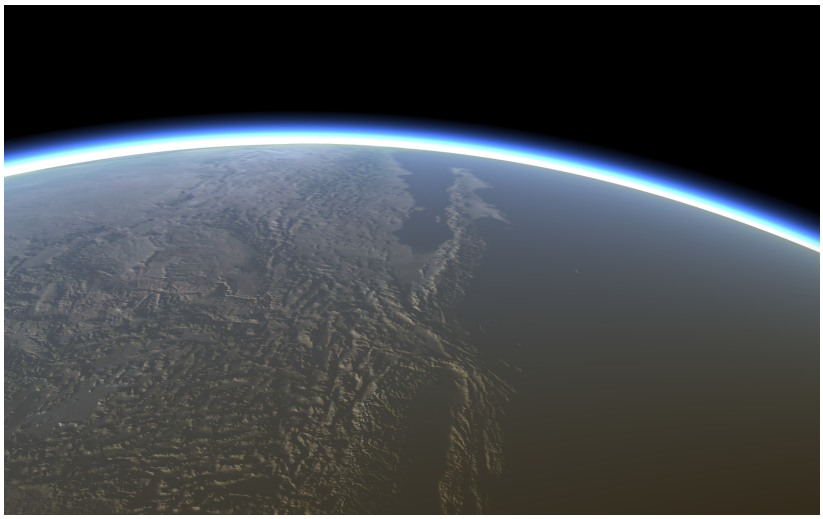

Figura 10.1: América do Norte vista de órbita.

Todas estas imagens foram produzidas numa máquina modesta, com um processador Intel Core 2 Duo a 2.0GHz, com uma placa gráfica não dedicada. Esta placa gráfica tem 256MB RAM partilhadas com a RAM principal do sistema e é uma NVIDIA

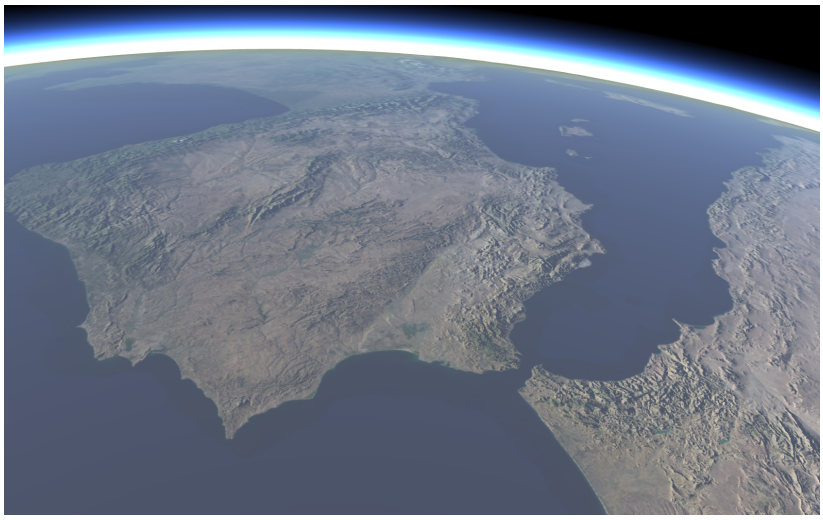

Figura 10.2: Península Ibérica vista de órbita.

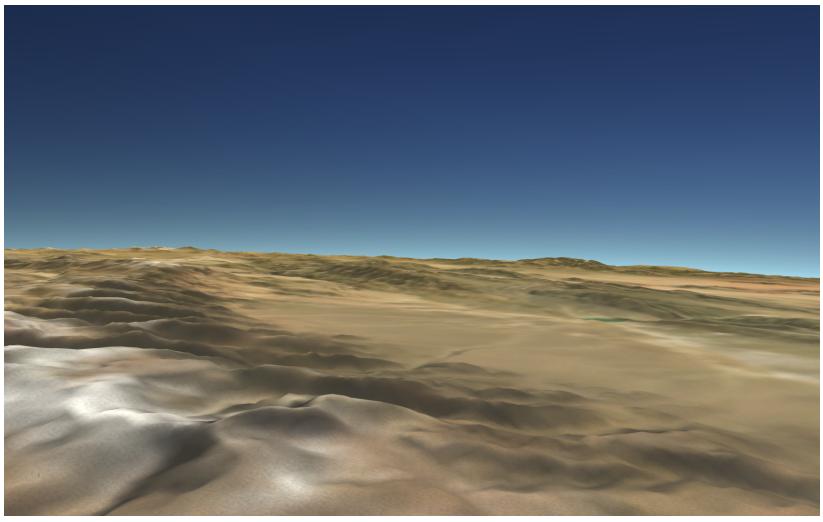

Figura 10.3: Cordilheira dos Andes vista do terreno.

GeForce 9400M. As imagens de controlo aqui presentes foram geradas em Mac OS X 10.6 com uma implementação de OpenGL 2.1, várias gerações anterior à actual 4.1.

A utilização de uma placa gráfica com características mais evoluídas poderia apenas acelerar a execução dos programas em GPU e permitir a utilização de *antialiasing*.

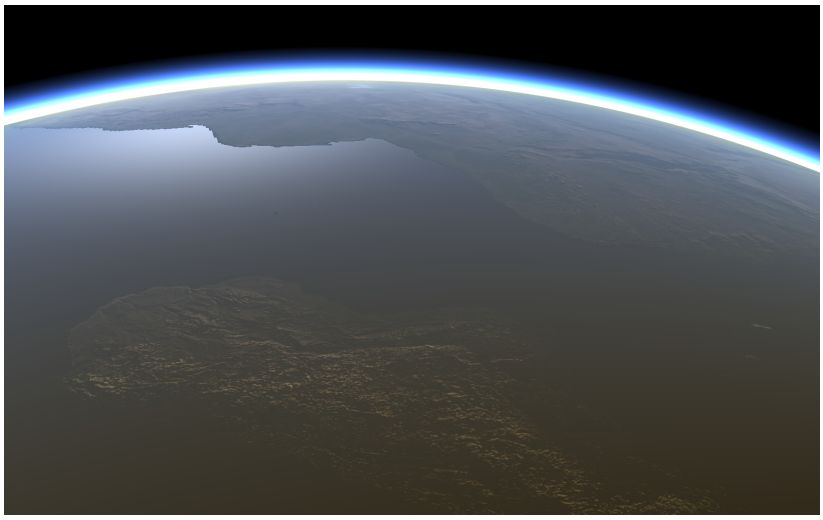

Figura 10.4: Pôr-do-sol sobre Madagáscar visto de órbita.

10.2 Baixa complexidade tecnológica

O motor gráfico desenvolvido é constituído por um número pequeno de classes, completamente estanques e modulares. Toda a lógica de desenho de terrenos é implementada pela classe que lida exclusivamente com geometria do terreno. Esta classe solicita a leitura de informação topográfica, de texturas e de normais a uma classe auxiliar. Esta classe auxiliar *BlueMarbleTopo* implementa código para interpretar os dados BMNG e fornece-los numa representação interna ao motor gráfico. Este método permite que se escrevam adaptadores diferentes para formatos externos diferentes, mantendo a arquitectura do motor inalterada.

O total de linhas de código, incluindo comentários extensos, protótipos, declarações de estruturas de dados e bibliotecas auxiliares, para a leitura de dados do disco e para o seu desenho, é de 1111. O total de linhas de código na implementação completa do motor, incluindo *Makefiles* que suportam Linux e Mac OS X, é de 4649.

Esta decisão de desenho permite, por um lado, conferir modularidade aos componentes do motor, e por outro, implementar o desenho de geometria complexa de fontes de dados de dimensão arbitrária de uma forma extremamente simples.

Seria possível simplificar ainda mais a arquitectura e generalizar algumas das regras de desenho que estão, de momento, replicadas em várias classes. Tal não foi conseguido por constrangimentos temporais na elaboração deste trabalho, mas será proposto como trabalho futuro para uma nova versão deste motor gráfico.

10.3 Performance

Não é, para além da percepção de movimento, possível classificar em termos de performance, o desempenho de um motor deste tipo, utilizando outras métricas que não os perfis de execução dos binários. Assim, apresento os perfis que demonstram a quantidade de memória RAM, utilização de CPU e operações IO de acesso aos dados em disco durante a execução de uma aproximação a um ponto do terreno, a partir de órbita.

Os perfis de execução correspondem ao tempo total decorrido entre o início da aproximação e, após atingido o ponto final, o momento em que todo o terreno está carregado com o nível máximo de pormenor.

Para estas execuções, a cache de blocos é aquecida com todos os que serão desenhados na aproximação. Este corresponde ao caso típico de leitura destes blocos pré-gerados a partir de disco ou de outro meio, como, por exemplo, num cenário de streaming, de um servidor de blocos remoto.

A máquina em que esta execução acontece é a mesma utilizada para a geração das imagens anteriores. O desenho acontece em *fullscreen* correspondente ao modo de aceleração máxima em Mac OS X, com uma resolução de 1280x800 *pixels*, 24 bits de profundidade de cor e sem *antialias*.

A figura 10.5 é um gráfico de toda de memória alocada entre estes dois instantes. O ponto máximo corresponde ao total alocado. Este valor é o valor que aparece na tabela da figura 10.6. A alocação de blocos começa no primeiro salto do gráfico, aos 7 segundos. O final acontece aos 25 segundos. Durante todo este tempo não houve, em

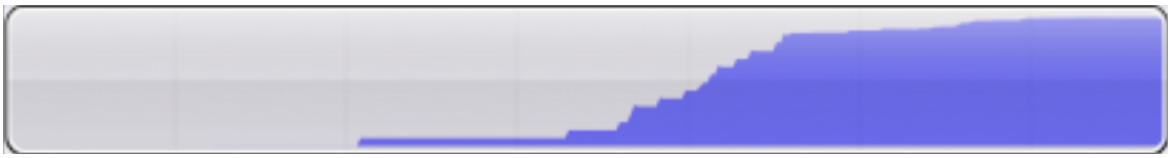

Figura 10.5: Memória alocada ao longo do tempo.

qualquer momento, nenhuma espera perceptível pela leitura dos dados.

A tabela na figura 10.6 corresponde ao total de memória alocada por tamanho de bloco pedido à função de alocação. Os 3 maiores blocos, de 512KB a 1MB correspondem à alocação de espaço para terreno, normais, e texturas. A coluna *Transitory* conta o total de blocos alocados e a coluna *Overall* o total de espaço alocado para todos os blocos de um tipo. Pela análise conclui-se que uma aproximação a um ponto no terreno e respectivo desenho, com qualidade máxima, necessita de aproximadamente 382,98MB de RAM. Este valor corresponde a aproximadamente 1/5 do total de memória numa máquina modesta com 2GB de RAM.

Uma vez desenhado algum terreno no nível máximo de pormenor, caso o utilizador se afaste desse ponto e escolha outra zona de foco, as zonas que deixam de ser desenhadas vão expirando por inactividade e vão sendo desalocadas.

A figura 10.7 ilustra quanto foi a utilização de CPU pela aplicação face ao total dos recursos disponíveis no sistema e a figura 10.8 representa o tempo total de CPU gasto nesta execução.

Como se pode observar, a utilização de blocos pré-gerados faz com que praticamente todo o tempo de CPU seja gasto na análise e desenho de geometria. Nesta lista, as percentagens são relativas ao total gasto pela aplicação.

Os níveis de recursos necessários para a construção deste resultado final, quer no que diz respeito à utilização de CPU, quer de consumo de memória e respectiva gestão, quer de saturação de *IO* ilustram o quão bem sucedido foi o trabalho e demonstram o cumprimento, em pleno, dos objectivos inicialmente propostos.

Graph	Category	Live Bytes	# Living	# Transitory	Overall Bytes
☑	* All Allocations *	1,73 MB	2658	595434	382,98 MB
☐	Malloc 1,00 MB	0 Bytes	0	114	114,00 MB
☐	Malloc 768,00 KB	0 Bytes	0	120	90,00 MB
☐	Malloc 512,00 KB	0 Bytes	0	113	56,50 MB
☐	Malloc 96,00 KB	0 Bytes	0	255	23,91 MB
☐	Malloc 80 Bytes	3,36 KB	43	248096	18,93 MB
☐	Malloc 1,01 MB	0 Bytes	0	14	14,16 MB
☐	Malloc 520,00 KB	0 Bytes	0	21	10,66 MB
☐	Malloc 64,00 KB	0 Bytes	0	158	9,88 MB
☐	Malloc 196,00 KB	0 Bytes	0	30	5,74 MB
☐	Malloc 256,00 KB	0 Bytes	0	20	5,00 MB
☐	Malloc 16 Bytes	3,14 KB	201	286501	4,37 MB
☐	Malloc 132,00 KB	0 Bytes	0	32	4,12 MB
☐	Malloc 26,00 KB	0 Bytes	0	112	2,84 MB
☐	Malloc 4,00 KB	0 Bytes	0	523	2,04 MB
☐	Malloc 260,00 KB	0 Bytes	0	8	2,03 MB
☐	Malloc 352 Bytes	1,03 KB	3	5415	1,82 MB
☐	Malloc 13,00 KB	0 Bytes	0	128	1,62 MB
☐	Malloc 48,00 KB	0 Bytes	0	32	1,50 MB
☐	Malloc 66,50 KB	0 Bytes	0	16	1,04 MB
☐	Malloc 516,00 KB	0 Bytes	0	2	1,01 MB
☐	Malloc 32,00 KB	0 Bytes	0	32	1,00 MB

Figura 10.6: Total de memória alocada por tipo de bloco.

Figura 10.7: Total de CPU utilizado face ao total disponível.

Running Time		Symbol Name
7299.3ms	56.6%	TWS::TerrainNode::draw() tws-mac
7299.3ms	56.6%	TWS::TerrainNode::analyse() tws-mac
4492.3ms	34.8%	TWS::SceneManager::run() tws-mac
418.4ms	3.2%	TWS::BlueMarbleTopo::resizeImage(unsigned char*, int, int, int, unsigned char*, int, int) tws-mac
198.1ms	1.5%	TWS::TerrainNode::gl_init() tws-mac
89.1ms	0.6%	TWS::BlueMarbleTopo::readNormals(TWS::Planet*, unsigned long, unsigned long, unsigned long, unsigned char*) tws-mac
80.1ms	0.6%	TWS::BlueMarbleTopo::readTextures(TWS::Planet*, unsigned long, unsigned long, unsigned long, unsigned char*) tws-mac
72.9ms	0.5%	SDL_SoftStretch SDL
48.5ms	0.3%	TWS::BlueMarbleTopo::readTopography(TWS::Planet*, unsigned long, unsigned long, unsigned long, unsigned char*) tws-mac
24.7ms	0.1%	TWS::Planet::drawTerrain() tws-mac
24.4ms	0.1%	TWS::TerrainManager::spawnThread(void*) tws-mac
20.2ms	0.1%	TWS::boxInFrustum(double*) tws-mac
19.4ms	0.1%	TWS::Planet::drawAtmosphere() tws-mac
15.6ms	0.1%	TWS::TerrainNode::analyse() tws-mac
10.1ms	0.0%	TWS::Shader::bind() tws-mac
9.9ms	0.0%	TWS::Vector3<double>::normalized() tws-mac
8.6ms	0.0%	TWS::Vector3<double>::z() tws-mac
8.4ms	0.0%	TWS::Vector3<double>::set(double, double, double) tws-mac
7.5ms	0.0%	TWS::SceneManager::draw() tws-mac
7.0ms	0.0%	TWS_Astro::sunpos(double, int, double*, double*, double*, double*) tws-mac
5.7ms	0.0%	TWS::AudioNode::setPosition(TWS::Vector3<double>) tws-mac
4.9ms	0.0%	TWS::Sun::draw() tws-mac
4.5ms	0.0%	TWS::Node::rotatez(double) tws-mac
2.7ms	0.0%	TWS::Shader::unbind() tws-mac
2.7ms	0.0%	TWS::Vector3<double>::lengthSquared() const tws-mac
2.3ms	0.0%	TWS::Vector3<double>::z() const tws-mac
2.2ms	0.0%	TWS::Vector3<double>::y() const tws-mac
1.9ms	0.0%	DYLD-STUB$$TWS::Vector3<double>::y() const tws-mac
1.3ms	0.0%	TWS::Vector3<double>::x() const tws-mac
0.0ms	0.0%	TWS::runSpamHandler(void*) tws-mac
0.0ms	0.0%	TWS::runHandler(void*) tws-mac

Figura 10.8: Total de tempo de CPU gasto em cada componente da aplicação.

10.4 Trabalho Futuro

Este projecto foi colocado num repositório público[1] para fácil acesso e contribuições da comunidade. Tendo os resultados ficado dentro do esperado e tendo sido atingido um patamar de desempenho desacoplado da dimensão espacial dos dados topográficos em uso, torna-se viável a adaptação do motor T.R.E. para utilização dos dados SRTM. Este desenvolvimento já foi iniciado e será adicionado ao repositório público do motor brevemente.

Durante a fase final de desenvolvimento deste trabalho foi publicado um novo conjunto de dados topográficos de cobertura global de ainda maior precisão que os do SRTM. Este novo conjunto de dados - ASTER GDEM[2] - resulta duma parceria entre o Ministério Economia e Indústria do Japão e a NASA e tem uma precisão constante de 30 metros por amostra. Existem, nesta altura, alguns constrangimentos no acesso

[1]Endereço: http://softwarelivre.sapo.pt/tws

[2]http://www.ersdac.or.jp/GDEM/E/index.html

aos dados que são públicos. No entanto, uma vez resolvidos, seria interessante adaptar o motor T.R.E. para os poder utilizar.

Para este último desenvolvimento seria necessário recriar o conjunto de programas auxiliares que tratam da fase de pré-processamento da informação e adaptar os módulos de leitura e desenho para os utilizar. Este trabalho será, a seu tempo, proposto à comunidade.

Neste momento, é também possível utilizar todos os programas já acabados para processar todos os dados topográficos de outros corpos planetários que estão disponíveis ao público. Já foi iniciado o trabalho de processamento da informação de topografia de Marte, da Lua, de Vénus e de Mercúrio. Será necessário fazer alguns ajustes e pequenas correcções devido a particularidades de cada um destes objectos no final. Também este trabalho será brevemente proposto à comunidade.

Referências

[1] MathWorld. Gnomonic projection, 2000. URL `http://mathworld.wolfram.com/GnomonicProjection.html`.

[2] MathWorld. Quaternions, 2000. URL `http://mathworld.wolfram.com/Quaternion.html`.

[3] Jean Meeus. *Astronomical Algorithms*. Willman-Bell, Inc., 2nd edition, 1999.

[4] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control. URL `http://tulrich.com/geekstuff/sig-notes.pdf`, 2002.

[5] Wikipedia. Quadtree space partitioning. URL `http://en.wikipedia.org/wiki/Quadtree`.

[6] Trent Polack. *Focus on 3D Terrain Programming*. Premier Press, 1st Edition, 2003.

[7] Michael Garland, Paul S. Heckbert. *Fast Polygonal Approximation of Terrains and Height Fields*. School of Computer Science, Carnegie Mellon University, 1995.

[8] Flip Strugar *Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD)*. 2010

[9] Yotam livny, Zvi Kogan, Jihad El-Sana *Seamless Patches for GPU-Based Terrain Rendering*. Dept Computer Science, University of Negev, Beer-Sheva, Israel, 2007.

[10] Frank Losasso, Hugues Hoppe *Geometry clipmaps: Terrain rendering using nested regular grids*. ACM Trans. on Graphics (SIGGRAPH), 23(3), 2004.

[11] Hugues Hoope *Smooth view-dependent level-of-detail control and its application to terrain rendering.* IEEE Visualization 1998 Conference, 35-42, 1998.

[12] B.D. Larsen, N.J. Christensen *Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail.* Journal of WSCG, vol. 11(2), pp. 282-9, 2003.

[13] Mark Duchaineau, LLNL, Murray Wolinsky, LANL, David E. Sigeti, LANL, Mark C. Miller, LLNL, Charles Aldrich, LANL, Mark B. Mineev-Weinstein, LANL *ROAMing Terrain: Real-time Optimally Adapting Meshes.* Los Alamos National Laboratory, Lawrence Livermore National Laboratory, 1997

[14] Thatcher Ulrich *Continuous LOD Terrain Meshing Using Adaptive Quadtrees.* 2000.

[15] Bryan Turner *Real-Time Dynamic Level of Detai Terrain Rendering with ROAM.* 2000.

[16] Willem H. de Boer *Fast Terrain Rendering Using Geometrical MipMapping.* 2000.

[17] Renato Pajarola *Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation.* ETH Zürich

[18] Steven C. Dollinds, Ph.D. *Modeling for the Plausible Emulation of Large Worlds.* Brown University, 2002