Universidade do Minho
Licenciatura em Engenharia Informática
2011

# Computação Gráfica
## Grupo 1
### Trabalho Prático

Ano Lectivo de 2010/2011

54745 **André Pimenta**
54808 **Cedric Pimenta**
54825 **Daniel Santos**
54738 **João Gomes**
54802 **Milton Nunes**

23 de Maio de 2011

**Resumo**

Neste relatório vão ser apresentados todos os passos e decisões na construção do trabalho prático de Computação Gráfica.
Este projecto consiste na apresentação de uma pequena aplicação em continuação do relatório apresentado na fase anterior, no qual se apresenta um jogo onde um agente se encontra num vale, no planeta XPTO. Neste vale, existem várias chaves dispersas que ele precisa de encontrar de encontrar de modo a abrir um tesouro que se encontra num edifício. Isto, sem que nenhuma das torres, que protegem este vale, o abatam.
Numa primeira parte será feita uma breve introdução ao projecto, seguindo-se a análise do desenvolvimento deste. Finalmente, é apresentada a conclusão e as perspectivas para a as fases futuras.

# Conteúdo

# Lista de Figuras

# Capítulo 1

# Introdução

## 1.1 Motivação e objectivos

No desenvolvimento deste projecto de Computação Gráfica pretendemos, para além de aplicar os conhecimentos leccionados na Unidade Curricular da cadeira, desenvolver sensibilidade para a criação de aplicações que envolvem elevado nível gráfico como é o caso neste desafio. Propomos-nos portanto a criar um jogo que seja atractivo e simples para quem o joga, apesar de ser a primeira vez que desenvolvemos algo deste género. Outro aliciante neste projecto é o facto de pela primeira vez desenvolvermos uma aplicação em C++, bem como para isso utilizarmos o OpenGL.

## 1.2 Contextualização

Pretende-se implementar um cenário constituído por um vale com cerca de $4km^2$. Nesse mesmo vale devem existir N chaves dispersas aleatoriamente e devem ser colocados, em extremos opostos, um agente (o jogador) e um edifício (inicialmente escondido) com um tesouro. O utilizador deverá conseguir conduzir este agente ao longo do vale, numa condução que pode ser feita em dois tipos de câmaras: First Person Shooter (FPS) e Third Person Shooter (TPS) que correspondem, respectivamente, à posição e orientação do agente, e a uma câmara exterior ao agente, sendo possível a comutação entre estas câmaras.

O objectivo do agente será encontrar as N chaves dispersas pelo vale com o objectivo de obter o tesouro guardado no edifício, o que permite terminar o jogo com sucesso. Para além disto, existem ainda M torres a proteger o vale; estas são capazes de detectar o agente se este estiver a menos de mil metros de distância e possuem uma capacidade de fogo capaz de disparar um tiro a cada 3 segundos.

Posto este problema que se encontra melhor descrito no enunciado do trabalho, deveremos implementar e tornar possível a criação deste mesmo cenário seguindo algumas exigências que serão apresentadas no capítulo seguinte.

# Capítulo 2

# Requisitos

Neste capítulo apresentamos os requisitos propostos para o desenvolvimento da aplicação Planeta XPTO, tantos os já realizados na primeira fase como os realizados na segunda fase ( fase actual) para que se possa perceber melhor as funcionalidades implementadas.

## 2.1 Requisitos da primeira fase (Fase Antiga)

- Implementação do terreno plano.
- Implementação da navegação do agente.
- Colocação das duas câmaras: FPS e TPS. Comutação entre estas câmaras com o teclado.
- Dispor as chaves aleatoriamente e implementar o detector de chaves.
- Carregar um modelo para o edifício do tesouro e outro para as torres de protecção.
- Orientar as torres segundo a posição do agente, desde que este esteja ao alcance da torre.

## 2.2 Requisitos da segunda fase (Fase Actual)

- Implementar a criação do terreno a partir de um mapa de alturas.
- Alterar a navegação do agente no terreno de forma a seguir a ondulação do terreno.
- Implementar a capacidade de fogo das torres, tendo em conta as especificações exigidas (tempo entre tiros e a velocidade destes).
- Detectar quando um projéctil disparado por uma torre atinge o agente.
- Implementação do processo *View Frustum Culling* de modo a optimizar a aplicação em relação ao edifício e às torres.

- Utilizar a extensão *Vertex Buffer Objects* (existente em OpenGL) no desenho do terreno.

# Capítulo 3

# Descrição do Trabalho e Análise de Resultados

Neste capítulo vamos descrever o desenvolvimento de cada um dos requisitos já apresentados na secção anterior, mas também algumas funcionalidades extra que decidimos criar e implementar de forma a tornar o jogo mas apelativo e agradável ao utilizador. Iremos, também, descrever os principais passos da sua implementação e estruturação e fazer uma breve analise dos resultados obtidos com estas mesmas implementações.

## 3.1 Terreno

A implementação do terreno foi um dos requisitos que mais alterações sofreu relativamente á implementação da primeira fase, muito se deve há existência de novos requisitos a ter em conta, nomeadamente a implementação do terreno partir de um mapa de alturas.
Passamos então a ter a necessidade de criar uma matriz com a leitura das alturas que vão derivar da imagem representante do mapa de alturas.
Para resolver este mesmo problema, as alturas serão guardadas numa matriz dinâmica que aloca memoria conforme as necessidades do mapa de alturas, procurando assim ser uma solução eficaz ao nível de memoria a utilizadar. De referenciar que decidimos criar um limite máximo de altura possível para o mapa, sendo que sempre que a leitura ultrapassar este limite será apenas considerada a altura máxima definida.
Após a criação do mapa de alturas e tendo em conta que este definirá as variáveis globais largura e comprimento com o respectivo comprimento e largura da imagem do mapa de alturas fazemos ainda uma ”suavização”do mapa de alturas, usando a função *smoothing*,função esta que ”suaviza”as alturas do terreno usando as alturas adjacentes em cada ponto e um factor de suavidade. Esta recebe o array das alturas,a altura e largura do array(matriz que este representa) e o numero de vezes que vai ser aplicado este método.

Já temos todas as necessidades relativas a alturas resolvidas, no entanto ainda falta tratar da textura. Tal como já apresentamos na etapa anterior as texturas nomeadamente a do terreno, mas também o mapa de alturas são carregadas recorrendo á biblioteca

**DevIL**.
A textura carregada será posteriormente aplicada na construção do terreno.
Após termos disponíveis um mapa de alturas e a textura a aplicar é possível criar o terreno.
Na criação do terreno carregamos a textura e aplicamos-a neste. Para cada unidade do mapa delimitado pelas variáveis comprimento e largura , usando a função de desenho e o tipo GL_TRIANGLE_STRIP e aplicando o calculo das normais para cada vertex construimos o terreno do jogo. Após a construção do terreno para cumprir o requisito de que este possua 4000m$^2$ aplicamos uma escala que defina as unidades de forma a ser considerado um terreno de 4000m$^2$. Todas as funções relativas á construção do terreno apresentam-se no ficheiro "terreni.cpp".

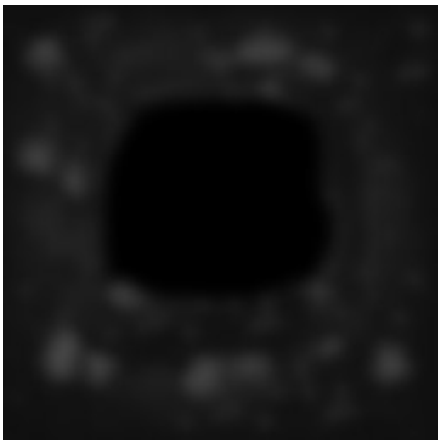

**Figura 3.1:** Mapa de alturas

**Figura 3.2:** Textura do terreno

## 3.2 Navegação

A navegação do agente efectua-se recorrendo às teclas 'w', 's' (respectivamente, movimentação para a frente e para trás) e 'a', 'd' (movimentação lateral para a esquerda e direita, respectivamente), enquanto o rato serve para rodar o agente e controlar o ângulo da câmara.
Todas estas funcionalidades são implementadas nas funções exclusivas pelo OpenGl para a utilização do rato e do teclado, e são utilizadas através dos registos de callback efectuados na função *main*.
Para que ocorra esta movimentação, temos variáveis globais que são activadas consoante a tecla pressionada e quando a tecla é solta, a mesma variável volta a ser desactivada. Enquanto essa variável global está activa, as variáveis que determinam a posição da câmara, o local para onde esta aponta e a posição do agente sofrem uma alteração no valor conforme nós pretendemos. No caso específico da movimentação para a frente e para trás, apenas a posição da câmara e do agente são alterados enquanto no caso das movimentações laterais, o local para onde a câmara aponta, também é alterado.
As funções de implementação do teclado e do rato encontram-se no ficheiro input.cpp que segue em anexo neste relatório.

## 3.3 Colocação das câmaras

A implementação das câmaras em diferentes perspectivas (FPS e TPS), é realizada com auxílio da função 'gluLookAt' que nos permite especificar a posição da câmara e a direcção para onde ela está a apontar.
Como parâmetros desta função usamos as variáveis:

- Posição da câmara em cada eixo: camX , camY, camZ.
- Posição onde a câmara aponta: eyeX, eyeY, eyeZ;

Para a implementação da câmara em modo Third Person Shooter foram necessárias mais três variáveis que indicam a posição do jogador (playerX, playerZ, playerY).
As coordenadas da posição da câmara e da posição para onde ela aponta, permitem calcular o vector director (dirX, dirY, dirZ), que vai ser utilizado para definir a direcção para onde o jogador deve estar a olhar. Este calculo é realizado na função 'updateDir'.
Quanto às rotações nos diferentes eixos, são usadas duas variáveis, 'alpha' e 'beta', que nos permitem definir o ângulo de rotação nos eixos dos 'yy' e 'xx' respectivamente. Assim é possível ao jogador movimentar a câmara em cada um desses eixos sem ter de se mover, podendo observar tudo em seu redor. Para tornar o jogo mais realista decidimos limitar o ângulo de visão da câmara segundo o eixo dos 'xx', pois uma pessoa também não consegue rodar a cabeça 360º.

A câmara TPS (Third Person Shooter) foi colocada ligeiramente recuada e elevada em relação ao agente, tal como é norma utilizar-se na maior parte dos jogos e de forma a facilitar a utilização do agente. Para além disto, por regra, a câmara fica apontada

para a posição do jogador.
A câmara FPS (First Person Shooter) foi colocada na posição do agente ligeiramente mais avançada para permitir que o utilizador tenha uma visão tal como se se encontrasse na posição do agente. A escolha das câmaras faz-se através da tecla 'v', alternando assim entre as duas câmaras possíveis. Quando se altera de FPS para TPS, as coordenadas do agente são igualadas às coordenadas da câmara e esta sofre a tal alteração já mencionada; quando se trata da alternativa inversa, são as coordenadas da câmara a serem igualadas às coordenadas do agente.
Para uma melhor jogabilidade foram usados mecanismos que permitem que o rato se encontre sempre no centro da janela. Assim não existe o problema do cursor se encontrar nas bordas do ecrã e o jogador não conseguir mover o câmara em todos os sentidos. Foi também usada a função 'glutTimerFunc' para reduzir o delay do rato e do teclado para a movimentação parecer mais fluída.

Todo o código relativo a esta secção encontra-se no ficheiro input.cpp que se está em anexo.

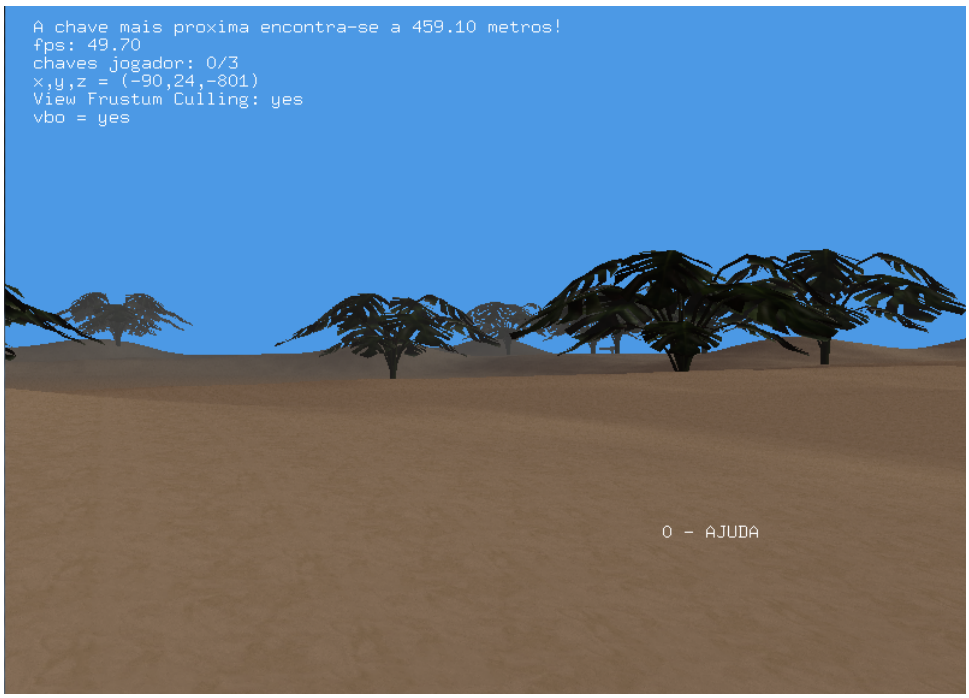


**Figura 3.3:** Camarâ FPS


**Figura 3.4:** Camarâ TPS

## 3.4 Chaves e Detector de Chaves

Existem N chaves que são dispostas aleatoriamente no terreno. O número de N chaves é configurável, pois encontra-se guardado numa variável global, bastando alterar esta para o número que desejarmos. Além disto, não permitimos que nenhuma chave seja criada para além das bordas do terreno.
Obtemos isto, gerando valores aleatórios para a dimensão do vale e geramos um número entre 0 e 180. Ao multiplicar o primeiro pelo co-seno (para a coordenada X) ou pelo sin (para a coordenada Z) do segundo, garantimos que as coordenadas estarão limitadas entre 0 e o valor que desejámos.
O agente navega ao longo do terreno e dispõe de um detector de metais com uma capacidade de detecção até 500 metros para localizar as chaves. Este detector mostra a informação no ecrã, indicando se detecta alguma chave (caso o agente esteja a menos de 500 metros de alguma); caso sejam detectadas duas chaves, o detector mostrará a que distância se encontra a chave mais próxima.
Para isto, guardamos as coordenadas de cada chave quando estas são inicializadas e como temos as coordenadas do agente, calculamos a norma entre estes dois pontos. Esse valor é comparado com o alcance que definimos para o detector e se for menor, guardámos essa distância. Ao percorrer todos as chaves, iremos guardar o menor valor e imprimi-lo para o ecrã.
Mais uma vez, o código relativo a estas funcionalidades encontra-se no ficheiro scene.cpp em anexo.

## 3.5 Carregar objectos

Para importar objectos para a aplicação foram encontrados vários modelos para utilização livre. Utilizámos, então, ficheiros do tipo **.obj** que foram carregados com a biblioteca *glm* fornecida nas aulas práticas, mas também decidimos começar a usar modelos **MD2** nomeadamente para o agente e as torres.

## 3.6 Disposição dos Objectos

Tal como já referimos anteriormente, as chaves são dispostas aleatoriamente ao longo do terreno (procedemos do mesmo modo para a colocação das torres de protecção e das árvores que se encontram espalhadas pelo terreno). No entanto, nem tudo é aleatório; o posição inicial do agente será sempre no início do vale enquanto o edifício (e o tesouro) se encontram no fim do vale, ou seja, no extremo oposto à posição inicial do agente.

## 3.7 Orientação das torres em direcção ao agente e disparo de Balas

Como nos é dito no enunciado, as torres que protegem o vale devem conseguir orientar a sua arma para o agente caso este se encontre num raio de 1000 metros em relação a estas. Assim sendo, antes de construir cada torre, calculámos a distância desta em relação ao agente e se este se encontrar a mais de mil metros, a torre é desenhada com uma orientação aleatória. Porém, se ele se encontrar a menos de 1000 metros, então, obtemos a posição dele e da torre, calculamos o ângulo entre eles e, quando a torre é desenhada, sofre uma rotação, do ângulo calculado, no eixo dos YY.
Depois de se ter adequado a rotação das torres em relação à posição do agente, na segunda fase do projecto desenvolvemos os restantes requisitos pretendidos para as torres, nomeadamente em relação aos disparos. Desta forma, implementamos os disparos nas torres cumprindo tudo o que era indicado no enunciado. Ou seja, com a ajuda do factorMetro, valor calculado que ajusta a medida em metros à escala do jogo, melhoramos a eficiência do radar de 1000 metros, e colocamos as torres a disparar em direcção à posição do jogador, sendo que cada torre só pode voltar a disparar 3 segundos após o disparo anterior efectuado pela mesma.
Para além disso tivemos em atenção o facto de as balas desaparecerem caso saiam do perímetro do jogo, bem como no caso de baterem contra uma árvore ou no castelo desaparecerem. As nossas balas tal como se pretendia no enunciado deslocam-se à velocidade de 30 km/hora e à distância de um metro do chão. Por fim, caso a posição da bala coincida com a posição do jogador, ou seja, caso a bala atinja o jogador, o utilizador perde o jogo Em anexo, no ficheiro scene.cpp está todo o código relativo a este requisito.

## 3.8 Vertex Buffer Objects

Tal como já tinha sido referenciado no primeiro relatório e no enunciado, um dos objectivos para esta etapa do projecto seria a utilização de Vertex Buffer Objects no desenho do terreno. Este método é uma das várias optimizações implementadas nesta segunda fase do projecto. Desenhando o terreno com este método, pretendemos poupar recursos na utilização da memória pois as coordenadas tanto de textura como de normais como dos vértices são armazenadas na placa gráfica, evitando o envio destas sempre que é executado o *renderscene()*.
De modo a utilizar Vertex Buffer Objects, extensão do OpenGL que providencia métodos para efectuar o *upload* de informação, criámos três arrays com o tamanho suficiente para guardar todas as coordenadas anteriormente mencionadas. Após se ter guardado todos os valores, criámos três buffers e passámos a informação contida nos arrays para estes. Com estas operações, os dados necessários para o desenho do terreno estão reunidos e a função *GenerateVBO* é executada aquando da chamada da função main para, tal como foi dito, guardar a informação na placa gráfica. Após isto, utilizámos a função *glBindBuffer* que nos permitirá usar os buffers anteriormente definidos com a função *glDrawElements*; por fim, para a utilização desta função, criámos um array em que guardámos as posições dos vértices pela ordem que estes devem ser utilizados no desenho da primitiva *GL_TRIANGLE_STRIP* e passamos tanto o array como a primitiva como parâmetro. Com isto, tal como esperado, obtemos um melhor desempenho na criação do terreno.

## 3.9 View Frustum Culling

A implementação do View Frustum Culling permite que não sejam desenhados objectos que não estão no campo de visão da câmara.
Para definir o VFC precisamos de definir os planos que delimitam essa região. Um plano pode ser definido por três pontos. Com o vector ”cam”que define a posição da câmara; o vector director ”look”para onde a camara aponta; o vector ”up”que define o eixo que aponta para cima; o vector ”direita”que aponta para o lado direito em relação ao ponto para onde a câmara está virada, e que servirá para obter os planos laterais; a distancia, altura e largura dos planos mais próximo e mais distante conseguimos definir pontos suficientes para definir todos os seis planos que constituem o VFC. Com estes pontos e vectores conseguimos ainda descobrir as normais que apontam, em cada plano, para o interior da área de visualização do VFC.
Para verificar se o edifício e as torres devem ser desenhadas é necessário verificar se estas se encontram dentro da área do VFC. Para isso definimos axis aligned bounding boxes para ambos os tipos de objectos, pois são a opção que melhor se adapta aos nossos modelos. Para cada modelo a AABB foi adaptada ao tamanho de cada objecto para que a aproximação fosse a melhor e a aproximação a melhor possível. O teste de localização das AABB em relação à área de visualização do VFC é feito através de dois vértices, o mais distante e o mais próximo e ainda do vectores que apontam para o interior da área de visualização do VFC. Se um desses pontos estiver dentro da área de visualização isso é suficiente para desenhar a o modelo referente a essa AABB.

## 3.10 Optimizações utilizadas

Para o melhoramento de desempenho da aplicação, para que esta esteja acessível a diversas maquinas com diferentes características usamos as seguintes optimizações:

- Vertex Buffer Objects
- View Frustum Culling
- Display List

## 3.11 Funcionalidades extra

Após as realização de todas as tarefas decidimos criar algumas funcionalidades extra de forma a tornar o jogo mas agradável ao utilizador, mas também muito mais apelativo. Posto isto nesta secção apresentamos algumas funcionalidades que achamos se serão uma mais valia para o jogo.

### 3.11.1 Estados do jogo

Para que fosse possível existir um inicio, meio e fim do jogo criamos duas variáveis, variável dead e end, que vão permitir e identificar o estado do jogo e do jogador. A variável *dead* será inicializada a 0 e assim se manterá até que o jogador seja atingido por uma bala. Caso este seja atingido por uma bala então a variável dead passará para o valor 1, e então o jogo terminará por *Game Over*, tendo depois o jogador a possibilidade de voltar a jogar ou sair do jogo.
Por outro lado a variável *end* que também será inicializada com o valor 0 representa quando o jogo foi terminado por sucesso do jogador, ou seja quando o jogador conseguir cumprir todos os objectivos do jogo (já expostos neste relatório) então o jogo terminará com uma mensagem de sucesso e permitindo uma vez mais ao utilizador voltar a jogar ou simplesmente terminar o jogo.

### 3.11.2 Menus

Para facilitar ao utilizador a percepção do jogo decidimos criar dois menus. O primeiro deles surge no final do jogo, isto é, no caso do utilizador chegar com sucesso ao castelo ou no caso de ser atingido por uma bala o jogo termina e surge de seguida no ecrã um menu que indica a vitória ou derrota do jogador e dá as opções de sair ou então voltar a jogar, iniciando-se assim um novo jogo. Para além disso criamos ainda o menu de ajuda. No caso do utilizador carregar na tecla '0' surge no ecrã várias informações que ajudam o utilizador a jogar.

### 3.11.3 Modelos MD2

Nesta fase optamos por substituir alguns dos nossos objectos por modelos MD2. Este modelos permitiram dar uma nova vida ao nosso projecto devido as animações que nos

permitiram fazer, tornando a jogabilidade do projecto mais agradável, e ainda conseguimos que este se tornasse mais leve.
Tal como os objectos os modelos md2 são carregados apartir da pasta DATA/MD2, que contem os ficheiros .md2 (ficheiro do vértices para o desenho) e os ficheiros .pcx (ficheiros que contem a textura do nosso "objecto"). Quando o jogo é iniciado, estes ficheiros são carregados na função main e posteriormente usados aquando do seu desenho.

### 3.11.4 Velocidades

Uma das especificações exigidas neste trabalho está relacionado com as velocidades a que se movem os tiros disparados pelas torres e o jogador. Assim, calculamos as velocistas respectivas para metros por segundo e esses são os valores iniciais atribuídos a duas variáveis globais. De modo a garantir as devidas velocidades, aquando do cálculo da posição do jogador e dos tiros, multiplicámos o valor obtido pela respectiva variável. De modo a melhorar a jogabilidade, tomámos a iniciativa de permitir ao utilizador alterar a velocidade tanto do jogador como dos tiros, existindo limites para estas alterações. Com isto, pretendemos que o jogo se possa tornar mais fácil ou difícil consoante a preferência de quem joga; para a alteração da velocidade do jogador, basta clicar na tecla '+' ou '-' para aumentar ou diminuir a velocidade, respectivamente. Quanto a aumentar ou diminuir a velocidade dos tiros, clica-se nas teclas 'o' ou 'l', respectivamente.

### 3.11.5 Factor Metro

Para que o nosso jogo mantenha a coerência caso as dimensões do terreno sejam redimensionadas, quando ele é executado , é calculado o factorMetro (trata-se de uma variável que representa a escala do terreno, comprimento do mapa de alturas pela dimensão desejada). Aplicando este factor às distâncias calculadas, como por exemplo, o detector de chaves ou o alcance de detecção das torres, garantimos que se o terreno for mais pequeno que o desejado, as distâncias são adaptadas. Para que se possa perceber melhor o que isto significa, apresentámos o seguinte cenário: o detector de chaves é 200, e o terreno tem de dimensões 100 por 100; sem a aplicação deste factor, onde quer que o jogador se encontrasse, a chave seria detectada mas com a aplicação deste, a distância diminui na devida proporção.

## 3.12 Arvores

Para melhorar a qualidade da aplicação decidimos implementar árvores ao longo do terreno. Estas são espalhadas ao longo do terreno de forma aleatória tal como as torres e as chaves. O numero de árvores tal como outras variáveis neste projecto é configurável.
Para tornar o jogo mais real implementamos colisões com as árvores, não permitindo o agente ultrapassar estas.

# Capítulo 4

# Conclusão

Fazendo, agora, uma análise crítica ao trabalho por nós desenvolvido, pensámos ter cumprido os requisitos exigidos para a aplicação, mas também alguns requisitos por nós propostos de forma a melhorar o jogo.
Relativamente aos objectivos exigidos pelo enunciado do trabalho, estes foram todos realizados com sucesso, embora alguns tenham nos causado maiores dificuldades, nomeadamente os objectivos da fase II. Porem e após a realização destes decidimos criar novos objectivos, alguns já referenciados como desafios futuros na etapa I, tal como o caso de usar modelos **MD2**, de forma a valorizar o trabalho, mas também melhorar alguns objectivos apresentados na etapa anterior.
Nesta fase começamos por responder a todos os objectivos propostos para a mesma, sendo que alguns nomeadamente a implementação do *View Frustum Culling* e de *Vertex Buffer Objects* causaram nos algumas dificuldades no entanto conseguimos resolver estas com sucesso e proporciono-nos uma maior aprendizagem sobre estes devido a essas mesmas dificuldades.
Tal como já referimos ao longo do relatório decidimos criar alguns objectivos extra com o intuito de dar mor jogabilidade e maior ênfase ao jogo. Estes objectivos estão referenciados no capitulo anterior na secção de funcionalidades extra. Destas mesmas funcionalidades extra destacamos o uso de objectos **MD2** que nos propusemos a usar nesta fase quando terminamos a anterior.
Podemos então afirmar que o desenvolvimento deste trabalho foi concluído, cumprindo os objectivos propostos e com a introdução de novas funcionalidades, no entanto poderiam ter sido aplicadas mais optimizações no projecto nomeadamente a utilização de optimizações *"level of detail"* entre outros que ficam referenciados para possíveis utilizações futuras.

# Bibliografia

## 4.1 Referências Bibliográficas

Astel Dave,Hawkins Kevin
"Beginning OpenGL Game Programming",
Course Technology PTR; 1 edition (March 19, 2004).

## 4.2 Referências WWW

"http://www.lighthouse3d.com/",
10 de Abril de 2011.

"www.opengl.org/documentation/",
09 de Abril de 2011.

# Capítulo 5

# Anexos

## 5.1 Elementos do Grupo

**Figura 5.1:** André Pimenta

**Figura 5.2:** Cedric Pimenta

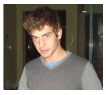

**Figura 5.3:** Daniel Santos

**Figura 5.4:** João Miguel

**Figura 5.5:** Milton Nunes