



Escola de Engenharia

Departamento de Informática

**Universidade do Minho**

Licenciatura em Engenharia Informática

## **Projecto de Computação Gráfica**

**“Bar”**

Bruno Ferreira, A61055

Serafim Pinto, A61056

*Grupo 19*

Braga, Maio de 2013

# Conteúdo

<b>1 Resumo</b>	<b>7</b>
<b>2 Introdução</b>	<b>8</b>
<b>3 Descrição do Trabalho e Análise de Resultados</b>	<b>9</b>
3.1 Requisitos . . . . .	10
3.1.1 Primeira Fase . . . . .	10
3.1.2 Segunda Fase . . . . .	11
3.1.3 Terceira Fase . . . . .	11
3.1.4 Quarta Fase . . . . .	12
3.2 Implementação . . . . .	13
3.2.1 Primeira Fase . . . . .	13
3.2.2 Segunda Fase . . . . .	18
3.2.3 Terceira Fase . . . . .	28
3.2.4 Quarta Fase . . . . .	30
<b>4 Navegação e Controlos</b>	<b>41</b>
<b>5 Optimização e Técnicas usadas</b>	<b>42</b>
5.1 Evitar erros de vírgula flutuante . . . . .	42

5.2	”Camadas” nas primitivas . . . . .	43
5.3	Coordenadas Polares e Esféricas . . . . .	43
5.4	Gestão de input . . . . .	45
5.5	Câmara FPS . . . . .	45
5.6	Sólidos de Revolução . . . . .	46
5.7	Evitar mudanças de estado . . . . .	47
5.8	Vertex Buffer Objects . . . . .	47
5.9	Backface Culling . . . . .	48
5.10	View Frustum Culling . . . . .	48
5.11	Bounding Volumes Hierárquicos . . . . .	50
5.12	Encadeamento de métodos . . . . .	51
<b>6</b>	<b>Organização e Descrição do Projecto</b>	<b>55</b>
6.1	Descrição Resumida . . . . .	55
6.2	Descrição Detalhada . . . . .	58
6.2.1	Classe Vec3 . . . . .	58
6.2.2	Classe Light . . . . .	62
6.2.3	Classe Input . . . . .	66
6.2.4	Classe Camera . . . . .	68
6.2.5	Classe CG_OBJ . . . . .	72
6.2.6	Classe Plano . . . . .	77
6.2.7	Classe Cubo . . . . .	79

6.2.8 Classe Cilindro . . . . .	81
6.2.9 Classe Esfera . . . . .	83
6.2.10 Classe SolidoRevolucao . . . . .	84
6.2.11 Classe Plano3D . . . . .	86
6.2.12 Classe Frustum . . . . .	88
6.2.13 Classe ObjectTree . . . . .	90
6.2.14 Classe Profiler . . . . .	95
6.2.15 Classe Textura . . . . .	98
6.2.16 Módulo Utilities . . . . .	101
6.2.17 Ficheiro main.c . . . . .	102
<b>7 Conclusão</b>	<b>103</b>
<b>8 Referências</b>	<b>105</b>
8.1 Referências bibliográficas . . . . .	105
8.2 Referências WWW . . . . .	105
<b>9 Membros do Grupo</b>	<b>106</b>

## Listas de Figuras

1	Plano com 5 de comprimento, 4 de largura e 3 camadas . . . . .	14
2	Esfera com 2 de raio, 20 camdas horizontais e verticais . . . . .	17
3	As seis paredes visíveis em modo linha . . . . .	19
4	Cadeira normal e banco de balcão . . . . .	22
5	Copo simples . . . . .	23
6	Copo de vinho . . . . .	24
7	Copos de champanhe . . . . .	25
8	Garrafa . . . . .	26
9	Candeeiro de tecto e de pé . . . . .	27
10	Composição do Bar no fim da fase 3 . . . . .	29
11	Parede exterior . . . . .	31
12	Perspectiva geral sobre o bar . . . . .	32
13	Balcão, copos, garrafas e bancos . . . . .	33
14	Pormenores do balcão e porta . . . . .	34
15	Mesa e cadeiras . . . . .	35
16	Mesas de bilhar com bolas . . . . .	36
17	Pormenor das pernas da mesa de bilhar . . . . .	37
18	Porta, decoração da parede, tecto e candeeiros . . . . .	38
19	Os copos e garrafas . . . . .	39
20	Perspectiva da outra mesa de bilhar . . . . .	40

21	Coordenadas Polares . . . . .	43
22	Coordenadas Esféricas . . . . .	44
23	Demonstração de Sólido de Revolução . . . . .	46
24	Demonstração de Bounding Volumes Hierárquicos . . . . .	50
25	Relações de inclusão entre as várias classes . . . . .	56

## 1 Resumo

Neste relatório apresentamos todos os passos e decisões tomadas na construção do trabalho prático de Computação Gráfica.

Este projecto consiste na representação de um Bar, enquanto contrução geométrica. O Bar poderá conter elementos como mesas, cadeiras, copos, entre outros.

Numa primeira parte deste relatório fazemos uma breve introdução ao projecto, seguindo-se a análise do seu desenvolvimento ao longo das várias fases. Depois apresentamos uma descrição detalhada das técnicas usadas e, por fim, temos a conclusão.

## 2 Introdução

No desenvolvimento do presente projecto de Computação Gráfica pretendemos, para além de aplicar os conhecimentos leccionados na Unidade Curricular, desenvolver sensibilidade para a criação de aplicações de elevada complexidade gráfica. Propomos-nos portanto a criar um espaço com elementos comumente observados num Bar.

Este trabalho está dividido em quatro fases distintas. Na primeira fase apenas são feitas as primitivas: plano, cubo, cilindro, esfera. Numa segunda fase, devem ser apresentados alguns itens que deverão estar no espaço final, tais como: mesas, cadeiras, copos e candeeiros. A terceira fase consiste na preparação das primitivas das fases anteriores para a inclusão de texturas e iluminação. Nessa fase, devemos também optimizar o projecto utilizando *Vertex Buffer Objects*. Na quarta e última fase deve ser montado o Bar propriamente dito, fazendo uso de tudo o que foi criado até ao momento. Além disto, a visualização do modelo deve ser feita recorrendo a uma câmara de movimento livre, com o uso do teclado e rato. Posto este problema, que está descrito no enunciado do trabalho, deveremos implementar e tornar possível a criação deste cenário seguindo os requisitos que serão apresentados no capítulo seguinte.

O projecto será desenvolvido em C++ no software Visual Studio 2012. Utilizaremos também a biblioteca gráfica do OpenGL, os utilitários GL (GLU), o GL Extension Wrangler (GLEW) e a extensão Devil.

Para criação de texturas utilizaremos o software Genetica Viewer 3.5.

### 3 Descrição do Trabalho e Análise de Resultados

Neste capítulo vamos descrever o desenvolvimento de cada um dos requisitos a ser apresentados. Descreveremos também os principais passos da sua implementação e estruturação e faremos uma breve análise dos resultados obtidos com estas mesmas implementações.

### 3.1 Requisitos

Durante esta secção apresentamos em detalhe os requisitos propostos em cada fase para o desenvolvimento do nosso projecto.

#### 3.1.1 Primeira Fase

- Construir uma biblioteca de primitivas: plano, cubo, cilindro e esfera;
- Cada primitiva deve ser feita numa função que desenhe a primitiva centrada na origem;
- A função para cada primitiva deve ter um conjunto de parâmetros que permita desenhar de acordo com a dimensão e resolução pretendidas;
- Construção de uma aplicação OpenGL que permita a visualização de cada primitiva separadamente.

### 3.1.2 Segunda Fase

- Apresentar rotinas que desenhem os seguintes objectos: mesas, cadeiras, copos e candeeiros;
- Desenhar o espaço limite do bar, ou seja, paredes chão e tecto;
- À exceção do copo, todos os outros objectos devem ser desenhados à custa da biblioteca de primitivas da primeira fase, e de transformações geométricas. O copo deve ter uma rotina própria para a sua construção;

### 3.1.3 Terceira Fase

- Definição das coordenadas de textura e definição das normais nas primitivas;
- Utilização de VBOs;
- Construção de uma aplicação que permita a visualização de cada composição geométrica.

### 3.1.4 Quarta Fase

- Aplicação com a apresentação da composição geométrica final, ou seja, um bar com mesas, candeeiros, cadeiras e copos (construídos apenas com recurso às primitivas e transformações geométricas desenvolvidas nas fases anteriores);
- Utilização de texturas e iluminação.
- Visualização da cena utilizando uma câmara de movimento livre, sem necessidade de detecção de colisões.

## 3.2 Implementação

Nesta seccção descrevemos de forma incremental o desenvolvimento do projecto ao longo das 4 fases.

### 3.2.1 Primeira Fase

A primeira fase pedia que fosse construída uma biblioteca com primitivas geométricas, aqui explicamos como nesta fase do projecto construímos cada uma delas. Estas primitivas foram numa classe à qual chamámos **Primitivas**. Para facilitar a visualização destas, elaborámos também uma *main.cpp*, onde é possível através do teclado mover a câmara e visualizar o objecto de ângulos diferentes.

Criámos também o módulo **Utilities**, onde temos pequenas funções que nos facilitam na escrita do código tornado-o mais fácil de ler (p.e função que passa de graus para radianos). Além disto, temos um menu associado ao botão direito do rato onde o utilizador pode escolher qual a primitiva a visualizar, e visualizar o objecto de várias formas (figura colorida, apenas linhas ou apenas pontos).

## Plano

A função que desenha o plano recebe como argumentos, o comprimento, a largura e ainda o número de camadas que o plano terá. Como queremos o nosso plano com um determinado número de camadas, dividimos as variáveis comprimento e largura pelas camadas pretendidas, e guardamos o resultado em  $x2$  e  $z2$  respectivamente (sobre camadas, ver secção [5.2](#)).

Como também queremos desenhar centrado na origem, dividimos o comprimento e a largura por dois, e guardamos o resultado nas próprias variáveis. E é assim, com base nessas variáveis que desenharmos os triângulos que compõem o nosso plano. A Figura 1 permite visualizar o nosso plano nesta fase:



Figura 1: Plano com 5 de comprimento, 4 de largura e 3 camadas

## Cubo

A função que desenha o cubo recebe como argumento o comprimento do lado e o número de camadas no eixo dos  $X$  e dos  $Z$ . Tal como no plano, para centrar o cubo na origem, a variável lado é dividida por dois. Para desenhar esta primitiva, como seria de esperar desenhamos seis faces, e cada uma delas desenhada da mesma maneira que é desenhado um plano.

Optámos por criar um cubo e não um paralelipipedo, pois um paralelipipedo pode ser obtido escalando um cubo.

Visto ser ineficiente, não recorremos à função que cria o plano para construir o cubo (ver secção [5.7](#))

## Cilindro

Este primitiva foi desenhada de um método muito semelhante ao usado no execício da aula prática. A função recebe como argumentos o raio, a altura, o número de camadas verticais e o número de camadas horizontais. Existe uma variável *delta* que resulta da operação

```
float delta = 2 * M_PI / fatias;
```

Ficando guardada nesta variável o ângulo necessário para obter as coordenadas polares. Mais uma vez como a primitiva deve ser desenhada na origem, dividimos a altura por dois. O Cilindro é assim desenhado com recurso a coordenadas polares

(ver secção 5.3). Posteriormente o cálculo deste ângulo foi modificado para passar a ser calculado a cada iteração para evitar erros de vírgula flutuante (ver secção 5.1).

## Esfera

A função que desenha esta primitiva recebe três argumentos: o raio da esfera, o número de camadas verticais e o número de camadas horizontais. É através de dois ângulos e funções trigonométricas que conseguimos desenhar a primitiva. Estas variáveis são:

```
float alpha = 2 * M_PI / fatias;  
float beta = M_PI / seccoes;
```

Estamos agora a utilizar coordenadas esféricas (ver secção 5.3). Existem três ciclos, um que desenha a parte de baixo, outro que desenha a parte de cima e por fim um que desenha as secções intermédias. Posteriormente o cálculo destes ângulos foi modificado para passar a ser calculado a cada iteração para evitar erros de vírgula flutuante (ver secção 5.1).

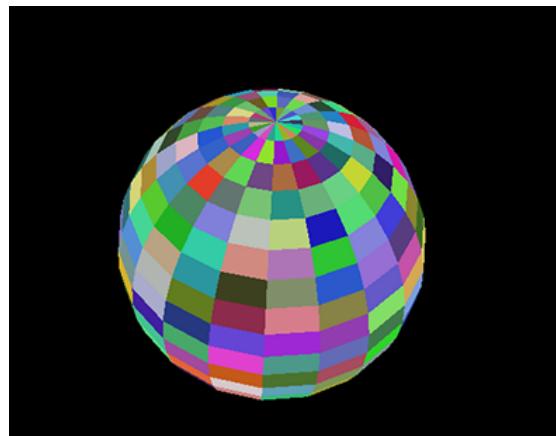


Figura 2: Esfera com 2 de raio, 20 camdas horizontais e verticais

### 3.2.2 Segunda Fase

Em relação à primeira etapa, reorganizámos o nosso trabalho com o objectivo de ter uma melhor organização e escalabilidade. Para isso foram criadas algumas novas classes.

Criámos uma classe **CG\_OBJ** que define um *Vertex Buffer Object* (sem índices). De forma simplificada, um *VBO* é buffer onde se guarda um conjunto de vértices correspondente a uma primitiva (ver secção 5.8).

Para compor vários *VBO* de modo a formar um objecto complexo foi criada a classe **Figuras**. Esta classe *static* apenas tem um *array* de **CG\_OBJ** e métodos que compõem elementos desse array para formar figuras complexas. Estes métodos são utilizados pelo *renderScene()* para desenhar objectos construídos por composição de primitivas.

Foram também criadas sub-classes da classe **CG\_OBJ** correspondentes às várias primitivas: **Plano**, **Cubo**, **Cilindro** e **Esfera**. De notar que a forma de desenhar as primitivas mudou e, como tal, a nova forma de desenhar primitivas difere consideravelmente da forma que constava na classe **Primitivas**. Além das quatro primitivas, foi adicionada a classe **SolidoRevolucao** que foi usada para desenhar copos (ver secção 5.6).

Também implementámos uma câmara FPS (ver secção 5.5) e alterámos a interacção através do teclado com a aplicação (ver secção 5.4). De forma a aumentar a escalabilidade do projecto, criámos duas novas classes para o controlo da câmara e teclado. São respectivamente a classe **Camera** e a classe **Input**. Ao longo das fases seguintes, ambas as classes **Camera** e **Input** foram sofrendo alterações conforme foi necessário.

## Espaço limite do Bar

Para simular o espaço do Bar, fizemos com que toda a cena fosse desenhada dentro de um cubo com as seis paredes visíveis.

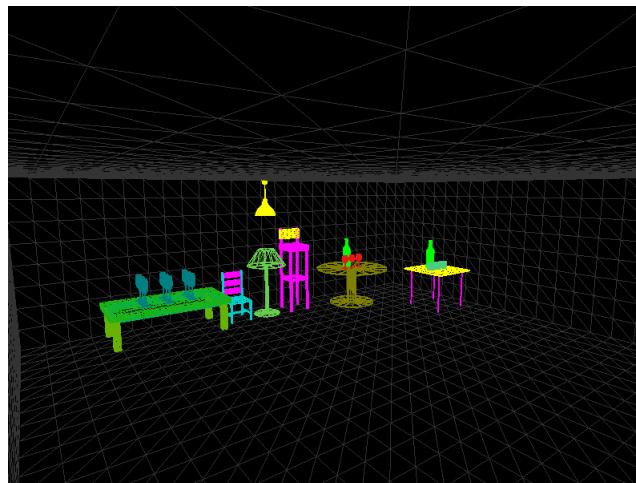


Figura 3: As seis paredes visíveis em modo linha

## Objectos

Para além do espaço limite do bar, foi também pedido que fossem construídos alguns elementos que se podem encontrar num bar. Estes elementos são desenhados recorrendo aos métodos da classe **Figuras**.

Para esta fase desenhámos três **mesas**, sendo que uma delas tem a base e o tampo redondo, outra mesa **quadrada** e por fim uma mesa **rectangular** baixa.

A mesa **quadrada** é a mais simples das três, contendo quatro cilindros que constituem as pernas da mesa e um rectângulo para a superfície.

Temos uma mesa rectangular mais baixa que a quadrada. A mesa é formada por 9 cubos: 4 para as pernas, 4 para o reforço e 1 para a superfície da mesa.

Por fim temos a mesa **redonda**, que é composta por uma base de suporte inferior e pela superfície superior. A unir estes itens está um cilindro de raio inferior à base e ao tampo. Esta mesa é desenhada com recurso a Sólidos de Revolução (ver secção [5.6](#)).

Outro elemento do Bar pedido eram as **cadeiras**, além de cadeiras optámos por também desenhar um **banco** alto. As pernas, assento e recosto da cadeira são cubos. O banco tem pernas cilíndricas e ambos o recosto e o assento são cubos.

Apresentamos agora os **copos** que criámos usando Sólidos de Revolução (ver secção [5.6](#) para detalhes sobre Sólidos de Revolução). Recorrendo à mesma técnica desenhámos também uma garrafa.

Finalmente o último item pedido eram candeeiros, aqui optámos por fazer dois estilos diferentes, um de tecto e outro de pé. Tanto o candeeiro de tecto e o *abajour* do candeeiro de pé são Sólidos de Revolução. O candeeiro de pé também é formado por uma composição de vários cilindros.

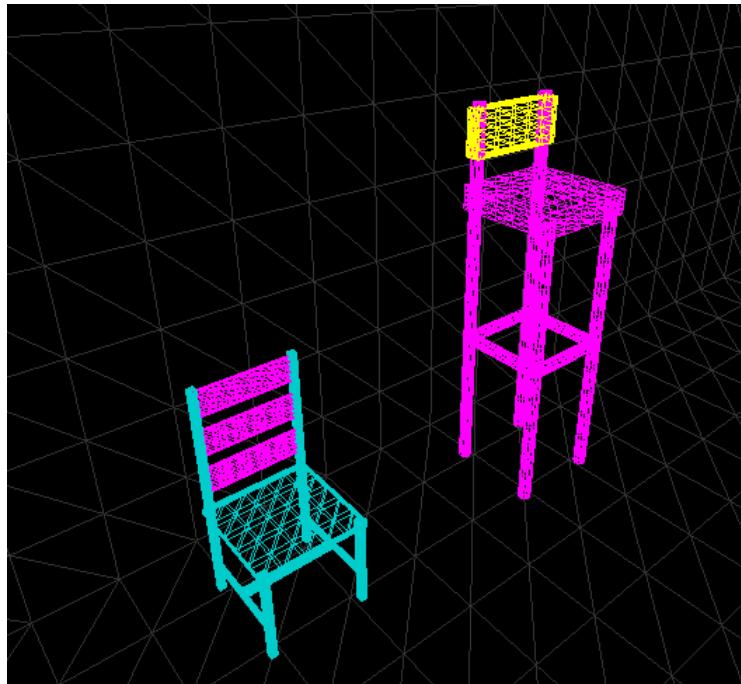


Figura 4: Cadeira normal e banco de balcão

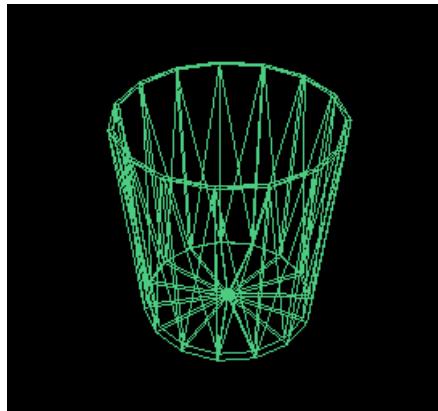


Figura 5: Copo simples

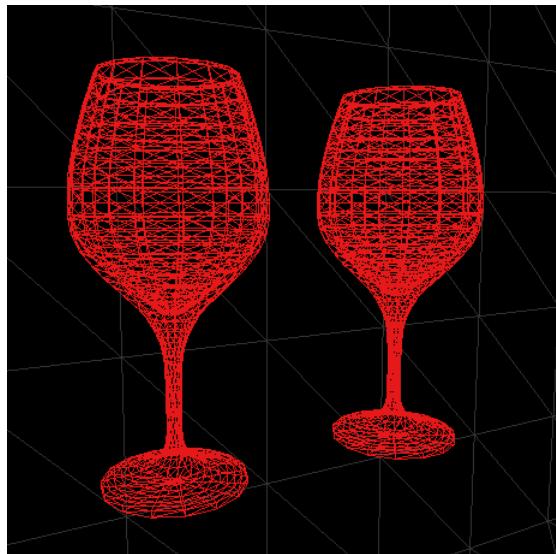


Figura 6: Copo de vinho



Figura 7: Copos de champanhe

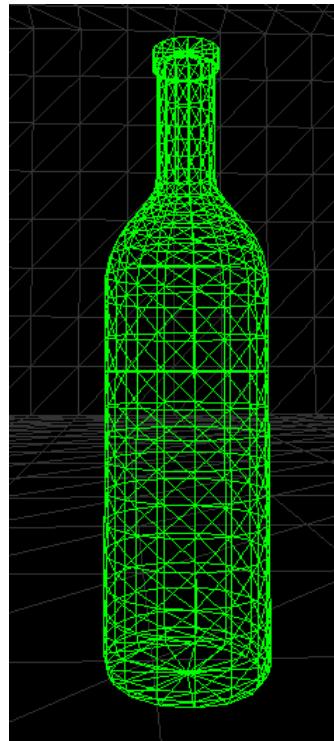


Figura 8: Garrafa

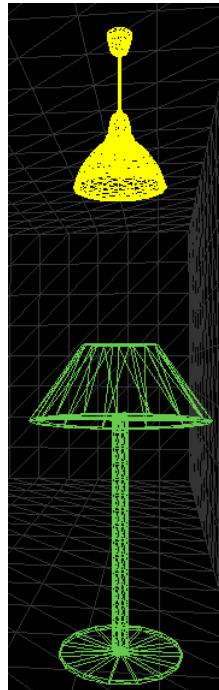


Figura 9: Candeeiro de tecto e de pé

### 3.2.3 Terceira Fase

Tal como já tinha sido referido anteriormente e no enunciado, um dos objectivos para esta etapa do projecto seria a utilização de *Vertex Buffer Objects* no desenho do Bar. Os *VBOs* que implementámos não são óptimos, pois não utilizam indices. Nesta fase serão implementados *VBOs* com indices (ver secção 5.8).

#### ***VBOs nas Primitivas***

Já tínhamos os *VBOs* implementados na segunda fase, portanto nesta fase apenas implementámos indices e adicionámos coordenadas de textura e normais aos vértices.

Nesta fase a classe **Figuras** continua a ser responsável pela composição de **CG\_OBJ** em objectos complexos.

#### **Iluminação e Texturas**

Utilizando as coordenadas de textura e as normais aos vértices nos *VBOs* implementámos texturas e iluminação no nosso Bar. Para não comprometer a escalabilidade do projecto, foram criadas duas novas classes: **Texturas** e **Light**.

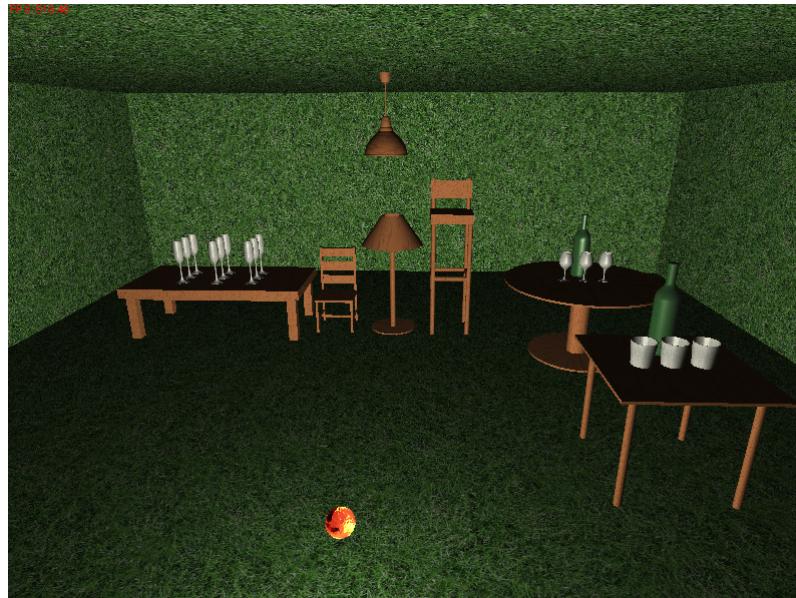


Figura 10: Composição do Bar no fim da fase 3

### 3.2.4 Quarta Fase

Nesta fase deve ser apresentado o produto final deste projecto: o Bar. Para esta fase também nos é sugerido que exista iluminação, texturas e uma câmara de movimento livre.

#### Visualização da cena utilizando uma câmara de movimento livre

A câmara estilo FPS está implementada desde a segunda fase sob a forma da classe **Camera**. Esta classe tem sofrido alterações de acordo com as necessidades. Nesta quarta fase a **Camera** também está encarregue de actualizar os planos do View Frustum (ver secção [5.10](#) sobre View Frustum Culling).

#### Utilização de texturas e iluminação

As texturas e a iluminação foram implementadas na fase anterior. Nesta fase apenas sofreram alterações para adicionar texturas ou facilitar o uso da classe.

#### Aplicação com a apresentação da composição geométrica final

Depois de compostas as várias primitivas criadas ao longo do projecto, obtemos o produto final: o Bar.

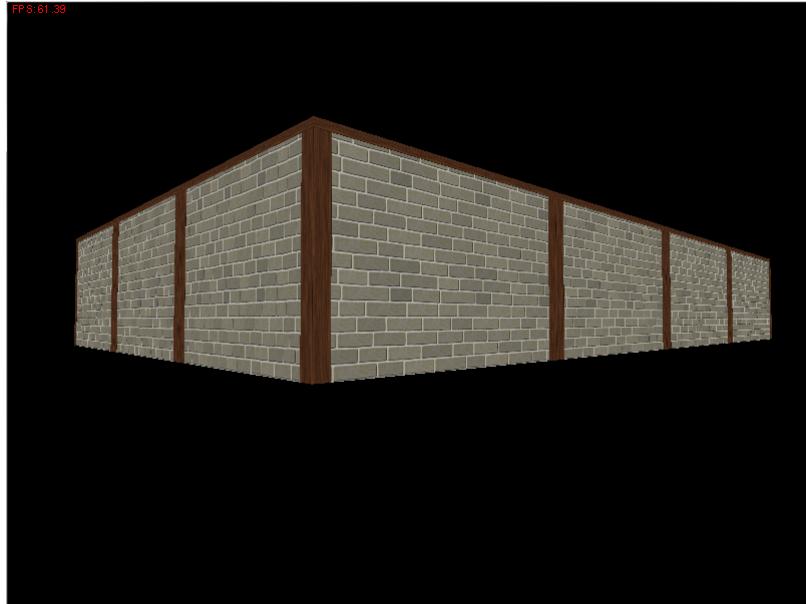


Figura 11: Parede exterior



Figura 12: Perspectiva geral sobre o bar



Figura 13: Balcão, copos, garrafas e bancos



Figura 14: Pormenores do balcão e porta



Figura 15: Mesa e cadeiras



Figura 16: Mesas de bilhar com bolas

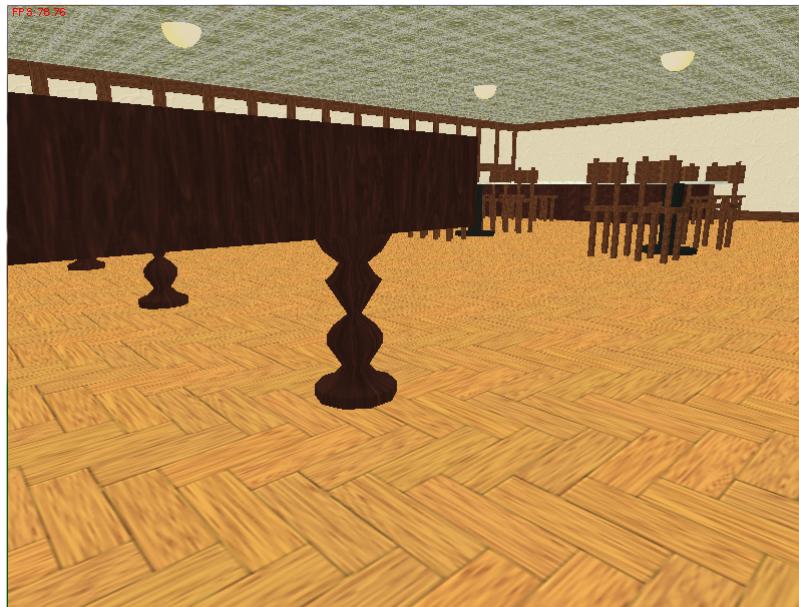


Figura 17: Pormenor das pernas da mesa de bilhar

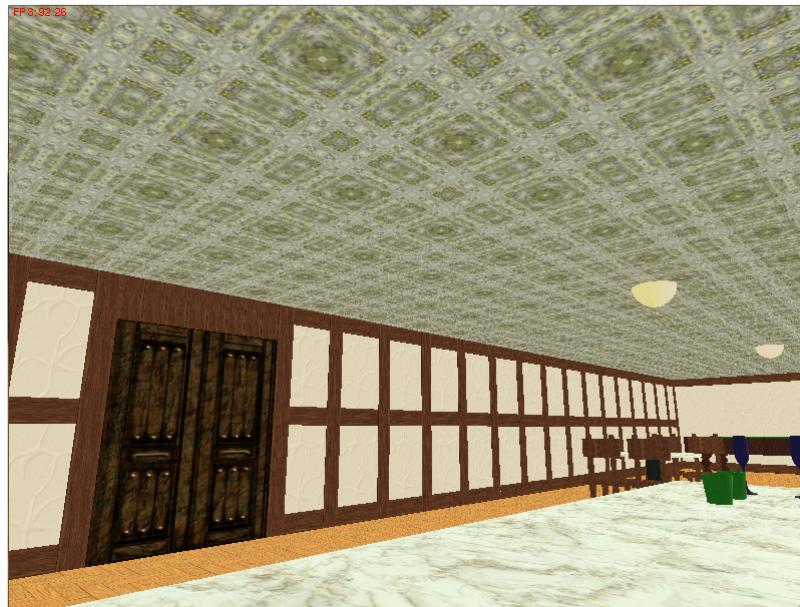


Figura 18: Porta, decoração da parede, tecto e candeeiros



Figura 19: Os copos e garrafas



Figura 20: Perspectiva da outra mesa de bilhar

## 4 Navegação e Controlos

Apresentamos agora os controlos relativos à etapa final do projecto. Estes controlos são responsabilidade das classes **Camera** e **Input**.

### Teclas W,S,A e D

A câmara é movimentada utilizando as teclas A,W,S e D para deslocar a câmara para a esquerda, para a frente, para trás e para a direita, respectivamente.

### Movimento do rato

O movimento do rato permite "olhar em volta".

### Tecla Z

A tecla Z permite alternar entre um modo de visualização em que apenas são desenhadas linhas e o modo de desenho natural.

### Tecla X

A tecla X permite activar e desactivar o View Frustum Culling.

### Barra de Espaços

A barra de espaços permite activar ou desactivar o modo de câmara FPS.

### Tecla + e -

Aumentar e reduzir a velocidade de deslocação da câmara.

## 5 Optimização e Técnicas usadas

Apresentamos agora uma descrição pormenorizada sobre as técnicas usadas e optimizações.

### 5.1 Evitar erros de vírgula flutuante

As variáveis *float*, ou de vírgula flutuante, têm uma precisão limitada. Esta limitação provoca, por vezes, efeitos inesperados ao calcular certos valores. Esta precisão limitada é uma falha aparentemente insignificante, mas que pode causar problemas quando há um acumular de erros de precisão.

Tomemos por exemplo a expressão  $x = 1 * (0.5 - 0.4 - 0.1)$ . É para nós óbvio que  $x = 0$ , mas quando representado em formato de vírgula flutuante em binário, obtemos um valor possivelmente inesperado:  $x = (-2.78 * 10^{-17})$  ou, de forma mais extensa,  $x = -0.000000000000000278$ . Isto acontece porque o valor 0.1 (como muitos outros valores) não é representável em formato de vírgula flutuante. Dependendo da utilização futura do nosso  $x$ , poderíamos ter erros na ordem dos milhões com alguma facilidade.

Para evitar estes erros substituem-se somas sucessivas por multiplicações.

Esta técnica é usada em vários ciclos da aplicação, principalmente quando existem ciclos *for* cujo incremento é um ângulo. Nestes casos tenta-se obter o valor desejado à custa de uma multiplicação em vez de incrementos sucessivos.

## 5.2 ”Camadas” nas primitivas

As camadas nas primitivas são uma forma de assegurar que um triângulo nunca fica muito ”esticado”. Quando um triângulo fica demasiado esticado o *openGL* tem dificuldade em calcular a iluminação para alguns pontos do triângulo e isso provoca um efeito irrealista ao desenhar.

## 5.3 Coordenadas Polares e Esféricas

Na matemática, o sistema de coordenadas polares é um sistema de coordenadas bidimensional no qual qualquer ponto de uma circunferência é determinado a partir de um ângulo e um raio.

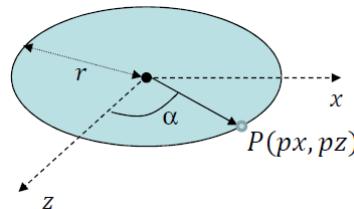


Figura 21: De acordo com a figura, temos que qualquer ponto  $P = (x, y)$  na circunferência pode ser representado sob a forma  $P = (r * \text{sen}(\alpha), r * \cos(\alpha))$

De forma análoga, temos que um sistema de coordenadas esféricas é um sistema de coordenadas tri-dimensional no qual qualquer ponto da esfera é determinado a partir de dois ângulos e um raio.

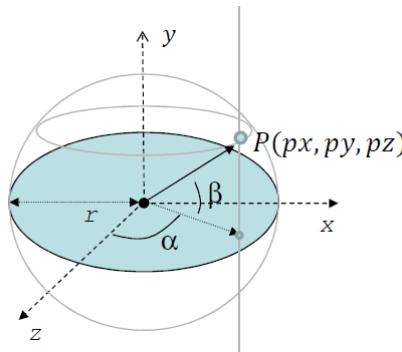


Figura 22: De acordo com a figura, temos que qualquer ponto  $P = (x, y, z)$  numa esfera de raio  $r$  pode ser representado sob a forma  $P = (r * \cos(\beta) * \sin(\alpha), r * \sin(\beta), r * \cos(\beta) * \cos(\alpha))$  para  $-90 < \beta < 90$

As coordenadas polares são utilizadas no cálculo dos vértices do cilindro e nos sólidos de revolução (ver secção 5.6). As coordenadas esféricas são utilizadas no cálculo dos vértices da esfera e no movimento da câmara.

## 5.4 Gestão de input

O input é gerido na classe **Input**. Essa gestão consiste em ter um *array* de 256 booleanos (correspondentes aos 256 caracteres da tabela ASCII) que têm o valor *true* caso a tecla correspondente esteja pressionada, ou *false* caso contrário.

Desta forma, a cada frame, é chamado um método da classe **Input** que realiza acções conforme determinadas teclas estejam pressionadas. Esta abordagem, em conjunto com a directiva *glutIgnoreKeyRepeat(1)*, permite (por exemplo) que o utilizador movimente a câmara simultaneamente em mais que uma direcção.

De notar que esta gestão depende do *frame rate*, ou seja, quanto mais imagens por segundo forem produzidas, mais rápido é o movimento da câmara. Inicialmente, a implementação da classe *input* era baseada num temporizador que a cada 10 milisegundos realizava acções conforme as teclas pressionadas, desta forma o movimento da câmara era independente do *frame rate*, mas em contrapartida processávamos as teclas pressionadas várias vezes por imagem.

## 5.5 Câmara FPS

A câmara FPS é uma implementação de coordenadas esféricas.

Imagine-se a câmara colocada no centro de uma esfera de raio  $r = 1$ . Quando se move o rato, os ângulos  $\alpha$  e  $\beta$  são actualizados e, na próxima chamada à função *gluLookAt*, o ponto *center* para onde a câmara está apontada será calculado utilizando Coordenadas

Esféricas, o raio  $r = 1$  e os ângulos  $\alpha$  e  $\beta$ .

O movimento da câmara FPS também usa os mesmos ângulos  $\alpha$  e  $\beta$ . Para mover a câmara (por exemplo) para a frente, o ponto  $P$  da posição da câmara é deslocado na direcção do vector *center* uma determinada quantidade de unidades.

## 5.6 Sólidos de Revolução

Os sólidos de revolução são uma metodologia para desenhar uma vasta gama de figuras.

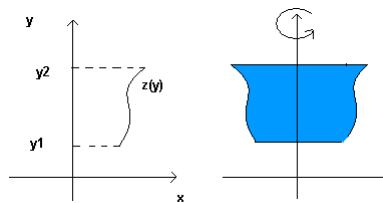


Figura 23: Para desenhar um sólido de revolução é necessário que a aplicação tenha conhecimento de uma lista de pontos sob a forma  $P = (x, y)$ . Depois de ter os pontos, a aplicação une-os, ordenadamente, e roda-os em torno do eixo dos  $yy$ .

O cálculo das normais dos sólidos de revolução produz normais de forma a suavizar a transição de um segmento do sólido de revolução para o próximo.

## 5.7 Evitar mudanças de estado

As placas gráficas processam os vértices em paralelo e de forma extremamente rápida, mas de cada vez que é necessário alterar o estado do *openGL* (p.e para mudar a cor dos vértices a desenhar), é necessário esperar que todos os processadores da placa gráfica acabem de processar os vértices actuais, para depois efectivamente mudar o estado e fornecer mais vértices à placa. Este processo atrasa a operação de desenho dos vértices.

Esta é a razão pela qual (por exemplo) um cubo não deve ser construído a partir de rotações de planos.

## 5.8 Vertex Buffer Objects

Os *Vertex Buffer Objects*, ou *VBOs*, são uma forma de optimizar um projecto de computação gráfica.

Os *VBOs* são buffers na memória da placa gráfica onde é colocado um conjunto de vértices de forma a tornar o desenho desses vértices mais rápido. Os buffers são inicializados e são transferidos os vértices para o buffer apenas uma vez, na inicialização da aplicação. Quando houver necessidade de desenhar um objecto cujos vértices estejam guardados num *VBO*, apenas é necessário que a placa gráfica os carregue a partir da sua memória reservada e os desenhe.

No modo de desenho imediato, os vértices são enviados para a placa gráfica uma vez por *frame*, para depois serem desenhados. Com *VBOs* os vértices são enviados para a

placa apenas uma vez, reduzindo assim o custo de desenho dos vértices para um mínimo.

Os *VBOs* "com indices" são outra optimização que se pode fazer que evita que a placa gráfica tenha que processar mais do que uma vez vértices iguais.

## 5.9 Backface Culling

*Backface culling* consiste em não desenhar a face oculta dos triângulos. A face oculta dos triângulos é definida na especificação dos pontos: se os vértices forem especificados no sentido dos ponteiros do relógio, trata-se de uma face oculta.

O programador da aplicação pode alterar esta norma, fazendo com que a face oculta dos triângulos corresponda aos vértices que foram especificados no sentido contrário ao dos ponteiros do relógio.

Embora esta optimização seja simples de implementar, não é ideal, pois os vértices continuam a ser processados e o ganho em termos de tempo é apenas por não ter de os desenhar.

## 5.10 View Frustum Culling

A implementação do *View Frustum Culling* permite desenhar apenas os objectos que estão no campo de visão da câmara.

Para este efeito, para cada objecto da cena, são testados alguns pontos de forma a verificar se o objecto está visível. Esta verificação requer que sejam conhecidos os planos

que limitam o *View Frustum* e as suas normais. Testar se um ponto está, ou não, dentro do *View Frustum* corresponde a calcular a distância com sinal entre cada um dos planos e o ponto testado.

Na nossa implementação de *View Frustum Culling* são testados um máximo de 8 pontos por objecto. Isto é verdade porque todos os objectos são envolvidos num volume em forma de *caixa*.

Verificar se uma caixa está dentro ou fora do *View Frustum* consiste em rejeitar a caixa se e só todos os pontos da caixa estão do lado errado de um dos planos que definem o *View Frustum*.

O cálculo dos planos e das normais aos planos que definem o *View Frustum* é feito de cada vez que a câmara é reposicionada e o cálculo que verifica se as caixas estão dentro ou fora do *View Frustum* é executado uma vez por caixa, por frame.

Introduzindo o conceito de *Bounding Volumes Hierárquicos* (ver secção 5.11) conseguimos optimizar o cálculo que verifica se as caixas estão dentro ou fora do *View Frustum*. Esta optimização consiste em assumir que *caixas-filho* que estejam dentro de uma *caixa-pai* que está fora do *View Frustum* estão fora do *View Frustum*. De modo semelhante, esta optimização também nos permite assumir que *caixas-filho* que estejam dentro de uma *caixa-pai* que está totalmente dentro do *View Frustum* estão, também elas, dentro do frustum.

## 5.11 Bounding Volumes Hierárquicos

*Bounding Volumes* são aquilo a que chamámos caixas ao descrever o *View Frustum Culling* (ver secção 5.10). Assim, cada *Bounding Volume*, ou caixa, envolve um objecto.

Podemos ter agora caixas que contêm caixas e, desta forma, ter uma hierarquia, ou árvore de caixas.

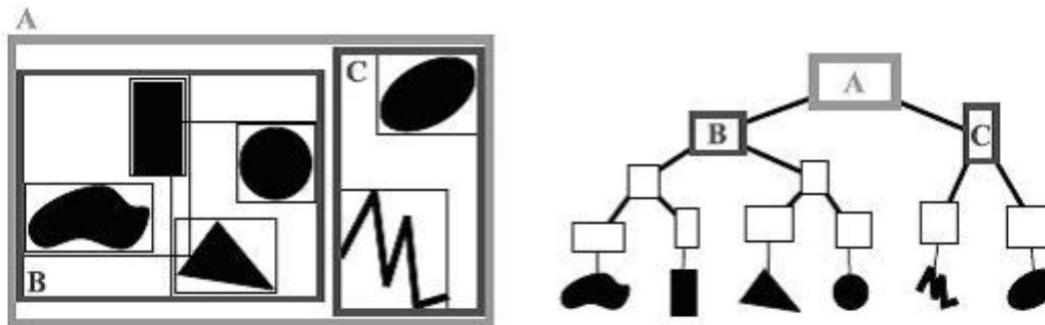


Figura 24: Esta figura ilustra o conceito de *Bounding Volumes Hierárquicos*. À esquerda temos uma representação da partição do espaço, e à direita temos a representação sobre a forma de uma árvore. Temos as caixas A, B e C e vários objectos, cada um com a sua caixa.

Na nossa implementação de *bounding volumes hierárquicos* permitimos que cada caixa tenha ou não um objecto associado, sendo que no caso de não ter um objecto associado, tem forçosamente de ter caixas-filho que lhe correspondam. Uma caixa que não tenha um objecto associado nem caixas-filho não teria volume, como tal não seria um *Bounding Volume Hierárquico*.

## 5.12 Encadeamento de métodos

O encadeamento de métodos é uma técnica que nos permite criar a árvore de *bounding volumes hierárquicos* (ver secção 5.11) sem ter de atribuir um elemento a uma variável para o usar.

Para poder utilizar métodos encadeados alterámos os métodos que modificam os elementos de forma a estes nos devolvam o próprio objecto que foi modificado.

Segue-se um exemplo que ilustra esta técnica.

Suponha-se que temos os seguintes métodos numa classe **Nodo**:

```
Nodo *Nodo::A();  
Nodo *Nodo::B(float x, float y, float z);  
Nodo *Nodo::C(int valor);
```

Que modificam um objecto da classe **Nodo** e devolvem um apontador para um nodo. Inspecionando os métodos, temos que todos eles têm pelo menos um

```
return this;
```

de forma a devolver um apontador para o próprio objecto.

Combinando estas definições com o conceito de árvore, podemos criar uma árvore da forma que abaixo se descreve:

```
Nodo *raiz = new Nodo();  
  
raiz  
    ->addFilho(  
        (new Nodo())->B(1,2,3)->C(9)  
            ->addFilho(  
                (new Nodo())->A()  
            )  
        )  
    ->addFilho(  
        (new Nodo())->C(60)  
    );
```

Desta forma, conseguimos criar a árvore apenas atribuindo valor à variável que representa a raiz da árvore. A árvore gerada teria a seguinte forma:

```
Nodo raiz
|
|-- Nodo ao qual foi aplicado B(1,2,3) e C(9)
|   |
|   '-- Nodo ao qual foi aplicado A()
|
'-- Nodo ao qual foi aplicado C(60)
```

No projeto, a classe **Nodo** do exemplo corresponde, em parte, à classe **ObjectTree**.

## 6 Organização e Descrição do Projecto

Pretendemos com esta secção dar a conhecer algumas características mais técnicas sobre o funcionamento da nossa aplicação.

### 6.1 Descrição Resumida

Esta secção visa esclarecer, de uma forma resumida, quanto à funcionalidade de cada classe presente no projecto, assim como as relações entre elas.

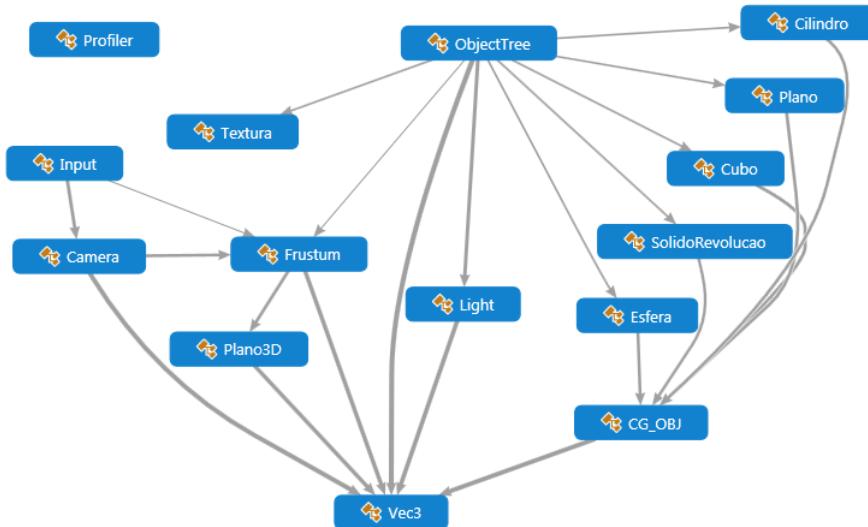


Figura 25: Relações de inclusão entre as várias classes

A classe **CG\_OBJ** (6.2.5) é onde são definidos os *VBOs* e, portanto, as classes **Plano** (6.2.6), **Cubo** (6.2.7), **Cilindro** (6.2.8), **Esfera** (6.2.9) e **SolidoRevolucao** (6.2.10) são todas sub-classes desta. O **Input** (6.2.3) e a **Camera** (6.2.4) estão relacionados, pois o movimento do rato modifica os ângulos da orientação da câmara. O **Input** também controla a activação e desactivação da funcionalidade de *View Frustum Culling* e como tal está relacionada com a classe **Frustum** (6.2.12). A **Camera** controla quando é que o **Frustum** deve ser actualizado e efectua essa actualização. A classe **Frustum** utiliza **Planos3D** (6.2.11) para definir os limites do *View Frustum*. A iluminação é definida na classe **Light** (6.2.2) e as texturas na classe **Textura** (6.2.15), ambas são utilizadas pela classe **ObjectTree** (6.2.13) visto que esta é a classe que agrupa todo o tipo de elementos para construir a cena final. Temos também a classe **Vec3** (6.2.1) que, sendo a definição de um vector com três posições e tratando-se esta de uma aplicação de computação gráfica a três dimensões, é natural que seja utilizada pela maioria das outras classes. Por fim temos a classe **Profiler** (6.2.14), que não depende de outras componentes do projecto e é apenas utilizada no ficheiro **main.c**. O ficheiro **main.c** (6.2.17) e o módulo **Utilities** (6.2.16) não estão representados no gráfico por não serem classes.

## 6.2 Descrição Detalhada

Esta secção e suas sub-secções explicitam com algum pormenor o funcionamento e propósito dos métodos, propriedades, enumerações e definições das várias classes do projecto.

### 6.2.1 Classe Vec3

Esta classe representa um conjunto de 3 vectores. Tem métodos que permitem que alguns cálculos sejam implementados de uma forma mais legível.

#### Propriedades private:

**float vec[3]**

O vector é guardado num array de três elementos.

#### Métodos public:

**Vec3()**

**Vec3(Vec3 \*elem)**

**Vec3(float A, float B, float C)**

**Vec3(float \*vec)**

Métodos construtores da classe.

**Vec3 clone()**

Método de clone da classe.

**float getVal(int pos)**

Obtém o valor da *pos*-ésima posição do array *vec*.

**float X()**

Obtém o valor da primeira posição do array *vec*.

**float Y()**

Obtém o valor da segunda posição do array *vec*.

**float Z()**

Obtém o valor da terceira posição do array *vec*.

**void setVal(int pos, float val)**

Coloca o valor *val* na posição *pos* do array *vec*.

**void setX(float val)**

Coloca o valor *val* na primeira posição do array *vec*.

**void setY(float val)**

Coloca o valor *val* na segunda posição do array *vec*.

**void setZ(float val)**

Coloca o valor *val* na terceira posição do array *vec*.

**float \*getAll(float \*arr)**

Coloca em *arr* os valores do array *vec*.

**void reset(float A, float B, float C)**

Redefine todos os elementos do array *vec*

**void incrementar( Vec3 \*inc )**

Soma os valores do objecto actual com os valores do objecto *inc* e substitui os valores do objecto actual pelo resultado da soma.

**void decrementar( Vec3 \*dec )**

Subtrai os valores do objecto actual pelos valores do objecto *dec* e substitui os valores do objecto actual pelo resultado da subtracção.

**Vec3 somar( Vec3 \*vec )**

Soma os valores do objecto actual com os valores do objecto *vec* e devolve um novo objecto com a soma.

**Vec3 multiplicar( float val )**

Multiplica os valores do objecto actual por um escalar *val* e devolve um novo objecto com o resultado.

**Vec3 crossProduct( Vec3 \*snd )**

Calcula o produto externo entre o objecto actual e o objecto *snd*.

**float innerProduct( Vec3 \*snd)**

Calcula o produto interno entre o objecto actual e o objecto *snd*.

**void normalizar()**

Normaliza o array *vec*. Ou seja, divide cada uma das posições por  $(\sqrt{x^2 + y^2 + z^2})$

**float norma()**

Devolve a norma do objecto actual  $(\sqrt{x^2 + y^2 + z^2})$ .

### 6.2.2 Classe Light

Esta classe controla a iluminação no *openGL*. Parte da classe (os métodos static) são apenas alternativas às funções do *openGL* para definir propriedades das luzes do *openGL*. Os outros métodos e as propriedades permitem agrupar um conjunto de definições relativas à iluminação de modo a permitir a static bool keyDown[256];

#### Propriedades private:

##### **Vec3 pos**

Localização da luz.

##### **float D**

Localização da luz (quarta componente das coordenadas homogéneas).

##### **Vec3 amb**

Cor da componente ambiente da luz.

##### **Vec3 dif**

Cor da componente difusa da luz.

##### **Vec3 esp**

Cor da componente especular da luz.

**Vec3 dir**

Direcção da luz, caso seja um foco de luz.

**float sExp**

Intendidade da luz (0-128).

**float sCutOff**

Abertura da luz ([0-90] ou o valor especial 180)

**GLenum nr**

Número da luz do *openGL* a que corresponde a luz actual

Métodos public:**static void posicao(GLenum light, float x, float y, float z, float w)**

Define a posição de uma determinada luz do *openGL*.

**static void ambiente(GLenum light, float r, float g, float b)**

Define a componente ambiente de uma determinada luz do *openGL*.

**static void difusa(GLenum light, float r, float g, float b)**

Define a componente difusa de uma determinada luz do *openGL*.

**static void especular(GLenum light, float r, float g, float b)**

Define a componente especular de uma determinada luz do *openGL*.

**static void direccao(GLenum light, float x, float y, float z)**

Define a direcção de uma determinada luz direccional do *openGL*.

**static void spotExp(GLenum light, float exp)**

Intendida de uma determinada luz do *OpenGL* (0-128).

**static void spotCutOff(GLenum light, float cutOff)**

Abertura de uma determinada luz do *openGL* ([0-90] ou o valor especial 180).

**static void enable(GLenum light)**

Liga uma determinada luz do *openGL*.

**static void disable(GLenum light)**

Desliga uma determinada luz do *openGL*.

**Light()**

**Light(LuzId id)**

Construtores da classe.

**Light \*setPos(Vec3 pos, float D)**

Define a posição do objecto **Light**.

**Light \*setAmb(Vec3 amb)**

Define a componente ambiente do objecto **Light**.

**Light \*setDif(Vec3 dif)**

Define a componente difusa do objecto **Light**.

**Light \*setEsp(Vec3 esp)**

Define a componente especular do objecto **Light**.

**Light \*setDir(Vec3 dir)**

Define a direcção do objecto **Light**.

**Light \*setSpotExp(float exp)**

Define a intensidade do objecto **Light**.

**Light \*setSpotCutOff(float cutOff)**

Define a abertura do objecto **Light**.

**void aplicarELigar()**

Aplica as propriedades definidas para o actual objecto **Light** à luz do *openGL* que lhe está associada e liga essa luz.

**void desligar()**

Desliga a luz do *openGL* associada ao objecto **Light**.

### 6.2.3 Classe Input

A classe **Input** é a classe essencial para a gestão de input.

#### Propriedades private:

**static bool keyDown[256]**

Array de 256 posições que tem o valor *true* se a posição associada a uma determinada tecla estiver pressionada.

#### Métodos e propriedades public:

**static void init()**

Definir funções para *glutKeyboardFunc*, *glutKeyboardUpFunc*, entre outras.

**static void keyPress(unsigned char tecla, int x, int y)**

Realiza acções para algumas teclas e define o *keyDown* no índice correspondente à tecla pressionada a *true*.

**static void KeyUp(unsigned char tecla, int x, int y)**

Coloca o *keyDown* no índice correspondente à tecla solta a *false*.

**static void processInput()**

Método que executa acções conforme as teclas pressionadas de acordo com o array *keyDown*.

**static bool apenasLinhas**

Permite alternar entre desenhar os objectos de forma opaca ou apenas os seus limites.

### 6.2.4 Classe Camera

Esta classe controla a funcionalidade da câmara FPS no nosso projecto (ver secção 5.5).

#### Propriedades private:

##### **static bool modoFPS**

Permite activar ou desactivar o modo FPS da câmara.

##### **static Vec3 fpsMousePos**

Ultima posição do rato antes de desactivar o modo FPS.

##### **static Vec3 pos**

Posição da câmara.

##### **static float alpha**

##### **static float beta**

Orientação da câmara.

##### **static float passo**

Distância percorrida de cada vez que a câmara se desloca numa determinada direcção.

**static float dnear**

Distância da câmara ao plano near.

**static float dfar**

Distância da câmara ao plano far.

**static float fov**

Parâmetro fov da função gluPerspective.

**static float ratio**

Rácio entre o comprimento e a largura da janela.

**static Vec3 up**

Vector up, como definido na função gluLookAt.

**static Vec3 center**

Ponto para onde a câmara está apontada

**Métodos public:****static void init(float x, float y, float z)**

Inicializa a câmara numa determinada posição.

**static void lookAt()**

Método que utiliza a função gluLookAt de acordo com as propriedades da câmara.

**static void toggleFPS()**

Altera entre o modo FPS e o modo sem controlo sobre a orientação da câmara.

**static void changeSize(int w, int h)**

Método que é chamado quando a janela é redimensionada.

**static void moverFrente()****static void moverTras()****static void moverEsquerda()****static void moverDireita()****static void moveTo(float x, float y, float z)**

Movimento da câmara nas várias direcções ou para um ponto específico.

**static void mouseMove(int x, int y)**

Actualizar os ângulos da orientação da câmara de acordo com a posição actual do rato.

**static void lookAt(float x, float y, float z)**

Forçar a câmara a apontar para um determinado ponto.

**static void passoMaior()**

Aumentar a velocidade de movimento da câmara.

**static void passoMenor()**

Reduzir a velocidade de movimento da câmara.

**static void setOrthographicProjection()**

Muda para a matriz de Projecção para escrever texto.

**static void restorePerspectiveProjection()**

Muda para a matriz Model View Projection.

**static void renderString( float x, float y, int spacing, Fonts font, char \*string)**

Escreve um texto no ecrã na posição  $(x, y)$  com um determinado espaçamento extra entre caracteres e o tipo de letra especificado.

### 6.2.5 Classe CG\_OBJ

Método que permite implementar *VBOs* (ver secção 5.8).

Todas as primitivas e sólidos de revolução são sub-classes desta classe. Desta forma o comportamento de um *VBO* é homogéneo e obtemos maior escalabilidade.

#### Métodos e propriedades private:

**static int maxBuffers**

**static int nBuffers**

Número máximo de buffers e o número de buffers em utilização.

**void addVertex(int \*indice, float x, float y, float z)**

**void addNormal(int \*indice, float x, float y, float z)**

**void addTextureCoord(int \*indice, float x, float y)**

**void addVertex(int \*indice, float x1, float y1, float z1, float x2, float y2,  
float z2, float x3, float y3, float z3)**

**void addNormal(int \*indice, float x1, float y1, float z1, float x2, float y2,**

**float x3, float y3, float z3)**

**void addTextureCoord(int \*indice, float x1, float y1, float x2, float y2,  
float x3, float y3)**

Adiciona vértices, normais ou coordenadas de textura aos arrays de vértices, normais ou coordenadas de textura, respectivamente.

**float comprimento**

Comprimento dos sólidos de revolução.

**void calculateBounds()**

Calcula os limites da caixa que deve envolver o objecto.

**int nTriangulos**

**int nVertices**

**int nFloats**

Número de triângulos, vértices e valores de vírgula flutuante do objecto.

**int bufferPos // posição no buffer onde está o objecto**

Número do primeiro *Vertex Object Buffer* onde está guardado o objecto.

**float \*vertexB // array de coordenadas**

**float \*normalB // array de normais**

**float \*textureB // array de coordenadas de textura**

Arrays de coordenadas, normais e coordenadas de textura, respectivamente.

**int \*vertexI // array de indices de coordenadas**

Array de índices de coordenadas.

**void guardarOBJ(int nTriangulos)**

Método que inicializa alguns valores para depois ser preenchido o array de coordenadas, normais e coordenadas de textura.

**virtual void preencherVertices()**

Método que preenche os arrays de vértices, normais e coordenadas de textura. Este método é implementado por todas as subclasses da classe **CG\_OBJ**, isto porque cada objecto é composto por diferentes vértices.

**void revolutionSolidClose(float \*x, float \*y, int count, int fatias)**

**void revolutionSolidOpen(float \*x, float \*y, int count, int fatias)**

Preenche o array de vértices de acordo com as coordenadas XY e constroi um sólido de revolução.

**Vec3 emissiva**

**Vec3** especular

**Vec3** ambiente

**Vec3** difusa

**float shininess**

Propriedades do material.

### Métodos e propriedades public:

**static GLuint \*buffers**

Array de identificadores de Vertex Buffer Objects.

**static void prepararBuffer(int maxBuffers)**

Aloca espaço e preenche o array de identificadores de VBOs.

**void preencherIndices()**

Identifica os vértices que ocorrem mais que uma vez no objecto e preenche o array de Indices de acordo com essa informação.

**void desenhar()**

Desenha o objecto.

**static int getMaxBuffers()**

Obtém o número máximo de buffers disponíveis.

**Vec3 \*minBound**

**Vec3 \*maxBound**

Obtém os limites do objecto nos vários eixos.

**CG\_OBJ \*setEmissiva(float r, float g, float b)**

**CG\_OBJ \*setEspectral(float r, float g, float b)**

**CG\_OBJ \*setAmbiente(float r, float g, float b)**

**CG\_OBJ \*setDifusa(float r, float g, float b)**

**CG\_OBJ \*setShininess(float s)**

Permite definir as propriedades do material com a possibilidade de encadeamento de métodos.

### 6.2.6 Classe Plano

Classe que implementa a construção de um plano. Sub-classe da classe CG\_OBJ (ver secção 6.2.5).

#### Métodos e propriedades private:

##### **int camadasx**

Número de camadas no eixo dos *xx*.

##### **int camadasz**

Número de camadas no eixo dos *zz*.

##### **float comprimento**

Comprimento no eixo dos *xx*.

##### **float largura**

Comprimento no eixo dos *zz*.

##### **void preencherVertices()**

Implementação específica para o plano do método preencherVertices.

#### Métodos public:

**Plano(float comprimento, int camadasx, int camadasz)**

**Plano(float compX, float compZ, int camadasx, int camadasz)**

Construtores da classe.

### 6.2.7 Classe Cubo

Classe que implementa a construção de um paralelipípedo. Sub-classe da classe CG\_OBJ (ver secção 6.2.5).

A classe chama-se **Cubo** porque começou por construir cubo ao qual aplicávamos escalas em determinados eixos para o transformar num paralelipípedo. Na ultima fase decidimos que seria mais fácil arranjar uma forma de construir um paralelipípedo e não achámos que existisse uma grande necessidade de substituir o nome original da classe.

Apesar destas modificações, a classe **Cubo** continua a ter um construtor que recebe o lado em vez dos vários comprimentos, construindo assim um cubo.

#### Métodos e propriedades private:

**int camadasx**

**int camadasy**

**int camadasz**

Número de camadas nos vários eixo

**float compX**

**float compY**

**float compZ**

Comprimento do cubo nos vários eixos. Apenas se todos os comprimentos forem iguais é que é realmente um cubo.

**void preencherVertices()**

Implementação específica para o cubo do método preencherVertices.

**Métodos public:****Cubo(float lado, int camadasx, int camadasy, int camadasz);****Cubo(float compX, float compY, float compZ, int camadasx, int camadasy, int camadasz)**

Construtores da classe.

### 6.2.8 Classe Cilindro

Classe que implementa a construção de um cilindro. Sub-classe da classe CG\_OBJ (ver secção 6.2.5).

#### Métodos e propriedades private:

##### **float raio**

Raio do cilindro.

##### **float altura**

Altura do cilindro.

##### **int fatias**

Número de fatias do cilindro.

##### **int seccoes**

Número de secções verticais do cilindro

##### **void preencherVertices()**

Implementação específica para o cubo do método preencherVertices.

#### Métodos public:

**Cilindro(float raio, float altura, int fatias, int seccoes)**

Construtor da classe.

### 6.2.9 Classe Esfera

Classe que implementa a construção de uma esfera. Sub-classe da classe CG\_OBJ (ver secção 6.2.5).

#### Métodos e propriedades private:

##### **float raio**

Raio da esfera.

##### **int fatias**

Número de fatias da esfera.

##### **int seccoes**

Número de secções da esfera.

##### **void preencherVertices()**

Implementação específica para o cubo do método preencherVertices.

#### Métodos public:

##### **Esfera(float raio, unsigned fatias, unsigned seccoes)**

Construtor da classe.

### 6.2.10 Classe SolidoRevolucao

Classe que implementa a construção de um sólido de revolução. Sub-classe da classe CG\_OBJ (ver secção [6.2.5](#)).

#### Enumerações:

##### **enum TipoSolidoRevolucao**

Forma de identificar os vários tipos de sólidos de revolução que podem ser criados com esta classe.

#### Métodos e propriedades private:

##### **int fatias**

Número de fatias.

##### **bool fechado**

Propriedade que indica se o sólido de revolução deve ser fechado ou se tem uma abertura no meio.

##### **float x[100]**

**float y[100]**

Arrays que permitem guardar 100 pontos na forma  $(x, y)$

**int pontos**

Número de pontos sobre a forma  $(x, y)$  com valores definidos.

**void preencherVertices()**

Implementação específica para os sólidos de revolução do método preencherVertices.

### Métodos public:

**SolidoRevolucao(TipoSolidoRevolucao tipo, int detalhe)**

Construtor da classe.

### 6.2.11 Classe Plano3D

A classe **Plano3D** permite guardar informação sobre um plano definido a 3 dimensões.

#### Propriedades private:

##### **Vec3 normal**

Normal do plano.

##### **float D**

Distância do plano à origem.

##### **Vec3 ponto**

Ponto no plano.

#### Métodos public:

##### **Plano3D()**

##### **Plano3D( Vec3 \*normal, Vec3 \*ponto )**

Construtores da classe.

**Vec3 getPonto()**

Obter o ponto no plano.

**Vec3 getNormal()**

Obter a normal do plano.

**float distancia( Vec3 \*ponto )**

Obter a distância do plano à origem.

### 6.2.12 Classe Frustum

Classe que define os planos do *View Frustum* e que permite indentificar as caixas que estão dentro, fora ou a intersectar o *View Frustum*.

#### Enumerações:

##### **enum PlanosFrustum**

Forma de identificar os vários planos do *frustum*.

##### **enum PosicaoNoFrustum**

Forma de identificar a posição de uma caixa em relação ao *frustum*.

#### Métodos e propriedades private:

##### **static bool frustumNeedsUpdate**

Permite saber se o frustum precisa de ser actualizado (devido ao movimento da câmara).

##### **static bool frustumCullingEnabled**

Permite saber se o *View Frustum Culling* está activado.

**static Plano3D plano[planoCOUNT\_ENUM]**

Os vários planos do *frustum*.

**static bool pointInFrustum(Vec3 \*ponto)**

Verifica se um ponto se encontra dentro do *view frustum*.

**Métodos public:**

**static void updateFrustum( Vec3 \*pos, Vec3 \*up, Vec3 \*center, float fov**

**float ratio, float dNear, float dFar)**

Actualiza os planos do frustum de acordo com os parâmetros da câmara.

**static PosicaoNoFrustum boxInFrustum(Vec3 \*min, Vec3 \*max)**

Verifica se uma caixa está dentro, fora ou a intersectar o view frustum.

**static void toggleFrustumCulling()**

Activar e desactivar o *View Frustum Culling*.

**static bool isCullingEnabled()**

Verifica se o *View Frustum Culling* está activo.

**static void scheduleUpdate()**

Informa a classe que os planos do frustum precisam de ser atualizados.

### 6.2.13 Classe ObjectTree

Esta classe organiza os elementos visuais do nosso projecto sob a forma de uma árvore (ver secção 5.11 sobre *Bounding Volumes Hierárquicos*). Esta classe também desenha toda a cena percorrendo a árvore da raiz para as folhas.

#### Enumeração:

##### **enum TipoMod**

Forma de identificar os vários tipos de modificações: translações, rotações e escalas.

#### Métodos e propriedades private:

##### **int modsCount**

Número de modificações efectuadas no objecto.

##### **TipoMod \*modsTipo**

Array do tipo de modificações.

##### **Vec3 \*modsVec**

Array de vectores das modificações.

**float \*modsExtra**

Array de ângulos necessários para efectuar rotações.

**bool texturar**

Indica se ao objecto deve ser aplicada uma textura.

**Vec3 cor**

Indica a cor do objecto caso não seja implementada uma textura.

**TipoTextura texTipo**

Tipo da textura a aplicar ao objecto.

**float texScaleX, texScaleY, texAnguloRotacao**

Escalas e ângulo de rotação da textura a aplicar.

**Light \*luzes[8 ]**

Luzes que irão iluminar o objecto.

**CG\_OBJ \*obj**

O objecto a desenhar.

**Vec3 \*boundsMin**

Os limites inferiores do objecto e dos filhos nos vários eixos.

**Vec3 \*boundsMax**

Os limites superiores do objecto e dos filhos nos vários eixos.

**bool \*toggle**

Activar ou desactivar o desenho do objecto.

**int numFilhos**

Número de filhos que o objecto tem.

**ObjectTree \*\*filhos**

Array de filhos do objecto.

**static ObjectTree \*raizObj**

A raiz da árvore.

**static void drawAux(ObjectTree \*raiz, PosicaoNoFrustum posNoFrustum)**

Desenhar o objecto caso este esteja no *frustum*. Fazer a mesma operação para todos os filhos do objecto. Ver [6.2.12](#) sobre a classe Frustum.

**static void checkBounds(ObjectTree \*tree)**

Verificar recursivamente os limites dos vários objectos da árvore.

**static void localEfectivo(ObjectTree \*tree, Vec3 \*ponto)**

Modifica o ponto de acordo com as rotações, translações e escalas associadas ao objecto.

**static Light \*luzesGlobais[8]**

Definição de luzes globais a todos os objectos.

### Métodos e propriedades public:

**ObjectTree()**

Construtor da classe.

**ObjectTree \*rotate( float angulo, Vec3 vec )**

**ObjectTree \*translate( Vec3 vec )**

**ObjectTree \*scale( Vec3 vec )**

Adiciona uma modificação ao objecto.

**ObjectTree \*color( Vec3 vec )**

Define a cor do objecto.

**ObjectTree \*texture(TipoTextura tipo, float sX, float sY, float anguloRot)**

Define a textura do objecto.

**ObjectTree \*addLight( Light \*luz )**

Adiciona uma luz ao objecto.

**ObjectTree \*addLights( Light \*luzes[], int i0=-1, int i1=-1, int i2=-1  
int i3=-1, int i4=-1, int i5=-1, int i6=-1, int i7=-1)**

Selecciona as luzes a adicionar ao objecto a partir de um array de luzes.

**ObjectTree \*objecto( CG\_OBJ \*O )**

Adiciona uma figura ao actual nodo da árvore.

**ObjectTree \*addFilho( ObjectTree \*filho )**

Adiciona um filho ao actual nodo da árvore.

**static void init()**

Inicializa algumas variáveis e prossegue para inicializar toda a toda a árvore recorrendo a encadeamento de métodos. Por fim calcula os limites (i.e as caixas) para cada nodo da árvore.

**static void draw()**

Desenhar a árvore. Este método apenas inicia o desenho recursivo da árvore começando pela raiz.

### 6.2.14 Classe Profiler

Esta classe permite a cronometragem de determinadas tarefas. Na quarta fase a classe apenas é utilizada para calcular as frames por segundo da aplicação. Ao longo do desenvolvimento do projecto, esta classe foi utilizada para medir e comparar os tempos de implementações alternativas para nos auxiliar numa tomada de decisão.

#### Enumerações:

##### **enum ProfilerData**

Forma de identificar os vários cronómetros existentes.

#### Propriedades private:

##### **static int \*cronometros**

Array de tempos, para poder manter registo de vários cronómetros simultaneamente.

##### **static int \*emPausa**

Array que indica se o cronometro correspondente no array de cronómetros está em pausa e quanto tempo tinha contado antes de ter sido posto em pausa.

**static int frames**

Valor utilizado para calcular as frames por segundo.

**static float fps**

Ultimo valor calculado para frames por segundo.

**Métodos public:****static void init()**

Alocar espaço para os vários cronómetros.

**static int diff(ProfilerData cronometro)**

Calcula quanto tempo já passou desde que o cronómetro foi iniciado.

**static void start(ProfilerData cronometro)**

Iniciar ou continuar um cronómetro.

**static void pause(ProfilerData cronometro)**

Pausar um cronómetro. Consiste em guardar o tempo actual e adicioná-lo ao cronómetro da próxima vez que este for iniciado.

**static int now()**

Obtém o tempo sob a forma de número de milisegundos que passaram desde a chamada à função glutInit.

**static void startFrame()**

Actualiza o contador de frames que passaram no segundo actual e, caso já tenha passado um segundo completo, regista quantas frames foram desenhadas.

**static float getFPS()**

Obter o número de frames por segundo (valor diz respeito ao ultimo segundo).

### 6.2.15 Classe Textura

A classe **Textura** permite carregar e utilizar diversas texturas no nosso bar.

#### Enumerações:

**enum TipoTextura : int**

Forma de identificação das várias texturas.

#### Propriedades e métodos private:

**static GLuint \*textureIds**

Identificadores de textura.

**static Textura \*texturas**

Objectos do tipo Textura que permitem aplicar a textura a uma figura.

**static ILuint \*imageIds**

Identificadores de imagem.

**static char \*\*filenames**

Array de string que guarda o nome do ficheiro fonte.

**int tw, th**

O comprimento e largura da imagem fonte.

**ILubyte \*imageData**

Identificador da imagem importada.

**Textura(TipoTextura tipoTextura)**

Construtor da classe.

### Métodos public:

**static void init()**

Carrega as imagens e converte-as para texturas do openGL.

**static int getWidth(TipoTextura tipoTextura)****static int getHeight(TipoTextura tipoTextura)**

Obter o comprimento e largura de uma imagem.

**static void setTextura(TipoTextura tipoTextura, float sx=1, float sy=1, float graus=0)**

Aplicar uma textura com uma determinada escala e rotação.

**static void setTextura(float sx, float sy, float graus=0)**

Aplica uma escala e rotação à textura previamente seleccionada.

**static void setColor(float R, float G, float B)**

Aplicar uma cor RGB em vez de uma textura.

**static void translate(float x, float y)**

Aplica uma translação à textura.

**static void unsetTextura()**

Activar a textura "vazia" do openGL.

**int getWidth()**

**int getHeight()**

Obter o comprimento e a largura da textura.

### 6.2.16 Módulo Utilities

Nas fases anteriores, este módulo tinha várias funções relacionadas com cálculo algébrico, mas com a introdução da classe **Vec3** na fase final do projecto, esses cálculos passaram a ser feitos na nova classe. O que restou nesta classe foram duas definições.

#### Definições:

**toDegree(rad)**

**toRadian(deg)**

Converter radianos em graus e o processo inverso. Os números apresentados na definição correspondem a uma aproximação de  $\frac{180}{\pi}$  e  $\frac{\pi}{180}$ , respectivamente.

### 6.2.17 Ficheiro main.c

#### Funções:

##### **renderScene**

A função renderScene pede à classe **Input** para interpretar qualquer input, indica à classe **Profiler** que está a ser desenhada uma nova frame e prossegue para o desenho da cena, que é feito através de uma chamada ao método draw da classe **ObjectTree**. Depois do desenho da cena é escrito o valor correspondente às frames por segundo no canto superior direito do ecrã e são trocados os buffers de desenho.

##### **main**

A função main inicializa todo o tipo de variáveis que podem ser inicializadas em modo *offline* e entrega o controlo da aplicação ao glut.

## 7 Conclusão

Fazendo uma retrospectiva ao projecto por nós desenvolvido, pensamos ter cumprido os requisitos exigidos para a aplicação, assim como algumas funcionalidades que implementámos para melhorar o nosso produto final.

Todos os pontos exigidos no enunciado foram cumpridos para as correspondentes fases com alguma facilidade, à excepção da fase final. Este facto deve-se, por um lado, à nossa implementação parcial de *VBOs* na segunda fase e assim reduzir a quantidade de funcionalidades a implementar na terceira fase. Por outro lado o tempo foi escasso para realizar a quarta fase, com a agravante de termos começado a implementar um modo de desenho do bar que, além de não ser facilmente escalável, acabou por se provar ineficiente. Este contratempo deixou-nos com apenas uma semana para compor o bar utilizando uma outra qualquer implementação. De forma a valorizar o projecto, decidimos implementar *View Frustum Culling* com *Bounding Volumes Hierárquicos*. Talvez esta ultima decisão tenha contribuido para o atraso na entrega do nosso bar, mas consideramos que o que mais nos prejudicou em termos de tempo foi a nossa primeira tentativa (falhada) de desenhar o bar. Esta decisão de utilizar *View Frustum Culling* com *Bounding Volumes Hierárquicos* também foi um desafio que, depois de superado, nos proporcionou uma aprendizagem mais completa na área da computação gráfica.

Desde o início do projecto houve uma grande preocupação da nossa parte em ter o projecto bem organizado, orientado a objectos e extremamente escalável. Esta preocupação deve-se a alguma experiência prévia em programação orientada a objectos e

a consciência de que seria sempre muito mais fácil acrescentar novas funcionalidades a um projecto escalável e orientado a objectos. Um exemplo desta nossa preocupação é a classe **CG\_OBJ**, que controla todos os aspectos referentes aos *VBOs* à excepção da introdução de vértices. Esta parte crucial da definição de um *VBO*, a introdução dos vértices, é feita de uma forma dinâmica através da definição de um método abstracto (ou virtual, em C++) em todas as sub-classes de **CG\_OBJ**. Outro exemplo é a forma como são definidas as texturas, que nos permitem adicionar uma nova textura especificando apenas um nome para a textura e a sua localização em disco. Foi também tido algum cuidado para manter um equilíbrio no que diz respeito a optimizações numa fase de desenvolvimento do projecto. Tentámos não optimizar o código, mas ao mesmo tempo não utilizar técnicas pouco eficientes.

Podemos então afirmar que o desenvolvimento desde projecto foi concluído, no entanto reconhecemos que poderiam ter sido aplicadas mais optimizações e técnicas no projecto, como por exemplo a redução do numero de mudanças de estado antes do desenho de cada objecto, uma melhor divisão dos objectos pela Hierarquia de Volumes ou a introdução de sombras.

## 8 Referências

### 8.1 Referências bibliográficas

#### Slides teóricos e práticos

Slides mostrados durante as aulas de Computação Gráfica.

### 8.2 Referências WWW

#### [www.lighthouse3d.com](http://www.lighthouse3d.com)

Lighthouse3D is a site devoted mostly to 3D computer graphics.

## 9 Membros do Grupo



**Bruno Ferreira**  
**A61055**



**Serafim Pinto**  
**A61056**