



UNIVERSIDADE DO MINHO
Departamento de Informática

Entrega 1 - Computação Gráfica



Realizado por:

André Mota - A81384 | Gonçalo Braga - A97541 | Rodrigo Pereira - A96561

<https://github.com/acmota2/CG2023>

10 de março de 2023

Conteúdo

1	Introdução	2
2	Objetivos	2
3	Ficheiros utilizados no gerador e no motor 3D	2
4	Criação de um gerador	4
4.1	Suporte aos pedidos efetuados pelos utilizadores	4
4.1.1	Tratamento dos pedidos	5
4.2	Cálculo dos pontos necessários para suportar o pedido do utilizador	5
4.2.1	DrawPlane - Cálculo dos pontos de um plano	5
4.2.2	DrawBox - Cálculo dos pontos de uma caixa	6
4.2.3	DrawSphere - Cálculo dos pontos de uma esfera	6
4.2.4	DrawCone - Cálculo dos pontos de um cone	7
4.3	Resposta ao pedido do utilizador	8
5	Motor 3D	8
5.1	Parse de .xml	8
5.2	Processamento de .obj	8
5.3	Câmara	8
6	Resultado Finais	9
7	Conclusão	11

Conteúdo

1 Introdução

O objetivo deste trabalho prático é desenvolver uma cena gráfica, tendo como base um **motor 3D**.

Para podermos ver o potencial deste motor 3D temos de fornecer exemplos para que se possa visualizar o funcionamento do mesmo, exemplos esses, que são criados por um **gerador**.

Este trabalho prático encontra-se subdividido em 4 partes. Neste relatório, em específico, iremos dar mais ênfase à **primeira parte** do trabalho prático.

Esta primeira fase do trabalho prático, consiste de uma forma geral, em criar um motor 3D para a cena gráfica, bem como um gerador, que tem como papel principal, *calcular* pontos, que quando compostos formam **triângulos** que irão ser consumidos pelo motor 3D.

2 Objetivos

Nesta secção do relatório estão devidamente definidos os **principais** objetivos da primeira parte do trabalho prático.

- Criação de um **motor 3D**;
- Criação de um **gerador**;
- Colocação de uma câmara na cena gráfica (processo realizado pelo motor 3D).

3 Ficheiros utilizados no gerador e no motor 3D

Como irá ser possível visualizar noutras secções, o gerador irá gerar figuras que, posteriormente, irão ser consumidas pelo motor 3D, dando origem às cenas.

Com isto, tem de existir um meio de comunicação entre o motor 3D e o gerador, isto é, aquilo que o gerador **produz** o motor 3D tem de **consumir** para criar as cenas.

Sendo assim, os ficheiros gerados são ficheiros **.obj**. Estes ficheiros têm uma particularidade em relação aos ficheiros **.txt** convencionais, isto é, não descrevem as instruções em si; apenas descrevem os parâmetros que as instruções irão necessitar.

Um exemplo de um ficheiro **.txt** que pode ser utilizado nesta fase é o seguinte:

```
glBegin(GL_TRIANGLES);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1000.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1000.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 1000.0f);
glEnd();
```

Neste trabalho prático, iremos utilizar os ficheiros **Wavefront OBJ** que são definidos da seguinte forma:

- v - Representa as coordenadas de um determinado vértice;

- vn - Representa a normal de um qualquer vértice;
- vt - Representa a textura de um qualquer vértice;
- f - Representa a face de uma figura.

Um ficheiro .obj pode ser representado da seguinte forma:

```
v 1 0 1
v 0 1 1
v 2 3 1
f 1/0/0 2/0/0 3/0/0
```

Nas faces, cada conjunto de x/x/x representa o seguinte:

- f 1/2/3 4/5/6 7/8/9 - A face é constituída pelos vértices que estão nas linhas 1, 4 e 7, cujas texturas estão na linha 2, 5 e 8 e cujas normais estão nas linhas 3, 6 e 9, respetivamente.

Uma das vantagens do formato deste ficheiro é o facto de usarmos um tipo de ficheiro **standard**. Isto leva-nos a que possamos importar modelos variados, se assim o precisarmos. Apenas para título de exemplo, importámos um ficheiro OBJ que possuía um modelo de um carro de fórmula 1. Este modelo quando foi submetido ao nosso motor deu-nos esta cena:

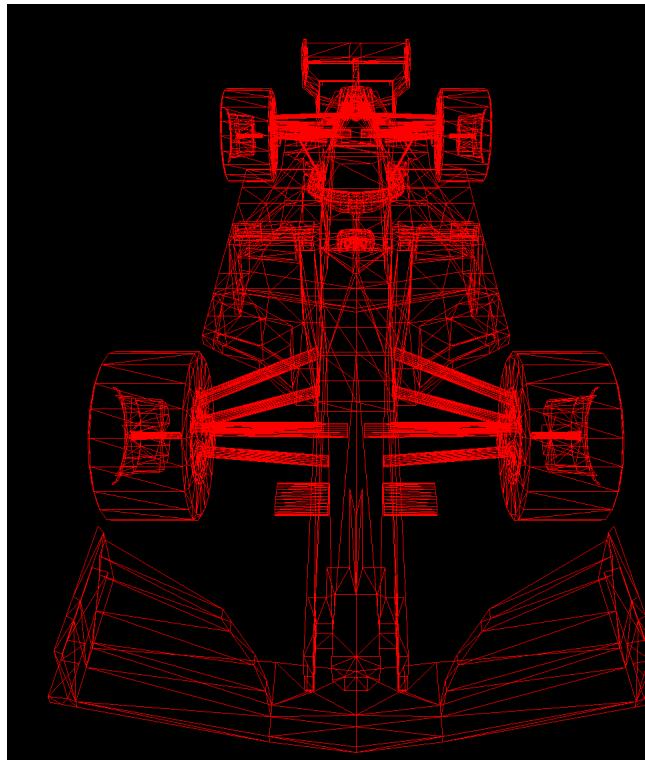


Figura 1: Modelo de fórmula 1 deduzido a partir de um ficheiro OBJ

Conseguimos perceber quais as vantagens que os ficheiros OBJ têm face aos ficheiros txt convencionais.

Sendo assim, conseguimos perceber como estão estruturados os ficheiros gerados pelo gerador e consumidos pelo motor 3D.

4 Criação de um gerador

O **gerador** tem de possuir três *papéis* distintos, isto é:

- Suportar os pedidos efetuados pelo utilizador;
- Calcular os pontos necessários para suportar o pedido do utilizador;
- Responder ao pedido do utilizador, com um ficheiro onde estão expressos os pontos que formam a cena pedida pelo utilizador.

4.1 Suporte aos pedidos efetuados pelos utilizadores

Os utilizadores podem pedir para serem calculados os pontos constituintes das seguintes figuras:

- Plano
- Caixa
- Esfera
- Cone

Sendo assim, necessitam de inserir linhas específicas quando executam o programa, tal como:

```
$ ./generator plane 1 3 plane.3d
```

Na imagem acima, podemos ver a linha de *input* que os utilizadores irão ter de colocar, caso pretendam obter os pontos de um plano no eixo XZ.

Nesse caso, o número **1** representa o *length* que o plano irá ter em cada lado e o número **3** representa o número de *divisões* que o plano irá ter no eixo do X e do Z.

Por ultimo, o nome **plane.3d** é o nome do ficheiro resultado que irá conter os pontos e as faces constituintes do plano pedido.

```
$ ./generator box 2 3 box.3d
```

Na imagem acima, podemos ver a linha de input que os utilizadores irão ter de colocar, caso pretendam obter os pontos de uma caixa centrada na origem.

Desta forma, o número **2** representa o *length* que a caixa irá ter em cada lado e o número **3** representa o número de *divisões* que a caixa irá ter no eixo do X,Y e Z.

Por fim, o nome **box.3d** é o nome do ficheiro resultado que irá conter os pontos e as faces constituintes da caixa pedida.

```
$ ./generator sphere 1 10 10 sphere.3d
```

Na imagem acima, podemos ver a linha de input que os utilizadores irão ter de colocar, caso pretendam obter os pontos de uma esfera centrada na origem.

Assim, o número **1** representa o *radius* que a esfera possui, o primeiro número **10** representa o número de *slices* que a esfera irá ter e o segundo número **10** representa o número de *stacks* que a esfera irá ter.

Por fim, o nome **sphere.3d** é o nome do ficheiro resultado que irá conter os pontos e as faces constituintes da esfera pedida.

```
$ ./generator cone 1 2 3 4 cone.3d
```

Na imagem acima, podemos ver a linha de input que os utilizadores irão ter de colocar, caso pretendam obter os pontos de um cone com base no plano XZ.

Assim, o número **1** representa o *radius* que o cone irá ter na base, o número **2** representa a *height* que o cone irá possuir, o número **4** representa o número de *slices* que o cone irá possuir e o número **3** representa o número de *stacks* que o cone irá possuir.

Por fim, o nome **cone.3d** é o nome do ficheiro resultado que irá conter os pontos e as faces constituintes do cone pedido.

4.1.1 Tratamento dos pedidos

Com base no que foi possível ver na secção abordada anteriormente, conseguimos perceber qual o *input* que o gerador irá receber no arranque do mesmo.

Sendo assim, para tratarmos desses pedidos, foram feitas funções para tratar cada um e para dividir corretamente estes pedidos. A função **main** do programa controla os inputs recebidos por argumento.

4.2 Cálculo dos pontos necessários para suportar o pedido do utilizador

Para atender ao pedido dos utilizadores criámos funções que calculem os pontos constituintes das figuras pedidas, sendo assim temos ...

4.2.1 DrawPlane - Cálculo dos pontos de um plano

Para que o utilizador possa desenhar um plano no eixo XZ, temos de calcular os pontos, que suportam os triângulos existentes no plano, tendo em conta a **regra da mão direita**. Sendo assim foram criados os valores base do plano. Os valores bases foram calculados da seguinte forma:

```
1 float right_most = length / 2.;
2 std::vector<float> base_points;
3 float division_size = length / ((float) divisions);
4 for (auto i = 0; i <= divisions; ++i) {
5     base_points.push_back(right_most - (division_size * i));
6 }
```

Os valores base são considerados o *esqueleto* do plano, uma vez que o plano é constituído, por múltiplas combinações de valores base, dando origem aos pontos do plano.

. Depois de calculados os pontos dos triângulos estes são agrupados em faces(triângulos), com base na **regra da mão direita**, para que seja possível visualizar o plano, que irá ser criado posteriormente pelo motor 3D.

O que foi abordado em cima é possível ser visualizado nesta imagem:

```
1 for (auto i = 0; i < divisions; ++i) {
2     for (auto j = 1; j <= divisions; ++j) {
3         auto index = (divisions + 1) * i + j;
4         f << "f " << index << "/0/0 ";
5         f << index + 1 << "/0/0 ";
6         f << index + divisions + 1 << "/0/0\n";
7         f << "f " << index + 1 << "/0/0 ";
8         f << index + divisions + 2 << "/0/0 ";
9         f << index + divisions + 1 << "/0/0\n";
```

```

10     }
11 }
```

4.2.2 DrawBox - Cálculo dos pontos de uma caixa

Funciona de forma homónima a DrawPlane. Como a caixa é centrada na origem, os valores das coordenadas serão os mesmos em todas as faces, mudando apenas os eixos em que estes valores se encontram.

Desta forma, um cubo pode ser definido de forma bastante idêntica a um plano, se o mesmo processo do plano for repetido para todas as faces, apenas mudando o eixo em que os pontos previamente calculados se encontram.

4.2.3 DrawSphere - Cálculo dos pontos de uma esfera

A forma de calcular os pontos dumha esfera em tudo se assemelha à forma de calcular os pontos dum cilindro, ou seja, através do cálculo dos pontos das bases e dos pontos da face. A diferença na esfera deve-se a estes pontos terem um *offset* em relação à origem dado pelo seno dum ângulo beta. O cálculo dos vértices é dado pelo seguinte código:

```

1 f << "v " << 0 << ' , ' << radius << ' , ' << 0 << '\n';
2 for (std::size_t i = 1; i < stacks; ++i) {
3     for (std::size_t j = 0; j < slices; ++j) {
4         float sinb = sin((i * M_PI) / stacks);
5         f << "v " << std::noshowpoint << std::fixed << std::setprecision(6)
6             << sin(j * (2 * M_PI / slices)) * sinb * radius
7             << ' , ' << cos((i * M_PI) / stacks) * radius
8             << ' , ' << cos(j * (2 * M_PI / slices)) * sinb * radius << '\n';
9     }
10 }
11 f << "v " << 0 << ' , ' << -radius << ' , ' << 0 << '\n';
```

Os triângulos formados por estes pontos são posteriormente calculados através dos índices de 1 a $stacks \times slices + 2$ correspondentes à ordem pela qual são impressos no ficheiro .obj que irá resultar. O primeiro índice de cada triângulo poderia ser escolhido de forma arbitrária, desde que os restantes respeitassem a ordem da **regra da mão direita**, de forma a garantir que ao desenhar a face pintada fique de fora da esfera.

De forma homónima a como os pontos são definidos, os casos das supostas bases são tratados em separado, e o caso dos restantes pontos é feito no mesmo ciclo:

```

1 // base 1
2 for(std::size_t i = 0; i < slices; ++i) {
3     f << "f " << 1 << "/0/0 ";
4     f << (i%slices) + 2 << "/0/0 ";
5     f << ((i+1)%slices) + 2 << "/0/0\n";
6 }
7 // "faces"
8 for(std::size_t i = 0; i < stacks - 2; ++i) {
9     for(std::size_t j = 0; j < slices; ++j) {
10         std::size_t r = (j%slices) + 2 + (i * slices);
11         std::size_t l = ((j+1)%slices) + 2 + (i * slices);
12         // |
13         f << "f " << r << "/0/0 ";
14         f << l + slices << "/0/0 ";
15         f << l << "/0/0\n";
16         // \
17         f << "f " << r << "/0/0 ";
18         f << r + slices << "/0/0 ";
19         f << l + slices << "/0/0\n";
```

```

20     }
21 }
22 // base 2
23 std::size_t bottom_most = slices*(stacks-1)+2;
24 for(std::size_t i = 0; i < slices; ++i) {
25     f << "f " << bottom_most << "/0/0 ";
26     f << bottom_most - slices + (i+1)%slices << "/0/0 ";
27     f << bottom_most - slices + (i%slices) << "/0/0\n";
28 }
29 }
```

4.2.4 DrawCone - Cálculo dos pontos de um cone

O cálculo dos pontos constituintes do cone, é feito de acordo com o número de **stacks** e **slices** existentes.

Para cada **stack** existente no cone vamos calcular os pontos constituintes das suas **slices**.

Desta forma, começamos a calcular os pontos opostos á base do cone,tendo em conta a **regra da mão direita**, fazendo assim a extremidade do mesmo, como podemos ver neste excerto de código.

```

1 int points=1;
2 std::ofstream f;
3 f.open(file);
4 float delta = (2*M_PI) / (float)slices;
5 float stack_h = height / (float)stacks;
6 float curr_h = height;
7 float stack_r = radius / (float)stacks;
8 float curr_r = 0;
9 int i = 0;
10 curr_r += stack_r;
11 while (i < slices)
12 {
13     float aCil = delta*i;
14     float px1 = (sin(aCil)), py1 = (cos(aCil));
15     float px2 = (sin(aCil + delta)), py2 = (cos(aCil + delta));
16
17     f << "v " << std::setprecision(6) << 0.0 << ', ' << curr_h << ', ' << 0.0 << '\n';
18     ++points;
19     f << "v " << std::setprecision(6) << px1 * curr_r << ', ' << curr_h - stack_h <<
20     ', ' << py1 * curr_r << '\n';
21     ++points;
22     f << "v " << std::setprecision(6) << px2 * curr_r << ', ' << curr_h - stack_h <<
23     ', ' << py2 * curr_r << '\n';
24     ++points;
25     i++;
26 }
```

Desta forma, depois de calculados os pontos opostos á base, vão sendo calculados os pontos mais próximos da base, todos correspondentes á mesma **stack**.

Depois de fazermos este processo para todas as **stacks**, o cálculo da base do cone, é idêntico ao cálculo da base do cilindro, que foi feito nas aulas práticas.

4.3 Resposta ao pedido do utilizador

Como sabemos, o **gerador** têm de dar como **output** um ficheiro, com os pontos constituintes da figura pedida. Sendo assim ao longo da execução, é transcrita para o ficheiro de output os vértices cálculos como podemos ver neste excerto.

```
1 f << "v " << std::setprecision(6) << 0.0 << ',' << curr_h << ',' << 0.0 << '\n';
```

Esta inscrição dos vértices no ficheiro, têm de ser acompanhada também pela inscrição das faces constituintes da figura no ficheiro, sendo assim, como os vértices se encontram seguidos no ficheiro, então o cálculo das faces, é feito da seguinte forma. Este exemplo, é demonstrativo de como as faces de um cone, são calculadas.

```
1 for (auto i=1; i+3<=points; i=i+3){  
2     f << "f " << i << "/0/0 ";  
3     f << i+1 << "/0/0 ";  
4     f << i+2 << "/0/0\n";  
5 }
```

Com base no que foi abordado em cima, conseguimos perceber como são escritos os vértices e as faces da figura pedida no ficheiro de output.

5 Motor 3D

5.1 Parse de .xml

O parsing de .xml é feito com recurso à biblioteca **rapidxml**.

5.2 Processamento de .obj

O processamento de .obj é feito tendo em consideração os campos pertencentes a estes. Como estes ficheiros representam um sistema de referenciamento de pontos por índice, os pontos são guardados num vetor, para posteriormente serem referenciados para quando são desenhadas as faces.

Para guardar cada modelo gerado, foi criada a **class model**, de forma a concentrar todas as operações possíveis para um modelo no mesmo sítio, e assim facilitar a implementação de mais funcionalidades no futuro.

5.3 Câmara

Este motor 3D implementa uma câmara ao estilo FPS (*First Person Shooter*), cujas *keybinds* são:

- $\uparrow\downarrow\leftarrow\rightarrow$: modificam o ângulo da câmara, i.e., \leftarrow e \rightarrow alteram o ângulo *alpha* da câmara e \uparrow e \downarrow alteram o *beta*.
- WASD: movem-se nos eixos x e z da câmara, sendo que A e D movem-se, respetivamente, para a esquerda e direita, e W e S movem-se, respetivamente, para a frente e para trás.
- Barra de espaço/*left shift*: movem-se no eixo y da câmara, sendo a barra de espaço correspondente à movimentação para cima e *left shift* correspondente à movimentação para baixo.

Um exemplo do movimento possível da câmara é o seguinte:

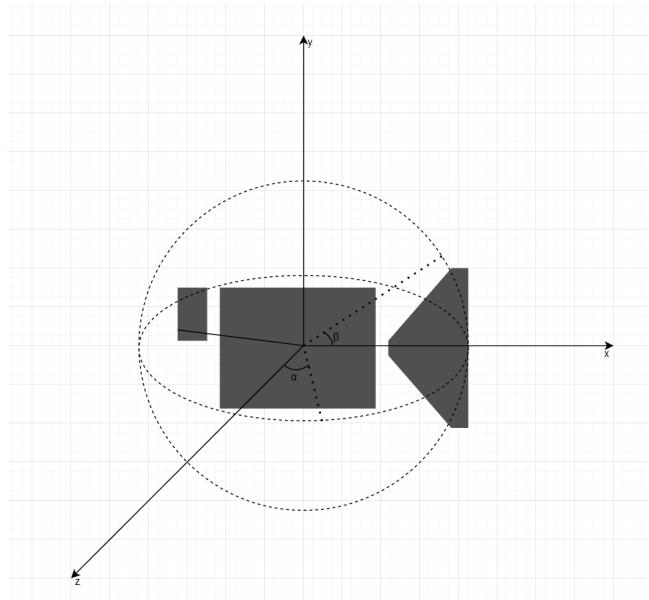


Figura 2: Movimento possível da câmara

6 Resultado Finais

Os resultados finais, da primeira parte do trabalho prático são:

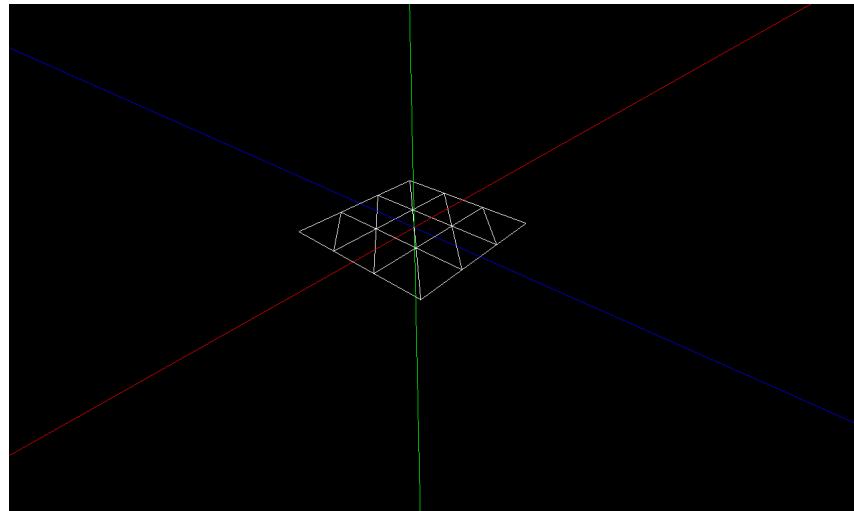


Figura 3: Plano obtido como resultado

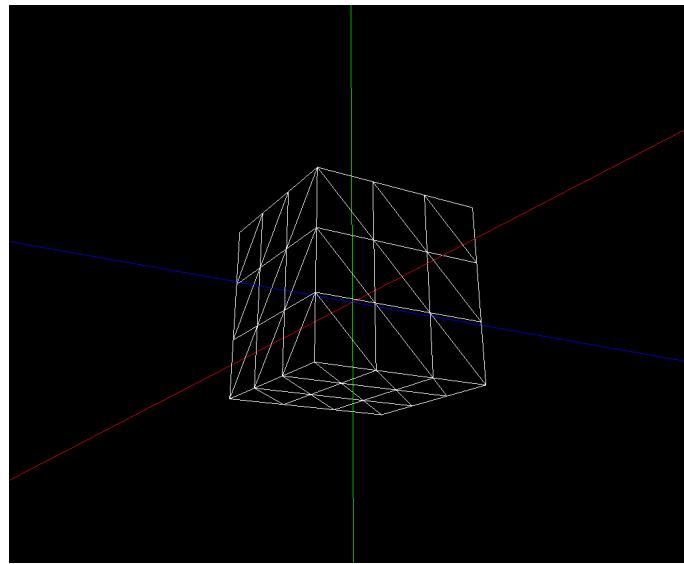


Figura 4: Box obtida como resultado

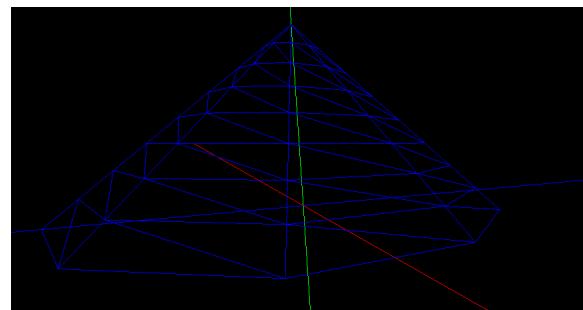


Figura 5: Cone obtido como resultado

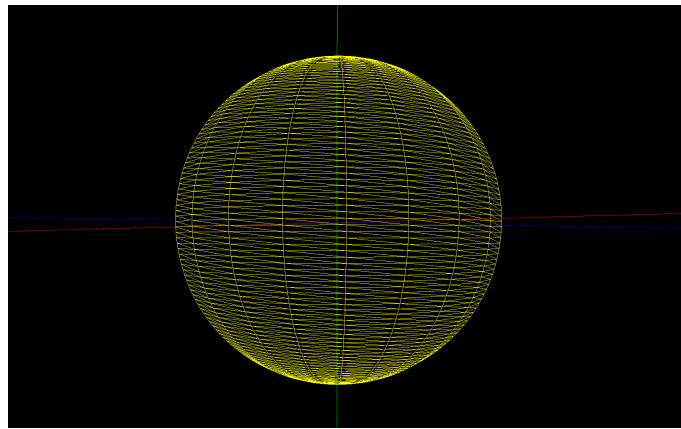


Figura 6: Esfera obtida como resultado

7 Conclusão

Como foi possível verificar por este relatório, nós nesta primeira fase do trabalho prático, criamos um gerador de ficheiros .obj que calcula os pontos necessários para criar a figura geométrica pedida pelo utilizador.

Posteriormente a isto, foi criado um motor 3D que utilizava os ficheiros gerados pelo **gerador**, e gerava as cenas, correspondentes aos dados existentes nos ficheiros .obj.

Sendo assim, conseguimos atingir todos os objetivos propostos pelo corpo docente para esta primeira parte do trabalho prático.