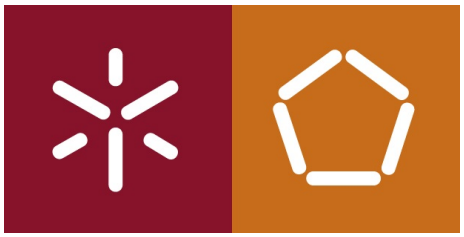
# Computação Gráfica

## Fase IV

30 de Maio de 2021

---

a85517 Duarte Oliveira
a82098 Melânia Pereira
a81195 Tiago Barata
a67645 Sérgio Gomes



Mestrado Integrado em Engenharia Informática
Universidade do Minho

# Conteúdo

# 1 Introdução

Inicia-se aqui o relatório da quarta e última fase do projeto prático da unidade curricular de Computação Gráfica no ano letivo de 2020/2021. Aqui será exposto todo o raciocínio associado à resolução do enunciado desta fase.

Nesta fase pretende-se a adição de luzes e texturas ao modelo gráfico desenvolvido nas fases anteriores. Com estas últimas adições, será possível, com a aplicação desenvolvida, desenhar qualquer cena que seja definida num ficheiro XML de configuração.

# 2 *Generator*

Para esta última fase foi necessário adicionar algumas informações aos ficheiros de pontos a gerar.

Para conseguir implementar luzes e texturas no desenho dos objetos foi necessário definir um conjunto de pontos normais e um conjunto de coordenadas de textura para cada vértice do objeto.

## 2.1 Vetores normais

Como forma de adicionar luz especular às superfícies constituintes do modelo é necessário, para cada ponto das mesmas, criar um vetor que seja normal à superfície modelada e que seja normalizado, isto é, de norma 1. Este processo é então aplicado a cada modelo fornecendo assim os meios necessários a que estes possam demonstrar reflexão. Os vetores gerados são então armazenados juntamente com os vértices que formam o respectivo modelo sendo crucial garantir que a ordem de vértices criada corresponde à ordem de vetores normais. Quando um mesmo ponto é utilizado na criação de múltiplos triângulos este deve ser considerado como sendo um novo ponto para cada um dos triângulos tendo de surgir duplicado na sequência de pontos a serem desenhados e também o vetor normal a este ponto deve surgir duplicado podendo este ser diferente para diferentes triângulos, apesar do ponto ser o mesmo.

### 2.1.1 Vetores Normais do Plano

Sendo o plano paralelo aos eixos coordenados X e Z temos que para qualquer um dos quatro pontos que o formam o plano, estes têm vetor normal como sendo o vetor normal à superfície que é colinear com o eixo Y e orientado para a porção positiva do mesmo, sendo, por isso, de coordenadas *(0, 1, 0)* como pode ser visto na seguinte figura.

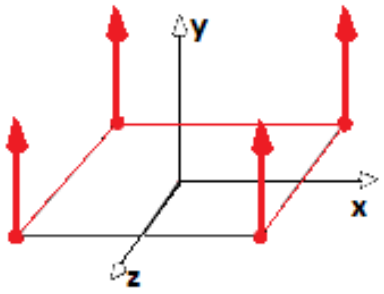



Figura 1: Plano com Vetores Normais

Na escrita do ficheiro, após a escrita das coordenadas dos 6 pontos que formam os dois triângulos do plano da-se a escrita dos 6 vetores normais, um para cada um dos vértices anteriormente referidos.

```
1  void drawPlane(float size, string fileName) {
2      float halfSize = size / 2;
3
4      ofstream file(fileName);
5      if (file.is_open()) {
6
7          file << halfSize  << " " << 0.0 << " " << halfSize  << "\n";
8          file << halfSize  << " " << 0.0 << " " << -halfSize  << "\n";
9          file << -halfSize << " " << 0.0 << " " << halfSize  << "\n";
10
11         file << halfSize  << " " << 0.0 << " " << -halfSize  << "\n";
12         file << -halfSize << " " << 0.0 << " " << -halfSize  << "\n";
13         file << -halfSize << " " << 0.0 << " " << halfSize  << "\n";
14
15         file << "---\n";
16
17         file << 0 << " " << 1 << " " << 0 << "\n";
18         file << 0 << " " << 1 << " " << 0 << "\n";
19         file << 0 << " " << 1 << " " << 0 << "\n";
20
21         file << 0 << " " << 1 << " " << 0 << "\n";
22         file << 0 << " " << 1 << " " << 0 << "\n";
23         file << 0 << " " << 1 << " " << 0 << "\n";
24
25         file << "---\n";
26
27         file << 1 << " " << 1 << "\n";
28         file << 1 << " " << 0 << "\n";
29         file << 0 << " " << 1 << "\n";
30
31         file << 1 << " " << 0 << "\n";
32         file << 0 << " " << 0 << "\n";
33         file << 0 << " " << 1 << "\n";
34
35         file << "---\n";
36     }
37     file.close();
38 }
```

### 2.1.2 Vetores Normais da Caixa

Qualquer ponto que forma uma determinada face de um paralelepípedo tem vetor normal igual ao vetor de norma unitária e perpendicular à face em questão.

Para cada vértice escrito em ficheiro é guardado um vetor normal igual ao vetor normal à face em questão sendo para as faces de cima e de baixo o vetor *(0, 1, 0)*, para as faces direita e esquerda o vetor *(1, 0, 0)* e para as faces da frente e de trás o vetor *(0, 0, 1)*. Estes vetores são guardados num array para escrita posterior.

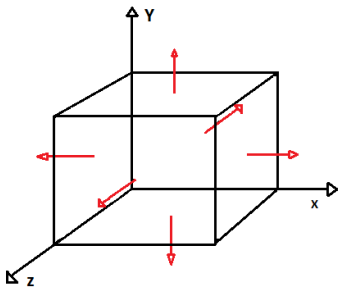



Figura 2: Cubo com Vetores Normais às Faces

```
1     file << i * xx    << " " << j * yy    << " " << z << "\n";
2     n.push_back(0);
3     n.push_back(0);
4     n.push_back(1);
5
6     file << i * xx + xx << " " << j * yy    << " " << z << "\n";
7     n.push_back(0);
8     n.push_back(0);
9     n.push_back(1);
10
11    file << i * xx + xx << " " << j * yy + yy << " " << z << "\n";
12    n.push_back(0);
13    n.push_back(0);
14    n.push_back(1);
15
16
17    file << i * xx    << " " << j * yy    << " " << z << "\n";
18    n.push_back(0);
19    n.push_back(0);
20    n.push_back(1);
21
22    file << i * xx + xx << " " << j * yy + yy << " " << z << "\n";
23    n.push_back(0);
24    n.push_back(0);
25    n.push_back(1);
26
27    file << i * xx    << " " << j * yy + yy << " " << z << "\n";
28    n.push_back(0);
29    n.push_back(0);
30    n.push_back(1);
```

### 2.1.3 Vetores Normais da Esfera

A utilização de coordenadas esféricas permite com maior facilidade a definição de vetores normais a qualquer ponto de uma esfera sendo que este pode ser calculado através dos ângulos das respetivas coordenadas esféricas desse ponto. Sendo um vetor *N = (Nx, Ny, Nz)* um vetor normal, temos que as suas componentes são dadas por

- Nx = $\sin\theta * \sin\phi$;
- Ny = $\cos\phi$;
- Nz = $\cos\theta * \sin\phi$;

Isto pode ser visualizado na seguinte figura.

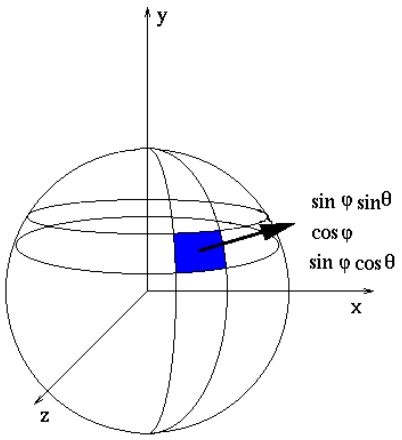



Figura 3: Esfera com Vetor Normal Genérico de um Polígono

Tomando este princípio, é possível, para cada vértice da esfera inserido no ficheiro, criar o seu respectivo vetor normal tomando os ângulos de coordenadas esféricas do ponto em questão como input, sendo o resultado adicionado a um array para escrita posterior. Segue-se um excerto da escrita de um vértice da esfera.

```
1  for (int i = 0; i < slices; i++) {
2              for (int j = 0; j < stacks; j++) {
3                  phi = j * deslocP;
4
5                  //inferior direito
6                  ax = radius * sin(theta) * sin(phi);
7                  ay = radius * cos(phi);
8                  az = radius * cos(theta) * sin(phi);
9
10                 file << ax << " " << ay << " " << az << "\n";
```

```
11                  n.push_back(sin(theta) * sin(phi));
12                  n.push_back(cos(phi));
13                  n.push_back(cos(theta) * sin(phi));
14                  t.push_back(saltoH * i);
15                  t.push_back(saltoV * j);
```

### 2.1.4 Vetores Normais do Cone

O vetor normal de um ponto constituinte da base de um cone, sendo a base do mesmo paralela ao plano XoZ, é o vetor de coordenadas X, Y e Z de valores 0, 1 e 0 respectivamente seja este qualquer ponto da base. No código, um triângulo da base é guardado em ficheiro a cada interação de um ciclo sobre o número de slices do cone a desenhar.

```
1   //base
2               file << " " << 0.0f << " " << base << " " << 0.0f << "\n";
3               n.push_back(0);
4               n.push_back(1);
5               n.push_back(0);
6               t.push_back(0.5);
7               t.push_back(0.5);
8
9               file << " " << radius * sin(alfa + deslocA) << " " << base << " " <<
                    radius * cos(alfa + deslocA) << "\n";
10              n.push_back(0);
11              n.push_back(1);
12              n.push_back(0);
13              t.push_back(0.5+0.5*sin(alfa + deslocA));
14              t.push_back(0.5+0.5*cos(alfa + deslocA));
15
16              file << " " << radius * sin(alfa) << " " << base << " " << radius * cos(
                    alfa) << "\n";
17              n.push_back(0);
18              n.push_back(1);
19              n.push_back(0);
20              t.push_back(0.5+0.5*sin(alfa));
21              t.push_back(0.5+0.5*cos(alfa));
```

Para a superfície lateral do cone é possível deduzir o vetor normal a cada um através do ângulo $\alpha$ correspondente ao ângulo de coordenadas cónicas de cada ponto. Para um ponto *P = (Px, Py, Pz)* lateral do cone, o seu vetor *N = (Nx, Ny, Nz)* normal a um ponto lateral de um cone é dado por:

- $Nx = \sin \alpha$
- $Ny = Py$

- $Nz = \cos \alpha$

Um vetor normal a um vértice lateral ao cone toma a seguinte forma.

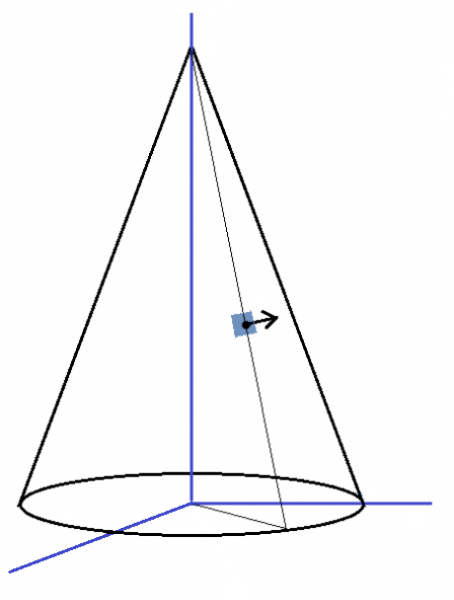


Figura 4: Cone com Vetor Normal Genérico de um Polígono

Para cada triângulo da lateral do cone três pontos são guardados em ficheiro assim como os respectivos vetores normais são armazenados num array para escrita futura.

```
1     file << " " << ax << " " << ay << " " << az << "\n";
2     n.push_back(sin(alfa));
3     n.push_back(base);
4     n.push_back(cos(alfa));
5
6     file << " " << bx << " " << by << " " << bz << "\n";
7     n.push_back(sin(alfa + deslocA));
8     n.push_back(base);
9     n.push_back(cos(alfa + deslocA));
10
11    file << " " << cx << " " << cy << " " << cz << "\n";
12    n.push_back(sin(alfa+deslocA));
13    n.push_back(base+hStack);
14    n.push_back(cos(alfa+deslocA));
```

### 2.1.5 Vetores Normais do Teapot

As normais a um patch de bezier são obtidas através de um método auxiliar que toma proveito da propriedade de produto externo de dois vetores e origina um vetor perpendicular ao mesmo para determinar o vetor normal de um ponto. Dois vetores, U e V, tangentes à curva no ponto, são calculados e através destes o vetor normal é calculado. Estes vetores tangentes são as derivadas à curva em ordem a cada uma das coordenadas.

```
1     float* normalBezier(vector<float>& patch_points, float u, float v) {
2
3     float* vetorU = new float[3];
4     float* vetorV = new float[3];
5     float* normal = new float[3];
6
7     for(int i=0;i<3;i++){
8          vetorU[i]=0.0f;
9          vetorV[i]=0.0f;
10         normal[i]=0.0f;
11    }
12
13    vetorU = derivateUBezier(patch_points, u, v,vetorU);
14    vetorV = derivateVBezier(patch_points, u, v,vetorV);
15
16    // normaliza os vetores
17    normalize(vetorU);
18    normalize(vetorV);
19
20    // calcula o produto vetorial
21    cross(vetorU, vetorV, normal);
22
23    // normaliza a normal
24    normalize(normal);
25
26    return normal;
27 }
```

## 2.2 Coordenadas de textura

As coordenadas de textura são necessárias para que o OpenGl possa mapear a imagem de textura no objeto pretendido. É necessário definir coordenadas de textura para cada um dos modelos possíveis de gerar. Depois de gerada a partir de uma imagem fornecida, os lados da textura têm comprimento 1.

De seguida apresenta-se uma breve explicação da definição destas para cada um dos modelos.

### 2.2.1 Coordenadas de textura para o plano

O plano trata-se de um simples quadrado ou retângulo, assim como o é a textura, logo, as coordenadas de textura serão simplesmente (0,0) para o canto inferior esquerdo do plano, (0,1) para o canto superior esquerdo e (1,0) e (1,1) para os cantos inferior e superior direitos, respetivamente.

### 2.2.2 Coordenadas de textura para a caixa

A caixa é um conjunto de planos organizados perpendicularmente entre si, assim, pode-se definir para cada face da caixa, as coordenadas de textura da mesma forma que foram definidas para o plano.

Mostra-se de seguida num excerto do código de uma face, a definição das coordenadas de textura para a face frontal da caixa. Lembrando apenas que o t é um vetor de texturas, que itera em dois ciclos, um i (representando o comprimento) e um j (representando a altura).

```
1   //Face da frente
2
3   t.push_back(i/l);
4   t.push_back(j/l);
5
6   t.push_back((i+1)/l);
7   t.push_back(j/l);
8
9   t.push_back((i+1)/l);
10  t.push_back((j+1)/l);
11
12  t.push_back(i/l);
13  t.push_back(j/l);
14
15  t.push_back((1+i)/l);
16  t.push_back((1+j)/l);
17
18  t.push_back(i/l);
19  t.push_back((j+1)/l);
```

### 2.2.3 Coordenadas de textura para o cone

A abordagem para calcular as coordenadas de textura foi pouco ortodoxa, mas pareceu a melhor solução. Após rever os exercícios feitos nas aulas sobre o cilindro, o grupo foi à procura de umas texturas para cones, do género do cilindo, isto é, com um plano e um esfera, para usar na base. No entanto, todos os exemplos encontrados era de planos completos. Pensou-se em 2 abordagens diferentes: Colocar a textura na lateral do cone, excluindo a base, ou então repetir a textura, na base.

Optou-se assim pela segunda abordagem, visto que o objetivo neste trabalho é aplicar uma textura completa.

Segue assim, um excerto das coordenadas de textura na base do cone, salientando que o alfa é o ângulo onde está a ser desenhada a base.

```
1      t.push_back(0.5);
2      t.push_back(0.5);
3
4      t.push_back(0.5+0.5*sin(alfa + deslocA));
5      t.push_back(0.5+0.5*cos(alfa + deslocA));
6
7      t.push_back(0.5+0.5*sin(alfa));
8      t.push_back(0.5+0.5*cos(alfa));
```

Segue ainda um excerto da abordagem do cálculo das coordenadas de textura para a lateral do cone, onde o j são as stacks do cone, e o i são as slices.

```
1      t.push_back((i*1.0)/slices);
2      t.push_back((j*1.0)/stacks);
3
4      t.push_back((i+1.0)/slices);
5      t.push_back((j*1.0)/stacks);
6
7      t.push_back((i+1.0)/slices);
8      t.push_back((j+1.0)/stacks);
9
10     t.push_back((i*1.0)/slices);
11     t.push_back((j*1.0)/stacks);
12
13     t.push_back((i+1.0)/slices);
14     t.push_back((j+1.0)/stacks);
15
16     t.push_back((i*1.0)/slices);
17     t.push_back((j+1.0)/stacks);
```

### 2.2.4 Coordenadas de textura para a esfera

Para calcular as coordenadas de textura para a esfera, começa por se definir duas deslocações, que é 1 (coordenada máxima de textura) a dividir pelo número de *stack* ou de *slices*. Após isso, e dependendo da ordem em que são desenhados os vértices, é feito o respetivo mapeamento, olhando a esfera como um planisfério.

Apresenta-se a seguir um excerto do cálculo das coordenadas de textura da esfera, onde o i é o ciclo que itera sobre as slices, e o j o ciclo que itera sobre as stacks.

```
1      t.push_back(saltoH * i);
2      t.push_back(saltoV * j);
```

```
3
4       t.push_back(saltoH * (i+1));
5       t.push_back(saltoV * j);
6
7       t.push_back(saltoH * (i+1));
8       t.push_back(saltoV * (j+1));
9
10      t.push_back(saltoH * i);
11      t.push_back(saltoV * j);
12
13      t.push_back(saltoH * (i+1));
14      t.push_back(saltoV * (j+1));
15
16      t.push_back(saltoH * i);
17      t.push_back(saltoV * (j+1));
```

### 2.2.5 Coordenadas de textura para o teapot

As coordenadas de textura do teapot, na verdade eram valores previamente calculados. Eram calculados 3 pontos com os incrementos dos ciclos, pontos esses que servem de referência para calcular as superfícies de bezier. Deste modo, esses pontos de referência, são usados como coordenadas de textura, para um plano retangular.

Segue um excerto do código que faz o cálculo desses pontos:

```
1       float x1 = increment * xx;
2       float x2 = increment * ( xx+1 );
3
4       float y1 = increment * yy;
5       float y2 = increment * ( yy+1 );
```

# 3 *Engine*

## 3.1 Composição do ficheiro XML

Para a implementação de luzes e texturas foi necessário que o ficheiro XML de configuração do desenho sofresse algumas alterações.

A primeira sendo um novo elemento `light` que define o tipo e posição da fonte de luz na cena. Outra foi a adição de um novo atributo ao elemento `model` chamado `texture` que terá como valor o *path* da imagem da textura a aplicar ao objeto em causa a ser desenhado. E, por fim, uma outra adição ao elemento `model` que é a possibilidade de definir a cor das componentes difusa, emissiva, ambiente e especular da luz, através dos atributos `diffR`, `diffG`, `diffB`, `emiR`, `emiG`, `emiB`, `ambR`, `ambG`, `ambB`, `specR`, `specG`, `specB`, respetivamente sendo cada componente descrita pelos três canais RGB.

## 3.2 Leitura do ficheiro de pontos

Devido à necessidade de ter vetores normais e coordenadas de textura para aplicar luzes e texturas aos objetos na cena, foi necessário adicionar ao ficheiro de pontos de vértices, mais estes dois conjuntos para cada objeto. Assim, a leitura do ficheiro passou a ser dividida em 3 fases: a primeira que lê todos os pontos desde o início do ficheiro até ao separador "− − −", sendo estes os pontos de cada vértice do objeto a desenhar, a segunda que lê desde o primeiro separador até ao seguinte, e onde se encontram os valores dos vetores normais a cada vértice; e finalmente, a terceira que lê desde o segundo separador até ao último, onde se encontram todas as coordenadas de textura para cada vértice.

```
1   void Objeto::readFile() {
2       (...)
3       std::ifstream infile(filename);
4       (...)
5       std::string buffer;
6       while (true) {
7           getline(infile, buffer);
8           std::stringstream ss(buffer);
9           ss >> x >> y >> z;
10          if(x == "---")
11              break;
12          points.push_back(std::atof(x.data()));
13          points.push_back(std::atof(y.data()));
14          points.push_back(std::atof(z.data()));
15      }
16      while (true) {
17          getline(infile, buffer);
18          std::stringstream ss(buffer);
19          ss >> x >> y >> z;
20          if(x == "---")
21              break;
```

```
22              normal.push_back(std::atof(x.data()));
23              normal.push_back(std::atof(y.data()));
24              normal.push_back(std::atof(z.data()));
25          }
26          while (true) {
27              getline(infile, buffer);
28              std::stringstream ss(buffer);
29              ss >> x >> y;
30              if(x == "---")
31                  break;
32              textura.push_back(std::atof(x.data()));
33              textura.push_back(std::atof(y.data()));
34          }
35          (...)
36      }
```

## 3.3 Luzes

Depois de "ligar"as luzes do OpenGL e de lido o ficheiro XML e conhecidos os valores da posição da luz e qual o seu tipo, e ainda os vetores normais para cada vértice do objeto, procede-se à definição da posição da fonte de luz, que, no caso presente, é um ponto de luz na origem do referencial.

Para iluminar os objetos com as cores pretendidas que são passadas nos ficheiro XML, é necessário que aquando do desenho do objeto se defina a cor da luz no material em cada uma das componentes com a função `glMaterialfv` e também um parâmetro de brilho. Este processo é feito na classe do objeto no momento do seu desenho.

```
1       float diff[3] = { this->diffR, this->diffG, this->diffB };
2       float spec[3] = { this->specR, this->specG, this->specB };
3       float emi[3] = { this->emiR, this->emiG, this->emiB};
4       float amb[3] = { this->ambR, this->ambG, this->ambB };
5
6       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, amb);
7       glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diff);
8       glMaterialfv(GL_FRONT_AND_BACK,GL_EMISSION, emi);
9       glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, spec);
10      glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 128);
```

## 3.4 Texturas

Para aplicar texturas aos objetos começou por se aceder ao site fornecido no enunciado para ir buscar as imagens de textura.

É necessário, numa primeira fase, fazer load da textura, recorrendo ao *DevIL*, onde será gerado um ID para a textura carregada que será usado depois para fazer o bind da mesma para esta ser desenhada.

Depois disso e com as coordenadas de textura já conhecidas para cada objeto (são lidas do ficheiro e guardadas num vetor), procedeu-se à geração de buffers que conterão as coordenadas e, depois, ao bind do buffer, já contendo os dados, assim como ao bind da textura a aplicar recorrendo ao seu ID.

Apresentamos de seguida excerto do código que descrevem este processo:

```
1   void Objeto::readFile() {
2       (...)
3       loadTexture();
4       (...)
5       glBindBuffer(GL_ARRAY_BUFFER, buffers[2]);
6       glBufferData(GL_ARRAY_BUFFER, textura.size() * sizeof(float), textura.data()
            , GL_STATIC_DRAW);
7       (...)
8   }
9
10  void Objeto::loadTexture() {
11      (...)
12      ilGenImages(1,&t);
13      ilBindImage(t);
14      ilLoadImage((ILstring)texfilename.c_str());
15      tw = ilGetInteger(IL_IMAGE_WIDTH);
16      th = ilGetInteger(IL_IMAGE_HEIGHT);
17      ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
18      texData = ilGetData();
19
20      glGenTextures(1,&texID);
21      (...)
22  }
23
24  void Objeto::draw() {
25      glBindTexture(GL_TEXTURE_2D, texID);
26      glBindBuffer(GL_ARRAY_BUFFER, buffers[2]);
27      glTexCoordPointer(2,GL_FLOAT,0,0);
28      (...)
29      glDrawArrays(GL_TRIANGLES, 0, sizeP);
30      glBindTexture(GL_TEXTURE_2D,0);
31  }
```

# 4 Notas e apreciações finais

Esta fase, apesar de parecer simples, deu bastante trabalho no cálculo das coordenadas dos vetores normais e de texturas. A implementação no *engine* foi razoavelmente simples seguindo todos os exemplos feitos nas aulas práticas da UC.

No entanto, o grupo deparou-se com alguns erros e problemas, principalmente com as texturas que não estavam a ser desenhadas corretamente, mas depois de bastantes tentativas e pesquisa conseguiu perceber as razões e, consequentemente, resolver esses erros.

O grupo está muito contente em relação ao resultado final desta fase e, sendo esta a última, também do trabalho como um todo. Apesar das dificuldades encontradas ao longo do desenvolvimento da aplicação, considera que fez um bom trabalho e que, com a ajuda do docente, conseguiu perceber alguns erros que estava a cometer e corrigi-los, levando ao sucesso completo deste trabalho prático.