

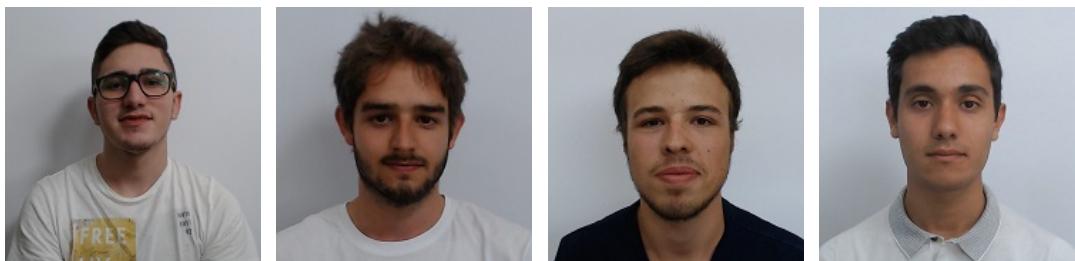
**Universidade do Minho**  
Escola de Engenharia

# Computação Gráfica

Trabalho Prático - Fase 4  
30 de Maio de 2021

## Grupo 23

Carlos Filipe Coelho Ferreira, A89542  
Joel Salgueiro Martins, A89575  
José Carlos Leite Magalhães, A85852  
Paulo Ricardo Antunes Pereira, A86475



# Conteúdo

1	Introdução . . . . .	3
2	Motor . . . . .	4
2.1	Luzes . . . . .	4
2.2	Texturas, Iluminação e Componente Material . . . . .	6
2.3	Interpretação de ficheiros XML . . . . .	9
2.3.1	Luzes e derivados . . . . .	9
2.3.2	Componente Material . . . . .	11
2.3.3	Modificações adicionais . . . . .	12
3	Gerador . . . . .	14
3.1	Normais . . . . .	14
3.1.1	Plano . . . . .	14
3.1.2	Cubo . . . . .	14
3.1.3	Esfera . . . . .	15
3.1.4	<i>Teapot</i> . . . . .	16
3.2	Texturas . . . . .	17
3.2.1	Plano . . . . .	17
3.2.2	Cubo . . . . .	17
3.2.3	Esfera . . . . .	18
3.2.4	<i>Teapot</i> . . . . .	19
4	Execução . . . . .	20
4.1	Sistema Solar . . . . .	20
5	Conclusão . . . . .	22

# 1 Introdução

No âmbito da unidade curricular de Computação Gráfica foi proposto desenvolver um motor 3D, baseado num cenário gráfico, cujo propósito passa por explorar todas as suas potencialidades e capacidades. Esta exploração é feita através de diversos exemplos visuais e interativos, sendo utilizadas, para esse efeito, todas as ferramentas abordadas nas aulas práticas da disciplina.

O projeto encontra-se dividido em quatro fases, sendo que este relatório versa sobre a última, ao qual é pedida a introdução, no nosso sistema, das funcionalidades relativas às luzes e às texturas. Esta novidade obrigou não só à implementação de novas ideias e conceitos, bem como a uma reestruturação de fases passadas.

O Motor da solução terá agora novas funcionalidades, capaz de responder a toda componente de luzes, independentemente do tipo, bem como às texturas. Ao Gerador foram adicionados mecanismos para, no momento de criar primitivas gráficas, calcular as respetivas normais e pontos de textura. Por fim, toda a estrutura envolvendo os ficheiros XML teve de ser ampliada, com novos elementos e respetivos atributos.

Mais uma vez todos estes novos requisitos serão aplicados no já complexo Sistema Solar, sendo este a prova gráfica de que tudo está a funcionar como deveria.

## 2 Motor

### 2.1 Luzes

Um dos objetivos da etapa em que nos encontramos é que todos os objetos envolventes do sistema fossem coloridos, sendo influenciados pela luz presente, estando assim representados de uma forma mais realista. As funcionalidades vistas de seguida são baseadas na biblioteca *OpenGL*, mais concretamente na componente correspondente à luz de objetos e respetivas capacidades.

Para suportar estas alterações foi necessário criar e estabelecer diferentes fontes de luzes, de modo a cobrir todos os tipos que estão disponíveis nos ficheiros XML. Para concluir este objetivo inicial, foi adicionado, à classe **Instrucao** - detalhadamente explicada nos relatórios anteriores - um novo atributo, do tipo, chamado "luz", pertencendo a uma nova classe, com o mesmo nome. Recordemos a referida classe:

```
1 class Instrucao {
2     string nome;
3     float x=0;
4     float y=0;
5     float z=0;
6     float angle=0;
7     //util para rotacao animada
8     float time=0;
9     //util para translacoes animadas
10    TranslateAnimated* tAnimated;
11    Modelo* model;
12    //novo atributo, de uma nova classe
13    Luz* luz;
14    //(...)
15 };
```

A classe **Luz**, por sua vez, foi definida da seguinte maneira:

```
1 class Luz {
2     string tipo; //tipo de luz, ("Spot,Directional ou Point")
3     GLenum id; //Id da luz
4     float* pos; //Posicao da luz
5     float* dir; //Direcao dos feixes caso seja do tipo Spot
6     float* diff, *spec, *amb; //Componente difusa, especular e
                                ambiente da luz
7     float atenuacao,cutoff, exp; //Outros dados importantes
8     //(...)
9 };
```

Assim como na classe **Modelo** ou **TranslatedAnimated**, esta classe também necessita de uma fase de preparação, feita uma única vez em cada execução. Ou seja, antes do desenho das demais figuras, é necessário, para cada luz, executar o *prepare()*.

De seguida, em cada *renderScene*, é executado a função *apply()*, tendo diversos comportamentos, consoante o tipo de luz:

- Direcional: luz que ilumina um determinado lado do modelo, com uma execução bastante simples. Neste tipo, o valor da variável **pos[3]**, referida na definição da classe, deve ser **0**;
- Point: luz que ilumina da sua posição para todas as direções. Apresenta um valor de atenuação e **pos[3]=1**;
- Spot: luz que imita o comportamento de um halofote, pelo que deve ter:
  - direção dos feixes de luz;
  - um ângulo de **cutoff**, a variar entre [0,90] ou 180 graus;
  - Um valor de **exponte** no intervalo [0,128].

Vejamos então a definição das funções anteriormente mencionadas:

```

1 class Luz {
2 //(...)
3     void prepare() {
4         glEnable(GL_LIGHT0 + this->id);
5     }
6
7     void apply() {
8         glLightfv(GL_LIGHT0 + this->id, GL_AMBIENT, amb);
9         glLightfv(GL_LIGHT0 + this->id, GL_DIFFUSE, diff);
10        glLightfv(GL_LIGHT0 + this->id, GL_SPECULAR, spec);
11        glLightfv(GL_LIGHT0 + this->id, GL_POSITION, pos);
12        if (tipo == "POINT")
13            glLightfv(GL_LIGHT0 + this->id,
14                      GL_QUADRATIC_ATTENUATION, &(this->atenuacao));
15        else if (tipo == "SPOT" ) {
16            glLightfv(GL_LIGHT0 + this->id, GL_SPOT_DIRECTION,
17                      this->dir);
18            glLightfv(GL_LIGHT0 + this->id, GL_SPOT_EXPONENT, &(
19                          this->exp));
20            glLightfv(GL_LIGHT0 + this->id, GL_SPOT_CUTOFF, &(
21                          this->cutoff));
22        }
23    }
24 //(...)
25 }
```

## 2.2 Texturas, Iluminação e Componente Material

Devido à complexidade dos requisitos desta fase, também a classe **Modelo** sofreu algumas alterações, nomeadamente o acréscimo de atributos. Estes, com fins distintos, contém informação sobre a existência de texturas, normais, bem como as características dos materiais: difusa, ambiente, especular e emissiva.

Vejamos então a nova classe **Modelo**:

```
1 class Modelo {  
2     string nome;  
3     vector<float> pontos; //posicoes  
4     vector<float> normais; //normais em cada ponto  
5     vector<float> text; //grelha da textura  
6     string textura; //nome da textura  
7     GLuint vboID;  
8     float *ambient, *diffuse; //componente ambiente e difusa  
9     float *emissive,*specular;//componente emissiva e especular  
10    float size = 0;  
11    int idTextura; //id da textura  
12 //(...)  
13 };
```

A necessidade de trabalhar com ficheiros de várias imagens, de modo a definir as texturas, levou à utilização da API **DevIL**, tal como feito nas aulas práticas. A função *loadTexture()* é encarregue de ler o objeto imagem e de devolver o ID da textura, para que esta possa, posteriormente, ser utilizado.

```
1 int loadTexture () {  
2     unsigned int t, tw, th;  
3     unsigned char* texData;  
4     unsigned int texID;  
5  
6     ilInit();  
7     ilEnable(IL_ORIGIN_SET);  
8     ilOriginFunc(IL_ORIGIN_LOWER_LEFT);  
9  
10    ilGenImages(1, &t);  
11    ilBindImage(t);  
12    string texfile = DIR + this->textura;  
13    ilLoadImage((ILstring)(texfile.c_str()));  
14    tw = ilGetInteger(IL_IMAGE_WIDTH);  
15    th = ilGetInteger(IL_IMAGE_HEIGHT);  
16    ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);  
17    texData = ilGetData();  
18  
19    glGenTextures(1, &texID);  
20  
21    glBindTexture(GL_TEXTURE_2D, texID);
```

```

22     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
23                     GL_REPEAT);
24     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
25                     GL_REPEAT);
26
27     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
28                     GL_LINEAR);
29     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
30                     GL_LINEAR_MIPMAP_LINEAR);
31
32     glBindTexture(GL_TEXTURE_2D, 0);
33     return texID;
34 }
```

De seguida, foi necessário adaptar a função *formaVbo(...)* de modo a incluir a textura, as normais (para as luzes) e as componentes materiais do objeto. O modo de pensamento foi semelhante ao da fase anterior, etapa em que se implementou o conceito de *Vertex Buffer Object*.

```

1 void formaVbo(GLuint* figuras,GLuint* normais,GLuint* texturas)
2 {
3     size = this->pontos.size() / 3;
4     float* p = new float[size * 3]; //array para os pontos
5     float* n = new float[size * 3]; //array para as normais
6     float* t = new float[size * 3]; //array para as
7         texturas
8     for (int j = 0; j < size; j++) {
9         p[i] = this->pontos.at(i);
10        (...)

11        i += 3;
12        k += 2;
13    }
14    //criar buffer das figuras na placa grafica:
15    glBindBuffer(GL_ARRAY_BUFFER, figuras[vboID]);
16    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 3,
17                 p, GL_STATIC_DRAW);

18    //criar buffer das normais na placa grafica:
19    glBindBuffer(GL_ARRAY_BUFFER, normais[vboID]);
20    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 3,
21                 n, GL_STATIC_DRAW);

22    //criar buffer das texturas na placa grafica
23    glBindBuffer(GL_ARRAY_BUFFER, texturas[vboID]);
24    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 2,
25                 t, GL_STATIC_DRAW);
```

```

23
24     if (this->textura.compare("nada") != 0) { //Se tiver
25         texturas
26             int txId=loadTexture(); //vai ler o ficheiro
27             this->idTextura = txId; //guarda o id
28     }else {
29         this->idTextura = -1;
30     }

```

Por último, também a função *desenhaPontos(...)* sofreu alterações, de modo a conseguir tratar argumentos como as normais e as texturas. Recebe dois novos argumentos, dois arrays de índices dos VBOs: um para as normais e outro para as texturas.

```

1 void desenhaPontos(GLuint* figuras,GLuint* normais,GLuint*
2      texturas) {
3      //Apontador para o buffer das figuras:
4          glBindBuffer(GL_ARRAY_BUFFER, figuras[vboID]);
5          glVertexPointer(3, GL_FLOAT, 0, 0);
6
7      //Apontador para o buffer das normais:
8          glBindBuffer(GL_ARRAY_BUFFER, normais[vboID]);
9          glNormalPointer(GL_FLOAT, 0, 0);
10
11     //Colocar as componentes do material:
12         glMaterialfv(GL_FRONT, GL_AMBIENT, this->ambient);
13         glMaterialfv(GL_FRONT, GL_DIFFUSE, this->diffuse);
14         glMaterialfv(GL_FRONT, GL_SPECULAR, this->specular);
15         glMaterialfv(GL_FRONT, GL_EMISSION, this->emissive);
16
17     if (this->textura=="nada") { //Se nao tiver textura
18         glDrawArrays(GL_TRIANGLES, 0,size);
19     }
20     else { //Caso tenha textura~
21         //Apontador para o buffer das texturas:
22             glBindBuffer(GL_ARRAY_BUFFER, texturas[vboID]);
23             glTexCoordPointer(2, GL_FLOAT, 0, 0);
24             glBindTexture(GL_TEXTURE_2D, idTextura); //
25                 buscar a textura
26             glDrawArrays(GL_TRIANGLES, 0, size); //desenha
27             glBindTexture(GL_TEXTURE_2D, 0); //limpar o
28                 apontador da textura
29     }
30     resetColor(); //"resetar" os valores do componente
31         material para defeito.
32 }

```

## 2.3 Interpretação de ficheiros XML

Com a adição de texturas, cores, luzes e componentes materiais foi necessário adaptar a nossa solução de modo a que esta consiga ler e interpretar todos estes novos campos, fornecidos pelo utilizador através de ficheiros XML.

### 2.3.1 Luzes e derivados

Assim, o ficheiro XML, no caso das luzes, terá novos elementos, cada um com os respetivos atributos, como podemos ver no seguinte exemplo, contento os três tipos de luzes:

```
1 //(...)
2 <lights>
3 //No caso de ser do tipo DIRECTIONAL:
4     <light type="DIRECTIONAL" posX="1.0f" posY="0.0f" posZ="1.0
      f" diffB="0.0f" diffG="0.0f" diffR="1.0f"/>
5
6 //No caso de ser do tipo POINT:
7     <light type="POINT" posX="0.0f" posY="0.0f" posZ="0.0f"
      ambB="1.0f" ambG="1.0f" ambR="1.0f" atenuacao="0.0f" />
8
9 //No caso de ser do tipo SPOT:
10    <light type="SPOT" posY="120.0f" dirY="-1" cutoff="60"
       specR="1.0f" specB="1.0f" specG="1.0f" expoente="1.0f"/>
11 </lights>
12 //(...)
```

Recordemos, então, a classe criada na fase passada, responsável pela leitura e interpretação de ficheiros XML, *leitorXML*, definida pelo seguinte:

```
1 class leitorXML {
2     string ficheiro;
3     Transformacao* transf;
4     //...
5 };
```

O processo de leitura e interpretação é feito utilizando várias funções, auxiliadas pela biblioteca *tinyxml*, já usada nas fases anteriores. Entre elas, a função *parseGroup(...)*, responsável por verificar o tipo dos elementos que estão a ser lidos, bem como proceder ao processamento adequado. Todos os tipos de elementos das fases anteriores são mantidos integralmente, bem como o seu tratamento.

Para esta fase foi então necessário acrescentar uma condição, que verifique se o elemento lido é do tipo *lights*. Caso seja, então é necessário interpretar devidamente os campos que se seguem e, para isso, é invocada uma nova função, *parseLights(...)*, criada para este o efeito:

```

1 void parseLights(XMLElement* lights) {
2     string tipo; float* pos,*dir,*amb,*diff,*spec;
3     float atenuacao=0.0f, cutoff=0.0f, expoente=0.0f;
4     for (XMLElement* elem = //(...)) {
5         //1.Inicializa array para guardar <atributo>: <"nomeArray">
6             // -> posicoes: "pos"
7             pos = new float[4];
8             pos[0]=0.0f; pos[1]=0.0f; pos[2]=0.0f; pos[3]=0.0f;
9             /* Analogamente:
10                 -> direcoes: "dir"
11                 -> ambiente: "amb"
12                 -> difusa: "diff" */
13
14     /*2.Verifica se os varios atributos estao presentes no XML:
15        -> Se estiverem, entao tomam esse valor;
16        -> Caso nao estejam, ficam com o valor ja definido; */
17         elem->QueryFloatAttribute("diffR", &diff[0]);
18     //Analogo para ("diffG", &diff[1]) e para ("diffB", &diff[2]);
19     elem->QueryFloatAttribute("ambR", &amb[0]);
20     //Analogo para ("ambG", &amb[1]) e para ("ambB", &amb[2]);
21     elem->QueryFloatAttribute("specR", &spec[0]);
22     //Analogo para ("specG", &spec[1]) e para ("specB", &spec[2]);
23     elem->QueryFloatAttribute("posX", &pos[0]);
24     //Analogo para ("posY", &pos[1]) e para ("posZ", &pos[2]);
25
26     //3.Verifica os tipos de luz e toma a respetiva acao:
27     tipo = elem->Attribute("type");
28     if (tipo.compare("DIRECTIONAL") == 0) {
29         (this->transf)->addLight(tipo,pos,dir,atenuacao,
30             cutoff, expoente,diff,amb,spec); }
31     else if (tipo.compare("POINT") == 0) {
32         pos[3]= 1.0f;
33     //Procura se existe atenuacao. Se nao tiver, fica o inicial
34     elem->QueryFloatAttribute("att", &atenuacao);
35         (this->transf)->addLight(...); }
36
37     else if (tipo.compare("SPOT") == 0) {
38         pos[3] = 1.0f;
39     //Procura varios atributos. Se nao existirem, ficam os iniciais
40         elem->QueryFloatAttribute("cutoff", &cutoff);
41             //Analogo para "exp","dirX", "dirY" e "dirZ"
42             (this->transf)->addLight(...); }
43 }
```

Então, analisando o código anterior, conclui-se que, independentemente do tipo, o passo seguinte será adicionar esta transformação à estrutura, utilizando a *addLight*:

```

1 void addLight(string tipo, float* posicao, float* direcao,
   float at, float cut, float expoente, float* diff, float* amb,
   float* spec) {
2     Instrucao* i = new Instrucao();
3     Luz* luz = new Luz(tipo, nLuzes, posicao, direcao, at, cut,
4                         expoente, diff, amb, spec);
5     i->luzIns(luz);
6     instrucoes.push_back(i);
7     nLuzes++;

```

Como podemos observar, o processo é análogo ao realizado nas fases interiores, à exceção da adição na nova classe, *Luz*, já descrita anteriormente.

### 2.3.2 Componente Material

No caso de estarmos perante a componente material e texturas, então terá que responder de maneira diferente. Então, o ficheiro XML referente ao elemento **models** teve que sofrer algumas alterações, nomeadamente a adição da textura e da componente material, ambos opcionais, como vemos pelos exemplos seguintes:

```

1 // ...
2 <models>
3   <model file="Sphere.3d" texture="sun.jpg" emissR="1.0f"
4         emissG="1.0f" emissB="1.0f"/>
5   <model file="teapot.3d" texture="moon.jpg" diffR="1.0f"
6         diffB="0.0f" diffG="0.0f"/>
7   <model file="Sphere.3d" diffR="0.9f" diffG="0.87f" diffB="0.5
      f" />
8 </models>
9 // ...

```

A componente material de um determinado objeto, como se pôde verificar no extracto anterior, é dividida em vários componentes: **difusa**, **ambiente**, **specular** e **emissiva**.

Acrescenta-se a informação relevante de que se uma ou mais componentes materiais não estiverem presentes no ficheiro XML, então utiliza-se valores por defeito. Caso contrário, então é necessário "recolher"esses mesmos valores. O atributo **texture**, analogamente às componentes materiais, será recolhido no caso deste estar "preenchido".

A função responsável pelo processo descrito é a seguinte:

```
1 void parseModelo(XMLElement* models) {
2 //itera ao longo do XML:
3     for (XMLElement* elem = models->FirstChildElement("model"));
4         //(...){
5             string file = elem->Attribute("file"); float
6                 controlPoint;
7             vector<float> diffuse; vector<float> ambiente;
8             vector<float> specular; vector<float> emissive;
9             //se existir o elemento difusa, entao recolhe valores
10            if (elem->FindAttribute("diffR")) {
11                diffuse.push_back(elem->FloatAttribute("diffR"));
12                diffuse.push_back(elem->FloatAttribute("diffG"));
13                diffuse.push_back(elem->FloatAttribute("diffB"));
14                diffuse.push_back(1.0f);
15            } //Caso contrario, assume valores default
16            else diffuse = { 0.8f,0.8f,0.8f,1.0f };
17
18            //Analogamente, para os restantes elementos
19            if (elem->FindAttribute("ambR")) { //...
20                else ambiente = { 0.2,0.2,0.2,1.0f };
21                if (elem->FindAttribute("specR")) { //...
22                    else specular = { 0.0f,0.0f,0.0f,1.0f };
23                    if (elem->FindAttribute("emissR")) { //...
24                        else emissive = { 0.0f,0.0f,0.0f,1.0f };
25                        const char* textura = elem->Attribute("texture");
26                        //Por fim, e adicionado a nossa estrutura, diferindo no
27                            parametro textura
28                        if (textura)
29                            (this->transf)->addModel(file,diffuse,emissive,
30                                ambiente,specular,textura);
31                        else
32                            (this->transf)->addModel(file, diffuse, emissive,
33                                ambiente, specular, "nada"); //...
34
35 }
```

### 2.3.3 Modificações adicionais

Na fase anterior, uma das funcionalidades implementadas mais relevantes era a classe **TranslatedAnimated**, encarregue de um tipo de translações que descrevem um movimento seguindo pontos gerados por curvas do tipo Catmull-Rom.

Uma das características desta animação era que, durante o movimento, o objeto tinha a capacidade de manter a sua direção, sempre alinhada com a tangente da trajetória.

Nesta fase, essa funcionalidade é opcional, sendo possível desligar, bastando adicionar, no ficheiro XML, a declaração **rotate="false"**, no elemento *translate*.

```

1 // (...)
2 <translate time="10" trace="100" rotate="false">
3     <point x="-30.0" y="0" z="0"/>
4     ...
5 </translate>
6 // (...)

```

Outra alteração adicional passou pela funcionalidade extra da fase anterior. Esta, utilizando o tipo de translação ***TranslatedAnimated***, tinha a capacidade de desenhar a rota de um determinado objeto.

Porém, ao aplicar as componentes das luzes, a primitiva *glColor3f* ficou aquém do pretendido, traduzindo-se no efeito indesejado de algumas partes das rotas ficarem mais escuras que outras.

Então, adaptou-se a função encarregue de desenhar as rotas, já definida na etapa anterior e denominada *traceCurve*:

```

1 void traceCurve(GLuint* trajetorias) {
2     if (desenhaTrajetoria != 0) { //Se for para desenhar a rota
3
4         glEnable(GL_COLOR_MATERIAL); //Ativa-se o
5             GL_COLOR_MATERIAL
6         glColor3f(1.0f, 1.0f, 1.0f); //Faz-se o glColor3f com a
7             cor branco
8         glBindBuffer(GL_ARRAY_BUFFER, trajetorias[this->indice
9             ]); //Buscar o buffer da trajetoria
10        glVertexPointer(3, GL_FLOAT, 0, 0);
11        glDrawArrays(GL_LINE_LOOP, 0, desenhaTrajetoria); //
12            Desenhar a trajetoria
13        glDisable(GL_COLOR_MATERIAL); //Desativar o
14            GL_COLOR_MATERIAL
15    }
16 }

```

### 3 Gerador

O gerador desta fase, de modo a cumprir com os requisitos da mesma, como por exemplo a introdução da iluminação, sofreu algumas alterações. Assim, além de gerar os pontos referentes às figuras pretendidas, adquire também capacidade de calcular as respetivas normais, bem como os pontos do mapeamento de texturas.

Acrescenta-se em forma de nota que, relativamente às figuras implementadas nas fases anteriores, apenas o plano teve que sofrer alterações mais profundas. Na fase 1 considerou-se que este se encontrava sobre o eixo  $xOz$  sendo que, nesta fase, foi adicionada a hipótese deste poder assente em qualquer um dos eixos:  $xOz$ ,  $xOy$  e  $yOz$ .

Refira-se que cada tipo de sólido terá a sua fórmula de cálculo e, por isso, cada um destes foi tratado de forma diferente, conforme explicado de seguida.

#### 3.1 Normais

##### 3.1.1 Plano

Para esta figura, o valor das normais varia consoante o plano em questão:

- Plano  $xOz$ , tem como vetor normal  $(0, 1, 0)$ ;
- Plano  $xOy$ , tem como vetor normal  $(0, 0, 1)$ ;
- Plano  $yOz$ , tem como vetor normal  $(1, 0, 0)$ .

##### 3.1.2 Cubo

Para esta figura geométrica a obtenção das normais é feita de forma direta, variando de face para face, como visível na Figura X. Então, supondo que a face do eixo  $yOx$  é a face frontal da caixa:

- A face frontal tem como vetor normal  $(0, 0, 1)$ ;
- A face contrária tem como vetor normal  $(0, 0, -1)$ ;
- A face lateral direita tem como vetor normal  $(1, 0, 0)$ ;
- A face lateral esquerda tem como vetor normal  $(-1, 0, 0)$ ;
- A face superior tem como vetor normal  $(0, 1, 0)$ ;
- A face inferior tem como vetor normal  $(0, -1, 0)$ .

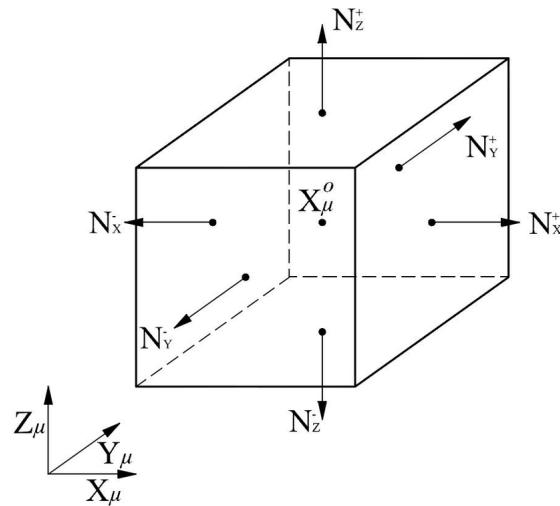


Figura 1: Representação do cubo e respetivos vetores normais

### 3.1.3 Esfera

Para a esfera, o cálculo dos vetores normais apenas passa por normalizar os seus próprios vértices. Então, cada ponto  $(x, y, z)$  da esfera, terá as suas normais calculadas da seguinte forma:

$$\vec{n} = \left( \frac{x}{raio}, \frac{y}{raio}, \frac{z}{raio} \right) \quad (1)$$

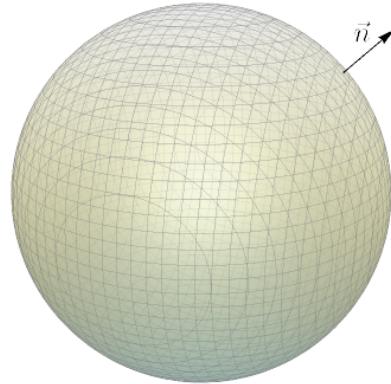


Figura 2: Representação da esfera e uma normal genérica

### 3.1.4 Teapot

Para obter as normais do *Teapot* foram utilizadas fórmulas dadas nas aulas, pelo que estas se podem aplicar a qualquer outra figura gerada por pontos de Bezier.

Acrescenta-se ainda que, devido à semelhança entre fórmulas e em todo o processo, os cálculos demonstrados de seguida são efetuados no momento em que são calculadas as coordenadas de cada ponto Bezier.

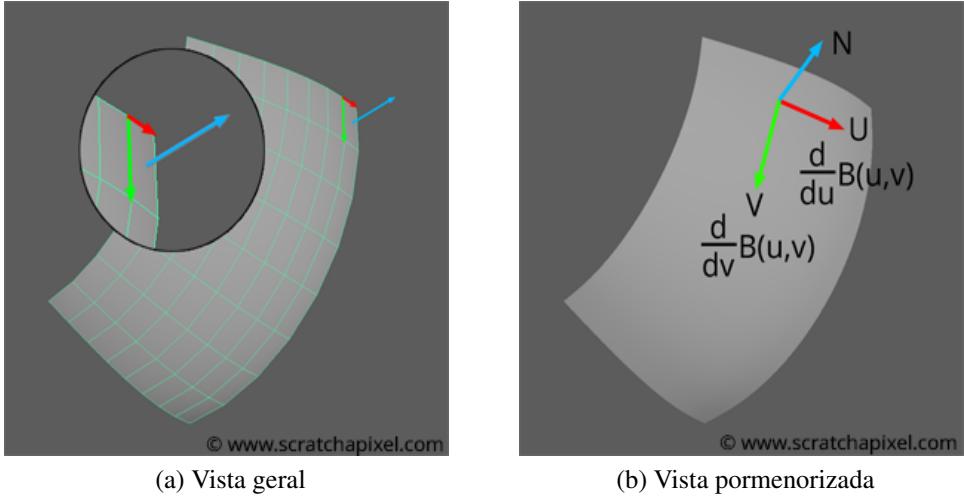


Figura 3: Representação das normais numa superfície Bezier

$$\frac{\partial p(u, v)}{\partial u} = [3u^2 \quad 2u \quad 1 \quad 0] M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T V^T \quad (2)$$

$$\frac{\partial p(u, v)}{\partial v} = U M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix} \quad (3)$$

Por fim, é necessário calcular o produto entre os dois vetores anteriormente calculados:

$$\vec{n} = \vec{v} \times \vec{u} \quad (4)$$

## 3.2 Texturas

### 3.2.1 Plano

O valor das texturas do plano é obtido de forma simples e direta, variando consoante a posição:

- "Canto" inferior esquerdo, tem como coordenadas  $(0, 0)$ ;
- "Canto" inferior direito, tem como coordenadas  $(1, 0)$ ;
- "Canto" superior esquerdo, tem como coordenadas  $(0, 1)$ ;
- "Canto" superior direito, tem como coordenadas  $(1, 1)$ .

### 3.2.2 Cubo

A textura do cubo exigiu uma pequena alteração na função responsável pelo sua formação, de forma a mapear toda a textura. Bastou, então, criar duas novas variáveis principais: **alturaAtual** e **ladoAtual**. Posteriormente, e dependendo da face em questão, estas são devidamente atualizadas.

Vejamos o exemplo seguinte. Sabe-se que a textura, de cada face, corresponde a um quadrado (a). Esse quadrado, por sua vez, corresponde a  $1/4$  da largura da imagem e a  $1/3$  da sua altura. Então se, por exemplo, tivermos em conta a face *front*, então a posição desta será, de largura,  $1/4$  a  $2/4$  e de altura, de  $1/3$  a  $2/3$ .

Por fim, em (b) vemos o resultado final.

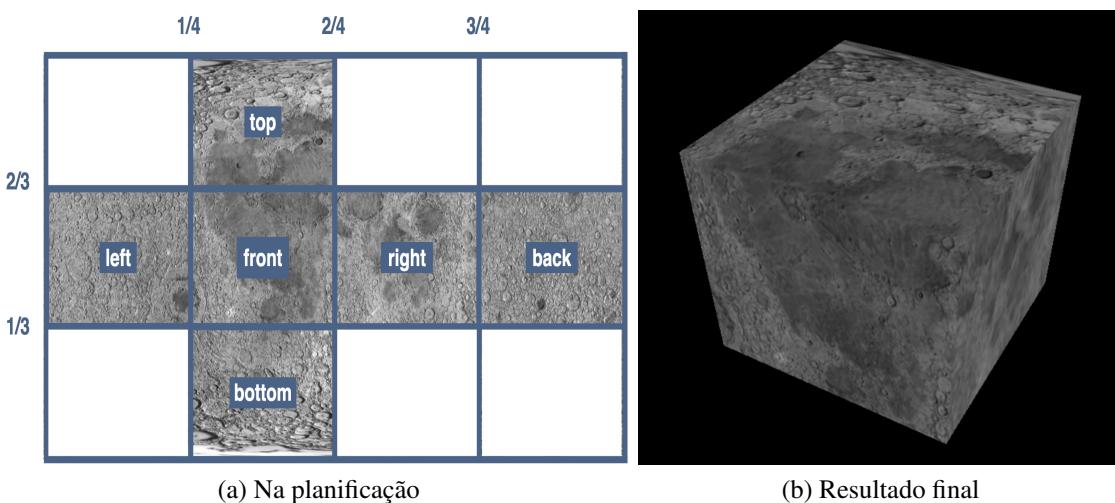


Figura 4: Textura aplicada ao Cubo

### 3.2.3 Esfera

O cálculo do mapeamento de texturas foi feito introduzindo, na nossa solução, duas novas variáveis: *cima* e *lado*. A primeira, responsável por iterar as pilhas (*stacks*), sendo a outra responsável por iterar sobre as faces (*slices*). Os seus valores de incremento são, respetivamente,  $1.0/\text{pilhas}$  e  $1.0/\text{faces}$ .

Vejamos, então, o pseudo-código responsável por implementar parágrafo interior:

```
1 void desenhaEsfera(float raio, int faces, int pilhas, char* fi) {
2     // ...
3     float cima1, cima2, lado1, lado2;
4     float cimaIncrementa = 1.0 / pilhas;
5     float ladoIncrementa = 1.0 / faces;
6
7     for (int i = 0; i < pilhas; i++) { //itera sobre pilhas
8         cima1 = cimaIncrementa * i;
9         cima2 = cimaIncrementa * (i + 1.0f);
10        // ...
11        for (int j = 0; j < faces; j++) { //itera sobre faces
12            lado1 = ladoIncrementa * j;
13            lado2 = ladoIncrementa * (j + 1.0f);
14            // ...
15        }
16    }
```

As coordenadas calculadas representam então a posição, do ponto atual, numa determinada imagem 2D referente à textura, como podemos ver represenado em (a). Em (b), por sua vez, vemos o aspetto do resultado final.

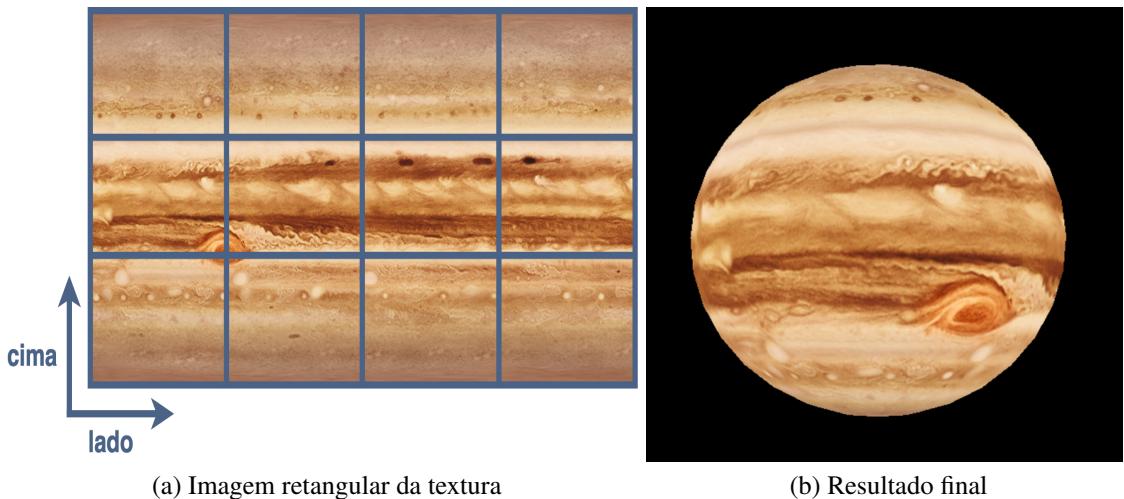


Figura 5: Textura aplicada à Esfera

### 3.2.4 Teapot

Para calcular as texturas do *teapot*, a lógica usada foi semelhante à da esfera, já explicada anteriormente, passando esta por dividir a textura pelo número de *Patches*. Por fim, basta apenas definirmos como calcular os valores da largura e altura, de modo a mapear a textura por completo.

Vejamos, para simplificar, o seguinte pseudo-código, correspondente ao momento em que são gerados os pontos Bezier:

```
1 //(...)  
2     float divText = 1.0 / dados.numPatch;  
3         for (int a = 0; a < dados.numPatch; a++) {  
4             for (int u = 0; u < tess; u++) {  
5                 for (int v = 0; v < tess; v++) {  
6                     float textura = (divText * a);  
7                     u1 = (float)u / tess;  
8                     v1 = (float)v / tess;  
9                     u2 = (float)(u + 1) / tess;  
10                    v2 = (float)(v + 1) / tess;  
11 /* altura = textura + (u1 || u2)  
12 largura = textura + (v1 || v2)  
13 0 . ---- . 1  
14 |  
15 |  
16 3 . ---- . 2  
17 Por exemplo, para o triangulo 012, teremos:  
18 (cX,cY,cZ,nX,nY,nZ, textAlt, textLarg) */  
19 (_____,_____,textura + u1, textura + v1);  
20 (_____,_____,textura + u1, textura + v2);  
21 (_____,_____,textura + u2, textura + v2);
```



Figura 6: *Teapot* - Resultado final

## 4 Execução

### 4.1 Sistema Solar

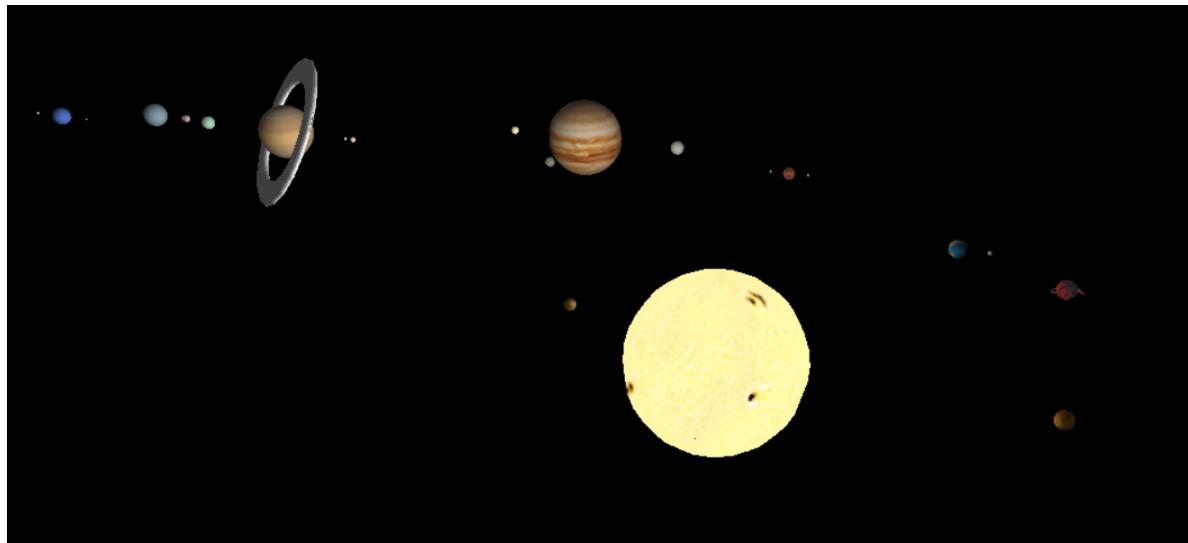


Figura 7: Sistema Solar iluminado sem trajetória gráfica desenhada

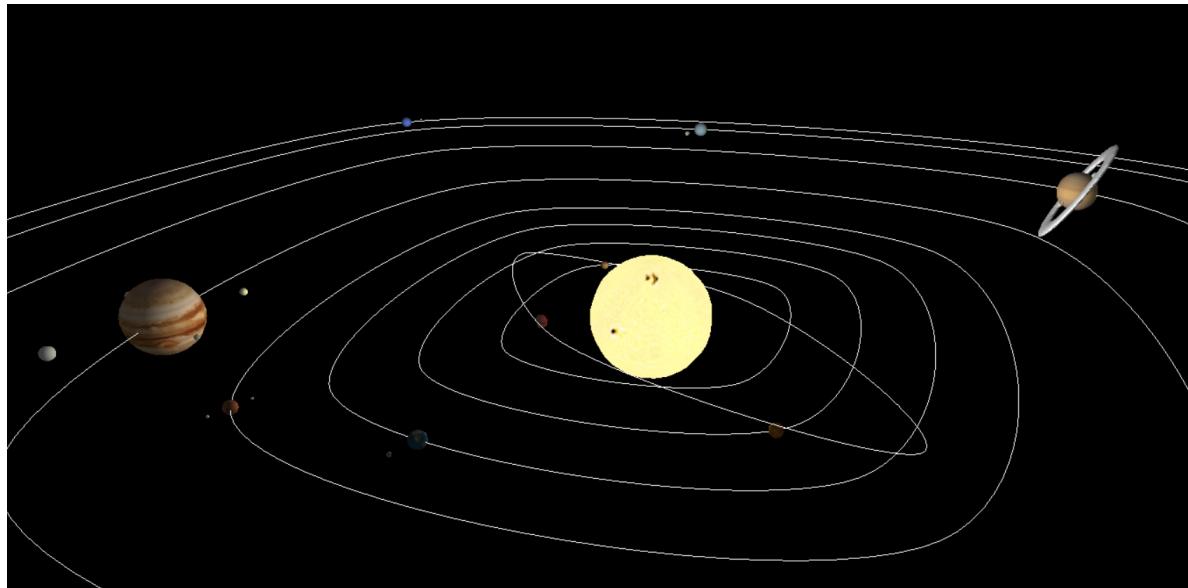


Figura 8: Sistema Solar iluminado com trajetórias representadas - 1

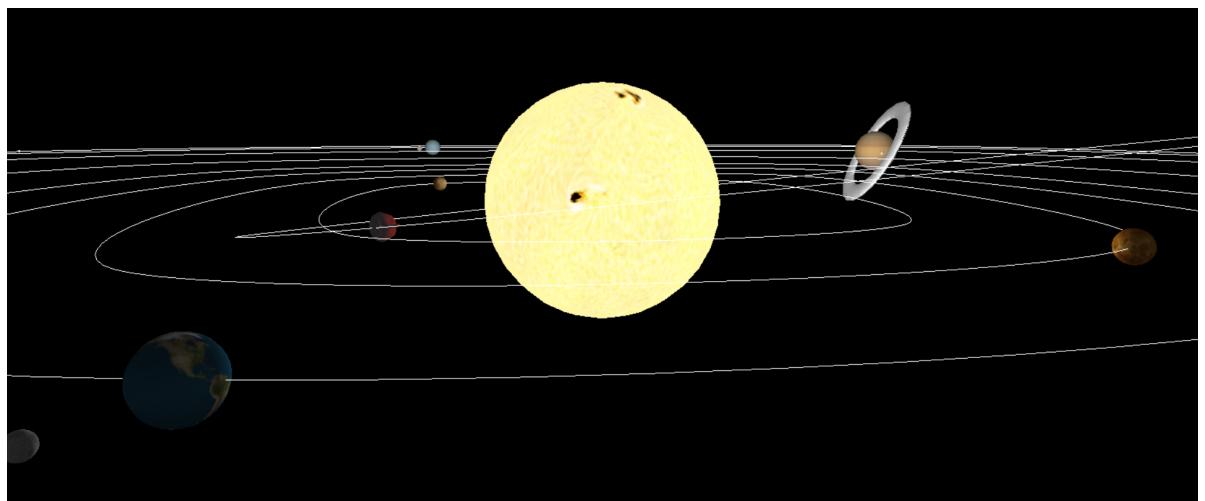


Figura 9: Sistema Solar iluminado com trajetórias representadas - 2

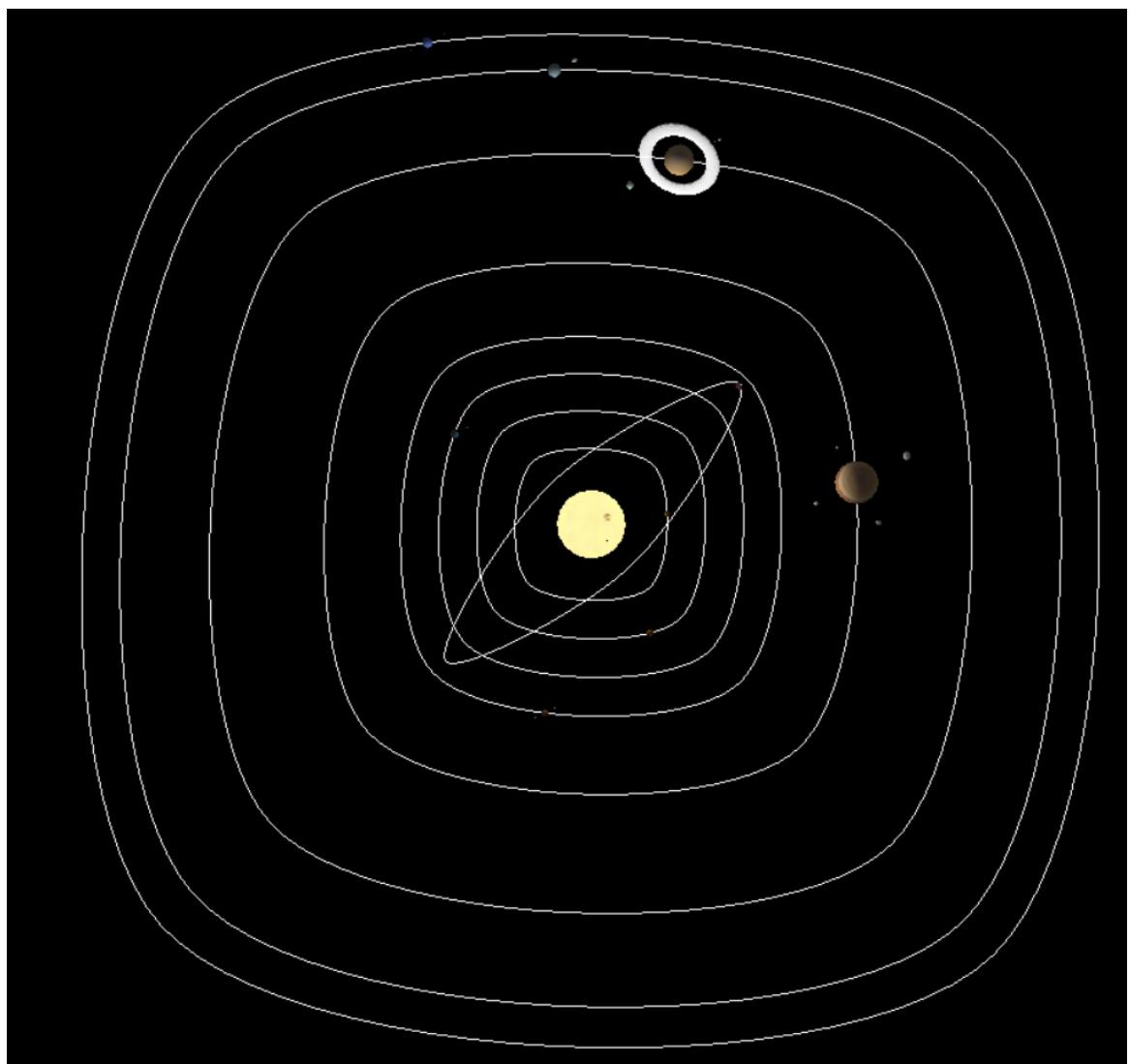


Figura 10: Sistema Solar iluminado com trajetórias representadas - 3

## 5 Conclusão

Quanto à parte pedagógica, concluímos que este projeto foi muito enriquecedor para o nosso coletivo, pois este trabalho permitiu que melhorássemos as nossas competências a nível prático relacionadas com a unidade curricular de Computação Gráfica.

Esta fase do trabalho permitiu a que chegássemos a várias conclusões, pois lidámos com vários conceitos. Por um lado, ambas as componentes introduzidas (luzes e texturas) foram satisfatórias de manipular, pois só trouxeram aspectos positivos à nossa solução, tal como o impacto visual que estas causaram. Por outro, houve um reforço de conhecimento da biblioteca *OpenGL*, um dos objetivos do projeto, tão explorada pelo grupo desde o primeiro dia.

Estando na fase final de todo o projeto, e tendo este sido fruto de um trabalho contínuo, ao longo de várias etapas, consideramos que os objetivos inicialmente traçados foram todos cumpridos. A cada transição foram sempre feitos pequenos refinamentos da fase que antecede, algo desafiante, às vezes por necessidade, outras apenas por vontade do grupo, mas ao mesmo tempo os resultados foram sempre satisfatórios.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto terminando, assim, esta viagem de quatro fases.