Computação Gráfica

Universidade do Minho

Trabalho Prático 4

Ângela Fernandes
A72053

Eduarda Santos
A78038

Leonel Gonçalves
A72305

Luís Lopes
A71016

May 8, 2019

Contents

List of Figures

Chapter 1

Introdução

Este trabalho contempla as várias fases do trabalho prático de Computação Gráfica e tem como objetivo a criação de um sistema solar. Tem por base o uso da ferramenta *OpenGL* e versa essencialmente em duas componentes, a criação de um gerador e a criação de um motor.
Ao longo deste relatório encontra-se uma descrição informal do problema, a resolução de cada fase e uma breve explicação acerca dos problemas encontrados. Irão também ser mostrados alguns excertos de código relevantes ao trabalho e imagens de forma a ser possível ver a evolução do cenário, Sistema Solar ao longo de cada fase.

1.1 Estrutura do relatório

Este relatório encontra-sa dividido em seis capítulos.
O primeiro capítulo é referente à introdução do relatório e respetiva estrutura.
O segundo capítulo aborda a análise e especificação do problema onde é feita uma breve descrição do problema e requisitos.
O terceiro capítulo resume o desenvolvimento do trabalho. Este capítulo encontra-se ainda divido em quatro subcapítulos referentes às quatro fases do trabalho respetivamente.
O quarto capítulo descreve algumas das dificuldades encontradas assim como os pontos do trabalho que ficaram por tratar.
O quinto capítulo trata a conclusão do trabalho.
O sexto capítulo é referente à bibliografia do trabalho.
No final do relatório exustem ainda quatro apêndices com excertos de código relevantes à execução do trabalho.

Chapter 2

Análise e Especificação

2.1 Descrição informal do problema e requísitos

Este trabalho prático encontra-se dividido em quatro fases.
A primeira fase tem como objetivo principal o desenvolvimento de um gerador em C++, que recebe informação de modelos a desenhar com o intuito de ser renderizada uma determinada figura, e o desenvolvimento de um motor que permitisse a leitura de um ficheiro XML de configuração que contém as figuras a desenhar propostas para esta fase, o plano, a esfera, a caixa e o cone.
O propósito da segunda fase consiste no desenvolvimento de um cenário, o Sistema Solar, recorrendo à renderização de figuras criadas na primeira fase, com especial atenção na esfera, através da leitura de um ficheiro XML.
Foi usado um modelo idêntico ao sugerido no enunciado, definindo assim os grupos hierarquicamente. Nesta fase do trabalho o gerador apenas indicava o número de triângulos a desenhar tal como as suas coordenadas correspondentes, sendo o motor o responsável pelo desenho das figuras tal como na primeira fase.
A terceira fase consiste em enriquecer o gerador de forma a que fosse capaz de produzir animações baseadas nas curvas de Catmull-Rom, tal como a criação de um novo tipo de modelo com base nas patches de Bezier.
Outro requisito resume-se à translação e rotação dos elementos, agora também conhecidos como planetas, de forma a aproximar o resultado obtido com o sistema solar.
Faz também parte dos objetivos desta fase que seja adotada uma forma de desenhar os modelos através do *Vertex Buffer Object*, em vez de serem desenhados de forma imediata, como aconteceu até esta fase. Além disto,é pedida a criação de um cometa cuja a trajetória é definida através da curva de Catmull-Rom.
A quarta e última fase do trabalho tem como objetivo criar iluminação e a texturização dos objetos. O gerador deve ser atualizado de modo a gerar as coordenadas de textura e as normais para cada vértices.
O resultado final deste trabalho prático é, portanto, a criação de um sistema solar através de uma junção de todos os pontos enunciados anteriormente.

Chapter 3

Desenvolvimento do trabalho

3.1 Fase 1

Na primeira fase do trabalho foram criados os quatro modelos pedidos, o plano, a caixa, a esfera e o cone.
Criou-se um primeiro gerador que recebe input parâmetros tais como o tipo de figura geométrica escolhida, as instruções relativas a essas figuras, e o nome do ficheiro criado como *output*. Este gerador emite um erro sempre que recebe parâmetros incorretos.
Para além disso, criou-se um motor que recebe os ficheiros gerados pelo gerador e desenha a figura pretendida.
Cada figura tem um ficheiro XML associado, pelo que só é possível desenhar uma figura de cada vez. Nas fases seguintes foi corrigido o problema de existir um ficheiro XML para cada figura e passou a existir apenas um ficheiro XML com toda a informação necessária às fases seguintes.

3.1.1 Resultado final da fase

Seguidamente é apresentada uma das figuras que é possível criar nesta fase, a esfera.

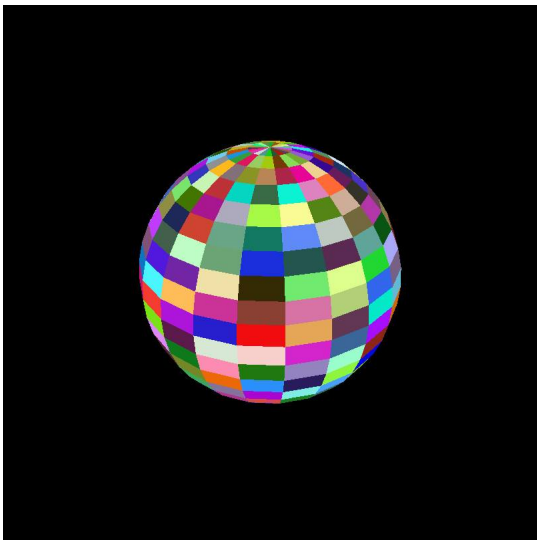

Figure 3.1: Esfera

3.2 Fase 2

3.2.1 Ficheiro XML

Nesta fase do trabalho começou-se por refazer o ficheiro XML. Este ficheiro tem como objetivo apresentar os vários planetas e os grupos hierárquicos onde estão inseridos. Seguidamente é mostrado um exemplo do ficheiro apresentado.

Exemplo do ficheiro XML

```
<group>
   Terra
    <rotate time=15 X=0 Y=1 Z=0/>
   <translate X=2.5 Y=0 Z=0 />
   <scale X=0.075 Y=0.075 Z=0.075/>
   <color R= 0.196078 G=0.6 B=0.8 />
   <models>
      <model file = "sphere_light.3d" />
   </models>

   <group>
      Lua
         <rotate time=5 X=0 Y=1 Z=0/>
         <translate X=2.7 Y=0.05 Z=0 />
         <scale X=0.4 Y=0.4 Z=0.4/>
```

```
        <color R=0.329412 G=0.329412 B=0.329412/>
      <models>
        <model file = "sphere_lightest.3d" />
      </models>
    </group>
</group>
```

No excerto de código anterior é apresentado o exemplo do planeta Terra e da Lua, onde a Lua é apresentada como um subgrupo do planeta Terra. [1]
Para a leitura do ficheiro XML é utilizada uma função denominada *parserXML* onde é feita a tradução do XML para as figuras nele mencionadas, onde existem classes para a rotação, translação e escala. Estas figuras são, na sua maioria, os planetas do sistema solar.
No apêndice A deste relatório é apresentada parte da função parserXML onde ocorre a tradução dos parâmetros recebidos do ficheiro XML para a criação da figura.

3.2.2 Classes do XML

A leitura do ficheiro XML é possível apenas quando o ficheiro tem uma estrutura específica.
É obrigatório que o ficheiro comece com uma tag denominada scene. Seguidamente, e para inicializar cada grupo, é apresentada a tag *group*. Dentro de cada grupo tems as informações respetivas ao mesmo tais como o modelo a renderizar e as transformações geométricas a executar, o grupo contém também uma informação relativa à cor do planeta. [2]
Cada elemento enunciado anteriormente tem uma classe associada no Motor onde existem as variáveis de instância e os métodos que permitem a transformação de figuras.
São criadas várias classes. A classe *Model*, que utiliza o método Apply() glVertex3f(x,y,z); a classe *Scale*, que utiliza o método Apply() glScale(x,y,z); a classe *Translate*, que utiliza o método Apply() glTranslate(x,y,z); a classe *Rotate*, que utiliza o método Apply() glRotate(angle,x,y,z) e, por último, a classe *Color*, que utiliza o método Apply() glColor(r,g,b).
No apêndice B é possível encontrar as várias classes apresentadas no Motor e enunciadas no parágrafo anterior.

[1] Este excerto de código pertence, na verdade, à fase 3. Na fase 2 o grupo não conseguiu resolver o problema das hierarquias. Por questões de simplificação do relatório e visto que o grupo resolveu este problema de hierarquias na fase 3, este excerto encontra-se neste subcapítulo e não no seguinte.
[2] Na fase 4 a cor deixa de existir quando passarem a existir texturas.

3.2.3 Resultado final da fase

Na figura abaixo é possível observar o resultado final da segunda fase do trabalho. É apresentada a primeira versão do sistema solar criada.

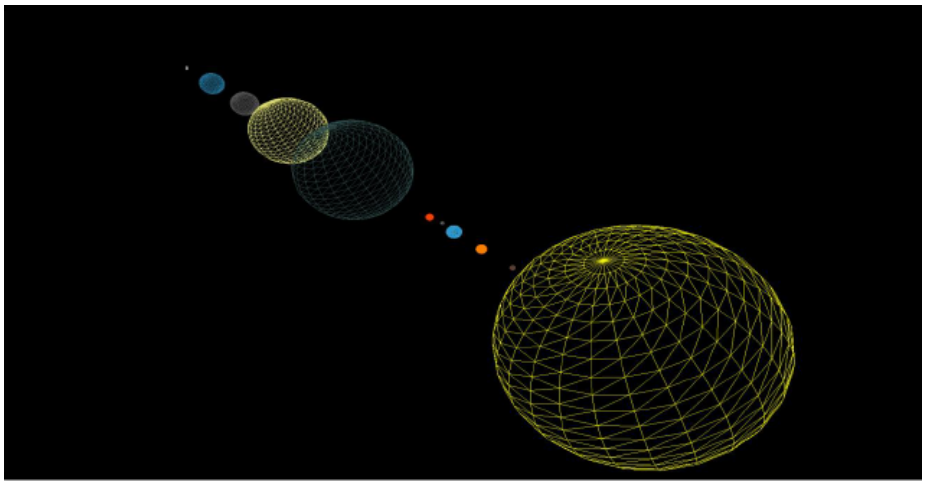

Figure 3.2: Sistema solar - Fase 2

3.3 Fase 3

Nesta fase do trabalho prático era pedida a rotação e translação de elementos. Considerando as translações, um conjunto de pontos é fornecido de forma a definir a curva cúbica de Catmull-Rom, assim como o número de segundos que demora a percorrer essa mesma curva.
Esta fase tinha como objetivo, portanto, criar animações baseadas nessas curvas. É ainda pedido que os modelos sejam desenhados com *Vertex Buffer Object*, explicados mais à frente, em vez de serem desenhados de modo imediato.

3.3.1 Vertex Buffer Object

O Vertex Buffer Object, também conhecido como VBO, permite que arrays de vértices sejam armazenados na memória gráfica de alta performance de forma a promover a transferência de dados.
Tem ainda vantagens como o facto de ser possível reduzir o número de chamadas à função e usos desnecessários de vértices compartilhados.
A extensão *GL_ARB_vertex_buffer_object* utilizada no OpenGL tem como objetivo providenciar métodos para importar informação de um vértice para um dis-

positivo de vídeo de modo a renderizar imagens de forma não imediata, havendo um aumento na performance.
Assim, o motor que está implementado na função *apply* da classe Model, pesquisa na estrutura *Map* por informações sobre a figura .3d. Se não encontrar, lerá o ficheiro .3d uma única vez e passará as suas informações para a estrutura *Map*. Seguidamente, são usadas as informações já na Map de modo a passar os vértices para o *buffer* do VBO e assim desenhar as figuras pretendidas. São usadas as funções: *glBindBuffer*, *glBufferData* e *glVertexPointer* que criam, ativam e inicializam o armazenamento de dados no buffer, tal como foram usadas nas aulas práticas.
No apêndice C deste relatório é mostrada a parte da função no Motor que implementa o VBO.

3.3.2 Curvas de Catmull-Rom

As curvas de Catmull-Rom vêm de uma família de curvas interpoladoras cúbicas onde a tangente de cada ponto de controlo é dada pelos pontos anterior e posterior ao atual.

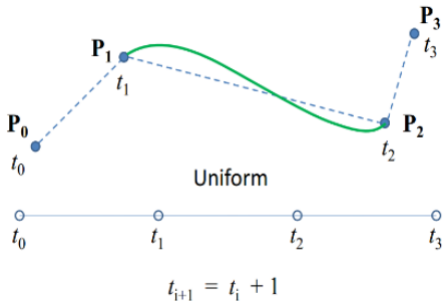

Figure 3.3: Curva de Catmull-Rom

Na imagem anterior é possível observar o cálculo da curva de P1 para P2. Tem-se que t começa a 0 em P0 e continua a aumentar ao longo dos quatro pontos de controlo. Sabe-se que as curvas de Catmull- Rom passam por todos os pontos de controlo com exceção do primeiro e último.
Tem como vantagens a continuidade garantida de forma automática e a propriedade de controlo local nos pontos de controlo.

Classe Catmull

Nesta fase do trabalho foi criada uma classe Catmull que engloba os parâmetros de translação e rotação.

No caso de ser realizada uma translação, o motor recebe um pârametro tempo, em segundos, e um *array* de números reais que correspondem aos pontos de controlo, ao entrar neste construtor será invocada a função *getGlobalCatmull-RomPoint* de forma a calcular os pontos globais da curva a percorrer. No caso da rotação, o motor recebe igualmente um tempo e coordenadas (x,y,z) para saber o eixo em que a rotação vai ser executada.
Seguidamente, o método *apply* calcula um ângulo, adiciona o ângulo ao calculado anteriormente e invoca a função *glRotate*. Sendo que este processo irá ser repetido para cada *frame per second*.

Função que calcula os pontos globais da curva

```
void getGlobalCatmullRomPoint(float gt, float *pos, float *deriv, vector<
    vector<float> > pc, int npc) {
  const int POINT_COUNT = npc;

  //float p[4][3] = { { -1,0,-1 },{ -1,0,1 },{ 1,0,1 },{ 1,0,-1 } };
  float** p = new float*[POINT_COUNT];
  for (int i = 0; i < POINT_COUNT; i++)
    p[i] = new float[3];

  for (int i = 0; i < npc; i++)
    for (int j = 0; j < 3; j++)
      p[i][j] = pc[i][j];

  // Points that make up the loop for catmull-rom interpolation
  float t = gt * POINT_COUNT;
  int index = floor(t);
  t = t - index;

  int indices[4]; //store the points
  indices[0] = (index + POINT_COUNT - 1) % POINT_COUNT;
  indices[1] = (indices[0] + 1) % POINT_COUNT;
  indices[2] = (indices[1] + 1) % POINT_COUNT;
  indices[3] = (indices[2] + 1) % POINT_COUNT;

  getCatmullRomPoint(t, p[indices[0]], p[indices[1]], p[indices[2]], p[
      indices[3]], pos, deriv);

  //free
  for (int i = 0; i < POINT_COUNT; i++)
    delete[] p[i];
  delete[] p;
}
```

No apêndice D do relatório é possível observar a classe Catmull na sua totali-

dade.

3.3.3 Eficiência na leitura de ficheiros

Apesar de já não ser um requisito desta fase, o grupo resolveu o problema dos ficheiros .3d serem lidos em cada frame.
Recorreu-se, para este efeito, a uma estrutura *Map* cuja chave é o nome do modelo a ler e o valor um *array* que armazena as linhas do ficheiro .3d.
Por cada ficheiro lido, o conteúdo do mesmo é armazenado nessa mesma estrutura de forma a poder ser lida, a cada *frame*, pelo motor. Desta forma, cada ficheiro .3d é lido somente uma vez.
Seguidamente é apresentada parte da estrutura Map utilizada pelo grupo para o efeito.

Parte da estrutura *Map* utilizada

```
//Definido como variavel global
map<string,vector<string> > modelStorage;
//Na classe Model
    int apply() {
        if (modelStorage.find(modelo) == modelStorage.end()){
            ifstream  ficheiro (modelo);
            string  s;
            vector<string> lines;
            while(getline( ficheiro ,s))
                lines.push_back(s);
            modelStorage[modelo] = lines;
            ficheiro.close();
        }
    vector<string> currentFile = modelStorage.find(modelo)->second;
```

3.3.4 Resultado final da fase

Na primeira figura pode observar-se o sistema solar visto de cima. Apesar de não ser possível mostrar o movimento na imagem, cada planeta executa um movimento de rotação e translação, para se movimentarem em torno do sol, bem como um movimento de rotação extra para se movimentarem em torno de si próprios, segundo um tempo e eixo defenidos no ficheiro XML.

Na segunda figura é possível observar novamente o sistema solar mas desta vez através de uma visão mais frontal.

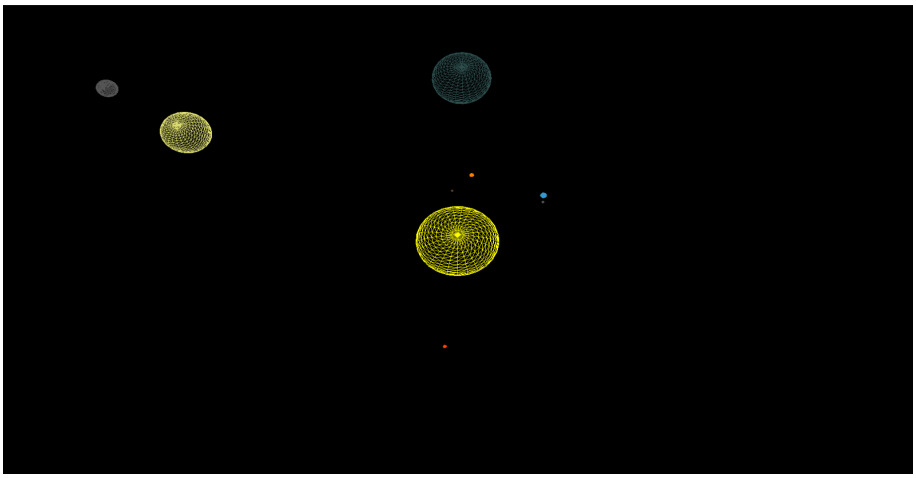

Figure 3.4: Sistema solar visto de cima

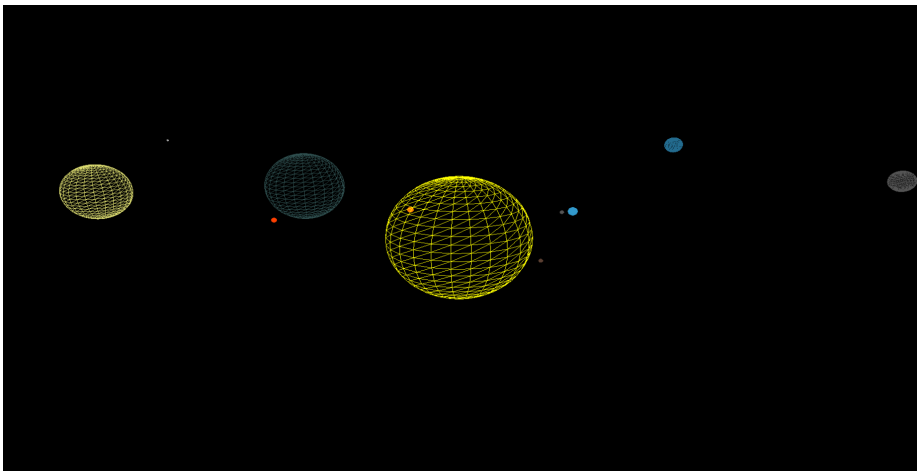

Figure 3.5: Sistema solar visto de frente

3.4 Fase 4

Nesta fase do trabalho o grupo tratou da texturização dos planetas. Até esta fase os planetas apenas possuíam uma cor fixa semelhante à cor original dos planetas do sistema solar.

3.4.1 Texturas

Para a sequência desta fase foram necessárias algumas alterações a nível do gerador, nomeadamente, ao código que renderiza as esferas. Foram adicionadas coodernadas referentes ao vetor normal e às texturas.
Tal como os VBOs, mencionados na fase 3, as texturas são objetos que precisam de ser gerados inicialmente, através da invocação de uma função. Para este trabalho são utilizadas imagens reais para decorar os planetas do sistema solar
Além disso, a fim de manter a eficiência do código, foi também criada uma estrutura *Map* para armazenar os dados sobre as texturas, a fim destas não terem que ser carregadas do disco a cada *frame*.
Seguidamente é mostrado o excerto de código do motor que serve para carregar uma textura no motor. Começa-se por abrir o ficheiro da imagem, é definida a origem do espaço da mesma no *DevIL* e, posteriormente, converte-se a imagem para RGBA.

Parte da função que serve para carregar texturas

```
int loadTexture(string s) {
    (...)
    if (textureStorage.find(s) == textureStorage.end()){
        ilInit();
        ilEnable(IL_ORIGIN_SET);
        ilOriginFunc(IL_ORIGIN_LOWER_LEFT);
        ilGenImages(1, &t);
        ilBindImage(t);
        ilLoadImage((ILstring)s.c_str());
        width = ilGetInteger(IL_IMAGE_WIDTH);
        height = ilGetInteger(IL_IMAGE_HEIGHT);
        ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
        texData = ilGetData();
        glGenTextures(1, &texID);
        glBindTexture(GL_TEXTURE_2D, texID);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
            GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
            GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
            GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_LINEAR_MIPMAP_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
            GL_RGBA, GL_UNSIGNED_BYTE, texData);
        glGenerateMipmap(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, 0);
        textureStorage[s] = texID;
```

```
    }
  return textureStorage[s];

}
```

São assim obtidas as informações necessárias, em termos de altura, comprimento, entre outros. É ainda criado um *slot* de textura e faz-se *bind*.

3.4.2 Luminosidade

O grupo adicionou apenas uma luz ambiente ao trabalho, sendo esta a única luz implementada.

Luz ambiente

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0)
```

3.4.3 Resultado final do trabalho

Seguidamente é mostrada uma imagem do resultado final com as várias componentes do trabalho.

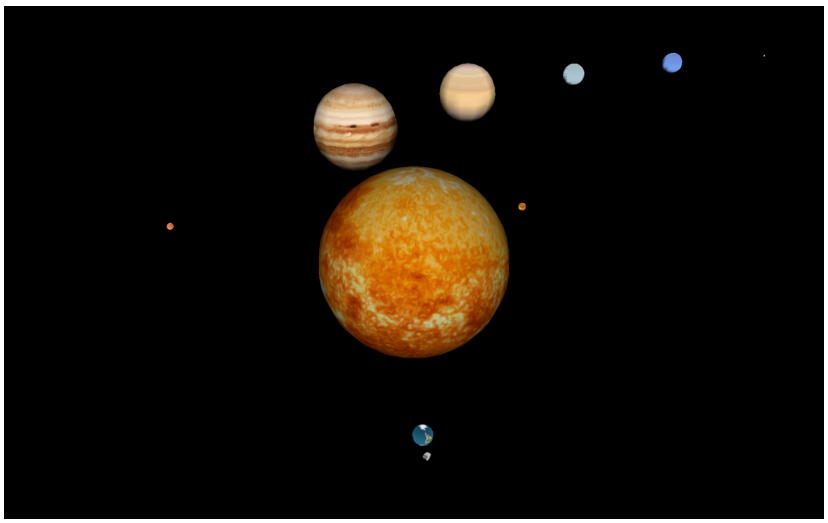

Figure 3.6: Sistema solar final

Chapter 4

Dificuldades encontradas

Ao longo das várias fases do trabalho foram corrigidos problemas das fases anteriores e melhorados alguns excertos de código, no entanto o grupo não conseguiu resolver algumas das especificaçõs pedidas.
O grupo não conseguiu implementar com sucesso os *patches* de Bezier, pelo que o gerador manteve-se, na sua maioria, inalterado. Assim sendo, não foi possível desenhar o cometa pedido na terceira fase.
Para além disso o grupo não conseguiu implementar a iluminação para além da luz ambiente na quarta fase.

Chapter 5

Conclusão

Com a elaboração deste projeto foi possível atingir com sucesso todos os objetivos definidos no início, em grande parte devido ao facto de os requisitos para cada fase que estarem bem definidos. O grupo fez um estudo aos conteúdos abordados nas aulas de forma a ser possível entender a melhor forma de proceder ao desenvolvimento do trabalho.
Com a evolução do trabalho, e em cada fase, muitas coisas foram melhoradas de forma a que o sistema solar ficasse mais eficente. Por exemplo, a implementação dos VBOs que inicialmente não estavam a ser usados mas mais tarde permitiram uma maior eficiência do trabalho.
O objetivo deste trabalho foi atingido com sucesso dado que foi possível ao grupo criar todo um cenário com os conhecimentos adquiridos nas aulas.

Chapter 6

Bibliografia

- RAMIRES, Antonio. 2019: The Geometric Pipeline. Notes of an Undergraduate Course in Computer Graphics. University of Minho.
- RAMIRES, Antonio. 2017: Curves and Surfaces. Notes of an Undergraduate Course in Computer Graphics. University of Minho.
- RAMIRES, Antonio. 2017: OpenGL Lighthing. Notes for a course in Computer Graphics. University of Minho.
- DA SILVA, André. Curvas e Superfícies. Acedido em 12 de abril de 2019, em http://www.joinville.udesc.br/portal/professores/andretavares/materiais/CGR0001_23_curvas_superficies.pdf.
- OpenGL Vertex Buffer Object (VBO). Acedido em 12 de abril de 2019, em https://www.klebermota.eti.br/2014/01/22/opengl-vertex-buffer-object-vbo/.
- What is the best way to use a HashMap in C++?. Acedido em 15 de abril de 2019, em https://stackoverflow.com/questions/3578083/what-is-the-best-way-to-use-a-hashmap-in-c.
- Read file, line by line, and store values into a array/string?. Acedido em 15 de abril de 2019, em https://stackoverflow.com/questions/30409547/read-file-line-by-line-and-store-values-into-a-array-string.
- Planet Texture Maps. Acedido em 1 de maio de 2019, em http://planetpixelemporium.com/planets.html

Appendix A

Função parserXML

Neste excerto da função ParserXML é mostrada a tradução de parâmetros do ficheiro XML em figuras.

Listing A.1: Função parserXML

```
while (elementos != NULL){
    if (strcmp(elementos->Value(), "group") == 0){
        stackgroups.push_back(elementos->FirstChildElement());
        Tree aux = new struct node;
        aux->g = new Group(idx++);
        aux->label = "group";
        aux->sons.clear();
        taux->sons.push_back(aux);
        stack_nodes_group.push_back(aux);
        cap++;
    }

    if (strcmp(elementos->Value(), "scale") == 0 || strcmp(elementos->
        Value(), "translate") == 0){
        float x, y, z;
        x = y = z = 0.0;
        if (elementos->Attribute("X")) {
            x = atof(elementos->Attribute("X"));
        }
        if (elementos->Attribute("Y")) {
            y = atof(elementos->Attribute("Y"));
        }
        if (elementos->Attribute("Z")){
            z = atof(elementos->Attribute("Z"));
        }
        if (strcmp(elementos->Value(), "scale") == 0){
            Tree aux = new struct node;
```

```
            aux->g = new Scale(x,y,z);
            aux->label = "scale";
            aux->sons.clear();
            taux->sons.push_back(aux);
        }
        else{
            Tree aux = new struct node;
            aux->g = new Translate(x, y, z);
            aux->label = "translate";
            aux->sons.clear();
            taux->sons.push_back(aux);
        }
    }
```

Appendix B

Classes do XML

Listing B.1: Classe Model

```
class Model : public Group {
    public:
        Model(string s) {
            modelo = s;
        }
        string getModelo() {
            return modelo;
        }
        // MOTOR
        int apply() {
            ifstream ficheiro (modelo);
            string s;
            getline( ficheiro ,s);
            int triangulos = atoi(s.c_str());

            for (int i = 0; i< triangulos; i++) {
                glBegin(GL_TRIANGLES);
                for (int j = 0; j < 3; j++) {
                    getline( ficheiro , s);
                    float x, y, z;
                    istringstream lineSplit (s);
                    lineSplit >> x;
                    lineSplit >> y;
                    lineSplit >> z;
                    glVertex3f(x, y, z);
                }
                glEnd();
            }
            ficheiro.close();
```

```
            return 4;
        }
    public:
        string modelo;
};
```

Listing B.2: Classe Scale

```
class Scale : public Group {
    public:
        Scale(float a, float b, float c) {
            x = a;
            y = b;
            z = c;
        }
        Scale() {
            x = 0;
            y = 0;
            z = 0;
        }
        int apply() {
            glScalef(x, y, z);
            return 2;
        }
    private:
        float x, y, z;
};
```

Listing B.3: Classe Translate

```
class Translate : public Group {
    public:
        Translate(float a, float b, float c) {
            x = a;
            y = b;
            z = c;
        }
        Translate() {
            x = 0;
            y = 0;
            z = 0;
        }
        int apply() {
            glTranslatef(x, y, z);
            return 3;
        }
```

```
    private:
        float x, y, z;
};
```

Listing B.4: Classe Rotate

```
class Rotate : public Group {
    public:
        Rotate(float l, float a, float b, float c) {
            angle = l;
            x = a;
            y = b;
            z = c;
        }
        Rotate() {
            x = 0;
            y = 0;
            z = 0;
            angle = 0;
        }
        int apply() {
            glRotatef(angle, x, y, z);
            return 1;
        }
    private:
        float x, y, z, angle;
};
```

Listing B.5: Classe Color

```
class Color : public Group {
    public:
        Color(float x, float y, float z){
            r = x;
            g = y;
            b = z;
        }
        Color(){
            r = 0;
            g = 0;
            b = 0;
        }
        int apply(){
            glColor3f(r, g, b);
            return 4;
        }
```

```
    private:
        float r, g, b;
};
```

Appendix C

Vertex Buffer Object

Listing C.1: VBO

```
//Motor VBO
    int apply() {
        if (modelStorage.find(modelo) == modelStorage.end()){
            ifstream ficheiro(modelo);
            string s;
            vector<string> lines;
            while(getline(ficheiro,s))
                lines.push_back(s);
            modelStorage[modelo] = lines;
            ficheiro.close();
        }
        vector<string> currentFile = modelStorage.find(modelo)->second;
        int vertexCount;
        position.clear();
        bool vertexCountNotRead = true;
        // Fill Buffer
        for(auto s: currentFile){
            if (vertexCountNotRead){
                vertexCount = atoi(s.c_str());
                vertexCountNotRead = false;
            } else{
                istringstream lineSplit(s);
                float x, y, z;
                lineSplit >> x;
                lineSplit >> y;
                lineSplit >> z;
                position.push_back(x);
                position.push_back(y);
                position.push_back(z);
```

```
            }
        }
        // VBO Render
        glGenBuffers(1, &buffer);
        glBindBuffer(GL_ARRAY_BUFFER, buffer);
        glBufferData(GL_ARRAY_BUFFER, position.size() * sizeof(float),
            &(position[0]), GL_STATIC_DRAW);
        glVertexPointer(3, GL_FLOAT, 0, 0);
        glDrawArrays(GL_TRIANGLES, 0, vertexCount*3);
        glDeleteBuffers(1, &buffer);
        return 4;
    }
```

Appendix D

Curvas de Catmull

Listing D.1: Classe Catmull

```
class Catmull : public Group {
    public:
        Catmull(float t, vector<float> pontos) {
            flag = 1;
            time = t;
            for (int i = 0; i < pontos.size(); i+=3) {
                vector<float> aux;
                aux.push_back(pontos[i]);
                aux.push_back(pontos[i+1]);
                aux.push_back(pontos[i+2]);
                pc.push_back(aux);
            }
            rangle = 0;
        }

        Catmull(float t, float xx, float yy, float zz) {
            xa = xx;
            ya = yy;
            za = zz;
            time = t;
            flag = 0;
            rangle = 0;
        }

        int apply() {
            static float t = 0;
            static float up[4] = { 0, 1, 0, 0 };
            float pos[4], deriv[4];
            float m[4 * 4];
```

```
            float y[4], z[4];

            if (flag) { // Time
                getGlobalCatmullRomPoint(t, pos, deriv, pc, pc.size());
                cross(deriv, up, z);
                cross(z, deriv, y);
                normalize(deriv);
                normalize(y);
                normalize(z);
                glTranslatef(pos[0], pos[1], pos[2]);
            }

            else // Rotate
                if (fps) {
                    float angaux = (360 / (fps*time));
                    rangle += angaux;
                    rangle=fmod(rangle, 360);
                    glRotatef(rangle, xa, ya, za);
                }

            up[0] = y[0]; up[1] = y[1]; up[2] = y[2]; up[3] = y[3];

            if (fps)
                t += 1 / (fps*time);

            return 2;
    }

    private:
        float time;
        int flag;
        float xa, ya, za;
        vector<vector<float> > pc;
        float rangle;
};
```