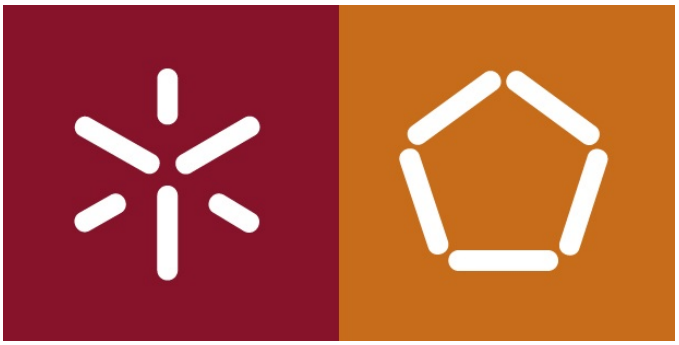
Universidade do Minho

Mestrado Integrado em Engenharia Informática



Computação Gráfica

Relatório do Trabalho Prático 3ªFase

Curves, Cubic Surfaces and VBOs

Grupo 36

Carlos Gomes
A77185

Nuno Silva
A78156

20 de Abril de 2019

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Foi nos proposto, através da unidade curricular Computação Gráfica, a criação de um mecanismo 3D baseado num cenário gráfico, mediante do qual teríamos de utilizar várias ferramentas, entre as quais C++ e OpenGl.

Para esta terceira fase terá como objetivo a inclusão de curvas e superfícies cúbicas ao trabalho anteriormente desenvolvido, tendo como finalidade a criação de um modelo dinâmico do Sistema Solar com um cometa incluído.

1.2 Resumo

Visto se tratar de uma terceira fase de um trabalho prático, é natural que se mantenha m algumas funcionalidades de fases anteriormente desenvolvidas, e por outro lado algumas delas sejam alteradas e otimizadas, de modo a cumprir com os requisitos necessários.

Assim esta fase trará consigo várias novidades que levarão a várias alterações, tanto a nível do *engine* como do *generator*.

Analisando primeiramente o *generator*, nesta fase, este passará a conseguir ser capaz de criar um novo tipo de modelo baseado no algortimo Bezier.

Posto isto analisaremos o *engine*, este sofrerá múltiplas modificações e, simultâneamente receberá novas funcionalidades. Os elementos *translate* e *rotate*, presentes no XML, sofrerão modificações. O elemento *translation* será acompanhado por um conjunto de pontos, que no seu todo, definirão uma *catmull-rom curve*, assim como o número de segundos para percorrer toda essa curva. Isto surge com objetivo de possibilitar a criação de animações baseadas nessas mesmas curvas. Mudando para o elemento *rotation*, o ângulo anteriormente definido poderá agora ser substituido pelo tempo, correspondente

este ao número de segundos que o objeto demora a comletar uma rotação de 360 graus em torno do eixo definido. Deste modo, terá que ser alterado não só *parser* responsável por ler esses mesmos ficheiros, assim como, o modo como é processada toda a informação recebida com o intuito de gerar todo o cenário pretendido.

Um última modificação implementada, e não menos importante, está relacionada com os modelos gráficos, pois estes passarão a ser desenhados com o auxílio de VBOs, ao contrário da fase anterior, no qual estes eram desenhados de forma imediata.

Tudo isto tem como finalidade conseguirmos gerar eficazmente um modelo Sistema Solar ainda mais realista do que o elaborado na fase anterior, pois este passará de um modelo estático a dinâmico após estas novas modificações, acima descritas.

Capítulo 2

Arquitetura do Código

Tendo em mente a continuação do trabalho desenvolvido na fase anterior, e de várias opções de desenvolvimento, foi decidido manter as duas aplicações (**generator** e **engine**). Sendo esta última alvo de algumas modificações mais acentuadas tendo em vista o cumprimento dos requisitos necessários. Ficou tambem decidido a alteração da estrutura do trabalho, de forma a torna-lo mais organizado.

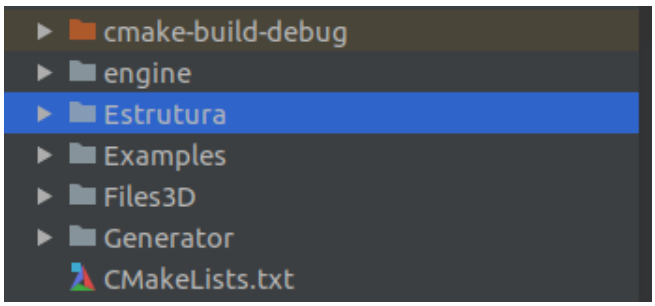


Figura 2.1: Organização do trabalho

2.1 Aplicações

Esta secção é destinada a elucidação sobre as aplicações necessárias para exibir e gerar os diversos modelos requiridos. Na realização desta fase houve alterações significativas em ambas as aplicações do projeto, de modo a que os requisitos mínimos a nós apresentados, fossem concluídos na sua totalidade.

2.1.1 Gerador

generator.cpp : Aplicação onde estão definidos os métodos que posteriormente irão gerar os respetivos vértices das diferentes formas geométricas, para que mais tarde possam ser renderizados pelo *engine*.

Para além das primitivas gráficas anteriormente desenvolvidas, com esta nova fase foi implementada um novo método de construção de modelos com base em curvas *Bézier*, obrigando assim a acrescentar novas funcionalidades ao gerador.

```
|---------------------------------------> FIGURA <-----------------------------------------|
|                                                                                           |
|                        Figuras possíveis: sphere                                          |
|                                           cone                                            |
|                                           box                                             |
|                                           plane                                           |
|                                                                                           |
|                       Como gerar:                                                         |
|                            -> box: ./generator box <largura> <altura> <comprimento> <camadas>     |
|                                (Atenção, se não quiser camadas utilize 0)                 |
|                                                                                           |
|                           -> sphere: ./generator sphere <raio> <slices> <stacks>          |
|                                                                                           |
|                           -> cone: ./generator cone <raio> <altura> <slices> <stacks>     |
|                                                                                           |
|                           -> plane: ./generator plane <lado>                              |
|                                                                                           |
|---------------------------------------> Controlos 3D <-----------------------------------|
|                                                                                           |
|                       * TRANSLAÇÃO: w, a, s, d  | W, A, S, D                                |
|                                                                                           |
|                       * ROTAÇÃO: Seta cima, baixo, esquerda, direita                      |
|                                                                                           |
|                       * ZOOM: + | -                                                       |
|                                                                                           |
|                       * REPRESENTAÇÃO DO SÓLIDO:                                          |
|                           -> por linhas: l | L                                            |
|                           -> por pontos: p | P                                            |
|                           -> preenchido: o | O                                            |
|                                                                                           |
|-------------------------------------------><-------------------------------------------|
```

Figura 2.2: Menu de ajuda do Generator

2.1.2 Motor

engine.cpp : Aplicação que possui as funcionalidades principais. Permite a apresentação dos modelos graficamente, através de uma janela.

As mudanças relativamente à estruturação do ficheiro de configuração(XML) obrigaram, a algumas modificações nos métodos de *parsing* e consequentemente no armazenamento de dados durante a leitura do respetivo ficheiro. Para além disso é também implementado o *Catmull-Rom Cubic Curves* e de *Vertex Buffer Objects*.

2.2 Classes

Para esta fase, foi apenas necessário fazer alterações nas classes previamente criadas na fase anterior. Mantendo assim a organização delas mesmas e da directoria onde se encontram, *"Estrutura"*. De seguida mostramos em quais classes fizemos alterações e explicando as mesmas.

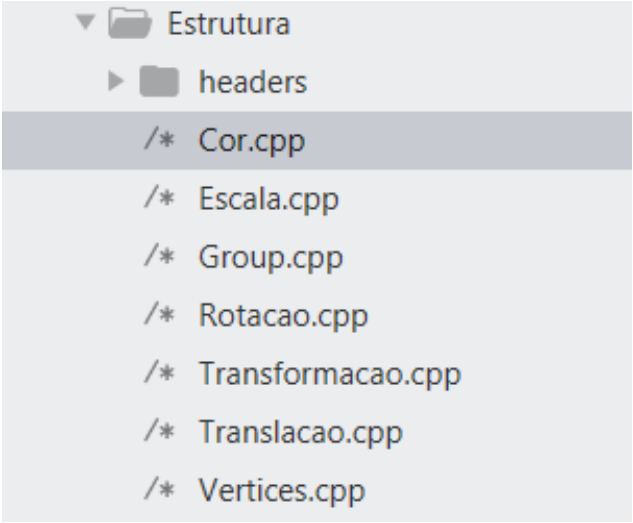


Figura 2.3: Classes contidas em "Estrutura"

2.2.1 Translação

Pretendemos que esta classe guarde toda a informação referente a uma translação a todos os pontos de controlo de uma curva e a um conjunto de pontos finais de translação, após ser calculada a curva de maneira a criar as mesmas com o método ***CatmullRom***.

```
#include "headers/Translacao.h"
#include <iostream>
#include <math.h>
Translacao::Translacao() {
    //xT = 0.0;
    //yT = 0.0;
    //zT = 0.0;
};
/*
Translacao::Translacao(float x, float y , float z){ ...
};
void Translacao::insereX(float x) {xT = x;}
void Translacao::insereY(float y) {yT = y;}
void Translacao::insereZ(float z) {zT = z;}
float Translacao::getX() {return xT;}
float Translacao::getY() {return yT;}
float Translacao::getZ() {return zT;}

*/
void Translacao::inserePonto(Vertices* ponto) {
    pontosT.push_back(ponto);
}

void Translacao::insereTempo(float t) {
    tempo = t;
}

std::vector<Vertices*> Translacao::getPontos() {
    return pontosT;
}

std::vector<Vertices*> Translacao::getCurva() {
    return pontosC;
}

float Translacao::getTempo() {
    return tempo;
}
```

```cpp
void Translacao::inserePontosT(std::vector<Vertices*> pontos) {
    pontosT = pontos;
}

void Translacao::inserePontosC(std::vector<Vertices*> pontos) {
    pontosC = pontos;
}

void Translacao::getCatmullRomPoint(float t, int* indices, float* resultado, std::vector<Vertices*> pontosT) {

    float m[4][4] = {{-0.5f, 1.5f,  -1.5f, 0.5f},
                     {1.0f,  -2.5f, 2.0f,    -0.5f},
                     {-0.5f, 0.0f,    0.5f,  0.0f},
                     {0.0f,    1.0f,    0.0f,    0.0f}};

    float quad = t*t;
    float cubo = t*t*t;
    float aux[4];

    resultado[0] = 0;
    resultado[1] = 0;
    resultado[2] = 0;

    aux[0] = (cubo * m[0][0]) + (quad * m[1][0]) + (t * m[2][0]) + (m[3][0]);
    aux[1] = (cubo * m[0][1]) + (quad * m[1][1]) + (t * m[2][1]) + (m[3][1]);
    aux[2] = (cubo * m[0][2]) + (quad * m[1][2]) + (t * m[2][2]) + (m[3][2]);
    aux[3] = (cubo * m[0][3]) + (quad * m[1][3]) + (t * m[2][3]) + (m[3][3]);

    int i1 = indices[0];
    int i2 = indices[1];
    int i3 = indices[2];
    int i4 = indices[3];


    Vertices* p1 = pontosT[i1];
    Vertices* p2 = pontosT[i2];
    Vertices* p3 = pontosT[i3];
    Vertices* p4 = pontosT[i4];
    resultado[0] = (aux[0]* p1->getX1()) + (aux[1]*p2->getX1()) + (aux[2]*p3->getX1()) + (aux[3]*p4->getX1());
    resultado[1] = (aux[0]* p1->getY1()) + (aux[1]*p2->getY1()) + (aux[2]*p3->getY1()) + (aux[3]*p4->getY1());
    resultado[2] = (aux[0]* p1->getZ1()) + (aux[1]*p2->getZ1()) + (aux[2]*p3->getZ1()) + (aux[3]*p4->getZ1());
}

void Translacao::getGlobalCatmullRomPoint(float gt, float *resultado, std::vector<Vertices *> verts) {
    int tamanho = pontosT.size();
    int* ind = new int[4];
    int i = 0;
    float t;

    t = tamanho * gt;
    i = floor(t); // Serve para truncar
    t = t-i;

    ind[0] = (i+tamanho-1)%tamanho;
    ind[1] = (ind[0]+1)%tamanho;
    ind[2] = (ind[1]+1)%tamanho;
    ind[3] = (ind[2]+1)%tamanho;

    getCatmullRomPoint(t, ind, resultado, verts);
}

std::vector<Vertices*> Translacao::curva() {
    float res[3];
    float t;
    for (t = 0; t < 1; t += 0.01) {
        getGlobalCatmullRomPoint(t, res, pontosT);
        //    std::cout << res[0] << " " << res[1] << " " << res[2] << std::endl;
        Vertices* v = new Vertices(res[0], res[1], res[2]);
        pontosC.push_back(v);
    }
    return pontosC;
}

bool Translacao::vazioT() {
    return (tempo==0 && pontosT.empty());
}
```

Assim sendo teve de ser adicionado o campo **time** no ficheiro XML.

```xml
        <!--MERCURIO-->
        <group>
            <translate Time="10">
                <point X="30" Y="0" Z="0"/>
                <point X="27.406363729278027" Y="0" Z="12.202099292274005"/>
                <point X="20.073918190765745" Y="0" Z="22.294344764321828"/>
                <point X="9.270509831248424" Y="0" Z="28.531695488854606"/>
                <point X="-3.1358538980296" Y="0" Z="29.835656861048204"/>
```

Figura 2.4: Exemplo de alteração no XML, adicionando o campo time

2.2.2 Rotação

Classe que guarda toda a informação referente a uma rotação. Sendo assim, é composta por quatro variáveis **angulo**, **x**, **y** e **z**, que identificam em qual dos eixos vai ser efetuada a rotação. Nesta fase adicionamos também uma variável **tempo** que representa, o número em segundos que demora a fazer uma rotação de 360º sobre o eixo especificado.

```
#include "headers/Rotacao.h"

Rotacao::Rotacao() {}

void Rotacao::insereAngulo(float angulo) {
    angleR = angulo;
}
void Rotacao::insereX(float x) {
    xR = x;
}
void Rotacao::insereY(float y) {
    yR = y;
}
void Rotacao::insereZ(float z) {
    zR = z;
}
void Rotacao::insereTempo(float t) {
    timeR = t;
}

float Rotacao::getAngulo() {
    return angleR;
}

float Rotacao::getTempo(){
    return timeR;
}
float Rotacao::getXR() {
    return xR;
}
float Rotacao::getYR() {
    return yR;
}
float Rotacao::getZR() {
    return zR;
}

bool Rotacao::vazioR() {
    return (timeR==0 && angleR==0 && xR==0 && yR==0 && zR==0);
}
```

2.2.3 Escala

Classe responsável por obter e aplicar as escalas correspondentes a cada modelo.

```
2   #include "headers/Escala.h"
3
4   Escala::Escala() {
5       xe = 1;
6       ye = 1;
7       ze = 1;
8   };
9
10  Escala::Escala(float x, float y, float z){
11      x = x;
12      y = y;
13      z = z;
14  };
15
16  void Escala::insereXE(float x) {
17      xe = x;
18  }
19  void Escala::insereYE(float y) {
20      ye = y;
21  }
22  void Escala::insereZE(float z) {
23      ze = z;
24  }
25
26  float Escala::getXE() {
27      return xe;
28  }
29  float Escala::getYE() {
30      return ye;
31  }
32  float Escala::getZE() {
33      return ze;
34  }
35
36  bool Escala::vazioE() {
37      return (xe==0 && ye==0 && ze==0);
38  }
39
```

2.2.4 Transformação

Classe criada com o obejtivo de aplicar todas as transformações que irão ser utilizadas num modelo geométrico.

```
#include "headers/Transformacao.h"

Transformacao::Transformacao() {
}

Transformacao::Transformacao(Translacao* t, Rotacao* r, Escala* e, Cor* c) {
    transl = t;
    rotacao = r;
    escala = e;
    cor = c;
}

void Transformacao::insereTranslacao(Translacao* t) {
    transl = t;
}
void Transformacao::insereRotacao(Rotacao* r) {
    rotacao = r;
}
void Transformacao::insereEscala (Escala* e) {
    escala = e;
}
void Transformacao::insereCor (Cor* c) {
    cor = c;
}
bool Transformacao::verificaVazio() {
    return (transl->vazioT() && rotacao->vazioR() && escala->vazioE() && cor->vazioC());
}
```

2.2.5 Group

Consiste na divisão de um ficheiro **XML** em grupos, para posteriormente a cada grupo ser aplicado transformações geométricas tais como *scale, translate* e *rotation*.

```
#include <GL/glew.h>
#include <GL/glut.h>
#include <GL/gl.h>
#include "headers/Group.h"

Group::Group(){

}

void Group::insereTransformacoes(Transformacao* t) {
    tranformacoes = t;
}

void Group::insereVerts(std::vector<Vertices*> vert) {

    vertics = vert;
}

void Group::insereFilho(Group* f) {
    filhos.push_back(f);
}

void Group::insereNome(std::string name) {
    nome = name;
}
Transformacao* Group::getTransformacoes() {
    return tranformacoes;
}

std::vector<Vertices*> Group::getVertices() {
    return vertics;
}

std::vector<Group*> Group::getFilhos() {
    return filhos;
}

std::string Group::getNome() {
    return nome;
}
```

```
void Group::desenha() {
    glBindBuffer(GL_ARRAY_BUFFER,buffer[0]);
    glVertexPointer(3,GL_FLOAT,0,0);
    glDrawArrays(GL_TRIANGLES,0,nvertices);
}

void Group::VBO() {
    glEnableClientState(GL_VERTEX_ARRAY);
    float* vert = (float*) malloc(sizeof(float)*vertics.size()*3);

    for(int i=0;i<vertics.size();i++) {

        vert[nvertices] = vertics[i]->getX1();
        vert[nvertices+1] = vertics[i]->getY1();
        vert[nvertices+2] = vertics[i]->getZ1();
        nvertices+=3;
    }

    glGenBuffers(1,buffer);
    glBindBuffer(GL_ARRAY_BUFFER,buffer[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float)*vertics.size()*3,vert,GL_STATIC_DRAW);
    free(vert);
}

void Group::setFilho(std::vector<Group*> g) {
    filhos = g;
}

void Group::insereN(int n) {
    nvertices = n;
}
```

2.3 Parsing

Para conseguir atingir o propósito final, tivemos de alterar a forma como faziamos o *parse* do ficheiro XML. Assim sendo a função parseElementos e o ficheiro *parser.cpp* foram submetidos a algumas alterações como poderemos comprovar de seguida.

```
void parserElementos(tinyxml2::XMLElement* elemento,Transformacao* transf, std::string identif) {

    Transformacao* transform = new Transformacao();
    Translacao* translacao = new Translacao();
    Rotacao* rotacao = new Rotacao();
    Escala* escala = new Escala();
    Cor* cor = new Cor();

    if((strcmp(elemento->FirstChildElement()->Value(),"group"))==0) {

        elemento = elemento->FirstChildElement();
    }

    for( tinyxml2::XMLElement* atual=elemento->FirstChildElement(); (strcmp(atual->Value(),"models"))!=0; atual = atual->NextSiblingElement()) {

        if((strcmp(atual->Value(),"translate"))==0) {

            translacao = parserTranslate(atual);
            achouT=1;
        }

        if((strcmp(atual->Value(),"scale"))==0) {

            escala = parserScale(atual);
            achouS=1;
        }

        if((strcmp(atual->Value(),"rotate"))==0) {

            rotacao = parserRotation(atual);
            achouR=1;
        }

        if((strcmp(atual->Value(),"color"))==0) {

            cor = parserColor(atual);
            achouC=1;
        }
    }
```

```
352
353     if(achouS==1) transform->insereEscala(escala);
354     else {
355         escala->insereXE(1);escala->insereYE(1);escala->insereZE(1);
356         transform->insereEscala(escala);
357     }
358     if(achouC==1) transform->insereCor(cor);
359     else {
360         cor->insereR1(1);cor->insereG1(1);cor->insereB1(1);
361         transform->insereCor(cor);
362     }
363     if(achouR==1) transform->insereRotacao(rotacao);
364     else {
365         rotacao->insereAngulo(0);rotacao->insereX(0);rotacao->insereY(0);rotacao->insereZ(0);
366         transform->insereRotacao(rotacao);
367     }
368     if(achouT==1) transform->insereTranslacao(translacao);
369     else {
370         Translacao* transl = new Translacao();
371         transform->insereTranslacao(transl);
372
373     }
374
375     achouT=achouS=achouR=achouC=0;
376
377     for(tinyxml2::XMLElement* modelo=elemento->FirstChildElement("models")->FirstChildElement("model");modelo;modelo=modelo->NextSiblingElement("model")) {
378
379         Group* grupo = new Group();
380         grupo->insereNome(modelo->Attribute("file"));
381
382         std::vector<Vertices*> verts;
383         std::string caminho3D= "../Files3D/";
384         caminho3D.append(grupo->getNome());
385
386         verts = lerFicheiro(caminho3D);
387
388         grupo->insereVerts(verts);
389         grupo->insereN(0);
390         grupo->insereTransformacoes(transform);
391
392         if(identif=="filho") {
393             int pos = groups.size() - 1;
394
395             groups[pos]->insereFilho(grupo);
396         }
397             else if(identif=="pai") {
398                 int pos = groups.size() -1;
399                 groups[pos]->insereFilho(grupo);
400             }
401             else {
402                 groups.push_back(grupo);
403             }
404         }
405
406         if (elemento->FirstChildElement("group")) {
407
408             parserElementos(elemento->FirstChildElement("group"),transform,"filho");
409         }
410
411         if ( (identif=="filho" || identif=="pai") && (elemento->NextSiblingElement("group"))) {
412             parserElementos(elemento->NextSiblingElement("group"),transf,"pai");
413         }
414
415         if ((identif!="filho" && identif!="pai") && (elemento->NextSiblingElement("group"))) {
416
417             parserElementos(elemento->NextSiblingElement("group"),transf,"irmao");
418         }
419
420     }
```


Figura 2.5: Função parseElemntos, extraída do ficheiro engine.cpp

Assim sendo durante a execução da função parseElementos, são chamadas diversas funções presentes no ficheiro *parser.cpp*, que por sua vez sofreu alterações, tal como tinha sido indicado em cima.

```cpp
#include "../Estrutura/headers/Group.h"
#include "tinyxml2.h"


int achouR = 0;
int achouS = 0;
int achouT = 0;
int achouC = 0;

// Função que recebe um string que é o caminho e vai ler o ficheiro e a medida que vai lendo o ficheiro vai meter na estrutura as coordenadas
std::vector<Vertices*> lerFicheiro(std::string caminho) {

    std::vector<Vertices*> verts;
    std::ifstream ficheiro(caminho);
    std::string linha;


    if(ficheiro.fail()) {
        std::cout << "Bip bip! não consegui encontrar o ficheiro 3D!"<< std::endl;
    }
    else {
        while(getline(ficheiro,linha)) {
            size_t pos;
            Vertices* vertice = new Vertices();


            vertice->insereX1(std::stof(linha,&pos));

            linha.erase(0,pos+1);
            vertice->insereY1(std::stof(linha,&pos));

            linha.erase(0,pos+1);
            vertice->insereZ1(std::stof(linha,&pos));

            verts.push_back(vertice);
        }
    }
    return verts;
}
```

```cpp
Rotacao* parserRotation(tinyxml2::XMLElement* elemento) {
    float angulo = 0, x = 0, y = 0, z = 0, tempo = 0;


    elemento->QueryFloatAttribute("Time",&tempo);
    elemento->QueryFloatAttribute("angle",&angulo);
    elemento->QueryFloatAttribute("axisX",&x);
    elemento->QueryFloatAttribute("axisY",&y);
    elemento->QueryFloatAttribute("axisZ",&z);

    Rotacao* rotacao = new Rotacao();
    rotacao->insereAngulo(angulo);
    rotacao->insereTempo(tempo);
    rotacao->insereX(x);
    rotacao->insereY(y);
    rotacao->insereZ(z);

    return rotacao;

}


Cor* parserColor(tinyxml2::XMLElement* elemento) {
    float r = 0, g = 0, b = 0;

    elemento->QueryFloatAttribute("R",&r);
    elemento->QueryFloatAttribute("G",&g);
    elemento->QueryFloatAttribute("B",&b);


    Cor* cor = new Cor();
    cor->insereR1(r/255);
    cor->insereG1(g/255);
    cor->insereB1(b/255);

    return cor;

}
```

```
82
83  Translacao* parserTranslate(tinyxml2::XMLElement* elemento) {
84
85      float x = 0, y = 0, z = 0, time=0;
86      Translacao* transl= new Translacao();
87      std::vector<Vertices*> pontos;
88      elemento->QueryFloatAttribute("Time",&time);
89
90
91
92      for (tinyxml2::XMLElement* aux = elemento->FirstChildElement();aux; aux = aux->NextSiblingElement()) {
93          aux->QueryFloatAttribute("X",&x);
94          aux->QueryFloatAttribute("Y",&y);
95          aux->QueryFloatAttribute("Z",&z);
96
97          Vertices* v =  new Vertices(x,y,z);
98
99          pontos.push_back(v);
100         //v->insereX1(x);v->insereY1(y);v->insereZ1(z);
101     }
102
103     transl->inserePontosT(pontos);
104     transl->insereTempo(time);
105
106
107     return transl;
108
109 }
110
```

```
111 Escala* parserScale(tinyxml2::XMLElement* elemento) {
112
113     float x = 0, y = 0, z = 0;
114
115     elemento->QueryFloatAttribute("X",&x);
116     elemento->QueryFloatAttribute("Y",&y);
117     elemento->QueryFloatAttribute("Z",&z);
118
119
120
121     Escala* escala = new Escala();
122     escala->insereXE(x);
123     escala->insereYE(y);
124     escala->insereZE(z);
125
126     return escala;
127
128 }
```

Assim sendo, é de referenciar que algumas funções acima apresentadas, tais como *getTransformacoes()*, *getRotacao()*, *getEscala()*, *getTranslacao()*, *getCor()* e *getTempo()* serão utilizadas na função *renderScene* presente no ficheiro *engine.cpp*, de forma a apresentar da melhor forma os planetas.

Capítulo 3

Gerador

3.1 Bézier

3.1.1 Processamento do ficheiro input

Antes de processar o ficheiro de input *(.patch)*, tivemos que perceber o formato do ficheiro para posteriormente gerar a figura. Após uma análise ao ficheiro, concluímos que não representa uma grande complexidade:

- Na primeira linha surge o número de *patches* (**numeroPatch**);
- As próximas linhas que são (**numeroPatch**), têm cada uma, 16 números inteiros que correspondem aos indíces de cada um dos pontos de controlo que fazem parte desse *patch*;
- Posteriormente aparece um inteiro que representa o número de pontos de controlo (**controlPontos**);
- Por fim surgem os pontos de controlo;

3.1.2 Estruturas de Dados

Após termos analisado a estrutura do ficheiro de input, decidimos guardar toda a informação fornecida num *array de arrays*.

Assim sendo guardamos os 16 números inteiros correspondentes a um determinado *patch* num *array de arrays* de inteiros, *indice[i][j]*, onde i representa o indíce do patch em questão, varia entre $0 \leq \text{i} <$ **numeroPatch**. Por outro lado, j varia entre $0 \leq \text{j} < 16$.

De forma a guardar os pontos de controlo, utilizamos a mesma técnica. Foi criado um *array de arrays* capaz de armazenar *floats*, *pontos[i][j]*, onde i representa o índice do

ponto de controlo que varia entre $0 \leq i < \textbf{controlPontos}$. Já que j, irá representar as várias coordenadas X, Y e Z dos pontos, ou seja $0 \leq j < 3$.

Desta forma de armazenamento conseguimos prosseguir para a fase em que iremos formalizar um processamento de todos os dados.

3.1.3 Processamento dos *patches*

Antes de passarmos à explicação do algoritmo que traduz o ficheiro de input num modelo geométrico, temos primeiro que entender como funcionam as curvas *Bézier*.

Para a criação de uma curva, é necessário 4 pontos, definidos num espaço 3D, ou seja, constituídos por três coordenadas: **X**, **Y** e **Z**.

Esse pontos só por si, não geram uma curva, para processar a curva será preciso combina-los com alguns coeficientes.

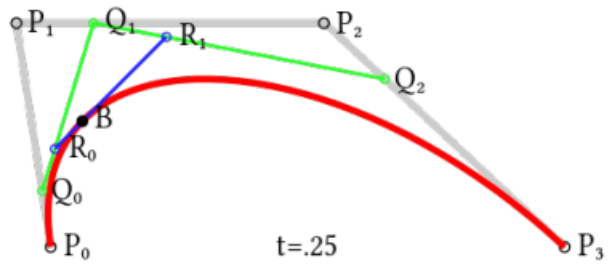


Figura 3.1: Exemplo de uma curva Bézier

Sendo esta curva uma equação, existe uma variável t, que varia entre 0 e 1. O resultado da equação de qualquer t corresponde a uma determinada posição na curva. De seguida apresenta-mos a fórmula para obter uma curva.

$$B(t) = (1-t)^3 * P0 + 3 * t * (1-t)^2 * P1 + 3 * t^2 * (1-t) * P2 + t^3 * P3$$

Ao qual nós a interpretamos da seguinte maneira ao aplicar no nosso projeto.

```
float* formula (float t, float* p1, float* p2, float* p3, float* p4) {

    float aux = 1.0 - t;
    float *result = new float[3];

    float ponto1 = aux*aux*aux;
    float ponto2 = 3*(aux*aux)*t;
    float ponto3 = 3*aux*(t*t);
    float ponto4 = t*t*t;

    result[0] = (ponto1 * p1[0]) + (ponto2 * p2[0]) + (ponto3 * p3[0]) + (ponto4 * p4[0]);
    result[1] = (ponto1 * p1[1]) + (ponto2 * p2[1]) + (ponto3 * p3[1]) + (ponto4 * p4[1]);
    result[2] = (ponto1 * p1[2]) + (ponto2 * p2[2]) + (ponto3 * p3[2]) + (ponto4 * p4[2]);

    return result;
}
```

De notar que **P0**, **P1**, **P2** e **P3** são pontos de controlo. Resolvendo a equação várias vezes substituindo t, por vários valores entre 0 e 1 conseguimos descobrir toda a curva. A partir deste momento, já estamos capazes de desenvolver o algoritmo que apresenta um princípio parecido com aquele que é utilizado nas curvas de Bézier, só que em vez de termos 4 pontos de controlo teremos 16.

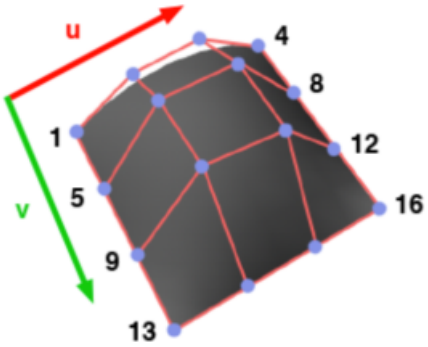


Figura 3.2: Bézier Patch e os seus pontos de contrlo

Para conseguirmos uma figura muito semelhante a esta apresentada, ainda à instantes desenvolve-mos um algoritmo ao qual reaproveitamos os conhecimentos do algortimo que produz uma curva Bézier mas desta vez aplicamo-lo em "larga escala".

```
float* bezier(float a, float b,float** pontos, int* indice) {

    float* ponto = new float[3];
    float  alt[4][3];
    float  res[4][3];

    int i,j,x;
    i=j=x=0;
    float *resultado;

    for(i=0; i<16; i++) {
        alt[j][0] = pontos[indice[i]][0];
        alt[j][1] = pontos[indice[i]][1];
        alt[j][2] = pontos[indice[i]][2];
        j++;

        if(j%4==0) {
            ponto = formula(a,alt[0],alt[1],alt[2],alt[3]);
            res[x][0] = ponto[0];
            res[x][1] = ponto[1];
            res[x][2] = ponto[2];
            x++;
            j=0;
        }
    }

    resultado = formula(b,res[0],res[1],res[2],res[3]);

    return resultado;
}
```

Capítulo 4

Motor

4.1 VBOs

Tal como já foi referido anteriormente, uma das mudanças feitas no nosso trabalho foi implementar os VBOs para desenhar os diferentes. Os *Virtual Buffer Objects* são uma funcionalidade oferecida pelo OpenGL, oas quais nos permitem inserir informação sobre os vértices diretamente na placa gráfica do nosso dispositivo. Assim é fornecida uma performance muito melhor, devido ao facto de a renderização ser feita de imediato pois a informação já se encontra na placa gráfica em vez de no sistema, diminuindo assim a carga de trabalho no processador e assim os pontos em vez de serem desenhados um a um, passam a estar todos guardados num *buffer*.

- **setVBO()** - método que após criar e preencher um array com os pontos para desenhar os triângulos, preenche um buffer de forma ordenada com os pontos do array previamente criado.
- **desenha()** - método que desenha os triângulos com os pontos guardados num *buffer*

4.2 Curva Catmull-Rom

4.2.1 Rotação

Para podermos implementar as novas formas de rotação, foi necessário acrescentar uma variável **time** para podermos calcular o tempo que demora a fazer uma rotação 360º. Para isto, aplicamos as seguintes formulas para aplicar a rotação ao objeto:

```
float r1 = glutGet(GLUT_ELAPSED_TIME) % (int) (transl->getTempo()*1000);
float r2 = (r1*360) / (transl->getTempo()*1000);
glRotatef(r2,rotacao->getXR(),rotacao->getYR(),rotacao->getZR());
```

Usamos a função glutGet na renderScene para determinarmos o tempo decorrido na execução do programa, mas, como este vai cada vez mais aumentando ao longo do tempo precisamos de estabelecer um limite para este valor, sendo que, usamos o resto da divisão pelo tempo dado e multiplicamos por 1000(tempo em milisegundos). Conseguimos assim de seguida determinar a amplitude que o objeto vai rodar no momento simplesmente dividindo o valor calculado anteriormente multiplicado por 360(graus) por o tempo multiplicado por 1000. Desta forma, o valor tempo é aplicado à função glRotatef fazendo o objeto rodar durante uma porção de tempo.

4.2.2 Translação

Além da nova forma de representar uma rotação, também houve a necessidade de criarmos uma nova representação para a translação. Para esse efeito, criamos uma nova variável tempo e 2 arrays pontosC e pontosT onde serão preenchidos em conjunto com a variável tempo através do parser do ficheiro XML. O valor do tempo vai ser calculado da seguinte maneira:

```
float t1 = glutGet(GLUT_ELAPSED_TIME) % (int) (transl->getTempo()*1000);
float t2 = t1 / (transl->getTempo()*1000);
```

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix},$$

$$T' = \begin{bmatrix} 3*t^2 & 2*t & 1 & 0 \end{bmatrix},$$

Assim sendo um dos arrays será preenchido com auxilio do parser (pontosT) e o outro será preenchido com o auxilio da função **getCatmullRomPoint**, pontosC, que obtem os valores para os mesmos através da multiplicação de uma matriz **M** por dois vetores e pontos.

4.2.3 Desenhar as curvas Catmull-Rom

Para a implementação desta secção foi criada uma nova função denominada encurvar que, com o resultado da getGlobalCatmullRomPoint gera os pon tos da curva a partir dos pontos dados no ficheiro XML. O resultado da auxiliar perminte-nos obter as coordenadas do próximo ponto da curva para um dado valor t. Por fim, é utilizada a função renderCatmullRomCurve que desenha a curva pretendida com a respetiva cor.

Capítulo 5

Análise de Resultados - Sistema Solar

O resultado final correspondeu as expectativas apesar do problema com o anel saturno, ao qual ocorria-nos consecutivamente erros e que por sua vez metia em causa a integralidade do trabalho, sendo assim resolve-mos retira-lo comprementendo-nos implementa-lo na próxima fase. Assim sendo, deixa-mos algumas imagens do trabalho no seu estado atual.

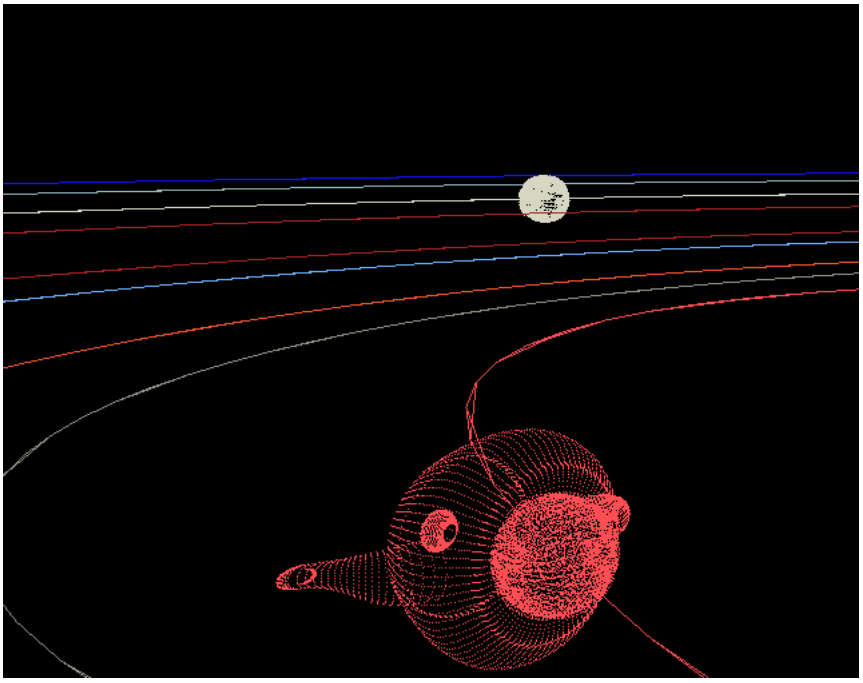


Figura 5.1: O "cometa", teapot, feito em linhas

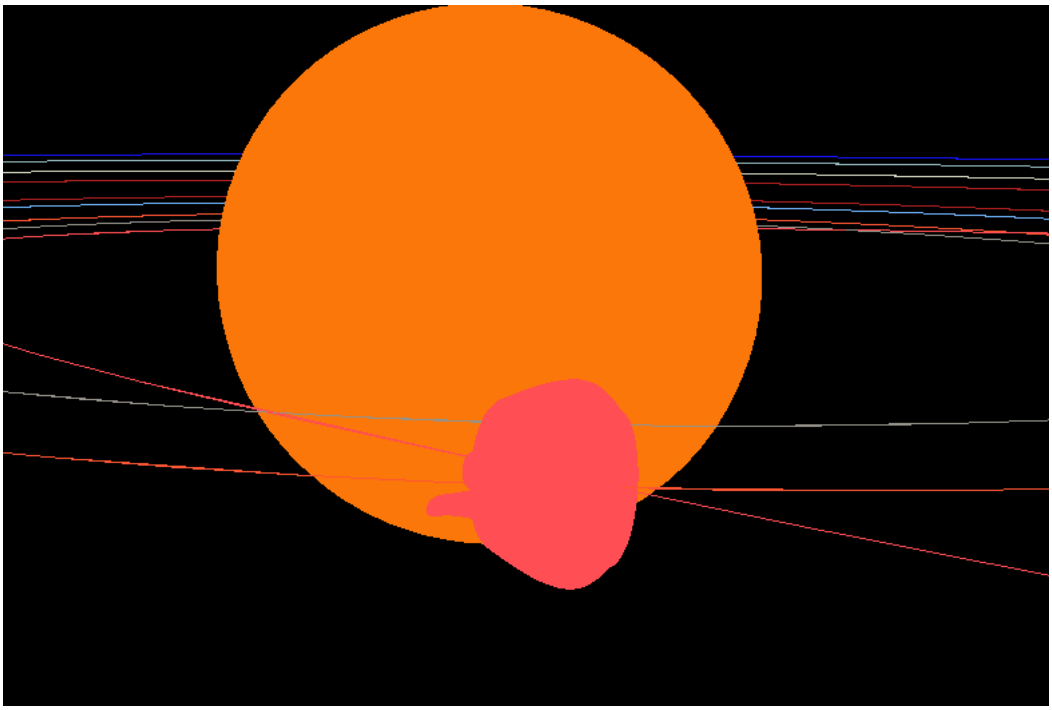


Figura 5.2: O "cometa", teapot, preenchido com a cor escolhida

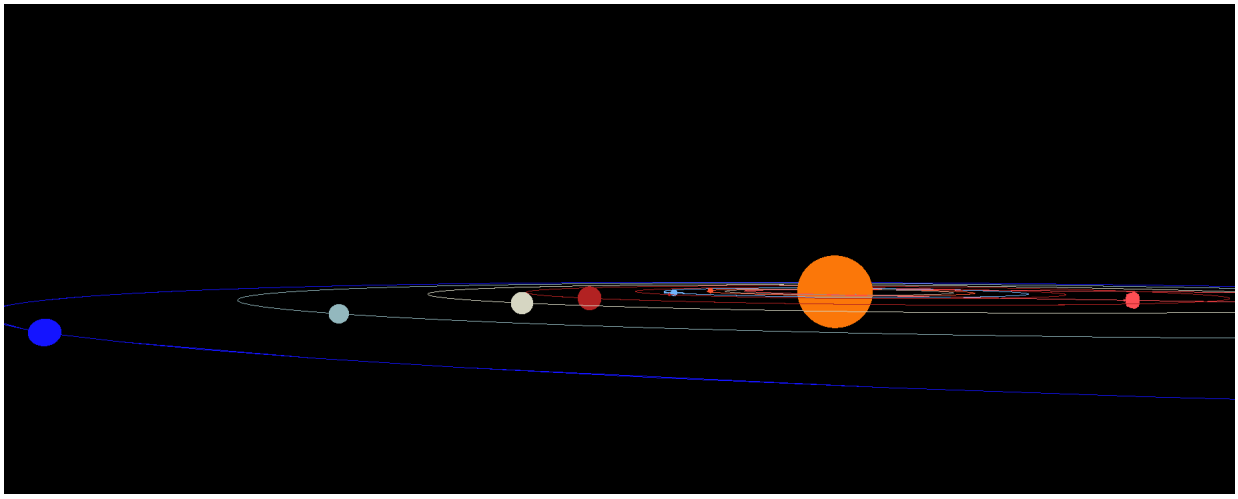


Figura 5.3: Uma das vistas sobre o sistema solar

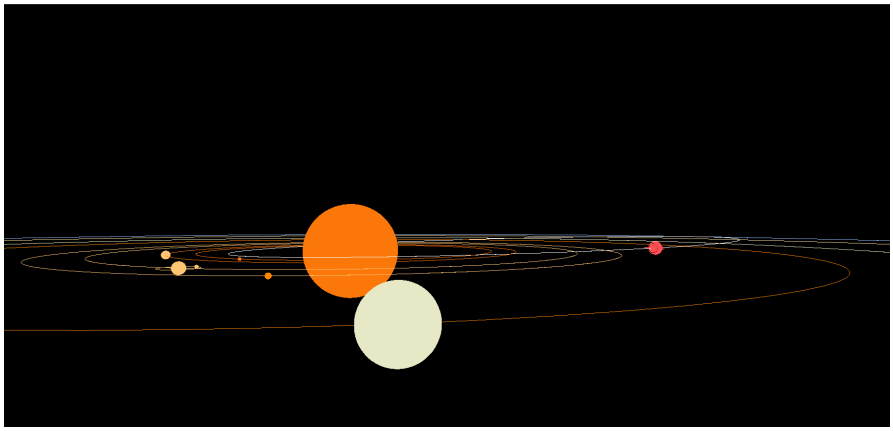


Figura 5.4: Outra vista possivel sobre o sistema solar

Capítulo 6

Conclusão/Trabalho futuro

Ao contrário das fases desenvolvidas anteriormente, a elaboração desta terceira fase, foi notoriamente mais demorada e complexa. Isto deve-se não só ao número de requisitos exigidos, mas também ao nível de complexidade que estes apresentavam.

Durante a elaboração desta fase deparamo-nos com algumas dificuldades entre as quais relacionada com os *Bézier* e tudo aquilo que eles envolvem uma vez que nunca antes tínhamos trabalhado com estes e, desta forma, não possuíamos conhecimento suficiente para, a partir destes, elaborar um algoritmo capaz de representar os modelos gráficos correspondentes. No entanto, após longas pesquisas conseguimos superar esta adversidade nos ocorreu durante o desenvolvimento desta fase.

A implementação tanto dos VBOs como das Catmull-Rom curves foram mais um desafio que tivemos de contornar, porém com os conhecimentos adquiridos nas aulas práticas conseguimos implementa-los com sucesso.

Depois da conclusão desta fase, pensamos que o resultado final desta fase corresponde às expectativas, na medida em que conseguimos desenvolver um modelo dinâmico, do Sistema Solar, tal como pedido no enunciado.

Assim aguarda-mos ansiosamente pela próxima e última fase que se avizinha, para assim optimizar-mos ainda mais o nosso projeto e torna-lo cada vez mais realista e visualmente agradável.

Capítulo 7

Anexos

Em anexo irá um script que criamos em javascript, de forma a obter os vários pontos que inserimos no ficheiro XML, de forma a criar as orbitas.

E um *gif* em que damos um pequeno *preview* do trabalho com algumas das suas funcionalidades no seu estado atual.