**Universidade do Minho**
Escola de Engenharia
Licenciatura em Engenharia Informática

## Unidade Curricular de Computação Gráfica

Ano Letivo de 2022/2023

# Parte 3 - Curvas, Superfícies Cúbicas e VBOs

Filipa Gomes (A96556) Miguel Gomes (A93294) Pedro Pacheco (A61042)
Rita Lino (A93196)

5 de maio

CG

# Resumo

Neste relatório explicitaremos as nossas decisões na realização da terceira fase do trabalho da Unidade Curricular de Computação Gráfica. Visando, deste modo, explicar como foram implementadas as curvas, superfícies cúbicas e VBOs no *engine*, que lê a configuração dos ficheiros e renderiza os modelos.

**Área de Aplicação**: Computação Gráfica.

**Palavras-Chave**: OpenGL, C++, XML, Transformações geométricas, Primitivas gráficas, Catmull-Rom, Bezier patches, VBOs, VAOs...

# Índice

**1 Introdução** **1**

**2 Engine** **2**
2.1 Suporte de *VBO* . . . . . 2
2.2 Transformações com Tempo . . . . . 4
2.3 *Keybinds* . . . . . 6

**3 Generator** **7**
3.1 Leitura do ficheiro Patch . . . . . 7

**4 Sistema Solar** **8**

**5 Testes** **9**

**6 Conclusões e trabalhos futuros** **11**
6.1 Extras . . . . . 11
6.2 Observações . . . . . 12

**7 Referências** **13**

# Lista de Figuras

4.1 Sistema Solar - Animado . . . . . 8

5.1 Ficheiro de Teste teste_3_1.xml . . . . . 9
5.2 Ficheiro de Teste teste_3_2.xml . . . . . 10

6.1 Resultado do erro de implementação Fase2 . . . . . 12

# 1 Introdução

O objetivo desta tarefa prática, a aplicação do gerador deve ser capaz de criar um novo tipo de modelo baseado em Bezier patches. O gerador receberá como parâmetros o nome de um ficheiro onde os pontos de controlo de Bezier são definidos, bem como o nível de *tesselation* necessário. O ficheiro resultante conterá uma lista dos triângulos para desenhar a superfície.

Em relação ao **engine**, serão expandidos os elementos de translação e rotação. Considerando a translação, um conjunto de pontos será fornecido para definir uma curva cúbica de Catmull-Rom, bem como **time** (o número de segundos) para percorrer toda a curva.

O objetivo é realizar animações baseadas nessas curvas. Essa transformação também inclui um campo **align** para especificar se o objeto deve ser alinhado com a curva. Os modelos podem ter uma transformação dependente do tempo ou estática como tal como nas fases anteriores. No nó de rotação, o ângulo pode ser substituído pelo tempo, ou seja, o número de segundos para executar uma rotação completa de 360 graus em torno do eixo especificado.

# 2 Engine

## 2.1 Suporte de *VBO*

Nas fases anteriores, os modelos eram desenhados de modo imediato, no entanto, notava-se algum atraso na inicialização da cena e por consequência um elevado consumo de memória RAM. Para combater esses problemas observados nesta fase foram utilizados *VBOs*. Estes podem ser vistos como um *array* que reside na memória da placa gráfica e permitem melhorias significativas desempenho do programa.

Para sua implementação foi alterada a classe *Model* para suportar *VBOs* e *VAO*. Esta classe contem ainda um método *(prepare_data())* utilizado para carregar os vértices para um vetor e, posteriormente, um *VBO* e *VAO*.

```
1  class Model {
2      public:
3              std::string file;
4              std::vector<Point> points;
5              GLuint vertices = 0;
6              GLuint vao = 0;
7              GLint vertice_count = 0;
8              Color color;
9
10         auto load_file() -> void;
11         auto prepare_data() -> void;
12         auto render() -> void;
13
14 };
```

```
auto Model::prepare_data() -> void {
// criar um vector com os dados dos pontos
        vector<float> p;
        for (Point point : this->points){
                p.push_back(point.x);
                p.push_back(point.y);
                p.push_back(point.z);
        }
        this->vertice_count = p.size() / 3;

// criar o VAO
        glGenVertexArrays(1, &(this->vao));
        glBindVertexArray(this->vao);

// criar o VBO
        glGenBuffers(1, &(this->vertices));

// copiar o vector para a memória gráfica
        glBindBuffer(GL_ARRAY_BUFFER, this->vertices);
        glBufferData(
                GL_ARRAY_BUFFER, // tipo do buffer, só é relevante na altura do desenho
                sizeof(float) * p.size(), // tamanho do vector em bytes
                0, // os dados do array associado ao vector
                GL_STATIC_DRAW // indicativo da utilização (estático e para desenho)
        );
        glBufferSubData(
                GL_ARRAY_BUFFER, // tipo do buffer, só é relevante na altura do desenho
                0, // offset
                sizeof(float) * p.size(), // tamanho do vector em bytes
                p.data() // os dados do array associado ao vector
        );

// indicar que o VBO está associado ao VAO
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(
                0, // índice do atributo
                3, // número de componentes (x, y, z)
                GL_FLOAT, // tipo dos componentes
                GL_FALSE, // normalização
                0, // stride
                0 // offset
        );

// desligar o VAO
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindVertexArray(0);
}
```

Foi ainda alterado o método *render* para utilização de *VBOs* e *VAOs*.

```
auto Model::render() -> void {
        this->color.apply();
        glBindVertexArray(this->vao);
        glDrawArrays(GL_TRIANGLES, 0, this->vertice_count);
        glBindVertexArray(0);
}
```

## 2.2 Transformações com Tempo

Nesta fase foi necessário adicionar a implementação de transformações com o fator **time**, ou seja, o número de segundo que teria que fazer a transformação, tal como o tipo de transformação (estática ou não).

```
enum class TransformationType {
    None,
    TimedTranslate,
    Translate,
    TimedRotate,
    Rotate,
    TimedScale,
    Scale
};
```

Alterou-se a classe *Transformation* original para suportar transformações com tempo.

```
class Transformation {
    public:
        TransformationType type;
        float time;
        int angle;
        float x;
        float y;
        float z;
        bool align;
        Curve curve;
};
```

Foi também implementada a classe *Curve*, que contem os pontos de controlo para calcular a curva através do método *Catmull-Rom*.

```
class Curve{
  private:
    vector<Point> control_points;
    int POINT_COUNT;
    Point position;
    Point derivated;
    Point previous_y;
};
```

Uma transformação, como mencionado anteriormente, poderá ser estática ou realizada em um certo número de segundos definidos. Para tal, foi necessário que na leitura do ficheiro de cena *.xml* se diferenciem entre as transformações estáticas ou não estáticas.

É usado um valor negativo como *flag* de controlo, pois o tempo não deve ser negativo. Sendo depois lidos todos os campos necessários para a sua realização, em caso de um campo incompleto, será utilizado um valor *'default' "0"*.

### Rotação

Leitura de uma **Rotação** pelo *parser*.

```
//(...)
if (strcmp(child_element->Name(), "rotate") == 0){
  float time = child_element->Attribute("time") ? atof(child_element->Attribute(
   "time")) : -1; // Se existir o valor de "time" é lido, senão é colocado o
   valor -1.
  if (time > -1){
    //(...)
    transformations.push_back(Transformation(TransformationType::TimedRotate, x,
    y, z, time));
  }
   else{
    //(...)
    transformations.push_back(Transformation(TransformationType::Rotate, x, y, z
    , angle));
  }
```

### Translação

Leitura de uma **Translação** pelo *parser*. Para esta é necessário fazer a leitura de *pontos de controlo* para a criação da curva no momento da renderização.

```
if (strcmp(child_element->Name(), "translate") == 0){
  float time = child_element->Attribute("time") ? atof(child_element->Attribute(
   "time")) : -1;
  if (time > -1){
    //(...)
    while (points_element != nullptr){
      // Leitura dos pontos de controlo.
      x = points_element->Attribute("x") ? atof(points_element->Attribute("x"))
   : 0;
      y = points_element->Attribute("y") ? atof(points_element->Attribute("y"))
   : 0;
      z = points_element->Attribute("z") ? atof(points_element->Attribute("z"))
   : 0;
      c.add_control_point(Point(x, y, z));
      points_element = points_element->NextSiblingElement();
    }
    transformations.push_back(Transformation(TransformationType::TimedTranslate,
    time, align, c));
  }
else{
  //(...)
  transformations.push_back(Transformation(TransformationType::Translate;, x, y,
    z));
}
```

### Escala

Leitura de uma **Escala** pelo *parser*. Foi também implementada uma escala, ainda que muito rudimentar, esta suporta o "aparecimento"do objeto num certo ponto.

```
else if (strcmp(child_element->Name(), "scale") == 0){
  float time = child_element->Attribute("time") ? atof(child_element->Attribute(
   "time")) : -1;
  if (time > -1)
      type = TransformationType::TimedScale;
  else
      type = TransformationType::Scale;
  //(...)
  transformations.push_back(Transformation(type, x, y, z,time));
}
```

## 2.3 *Keybinds*

Foram definidas algumas *keybinds* que alteram o modo de visualização:

- ***M*** : Alterna entre modos de desenho de polígonos (**GL_FILL**, **GL_LINE** ou **GL_POINT**);
- ***Setas*** : Alterar a posição da câmara relativa ao ponto *LookAt*;
- ***Q*** : Sair do programa;
- ***+ (plus) / - (minus)*** : Alterar a distância da câmara relativa ao ponto *LookAt*.

# 3 Generator

## 3.1 Leitura do ficheiro Patch

### *Bezier Patches*

Para a criação das Bezier Patches foi necessário criar uma classe **BezierPatch**, que tem como campos a *tesselation*, número de *patches*, número de pontos, vetor de vetores ($*_{matriz}$) de índices e um vetor de pontos de controlo.

```
class BezierPatch {
  public:
    int tessellation;
    int n_patches;
    int n_points;
    vector<vector<int>> indices;
    vector<Point> control_points;
};
```

Visando encontrar os pontos de Bezier, foi criada uma função **bezier** que utiliza os polinómios de Bernstein para calcular os pontos no instante *tt*.

```
Point bezier(float t, float tt, std::vector<Point> points, std::vector<int>
    indexes){
  Point initial[4] = {Point(), Point(), Point(), Point()};
  std::vector<Point> curve;
  int i, j = 0;
  for (i = 0; i < 16; i++){
    initial[j] = Point(points[indexes[i]].x, points[indexes[i]].y, points[
    indexes[i]].z);
    if(j == 3) {
      curve.push_back(bernstein(t, initial[0], initial[1], initial[2], initial
    [3]));
      j = 0;
    }
    else j++;
    }
    return bernstein(tt, curve[0], curve[1], curve[2], curve[3]);
}
```

# 4 Sistema Solar

Após a implementação das novas funcionalidades, é possível observar algumas alterações no cenário do Sistema Solar (apresentado na fase anterior). O Sistema Solar tem agora alguns movimentos, ou seja, rotações entre os astros e planetas que ajudam a simular um verdadeiro Sistema Solar. Podemos também observar que alguns cometas e planetas têm as suas órbitas diferentes das restantes.

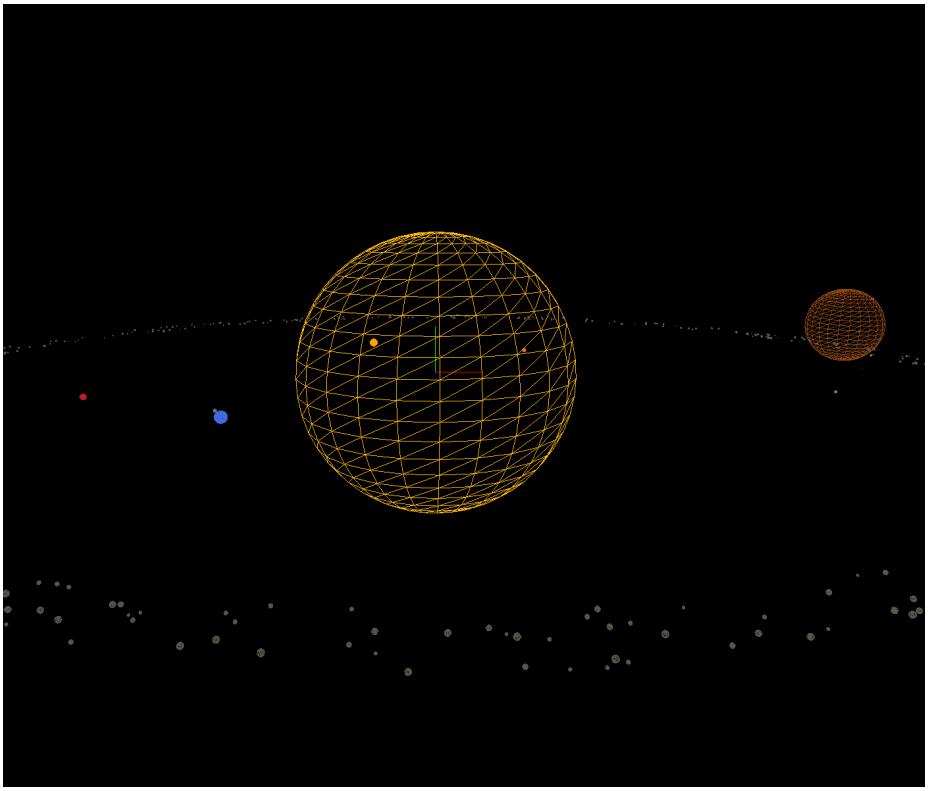

Figura 4.1: Sistema Solar - Animado

*(Demonstração em Vídeo $_{youtube}$)

# 5 Testes

Por meio de ficheiros disponibilizados é possível fazer a validação da implementação efetuadas usando os cenários de teste. Seguem os cenários resultantes dos mesmos:

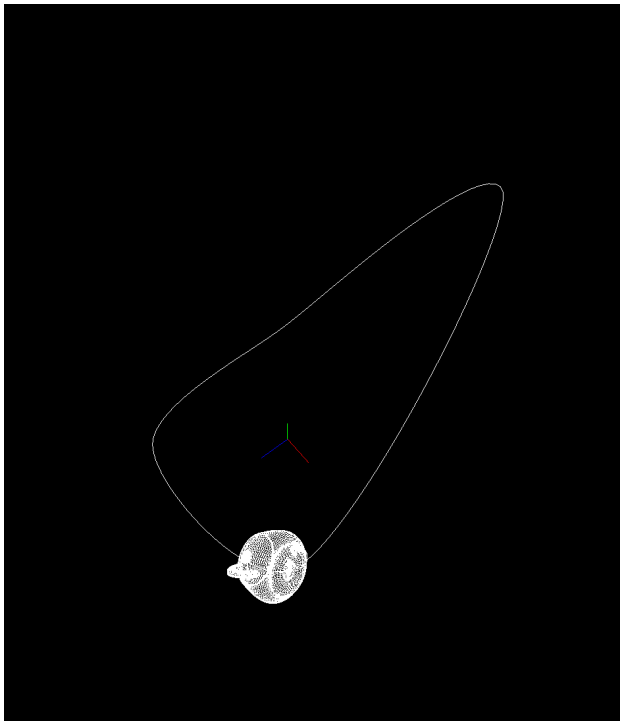

Figura 5.1: Ficheiro de Teste teste_3_1.xml

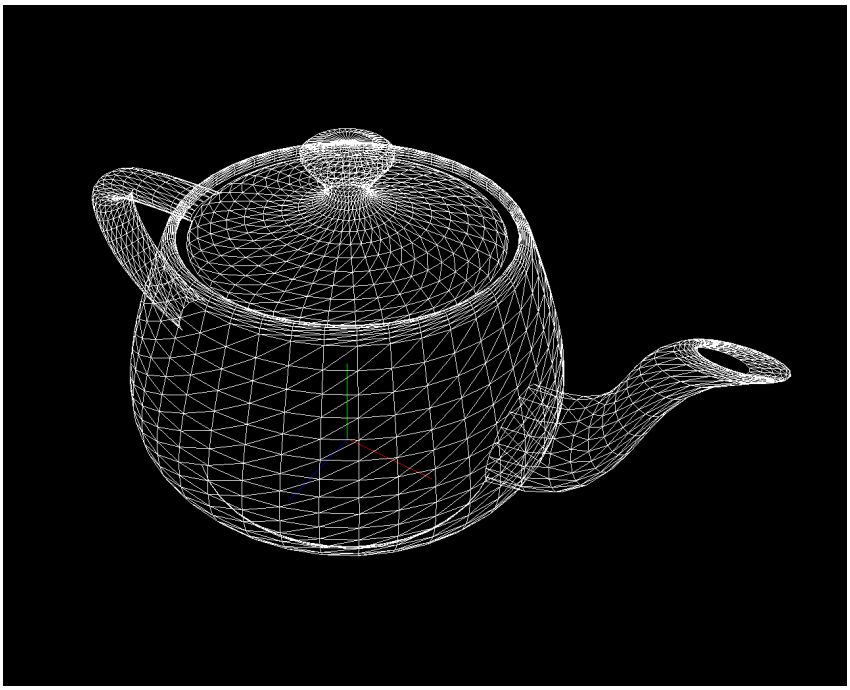

Figura 5.2: Ficheiro de Teste teste_3_2.xml

# 6 Conclusões e trabalhos futuros

No final da realização desta terceira fase, conseguimos consolidar os conhecimentos adquiridos durante as aulas acerca de transformações de modelos 3D, utilização de VBOs e VAOs, e perceber melhor como funcionam as Bezier Patches tal como a Catmull-Rom. Graças à aplicação destes novos conhecimentos foi nos possível criar um modelo 3D de um Sistema solar mais realista que o criado na fase anterior.

Considera-se que foi feito um bom trabalho tendo em conta do esforço de todos os elementos do grupo na implementação e complementação desta terceira fase do projeto.

## 6.1 Extras

O *engine* contém uma nova funcionalidade extra, que é aplicação de escalas com tempo, como foi possível visualizar na secção ***Escala***

As funcionalidades extra que ainda se encontram em desenvolvimento são *Pick and Click* para identificação de vários modelos com o nome desse grupo/modelo com utilização do rato, a criação de uma câmara *FPV*, menus para facilitar a interação com o utilizador, carregamento dinâmico de cenas e modelos e alteração das cenas interativamente (através de rato e teclado).

## 6.2 Observações

Foi descoberto que a implementação apressada da funcionalidade extra *Identificação de grupos* para utilização com outra funcionalidade extra *Pick and Click* levou a um problema no carregamento dos ficheiros de teste fornecidos na fase anterior, sendo esse problema resolvido nesta fase.

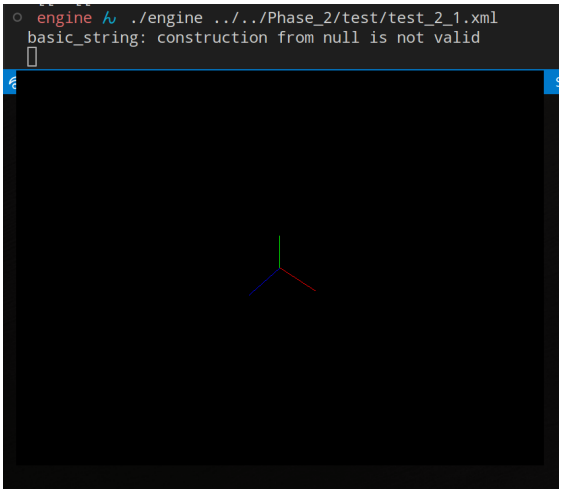


Figura 6.1: Resultado do erro de implementação Fase2

Para corrigir esse problema foi apenas necessário alterar uma linha de código:

```
string name = element_group->Attribute("name");
        VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
string name = element_group->Attribute("name") ? element_group->Attribute("name") : "";
```

# 7 Referências


1. **Dados sobre planetas e luas (NASA)**. (04/04/2022).
2. **Distancia relativa entre planetas e luas (National Geographic)**. (04/04/2022).
3. **Referência para o *'script'* gerador do *Sistema Solar* (Github)**. (05/05/2023).
4. **Demonstração em vídeo do sistema solar - https://youtu.be/-gOF6WKwT5E**