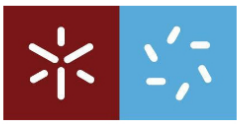

Universidade do Minho

Licenciatura em Ciências da Computação

Fase 3 - Curvas, Superfícies Cúbicas e VBOs

Computação Gráfica

Grupo 7

Cláudia Rego Faria
(A105531)

Diogo José Borges Dias
(A102943)

Maria Inês Barros de Matos
(A102937)

Patrícia Daniela Fernandes Bastos
(A102502)

27 de abril de 2025

Conteúdo

1 Introdução **3**

2 Estrutura do projeto **4**

3 Engine e Estruturas de Dados **5**
3.1 Superfícies Cúbicas 5
3.1.1 Alterações nas estruturas de dados 5
3.1.2 Alterações ao *parser XML* 6
3.1.3 Integração das transformações animadas no engine . . 7
3.2 VBOs 8
3.2.1 Alterações nas estruturas de dados 9
3.2.2 Alterações no Engine 9
3.2.3 Alterações no ficheiro *CMakeLists.txt* 9
3.3 Movimento da Câmara e Modos de Desenho 10

4 Generator **11**
4.1 Geração de Superfície Bézier 11
4.1.1 Implementação 11
4.1.2 Funcionamento Detalhado 11
4.1.3 Função *bernstein* 12
4.1.4 Cálculo de Pontos na Superfície — *bezierPatch* 12

5 O Sistema Solar **14**

6 Testes **17**

7 Conclusão **19**

Capítulo 1

Introdução

Este relatório é o terceiro de um trabalho de quatro partes, no âmbito da UC Computação Gráfica da Licenciatura em Ciências da Computação e referente ao ano letivo de 2024/2025.

Para a Terceira Fase, foram feitas alterações ao *Engine*, ao *Generator* e ao ficheiro *XML* do sistema solar. No *Engine* foram necessárias alterações tanto para suportar a utilização de *VBOs* como a animação baseada em curvas cúbicas de *Catmull-Rom*. Com o *Generator* agora é possível criar um novo tipo de modelos baseados em patches de *Bezier*.

Esta fase teve como objetivo a aprendizagem e implementação de curvas, superfícies cúbicas e *VBOs*, usando os conhecimentos adquiridos na fase anterior.

Capítulo 2

Estrutura do projeto

Em semelhança da fase anterior, para uma melhor organização do projeto, este foi dividido nos seguintes diretórios:

- **3d**: armazena os ficheiros criados pelo generator que serão posteriormente utilizados pelo engine para a renderização dos modelos
- **engine**: contém a implementação do motor gráfico responsável pela leitura e visualização dos modelos 3d gerados
- **gen**: inclui o código responsável pelo cálculos dos pontos necessários para a construção das primitivas e pela geração dos ficheiros 3d
- **data_structs**: contém as structs desenvolvidas para um melhor e mais fácil funcionamento do projeto
- **test_files**: contém os ficheiros de teste que serão utilizados ao longo das diferentes fases do projeto, e o ficheiro *teapot.patch*
- **TinyXml**: contém a biblioteca *TinyXML* utilizada para auxiliar na leitura e manipulação de ficheiros .xml
- **demo_scenes**:
  - contém o ficheiro XML para a verificação desta terceira fase, neste caso o Sistema Solar
  - **help_functions**: contém os ficheiros python que auxiliam a criação dos pontos necessários para as curvas de *Catmull-Rom*
- **outputs**: contém o ficheiro *CMakeLists.txt*

Capítulo 3

Engine e Estruturas de Dados

Nesta fase, foram feitas alterações ao *Engine* e a certas estruturas de dados necessárias para a implementação das curvas cúbicas de *Catmull-Rom* e dos *VBOs*.

3.1 Superfícies Cúbicas

Para implementar as novas rotações e translações foi necessário:

- Alterar as estruturas de dados
- Alterar o parser do XML
- Integrar as transformações animadas no *engine*

3.1.1 Alterações nas estruturas de dados

Primeiramente, para que seja possível suportar as novas rotações e translações foi necessário fazer algumas adições na *struct Transform*. Na fase 2 apenas eram permitidas transformações estáticas, enquanto que agora também são possíveis transformações animadas.

Listing 3.1: struct Transform na Fase 2

```
1        struct transform{
2            char type;
3            Point transformation;
4            float angle;
5        }
6
```

Listing 3.2: struct Transform na Fase 3

```
struct transform{
    char type;
    Point transformation;
    float angle;
    float time;
    bool align;
    std::vector<Point> points;
}

```

Foram adicionados os campos:

- ***time*** : Em rotações, é o tempo necessário para completar 360º e, em translações, é o tempo para percorrer a curva
- ***align***: Este campo é do tipo booleano, caso seja *true* vai ativar o alinhamento dinâmico com a curva
- ***points***: Vai armazenar os pontos de controlo da curva *Catmull-Rom* (esta curva precisa de um mínimo de 4 pontos)

Para estes novos campos também foi necessário adicionar as respectivas funções *get*.

3.1.2 Alterações ao *parser XML*

Também foi necessária a atualização do *parser*, implementado no ficheiro *Settings.cpp*, que para além de ser capaz de ler transformações estáticas (como já fazia anteriormente), agora deveria ser também capaz de ler transformações dinâmicas.

O parser funciona através da função *parseGroup()* que recursivamente processa elementos XML, identificando-os e criando transformações para os *models*.

As transformações dividem-se em três categorias principais :

1. **Translações (*translate*)**
   Após a verificação de que se trata de uma translação, é necessário identificar se é uma translação estática ou animada. Para isso é verificada a existência do atributo *time*. Caso este atributo *time* exista significa que estamos perante uma translação animada.

(a) **Translação animada**
Após identificar o tipo da translação o parser vai guardar as informações do XML relativas ao *time* e ao *align*. Depois vai criar um vetor de pontos que irá armazenar os pontos de controlo da curva. Como as curvas *Catmull-Rom* necessitam de pelo menos 4 pontos foi necessário controlar a criação das translações. Caso o vetor de pontos tenha pelo menos 4 pontos a translação vai ser criada e o *parsing* vai continuar. Em caso contrário vai ser mostrada uma mensagem de erro e a translação irá ser ignorada.

(b) **Translação estática**
Este tipo de translação acontece quando não existe presença de um atributo *time* e então a translação vai ser tratada da forma implementada na fase 2.

2. **Rotações (*rotate*)**
O tratamento de rotações acontece de forma parecida ao das translações mas, neste caso, se estiver presente o atributo *time* sabe-se que se trata de uma rotação animada, enquanto que estando o atributo *angle* trata-se de uma rotação estática.

3. **Escalas (*scale*)**
As escalas não tiveram qualquer tipo de mudança de uma fase para a outra.

A lógica dos grupos e subgrupos continua a mesma.

3.1.3 Integração das transformações animadas no engine

O *engine* sofreu algumas alterações para suportar as novas transformações, particularmente na função *drawFigures()* e no sistema de matrizes (implementado em *Matrix.cpp*). Foi também criada uma nova função para desenhar as curvas chamada *drawCatmullRomCurve()*.

As alterações em *drawFigures* (processamento de transformações) foram as seguintes:

- **Rotação Animada**
Quando é encontrada uma rotação com tempo definido (time > 0), o ângulo atual é calculado baseado no tempo decorrido desde o início da animação.

```
1    float angle = (glutGet(GLUT_ELAPSED_TIME)/1000.0f/time) *
     360.0f;
2
```

A fórmula acima converte o tempo em um ângulo de 0 a 360 graus, criando assim um efeito de rotação contínua. Esta rotação é aplicada com o uso do *glRotatef*.

Listing 3.3: processamento completo de rotações animadas

```
1    if(type == 'R' && time > 0) {
2        float angle = (glutGet(GLUT_ELAPSED_TIME)/1000.0f/time)
     * 360.0f;
3        glRotatef(angle, getX(tv), getY(tv), getZ(tv));
4    }
5
```

- **Translação Animada**
  Quando está presente uma translação animada é necessário começar por converter os pontos de controlo para o formato esperado pela função *getGlobalCatmullRomPoint()*. Se a opção *showCurve* estiver ativada, a trajetória da curva é desenhada com o uso da função *drawCatmullRomCurve()*.

  Após o desenho da curva é preciso calcular a posição na curva *Catmull-Rom* , ou seja, onde o objeto deve estar a cada momento da animação. Primeiro, o tempo decorrido é normalizado para um valor entre 0 e 1, que representa o progresso total da animação. Esse valor é passado para a função *getGlobalCatmullRomPoint()*, que retorna a posição exata (x , y, z) na curva e a direção do movimento naquele instante. O objeto vai ser então movido para essa posição com o uso do *glTranslatef*.

  Por fim, é preciso verificar o alinhamento. Caso *align* seja *true*, o objeto gira automaticamente para acompanhar a curva, alinhando a frente com a direção do movimento. Caso seja *false*, o objeto mantém a sua rotação original enquanto se move pela curva, como se estivesse a deslizar sem inclinação.

- **Transformações estáticas**
  Para as transformações estáticas o *drawFigures* não precisa de cálculos auxiliares e por isso utiliza diretamente o *glRotatef*, *glTranslatef* e *glScalef* do *OpenGL*.

3.2 VBOs

Nesta fase do trabalho a renderização dos *Models* é feita com VBOs, substituindo o método usado nas fases anteriores.

Para isso foi necessário:

- alterar as estruturas de dados
- alterar a forma como as figuras são desenhadas no Engine
- alterar o ficheiro cmakelists

3.2.1 Alterações nas estruturas de dados

Model
Para a utilização de VBOs é necessário implementar a inicialização e gestão de buffers.

Para isso foi necessário fazer algumas adições na *struct Model*. Foram adicionados os campos:

- **vertexB** : substitui o vetor de pontos *point* de forma a suportar *VBOs*
- **vertexCount** : quantidade total de vértices
- **buffer** : array que armazena o *buffer*, e futuramente, o vetor nomal
- **bufferInitialized** : campo do tipo boolenao que identifica se o buffer do *model* já foi inicializado

Além disso foram implementadas os respectivos métodos de *get* e a função *initModelBuffers*, que inicializa o *buffer* do *model* em questão caso não tenha sido inicializado.

Group
Foi adicionada a função *initGroupBuffers* que inicializa os buffers para cada *model*, com *initModelBuffers*, e para cada subgrupo, recursivamente, de um certo grupo.

3.2.2 Alterações no Engine

Com o auxílio das funções anteriores, as figuras que anteriormente eram desenhadas a partir de um ciclo *for* e de *glVertex3f* agora são renderizadas apenas através de *glDrawArrays*.

3.2.3 Alterações no ficheiro *CMakeLists.txt*

O ficheiro foi alterado com o auxilio do ficheiro *CMakelists.txt* fornecido nas aulas práticas.

3.3 Movimento da Câmara e Modos de Desenho

Em complementação à fase anterior, foi adicionada opção de alterar a visualização das curvas de *Catmull-Rom*.

Movimento

W: Mover para a frente

S: Mover para trás

A: Mover para a esquerda

D: Mover para a direita

+: Mover para cima

- : Mover para baixo

Cursor: Alterar a direção da câmara

Modo de Desenho

M: Alternar entre preenchimento, linhas e pontos

P: Alternar entre mostrar ou não os eixos *xyz*

C: Alternar entre mostrar ou não as curvas de *Catmull-Rom*

Capítulo 4

Generator

4.1 Geração de Superfície Bézier

Um dos objetivos desta fase do projeto consistiu em implementar no *Generator* a capacidade de criar ficheiros 3D baseados em superfícies definidas por patches de *Bézier*.

4.1.1 Implementação

A principal função desenvolvida para esta fase foi *genPatch*, que encapsula todo o processo descrito acima. Esta função recebe como parâmetros o caminho para o ficheiro que contém os patches e pontos de controlo e o nível de tesselação pretendida.

4.1.2 Funcionamento Detalhado

O processo dentro de *genPatch* segue as seguintes etapas:

- **Leitura do Ficheiro de Entrada**
  - O ficheiro contém inicialmente o número de *patches*.
  - Para cada *patch*, são lidos 16 índices correspondentes aos pontos de controlo e são armazenados num vetor que vai conter todos os *patches*.
  - Depois, é lido o número total de pontos de controlo.
  - Finalmente, são lidas as coordenadas (x,y,z) de cada ponto de controlo.
- **Geração dos Triângulos**
  Para cada *patch*:
  - São recolhidos os 16 pontos de controlo necessários.

- A superfície é subdividida em pequenos quadrados (dependendo do nível de tesselação).
- Para cada quadrado, são calculados 4 pontos sobre a superfície de *Bézier* usando a função *bezierPatch*.
- Cada quadrado é dividido em dois triângulos.

Este processo gera uma aproximação da superfície suave através de triângulos.

- **Escrita no Ficheiro de Saída**
  Os vértices dos triângulos gerados são depois escritos no ficheiro de saída 3d.

4.1.3 Função *bernstein*

A função de *bernstein* calcula os polinómios cúbicos de *Bernstein*, que são a base matemática das curvas e superfícies de Bezier. Para curvas de Bezier cúbicas, existem quatro funções de base que representam a influência dos pontos de controlo. Os polinómios de *Bernstein* de grau 3 definidos pela função são:

$$B0(t) = (1-t)^3$$
$$B1(t) = 3t(1-t)^2$$
$$B2(t) = 3t^2(1-t)$$
$$B3(t) = t^3$$

Estes polinómios calculam o peso de cada ponto de controlo num patch de *Bézier* para um dado valor de t.

A função *genPatch* usa bernstein através da função *bezierPatch*, para calcular os pontos da superfície a partir dos pontos de controlo e dos parâmetros u e v, gerando assim os triângulos que desenham a superfície.

4.1.4 Cálculo de Pontos na Superfície — *bezierPatch*

A função *bezierPatch* calcula a posição de um ponto numa superfície de *Bézier*, para umas coordenadas (u,v).

Esta função:

- recebe um vetor de pontos de controlo que definem a forma do *patch*;
- calcula todas as combinações de polinómios de *Bernstein* para os parâmetros u e v;
- calcula a soma ponderada de todos os pontos de controlo;
- Devolve o ponto final (x,y,z) que pertence à superfície de *Bézier*.

Essencialmente, bezierPatch avalia o patch aplicando a fórmula:

$$p(u,v) = \sum_{i=0}^{\mathrm{n}} \sum_{j=0}^{\mathrm{m}} {B_i}^n(u) {B_j}^m(v) k_{i,j}$$

Esta função é usada em *genPatch* para calcular os vértices da superfície, que depois são ligados em triângulos para desenhar o modelo.

Capítulo 5

O Sistema Solar

Nesta fase do trabalho a *demo scene* do sistema solar é dinâmica, em contrário à fase anterior em que se tratava de um sistema solar estático.

Para que isso fosse possível o ficheiro XML foi alterado para conter as curvas de *Catmull-Rom* desenvolvidas nesta fase.

Cada planeta, para além de rodar em torno do próprio eixo, também orbita o Sol com tempos ligeiramente ajustados, em compração com o tempo real, para melhor visualização. As luas também orbitam os seus respetivos planetas.

Luas			
Planeta	Lua	Período orbital (dias)	Inclinação
Júpiter	Io	1.77	0.04°
Júpiter	Europa	3.55	0.47°
Júpiter	Ganymede	7.15	0.18°
Júpiter	Callisto	16.69	0.19°
Saturno	Titan	15.95	0.33°
Neptuno	Triton	5.88	157.35°
Terra	Lua	27.32	5.145°

Tabela 5.1: Características orbitais das luas

Planetas			
Planeta	Rotação (duração de um dia)	Período orbital (dias)	Inclinação
Mercúrio	1408	12.567	0°
Vénus	5832.26	32.1	177.4°
Terra	23.56	52.18	23.4°
Marte	24.36	98.096	25.2°
Júpiter	9.55	618.84	3.1°
Saturno	10.33	1558.05	26.7°
Urano	17.14	4383.52 (2000)	97.8°
Neptuno	16	8598.51 (3000)	28.3°

Tabela 5.2: Características rotacionais e orbitais dos planetas

Foi também adicionado um cometa em forma de *teapot*, gerado a partir do comando *./generator patch ../test_files/teapot.patch 2 bezier_2.3d*.

O cometa tem uma trajetória eliptica em torno do sol, tendo sido aplicada uma escala para que fique mais semelhante a um cometa.

Para que fosse possível obter os pontos de controlo das curvas de *Catmull-Rom* mais rapidamente foram criados dois scripts em python que são capazes de gerar os pontos tanto para uma tragetória ciclica (*calcPointsCircle.py*) como para uma eliptica (*calcPointsElipse.py*), utilizando apenas a distância do raio (ou maior raio, no caso da elipse) nas coordenadas x e z.

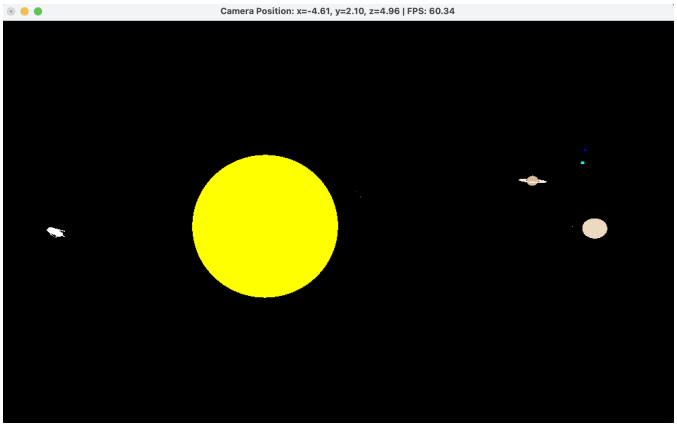

Figura 5.1: *solar_system.xml*

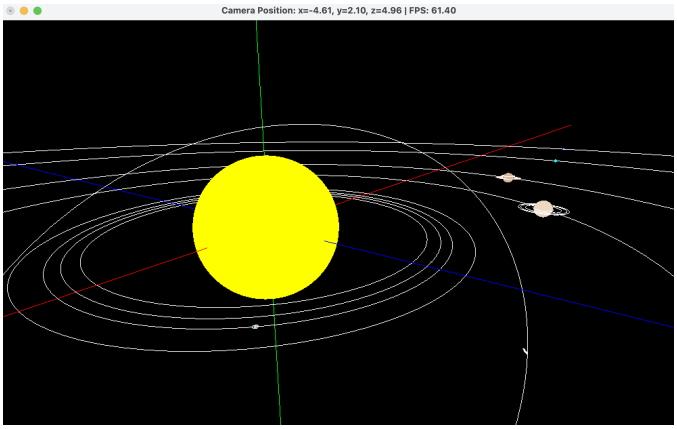

Figura 5.2: *solar_system.xml com eixos e curvas visíveis*

Capítulo 6

Testes

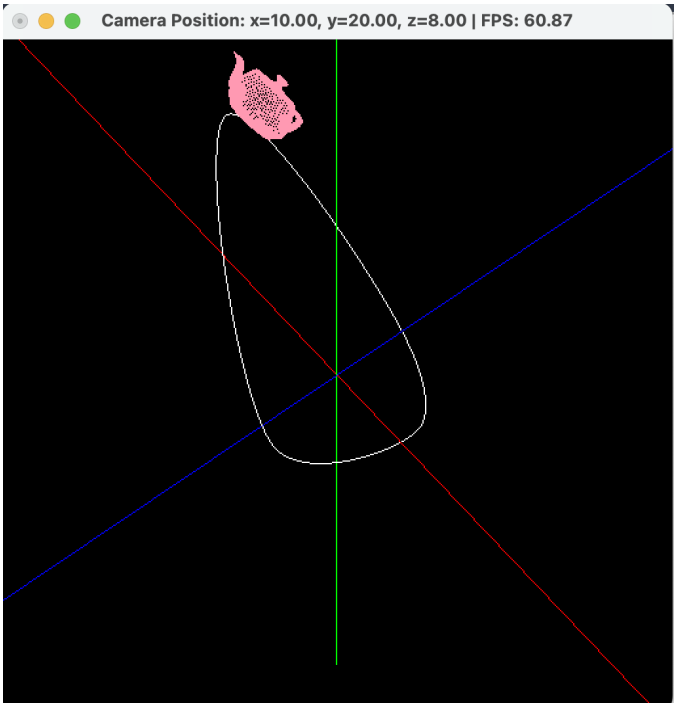

Figura 6.1: *test_3_1.xml*

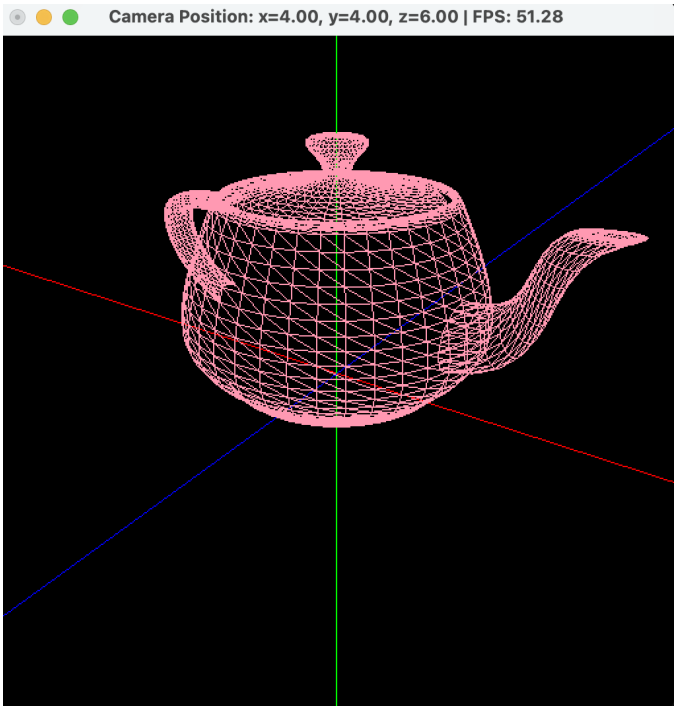

Figura 6.2: *test_3_2.xml*

Capítulo 7

Conclusão

Nesta Terceira Fase do trabalho, foi possível consolidar os conhecimentos práticos e teóricos das aulas de Computação Gráfica. Aprendeu-se a suportar a utilização de *VBOs* e a implementar animação baseada em curvas cúbicas de *Catmull-Rom*. Foram implementadas alterações de maneira a suportar patches de *Bezier*, que permitem criar superfícies lisas e contínuas que podem ser controladas com precisão e renderizadas com eficiência.

Os conhecimentos adquiridos nesta fase dão continuidade à fase anterior e são essenciais para a etapa subsequente, onde irão ser exploradas técnicas mais avançadas de Computação Gráfica, nomeadamente a iluminação e as texturas.