COMPUTAÇÃO GRÁFICA

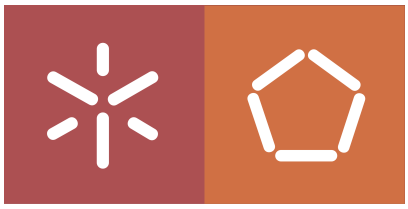
Fase 3 - Curvas, Superfícies Cúbicas e VBOs

Diogo Esteves
a104004

Rodrigo Fernandes
a104175

Diogo Barros
a100751

Fevereiro 2025



Universidade do Minho
Escola de Engenharia

Conteúdo

1 Introdução

Este relatório descreve o processo de evolução do motor 3D baseado em *scene graph*, desenvolvido nas últimas duas fases, como parte do projeto prático da unidade curricular de **Computação Gráfica** da Universidade do Minho.

Nesta fase houve a necessidade de gerar superfícies cúbicas a partir de ficheiros de formato fixo e ser possível a translação de primitivas em sentido curvo.

2 *Generator*

Nesta fase de projeto foi necessário o generator ser capaz de gerar pontos de superfícies de *Bezier* a partir de ficheiros de formato fixo com os pontos de controlo.

2.1 *Patch*

Nesta fase foi implementada a capacidade de gerar superfícies cúbicas de Bézier a partir de um ficheiro com patches de controlo e pontos. Cada patch define um conjunto de 16 índices que referenciam pontos de controlo num espaço tridimensional. O objetivo é, dada uma tesselação, produzir os triângulos que representam visualmente a superfície.

A geração das superfícies baseia-se no conceito de **superfícies de Bézier**, que são definidas por uma malha bidimensional de pontos de controlo $P_{i,j}$ com $i \in [0, m]$ e $j \in [0, n]$. A superfície de Bézier é uma função paramétrica de duas variáveis u e v, ambas no intervalo $[0, 1]$, dada pela seguinte equação:

$$S(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(u) \cdot B_{n,j}(v) \cdot P_{i,j} \tag{1}$$

onde $B_{m,i}(u)$ e $B_{n,j}(v)$ são as funções base de Bézier, definidas por:

$$B_{n,i}(t) = ni(1-t)^{n-i}t^i \tag{2}$$

Estas funções base, conhecidas como **funções de Bernstein**, determinam o peso de cada ponto de controlo na construção da superfície.

Cada patch é processado individualmente. Para cada par de parâmetros (u, v) definido pela tesselação, é calculado um ponto na superfície utilizando o produto tensorial das funções de Bernstein nos dois eixos. Isto permite interpolar a superfície a partir dos 16 pontos de controlo definidos no patch.

Para cada patch, a malha de pontos gerada com base na tesselação é usada para criar os triângulos. Cada célula da grelha é composta por dois triângulos definidos a partir dos quatro pontos do quadrado correspondente. Estes triângulos são então escritos no ficheiro de saída como coordenadas tridimensionais.

Esta abordagem permite representar superfícies suaves e contínuas com base num número reduzido de pontos de controlo, sendo extremamente útil para modelação de formas curvas em computação gráfica.

Seguem uns exemplos de um figuras geradas a partir de ficheiros patch:

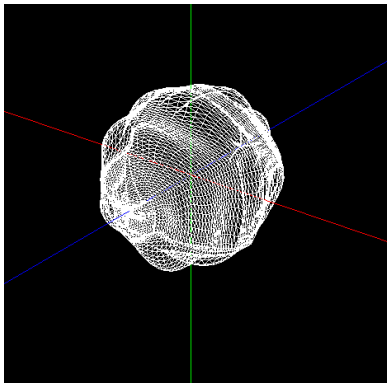


Figura 1: Cometa gerado com superfícies de Bezier

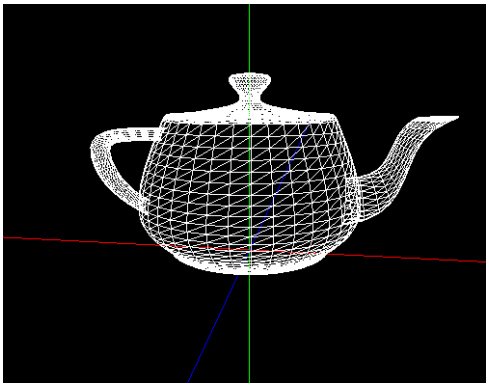


Figura 2: Teapot gerado com superfícies de Bezier

3 Engine

Nesta 3ª fase, o programa *engine* sofreu alterações para comportar diversas funcionalidades adicionais tais como: técnicas de otimização com recurso a VBOs, desenho de animações através de curvas de Catmull-Rom, otimizações no que toca às transformações geométricas da 2ª fase e a adição de uma camara orbital, para uma melhor visualização da cena.

3.1 Câmara Orbital

A câmara orbital foi implementada de forma a permitir ao utilizador observar a cena a partir de uma perspetiva esférica em torno de um ponto de interesse (habitualmente a origem do sistema de coordenadas). Esta abordagem é ideal para visualização de sistemas complexos, como o sistema solar, pois permite ao utilizador explorar livremente o espaço tridimensional mantendo um foco central constante.

A posição da câmara é calculada em coordenadas esféricas (r, θ, ϕ), onde:

- r representa a distância radial entre a câmara e o ponto de interesse;
- θ (theta) representa o ângulo de rotação em torno do eixo vertical (azimute);
- ϕ (phi) representa o ângulo de elevação (ângulo zenital) em relação ao plano horizontal.

Estas coordenadas esféricas são posteriormente convertidas para coordenadas cartesianas (x, y, z) utilizando as seguintes equações:

$$x = r \cdot \cos(\phi) \cdot \sin(\theta) \tag{3}$$

$$y = r \cdot \sin(\phi) \tag{4}$$

$$z = r \cdot \cos(\phi) \cdot \cos(\theta) \tag{5}$$

A câmara é, então, posicionada na cena com base nestas coordenadas e orientada para o ponto de interesse utilizando a função `gluLookAt`, que define a direção do olhar da câmara, o ponto de observação e o vetor *up*.

A interação com a câmara foi implementada utilizando o rato: ao pressionar e arrastar com o botão esquerdo, os valores dos ângulos θ e ϕ são atualizados com base no movimento horizontal e vertical do rato, respetivamente. Assim, o utilizador consegue "orbitar"em torno do centro da cena de forma fluida.

Para além disso, foi ainda implementada a funcionalidade de mostrar ou esconder os eixos da cena pressionando a tecla `T`, alternando o valor de uma variável booleana interna que controla esse comportamento visual.

Esta câmara proporciona uma navegação intuitiva e é particularmente útil para visualização exploratória, sendo uma abordagem comum em motores gráficos e aplicações de visualização 3D.

3.2 Otimizações de Desempenho

A primeira mudança nesta fase foram as otimizações de desempenho proporcionadas pela implementação de VBOs (Vertex Buffer Objects), utilizadas para desenho da geometria do OpenGL.

Um VBO pode ser visto como um array alocado na memória da placa gráfica, ao invés de alocado na memória RAM (como implementado nas fases anteriores). Assim obtemos ganhos de desempenho significativos, uma vez que foi reduzida a comunicação entre o CPU e a GPU, uma vez que agora a informação relatica aos vértices encontra-se alocada localmente na GPU. Além disso, torna-se possível tirar partido do paralelismo da GPU, uma vez que a distribuição de carga é agora gerida pelo driver da GPU.

Para que fosse possível implementar VBOs para cada modelo de uma dada cena 3D, começamos por, durante o processo de parsing dos ficheiros '.3d', iniciar para cada modelo um array/vector em C e copiamos o mesmo para a GPU, associando ao mesmo um ID (para que mais tarde seja possível o desenho do mesmo) que fica também armazenado em memória para consulta posterior. Esta operação é realizada apenas no parsing, não se repetindo mais nenhuma vez durante a execução do programa, uma vez que a geometria do mesmo não é dinamica.

Durante o processo de desenho, para cada modelo basta passar o ID do VBO armazenado, para garantir que estamos a desenhar o modelo correto, e o tipo dos dados que o array que armazenamos posteriormente contém.

Com a implementação desta técnica, conseguimos ganhos de desempenho significativos.

3.3 Otimizações das Transformações Estáticas

Nesta fase, houve a necessidade de distinguir as transformações estáticas de transformações relativas a animações, em especifico para os casos das rotações e translações, onde esta última utiliza as curvas de Catmull-Rom para cálculo da posição seguinte ao longo de um dado tempo.

Para simplificar este processo, durante o processo de parsing, e com recurso à biblioteca glm, fomos multiplicando numa matriz local ao grupo, a matriz relativa à transformação que está a ser processada, garantindo que o resultado final, para um determinado group, é uma matriz com as transformações

estáticas do xml aplicadas na ordem correta. Assim, no processo de desenho do group em questão, basta multiplicar a matriz do OpenGL pela matriz 'static_transformations', armazenada em cada group. Esta matriz tem como valor inicial a matriz identidade, para que caso não hajam transformações estáticas associadas a um dado group, esta multiplicação não anule as transformações já aplicadas pelo group-pai.

3.4 Curvas de Catmull-Rom

As curvas de **Catmull-Rom** foram introduzidas nesta fase para permitir animações de translação suaves e contínuas entre um conjunto de pontos de controlo. Esta técnica é essencial para simular movimentos curvos, como o trajeto de planetas ou satélites no espaço tridimensional.

Uma curva de Catmull-Rom é uma curva paramétrica que passa diretamente pelos pontos de controlo fornecidos. A curva entre cada par de pontos é influenciada pelos dois pontos anteriores e seguintes, garantindo continuidade C^1 (derivada contínua), ideal para animações suaves.

A equação de uma curva de Catmull-Rom para um parâmetro $t \in [0, 1]$ e quatro pontos de controlo consecutivos P_0, P_1, P_2, P_3 é dada por:

$$P(t) = 0.5 \cdot \big(2P_1 + (-P_0 + P_2)t + (2P_0 - 5P_1 + 4P_2 - P_3)t^2 + (-P_0 + 3P_1 - 3P_2 + P_3)t^3\big) \tag{6}$$

No processo de parsing da cena, os pontos de controlo de uma curva são lidos do XML e armazenados para posterior interpolação. Durante a execução do programa, a posição ao longo da curva é calculada com base no tempo decorrido e no total de tempo especificado para a animação. Esse valor é então usado como o parâmetro t da equação, determinando a nova posição do objeto no espaço.

Para calcular também a orientação do objeto ao longo da curva, foi implementada a extração do vetor tangente $\vec{T}$ à curva em cada ponto. Este vetor é calculado derivando a equação de Catmull-Rom relativamente a t. O vetor tangente permite construir uma base local para aplicar a transformação `glRotatef`, garantindo que o objeto se orienta corretamente ao longo do percurso. Este vetor, apenas é aplicado à matriz de translação caso o parametro 'align' do XML esteja marcado como 'true'. Caso contrário, apenas uma translação é executada para a nova posição cálculada no polinómio.

Esta funcionalidade proporciona animações mais realistas e visualmente agradáveis, sendo essencial para cenas em que o movimento não é linear, como simulações astronómicas ou percursos complexos em ambientes 3D.

4 Demo do Sistema Solar

Na demo para esta fase decidimos adicionar um cometa em órbita ao sol numa elipse e meter os planetas com as suas luas em rotação ao Sol.

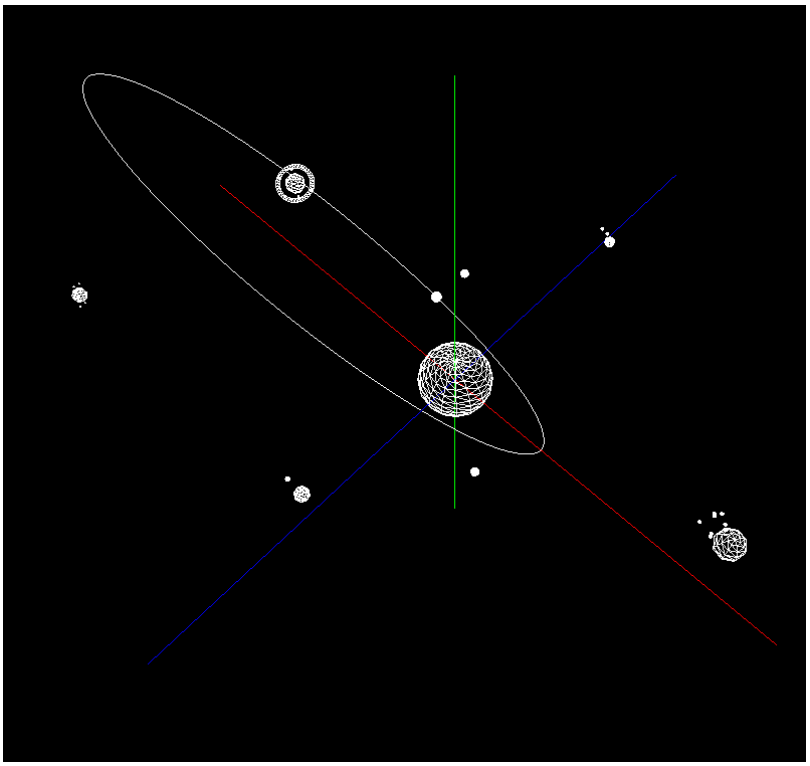


Figura 3: Demo do sistema solar desta fase

5 Referências

Referência utilizada para a geração dos pontos das superfícies cúbicas:

- https://pages.mtu.edu/ shene/COURSES/cs3621/NOTES/surface/bezier-construct.html