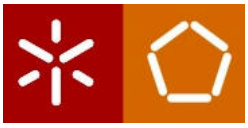

Universidade do Minho
Departamento de Informática

Computação Gráfica

Trabalho Prático - Fase 3

Grupo 5

João Coelho - A100596

José Rodrigues - A100692

Duarte Araújo - A100750

2024-04-26

Índice

1. Introdução 4
2. Patches de Bezier 4
3. Curvas Catmull-Rom 6
4. Rotação à volta de um eixo 8
5. VBOs 8
6. Demo do Sistema Solar 9
7. Resultados Obtidos 10
8. Conclusões 10

Lista de Figuras

Figure 1: Cálculo do ponto na curva de Bezier com u=0,1 5
Figure 2: Cálculo do ponto do Patch de Bezier com u=0,1 e v=0,6 5
Figure 3: Demo do Sistema Solar 9
Figure 4: “Resultados Obtidos nos Testes da Fase 3” 10

1. Introdução

Nesta fase do projeto o principal foco foi o aprimoramento da renderização 3D, possível através da melhoria das capacidades do nosso motor devido à implementação de curvas, VBOs e animações dinâmicas. Assim, esta fase poder ser caracterizada pela introdução de funcionalidades mais avançadas que permitem criar cenas mais dinâmicas e "vivas".

Os principais objetivos desta fase foram a estensão da nossa ferramenta de geração de modelos para suportar *Patches de Bezier*, porporcionando um meio mais versátil para a criação de superfícies complexas, e enriquecer o nosso motor com a capacidade de animar objetos ao longo de curvas cúbicas de *Catmull-Rom*. Além disso, adotámos VBOs para renderizar modelos, melhorando o desempenho e eficiência do nosso motor durante a renderização.

Ao longo deste relatório iremos aprofundar os detalhes técnicos da nossa implementação, bem como os desafios enfrentados e as decisões tomadas para superar tais desafios e, por fim, os resultados alcançados.

2. Patches de Bezier

Os *Patches de Bezier* são uma técnica fundamental em computação gráfica para a representação de superfícies complexas e suaves. Os mesmos permitem a criação de formas tridimensionais definindo um conjunto de pontos de controlo que influenciam a forma da superfície resultante, oferecendo uma abordagem flexível para a modelagem de objetos. Nesta secção exploraremos a implementação dos *Patches de Bezier* no nosso programa, apresentando os diversos detalhes técnicos, bem como os desafios enfrentados e as soluções adotadas durante o desenvolvimento.

Primeiramente, para calcular os pontos ao longo das curvas de Bezier, utilizamos a fórmula de Bernstein. Essa fórmula é derivada do Triângulo de Pascal e permite interpolar suavemente entre os pontos de controlo para obter um ponto na curva de Bezier para um dado parâmetro t.

$$b_0 = (1-t)^3$$

$$b_1 = 3t * (1-t)^2$$

$$b_2 = 3t^2 * (1-t)$$

$$b_3 = t^3$$

Os coeficientes de Bernstein b0, b1, b2 e b3 são calculados como combinações lineares dos termos (1 - t) e t, conforme definido pela fórmula de Bernstein. Esses coeficientes determinam a contribuição de cada ponto de controlo para o ponto final na curva de Bezier.

Ao aplicar os coeficientes aos pontos de controlo P[0], P[1], P[2] e P[3], obtemos o ponto na curva de Bezier correspondente ao parâmetro t.

$$p = P_0 * b_0 + P_1 * b_1 + P_2 * b_2 + P_3 * b_3$$

Assim sendo, cada *Patch de Bezier* é definido por uma matriz bidimensional de 16 pontos de controlo, organizados em 4 linhas e 4 colunas, onde cada linha representa uma curva de Bezier cúbica.

$$\begin{pmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{32} & P_{23} \\ P_{30} & P_{31} & P_{22} & P_{33} \end{pmatrix}$$

Para cada linha da matriz de pontos de controlo do *Patch de Bezier*, é calculado um ponto numa curva de Bezier cúbica para um determinado valor de u e esse ponto é então armazenado numa variável.

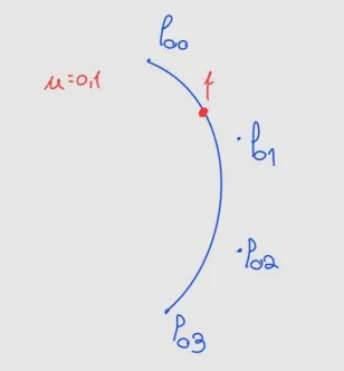

Figure 1: Cálculo do ponto na curva de Bezier com u=0,1

Após calcular os pontos ao longo das quatro curvas para o valor de u, os quatro valores calculados formam uma linha que será utilizada para calcular o ponto final do patch ao fornecer um certo valor v. Dessa forma, ao especificar um par de parâmetros u e v, podemos calcular um ponto do *Patch de Bezier* correspondente a esses parâmetros, representando uma coordenada na superfície do patch.

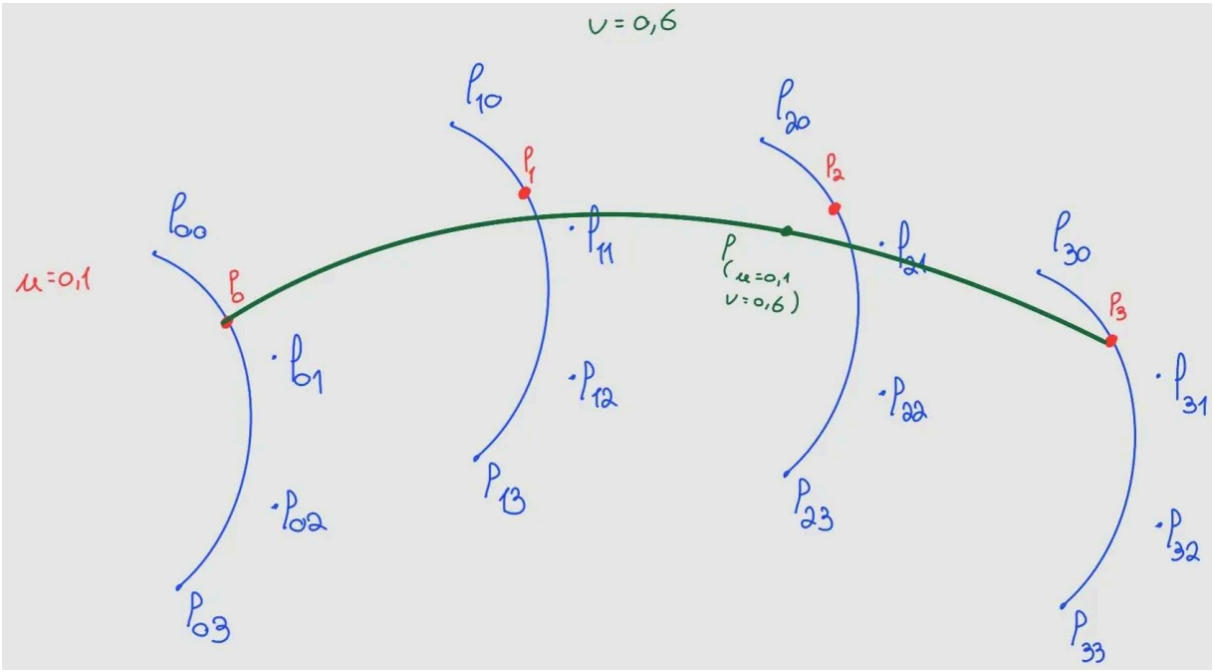

Figure 2: Cálculo do ponto do Patch de Bezier com u=0,1 e v=0,6

Este processo é repetido para cada ponto do patch, sendo que, por sua vez, os pontos calculados serão escritos no ficheiro de acordo com uma ordem que permita desenhar os quads recorrendo a dois triângulos.

Para além disso, ainda é importante referir o nível de tesselação, um valor que determina a suavidade e a resolução da superfície gerada pelos patches de Bezier. Esse valor é utilizado para dividir cada curva de Bezier em segmentos menores, permitindo um controlo mais firme sobre a aparência da superfície.

No processo de geração dos vértices do patch, o valor de tesselação é utilizado para definir o número de subdivisões ao longo de cada curva de Bezier. Isso significa que quanto maior o valor de tessellation, mais pontos serão calculados ao longo de cada curva, resultando numa superfície mais suave e detalhada. Por outro lado, um valor de tesselação menor pode resultar numa superfície mais simplificada, com menos detalhes.

O valor de tesselação também está diretamente envolvido no cálculo dos parâmetros u e v. Esses parâmetros determinam a posição ao longo das curvas de Bezier e são essenciais para calcular os pontos finais do patch. Ao dividir o intervalo [0, 1] num número específico de subdivisões determinado pelo valor de tesselação, podemos obter valores discretos para u e v, que são então utilizados no cálculo dos pontos do patch.

3. Curvas Catmull-Rom

As curvas *Catmull-Rom* são uma ferramenta poderosa na computação gráfica para criar trajetórias suaves e naturais, frequentemente utilizadas na animação de objetos e câmeras. Elas permitem definir uma curva suave através de uma série de pontos de controlo, proporcionando uma maneira flexível de animar objetos ao longo de caminhos complexos.

Nesta secção exploraremos a implementação das curvas *Catmull-Rom* no nosso programa, detalhando o processo de cálculo dos pontos ao longo da curva. Abordaremos os diversos aspetos técnicos envolvidos, desde o cálculo do parâmetro de tempo até a determinação das coordenadas da posição e da derivada ao longo da curva.

Para começar, o cálculo dos pontos da curva Catmull-Rom inicia-se com a determinação do parâmetro de tempo (t), que é crucial para definir a posição ao longo da curva. Esse parâmetro é calculado com base no tempo total da animação (valor definido no ficheiro XML) e no tempo decorrido desde o início da mesma. Isso garante que a animação progrida suavemente ao longo do tempo especificado.

```
float totalTime = transformation.values[0];
float elapsedTime = glutGet(GLUT_ELAPSED_TIME) / 1000.0f;
float timeParameter = fmod(elapsedTime, totalTime) / totalTime;
```

Uma vez que o parâmetro de tempo é calculado, os pontos de controlo relevantes são indexados. Isso é feito dividindo o parâmetro de tempo em partes iguais, cada uma correspondendo a um segmento da curva entre dois pontos de controlo consecutivos. Os índices dos quatro pontos de controle relevantes são então determinados com base no parâmetro de tempo e no número total de pontos de controle.

```
indices[0] = (index + nControlPoints - 1) % nControlPoints; // ponto de controlo anterior
indices[1] = (indices[0] + 1) % nControlPoints; // ponto de controlo atual
indices[2] = (indices[1] + 1) % nControlPoints; // ponto de controlo após o atual
indices[3] = (indices[2] + 1) % nControlPoints; // ponto de controlo após o do indices[2]
```

Com os pontos de controle identificados, as suas coordenadas são utilizadas para calcular a posição na curva *Catmull-Rom*. Isso é feito multiplicando os pontos de controlo pela matriz de Catmull-Rom. Essa

matriz contém os coeficientes que definem a curva e é aplicada aos pontos de controle para transformá-los nas coordenadas da curva.

$$\begin{pmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \end{pmatrix}$$

Dentro desse processo de cálculo, as coordenadas da posição e da derivada na curva *Catmull-Rom* são determinadas. A posição é calculada usando a fórmula de Catmull-Rom, que é uma combinação linear dos quatro pontos de controlo transformados. A derivada, por sua vez, é calculada com base na primeira derivada da fórmula de *Catmull-Rom*, que é uma função polinomial de segundo grau.

$$P(t) = a.t^2 + b.t^2 + c.t + d$$

$$P(t)' = 3 * a.t^2 + 2 * b.t + c$$

Onde a,b,c e d refletem a contribuição dos pontos de controle adjacentes ao ponto atual na curva.

Após calcular as coordenadas da posição e da derivada, estas são armazenadas e podem ser posteriormente utilizadas de diversas maneiras. No caso das coordenadas de posição, elas são frequentemente empregadas para controlar objetos animados ao longo da curva *Catmull-Rom*. Isso é feito aplicando uma translação das coordenadas da posição, movendo o objeto ao longo da trajetória definida pela curva.

Por outro lado, se o campo 'align' da transformação *Catmull-Rom* estiver definido como verdadeiro (True), as derivadas são utilizadas no cálculo do movimento do objeto. Isso garante que o objeto esteja sempre orientado na direção da curva.

Para realizar essa orientação, as derivadas são utilizadas para determinar a rotação do objeto ao longo da curva. Primeiro, o vetor derivada é normalizado para garantir que represente uma direção unitária tangente à curva naquele ponto. Esse vetor tangente é então utilizado como a direção principal de orientação, alinhando o objeto na mesma direção que a curva. O processo de normalização é feito da seguinte maneira:

$$|\vec{c}| = \sqrt{c_x^2 + c_y^2 + c_z^2}$$

$$\vec{c}_n = \frac{1}{|\vec{c}|} * \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix}$$

De seguida, são calculados vetores ortogonais adicionais para formar um sistema de coordenadas tridimensional que permitirá a rotação completa do objeto. O vetor "y0" é inicializado como o vetor unitário apontando para cima (0, 1, 0), e o produto cruzado entre o vetor tangente e "y0" resulta no vetor "z", que é ortogonal a ambos. Esses vetores são então normalizados para garantir que tenham a mesma escala que o vetor tangente.

Com os vetores de orientação definidos, uma matriz de rotação é construída, onde cada coluna representa um dos vetores de orientação, e a última linha é preenchida com zeros e um "1" no elemento diagonal final para garantir que a matriz seja uma matriz de transformação válida.

Por fim, essa matriz de rotação é aplicada ao objeto, utilizando-se a função `glMultMatrixf(matrix)`, o que resulta na rotação do objeto ao longo da curva, mantendo-o sempre alinhado com a sua direção de deslocamento.

4. Rotação à volta de um eixo

Para além da transformação responsável pelas curvas *Catmull-Rom* ainda era pedido nesta fase o desenvolvimento de um outro tipo de transformação, rotação de objetos em torno de um eixo. À semelhança da transformação previamente introduzida, esta também necessita de um valor temporal que equivale ao tempo total da animação (ou seja tempo total até uma rotação de 360º). Esta transformação é útil para criar animações onde os objetos giram em torno de um ponto fixo, sendo que, ao definir um ponto de rotação e um intervalo de tempo, a transformação calcula o ângulo de rotação com base no tempo decorrido.

```
float rotateTime = transformation.values[0];
float angle = (((float)glutGet(GLUT_ELAPSED_TIME) / 1000) * 360) / (float)(rotateTime);
```

De seguida, a transformação utiliza esse ângulo para girar o objeto em torno de um eixo específico. Isso é alcançado com a função `glRotatef`, que aplica uma rotação ao objeto, sendo necessário dar os valores do ponto de rotação e o ângulo.

```
glRotatef(angle, rotatePoint.x, rotatePoint.y, rotatePoint.z);
```

5. VBOs

Os *Buffers de Objeto de Vértice* (VBOs) são elementos fundamentais na renderização 3D em tempo real, proporcionando uma maneira eficiente de armazenar e aceder a dados de vértices de modelos 3D. Em resumo, um VBO é uma estrutura de dados que contém informações sobre os vértices de um objeto, como as suas coordenadas espaciais, cores e normais. Esses dados são transferidos para a GPU durante a inicialização do programa e são acessados diretamente pela GPU durante a renderização, o que resulta num desempenho significativamente melhor em comparação com a transferência de dados repetida entre a CPU e a GPU.

No contexto do nosso programa, os VBOs são utilizados para armazenar os dados dos vértices dos modelos 3D que queremos renderizar. Durante a fase de inicialização do programa, os dados dos vértices são carregados a partir dos ficheiros *.3d* e armazenados em VBOs. Isso é feito apenas uma vez para cada modelo, minimizando a sobrecarga de transferência de dados entre a CPU e a GPU durante a renderização.

Para começar, adicionamos um VBO à nossa estrutura de modelo, conforme mostrado abaixo:

```
struct Model {
    std::string filename;
    std::vector<Triangle> triangles;
    GLuint vbo;
};
```

De seguida, para cada modelo carregado, criamos um VBO para armazenar os dados dos vértices do modelo. Isso é feito usando as funções `glGenBuffers` e `glBufferData`, como podemos ver neste trecho de código:

```
while (inputFile >> x >> y >> z) {
    vertices.push_back({x, y, z});
}
inputFile.close();

glGenBuffers(1, &model.vbo);
glBindBuffer(GL_ARRAY_BUFFER, model.vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), vertices.data(),
GL_STATIC_DRAW);
```

Durante o processo de renderização, ativamos o VBO correspondente ao modelo que queremos renderizar usando glBindBuffer. Para além disso, habilitamos o estado de cliente para o array de vértices e especificamos o formato dos dados do vértice. Aqui está um trecho de código que ilustra isso:

```
glBindBuffer(GL_ARRAY_BUFFER, model.vbo);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(Vertex), nullptr);
glDrawArrays(GL_TRIANGLES, 0, model.triangles.size() * 3);
glDisableClientState(GL_VERTEX_ARRAY);
```

Essa abordagem permite-nos renderizar os triângulos do modelo diretamente a partir do VBO, minimizando a sobrecarga de transferência de dados entre a CPU e a GPU durante a renderização. Essa otimização resulta numa experiência de utilizador mais suave e eficiente ao lidar com modelos 3D complexos em tempo real.

6. Demo do Sistema Solar

A demo do Sistema Solar foi adaptada para utilizar as novas transformações desenvolvidas nesta fase do projeto. Essas transformações incluem o uso de curvas Catmull-Rom e rotações com tempo, que permitiram criar um ambiente mais realista do Sistema Solar.

Com as curvas Catmull-Rom, foi possível simular órbitas para os planetas e as suas luas. Agora, os planetas orbitam em torno do Sol, enquanto as luas giram em torno de seus planetas. Além disso, todos os corpos celestes continuam a girar em torno dos seus próprios eixos. Para além disso, foi ainda adicionado um cometa que utiliza os Patches de Bezier tal como era pedido no enunciado.

Segue uma imagem do modelo aprimorado:

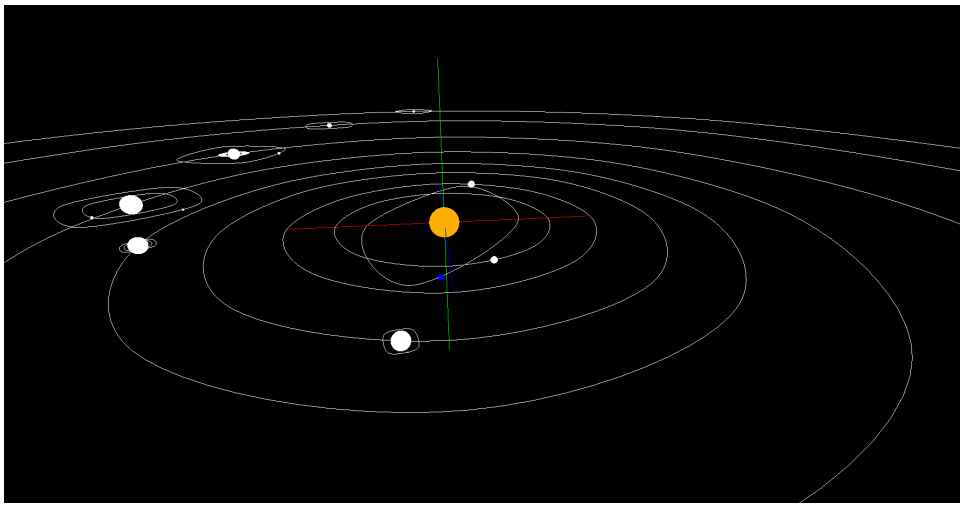

Figure 3: Demo do Sistema Solar

7. Resultados Obtidos

Após a implementação de todas as funcionalidades, uma série de testes foram realizados com o intuito de verificar o bom funcionamento das novas funcionalidades desenvolvidas. Os testes foram projetados para avaliar a eficácia das novas transformações, como as curvas de *Catmull-Rom* e as rotações com tempo, bem como a capacidade do sistema para simular um ambiente realista utilizando *Patches de Bezier*. Seguem-se os resultados dos testes:

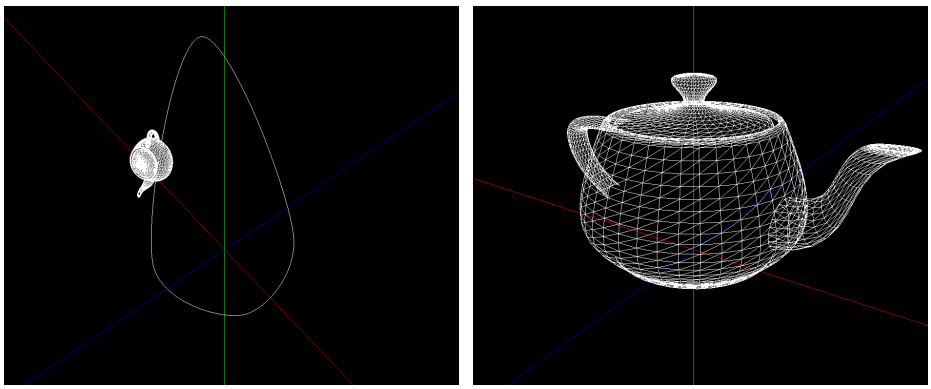

Figure 4: “Resultados Obtidos nos Testes da Fase 3”

Após a conclusão bem-sucedida dos testes da Fase 3 do projeto, podemos afirmar com confiança que o sistema atendeu aos requisitos estabelecidos e demonstrou funcionalidade consistente e integridade em relação às novas implementações. Os testes realizados para avaliar as novas transformações, como as curvas de Bezier foram executados com sucesso. Os modelos tridimensionais foram corretamente posicionados e animados de acordo com as especificações, refletindo a aplicação eficaz das técnicas desenvolvidas. Em suma, os resultados obtidos validam a conclusão satisfatória da Fase 3 do projeto, evidenciando a capacidade do sistema em lidar com simulações complexas de forma precisa e eficiente.

8. Conclusões

Após a implementação das novas transformações, dos VBOs e dos patches de Bezier, estamos satisfeitos por termos alcançado os objetivos estabelecidos para a Fase 3 do projeto. Essas funcionalidades representam avanços significativos na capacidade do sistema em lidar com ambientes 3D complexos e dinâmicos, contribuindo para uma experiência mais imersiva e realista.

A introdução dos patches de Bezier proporciona uma maneira flexível e poderosa de criar e animar superfícies curvas, permitindo a modelagem de formas mais complexas e orgânicas. As rotações com tempo agregam uma dimensão adicional à animação dos modelos, possibilitando movimentos suaves e naturais ao longo do tempo.

A utilização de VBOs para armazenar os dados dos vértices dos modelos 3D representa uma otimização crucial, minimizando a sobrecarga de transferência de dados entre a CPU e a GPU durante a renderização. Isso resulta numa experiência de utilizador mais suave e eficiente ao lidar com modelos 3D complexos em tempo real.

As curvas de Catmull-Rom oferecem uma maneira elegante de interpolar pontos em uma trajetória suave, sendo especialmente úteis na criação de caminhos de movimento para objetos animados. Essa técnica permite a criação de movimentos fluidos e realistas, contribuindo para a atmosfera geral do ambiente virtual.

No entanto, reconhecemos que ainda há áreas para melhorias e refinamentos nas próximas fases do projeto. Algumas sugestões incluem:

- **Melhorias na Interface do Utilizador:** Implementação do *Imgui* para aprimorar a interface do utilizador, tornando o sistema mais intuitivo e fácil de usar.
- **Otimização de Desempenho:** Identificar e resolver quaisquer gargalos de desempenho, como re-renders desnecessários, para garantir uma experiência fluida e responsiva para o utilizador.