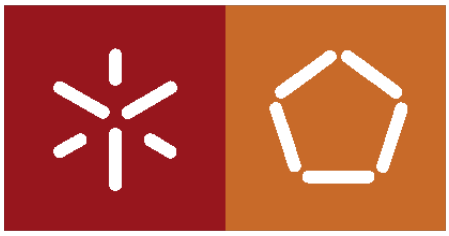

Universidade do Minho

MIEI - 3º ano - 2º semestre
Universidade do Minho

Trabalho Prático - Fase 3

Grupo 11


João Vieira
A76516


Manuel Monteiro
A74036


Bernardo Viseu
A74618


Fernando Pereira
A75496

April 20, 2019

Conteúdo

1 Introdução

1.1 Contextualização

No contexto curricular da disciplina de Computação Gráfica, foi nos proposto a criação de um mecanismo 3D com base num cenário gráfico, tendo por base todos os conhecimentos adquiridos até ao momento nas aulas e utilizando a linguagem de programação C++. Este trabalho prático está dividido em quatro fases, sendo este relatório referente à terceira fase, que tem como objetivo a criação de um modelo dinâmico do Sistema Solar recriado na segunda fase, com a adição de um cometa.

1.2 Resumo

Como foi referido, esta trata-se da terceira fase do projeto prático, é natural que se mantenham algumas das funcionalidades criadas nas fases anteriores, e por outro lado, de modo a cumprir com os requisitos necessários, algumas tiveram de ser alteradas.

Assim, esta fase traz consigo mudanças tanto ao nível do *engine* como também a nível do *generator*.

Começando pelo *generator*, este agora terá de criar um novo tipo de modelo baseando-se em *Bezier patches*. Este agora recebe um nome de um ficheiro de *input* no qual se encontram definidos os pontos de controlo dos vários *patches*, assim como um nível de tecelagem e, a partir destes, retorna um ficheiro contendo a lista de triângulos que definem essa superfície.

Já o *engine*, volta a sofrer alterações, pois tem de ter em conta outras funcionalidades. As translações e rotações presentes no ficheiro de XML também sofreram alterações. A translação será acompanhada por um conjunto de pontos, que irão definir uma curva *Catmull-rom*, assim como o tempo que o objeto irá demorar a percorrer a mesma. A rotação sofre uma alteração parecida, pois agora em vez de termos o ângulo de rotação, temos o período de rotação (tempo necessário para o objeto rodar 360 graus) em torno dos eixos escolhidos. Assim terá de ser alterada a maneira como o se processa toda a informação contida nesses ficheiros, para gerar o cenário pretendido. Outra modificação presente trata-se da maneira como são desenhados os modelos gráficos, agora com o auxílio de VBOs (*Vertex buffer object*), ao contrário da fase anterior em que eram desenhados de forma imediata.

Tudo isto tem a finalidade de gerar eficazmente um modelo do Sistema Solar ainda mais realista do que o elaborado na fase anterior, passando de um modelo estático para dinâmico com a implementação dos novos requisitos.

2 Arquitetura

Em relação à segunda fase do trabalho foram criadas novas classes, algumas classes foram alteradas e as restantes mantiveram-se inalteradas. Nesta secção só faremos menção às classes novas e às que foram modificadas.

2.1 Aplicações

Estas aplicações também marcaram presença na segunda fase, mas algumas tiveram de ser alteradas.

2.1.1 Gerador

generator.cpp: Como já foi explicado nas fases anteriores, esta é a aplicação responsável por definir as estruturas dos nossos modelos geométricos de modo a obter os respetivos vértices para que estes, posteriormente, possam ser renderizados pelo motor. As primitivas gráficas das fases anteriores foram mantidas mas foi necessário introduzir um novo método de construção de modelos com base em curvas de Bezier, sendo por isso necessário acrescentar ao gerador as novas funcionalidades.

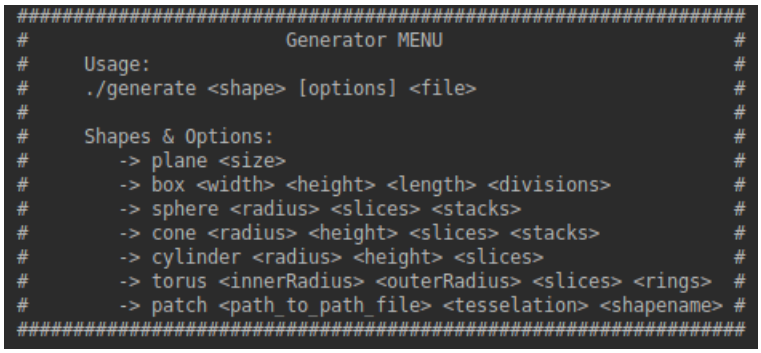

Figura 1: Menu de ajuda do Generator.

2.1.2 Motor

engine.cpp: Esta é a aplicação que permite visualizar as primitivas geométricas e interagir com elas através de comandos. Como a estrutura do ficheiro XML foi alterada, também foi necessário reajustar a aplicação de modo a realizar o parse com sucesso. Também foi necessário modificar o modo como as formas são renderizadas, para permitir o uso de *VBOs* passando a informação dos vértices para a placa gráfica o que resulta num melhor desempenho que é agora calculado a partir desta fase.

2.2 Classes

Todas classes da fase 2 também estão nesta fase, no entanto foi necessário acrescentar duas novas classes (Bezier.cpp e CatmullRom.cpp) de modo a ser possível responder às novas exigências do enunciado.

2.2.1 Bezier

bezier.cpp: Nesta classe é onde se encontram as funções que geram a superfície de Bezier, começando por ler o patch file. Este processo será explicado detalhadamente mais à frente no relatório.

2.2.2 Catmull-Rom

CatmullRom.cpp: Para definir e desenhar as trajetórias dos objetos, é necessário recolher do ficheiro xml os vários pontos de referência (necessitam de ser no mínimo 4) para gerar a curva de Catmull-Rom. Para guardar os pontos dados modificamos o parser das *translations*, utilizando uma nova variável `catmull_points` (vetor de Vertex). Depois disto são utilizados algoritmos de geração da curva completa de Catmull-Rom (definidos em `CatmullRom.cpp`). Os *Vertex* obtidos são então armazenados numa outra variável, `curve_points`.

2.2.3 Action

Action.cpp: Neste ficheiro estão definidas todo o tipo de ações disponíveis, ou seja `Translation`, `Scale`, `Rotation` e `Color`. Nesta fase do trabalho foi necessário modificar o parse da `Translation`, para se armazenar os pontos de Catmull-Rom dados no XML, e também o *time* que o objeto demora a realizar uma volta completa. Definimos também funções auxiliares que desenham a curva de Catmull-Rom completa. Por fim alteramos a maneira que as ações translation e *rotation* são aplicadas, o que é explicado melhor na secção de Catmull-Rom.

2.2.4 Shape

shape.cpp: Classe que contém a informação de cada um dos modelos a serem renderizados. Esta classe foi modificada para agora poder inicializar os *buffers* de cada objeto, a partir dos vértices processados nos ficheiros *.3d* e por sua vez desenha-los.

3 Gerador

3.1 *Bezier patches*

3.1.1 Leitura do *patch file*

Para entender como fizemos a leitura dos ficheiros patch, é necessário explicar primeiro o formato dos mesmos.

1. A primeira linha do ficheiro contém o número de patches.
2. As seguintes n linhas (em que n é o número de patches indicado anteriormente) contêm, para cada patch, uma lista de 16 números que representam os índices dos vários pontos de controlo de um dado patch.
3. A seguir a isto tem se uma linha com o número total de pontos de controlo.
4. Por ultimo aparecem os vários pontos de controlo de todos os patches, um ponto por linha, em que cada ponto é representado por três *floats*, separados por virgulas.

3.1.2 Processamento dos *patches*

O processamento de patches de Bezier é feito a partir do conceito de curvas de Bezier.

Curvas de Bezier são desenhadas a partir de 4 pontos (ou mais) definidos num espaço 3D, com coordenadas x, y e z. Estes pontos são designados pontos de controlo.

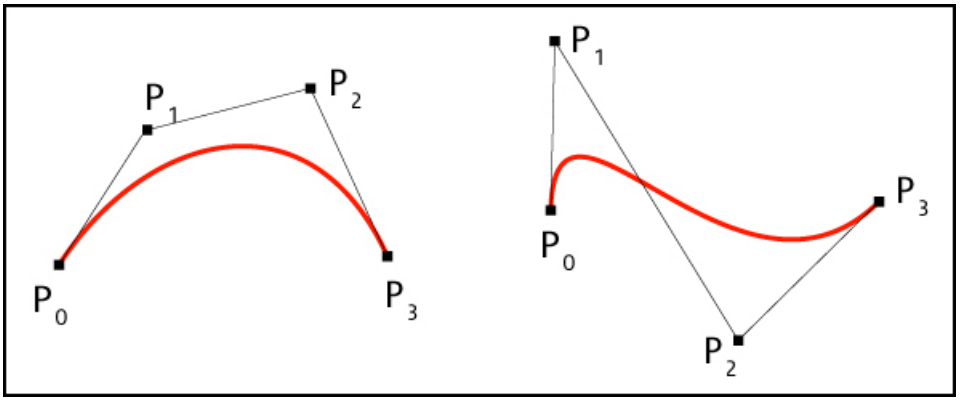

Figura 2: Exemplo de duas curvas de Bezier e os respetivos pontos de controlo.

Temos que a curva Bezier em si é definida por uma função, dada por um parâmetro, sendo este designado por *Tesselation* (representado por um t). Nas curvas de Bezier este t varia entre 0 e 1, ou seja qualquer valor de t no intervalo [0..1] representa um ponto da curva mo espaço 3D. Para se calcular a curva completa, é necessário calcular os pontos da curva à medida que incrementamos o valor de t.

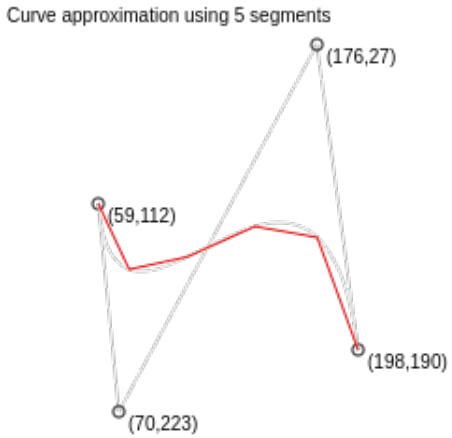

Figura 3: Curva de Bezier a mostrar uma aproximação com 5 segmentos (6 valores de t).

Como queremos uma aproximação mais precisa possível à curva de Bezier verdadeira, utilizamos valores muito pequenos para os intervalos de t. Para se calcular estes pontos, é utilizada esta equação:

```
P(t) = P0*k0 + P1*k1 + P2*k2 + P3*k3
```

Em que P0, P1, P2 e P3 são os pontos de controlo da curva, e k0, k1, k2 e k3 são os coeficientes utilizados para determinar o "peso" de cada ponto de controlo no cálculo de certo ponto da curva. Observamos que, para o ponto t = 0, dado que coincide com P0, temos que `k1=k2=k3=0`, e para o ponto t = 1 (que coincide com P4), `k0=k1=k2=0`.

Para o resto dos valores de t no intervalo `]0..1[`, os valores de k seguem estas equações:

```
k0 = (1 - t)^3
k1 = 3 * t * (1 - t)^2
k2 = 3 * t^2 * (1 - t)
k3 = t^3
```

A partir disto, podemos calcular os diferentes pesos para cada valor de t, e utiliza-los na equação dos pontos da curva, referido anteriormente. Isto é a base para se calcular curvas de Bezier.

Patches de Bezier diferem das curvas de Bezier no sentido de, utilizam se 16 pontos de controlo em vez de 4. Estes 16 pontos podem ser representados como uma grelha 4x4.

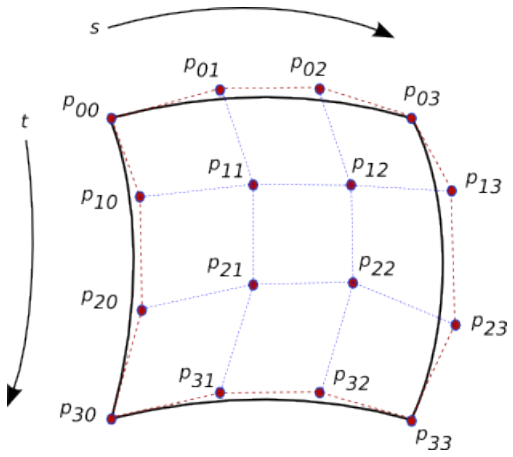

Figura 4: Exemplo de um patch de Bezier, demonstrando a matriz 4x4 com os pontos de controlo.

No caso das curvas de Bezier tínhamos apenas o parâmetro t, mas aqui utilizam-se dois: o u (ou s na Figura 3), e o v (ou t na Figura 3). Ambos estes parâmetros variam no intervalo `[0..1]` (como o t das curvas de Bezier).

Agora, para se calcular os pontos para cada conjunto de valores `(u,v)`, consideramos cada linha da matriz 4x4 como uma curva de Bezier, desta maneira, utilizamos apenas uma variável (u) para calcular o ponto correspondente a cada uma das curvas de Bezier do patch. Tendo em conta que a matriz 4x4 tem 4 curvas de Bezier, vamos então ter 4 pontos, que consideramos como pontos de controlo de uma curva de Bezier no sentido vertical (eixo v).

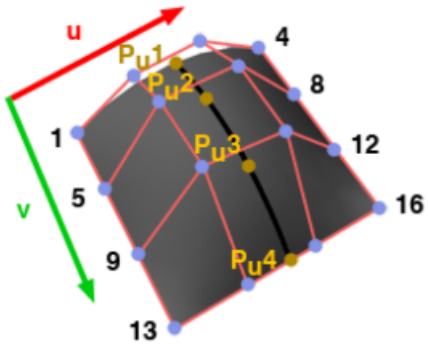

Figura 5: Demonstração dos pontos calculados neste passo.

Repetindo para esta nova curva no eixo v o mesmo processo que fizemos para o eixo u, vamos ter os pontos correspondentes à superfície da patch de Bezier, num dado par de valores (u, v).

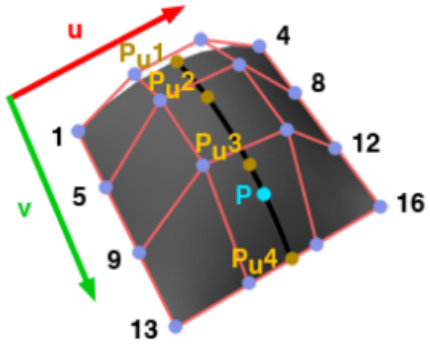

Figura 6: Ponto final calculado para um certo valor de u e v.

Depois disto já podemos desenhar a superfície completa, utilizando os pontos como vértices de triângulos. O numero de triângulos utilizados vai depender da tesselation dada no input, quanta maior tesselation, mais triângulos se utilizam, e consequentemente maior a qualidade da superfície desenhada.

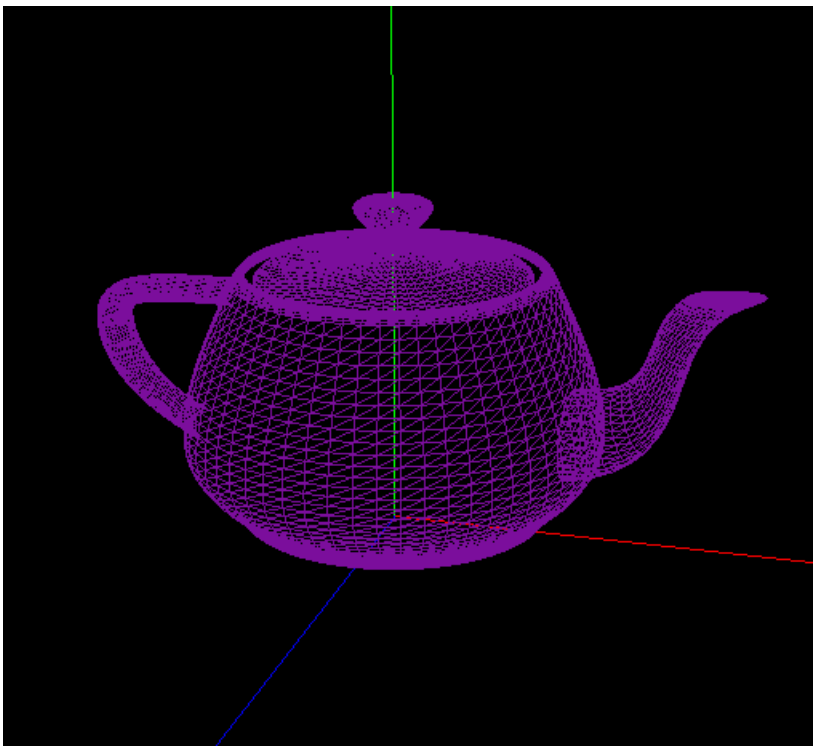

Figura 7: Teapot gerado com o patch de Bezier fornecido pelo professor.

4 Engine

4.1 *VBOs*

Tal como referido anteriormente, outra grande diferença desta fase para as anteriores é a utilização de *Vertex Buffer Objects* para desenhar todos os objetos, os quais anteriormente eram desenhados de modo imediato.

A utilização destes *buffers* permitem-nos inserir a informação dos vértices do objeto diretamente na placa de vídeo do nosso dispositivo, sendo isto uma funcionalidade oferecida pelo *OpenGL*.

O principal objetivo deste método é oferecer uma melhor *performance* aquela conseguida com o método de renderização imediato, pois a informação reside toda na placa gráfica em vez da memória do sistema, podendo desta forma ser renderizada pela mesma, diminuindo assim a sobrecarga do sistema. Assim os *frame per second* observados são inevitavelmente superiores àqueles que seriam observados utilizando o método das fases anteriores.

Em termos mais técnicos, para isto ser possível, tiveram de ser criados *arrays* de vértices para cada modelo que necessitamos de desenhar.

Já em código temos as funções, pertencentes à classe *Shape*, setUP() e draw(). A primeira é responsável por criar os *buffers* que terão a informação dos vértices. Já a última, trata de desenhar os objetos, já com a informação contida na placa de vídeo, fazendo uso da fnção *glDrawArrays* disponibilizada pelo *OpenGL*.

Já no motor, temos a função *initScene*, que prepara todos os modelos presentes no sistema, de modo que a função *render* só necessite de chamar a função *draw* para desenhar os objetos do sistema.

4.2 Curva *Catmull-Rom*

4.2.1 Rotação

Nesta fase do trabalho prático implementamos uma nova variável à ação Rotate: *time*.

Com esta variável podemos atribuir um tempo de rotação às shapes do sistema solar. O *time* representa o tempo que a shape demora a fazer uma rotação de 360 graus. Para aplicarmos isto no modelo, modificamos a maneira como aplicamos a rotação, para, no caso de o *time* ser diferente de zero:

```
t = glutGet(GLUT_ELAPSED_TIME) % (int) (time * 1000);
ang = (t * 360) / (time * 1000);
glRotatef(ang, x, y, z);
```

A variável t vai conter o tempo decorrido desde a invocação do glutInit, delimitado pelo *time* da rotação. Dado que `glutGet(GLUT_ELAPSED_TIME)` retorna em milissegundos, multiplicamos o *time* por 1000. A variável *ang* vai ter a quantidade que a shape deve rodar para o valor de tempo calculado antes. Depois disto aplicamos a rotação com esse valor. Caso seja dado uma rotação com $time = 0$, então apenas se aplica a rotação normalmente.

4.2.2 Translação

No caso das translações, estas também têm a variável *time* (representando o tempo que uma shape demora a fazer uma volta em torno do sol), e aqui utilizamos algoritmos para calcular curvas de Catmull-Rom, e, com o auxílio de novas variáveis uma sendo um vetor de Vertex `catmull_points` com os pontos dados para fazer a curva de Catmull-Rom, e outro vetor de Vertex denominado `curve_points`, que vai armazenar os pontos da curva de Catmull-Rom, depois de ser calculada.

5 Análise do Modelo

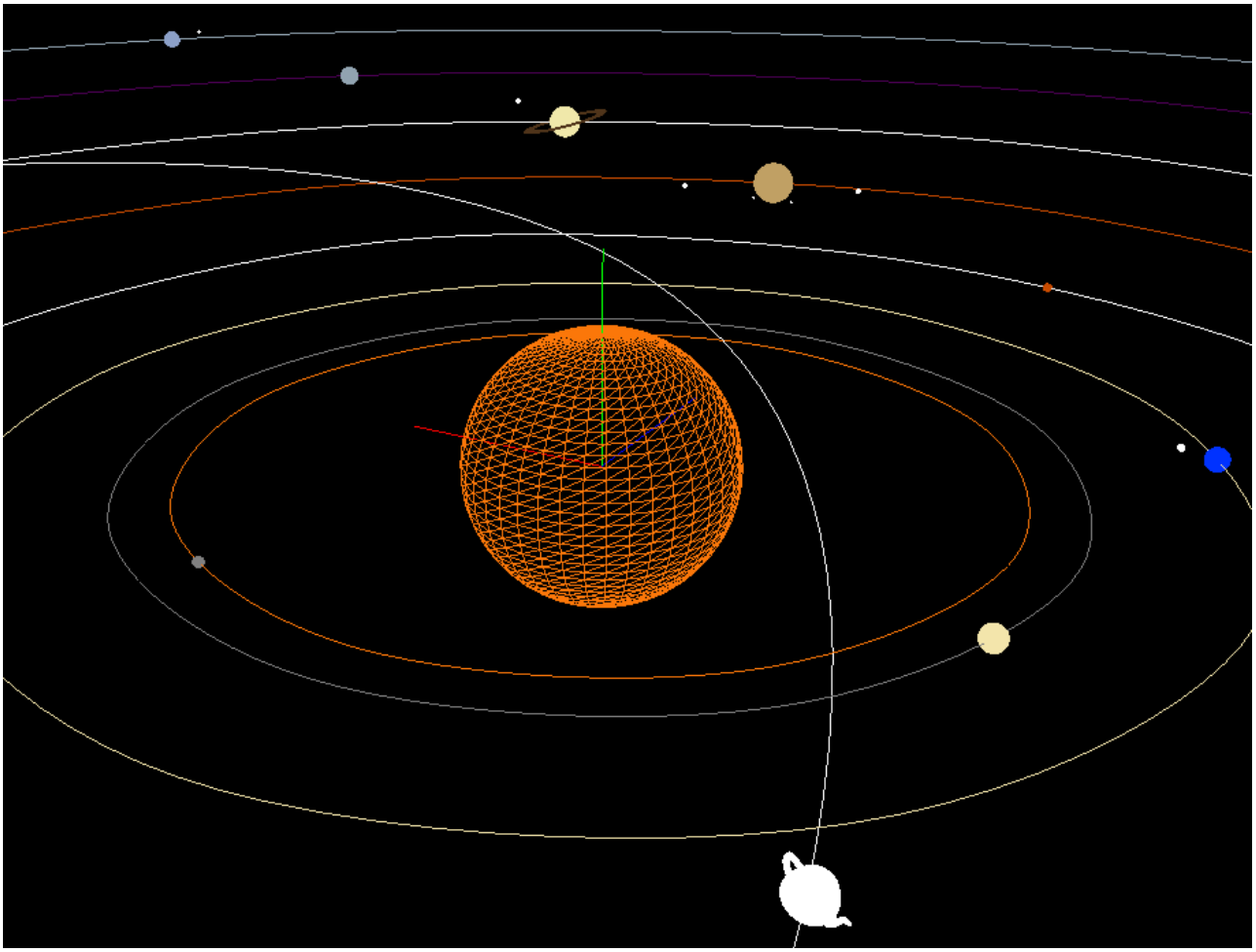

Figura 8: Visualização do Sistema Solar, com um cometa.

6 Conclusão

Esta fase do trabalho foi a mais trabalhosa até agora, pois tivemos de ganhar conhecimentos nas áreas das curvas e patches de Bezier, Catmull-Rom e também VBO's, e aplicar estes conhecimentos no nosso projeto. O facto de termos utilizado estas funções nas aulas práticas ajudou muito ao desenvolvimento desta fase. Quanto ao resto das mudanças, o facto de ser preciso modificar algumas partes do parser não foi grande dificuldade, e por fim conseguimos também modificar o XML do nosso Sistema Solar para um que representa bem a realidade, e achamos que mostra bem as novas funcionalidades do projeto.

Ficamos satisfeitos com o resultado final desta fase, e estamos ansiosos para completar a ultima fase do trabalho e termos o modelo completo e funcional.