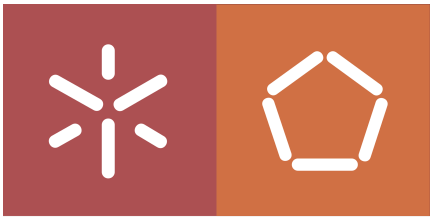


Universidade do Minho
Escola de Engenharia

Relatório da 3ª Fase do Projeto de

Computação Gráfica

Engenharia Informática
Universidade do Minho 2020/2021


Alexandre Gomes (a89549)


Carolina Oliveira (a89523)


Manuel Barros (a89560)


Maria Ramos (a89541)

Índice

1. Introdução

O trabalho a seguir apresentado foi realizado no âmbito da UC de Computação Gráfica do 2º semestre do 3º ano do curso de Engenharia Informática.

Nesta terceira fase do projeto foi nos proposta a implementação de diversas técnicas para enriquecer o tipo de cenas a construir, foi alterado o *generator* para permitir a criação de modelos baseados em *patches* de Bézier e foram implementados novos modos para rotação e translação tirando proveito de curvas de Catmull-Rom.

Foi também melhorada a *demo scene* do sistema solar criada na fase anterior, passando agora este a ser dinâmico.

2. Funcionalidades Obrigatórias

2.1. Bézier Patches

Nesta fase foi nos pedido que criássemos um novo tipo de modelo baseado em Bézier *patches*, o primeiro passo foi criar um novo ficheiro (bezier.cpp) para conter todas as funções que envolvem a construção deste modelo.

O primeiro passo foi ler o ficheiro de entrada e colocar todos os pontos num vetor e todos os *patches* (compostos por índices) numa matriz, usamos a biblioteca de regex para nos facilitar o processo.

$$p(u,v) = [u^3 \quad u^2 \quad u \quad 1]M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Analisando a fórmula de cima que calcula uma coordenada de um ponto dentro de uma Bézier *patch*:

- P00 a P11 representam os 16 pontos de controlo que nos são dados no ficheiro inicial
- M é a matriz correspondente à curva que queremos representar neste caso será a de Bézier $\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$
- M^T é a matriz transposta da anterior como esta é triangular então $M^T = M$
- Os valores de u e v (ambos compreendidos de 0 a 1) correspondem à posição do ponto dentro do *patch* em questão, quanto maior for a tesselação escolhida menor será o valor entre u e v adjacentes ficando a figura com um maior número de pontos e consequentemente com maior qualidade.

Temos de ter em atenção que temos de aplicar a fórmula para cada coordenada e para cada combinação de u e v, se tivermos por exemplo 2 *patches* com nível de tesselação 3 seria necessário aplicar a fórmula 96 vezes ($2(patches)$ * $3(coordenadas)$ * $(3+1)^2$ $(tesselação)$) o que corresponde a 32 pontos (16 por *patch*).

Tendo em conta a fórmula analisada foram codificadas as funções que efetuam todos os cálculos mostrados acima, primeiramente é usada a função calculatePatches para calcular todos os pontos dos que serão necessários no modelo, para cada *patch* é

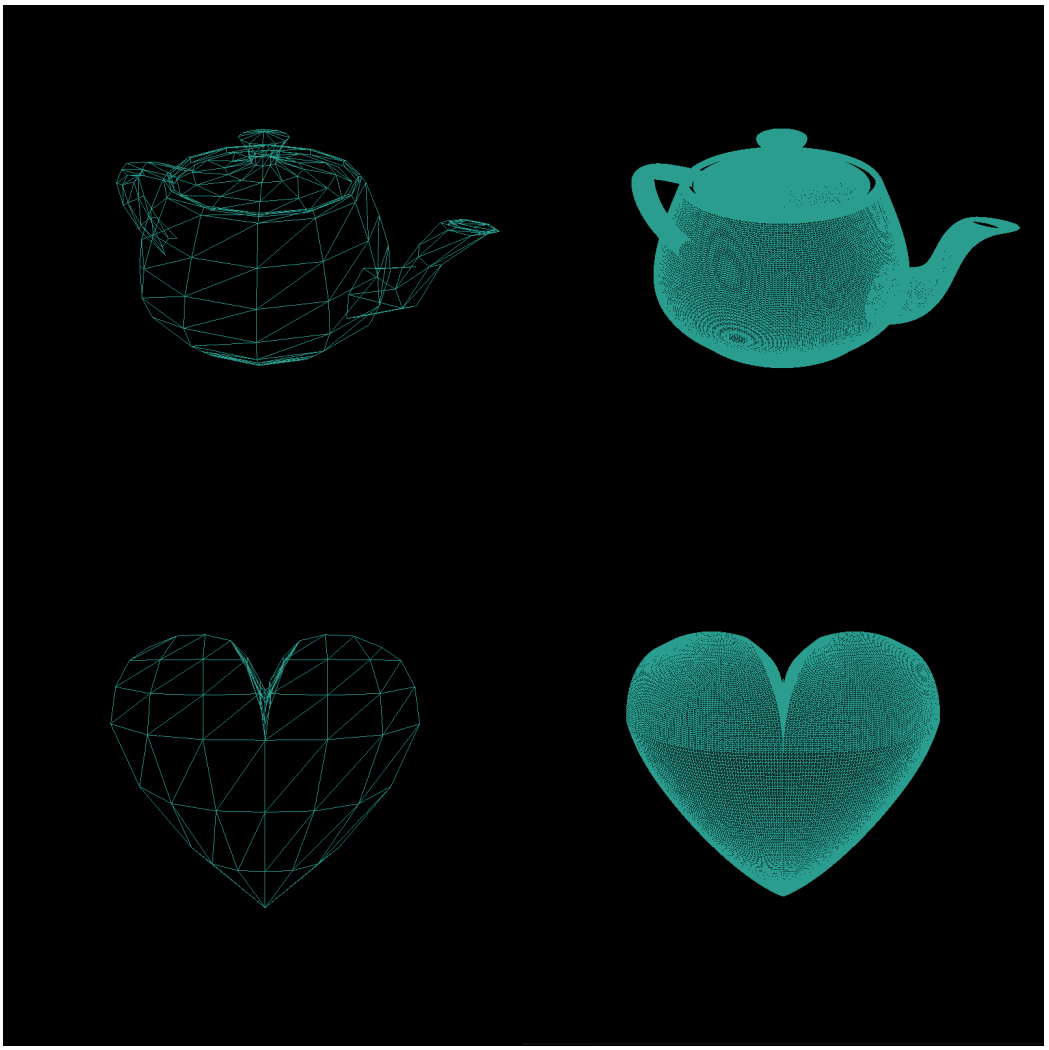
calculado $M * P * M^T$ sendo P a matriz dos pontos do controlo do *patch* em questão, a partir da função preCalculate que para além de fazer esta conta também é responsável por fazer a correspondência entre a matriz de índices e o vetor de pontos, este processo é efetuado para cada uma das coordenadas (x y z).

De seguida são efetuados dois ciclos *for* que abrangem todas as combinações possíveis entre u e v, no fim de cada iteração o ponto calculado é colocado num vetor de pontos pronto a ser imprimido no ficheiro output.

Seguidamente é usada a função calculateIndexes para calcular os índices necessários para fazer a triangulação entre os pontos determinados anteriormente, esta função é bastante simples sendo que apenas é necessário saber o número de *patches* e nível de tesselação para determinar a posição dos índices no ficheiro final.

Por fim basta imprimir os vectores de pontos e índices determinados no ficheiro final dado pelo utilizador.

De seguida mostramos o modelo de um bule de chá (gerado a partir dos pontos de controlo fornecidos pelos docentes) e de um coração (gerado a partir de pontos de controlo determinados pelos alunos), para níveis de tesselação 3 e 64.



O comando a usar para efetuar a geração de um modelo de Bézier *patches* necessita da identificação do tipo (bezierPatches), nível de tesselação ficheiro com os pontos de controlo e ficheiro de saída por esta ordem.

```
/Generator ./generator bezierPatch teapot.patch 3 bezierTeapotS.3d
```

2.2. Rotação com tempo

Fazer a rotação com tempo em torno de um eixo foi bastante simples, primeiramente fizemos com que a transformação de rotação criada em fases anteriores aceitasse mais parâmetros como o tipo de rotação (por tempo ou estático), o tempo necessário para efetuar uma rotação de 360º e o tempo desde o ínicio do programa até à frame anterior.

De seguida foi necessário calcular o ângulo de rotação a cada instante para isto foi usada a função glutGet(GLUT_ELAPSED_TIME) para descobrir o tempo passado desde o início do programa até ao instante atual, ao subtrair este pelo tempo passado até ao instante da frame anterior descobrimos o Δt.

Por fim é apenas necessário aplicar uma regra de três simples, multiplicando o Δt por 360º e dividindo pelo tempo total de uma rotação para obter o ângulo a rodar no instante em questão, este é somado à variável ângulo da classe que armazena o ângulo atual do objeto (este passo é necessário visto que depois de uma rotação o estado da matriz de transformação volta ao normal devido ao glPopMatrix que é efetuado eventualmente), só nos falta então efetuar a rotação com o novo ângulo.

```
if (this->rotating){
    float newTime = glutGet(GLUT_ELAPSED_TIME);
    float deltaTime = (newTime - this->oldTime)/1000.0f;
    this->oldTime = newTime;

    if (time>0){
        (this->angulo)+=(deltaTime*360.0f/time);
        glRotatef(this->angulo,x,y,z);
    }
```

2.3. Translação a partir de curvas de Catmull-Rom

Dado um conjunto de pontos necessários para definir uma curva de Catmull-Rom e dado o tempo que um dado objeto deve demorar para a percorrer é possível definir translações ao longo do tempo.

Para implementar esta funcionalidade primeiro é necessário permitir que seja possível definir um conjunto de pontos (no mínimo quatro) quando se declara uma translação no ficheiro xml.

$$p(t) = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Primeiro passo é descobrir um dado ponto em qualquer instante da curva, para determinar este ponto vamos analisar a fórmula acima

- A variável t (compreendida entre 0 e 1) indica-nos a posição ao longo da curva, 0 é o início e 1 é o fim, o valor de t neste caso vai depender do tempo que foi indicado para o objeto percorrer a curva, a matriz, podemos calcular o vetor tangente do ponto se substituirmos o vetor t pelas suas derivadas $T = [3t^2 \quad 2t \quad 1 \quad 0]$
- A matriz no meio da equação representa as curvas de Catmull-Rom
- Por fim temos o vetor de pontos P0 a P3, para encontrar estes pontos temos de descobrir quais são os dois pontos P1 e P2 sobre o qual Pt se encontra compreendido, depois temos de escolher mais um ponto P0 anterior a P1 e uma ponto P3 posterior a P2.

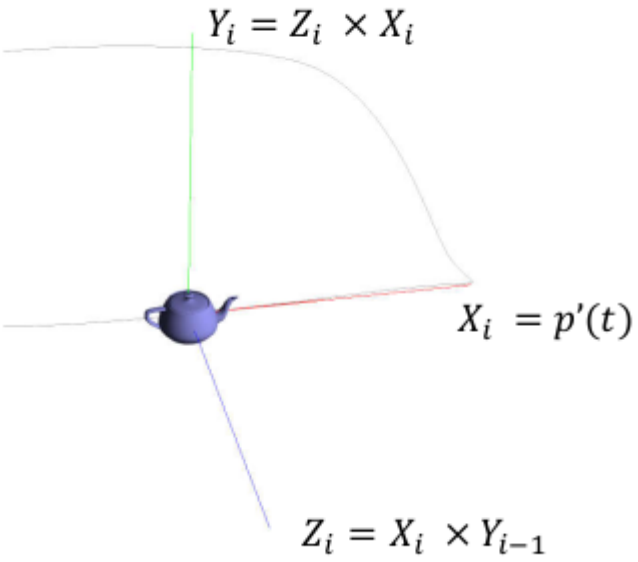
Temos de ter em atenção que temos de aplicar a fórmula para cada coordenada de P0 a P3.

Foi criado um novo módulo catmullRomCurve para abranger as próximas funções.

Tendo em conta a fórmula analisada foi declarada a função getGlobalCatmullRomPoint que dado um t descobre o valor de P0 a P3 e passa-os como parâmetro para a função getCatmullRomPoint, que aplica a fórmula a analizada acima para ambos os vetores t e t' dando-nos assim as coordenada do ponto a determinar e o seu vetor tangente.

A tangente tem um papel importante pois serve para calcular a orientação do objeto ao longo da curva.

A Partir destas funções já é possível desenhar a curva, esta funcionalidade não está aparente desde o início da aplicação mas se o utilizador usar a tecla ‘m’ todas as curvas de catmull-rom que foram declaradas vão ser desenhadas no ecrã.



$$\overrightarrow{X_i} = p'(t)$$
$$\overrightarrow{Z_i} = X_i \times \vec{Y}_{i-1}$$
$$\overrightarrow{Y_i} = \overrightarrow{Z_i} \times \overrightarrow{X_i}$$

$$M = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Analisando a fórmula e gráfico acima descobrimos qual a nova matriz de transformação que cada ponto deve ter em qualquer ponto da curva.

O vetor Xi é muito fácil de obter visto que é o vetor tangente calculado anteriormente.

Para calcular Zi temos de ter informação sobre o vetor Yi-1 ou seja o vetor Yi na última iteração da função, para resolver este problema foi criar uma variável na classe para a transformação de translação que guardará este valor, (é necessário que este tenha uma valor inicial).

Para calcular o novo Yi basta fazer o produto entre Xi e Yi-1 calculados anteriormente.

É de notar que todos os vetores devem estar normalizados, e que a matriz M resultante tem de ser uma matriz orientada por colunas.

Depois de calcular a matriz esta é multiplicada pela matriz de transformação atual.

Para podermos usar estas funções primeiro tem de se calcular o t atual, como dito acima este depende do tempo que um objeto deve demorar a percorrer uma curva, e o módulo de cálculo é bastante parecido ao do cálculo do ângulo para as rotações.

Primeiramente é necessário descobrir o Δt este processo é idêntico ao mostrado anteriormente, depois divide-se o Δt pelo tempo total que o objeto demora a percorrer a curva e temos o valor de t a incrementar ao t atual este é armazenado na classe da transformação.

2.4. Implementação de VBO's

Foi feita a implementação de VBO's a partir de um sistema de pontos mais índices, o primeiro passo para esta implementação de foi mudar a classe figura3d criada anteriormente, agora esta contém três variáveis dois indicadores para os arrays a ser criados em memória gráfica e um inteiro para o número de índices no array de índices.

A implementação foi bastante fácil sendo apenas necessário guardar os pontos e índices nos respectivos arrays e ligá-los à memória gráfica a partir das funções no módulo de opengl.

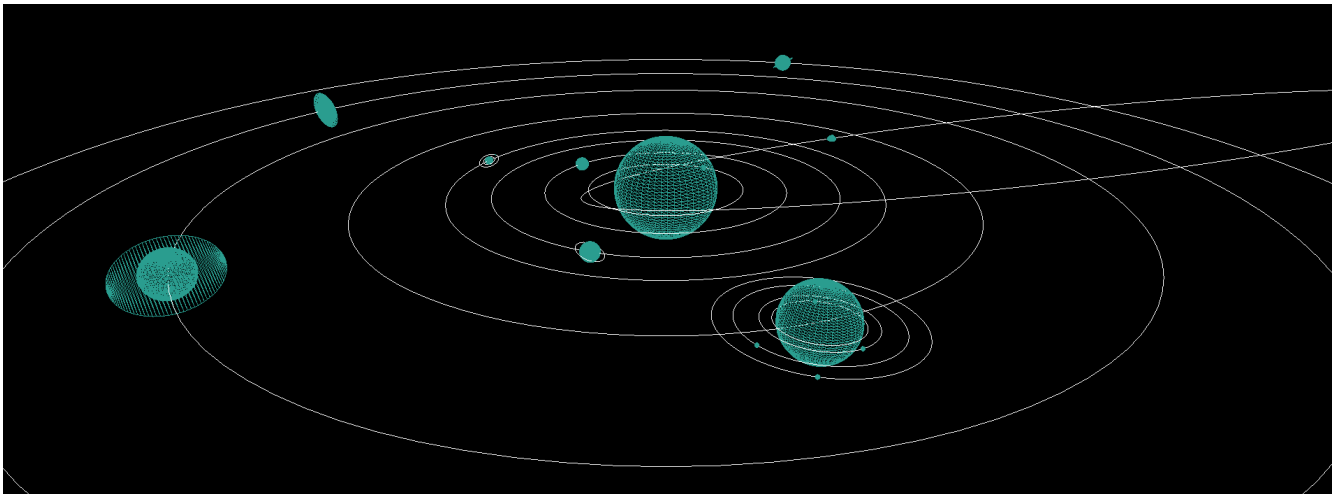
2.5. Sistema Solar Dinâmico

O primeiro passo para obtermos um sistema solar dinâmico foi adicionar rotações a todos os planetas, luas e sol, todos estes rodam na mesma direção exceto vênus.

Visto que quase todos os planetas e luas têm órbitas bastante próximas do circular foram calculados (a partir de um programa auxiliar) pontos necessários para definir cada curva de catmull-rom que corresponde a um círculo em que o raio é a distância de cada planeta ao sol.

Os tempos de rotação e translação foram baseados nos reais, mas não estão à escala.

Para o cometa foi usado o modelo coração mostrado anteriormente que foi formado por bezier *patches* cujos pontos de controlo foram calculados pelos alunos, este tem uma trajetória elíptica.



3. Funcionalidades Extras

3.1. Setup em XML

Foi adicionada a possibilidade de mudar as definições da cena num novo grupo xml chamado setup, este grupo tem de ser chamado no início do ficheiro antes da scene.

Até ao momento é possível selecionar se é chamada a função glutIdleScene, o raio e ângulos beta e alpha iniciais da câmara e o tamanho da janela.

```
<setup>
    <window width="1600" height="900" />
    <iddleFunc on="true" />
    <camera B="22.5" R="200" />
</setup>
```

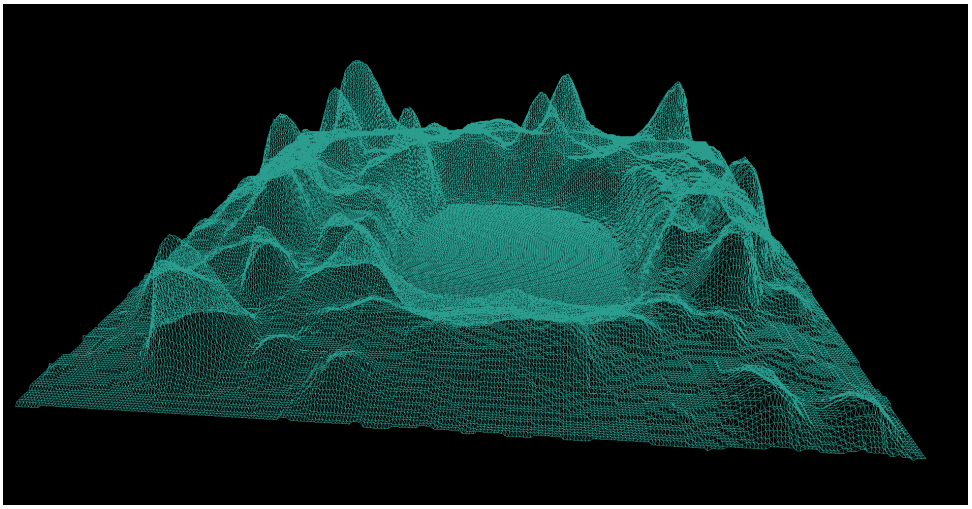
3.2. Geração de Terrenos

Agora é possível criar um novo tipo de modelo baseado em mapas de alturas, basta correr a aplicação, dizer que se trata de um terreno e dizer o caminho para a imagem.

```
Generator ./build/generator terrain terreno.jpg terrain.3d
```

Visto que o terreno vai ser desenhado com GL_TRIANGLE_STRIP o ficheiro output apenas conterá os pontos a desenhar e não índices.

O modo a implementar em xml é igual, sendo apenas necessário chamar o ponto 3d com a instrução “model”.



4. Conclusão

Enquanto que nas primeiras fases aprendemos os básicos como geração de modelos a partir de diversas primitivas e aplicação de transformações geométricas para a construção das diversas cenas, esta terceira fase do trabalho possibilitou-nos não explorar e aplicar diversas técnicas para nos permitir criar cenas mais sofisticadas como também melhorar a performance da nossa aplicação mudando a maneira como a imagens são processadas.

Verificamos também que a maior parte do nosso tempo foi passado a entender a matemática que estava por detrás das novas técnicas a implementar, sendo que depois a sua implementação se tornou mais fácil.