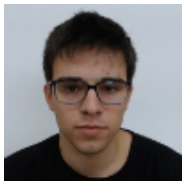
RELATÓRIO PRÁTICO FASE 3

GRUPO 30

Ana Almeida, A83916

André Figueiredo, A84807



Luís Ferreira, A86265

Rafael Lourenço, A86266

Mestrado Integrado em Engenharia Informática 2019/2020

Computação Gráfica

Universidade do Minho

4 de Maio de 2020

Conteúdo

1 Introdução **3**
- 1.1 Contextualização . . . 3
- 1.2 Resumo . . . 3

2 Arquitetura do código **4**
- 2.1 Aplicações . . . 4
  - 2.1.1 Gerador . . . 4
  - 2.1.2 Motor . . . 5
- 2.2 Classes . . . 5
  - 2.2.1 Base Element . . . 6
  - 2.2.2 Translate Element . . . 6
  - 2.2.3 Rotate Element . . . 6
  - 2.2.4 Scale Element . . . 6
  - 2.2.5 Model Element . . . 6
  - 2.2.6 Container Element . . . 7
  - 2.2.7 Scene Element . . . 7
  - 2.2.8 Group Element . . . 7
  - 2.2.9 Models Element . . . 7
  - 2.2.10 Patch . . . 7
- 2.3 Ficheiros Auxiliares . . . 8
  - 2.3.1 Figures . . . 8
  - 2.3.2 Camera . . . 8

3 Generator **9**
- 3.1 *Bézier Patches* . . . 9
  - 3.1.1 Ficheiro Input . . . 9
  - 3.1.2 Processamento dos patches . . . 9

4 Engine **12**
- 4.1 VBOs . . . 12
  - 4.1.1 Model Element . . . 12
- 4.2 Curva Catmull-Rom . . . 13
  - 4.2.1 Translate . . . 13
- 4.3 Rotate . . . 14

5 Resultados obtidos **15**
- 5.1 Sistema Solar - Visualização . . . 15

6 Conclusão **19**

Capítulo 1

Introdução

1.1 Contextualização

Foi-nos proposto, no âmbito da UC Computação Gráfica, a criação de um motor gráfico genérico para representar objetos a 3 dimensões, com o auxílio de duas ferramentas - *OpenGL* e *C++*.
O projeto está dividido em várias fases distintas, sendo que nesta terceira serão incluídas curvas, superfícies cúbicas e VBOs, com o objetivo de criar um modelo dinâmico do Sistema Solar com adição de um cometa (feito usando os pontos de controlo fornecidos para o *teapot*).

1.2 Resumo

Visto que esta se trata da terceira parte do projeto é natural que algumas das funcionalidades criadas na primeira e segunda partes se mantenham inalteradas, por outro lado, algumas delas foram alteradas de modo a cumprir os requisitos necessários, alterações estas que abrangem tanto o *generator* como o *engine*.
Primeiramente vamos olhar para o *generator*. Nesta fase passou a suportar a criação de um modelo geométrico baseado em *patches* de *Bézier*. Para isso, passa a receber como parâmetros o valor de tecelagem, *tessellation* e o ficheiro de *input*, que contém os pontos de controlo dos *patches* e, por fim, o ficheiro de *output*.
Passando agora para o *engine* que também sofreu várias alterações e tem novas funcionalidades. Os elementos *rotate* e *translate* presentes no *XML* foram alterados, pois passaram a ser dinâmicos. No caso do *translate*, este passou a conter o tempo necessário para completar uma translação e uma lista de pontos que definem a órbita/rota a percorrer usando uma curva de *Catmull-Rom*. Já no caso do elemento *rotate*, este passou também a receber o vetor e, em vez de um ângulo, o tempo que este demora a concluir uma rotação de 360^{o} em torno desse mesmo vetor. Estas duas pequenas alterações permitem criar animações dinâmicas que permitem, no nosso caso, criar um modelo do Sistema Solar.
Deste modo, teve de ser alterado não só o *parser*, mas também o modo como é processada a informação recebida, de forma a gerar o cenário da forma pretendida. Por fim, foi também alterada a forma como os modelos são desenhados, que passaram a sê-lo através de *VBOs*, ao contrário da fase anterior, em que eram desenhados de forma imediata. Foram ainda adicionadas funcionalidades no que toca à manipulação do tempo, visto que agora se tratam de animações dinâmicas.
Posto isto, considerámos que todas estas alterações nos permitiram gerar, de forma realista, o modelo do Sistema Solar proposto.

Capítulo 2

Arquitetura do código

Tendo em mente a continuação do trabalho desenvolvido na fase anterior, mantivemos as duas aplicações principais previamente desenvolvidas, *generator* e *engine*, alterando, contudo, alguma parte do código destes e, ainda, acrescentando novas partes, tendo em vista o cumprimento dos requisitos necessários propostos para esta fase.

2.1 Aplicações

Nesta secção são apresentadas as aplicações fundamentais que permitem gerar e exibir os diferentes cenários pretendidos. Conseguimos cumprir e concluir os requisitos propostos para esta fase, uma vez que se verificaram alterações significativas em ambas as aplicações.

2.1.1 Gerador

generator.cpp - Tal como explicado em fases anteriores, esta é a aplicação onde estão definidas as estruturações das diferentes formas/modelos geométricos a desenvolver de forma a gerar os respetivos vértices, para que, mais tarde, possam ser renderizados pelo motor (*engine*). Com a realização desta fase e para além das primitivas gráficas anteriormente desenvolvidas foi introduzido um novo método de construção de modelos com base em curvas de *Bézier*. Assim, foi necessário acrescentar ao gerador novas funcionalidades relativamente às fases anteriores.

```
#_____________________________ HELP ____________________________________#
|                                                                        |
|   Usage: ./generator {COMMAND} ... {OUTPUT FILE}                       |
|                      [-h]                                              |
|                                                                        |
|   COMMANDS:                                                            |
| - plane [SIZE]                                                         |
|      Creates a square in the XZ plane, centred in the origin.          |
|                                                                        |
| - box [SIZE X] [SIZE Y] [SIZE Z] [DIVISION X] [DIVISION Y] [DIVISION Z] |
|      Creates a box with the dimensions and divisions specified.        |
|                                                                        |
| - sphere [RADIUS] [SLICE] [STACK]                                      |
|      Creates a sphere with the radius, number of slices and            |
|      stacks given.                                                     |
|                                                                        |
| - cone [RADIUS] [HEIGHT] [SLICE] [STACK]                               |
|      Creates a cone with the radius, height, number of slices          |
|      and stacks given.                                                 |
|                                                                        |
| - torus [RADIUS] [WIDTH] [HEIGHT] [SLICE] [STACK]                      |
|      Creates a cone with the radius, width, height, number of slices   |
|      and stacks given.                                                 |
|                                                                        |
| - patch [TESSELLATION LEVEL] [INPUT FILE]                              |
|      Creates a new type of model based on Bezier patches.              |
|                                                                        |
|   OUTPUT FILE:                                                         |
| In the file section you can specify any file in which you wish         |
| to save the coordinates generated with the previous commands.          |
|                                                                        |
#________________________________________________________________________#
```

Figura 2.1: Menu de ajuda do Generator

2.1.2 Motor

engine.cpp - Aplicação que possui a capacidade de apresentar a janela, exibindo os modelos indicados no ficheiro *ss.xml* e interagir com o ambiente. Houve novamente uma alteração no ficheiro *XML*, desta vez mais pequena. As alteracões foram feitas no *translate* e *rotate*, o que levou a uma pequena modificação na função *parseXML()* para tratar desses novos casos.

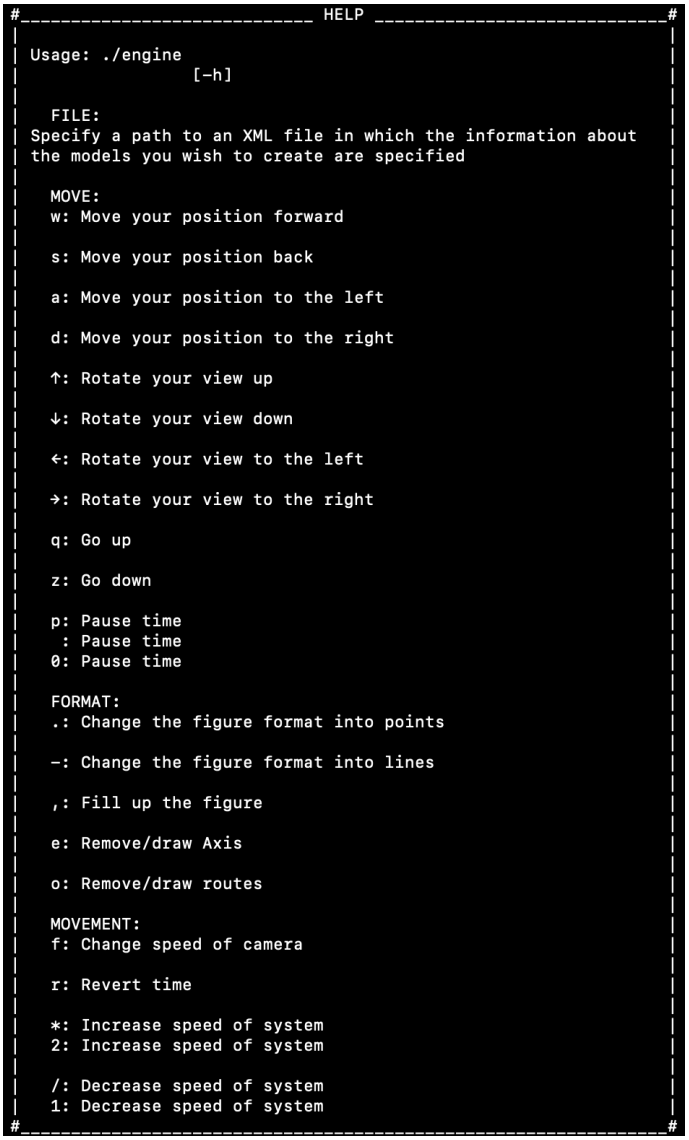


Figura 2.2: Menu de ajuda do Engine

2.2 Classes

A realização desta fase não levou à criação de mais nenhuma classe, antes pelo contrário, foi removida uma das classes auxiliar *triangle.cpp*. Foram então alterados e criados métodos nas classes já existentes. Deste modo, a tabela de classes mantém-se igual à última fase:

- Interface *BaseElement* que todos os outros implementam;
- *TranslateElement*;
- *RotateElement*;

- *ScaleElement*;
- *ModelElement*;
- Classe abstrata *ContainerElement*;
- *SceneElement*;
- *GroupElement*;
- *ModelsElement*.

2.2.1 Base Element

baseElement.h - Interface que todos os outros elementos terão de implementar, herdando o método *apply(float t, bool orb)* usado para renderizar os respetivos elementos. Foram acrescentados o *float t* e *bool orb* como argumentos para serem tratados pelas classes *TranslateElement* e *RotateElement*.

2.2.2 Translate Element

translateElement.h - Classe que armazena toda a informação necessária à execução de um *translate*. Esta foi a classe mais modificada, pois foi preciso adaptá-la aos novos inputs do ficheiro *XML*. Não só foram mantidas as variáveis de instância antes definidas, x, y e z, como foram acrescentadas as variáveis *float time*, *float **curvePoints* e ainda *float pos[nCurvePoints][3]*, assim como 6 novos métodos para o tratamento dos novos *translates* usando a *spline* de Catmull-Rom. Esta situação será explicada com mais detalhe na secção 4.2.1.

2.2.3 Rotate Element

rotateElement.h - Classe que guarda os atributos *angle*, *axisX*, *axisY* e *axisZ* necessários para a invocação da função OpenGL *glRotatef(angle, axisX, axisY, axisZ)*, realizada no método *apply(float t, bool orb)* que é por esta classe implementado. Nesta fase foi acrescentada uma variável *float time*, de modo a calcular o ângulo de rotação correspondente a esse tempo.

2.2.4 Scale Element

scaleElement.h - Classe que guarda os atributos x, y e z necessários para a invocação da função OpenGL *glScalef(x, y, z)* realizada no método *apply* que é por esta classe implementado. Esta classe, excetuando a adaptação à nova assinatura do método *apply*, não foi alterada nesta fase.

2.2.5 Model Element

modelElement.h - Classe que guarda a informação relativa a uma figura. Esta ficou também com o papel da classe *triangle* que foi descontinuada nesta fase. Foi removida a estrutura *Triangulo* de nome *figure* e substituída por um *GLuint vertexCount* que indica o número de vértices da figura, assim como um *GLuint buffers[1]*, onde vai ser guardada, na primeira e única posição, o ponteiro para a memória da *gpu* onde estão guardados os vértices da figura. Foi também criado um método *prepareModel* que vai tratar da leitura do ficheiro e colocação da informação na *gpu*

e nas variáveis de instância mencionadas acima e alterado o método *apply*, de modo a tratar do desenho com *VBOs*.

2.2.6 Container Element

containerElement.h - Classe abstrata extendida por todos os elementos que podem ter filhos, como é o caso da *scene*, *group* e *models*. Esta classe tem como atributo um vetor para ponteiros de *BaseElements* chamado *children*, onde serão guardados todos os *BaseElements* filhos. Implementa, ainda, um método de nome *addChild()* usado para guardar no vetor *children* o ponteiro para o filho recebido como argumento. Esta classe manteve-se inalterada nesta fase.

2.2.7 Scene Element

sceneElement.h - Classe "Root"que extende a classe *ContainerElement*, ou seja, é a classe que tem o vetor *children* principal, onde serão colocados todos os ponteiros dos seus filhos (*groups*). O seu único método, *apply(float t, bool orb)*, faz a invocação do *apply(float t, bool orb)* de todos os seus filhos. O Engine tem como variável global uma instância desta classe de modo a ser chamado o seu *apply(float t, bool orb)* quando for para renderizar o "mundo"na função *renderScene()*. Para além da adaptação aos novos argumentos do método *apply*, não sofreu mais nenhuma alteração nesta fase.

2.2.8 Group Element

groupElement.h - Classe que extende a classe *ContainerElement*. O seu método *apply* é similar ao do *SceneElement*, ou seja, também faz a invocação do método *apply* de todos os seus filhos, com uma pequena diferença: antes usa a função OpenGL *glPushMatrix()* e depois usa a função *glPopMatrix()*, de maneira a que qualquer transformação efetuada na matriz seja só aplicada nos seus filhos. Assim como o *SceneElement* só teve como novidade a adaptação aos novos argumentos do método *apply*.

2.2.9 Models Element

modelsElement.h - Classe que extende a classe *ContainerElement*. O seu método *apply*, assim como na classe *GroupElement*, é similar ao do *SceneElement*, ou seja, também invoca o *apply* de todos os seus filhos, com a ressalva dos filhos apenas poderem ser instâncias da classe *ModelElement*. As funções OpenGL *glBegin(GL_TRIANGLES)* e *glEnd()* foram removidas do *apply(float t, bool orb)*, pois o desenho das figuras é agora feito por *VBOs*. Foi também adaptado o método *apply* aos argumentos adicionados.

2.2.10 Patch

patch.h - Classe que contém os métodos que calculam os pontos e as normais de um *patch* de *Bézier*.
Um *patch* é um conjunto de 16 pontos que definem uma superfície de *Bézier*. Cada *patch* encontra-se dividido em 4 partes que definem arcos/curvas desta superfície. O arco será definido consoante o valor de *tessellation*, isto é, quanto maior este valor, mais pontos serão criados ao longo do arco.

No fim de se fazer este processo para os 4 arcos, é novamente aplicado este algoritmo sobre os 4 pontos resultantes, um de cada arco, que resulta num ponto pertencente à superfície.
Ao aplicar este procedimento a todos os pontos da superfície, obtemos uma rede de arcos que criam quadriláteros entre eles e cada quadrilátero é dividido em 2 triângulos. A forma como este algoritmo funciona será mais aprofundada no capítulo seguinte.

2.3 Ficheiros Auxiliares

2.3.1 Figures

figures.h - Módulo responsável pela criação das formas geométricas. Nesta fase foram adicionados métodos para a criação de um modelo geométrico descrito por um *patch* de *Bézier*, sendo este um ficheiro de *input* fornecido com um formato específico. No método de criação, é lido o ficheiro que contém os pontos de controlo do *patch* e de seguida são passados aos métodos da classe *Patch*. O resultado são os pontos do modelo geométrico e serão escritos de seguida no ficheiro de *output*.

2.3.2 Camera

camera.h - Módulo responsável pela interação com o utilizador, o posicionamento da câmera e o desenho dos eixos, entre outras funções. Nesta fase, a única alteração foi a possibilidade de usar várias teclas ao mesmo tempo, isto é, é possível mover-se para a frente e para os lados ao mesmo tempo. Considerámos que era uma funcionalidade importante, pois torna a movimentação no mundo muito mais fluída. Nesta fase, foram também adicionadas teclas para:

- mostrar/esconder órbitas/rotas;
- Parar, acelerar, abrandar e inverter o tempo do sistema;
- Aumentar a velocidade da movimentação da câmera.

Capítulo 3

Generator

3.1 *Bézier Patches*

3.1.1 Ficheiro Input

Antes de passarmos para o processo de leitura do ficheiro de *input*, vamos apresentar o formato do mesmo:

- A primeira linha contém o número de *patches*.
- As linhas seguintes, que ocupam o mesmo número de linhas que o valor do número de *patches* (lido na primeira linha) contêm, para cada *patch*, uma sequência de 16 índices para os pontos de controlo que lhe pertencem.
- A linha seguinte contém o número de pontos de controlo.
- Por fim, as linhas contêm os pontos correspondentes a cada índice.

Durante o processo de leitura do ficheiro de *input*, é criado um *vector* de *doubles* que armazena as coordenadas dos pontos de controlo por ordem, isto é, durante a leitura dos índices dos pontos de controlo de cada *patch*, sempre que é lido um índice, esse ponto é logo colocado nesse *vector*. Isto é feito através de saltos dentro do ficheiro de *input*.
Tal só é possível, pois, no ficheiro, a linha com o número de pontos é a linha (1+ número de *patches* +1), sendo que os 1s presentes na fórmula correspondem à linha que contém o número de *patches* e à linha que contém o número de pontos e, portanto, na linha seguinte tem o ponto de índice 0.
Desta forma, o ponto de índice i está na linha (3+ número de *patches* $+i$). Depois de termos o ponto referido, basta apenas adicionar as suas 3 componentes ao *vector*.

3.1.2 Processamento dos patches

Para melhor perceber o funcionamento do algoritmo, responsável por traduzir um *Bézier patch* para um modelo geométrico, temos primeiro de saber o que são *Bézier curves* e como é que estas são definidas.
Para criar uma curva precisámos apenas de 4 pontos. Estes pontos são designados por pontos de controlo e estão definidos no espaço $3D$, ou seja, são constituídos por 3 coordenadas: x, y e z. Portanto, a curva pode ser definida através de uma curva paramétrica que será criada através destes pontos em conjunto com a matriz de *Bézier* e um vetor de coeficientes.

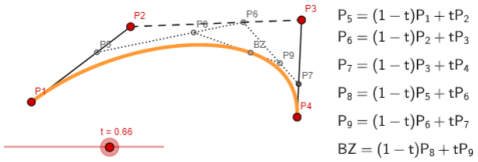


Figura 3.1: Curva gerada por 4 pontos de controlo

Assim sendo, esta curva pode ser tratada como uma curva paramétrica, isto é, basta definir uma equação e através de uma variável, neste caso a variável t, obtemos o ponto quando t tem um certo valor. Como estamos perante uma curva de *Bézier*, este varia entre [0,1].
Desta forma, o resultado da equação para qualquer t contido no intervalo [0,1] corresponde a uma determinada posição no espaço $3D$ pertencente à curva. Assim, se quisermos representar a curva no espaço, basta calcular o resultado da equação para muitos valores de t, espaçados por valores fixos. Por exemplo, se a *tessellation* é 4, então os valores de t serão $0, 0.25, 0.5, 0.75$ e 1. Tendo obtido esses pontos da curva, basta traçar segmentos entre esses mesmos pontos.
Facilmente se percebe que quantos mais pontos houver, ou seja, um valor maior de *tessellation*, mais próximos estes estarão (intervalos mais pequenos) e, consequentemente, a curva é representada de uma forma muito mais precisa.

No entanto, agora é necessário saber como é que podemos calcular tais pontos. Tal como foi dito anteriormente, a forma da curva corresponde ao resultado da combinação dos vários pontos de controlo, juntamente com outros agentes que, neste caso, são a matriz M e o vetor t.
Tomando o vetor P como o vetor que contém os pontos de controlo da curva, P_0, P_1, P_2 e P_3, M como a matriz das *curvas cúbicas* de *Bézier* e t como o vetor com os coeficientes de cada ponto, podemos calcular a curva criada pelos 4 pontos que é dada através da seguinte fórmula:

$$p(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Como se pode intuir, para o primeiro ponto da curva, $t = 0$, o resultado da equação coincide com o primeiro ponto de controlo, P_0. Analogamente, quando $t = 1$, o resultado da equação coincide com o último ponto de controlo, P_3, ou seja, o último ponto da curva.
Desta forma, quando queremos calcular a posição da curva para um determinado t, basta substituir o t no vetor e multiplicar as matrizes.

Depois desta pequena explicação sobre o funcionamento das curvas de *Bézier*, podemos ver como funcionam os *Bézier* patches. Na verdade, o princípio é similar ao que é utilizado nas curvas de *Bézier*, no entanto, em vez de termos apenas 4 pontos de controlo passámos a ter 16, que podem ser vistos como uma grelha de 4×4 pontos de controlo.

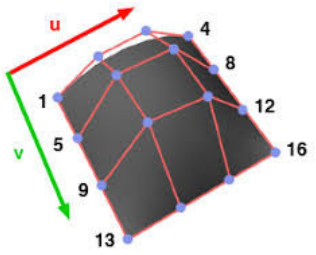


Figura 3.2: Superfície gerada por 16 pontos de controlo

No caso das curvas tínhamos apenas o paramêtro t para as movimentações ao longo da curva. Neste caso, vamos passar a ter dois parâmetros, o parâmetro u, para as movimentações horizontais pela grelha e o parâmetro v para as movimentações verticais. Tal como para a curva, ambos os parâmetros variam entre [0,1].
Então, falta calcular o ponto correspondente a (u, v) no *patch*. O método que usámos passa por considererar que o *patch* é uma grelha onde estão contidas 4 curvas de *Bézier*. Assim sendo, usámos o parâmetro u para calcular o ponto correspondente a cada uma destas curvas. Nesta fase, temos novamente 4 pontos e podemos considerar que estes 4 formam uma curva de *Bézier* e é nesta curva que será calculado agora o ponto em função do v.
Em suma, é criada uma curva de *Bézier* na direção de v, através dos 4 pontos em função do u nas 4 curvas de *Bézier* da grelha original. Na curva referida é calculado o ponto correspondente a (u, v) que é o equivalente a calcular o ponto na curva em função do v. Isto é feito através do mesmo método pelo qual foram calculados os pontos que deram origem a esta curva na vertical.

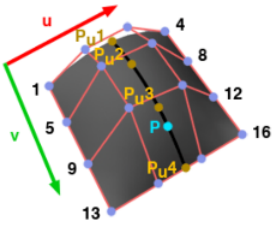


Figura 3.3: Ponto final na superfície de *patch* para um valor u e v

Portanto, para criar um modelo de um *patch* de *Bézier*, basta iterar para os vários valores de u e v, sendo o intervalo entre estes dado pela *tesselation*. Portanto, em cada iteração deste algoritmo são calculados os pontos para os pares (u_0, v_0), (u_0, v_1), (u_1, v_0) e (u_1, v_1), que formam um quadrilátero entre eles e que serão escritos no *array* de forma a formar os 2 triângulos correspondentes ao quadrilátero. Estes pontos do array, mais tarde, serão escritos por ordem para o ficheiro de *output*.
Esta escrita será feita através de triângulos, isto é, aquando da escrita no ficheiro serão retiradas as coordenadas dos vértices i, $i+1$ e $i+2$, que formam um triângulo entre si.

Capítulo 4

Engine

O Engine é responsável por receber ficheiros escritos em *XML*. Tendo como referência a fase anterior, as únicas alterações foram verificadas nas *tags translate* e *rotate*, o que foi simples de adaptar ao código já existente.
Usámos a função *glutGet(GLUT_ELAPSED_TIME)* para o cálculo do tempo passado para, assim, a velocidade dos planetas não estar dependente dos *frames* por segundo (*fps*) do programa.

4.1 VBOs

4.1.1 Model Element

Como mencionado anteriormente, uma das alterações que surgiu nesta fase foi a implementação dos *VBOs* para desenhar todos os modelos.
VBO é a sigla para *Vertex Buffer Object* e esta é uma funcionalidade oferecida pelo OpenGL que fornece métodos capazes de inserir a informação sobre os vértices diretamente na *gpu*.
O principal objetivo dos *VBOs* é oferecer uma performance substancialmente superior àquela conseguida com o método de renderização imediato, pois a informação reside na *gpu* em vez de na própria memória do sistema podendo, desta forma, ser diretamente renderizada pela *gpu*, diminuindo significativamente a sobrecarga do sistema. Logo, através deste método, os *fps* (*frames per second*) observados serão inevitavelmente superiores àqueles que seriam observados utilizando a renderização imediata da fase anterior.
Para implementar estes *VBOs* foi necessário criar aquilo a que se chamam *vertex buffers*, que são nada mais do que arrays, nos quais serão inseridos todos os vértices que constituem o modelo que queremos desenhar.
Assim, foi criado o método *prepareModel(const char* filename)* que vai abrir o ficheiro com os pontos dos triângulos do modelo e vai preencher um *vector<float> vbo* com as coordenadas referentes aos tais pontos dos triângulos das figuras.
Depois de percorrido todo o ficheiro, é calculado o número de vértices que foram inseridos no *vector<float> vbo* e esse valor é guardado na variável de instância *vertexCount*. Com o auxílio das funções OpenGL *glGenBuffers()*, *glBindBuffer()* e *glVertexPointer()* é guardado na *gpu* a informação relativa ao modelo, guardando, na variável de instância *buffers[0]*, o ponteiro para a *gpu* onde essa informação se encontra.
Para o desenho dos modelos através dos *VBOs* criados, ou seja, quando o método *apply* for invocado, ao invés de se usar a função *glVertex3f()* para o desenho dos pontos,

como era o caso da fase anterior, foram usadas as funções *glBindBuffer()*, *glVertexPointer()* e *glDrawArrays()*, com o auxílio do ponteiro *buffers[0]* e do número de vértices *vertexCount*.

4.2 Curva Catmull-Rom

4.2.1 Translate

Para a criação desta curva são passados no *XML* os pontos por onde esta tem de passar com a criação da *tag points*, que é filha da *tag translate*. Estes pontos são adicionados a um *vector<float> curveCoords*, da classe *TranslateElement* em causa, durante o parse dos filhos do *translate* com o método *addCoords(float x, float y, float z)* e, no final, é chamado o método *genCurveMatrix()* que vai alocar a matriz *curvePoints[nPoints][3]*, onde *nPoints* é o número de pontos da curva e o valor 3 corresponde às 3 coordenadas. Como referido anteriormente, a classe *TranslateElement* tem *float **curvePoints* como variável de instância, logo esse apontador vai ficar a apontar para a matriz criada. Essa matriz é preenchida com a informação retida pelo *vector<float> curveCoords*. Este método é também responsável pelo preenchimento da variável de instância *float pos[nCurvePoints][3]*, onde *nCurvePoints* é uma macro que guarda o número de pontos que vão constituir a órbita do planeta. Esta variável vai ser preenchida com o uso do método *getGlobalCatmullRomPoint()* que vai ser explicada em seguida.
Quando for chamado método *apply*, é verificado se existe a variável de instância *time* e, em caso positivo, é também verificado se o *bool orb* recebido como argumento toma o valor *true*. Se sim, significa que é para desenhar as órbitas dos planetas, com a evocação do método *renderCatmullRomCurve()* que vai percorrer a matriz *pos[nCurvePoints][3]* e executando a função OpenGL *glVertex3f(pos[i][0], pos[i][1], pos[i][2])* para cada um dos pontos. Depois do *if* do desenho da órbita, o tempo passado é dividido pelo tempo que demora a dar uma volta à órbita e é declarada uma variável *float xyz[3]* que vai ser preenchida com as coordenadas para executar a função OpenGL *glTranslatef(xyz[0], xyz[1], xyz[2])* com a evocação do método *getGlobalCatmullRomPoint(t, xyz, deriv)* juntamente com uma variável *float deriv[3]* que será preenchida, mas não será usada.
Caso o translate não tenha a variável de instância *time* inicializada, significa que se trata de um translate da fase anterior que leva à execução da função OpenGL *glTranslatef(x,y,z)*, onde x, y e z são variáveis de instância inicializadas no construtor.
Vamos agora proceder a uma breve explicação da execução do método *getGlobalCatmullRomPoint(float gt, float* pos, float* deriv)* e das suas funções auxiliares. Este método recebe um *float gt*, com o qual calcula o verdadeiro tempo através do número de pontos da curva; um *float* pos* e um *float* deriv*, que preenche com a posição e a derivada da curva, respetivamente. Este método calcula os 4 índices da curva que vão ser usados para o cálculo da posição e derivada, através do método auxiliar *getCatmullRomPoint()*.
Este método vai receber o tempo, os 4 pontos referentes aos índices calculados anteriormente e ainda os dois pontos: posição e derivada. Através da matriz de Catmull-Rom, de um vetor T, de um outro vetor ∂T (derivada de T) e das coordenadas dos 4 pontos recebidos como argumento, vai finalmente preencher as variáveis *pos* e *deriv*. Abaixo é demonstrado como é feito este cálculo:

$$p(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1.0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$p'(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1.0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

4.3 Rotate

Nesta classe foi apenas acrescentado o tratamento de um *rotate* no tempo. É verificado se o parâmetro *time* existe e, em caso positivo, é calculado o ângulo no tempo com $angle = 360/time \times t$, onde o *time* é o tempo que demora uma rotação completa e t é recebido como argumento da função *apply* e simboliza o tempo em que aquele frame se encontra. De seguida é executada a função OpenGL *glRotatef(angle, axisX, axisY, axisZ)*, assim como na fase anterior.

Capítulo 5

Resultados obtidos

5.1 Sistema Solar - Visualização

O resultado final do sistema solar dinâmico correspondeu ao esperado pelo grupo. Todos os planetas foram representados com o máximo de cuidado para respeitar diferenças de tamanho e distância, mas sem nunca perder de vista que se trata de uma animação e que, por isso, não consegue (nem deve) respeitar a 100% a escala real. Os seus movimentos de rotação e translação foram representados tentando também uma aproximação o mais possível à realidade. Houve ainda a preocupação de adicionar um cometa gerado através de um ficheiro de *input* de um *patch*.

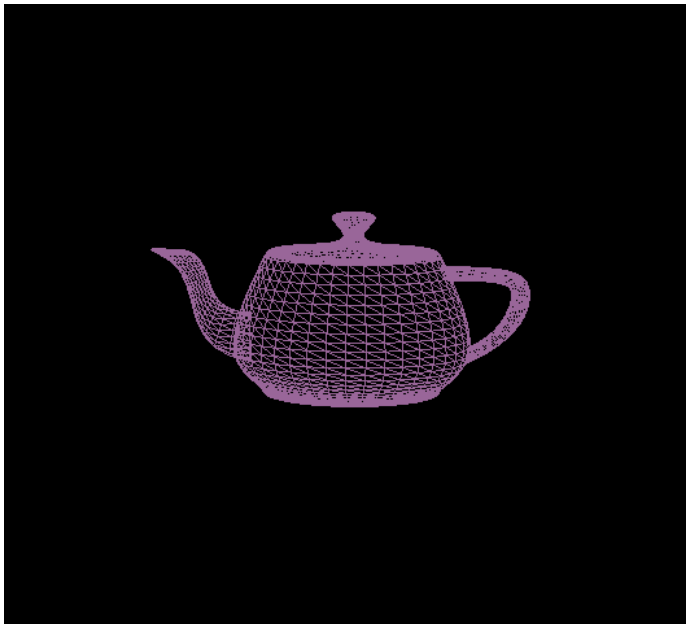


Figura 5.1: Teapot (cometa) gerado pelo *patch* de *Bézier*

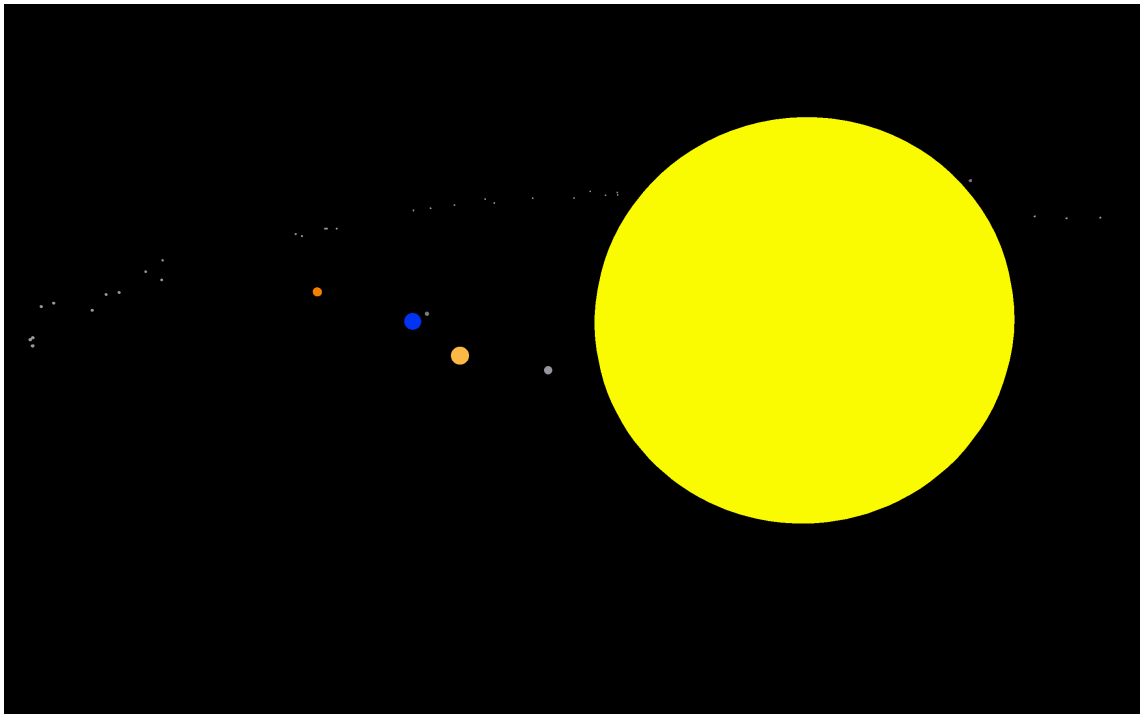


Figura 5.2: Representação dos quatro primeiros planetas do Sistema Solar com o satélite natural Lua

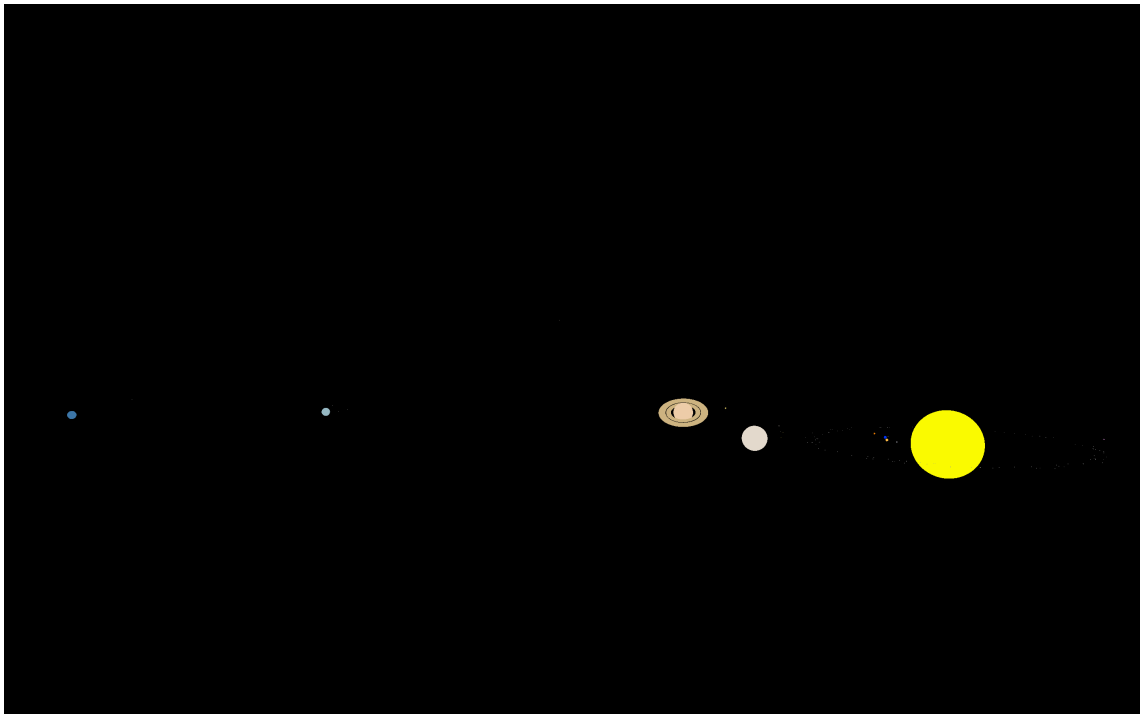


Figura 5.3: Representação do Sistema Solar

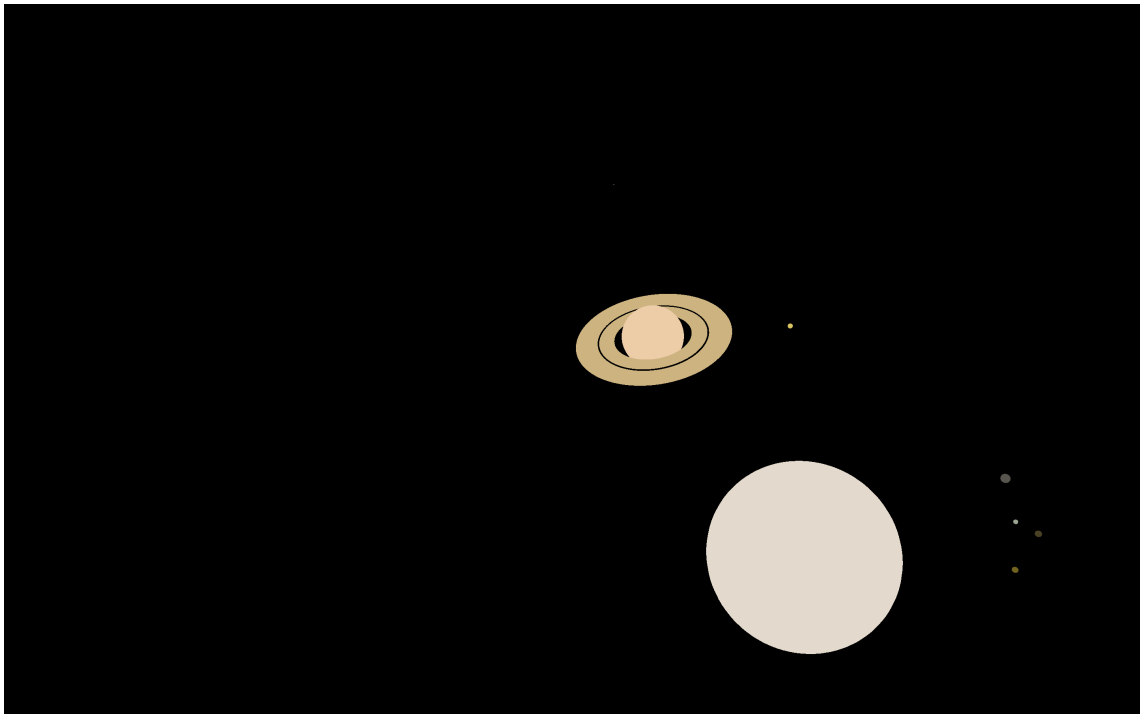


Figura 5.4: Representação de Júpiter e Saturno e respetivos satélites naturais

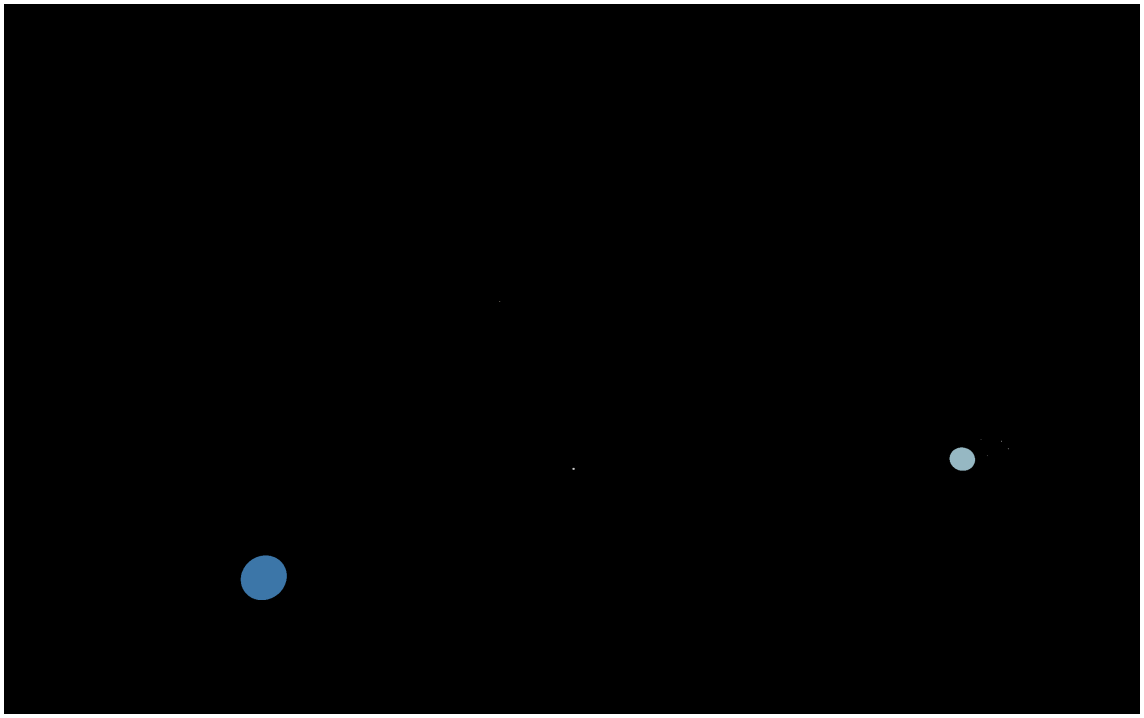


Figura 5.5: Representação de Urano e Neptuno e respetivos satélites naturais

Figura 5.6: Representação do Sistema Solar visto de Plutão

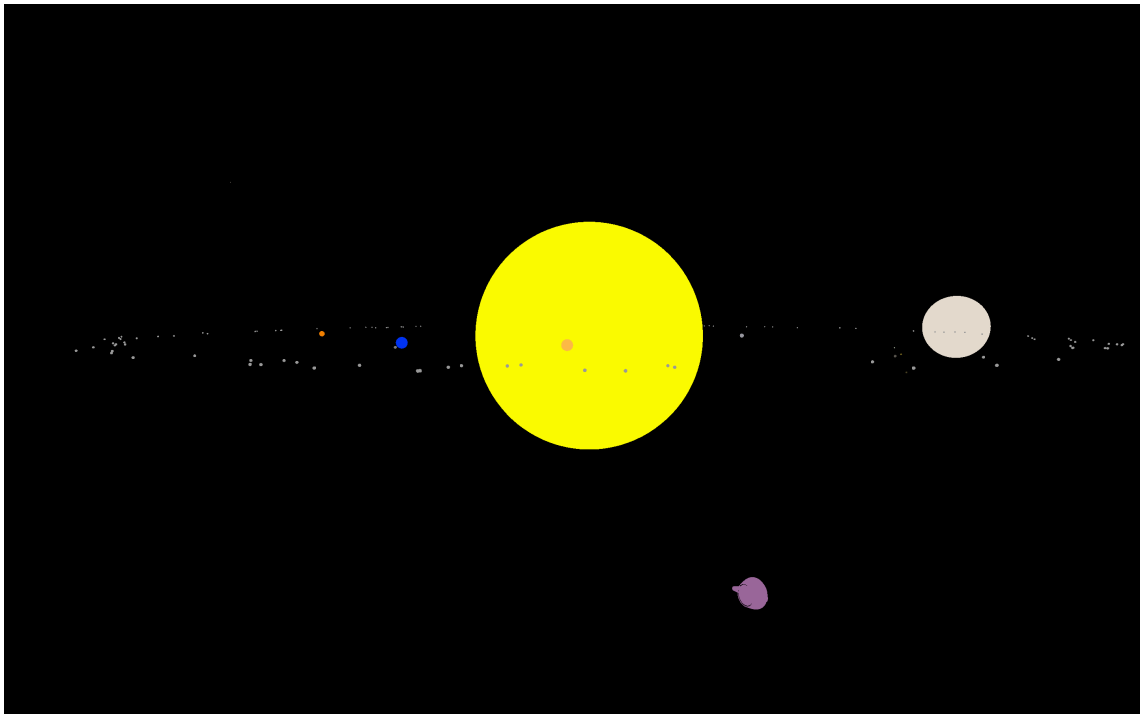


Figura 5.7: Representação do Sistema Solar visto do cometa

Capítulo 6

Conclusão

Ao contrário do que aconteceu na fase anterior, a elaboração desta terceira fase do projeto foi um pouco mais complexa, contudo, muito do código já tinha sido preparado para as alterações necessárias nesta fase.
Durante a elaboração desta fase deparámo-nos com algumas dificuldades. Uma delas esteve relacionada com os *Bézier patches* e tudo aquilo que eles envolvem, já que nunca tínhamos trabalhado com eles e, desta forma, não possuíamos conhecimento suficiente para, a partir destes, elaborar um algoritmo capaz de projetar os modelos gráficos correspondentes. No entanto, com o auxílio da documentação fornecida pelo docente e alguma pesquisa conseguimos superar este obstáculo.
A implementação, tanto das Catmull-Rom curves como dos *VBOs*, foi outro dos desafios que tivemos de resolver, no entanto, com a preparação prévia do código e os conhecimentos adquiridos durante a resolução dos guiões, facilmente estruturámos os passos necessários para a sua correta resolução.
Assim, considerámos que o resultado final desta fase corresponde às expectativas, na medida em que conseguimos desenvolver um modelo, agora dinâmico, do Sistema Solar, tal como era pedido no enunciado.
Desta forma, esperámos que para a próxima e última fase que se avizinha, consigámos concluir o projeto de forma a obter um resultado final ainda mais realista e visualmente agradável ao utilizador.