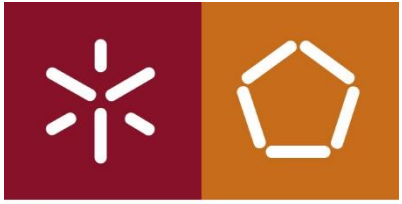

Universidade do Minho
Escola de Engenharia

Computação Gráfica

Relatório do projeto prático - Fase III

Curves, Cubic Surfaces and VBOs

Grupo de Trabalho

Ana Esmeralda Fernandes

A74321

Bárbara Nadine Oliveira

A75614

João Paulo Ribeiro Alves

A73542

Miguel Dias Miranda

A74726

Mestrado Integrado em Engenharia Informática

maio de 17

Conteúdo

Índice figuras

1. Introdução

Este relatório visa apresentar e explicar todas as decisões tomadas na realização da terceira fase do trabalho prático da Unidade Curricular de Computação Gráfica. Procuramos assim justificar todas as ponderações para a resolução dos problemas propostos no enunciado, assim como a abordagem tomada para gerar, a partir de um ficheiro de input em *xml* com elementos pré-definidos, uma maquete de um sistema solar de forma simples e fácil de interagir.

Nesta fase, é utilizado o conceito de Patches de Bezier, passando a ser possível gerar um determinado modelo a partir de um input com pontos e índices referentes aos *patches*. Foi ainda implementado o uso de estruturas de VBOs, para otimizar a performance e carga da GPU, e a possibilidade de descrever translações segundo curvas de Catmull-Rom, a partir de um conjunto de pontos que definem a curva e um parâmetro de tempo para o objeto percorrer a curva gerada.

Toda a modulação do problema foi feita com recurso à ferramenta de programação Visual Studio e à linguagem C++, abordada nas aulas práticas da UC.

Numa primeira fase serão descritas as alterações à aplicação *gerador,* onde se introduziu a funcionalidade de determinar os vértices de um objeto segundo patches de Bezier. Posteriormente será descrita a abordagem e as decisões relativas à aplicação *sistemaSolar,* que por sua vez passa a implementar as estruturas de VBOs e translações segundo curvas de Catmull-Rom.

2. Projetos Desenvolvidos

2.1. Aplicação Gerador

Para esta fase do projeto foi necessário remodelar a aplicação designada como Gerador, desenvolvida na primeira fase do trabalho. O programa era inicialmente responsável por gerar um ficheiro com um conjunto de vértices associados a um determinado modelo, sendo que estes modelos gerados podiam ser planos, cubos, cilindros, esferas ou cones, conforme os parâmetros de entrada introduzidos.

Com os novos requisitos desta terceira fase do projeto, é agora possível gerar um qualquer modelo, tendo este por base um ficheiro de input com informações que permitem construir *patches de Bezier,* e com estes, desenhar as superfícies de um determinado modelo.

Devido à formulação matricial envolvida na definição e no cálculo dos vértices que estão nas curvas de B*ezier*, além das necessárias alterações no módulo da classe Modelo, foram ainda criadas duas novas classes, designadas de Patch e Matriz, para estruturar e simplificar a forma como é gerado um qualquer modelo em três dimensões, alicerçado em *patches* de *Bezier*.

2.1.1. Descrição das alterações às classes existentes

2.1.1.1. Classe Modelo

Com o objetivo de manter a estruturação modular com que a aplicação *Gerador3d* se desenvolveu inicialmente, a implementação das funções necessárias para gerar os vértices de um modelo, a partir de um input com informações sobre *patches de Bezier,* foram associadas e criadas dentro do módulo da classe **Modelo.** Assim, passam a existir nesta classe, em acréscimo, as seguintes funções:

- Função de leitura de um ficheiro com as informações para construir os *patches de Bezier:*

```
vector<Patch> Modelo::lerFicheiro(char* file)
```

Considerando o formato apresentado para os ficheiros com os *patches de Bezier,* a função ***lerFicheiro*** começar por iterar o ficheiro, lendo e armazenando todas as informações contidas nele e, posteriormente, cria os P*atches* com os pontos de controlo associados a cada um.

Para isto existem na função três estruturas de dados principais:

- *vector* com todos os patches, representados ainda pelos índices dos pontos de controlo, tal como listados no ficheiro de input;
- vector com todos os vértices de controlo lidos;
- *vector de* Patches*,* sendo esta a variável retornada, depois de preencher cada *Patch* com os respetivos pontos de controlo existentes no *vector* de vértices.

Para o ficheiro utilizado, com os *patches* do teapot, na primeira linha é indicado que existem 32 *patches.* Deste modo, até à linha 33 surgem os 32 *patches* (compostos pelos índices) que serão armazenados na estrutura *vector de* Patches (representados por índices). Na linha 34 é indicado o número de pontos de controlo disponíveis para ser usados (neste caso 290) e, portanto, desde a linha 35 até ao final do ficheiro, existirá em cada linha a informação que permite criar um vértice, e que serão armazenados no *vector* de vértices.

Terminada a leitura do ficheiro, é necessário criar efetivamente os *Patches* com os pontos de controlo. Assim, para cada *Patch* lido e guardado do ficheiro, composto por 16 inteiros que representam o índice do vértice a utilizar, é iterado cada um destes inteiros e criado um Patch com os vértices de controlo correspondentes.

Por exemplo, se o patch a iterar for igual a [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], significa que este patch utiliza como vértices de controlo os vértices existentes nas primeiras 15 posições do *vector* de vértices. Se o patch for [3, 16, 17, 18, 7, 19, 20, 21, 11, 22, 23, 24, 15, 25, 26, 27], então o *patch* será composto pelos vértices existentes no índice 3,16,17, … ,26,27 do vector com todos os vértices.

- Função para criar um modelo a partir de Patches de *Bezier:*

```
void Modelo::bezier(char* file, int tesselation)
```

A função *Bezier* será responsável por criar os triângulos do modelo, a partir de um ficheiro de input com as informações para construir os *patches de Bezier* e com um pré-definido nível de tecelagem.

A função terá como estruturas de dados principais:

- vector de *Patches*, que será preenchido pelo valor de retorno da chamada da função *lerFicheiro*, anteriormente apresentada;
- um *vector de Triangulos* que representam o modelo geométrico a produzir.

Para cada *Patch* do *vector* de *Patches*, são criadas as matrizes Px, Py e Pz, que representam a conversão de um *patch* para uma matriz (neste caso uma matriz 4x4), construída apenas com as coordenadas do eixo do X, Y ou Z, respetivamente, dos pontos de controlo do *patch*.

Posteriormente, é feito um ciclo aninhado para implementar o nível de tecelagem indicado. Este nível de tecelagem deve ser obrigatoriamente superior a 1, porque o parâmetro t, utilizado na construção dos pontos através de um patch, é calculado com a divisão (1/ tecelagem) e deve estar contido entre 0 e 1. Este parâmetro foi designado por *passo* no código da função.

Para cada iteração deste ciclo aninhado, são criados 4 vértices, onde cada uma das suas coordenadas são calculadas segundo a fórmula de *Bezier* e com a matriz P correspondente, com a função *getPontoB* que será descrita na secção seguinte.

Por fim, de forma semelhante aos outros modelos possíveis de gerar, depois de ter o conjunto de triângulos do modelo já calculados, estes são escritos num ficheiro de output, tal como era o procedimento na primeira fase do projeto.

2.1.2. Descrição das novas Classes criadas

Para dar resposta à necessidade de guardar informações relativas a *patches de Bezier*, assim como ter um módulo responsável por realizar operações e métodos sobre matrizes, foram construídas as classes **Patch** e **Matriz**, que serão descritas brevemente nas seguintes secções.

Todas estas classes foram desenvolvidas segundo uma estrutura modelar, estando separadas entre o ficheiro com o código em extensão .cpp e o ficheiro com os cabeçalhos com a extensão .h.

2.1.2.1. Classe Patch

Classe criada simplesmente para guardar informações relativas a um *patch de Bezier.*

Tendo sido assumido, na descrição do formato dos *patches de Bezier***,** que estes contêm sempre 16 pontos de controlo, esta classe tem como variável um array de 16 Vertices, que representam os pontos de controlo do *patch.*

Sobre esta variável da classe foram ainda implementados os métodos de consulta (*gets*) de um Vertice numa posição específica do array, assim como os métodos de atribuição (*sets*) para colocar um Vertice numa determinada posição do array.

2.1.2.2. Classe Matriz

Classe criada para implementar a representação de uma matriz, assim como disponibilizar métodos e operações matemáticas sobre matrizes.

Para ser possível representar uma matriz, com número de linhas e colunas genérico, ou seja, não estático, optou-se por representar uma matriz num vetor de vetores, onde cada posição *[i][j]* contem elementos do tipo numérico *double.* Esta variável é assim designada por *elementos* e é do tipo *vector<vector<double>>.*

Foi ainda considerado relevante existirem mais duas variáveis da classe, para determinar em qualquer momento e de forma direta, o número de linhas e colunas da matriz.

A nível de métodos foram implementadas as comuns funções de consulta (*gets*) sobre as variáveis de classe já citadas, assim como definidas as funções do operador + (soma) e * (produto) para a classe Matriz, sendo assim possível adicionar e multiplicar matrizes usando diretamente o operador correspondente, aplicado a tipos desta classe.

De maior relevância para justificar o surgimento desta classe e relacionadas com a construção de um modelo partindo de *patches de Bezier,* existem ainda funções para:

- Determinar a transposta de uma Matriz:

```
Matriz transposta();
```

- Converter um *Patch* para uma matriz:

```
static Matriz patchToMatriz(Patch p, int cord);
```

Função que dado um *patch de Bezier*, o transforma numa matriz. Recebe como argumentos o *Patch* a transformar e, no segundo argumento, recebe um parâmetro que determina se deve usar a coordenada X, Y ou Z, dos vértices do *Patch,* para gerar a matriz.

Dado que um *Patch* tem sempre 16 vértices de controlo, o resultado será sempre, no contexto deste projeto, uma matriz 4x4;

Seja um patch P = [V_0, V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8, V_9, V_{10}, V_{11}, V_{12}, V_{13}, V_{14}, V_{15}] representado pelos seus 16 vértices de controlo, a matriz P_k correspondente será da forma:

$$Pk = \begin{bmatrix} V0k & V1k & V2k & V3k \\ V4k & V5k & V6k & V7k \\ V8k & V9k & V10k & V11k \\ V12k & V13k & V14k & V15k \end{bmatrix},$$

onde k representa qual a coordenada (x, y ou z) dos vértices do patch é que se usa para construir a matriz.

- Calcular as coordenadas de um vértice pertencente a um *Patch de Bezier:*

```
static double getPontoB(double u, double v, Matriz P);
```

Função que com os seus argumentos, e usando uma Matriz M predefinida, gera a coordenada (X, Y, ou Z, conforme a matriz P passada) de um vértice do *patch de Bezier*.

Os parâmetros u e v representam as coordenadas de um determinado ponto e são usados para calcular as matrizes U e V, respetivamente. A matriz P, recebida no terceiro argumento, é por sua vez o resultado da transformação prévia de um *Patch* para uma matriz, usando a função ***patchToMatriz*** já citada.

Seja a matriz U = $[u\text{^}3 \quad u\text{^}2 \quad u \quad 1]$, V = $[v\text{^}3 \quad v\text{^}2 \quad v \quad 1]^T$, que representam dois pontos, a as matrizes quadradas:

$$\text{M} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \text{ e } P = \begin{bmatrix} P0 & P1 & P2 & P3 \\ P4 & P5 & P6 & P7 \\ P8 & P9 & P10 & P11 \\ P12 & P10 & P14 & P15 \end{bmatrix},$$

é utilizada a fórmula $U * M * P * M^T * V$, apresentada no documento *"Formulário de Curvas e superfícies" para determinar a coordenada do vértice no patch de Bezier.*

2.1.3. Diagrama de Classes Gerador3d

A seguinte imagem procura descrever o diagrama de classes da aplicação gerador3d.

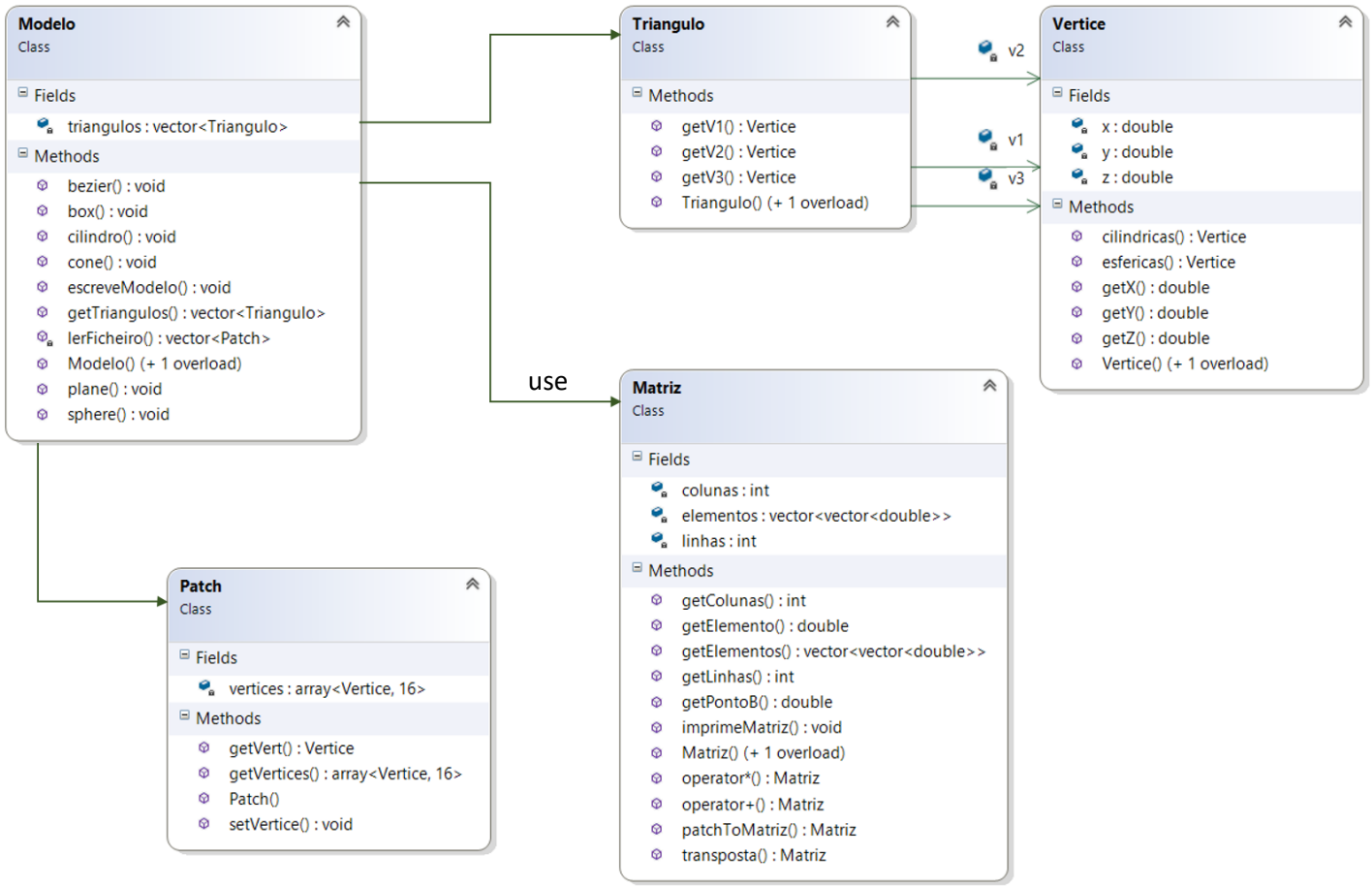

Figura 1 - Diagrama de Classes do Projeto gerador3d

2.2. Aplicação sistemaSolar

O projeto responsável por interpretar as informações contidas num ficheiro xml, foi designada como sistemaSolar. Apesar do seu nome ser bastante especifico, associado ao contexto das figuras que serão desenhadas, a aplicação será capaz de traçar qualquer tipo de cenário desde que o mesmo esteja representado de forma correta, segundo os parâmetros e hierarquias que compõe o ficheiro xml de input.

Para esta terceira fase, a aplicação sistemaSolar passa a utilizar estruturas de VBOs para o desenho dos objetos do sistema solar, ao contrário do uso direto de uma variável global, utilizada na fase anterior. Passa também a ser possível definir as translações dos objetos segundo curvas de Catmull-Rom. Estas curvas são determinadas perante um conjunto de pontos iniciais e a translação do objeto segundo a curva é calculado segundo o parâmetro de tempo, em segundos, no qual o objeto deve percorrer toda a curva.

Por fim, é ainda possível definir as rotações dos objetos com um parâmetro de tempo, sem que seja necessário utilizar o valor de um ângulo.

2.2.1. VBOs

2.2.1.1. Formulações iniciais

Para iniciar esta secção vamos enunciar e explicitar alguns dos conceitos que vão ser mencionados recorrentemente nos sucessivos tópicos:

- Vertex Buffer Objects (VBO):

  VBOs podem ser vistos como uma estrutura abstrata de dados que permite melhorar consideravelmente a performance geral do OpenGl, evitando as desvantagens associadas ao tempo de transferência de dados entre a memória central e a GPU.

  A implementação destas estruturas, permitem que um array de vértices seja guardado diretamente na memória da placa gráfica, promovendo um acesso aos dados por parte da GPU bastante eficientes. A utilização destes arrays de vértices possibilitam ainda a redução de chamadas de funções e o uso redundante e recorrente de dados, associados com vértices, guardados em estruturas partilhadas na memória central.

  A ideia chave da utilização de VBO's é a seguinte:

  - Sem a utilização de VBOs, utilizando o modo de desenho imediato do OpenGL, os vértices são definidos em memória RAM e copiados um a um, para a placa gráfica, conforme a ordem com que devem ser processados. Este processo é repetido sempre que os queremos desenhar.
  - Com a utilização VBOs, todos os vértices são definidos e copiados para um buffer, que é passado numa fase inicial, e uma única vez, para a memória da placa gráfica, antes de qualquer pedido para os processar e desenhar.

    Deste modo, o processamento dos triângulos a desenhar torna-se muito mais eficiente, porque na comunicação entre o CPU e o GPU surge uma situação de *bottleneck,* obrigando as operações de desenho da GPU a parar, enquanto esperam por mais dados para processar enviados pelo CPU.

Com o uso de VBOs, tentamos ter o máximo de dados e processamento possível no lado da placa gráfica.

Um VBO não é um objeto no sentido da programação orientada a objetos, é simplesmente um array de dados. O termo "objeto" é usado uma vez que o OpenGL nos fornece um identificador referente à totalidade do buffer, ao invés do primeiro elemento do array como geralmente sucede.

2.2.1.2. Implementação de VBOS

Para a implementação das estruturas *VBOs* no projeto *sistemaSolar,* foi seguido o procedimento utilizado na realização do guião 4 das aulas práticas, que abordava este tema.

Deste modo, foram feitas as seguintes alterações:

- Função ***main***:

  Tendo concluído a leitura dos dados do ficheiro de input, é possível determinar o número de modelos lidos e o número de curvas de Catmull-Rom a desenhar.

  Assumindo que existem tantos modelos quantos ficheiros referenciados nos atributos *model* (do elemento *models* do ficheiro *xml de input),* são inicializados tantos *VBOs* quantos modelos lidos, existindo assim um *VBO* por modelo.

  Do mesmo modo, considerando que é desenhada uma curva de Catmull-Rom sempre que numa translação, o tempo indicado for superior a 0 e forem dados pelo menos quatro vértices, então é também possível saber quantas curvas de Catmull-Rom são necessárias de desenhar, e com isso, criar um VBO para cada uma delas. Esta inicialização dos VBOs, com o seu respetivo tamanho, é feita com recurso à função ***glGenBuffers***.

  Feita a inicialização dos buffers é usada a função preencheVBOsModelos, para criar e preencher cada um dos VBOs com os dados associados a um determinado modelo, e a função **preencheVBOsCatmullRom**, para criar e preencher cada VBO com as coordenadas de cada vértice de uma determinada curva de Catmull-Rom.

  Antes da chamada do método ***glutMainLoop***, é feita a inicialização final dos VBOS com a função ***glEnableClienteState****(GL_VERTEX_ARRAY).*

- Função ***preencheVBOsModelos****:*

```
void preencheVBOsModelos(Grupo* grupos)
```

  Método chamado pela função *main* para preencher cada um dos *VBOs* com as coordenadas dos vértices, relativas a um determinado modelo.

  Recebe como único argumento o apontador para a raiz da estrutura Grupo. Esta estrutura, utilizada na segunda fase do projeto, tem como variáveis o conjunto de informações e transformações associados a um elemento *group*, lido do *xml.* Além destas variáveis, um *Grupo* tem ainda um apontador para o eventual *Grupo* irmão e para o eventual *Grupo* filho.

Para o nodo recebido no argumento, a função começa por determinar qual o conjunto de modelos que o mesmo contem. De seguida, é feito um ciclo para iterar cada um destes modelos, e preencher um dos índices do array de VBOs com as coordenadas dos vértices do respetivo modelo. O seguinte excerto de código procura mostrar parte deste procedimento:

```
(...)
vector<Modelo> modelos = grupos->getVecModels();
   for (auto modelo : modelos) {
          vector<Vertice> vertsModelo = modelo.getVertices();
          int size = 3 * vertsModelo.size();
          double* vecCoordenadas = new double[size];

          for (auto vertice : vertsModelo) {
                vecCoordenadas[iter++] = vertice.getX();
                vecCoordenadas[iter++] = vertice.getY();
                vecCoordenadas[iter++] = vertice.getZ();
          }
(...)
```

Recordando a implementação realizada na primeira fase do projeto, uma variável do tipo *Modelo* é caracterizada pelo conjunto de *Vertices* que o compõe, e um *Vertice* é identificado pelas três coordenadas do mesmo, no espaço a três dimensões.

Por esta razão, o tamanho do array que irá conter as coordenadas dos vértices do modelo, terá como tamanho fixo *3*nrVérticesModelo*.

Alocando espaço para este array, designado *vecCoordenadas,* é realizado um ciclo para iterar os vértices do modelo, e são colocadas as respetivas coordenadas de cada vértice no array de coordenadas. Estas coordenadas são inseridas segundo a ordem X, Y e Z.

Tendo preenchido o array com as coordenadas, é ativada a posição do array de VBOs sobre o qual se vão inserir os dados, com recurso à função ***glBindBuffer***. O referido índice é preenchido com os elementos do array de coordenadas determinado, usando para isso a função ***glbufferData***.

Como no preenchimento dos *VBOs* deve também ser tida em consideração a ordem pela qual os modelos são processados, caso o nó recebido como argumento tenha filhos, a função é chamada recursivamente para o respetivo Grupo filho. Só depois desta verificação, é que caso existam irmãos no nodo, a função é iterada recursivamente para o Grupo irmão.

Com este procedimento, será possível posteriormente, na função renderscene, garantir a hierarquia que inicialmente existia no ficheiro *xml* de *input*, reutilizando as transformações realizadas num nó pai, para os seus eventuais nós filhos.

- Função *preencheVBOsCatmullRom:*

```
void preencheVBOsCatmullRom(Grupo* grupos)
```

Método chamado pela função *main* para preencher cada um dos *VBOs* com as coordenadas dos vértices, relativas a uma curva Catmull-rom.

Depois da utilização de VBOs para os objetos do sistema solar, com a posterior implementação das translações segundo curvas de Catmull-Rom, o desenho destas curvas, que de certa forma indicavam a "orbita" do objeto, causou uma considerável perda de performance por parte da aplicação, obtendo em média cerca de 40FPS, face aos Esta situação levou à tomada de decisão de adaptar e desenvolver, também em VBOs, as curvas de Catmull-Rom.

Estes VBOs irão assim conter os pontos usados para descrever os segmentos de reta de cada uma das curvas de Catmull-Rom a serem desenhadas.

Para o nodo recebido no argumento, a função começa por determinar se nesse Grupo existe uma translação a ser implementada segundo uma curva de Catmull-Rom. Para isto, é feito um teste para saber se o tempo da translação é superior a 0 e se existem pelo menos quatro pontos para descrever a curva.

De seguida, a função executa o código associado ao método *renderCatmullRomCurve* utilizada no guião da aula prática, onde para uma variável **t**, entre 0 e 1, a faz variar em intervalos de 0.01 e para os sucessivos valores de **t** calcula o respetivo vértice na curva da Catmull-Rom. Todos os vértices determinados, que irão definir a curva, são guardados numa variável auxiliar. Pelo facto de **t** variar entre 0 e 1, e com um incremento de 0.01, é possível saber que neste caso serão sempre usados 101 vértices para definir cada uma das curvas de Catmull-Rom. A determinação destes pontos é feita utilizando a função *getGlobalCatmullRomPoint* que será apresentada e explicada na secção onde abordaremos a implementação das curvas de Catmull-Rom.

Feito este procedimento, a partir deste ponto a função é em tudo semelhante à função *preencheVBOsModelos* apresentada no tópico anterior. De forma sucinta, é ativado o buffer, no respetivo índice disponível, no array de VBOs de Catmull-Roms. Depois de ativo, são passados para lá todas as coordenadas dos vértices determinados da curva, novamente segundo a ordem X, Y e Z.

Novamente, pelos motivos já abordados, deve também ser tida em consideração a ordem pela qual os vértices das várias curvas são processados. Assim, caso o nó recebido como argumento tenha filhos, a função é chamada recursivamente para o respetivo Grupo filho. Só depois desta verificação, é que caso existam irmãos no nodo, a função é iterada recursivamente para o Grupo irmão.

Com este procedimento, será possível posteriormente, na função renderscene, garantir a hierarquia que inicialmente existia no ficheiro *xml* de *input*, reutilizando as transformações realizadas num nó pai, para os seus eventuais nós filhos.

2.2.1.3. Comparação performance da aplicação com e sem VBOs

Para analisar as diferenças de performance entre o projeto da Fase II, sem VBOs, e o atual projeto da Fase III, com uso de VBOs, foi tido em consideração o parâmetro *frames per second.* Para isso foi utilizado um método contador de FPS, chamado na função *renderscene* depois do desenho de uma cena. Esta função foi acrescentada tanto no projeto da Fase II e da Fase III.

Para uma comparação mais uniforme, utilizou-se o projeto da Fase II tal como foi entregue (apenas com acréscimo da função contadora de FPSs) e o projeto da Fase III sem a implementação das translações segundo curvas de Catmull-Rom, diferindo assim do anterior projeto apenas na implementação de VBOs. Tomamos esta decisão porque a posterior implementação de curvas de Catmull-Rom trás uma nova carga ao GPU que influenciaria esta comparação.

De forma genérica, com o desenho dos objetos através de uma variável global, onde a GPU fica à espera que o CPU lhe passe gradualmente a informação para que a mesma processe triângulos, é possível obter uma média de cerca de 350FPS. Com a utilização de VBOs, passando para o lado da GPU, num único comando, toda a informação para que a mesma faça o processamento dos triângulos, o ganho de FPS foi na ordem dos 200%, obtendo uma média de cerca de 720FPSs ao renderizar e interagir com a mesma cena.

As imagens apresentadas em anexo, na secção *Comparação performance VBOs,* procuram descrever estes valores obtidos. No canto superior esquerdo, são apresentados os FPSs determinados através da função contadora de FPS utilizada.

2.2.2. Curvas de Catmull-Rom

2.2.2.1. Formulações iniciais

Para a criação de curvas de Catmull-Rom, é necessário um conjunto de pontos previamente definidos e lidos a partir do input em xml. A curva criada será assim o resultado das várias splines formadas através da fórmula de Catmull-Rom, que segue uma interpolação cúbica, permitindo assim descrever uma curva suave que passa por todos os pontos inicialmente referidos.

De modo a conseguirmos obter a curva são então necessários pelo menos quatro pontos iniciais, e um parâmetro chave '**t**' que irá definir a interpolação entre os 2 pontos centrais.

Para determinar os vértices que descrevem a forma da curva é utilizada a fórmula **T * M * P**, apresentada no documento *"Formulário de Curvas e superfícies", onde:*

$$\text{T= } [t^3 \;\; t^2 \;\; t \;\; 1], \text{ M = } \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \text{ e } P = \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix}.$$

A multiplicação destas matrizes dá origem à matriz geométrica que contem a coordenadas do vértice na curva, segundo o parâmetro t.

Esta fórmula confere à curva de Catmull-Rom as seguintes características:

- A curva passa por todos os pontos;
- A curva é contínua na primeira derivada, garantindo que não ocorrem descontinuidades na direção ou magnitude da tangente;
- A curva não é continua na segunda derivada, a segunda derivada é interpolada linearmente em cada segmento fazendo a curvatura variar linearmente no comprimento do segmento;

2.2.2.2. Implementação das curvas de Catmull-Rom

Para a determinação dos vértices segundo a fórmula apresentada, foi seguido o procedimento utilizado na realização do guião 9 das aulas práticas, que abordava este tema. Pelo facto das curvas de Catmull-Rom estarem associadas a translações, foi na classe translação que se desenvolveram as funções que determinam os vértices dos segmentos das curvas.

Deste modo, foram feitas as seguintes alterações:

- Função ***preencheVBOsCatmullRom:***

```
void preencheVBOsCatmullRom(Grupo* grupos)
```

Este método já foi abordado na secção anterior, relativa à implementação de VBOs. É novamente referido aqui, de forma breve, por estar relacionado com as curvas de Catmull-Rom.

De forma genérica, esta função realiza o trabalho da **renderCatmullRomCurve** utilizada nas aulas, determinando o conjunto de vértices que serão usados para definir os segmentos de reta da curva, usando o parâmetro **t** entre 0 e 1, incrementado sucessivamente em 0.01. Para isto, utiliza a função **getGlobalCatmullRomPoint**.

Por serem utilizados VBOs, estes vértices são guardados no respetivo VBO, que será posteriormente utilizado pela renderscene, em vez de utilizar uma chamada recorrente à **renderCatmullRomCurve** em cada frame, passando assim diretamente para o lado da

GPU os vértices que a mesma precisa de processar para desenhar uma determinada curva.

- Função *getGlobalCatmullRomPoint*:

```
void Translacao::getGlobalCatmullRomPoint(float gt, float *res, float
                                          *deriv)
```

Função que determina, para o parâmetro gt recebido no primeiro argumento, qual é o segmento de reta a processar e o respetivo vértice interpolado nesse segmento.

Determinado qual o segmento, calcula o conjunto de índices a usar e com estes índices vai buscar o conjunto de vértices necessários para utilizar na interpolação.

Por se tratar de uma interpolação cúbica (N = 3) são necessários N+1 pontos, por cada segmento. Estes vértices são retirados da variável que contem todos os vértices a serem usados para definir a curva, lidos a partir do xml de input.

Para determinar a curvatura são assim usados 4 pontos que vão posteriormente formar as matrizes Px, Py e Pz.

Com os quatro vértices determinados e o parâmetro **gt** utiliza a função auxiliar **getCatmullRomPoint** para efetivamente calcular as coordenadas do vértice.

- Função *getGlobalCatmullRomPoint*:

```
void Translacao::getCatmullRomPoint( … )
```

Função que realiza o cálculo matricial **T * M * P** anteriormente apresentado, para determinar as coordenadas do vértice e ainda a derivada tangente a esse vértice, usando a interpolação de Catmull-Rom.

Para o cálculo do vértice, são criadas três matrizes P, nomeadamente Px, Py e Pz, que são formadas a partir das coordenadas X,Y e Z, respetivamente, dos quatro pontos recebidos como argumentos e pré determinados na função **getGlobalCatmullRomPoint**.

É inicialmente realizado o produto das matrizes M * P, criando assim uma matriz intermédia A. A matriz T, calculada conforme o parâmetro t recebido no primeiro argumento, é por fim multiplicada pela referida matriz A, de forma a calcular as coordenadas do vértice a interpolar.

Neste método é relevante referir a utilização da classe Matriz, criada inicialmente para simplificar a implementação dos *patches de Bezier*. Por ter nela implementada os métodos de criação e multiplicação de matrizes, foi novamente reutilizada para este contexto de Catmull-Rom.

2.2.3. Implementação de rotações segundo parâmetro tempo

Nesta fase do projeto, foi implementada a transformação de rotação com um parâmetro de tempo, contrariamente às duas fases anteriores, em que se recebia diretamente um ângulo a utilizar na rotação. Os restates parâmetro de uma rotação, relativos às restantes coordenadas X, Y e Z, continuam a indicar o eixo sobre o qual se irá fazer a rotação.

Sendo assim, no processo de leitura do ficheiro XML, uma vez encontrado o elemento xml rotação, será guardada numa variável o valor que se encontrar no atributo time, assim como as coordenadas já referidas.

Na altura de executar efetivamente a rotação, na função *desenhaGrupo* chamada pela *renderscene,* o parâmetro de tempo lido irá ser multiplicado por 360˚, que representa uma volta completa. A partir daqui o ângulo a usar na rotação será calculado dividindo o valor GLUT_ELAPSED_TIME pelo último valor de tempo referido.

Utiliza-se por fim, de forma semelhante às fases anteriores, a função glRotatef(angle, x, y, z) para realizar a rotação do objeto a desenhar.

2.2.4. Descrição do processo de leitura do XML

Para realizar a análise do input em formato *xml* foi reutilizada a biblioteca *TinyXML* já familiarizada nas fases anteriores do projeto.

Para conseguir lidar com a estrutura própria deste ficheiro e a hierarquia dos grupos que possam existir nele, foram tomadas as mesmas considerações apresentadas no relatório da fase anterior. Deste modo, a leitura do ficheiro xml é realizada apenas uma vez, logo na fase inicial de compilação do projeto, ficando assim na classe principal sistemaSolar a variável, do tipo Grupo*, com toda a informação sobre os grupos a desenhar.

Depois de estruturar esta variável, de forma praticamente igual ao processo utilizado na Fase II, esta será um apontador para o grupo raiz a desenhar, sendo todos os restantes grupos ou filhos ou irmãos deste nó. Ao passo que na fase anterior esta variável era utilizada diretamente na renderscene, para desenhar cada *frame*, assim que a mesma seja preenchida com a leitura do xml, ela será utilizada para gerar os VBOs necessários conforme as informações sobre modelos que a mesma contenha. Serão posteriormente usados os VBOs, na renderscene, para desenhar os modelos.

Como alterações significativas a nível do processo de leitura destacam-se as ligeiras alterações no parsing dos elementos XML, associados aos campos *translate* e *rotate:*

- Na primeira situação, relativa ao parsing de translações, passa a ser analisado o atributo *time* do elemento xml *translate.*

  Se o valor de *time* for superior a 0, então estaremos num grupo onde a translação dos seus modelos deve ser feita segundo uma curva de Catmull-Rom. Deste modo, o elemento *translate* irá ter como filhos um conjunto de pelo menos quatro pontos, que serão interpretados e guardados numa variável. Estes pontos irão posteriormente definir a curva de Catmull-Rom, dado que o método de interpolação cúbica usado obriga a curva a passar por estes pontos.

  O seguinte excerto procura mostrar um possível exemplo de input para a representação de uma translação segundo uma curva de Catmull-Rom, onde o objeto a deve percorrer totalmente demorando 30 segundos.

```
<translate time=30 >
        <point  X=1   Y=0.25  Z=0 />
        <point  X=0   Y=0.25  Z=-1 />
        <point  X=-1  Y=0.25  Z=0 />
        <point  X=0   Y=0.25   Z=1 />
</translate>
```

  Por outro lado, se o valor de *time* for igual a 0, é porque estamos perante uma translação estática, semelhante às utilizadas e realizadas na fase anterior do projeto. Neste contexto, o elemento xml *translate* terá apenas um único filho, que será o elemento point, que representa as coordenadas a serem usadas na translação.

O seguinte excerto procura mostrar um possível exemplo de input para a representação de uma translação estática, cujos parâmetros lidos serão diretamente utilizados como argumentos da função **glTranslatef**, deslocando o objeto para a abcissa X = 10 e ordenada Y = 2 e cota Z = 0.

```
<translate time=0 >
        <point X=10   Y=2   Z=0 />
</translate>
```

- Na segunda situação, relativa ao parsing de rotações, deixa de ser analisado o atributo designado na fase anterior como angle, representativo do angulo a utilizar na rotação, para se passar a ler o atributo time, que represente o intervalo de tempo que o objeto deve demorar a rodar 360˚ segundo o eixo especificado pelos valor dos atributos X, Y e Z.

  O seguinte excerto procura mostrar um possível exemplo de input para a representação de uma rotação segundo o eixo dos Y.

```
<rotate time=1 axisX=0 axisY=1 axisZ=0 />
```

Na seção em Anexos, designada "*Código do ficheiro XMl de entrada"*, será apresentado na integra o corpo do ficheiro xml utilizado como input.

2.2.5. Descrição do ciclo de rendering

2.2.5.1. Descrição da função renderscene

Tendo em conta os conceitos mencionados será descrito do processo utilizado para a execução do ciclo de rendering.

Durante a execução inicial do programa, após a leitura e tratamento do ficheiro xml de input terem sido realizadas, ficará guardada a informação nele contida em memória, quer numa variável global que será um apontador para uma estrutura do tipo Grupo, como em estruturas de VBOs associadas aos vértices dos modelos e das curvas de Catmull-Rom a desenhar.

Será então na função renderscene, com a invocação da função ***desenhaGrupo***, que se realiza a iteração destas variáveis globais e se obtém a informação necessária, de modo a se possível desenhar o cenário pretendido.

A implementação da referida função ***desenhaGrupo*** irá percorrer a estrutura de dados de forma idêntica ao modo com que esta foi preenchida, para ser possível obter as transformações a ser efetuadas à matriz de visualização atual do modelo (modelview Matrix) de cada um dos grupos "filhos". Este processo de iteração tem em consideração as transformações já realizadas pelo respetivo Grupo "pai" e permite assim ser possível reverter a modelViewMatrix para o ponto anterior, depois de todos os "filhos" de um dado grupo "pai" terem sido desenhados.

2.2.5.2. Descrição da função desenhaGrupo

Esta função ***desenhaGrupo*** recebe como argumento o apontador para um elemento da classe grupo, que na primeira iteração representa o nodo inicial da estrutura de dados e iterar.

O seu funcionamento pode ser descrito pelos seguintes passos:

- Antes de realizar qualquer alteração à matriz de visualização do modelo é efetuado um *glPushMatrix()* de modo a guardar o estado da matriz atual para que possam ser recuperadas posteriormente;
- Num segundo passo, é realizada a extração da informação necessária no nodo que recebe como argumento, e realiza as transformações correspondentes;
  Estas transformações são relativas a uma rotação, uma translação, uma cor e uma escala que vão ser aplicadas à matriz atual, utilizando as funções glut correspondentes;
- Para o caso da Translação, se na variável que representa esta alteração o parâmetro time for superior a 0 e existirem mais de quatro pontos, então significa que o modelo a processar tem uma translação definida segundo uma curva de Catmull-Rom.
  Nesta situação, a implementação da translação é feita ativando o VBO que contem os vértices da curva (órbita) a percorrer e á determinação da posição do objeto, segundo a função ***getGlobalCatmullRomPoint*** já abordada. O resultado da chamada desta função irá determinar as coordenadas sobre as quais se deve realizar a translação;
- De forma semelhante, no caso da rotação do objeto, se na variável que representa esta transformação o parâmetro time for também ele superior a 0, então é porque foi atribuído um tempo para o objeto rodar 360º sobre o eixo especificado. Assim, é determinado o angulo e usada a função ***glRotatef*** de forma normal;
- Realizadas as transformações, utilizando a informação contida no nodo recebido como argumento, será feito um ciclo para cada um dos modelos da variável que contem todos os modelos a desenhar daquele grupo. No corpo deste ciclo, com recurso à primitiva ***glDrawArrays***, será feito o pedido de desenho dos vértices do modelo a partir da ativação e referencia ao VBO que contem os dados do modelo a iterar.
- Após ter sido realizado o desenho dos triângulos, a função tenta identificar se o grupo atual tem algum nodo/Grupo filho. Para que não se percam as transformações já realizadas, caso se verifique que o nodo atual tenha descendência, a função desenhaGrupo é chamada recursivamente para o filho, sem que o *glPopMatrix* seja efetuada.
- Caso o grupo atual não tenha nenhum grupo filho, é efetuado o *glPopMatrix,* uma vez que vamos querer voltar ao estado da matriz de visualização do modelo anterior, sem as alterações que foram efetuadas pela atual iteração da função desenhaGrupo.
  Este processo é efetuado porque as transformações dos grupos seguintes (irmãos) já não serão realizas sobrepostas às transformações atuais.
- Posteriormente, e de forma análoga ao tópico anterior, a função tenta identificar se o grupo atual tem algum irmão, e será invocada recursivamente para este, caso exista.

A natureza recursiva da função e o modo como se encontra definida garante que as transformações à matriz de visualização do modelo para os filhos de um dado grupo são sempre obtidas a partir de transformações já efetuadas para o seu respetivo grupo pai, pelo razão de não existir nenhum *glPopMatrix* entre as transformações realizadas para o pai, e a invocação recursiva da função *desenhaGroup* para o filho, invocação esta que irá efetuar novas transformações a partir das anteriores.

Por motivos idênticos, todos os irmãos de um dado grupo serão desenhados a partir das mesmas transformações, ou seja, a partir da matriz de visualização do modelo que foi obtida a partir do grupo pai de todos eles, isto porque, quando a função é invocada para um grupo a partir do seu irmão, é feito um *glPopMatrix* que faz a matriz reverter para o estado anterior.

Este processo anula as transformações executadas para o grupo atual e assim a matriz de visualização do modelo irá ser a mesma que foi obtida a partir das transformações executadas pelo pai do grupo atual.

Na função renderScene invocamos a função desenhaGrupo passando-lhe a "raíz" da variável global que corresponde ao primeiro grupo a ser desenhado e a partir daí, a função irá processar de forma semelhante ao processo descrito acima.

O seguinte código mockup descreve de forma sucinta este processo:

```
void DesenhaGrupo(grupo){
      glPushMatrix();
   ➢ obtém transformações guardadas no grupo;
   ➢ obtém informações (cor e vértices a serem desenhados)
     guardadas nos modelos do grupo;

   ➢ efetuar transformações;
   ➢ se (catmull-Rom)
        glBindBuffer(GL_ARRAY_BUFFER, VBOCatmullRon[i1]);
        glVertexPointer(3, GL_DOUBLE, 0, 0);
        glDrawArrays(GL_LINE_LOOP, 0, 101 );
        i1++;

   ➢ desenhar vértices dos modelos (utilizando VBO's):
          glBindBuffer(GL_ARRAY_BUFFER, VBOModelos[i2]);
          glVertexPointer(3, GL_DOUBLE, 0, 0);
          glDrawArrays(GL_TRIANGLES, 0, nrVerticesModelo);
          i2++;
   ➢ se (Grupo atual tiver filho) =>
          desenhaGroup (filho do Grupo atual);
     glPopMatrix();

   ➢ se (Grupo atual tiver irmão) =>
          desenhaGroup (irmão do Grupo atual);

   ➢ termina a função;
}

renderScene(){
      (…)
      /* grupo* raiz -> raiz da variável global, que contem
a informação obtida pelo parsing do ficheiro de input xml */
      DesenhaGrupo(raiz);
      (…)
}
```

2.2.6. Diagrama de Classes sistemaSolar

A seguinte imagem procura descrever o diagrama de classes da aplicação sistemaSolar, representando tanto as classes e os seus respetivos métodos.

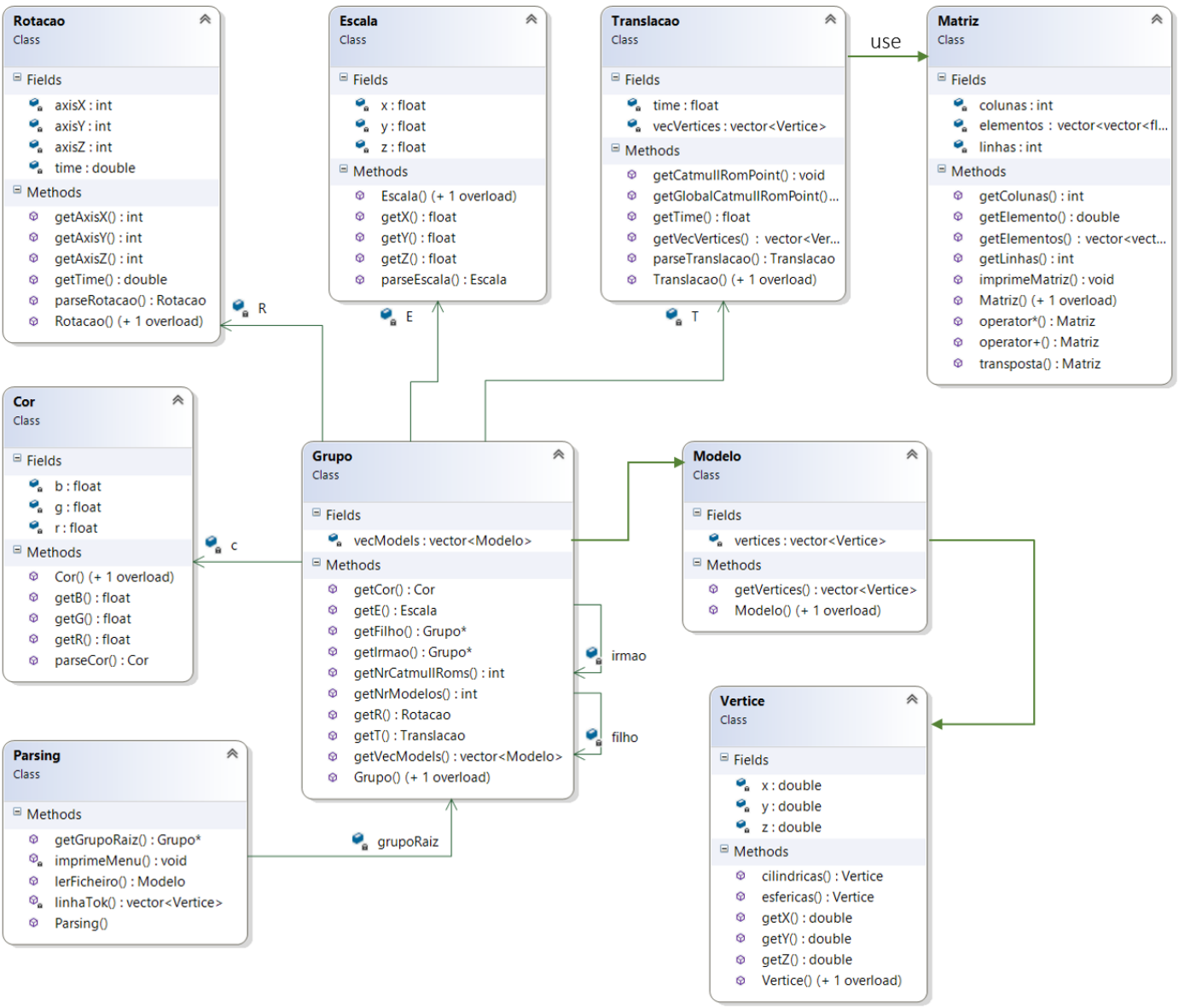

Figura 2 - Diagrama de Classes do Projeto sistemaSolar

3. Instruções de funcionamento do projeto

3.1. Aplicação sistemaSolar

A execução da aplicação sistemaSolar depende apenas da passagem de um único parâmetro à função main, que será o nome do ficheiro xml utilizado para gerar a cena.

Face ao formato do ficheiro xml, destaca-se que deve existir um campo ***diretoria***, com o atributo ***dir***, que irá conter a diretoria para onde se encontram os ficheiros (referentes aos vértices de um determinado modelo a desenhar) que serão usados. Estes ficheiros, identificados pelo seu nome, devem estar contidos no elemento ***models*** e o seu nome especificado no atributo ***file.*** A seguinte imagem exemplifica a forma do ficheiro xml de input utilizado e disponibilizado para o teste da aplicação sistemaSolar.

Na possibilidade de encontrar vários nomes de ficheiros no elemento *models*, estes serão todos considerados e desenhados tendo por base a mesma "origem". Esta origem será influenciada pelas transformações que possam ter sido feitas pelo grupo pai (se existir grupo pai e este tiver transformações como rotações, escalas ou translações) e consoante as transformações existentes no grupo que contem o referido conjunto de modelos.

A nível da aplicação sistemaSolar foi também tido em conta o controlo de erros, quando o nome dos ficheiros não existir ou estes não forem encontrados na diretoria especificada.

Como aspeto negativo, existe o facto do acesso ao ficheiro *.xml*, como este é passado ao main apenas pelo seu nome, é necessário alterar no código o valor de uma variável global que indica a diretoria do ficheiro *.xml* a utilizar. Esta diretoria é identificada numa variável global do tipo string, a aplicação sistemaSolar concatena o nome do ficheiro passado ao main com esta string e chega assim até ao caminho para o ficheiro *.xml* de input.

Como aspetos positivos, destacamos as diversas formas de interação e visualização dos modelos renderizados, através de diversas teclas. Estas funcionalidades serão descritas na secção seguinte e apresentam funcionalidades como zoom, várias formas de apresentação do sistema (estático ou dinâmico, com os planetas a rodar em torno do sol) e escolha do modo de desenho (linha, pontos ou a cheio).

3.2. Interação com o cenário desenhado

Para possibilitar uma melhor visualização do sistema solar desenhado foram criadas as seguintes opções de interação com a maqueta, baseadas nos conhecimentos apreendidos nas aulas práticas. Estas opções incluem funcionalidades como deslocar o ponto de visão sobre os eixos, rodar o ponto de visão da camara e realizar zoom.

Sugestões de interação com o sistema solar gerado:

1. Alteração do ponto visão da camara

Tecla Up - Deslocar para cima
Tecla Down - Deslocar para Baixo
Tecla Right - Deslocar para esquerda
Tecla Left - Deslocar para direita

2. Opções de visualização/desenho do modelo

Tecla 1 - Só linhas Tecla 2 - Preenchido Tecla 3 - Só vértices

3. Movimentação do modelo

Tecla W - Mover no sentido positivo do eixo Z
Tecla S - Mover no sentido negativo do eixo Z
Tecla A - Mover no sentido positivo do eixo X
Tecla D - Mover no sentido negativo do eixo X

4. Animações e Zoom

Tecla + - Aumentar o zoom da camara
Tecla - - Diminuir o zoom da camara
Tecla O - Restaurar todas as definições de visualização iniciais

Devido à escala dos planetas face ao sol, e pelo facto da distância de visão inicial ser grande, possibilitando a vista de todos os astros renderizados, recomenda-se como forma de interação o uso das teclas + e – para aumentar ou reduzir o zoom e das teclas A ou D para deslocar sobre o eixo dos X, sendo assim possível "percorrer" todos os objetos representados.

4. Conclusão

Como sumário final avaliamos de forma positiva o desenvolvimento desta fase do projeto, resolvendo todos os requisitos solicitados e mantendo os aspetos positivos realizados em fases anteriores do projeto.

As principais dificuldades encontradas durante a realização do trabalho estarão associadas com o desenvolvimento e implementação das curvas de Catmull-Rom, devido a algumas dificuldades na definição matricial do problema e uso correto dos pontos fornecidos para descrever a curva.

No nosso entender, todas as eventuais dificuldades foram totalmente ultrapassadas e compreendidas em grupo, tendo conseguido utilizar todos os conceitos aprendidos, cumprindo assim todos os requisitos básicos do enunciado do projeto.

Para demonstrar as funcionalidades, apresentamos uma maquete do sistema solar completa, enriquecida com elementos como luas, cores dos planetas e as rotações dos mesmos segundo uma órbita, criada com uma curva de Catmull-Rom, em torno do sol.

No final desta fase, referimos ainda a boa reutilização das estruturas e classes já utilizadas na fase I e II. Num sentido de valorização, focamos a utilização da classe Matriz para abstrair e simplificar todas as operações sobre matrizes, necessárias para realizar esta fase, assim como o uso de VBOs para guardar os vértices que descrevem os segmentos das curvas de Catmull-Rom, obtendo assim ganhos consideráveis no desempenho da aplicação.

5. Referências

Lighthouse3d. (2017). GLUT Tutorial. [online] Available at:
http://www.lighthouse3d.com/tutorials/glut-tutorial/
[Accessed 27 Feb. 2017].

Opengl-tutorial.org. (2017). *Tutorial 3 : Matrices*. [online] Available at:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
[Accessed 29 Mar. 2017].

Lighthouse3d.(2017).Catmull-Rom Spline. [online] Available at:
http://www.lighthouse3d.com/tutorials/maths/catmull-rom-spline/
[Accessed 26 Apr. 2017]

6. Anexos

6.1. Comparação performance VBOs

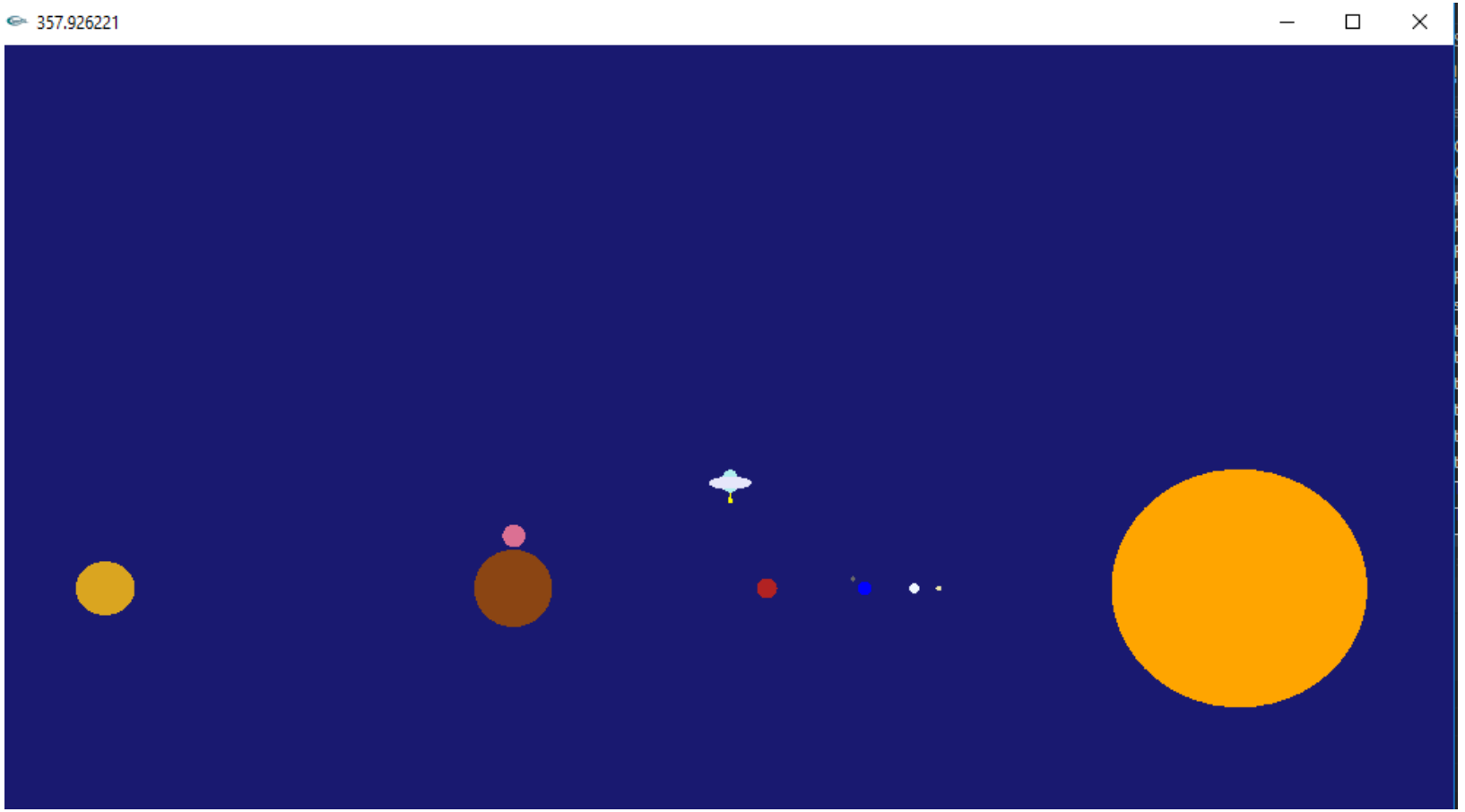

Figura 3 - Sistema solar Fase II sem recurso a VBOs – I

Exemplo do resultado obtido após correr a aplicação desenvolvida para a Fase II.
Sem utilização de VBOs para os modelos desenhados no cenário desenvolvido, a média de FPS ronda os 350fps.

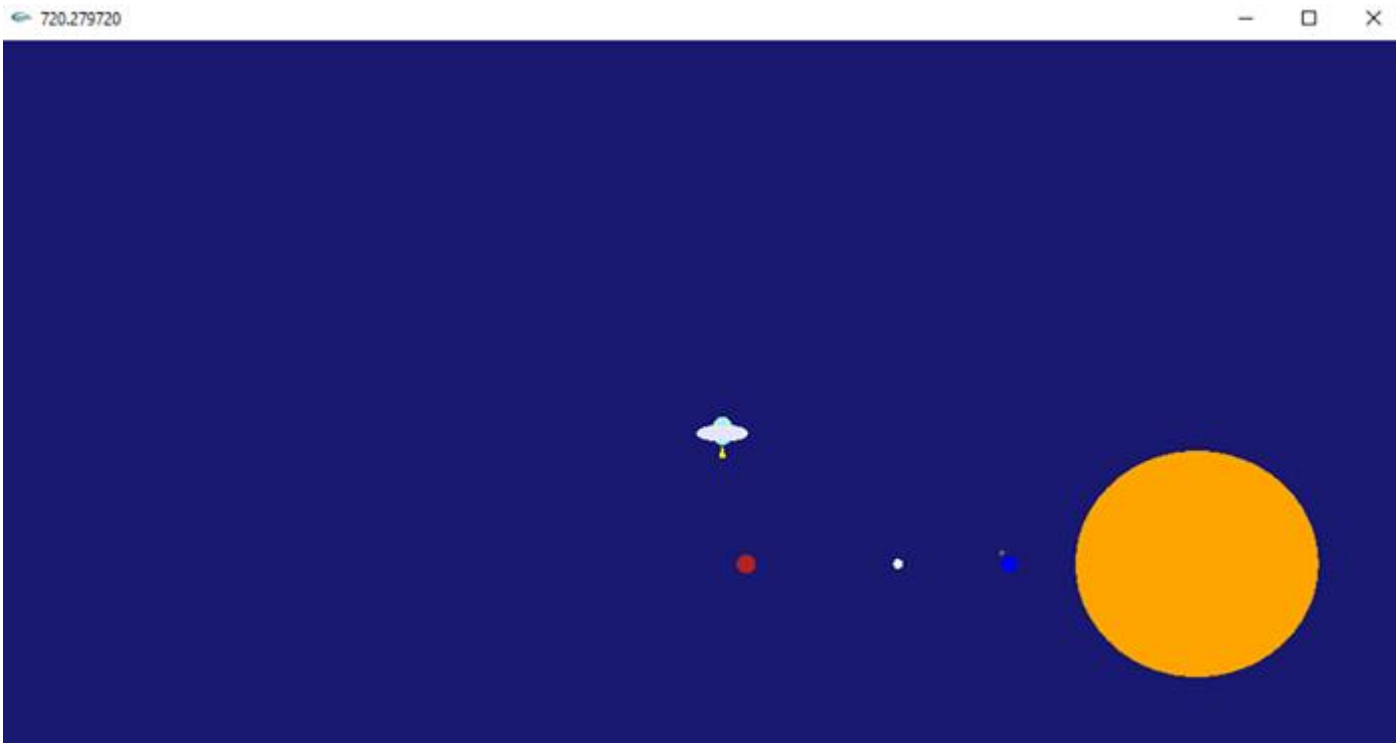

Figura 4 - Sistema solar Fase II com recurso a VBOs – I

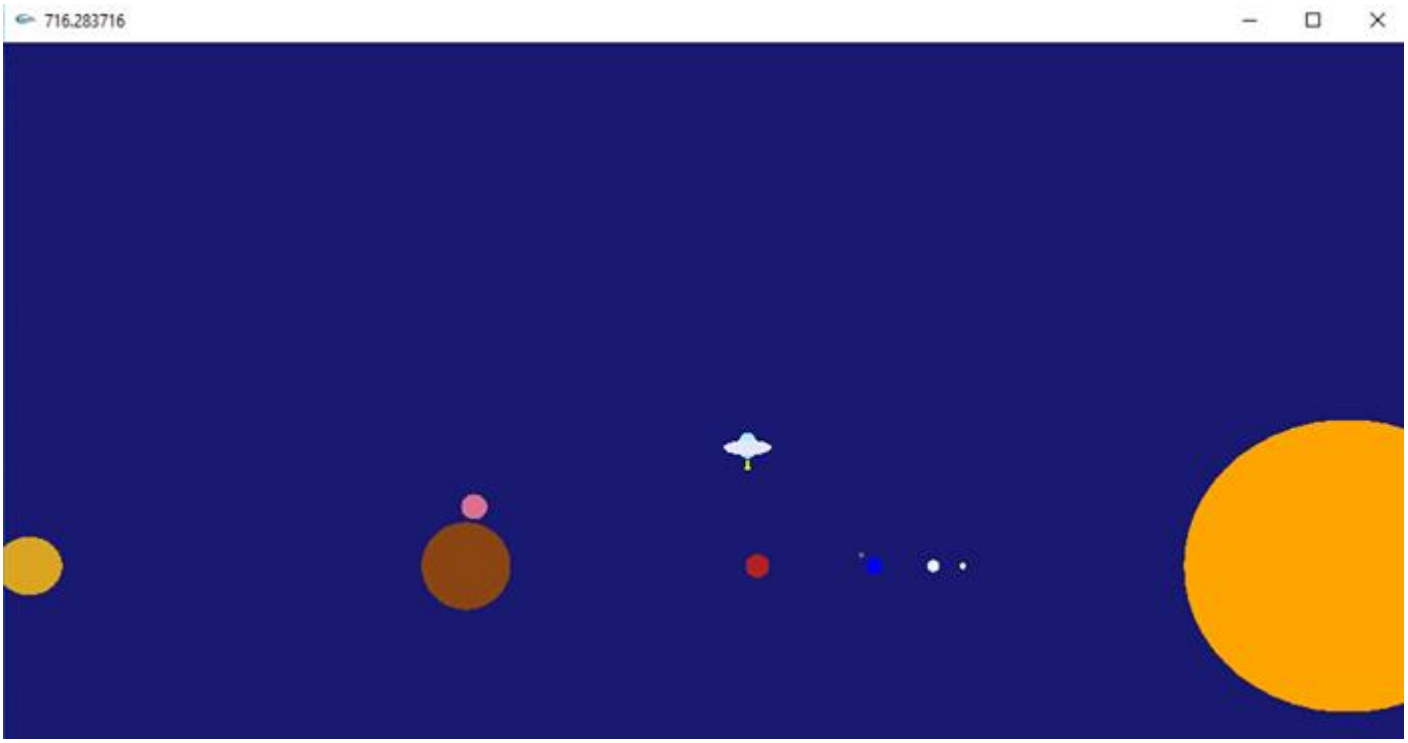

Figura 5 - Sistema solar Fase II com recurso a VBOs – II

Exemplo do resultado obtido após alterar a aplicação desenvolvida para a Fase II para utilizar VBOs.
Com recurso a VBOs n cenário desenvolvido, a média de FPS ronda os 720fps, mais do dobro dos obtidos na Fase II entregue.

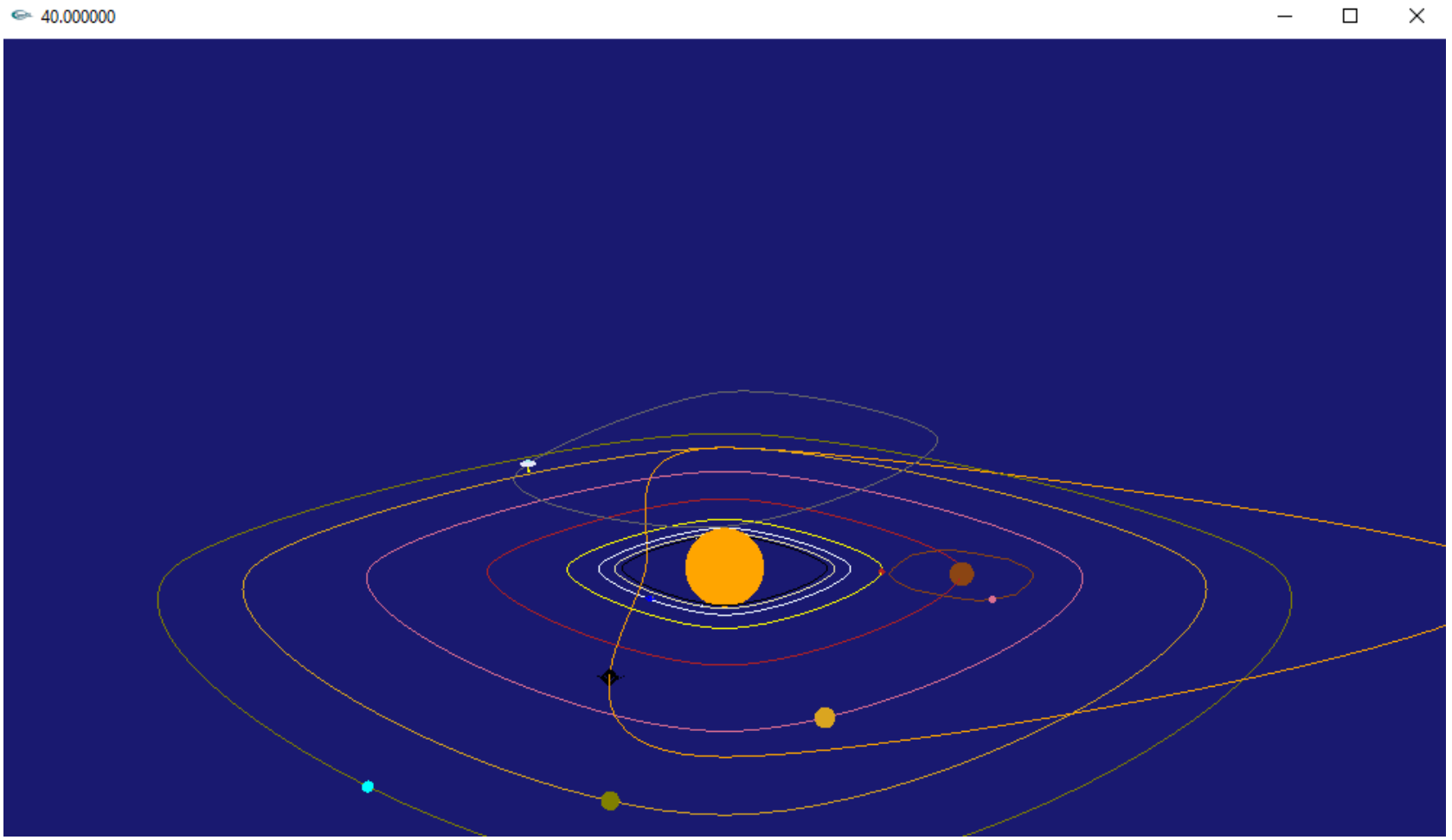

Figura 6 - Sistema solar final da Fase III – I

Exemplo obtido após correr a aplicação final, desenvolvida para a Fase III, com a utilização de VBOs para os modelos e usando uma função para determinar os vértices dos segmentos de reta das curvas de Catmull-Rom, em real time.

Com a uso de VBOs apenas para os modelos, a média de FPS ronda os 40fps, face aos perto de 650 obtidos em etapas anteriores do projeto.

Esta quebra de performance justificou assim a passagem dos vértices que definem os segmentos de reta das curvas de Catmull-Rom para VBOs.

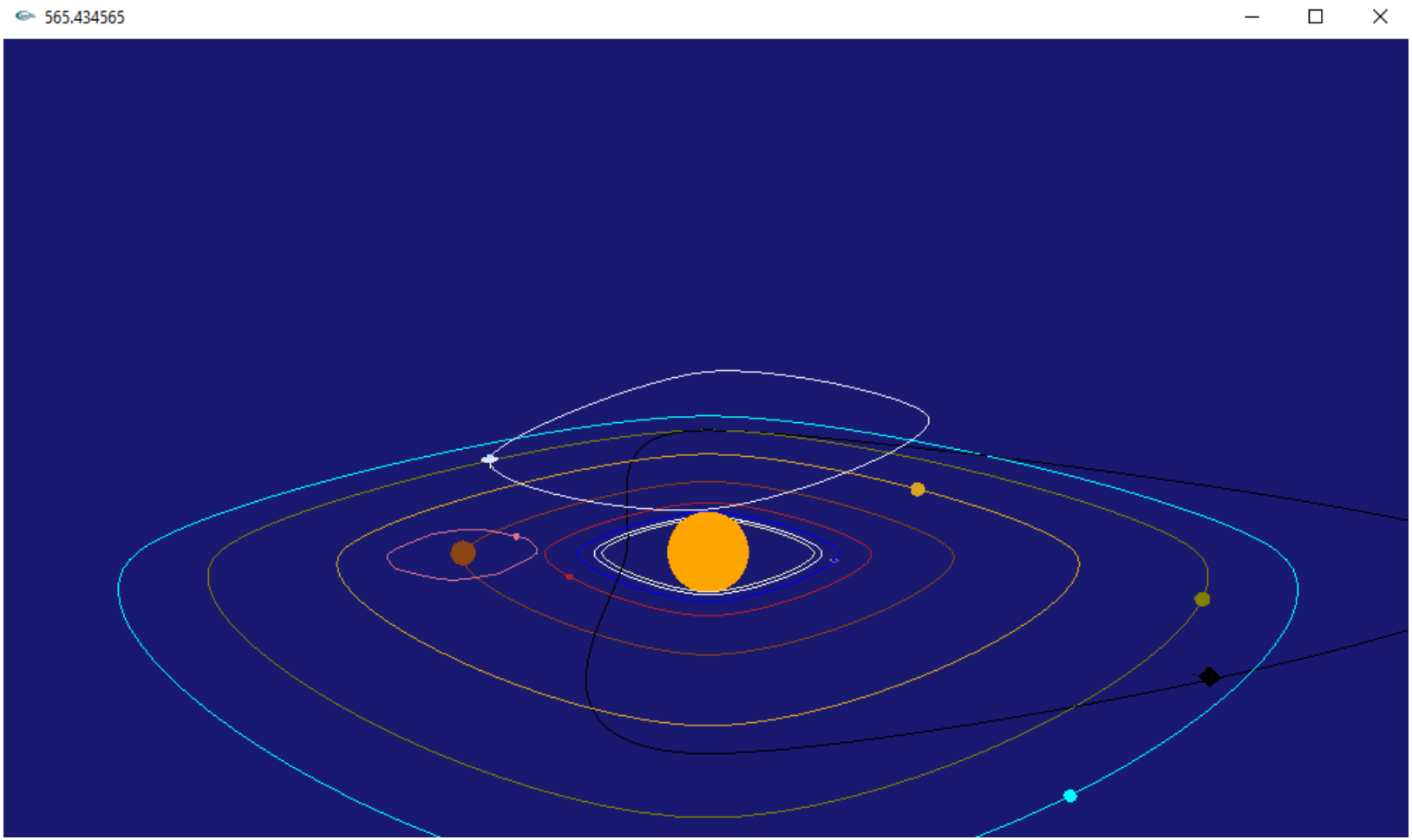

Figura 7 - Sistema solar final da Fase III – II

Exemplo do resultado obtido após correr a aplicação desenvolvida para a Fase III, com a utilização de VBOs para os modelos e para os segmentos de reta das curvas de Catmull-Rom determinadas.

Com a utilização destas duas estruturas de VBOs para o cenário desenhado, a média de FPS ronda os 565fps, face aos rudimentares 40fps obtidos sem o uso de VBOs para os vértices dos segmentos das curvas de Catmull-Rom.

6.2. Código do ficheiro XMl de entrada

```
<?xml version="1.0" encoding="UTF-8"?>
<scene>
	<diretoria dir="C:\Users\migue\OneDrive\Trabalho CG\Fase
III\sistemaSolar\Ficheiros\" />
	<group>
			<!-- SOL -->
			<cor R=1.0 G=0.647 B=0 />
			<scale X=23 Y=23 Z=23 />
			<models>
				<model file="esfera.3d" />
			</models>
			<!-- Cometa teapot -->
			<group>
				<translate time=17 >
					<point X=1 Y=0 Z=0 />
					<point X=0 Y=0 Z=-5 />
					<point X=-10 Y=0 Z=0 />
					<point X=0 Y=0 Z=6 />
				</translate>
				<rotate time=90 axisX=1 axisY=0 axisZ=0 />
				<scale X=0.05 Y=0.05 Z=0.05 />
				<models>
					<model file="bezier.3d" />
				</models>
			</group>
			<!-- Mercurio -->
			<group>
				<translate time=4 >
					<point X=1.3 Y=0 Z=0 />
					<point X=0 Y=0 Z=-1.3 />
					<point X=-1.3 Y=0 Z=0 />
					<point X=0 Y=0 Z=1.3 />
				</translate>
				<cor R=0.933 G=0.910 B=0.667 />
				<scale X=0.02 Y=0.02 Z=0.02 />
				<models>
					<model file="esfera.3d" />
				</models>
			</group>
			<!-- Venus -->
			<group>
				<translate time=6 >
					<point X=1.4 Y=0 Z=0 />
					<point X=0 Y=0 Z=-1.4 />
					<point X=-1.4 Y=0 Z=0 />
					<point X=0 Y=0 Z=1.4 />
				</translate>
				<cor R=0.941 G=0.973 B=1 />
				<scale X=0.04 Y=0.04 Z=0.04 />
				<models>
					<model file="esfera.3d" />
				</models>
			</group>
			<!-- Terra -->
```

```
<group>
	<translate time=9 >
		<point X=1.6 Y=0 Z=0 />
		<point X=0 Y=0 Z=-1.6 />
		<point X=-1.6 Y=0 Z=0 />
		<point X=0 Y=0 Z=1.6 />
	</translate>
	<cor R=0.0 G=0.0 B=1 />
	<scale X=0.055 Y=0.055 Z=0.055 />
	<models>
		<model file="esfera.3d" />
	</models>
	<!-- Lua -->
	<group>
		<translate time=30 >
			<point X=1 Y=0.25 Z=0 />
			<point X=0 Y=0.25 Z=-1 />
			<point X=-1 Y=0.25 Z=0 />
			<point X=0 Y=0.25 Z=1 />
		</translate>
		<cor R=0.412 G=0.412 B=0.412 />
		<scale X=0.35 Y=0.35 Z=0.35 />
		<models>
			<model file="esfera.3d" />
		</models>
	</group>
</group>
<!-- OVNI -->
<group>
	<translate time=20 >
		<point X=2.6 Y=1.2 Z=0 />
		<point X=0 Y=1.4 Z=-2.6 />
		<point X=-1 Y=1 Z=0 />
		<point X=-2.6 Y=1.7 Z=0 />
		<point X=0 Y=1.7 Z=2.6 />
		<point X=1 Y=1 Z=0 />
	</translate>
	<cor R=0.902 G=0.902 B=0.980 />
	<scale X=0.2 Y=0.06 Z=0.2 />
	<models>
		<model file="esfera.3d" />
	</models>
	<group>
		<translate time=10 >
			<point X=0 Y=0.3 Z=0 />
			<point X=0 Y=0.3 Z=0 />
			<point X=0 Y=0.3 Z=0 />
			<point X=0 Y=0.3 Z=0 />
		</translate>
		<cor R=0.686 G=0.933 B=0.933 />
		<scale X=0.4 Y=1.5 Z=0.4 />
		<models>
			<model file="esfera.3d" />
		</models>
		<group>
			<translate time=10 >
			<point X=0 Y=-0.7 Z=0 />
			<point X=0 Y=-0.7 Z=0 />
			<point X=0 Y=-0.7 Z=0 />
```

```
                        <point X=0 Y=-0.7 Z=0 />
                </translate>
                        <cor R=1.0 G=1 B=0 />
                        <scale X=0.4 Y=0.3 Z=0.4 />
                        <models>
                                <model file="cone.3d" />
                        </models>
                </group>
        </group>
</group>
<!-- Marte -->
<group>
                <translate time=13 >
                        <point X=2 Y=0 Z=0 />
                        <point X=0 Y=0 Z=-2 />
                        <point X=-2 Y=0 Z=0 />
                        <point X=0 Y=0 Z=2 />
                </translate>
                <cor R=0.698 G=0.133 B=0.133 />
                <scale X=0.08 Y=0.08 Z=0.08 />
                <models>
                        <model file="esfera.3d" />
                </models>
</group>
<!-- Jupiter -->
<group>
        <translate time=40 >
                        <point X=3 Y=0 Z=0 />
                        <point X=0 Y=0 Z=-3 />
                        <point X=-3 Y=0 Z=0 />
                        <point X=0 Y=0 Z=3 />
        </translate>
        <cor R=0.545 G=0.271 B=0.075 />
        <scale X=0.3 Y=0.3 Z=0.3 />
        <models>
                <model file="esfera.3d" />
        </models>
        <!-- Lua Júpiter -->
        <group>
                <translate time=15 >
                        <point X=3 Y=0 Z=0 />
                        <point X=2.5 Y=0 Z=-1.5 />
                        <point X=2 Y=0 Z=-2 />
                        <point X=0 Y=0 Z=-3 />
                        <point X=-1.5 Y=0 Z=-2.5 />
                        <point X=-2 Y=0 Z=-2 />
                        <point X=-3 Y=0 Z=0 />
                        <point X=-2.5 Y=0 Z=1.5 />
                        <point X=-2 Y=0 Z=2 />
                        <point X=0 Y=0 Z=3 />
                        <point X=1.5 Y=0 Z=2.5 />
                        <point X=2 Y=0 Z=2 />
                </translate>
                <cor R=0.859 G=0.439 B=0.576 />
                <scale X=0.3 Y=0.3 Z=0.3 />
                <models>
                        <model file="esfera.3d" />
                </models>
        </group>
```

```
                <group>
                <!-- Saturno -->
                <group>
                        <translate time=65 >
                                <point X=4.5 Y=0 Z=0 />
                                <point X=0 Y=0 Z=-4.5 />
                                <point X=-4.5 Y=0 Z=0 />
                                <point X=0 Y=0 Z=4.5 />
                        </translate>
                        <cor R=0.855 G=0.647 B=0.125 />
                        <scale X=0.2 Y=0.2 Z=0.2 />
                        <models>
                                <model file="esfera.3d" />
                        </models>
                </group>
                <!-- Urano -->
                <group>
                        <translate time=90 >
                                <point X=6 Y=0 Z=0 />
                                <point X=0 Y=0 Z=-6 />
                                <point X=-6 Y=0 Z=0 />
                                <point X=0 Y=0 Z=6 />
                        </translate>
                        <cor R=0.502 G=0.502 B=0 />
                        <scale X=0.16 Y=0.16 Z=0.16 />
                        <models>
                                <model file="esfera.3d" />
                        </models>
                </group>
                <!-- Neptuno -->
                <group>
                        <translate time=120 >
                                <point X=7 Y=0 Z=0 />
                                <point X=0 Y=0 Z=-7 />
                                <point X=-7 Y=0 Z=0 />
                                <point X=0 Y=0 Z=7 />
                        </translate>
                        <cor R=0 G=1 B=1 />
                        <scale X=0.1 Y=0.1 Z=0.1 />
                        <models>
                                <model file="esfera.3d" />
                        </models>
                </group>
        </group>
</scene>
```