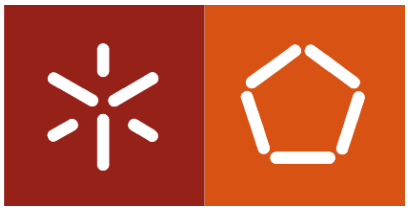

UNIVERSIDADE DO MINHO
Mestrado Integrado em Engenharia Informática
Computação Gráfica

TRABALHO PRÁTICO: 3º FASE

Curvas, Superfícies Cúbicas e VBOs

Grupo 3:
Ricardo Silva — 60995
Bruno Pereira — 72628

Braga, 1 de Maio de 2017

Conteúdo

Lista de Figuras

Lista de Algoritmos

Introdução

Para este projeto é requerido que se criem cenas hierárquicas com transformações geométricas descritas num ficheiro XML. Cada cena é uma árvore onde cada nodo contem um conjunto de transformações geométricas e conjuntos de modelos com os nomes de ficheiros onde estão vértices para a criação da cena. O resultado final é *renderização* de um modelo do sistema solar.

Adicionalmente, os objetivos passam por criar uma função de leitura dos dados a partir de ficheiros, com a criação de uma estrutura ou estruturas de estruturas para armazenamento dos dados, sendo este processo feito apenas uma vez. Também é necessário definir primitivas gráficas e um processo de leitura das estruturas para ser incluído no processo de *rendering*.

Este relatório está divido em duas secções: a secção que documenta o processo de criação e programação de algoritmos para criação de objetos para o sistema solar, no programa `Engine`, e uma segunda secção que documenta o processo de leitura dos dados gerados pelo primeiro programa, estrutura utilizadas para guardar esse dados, e processo de *rendering* iterando sobre as estruturas de dados e aplicando transformações, de forma a construir o sistema solar.

1 Gerador

O `Generator` tem, como função, recebendo parâmetros como comprimento, largura, raio, etc., gerar ficheiros de texto com a extensão `.3d`, cujo conteúdo é a informação sobre as figuras a criar.

Nesta secção ir-se-á descrever o processo de desenvolvimento das figuras necessárias do sistema solar. As figuras pertinentes a desenhar são a esfera (para os planetas e sol) e um disco (para alguns planetas que os tenham, como por exemplo Saturno).

1.1 Esfera

1.1.1 Análise do Problema

Para a construção da esfera teve-se que ter em conta coordenadas esféricas modificadas para o referencial rodado com Y para cima, Z como eixo das abcissas e X como eixo das ordenadas, como demonstra a Equação 1.

$$\begin{cases} x = \cos(\phi) * \sin(\theta) * \rho \\ y = \sin(\phi) * \rho \\ z = \cos(\phi) * \cos(\theta) * \rho \end{cases} \tag{1}$$

Na Equação 1, ρ representa o raio, ϕ o ângulo polar sendo $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, θ representa o ângulo azimutal sendo $\theta \in [0, 2\pi]$.

1.1.2 Diagrama

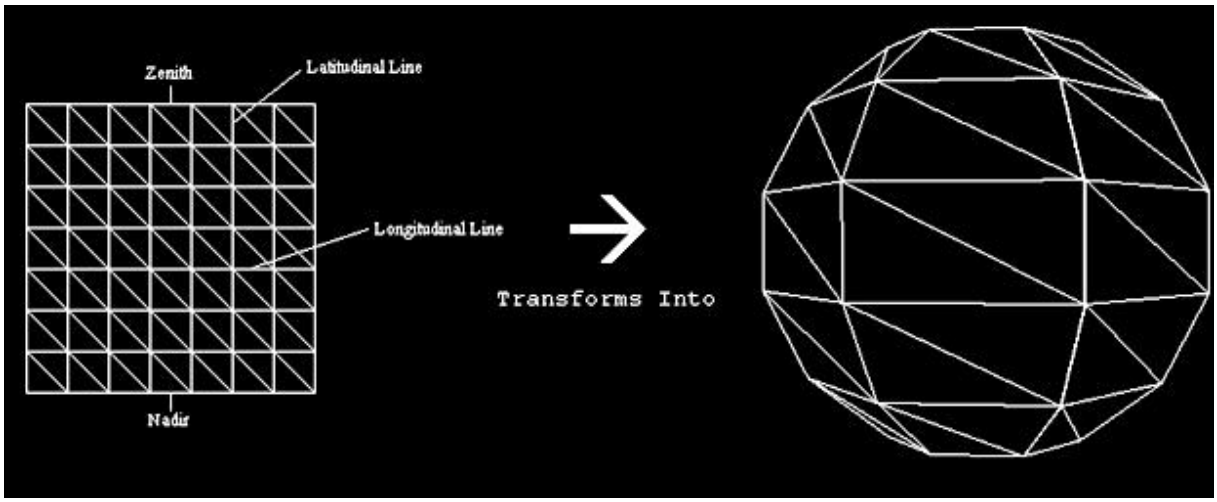

Figura 1: Objetivo do algoritmo de construção de esfera

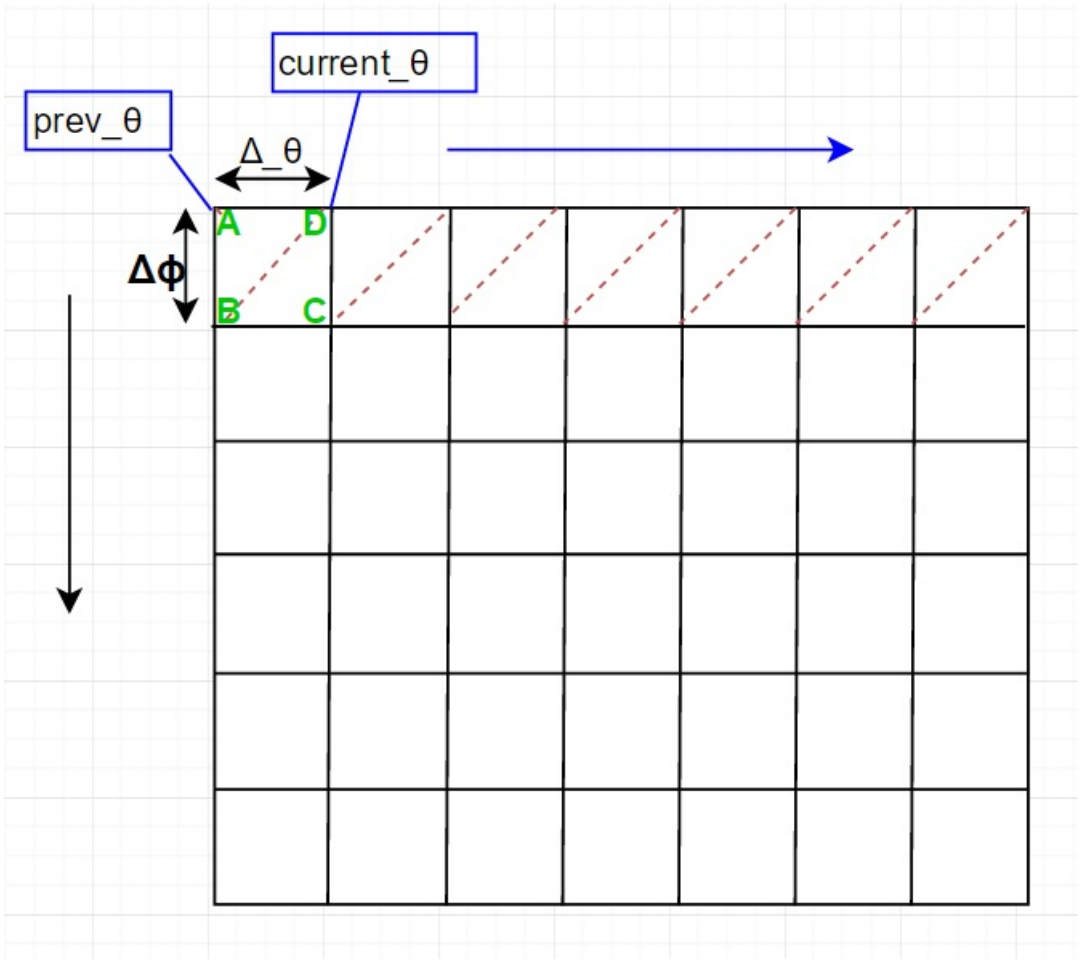

Figura 2: Diagrama de representativo de construção de esfera

No *Figura 2* pode-se ver uma matriz, que representa a esfera nos graus de ϕ e θ para

6 *stacks* e 7 *slices*. Assim como um mapa representativo da Terra, pretende-se mostrar os pontos se a esfera fosse aplanada (ver *Figura 15*).

Em cada quadrícula são calculados 4 pontos iniciais, com base nos cálculos apresentados pelo fórmula anterior. Note-se que, se usou duas variáveis para guardar o ϕ anterior e o ϕ corrente, e θ anterior e θ corrente. Adicionalmente é calculada a diferença de graus entre *slices* e *stacks*, representados por $\Delta\phi$ e $\Delta\theta$, respetivamente.

A intenção é calcular cada quadricula para cada linha e coluna, com auxilio das diferenças dos ângulos e à medida que se avança em cada quadricula, guardar o último grau calculado (ϕ e θ) e calcular nos pontos com o incremento nestes ângulos. Assim desloca-se para a direita na matriz, conforme θ avança de 0 para 2π e para baixo, conforme ϕ avança de $\frac{\pi}{2}$ para $-\frac{\pi}{2}$ (sentido dos ponteiros do relógio). O *Algoritmo 1* representa este processo e a *Figura 3* demonstra o que se mencionou.

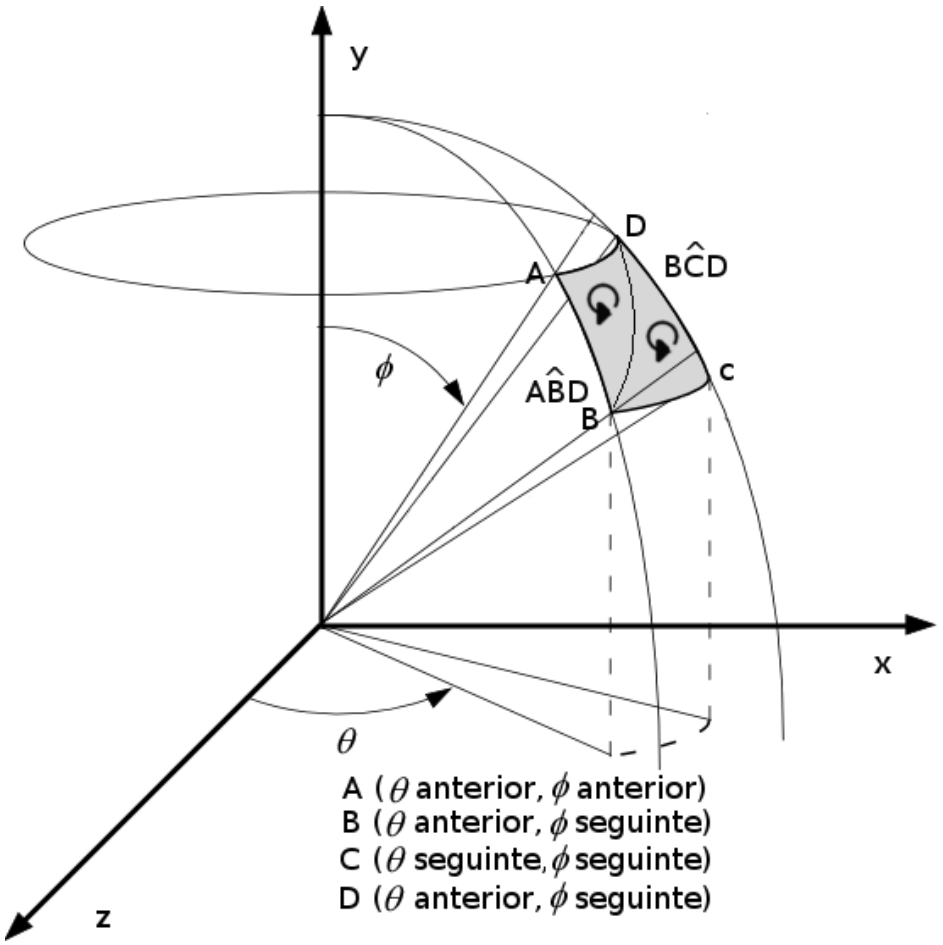

Figura 3: Diagrama de construção de esfera, com eixos, ordem do vértices e ângulos

O resultado pode-se ver na *Figura 4*, que demonstra uma esfera em *wireframe* gerada com a aplicação.

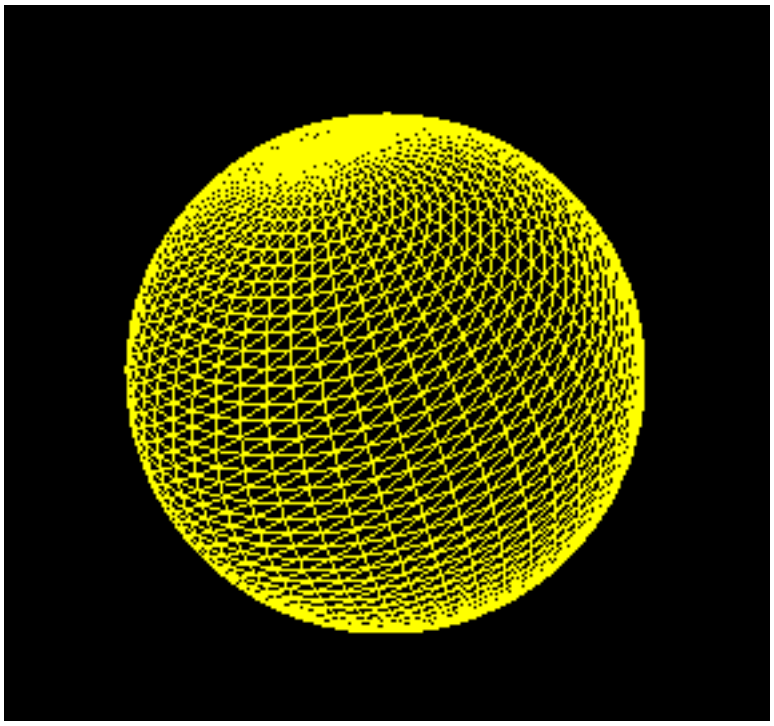

Figura 4: Esfera gerada

Algoritmo 1 Esfera

1: $\Delta\_\theta \leftarrow \frac{2\pi}{slices}$
2: $\Delta\_\phi \leftarrow \frac{2\pi}{stacks}$
3: $prev\_\phi \leftarrow \frac{\pi}{2}$
4: $current\_\phi \leftarrow prev\_\phi - \Delta\_\phi$
5: $i \leftarrow 0$
6: **while** $i \leq stacks$ **do**
7: $prev\_\theta \leftarrow 0$
8: $current\_\theta \leftarrow \Delta\_\theta$
9: $j \leftarrow 0$
10: **while** $j \leq slices$ **do**
11: $PontoA \leftarrow raio * \cos(prev\_\phi) * \sin(prev\_\theta),\ raio * \sin(prev\_\phi),\ raio * \cos(prev\_\phi) * \cos(prev\_\theta)$
12: $PontoB \leftarrow raio * \cos(current\_\phi) * \sin(prev\_\theta), raio * \sin(current\_\phi), raio * \cos(current\_\phi) * \cos(prev\_\theta)$
13: $PontoC \leftarrow raio * \cos(prev\_\phi) * \sin(current\_\theta),\ raio * \sin(prev\_\phi), raio * \cos(prev\_\phi) * \cos(current\_\theta)$
14: $PontoD \leftarrow raio * \cos(current\_\phi) * \sin(current\_\theta), raio * \sin(current\_\phi), raio * \cos(current\_\phi) * \cos(current\_\theta)$

15: $Triangulo(PontoA, PontoB, PontoD)$ ▷ Guardado em ficheiro
16: $Triangulo(PontoB, PontoC, PontoD)$ ▷ Guardado em ficheiro

17: $prev\_\theta \leftarrow current\_\theta$
18: $current\_\theta \leftarrow current\_\theta + \Delta\_\theta$

19: $j \leftarrow j + 1$
20: $prev\_\phi \leftarrow Current\_\phi$
21: $current\_\phi \leftarrow Current\_\phi - \Delta\_\phi$
22: $i \leftarrow i + 1$

1.2 Disco

Nesta secção descreve os procedimentos usados para desenvolver um disco. A motivação para o desenvolvimento desta figura provém da necessidade de representar os anéis que rodeiam os planetas Saturno e Úrano.

1.2.1 Análise do Problema

Existem certos elementos do sistema solar, que são característicos de um modelo do mesmo: anéis e órbitas. Apesar do significado de ambos ser diferente, ambos podem ser desenhados com o mesmo objeto, variando apenas no raio interno e externo.

Com efeito, requer-se para este projeto que se criem discos de vários tamanhos para os anéis de Saturno e Neptuno, e para as órbitas de cada planeta. Note-se que cada anel tem que ter alguma espessura, uma vez que, num plano, no *OpenGL* não se consegue ver o objeto. Assim cada disco terá duas circunferências, uma interior e outra exterior, com raio interno e externo respetivamente. Assim, as duas circunferências têm os mesmos pontos *xx* e *zz* mas com uma distancia fixa no eixo *yy*.

A fórmula para desenhar uma circunferência está representada na *Equação 2*

$$\begin{cases} x = \sin(\theta) * r \\ z = \cos(\theta) * r \end{cases} \tag{2}$$

Nesta secção apresentam-se diagramas que explicam o processo de criação de uma disco.

A *Figura 5* representa a forma como a iteração será feita, bem como apresenta de lado a espessura do disco.

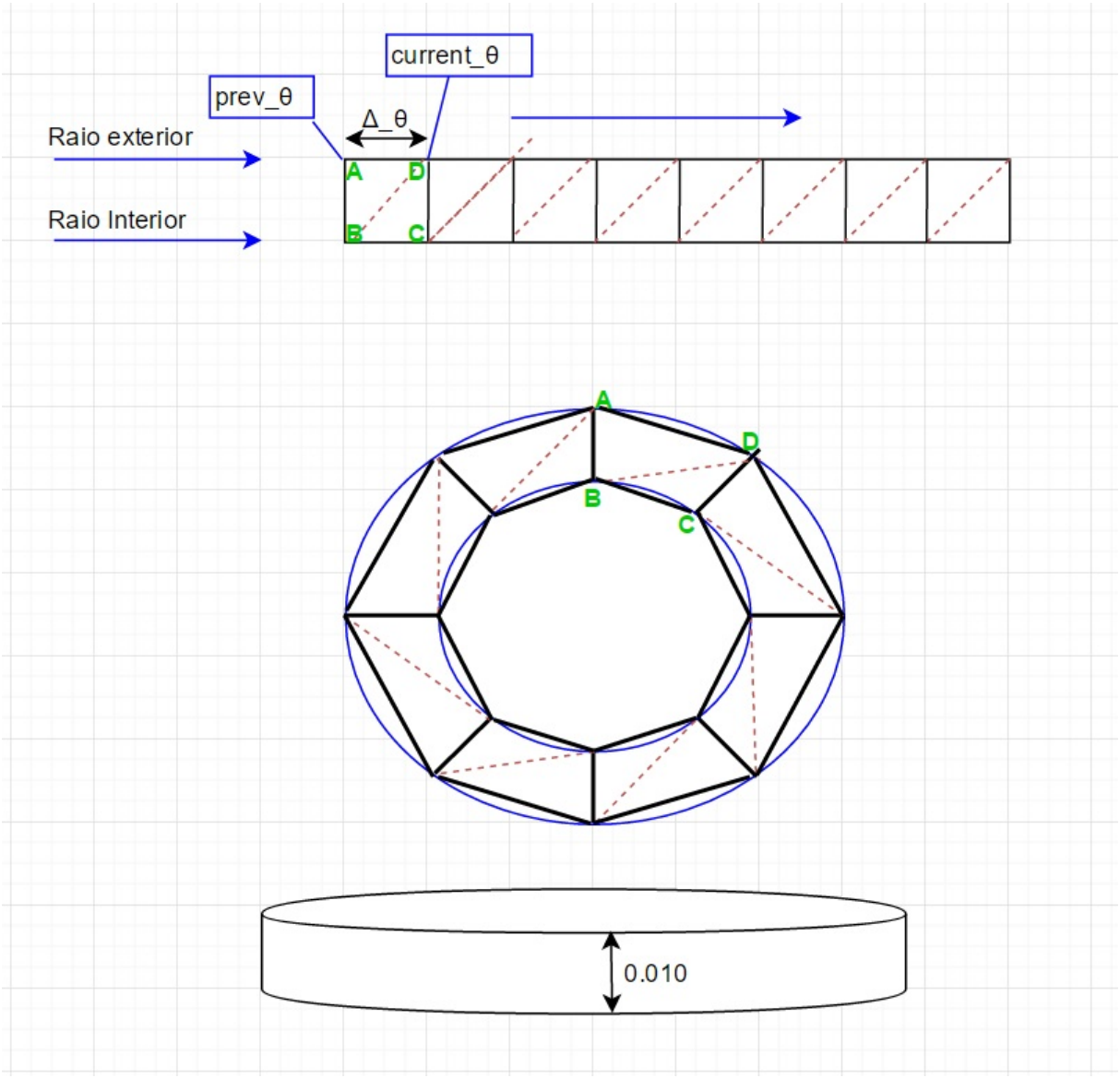

Figura 5: Diagrama Disco

Como se pode verificar o diagrama é relativamente semelhante ao da esfera. A matriz aqui observada apenas tem uma linha porque não se consideram *stacks* na representação do disco. As 8 colunas que representam as 8 *slices* (estas 8 slides servem meramente para propósitos exemplificativos).

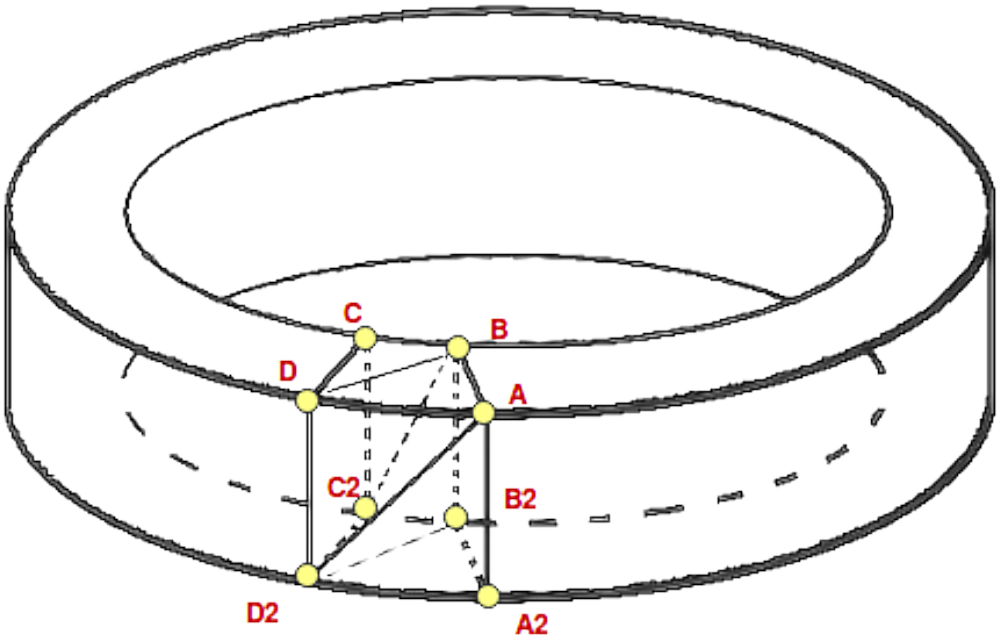

Figura 6: Pormenor dos vértices para desenho de um disco

Como se pode observar, o raciocínio é desenhar quadricula a quadricula com dois triângulos cada, neste exemplo verifica-se que a primeira quadricula é constituída pelos triângulos ABD e BCD. As coordenadas de cada ponto (vértice dos triângulos) são calculados com o auxílio das variáveis angulares prev_θ e current_θ usando a formulação das coordenadas esféricas. A diferença entre esta é o comprimento/largura da quadricula que corresponde a $\frac{2\pi}{slices}$.

Ora este processo é referente à face superior do disco. Para desenhar a face inferior faz-se o mesmo processo mas com outros vértices equivalentes nos eixos *xx* e *zz* mas com uma diferença fixa de 0.010 *yy* para representar a altura.

Quanto à face lateral do disco o processo é idêntico ao representado na matriz acima, mas enquanto que, para representar tanto a face superior como a inferior, os pontos usados têm todos os o mesmo valor *yy*, para representar o lado do disco usa-se uma combinação dos pontos de ambas as faces.

Este processo está representado no *Algoritmo 2*, e um resultado figura na *Figura 7* e na *Figura 8*.

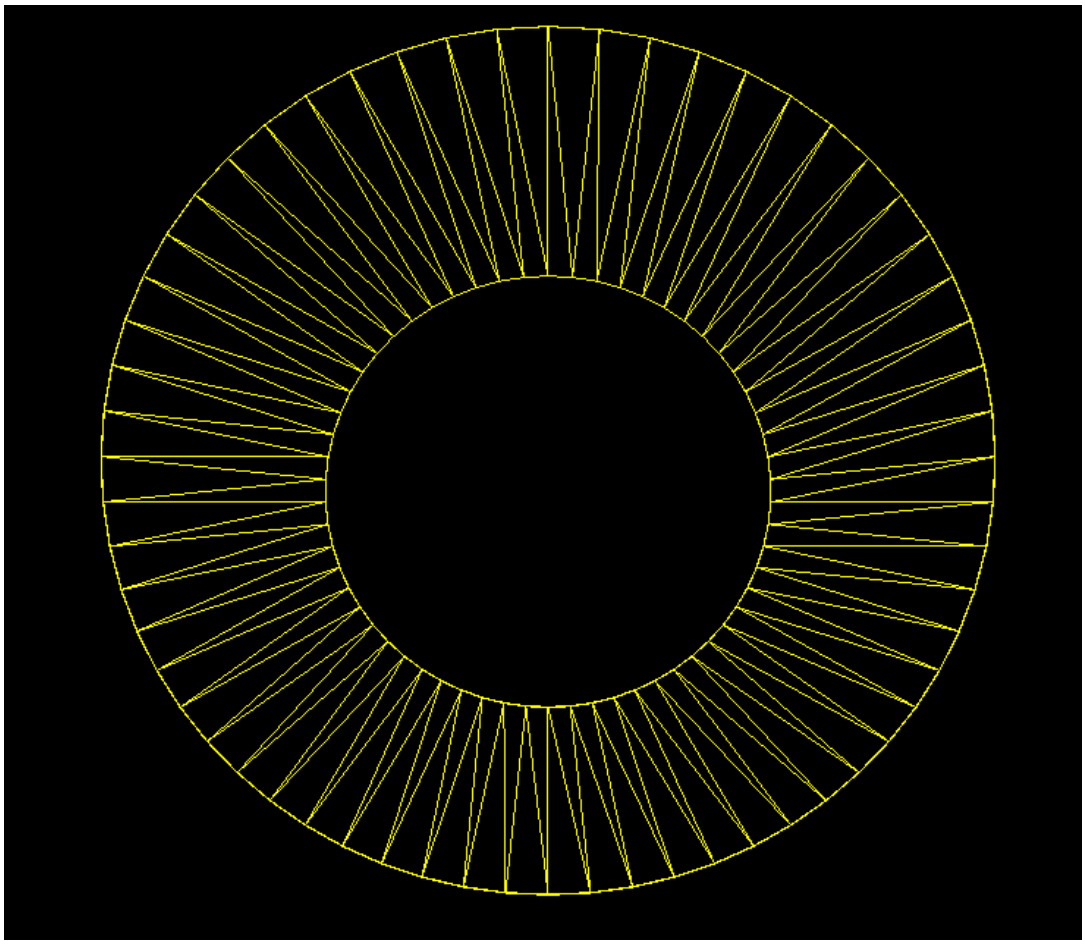

Figura 7: Resultado de um disco em *wireframe*, visto de baixo

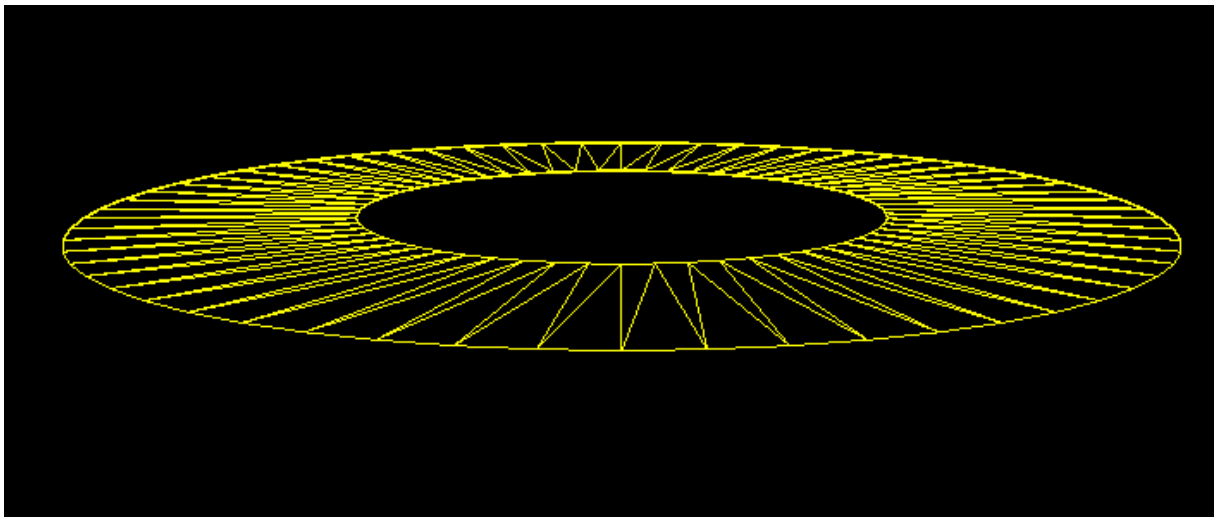

Figura 8: Resultado de um disco em *wireframe*, visto de outro ângulo

Algoritmo 2 Disco

1: $\Delta\_\theta \leftarrow \frac{2\pi}{slices}$
2: $prev\_\theta \leftarrow 0$
3: $current\_\theta \leftarrow \Delta\_\theta$
4: $i \leftarrow 0$
5: **while** $i \leq slices$ **do**
6: $PontoA \leftarrow raioOut * \sin(prev\_\theta), 0.005, raioOut * \cos(prev\_\theta)$
7: $PontoB \leftarrow raioIn * \sin(prev\_\theta), 0.005, raioIn * \cos(prev\_\theta)$
8: $PontoC \leftarrow raioIn * \sin(current\_\theta), 0.005, raioIn * \cos(current\_\theta)$
9: $PontoD \leftarrow raioOut * \sin(current\_\theta), 0.005, raioOut * \cos(current\_\theta)$
10: $PontoA2 \leftarrow raioOut * \sin(prev\_\theta), -0.005, raioOut * \cos(prev\_\theta)$
11: $PontoB2 \leftarrow raioIn * \sin(prev\_\theta), -0.005, raioIn * \cos(prev\_\theta)$
12: $PontoC2 \leftarrow raioIn * \sin(current\_\theta), -0.005, raioIn * \cos(current\_\theta)$
13: $PontoD2 \leftarrow raioOut * \sin(current\_\theta), -0.005, raioOut * \cos(current\_\theta)$

▷ Lado de cima

14: $Triangulo(PontoD, PontoB, PontoA)$ ▷ Guardado em ficheiro
15: $Triangulo(PontoC, PontoB, PontoD)$ ▷ Guardado em ficheiro

▷ Lado de baixo

16: $Triangulo(PontoA2, PontoB2, PontoD2)$ ▷ Guardado em ficheiro
17: $Triangulo(PontoD2, PontoB2, PontoC2)$ ▷ Guardado em ficheiro

▷ Lado de externo

18: $Triangulo(PontoA2, PontoA, PontoD2)$ ▷ Guardado em ficheiro
19: $Triangulo(PontoD, PontoD2, PontoA)$ ▷ Guardado em ficheiro

▷ Lado de interno

20: $Triangulo(PontoB, PontoC2, PontoB2)$ ▷ Guardado em ficheiro
21: $Triangulo(PontoC, PontoC2, PontoB)$ ▷ Guardado em ficheiro
22: $prev\_\theta \leftarrow current\_\theta$
23: $current\_\theta \leftarrow current\_\theta + \Delta\_\theta$
24: $i \leftarrow i + 1$

2 Motor

Nesta secção pretende-se descrever três pontos essenciais desta fase: as estruturas de dados, algoritmos e processos de *rendering*.

2.1 Estruturas de Dados

Como a estrutura do ficheiro XML é uma árvore *n-ária* escolheu-se uma estrutura que se seguiu a mesma lógica. Assim construiu-se um tipo de dados, a partir de um *struct* com vários campos para guardar os valores de cada grupo, onde se encontra um vetor de apontadores para outras estruturas deste tipo `Group`, como se pode ver na *Figura 9*. Além do mais, a estrutura base possui um vetor de *strings* para guardar os nomes de modelos que se encontrarem descritos no ficheiro XML. Existe também, um outro vetor para guardar apontadores de objetos do tipo `Transformation`, que representa de forma genérica qualquer transformação geométrica.

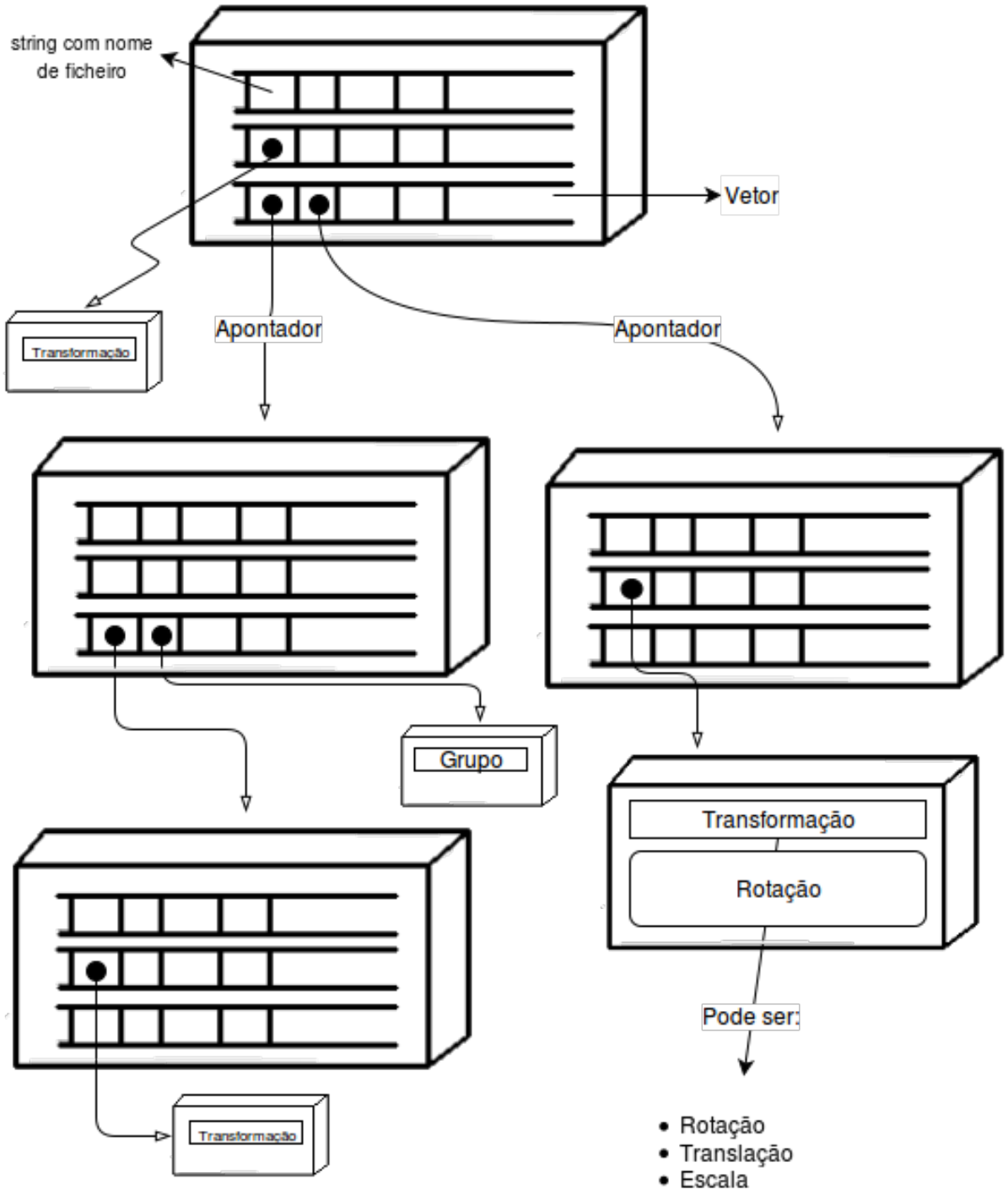

Figura 9: Árvore *n-ária* para armazenamento de grupos

Para a representação genérica de qualquer transformação geométrica, criou-se uma estrutura de classes, onde a classe `Transformation` funciona como superclasse abstrata, com três subclasses: `Rotation`, `Translation` e `Scale`. A superclasse possui um método virtual para aplicação da transformação (`applyTrasformation`), que é aplicado no contexto das suas subclasses quando invado diretamente, ou como parte da estrutura da hierarquia, à custa do

polimorfismo que a linguagem de programação C++ permite. Esta hierarquia pode ser vista com mais detalhe na *Figura 10*. Parte da estrutura de classes é a classe `Point3d` que representa um triplo de `doubles` para as coordenadas geométricas de um ponto em 3 dimensões. Adicionalmente a classe implementa alguns métodos para cálculos de pontos e vetores, embora o significado seja diferente entre estes dois conceitos.

Por último criou-se uma outra estrutura, em que se denominou o tipo `Modelos`, que é composta por um apontador para a raiz da árvore *n-ária* já mencionada, um vetor de `Triangles`, que possuirão os dados dos pontos a serem desenhados. A classe `Triangles` é composta por três `Point3d`, representando os três vértices de um triângulo. O uso daquele vetor é uma medida de eficiência em memória, uma vez que uma cena pode ser composta por objetos criados com o mesmo conjunto de pontos, sendo aplicadas transformações para o efeito desejado.

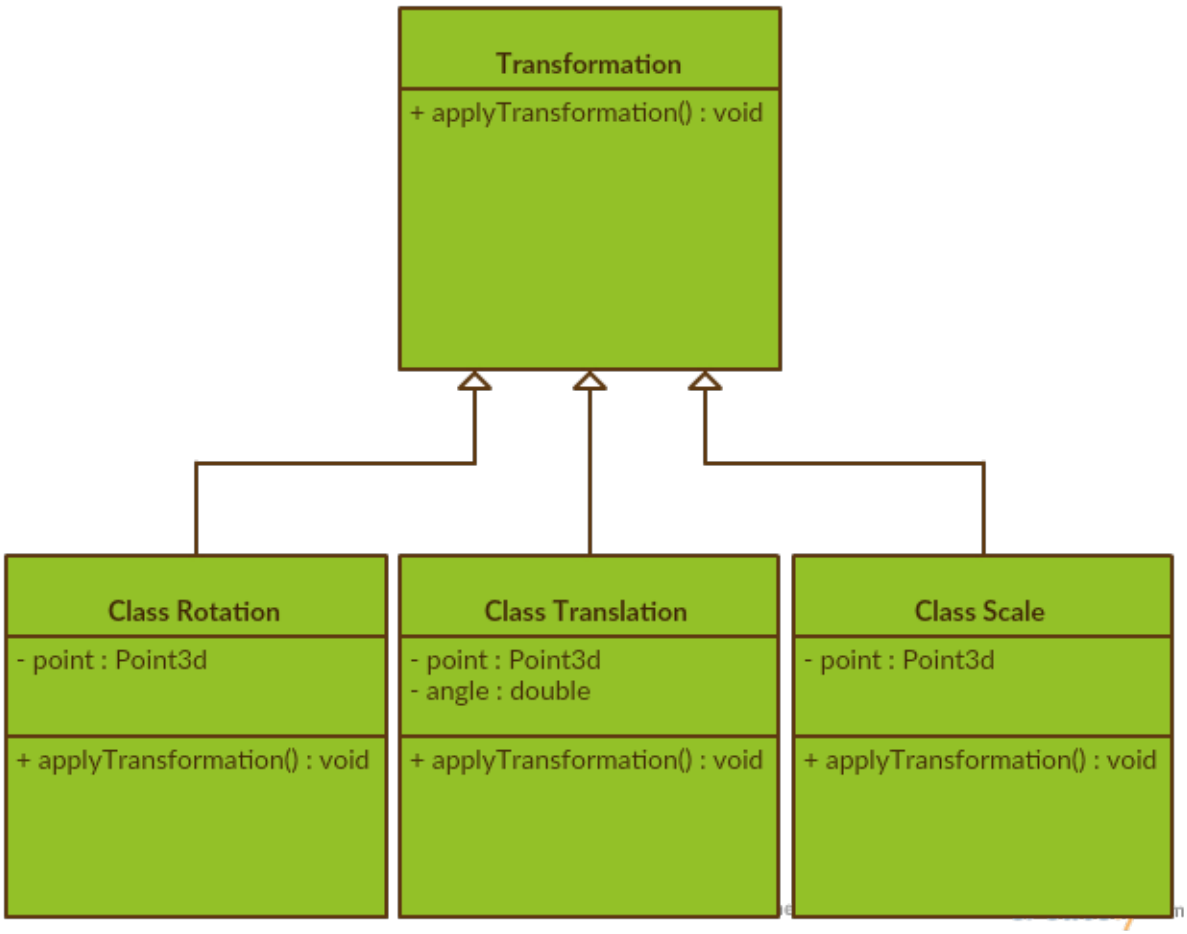

Figura 10: Hieraquia de classes de transformações geométricas

2.2 Descrição do processo de leitura

A função principal de leitura é a que está demonstrada no *Algoritmo 3* e representa parte do processo de leitura, no entanto decidiu-se remover partes acessórias de inicialização de estruturas e afins. Esta função serve-se da estrutura de apontadores da árvore que compõem a estrutura do documento XML para navegar na estrutura recursivamente.

Para além do apontador para o estrutura XML, passou-se por parâmetro um apontador para a estrutura `Modelos`, que contem um apontador para a raiz da árvore *n-ária* e uma tabela (ou mapa) com os valores dos pontos para *rendering* com o nome do ficheiro associado (chave).Note-se que as estruturas em que se guardam valores dos elementos fazem todas parte de um `Group`, tanto como o vetor de *strings* com o valor dos ficheiro, bem como o vetor de transformações e outros `Group` no vetor de grupo, o seu acesso é por uma apontador para `Group`, exceto os pares chave/valor guardados numa tabela na estrutura `Modelos`, com o acesso feito a estrutura por apontador para a mesma.

Algoritmo 3 Função Principal Leitura

```
1: procedure READXMLFROMROOTELEMENT(XMLElement * elem, Modelos * models, Group * grupo)
2:     if elem = NULL then
           return
3:     for all elem ∈ SIBLINGS do
4:         if elem = TRANSLATION ∨ elems = SCALE then
           ▷ os atributos x, y, z são alternativos.Se aparecerem é lhes atribuído um valor. Caso contrário ficam com o valor 0
5:             if elem = TRANSLATION then
6:                 Guardar x, y, z numa transformação como translação
7:             else
8:                 Guardar x, y, z numa transformação como escala
9:         else if elem = ROTATION then
           ▷ os atributos axis_{x,y,z} e angle são alternativos. Se aparecerem é lhes atribuído um valor. Caso contrário ficam com o valor 0
10:            Guardar axis_{x,y,z} e angle numa transformação como rotação
11:        else if elem = MODELS then
12:            for all model ∈ MODELS do
13:                Guardar atributo file no vetor de vector<string> apontado por grupo
14:                Carregar lista de triângulos num map associado a file, apontados por models
15:        else if elem = GROUP then
16:            Criar novo *novo_grupo
17:            READXMLFROMROOTELEMENT(elem, novo_grupo)
```

Como se pode ver no *Algoritmo 3*, as primeira instruções referem-se ao caso de paragem da função recursiva que verifica se o apontador do elemento, representa é um apontador não nulo. Em caso de nulo, a função retorna para o sítio de onde foi invocada. Em seguida itera-se cada elemento que esteja no mesmo nível da árvore do documento XML. podendo o elemento representar uma rotação, uma escala, uma translação, um modelos ou grupos de modelos, e por fim um grupo.

Com efeito, se o elemento encontrado representar uma escala ou uma translação obtêm-

se os atributos *x*, *y* e *z* e cria-se um apontador, alocando memória para um instância da classe `Translation` ou `Rotation` com esse valores, conforme a sua existência. Note-se que no algoritmo, se descreveu de forma breve como se processa cada atributo, ou seja, cada a existência de cada atributo é verificada, sendo lhe atribuído o valor que figura na *tag* caso exista, ou *0* caso contrário.

Por outro lado, se o elemento XML representar uma rotação, cria-se um apontador alocando memória para um instância da classe `Rotation`, com os valores *axisX*, *axisY*, *axisZ* e *angle*. Uma vez mais, a existência deste atributos é verificada, sendo o valor por defeito *0*, o valor contido na *tag*.

Há que fazer notar que, um objeto `Rotation`, `Translation` ou `Scale` são guardados, imediatamente a serem criados, no vetor de `Transformation`, que é a superclasse dos tipos anteriores. O *upcast* é imediato e é uma das caraterísticas da linguagem e de uma hierarquia de classes.

Se o elemento representar um conjunto de modelos, é efetuada uma navegação por apontador para todos os elementos desse conjunto, obtendo o valor do atributo *file*. Este é guardado no vetor de vetor de *strings* de `Group`. De igual modo, através do valor do ficheiro é feita uma leitura do mesmo, que tem os valores dos pontos dos vértices dos triângulos para *rendering*. Note-se que cada linha do ficheiro representa um triângulo, dado que cada linha tem os valores dos vértices separados por ponto e vírgula, e cada vértice tem as coordenadas *x*, *y* e *z* separadas por vírgula. Após o *parsing* da linha os valores são inseridos em instâncias de objetos `Triangle`, que por sua vez serão adicionados a um vetor. Aqui é verificada a existência do nome do ficheiro no conjunto de chaves da tabela, e inserido o par nome do ficheiro/vetor de triângulos no *map* ou tabela.

Por último, caso seja encontrado um elemento grupo, podem ocorrer uma de duas situações: se o grupo está ao mesmo nível do grupo que antecedeu, ou seja é um elemento *irmão* ou se está dentro de um grupo, isto é, é um *filhos* do grupo anterior. No entanto, note-se que não existe uma verificação dos dois caso no algoritmo, sendo efetuada uma chamada recursiva, com um novo elemento grupo. Para ilustrar o caso, atente-se na *Figura 11*. A primeira invocação deste função, representada pelo algoritmo, é precedida pela inicialização de um `Group`, sendo este passado como parâmetro para esta função. Ou seja é a raiz da árvore de `Group` e não possui quaisquer valores. O primeiro elemento que a função encontra é um grupo, como se pode ver na figura, logo tem que ser inicializado e adicionado à raiz, e apontador deste novo `Group` é passado por parâmetro para a chamada recursiva da função. Dentro da chamada recursiva, vão sendo adicionadas as transformações e modelos e se houver um novo grupo, o processo repete-se. Algo de salientar é que a raiz, não tem transformações nem modelos nos respetivos, nunca, e apenas o vetor com os apontadores para os filhos é que é preenchido. Dado que um elemento com a *tag* `scene` pode apenas ter grupos e não transformações, assim se justifica a raiz não ter transformações. Para o caso da *Figura 11*, a raiz terá apenas um *filho*, que será o *pai* de todos os outros grupos.

À medida que vão sendo encontrados elementos XML nulos, ou seja, que não há mais

nada no mesmo nível, a função ainda entra na chamada recursiva, mas logo retorna. Para clarificar, o `Group` para qual foi criada memória, logo antes desta chamada recursiva tem de existir e é uma folha. Além do mais, como demonstra a figura, as sucessivas chamadas recursivas vão retornando e voltando para o sítio onde forma invocadas. Nesta chamadas recursivas, como o apontador para o `Group` pai permanece localmente nessa função, vão sendo adicionados novos os *irmãos*.

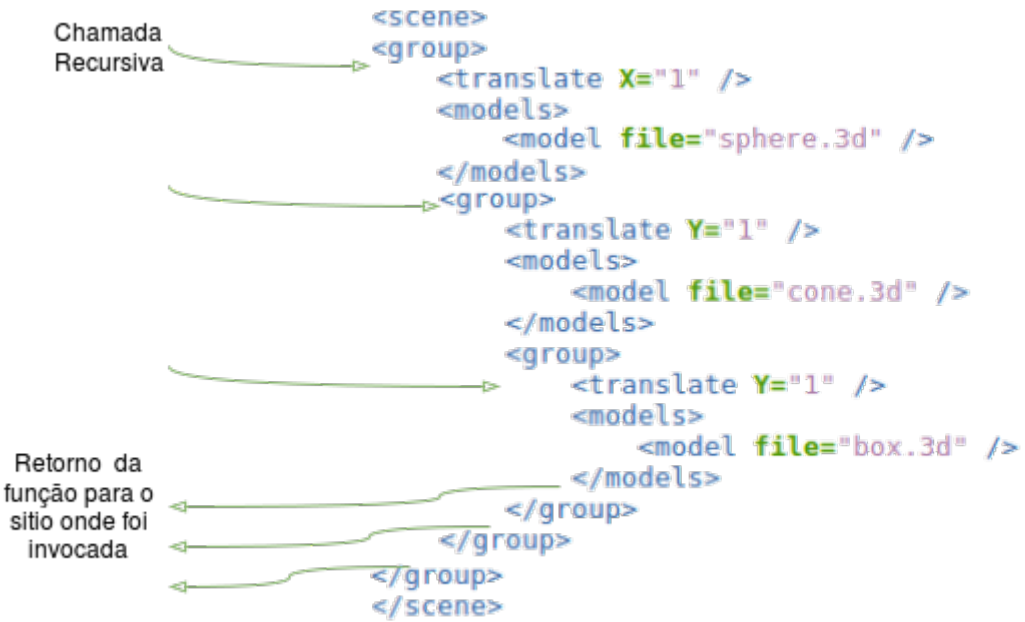

Figura 11: Diagrama representativo da recursividade do processo de leitura

2.3 Descrição do ciclo de *rendering*

Para fazer o *rendering* da estrutura de dados em memória, implementou-se uma função de travessia da árvore, colocada na função `renderScene`, após a função `glLoadIndentity` e `gluLookAt`, nesta sequência. O *Algoritmo 4* representa a função que efetua esta travessia.

Com efeito, a primeira instrução é a `glPushMatrix`, uma vez que se pretende colocar uma matriz para aplicação das transformações no topo da *stack* de matrizes do *OpenGL*. Em seguida, para cada transformação contida num `Group`, é invocada a função `applyTrasformation`, já mencionada na *Secção 2.1*, que aplica as transformações conforme o contexto, como já foi explicado.

Em seguida, é invocada a função `drawElement`, descrita no *Algoritmo 5*. Esta função especifica a primitiva para que será criada com os vértices em memória, neste caso com um triplo de vértices. Os vértices estão contidos, em objetos do tipo `Triangle` que por sua vez, contêm objetos do tipo `Point3d` com as coordenadas de cada vértice. Para obter estes vértices e criar a primitiva com `glVertex3f`, é necessário obtêm todas as *strings* com o nome dos ficheiros no vetor de *string* de cada `Group`. Como cada nome, procura-se pelo nome de ficheiro na tabela a partir do apontador para `Modelos`. Se a entrada existir, obtêm-se todos valores de `Triangle`, respetivos vértices e coordenadas.

No seguimento desta instrução existe um ciclo para aceder a elementos do vetor de apontadores pelo índice para `Group`. Cada elemento (apontador para `Group`) em determinada posição do *array* é passado por argumento para a chamada recursiva da função. Quando o chamada recursiva retorna, o índice é incrementado e o processo repete-se. A função termina quando não houver mais elementos no vetor. Ou seja, o índice é incrementando à medida que a chamadas recursivas entram na memória automática (*stack*) e retornam, saindo da *stack*.

Por último, note-se que a `glPopMatrix` é executada logo após o retorno da chamada recursiva. Uma vez que as transformações sejam herdadas é necessário colocar matrizes na *stack* de matrizes do *OpenGL*, conforme se vai descendo na árvore. Quando a função faz o *pop* à *stack* de matrizes, faz-lo na mesma chamada recursiva da função.

Algoritmo 4 Função de travessia da árvore de `Group`

```
1: procedure TRAVERSETREE(Modelos * models, Group * grupo)
2:     GLPUSHMATRIX( )
3:     for all transformation ∈ grupo → TRANSFORMATIONS do
4:         transformation → APPLYTRANSFORMATION()
5:     DRAWELEMENT(models, grupo)
6:     i ← 0
7:     while i < tamanho do array grupo → FILHOS do
8:         TRAVERSETREE(models, grupo → FILHOS_i)
9:         GLPOPMATRIX( )
10:        i ← i = i + 1
```

Algoritmo 5 Função de *rendering* dos pontos

```
1: procedure DRAWELEMENT(Modelos * models, Group * grupo)
2:     GLBEGIN(GL_TRIANGLES)
3:     for all String ∈ vetor de strings pontado por grupo do
4:         Procurar entry com a string (nome do ficheiro), no map apontado por models
5:         if entry existe then
6:             for all Triângulo ABC ∈ valores (vetor de triângulos) associado à chave do
7:                 Obter vértices do triângulo ABC
8:                 GLVERTEX3F(xA, yA, zA)
9:                 GLVERTEX3F(xB, yB, zB)
10:                GLVERTEX3F(xC, yC,zC)
11:    GLEND( )
```

3 Resultados

Para criar o modelo do sistema solar tomaram-se algumas liberdades em relação à distância dos planetas ao Sol, e como tal os valores das translações são mais ou menos obtidos por experimentação. Não obstante, os planetas estão colocados sobre o eixo dos *xx*. Para as órbitas, teve-se que ter em conta o raio interior do anel (como o anel é bastante fino, é redundante calcular o valor médio), e fazer uma escala ao anéis, é preciso dividir a distância à origem, pelo valor do raio interior do anel. As rotações tiveram como base valores reais, sendo o eixo de rotação o vetor (0,0,1).

Os resultados obtidos estão nas imagens seguintes.

A *Figura 12* mostra o grupo de planetas terrestres e lua da Terra.

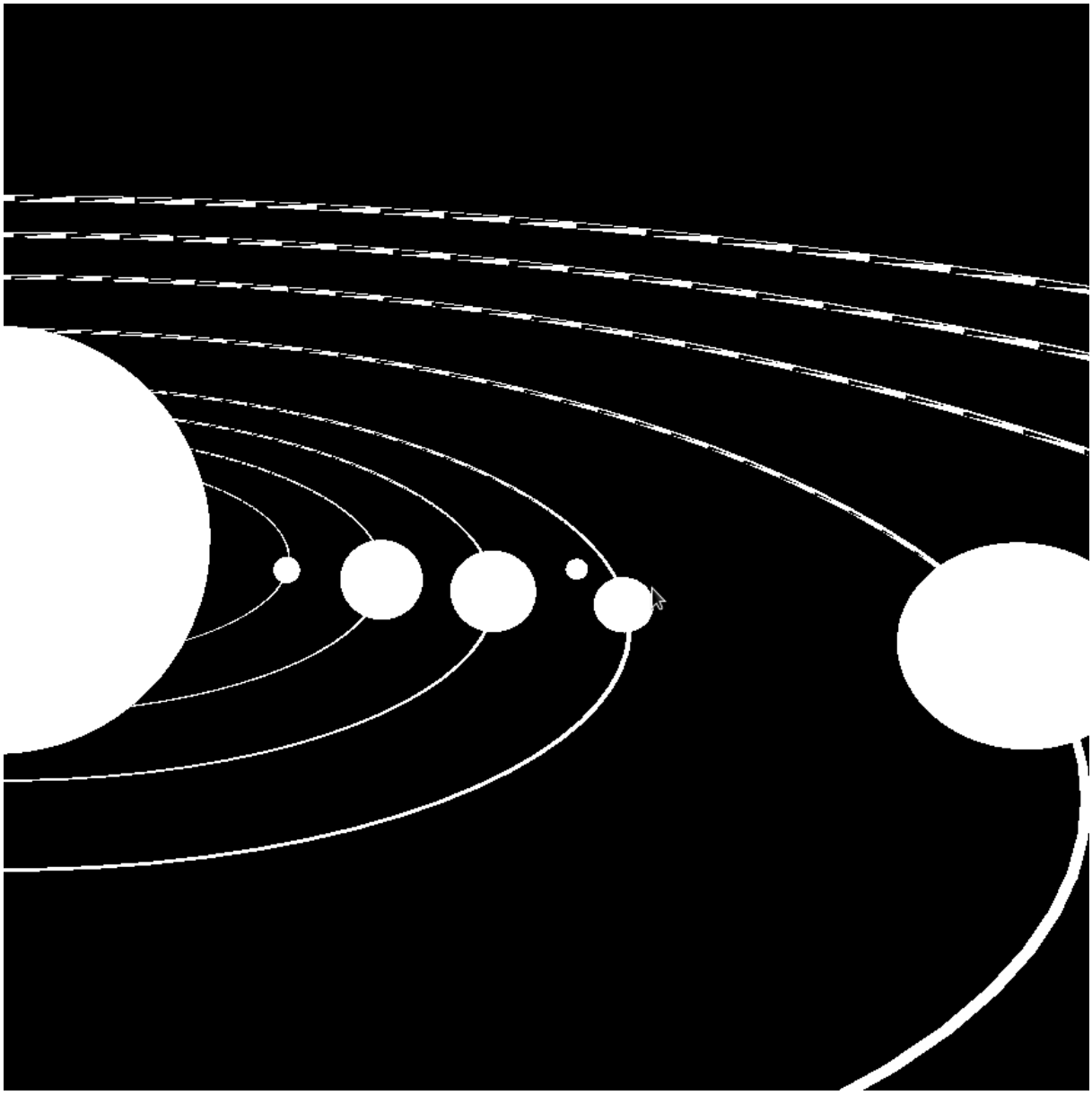

Figura 12: *Rendering* do modelo com foco do grupo de planetas terrestres e Lua

A *Figura 13* mostra Urano e Neptuno, com o *tilt* axial aplicado.

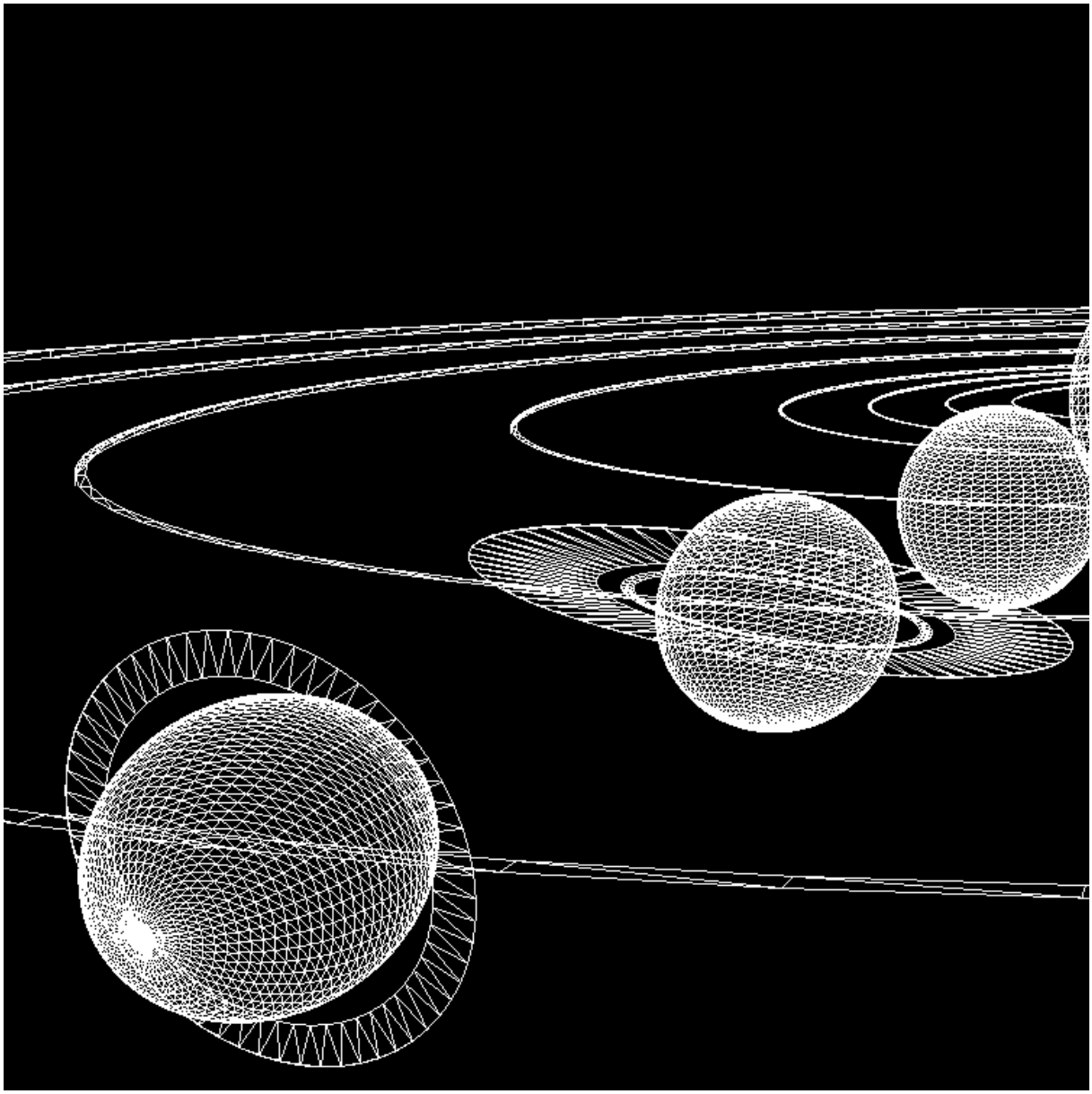

Figura 13: *Rendering* do modelo com foco no *tilt* axial de Urano e Neptuno

A *Figura 14* mostra o modelo completo.

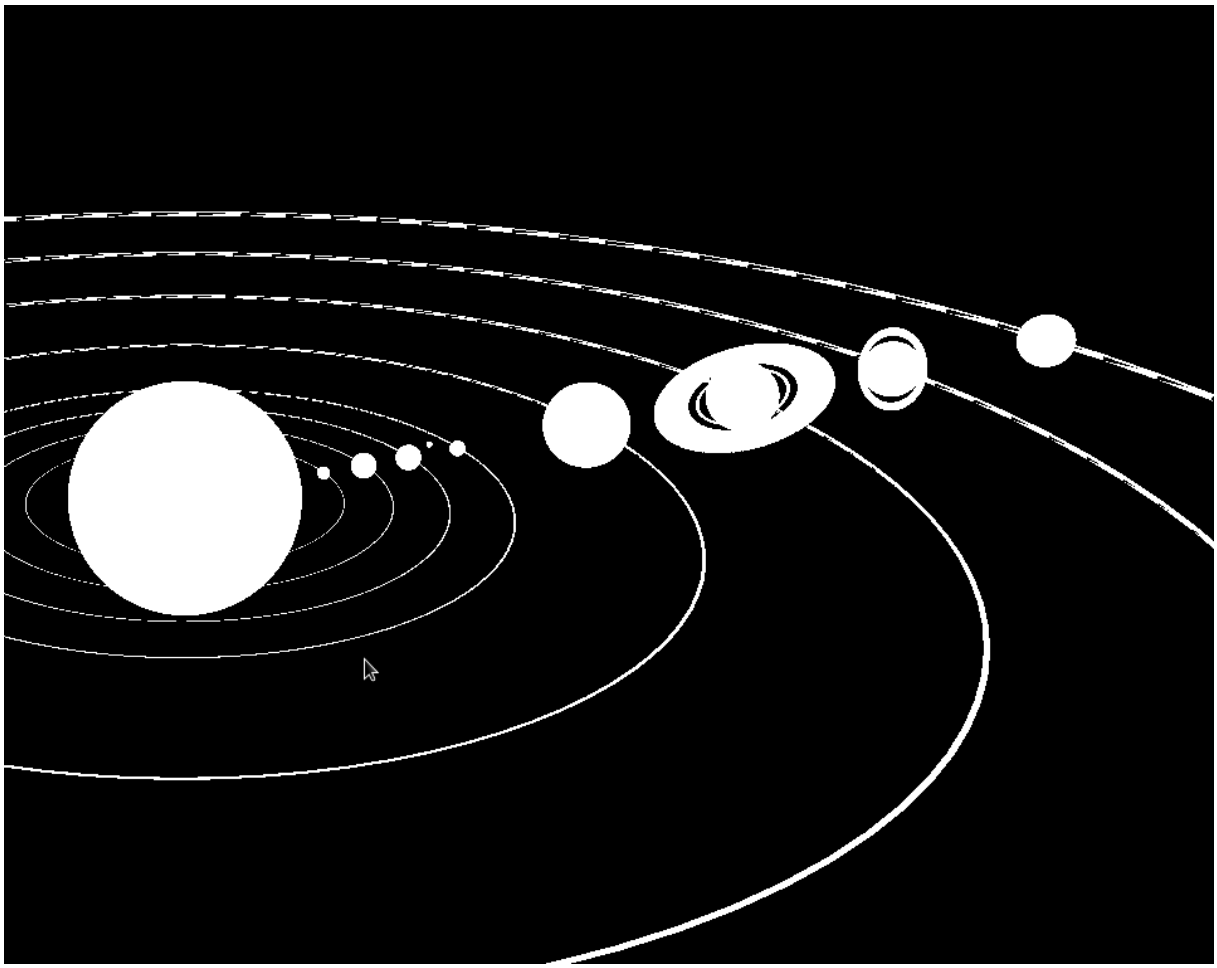

Figura 14: *Rendering* do modelo final

4 Bezier

Uma curva bezier pode ser definida por um qualquer numero de pontos, pontos estes chamados pontos de controlo da curva. Transformações como translação e rotação podem ser aplicadas na curva manipulando estes pontos. Para demonstrar estras tranformações segue-se o exemplo de uma curva de bezier cúbica, isto é, uma curva definida por 4 pontos de controlo:

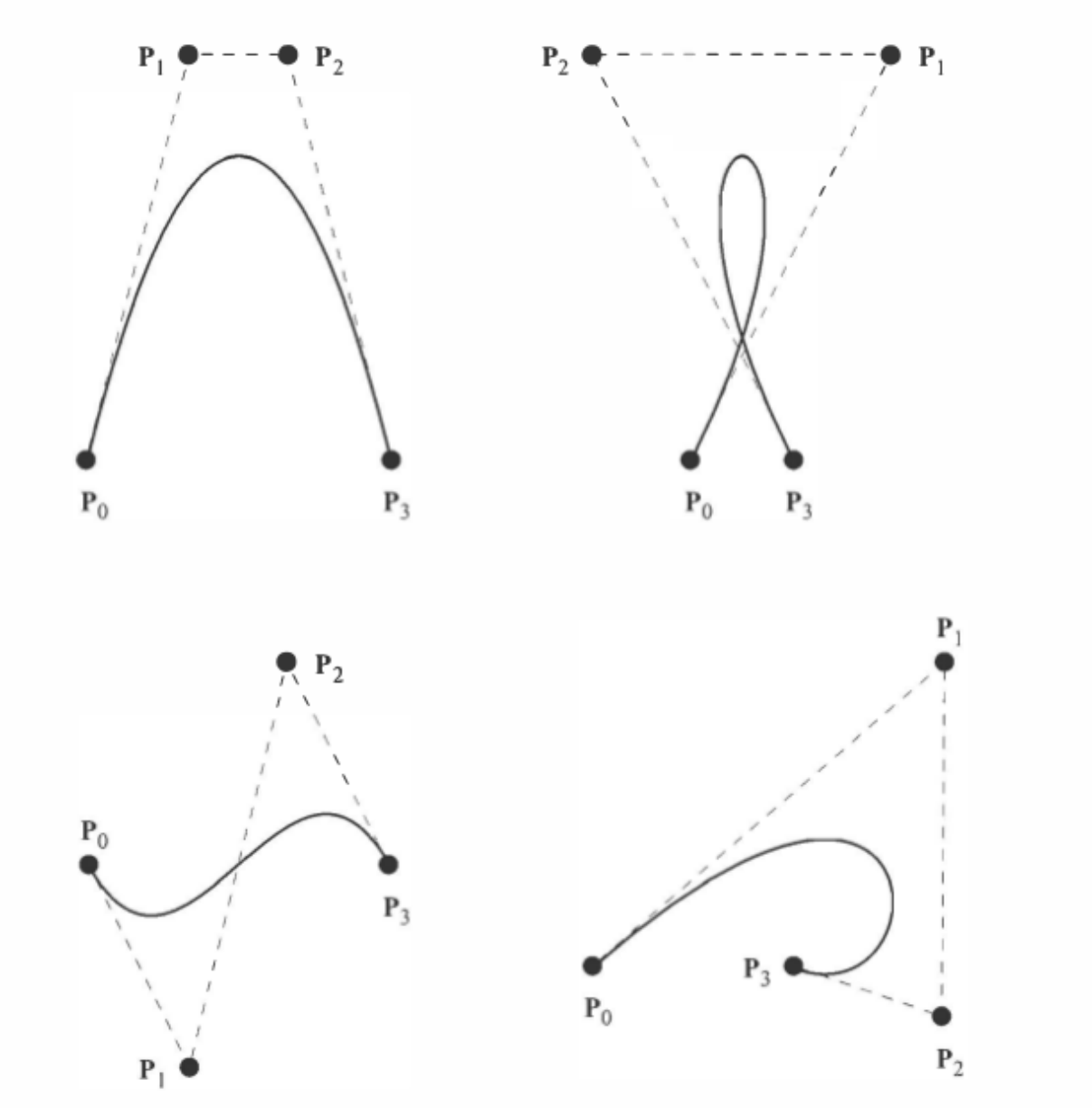

Figura 15: Curvas exemplo de Bezier

Como se pode observar cada uma das curvas observadas possui 4 pontos de controlo, P0, P1, P2 e P3.

As curvas de Bezier, independentemente do número de pontos de controlo que possam ter, são sempre definidas pela seguinte fórmula:

$$B(t) = \sum_{k=0}^{n} B_{n,k}(t) P_k \tag{3}$$

onde n corresponde ao número de pontos de controlo, e t é o polinómio Bernstein que tem sempre um valor positivo em [0,1] e o seu somatório é sempre igual a 1.

Com 4 pontos de controlo, uma curva de Bezier diz-se cúbica e pode ser definida pela seguinte formula:

$$B(t) = \sum_{k=0}^{3} B_{3,k}(t) P_k \tag{4}$$

$$B(t) = t^3 P_3 + 3t^2(1-t)P_2 + 3(t(1-t)^2)P_1 + (1-t)^3 P_0 \tag{5}$$

Quanto maior o número de pontos de controlo maior o custo computacional de as representar.

Conclusão

Em suma, pode-se afirmar que a maioria dos objetivos foram cumpridos, ou seja existe um modelo do sistema solar que pode ser *renderizado* com primitivas gráficas personalizadas, a partir da leitura de um ficheiro XML, cujo os dados são carregados numa estrutura em memória. Além do mais, existe a possibilidade de aplicar transformações a essa primitiva, tais como rotações, translações e escalas.

Com efeito, na primeira secção, relativa ao programa `Generator` que calcula os vértices de triângulos, foram implementas duas funções para desenhar esferas e discos com determinada espessura, para serem utilizados na construção do sistema solar. Adicionalmente, nesta secção são apresentados os algoritmos para o cálculos dos vértices, bem como diagramas explicativos, como também resultados da implementação desse algoritmos, e por último, fórmulas.

Na segunda secção, relativa ao programa `Engine`, que *renderiza* os vértices guardados em ficheiros, foram implementadas rotinas para a leitura de dados, sendo, este processo, feito apenas uma vez, guardando os dados lidos em estruturas de dados adequadas. De igual modo criou-se um algoritmo para iteração nessas estruturas de dados, durante o processo de *rendering*. À semelhança, com a secção anterior, este secção apresenta resultados, algoritmos e diagramas.

No entanto ficaram duas coias por fazer: otimização da função de leitura (*parsing* de linhas), uma vez que este processo tem várias iterações em cada resultado da linha obtida, e a implementação da câmara em primeira pessoa não foi possível, uma vez que, não houve tempo para amadurecer conhecimentos.

Referências

[1] A. Boreskov and E. Shikin, *Computer Graphics: From Pixels to Programmable Graphics Hardware*, 2013.

[2] F. Dunn and I. Parberry, *3D Math Primer for Graphics and Game Development*, 2nd ed. Wordware Pub, 2002.

[3] B. Eckel, *Thinking in C++*. Prentice Hall, 2000.

[4] ——, *Thinking in C++*. Prentice Hall, 2000.

[5] A. Koenig and B. E. Moo, *Accelerated C++ : Practical Programming by Example*. Addison-Wesley, 2000.

[6] E. Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*, 3rd ed. Course Technology PTR, 2012.

[7] S. B. Lippman, J. Lajoie, and B. E. Moo, *C++ Primer*, 5th ed. Addison-Wesley, 2013.

[8] A. Ramires, "GLUT Tutorial," 2011. [Online]. Available: http://www.lighthouse3d.com/tutorials/glut-tutorial/

[9] P. Shirley and S. Marschner, *Fundamentals of Computer Graphics*. CRC Press, 2015.

[10] D. Shreiner and Khronos OpenGL ARB Working Group., *OpenGL Programming Guide : The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*, 7th ed. Addison-Wesley, 2010.

[11] B. Stroustrup, *A Tour Of C++*. Addison-Wesley, 2014.

ANEXOS

A Modelo do Sistema Solar

```
<?xml version="1.0"?>
<scene>
  <!-- Sun -->
  <group>
    <rotate angle="7.25" axisZ="1"/>
    <scale X="17" Y="17" Z="17"/>
    <models>
      <model file="sphere.3d"/>
    </models>
  </group>
  <!-- Mercury -->
  <group>
    <translate X="23"/>
    <rotate angle="0.03" axisZ="1"/>
    <models>
      <model file="sphere.3d"/>
    </models>
  </group>
  <group>
    <scale X="15.33" Y="15.33" Z="15.33"/>
    <models>
      <model file="orbit.3d"/>
    </models>
  </group>
  <!-- Venus -->
  <group>
    <translate X="30"/>
    <scale X="2" Y="2" Z="2"/>
    <rotate angle="2.64" axisZ="1"/>
    <models>
      <model file="sphere.3d"/>
    </models>
  </group>
  <group>
    <scale X="20" Y="20" Z="20"/>
    <models>
      <model file="orbit.3d"/>
    </models>
  </group>
  <!-- Earth -->
  <group>
    <translate X="38"/>
    <scale X="2" Y="2" Z="2"/>
    <rotate angle="23.44" axisZ="1"/>
    <models>
      <model file="sphere.3d"/>
```

```
  </models>
  <!-- Moon -->
  <group>
    <translate X="2"/>
    <scale X="0.25" Y="0.25" Z="0.25"/>
    <rotate angle="6.68" axisX="1"/>
    <models>
      <model file="sphere.3d"/>
    </models>
  </group>
</group>
<group>
  <scale X="25.33" Y="25.33" Z="25.33"/>
  <models>
    <model file="orbit.3d"/>
  </models>
</group>
<!-- Mars -->
<group>
  <translate X="47"/>
  <scale X="1.25" Y="1.25" Z="1.25"/>
  <rotate angle="25.19" axisZ="1"/>
  <models>
    <model file="sphere.3d"/>
  </models>
</group>
<group>
  <scale X="31.33" Y="31.33" Z="31.33"/>
  <models>
    <model file="orbit.3d"/>
  </models>
</group>
<!-- Jupiter -->
<group>
  <scale X="48" Y="48" Z="48"/>
  <models>
    <model file="orbit.3d"/>
  </models>
</group>
<group>
  <translate X="72"/>
  <scale X="7" Y="7" Z="7"/>
  <rotate angle="3.13" axisZ="1"/>
  <models>
    <model file="sphere.3d"/>
  </models>
</group>
<!-- Saturn -->
<group>
  <scale X="70" Y="70" Z="70"/>
  <models>
    <model file="orbit.3d"/>
  </models>
```

```
  </group>
  <group>
    <translate X="105"/>
    <scale X="6" Y="6" Z="6"/>
    <rotate angle="26.63" axisZ="1"/>
    <models>
      <model file="sphere.3d"/>
      <model file="ring.3d"/>
      <model file="ring3.3d"/>
    </models>
  </group>
  <!-- Uranus -->
  <group>
    <scale X="93.33" Y="93.33" Z="93.33"/>
    <models>
      <model file="orbit.3d"/>
    </models>
  </group>
  <group>
    <translate X="140"/>
    <scale X="5" Y="5" Z="5"/>
    <rotate angle="82.23" axisZ="1"/>
    <models>
      <model file="sphere.3d"/>
      <model file="ring2.3d"/>
    </models>
  </group>
  <!-- Neptune -->
  <group>
    <scale X="120" Y="120" Z="120"/>
    <models>
      <model file="orbit.3d"/>
    </models>
  </group>
  <group>
    <translate X="180"/>
    <scale X="5" Y="5" Z="5"/>
    <rotate angle="28.32" axisZ="1"/>
    <models>
      <model file="sphere.3d"/>
    </models>
  </group>
</scene>
```

Listing 1: Código XML com parãmetros para o sistema solar