

**Computação Gráfica**

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

Trabalho Prático - Parte 3

|  |  |
| --- | --- |
| **Nome** | **Nº. Mecanográfico** |
| **António Jorge Monteiro Chaves** | A75870 |
| **Carlos José Gomes Campos** | A74745 |
| **Cesário Miguel Pereira Perneta** | A73883 |
| **Luís Miguel Bravo Ferraz** | A70824 |

# Introdução

No âmbito da Unidade Curricular de Computação Gráfica, numa primeira fase, foi-nos proposto, o desenvolvimento de um gerador de vértices de algumas primitivas gráficas (plano, caixa, esfera e cone).

Para além disto, também foi desenvolvido um mecanismo de leitura de ficheiros de configuração em XML2, que tem como objetivo desenhar os vértices das primitivas gráficas anteriormente geradas, a partir de ficheiros escolhidos pelo utilizador.

Já numa segunda fase, tivemos como objetivo o desenvolvimento de um gerador de vértices de algumas primitivas gráficas que nos permitem desenhar uma maquete do sistema solar, dos quais se destacam a esfera e o anel, tendo sido também desenvolvido um mecanismo de leitura de ficheiros de configuração em XML2, que para além de desenhar, dá cor, translação, rotação e escala às figuras.

Relativamente a esta terceira fase, tivemos como intuito incluir curvas e superfícies cúbicas ao trabalho previamente realizado, desenvolvendo deste modo um modelo dinâmico do Sistema Solar, incluindo também a criação de um cometa.

# Fase 1: Primitivas Gráficas

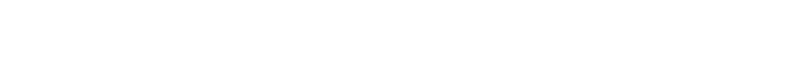
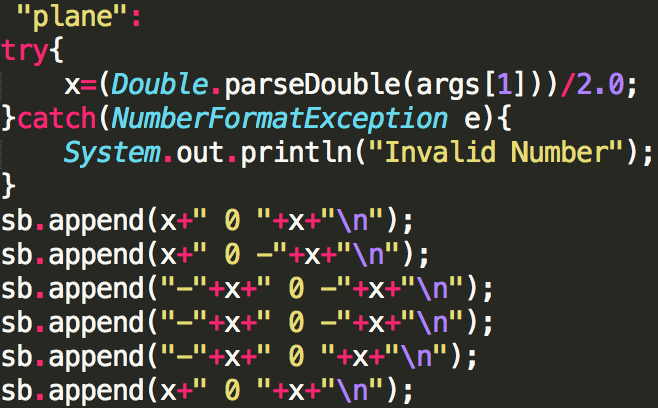
## Plano ZX

O desenho de um plano horizontal no eixo *zOx* é composto apenas por 2 triângulos. A chamada de geração de um plano requer a introdução de um número real, atribuído ao lado do plano. O valor

calculado de 𝑠𝑖𝑑𝑒 = 𝑥 permite-nos desenhar o plano com centro exato no meio do referencial, apenas

2

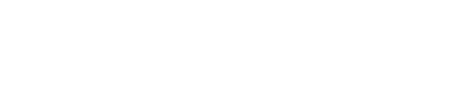
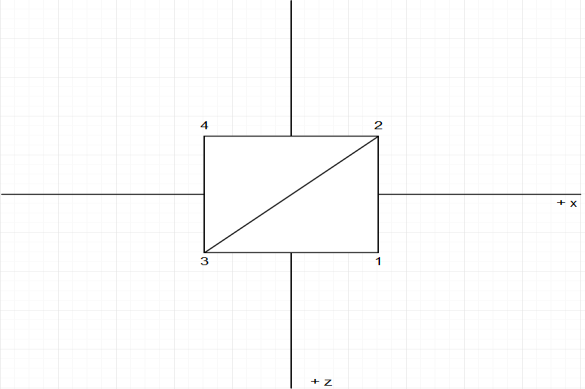
variando os valores de x e z entre [−𝑠𝑖𝑑𝑒, 𝑠𝑖𝑑𝑒]. De modo a que as figuras sejam visíveis pelo seu exterior, todos os triângulos devem ser desenhados tendo em conta a regra da mão direita.



*Figura 1 Pontos do Plano*

O desenho do plano na sua vista aérea (eixo *yy*) está representado no gráfico em baixo. Os pontos assinalados correspondem aos extremos do plano, com os quais serão formados os triângulos.

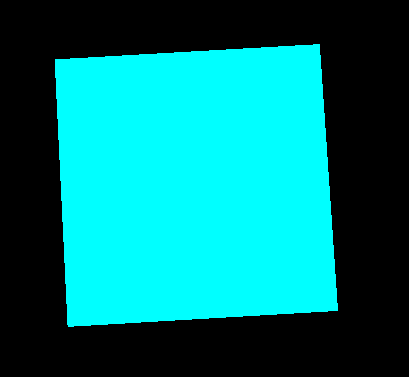
Para que o desenho seja visto desta perspetiva, a ordem pela qual os pontos dos triângulos devem ser escritos é:



*Figura 2 Gráfico do Plano*

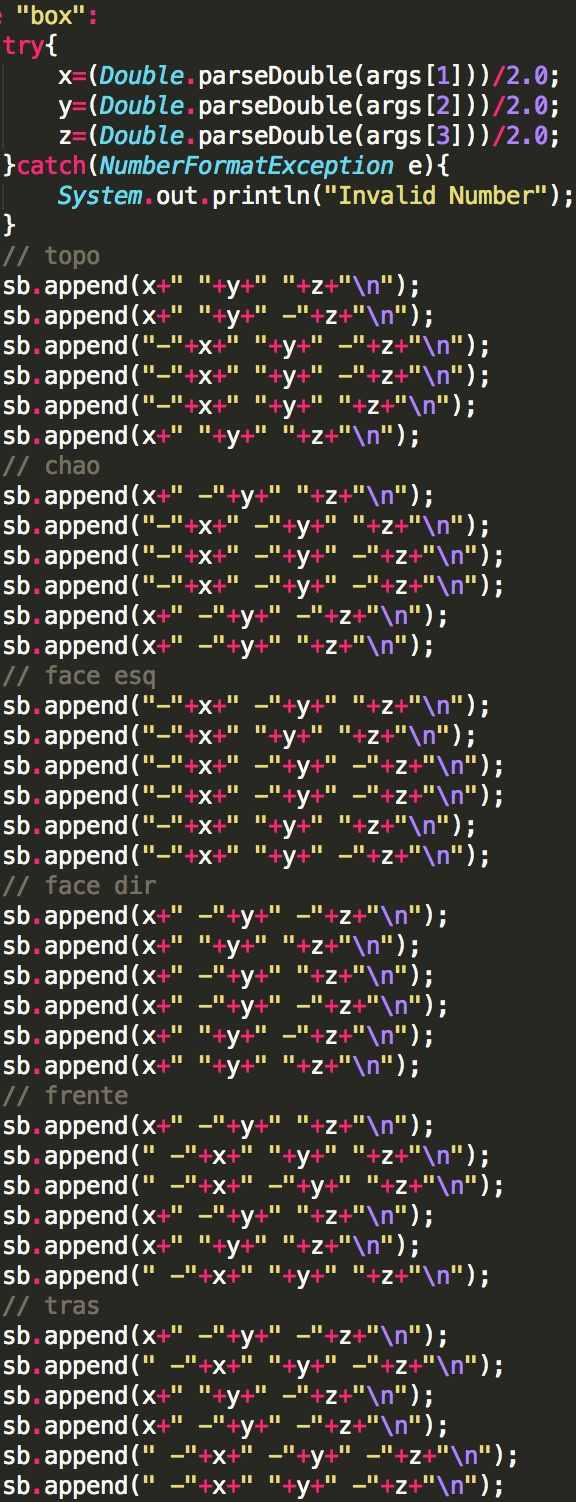
* + - 1,2,3 Para o inferior;
    - 2,4,3 Para o superior.

Os pontos são escritos linha a linha para um ficheiro cujo nome deve ser fornecido aquando da compilação do ficheiro. Esta operação é semelhante para todos casos de utilização do gerador.

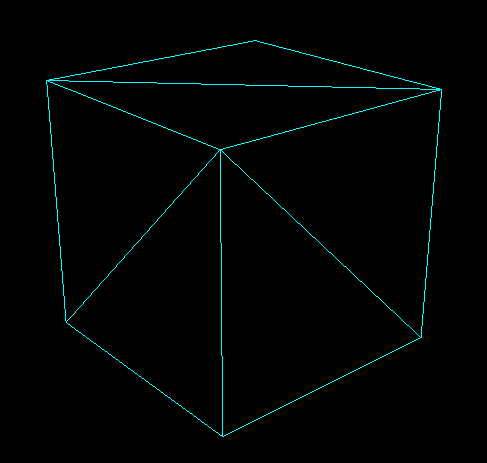
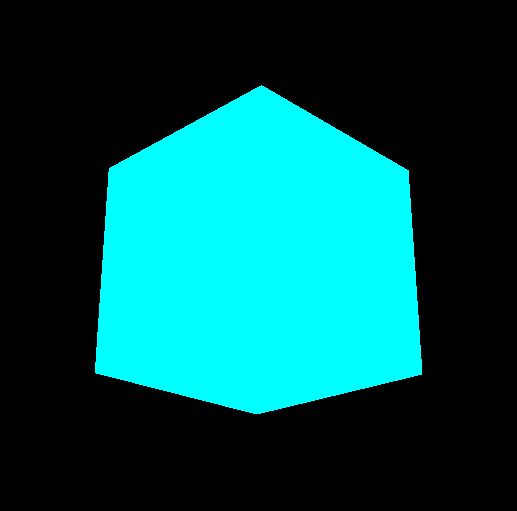


### Figura 3 1ª Demo Scene Plano Figura 4 2ª Demo Scene Plano

* 1. **Caixa**

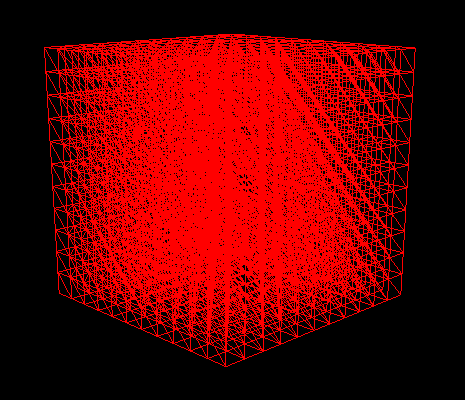
O desenho da caixa é dado por uma estratégia semelhante à do passo anterior. Uma caixa é composta por 6 planos, centrados na origem dos referenciais.

### Figura 5 Pontos Caixa

### Figura 6 Demo Scene Caixa (GL\_LINE) Figura 5 Demo Scene Caixa (GL\_FILL)

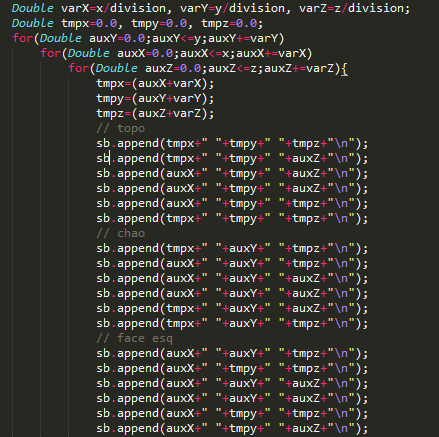
Também é possivel definir a caixa como uma junção de várias caixas de tamanho inferior. Para esse efeito itera-se pelas variáveis e para cada ponto são calculados 12 triângulos. A figura 8 exemplifica o código de três faces de cada Caixa contida dentro da Caixa principal.



### Figura 7 Demo Scene Caixa (iterativa)

* + - Division: Número de stacks e slices;
    - varX: Comprimento do lado do cubo sobre o número de divisões pretendido;
    - varY: Altura do cubo sobre o número de divisões pretendido;
    - varZ: Comprimento do lado do cubo sobre o número de divisões pretendido;
    - tmpx, tmpy, tmpz: Variáveis auxiliares para calcular as coordenadas do ponto seguinte.

Partindo de zero, as variáveis auxY, auxX e auxZ iteram até ao valor das coordenadas respetivas, sendo incrementadas por varY, varX e varZ respetivamente. Em cada iteração de z é desenhada uma linha de cubos, que em conjunto com as iterações de x dá origem ao desenho de uma stack. As iterações em y, por sua vez, indicam a altura de cada stack e o seu valor inicial de y, para que os triângulos sejam desenhados uniformemente e com extremidades concorrentes.



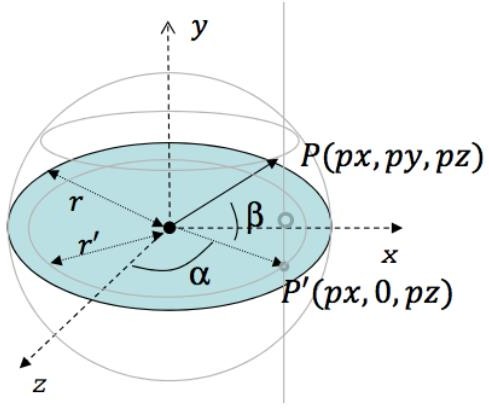
### Figura 8 Pontos Caixa

* 1. **Esfera**

O desenho da esfera requer a discriminação prévia de 3 variáveis, representando o raio da esfera, o número de *slices* horizontais e o número de *stacks* verticais, que definem o número de pontos a calcular. Para utilizar coordenadas esféricas no cálculo dos pontos de cada triângulo, definem-se 2 ângulos – *alfa* e *beta*, em radianos - e 2 incrementos angulares para cada iteração:



### Figura 9 Pontos Esfera

Para cada iteração, partindo de 0.0 e até ao número de slices dado, são calculados 4 pontos (P1, P2, P3 e P4 das páginas 9 e 10) e escritos para o ficheiro de texto indicado na compilação. Este ciclo é repetido para cada valor de beta, que percorre o diâmetro da esfera, ao longo do eixo yy.

1. 𝑎𝑙𝑓𝑎 = 0.0, 𝑎𝑙𝑓𝑎 ∈ [0, 2𝜋], 𝑣𝑎𝑟𝐴𝑙𝑓𝑎 = 2𝜋

𝑠𝑙𝑖𝑐𝑒𝑠

### Figura 10 Coordenadas Esféricas

2. 𝑏𝑒𝑡𝑎 = − 𝜋

2

, 𝑏𝑒𝑡𝑎 ∈ [− 𝜋

2

, 𝜋 2

] , 𝑣𝑎𝑟𝐵𝑒𝑡𝑎 =

2𝜋

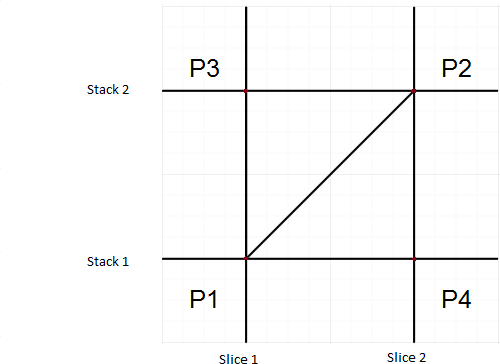
𝑠𝑡𝑎𝑐𝑘𝑠

A descrição de um ponto no espaço, partindo de coordenadas esféricas para cartesianas segue a seguinte transformação:

𝑧 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎) ∗ sin(𝑎𝑙𝑓𝑎) (𝑎𝑙𝑓𝑎, 𝑏𝑒𝑡𝑎, 𝑟) => {𝑥 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎) ∗ cos(𝑎𝑙𝑓𝑎)

𝑦 = 𝑟 ∗ sin(𝑏𝑒𝑡𝑎)

Sendo assim, dados os 2 ângulos e o raio da esfera, podemos calcular as coordenadas cartesianas de qualquer ponto, com a vantagem de se poder calcular pontos próximos, apenas incrementando o

ângulo *alfa* ou *beta*. Para cada *Stack*, *alfa* é inicializado a 0 e percorre os pontos de acordo com o incremento *varAlfa*. Para cada ponto, é preciso calcular as coordenadas de 4 pontos.

O primeiro ponto é o que coincide com a interseção da *slice* e *stack* iniciais de cada iteração, daí os valores das variáveis

𝑎𝑙𝑓𝑎 = 𝑎𝑙𝑓𝑎 & 𝑏𝑒𝑡𝑎 = 𝑏𝑒𝑡𝑎.

### Figura 11 Pontos Triângulos

𝑧 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎) ∗ sin(𝑎𝑙𝑓𝑎)

𝑃1 => {𝑥 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎) ∗ cos(𝑎𝑙𝑓𝑎)

𝑦 = 𝑟 ∗ sin(𝑏𝑒𝑡𝑎)

O segundo ponto situa-se no vértice oposto ao primeiro, na

interseção da *slice* e *stack* seguintes. O cálculo de (*x, y, x*) faz-se agora com 𝑎𝑓𝑙𝑎 = 𝑎𝑙𝑓𝑎 +

𝑣𝑎𝑟𝐴𝑙𝑓𝑎 & 𝑏𝑒𝑡𝑎 = 𝑏𝑒𝑡𝑎 + 𝑣𝑎𝑟𝐵𝑒𝑡𝑎.

𝑧 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎 + 𝑣𝑎𝑟𝐵𝑒𝑡𝑎) ∗ sin(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

𝑃2 => {𝑥 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎 + 𝑣𝑎𝑟𝐵𝑒𝑡𝑎) ∗ cos(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

𝑦 = 𝑟 ∗ sin(𝑏𝑒𝑡𝑎 + 𝑣𝑎𝑟𝐵𝑒𝑡𝑎)

Relativamente ao cálculo das coordenadas do terceiro ponto, pode observar-se que, em comparação com P1, se mantém o valor do ângulo horizontal *alfa*, enquanto que o valor de *beta* corresponde ao da próxima *stack*. O processo é análogo para P4, que no seu caso vê *alfa* a ser incrementado.

𝑧 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎 + 𝑣𝑎𝑟𝐵𝑒𝑡𝑎) ∗ sin(𝑎𝑙𝑓𝑎)

𝑃3 => {𝑥 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎 + 𝑣𝑎𝑟𝐵𝑒𝑡𝑎) ∗ cos(𝑎𝑙𝑓𝑎)

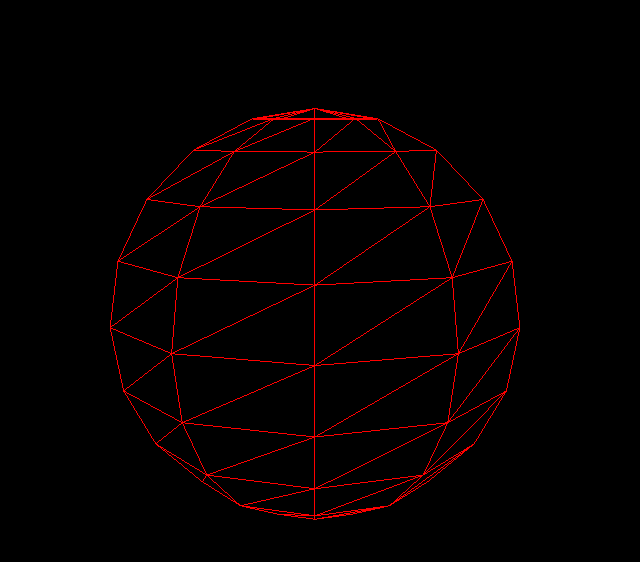
𝑦 = 𝑟 ∗ sin(𝑏𝑒𝑡𝑎 + 𝑣𝑎𝑟𝐵𝑒𝑡𝑎)

𝑧 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎) ∗ sin(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

𝑃4 => {𝑥 = 𝑟 ∗ cos(𝑏𝑒𝑡𝑎) ∗ cos(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

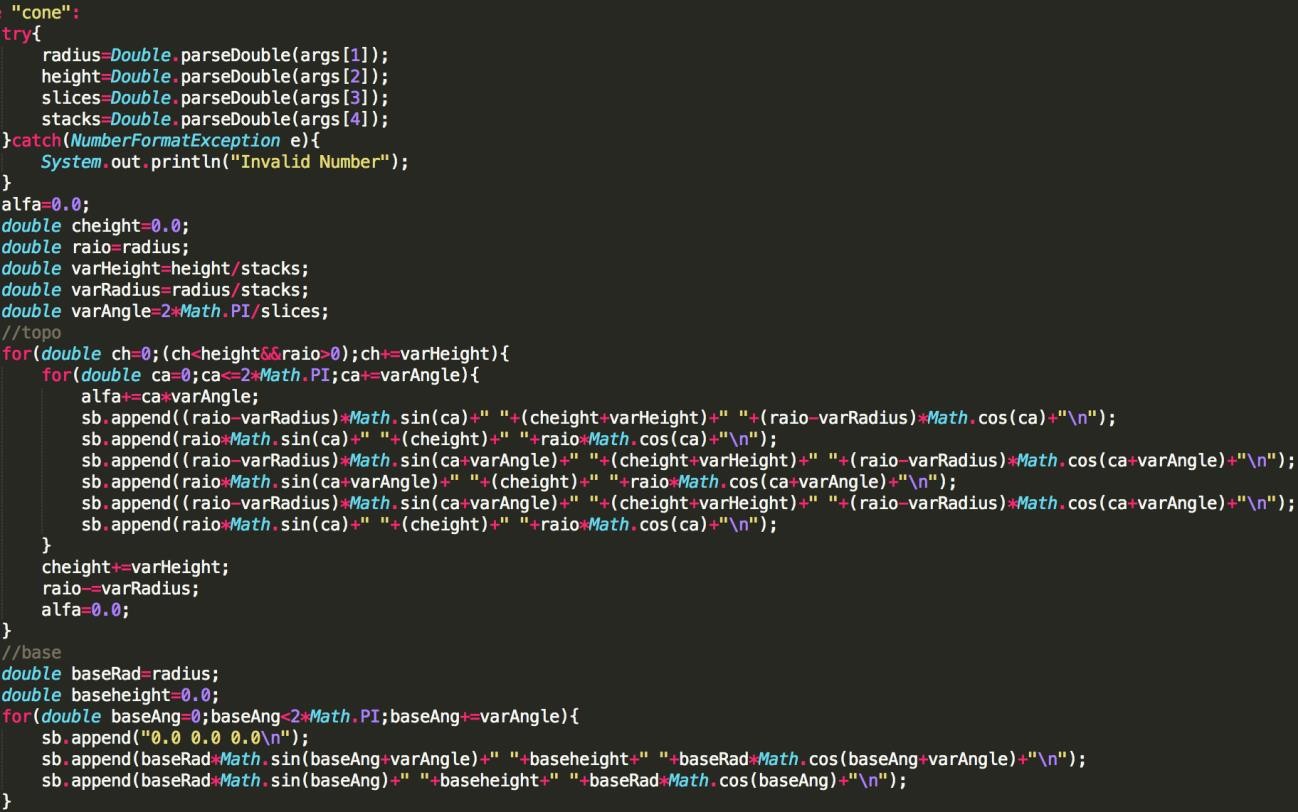
𝑦 = 𝑟 ∗ sin(𝑏𝑒𝑡𝑎)

Os 4 pontos calculados permitem desenhar 2 triângulos por ponto, por ordem P1, P2, P3 e P1, P4, P2. O programa gera para cada *stack* um “anel” de triângulos, ou seja, para cada iteração em *beta*, *alfa* percorre os pontos todos do anel.



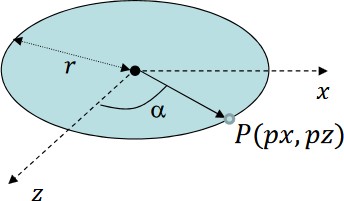
*Figura 12 Demo Scene Esfera (GL\_LINE)*

## Cone

Por último, o desenho do cone parte da transformação de variáveis polares em variáveis cartesianas, passando a altura do cone no parâmetro *y*. O conjunto de pontos calculados origina a face cónica do cone. Assim como no ponto anterior, a estrutura é percorrida gerando um anel de triângulos para cada *stack.* O valor da altura é fornecido aquando da compilação do ficheiro, junto com os valores do raio da base, número de *stacks* e número de *slices*.

### Figura 13 Pontos Cone

Para cada iteração de ch, variável que representa as stacks pelo incremento da altura em yy, são escritos, num ficheiro de texto, 3 coordenadas (1 ponto) por linha, referentes aos 4 diferentes pontos da face lateral do cone.

Os valores iniciais atribuídos às variáveis de altura e ângulo são os seguintes

ℎ𝑒𝑖𝑔ℎ𝑡 = 0.0, 𝑣𝑎𝑟𝐻𝑒𝑖𝑔ℎ𝑡

= ℎ𝑒𝑖𝑔ℎ𝑡/𝑠𝑡𝑎𝑐𝑘𝑠

2𝜋

𝑎𝑙𝑓𝑎 = 0.0, 𝑣𝑎𝑟𝐴𝑙𝑓𝑎 =

𝑠𝑡𝑎𝑐𝑘𝑠

### Figura 14 Coordenadas Polares

A construção da primeira parte da figura calcula os pontos dos triângulos por uma ordem igual à das esferas, tendo em conta a transformação, neste caso, a partir de coordenadas polares.

(𝑎𝑙𝑓𝑎, 𝑟) = {

𝑥 = 𝑟 ∗ cos(𝑎𝑙𝑓𝑎)

𝑦 = ℎ𝑒𝑖𝑔ℎ𝑡

𝑧 = 𝑟 ∗ sin(𝑎𝑙𝑓𝑎)

Os 4 pontos calculados para cada iteração de *alfa* são agora dados por:

𝑃1 = {

𝑥 = 𝑟 ∗ cos(𝑎𝑙𝑓𝑎)

𝑦 = ℎ𝑒𝑖𝑔ℎ𝑡

𝑧 = 𝑟 ∗ sin(𝑎𝑙𝑓𝑎)

𝑥 = (𝑟 − 𝑣𝑎𝑟𝑅𝑎𝑑𝑖𝑢𝑠) ∗ cos(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

𝑃2 = {

𝑦 = ℎ𝑒𝑖𝑔ℎ𝑡 + 𝑣𝑎𝑟𝐻𝑒𝑖𝑔ℎ𝑡

𝑧 = (𝑟 − 𝑣𝑎𝑟𝑅𝑎𝑑𝑖𝑢𝑠) ∗ sin(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

𝑃3 = {

𝑥 = (𝑟 − 𝑣𝑎𝑟𝑅𝑎𝑑𝑖𝑢𝑠) ∗ cos(𝑎𝑙𝑓𝑎)

𝑦 = ℎ𝑒𝑖𝑔ℎ𝑡 + 𝑣𝑎𝑟𝐻𝑒𝑖𝑔ℎ𝑡

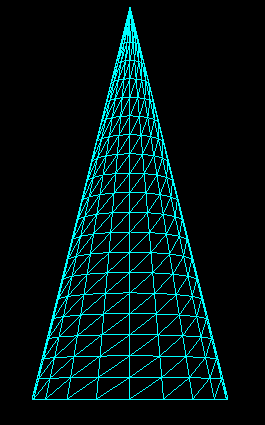
𝑧 = (𝑟 − 𝑣𝑎𝑟𝑅𝑎𝑑𝑖𝑢𝑠) ∗ sin(𝑎𝑙𝑓𝑎)

𝑃4 = {

𝑥 = 𝑟 ∗ cos(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

𝑦 = ℎ𝑒𝑖𝑔ℎ𝑡

𝑧 = 𝑟 ∗ sin(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

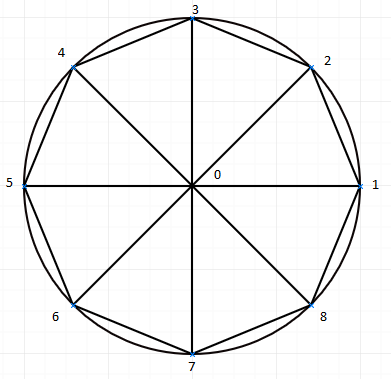


### Figura 16 Demo Scene Cone

Observando o cone, pudemos afirmar que sempre que se pretende calcular um ponto que se encontra uma *stack* acima do original (pontos P2 e P3) é necessário incrementar a altura e decrementar o raio, ambos em proporção. O ângulo *alfa*, por sua vez, a cada ciclo da função oscila entre 0 e 2 𝜋.

O desenho da base do cone faz-se partindo do centro da figura, juntando, para cada iteração, o ponto de ordem original, precedido do ponto de ordem seguinte.

𝑥 = 0.0

𝑃1 = {𝑦 = 0.0

𝑧 = 0.0

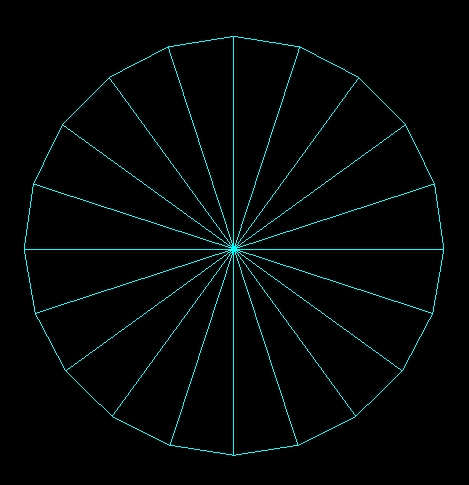
𝑥 = 𝑟 ∗ cos(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

𝑃2 = {

𝑦 = 0.0

𝑧 = 𝑟 ∗ sin(𝑎𝑙𝑓𝑎 + 𝑣𝑎𝑟𝐴𝑙𝑓𝑎)

### Figura 15 Exemplo de divisão do círculo



*Figura 17 Exemplo de divisão do círculo*

𝑃3 = {

𝑥 = 𝑟 ∗ cos(𝑎𝑙𝑓𝑎)

𝑦 = 0.0

𝑧 = 𝑟 ∗ sin(𝑎𝑙𝑓𝑎)

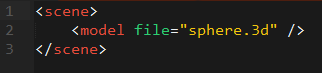
## Motor

O ficheiro *Engine.h* tem como função a leitura dos pontos previamente gerados nos ficheiros “.3d”. A função *xmlParser* começa por ler o ficheiro *config.xml* e retira a informação das figuras geométricas a desenhar. Para cada ficheiro, são lidas linha a linha as coordenadas *x, y e z*, e guardados numa instância da *class Point*, que por sua vez é adicionada a uma instância da *class Model*. No final da função o modelo é guardado num vetor invocado na função *processModels* que, para uma lista de modelos já compilados desenha os respetivos triângulos, através da sua chamada na função *renderScene*.



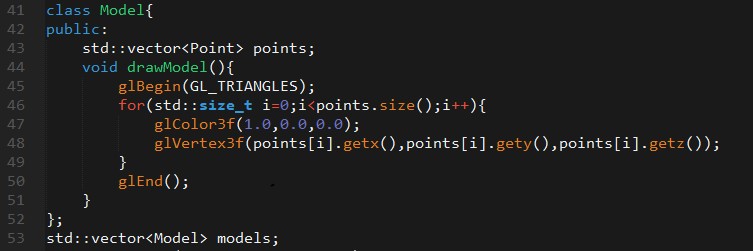
### Figura 18 Ciclo xmlParser

A cada tag *<model>* lida, copia-se para uma string o nome do ficheiro *.3d* que é o argumento do atributo *file* de cada tag. O conjunto destas tags está sempre delimitado pelas tags <scene> </scene>.



### Figura 19 Exemplo de ficheiro de configuração

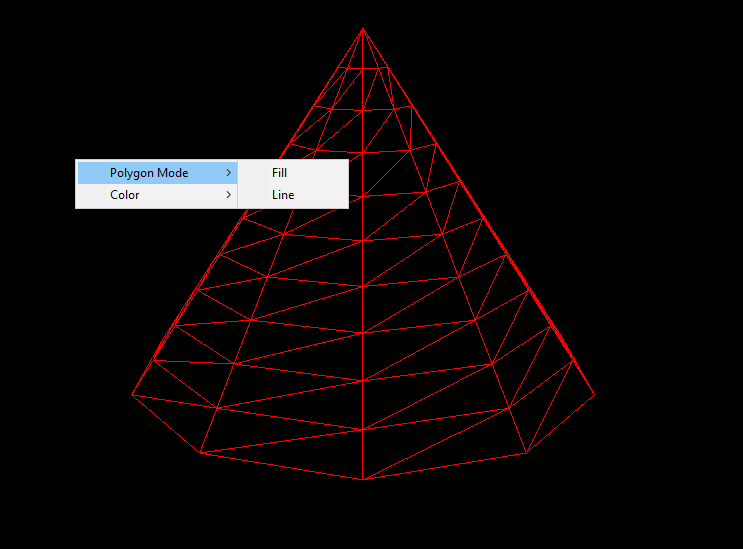
É aberta uma stream do ficheiro .3d cujos pontos já previamente escritos são lidos por uma *stringstream* para cada linha e separados pelas variáveis px, py e pz, antes de serem usados como parâmetros de uma nova instância da classe Point. Ao final da leitura do ficheiro, é procurada a próxima tag *<model* e o processo repete-se, para uma nova instância da class Model.



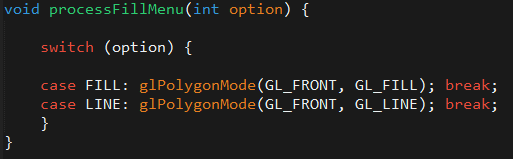
### Figura 20 Classe Model e seu método drawModel

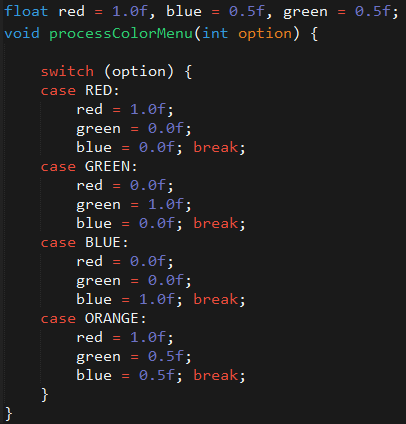
Para desenhar as figuras, é chamada a função drawModel de cada Model e chamada a função glVertex3f para o desenho de cada ponto, obtendo as suas coordenadas através de getx, gety e getz. A cada conjunto de 3 iterações da função é desenhado um triângulo.

Após o desenho das primitivas das figuras, foram inseridos Menus que permitem alterar a o preenchimento e cor das mesmas. Para aceder ao Menu basta apenas carregar na figura com o botão direito do rato.



### Figura 21 Exemplo Menu

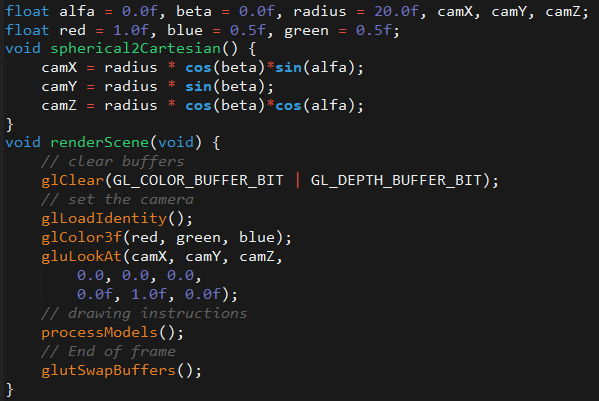


*Figura 22 Menu de tipo de desenho*

*Figura 23 Menu de Cor*

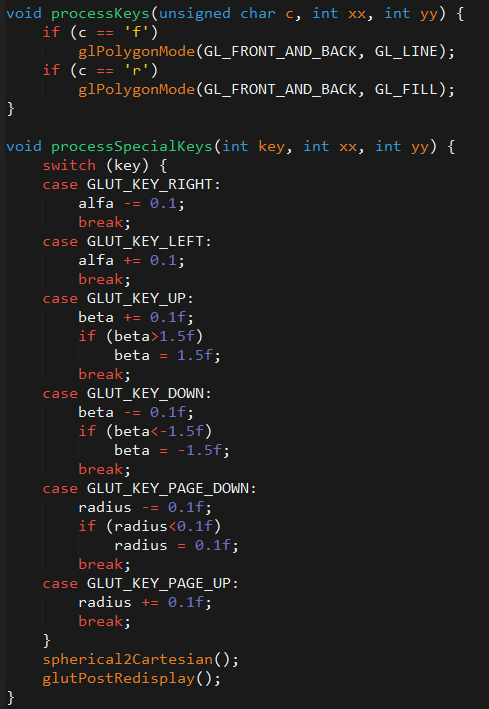
**Câmara**

A câmara está posicionada com recurso a coordenadas polares nas variáveis camX, camY e camZ, sempre com a direção do centro do eixo referencial (0,0,0). O vetor *up* coincide com um versor no eixo yy (0,1,0).



### Figura 24 Câmara

É possível alterar a posição da câmara com recurso às funções *glutSpecialFunc (processSpecialKeys)* e *glutKeyboardFunc (processNormalKeys)*.



### Figura 25 Introdução por teclado

As teclas *f e r* alteram o preenchimento dos triângulos entre *LINE* e *FILL*, sendo que as seguintes são as teclas que alteram a posição da câmara:

* ←, →: incremento e decremento,

respetivamente, em 0.1 unidades, do ângulo

𝑎𝑙𝑓𝑎.

* ↑, ↓: incremento e decremento,

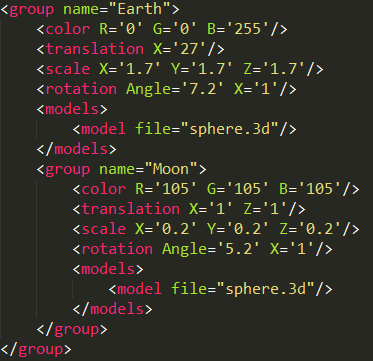
respetivamente, em 0.1 unidades, do ângulo

𝑏𝑒𝑡𝑎. Este é limitado superior e inferiormente pelo módulo de 1.5.

* *PG\_UP* e *PG\_DOWN*: decremento e incremento, respetivamente da distância da câmara à origem, que não pode ultrapassar, no limite inferior, 0.1 unidades.

# Fase 2: Transformações Geométricas

A proposta para a segunda etapa do trabalho prático era a inclusão de transformações geométricas na definição da cena, segundo um conjunto de transformações a aplicar a modelos dentro da mesma árvore.

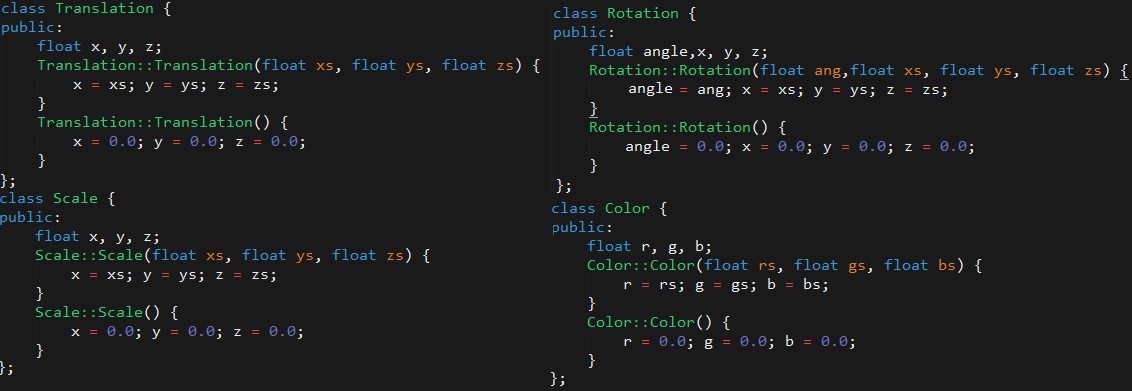


### Figura 26 Exemplo de Transformações

São adicionadas 4 novas classes ao ficheiro engine.h, referentes aos tipos de alterações que podem ser feitas aos modelos, representados pelas tags XML *<color>, <translation>, <rotation>* e

*<scale>*. Os parâmetros destas classes estão em conformidade com a assinatura das funções do *GLUT*

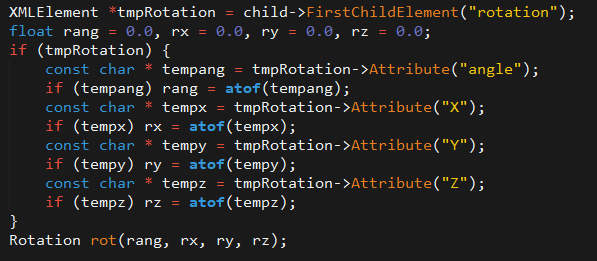
𝑔𝑙𝑇𝑟𝑎𝑛𝑠𝑙𝑎𝑡𝑒𝑓, 𝑔𝑙𝑅𝑜𝑡𝑎𝑡𝑒𝑓, 𝑔𝑙𝑆𝑐𝑎𝑙𝑒𝑓 e 𝑔𝑙𝐶𝑜𝑙𝑜𝑟3𝑓.



### Figura 27 Classes de Transformações

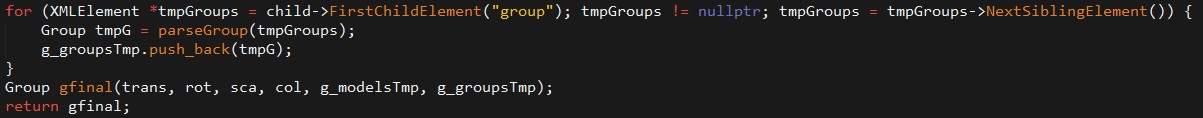
A função *xmlParser* foi alterada para passar a chamar uma função auxiliar *parseGroup* a cada tag

<group> no ramo imediato de <scene>. De seguida, itera-se pelas tags de cada classe e lê-se cada um dos seus parâmetros, armazenando-os em instâncias temporárias das respetivas. Na eventualidade de existência de *Groups* subsequentes, é corrida a função *parseGroup* recursivamente para esses grupos.

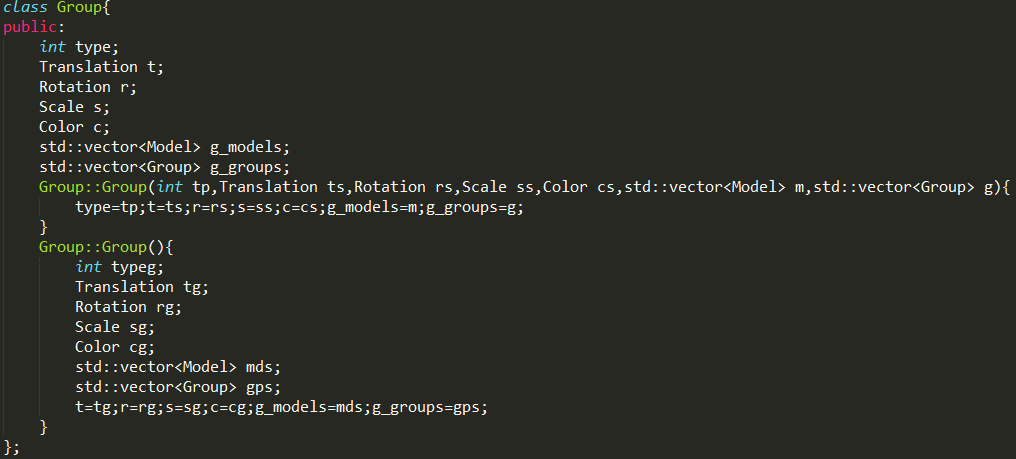


### Figura 28 Leitura de parâmetros de Rotação

O funcionamento do código em relação às escalas, translações e cores é idêntico ao código das rotações que é ilustrado na imagem de cima. A classe Group é definida segundo a Figura 29.

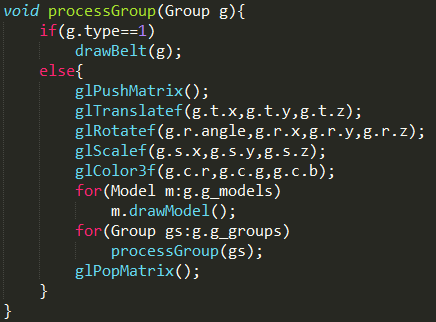


### Figura 29 Leitura de Groups subsequentes



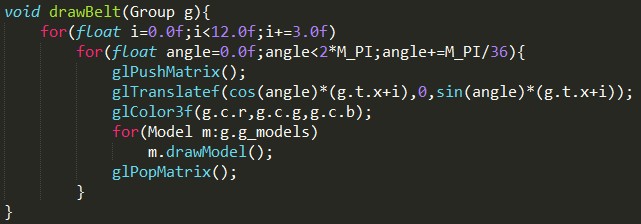
*Figura 30 Classe Group*

No final de cada iteração, a instância da classe Group gfinal é armazenada num vetor do tipo, junto com todos os conjuntos de figuras indicadas no ficheiro system.xml. Para o desenho da cena, percorre-se o vetor que contém todos os grupos, aplicando as transformações à matriz própria de cada Group. Todas as transformações de grupos subsequentes são aplicadas em relação à matriz do grupo “pai”. A classe Group também possui um int, que serve para distinguir entre os planetas, luas e o sol da cintura de asteróides, que é gerada automaticamente, que irá ser demonstrada mais à frente.



### Figura 31 Aplicação de Transformações

**Cintura de Asteróides**

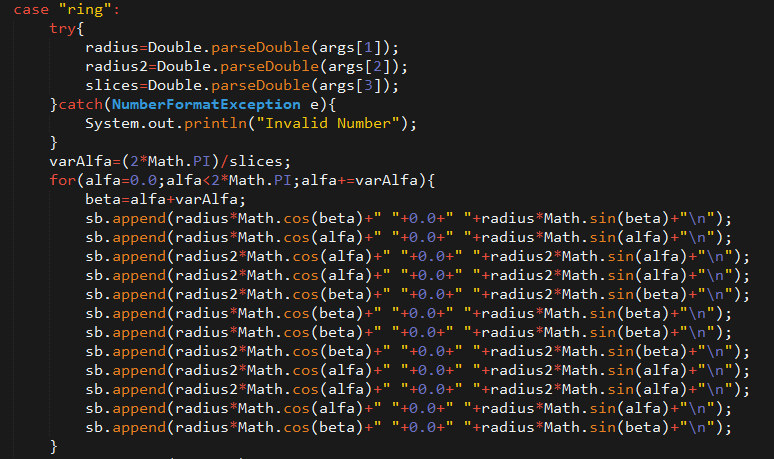


*Figure 32 Código que gera a cintura de asteróides*

A cintura é gerada de forma automática e simples, o ciclo de dentro serve para desenhar um círculo de asteróides, fasados em PI/36 rad. O ciclo de fora desenha mais 3 círculos de asteróides, separados por 3 unidades de comprimento. Cada asteróide é nada mais do que uma esfera muito irregular, criados também pelo generator.java. O cálculo das translações de cada um é feito com ajuda do angulo, seguindo a lógica trignométrica convencional. A cor é dada de igual modo aos restantes elementos do sistema solar.

**Gerador**

Nesta fase do trabalho prático foi incluída a possibilidade de gerar os pontos de um anel através do ficheiro generator.java.



### Figura 33 Código de geração de anel

Para gerar um anel, é necessário fornecer os valores dos raios inferior e superior, assim como o número de divisões do anel. As coordenadas dos pontos de cada triângulo são calculadas recorrendo às

𝑥 = 𝑟 ∗ cos(𝑎𝑙𝑓𝑎)

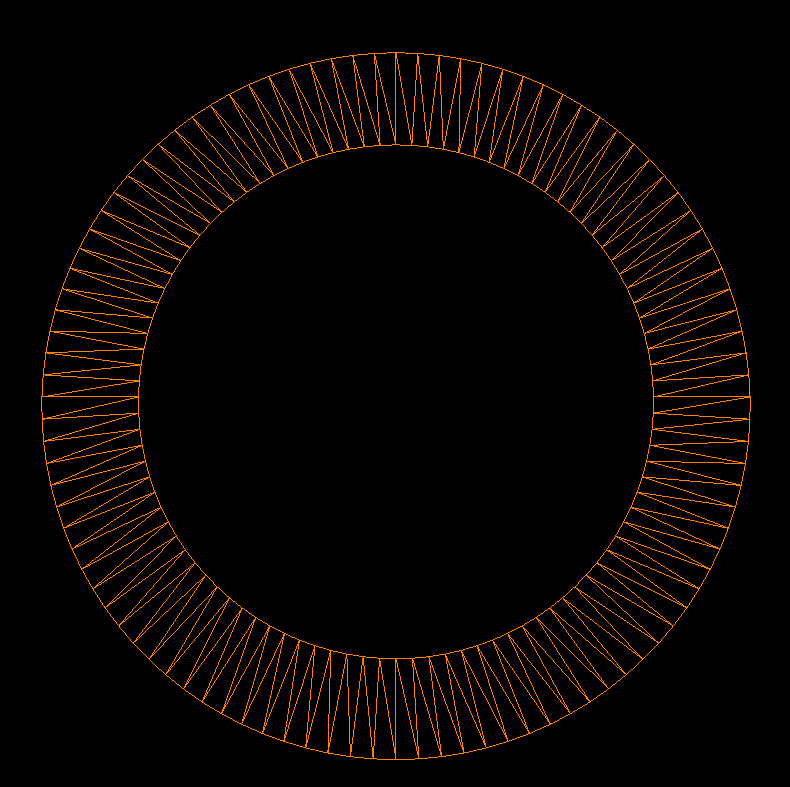
coordenadas polares (𝑎𝑙𝑓𝑎, 𝑟) = {

𝑦 = 0.0

𝑧 = 𝑟 ∗ sin(𝑎𝑙𝑓𝑎)

. O parâmetro de y=0.0 indica que os anéis serão

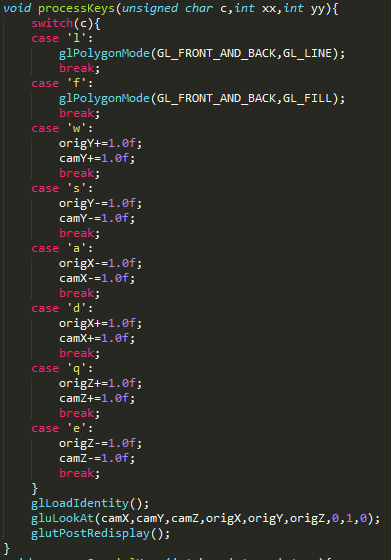
sempre desenhados concorrentemente com o plano xOz. A cada iteração em torno do ângulo alfa são calculados 4 triângulos, agrupando 2 para desenho com vista superior e o resto com vista inferior. O cálculo dos pontos segue um raciocínio semelhante ao do cone.



### Figura 34 Demo Scene Anel (GL\_FILL)

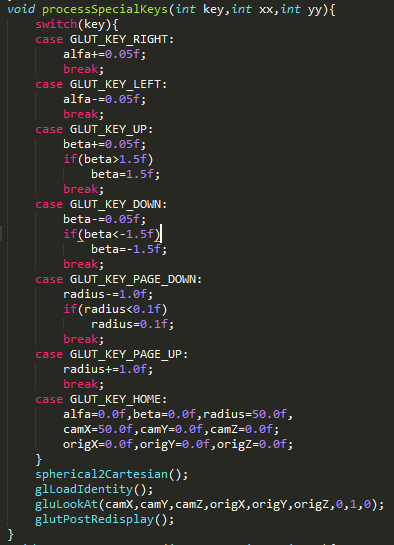
**Câmara**

A câmara nesta fase, continua a movimentar-se de maneira esférica, mas possui uma nova funcionalidade, que é poder alterar o centro de visão, ou seja, através de das teclas ‘a’, ‘d’, ‘w’, ‘s’, ‘q’ e ‘e’, alterar os valores de origX, origY e origZ que são os valores que representam o ponto para o qual está a olhar, e alteram também os valores de camX, camY e CamZ, que são as coordenadas do ponto onde está posicionada a câmera. Em nota, salienta-se que houve uma alteação das teclas que alternam o preenchimento dos triângulos, para ‘l’ de *LINE* e ‘f’ de *FILL*. No fim da função processKeys é chamada a função gluLookAt para reposicionar a camara na nova posição.



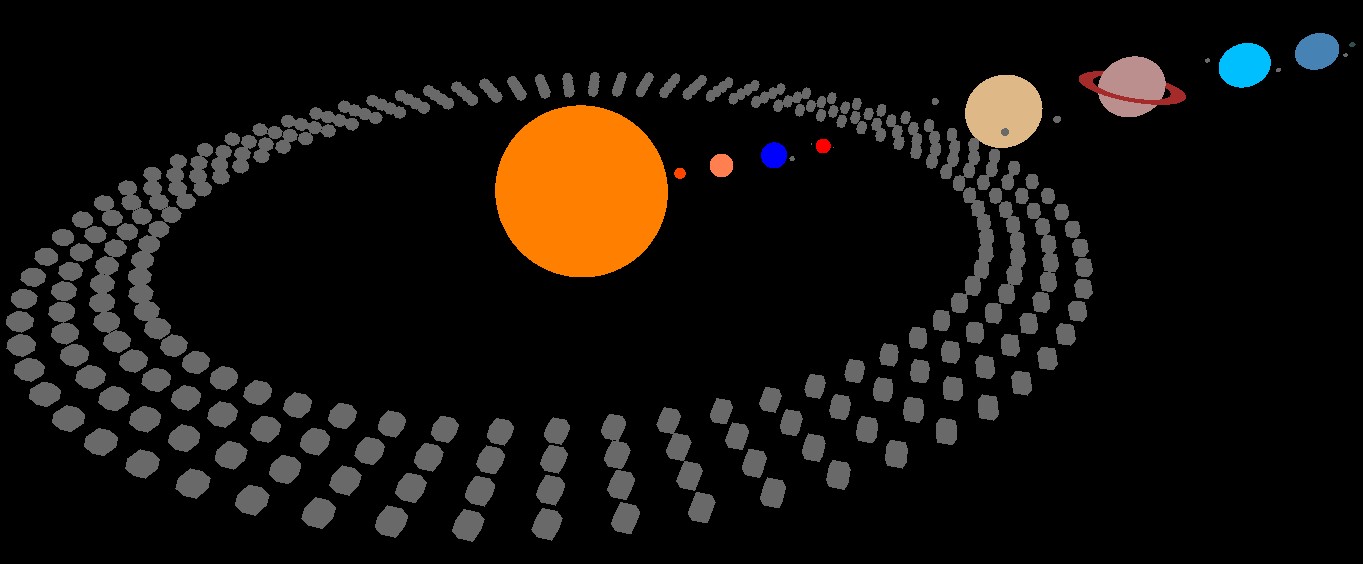
### Figura 35 Código da nova câmera

Também foi adicionado a tecla HOME como o intuito de posicionar a câmera na posição original, reiniciando os parâmetros que influenciam a câmera.

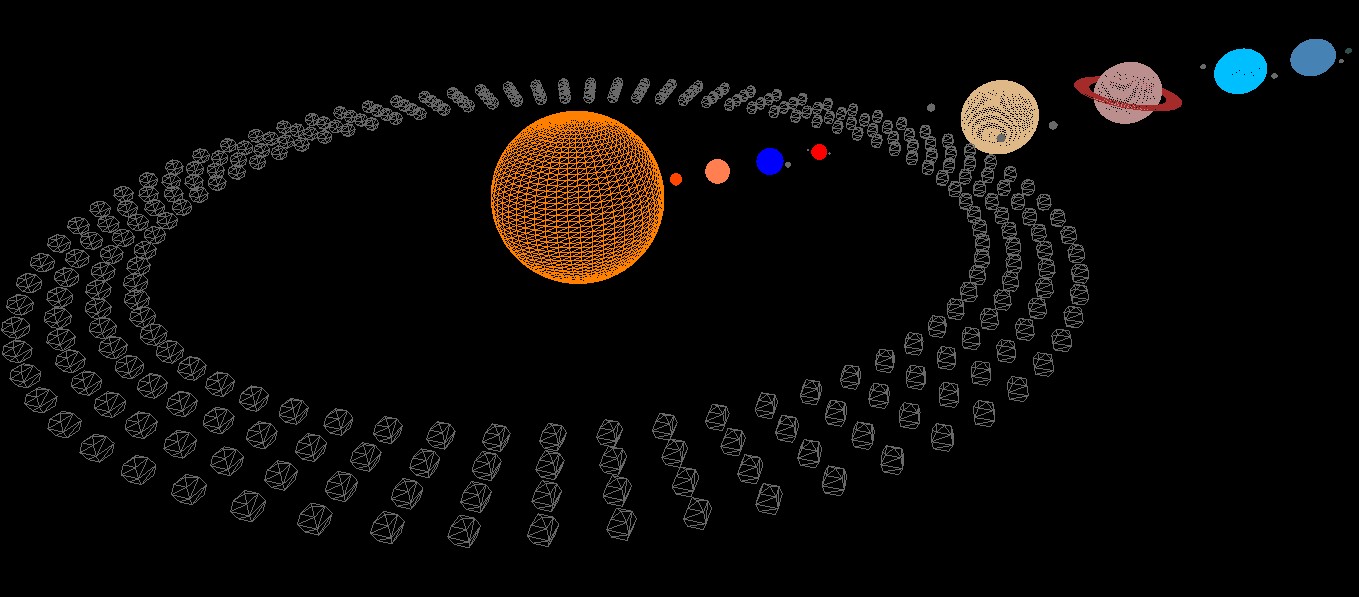


### Figura 36 Código da nova câmera

**Demo Scene do Sistema Solar**



*Figura 37 Demo Scene Fase 2 FILL*



*Figura 38 Demo Scene Fase 2 LINE*

# Fase 3: Curva, Superfícies Cúbicas e VBOs

**Cintura de Asteróides**

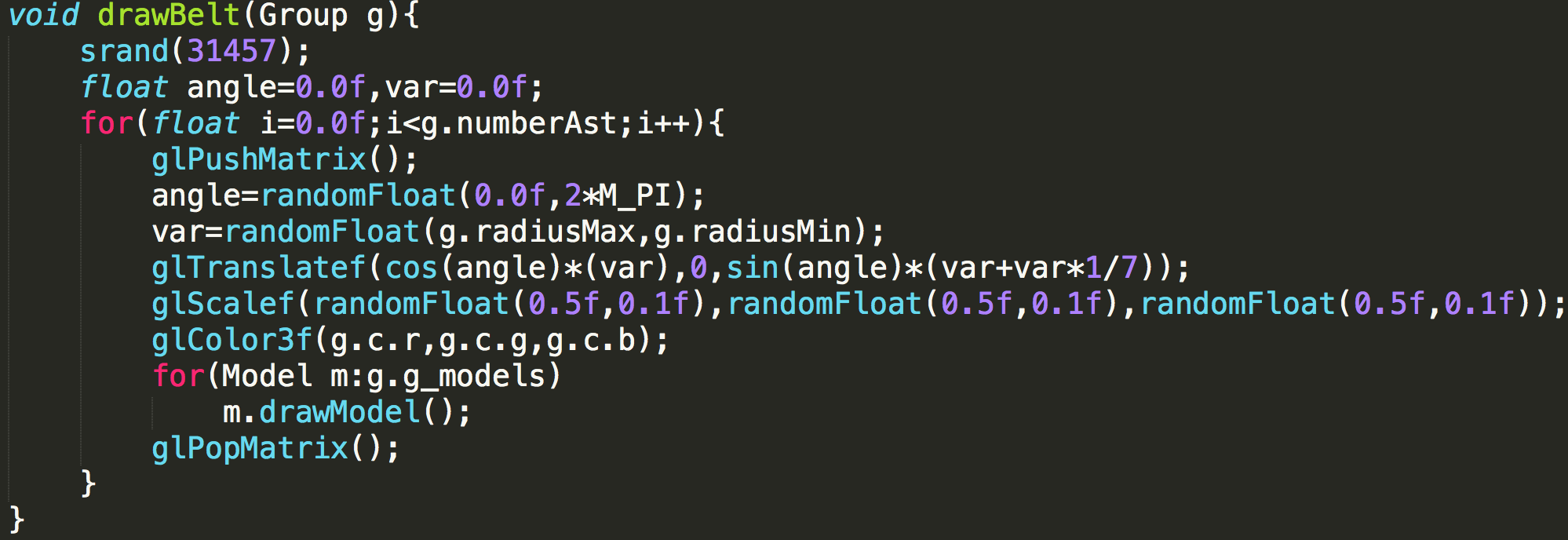


Figura 39 Cintura de Asteróides

Nesta 3ª fase, modificamos a cintura de asteróides previamente realizada. Com esta alteração, fazemos com que sejam gerados asteróides em diferentes posições, devido ao *angle* entre [0 ; 2π] e com diferentes tamanhos, dentro da gama de valores [*radiusMax ; radiusMin*] previamente definidos. Com o *glTranslatef*, *glScalef* e *glColor3f* definimos a posição, tamanho e cor do próximo asteroide a desenhar tendo em conta o *angle* e *var* anteriormente calculados.

**Órbitas**

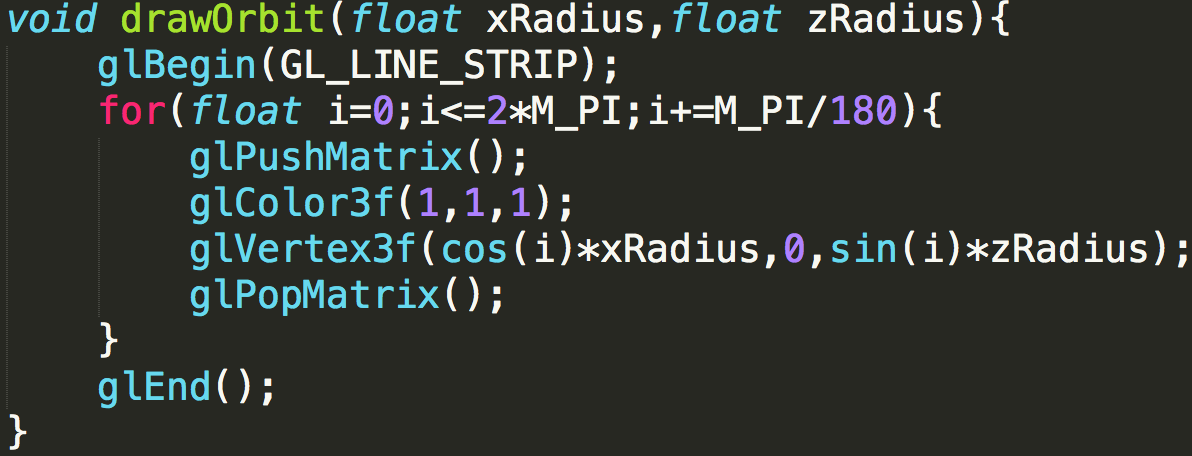


Figura 40 Órbitas

As órbitas dos planetas são desenhadas recorrendo ao *GL\_LINE\_STRIP*, fazendo com que sejam desenhados vários segmentos de reta entre o primeiro e último ponto, sendo que o seu centro fica no ponto (0,0,0). Em relação ao raio maior destas órbitas, o seu raio maior é no eixo dos XX e o menor no eixo dos ZZ.

Depois de iniciado o *GL\_LINE\_STRIP*, será através de um ciclo, que incrementa π/180 a cada iteração, que se vão desenhando os vários pontos necessários, entre [0 ; 2π], à construção dos vários segmentos de reta, que constituem cada órbita.

**Órbita do Cometa**

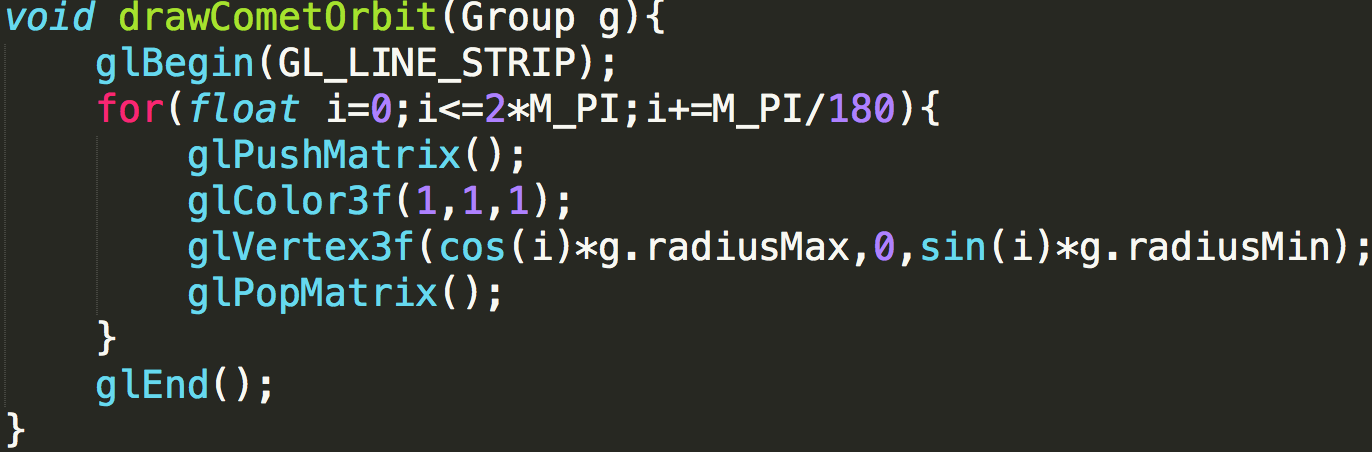
****

Figura 41 Órbita do Cometa

Em relação à orbita do cometa, que é um Bule de Chá, esta é desenhada da mesma forma que a anterior, no entanto, o seu centro irá ser no ponto (95,35,0). Em relação às orbitas previamente explicadas, estas diferem também pelo facto de que o seu raio maior é sobre o eixo dos ZZ e o menor sobre o eixo dos XX.

**Motor**

Tendo em conta as fases prévias, não foram adicionadas novas classes na presente fase, no entanto houve algumas alterações a ter em consideração em certas classes.

* Na classe *Translate*, foi adicionado a esta um parâmetro tempo, o qual corresponde ao tempo de uma translação completa. Além disso, foi também adicionado um conjunto de pontos representativos de uma curva Catmull-Rom, coincidentes com os pontos de controlo desta, sendo também guardado o tamanho desse conjunto;

Figura 42 Parsing de tags Translate

* A classe *Rotate* está idêntica à fase anterior, à exceção do parâmetro *angle* ,substituído agora pelo *time*, correspondendo este ao tempo de uma rotação completa;



Figura 43 Parsing de tags Rotate

* A classe *Model* foi reescrita para conter um *buffer* do tipo *GLuint* e um inteiro com o número de pontos de cada ficheiro 3d;
* Em relação ao *parseGroup*, foi alterada a maneira como são retirados os pontos dos ficheiros .3d, correspondendo a 1ª linha ao número total de pontos de cada ficheiro. São também extraídos pontos 3 a 3, dos ficheiros .3d, para um *array* dinâmico de *floats* previamente alocado. No final é mandado o *buffer* para a GPU, também associando ao modelo que está em leitura.



Figura 44 Parsing de tags Model

***GetTranslate***

A animação das translações é desenhada com recurso a curvas de Catmull-Rom. Os parâmetros para animação são o tempo e o vetor de pontos previamente calculados. Partindo da matriz de Catmull-Rom calcula-se em simultâneo os pontos res = T\*M\*P, que indica as coordenadas do próximo ponto da translação. No final é libertada a matriz que foi dinamicamente alocada com os pontos da curva e aplicada a translação.

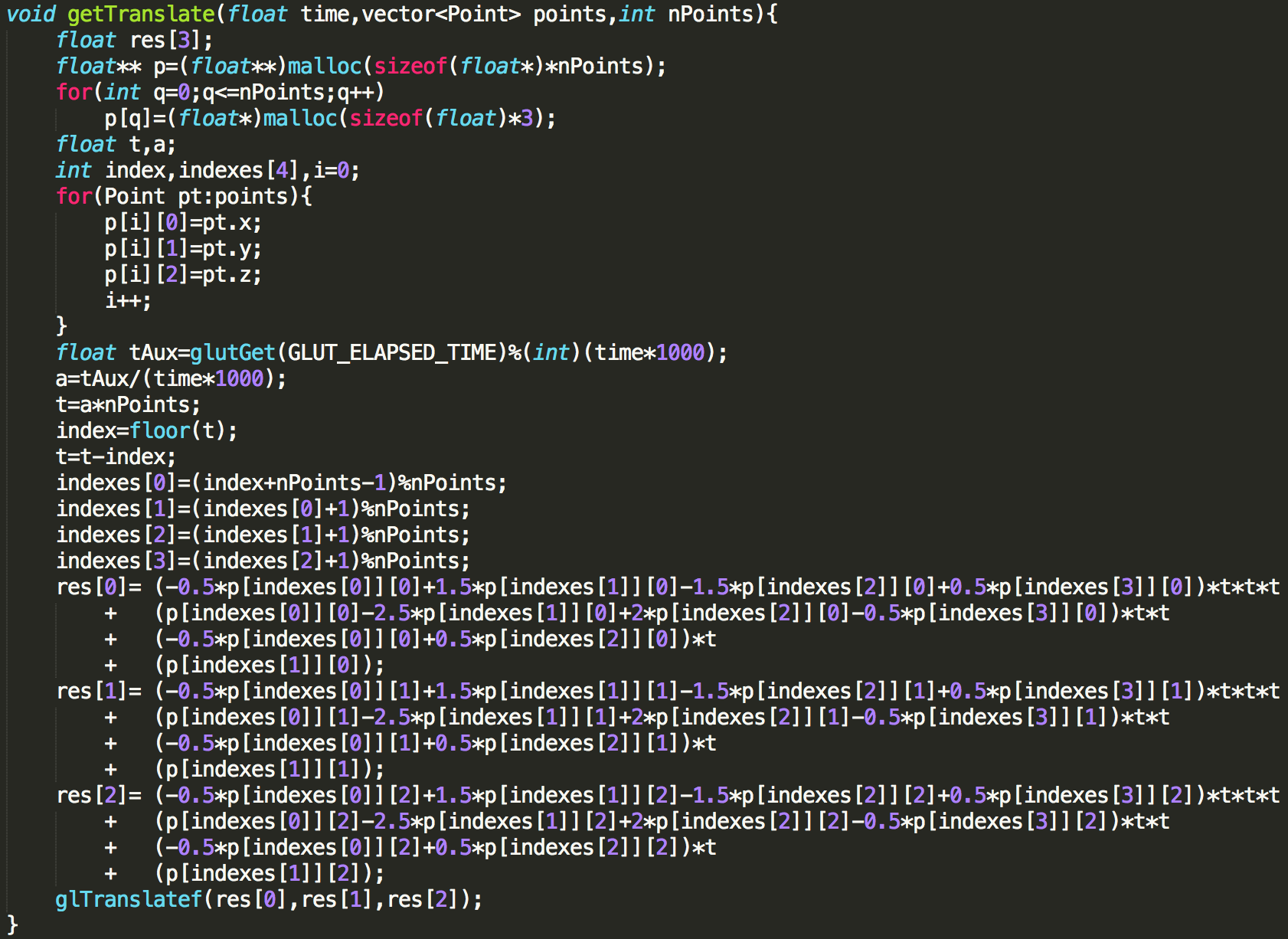


Figura 45 Translações Catmull-Rom

***GetRotate***

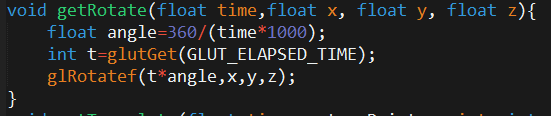
De modo a determinar as próximas posições para efetuar a rotação, divide-se o ângulo giro pelo fator de tempo, dado como parâmetro no ficheiro XML, multiplicado por 1000 para que este seja calculado em segundos. Ao aplicar a rotação, o efeito de transição é suavizado pois o incremento é constante.

Figura 46 Rotações

**Demo Scene do Sistema Solar**

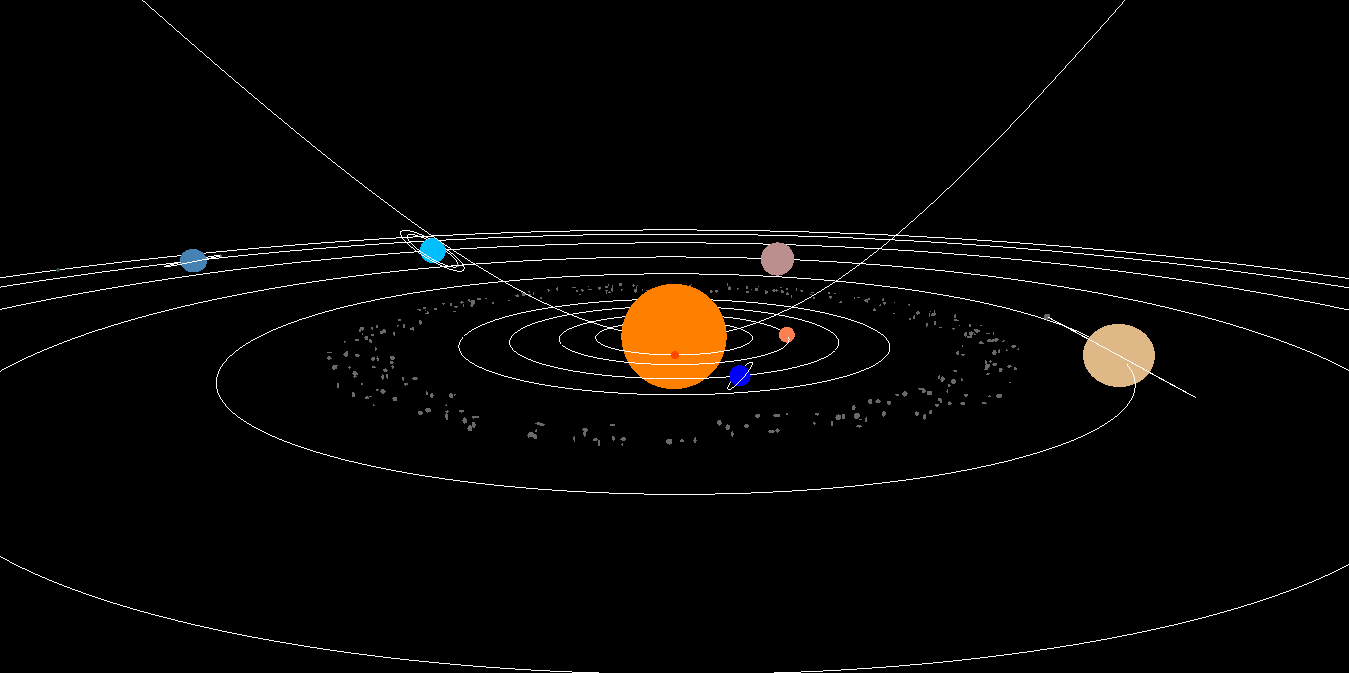


Figura 47 Demo Scene Sistema Solar