Universidade do Minho

Escola de Ciências

Licenciatura em Ciências da Computação

Ano Letivo de 2018/2019

Computação Gráfica

Trabalho Prático

1ª-2ª Fase

Grupo 2

Arlindo da Fonte Torres - A80298

Nelson José Dias Teixeira – A80584

João Ribeiro Imperadeiro – A80908

Índice

Introdução

O objetivo deste trabalho prático elaborado no âmbito da UC de Computação Gráfica consiste no desenvolvimento de um motor 3D baseado em pequenas cenas compostas por gráficos e fornecer exemplos da sua utilização que evidenciem o seu uso. Este trabalho prático está dividido em 4 fases, sendo que cada uma delas é submetida na plataforma *BlackBoard*.

A **1º fase** é composta por duas aplicações: uma para gerar os ficheiros com as informações dos modelos 3D (nesta fase apenas com a informação dos vértices); outra serve como motor gráfico que lê um ficheiro de configuração em *XML* e apresenta os modelos 3D solicitados.

Nesta fase pretende-se ter a possibilidade de representar as seguintes primitivas gráficas:

- Plano
- Caixa
- Esfera
- Cone

Para cada um destes modelos, podemos passar parâmetros para alterar as características de cada um, tendo em conta o que foi proposto no enunciado.

Para atingir os objetivos propostos, teremos de definir a implementação de cada uma das primitivas gráficas, definir o formato dos ficheiros com a informação de cada uma das figuras, processar os ficheiros XML com a informação relativa a uma cena, estudar as API's do *GLUT*, decidir a interação que será possível ter com a imagem final e aprender a utilizar o *CMake*.

Quanto à **2ª fase**, é pedida a criação de cenas hierárquicas, onde se aplicam transformações geométricas, tais como a translação, a rotação e a escala. Para tal, cada cena é definida como uma árvore, onde cada nodo é composto por um conjunto de transformações geométricas e, opcionalmente, por um conjunto de modelos.

De notar que existe a possibilidade de cada nodo ter nodos-filhos. Nessas situações, este irá herdar todas as transformações geométricas do nodo original (nodo-pai).

Tal como já foi abordado nas aulas, a ordem pela qual são implementadas as transformações geométricas é relevante, pelo que são aplicadas a todos os modelos e subgrupos respetivos pela ordem de aparição. Estas só podem existir dentro de um bloco *<group> … </group>* ou como caraterísticas da cena principal.

Por forma a demonstrar o funcionamento do resultado esperado nesta fase, é proposta a exibição de um modelo estático do sistema solar (incluindo o sol, os planetas intervenientes e a lua).

Trabalho Prático - 1ª Fase

Estrutura - 1ª Fase

Como sugerido no enunciado, decidimos dividir o nosso trabalho em dois programas complementares: o *generator* e o *engine*. Por um lado, o *generator* é responsável pela geração dos ficheiros que contêm os vértices de cada uma das figuras pedidas no enunciado e que serão geradas pelo utilizador. Por outro, temos o *engine*, que interpreta um ficheiro no formato *XML*, extrai as referências aos ficheiros gerados pelo *generator* e apresenta as respetivas formas geométricas, tendo em conta os vértices dados.

Durante o desenvolvimento dos dois, tentamos manter os comentários associados a cada função o mais atualizados possível, de forma a facilitar a leitura do código desenvolvido anteriormente e a possibilitar a compreensão do código por parte de pessoas externas ao grupo (como é o caso dos docentes que irão avaliar o presente trabalho).

Nos dois subcapítulos seguintes, iremos entrar em mais detalhes sobre o papel de cada um destes subprogramas do nosso trabalho.

- ***Generator***

Quanto ao *generator*, este é uma aplicação escrita em C++, que pode ser invocada a partir do terminal da seguinte forma:

./generator figura param1 [...paramN] nome.3d

Aqui, "figura" é o nome da figura geométrica pretendida, de parâmetros "param1 [...paramN]", cujos vértices serão enviados para o ficheiro "nome.3d" (adotamos a extensão ".3d" por convenção do enunciado, mas tenha-se em conta que o seu uso não é obrigatório).

Como pedido no enunciado, as figuras aceites são: “plane”, “box”, “sphere” e “cone”. Estas serão divididas num certo número de triângulos (que, como é evidente, serão compostos por 3 vértices cada um) dependendo dos parâmetro escolhidos pelo utilizador. A implementação de cada uma delas e os respetivos parâmetros serão explicados no capítulo seguinte, que abordará cada uma das primitivas gráficas.

Quanto ao formato que adotamos para a impressão dos vértices no ficheiro fornecido (“nome.3d” no caso do exemplo acima), decidimos imprimir as coordenadas de cada um deles numa linha, separadas por espaços, como se pode ver a seguir:

x.x y.y z.z\n

Cada uma das coordenadas pode ser um valor real, apesar de os parâmetros que são passados ao *generator* serem inteiros. Isto poderia ser alterado com relativa facilidade, mas como no enunciado é sugerido o uso de inteiros optamos por fazê-lo no nosso trabalho.

Aquando da invocação do *generator*, temos o cuidado de validar que a figura pedida é uma das implementadas e que o número de parâmetros está correto, dada essa figura.

- ***Engine***

Passando ao *engine*, este é também uma aplicação desenvolvida em C++, que pode ser invocada a partir do terminal da seguinte forma:

./engine ficheiro.xml

Nesta invocação, “ficheiro.xml” é um ficheiro de configuração que faz referência a um ou mais ficheiros produzidos pelo *generator* e tem a seguinte estrutura:

<scene>

<model file="figura1.3d" />

...

<model file="figuraN.3d" />

</scene>

Aquando da invocação do *engine*, procedemos à validação do ficheiro passado como parâmetro, certificando-nos de que ele foi mesmo passado e, em caso afirmativo, passamos à sua leitura/interpretação.

Para nos auxiliar no processamento do ficheiro de entrada, e tendo em conta que é um ficheiro no formato *XML*, seguimos a recomendação do enunciado e recorremos à biblioteca *TinyXML-2*.

Seguindo a documentação desta biblioteca, começamos por criar um objeto *XMLDocument*, invocando sobre ele o método *LoadFile* e passando como parâmetro o nome do ficheiro *XML* recebido na linha de comandos. De seguida, utilizando a abstração *XMLElement*, extraímos o elemento *scene* e, caso este exista, percorremos os elementos *model* que são descendentes deste. Para cada um deles, pegamos no atributo *file*, que contém a informação relativa ao nome do ficheiro onde estão os vértices, e carregamos todos os seus vértices para uma estrutura global (neste caso, um vetor de vértices, que são triplos de números reais).

Processado o ficheiro *XML* e tendo todos os vértices na nossa estrutura global, procedemos à invocação das primitivas do *GLUT*, as quais nos permitem criar uma janela e desenhar as figuras pedidas pelo utilizador. Para não tornar o relatório muito verboso, os parâmetros passados ao *GLUT* podem ser consultados no ficheiro "main.cpp" da pasta "*engine*". Tenha-se em atenção que a sua utilização está intimamente ligada ao uso que lhes demos nas aulas.

Passando agora ao desenho das figuras em si, estas são desenhadas a partir de triângulos, como referido anteriormente, sendo que três vértices consecutivos no ficheiro formam um triângulo. Quanto à cor de cada um dos triângulos, para facilitar a sua visualização, tomamos a decisão de não os pintar de uma forma monocromática. Sendo assim, fomos ao extremo oposto, pintando cada um dos vértices de uma cor aleatória, surgindo em cada um dos triângulos um gradiente que nos agradou.

Tendo em conta a experiência das aulas práticas, decidimos permitir a interação do utilizador com os vértices, possibilitando o redimensionamento da janela, a movimentação da figura através das teclas “WASD”, a aproximação ou afastamento da câmara através das teclas “Q” e “E”, a movimentação da câmara através das setas direcionais do teclado e a alteração do modo de desenho dos triângulos, usando “P” para colorir apenas os vértices, “L” para colorir apenas as arestas e “F” para colori-lo na íntegra.

Uma característica particular do nosso *engine* é que quando o utilizador interage com a figura, utilizando as funcionalidades abordadas acima, cada um dos triângulos muda de cor. Esta particularidade foi algo que nos pareceu criar um efeito interessante, por isso decidimos mantê-la no nosso trabalho.

Primitivas Gráficas

Tal como foi dito anteriormente, iremos agora abordar a implementação das figuras em si. As que foram solicitadas neste trabalho são as seguintes:

➔ plano

➔ caixa

➔ cone

➔ esfera

De notar que em todas as figuras teve-se em conta a regra da mão direita aquando do desenho de todos os vértices dos triângulos das mesmas, de forma a que os triângulos fossem corretamente desenhados.

- **Plano**

  void printPlane(int dist)

De maneira a criar o plano pedido num espaço 3D com um referencial associado (x,y,z), foi necessário passar como argumento à função que iria imprimir os vértices da figura a distância máxima do plano ao longo do eixo x e do eixo z. Assim, desenharam-se os 4 triângulos que iriam formar o plano, isto é, dois quadrados, um para cada uma das "faces" do plano. De salientar que era solicitado que este estivesse centrado na origem do referencial.

Tendo isto em conta, partiu-se para a implementação do plano, isto é, o seu desenho. Foram impressos 12 vértices no referencial (x,y,z), 3 para cada triângulo do futuro plano. Por forma a garantir que o plano se encontrasse centrado na origem do referencial, uma das componentes (x ou z) de cada ponto teria de ser a metade da distância máxima passada

como argumento à função. Respeitando a regra da mão direita, foram desenhados os pontos A, C e D para formar o primeiro triângulo e os pontos D, B e A para formar o segundo triângulo. O mesmo foi feito para a face oposta. Apresenta-se de seguida uma imagem ilustrativa do plano.

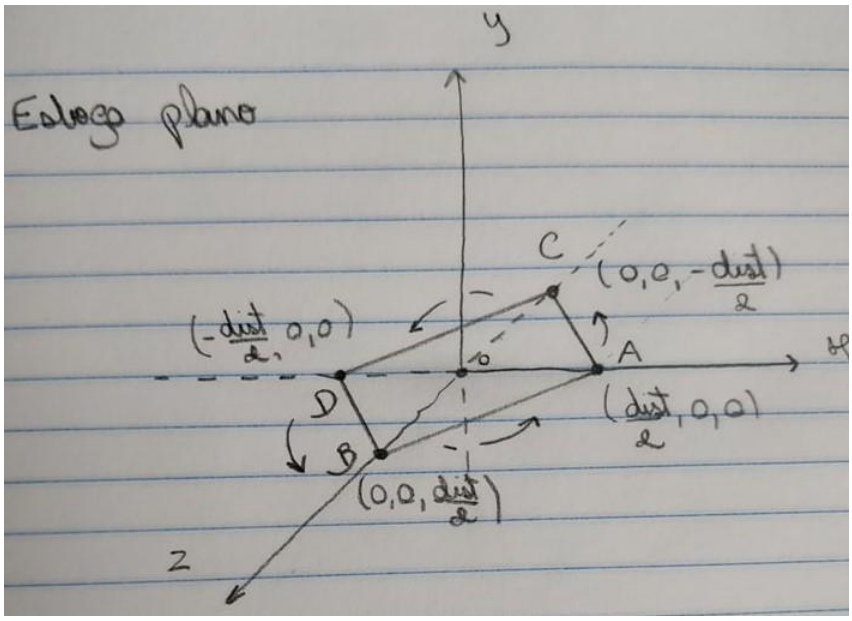
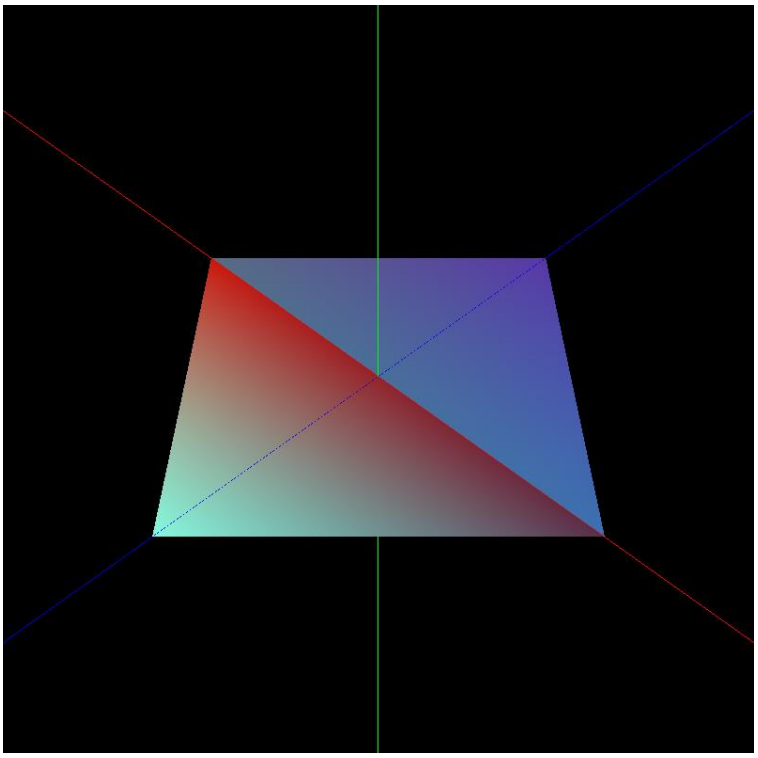


Figura *OpenGL* (Exemplo ao invocar printPlane(6))



- **Caixa**

  void printBox(int dimX, int dimY, int dimZ, int divisions)

De maneira a criar a caixa solicitada num espaço 3D com um referencial associado (x,y,z), são passados como argumentos as suas dimensões máximas ao longo dos 3 eixos. Para além destes, existe um outro parâmetro que diz respeito ao nº de divisões da caixa. Este parâmetro é opcional, está pré-definido para ser 1 e está associado ao número de quadrados em que cada face está dividida. Ou seja, se *divisions* for igual a N, significa que cada face estará subdividida em N*N quadrados mais pequenos, onde cada um deles é composto por 2 triângulos. Optou-se, adicionalmente, por colocar o centro da caixa na origem do referencial, tal como no caso anterior.

A partir das dimensões máximas mencionadas anteriormente, procedeu-se à definição dos vértices da figura. O raciocínio que está por trás da criação de cada face da caixa é semelhante ao do plano, sendo que uma diferença é o facto de que cada face pode ser retangular, para além de quadrangular.

Para além disto, temos a particularidade de poderem existir divisões, o que nos fez repensar a nossa implementação. Para atingir este objetivo, decidimos desenhar um subquadrado de cada face de cada vez. Ou seja, temos dois ciclos, um dentro do outro, em que uma iteração do ciclo exterior representa uma linha de quadrados de cada uma das faces da figura final e uma iteração do *loop* interior representa um único quadrado de cada uma das faces (concretizado nos triângulos que dele fazem parte).

Figura *OpenGL* (Exemplo ao invocar printBox(4,4,4,3) - cubo)

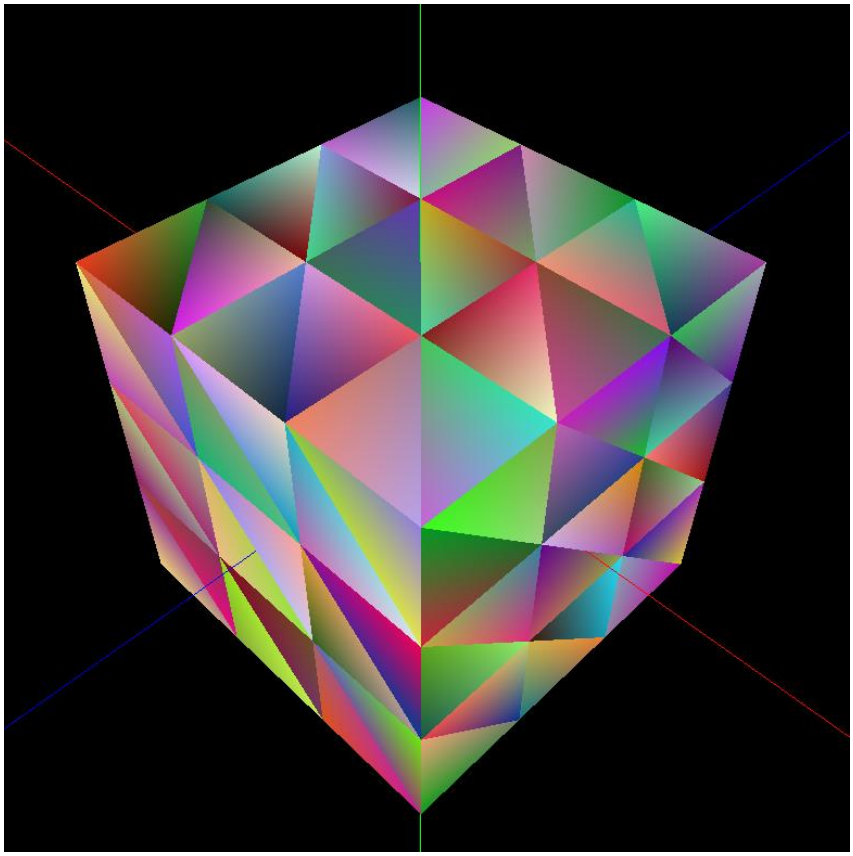
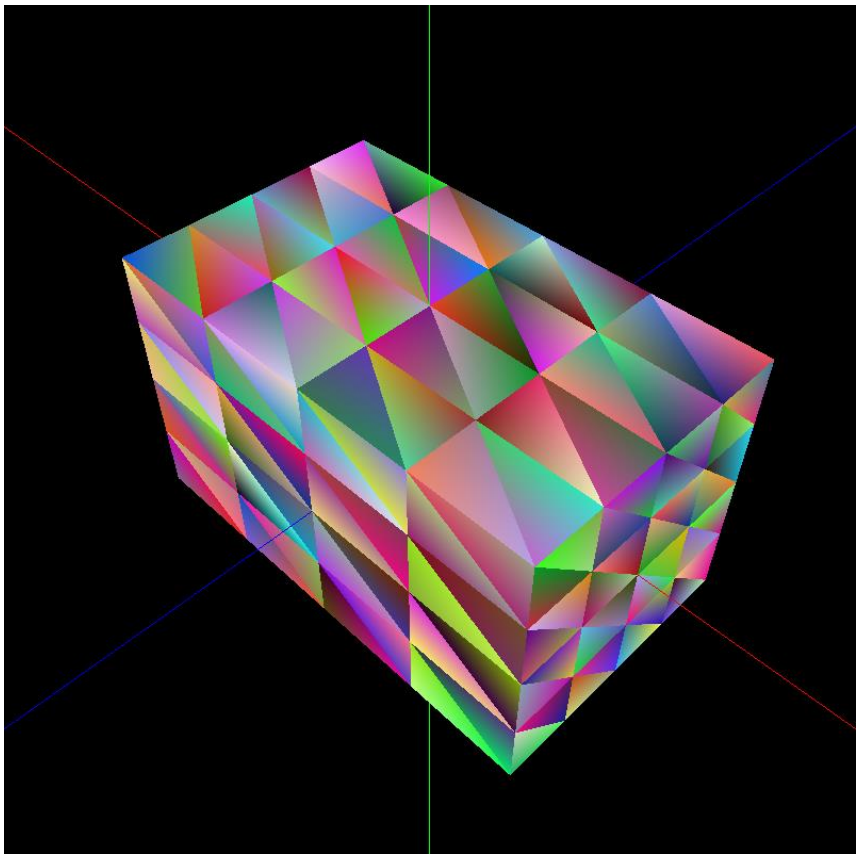
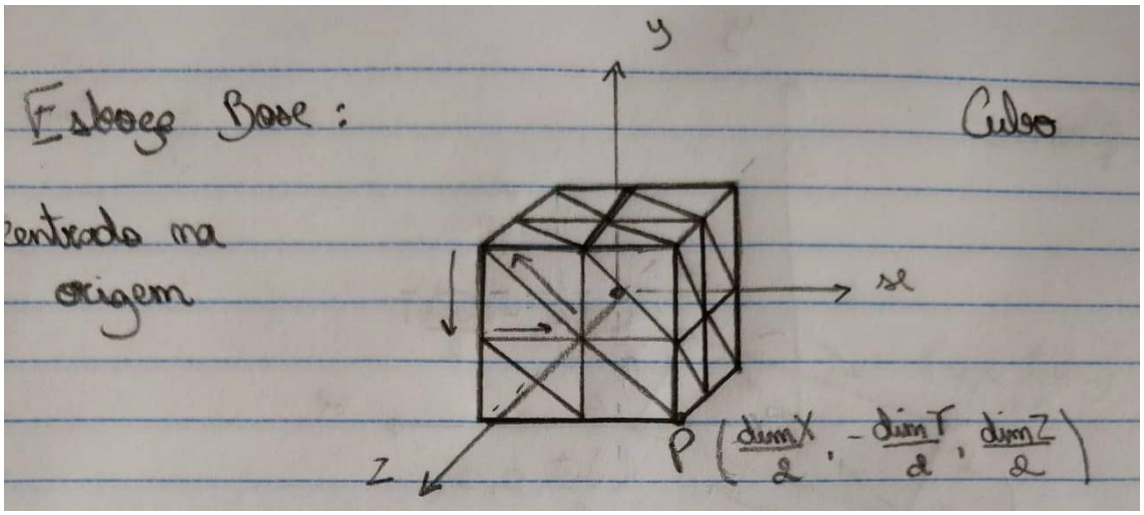


Figura *OpenGL* (Exemplo ao invocar printBox(5,3,3,4) - caixa retangular)




Esboço Base:
Cubo
Centrado na origem
y
x
z
P $\left(\frac{dim X}{2}, -\frac{dim Y}{2}, \frac{dim Z}{2}\right)$

- **Cone**

  void printCone(int bottomRadius, int height, int slices, int stacks)

De forma a desenharmos um cone composto por triângulos, com a ponta virada para a parte positiva do eixo do y, e que esteja dividido em *slices* (número de subdivisões em volta do eixo y) e *stacks* (número de subdivisões ao longo do eixo y, sendo que o seu raio vai decrescendo à medida que o valor de y aumenta), começamos por definir uma variável ***sliceDelta = (2 * π) / slices,*** que guarda o valor do ângulo de cada *slice*. Para além desta, é também definida uma variável ***r*** que armazena o valor inicial de *bottomRadius,* isto é, o raio da base do cone.

De seguida é construído o círculo da base do cone, com a face visível para baixo, e desenhando tantos triângulos quanto o número de slices.

De seguida, procedemos ao desenho de cada uma das *stacks*. Com esse fim, é definida numa nova variável ***rnext***, que representa o raio da *stack* seguinte, e itera-se, desenhando cada *stack* do cone (uma de cada vez). Para cada *stack*, desenham-se triângulos a apontar para cima e para baixo. À medida que se vai construindo a figura (de baixo para cima), vão-se atualizando os valores da altura da *stack* atual e da seguinte (***h*** e ***hnext***), e do raio da *stack* atual e da seguinte (***r*** e ***rnext***).

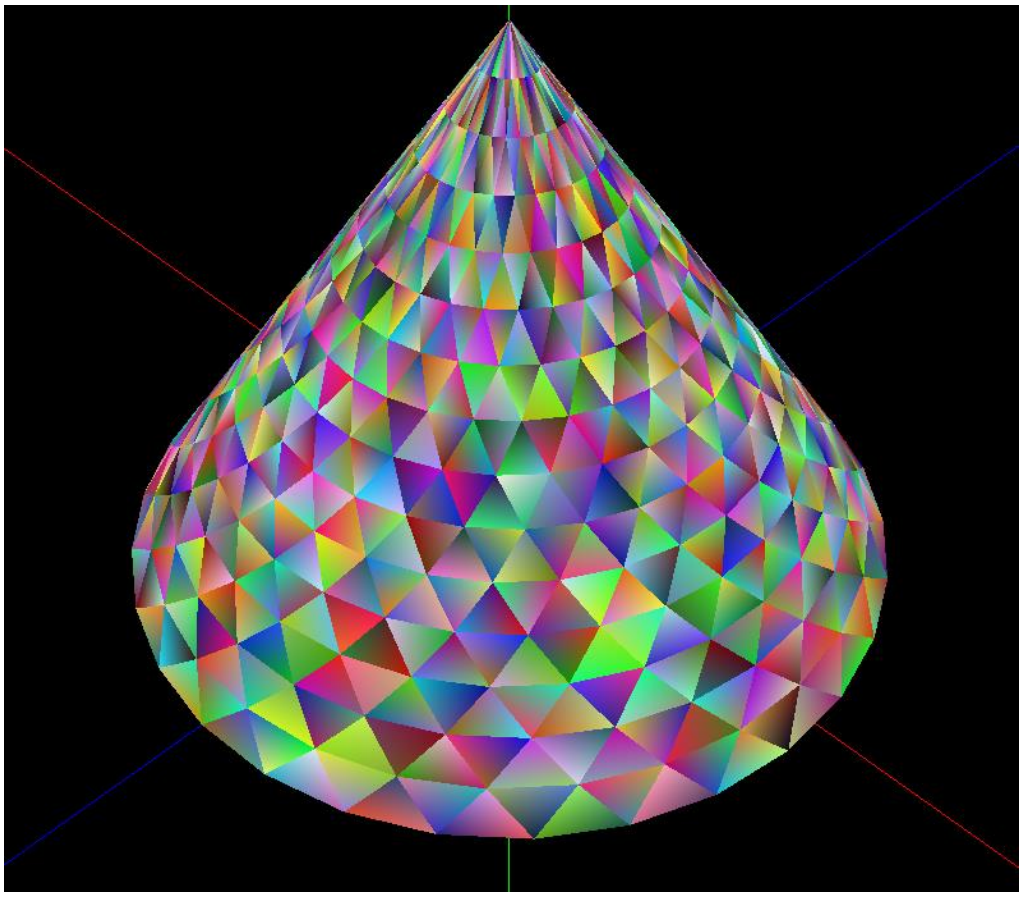
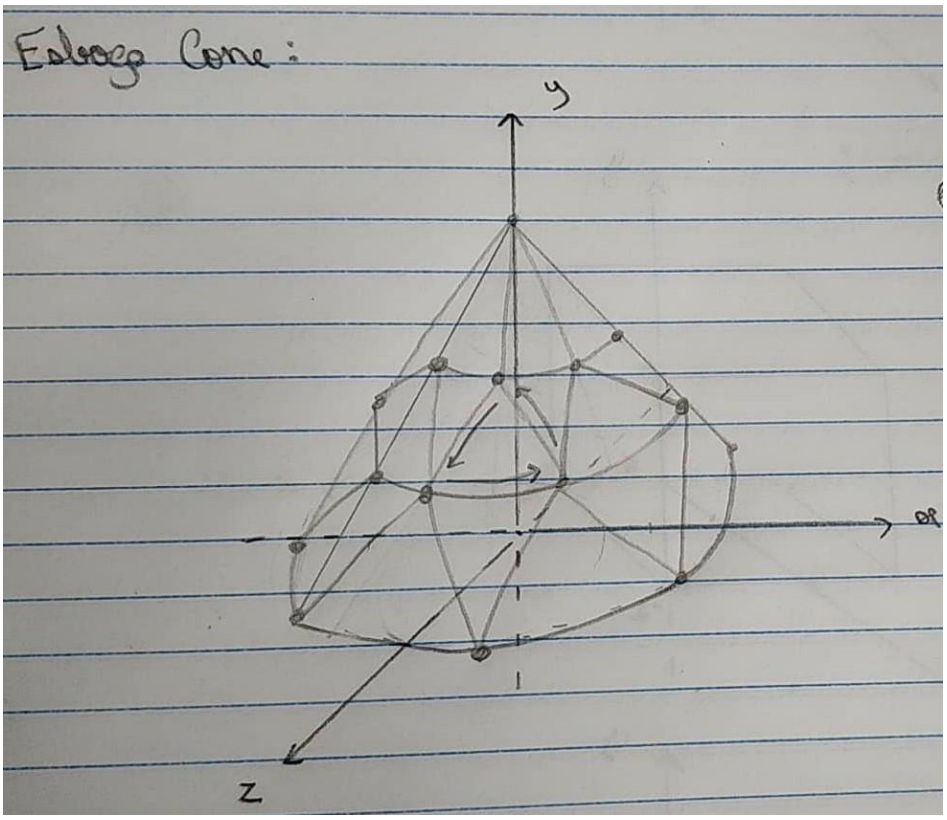


Figura *OpenGL* (Exemplo ao invocar printCone(3, 4, 30, 15))


Esboço Cone:
y
z

- **Esfera**

  void printSphere(int radius, int slices, int stacks)

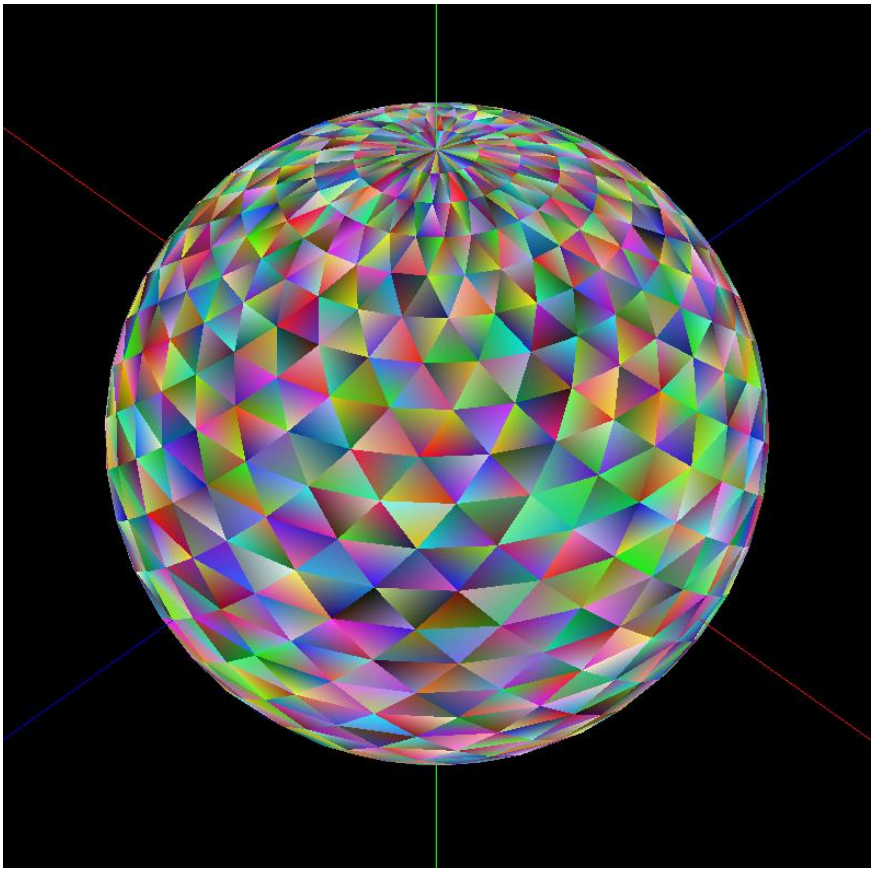
De todas as figuras pedidas no enunciado, esta foi, para nós, a mais desafiante e, consequentemente, a mais interessante de implementar.

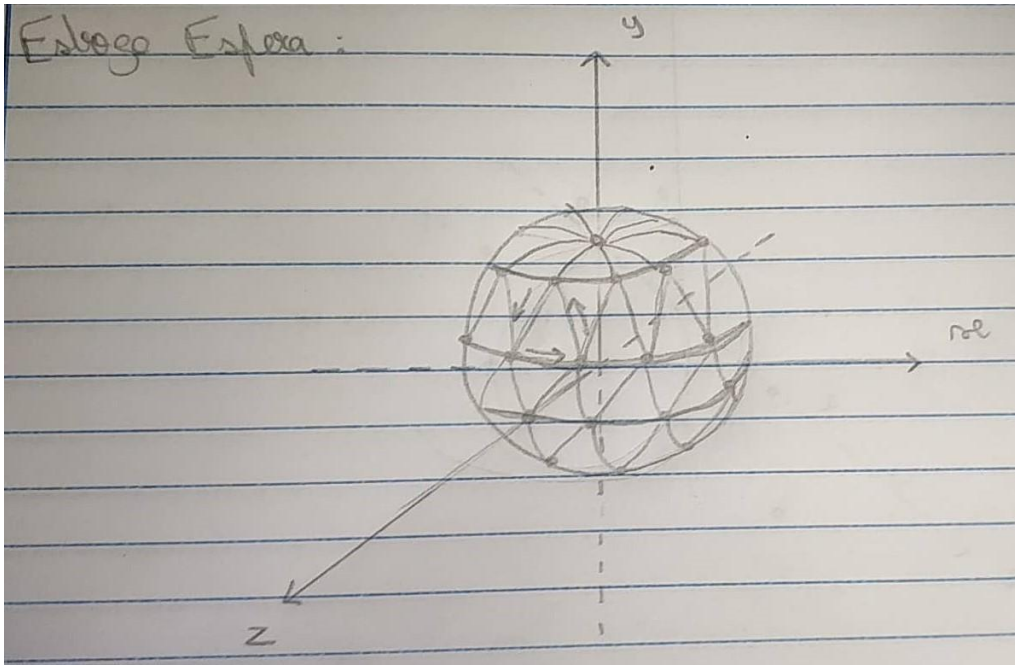
De forma a conseguirmos gerar os vértices necessários para o nosso *engine* conseguir desenhar a esfera, tomamos uma abordagem similar à do cone, em que desenhamos a esfera por *stacks*, tendo em conta que desta vez o fazemos de uma forma espelhada (isto é, desenhamos *stacks* a afastarem-se do equador). Para isso, decidimos centrar a esfera na origem e começámos por definir várias variáveis na função acima referida, sendo que entre elas se destacam:

1. ***sliceDelta = (2 * π) / slices*** (ângulo de cada *slice*);
2. ***betaDelta = (2 * π) / (stacks / 2)*** (ângulo entre duas *stacks*);
3. ***r*** (raio da *stack* atual);
4. ***nextR*** (raio da próxima *stack*);
5. ***height*** (altura da *stack* atual);
6. ***nextHeigth*** (altura da próxima *stack*).

Definidas as variáveis, passamos ao desenho de cada uma das *stacks*. Para isso, itera-se, desenhando duas *stacks* da esfera de cada vez (uma de cada cúpula da mesma). Para cada *stack*, desenham-se triângulos a apontar para cima e para baixo. À medida que se vai construindo a figura (do equador para os polos), vão-se atualizando os valores da altura da *stack* atual e da seguinte (***height*** e ***nextHeigth***), e do raio da *stack* atual e da seguinte (***r*** e ***nextR***).

Figura *OpenGL* (Exemplo ao invocar printSphere (3, 30, 30))





Composição de primitivas num mesmo referencial

Às funcionalidades implementadas na 1ª fase deste trabalho prático, acrescenta-se a possibilidade, tal como o nome deste capítulo sugere, de desenhar várias figuras num mesmo referencial. De seguida apresentam-se alguns exemplos que evidenciam a composição de primitivas no mesmo espaço:

Figura *OpenGL* (printBox(3,3,3,5) (cubo) + printSphere(2,30,30))

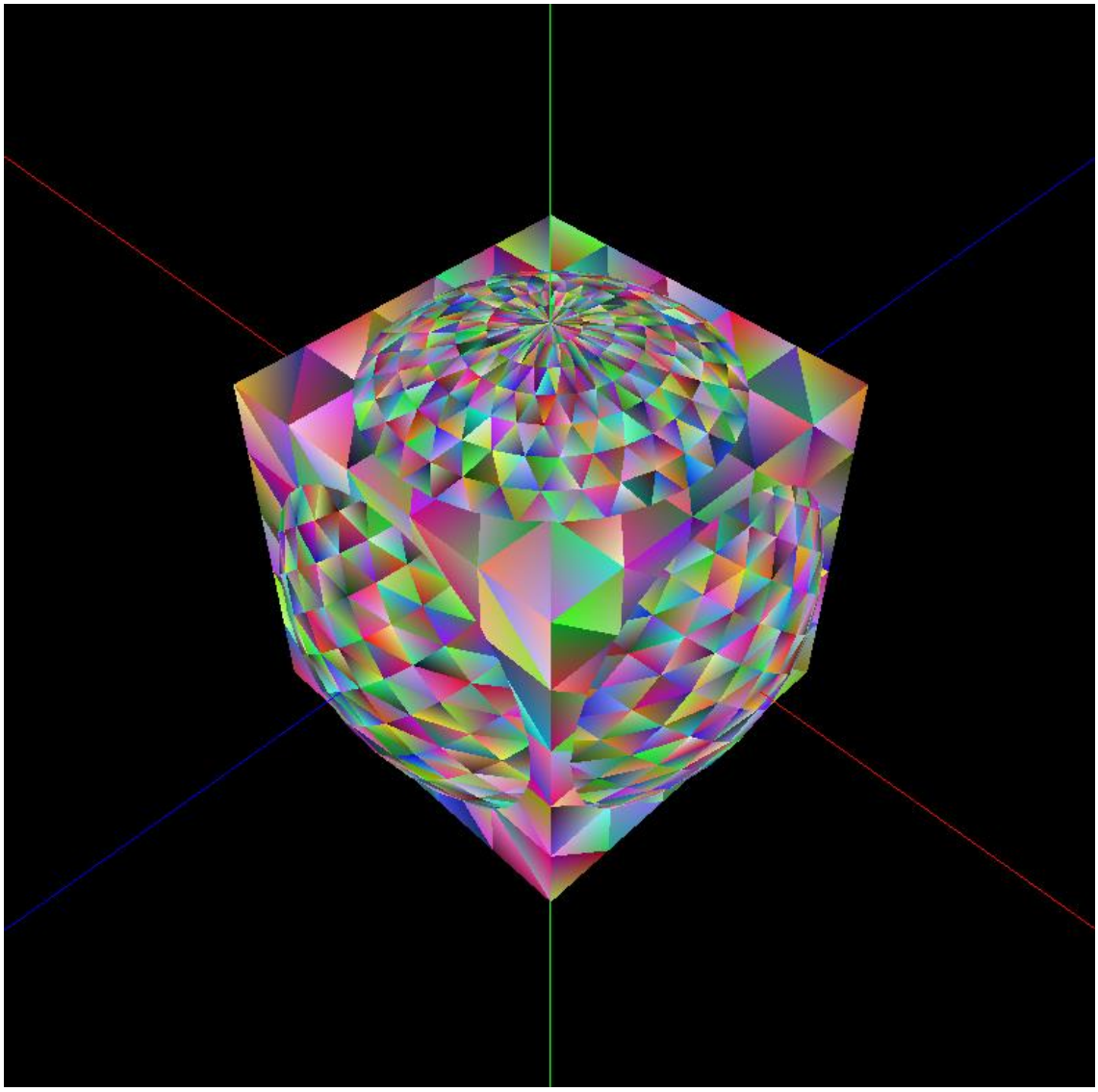


Figura *OpenGL* (printPlane(6) + printCone(2, 4, 30, 20))

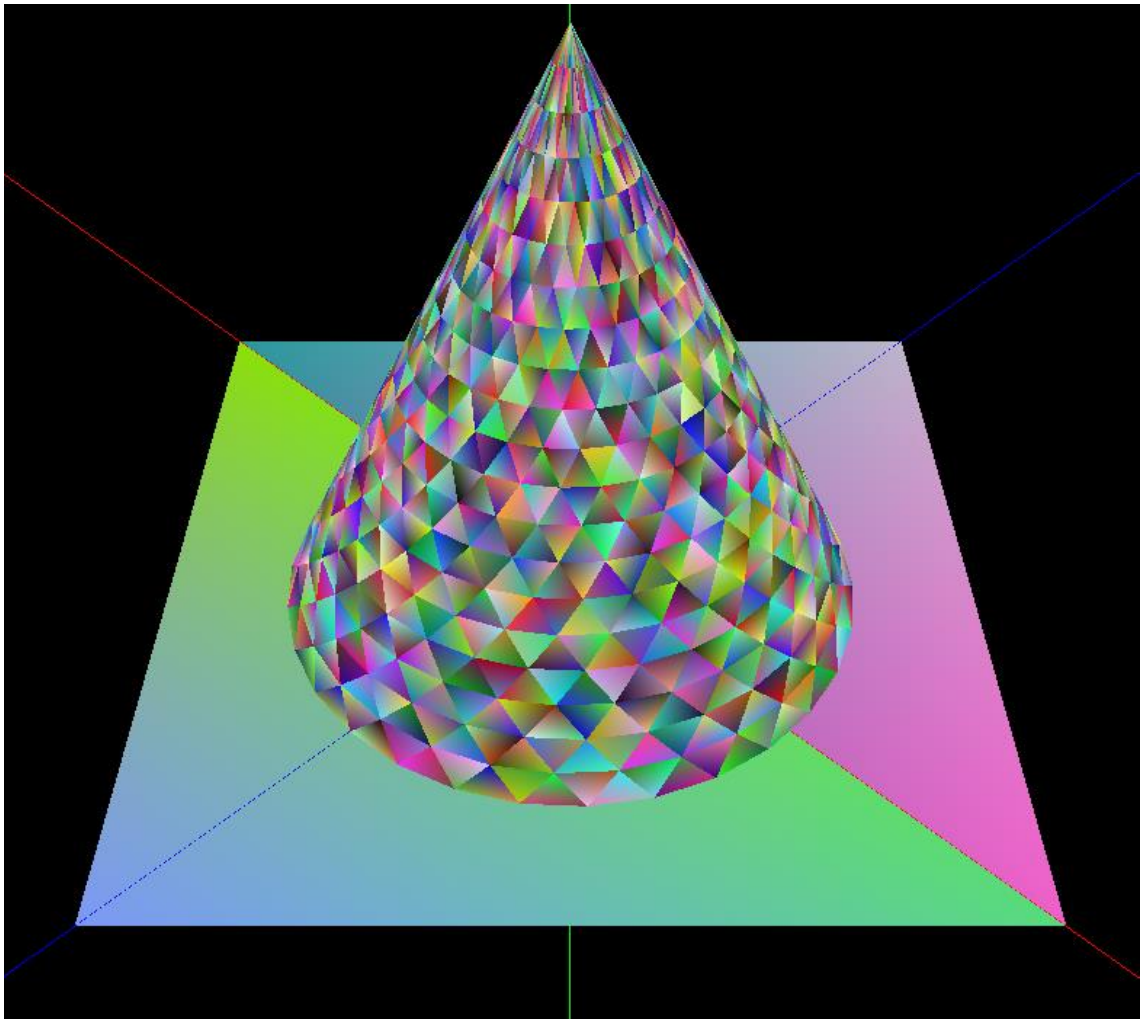
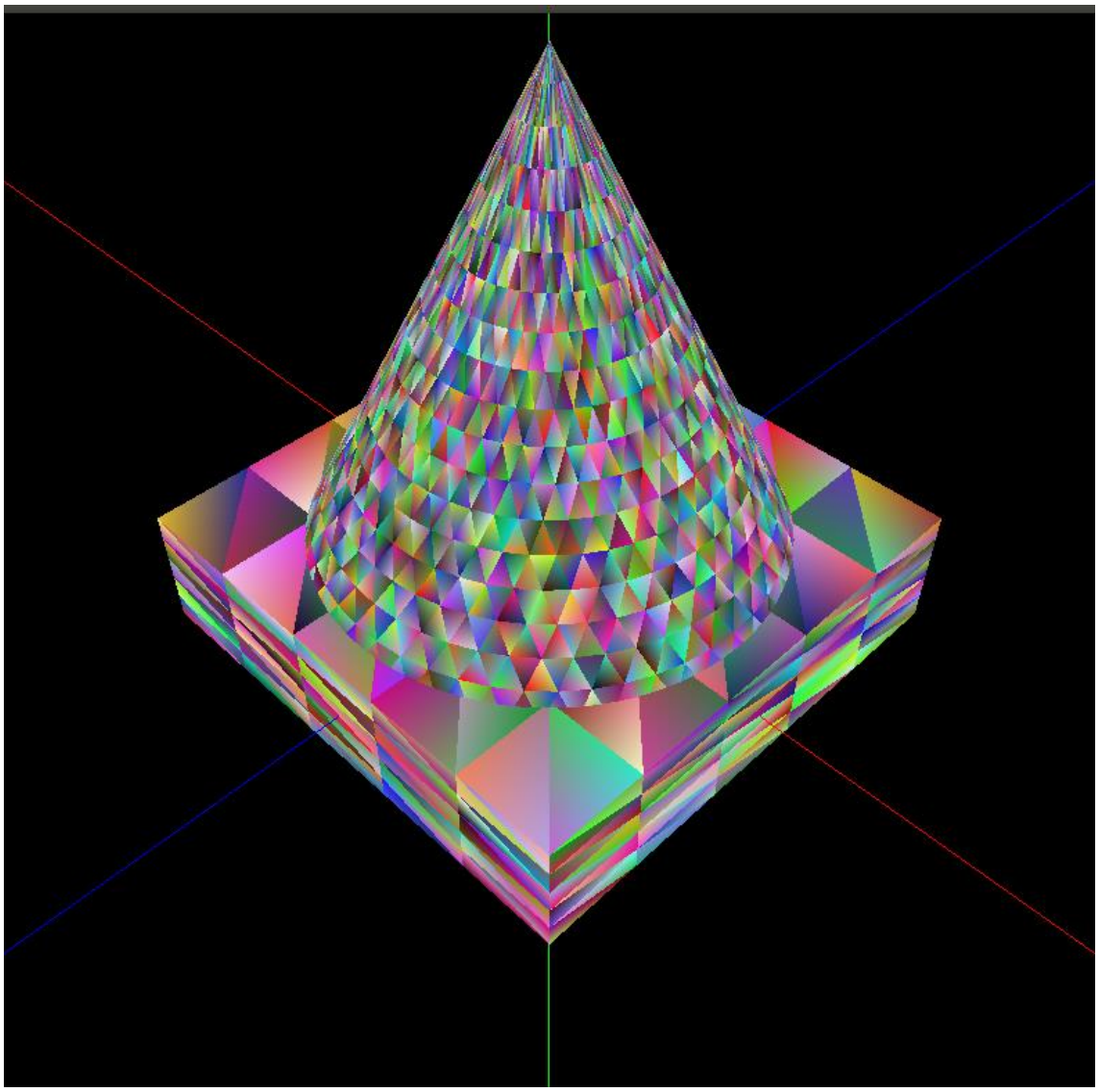


Figura *OpenGL* (printBox(4, 1, 4, 5) + printCone(2, 40, 50, 20))



Trabalho Prático – 2ª Fase

Estrutura - 2ª Fase

Por forma a satisfazer o que é pedido nesta etapa do trabalho prático, demos seguimento ao que foi feito na 1ª fase. Assim, alteramos a estrutura do *engine*, deixando a do *generator* inalterada.

Foram acrescentadas algumas estruturas de dados para armazenar informação relativa às transformações geométricas e aos blocos *<group> … </group>*. Apresentam-se de seguida as mesmas:

Estrutura de dados (1) - Transformações Geométricas

```
typedef struct {
    char type; /* T - Translate, R - Rotate, S - Scale, C-
Color */
    float param1;
    float param2;
    float param3;
    float param4;
} Transformation;
typedef std::vector<Transformation> Transformations;
```

De notar que os parâmetros acima mencionados variam consoante a variável *type* que indica qual a transformação ou coloração a ser aplicada.

Estrutura de dados (2) - Grupos

```
typedef struct group {
    Transformations trans;
    Vertices vert;
    std::vector<struct group> subGroups;
} Group;
```

É de salientar a presença de uma estrutura recursiva, *subGroups*, que contém os grupos aos quais serão aplicadas as transformações do grupo-pai.

A segunda componente desta construção diz respeito aos vértices do grupo em questão. Assim, foi atualizada a estrutura da mesma para a que se mostra a seguir:

Estrutura de dados (3) : Vértices

```
typedef struct {
	float x;
	float y;
	float z;
} Vertex;
typedef std::vector<Vertex> Vertices;
```

Após a definição destas estruturas de dados, partimos para a declaração de algumas funções que irão manipular a mesma. Foram criadas 2 funções: uma delas para adicionar os vértices das primitivas, previamente presentes num ficheiro, num vetor apropriado; a outra para processar a informação contida nos blocos *<group> … </group>* e adicioná-la aos subgrupos do respetivo bloco.

Uma vez definidas estas funções, passamos para a re-implementação do desenho dos vértices das primitivas geométricas, adaptando o código às novas estruturas de dados.

Por forma a traçar os vértices das figuras é percorrida a estrutura de dados *Group*, interpretando as transformações das mesmas através da variável *type*, tal como já foi dito anteriormente.

Depois deste processo são desenhados os vértices que caracterizam as primitivas, atribuindo-lhes as cores especificadas na *tag model*, aplicando um ligeiro gradiente para que a figura não fique totalmente uniforme, permitindo visualizar mais convenientemente no espaço 3D os planetas, a lua e o sol que compõem o modelo proposto.

Todo este processo mencionado atrás é aplicado a todos os blocos *<group> … </group>* presentes no ficheiro *XML*. Se eventualmente houver

um “subgrupo” num determinado bloco deste tipo são aplicados estes passos, recursivamente, ao mesmo.

Análise e tratamento do ficheiro *XML*

Após estas alterações, houve a necessidade de adaptar o tratamento do ficheiro *XML* que é passado como argumento ao executável do *engine*. Tal como na 1ª fase do trabalho, é averiguado se o ficheiro em causa possui o formato/extensão corretos e se se encontra delimitado pelas *tags* da *scene*, isto é, <*scene*> … </*scene*>. Depois de verificarmos o ficheiro, passamos à adição de toda a informação do ficheiro à nossa estrutura global, à qual chamamos *mainGroup*, de forma a termos todo o conteúdo disponível quando for necessário desenhar a cena. Note-se que esta adição é feita recursivamente, iterando por todos os subgrupos da *tag scene*.

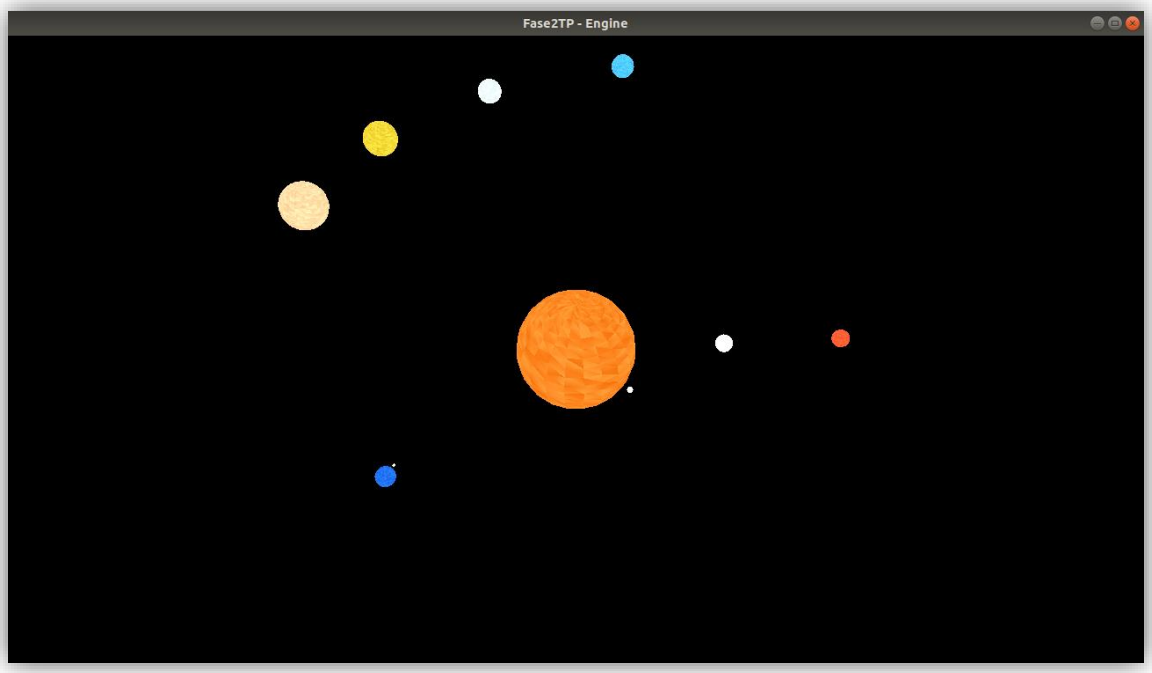
Ficheiro *XML* de exemplo

De seguida, apresentamos um exemplo do conteúdo de um ficheiro *XML*, que ilustra todas as funcionalidades que o nosso engine fornece ao utilizador. Por um lado, demonstra o uso dos três tipos de transformações (translação, escala e rotação), por outro, demonstra a possibilidade de existirem grupos que herdam as propriedades de grupos-pai. Demonstra ainda como podemos importar ficheiros de pixéis para uma cena, incluindo a respetiva cor (que será usada como referência, pois, como referido anteriormente, será aplicado um ligeiro gradiente), através do uso de modelos.

```
<scene>
	<group>
		<translate Z="-15"/>
		<scale X="0.3" Y="0.3" Z="0.3"/>
		<models>
			<model file="../planet.3d" R="0.1" G="0.4" B="0.9"/>
		</models>
		<group>
			<rotate angle="-35" axisX="1"/>
			<translate Z="2.7"/>
			<scale X="0.15" Y="0.15" Z="0.15"/>
			<models>
				<model file="../planet.3d" R="1" G="1" B="0.8"/>
			</models>
		</group>
	</group>
</scene>
```

Resultado obtido

Abaixo, podemos ver o nosso modelo estático do sistema solar, do qual o exemplo do ponto anterior faz parte. Este modelo é uma parte integrante do trabalho, pois representa a cena de demonstração pedida.



Conclusão

Durante a realização da 1ª fase deste trabalho prático foi observado pelos elementos deste grupo que, de facto, é preciso ter atenção quanto à definição dos vértices da figura final que se pretende construir. Este aspeto é importante uma vez que é necessário respeitar a regra da mão direita. Para além desta observação, com a primeira fase deste trabalho adquirimos não só conhecimentos a nível do tratamento de documentos anotados em *XML* bem como a nível da perceção do espaço 3D para a construção das primitivas gráficas pedidas.

O objetivo principal desta fase do trabalho foi cumprido na íntegra, ou seja, gerar ficheiros com a informação dos vértices das primitivas requisitadas para o futuro modelo e desenvolver um motor que interpretasse o ficheiro de configuração, anotado em *XML*, fazendo a sua representação gráfica. Para tal foi definido com sucesso o formato dos ficheiros com a informação de cada uma das figuras, processando corretamente os ficheiros *XML* com a informação relativa a uma cena. Foi ainda acrescentada a possibilidade de movimentar as figuras ao longo dos eixos x e z, podendo o utilizador aproximar-se ou afastar-se da mesma.

Quanto à 2º etapa deste trabalho, foi implementado com sucesso um novo *engine* que permite aplicar transformações geométricas (translação, rotação e escala) às cenas definidas previamente. Tal como foi proposto nesta fase, também demos a possibilidade de blocos *<group> … </group>* herdarem as transformações geométricas do bloco *<group> … </group>* pai. Com estas modificações no *engine,* elaboramos um modelo estático que simula o sistema solar, representando os planetas, o sol e a lua.

A 2ª fase deste trabalho prático foi concluída com alguma celeridade por parte dos elementos do grupo, uma vez que apenas era necessário modificar a estrutura do *engine*.