

Universidade do Minho
Departamento de Informática

Mestrado Integrado em Engenharia Informática

Geometric Transforms



Graphical primitives

A82238 João Gomes
A81953 Pedro Barros
A80328 Pedro Lima
A80624 Sofia Teixeira

Braga
2020

1 Introdução

Esta segunda fase do trabalho proposto na unidade curricular de computação gráfica, inserida no Mestrado Integrado de Engenharia Informática, teve como principal objetivo o desenvolvimento de um sistema solar através de transformações geométricas às nossas primitivas gráficas.

Mais uma vez, isto foi conseguido através da utilização de ferramentas sugeridas pelos docentes, nomeadamente o Microsoft Visual Studio, assim como alguns toolkits, sendo o *GLUT (OpenGL Utility Toolkit)* o único a ser utilizado nesta fase inicial. Estas ferramentas permitiram, em suma, trabalhar em OpenGL de modo a criar modelos tridimensionais e alterá-los de uma forma apropriada, tendo agora como objetivo a criação de um "mapa" tridimensional que é o nosso sistema solar.

2 Arquitetura do Projeto

Após efetuarmos uma avaliação geral do enunciado proposto, deliberamos que a melhor abordagem seria dividir o projeto em dois pedaços principais:

1. **Generator** - Aqui estão definidas as respetivas formas geométricas. Este generator vai criar os vértices de modo a que seja possível representar as formas da maneira pretendida (dado um certo comprimento, altura...).
2. **Engine** - Esta é a aplicação que contém as funcionalidades requiridas para demonstrar as figuras geométricas pretendidas. Permite a exibição tridimensional das mesmas que são especificadas no ficheiro *xml*.

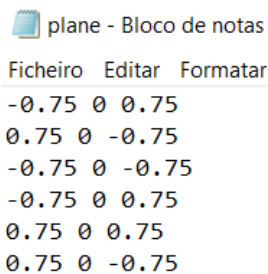
2.1 Generator

O *generator* ao ser executado vai gerar sempre um ficheiro *config.xml* e dependendo dos argumentos gera o ficheiro *.3d* pretendido.

Estes ficheiros podem posteriormente ser utilizados na config, que será parsed mais tarde pela nossa *engine*.

O generator é composto neste momento pelos seguintes componentes:

- **main.cpp** - A *main* do generator receberá os parâmetros do sólido que pretendemos gerar, e cria os pontos através do respetivo ficheiro dependendo do input recebido. No final deste processo, o writer escreverá ainda um ficheiro *.3d* onde estarão os pontos da forma geométrica criada.
- **writer.cpp** - O *writer* escreve os pontos gerados para um ficheiro *.3d*. Tal como é possível observar no exemplo da figura 1, são escritas em cada linha as coordenadas *x*, *y* e *z* de cada ponto.



```
plane - Bloco de notas
Ficheiro Editar Formatar
-0.75 0 0.75
0.75 0 -0.75
-0.75 0 -0.75
-0.75 0 0.75
0.75 0 0.75
0.75 0 -0.75
-0.75 0 0.75
0.75 0 -0.75
```

Figura 1: Ficheiro com pontos.

Os seguintes componentes formulam os pontos da respetiva primitiva gráfica que serão recebidos pelo **writer.cpp**:

- **plane.cpp**
- **box.cpp**
- **cone.cpp**
- **sphere.cpp**

2.2 *Engine*

Tal como foi mencionado no ponto anterior, o *Engine* tem como função principal gerar a representação gráfica dos sólidos a partir do nosso ficheiro *config.xml*, que por sua vez conterá os respetivos ficheiros *.3d*.

Assim como o *Generator*, o *Engine* é constituído por uma série de ficheiros auxiliares que facilitam todo o processo de criação da forma geométrica:

- **main.cpp** - A *main* realiza o *parser* do ficheiro *config.xml*, associando cada ficheiro às suas respetivas transformações através do uso da classe **Object**. Por fim, envia estas informações ao *renderer*.
- **renderer.cpp** - O *renderer*, após invocar o *reader* para este obter os pontos pretendidos, formula os triângulos que formarão a primitiva geométrica e aplica-lhes as respetivas transformações geométricas.
- **reader.cpp** - O *reader* lê os pontos dos ficheiros *.3d* indicados no ficheiro *config.xml* e consequentemente transforma-os num vetor.

2.3 *Common*

A pasta *Common*, como o nome sugere, é comum a ambas as aplicações mencionadas previamente.

- **Point.cpp** - Neste ficheiro está definida a nossa estrutura de dados, um *Point*. Como o nome da classe sugere, representa um ponto através das suas coordenadas (x,y,z).
- **Object.cpp** - Este ficheiro vai ser utilizado para associar os ficheiros às suas respetivas transformações. Deste modo, vai ter um *filename*, um vetor de *floats* para as rotações e vários *floats* com as informações de cada eixo do *translate* e do *scale*.

2.4 *Demos*

Nesta pasta, tal como na *Common*, são colocados ficheiros partilhados pelo *Generator* e pelo *Engine*. Assim, são aqui guardados os ficheiros com os pontos dos **modelos 3D** criados pelo *Generator* que vai ser posteriormente lido pelo *Engine*. Para além destes ficheiros, encontra-se também o ficheiro XML onde estes vão ser referidos:

- **config.xml** - ficheiro que é limpo pelo *Generator* onde se irá colocar o código *xml* relativo às figuras e transformações geométricas que o *Engine* irá posteriormente ler e desenvolver

3 Primitivas Gráficas

Nesta secção do relatório faremos uma breve introdução de cada primitiva gráfica, dos respetivos parâmetros e processos de desenvolvimento. Neste projeto temos como primitivas gráficas o **plano**, a **caixa**, o **cone**, e a **esfera**. Estas denominam-se primitivas gráficas uma vez que são formas geográficas irreduzíveis (para além do triângulo, a forma geográfica que permitirá a criação de todas as outras).

3.1 Triângulos

Como mencionado em cima, a forma geográfica fundamental deste projeto será o triângulo, uma vez que é este que permite todas as outras formas. Este é construído seguindo a regra da mão direita, para que seja possível a sua representação gráfica após ter sido processado pela máquina.

Esta regra é demonstrada na figura seguinte, onde está representada a construção de um triângulo para que este seja visível após ter sido criado.

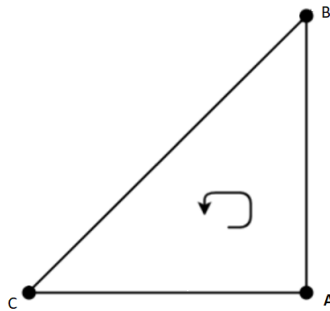


Figura 2: Construção do Triângulo.

3.2 Plano

Parâmetros:

- Dimensão

O plano tem como input uma dimensão. Esta dimensão será o tamanho de cada aresta do plano. A solução conseguida pelo grupo para esta primitiva gráfica foi através da dimensão recebida pelo programa, descobrir as coordenadas para criar um plano centrado à origem.

Sendo **d** a dimensão recebida, querendo centrar o plano à origem, sabemos que os vértices serão em **x** e **z** respetivamente ordenados por quadrante (como podemos verificar na figura 3):

1. $(d/2, d/2)$
2. $(-d/2, d/2)$
3. $(-d/2, -d/2)$
4. $(d/2, -d/2)$

Sabendo as coordenadas, foram então facilmente construídos dois triângulos, segundo as regras acima mencionadas, de modo a criar um plano, tal como se pode reparar na figura 4.

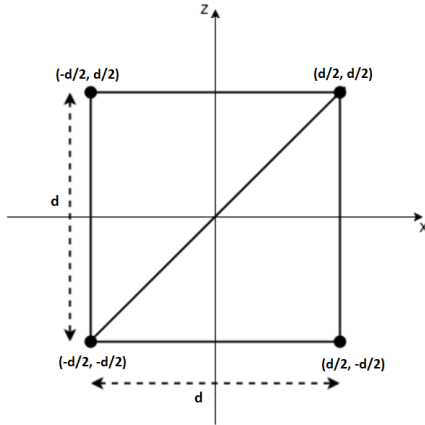


Figura 3: Construção do Plano.

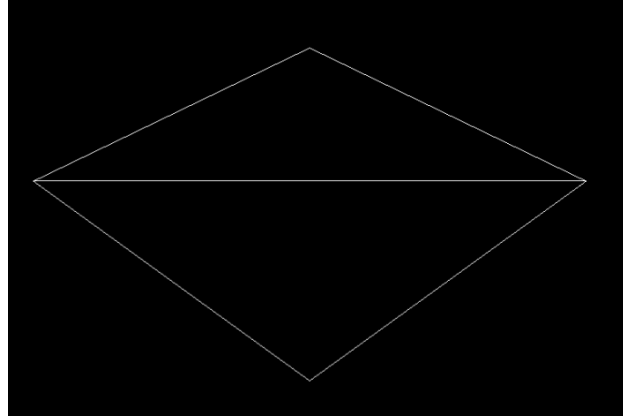


Figura 4: Plano.

3.3 Caixa

Parâmetros:

- **Altura**
- **Comprimento**
- **Profundidade**
- **Divisões**

Após receber a altura, comprimento e largura, semelhante ao plano, estes valores serão divididos por 2 de modo a obtermos as coordenadas que corresponderão aos vértices da nossa caixa de centro na origem.

Para realizar uma caixa com n partições, as arestas serão então divididas por n , sendo então obtido o tamanho das arestas dos "blocos" que vão constituir a caixa.

O processo de construção da caixa efetua-se desenhando os triângulos necessários conforme se vai percorrendo as zonas dos blocos. A ordem de construção trata-se então de $-z$ para z , $-x$ para x e $-y$ para y , tal como se pode observar na figura 5. Começa-se então pela camada inicial, pela linha da esquerda. A partir do fundo, começam a ser desenhados os triângulos da area do fundo avançando assim na direção de $-z$ para z . De seguida passa-se para a linha seguinte da direita e realiza-se o mesmo processo. Quando as linhas forem todas percorridas, reinicia-se o processo na camada de cima.

De modo a permitir uma correta utilização do nosso *Generator* é importante realçar que ao introduzir os parâmetros da *box*, quando o parâmetro **divisões** é passado com o valor 0, construir-se-á uma caixa sem qualquer divisão, tal como representada na figura 6.

Quando este mesmo parâmetro tem um valor x , isto representa o número de divisões que serão efetuadas em cada aresta da figura geométrica. Isto significa que as divisões aumentam exponencialmente e podemos calcular o número de divisões (número de "blocos") através da expressão 2^{x+1} como se pode verificar nas figuras 7 e 8.

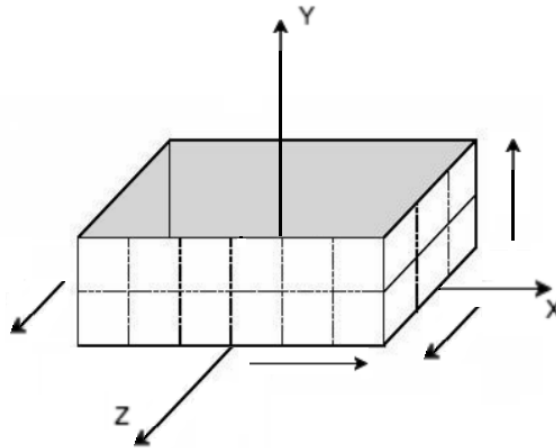


Figura 5: Construção da Caixa.

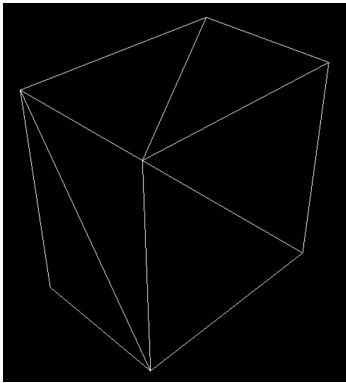


Figura 6: Caixa - sem divisões.

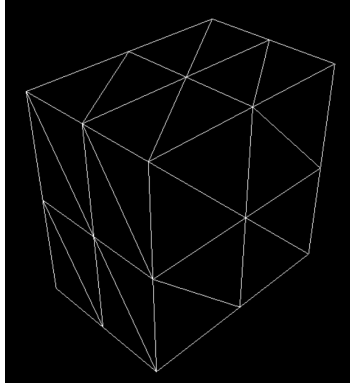


Figura 7: Caixa - 1 divisão.

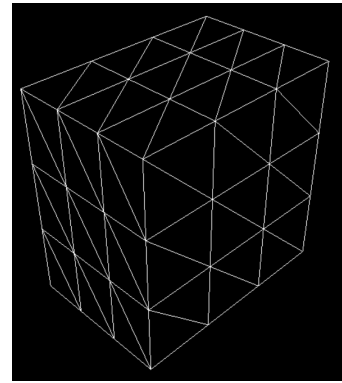


Figura 8: Caixa - 2 divisões.

3.4 Cone

Parâmetros:

- altura
- raio
- *slices*
- *stacks*

Para construir o cone, a nossa aplicação recebe como *input* os 4 parâmetros acima referidos.

A altura e o raio são parâmetros bastantes intuitivos quanto à sua utilização. Uma vez que as nossas primitivas são construídas a partir de triângulos, surgiu a necessidade de dividir a superfície do cone para possibilitar o desenho dos triângulos necessários. É aqui que entram os parâmetros *slices* e *stacks*. As *slices* representam cortes verticais perpendiculares à base do cone, enquanto que as *stacks* representam cortes horizontais paralelos à base do cone, estando ambas estas divisões encontradas na figura 9.

Após todas as *slices* e *stacks* serem aplicadas, o nosso cone fica dividido em múltiplas secções, todas idênticas entre si. Cada uma destas secções servirá para desenhar dois triângulos que partilham entre si dois vértices.

Usando coordenadas polares e fazendo uso das expressões que as convertem em coordenadas cartesianas, facilmente representamos um vértice. Existem duas formas diferentes de desenhar um cone. A primeira, iterando primeiro as *stacks* e depois as *slices*; a segunda, simplesmente fazendo as mesmas em ordem contrária. Nós optamos pela primeira opção, ou seja, em cada *slice* eram desenhadas as suas *stacks* antes de passar para a *slice* seguinte. Para realizar este desenvolvimento das *stacks*, o nosso raio base é diminuído sempre num valor constante, dado por **raio** - (**raio** / *stacks*), de modo a conceber o formato do cone. Para iterar as *slices*, o ângulo *alpha* vai aumentando um certo valor fixo, valor esse que é dado pela expressão $360^\circ / \textit{slices}$, e o raio é retornado ao valor inicial.

Com efeito, conforme o processo explicado, foi desenhado o cone da figura 10.

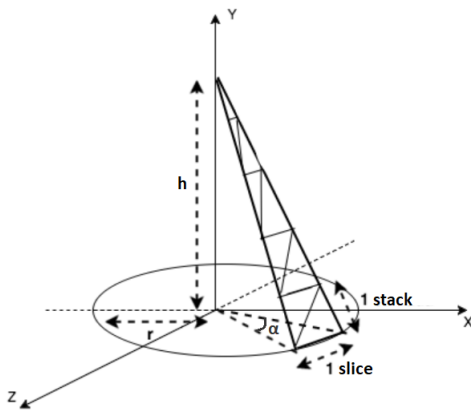


Figura 9: Construção do Cone.

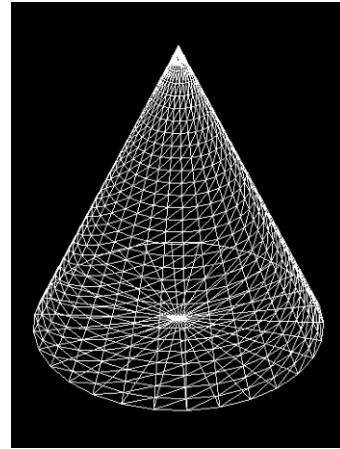


Figura 10: Cone.

3.5 Esfera

Parâmetros:

- **raio**
- *slices*
- *stacks*

O desenvolvimento da esfera assemelha-se ao do cone, uma vez que as *slices* e as *stacks* desempenham o mesmo papel em ambos. A maior diferença entre o a esfera e o cone consiste que na esfera os pontos encontram-se sempre à mesma distância do centro. Assim, para conseguir calcular as coordenadas dos pontos, recorreremos às coordenadas esféricas e às formulas de conversão destas para coordenadas cartesianas. Para tal, além do ângulo *alpha* já utilizado no desenho do cone, usamos um outro ângulo *beta*, como se pode verificar na figura 11.

O processo de desenho foi contrário ao usado no cone, ou seja, foram iteradas as *slices* e depois as *stacks*. Desta forma, entre cada *slice* apenas temos de incrementar o valor do *alpha* como feito no cone. E entre cada *stack* apenas temos de incrementar o valor de *beta*. Este ângulo *beta* é diferente do ângulo *alpha* em alguns aspetos. Como percorre a esfera de baixo a cima (ou vice-versa), começa em -90° e é iterado até chegar aos 90° . Para calcular o quanto somar ao ângulo *beta* em cada iteração das *stacks*, apenas dividimos $180^\circ / \textit{stacks}$.

Efetivamente, conforme o processo explicado, foi desenhado a esfera da figura 12.

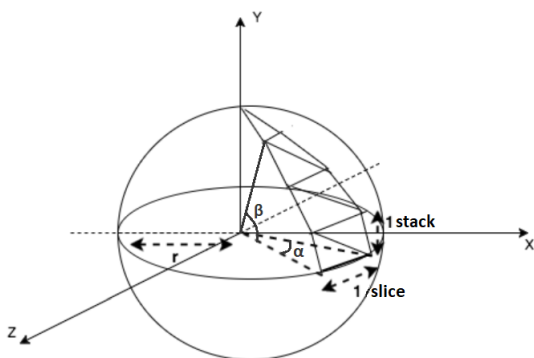


Figura 11: Construção da Esfera.

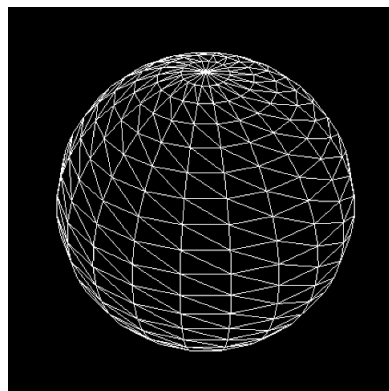


Figura 12: Esfera.

3.6 Conjuntos

Esta secção serve apenas para demonstrar a possibilidade de executar, através do nosso ficheiro *config.xml* múltiplas figuras geométricas em simultâneo. Como exemplo disto temos na figura 13 um cone e um plano, que denominamos *Hat*, e na figura 14 uma esfera em simultâneo com uma caixa.

É importante mencionar que o ficheiro *.xml* é criado (ou se já existir, limpo) no *Generator*, porém, este ficheiro é alterado manualmente com os objetos *.3d* que pretendemos criar.

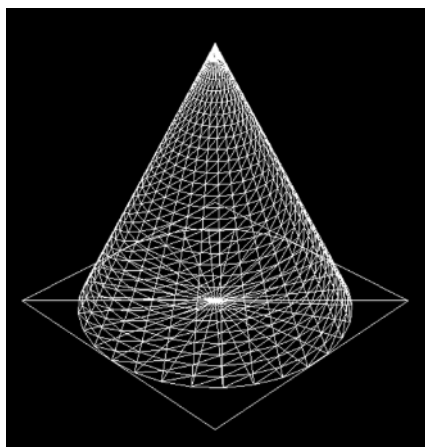


Figura 13: Conjunto "Hat": Cone e Plano.

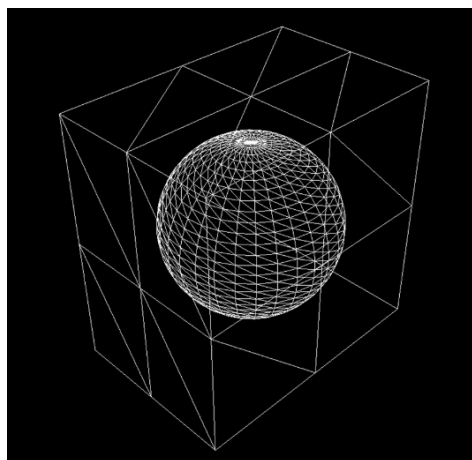


Figura 14: Conjunto: Esfera e Caixa.

4 Transformações Geométricas

Para a segunda fase deste projeto, foi-nos requerida a definição de métodos de transformação de um objeto tridimensional, nomeadamente o **scale**, **rotate** e **translate**.

Estes métodos foram desenvolvidos na nossa classe **renderer** e é nesta onde os pontos são criados consoante os parâmetros recebidos.

Os métodos de **scale**, **rotate** e **translate** existentes na biblioteca do **glut** afetam toda a perspetiva do projeto, portanto, surgiu a necessidade aplicar as funções **glPushMatrix** e **glPopMatrix** de modo a que os funções mencionados inicialmente fossem aplicadas não ao desenho como um todo, mas sim a cada respetiva figura geométrica a desenhar.

De modo a implementar o referido a cima foi então chamada a função **glPushMatrix** seguida de **glScalef**, **glRotatef** e **glTranslatef**. Consequentemente concebe-se a figura geométrica pretendida e chama-se a função **glPopMatrix**.

5 Parse XML

Para esta segunda fase do projeto, foi requerida a criação ou utilização de um parser que a partir do nosso ficheiro XML automatizasse a criação dos objetos tridimensionais. Isto foi conseguido através da utilização de uma ferramenta denominada *TinyXML2*.

O *TinyXML2* é uma ferramenta criada com o intuito de ajudar no *parsing* de informação de ficheiros *XML* (*eXtensible Markup Language*). A maneira como esta funciona é, fundamentalmente, construir um *DOM* (*Document Object Model*) que pode ser lido, alterado e guardado. Isto permite-nos dar "load" ao nosso ficheiro **config.xml** e retirar a informação necessária a ser enviada para o nosso **renderer**.

De modo a organizar a informação e a permitir a separação de rotações, translações ou escalonamentos de figuras geométricas diferentes, o nosso ficheiro **config.xml** está subdividido em grupos.

O subgrupos mencionados herdam os atributos dos grupos ascendentes. Assim, de modo a fundir as várias transformações geométricas que os subgrupos vão eventualmente ter precisamos de ter alguns cuidados:

- No caso do **translate**, necessitamos de somar as respetivas variáveis dos vários **translates**, ou seja, tendo um Grupo A, Subgrupo B e neste um Subgrupo C, o **translate** final do subgrupo C seria **translate A + translate B + translate C**.
- O **scale** funcionará do mesmo modo que o **translate** porém as variáveis serão multiplicadas e não somadas.
- O **rotate**, uma vez que não segue o mesmo formato que as funções anteriores, não seria tão simples de apenas somar ou multiplicar as respetivas coordenadas, portanto, criamos um vetor com as informações de cada rotate para que em cada respetivo grupo se aplique os rotates efetuados até esse momento, dos respetivos antecessores.

Ao efetuar o *parser*, sempre que este atinge o atributo **model** é criado um **Object** com o *filename* desse model e as transformações geométricas de acordo com os métodos acima referidos. Este **Object** é adicionado a um vetor de **Object** que posteriormente vai ser enviado para o **renderer.cpp** onde este vai iterar o vetor e desenhar cada figura geométrica conforme as suas especificações.

6 Keys

Embora não fosse necessário, foi decidido implementar as seguintes *keys* de modo a facilitar a visualização das figuras geométricas:

- **GLUT_KEY_F2** - aproxima a câmara das figuras geométricas
- **GLUT_KEY_F1** - distancia a câmara das figuras geométricas
- **GLUT_KEY_UP** - desloca a câmara para cima
- **GLUT_KEY_DOWN** - desloca a câmara para baixo
- **GLUT_KEY_RIGHT** - desloca a câmara para a direita
- **GLUT_KEY_LEFT** - desloca a câmara para a esquerda

7 Sistema Solar XML

Após ter sido definida a metodologia para desenvolvimento de uma cadeia de figuras que partilham propriedades com um sistema de hereditariedade, foi então desenvolvido o ficheiro **sistemasolar.xml** de modo a realizar uma simulação estática do sistema solar.

Para garantir uma aproximação à realidade nesta simulação do sistema solar, utilizamos a transformação geométrica **scale** para obter os diferentes tamanhos dos planetas e luas em relação ao sol, assim como a transformação geométrica **translate** para distanciar os astros uns dos outros com valores proporcionais à realidade.

Para tal, usamos como suporte os ficheiros criados da seguinte maneira:

- **sun.3d** - ./generator sphere 9 50 50 sun.3d
- **sphere.3d** - ./generator sphere 6 30 30 sphere.3d

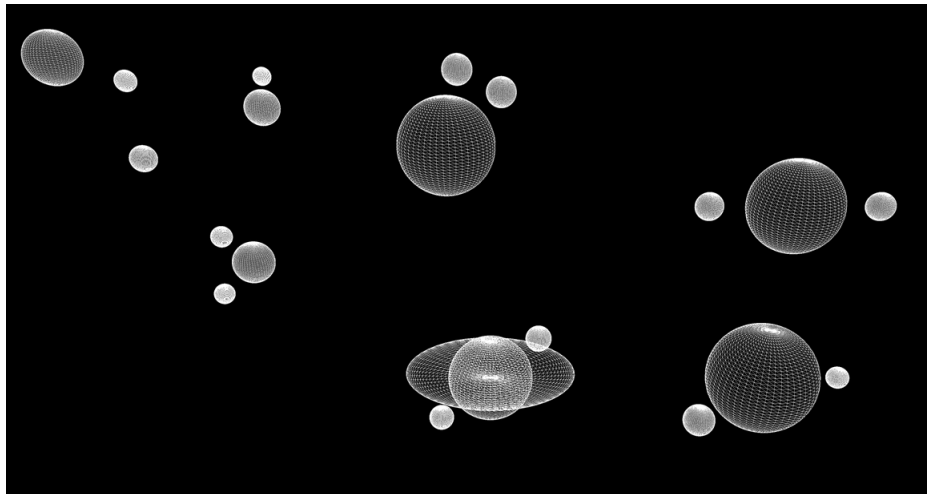


Figura 15: Sistema Solar.

8 Exemplos XML

Para além do proposto, foi decido criar mais ficheiros XML de modo a melhor testar as possibilidades e variedades dos vários parâmetros e agrupamentos dos mesmos.

De modo a utilizar estes ficheiros, é necessário copiar o seu conteúdo e colocá-lo no ficheiro **config.xml**.

Para as seguintes construções foram usados os seguintes ficheiros que foram criados a partir dos seguintes parâmetros/comandos:

- **sphere.3d** - ./generator sphere 1 20 20
- **cone.3d** - ./generator cone 1 20 20
- **plane.3d** - ./generator plane 1
- **box.3d** - ./generator box 1 1 1 0
- **box4.3d** - ./generator box 10 2 1 4

8.1 Castelo

Ficheiro XML usado para obter a figura 16:

- `castle.xml`

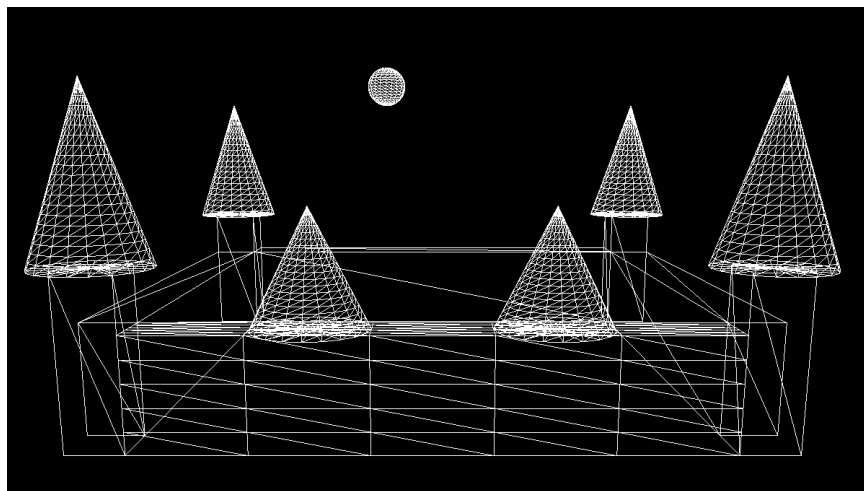


Figura 16: Castelo.

8.2 Homem de Neve

Ficheiro XML usado para obter a figura 17:

- `snowman.xml`

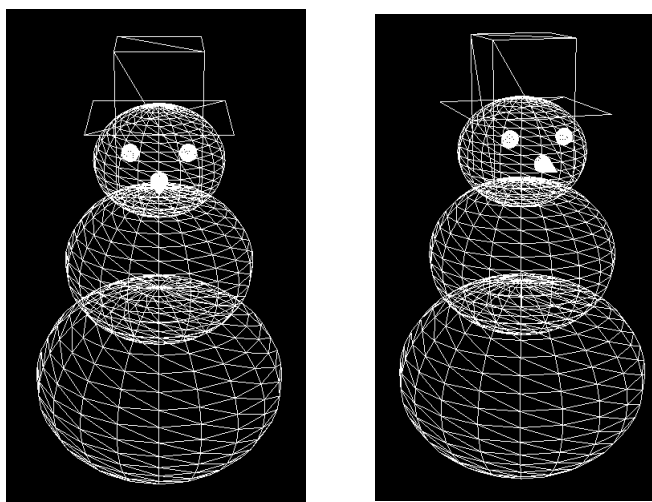


Figura 17: Homem de Neve.

8.3 Gelado

Ficheiro XML usado para obter a figura 18:

- `icecream.xml`

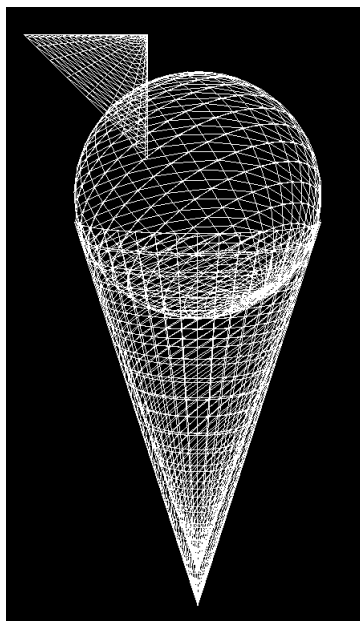


Figura 18: Gelado.

9 Conclusão

Na segunda fase deste projeto denotou-se a necessidade de organizar de um modo funcional as respetivas classes e ficheiros a ser criados, assim como consolidamos o nosso conhecimento com a biblioteca **GLUT**.

Outro conceito fundamental nesta fase do trabalho foi o pensamento lógico de modo a desenvolver um sistema solar que consiga simular decentemente um Sistema Solar.

Quanto ao parse de ficheiros XML esta etapa não se provou particularmente difícil uma vez que sendo alunos de engenharia informática já teríamos lidado com problemas e requerimentos semelhantes.

Em suma, consideramos que esta fase correu como esperada, sem nenhum problema particularmente difícil quanto à sua resolução, tendo sido apenas necessário consolidar os nossos conhecimentos face a uma agora "não tão nova" linguagem de programação.