Universidade do Minho
Departamento de Informática

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Normals and Texture Coordinates

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

Braga
2020

1 Introdução

Esta quarta fase do trabalho proposto na unidade curricular de computação gráfica, inserida no Mestrado Integrado de Engenharia Informática, tem como fim dar uma aparência sólida 3D às figuras geométricas. Este objetivo está dividido 3 etapas: luminosidade, material e textura.

Estas 3 etapas serão desenvolvidas através da utilização das ferramentas sugeridas, como é o caso das bibliotecas **GLUT, GLEW** e **Devil**. Estas permitiram trabalhar com **openGL** de modo a manipular os modelos tridimensionais criados, sendo possível gerar uma simulações mais realistas.

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 num ficheiro *xml*.

2.1 *Generator*

O *generator* ao ser executado vai gerar a figura geométrica pretendida e criar o respetivo ficheiro *.3d* onde serão colocados os pontos da mesma.

O *generator* é composto neste momento pelos seguintes componentes:

- **main.cpp** - A *main* recebe os parâmetros da figura geométrica que pretendemos gerar e cria os pontos através do respetivo ficheiro dependendo do *input* recebido.

- **writer.cpp** - O *writer* escreve os pontos gerados para um ficheiro *.3d*.

Os seguintes componentes formulam os pontos da respetiva primitiva gráfica, que serão recebidos pelo *writer*, as suas normais e os pontos da textura:

- **plane.cpp**

- **box.cpp**

- **cone.cpp**

- **sphere.cpp**

Para além dos acima mencionados, foi necessário criar outro componente que permitisse formular os pontos de uma figura geográfica apartir de um ficheiro com *patches*:

- **patches.cpp** - analisa o ficheiro que contém os *patches* e os pontos de controlo e, através das curvas de *Bezier*, formula os pontos de uma primitiva gráfica.

2.2 *Engine*

Tal como foi mencionado no ponto anterior, o *Engine* tem como função principal gerar a representação gráfica dos sólidos a partir de um ficheiro *xml*, que por sua vez conterá a menção de ficheiros *.3d*. Assim, a *Engine* recebe como argumento o nome do ficheiro *xml* que se pretende ler da seguinte maneira:

- **./engine nomeficheiro.xml**

Tal 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 *xml* indicado, associando cada ficheiro às suas respetivas transformações através do uso da classe **Object**. Por fim, envia estas informações ao *renderer*.
- **renderer.cpp** - O *renderer* 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 *xml* e consequentemente transforma-os num vetor, criando também um vetor com índices.
- **catmull.cpp** - O *catmull* recebe pontos e com estes forma a linha em que a primitiva gráfica se vai deslocar.

2.3 *Common*

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

- **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 e luzes.

  Deste modo, vai ter um vetor de vetor *floats* correspondestes às coordenadas dos diversos pontos, um vetor de *unsigned ints* correspondente aos índices, uma *string* para a textura, quatro vetores de *floats* correspondestes às informações da cor, um vetor de vetores de *floats* para as translações que usam *Catmull-Rom Curves*, um vetor de *floats* para as rotações e vários *floats* com as informações de cada eixo do *translate* regular 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*.

2.5 XML

A pasta *XML* trata-se do local onde se coloca o ficheiros *xml* que se pretende utilizar na *engine*. A pasta já contém alguns ficheiros exemplo que vão ser referidos numa secção posterior.

2.6 *Patches*

Todos os ficheiros *patch* que se pretende utilizar devem ser colocados na pasta *Patches*.

2.7 Texturas

Todos as imagens que se pretende implementar como textura devem ser colocadas na pasta *Texturas*.

3 Primitivas Gráficas

Nesta secção do relatório faremos uma breve introdução de cada primitiva gráficas, 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 irredutíveis (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.

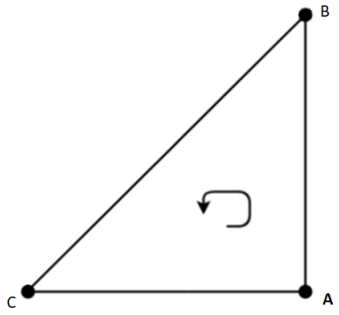

Figura 1: Construção do Triângulo.

3.2 Plano

Comando para criar um ficheiro com os pontos de um plano:

- **./generator plane dimensão nomeficheiro.3d**

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 2):

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 3.

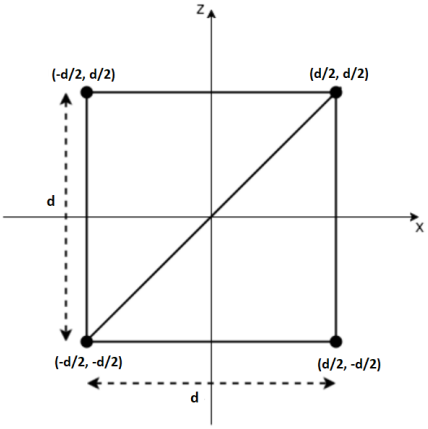

Figura 2: Construção do Plano.

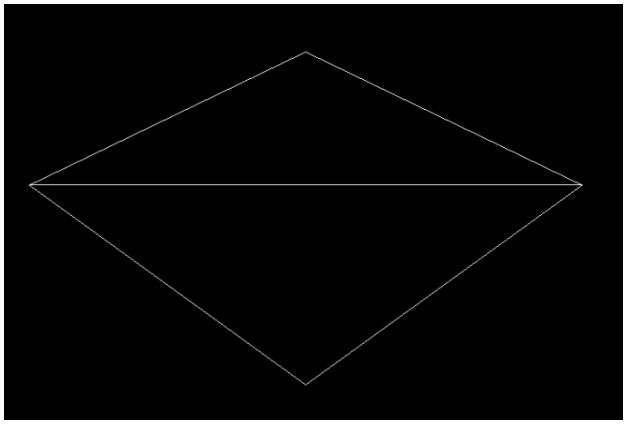

Figura 3: Plano.

3.3 Caixa

Comando para criar um ficheiro com os pontos de uma caixa:

- **./generator box largura altura profundidade divisões nomeficheiro.3d**

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 4. Começa-se então pela camada inicial, pela linha da esquerda. A partir do fundo, começam a ser desenhados os triangulos da area do fundo avançando assim na direçao 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.

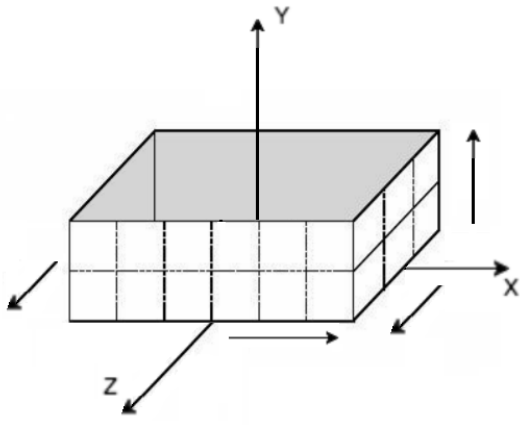

Figura 4: Construção da Caixa.

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 5.

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 6 e 7.

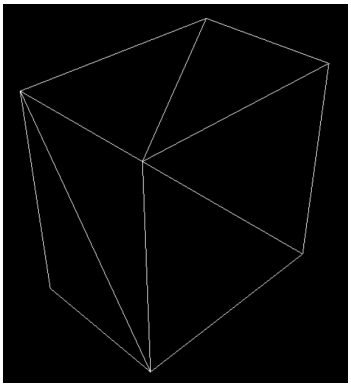

Figura 5: Caixa - sem divisões.

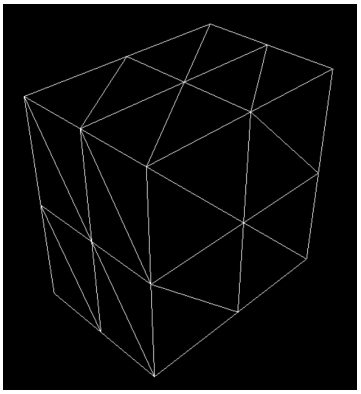

Figura 6: Caixa - 1 divisão.

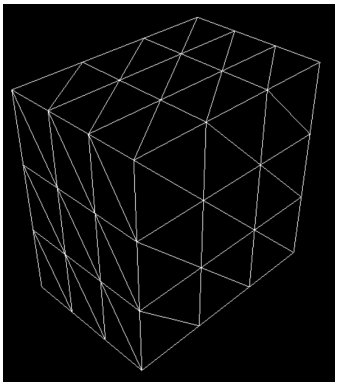

Figura 7: Caixa - 2 divisões.

3.4 Cone

Comando para criar um ficheiro com os pontos de um cone:

- **./generator cone raio altura slices stacks nomeficheiro.3d**

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 encontram-se representadas na figura 8.

Após todas as *slices* e *stacks* serem aplicadas, o nosso cone fica divido 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 360º/ ***slices***, e o raio é retornado ao valor inicial.

Com efeito, conforme o processo explicado, foi desenhado o cone da figura 9.

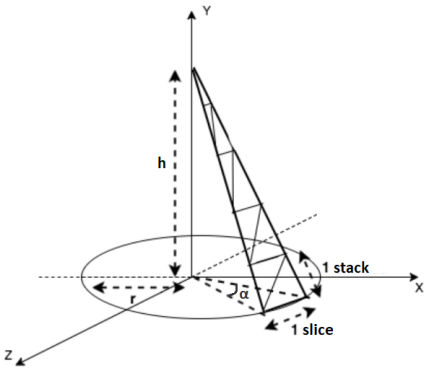

Figura 8: Construção do Cone.

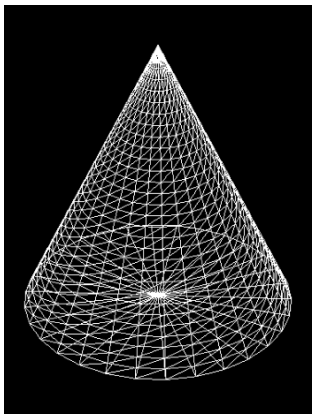

Figura 9: Cone.

3.5 Esfera

Comando para criar um ficheiro com os pontos de uma esfera:

- **./generator sphere raio slices stacks nomeficheiro.3d**

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, recorremos 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 10.

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 -90º e é iterado até chegar aos 90º. Para calcular o quanto somar ao ângulo *beta* em cada iteração das *stacks*, apenas dividimos 180º / ***stacks***.

Efetivamente, conforme o processo explicado, foi desenhado a esfera da figura 11.

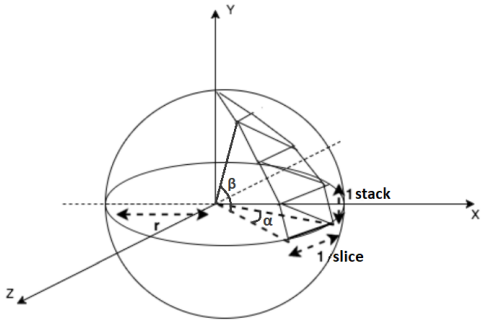

Figura 10: Construção da Esfera.

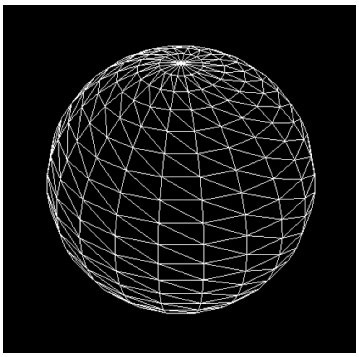

Figura 11: Esfera.

3.6 Conjuntos

Esta secção serve apenas para demonstrar a possibilidade de executar múltiplas figuras geométricas em simultâneo. Como exemplo disto temos na figura 12 um cone e um plano, que denominamos *Hat*, e na figura 13 uma esfera em simultâneo com uma caixa.

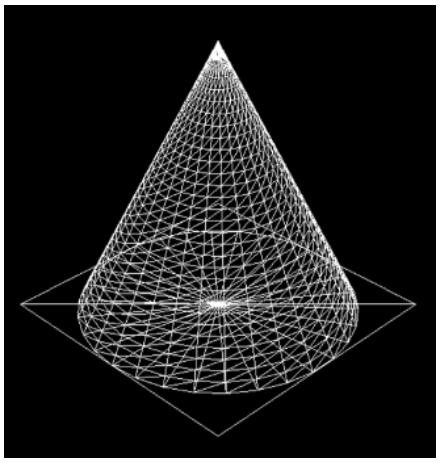

Figura 12: Conjunto "*Hat*": Cone e Plano.

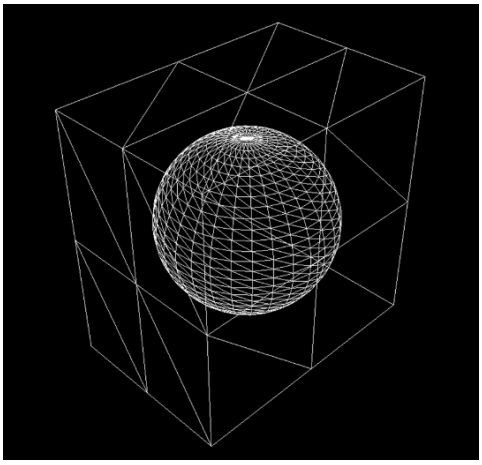

Figura 13: Conjunto: Esfera e Caixa.

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 acima, 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

Foi necessário a criação de um *parser* que, a partir de um 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 ficheiro *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, os ficheiros *xml* estão 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:

- O **translate** pode ser efetuado por dois métodos diferentes: o método normal, que recebe os valor do *x, y* e *z*, e por Catmull-Rom Curves, que recebe o valor do tempo e as coordenadas de vários pontos.
  - No primeiro caso, é necessário somar as respetivas variáveis das várias translações, ou seja, tendo um Grupo A, Subgrupo B e neste um Subgrupo C, a translação final do Subgrupo C seria **translate A + translate B + translate C**.
  - No segundo caso, uma vez que não é possível juntar as diversas translações, é criado um vetor de vetores de *floats* onde são guardados o tempo e as coordenadas de cada translação. Esta estratégia é realizada de maneira a garantir que em cada grupo se aplica as translações efetuados até esse momento, incluindo as dos respetivos antecessores.
- O **scale** funcionará do mesmo modo que o primeiro caso do **translate**, porém as variáveis serão multiplicadas e não somadas.
- O **rotate** pode ser efetuado por dois métodos diferentes: com um ângulo ou com um tempo.

  Em ambos os métodos, tal como no segundo caso do **translate**, não seria simples juntar as várias rotações. Consequentemente, foi criado um vetor com as informações de cada rotação para que em cada grupo se aplique as rotações efetuadas até esse momento, incluindo assim os respetivos antecessores.

  Para distinguir o método utilizado é adicionado ao vetor uma variável que, estando a 0, indica que é para aplicar o primeiro método e, estando a 1, indica que é para aplicar o segundo.

Ao efetuar o *parser*, sempre que este atinge o atributo **model** é criado um **Object**. Este vai conter as seguintes informações:

- os índices e as coordenadas dos pontos, das normais e das texturas do ficheiro mencionado no atributo **filename**, obtidos através do *reader*;
- as especificações de cor difusa (**diffR, diffG, diffB**), emissisva (**emisR, emisG, emisB**), especular (**specR, specG, specB**) e ambiente (**ambR, ambG, ambB**);
- a textura (**texture**);
- 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* onde este vai iterar o vetor e desenhar cada figura geométrica conforme as suas especificações.

Para além dos grupos, existe também uma secção de **lights** relativa às luzes **DIRECTIONAL, POINT** e **SPOT**. Assim, foi criado um vetor para cada tipo de luz que irá guardar as informações das respetivas luzes registadas nesta secção do XML.

6 VBOs

Nesta fase do trabalho foram implementados os **Vertex Buffer Objects** (**VBOs**) que aumentam significativamente a eficiência do projeto. Contrariamente ao modo imediato, através de VBOs é possível enviar a informação e guardá-la na GPU (*Graphical Processing Unit* ou Placa gráfica). Isto significa que a informação apenas necessita de ser enviada uma vez, sendo depois invocada através da sua *handle*.

Uma vez que a informação não está a ser enviada para a GPU em todos os *frames*, poupa-se *bandwidth* e logicamente aumenta a eficiência do trabalho.

6.1 Redução de pontos

Neste projeto, a ocorrência mais comum é a necessidade de desenhar dois triângulos que partilham dois pontos, tal como representado na figura 14. Por consequência, torna-se claro que o método de escrever no ficheiro as coordenadas *x, y* e *z* de seis pontos dos quais dois são repetidos não é eficiente.

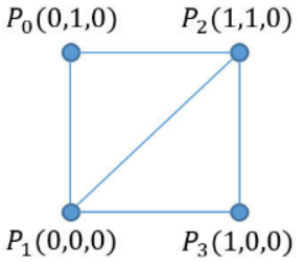

Figura 14: Rectangulo constituído por dois triângulos.

Por esse motivo, passaram a ser escritos apenas os quatro pontos não repetidos e foi atribuída uma ordem à sua escrita de forma a ser possível criar um índice comum, tal como demonstrado na figura 15.

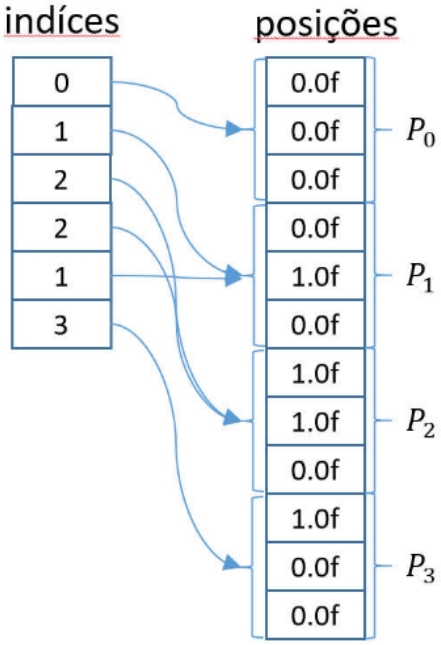

Figura 15: Ordem dos índices.

No entanto, nem todos os casos são semelhantes ao da figura 14, como é o caso da base do cone. Esta situação cria um problema ao índice previamente criado, uma vez que não segue a mesma regra.

Previamente, no *writer*, as coordenadas *x, y* e *z* de cada ponto eram escritas em cada linha do ficheiro, tal como é possível observar no exemplo da figura 16. No entanto, face à adversidade referida, nos casos da figura 14, serão escritas as coordenadas de quatro pontos numa linha e no caso da base do cone, seram escritas as coordenads de três pontos numas linha. É possível observar o novo formato na figura 17.

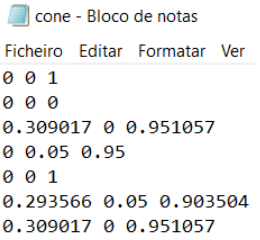

Figura 16: Excerto do ficheiro de pontos **cone.3d** - Antes.

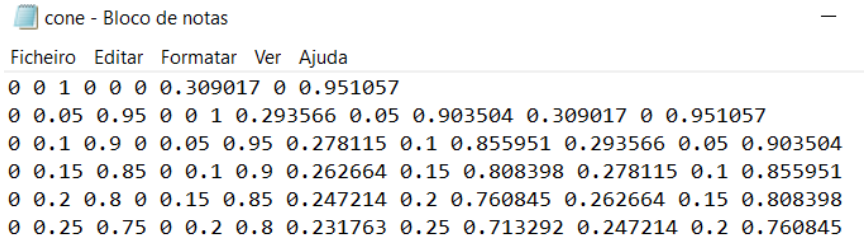

Figura 17: Excerto do ficheiro de pontos **cone.3d** - Depois.

Com efeito, através deste novo método, o *reader* consegue saber que, quando a linha tem apenas as coordenadas de três, os índices adicionados ao vetor devem retratar a ordem {*0,1,2*} e que, quando a linha tem as coordenadas de quatro pontos, os índices adicionados ao vetor devem retratar a ordem {*0,1,2,2,1,3*}.

6.2 Implementação

Previamente, os vários triângulos eram concebidos através das funções **glBegin, glEnd** e **glVertex3f**, sendo as primeiras duas usadas para delimitar os vértices definidos na terceira. Neste novo método, estas funções já não vão ser utilizadas.

A implementação de VBOs consiste em duas partes: a geração e preparação dos VBOs e o desenho das primitivas gráficas.

Na primeira parte, foram criados dois VBOs: um para os índices e um para os vértices. Cada um vai ser gerado atráves da função **glGenBuffers**, sendo de seguida ativado com a função **glBindBuffer** e preenchido pela função **glBufferData**.

Quanto à segunda parte, após ativados os VBOs, é definida a semantica através da função **glVertexPointer** e efetua-se o desenho das primitivas gráficas através da função **glDrawElements**. Denota-se que esta parte é implementada entre as funções **glPushMatrix** e **glPopMatrix**, tal como explicado anteriormente na secção 4.

7 *Bezier Curves*

Foi implementado no *Generator* uma funcionalidade que consiste em desenhar uma primitiva gráfica a partir de um ficheiro ***patch*** fornecido pelo utilizador.

Foi também necessário criar o seguinte comando para o *generator* executar este novo método:

- **./generator patch nomeFile.patch tesselation nomeFile.3d**

7.1 Ficheiro de *Patches*

O ficheiro ***patch*** tem uma estrutura muito específica que deve ser seguida.

A primeira linha do ficheiro contém o número de *patches* presentes no ficheiro. As próximas n linhas, uma para cada *patch*, contêm os 16 índices relativos a *control points* a considerar para o respetivo *patch*.

Após estas linhas de informação referente aos *patches*, a linha seguinte contem o número de *control points* existentes no ficheiro.

Por fim, todas as linhas seguintes até ao fim do ficheiro, contêm 3 coordenadas referentes a cada *control point*.

Estas características são possíveis de observar no exemplo da figura 18.

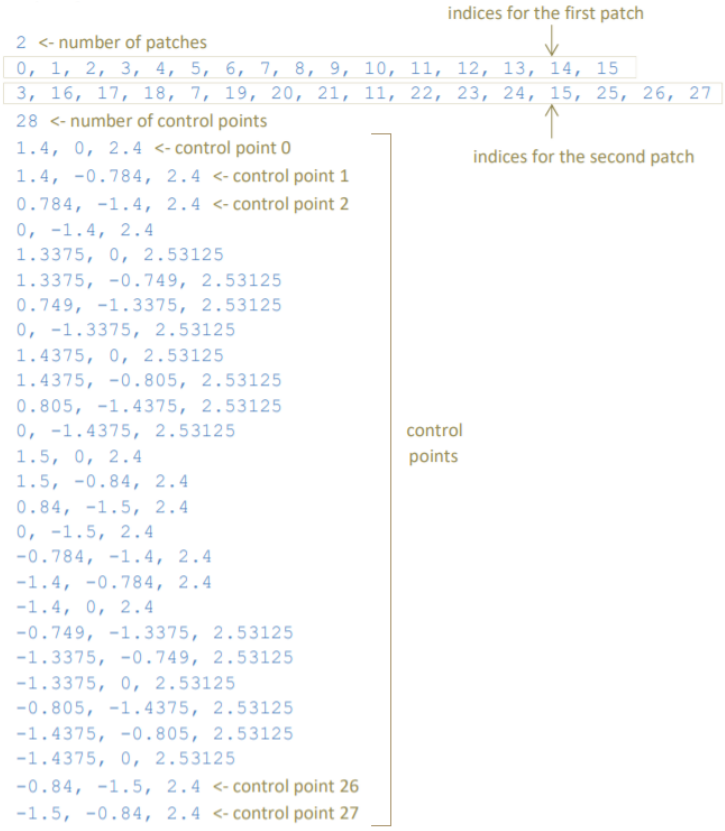

Figura 18: Ficheiro de *patches*.

7.2 Implementação

Para desenhar a primitiva gráfica, temos de tratar cada *patch* individualmente. Cada *patch* do ficheiro é formado por 16 *control points* que se obtêm através dos 16 índices, tal como mencionado acima. Estes 16 *control points* serão agrupados em 4 grupos de 4 pontos, cada grupo representando uma **Bezier Curve**.

A equação que define cada uma destas *Bezier Curves* é dada pela expressão $\mathbf{t^3 * P3 + 3 * t^2 * (1-t) * P2 + 3t * (1-t)^2 * P1 + (1-t)^3 * P0}$, onde **P0, P1, P2** e **P3** representam os 4 pontos do grupo que forma a curva.

Variando o valor de t entre 0 e 1, é possível obter os diferentes valores da curva. A partir do número de *tesselation* introduzido pelo utilizador, sabe-se quantos pontos calcular para cada curva, sendo calculado os valores de t conforme.

Para cada valor de t calculado, é calculado um ponto para cada uma das quatro *Bezier Curves*. Com estes 4 pontos, forma-se outra *Bezier Curve*, onde se vai variar o valor de t de modo a obter novos pontos.

Considerando t como *input* das primeiras curvas e *t2* como *input* das segundas curvas, temos que, para cada par de valores t, se obtêm duas curvas novas e para cada par de valores *t2* dessas novas curvas, se obtêm quatro pontos. Estes quatro pontos são os pontos usados para desenhar os triângulos.

7.3 *Teapot*

Por fim, foram postos em prática os métodos acima explicados, tendo sido criada a figura geométrica de um *teapot*, tal como é demonstrado na figura 19.

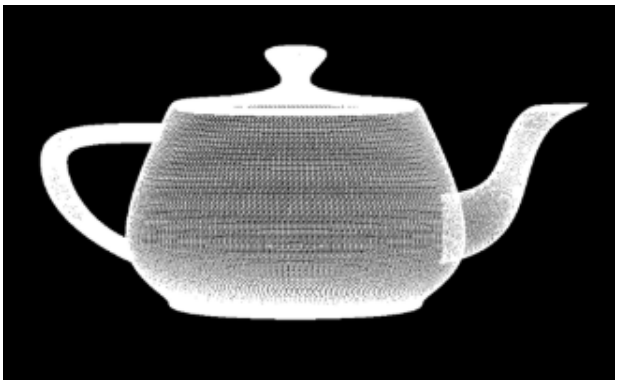

Figura 19: Bezier Curves - Teapot.

8 *Catmull-Rom Curves*

Nesta fase foi implementado outro tipo de translação. Este novo tipo de translação é calculado em função do tempo e apartir de *Catmull-Rom Curves*. As *Catmull-Rom Curves* são curvas que necessitam de 4 pontos para poderem ser definidas.

Internamente, as *Catmull-Rom Curves* são definidas tendo em conta *Hermit Curves*. As *Hermit Curves* são curvas definidas por 2 pontos. Considerando então 4 pontos definidos no espaço: P0, P1, P2 e P3, temos que a *Catmull-Rom Curve* definida por estes 4 pontos é na verdade a junção de 3 *Hermit Curves*, cada uma formada por um par de entre estes pontos. As *Hermit Curves* são então definidas pelos pares: P0 e P1, P1 e P2, P2 e P3. Juntando sempre o final de cada *Hermit Curve* com o início da próxima, obtemos a *Catmull-Rom Curve*.

Na prática, a implementação destas curvas é muito mais simples do que na teoria. Não é necessário preocupar com as *Hermit Curves* e a sua junção. Existem fórmulas e equações, que dados 4 quaisquer pontos no espaço, definem a *Catmull-Rom Curve* definida por estes pontos. Uma destas fórmulas é apresentada sob a forma de multiplicação de duas matrizes, apresentadas a seguir:

$$T = \begin{bmatrix} \mathrm{t}^3 & \mathrm{t}^2 & t & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Substituindo pela primeira matriz o valor de tempo pretendido, podemos calcular o valor da curva multiplicando a matriz resultante pelos 4 pontos que definem a curva.

9 Pontos Complementares

De modo a ser possível implementar as componentes relativas à luminosidade, ao material e à textura, foi necessário gerar pontos auxiliares. Assim, procedeu-se à elaboração de normais, para possibilitar a luminosidade e o material, e pontos de textura, para possibilitar a aplicação da mesma.

9.1 Geração de Normais

Um vetor normal é um vetor normalizado (norma 1) perpendicular a uma dada superfície num ponto.

9.1.1 Plano

Num plano, todos os pontos terão a eles associados o mesmo vetor normal, pois estão todos "assentes"na mesma superfície reta. Temos então, que para um plano assente num dos eixos, o vetor normal terá a direção do eixo cujas coordenadas não variam nesse plano.

- Se for um plano assente em xz, y não varia, logo o vetor normal será (0,1,0) ou (0,-1,0).
- Se for um plano assente em xy, z não varia, logo o vetor normal será (0,0,1) ou (0,0,-1).
- Se for um plano assente em yz, x não varia, logo o vetor normal será (1,0,0) ou (-1,0,0).

9.1.2 Caixa

Na caixa a lógica é a mesma. Como a caixa é composta por 6 planos, a cada ponto associa-se a normal tendo em conta o plano em que está assente.

No entanto, os vértices da caixa são pontos que pertencem a 3 planos. Estes pontos vão ter 3 normais associadas a eles, sendo que, quando se desenha uma face, é usada a normal correspondente ao plano desse face.

9.1.3 Cone

Para um cone, é necessário dividir os pontos em 2 categorias: os da base e o da superfície lateral.

Os da base, uma vez que se encontram assentes num plano, as normais serão iguais, (0,-1,0) em todos os pontos da mesma e escolhidos a maneira já foi explicada no plano.

Nos pontos da superfície do cone, é necessário calcular o ângulo que um dado ponto faz com a base do cone. Para tal, usamos a expressão arctangente (altura /raio). Após termos este ângulo, é facilmente calculado as coordenadas do vetor normal, usando coordenadas polares.

9.1.4 Esfera

Para a esfera, como esta se encontra sempre centrada na origem, para calcular as normais destas apenas é preciso normalizar as coordenadas dos pontos.

9.2 Geração de Pontos de Textura

A implementação de pontos de textura consiste em associar a cada vértice da figura geométrica um ponto da figura que pretendemos aplicar.

Para fazer esta conexão, os pontos de textura vão ser constituídos por duas coordenadas cujos valores variam entre 0 e 1, ou seja, os pontos vão pertencer à grelha representada na figura 20. Os valores da grelha serão posteriormente ajustados à figura pretendida, adaptando também os pontos de textura gerados.

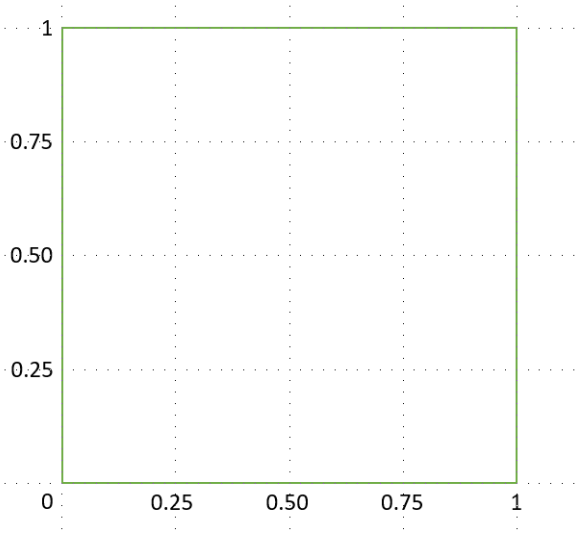

Figura 20: Grelha dos pontos de textura.

9.2.1 Plano

A associação do plano à grelha é bastante simples uma vez que o plano consiste em apenas uma superfície com 4 vértices.

Com efeito, tal como é possível observar na figura 21, não foi necessário fazer qualquer divisão da área. Assim, cada vértice do plano foi respetivamente associado a cada um dos vértices da grelha.

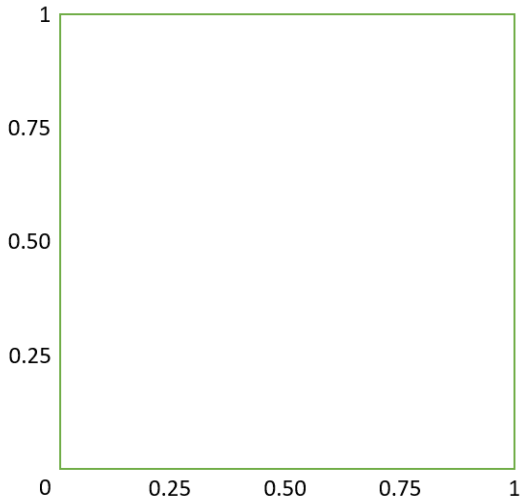

Figura 21: Superfícies - Plano.

9.2.2 Caixa

A associação da caixa à grelha é mais complicada uma vez que esta tem 6 superfícies.
Com efeito, foi necessário dividir a área em 6 partes nos seguintes intervalos:

- **base**: [0..1][0..0,25]
- **faces**: [0..1][0,25..0,5]
  - **face esquerda**: [0..a][0,25..0,5]
  - **face frente**: [a..b][0,25..0,5]
  - **face direita**: [b..c][0,25..0,5]
  - **face trás**: [(c..1][0,25..0,5]
- **topo**: [0..1][0,5..1]

Tal como acima indicado, foi dedicado uma área fixa para a face superior, uma para a face inferior e uma para as faces laterais.

entanto, foi concluído que a área correspondente a cada face lateral deverá depender das dimensões do cubo.

Efetivamente, existem então dois comprimentos, x e y, sendo x o tamanho dedicado às faces esquerda e direita e y o tamanho dedicado às faces frente e trás. Sendo assim, temos que os valores a, b e c referidos na divisão da grelha, são atribuídos da seguinte maneira:

$$a = x, b = x + y, c = x + y + x \tag{1}$$

Assim, é possível observar na figura 22 como a grelha seria distribuída caso as faces laterais da caixa tenham todas as mesmas dimensões: grelha com todas as faces laterais com 0.25 de largura .

Do mesmo modo, está demonstrado na figura 23 um exemplo de uma caixa cujas faces laterais não têm todas as mesmas dimensões: grelha com as faces laterais com as larguras x e y previamente explicadas.

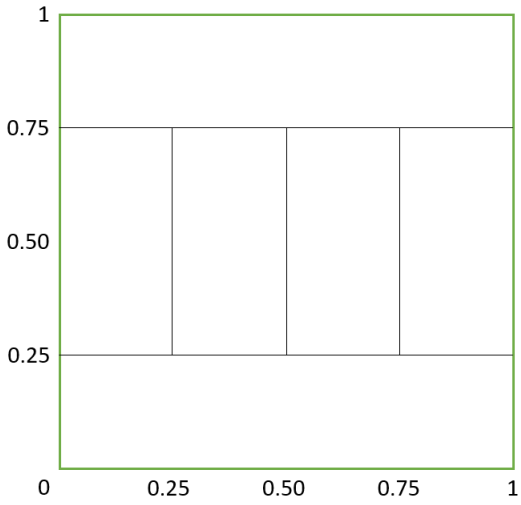

Figura 22: Superfícies - Caixa Proporcional.

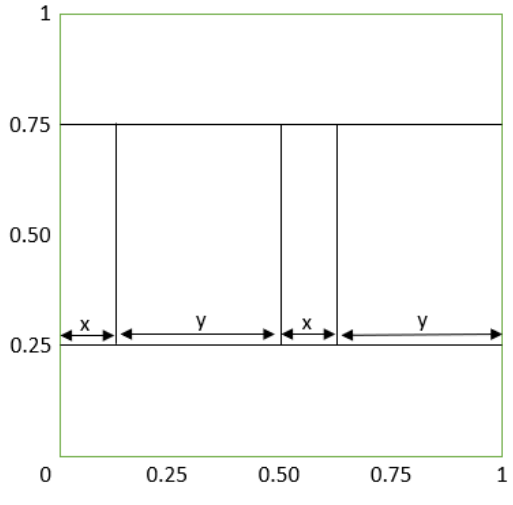

Figura 23: Superfícies - Caixa Desproporcional.

Por fim, é de ter em conta que a caixa pode ser implementada com divisões. Consequentemente, é necessário permitir a divisão de ambas as arestas de cada área/face x vezes, sendo este x o número de divisões pretendidas.Os pontos destas divisões nas grelhas foram implementados através do cálculo de cada pedaço de aresta.

Observando a figura 24, que representa a caixa exemplo da figura 22 com uma divisão, constata-se que as distâncias assinaladas na figura foram calculadas através das seguintes equações:

$$x = \frac{1}{nr\_divisions}, y = \frac{0.25}{nr\_divisions}, w = \frac{0.25}{nr\_divisions}, z = \frac{0.5}{nr\_divisions} \tag{2}$$

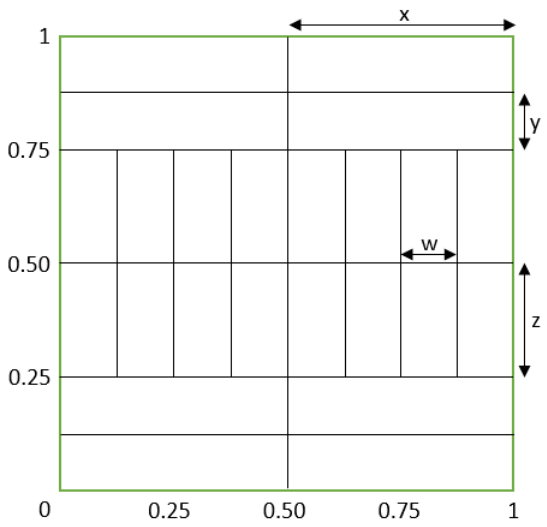

Figura 24: Divisões das Superfícies - Caixa Proporcional.

9.2.3 Esfera

A esfera, tal como o plano, é constituída por apenas uma superfície, portanto a sua área, tal como observado na figura 25, não foi dividida.

No entanto, no desenho da esfera, esta vai ser constituída por várias divisões, as *slices* e as *stacks*. Assim, vai ser necessário criar repartições na grelha para associar aos vértices formados por estas divisões.

Observando a figura 26, pode ser constatado a que a grelha foi repartida de maneira uniforme, sendo as distâncias entre pontos calculada da seguinte maneira:

$$x = \frac{1}{nr\_slices}, y = \frac{1}{nr\_stacks} \tag{3}$$

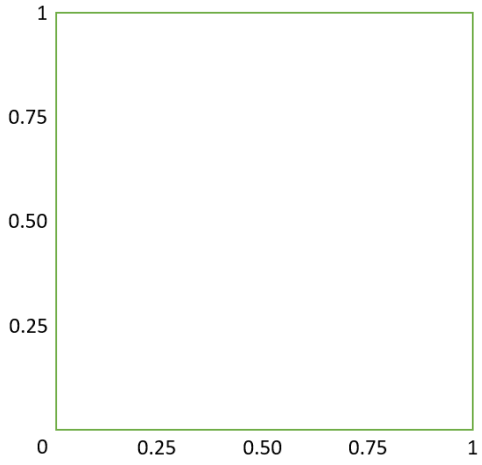

Figura 25: Superfícies - Esfera

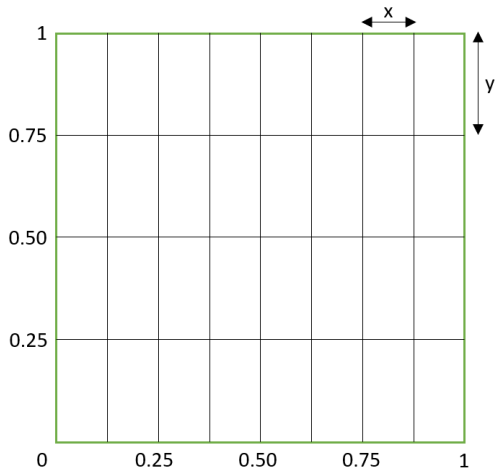

Figura 26: Divisões das Superfícies - Esfera

9.2.4 Cone

Por sua vez, o cone é constituído por duas superfícies, a base e a superfície lateral. Tendo isto em conta, foi repartida a área da grelha nos seguintes intervalos:

- **base**: [0..1][0..0,25]
- **superfície lateral**: [0..1][0,25..1]

Para além disso, a superfície lateral do cone vai ser implementada com as *slices* e as *stacks* que, tal como na esfera, vão criar vértices que são necessários associar na grelha. Assim, tal como demonstrado na figura 28, a superfície lateral vai ser repartida de maneira uniforme com as seguintes distâncias:

$$x = \frac{1}{nr\_slices}, y = \frac{0.75}{nr\_stacks}, \tag{4}$$

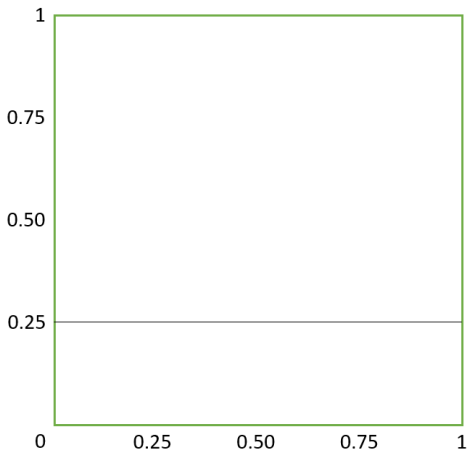

Figura 27: Superfícies - Cone

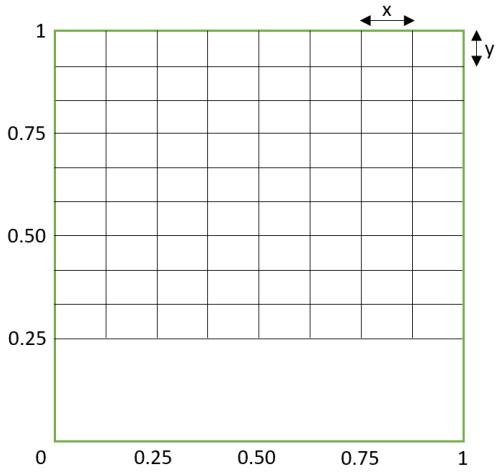

Figura 28: Divisões das Superfícies - Cone

9.3 Organização do Ficheiro

Para guardar estas informações, o formato do ficheiro teve de ser alterado uma vez que este possuía apenas as coordenadas dos pontos necessárias para formar desenhar os triângulos.

Estes novos elementos passam a estar escritos no ficheiro da seguinte maneira:

1. linha de coordenadas de 3/4 pontos da figura geométrica
2. linha das coordenadas das normais correspondentes aos pontos da primeira linha
3. linha das coordenadas dos pontos de textura correspondentes aos pontos da primeira linha
4. repetição das linhas 1, 2 e 3 até os pontos estarem todos registados

No caso do ficheiro já demonstrado na figura 17, este passa para o formato representado na figura 29, onde as linhas sublinhadas representam as linhas das normais e das texturas relativas à primeira linha.

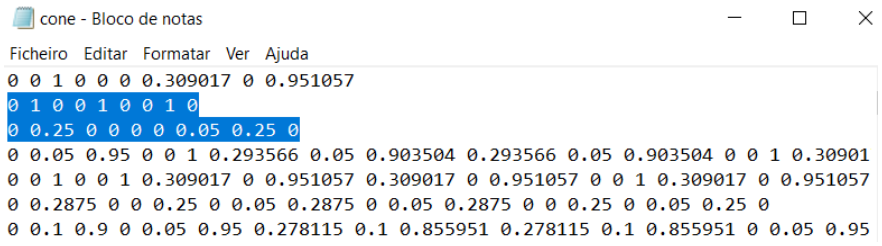

Figura 29: Excerto do ficheiro de pontos **cone.3d** - Pontos complementares.

9.4 Implementação

Estes pontos foram separados num vetor de normais e num vetor de texturas. Assim, estes vão ser implementados pelo método dos VBOs.

Tal como descrito na secção 6, estes são gerados através da função **glGenBuffers**, sendo de seguida ativados com a função **glBindBuffer** e preenchidos pela função **glBufferData**.

No entanto, na parte de desenhar os VBOs, surgiu a questão que, devido aos pontos de texturas possuírem apenas duas coordenadas e o de pontos e de normais possuírem três, torna-se impossível usar o vetor de índices criado para os pontos. Devido a esta adversidade, foi necessário reverter a redução de pontos previamente explicada na secção 6.1 de maneira a se desenhar os VBOs sem vetor de índices.

Com efeito, foi necessário alterar a parte do desenho dos VBOs. Enquanto que previamente (secção 6.2) se usava a função **glDrawElements**, agora vai ser usada a função **glDrawArrays**. Tal como anteriormente (secção 6.2), o desenho das VBOs continua a ser implementado entre as funções **glPushMatrix** e **glPopMatrix**.

10 *Lighting*

Foi implementado, através da função **glLightfv**, os componentes das várias luzes que vão incidir nas primitivas gráficas.

Existem 3 tipos de luzes: **directional light, point light** e **spotlight**. Podem ser executadas várias luzes da cada tipo desde que o número total de luzes seja igual ou menor que 8.

10.1 *Directional Light*

A *directional light* representa uma luz vinda de todos os pontos de uma determinada direção.
Esta luz tem os seguintes componentes com os seguintes respetivos parâmetros:

- direção
  - dirX, dirY, dirZ

10.2 *Point Light*

A *point light* é uma luz tipo lâmpada que apartir de um determinado ponto ilumina o que está à sua volta.
Esta luz tem os seguintes componentes com os seguintes respetivos parâmetros:

- posição
  - posX, posY, posZ
- atenuação
  - at

10.3 *Spotlight*

A *spotlight* é uma luz numa determinada posiçõa que ilumina numa direção e afeta uma área com a forma de um cone.
Esta luz tem os seguintes componentes com os seguintes respetivos parâmetros:

- posição
  - posX, posY, posZ
- direção
  - dirX, dirY, dirZ
- atenuação
  - at
- expoente
  - exp
- *cutoff*
  - cut

11 Cor/Material

Foi implementado, através da função **glMaterialfv**, os componentes do material que vão dar à primitiva gráfica uma aparência sólida 3D.
O modelo de cor utilizdo é o **RGBA** que, em que cada componente, tem um valor no intervalo [0.0, 1.0].

- **Difusa** - simula a iluminação causada por uma luz no objeto
  - Tem como *standard* os valores:
    (0.8f, 0.8f, 0.8f, 0.1f)

- **Especular** - simula o brilho produzido pela luz no objeto
  - Tem como *standard* os valores:
    (0.0f, 0.0f, 0.0f, 0.1f)
- **Ambiente** - simula a luz ambiente que impacta todo o objeto
  - Na vida real, mesmo quando está escuro, o normal é não estar escuro o suficiente para não se conseguir ver os objetos, logo esta luz tem como *standard* os valores:
    (0.2f, 0.2f, 0.2f, 0.1f)
- **Emissiva** - simula o aparência de emissão de luz, como o efeito de uma lâmpada
  - Tem como *standard* os valores:
    (0.f, 0.0f, 0.0f, 0.1f)

12 Keys

De modo a facilitar a visualização das figuras geométricas, foram implementadas determinadas *keys* conectadas com o teclado e o rato.

12.1 Teclado

Quanto ao teclado, é possível fazer os seguintes comandos:

- $+$ - aproxima a câmara das figuras geométricas
- **-** - afasta a câmara das figuras geométricas
- $\uparrow$ - desloca a câmara para cima
- $\downarrow$ - desloca a câmara para baixo
- $\rightarrow$ - desloca a câmara para a direita
- $\leftarrow$ - desloca a câmara para a esquerda
- **F1** - desloca a câmara para a frente
- **F2** - desloca a câmara para a trás
- **F3** - coloca a câmara na posição inicial
- **F4** - foca a câmara numa primitiva gráfica: à medida que se vai usando a *key* esta vai focando nas diferentes primitivas gráficas

12.2 Rato

Quanto ao rato, é possível fazer os seguintes comandos:

- ***Mouse Wheel*** - aproxima ou distancia a câmara das figuras geométricas
- ***Left Click; Right Click*** - aciona a captação do movimento do rato
  - ***Mouse Movement*** - desloca a câmara num sentido rotativo conforme o movimento

13 *Following*

Tal como foi anteriormente referido, foi implementado a *key* **F4** que altera a câmara conforme a posição das primitivas gráficas.

Esta *key* vai coloca a câmara a sempre à mesma distância da primitiva gráfica com esta centrada no espaço câmara, independentemente do movimento da mesma.

Se a *key* for usada, a câmara vai passar a focar na primeira primitiva que foi desenhada, movendo-se com a mesma de maneira a que esta permaneça no centro do espaço câmara. Se a *key* for utilizada novamente, a câmara salta para a próxima primitiva gráfica, comportando-se da maneira descrita. Este processo é repetido sempre que se usa a *key* percorrendo as primitivas gráficas todas até voltar à *normal view*. Nesta situação, a câmara não segue nenhuma primitiva gráfica, reiniciando o *loop* da *key*.

De modo a se saber em que modo a câmara está, foi implementado, através da função **glutSetWindowTitle**, que no título da janela se observasse as frases *"Normal View"*, tal como observado na figura 30, e *"Follow figure X"*, sendo *X* um número atribuído à primitiva que se encontra a ser seguida, tal como é possível observar na figura 31.

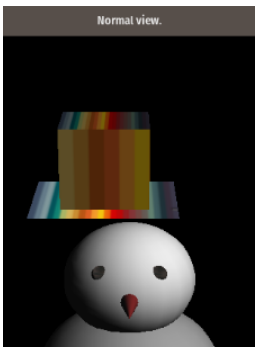

Figura 30: *Following*: Vista normal.

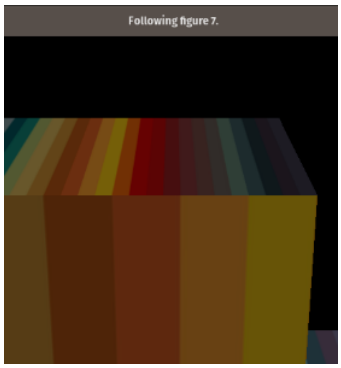

Figura 31: *Following*: Primitiva gráfica 7.

14 Sistema Solar XML

Após ter sido definida a metodologia para desenvolvimento de uma cadeia de figuras que partilham propriedades com um sistema de hereditariedade, foram então criados dois ficheiros para representar o sistema solar.

14.1 Estático

Para esta simulação foi utilizada a transformação geométrica **scale** para obter os diferentes tamanhos dos planetas e luas em relação ao sol, assim como a transformação geométrica **translate** mais simples para distanciar os astros uns dos outros.

Foi desenvolvido o ficheiro **sistemasolar.xml** com as características mencionadas de modo a realizar uma simulação estática do sistema solar.

Para a execução deste ficheiro, foi usado como suporte os ficheiros criados da seguinte maneira:

- **sun.3d**

  ./generator sphere 9 50 50 sun.3d

- **planet.3d**

  ./generator sphere 6 30 30 planet.3d

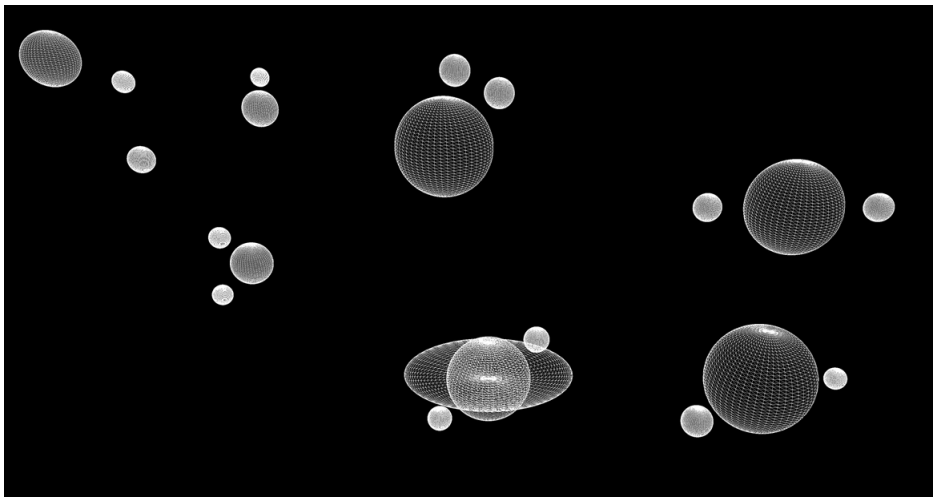

Figura 32: Sistema Solar - estático.

14.2 Com movimento

O **translate**, que permite definir uma rota e o tempo de percurso da mesma, e o **rotate**, que permite uma rotação permanente com um determinado tempo, é possível realizar uma representação do sistema solar muito mais exata na simulação estática.

Considerando isto, foi criado o ficheiro **fun-sistemasolar.xml** no qual se tentou fazer uma simulação mais correta usando o **scale** e as transformações geométricas mencionadas. Para a execução deste ficheiro, foi usado como suporte o ficheiro criado da seguinte maneira:

- **planet.3d**

  ./generator sphere 6 30 30 planet.3d

- **teapot.3d**

  ./generator patch teapot.patch 20 teapot.3d

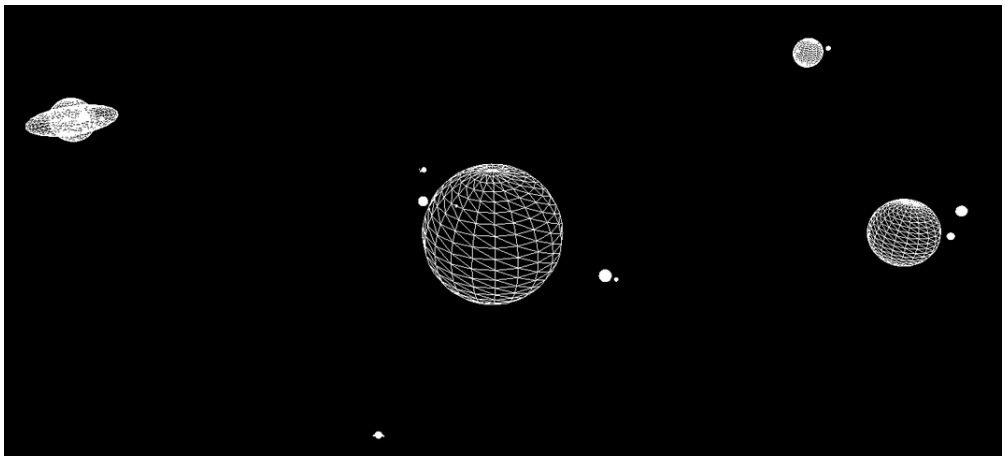

Figura 33: Sistema Solar - com movimento.

14.3 Com luminosidade, cor e textura

Com a implementação de texturas, luzes e material, é possível fazer uma representação do sistema solar mais semelhante à realidade. Com isto em vista, copiou-se o conteúdo do ficheiro **fun-sistemasolar.xml** e apartir dessa informação desenvolveu-se o ficheiro **luz-fun-sistemasolar.xml**, ao qual foram adicionados texturas, materiais e luzes.

Com efeito, para personalizar o sistema solar, obteve-se as texturas do sol, das luas e de cada um dos planetas e adicionou-se ao XML de modo a simular a realidade. Para além disso, foram implementados elementos de material às várias primitivas gráficas. Por fim, foi implementado um *point light* no local onde se encontra representado o sol, de maneira a simular a luz do mesmo.

No fim deste processo, foi obtido um resultado semelhante à realidade, tal como representado na figura 34.

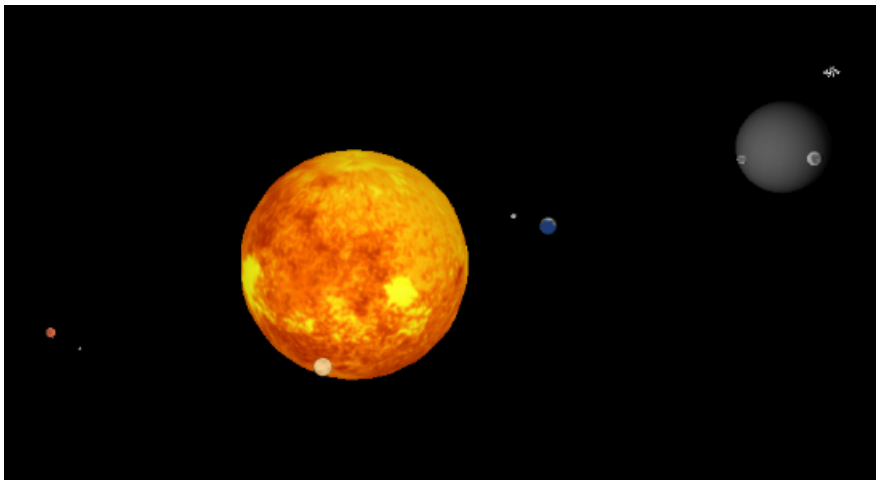

Figura 34: Sistema Solar - com movimento, luminosidade, cor e textura.

15 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.

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 sphere.3d

- **cone.3d**

  ./generator cone 1 1 20 20 cone.3d

- **plane.3d**

  ./generator plane 1 plane.3d

- **box.3d**

  ./generator box 1 1 1 0 box.3d

- **box4.3d**

  ./generator box 10 2 1 4 box4.3d

15.1 Castelo

Ficheiros XML e comandos utilizado para obter as figuras 35 (estática), 36 (com movimento), 37 (estático, com luminosidade *pointlight*, cor e textura) e 38 (com movimento, luminosidade *pointlight* e cor), respetivamente:

- **castle.xml**

  ./engine castle.xml

- **fun-castle.xml**

  ./engine fun-castle.xml

- **luz-castle.xml**

  ./engine luz-castle.xml

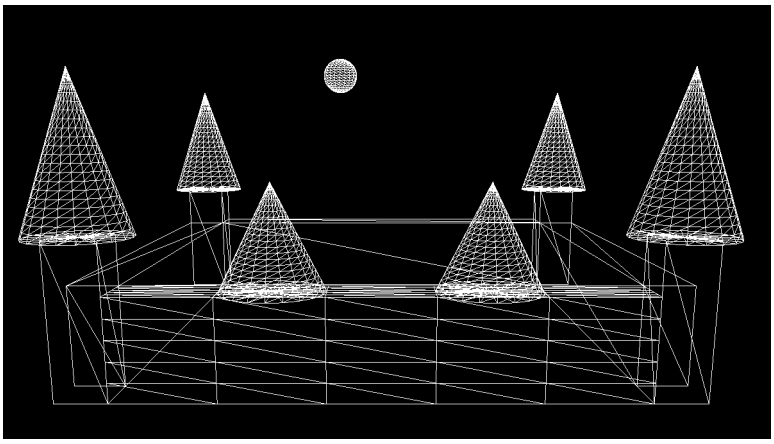

Figura 35: Castelo - estático.

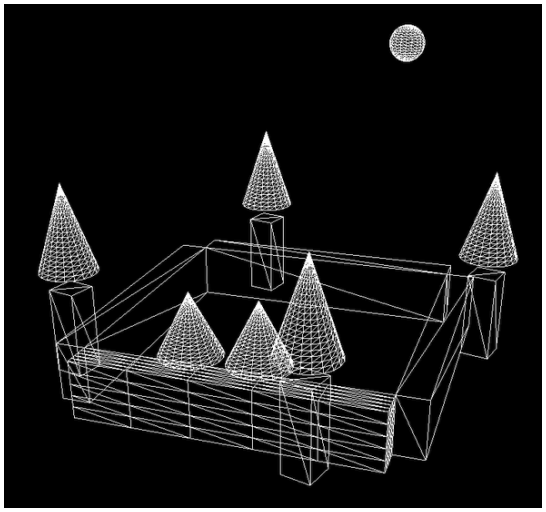

Figura 36: Castelo - movimento.

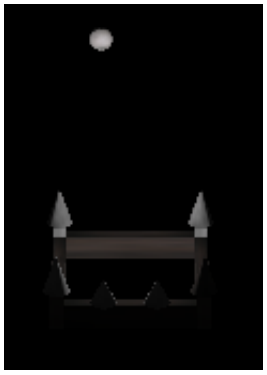

Figura 37: Castelo - estático, com luminosidade, cor e textura.

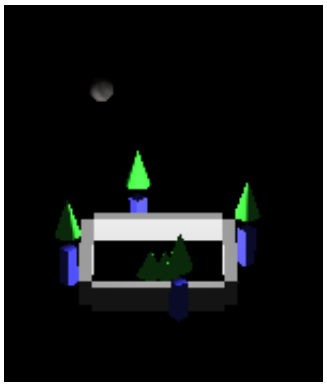

Figura 38: Castelo - com movimento, luminosidade e cor.

15.2 Boneco de Neve

Ficheiros XML e comandos utilizado para obter as figuras 39 (estática) e 40 (com movimento), 41 (estático, com luminosidade direcional, cor e textura) e 42 (com movimento, luminosidade direcional, cor e textura), respetivamente:

- **snowman.xml**

  ./engine snowman.xml

- **fun-snowman.xml**

  ./engine fun-snowman.xml

- **luz-snowman.xml**

  ./engine luz-snowman.xml

- **luz-fun-snowman.xml**

  ./engine luz-fun-snowman.xml

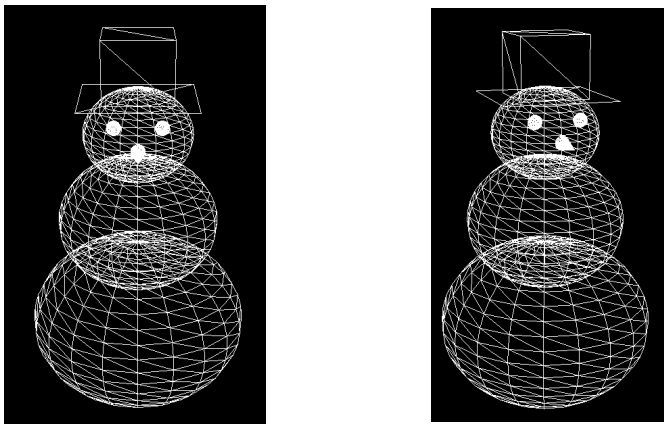

Figura 39: Boneco de Neve - estático.

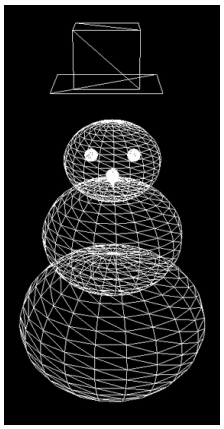

Figura 40: Boneco de Neve - com movimento.

Figura 41: Boneco de Neve - estático, com luminosidade, cor e textura.

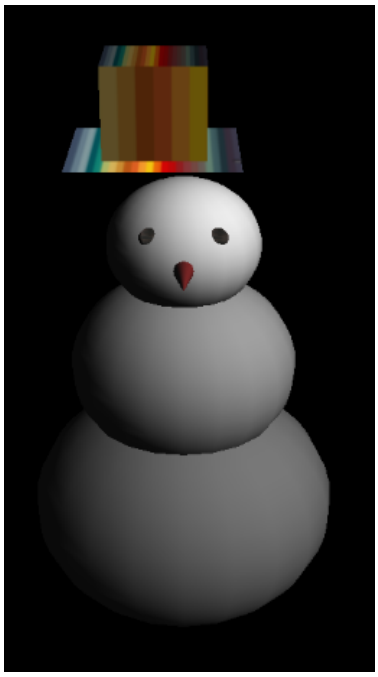

Figura 42: Boneco de Neve - com movimento, luminosidade, cor e textura.

15.3 Gelado

Ficheiros XML e comandos utilizado para obter as figuras 43 (estático) e 44 (com movimento), 45 (estático com luminosidade *spotlight*, cor e textura) e 46 (com movimento, luminosidade *spotlight*, cor e textura), respetivamente:

- **icecream.xml**

  ./engine icecream.xml

- **fun-icecream.xml**

  ./engine fun-icecream.xml

- **luz-icecream.xml**

  ./engine luz-icecream.xml

- **luz-fun-icecream.xml**

  ./engine luz-fun-icecream.xml

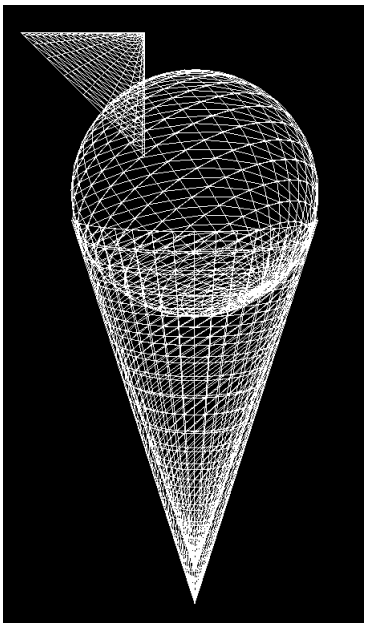

Figura 43: Gelado - estático.

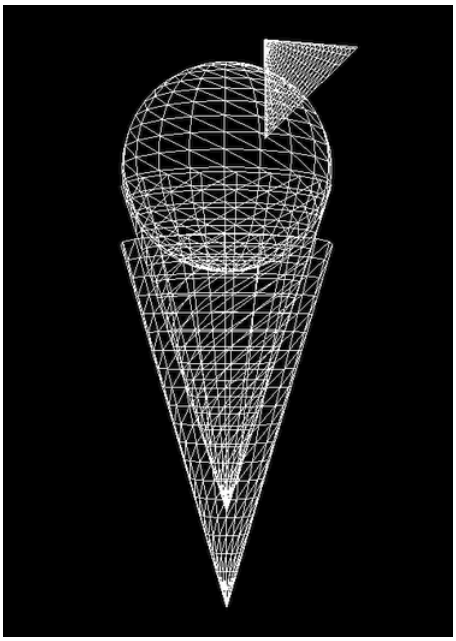

Figura 44: Gelado - com movimento.

Figura 45: Gelado - estático, com luminosidade, cor e textura.

Figura 46: Gelado - com movimento, luminosidade, cor e textura.

16 Conclusão

Para a quarta fase deste projeto, introduziu-se uma nova biblioteca: **Devil**. Através da utilização da nova biblioteca foi possível melhorar graficamente a versão anterior, dando a esta um aspeto mais realista.

Ao longo deste trabalho foi então possibilitada a adição de texturas, luzes e personalização de materiais.

Recorrendo às novas funcionalidades foi ainda possível criar exemplos XML que demonstram a diversidade das mesmas, retratando possíveis figuras obtidas pelo programa.

A maior adversidade encontrada pelo grupo terão sido as localizações das luzes. Não se conseguiu que estas ficassem fixas no espaço global, tendo assim ficado fixas no espaço câmara, movendo-se com a mesma.

Tirando este pormenor, foi realizada uma boa implementação dos vários requerimentos.