Universidade do Minho
Escola de Engenharia

Fase I
Primitivas Gráficas

Trabalho Prático
Computação Gráfica

Grupo 48

Gabriela Santos Ferreira da Cunha - a97393

João Gonçalo de Faria Melo - a95085

Nuno Guilherme Cruz Varela - a96455

a97393

a95085

a96455

março, 2023

Conteúdo

1 Introdução

No âmbito deste projeto, foi-nos proposto o desenvolvimento de um mecanismo 3D baseado num cenário gráfico, dividido em 4 fases.

O presente relatório é referente à primeira fase do trabalho, onde nos foi proposta a implementação de 2 aplicações: um *generator*, capaz de gerar ficheiros com a informação necessária ao modelo e um *engine*, responsável por ler ficheiros de configuração, no formato XML, e expor os modelos pretendidos para visualização.

2 Arquitetura

A arquitetura seguida pelo grupo pode ser representada pela figura seguinte.

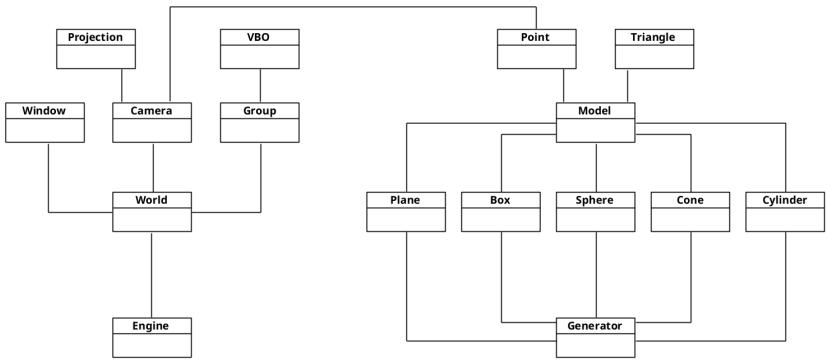


Figura 1: Arquitetura do sistema.

De modo a gerar a informação necessária, o *generator* recorre aos módulos responsáveis por gerar o modelo de cada uma das primitivas gráficas implementadas, que se encontram na pasta *shapes*. Por sua vez, cada um destes módulos devolve um "Model" correspondente ao pedido, que é um conjunto de pontos e triângulos. Em relação ao *engine*, este lê o ficheiro de configuração e regista a informação com recurso ao módulo "World", que contém as configurações gráficas necessárias para a representação das primitivas escolhidas.

2.1 Classes

2.1.1 *Point*

Para o desenvolvimento deste mecanismo 3D é necessário utilizar posições no espaço. Desta forma, utilizamos a unidade básica, o ponto, para indicar estas posições. Criamos, então, uma classe *Point* que guarda três coordenadas, em relação aos eixos x, y e z.

2.1.2 *Triangle*

De modo a evitar a repetição de vértices, que será abordada posteriormente, esta classe representa um triângulo que contém as posições no vetor de pontos, criado ao gerar a primitiva gráfica, de cada ponto que o forma.

2.1.3 *Model*

Com vista a representar as primitivas gráficas que pretendemos gerar, decidimos criar uma classe *Model* que contém 2 estruturas: uma que guarda todos os vértices da primitiva e outra referente aos triângulos que a vão compor.

2.1.4 VBO

De modo a aumentar a eficiência do nosso *engine*, recorremos a *Vertex Buffer Objects* e, portanto, criamos uma classe para representar um VBO. Esta classe é responsável por guardar dois *buffers* de um certo modelo: um para guardar os vértices e outro para guardar os índices dos vértices para representar os triângulos.

2.1.5 *World*

O *engine* é responsável por ler toda a configuração proveniente do ficheiro XML. Toda a informação deste ficheiro vai ser guardada num objeto "World" que contém todas as configurações relativas à janela, câmara e grupos.

2.2 Ferramentas utilizadas

2.2.1 TinyXML2

Para além da bilbioteca OpenGL, utilizada para implementar as funcionalidades gráficas do projeto, fizemos uso da biblioteca TinyXML2, de modo a simplificar o processo de *parsing* do ficheiro que contém as configurações gráficas e a indicação dos ficheiros previamente gerados que serão carregados e renderizados.

3 Aplicações

3.1 *Generator*

Como referido anteriormente, o *generator* deve gerar os ficheiros que contêm a informação sobre as primitivas gráficas a desenhar, isto é, neste caso, o conjunto de vértices que as compõem, conforme os parâmetros recebidos.

3.1.1 *Parsing* dos argumentos

Num primeiro momento, é realizado o *parse* do *input* com recurso a expressões regulares, com vista a verificar se é possível gerar o sólido pedido e se foram fornecidos todos os parâmetros necessários à criação do mesmo.

3.1.2 Geração do modelo

Em seguida, são calculados os vértices da primitiva gráfica escolhida, fase esta que foi alvo de principal foco em relação a esta aplicação, visto que procuramos fornecer ao *engine* um modelo eficiente. Desta forma, o desenvolvimento desta tarefa estendeu-se em 3 abordagens diferentes.

Inicialmente, a nossa aplicação gerava apenas os pontos de todos os triângulos presentes na primitiva a ser construída. No entanto, esta abordagem acabava por ser pouco eficiente, uma vez que, ao serem gerados todos os pontos, alguns deles acabavam por ser repetidos, devido ao facto de um mesmo ponto poder estar presente em vários triângulos. Isto levava a uma grande utilização de memória, pelo que não nos pareceu a solução ideal.

Em seguida, pensamos numa solução com 2 estruturas: uma com todos os vértices da primitiva, sem repetições, e outra com os triângulos a conter os índices, em relação à primeira estrutura, dos pontos que os formam. Em contrapartida, os acessos à memória desta solução não tiram partido da localidade espacial, uma vez que não repetimos vértices.

Por último, optamos por uma abordagem que resulta num equilíbrio das anteriores, isto é, repetir vértices moderadamente, com vista a garantir a localidade espacial. Esta foi a abordagem seguida pelo grupo, sendo a mais viável e eficiente das apresentadas. Assim sendo, após o cálculo destes vértices e triângulos, é devolvido um objeto "Model".

3.1.3 Estrutura do ficheiro gerado

O *generator* escreve toda a informação do modelo gráfico num ficheiro com extensão "3d". Como podemos observar pela figura 2, utilizamos a primeira linha para indicar o número de vértices e de triângulos. As linhas seguintes contêm as coordenadas dos vértices e, por último, os triângulos com os respetivos índices.

```
4 2
0.500000 0.000000 0.500000
0.500000 0.000000 -0.500000
-0.500000 0.000000 -0.500000
-0.500000 0.000000 0.500000
0 1 2
0 2 3
```

Figura 2: Estrutura do ficheiro gerado.

3.2 *Engine*

A aplicação *engine* recebe um ficheiro de configuração, em formato XML, e interpreta-o. Esta começa por fazer o *parse* da informação, com o auxílio da biblioteca TinyXML2, como já foi referido, e coloca-a num objeto "World". O *engine* é responsável por, a partir deste objeto, ajustar as definições da câmara, definir a janela de exibição e representar os resultados (primitivas gráficas) dos ficheiros relativos aos modelos 3D gerados previamente pelo *generator*.

No contexto desta aplicação, preocupamo-nos igualmente em garantir uma boa *performance*. Deste jeito, utilizamos *Vertex Buffer Objects* (VBOs), que consiste num *buffer* armazenado na memória da placa gráfica, de modo a que o modelo possa ser diretamente renderizado pela mesma, evitando que os dados dos vértices sejam constantemente transferidos do CPU para o GPU, algo que acontece no modelo de renderização primário, onde a informação está guardada na memória do sistema. Estes *buffers* são criados e preenchidos no momento da leitura do ficheiro do modelo, sendo que são gerados 2 *buffers* - um para os vértices e outro para os índices - para cada modelo.

De modo a proporcionar uma melhor visualização de cada primitiva gráfica, o grupo implementou o modo explorador da câmara. Desta forma, é possível observar as diferentes faces das várias primitivas gráficas, o que facilitou também no processo de *debug*, na medida em que é possível visualizá-las de diferentes ângulos e circular à volta das mesmas, contribuíndo para a verificação da obediência à regra da mão direita.

Assim, para ativar ou desativar o modo explorador, o utilizador deverá premir a tecla "c", sendo que o modo não está ativo por predefinição. Para diminuir e aumentar o ângulo vertical e horizontal da câmara, são utilizadas as "arrows". Para aproximar ou afastar a câmara do objeto, são utilizadas as teclas "+" e "-", respetivamente.

Outra funcionalidade implementada foi a contagem do número de *frames* por segundo, o que facilitou a comparação das diferentes estratégias de geração de modelos apresentadas anteriormente.

4 Primitivas Gráficas

Para esta fase, as primitivas geométricas implementadas foram o plano, a caixa, a esfera, o cone e o cilindro.

4.1 Plano

O plano desenvolvido está contido no plano $x0z$ e centrado na origem e tem a si associado um determinado valor de comprimento de lado e um certo número de divisões ao longo de cada eixo.

De modo a representar graficamente este comportamento, começamos por definir o ponto por onde iríamos iniciar a construção do plano e, como tal, optamos pela extremidade do plano que se situa no primeiro octante (x e z positivos). Admitindo que o plano tem comprimento l e n divisões, o ponto inicial será da forma $P_i(\frac{l}{2}, 0, \frac{l}{2})$. Os restantes pontos para formar o plano são calculados através de dois ciclos aninhados que vão percorrer o eixo do x e do z. Cada um destes ciclos tem n iterações, que corresponde ao número de divisões.

Através deste raciocínio, derivamos uma fórmula geral para todos os pontos do plano:

$$P\ (initialX - j \times squareLength,\ 0,\ initialZ - i \times squareLength),$$

onde $initialX = \frac{l}{2}$, $initialZ = \frac{l}{2}$, $squareLength = \frac{l}{n}$, i e j correspondem às iterações dos ciclos pelo eixo z e x respetivamente.

A figura seguinte resume o processo acima explicado, para um plano com 3 divisões. Como podemos observar, na primeira iteração de ambos os ciclos, são gerados os primeiros 4 pontos, situados mais à direita. Na segunda iteração do ciclo interior, que percorre o eixo x, os pontos de índice 2 e 3 são repetidos e passam a ter índices 4 e 5, respetivamente, de modo a garantir a localidade espacial na construção dos triângulos.

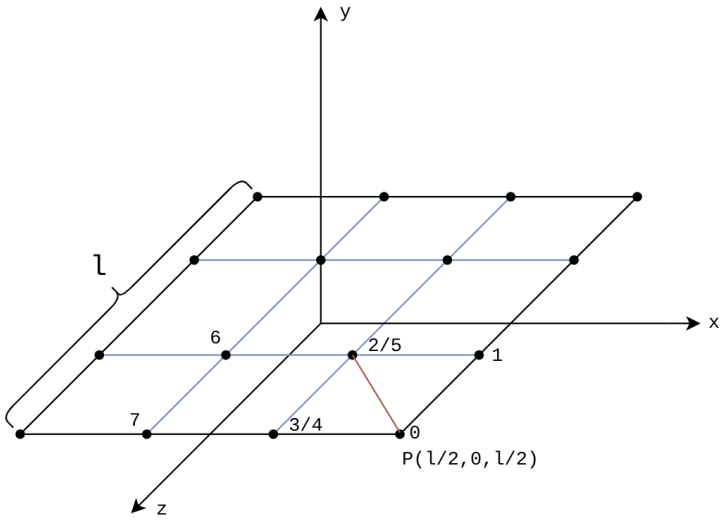


Figura 3: Geração de pontos do plano.

Outra das preocupações a desenhar o plano, era garantir que ele ficasse visível para cima. Para tal, recorremos à regra da mão direita que nos garante que os triângulos ficam visíveis. Utilizamos então o sentido contra os ponteiros do relógio na construção de cada triângulo, isto é, por exemplo, na figura acima, o sentido de indíce 0, 1 e 5.

4.2 Caixa

Uma das primitivas gráficas proposta para desenvolver foi uma caixa, na forma de cubo, centrada na origem com um certo comprimento de lado e número de divisões. Para tal, vimos a necessidade de criar 6 planos, que correspondem às 6 faces, para gerar a caixa.

A estratégia seguida para gerar estes planos é similar ao processo explícito acima para o plano. Como tal, começamos por definir os pontos iniciais por onde iríamos começar construir os planos e através de dois ciclos aninhados percorremos todos os pontos necessários para a sua construção. Assumindo que a caixa tem comprimento l e n divisões, os pontos iniciais utilizados foram os seguintes:

Plano x0z	**Plano y0z**	**Plano x0y**
$P_i(\frac{l}{2}, \frac{l}{2}, \frac{l}{2})$	$P_i(\frac{l}{2}, \frac{l}{2}, \frac{l}{2})$	$P_i(-\frac{l}{2}, -\frac{l}{2}, \frac{l}{2})$
$P_i(\frac{l}{2}, -\frac{l}{2}, \frac{l}{2})$	$P_i(-\frac{l}{2}, \frac{l}{2}, \frac{l}{2})$	$P_i(-\frac{l}{2}, -\frac{l}{2}, -\frac{l}{2})$

Tabela 1: Pontos iniciais da construção dos planos.

Graficamente, os pontos de partida encontram-se representados na figura abaixo.

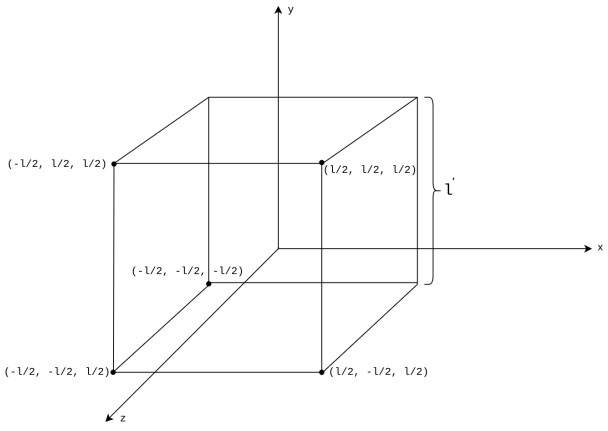


Figura 4: Pontos iniciais para a geração do planos da caixa.

Para gerar planos paralelos aos planos que contêm os eixos, $x0z$, $y0z$ e $x0y$, utilizamos 3 funções que vão receber os pontos iniciais acima definidos. Cada uma destas funções calcula os pontos e os triângulos necessários à construção do respetivo plano.

Também na caixa, alguns vértices dos quadrados são repetidos, tal como no plano, de maneira a garantir uma boa localidade espacial nos triângulos que vão ser criados.

4.3 Esfera

A esfera é um sólido geométrico que pode ser dividido em *slices* e *stacks*, tal como representa a figura 5.

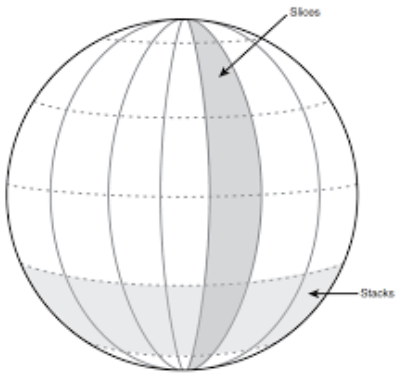


Figura 5: Divisão da esfera.

A estratégia de implementação usada para criar o modelo 3D consiste em gerar uma série de pontos e triângulos para representar a geometria da esfera. À semelhança do plano e da caixa, pretendemos tratar esta estratégia como um processo iterável sobre o número de *slices* e *stacks*.

Assumindo que α é o ângulo segundo os eixos do z e x, β é o ângulo segundo o eixos do y e x e r é o raio da esfera, podemos determinar qualquer ponto presente na superfície esférica:

$$P\ (r \times cos\beta\ \times sin\alpha,\ r \times sin\beta,\ r \times cos\beta\ \times cos\alpha)$$

De modo a calcular os vértices da esfera, são utilizados dois ciclos: um primeiro que vai iterar pelas *slices* e um segundo que itera pelas *stacks*. Para cada *slice* o ângulo α é incrementado em $\frac{2\pi}{slices}$ radianos. Já para as *stacks*, o ângulo β sofre alterações de $\frac{\pi}{stacks}$ radianos. Desta forma, e através da expressão acima definida, conseguimos exprimir todos os vértices necessários para a representação gráfica da esfera.

Em cada iteração de uma *slice* são adicionados ao vetor de vértices todos os vértices dessa mesma *slice*, inclusivamente o ponto superior e inferior da esfera. Isto vai gerar a repetição desses dois pontos e de todos os pontos comuns entre a *slice* atual e a seguinte. No entanto, desta maneira, conseguimos usufruir de uma boa localidade espacial, algo que melhorará a eficiência do *engine*. Para calcular os triângulos, e como percorremos a esfera de baixo para cima, começamos por desenhar o triângulo mais inferior, de seguida os triângulos intermediários e, por último, o triângulo mais superior da esfera.

4.4 Cone

O cone é outra das primitivas gráficas que nos foi proposta a desenvolver. Como tal, terá a sua base no plano $x0z$ e recebe como parâmetros o raio, a altura, o número de *slices* e *stacks*.

A estratégia seguida para desenhar o cone é semelhante à estratégia utilizada na esfera. Assim, utilizamos um ciclo externo para iterar nas *slices* e dentro deste iteramos pelas *stacks*.

A base do cone é uma circunferência, pelo que, para cada *slice*, o ângulo é incrementado em $\frac{2\pi}{slices}$ radianos. Desta forma, todos os pontos da base podem ser definidos da seguinte forma:

$P\ (r \times sin\alpha,\ 0,\ r \times cos\alpha)$, em que r é o raio inicial do cone e α é o ângulo.

Os restantes vértices do cone, ou seja, vértices das outras *stacks*, seguem a seguinte fórmula:

$$P\ (newR \times sin\alpha,\ newH,\ newR \times cos\alpha), \text{ onde}$$
$$newH = stackIndex \times \frac{h}{stacks} \text{ e } newR = r - stackIndex \times \frac{r}{stacks}.$$

Nesta primitiva gráfica, o raio e a altura são aspetos a ter em conta, visto que para cada *stack* teremos de os recalcular. Desta forma, ao longo das iterações nas *stacks*, o raio é calculado em função da *stack* presente $(r - stackIndex \times \frac{r}{stacks})$. Depois disto, podemos então calcular os pontos da seguinte circunferência paralela à base. Para cada *stack*, a altura de cada ponto vai aumentar, podendo ser calculada com a expressão $newH = stackIndex \times \frac{h}{stacks}$. Todas as *stacks* de uma *slice*, à exceção da *stack* do topo, são constituídas por 2 triângulos, pelo que temos a necessidade de os criar.

Através das fórmulas acima descritas, conseguimos calcular todos os pontos de uma *slice* necessários à construção dos triângulos. À semelhança da esfera, os vértices comuns entre *slices* são repetidos, de forma a garantirmos um bom equilíbro entre a repetição de vértices e a localidade espacial.

A seguinte figura contém a representação gráfica de uma *slice* do cone e os respetivos vértices, cujas coordenadas encontram-se na tabela abaixo. Vamos assumir que r é o raio original, i é o índice da *slice*, j é o índice da *stack*, *slices* é o número de *slices*, *stacks* é o número de *stacks* e $\alpha = \frac{2\pi}{slices}$.

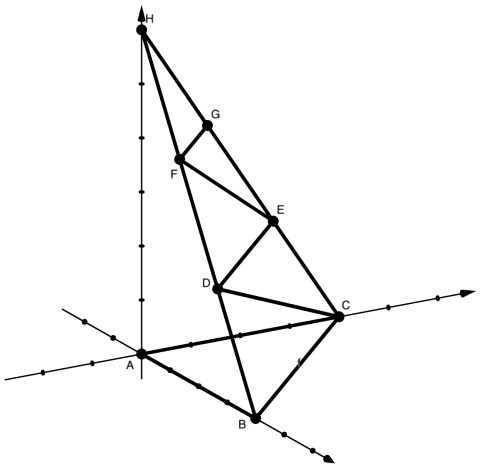


Figura 6: *Slice* de um cone.

Ponto	Coordenadas
A	$(0, 0, 0)$
B	$((r - j \times \frac{r}{stacks}) \times sin(i \times \alpha)\ ,\ j \times \frac{h}{stacks}\ ,\ (r - j \times \frac{r}{stacks}) \times cos(i \times \alpha))$
C	$((r - j \times \frac{r}{stacks}) \times sin((i+1) \times \alpha)\ ,\ j \times \frac{h}{stacks}\ ,\ (r - j \times \frac{r}{stacks}) \times cos((i+1) \times \alpha))$
D	$((r - (j+1) \times \frac{r}{stacks}) \times \sin(i \times \alpha)\ ,\ (j+1) \times \frac{h}{stacks}\ ,\ (r - (j+1) \times \frac{r}{stacks}) \times cos(i \times \alpha)))$
E	$((r - (j+1) \times \frac{r}{stacks}) \times \sin((i+1) \times \alpha)\ ,\ (j+1) \times \frac{h}{stacks}\ ,\ (r - (j+1) \times \frac{r}{stacks}) \times cos((i+1) \times \alpha))$
F	$((r - (j+2) \times \frac{r}{stacks}) \times sin(i \times \alpha)\ ,\ (j+2) \times \frac{h}{stacks}\ ,\ (r - (j+2) \times \frac{r}{stacks}) \times cos(i \times \alpha))$
G	$((r - (j+2) \times \frac{r}{stacks}) \times \sin((i+1) \times \alpha)\ ,\ (j+2) \times \frac{h}{stacks}\ ,\ (r - (j+2) \times \frac{r}{stacks}) \times cos((i+1) \times \alpha))$
H	$(0, h, 0)$

Tabela 2: Coordenadas dos pontos da *slice*.

4.5 Cilindro

O cilindro é uma primitiva gráfica extra, abordada nas aulas práticas, que decidimos adicionar nesta fase. Este vai estar centrado na origem e vai receber como parâmetros o raio, a altura e o número de *slices*.

O cilindro é, novamente, iterado pelas suas *slices* e para cada uma destas calculamos 6 vértices que vão dar origem a 4 triângulos. A base superior e inferior terão uma componente em y positiva e negativa, respetivamente, que é metade da altura do cilindro, uma vez que está centrado na origem. Como ambas as bases são circunferências, vimos a necessidade de recorrer a coordenadas polares para exprimir os seus pontos, tal como no cone e na esfera. Na figura seguinte encontra-se a representação de uma *slice* de um cilindro.

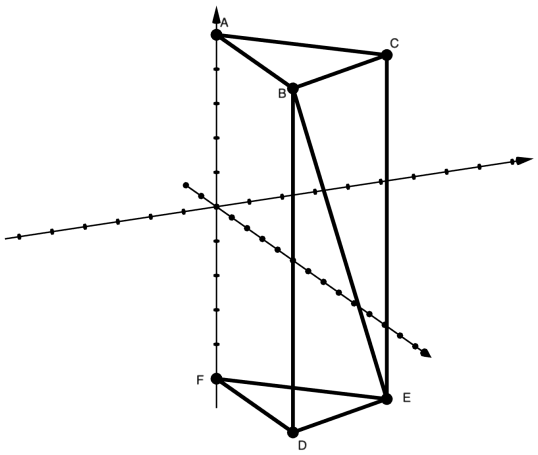


Figura 7: *Slice* de um cilindro.

Assumindo que a *slice* presente na figura 7 tem índice i e pertence a um cilindro com altura h, raio r, *slices* slices e $\alpha = \frac{2\pi}{slices}$, podemos calcular os seus vértices da seguinte maneira:

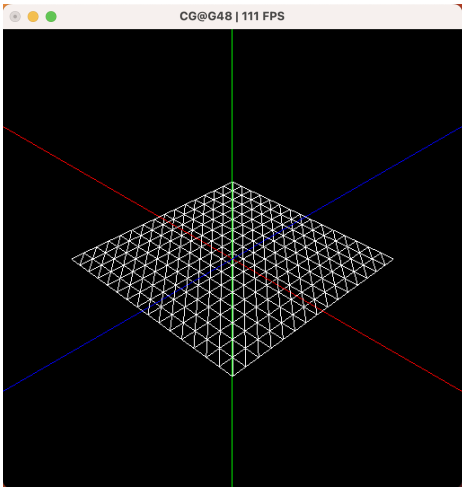
Ponto	**Coordenadas**
A	$(0, \frac{h}{2}, 0)$
B	$(r \times sin(i \times \alpha), \frac{h}{2}, r \times cos(i \times \alpha))$
C	$(r \times sin((i+1) \times \alpha), \frac{h}{2}, r \times cos((i+1) \times \alpha)$
D	$(r \times sin(i \times \alpha), -\frac{h}{2}, r \times cos(i \times \alpha))$
E	$(r \times sin((i+1) \times \alpha), -\frac{h}{2}, r \times cos((i+1) \times \alpha))$
F	$(0, -\frac{h}{2}, 0)$

Tabela 3: Coordenadas dos pontos da *slice*.

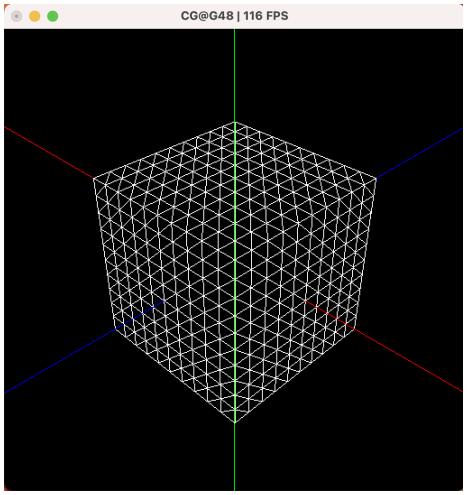
5 Resultados finais

Nesta secção encontram-se os resultados finais obtidos nas *demo scenes* criadas pelo grupo, anexadas com o projeto, e, ainda, a comparação entre os resultados obtidos e os resultados esperados em cada teste disponibilizado no enunciado do projeto.

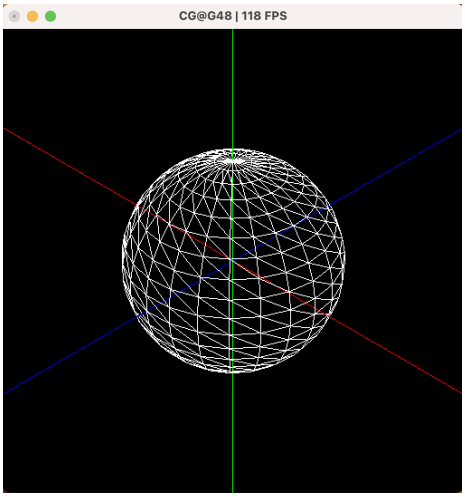
5.1 Primitivas gráficas



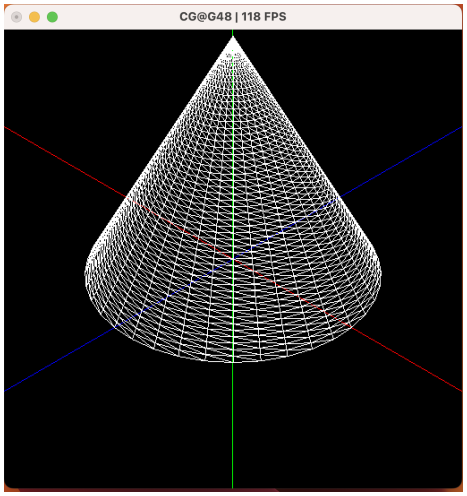
(a) Plano (*demo scene* 1).



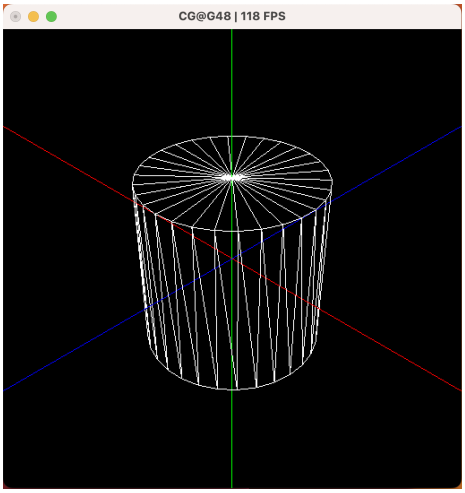
(b) Caixa (*demo scene* 2).



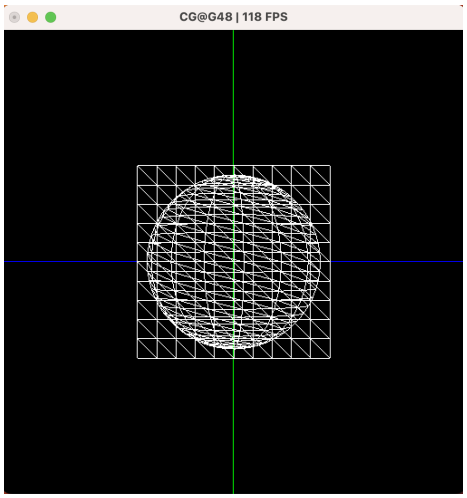
(c) Esfera (*demo scene* 3).



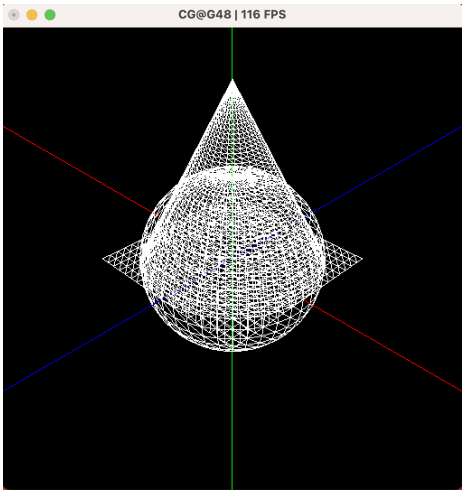
(d) Cone (*demo scene* 4).



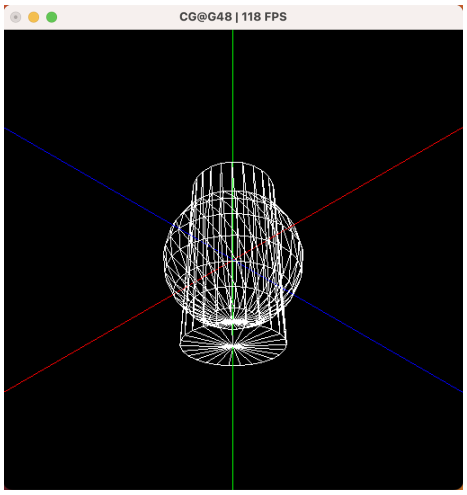
(e) Cilindro (*demo scene* 5).



(f) *Demo scene* 6.



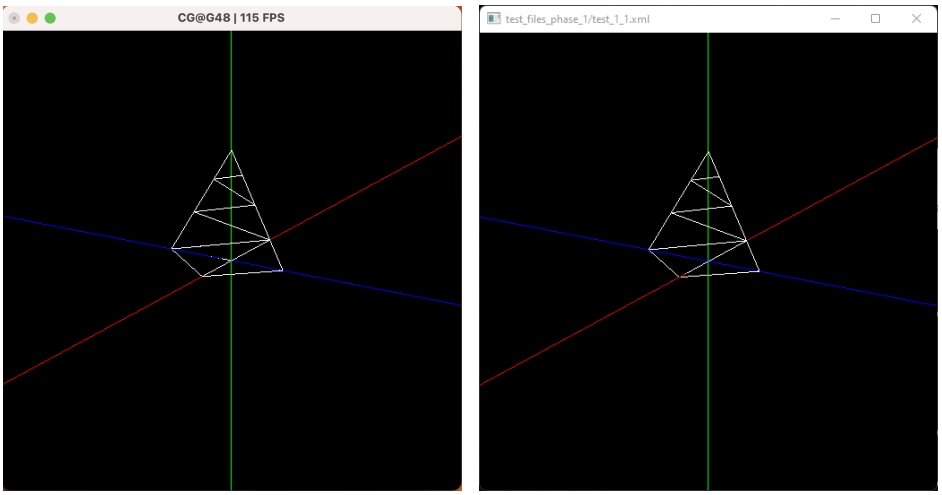
(g) *Demo scene* 7.



(h) *Demo scene* 8.

5.2 Testes

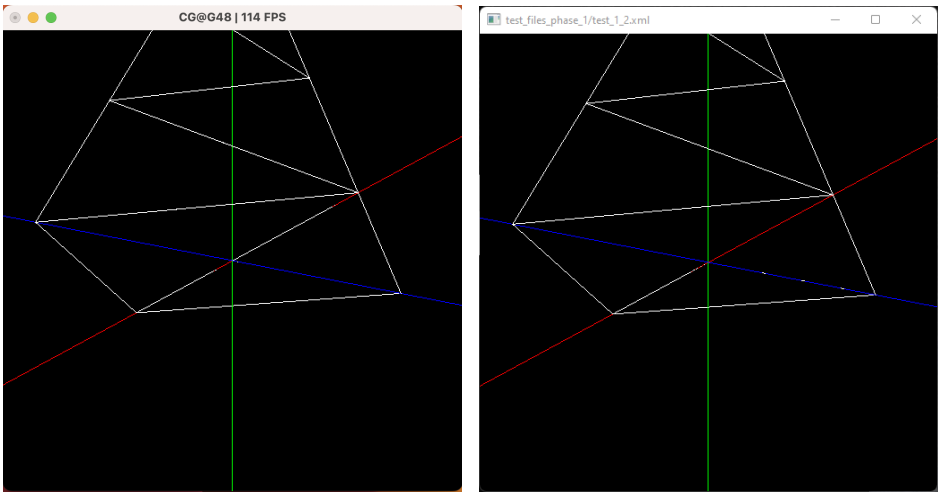
Teste 1



(i) Resultado obtido. (j) Resultado esperado.

Figura 8: Resultados do Teste 1.

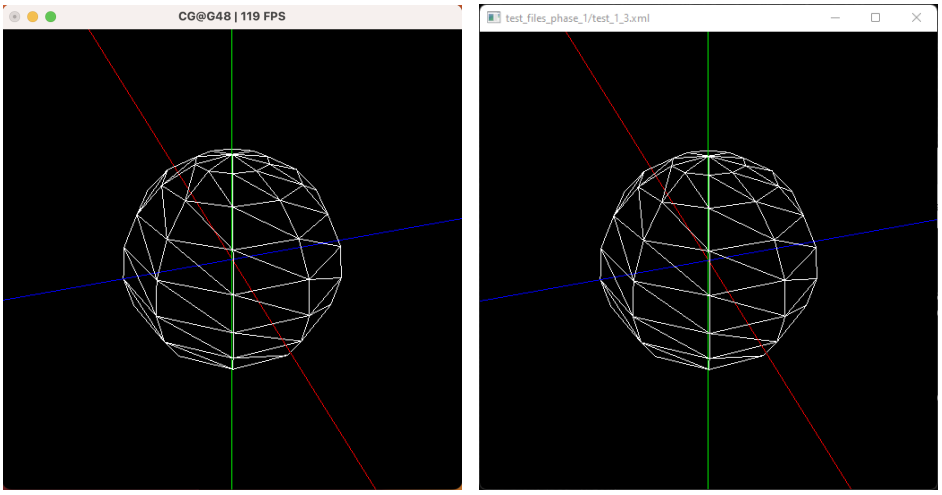
Teste 2



(a) Resultado obtido. (b) Resultado esperado.

Figura 9: Resultados do Teste 2.

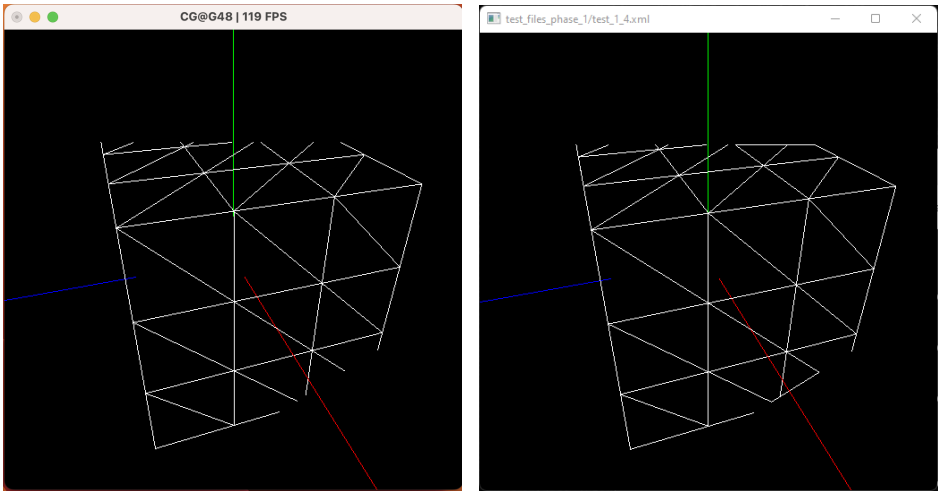
Teste 3



(a) Resultado obtido. (b) Resultado esperado.

Figura 10: Resultados do Teste 3.

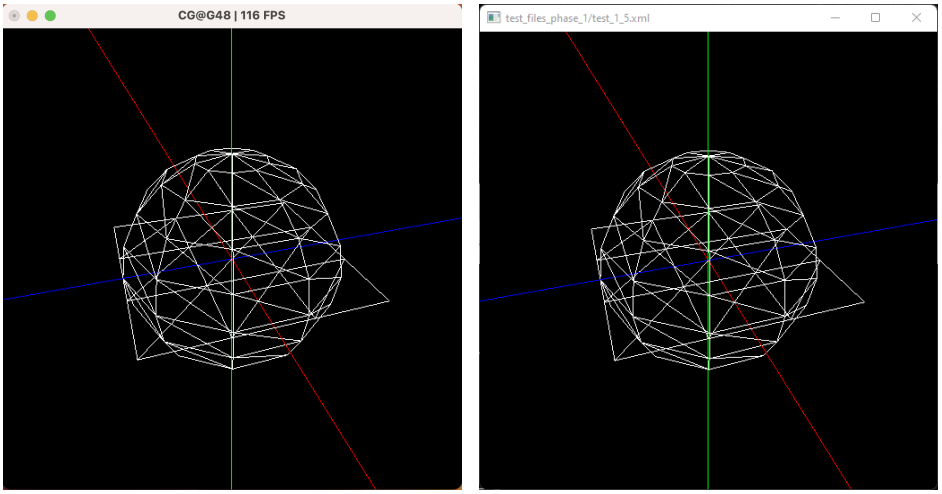
Teste 4



(a) Resultado obtido. (b) Resultado esperado.

Figura 11: Resultados do Teste 4.

Teste 5



(a) Resultado obtido. (b) Resultado esperado.

Figura 12: Resultados do Teste 5.

6 Conclusão

A realização desta primeira fase do projeto permitiu a consolidação da matéria lecionada nas aulas, nomeadamente sobre a utilização e domínio de ferramentas e conceitos associados à Computação Gráfica como OpenGL e Glut. Para além disso, foram também adquiridos conhecimentos de uma nova linguagem, C++, e, ainda, a exploração de novos domínios como a leitura de ficheiros com extensão XML.

De um modo geral, consideramos que os objetivos estabelecidos para esta fase foram cumpridos na íntegra e o trabalho desenvolvido foi também complementado com funcionalidades extra como uma nova primitiva gráfica (cilindro), o modo explorador, a implementação dos *Vertex Buffer Objects* e a contagem dos *frames* por segundo. Uma das tarefas que gostaríamos de ter realizado era a construção da primitiva gráfica torus, pelo que se trata de um objetivo para a fase seguinte. É importante visar também que, ao longo desta fase, o grupo preocupou-se sempre em manter a eficiência e boa *performance* das 2 aplicações desenvolvidas. Fazemos, portanto, uma avaliação positiva e estamos bastante satisfeitos com o trabalho realizado.