Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica

Grupo 30

Trabalho Prático - **Fase 1**

Primitivas Gráficas

Joana Alves (A93290)

Maria Cunha (A93264)

Vicente Moreira (A93296)

Março 2023

Conteúdo

1 Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Computação Gráfica da Licenciatura em Engenharia Informática, tendo como objetivo o desenvolvimento de um *mini scene graph based 3D engine*.

Este projeto está dividido em diversas fases de desenvolvimento, de forma a auxiliar e a agilizar o processo. Assim, este relatório integra a primeira fase deste projeto relativo à construção de primitivas gráficas através de uma aplicação *generator* e da sua visualização numa aplicação *engine*.

Relativamente ao **generator**, este é responsável, tal como o nome indica, de gerar (para um ficheiro de *output*) os vértices necessários para criar as figuras dadas como *input* pelo utilizador, respeitando as características cedidas.

No caso do **engine**, este terá como papel principal a leitura de um ficheiro de configuração *XML*, ficando responsável pela correta configuração da cena e *display* dos modelos incluídos no mesmo. É de notar que os modelos lidos são o resultado da aplicação *generator*.

Em suma, ao longo deste relatório irão ser apresentados os pormenores relativos à arquitetura, comportamento e decisões tomadas pelo grupo nesta primeira fase de desenvolvimento.

2 Generator

Tal como referido anteriormente, esta aplicação é responsável pela geração de ficheiros contendo o número de vértices totais e a sua respetiva descrição de coordenadas de certas figuras geométricas cedidas como *input* pelo utilizador. Assim, apresentamos os diversos componentes e características desta aplicação.

2.1 Formato .3d

O grupo começou por definir o formato dos ficheiros das primitivas a serem geradas. Inicialmente, foi planeado a escrita de um ficheiro num formato semelhante a **CSV**, onde este ficheiro começava com um número inteiro indicativo do número de pontos das primitivas, e de seguida, a descrição das coordenadas x,y,z dos vários pontos, com *floats* de 6 casas de precisão, assim como este exemplo:

```
3
2.000000;0.000000;0.000000
0.000000;0.000000;-2.333333
-2.000000;0.000000;0.000000
```

No entanto, este formato observou-se ser relativamente **ineficiente** na velocidade de leitura e escrita de pontos, assim como na geração de ficheiros de **tamanho elevado**. Dado este problema, decidimos alterar o formato do ficheiro ***.3d*** para armazenar informação "**binária**", ou seja, os valores do inteiro e dos pontos são escritos "diretamente"no ficheiro (ao invés de serem escritos "como texto").

Isto trouxe vários benefícios como maior precisão nos *floats* com uma menor utilização de espaço de disco, visto que estes só ocupam 4 *bytes* ao invés dos 9-10 *bytes* da versão anterior, e também facilitou o processo de escrita e leitura, pois é possível ler e escrever todos os pontos num só comando, através do cálculo do tamanho do *buffer* com **"NumPontos * sizeof(float)"**.

2.2 PointHolder.h

Para auxiliar na escrita dos vários pontos das primitivas, assim como fornecer um método que resolvesse o problema da escrita dos números dos pontos antes de calcular estes, decidimos criar uma classe auxiliar **PointHolder**. Esta classe permite às várias funções responsáveis pela geração de primitivas utilizar os mesmos métodos de armazenamento e escrita de pontos, o que traz vantagens na estabilidade e desenvolvimento destas funções geradoras.

A classe é composta por um *array* de *floats* dinâmico, ou seja, conforme este necessita de mais memória para armazenamento, esta classe é responsável por automaticamente alocar a memória necessária. A classe também fornece um método de escrita do seu conteúdo para um ficheiro, seguindo o formato especificado. Assim, as funções geradores de primitivas apenas necessitam de utilizar as funções ***addPoint*** e ***printToFile***, sem terem de se preocupar com o seu funcionamento interno.

2.3 Plane.cpp

Sendo esta a primitiva mais fácil de gerar, inicialmente criamos uma função simples, geradora do plano com os parâmetros fornecidos (no plano XZ, com Y=0) como pretendido pela primitiva. Este começa por calcular o tamanho das sub-divisões das divisões a serem geradas e, num ciclo iterativo para linhas e colunas, o plano é desenhado em pequenos quadrados através de dois triângulos, assim como a figura demonstra:

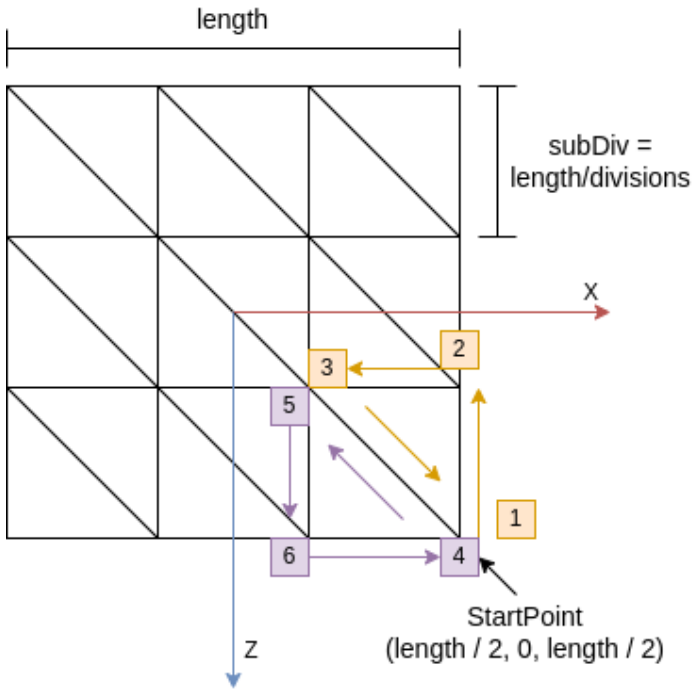


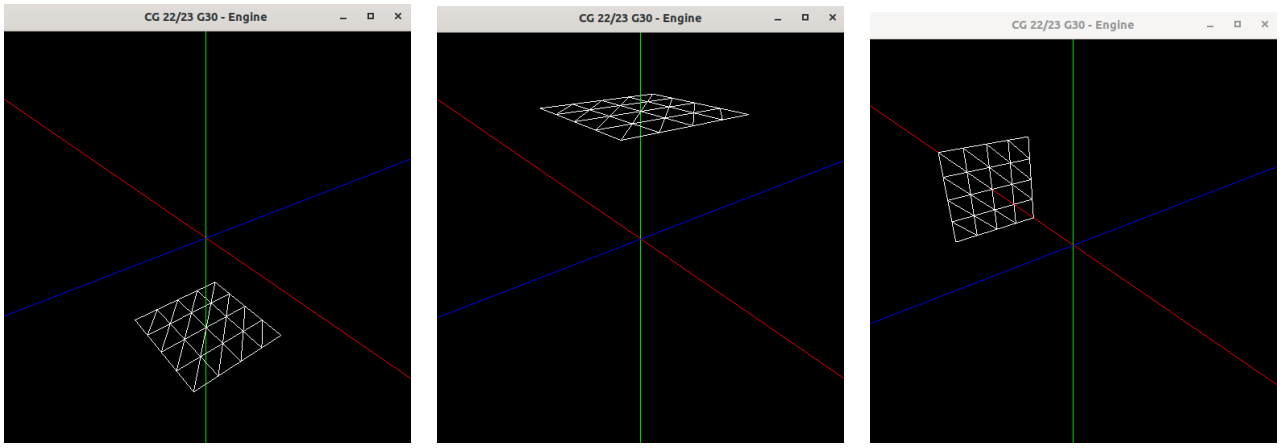
Figura 1: Estratégia de geração do plano

No entanto, como será descrito na primitiva *box*, para o desenvolvimento desta, e visto que a primitiva é composta por 6 planos, seria útil que a função geradora de plano pudesse ser mais versátil. Com este objetivo, desenvolvemos uma função mais genérica de geração do plano com os seguintes argumentos:

addGenericPlane(PointHolder ph, int length, int divisions, char axis, bool direction, float distance):

- **PointHolder ph:** PointHolder onde os pontos gerados serão armazenados.
- **int length:** Tamanho do plano.
- **int divisions:** Número de sub-divisões.
- **char axis:** Eixo normal ao plano (Valores: 'x','y','z', p.ex.: 'y' == Plano XZ)
- **bool direction:** Direção de visualização do plano. *True* indica que o plano está virado para o eixo positivo definido, ou seja, poderá ser visualizado por alguém situado nesse eixo positivo.
- **float distance:** Distância de deslocamento do plano, relativo ao eixo definido.

Com esta função definida é possível gerar planos genéricos perpendiculares a qualquer eixo x, y ou z, tal como os seguintes exemplos demonstram:



$addGenericPlane(ph, 2, 4, 'y', true, -2);$ $addGenericPlane(ph, 2, 4, 'y', true, 2);$ $addGenericPlane(ph, 2, 4, 'x', true, -3);$

Figura 2: Exemplos de geração do plano

2.4 Box.cpp

Para a primitva *box*, uma vez que já foi desenvolvida uma função de geração de plano genérica, é possível gerar o cubo recorrendo a esta função, efetuando uma chamada por cada face do cubo, resultando numa função de geração simples e de fácil compreensão, como descrito no sua totalidade abaixo:

```
void generateBox(int length,int divisions,FILE* file){
    PointHolder ph = initPointHolder();
    float subLength = ((float) length) / ((float) 2);

    addGenericPlane(ph,length,divisions,'y', true,subLength);
    addGenericPlane(ph,length,divisions,'y', false,-subLength);

    addGenericPlane(ph,length,divisions,'x', true,subLength);
    addGenericPlane(ph,length,divisions,'x', false,-subLength);

    addGenericPlane(ph,length,divisions,'z', true,subLength);
    addGenericPlane(ph,length,divisions,'z', false,-subLength);

    printToFile(ph,file);
}
```

2.5 Sphere.cpp

Para a geração da primitiva da esfera é necessário definir dois ângulos, **alpha** (α) e **beta** (β), para efetuar os cálculos dos pontos. *Alpha* representa o ângulo de abertura de um círculo imaginário no plano XZ, onde o eixo positivo do Z corresponde a 0^{o} e o eixo positivo do X 90^{o}, sendo que este ângulo poderá efetuar um completo 360^{o}. Já o ângulo *Beta* está situado no plano XY (ou ZY) e este começa no eixo negativo do Y, com um valor de -90^{o} e pode abrir até o eixo positivo do Y, onde fica a 90^{o}, totalizando uma abertura de 180^{o}.

Com estes ângulos e um raio (r) definido, é possível calcular as coordenadas de um ponto numa esfera através das seguintes fórmulas:

$$\mathbf{X} = r * cos(\beta) * sin(\alpha)$$
$$\mathbf{Y} = r * sin(\beta)$$
$$\mathbf{Z} = r * cos(\beta) * cos(\beta)$$

Assim, na função de geração, começamos por calcular os valores dos sub-ângulos *alpha* e *beta*, provenientes do número de *slices* e *stacks* fornecidas, respetivamente. Depois as várias *slices* e *stacks* são iteradas, e, em cada iteração, são calculados 4 pontos correspondentes a uma "*plate*", como ilustrado na figura:

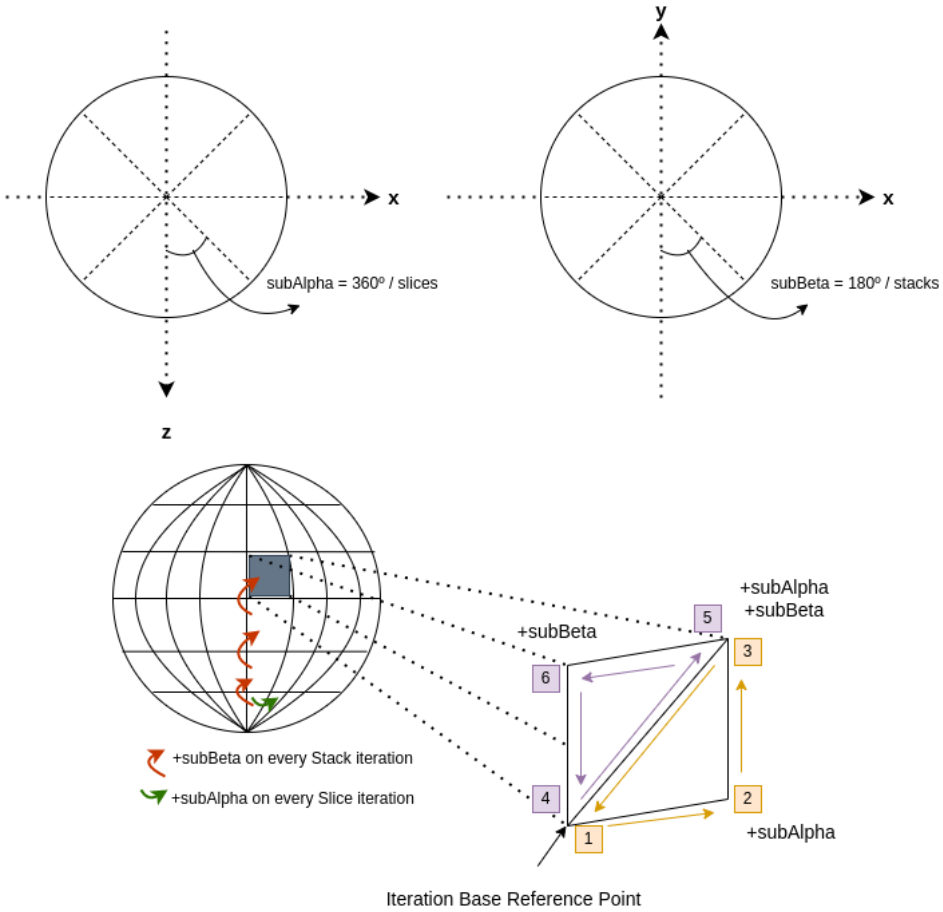


Figura 3: Construção da primitiva Sphere

2.6 Cone.cpp

A função responsável pelo desenvolvimento do cone recebe como argumentos o raio, a altura, o número de *slices* e *stacks* do cone e, por fim, o apontador para o ficheiro de escrita.

A construção do cone foi dividida em 3 partes, usando um ciclo para cada uma. A primeira define-se pela inicialização do ponto centro da base com todas as coordenadas a 0 sendo este ponto estático durante o ciclo inteiro, ou seja, é um ponto comum em todos os triângulos que se desenha até que uma rotação inteira seja concluída, originando assim a base esférica.

```
float slice = doisPI/slices;

    for (float i = 0; i < doisPI; i += slice) {
        addPoint(ph, 0.0f, 0.0f, 0.0f);
        addPoint(ph, (sin(i + slice) * radius),0.0f, (cos(i + slice) * radius));
        addPoint(ph, sin(i) * radius,0.0f, cos(i) * radius);
    }
```

Assim, os restantes ciclos continuam esta geração de pontos, construindo de baixo até ao topo do cone. A matemática utilizada é semelhante à que já foi apresentada. Por exemplo, uma *slice* foi definida como $2 * pi/slices$, o que é um simples desenvolvimento da fórmula $(360 * pi)/(180 * slices)$. Esta definição de variáveis unitárias provou ser útil e de fácil compreensão num contexto cíclico.

2.7 Cylinder.cpp

Por último, com o auxílio de cálculos e análises elaboradas nas aulas práticas, decidimos incluir uma figura geométrica extra: o **cilindro**. Desta forma, criamos uma função que calcula os pontos do cilindro dado as suas características como: raio, altura e número de *slices*. A forma como esta função calcula os triângulos do cilindro pode ser descrita em fatias, isto é, através do número de *slices* conseguimos definir o ângulo de abertura de cada fatia do cilindro. Assim, em cada iteração do ciclo de cálculo, são calculados os pontos e respetivos triângulos de uma fatia, começando pelo topo, de seguida os dois triângulos laterais e, por fim, a base. Para uma melhor perceção incluímos a figura 4:

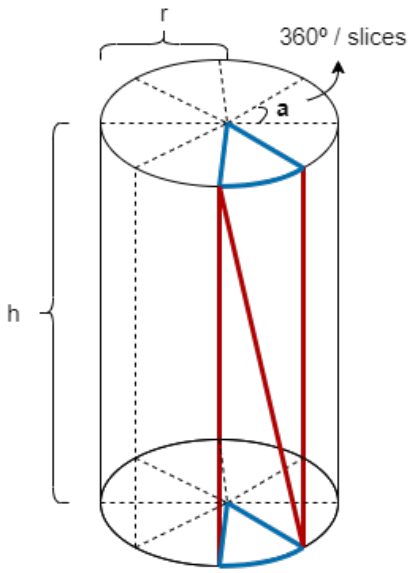


Figura 4: Estratégia de geração do cilindro

3 Engine

Passamos ao desenvolvimento do *Engine* do nosso projeto. Este é responsável por, dado um ficheiro de configuração em *XML* que descreve os vários aspetos de uma cena, montar esta e apresentá-la ao utilizador através das diretivas do openGL.

3.1 Estrutura de dados

Para evitar a necessidade de ler o ficheiro de configuração por cada nova frame ou manipulação da cena, o que pode levar a tempos de processamento elevados, decidimos que esta configuração inicial começa por ser carregada em memória na sua totalidade, utilizando uma estrutura de dados definida pelo *Engine*.

Para facilitar o processo de leitura, assim como manter a simplicidade na compreensão da estrutura de dados, decidimos criar um modelo semelhante à sintaxe do ficheiro *XML*. Este é definido por uma classe principal **World** e contém várias classes com a informação da cena, assim como o nodo *world* contém os vários *children nodes*, no ficheiro *XML*.

Optamos por esta estrutura visto que irá permitir facilmente a expansão de novos nodos/-classes, quer seja pela a introdução de transformações e luzes, ou a modificação das próprias classes, como os modelos, para adicionar o suporte de texturas.

Eis um sumário da estrutura de dados utilizada pelo *Engine*:

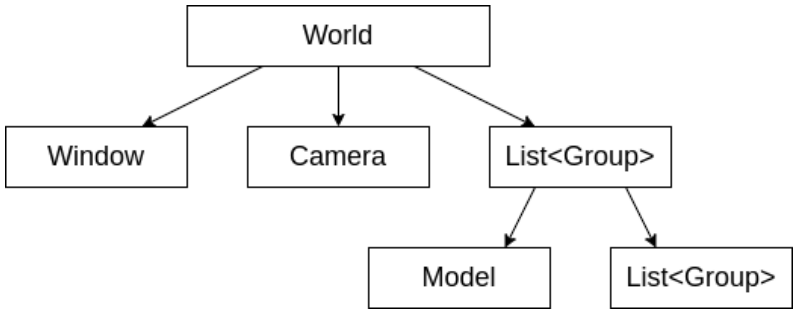


Figura 5: Estrutura de Dados do Engine

- **world.h:** Classe principal, contém as classes *Window*, *Camera* e uma lista de instâncias da classe *Group*. Esta classe também contém a função de carregamento da configuração *XML* (***loadXML***) e a função principal de desenho (***draw***).
- **window.h:** Classe de informação simples, contém as dimensões da janela a ser criada, caso não seja definida, utiliza um valor *default* de **512 x 512**.
- **camera.h:** Classe de informação simples, contém os valores de posição, direção, "*up*"e de perspetiva da câmara.
- **group.h:** Classe responsável por armazenar os modelos, assim como a sua lista de sub-grupos, poderá ser evoluído para suportar transformações.
- **model.h:** Classe de armazenamento do modelo, lido previamente na fase de configuração. Este segue uma estrutura semelhante à definida no *PointHolder* e poderá ser expandido para suportar novas features.

3.2 Pugixml

Para a leitura do ficheiro de configuração inicial, utilizamos a biblioteca **pugixml**, que permite facilmente pesquisar, ler e manipular elementos num ficheiro *XML*. Esta biblioteca contém funções relevantes como:

- **xml_document :: load_file:** Função de abertura/leitura do ficheiro XML.
- **xml_node :: child:** Função de pesquisa e leitura de nodos no ficheiro XML. Devolve o primeiro nodo que satisfaz a pesquisa.
- **xml_node :: next_sibling:** Função de iteração de nodos num ficheiro XML, pesquisa pelo nodo indicado no mesmo nível que o nodo atual, ou seja, os seus "irmãos".
- **xml_node :: attribute:** Função de leitura de atributos de um nodo.

A leitura do ficheiro de configuração é efetuada pela classe *World*, que utiliza a função ***loadXML***. Esta começa por ler os valores dos nodos *window* e *camera*, casos estes não existam, são utilizados valores *default* definidos. Por último, são lidos os grupos de forma recursiva, sendo estes responsáveis pelo carregamento dos modelos a serem usados no seu grupo.

3.3 OpenGL

Com a estrutura de dados definida e a leitura do ficheiro de configuração inicial completa, podemos passar à fase de desenho e visualização da cena. Visto que a classe *World* contém toda a informação da cena, é apenas necessário aceder a esta ou executar as funções de desenho na classe.

Para a definição da *window* e da câmara, os vários valores são acedidos através das variáveis presentes em ***myWorld.myWindow*** e ***myWorld.myCamera***, como a *width*/*height*, ou os valores de posX,posY, lookX, lookY, etc.

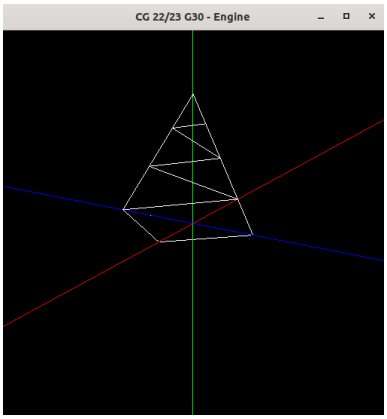
Para o desenho da cena, é invocada a função ***draw()*** na classe ***myWorld***, esta executa de forma recursiva o desenho dos vários grupos e subgrupos descritos na estrutura de dados. Para os modelos, é executada a função ***drawModel()***, presente na classe. Esta função percorre a lista de pontos armazenada, previamente carregada na fase de leitura e invoca a diretiva **glVertex3f** do *openGL*.

4 Testes e Resultados

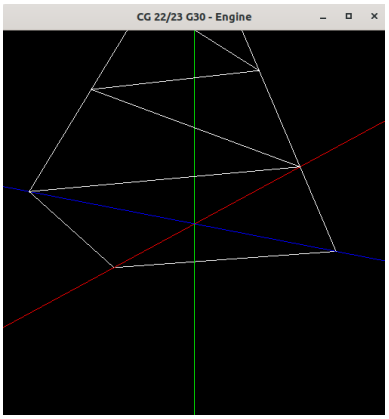
Nesta secção vamos apresentar os diversos testes executados pelo grupo de modo a conseguir avaliar ambas as aplicações quanto à sua correção e exatidão de resultados.

4.1 Testes Docentes

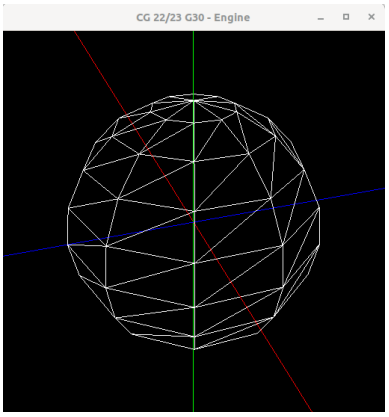
Alguns dos testes implementados foram cedidos pelos docentes de modo a verificar o comportamento de ambas as aplicações de maneira mais eficiente, uma vez que os resultados esperados foram divulgados em conjunto com os ficheiros de configuração. Assim, apresentamos os resultados obtidos em cada um dos testes pela nossa aplicação, sendo de notar que todos estes foram de encontro ao esperado.



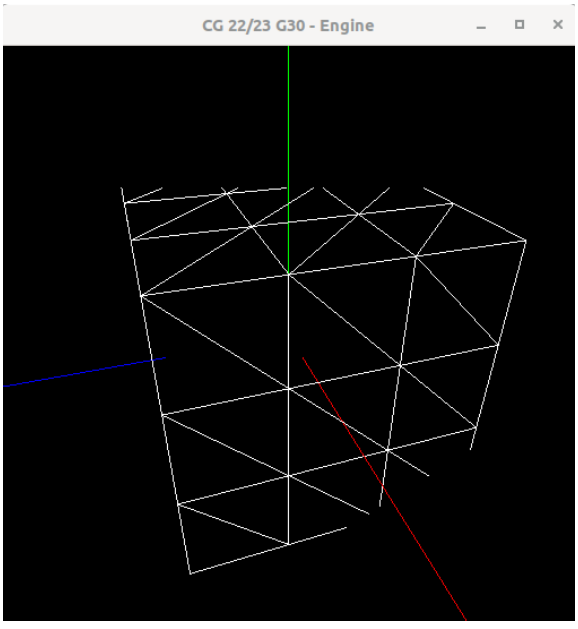
test_1_1



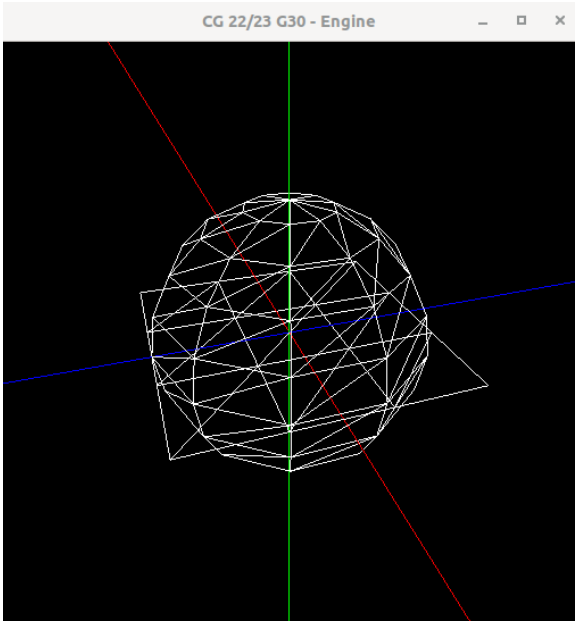
test_1_2



test_1_3



test_1_4



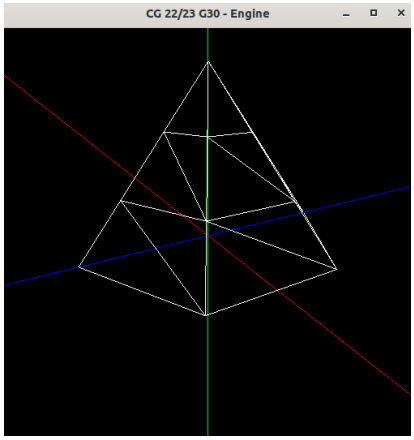
test_1_5

4.2 Testes Grupo

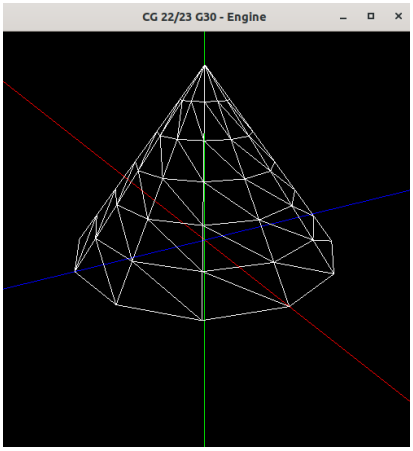
De seguida, para além de testes básicos de comportamento, o grupo decidiu realizar três testes para as figuras Cone e Esfera, de modo a acentuar o comportamento das mesmas quando o número de *slices* e *stacks* vai alterando, assim como a adição de testes para a primitiva extra desenvolvida (cilindro).

4.2.1 Cone

Para o Cone, começamos por realizar um teste com 6 *slices* como parâmetro, ficando a parecer uma pirâmide hexagonal. De seguida, duplicamos o número de *slices* onde podemos denotar um aumento na suavidade das curvas da base. Por fim, duplicamos novamente o número de *slices* para maximizar a suavidade da base, tendo o resultado apresentado na figura 6.



6 *slices* - 3 *stacks*



12 *slices* - 6 *stacks*

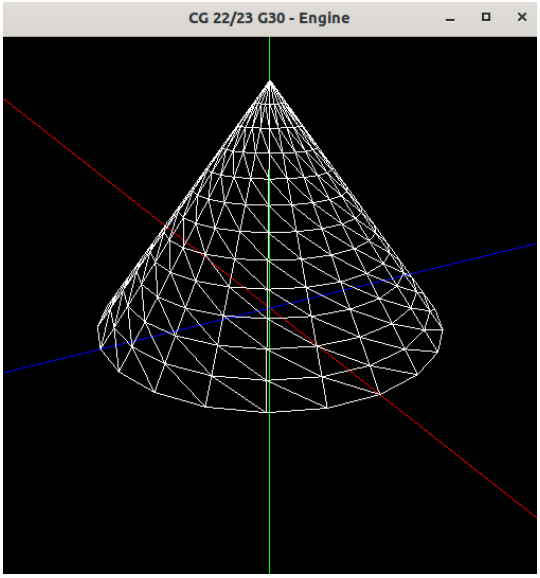
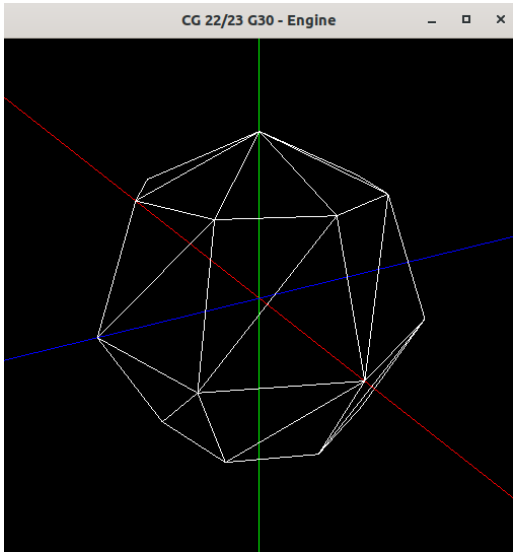


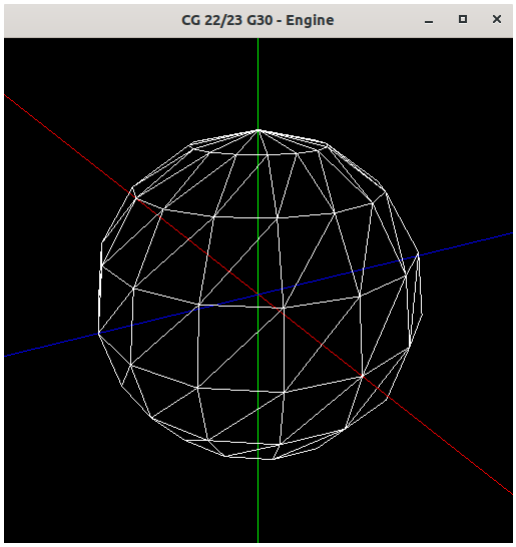
Figura 6: 24 *slices* - 12 *stacks*

4.2.2 Esfera

Para a esfera, seguimos o mesmo método que o Cone, ou seja, começamos com número reduzidos tanto de *slices* como de *stacks*, obtendo uma figura com arestas muito acentuadas. De seguida, duplicamos os dois parâmetros para começar a suavizar as arestas. Por fim, tentamos maximizar a suavidade da esfera obtendo o resultado da figura 7.



8 *slices* - 4 *stacks*



16 *slices* - 8 *stacks*

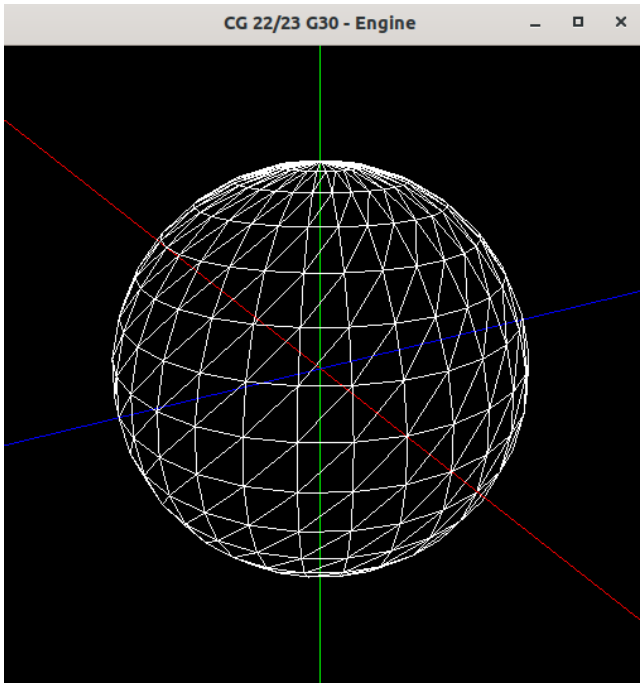


Figura 7: 32 *slices* - 16 *stacks*

4.2.3 Cilindro e Cubo

Por último, decidimos incluir um teste da nova primitiva Cilindro (figura 8), assim como testar a primitiva do Cubo (figura 9), obtendo os seguintes resultados:

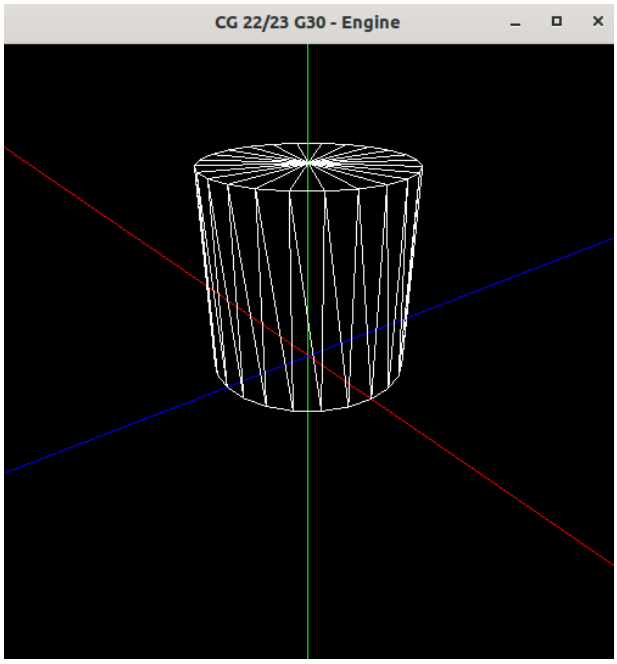


Figura 8: 2 *radius* - 24 *slices* - 2 *height*

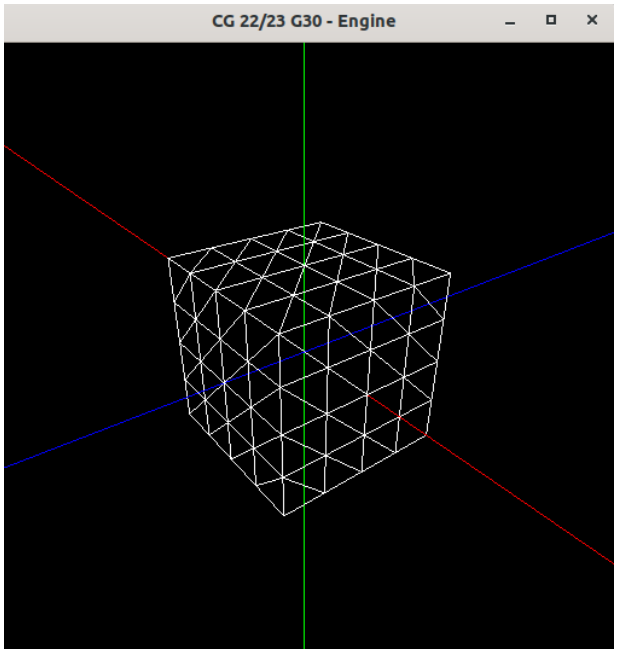


Figura 9: 2 *length* - 4 *divisions*

5 Conclusão

Após a realização desta primeira fase do projeto, o grupo encontra-se satisfeito com o trabalho desenvolvido, tendo sido alcançados todos os objetivos estabelecidos quer pelos docentes, como pelo próprio grupo. No entanto, acreditamos que algumas abordagens a certos aspetos poderão vir a ser alteradas nas fases seguintes.

Em suma, aprofundamos o conhecimento lecionado nas aulas teóricas e práticas da unidade curricular em relação à linguagem de programação C++, às várias primitivas gráficas a implementar, assim como a utilização da biblioteca *Glut* e da API *openGL*.