Computação Gráfica

Trabalho Prático - Fase 1

Universidade do Minho, Departamento de Informática
José Correia, Diogo Abreu, e Rodrigo Monteiro
{100610, 100646, 100706}@alunos.uminho.pt

Grupo 69

1. Introdução

Este relatório apresenta o resultado da primeira fase do projeto desenvolvido na disciplina de Computação Gráfica. Nesta primeira fase foi nos proposto o desenvolvimento de duas aplicações: um **Generator**, gerador de vértices para primitivas gráficas, e uma **Engine**, aplicação que lê ficheiros de configuração XML e desenha os vértices das primitivas gráficas gerados previamente. Para implementação destas aplicações recorremos ao uso da ferramenta OpenGL, utilizando a linguagem de programação C++.

2. Objetivos

O objetivo consistiu no desenvolvimento de cenários gráficos em 3D específicos, onde as primitivas gráficas, plano, caixa, cone e esfera, estivessem presentes. A representação gráfica destas quatro figuras é construída com base em múltiplos triângulos, sendo primeiramente necessário determinar os vértices que compõem cada um desses triângulos, para posteriormente desenharmos os triângulos de forma a gerar as figuras nos cenários pretendidos.

3. Arquitetura do projeto

O projeto está divido nos seguintes *packages*/*libraries* de modo a que que seja possível haver partilha e reutilização de código dentro da mesma solução para diferentes projetos de inicialização, como o *generator* e a *engine*.

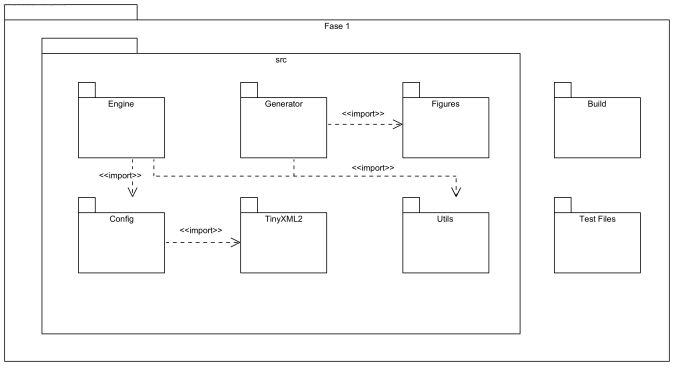


Figure 1: Diagrama de *packages*

Assim, a *engine* utiliza a biblioteca *config* que, a partir da bilbioteca *tinyXML2*, faz o parsing de um ficheiro de configuração, guardando uma série de informações em estruturas de dados.

Para além disso, tanto a *engine* como o *generator* utilizam a biblioteca *utils* que contém as classes *Figure* e *Point*, necessárias para a construção dos diversos tipos de figuras:

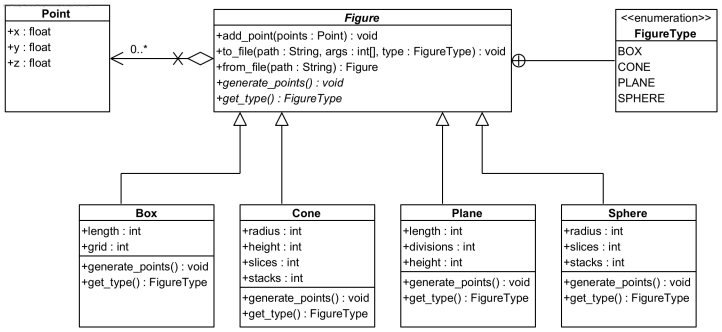


Figure 2: Diagrama de classes

Assim, foi utilizada uma classe abstrata *Figure* que possui um *vector* de *Point*, e os seguintes métodos: adição de um ponto; figura para ficheiro; ficheiro para figura (*static*); geração de pontos (*abstract*); e um *getter* para o tipo de figura (*abstract*). As subclasses de *Figure* implementam, então, os métodos `generate_points` e `get_type`.

Deste modo, dispomos de polimorfismo, e caso se necessite de fazer *dynamic cast*, é possível fazê-lo de forma segura através da garantia do `get_type` – dado que o *dynamic cast* é uma operação dispendiosa.

4. Generator

O generator é a aplicação que trata da geração de todos os pontos necessários para o desenho das primitivas gráficas pretendidas. As funções e algoritmos utilizados têm sempre em conta que todas as figuras serão formadas apenas pela junção de múltiplos triangulos. Durante a geração dos pontos estes são armazenados num vetor, e por fim escritos num ficheiro *.3d*.

4.1. Cálculo de vértices

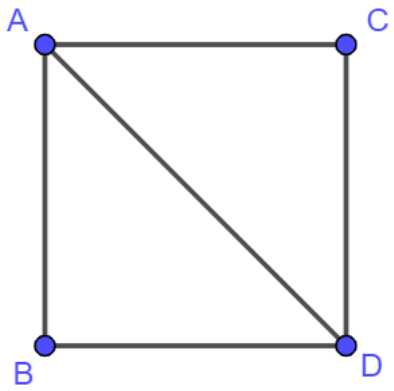
4.1.1. Plane

O plano está contido no plano XZ com centro na origem, ponto (0, 0, 0) e tem como variáveis o seu comprimento (*length*) e o número de divisões iguais por aresta (*divisions*).

Dado que a figura é um quadrado posicionado na origem sabemos que os limites da figura estão a *length/2* dos respetivos eixos paralelos e que todos os pontos têm o valor de *y* constante (no caso 0). Então abordamos este problema como uma matriz *divisions* x *divisions*, onde cada elemento é um quadrado de comprimento *length/divisions*.

```
float dimension2 = static_cast<float>(length) / 2;
float div_side = static_cast<float>(length) / divisions;
float f_height = static_cast<float>(height);
```

Cada um destes elementos pode ser desenhado pela junçao de dois triâgulos cuja hipótenusa é a diagonal do quadrado. Assim sendo, para cada triângulo, respeitando a regra da mão direita, usamos A-B-D para o primeiro triângulo e D-C-A para o segundo triâgulo.



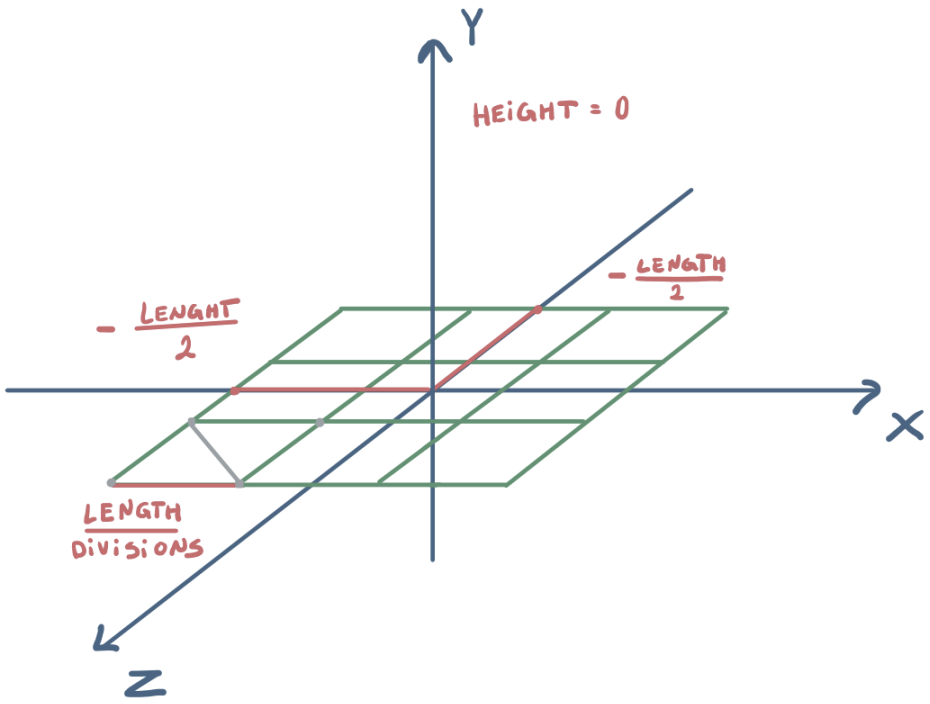
A nível de código conseguimos obter todos os pontos a partir de dois ciclos *for*, onde cada iteração completa gera uma fila de quadrados compostos por dois triângulos. O ciclo começa na extremidade pertencente ao 2º quandrante e termina na extremidade oposta.

```
for (int line = 0; line < divisions; line++) {
    float z1 = -dimension2 + line * div_side;
    float z2 = z1 + div_side;

    for (int column = 0; column < divisions; column++) {
        float x1 = -dimension2 + column * div_side;
        float x2 = x1 + div_side;

        // First triangle
        add_point(Point(x1, f_height, z1)); // A
        add_point(Point(x1, f_height, z2)); // B
        add_point(Point(x2, f_height, z2)); // D

        // Second triangle
        add_point(Point(x2, f_height, z2)); // D
        add_point(Point(x2, f_height, z1)); // C
        add_point(Point(x1, f_height, z1)); // A
    }
}
```

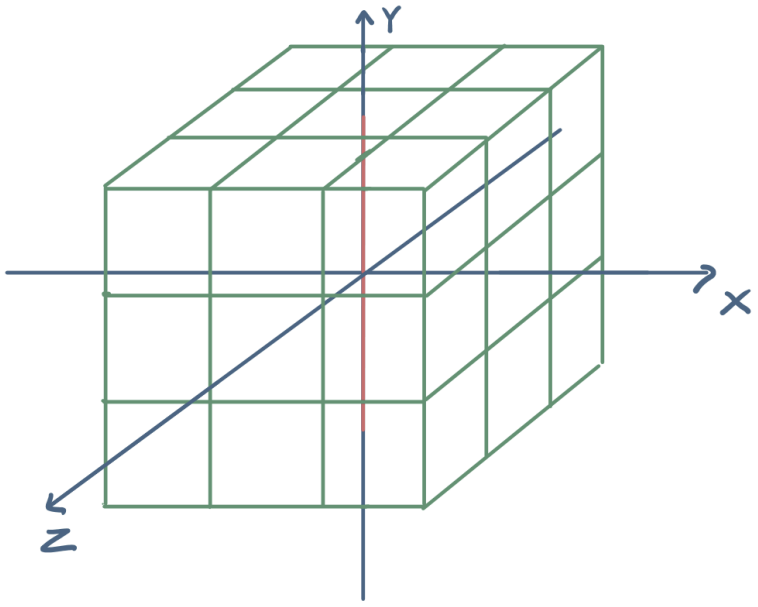


4.1.2. Box

A caixa está centrada na origem, ponto (0, 0, 0) e tem como variáveis o seu comprimento (*length*) e o número de divisões iguais por aresta (*divisions*). Como a caixa é constituída por seis planos quadrangulares, utilizou-se exatamente o mesmo raciocínio utilizado na geração dos pontos do plano XZ, mas agora aplicado também aos planos XY e YZ.

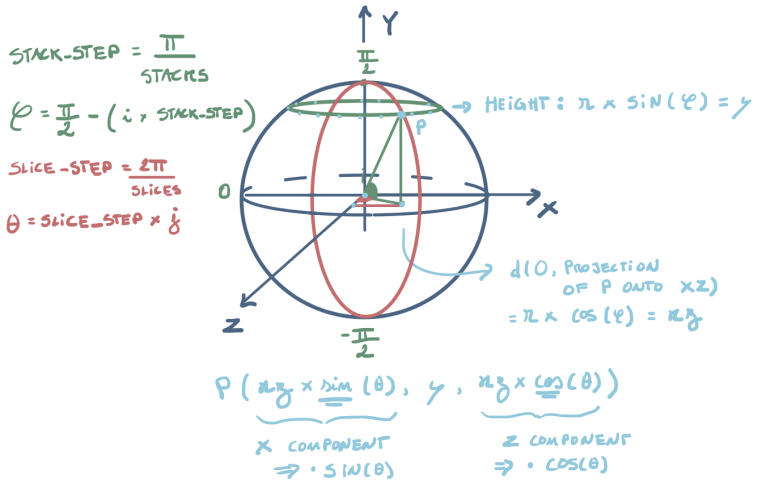
Como a caixa está centrada na origem cada face está distanciada *length/2* dos respetivos planos paralelos XZ, XY, YZ.

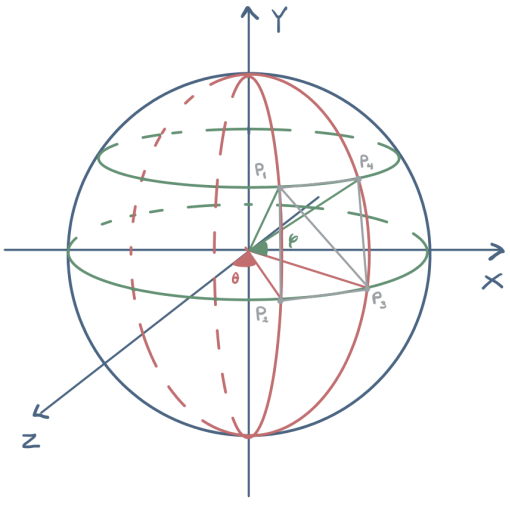
Face Frontal –> (x, y, *length/2*)
Face Traseira –> (x, y, *length/2*)
Face de Baixo –> (x, *length/2*, z)
Face de Cima –> (x, *length/2*, z)
Face da Direita –> (*length/2*, y, z)
Face da Esquerda –> (*length/2*, y, z)



4.1.3. Sphere

Primeiramente, são calculados os valores angulares dos intervalos entre *stacks* e entre *slices*, utilizados para incrementar φ e θ. De seguida, são feitas *stacks* $\times$ *slices* iterações. O *outer loop* é relativo às stacks: onde o φ ($\frac{\pi}{2}$ até $-\frac{\pi}{2}$), a altura do ponto, e a distância da origem até à projeção do ponto no plano XZ são calculados. Com o *inner loop* é calculado o θ (0 até 2π), sendo a criação de pontos feita em circunferências horizontais, os quais formam triângulos – dois por "célula".

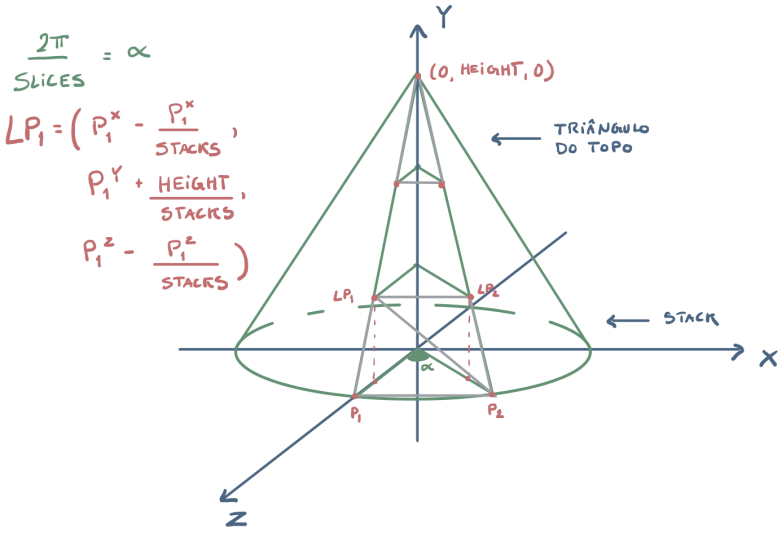





```
Point p1 = Point(xy      * sinf(theta),      y,      xy      * cosf(theta) );
Point p2 = Point(xy_next * sinf(theta),      y_next, xy_next * cosf(theta) );
Point p3 = Point(xy_next * sinf(theta_next), y_next, xy_next * cosf(theta_next) );
Point p4 = Point(xy      * sinf(theta_next), y,      xy      * cosf(theta_next) );
```

4.1.4. Cone

Os pontos da base são calculados, $P(r \times \sin(\alpha), 0, r \times \cos(\alpha))$, e guardados num `vector`. Depois, para cada dois pontos da base, são criados os pontos que formam a diagonal com origem nos pontos da base até ao topo do cone. Assim, depois dos triângulos da base, são calculados os triângulos de cada "célula" formada pelas *stacks* e *slices*. No fim de cada duas diagonais é construído o triângulo do topo.



4.2. **Criação e escrita nos ficheiros**

A criação e escrita em ficheiros possui a mesma lógica para todos os tipos de figuras. Portanto, os métodos responsáveis por isso estão classe *Figure*.

```
void to_file(const string& path, const vector<int>& args, FigureType
type)
```

O método `to_file` escreve, para um determinado ficheiro `.3d`, uma linha inicial com os seguintes elementos, separados por um ponto e vírgula:

```
«Índice do tipo de figura»;«Nº de pontos»;«Arg nº1»; ... ;«Arg nºN»
```

De seguida, escreve as coordenados dos pontos, com um ponto por linha:

```
p1x;p1y;p1z
...
pNx;pNy;pNz
```

```
Figure* from_file(const string& path)
```

O método `from_file` lê os dados estruturados da forma descrita anteriormente, e, a partir deles, instância a figura de acordo com o seu tipo – caso tenha o número correto de argumentos.

```
Figure* instance = nullptr;
if (type == FigureType::BOX && args.size() >= 2) {
        instance = new Box(args.at(0), args.at(1), points);
// ...
```

5. Engine

5.1. XML parser

A *engine* utiliza uma classe `Config` que guarda configurações da janela e da câmera, e os caminhos para os ficheiros das figuras a construir. Estas configurações são lidas de um ficheiro `.xml` com recurso à biblioteca `tinyXML2.h`:

```
tinyxml2::XMLDocument doc;
if (doc.LoadFile(file_path) == tinyxml2::XML_SUCCESS) {
    tinyxml2::XMLElement* root = doc.FirstChildElement("world");
    tinyxml2::XMLElement* cam = root->FirstChildElement("camera");
    tinyxml2::XMLElement* pos_cam = camera-
>FirstChildElement("position");
    // ...
```

5.2. Desenho das figuras

A partir dos caminhos para os ficheiros guardados classe `Config`, são instanciadas as figuras a partir do método `from_file` e adicionados a um `vector`.

```
for (unsigned int i = 0; i < config.models.size(); i++) {
    Figure* f = Figure::from_file(config.models.at(i));
    figures.push_back(f);
  }
```

Posteriormente, são renderizadas percorrendo os pontos de cada figura e contruindo os triângulos com esses pontos.

```
void drawFiguras() {
  glBegin(GL_TRIANGLES);
  for (Figure* f : figures)
    for (Point p : f->points)
      glVertex3d(p.x, p.y, p.z);
  glEnd();
```

5.3. Movimentação da Câmara

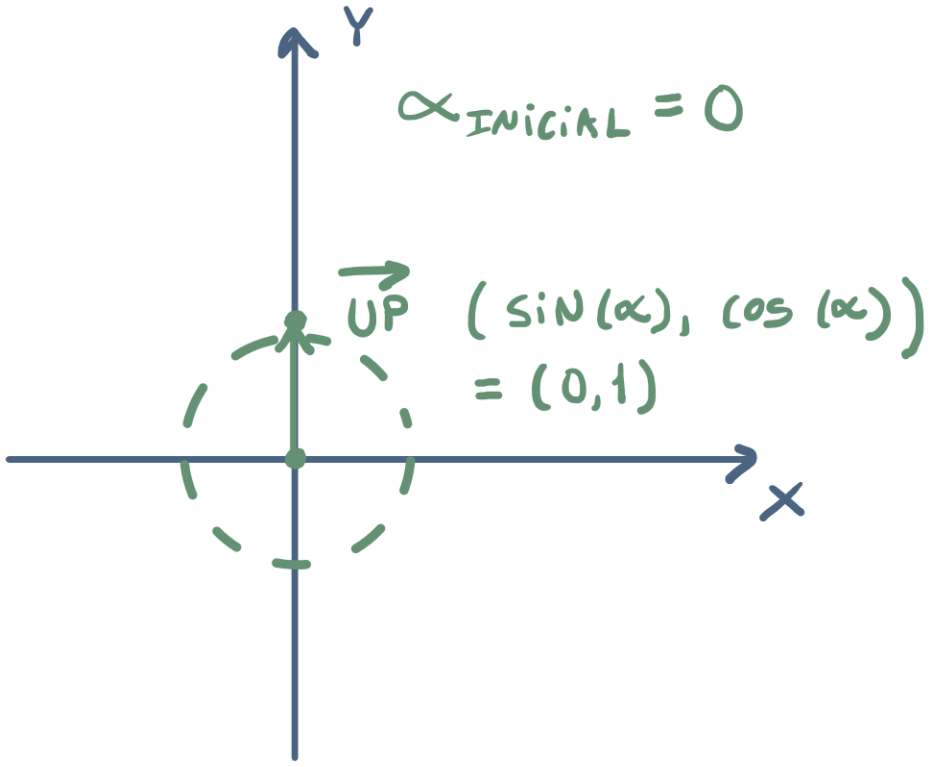
5.3.1. Coordenadas esféricas

De forma a poder movimentar a câmara em torno dos cenários criados optamos por utilizar coordenadas esféricas, com estes método conseguimos posicionar a câmara em qualquer ponto num determinado raio esférico.

```
px = radius * cos(beta_) * sin(alpha);
py = radius * sin(beta_);
pz = radius * cos(beta_) * cos(alpha);
```

5.3.2. Camera Roll

Este mecanismo foi implementado para que conseguissemos rodar o nosso campo de visão. Para obter este efeito fazemos a câmara rodar em torno do própio eixo de direção, ou seja, uilizando coordenadas polares alteramos os valores do x e do y do *up-vector*.



6. Demonstração

6.1. Manual de utilização

A Rotação para a esquerda
D Rotação para a direita
S Rotação para a baixo
W Rotação para a cima
← Rotação da câmara no sentido CCW
→ Rotação da câmara no sentido CW
↓ Zoom Out
↑ Zoom In
L Modo GL_LINE
P Modo GL_POINT
F Modo GL_FILL

6.2. Execução do programa

O nosso projeto é compilado com base num ficheiro CMakeList, que nos permite compilar o projeto em qualquer máquina que tenha o cmake instalado. No nosso caso o ambiente em que trabalhamos foi o Visual Studio.

A execução está dividida em dois executáveis, o **genaretor** e o **engine**. Podemos tomar como exemplo, a geração de uma box com $length = 2$ e $grid = 3$:

```
C:\build\Debug>generator box 2 3 box_2_3.3d
```

Com o ficheiro .3d criado, podemos avançar para a execução dos testes.xml no engine:

```
C:\build\Debug>engine "..\..\test_files\test_1_4.xml"
```

6.3. **Output**

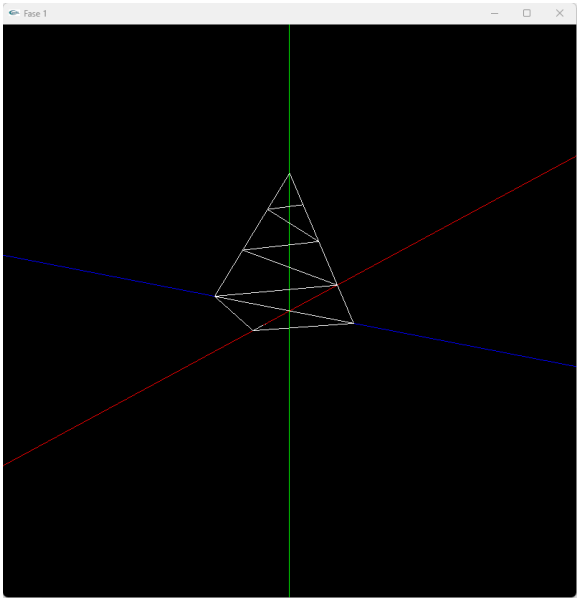


Figure 11: Test1 - Cone with *radius* = 1, *height* = 2, *slices* = 4, *stacks* = 3, fov = 60

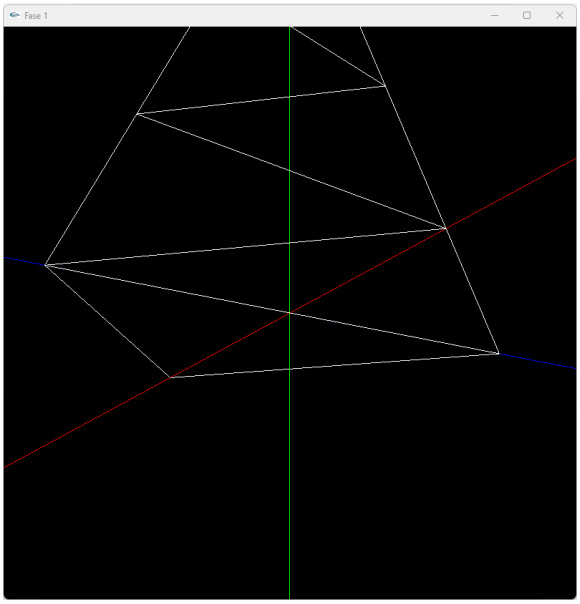


Figure 12: Test2 - Cone with *radius* = 1, *height* = 2, *slices* = 4, *stacks* = 3, fov = 20

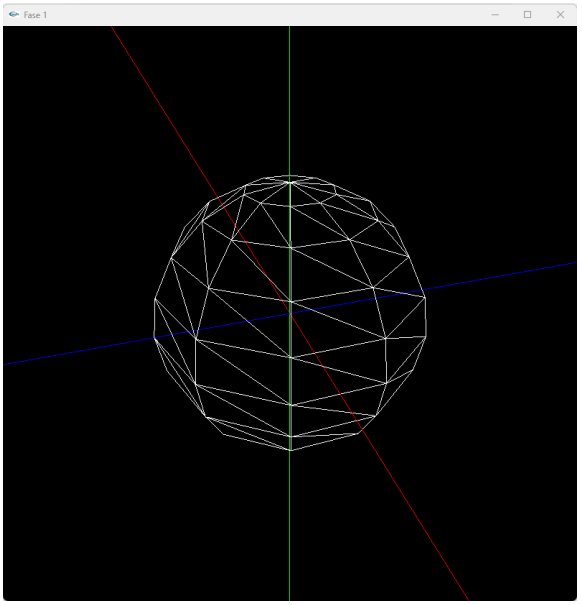


Figure 13: Test3 - Sphere with radius = 1, *slices* = 10, *stacks* = 10, fov = 60

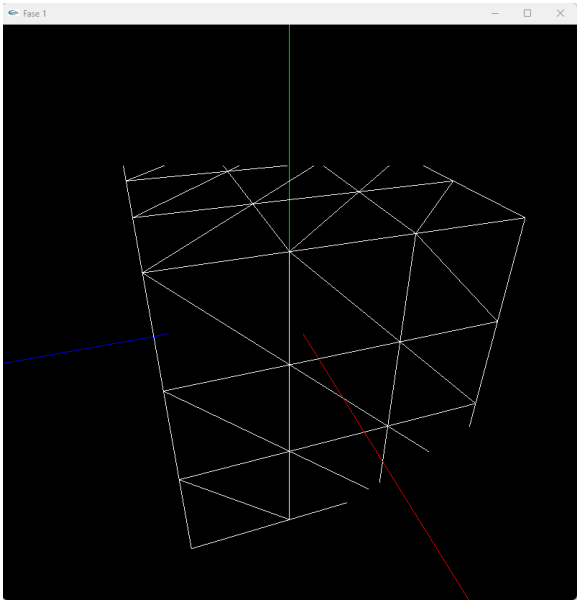


Figure 14: Test4 - Box with *length* = 2, *grid* = 3, *fov* = 60, *far* = 3.5

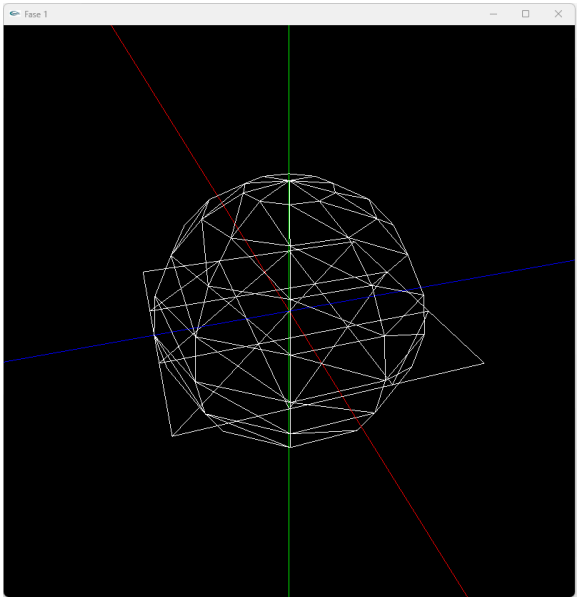


Figure 15: Test5 - Sphere with radius = 1, *slices* = 10, *stacks* = 10; Plane with *length* = 2, *divisions* = 3

References

1. Ahn, S. H.: OpenGL Sphere, `https://www.songho.ca/opengl/gl_sphere.html`