

Relatório

Fase 1

8-Março-2024
Computação Gráfica

Luís Ribeiro, A100608
Martim Félix, A100647
Diogo Santos, A100714

INTRODUÇÃO

A primeira fase do trabalho prático de Computação Gráfica para o ano letivo de 2023/24 propõe a construção de uma mini engine 3D. Esta fase é essencial para estabelecer as bases sobre as quais as etapas seguintes serão desenvolvidas, incorporando progressivamente conceitos e funcionalidades mais avançadas.

Denominada "Primitivas Gráficas", esta etapa concentra-se em gerar ficheiros que contenham informações dos modelos (especificamente, os vértices) e no desenvolvimento do motor que, lendo um ficheiro de configuração em XML, seja capaz de visualizar os modelos. É requerida a criação de duas aplicações distintas: uma dedicada à geração dos ficheiros de modelos, que opera independentemente do motor e que aceita como parâmetros o tipo de primitiva gráfica a ser gerada, os parâmetros necessários para a sua criação e o ficheiro de destino para o armazenamento dos vértices (denominada de generator); e o motor (engine), que processa um ficheiro de configuração em XML detalhando as configurações da câmara e os ficheiros dos modelos previamente gerados para serem carregados.

As primitivas gráficas a serem implementadas nesta fase incluem:

- ⇒ **Plano:** um quadrado no plano XZ, centrado na origem e subdividido nas direções X e Z.
- ⇒ **Caixa:** com especificação das suas dimensões e número de divisões por aresta, centrada na origem.
- ⇒ **Esfera:** definida pelo raio, número de fatias e pilhas, igualmente centrada na origem.
- ⇒ **Cone:** caracterizado pelo raio da base, altura, número de fatias e pilhas, com a base posicionada no plano XZ.
- ⇒ **Cilindro (Extra):** é produzido pelo raio das bases, altura e fatias. O centro da base inferior está apoiado no plano XZ e centrado na origem.

Este segmento é determinante para que nós ganhássemos familiaridade com os conceitos básicos de modelação 3D, a estrutura de dados de um motor gráfico e a gestão de ficheiros e configurações através de XML. Como dito anteriormente, alcançar os objetivos desta fase prepara o terreno para enfrentar desafios mais complexos em etapas futuras, que introduzirão transformações geométricas, superfícies cúbicas, Buffers de Objectos de Vetores (VBOs), normais, coordenadas de textura, entre outros element

Estrutura do projeto

O projeto de Computação Gráfica está organizado de maneira lógica e funcional, dividido em componentes principais que facilitam a compreensão, desenvolvimento e manutenção do código. A seguir, detalhamos os diretórios e ficheiros principais que compõem a estrutura do projeto:

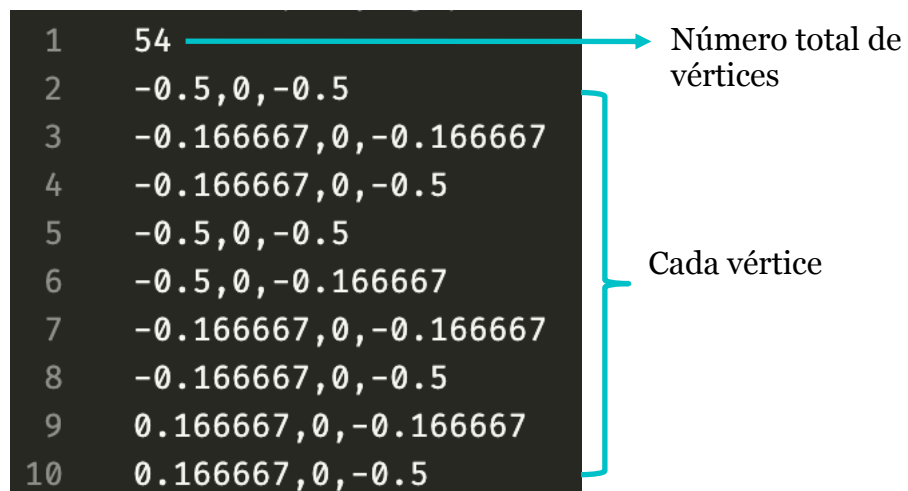
- **include** - Este diretório armazena os ficheiros de cabeçalho (.hpp), que declaram as funções, classes e métodos utilizados nos diferentes módulos do projeto:
 - *generators.hpp*: Define a interface para os geradores das primitivas gráficas.
 - *parser.hpp*: Contém as declarações para o parsing do ficheiro XML de configuração.
 - *point.hpp*: Define a estrutura de dados para um ponto no espaço 3D.
 - *tinyxml2.h*: Ficheiro de cabeçalho para a biblioteca de parsing de XML utilizada no projeto.
- **src** - O diretório fonte, onde o código-fonte do projeto é armazenado:
 - **engine** - Contém os ficheiros fonte do motor de renderização 3D:
 - *engine_main.cpp*: Ponto de entrada para a aplicação do motor gráfico.
 - *parser.cpp*: Implementação das funções declaradas em *parser.hpp*.
 - **generator** - Inclui os ficheiros fonte responsáveis pela geração das primitivas:
 - *box.cpp*, *cone.cpp*, *plane.cpp*, *sphere.cpp*: Implementam os geradores para as primitivas gráficas correspondentes.
 - *generator_main.cpp*: Ponto de entrada para a aplicação do gerador de primitivas.
- **shared** - Contém ficheiros compartilhados entre diferentes partes do projeto:
 - *point.cpp*, *tinyxml2.cpp*: Implementações das classes e funções declaradas em **include**.
- **Raiz do Projeto** - Na raiz, encontramos ficheiros de configuração e descrição do projeto:
 - *CMakeLists.txt*: Ficheiro de script para o sistema de construção CMake, que automatiza o processo de compilação.
 - Ficheiros como *plane.3d* que contêm os dados de primitivas gráficas utilizadas

Generator

A função desta aplicação reside em criar os pontos necessários para formar os triângulos que compõem as várias formas geométricas, recorrendo para isso a métodos de cálculo específicos. Uma vez criados, os vértices são armazenados num arquivo com a extensão .3d, o qual é mencionado dentro de um documento XML. O processo de compilação do projeto é facilitado pelo uso de um arquivo CMakeLists, que permite a geração do código através do uso do cmake.

Ficheiro 3d

O formato dos ficheiros .3d mantém-se consistente para todas as primitivas gráficas. Cada ficheiro inicia com uma linha que indica a contagem total de vértices. Subsequentemente, as linhas seguintes listam individualmente cada vértice, colocados um em cada linha e distinguidos entre si por uma vírgula. Uma figura exemplificativa acompanha para visualizar melhor esta configuração do ficheiro.



O diagrama mostra um ficheiro .3d com 10 linhas. A primeira linha contém o número 54, que é apontado por uma seta vermelha com o rótulo 'Número total de vértices'. As linhas seguintes contêm coordenadas de vértices, separadas por vírgulas. Uma bracejada vermelha engloba as linhas 2 a 10, com o rótulo 'Cada vértice'.

Linha	Conteúdo
1	54
2	-0.5,0,-0.5
3	-0.166667,0,-0.166667
4	-0.166667,0,-0.5
5	-0.5,0,-0.5
6	-0.5,0,-0.166667
7	-0.166667,0,-0.166667
8	-0.166667,0,-0.5
9	0.166667,0,-0.166667
10	0.166667,0,-0.5

Figura 1: Ficheiro corresponde ao plane.3d

Primitivas

Para o projeto atual, foi definido como objetivo a programação de um conjunto de formas tridimensionais fundamentais que são utilizadas na construção de cenários virtuais em 3D. As formas básicas incluem uma 'box', um plano, uma esfera e um cone. Foi também incorporado um cilindro à lista de formas disponíveis. Estes objetos são gerados por meio de funções que processam parâmetros específicos, tais como dimensões e subdivisões, para assegurar a correta representação geométrica. A validade dos dados inseridos é essencial, sendo descartados valores que não sejam positivos. A geração de cada forma é realizada através de instruções definidas para o gerador de modelos, que incluem os parâmetros necessários para cada tipo de primitiva, como raio, altura e número de divisões.

Os comandos possíveis são os seguintes:

- ⇒ plane length divisions
- ⇒ box dimension divisions
- ⇒ sphere radius slices stacks
- ⇒ cone radius height slices stacks
- ⇒ cylinder radius height sides

Sphere

Como referido anteriormente, para além do habitual raio da esfera temos também parâmetros *slices* e *stacks*. Esta maneira de construir uma esfera é bastante comum, e é aliás a utilizada pela função disponível **glutWireSphere()**. Na imagem abaixo dá para perceber isto mesmo:

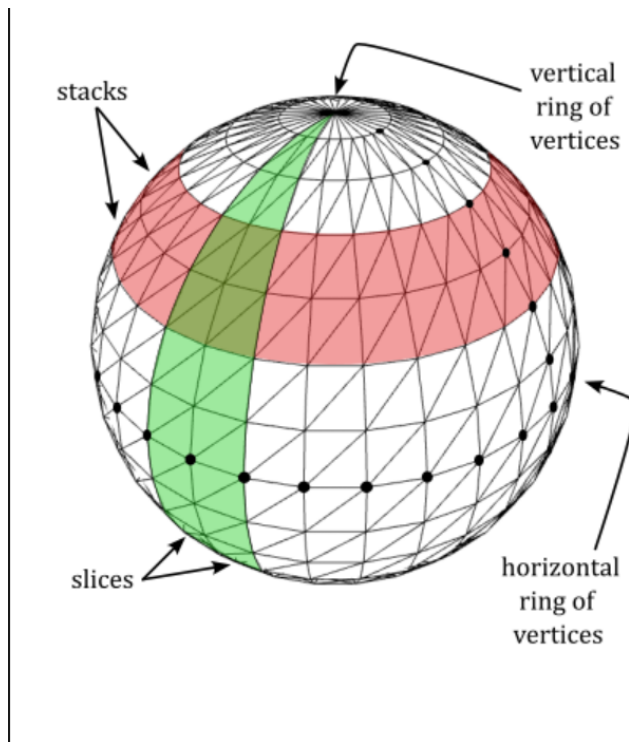


Figura 2: Constituição da esfera

Cada segmento (*stack* numa *slice*) é composto por um quadrilátero, tal como praticamente todas as figuras geométricas feitas até agora, exceto no topo e no fim, o "polo norte" e "polo sul" pois aqui é como se dois pontos consecutivos do quadrilátero convergissem num só.

Para agora começarmos a gerar os pontos, vamos começar pelo topo da esfera. Vamos também definir dois ângulos, *theta* e *phi* que serão os abaixo representados.

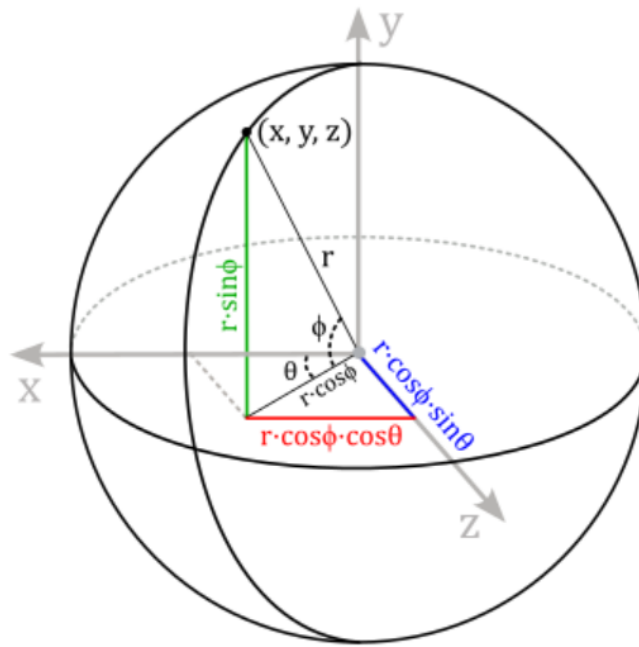


Figura 3: Coordenadas cartesianas de um ponto de uma esfera

Como é possível ver na imagem, determinar pontos fica bastante simples, mas é importante referir que utilizamos um sistema de eixos diferente do demonstrado, pois temos o eixo X e Z trocados:

$$\begin{aligned}x &= \text{raio} * \cos(\text{phi}) * \sin(\text{theta}) \\y &= \text{raio} * \sin(\text{phi}) \\z &= \text{raio} * \cos(\text{phi}) * \cos(\text{theta})\end{aligned}$$

Sabendo também o número de *slices* e *stacks* conseguimos calcular o *step* que ambos os ângulos tomam para progredir:

$$\text{phi_step} = (\pi) / \text{num_de_stacks}$$

Desenhámos as *stacks* ao longo do eixo Y, sendo assim 180° ou então π . Dividindo pelo número de *stacks* temos o *step* pretendido.

$$\text{theta_step} = 2 * (\pi) / \text{num_de_slices}$$

Semelhante ao anterior, mas desta vez estamos a desenhar em torno do eixo Y, fazendo 360° ou então 2π .

O processo agora torna-se bastante semelhante ao de figuras anteriores, temos de descobrir os 4 pontos que fazem um segmento. Cada ponto segue a fórmula para as suas coordenadas mostrada anteriormente. Se fizermos as combinações dos ângulos *phi*, *theta* e *phi + phi_step*, *theta + theta_step* temos os 4 pontos que precisávamos.

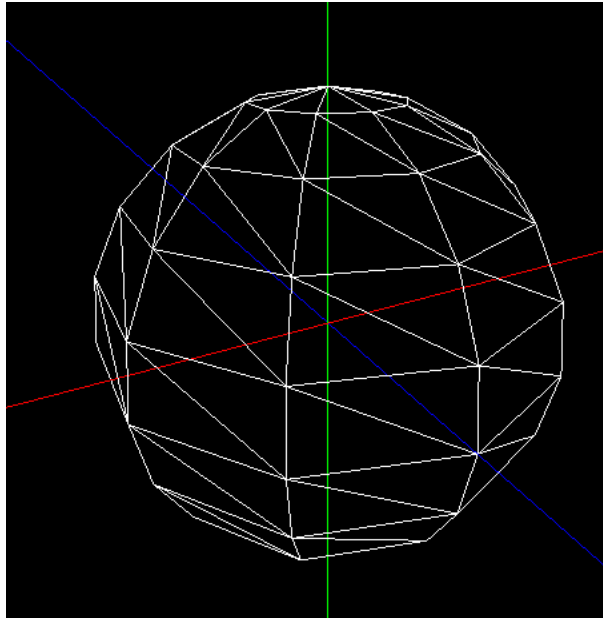


Figura 4: Esfera gerada pelo programa (1,8,8)

Cone

Para modelar um cone em computação gráfica, utilizamos parâmetros fundamentais: o raio da base (*radius*), a altura (*height*), o número de fatias verticais (*slices*) e o número de camadas horizontais (*stacks*). A base do cone é um círculo dividido em *slices*, e a altura é segmentada em *stacks*, permitindo uma representação detalhada do cone.

Cálculo dos Ângulos e Coordenadas:

Cada *slice* representa uma fatia do círculo da base, e os ângulos para cada *slice* são calculados como segue:

Ângulo para a *slice* atual: $angle = 2 * \pi * i / slices$

Ângulo para a próxima *slice*: $nextAngle = 2 * \pi * (i + 1) / slices$

Onde *i* varia de 0 a *slices* - 1.

Base do Cone:

Para cada *slice* na base, criamos um triângulo que conecta dois pontos na borda da base ao centro do círculo (0,0,0), utilizando as coordenadas polares para converter os ângulos em coordenadas cartesianas:

Centro da base: (0, 0, 0)

Ponto na borda da base: $(radius * \cos(angle), 0, radius * \sin(angle))$

Ponto na borda da próxima *slice*: $(radius * \cos(nextAngle), 0, radius * \sin(nextAngle))$

Corpo do Cone:

Para modelar o corpo do cone, segmentamos a altura em *stacks*, onde cada camada é uma circunferência cujo raio diminui linearmente do *radius* na base até o no vértice superior do cone.

Para cada camada *j*, calculamos:

Altura atual: $currentHeight = height * j / stacks$

Altura da próxima camada: $nextHeight = height * (j + 1) / stacks$

Raio atual: $currentRadius = radius * (1 - j / stacks)$

Raio da próxima camada: $nextRadius = radius * (1 - (j + 1) / stacks)$

Utilizando estes valores, determinamos as coordenadas dos quatro pontos que definem cada segmento (um quadrilátero distorcido que se aproxima a um triângulo em direção ao vértice):

$p1 = (currentRadius * \cos(angle), currentHeight, currentRadius * \sin(angle))$

$p2 = (nextRadius * \cos(angle), nextHeight, nextRadius * \sin(angle))$

$p3 = (currentRadius * \cos(nextAngle), currentHeight, currentRadius * \sin(nextAngle))$

$p4 = (nextRadius * \cos(nextAngle), nextHeight, nextRadius * \sin(nextAngle))$

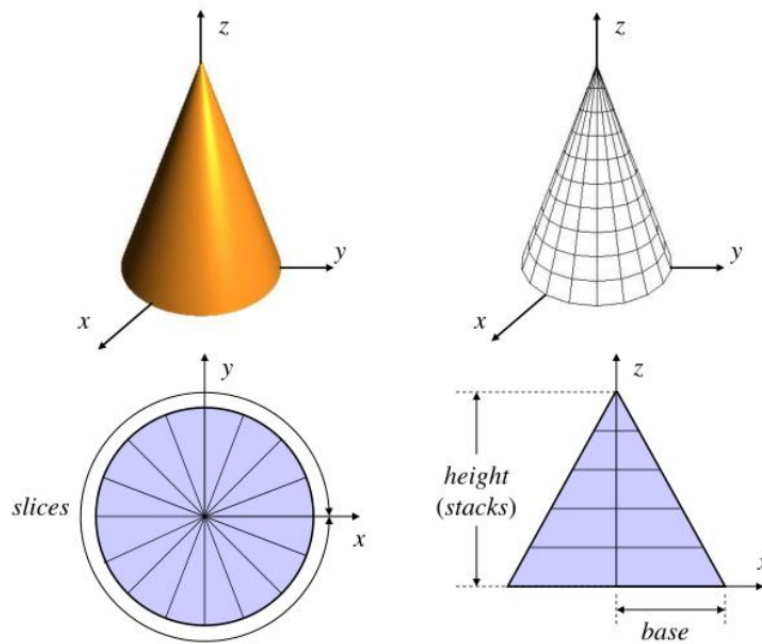


Figura 5: Representação de um Cone e sua Decomposição em Primitivas Gráficas

Cada par de triângulos adjacentes formado por esses pontos ($p1, p2, p4$) e ($p1, p4, p3$) compõe uma fatia do cone. Repetindo esse processo para cada *slice* e *stack*, obtemos uma representação detalhada do cone.

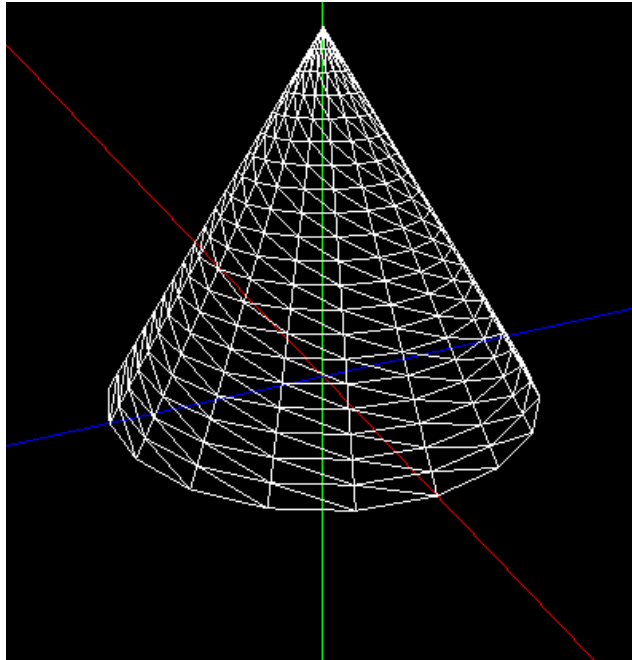


Figura 6: Cone de radius = 1, height = 2, slices = 20 e stacks = 20

Box

Para a modelagem de uma caixa tridimensional em computação gráfica, utilizamos um método sistemático que se baseia na divisão da caixa em planos que compõem as faces todas: XZ (topo e base), XY (frentes e costas), e YZ (lados). A caixa é essencialmente formada por um conjunto de quadriláteros, cada um subdividido em triângulos para simplificar o desenho gráfico. Vamos detalhar a criação de cada face utilizando o código fornecido como base.

Plano XZ (Topo e Base):

Os planos XZ representam o topo e a base da caixa. Para gerar os vértices desses planos, calculamos uma malha de quadrados subdivididos, baseados na **dimensão** total da caixa e no número de **divisões** desejadas.

- ⇒ A variável *step* representa a distância entre cada divisão, calculada por **dimensão / divisões**.
- ⇒ *halfDimension* é a metade da dimensão total da caixa, utilizada para centralizar a malha.

Para cada quadrado na malha do plano XZ:

- ⇒ *x0* e *z0* são as coordenadas iniciais do quadrado.
- ⇒ *x1* e *z1* são as coordenadas finais, derivadas pela adição do *step* às coordenadas iniciais.

A partir desses pontos, formamos dois triângulos por quadrado:

- Um triângulo inferior e um superior para a face de topo.
- Um triângulo inferior e um superior invertidos para a face de base.

Plano XY (Frentes e Costas):

O plano XY forma as faces frontal e traseira da caixa. O processo é análogo ao plano XZ, porém, neste caso, movemo-nos ao longo dos eixos X e Y, mantendo Z constante para cada face.

Para cada quadrado na malha do plano XY:

- ⇒ Utilizamos x_0 , y_0 (pontos iniciais) e x_1 , y_1 (pontos finais) para definir os limites de cada quadrado.
- ⇒ Formamos dois triângulos: um inferior e um superior para cada quadrado, tanto na face frontal quanto na traseira.

Plano YZ (Lados):

Os planos YZ formam os lados esquerdo e direito da caixa. Similarmente aos planos anteriores, a malha é gerada movendo-nos ao longo dos eixos Y e Z, com X mantido constante para cada lado.

Para cada quadrado na malha do plano YZ:

- ⇒ y_0 e z_0 definem os pontos iniciais, enquanto y_1 e z_1 definem os pontos finais de cada quadrado.
- ⇒ Dois triângulos são formados para cada quadrado: um inferior e um superior, repetidos para ambos os lados da caixa.

Construção Final da Caixa:

Após a geração de vértices para cada um dos planos (XZ, XY, YZ), combinamos todos para formar a caixa completa. Cada conjunto de vértices é inserido numa lista final (*box*), que representa a malha total da caixa.

Este método detalhado permite uma flexibilidade considerável na criação de caixas, adaptando a quantidade de detalhes através do número de *divisões*, e ajustando a *dimensão* total para escalar o objeto conforme necessário.

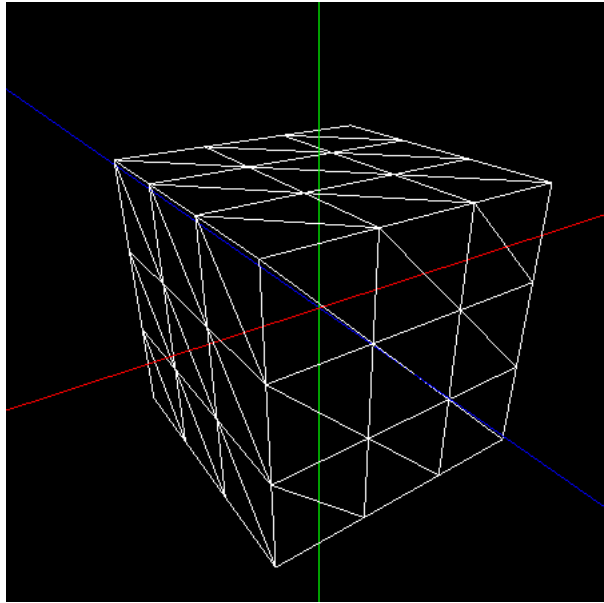


Figura 7: Cubo de dimension = 2 e divisions = 3

Plane

A função que trata de construir os pontos é **generatePlane(int length, int divisions)**.

Como o plano está centrado na origem, começamos por calcular metade do seu comprimento e o comprimento de cada passo (*step*), e utilizando a variável de ciclo conseguimos calcular as coordenadas dos 4 pontos utilizados para desenhar um segmento. Cada segmento é um quadrado feito por 2 triângulos, portanto após o X_0 e o Z_0 serem calculados ($\text{metadeComprimento} + \text{variávelDeCiclo} * \text{step}$), X_1 e Z_1 é apenas $X_0 + \text{step}$ e $Z_0 + \text{step}$, respectivamente. Isto é repetido divisions^2 .

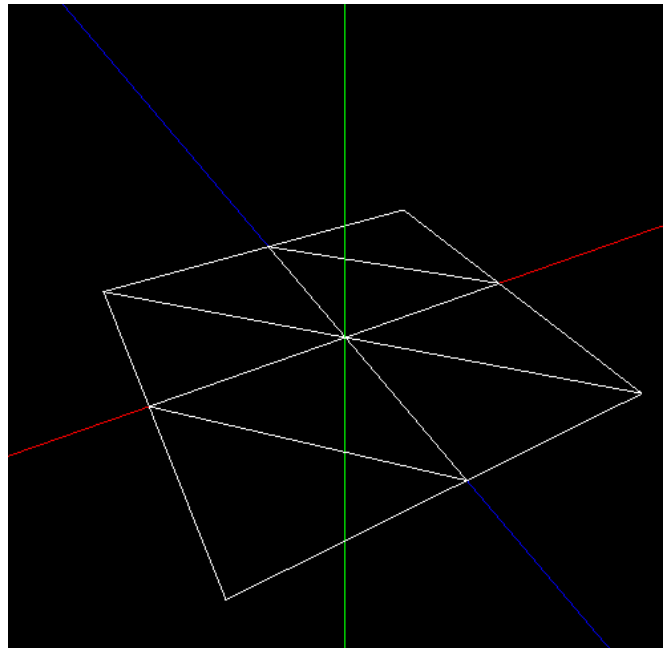


Figura 8: Plano de lenght = 2 e divisions = 2.

Cylinder – Extra

Como extra incluímos também a possibilidade de criar um cilindro a partir do *generator* sendo fornecidos os parâmetros *radius*, *height* e *slices* que definem o número de divisões das circunferências que fazem o topo e a parte de baixo do cilindro. No fundo, aquilo que é desenhado é um prisma com n lados em cada ponta. Quando o número n de lados é suficientemente grande o prisma resultante assemelha-se a um cilindro.

Cálculo dos Vértices:

Para definir os vértices necessários para formar o cilindro procede-se à definição do mesmo, *slice* a *slice*. Iteramos então pelo número de slices.

Os pontos de centro das circunferências são sempre utilizados e por isso mantêm-se definidos durante a execução de todo o programa. Os outros 4 pontos são definidos de forma iterativa e são calculados a partir do ângulo da slice que se pretende desenharmos. A cada iteração este ângulo é incrementado permitindo calcular um novo par de vértices que permitem definir os 4 triângulos que compõem a fatia. Como 2 desses pontos se mantêm de uma iteração para a seguinte, a implementação reflete isso mesmo. Cada vértice ou ponto usado para a definição do cilindro é calculado apenas uma vez.

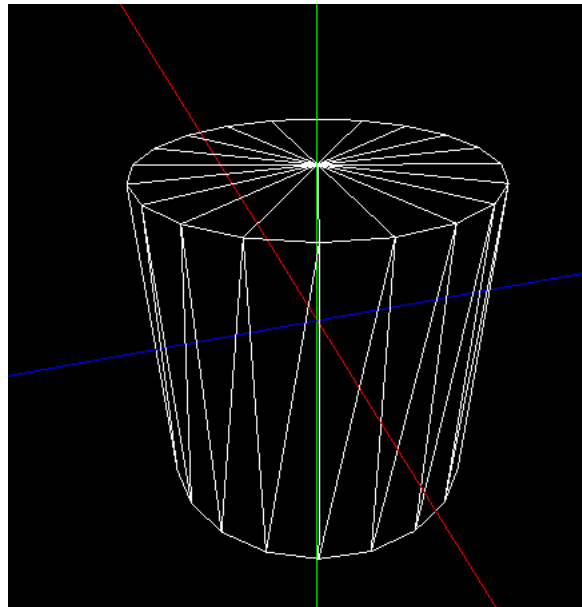


Figura 9: Cilindro com $radius = 1$, $height = 2$ e $slices = 20$

Engine

Estrutura

O segmento da aplicação Engine desempenha um papel crucial dentro do contexto do nosso software de visualização 3D. Esta componente está encarregue de interpretar ficheiros XML, utilizando a biblioteca tinyXML para essa finalidade. O processo inicia-se com a leitura única do ficheiro XML, a partir do qual a Engine identifica as primitivas definidas e procede à leitura dos respectivos ficheiros .3d que contêm os vértices das formas.

Para acomodar a variedade de primitivas e os seus dados associados na memória, recorreremos à implementação das classes Primitive e Point. A classe Point é responsável por representar um ponto no espaço 3D através de três coordenadas – x, y e z. Por sua vez, a classe Primitive serve como um recipiente para um vetor de pontos, correspondendo cada instância a uma primitiva única, seja ela um plano, uma caixa, uma esfera, um cone ou um cilindro, contendo todas as coordenadas necessárias para descrever as faces da primitiva em questão.

Para compilar a aplicação Engine, utilizamos um ficheiro CMakeLists.txt que integra todos os ficheiros necessários, incluindo os mencionados anteriormente, e prepara o projeto para a geração via cmake. Este método assegura que todos os componentes da aplicação sejam corretamente compilados e linkados, permitindo um funcionamento harmonioso do programa.

Em relação ao código em si, a estrutura da aplicação Engine é delineada no ficheiro engine_main.cpp, onde a lógica para a renderização das formas em 3D é implementada. Através das funções definidas neste ficheiro, estabelecemos as configurações iniciais da câmara, os parâmetros de visualização e os comandos para interação do utilizador, que permitem navegar pelo ambiente 3D. O código facilita a interação com a cena, permitindo ao utilizador ajustar a visualização ao seu critério.

Extras

No desenvolvimento da aplicação Engine, foram introduzidas funcionalidades adicionais não explicitadas no documento de requisitos. Entre estas, destacamos a inclusão do movimento da câmara, que acreditamos enriquecer significativamente o projeto desenvolvido.

Câmara

Nesta fase decidimos acrescentar como extra para além do cilindro uma câmara em modo *explorer*, significando isto que a câmara anda em torno de uma espécie de esfera invisível. Depois de se perceber os cálculos da esfera, fazer a câmara neste modo torna-se trivial pois funciona da mesma forma:

$$\begin{aligned}camX &= cam_radius * cos(cam_phi) * sin(cam_theta) \\ camY &= cam_radius * sin(cam_phi) \\ camZ &= cam_radius * cos(cam_phi) * cos(cam_theta)\end{aligned}$$

Para já temos ambos os ângulos a começar em 0 e o raio em 3. Na próxima fase pretendemos que a câmara parta já do ponto inicial. O utilizador pode usar o teclado para andar em torno do plano e também pode dar **zoom in** e **zoom out** (que é apenas aumentar e diminuir o *cam_radius*)

Conclusão

Concluída a primeira fase do projeto, torna-se importante realizar uma avaliação crítica do que foi feito. Ao finalizar esta etapa, observamos que houve elementos destacáveis, como a inclusão de uma primitiva adicional – o cilindro, a implementação de movimentação da câmara.

Foram sentidas algumas dificuldades, principalmente no tratamento da posição inicial da câmara em modo de exploração como referido na sua secção. Esta dificuldade advém principalmente de não ser tão intuitivo a noção dos ângulos e de como chegar a um através do ponto inicial com o *arctan*. No entanto prevemos ter isto resolvido para a próxima fase.

Em suma, o resultado do esforço empregado é visto de forma positiva, uma vez que os desafios encontrados foram devidamente ultrapassados e todos os objetivos propostos inicialmente foram atingidos.