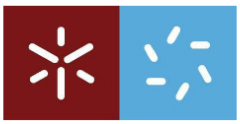

Universidade do Minho

Licenciatura em Ciências da Computação

Fase 1 - Primitivas Gráficas

Computação Gráfica

Grupo 7

Cláudia Rego Faria (A105531)

Diogo José Borges Dias (A102943)

Maria Inês Barros de Matos (A102937)

Patrícia Daniela Fernandes Bastos (A102502)

2 de março de 2025

Conteúdo

Capítulo 1

Introdução

Este relatório é o primeiro de um trabalho de quatro partes, no âmbito da UC Computação Gráfica da Licenciatura em Ciências da Computação e referente ao ano letivo de 2024/2025.

Para a Primeira Fase, foram desenvolvidas duas aplicações: Generator e Engine. Esta fase teve como objetivo a aprendizagem e implementação prática do *OpenGL* e da linguagem C++ através da geração de figuras geométricas (as Primitivas Gráficas) e a renderização de cenas.

Capítulo 2

Estrutura do projeto

Para uma melhor organização do projeto, este foi dividido nos seguintes diretórios:

- **3d**: armazena os ficheiros criados pelo generator que serão posteriormente utilizados pelo engine para a renderização dos modelos
- **engine**: contém a implementação do motor gráfico responsável pela leitura e visualização dos modelos 3d gerados
- **gen**: inclui o código responsável pelo cálculos dos pontos necessários para a construção das primitivas e pela geração dos ficheiros 3d
- **data_structs**: contém as structs desenvolvidas para um melhor e mais fácil funcionamento do projeto
- **test_files**: contém os ficheiros de teste que serão utilizados ao longo das diferentes fases do projeto
- **TinyXml**: contém a biblioteca TinyXML utilizada para auxiliar na leitura e manipulação de ficheiros .xml

Capítulo 3

Generator

O generator é responsável pelo cálculo dos pontos necessários para a construção de primitivas gráficas em formato .3d. Todas as primitivas são representadas por triângulos, ou seja, conjuntos de três pontos, que vão permitir a sua renderização posterior.

O ficheiro gerado pelo generator vai seguir um formato padrão onde a primeira linha indica o número total de pontos presentes no ficheiro e as restantes linhas vão conter as coordenadas dos pontos. Os pontos encontram-se especificados em cada linha por três valores do tipo float separados por vírgulas, onde cada um desses valores representa a coordenada x, y e z ,respectivamente.

Cada grupo de três linhas consecutivas no ficheiro vai representar um triângulo.

Figura 3.1: Excerto de um ficheiro .3d

O ficheiro é gerado através da função *writeToFile*.

A execução do generator é feita através do seguinte comando:

./ generator [primitiva] [argumentos] [fileName.3d]

Onde:

- **primitiva** indica a primitiva a ser gerada, tendo como opções: "box", "plane", "sphere" e "cone"
- **argumentos** corresponde aos parâmetros necessários para definir as características das primitivas:
  - *box*: [*length*] [*divisions*]
  - *plane*: [*length*] [*divisions*]
  - *sphere*: [*radius*] [*slices*] [*stacks*]
  - *cone*: [*radiu*] [*height*] [*slices*] [*stacks*]
- ***filename.3d*** indica o nome do ficheiro a ser criado e onde os pontos vão ser escritos

Em seguida, irá ser explicado com mais detalhe a estratégia adotada para o cálculo dos pontos de cada uma das primitivas.

3.1 Caixa

A função *genBox* recebe como parâmetros *length* e *grid*.

A caixa é constituída por seis faces, onde cada face é dividida numa grelha de *grid*× *grid* quadrados. Cada quadrado é constituído, por sua vez, por dois triângulos. Para a sua construção é necessário o uso das seguintes variáveis:

- part: divide *length* por *grid* de forma a obter o tamanho dos lados de cada subquadrado;
- half: divide a altura em metade.

Para encontrar a coordenada mais à esquerda, mais abaixo e mais atrás calcula-se:

a = -*half* + i * *part*;

b = -*half* + j * *part*;

onde i e j são parte de loops que vão incrementando de 0 a grid-1.

Desta forma é assegurado que cada quadrado está a ser formado à volta da origem de forma simétrica.

Para cada quadrado em cada face, a função gera 6 vértices (representando 2 triângulos) que vão ser adicionados aos *controlPoints*, que armazenam os vértices da caixa. Este processo é repetido para todas as seis faces, gerando assim a caixa completa.

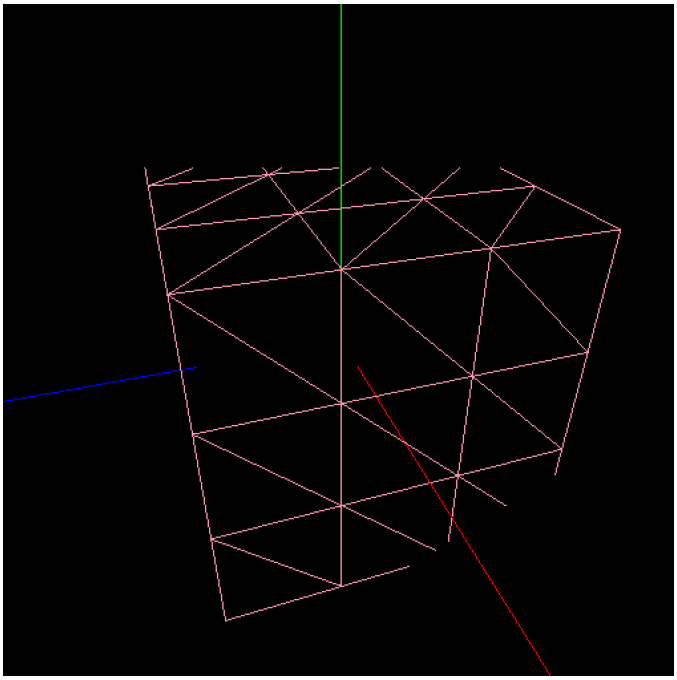

Figura 3.2: Caixa com 2 de *lenght* e 3 de *grid* (*test_1_4.xml*)

3.2 Plano

A função *genPlane* é responsável por gerar um plano 3D dividido em pequenos quadrados, que são representados por dois triângulos cada. Esse plano é criado no eixo **XZ**, com altura fixa em $\mathbf{Y = 0}$. A função *genPlane* recebe como parâmetros *length* (o tamanho do plano em ambas as direções, x e z) e *divisions* (o número de divisões do plano, isto é, quantos quadrados ele terá).

O plano é dividido em pequenos quadrados, e cada quadrado é formado por dois triângulos.

Dividindo o plano em partes menores:

- *part*: divide o comprimento total pelo número de divisões para obter o tamanho de cada quadrado.
- *half*: representa a metade do comprimento, para centralizar o plano na origem (0,0,0).

Cada quadrado do plano é definido por quatro pontos:

- (x0, z0): canto inferior esquerdo
- (x1, z0): canto inferior direito
- (x0, z1): canto superior esquerdo
- (x1, z1): canto superior direito

Para cada divisão (i, j), os pontos são calculados como:

x1 = -*half* + i * *part*

z1 = -*half* + j * *part*

x2 = x1 + *part*

z2 = z1 + *part*

Usam-se esses pontos para formar dois triângulos:

- Triângulo 1: (x0, z0), (x1, z0), (x0, z1)
- Triângulo 2: (x1, z0), (x1, z1), (x0, z1)

Esses triângulos são adicionados aos *controlPoints*, que armazenam os vértices do plano.

Como o plano é paralelo ao eixo XZ, a coordenada y é sempre 0.

Gera-se assim um plano.

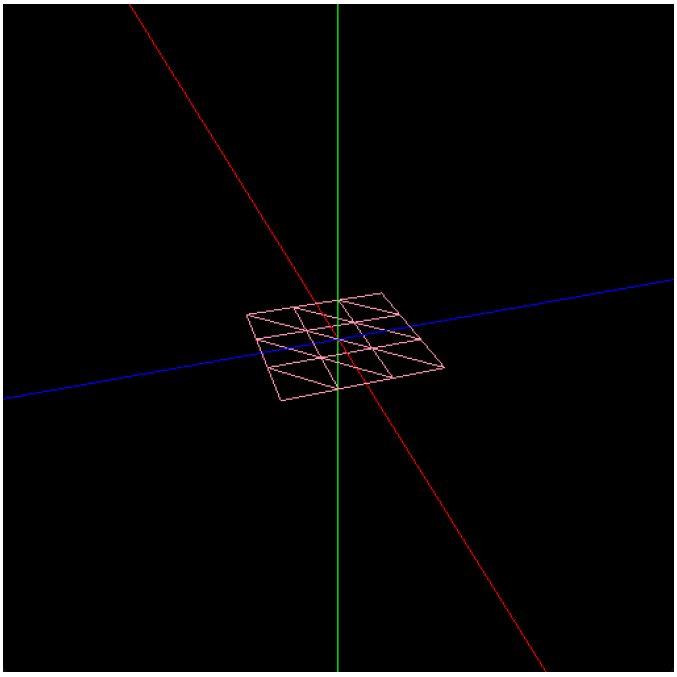

Figura 3.3: Plano com 1 de comprimento dividido em 3x3 quadrados

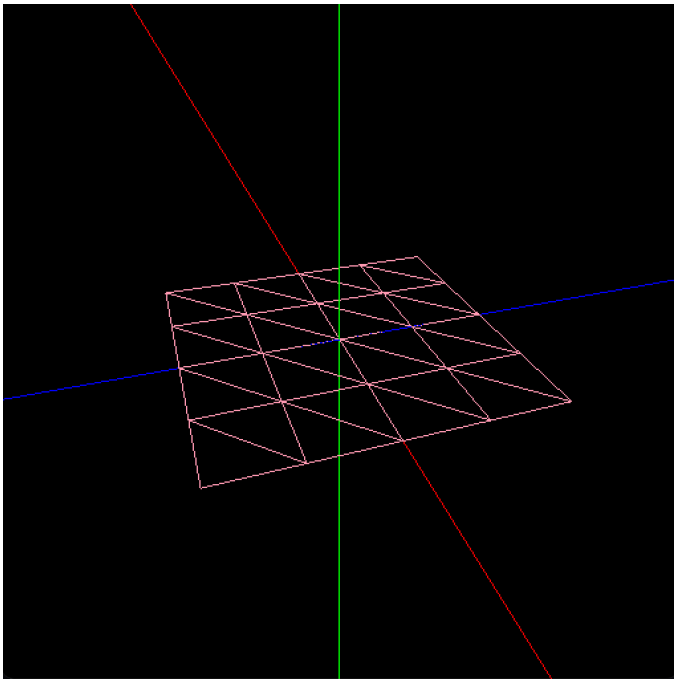

Figura 3.4: Plano com 2 de comprimento dividido em 4x4 quadrados

3.3 Esfera

A função *genSphere* recebe como parâmetros *radius*, *slices* e *stacks*.

A esfera é dividida em ângulos:

- *anglePerSlice*: divide 360° em fatias horizontais iguais;
- *anglePerStack*: divide 180° em pilhas verticais iguais.

Usando coordenadas esféricas para converter ângulos em pontos 3D calcula-se:

- $x = raio \times \sin(beta) \times \sin(alpha)$;
- $y = raio \times \cos(beta)$;
- $z = raio \times \sin(beta) \times \cos(alpha)$.

Através destas fórmulas e considerando beta como o ângulo *anglePerStack* atual e o seguinte, e alpha como o ângulo *anglePerSlice* atual e o seguinte, calculam-se quatro pontos: p1, p2, p3 e p4. Os pontos p1, p2 e p4 formarão o triângulo superior de cada secção, enquanto p1, p3 e p4 formarão o triângulo inferior.

Para cada secção serão formados triângulos que serão adicionados aos *controlPoints*, que vão gerar a esfera.

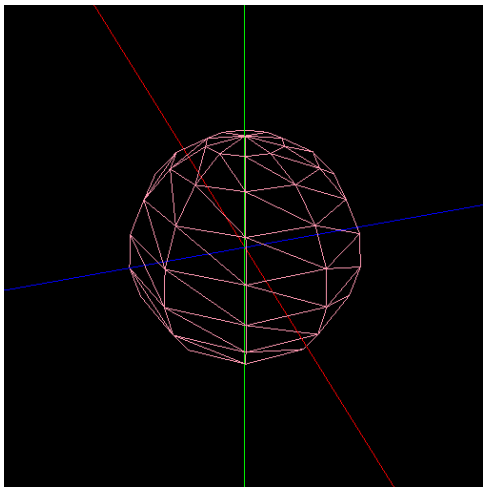

Figura 3.5: Esfera com 1 de *radius*, 10 de *slices* e 10 de *grid* (*test_1_3.xml*)

3.4 Cone

A função *genCone* recebe como parâmetros: *radius*, *height*, *slices* e *stacks*, onde as *slices* correspondem às divisões equivalentes do cone e as *stacks* às divisões equivalentes da altura do cone.

Para cada *slice*, é calculado o ângulo correspondente:

- Ângulo de cada *Slice* (alpha) = $2 \times \pi \div$ Nº de *Slices* (*slices*)

Para cada stack, é necessário calcular a sua altura e raio:

- Altura de cada *Stack* (*heightPerStack*) = Altura do Cone (*height*) ÷ Nº de *Stacks* (*stacks*)
- Ângulo para cada Stack = - (*radius* ÷ *stacks*)

As coordenadas (x,y,z) de cada ponto são calculadas da seguinte forma:

- Y: A altura do cone é recebida como argumento e é fácil perceber qual é a altura dos pontos para cada iteração. Sendo *heightPerStack* o valor da altura total dividido pelo número de *stacks* e *stack* o número da *stack* atual então:
  - y = stack * heightPerStack
- X e Z:
  - x = Raio da Stack * sin (alpha)
  - z = Raio da Stack * cos (alpha)

A construção do cone é um processo iterativo que está dividido em duas partes:

1. Superfície lateral: São gerados, para cada *slice*, e para cada *stack*, 4 pontos: *bottom_left, bottom_right, top_left, top_right* que vão ser usados para criar dois triângulos por *stack* sendo eles:
   - (*top_left, bottom_left, bottom_right*)
   - (*top_left, bottom_right, top_right*)
2. Base do cone: A base do cone é formada por um conjunto de triângulos que ligam o ponto central da base (*base_middle*) a pontos na circunferência inferior do cone. Para cada *slice* são criados 2 pontos *base_bottom_right* e *base_bottom_left* que são ligados ao ponto central *base_middle* formando assim um triângulo.

A função genCone segue esta estratégia e vai armazenando os pontos dos respectivos triângulos ao vetor *controlPoints*, gerando o cone.

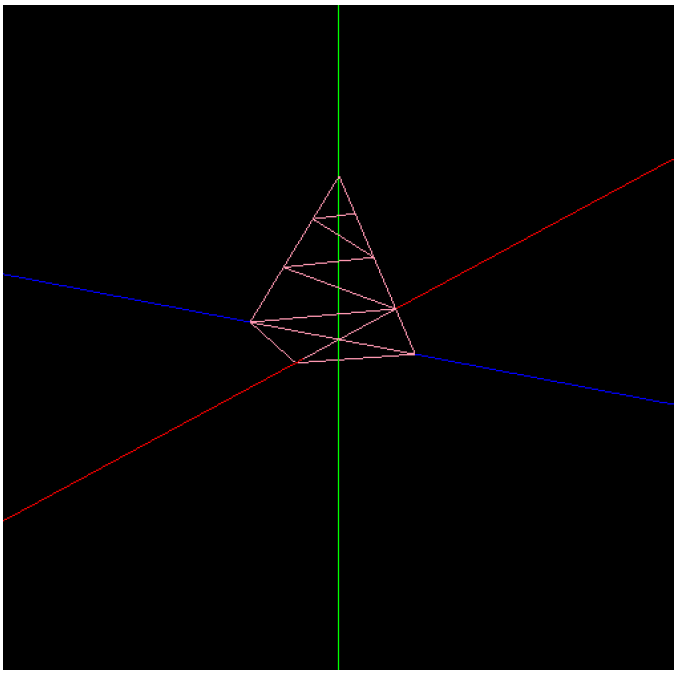

Figura 3.6: Cone com 1 de *radius*, 2 de *height*, 4 de *slices* e 3 de *stacks* (*test_1_1.xml*)

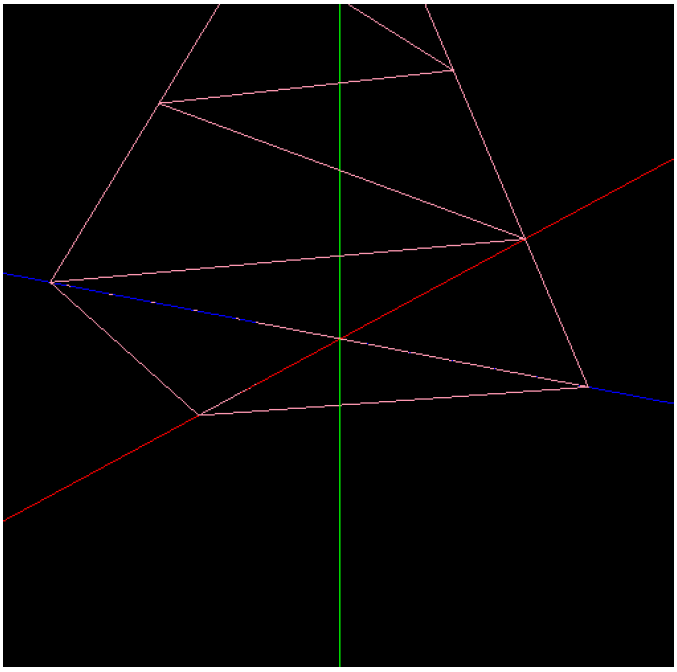

Figura 3.7: Cone com 1 de *radius*, 2 de *height*, 4 de *slices* e 3 de *stacks* (*test_1_2.xml*)

Capítulo 4

Engine

O engine é responsável pela leitura dos ficheiros XML, com o auxílio da *struct Settings*, e pela renderização dos ficheiros 3d.

Após a leitura do ficheiro XML as configurações da câmara são guardadas nas variáveis:

- *height, width*: usadas na inicialização do tamanho da janela
- *camx, camy, camz, lookAtx, lookAty, lookAtz, upx, upy, upz*: usadas na configuração da câmara com o auxílio das variáveis *radius, alpha e beta*. Em que:

$$alpha = acos(camz \div \sqrt{camx \times camx + camz \times camz})$$
$$beta = asin(camy \div radius)$$
$$radius = \sqrt{camx \times camx + camy \times camy + camz \times camz}$$

- *fov, nearPlane, farPlane*: usados na configuração da perspetiva

Com o auxílio da *struct Model* são guardados os pontos necessários para cada figura no vetor de *Models models* para que sejam desenhados os triângulos necessários para cada figura.

A execução do engine é feita através do seguinte comando:

./engine [pathToFile/fileName.xml]

Onde:

- ***fileName.xml*** indica o nome do ficheiro XML de teste a ser lido
- ***pathToFile*** indica o caminho até a pasta onde se encontra o ficheiro XML

De maneira a proporcionar uma experiência mais interativa e intuitiva na observação dos modelos, implementámos algumas funcionalidades para o controlo da câmara.

Movimento Orbital e Rotação

A câmara implementada é do tipo orbital e move-se em torno de um ponto (sendo este definido pelo *lookAt*). A rotação é controlada através das seguintes teclas:

W : Rotação para cima

S : Rotação para baixo

A : Rotação para a esquerda

D : Rotação para a direita

Zoom

Para ajustar a proximidade da câmara ao modelo, utilizan-se:

+ : Zoom in

- : Zoom out

Modos de Desenho

Para facilitar a observação da cena, é possível alternar entre diferentes modos de renderização:

M : Alterna entre preenchimento, linhas e pontos

Capítulo 5

Utilitários

5.1 Structs

Para simplificar o código e facilitar a sua reutilização foram criadas três *Structs*:

1. ***Settings***: guarda as configurações da cena 3D a representar.

   Contém as variáveis:

   - *window* (dimensões da janela);
   - *poscam* (posição da câmara);
   - *lookAt* (direção da câmara);
   - *up*;
   - *projection* (campo de visão fov, plano próximo near, plano distante far);
   - models: (vetor com caminhos para os ficheiros dos modelos 3D).

   Funções principais:

   - *newSettings()*: Cria uma nova estrutura de configurações.
   - *xmlToSettings(filePath)*: Lê as configurações a partir de um ficheiro XML. Para isso, utiliza o TinyXML, ferramenta sugerida pelo docente da UC.
   - *getPaths()*: Retorna a lista de caminhos dos modelos.
   - *setCamPosition()*: Define a posição da câmara.
   - Funções de *get* para obter parâmetros individuais como posição da câmara, vetores *lookAt, up*, e projeção.
   - *deleteSettings()*: Liberta a memória alocada.

2. ***Point:*** contém as variáveis x, y e z.

Funções principais:

- *createPoint()*: Cria um ponto na origem (0,0,0).
- *makePoint(x, y, z)*: Cria um ponto com coordenadas específicas.
- Funções *getX(), getY(), getZ()*: Retornam as coordenadas individuais.

3. ***Model***: contém um modelo composto por um vetor de pontos 3D, que vão gerar uma figura geométrica.

   Funções principais:

   - *createModel()*: Cria um modelo vazio.
   - *makeModel(points)*: Cria um modelo com uma lista de pontos.
   - *getPoints()*: Retorna a lista de pontos do modelo.
   - *addPoint()*: Adiciona um ponto ao modelo.
   - *readFromFile(fileName)*: Lê pontos de um ficheiro para criar o modelo.
   - *writeToFile(controlPoints,fileName)*: Escreve pontos num ficheiro.

5.2 Biblioteca TinyXML

Para a leitura dos ficheiros XML foi optado pelo uso da biblioteca TinyXML devido à sua simplicidade de uso. A biblioteca permitiu a fácil extração das informações necessárias dos ficheiros, como as coordenadas da câmera, dimensões da janela e caminhos para os modelos 3D.

Capítulo 6

Conclusão

Nesta Primeira Fase do trabalho, foi possível consolidar os conhecimentos práticos e teóricos das aulas de Computação Gráfica. Aprendeu-se a utilizar o *OpenGL* juntamente com a linguagem C++, assim como a criar primitivas gráficas, em particular renderização de cenas, geração de figuras geométricas básicas e sistemas de coordenadas.

Os conhecimentos adquiridos nesta fase constituem a fundação essencial para as etapas subsequentes, onde irão ser exploradas técnicas mais avançadas de Computação Gráfica.