

Computação Gráfica (3^a ano de LCC)

Trabalho Prático

1^a Fase

Relatório

Grupo 15

Pedro Manuel Pereira dos Santos	(A100110)
João Manuel Franqueira da Silva	(A91638)
David Alberto Agra	(A95726)
João Pedro da Silva Faria	(A100062)

March 7, 2024

Índice

1	Introdução	3
1.1	Generator	3
1.2	Engine	3
2	Apresentação das soluções	4
2.1	Generator	4
2.1.1	Plane	5
2.1.2	Box	6
2.1.3	Sphere	7
2.1.4	Cone	8
2.2	Engine	9
2.2.1	Variáveis globais	9
2.2.2	Funções auxiliares	9
3	Demonstração	10
3.1	Execução dos programas	10
4	Conclusão	11

Lista de Figuras

2.1	parte de um ficheiro .3d gerado	4
2.2	plane, unit = 2, slices = 15	5
2.3	box, unit = 2.5, slices = 30	6
2.4	sphere, radius = 1.5, slices = 65, stacks = 60	7
2.5	cone, radius = 1, height = 3, slices = 20, stacks = 10	8

Chapter 1

Introdução

No âmbito da unidade curricular de Computação Gráfica, foi-nos proposto o desenvolvimento de duas aplicações utilizando a linguagem C++, recorrendo à ferramenta OpenGL.

O objetivo deste trabalho é desenvolver um sistema para gerar objetos geométricos (planos, caixas, esferas e cones) em 3D e desenhá-los com base nas configurações de uma câmara definidas num arquivo xml.

O gerador, produz os vértices dos objetos, enquanto a engine utiliza o OpenGL para renderizá-los.

Neste relatório, será fornecida uma análise detalhada das decisões e abordagens adotadas para viabilizar a implementação destas aplicações propostas.

1.1 Generator

Deve guardar em ficheiro, todos os vértices necessários para desenhar as primitivas gráficas requisitadas (planos, caixas, esferas ou cones) dadas informações como tamanho, raio, divisões, stacks, etc.

1.2 Engine

Deve ler um ficheiro em formato xml, obter as configurações da câmara e os ficheiros com os vértices armazenados, desenhando então as primitivas correspondentes.

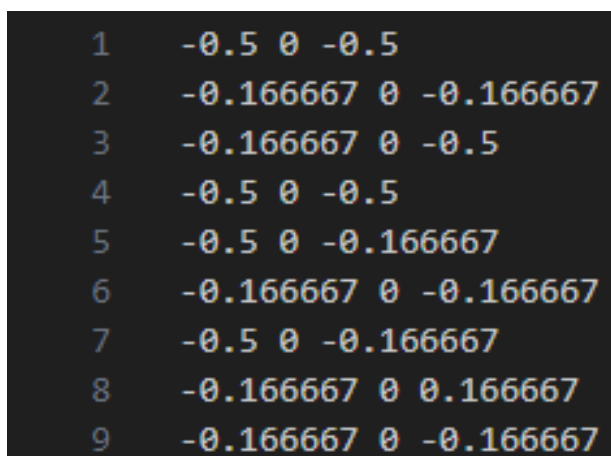
Chapter 2

Apresentação das soluções

Adotamos a ferramenta TinyXML2 para interpretar as informações a partir de um arquivo XML na engine. Além disso, implementamos um conjunto de funções personalizadas nos ficheiros "functions.h" e "functions.cpp" para melhorar a organização do generator.

2.1 Generator

Esta aplicação recebe como input, a figura geometrica e os seus respetivos atributos, bem como o nome do ficheiro onde os pontos serão guardados. Como todas as figuras serão construidas apartir de triângulos, decidimos guardar 3 números em cada linha do ficheiro, visto que cada coordenada é descrita por 3 valores, sendo elas X, Y e Z, respetivamente.



```
1 -0.5 0 -0.5
2 -0.166667 0 -0.166667
3 -0.166667 0 -0.5
4 -0.5 0 -0.5
5 -0.5 0 -0.166667
6 -0.166667 0 -0.166667
7 -0.5 0 -0.166667
8 -0.166667 0 0.166667
9 -0.166667 0 -0.166667
```

Figure 2.1: parte de um ficheiro .3d gerado

Para gerar os pontos para diferentes primitivas, é preciso fornecer ao generator os seguintes parâmetros, dependendo da primitiva em questão: Para o Plano (comprimento, divisões), para a Caixa (comprimento, divisões), para a Esfera (raio, fatias, pilhas) e para o Cone (raio, altura, fatias, pilhas) e finalmente o nome do ficheiro .3d que vai exportar.

Seguimos agora, para uma descrição mais detalhada de como é gerado cada primitiva.

2.1.1 Plane

É um quadrado centrado no plano XZ, dividido em secções, sendo estas também quadrados, que serão desenhados através de 2 triângulos. Para a criação dos pontos desta figura, podemos iterar com dois ciclos. Inicialmente começámos pelo canto do plano, tendo os 4 pontos para construir os dois triângulos, e, em cada iteração arrastamos os pontos para formar os próximos triângulos, e assim sucessivamente.

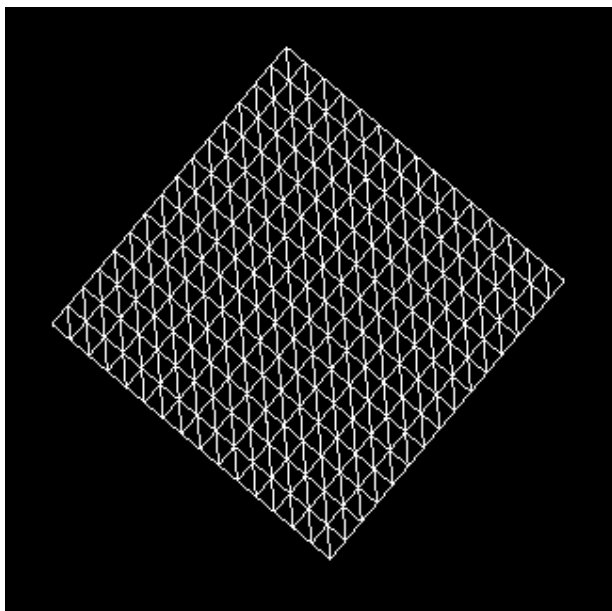


Figure 2.2: plane, unit = 2, slices = 15

2.1.2 Box

É uma caixa centrado na origem, também dividida em várias secções quadradas formadas por dois triângulos. Para obter os seus pontos, podemos também iterar 3 vezes com dois ciclos for. Assumindo que queremos um cubo com comprimento **unit**, na primeira realização dos ciclos obtemos dois planos no plano XZ, com centro de coordenadas $(0, \text{unit}/2, 0)$ e $(0, -\text{unit}/2, 0)$. Na segunda realização dos ciclos obtemos também dois planos no plano XY, com centro de coordenadas $(0, 0, \text{unit}/2)$ e $(0, 0, -\text{unit}/2)$. Por ultimo, obtemos dois planos no plano YZ com centro de coordenadas $(\text{unit}/2, 0, 0)$ e $(-\text{unit}/2, 0, 0)$. Todos os pontos destes planos são obtidos da mesma forma que a os pontos da primitiva **plane**, começando pelos cantos com 4 pontos que representam os primeiros 2 triângulos, e sucessivamente arrastando esses pontos por cada iteração.

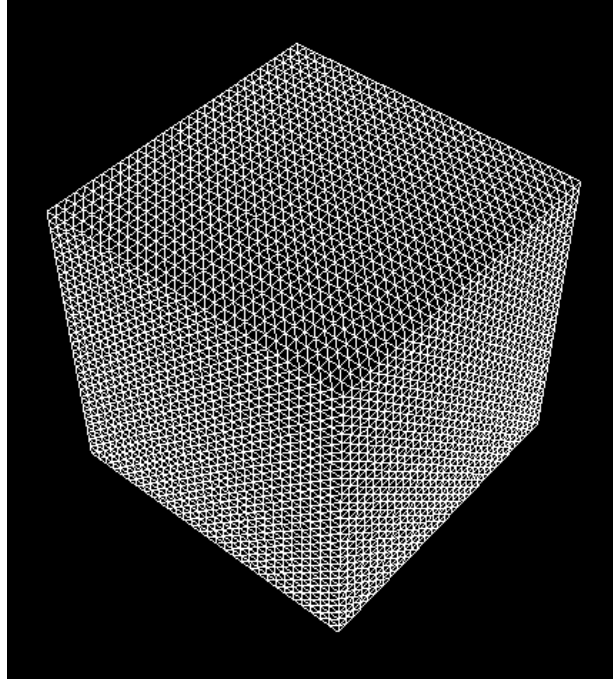


Figure 2.3: box, unit = 2.5, slices = 30

2.1.3 Sphere

È uma esfera centrada na origem, dividida em secções (slices), em que cada uma é dividida por subsecções (stacks). Primeiro itera-se sobre o movimento angular de um círculo. Para cada iteração obtemos os pontos que compõem o topo da esfera, sendo calculados 2 vértices usando coordenadas esféricas com os ângulos longitudinais e latitudinais e dos valores destes na próxima iteração, através do *sen* e *cos*, sendo o outro vértice, o topo da esfera, de coordenada $(0, \text{raio}, 0)$. Usamos o mesmo método para obter os vértices para a base da esfera, desta vez, conectando-os ao vértice mais fundo da esfera, com coordenadas $(0, -\text{raio}, 0)$. Para obter os vértices pertencentes ao corpo da esfera iteramos sobre as stacks e os slices, em que para cada iteração calculamos as coordenadas dos vértices através de coordenadas esféricas, a partir dos ângulos n e i , que representam a respetiva stack e slice que está a ser iterada.

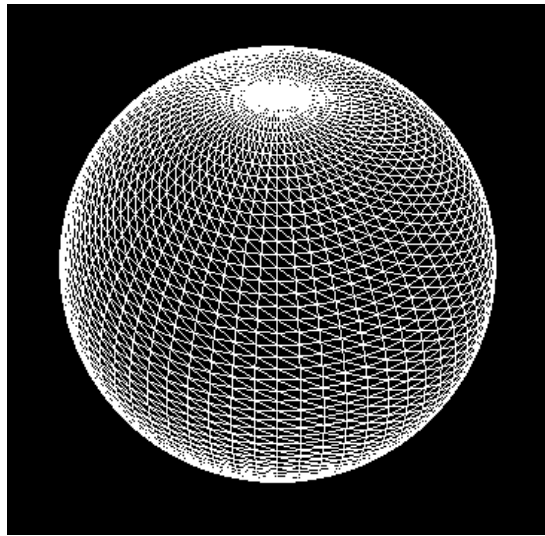


Figure 2.4: sphere, radius = 1.5, slices = 65, stacks = 60

2.1.4 Cone

É um cone com a base centrada no plano XZ dividida em secções (slices), cada uma dividida também por subsecções (stacks). Primeiro itera-se sobre o número de slices, obtendo as coordenadas dos vértices que compõem a base do cone, calculando o *sen* e o *cos* do ângulo *arch_alfa*, conectando esses vértices à origem. Por último, para calcular os vértices que formam o corpo do cone, itera-se sobre as stacks e as slices, para cada slice, calculando as coordenadas dos 4 pontos que irão compor os dois triângulos. Os valores x e z dos vértices são obtidos a partir do *sen* e *cos* do ângulo da fatia atual, e o y através da stack em que nos encontramos.

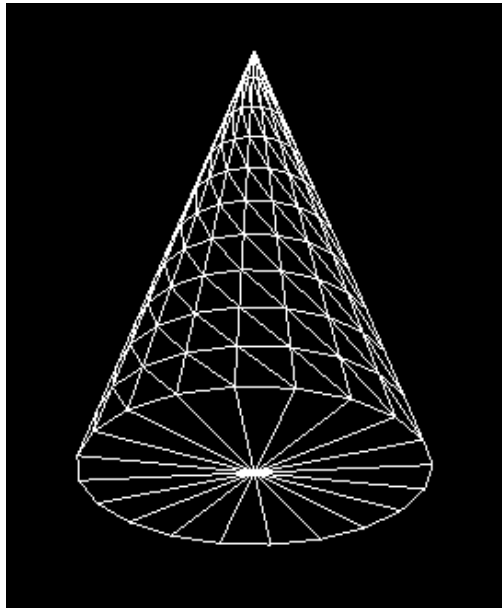


Figure 2.5: cone, radius = 1, height = 3, slices = 20, stacks = 10

2.2 Engine

Esta aplicação recebe como input um ficheiro xml, e tem como função extrair deste informações sobre a posição e para que ponto a câmara aponta, valores para near e far, tamanho da janela, e por último, os ficheiros com vertices guardados para serem lidos. Para analisar todos estes dados, foi utilizado o parser TinyXML2.

2.2.1 Variáveis globais

Depois de extraída a informação necessária do ficheiro xml, necessitamos de variáveis globais, para a armazenar.

- **camx, camy, camz** - Referentes à coordenada do ponto onde a câmara se encontra.
- **lookx, looky, lookz** - Referentes à coordenada do ponto para que a câmara está a apontar.
- **upx, upy, upz** - Referentes ao vetor up da câmara.
- **fov** - Referente à amplitude do campo de visão da câmara.
- **near, far** - Planos que determinam quais os objetos a serem renderizados
- **width, height** - Largura e altura da janela.
- **ficheiros** - Vetor de strings que guarda os nomes dos ficheiros com os vertices a serem lidos.
- **numbersArray** - Vetor de vetores de floats, em que, para cada ficheiro a ler, guarda um vetor com todos os vertices dos triângulos a serem desenhados.

Para melhor visualização das figuras desenhadas, decidimos implementar movimento para a nossa câmara.

- **dist** - Correspondente à distancia entre a câmara e a origem, para que ao mover a câmara, esta sempre se mantenha à mesma distância.
- **alpha** - Ângulo que representa a rotação da câmara em torno do eixo Y.
- **beta** - Ângulo que representa a rotação da câmara em torno do plano XZ.

2.2.2 Funções auxiliares

- **get_attributes** - Função que processa os atributos de um elemento XML, armazenando as informações relevantes nas variáveis globais.
- **traverse_elements** - Função recursiva que percorre os elementos do ficheiro xml.
- **get_vertices** - Função que lê os ficheiros referidos no xml, armazenando os vêrtices contidos nestes e guardando-os num vetor.

Chapter 3

Demonstração

3.1 Execução dos programas

Para executar os nossos programas, precisamos primeiro de os compilar, podemos correr no nosso terminal

cmake CMakeLists.txt

e de seguida

make

para gerar os executáveis.

Para escrever os pontos de uma primitiva em ficheiro, é necessário escolher o nome de uma primitiva válida, os seus atributos, e o nome do ficheiro para armazenar os vértices, por exemplo:

./generator plane 2 10 plane.3d

Para desenhar a figura pedida por um ficheiro xml, basta apenas executar:

./engine test_1_3.xml

Chapter 4

Conclusão

Durante a realização desta primeira fase do projeto deparámo-nos com alguns obstáculos. Tivemos alguma dificuldade na geração dos vértices para o cone e para a esfera, devido ao facto de termos de usar coordenadas esféricas. A leitura e o parsing dos ficheiros em formato xml foi também um desafio, visto que nunca tínhamos trabalhado previamente com nenhuma ferramenta de apoio.

No entanto, sentimos ter cumprido o que nos foi pedido, considerando então um balanço positivo desta entrega, e também nos sentimos melhor preparados para a realização das próximas fases deste projeto.