Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

Trabalho Prático
Fase 1 - Primitivas gráficas

António Filipe Castro Silva(a100533) Diogo Rafael dos Santos Barros(a100600)
Duarte Machado Leitão(a100550) Pedro Emanuel Organista Silva(a100745)

8 de março de 2024

CG

Índice

Lista de Figuras

1 Introdução

Este relatório foi realizado no âmbito da 1.ª fase do Trabalho Prático da Unidade Curricular de Computação Gráfica, onde nos foi proposto o desenvolvimento de duas aplicações distintas, mas interligadas:

- **Generator**: um programa que trata de gerar vértices para variadas primitivas gráficas, sendo elas: planos, caixas, esferas e cones. Esses dados ficam guardados posteriormente num ficheiro ".3d".

- **Engine**: um programa que recebe um ficheiro de configuração do tipo XML, que contem várias informações e a partir destas informações (sendo uma delas o ficheiro criado pelo **Generator**) desenha todos os vértices indicados e demonstra um certo ângulo de câmara.

Estas aplicações foram desenvolvidas utilizando a linguagem C++ e utilizando a ferramenta OpenGL e a Biblioteca *tinyxml* para tratar da leitura dos ficheiros XML.

Neste relatório, vamos relatar a forma como arquiteturamos a nossa resolução, de que forma tratamos de cada primitiva gráfica no **Generator**, como o programa da **Engine** lê, corre e apresenta essas primitivas e, por fim, explicar como executamos as aplicações .

2 Arquitetura da Solução

Para produzirmos a solução para este trabalho prático, tivemos de dividi-lo em algumas partes, mais especificamente, dividimos as partes por pastas diferentes:

- **Pasta Engine**: nesta pasta temos o ficheiro **engine** que trata de tudo o que tem a ver com o GLUT, desde dar *Render* à cena em questão, para tratar de desenhar as diferentes figuras, de acordo com os pontos obtidos que são lidos na main, para tratar dos casos de processar as várias *keys* do teclado que pressionamos. A leitura dos ficheiros XML é tratado no ficheiro **leitor**, tendo este um hpp, para facilitar a passagem das funções para a engine. Este ficheiro cria um leitor, extrai os dados do XML e tem as funções de get para facilitar o acesso a cada dado do Leitor.

- **Pasta Generator**: nesta pasta somente temos o ficheiro **generator** que trata de gerar os ficheiros ".3d"que contêm os pontos de cada primitiva gráfica que nos foi pedida. Sendo assim temos quatro funções **generateX**: **generatePlane**, que gera uma *plane* dependendo de vários dados fornecidos, sendo os principais, as *axis*, pois são estas que determinam onde a *plane* fica localizada; **generateBox**, que gera uma *box* a partir de várias *planes*; **generateSphere**, que gera uma *sphere*, com diferentes parâmetros, dependendo do que se dá; por fim, a **generateCone**, que gera um cone, criando inicialmente a face à volta e depois por fim a sua base. Este ficheiro, além dos *generates*, tem a sua **main**, na qual dependendo dos argumentos que recebe escolhe a *generate* que tem de utilizar e que parâmetros dá a essa função.

- **Pasta tinyxml**: esta pasta simplesmente contem todos os ficheiros da biblioteca *tinyxml* que poderiam ser utilizados e que forem indicados pelo professor no início deste trabalho prático. Estes foram utilizados no ficheiro **leitor** para obtermos as informações que vinham nos ficheiros de teste XML.

- **Pasta utils**: esta última pasta importante contem os dois ficheiros que utilizamos como base para estruturar o que o *generator* cria para colocar dentro dos ficheiros ".3d". São estes o ficheiro **figura** e o ficheiro **ponto**, cada um contendo um *hpp* também. O ficheiro **ponto** contêm a estrutura de um ponto, os seus getters e algumas funções que poderão ser necessários. O ficheiro **figura** tem a estrutura de uma figura (uma lista de pontos), as funções de pegar nos pontos, criar a figura, adiiconar um ou vários pontos em simultâneo, criar file para essa figura, criar uma figura de um ficheiro, apagar figura (tratamento pequeno de memory leaks) e criar lista de várias figuras.

3 Generator

O Generator é a componente do projeto que gera os pontos que constituem as figuras, com base num dado input. Os pontos serão guardados num ficheiro .3d que, eventualmente será usado pela Engine.

No input do Generator deve constar o tipo de figura que se pretende gerar (plane, box, sphere, cone), bem como outros parâmetros específicos para cada tipo de figura (length, slices, ...).

3.1 Figuras

3.1.1 Plane

A construção de planos baseia-se na junção de triângulos no eixo inserido como argumento na função geradora. O primeiro passo consiste em calcular o tamanho de cada uma das divisões que constituem o plano. De seguida, serão calculadas as coordenadas iniciais que serão posteriormente usadas no calculo das restantes coordenadas com constituem o plano.

```
1    float divisionSize = (float)length
2    float start_x = -(length / 2.0f);
3    float start_z = -(length / 2.0f);
4    float start_y = -(length / 2.0f);
```

O plano pode ser visto como uma matriz com o número de colunas e o número de linhas igual ao valor de divisions. Com isto em mente, foram iniciados dois ciclos for para iterar sobre as linhas e colunas do plano e gerar os vértices que o constituem. Dentro do ciclo, há condicionais que dependem dos eixos especificados (axis1 e axis2). Cada conjunto de cálculos gera as coordenadas dos vértices com base nos eixos selecionados.

Os cálculos das coordenadas são feitos da seguinte forma:

- Caso 1: Eixo X e Y (axis1 == 'x' && axis2 == 'y'):
  - x1 = start_x + coluna x divisionSize
  - z1 = start_z + linha x divisionSize
  - x2 = x1 + divisionSize
  - x3 = x1
  - z3 = z1 + divisionSize
- Caso 2: Eixo X e Z (axis1 == 'x' && axis2 == 'z'):
  - x1 = start_x + coluna x divisionSize
  - z1 = start_z + linha x divisionSize
  - z2 = z1
  - x3 = x1
  - z3 = z1 + divisionSize
- Caso 3: Eixo Y e Z (axis1 == 'y' && axis2 == 'z'):
  - x = start_x
  - y1 = start_y + coluna x divisionSize
  - z1 = start_z + linha x divisionSize
  - y2 = y1
  - z2 = z1 + divisionSize
  - y3 = y1 + divisionSize
  - z3 = z1

Caso seja necessário inverter a orientação do triângulo, ou seja, a flag inverte tem valor 1, deverão ser feitas trocas de vértices.

- Caso 1: Eixo X e Y (axis1 == 'x' && axis2 == 'y'): os vértices x2, z2 e x3, z3 são trocados
- Caso 2: Eixo X e Z (axis1 == 'x' && axis2 == 'z'): os vértices x2, z2 são trocados e x3=x1+divisionSize
- Caso 3: Eixo Y e Z (axis1 == 'y' && axis2 == 'z'): os vértices y2, z2 e y3, z3 são trocados

No fim do cálculo das coordenadas, os pontos devem ser criados e adicionados à figura que constitui o plano.

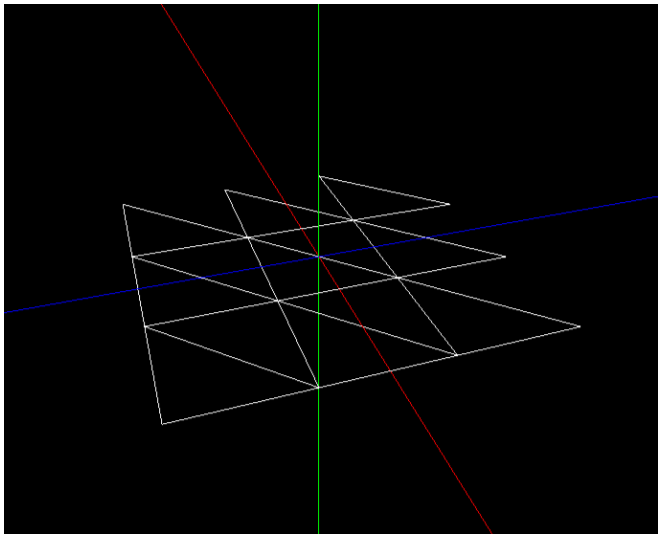

Figura 3.1: Plane 2 3

3.1.2 Box

A caixa pode ser vista como um conjunto de planos. Cada um destes planos representa as faces superior, inferior, frontal, traseira, lateral esquerda e lateral direita.

São criadas figuras para cada um dos planos que constituem a caixa com recurso à função geradora de planos.

Os pontos de cada plano são posteriormente passados para a figura da caixa. No fim deste processo são apagadas as figuras dos planos, visto que não voltarão a ser usadas.

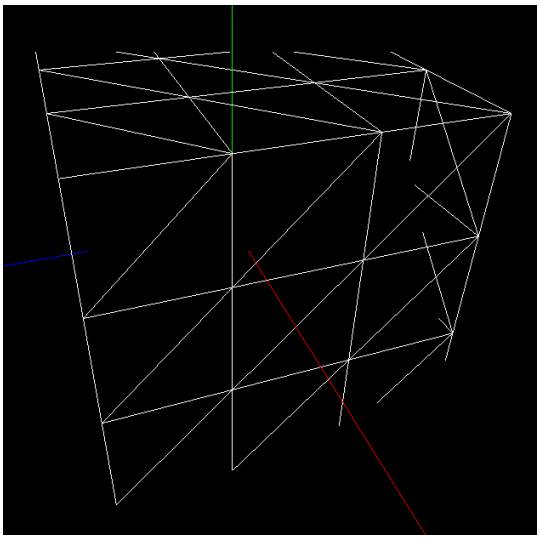

Figura 3.2: Box 2 3

3.1.3 Sphere

A esfera é aproximada por uma série de fatias e anéis, onde o número de fatias é determinado pelo parâmetro slices e o número de anéis é determinado por stacks. Cada fatia é composta por dois triângulos adjacentes.

A lógica de geração da esfera envolve a iteração através de cada anel (stack) e fatia (slice) para calcular as coordenadas dos vértices correspondentes. Dentro dos ciclos, o stackAngle e o sliceAngle são utilizados para determinar as coordenadas esféricas dos vértices. As coordenadas esféricas (x, y, z) de um ponto em uma esfera com raio radius são calculadas da seguinte forma:

```
1     x1 = radius * sin(stackAngle1) * cos(sliceAngle1);
2     y1 = radius * sin(stackAngle1) * sin(sliceAngle1);
3     z1 = radius * cos(stackAngle1);
```

Estas equações representam a projeção dos pontos em coordenadas esféricas para coordenadas cartesianas. Os vértices que constituem cada fatia são calculados desta maneira, e os pontos resultantes são adicionados à figura final.

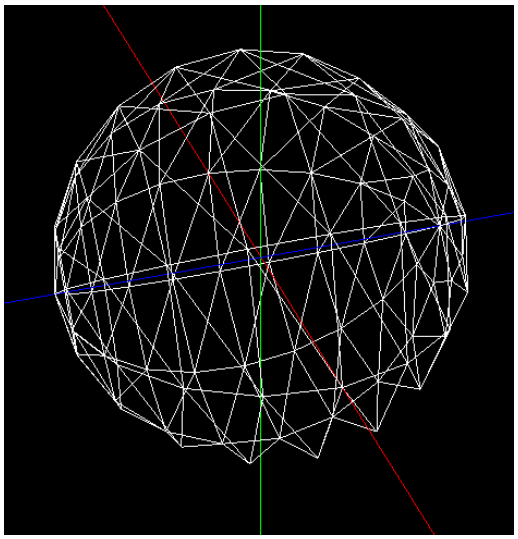

Figura 3.3: Sphere 1 10 10

3.1.4 Cone

A construção do cone, à semelhança da esfera, recorre a fatias e aneis. O número de fatias é determinado pelo parâmetro slices, enquanto o número de anéis é determinado por stacks. O cone é dividido em secções horizontais, e cada seção é aproximada por triângulos. Para obter as coordenadas das secções laterais do cone, são criados dois ciclos. O primeiro ciclo percorre os anéis e o segundo as fatias.

```
1     x1 = radius * (1 - stackHeight1 / height) *
          cos(sliceAngle1);
2     y1 = stackHeight1;
3     z1 = radius * (1 - stackHeight1 / height) *
          sin(sliceAngle1);
4
5     x2 = radius * (1 - stackHeight1 / height) *
          cos(sliceAngle2);
6     y2 = stackHeight1;
7     z2 = radius * (1 - stackHeight1 / height) *
          sin(sliceAngle2);
8
9     x3 = radius * (1 - stackHeight2 / height) *
          cos(sliceAngle1);
10    y3 = stackHeight2;
11    z3 = radius * (1 - stackHeight2 / height) *
          sin(sliceAngle1);
12
13    x4 = radius * (1 - stackHeight2 / height) *
          cos(sliceAngle2);
14    y4 = stackHeight2;
15    z4 = radius * (1 - stackHeight2 / height) *
          sin(sliceAngle2);
```

A obtenção destas coordenadas envolve a parametrização das coordenadas dos vértices em termos de dois ângulos, stackAngle stackAngle e sliceAngle sliceAngle, que representam a posição relativa ao longo do eixo vertical e ao redor do eixo horizontal do cone, respetivamente. A stackHeight representam as alturas dos anéis ao longo do eixo vertical do cone.

Ao variar stackHeight de 0 a height e sliceAngle de 0 a 2π, os vértices são gerados para formar as seções laterais do cone. Esses vértices são então conectados para formar triângulos, preenchendo assim as superfícies laterais do cone.

Depois de geradas as secções laterais do cone, é necessário gerar a sua base. Os pontos que constituem a base são gerados em outro ciclo. Para cada fatia ao longo do plano horizontal da base são geradas as coordenadas que constituem um triângulo.

```
    float x1 = 0;
    float y1 = 0;
    float z1 = 0;

    float x2 = radius * cos(sliceAngle1);
    float y2 = 0;
    float z2 = radius * sin(sliceAngle1);

    float x3 = radius * cos(sliceAngle2);
    float y3 = 0;
    float z3 = radius * sin(sliceAngle2);
```

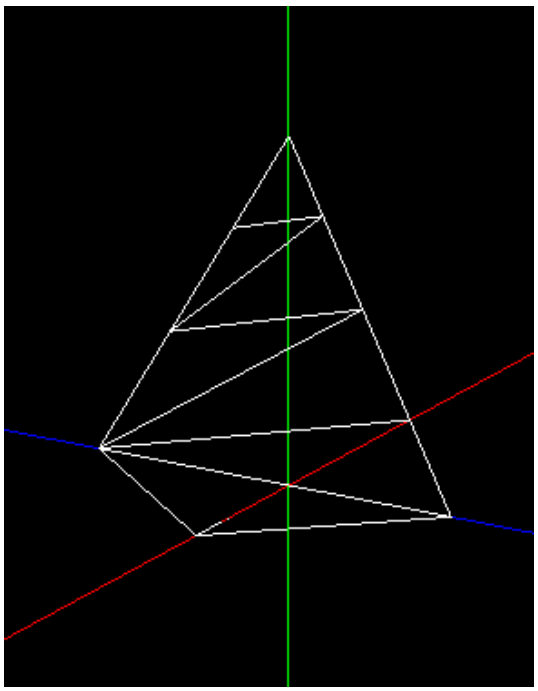

Figura 3.4: Cone 1 2 4 3

4 Engine

Depois da formulação do *Generator* e da criação dos ficheiros ".3d", procedemos à implementação da aplicação **Engine** que vai tratar de ler a informação que vem dos ficheiros de configuração XML(posição da câmara e o ficheiro .3d, ou mais do que um, para ler). Depois de lidos esses dados, este trata de desenhar os triângulos que originaram das primitivas que implementamos no *Generator*.

A função que trata da leitura dos ficheiros XML está implementada no ficheiro **leitor.cpp**. Os dados lidos do ficheiro XML são depois convertidos para uma *struct* também chamada **Leitor**.

A nossa estrutura Leitor guarda 4 arrays e uma lista:

- **posicao**: um array de 3 elementos *float*, sendo que cada um representa a coordenada x,y e z da posição da câmara
- **lookAt**: muito semelhante à **posicao**, mas desta vez guarda as coordenadas da posição lookAt da câmara
- **up**: igual às últimas duas, mas guarda as coordenadas do vetor *up* da câmara
- **projection**: o último array de 3 floats, que guarda os parâmetros fov, near e far da câmara.
- **file**: a lista *file* guarda em string o path das figuras ".3d"que vêm no ficheiro XML e queremos desenhar.

Nesse ficheiro **leitor.cpp**, além da estrutura do Leitor e da função que trata de ler os ficheiros XML (extrair_XML), que utiliza várias vezes a biblioteca *tinyxml*, também temos getters de cada parte da estrutura, a criação inicial do Leitor e a função que trata do *delete* desse leitor que acontece na conclusão do programa.

Depois de criado com sucesso o **Leitor**, realmente criamos o ficheiro **Engine**, onde estará a execução desta aplicação em concreto. Neste ficheiro, depois de tratarmos de todos os *includes* que este necessita(leitor.hpp, por exemplo), criamos várias variáveis globais, sendo as principais as que simbolizam as posições da câmara. Além dessa também criamos uma alpha, uma beta e uma radius para futuramente auxiliar na manipulação da câmara. Também criamos o mode (modo de exibição na consola(LINE,POINT,FILL)), uma lista de figuras e inicializamos o Leitor.

Claramente, neste ficheiro, a parte principal será utilizar a função de ler o XML para obter todos os dados, mas depois também tem as importantes funções que tratam de mostrar as figuras e movimentar a câmara.

Primeiro lemos os dados e, além de designarmos cada valor à sua variável global, calculamos o alpha, beta e radius com os dados da câmara, para depois facilitar e demonstrar de melhor forma a alteração das vistas de câmara, sendo estas variáveis calculadas deste modo:

```
1    radius = sqrt(cameraPosX * cameraPosX + cameraPosY *
        cameraPosY + cameraPosZ * cameraPosZ);
2    alpha = acos(cameraPosZ / sqrt(cameraPosX * cameraPosX +
        cameraPosZ * cameraPosZ));
3    beta1 = asin(cameraPosY / radius);
```

Depois de calculados estes valores, realmente começa-se a chamar as funções de GLUT para expor o resultado da leitura do XML e utiliza-se algumas das funções criadas por nós e vamos falar sobre elas pela ordem que aparecem no ficheiro:

- **changeSize**: função que trata dos vários parâmetros de tamanho da window e, mais importante, a perspetiva, que utiliza **FOV**, **near** e **far**, dados que são retirados do XML.
- **drawFiguras**: função que recebe a lista das figuras lidas, pega nos pontos delas e as desenha, usando *glVertex3f* e *GL_TRIANGLES*
- **renderScene**: função principal que trata do que a *Engine* mostra. Começa por definir o *lookAt* com todas as variáveis descobertas no XML ou calculadas previamente. Depois trata de desenhar as linhas do referencial e, por fim, chama a função drawFiguras para desenhar as primitivas gráficas pedidas.
- **processSpecialKeys**: esta é a primeira de duas funções que trata das *keys* clicadas no teclado. Esta chama-se *SpecialKeys* pois trata da utilização das setas no teclado, tendo cada uma a sua função específica. Seta para a frente mexe a camara para a frente, sexta para baixo mexe a camara para trás, seta para a esquerda mexe a câmara para a esquerda (pelo eixo x), seta para a direita mexe a câmara para a direita (pelo eixo x)
- **processKeys**: esta função já trata das letras clicadas. AWSD fazem o mesmo que seta para a esquerda, seta para cima, seta para baixo, seta para a direita. FLP alteram o modo de apresentação entre FILL, LINE, POINT. E o + e o - aumentam e diminuem o zoom na imagem, utilizando o radius.

5 Execução

Para compilarmos este projeto, começamos por usar um ficheiro CMakeLists que nos permite compilar todo o projeto através do cmake. Depois de compilado, obtemos os dois executáveis para cada aplicação, o **generator.exe** e o **engine.exe**. A forma como arranjamos para correr o generator foi imediatamente após a compilação, para isso criamos um makefile. Nesse makefile, utilizamos o CMakeLists criado por nós, mas também tratamos de mencionar que é para compilar em modo Win32 e Release. Sendo assim a nossa makefile ficou assim:

```
build_f1:
    cmake -B FASE1/build -S FASE1/src -A Win32
    cmake --build FASE1/build/ --config Release

    @echo "Build␣FASE1␣done"
    ./FASE1/build/Release/generator.exe plane 2 3
        FASE1/out/plane_2_3.3d
    @echo "Plane␣done"
    ./FASE1/build/Release/generator.exe box 2 3
        FASE1/out/box_2_3.3d
    @echo "Box␣done"
    ./FASE1/build/Release/generator.exe cone 1 2 4 3
        FASE1/out/cone_1_2_4_3.3d
    @echo "Cone␣done"
    ./FASE1/build/Release/generator.exe sphere 1 10 10
        FASE1/out/sphere_1_10_10.3d
    @echo "Sphere␣done"
```

Como podemos ver, logo após correrrmos o cmake, compilando assim o código e produzindo os dois executáveis, procedemos à execução do *generator* em cada uma das primitivas gráficas pedidas, criando assim 4 ficheiros ".3d". A forma de execução como conseguimos ver é: executável primitiva_gráfica parâmetros necessários para esta ficheiro_out

Para o *engine* já tivemos de proceder de uma forma diferente. Este ao contrário do *generator.exe* só vai necessitar que lhe seja dado o caminho para o ficheiro de teste XML. Sendo assim a sua execução seria deste modo, utilizando uma linha de cada vez, dependendo do teste que se quer executar:

```
1     start Release/engine ../test_fase1/test_1_1.xml
2
3     start Release/engine ../test_fase1/test_1_2.xml
4
5     start Release/engine ../test_fase1/test_1_3.xml
6
7     start Release/engine ../test_fase1/test_1_4.xml
8
9     start Release/engine ../test_fase1/test_1_5.xml
```

Ao contrário do *generator*, a *engine* já tem de ser executada dentro da pasta *build*, onde está todo o código compilado e os executáveis, pois como o generator é executado com o makefile, este é executado na raiz do projeto na pasta "Trabalho Prático - Grupo9".

6 Conclusão

Com a resolução da primeira fase deste trabalho prático conseguimos de algum modo adaptar o que temos aprendido neste primeiro mês de semestre de aulas teóricas e teórico-práticas em algum funcional, mas também nos ajudou a ser mais autónomos em termos de aprendizagem no que toca a Computação Gráfica e à linguagem C++.

Depois de uma breve análise, acreditamos que produzimos um *Generator* e um *Engine* razoavelmente bem conseguidos, só com alguma necessidade de melhoramento na precisão dos pontos, implementando todos os *Generates* que o professor pedia e conseguindo ver as figuras com o posicionamento de câmara correto a partir da Engine. Em termos de Extras, temos a introdução da movimentação da câmara e alteração de visualização das figuras, tal como aprendemos nas aulas práticas.

Em conclusão, analisando tudo o que foi feito, consideramos que temos tudo o que é necessário para prosseguirmos com sucesso para a segunda fase deste trabalho prático.