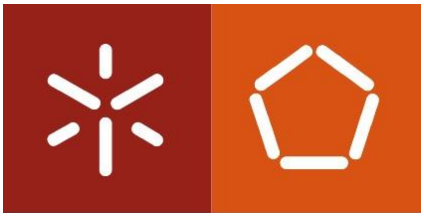

Universidade do Minho

Departamento de Informática

Computação Gráfica
Grupo 45
Trabalho Prático
Fase 1

14 de março de 2021

Ariana Lousada (A87998)

Rui Armada (A90468)

Carolina Vila Chã (A89495)

Sofia Santos (A89615)

Conteúdo

1 Introdução **3**

2 Desenvolvimento **4**
2.1 Generator 4
2.1.1 Plano 4
2.1.2 Caixa 4
2.1.3 Esfera 5
2.1.4 Cone 5
2.2 Engine 6
2.2.1 Parse de XML 6
2.2.2 Renderização dos modelos 7
2.2.3 Câmara 7

3 Conclusão **9**

Capítulo 1

Introdução

O trabalho aqui detalhado foi realizado no âmbito da UC de **Computação Gráfica**, e tem como objetivo desenvolver um **motor 3D** e fornecer exemplos de **cenas** que demonstrem o seu potencial.

Este relatório engloba a **primeira** de 4 fases deste projeto, que consiste na criação de **duas aplicações**.

- **Generator** - gera ficheiros com as informações dos modelos 3D das figuras.[1]
- **Engine** - lê um arquivo XML com os ficheiros *.3d* gerados pela aplicação anterior e apresenta os respetivos modelos.

Ao longo deste documento vamos explicar o funcionamento de cada uma destas aplicações.

[1] Nesta fase apenas gera os vértices necessários para desenhar cada figura.

Capítulo 2

Desenvolvimento

2.1 Generator

O principal objetivo do gerador deste trabalho prático é criar ficheiros de pontos no espaço 3D com base na informação introduzida. O utilizador tem quatro opções de figuras diferentes: plano, esfera, caixa e cone. De modo a facilitar a representação de dados, criámos uma estrutura **Point** que representa um ponto num referencial tridimensional com coordenadas cartesianas. Após a inserção do comando pelo utilizador, o *generator* cria um ficheiro com o nome especificado, no qual consta o número total de pontos que constituem a figura, tal como as coordenadas de cada ponto, calculadas pela função *draw* da figura respetiva, que retorna um vetor de *Points*. Vamos passar a explicar sucintamente como cada figura é construída.

2.1.1 Plano

O plano é a figura mais simples, consistindo apenas em dois triângulos no plano $y = 0$, formando um quadrado de comprimento igual ao valor fornecido. A função responsável por este processo é a função *drawPlane*, que apenas recebe o comprimento do lado do "quadrado"fornecido pelo utilizador.

2.1.2 Caixa

A caixa é uma versão mais complexa do plano. Esta possui seis faces retangulares, unidas pelas suas arestas, de forma a formar um paralelepípedo. Os valores do comprimento, largura e altura são fornecidos pelo utilizador, tal como um valor opcional de divisões, que estabelece o número de retângulos em que cada face deve estar dividida. Por exemplo, 3 divisões correspondem a $3^2 = 9$ retângulos por face da caixa.

Tal como no plano, os retângulos da caixa são formados por pares de triângulos.

As funções responsáveis por todo este processo são a função *drawBox*, que apenas recebe as dimensões da caixa, e a função *drawBoxD*, que recebe ainda o número de divisões.

2.1.3 Esfera

Para gerar uma esfera, devemos ter em conta o número de *slices* e de *stacks*. As *slices* são as divisões horizontais, enquanto que as *stacks* são as divisões verticais.

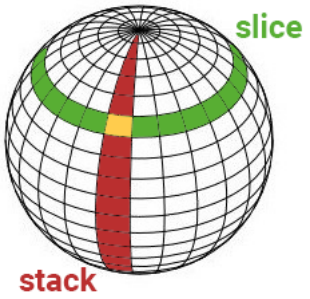

Figura 2.1: *slices* e *stacks* de uma esfera.

A forma mais fácil de gerar os vértices da esfera é calcular as coordenadas polares de cada vértice e convertê-las para coordenadas cartesianas. Assim, apenas temos de calcular o ângulo de cada *slice* e *stack* e podemos determinar a posição de todos os vértices que precisamos para gerar uma esfera. Para além do número de *slices* e de *stacks* que a esfera deve ter, precisamos ainda de saber o raio da mesma, valores estes que são fornecidos à função *drawSphere*.

2.1.4 Cone

Para gerar os vértices do cone usamos um algoritmo bastante semelhante ao que usamos para gerar a esfera, visto que ambos estes sólidos devem estar divididos em *slices* e em *stacks*. Apenas temos que ter em conta o raio do cone e a sua altura, que aqui são valores diferentes, enquanto que na esfera são iguais. Para além disso, o cone possui uma base, que não sendo mais do que um círculo, não é difícil de gerar.

A função responsável por este processo é a função *drawCone*, que recebe o valor do raio da base, altura, *slices* e *stacks* do utilizador.

2.2 Engine

O engine é a aplicação responsável por *renderizar* as figuras construídas previamente pelo *generator*. Para isto, recebe um ficheiro XML (que por sua vez contém os ficheiros dos vértices dos modelos a desenhar) que é lido e interpretado posteriormente.

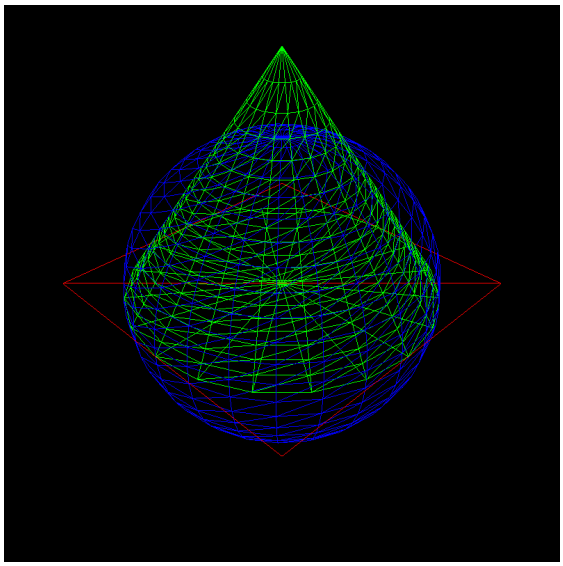

Figura 2.2: Figuras desenhadas com o *generator* e *engine* desenvolvidos.

2.2.1 Parse de XML

De modo a conseguirmos interpretar os ficheiros *.xml* utilizamos a versão 2 do *tinyXML*. A parte do código responsável por interpretar estes ficheiros é a seguinte:

```
doc.LoadFile(argv[1]);
    if(doc.ErrorID()) {
        doc.LoadFile(std::string("../").append(argv[1]).c_str());
        if(doc.ErrorID()) {
            printf("%s\n", doc.ErrorStr());
            return doc.ErrorID();
        }
    }

    XMLElement* scene = doc.FirstChildElement("scene");
    if(scene == NULL) {
        puts("No <scene> found.");
        return 1;
    }

    XMLElement* model = scene->FirstChildElement();
    while(model) {
        if(!strcmp(model->Name(), "model"))
            solids.push_back(vectorize(model->Attribute("file")));
        model = model->NextSiblingElement();
    }
```

Começamos por fazer *error checking* (linhas 2 a 8), de modo a verificar se ocorre algum erro na abertura do ficheiro XML. Aplicações como o *Visual Studio* executam o projeto a partir da pasta `"build/"`. Por esta razão, se o ficheiro *.xml* não for encontrado na pasta de onde o projeto está a ser executado, ele tenta encontrá-lo na pasta anterior, o que nos permite correr o nosso *engine* através do *Visual Studio* sem ser preciso mudar a estrutura dos nossos ficheiros.

Caso não ocorra nenhum erro, vamos passar a procurar *scenes*, nas quais vamos ter os nossos ficheiros de vértices. Caso uma *scene* seja encontrada, a figura do modelo é construída analisando cada ficheiro de vértices e posteriormente desenhando-a com o auxílio do openGL. Exemplos de ficheiros xml utilizados no projeto:

```
<scene>
    <model file='sphere.3d'/>
</scene>
```

```
<scene>
    <model file='plane.3d'/>
    <model file='cone.3d'/>
    <model file='sphere.3d'/>
</scene>
```

2.2.2 Renderização dos modelos

À medida que fazemos *parsing* dos ficheiros *.3d* contidos nas cenas fornecidas ao programa, vamos armazenando os valores relativos aos vértices de cada num `vector<Point>`, em que `Point` é uma classe que representa um ponto, tal como a usada pelo nosso *generator*.

Depois de todos os ficheiros serem lidos, o *engine* pode finalmente renderizá-los. Para isso, vai ler todos os `vector<Point>` gerados anteriormente e transformar esses pontos em vértices, através da função `glVertex3d`.

2.2.3 Câmara

Para o movimento da câmara desenvolvemos a função *keyboardFunc*:

```
void keyboardFunc(unsigned char key, int x, int y) {
  switch(key) {
      case 'a':
          camPos.alpha -= M_PI / 16;
          break;
      case 'd':
            camPos.alpha += M_PI / 16;
            break;
      case 's':
            camPos.beta -= M_PI / 16;
            break;
        case 'w':
            camPos.beta += M_PI / 16;
            break;
        case 'q':
            if (camPos.radius > 1) camPos.radius -= 1;
            break;
        case 'e':
            camPos.radius += 1;
            break;
  }
  if (camPos.alpha < 0) camPos.alpha += M_PI * 2;
  else if (camPos.alpha > M_PI * 2) camPos.alpha -= M_PI * 2;

```

```
25    if (camPos.beta < - M_PI) camPos.beta += M_PI * 2;
26    else if (camPos.beta > M_PI) camPos.beta -= M_PI * 2;
27
28    glutPostRedisplay();
29  }
```

Esta função reage de acordo com o input do utilizador, calculando adequadamente a próxima posição da câmara. As teclas `a` e `d` permitem rodar a câmara na horizontal, as teclas `w` e `s` permitem rodar na vertical e as teclas `q` e `e` mudar a distância da câmara à origem.

Capítulo 3

Conclusão

Com o desenvolvimento da primeira fase deste projeto conseguimos aplicar na prática os conceitos lecionados nesta unidade curricular, auxiliando a consolidação de vários tópicos, nomeadamente como utilizar corretamente a biblioteca GLUT do OpenGL, assim como o raciocínio por detrás da construção tridimensional de algumas figuras simples, o que ajudou a entender a importância desta primeira fase.

Uma vez que conseguimos desenvolver um *generator* capaz de gerar modelos básicos e um *engine* que os renderiza com sucesso, esta aplicação vai-se mostrar extremamente útil em futuras fases deste trabalho prático.