Universidade do Minho
Escola de Engenharia

Licenciatura em Engenharia Informática
Computação Gráfica

Trabalho Prático

Phase 1 – Graphical primitives

Grupo 10

Jéssica Cunha (a100901)
Martim Redondo (a100664)
Rodrigo Castro (a100694)
Tiago Moreira (a100541)

Ano letivo 2023/24

Índice

Índice de figuras

Introdução

Este projeto, desenvolvido no âmbito da unidade curricular de Computação Gráfica, tem por objetivo desenvolver um 3D *engine* de pequena escala, visando criar uma experiência visual tridimensional.

Nesta primeira fase, o objetivo é a implementação de primitivas gráficas. Para isso, será preciso desenvolver duas aplicações: o *Generator* e a *Engine*.

O Generator deverá possibilitar a criação das figuras (planos, caixas, esferas e cones), com parâmetros específicos.

O Engine, por sua vez, deverá ser capaz de ler os ficheiros XML durante a inicialização, interpretando as configurações da câmara e carregando os modelos especificados. A renderização resultante proporcionará uma experiência visual 3D.

No decorrer, serão descrevidos os processos da implementação do projeto, destacando as decisões tomadas durante o desenvolvimento, aliados à componente teórica de Computação Gráfica.

Estrutura

A estrutura desta primeira fase foi pensada em tornar o código o mais simples e intuitivo possível. Começamos por dividir em pastas, sendo criadas 5 pastas:

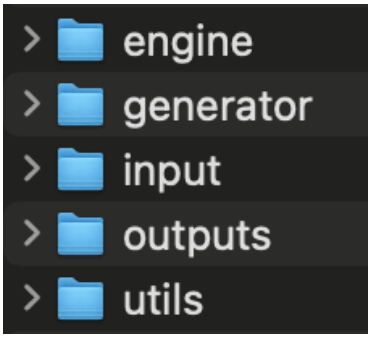

Figura 1 - Estruturação do código

Cada diretório está devidamente organizado com os arquivos correspondentes. A pasta "*engine*" contém os arquivos "*xml_parser.cpp*" e "*engine.cpp*", responsáveis por iniciar a funcionalidade do *engine*. No diretório "*generator*", além do "*generator.cpp*", encontram-se todas as formas geométricas, como "*box.cpp*", "*shpere.cpp*", "*plane.cpp"* e "*cone.cpp*". Os arquivos *XML* necessários para o *engine* estão na pasta "*input*". Os resultados produzidos pelo *generator* são armazenados na pasta "outputs". Por fim, o diretório "*utils*" contém arquivos genéricos utilizados tanto pelo *engine* quanto pelo *generator*, como "*figure.cpp*", "*triangle.cpp*", "*point.cpp*" e "*utils.cpp*".

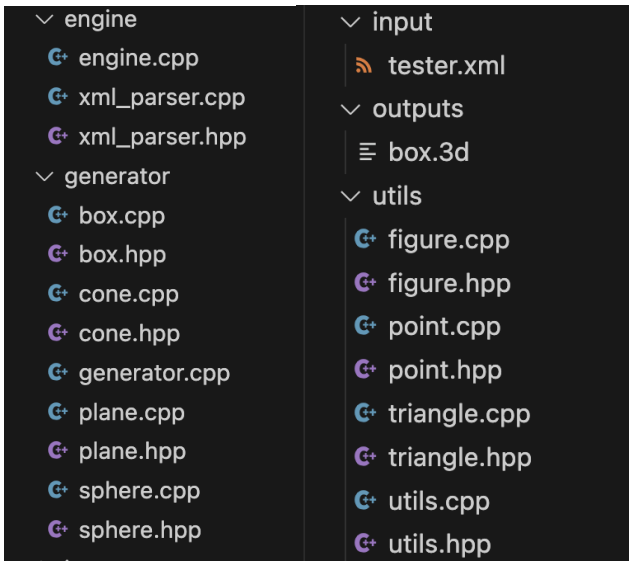

Figura 2 – Ficheiros pertencentes a cada diretório

Structs

O nosso projeto está estruturado de uma maneira ramificada. No começo começamos por definir o que é um ponto:

```
typedef struct point {
    float x;
    float y;
    float z;
} *POINT;
```

Figura 3 - *Struct point*

Depois de definido um ponto começamos por definir o que é um triângulo, que nada mais é do que um conjunto de 3 pontos:

```
typedef struct triangle {
    int num_points; // número de pontos
    std::vector<POINT>* vertices;
}*TRIANGLE;
```

Figura 4 - *Struct triangle*

Por fim, definimos o que é uma figura, que, no nosso projeto, nada mais é do que um conjunto de triângulos. Contudo, teríamos de distinguir cada figura já que cada uma tem as suas características, então tivemos de criar uma estrutura auxiliar para o tipo e uma estrutura principal par a figura:

```
enum FIGURE_TYPE {
    UNKNOWN,
    SPHERE,
    CONE,
    PLANE,
    BOX
};
```

Figura 5 - Estrutura de dados para os tipos de figura

```
typedef struct figure {
    FIGURE_TYPE type;
    std::vector<TRIANGLE>* triangles;
    union {
        struct {
            float radius;
            int slices;
            int stacks;
        } sphere;
        struct {
            float height;
            float radius;
            int slices;
            int stacks;
        } cone;
        struct {
            int length;
            int divisions;
        } plane;
        struct {
            int length;
            int divisions;
        } box;
    };
} *FIGURE;
```

Figura 6 - *Struct figure*

Generator

O *generator* é uma função do nosso programa encarregado de construir uma figura, no entanto, esta deve ser guardada num arquivo. No nosso caso, optamos por salvar as informações sobre a figura, que incluem os triângulos e os pontos que os constituem.

Para compilar, o gerador requer de alguns parâmetros, como o tipo da figura, os dados para a construção da figura e o local onde os dados serão salvos.

Plane

O primeiro passo consistiu em centralizar o plano, o que foi alcançado dividindo o comprimento total por 2. Em seguida, para dividir o plano em várias partes iguais, calculamos o tamanho de cada divisão dividindo o comprimento total pelo número desejado de divisões. Uma vez estabelecidas essas medidas, prosseguimos com a definição dos pontos extremos que delimitam o plano. O ponto 1 corresponde ao canto inferior esquerdo, o ponto 2 ao canto superior esquerdo, o ponto 3 ao canto inferior direito e o ponto 4 ao canto superior direito.

Posteriormente, implementamos um ciclo para percorrer as linhas e colunas do plano. A cada iteração, os pontos são deslocados ao longo do plano de acordo com a posição relativa dentro da grade de divisões. Estes pontos são então organizados em triângulos e armazenados na estrutura de dados que representa o plano.

Box

A box nada mais é do que vários planos, para tal, no ficheiro "*plane.cpp*" foram criadas várias maneiras de fazer um plano para conseguirmos obter as diversas faces da box. Depois das faces devidamente criadas só teríamos de ter o trabalho de as guardar na estrutura da box concatenadas.

Sphere

Para a criação da esfera foi preciso decidirmos por onde a começaríamos a construir e decidimos começar pelo "polo Sul", interagindo entre cada *slice* de cada *stack* e subindo nas *stack* até perfazer o número de *stacks* do input.

Tendo isto em mente, foram criados três ângulos que nos auxiliariam na criação dos triângulos ao longo do percorrer das *stacks* e *slices* da esfera. Esses ângulos são o φ, o θ e o *nextTheta.* O φ representa a rotação ao redor do eixo vertical (eixo y) da esfera. Ele varia de 0 a 2π. Cada valor de φ corresponde a uma fatia horizontal da esfera. O θ representa a inclinação vertical da esfera. Ele varia de 0 a π. O *nextTheta* é o ângulo θ para a próxima slice da esfera, com isto somos capazes de gerar os pontos suficientes para fazer os triângulos daquela *slice*.

Após a geração dos ângulos, estes vão ser usados para criar pontos. Todas as fórmulas para criar os pontos são facilmente geradas através das relações trigonométricas entre as coordenadas esféricas e as coordenadas cartesianas.

Por fim, criamos a camada final, correspondente ao "topo norte". A abordagem é semelhante à da base, onde criamos um ponto com coordenadas $(0, raio, 0)$ e conectamos todos os pontos da camada anterior a esse ponto para formar os triângulos finais que completam nossa esfera. Com isso, a esfera está concluída.

Cone

Inicialmente, iniciamos a criação do cone dividindo-o em *stacks* e *slices*. A abordagem que adotamos é tratar toda a informação de uma *stack* antes de passar para a *stack* seguinte. Ao encontrar a interseção de uma *stack* e com uma *slice*, um ponto é gerado.

Esses pontos de interseção são então utilizados para formar triângulos. Isto é considerado apenas para os triângulos que contêm pontos da mesma *slice* ou da *slice* seguinte. Em particular, dois triângulos são formados por combinações específicas de pontos, incluindo $(slice\ atual, slice\ atual)$, $(slice\ atual + 1, stack\ atual)$, $(slice\ atual + 1, stack\ atual + 1)$ e $(slice\ atual, stack\ atual + 1)$. Os cálculos são realizados para determinar as coordenadas exatas desses pontos, que são então usadas para construir os triângulos correspondentes.

Esse processo é repetido à medida que avançamos pelas *slices*. Quando chegamos à última *slice*, o mesmo método é aplicado, mas tendo em consideração de que a $slice\ atual + 1$ é tratada como a *slice* inicial (*slice* 1).

Os triângulos formados pela base e pelo topo do cone são tratados de maneira especial, reconhecendo a singularidade desses casos.

Engine

O *Engine* é a aplicação responsável por converter a configuração do "mundo", fornecida pelo ficheiro XML, e desenhar de acordo com as informações fornecidas nesse ficheiro, com *OpenGL*.

Estruturas de dados

Primeiramente, começamos por definir estruturas de dados de suporte à configuração da câmara, bem como o armazenamento de outras informações, tal como os ficheiros 3D com as informações das figuras a desenhar.

Assim, criamos as *structs*: *Camera*, *Model*, *Group* e *WORLD*.

A *Camera* armazena as informações sobre a posição, direção e projeção da câmara. As três primeiras variáveis desta struct são constituídos por arrays de 3 elementos para conseguirmos representar o x, y e z para cada variável, respetivamente.

O *Model* guarda o caminho dos ficheiros do modelo 3D.

O *Group* contém uma lista de modelos.

E, por último, o *WORLD* agrega todas essas informações e ainda o comprimento e a largura que a janela deve ter.

```
struct Camera {
    float position[3];
    float lookAt[3];
    float up[3];
    struct Projection {
        float fov;
        float near;
        float far;
    } projection;
};

struct Model {
    std::string file;
};

struct Group {
    std::vector<Model> models;
};

struct WORLD {
    int windowWidth;
    int windowHeight;
    Camera camera;
    Group group;
};
```

Figura 7 - *struct* de suporte à *scene*

Parse

Para conseguirmos povoar a estrutura de dados *WORLD* utilizamos a função ***parse_config_file***. Essa função recorre à biblioteca *tinyXML* para analisar um ficheiro *XML*, que descreve a configuração da *scene*. Essa análise inclui detalhes sobre a janela, câmara e modelos a serem renderizados. E por último fazer a atribuição de cada valor aos elemos da *struct WORLD*.

Interação do utilizador

De maneira, a conseguirmos interagir com aplicação e conseguirmos alterar algumas perspetivas de visualização a função ***keyboardFunc*** que trata eventos de teclado, permitindo ao utilizador interagir com o sistema, como movimentar a câmara e realizar ajustes visuais.

Assim, poderão ser utilizadas as seguintes teclas para os seguintes propósitos:

- **a** → Percorrer o eixo do x no sentido negativo;
- **d** → Percorrer o eixo do x no sentido positivo;
- **s** → Percorrer o eixo do z no sentido positivo;
- **w** → Percorrer o eixo do z no sentido negativo;
- **i** → Movimentar o ponto de foco da câmara para baixo;
- **k** → Movimentar o ponto de foco da câmara para cima;
- **u** → Aumentar a altura da câmara;
- **o** → Diminuir a altura da câmara;
- **+** → *Zoom in* (aproximar a câmara da *scene*);
- **-** → *Zoom out* (afastar a câmara da *scene*).

Testes

De forma a conseguirmos testar o funcionamento da aplicação da *Engine*, desenvolvemos testes, respeitando a configuração dada pelo enunciado e obtivemos os seguintes resultados:

Para o plano:

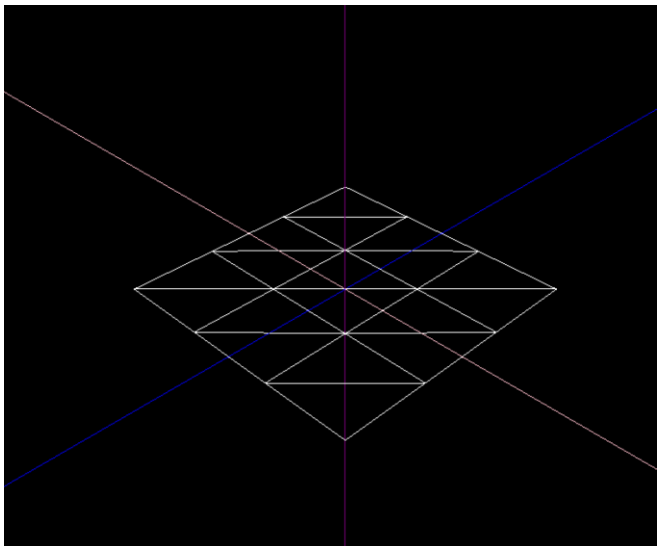

Figura 8 - Resultado do teste para o plano

Para a box:

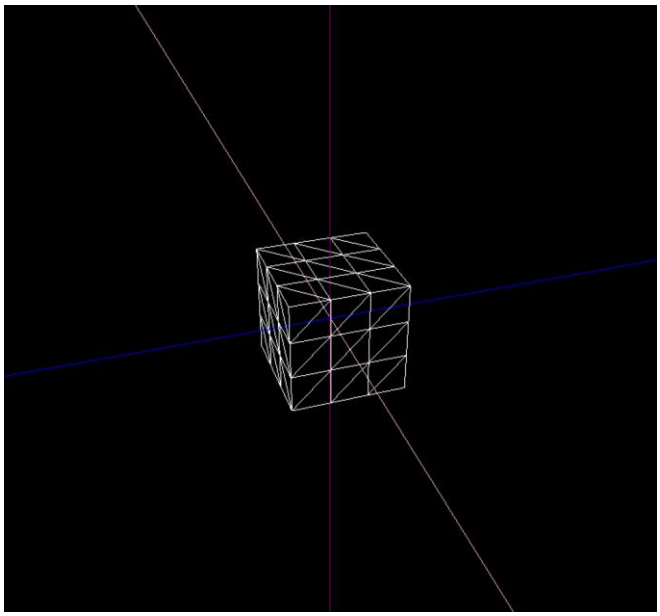

Figura 9 - Resultado do teste para a *box*

Para o cone:

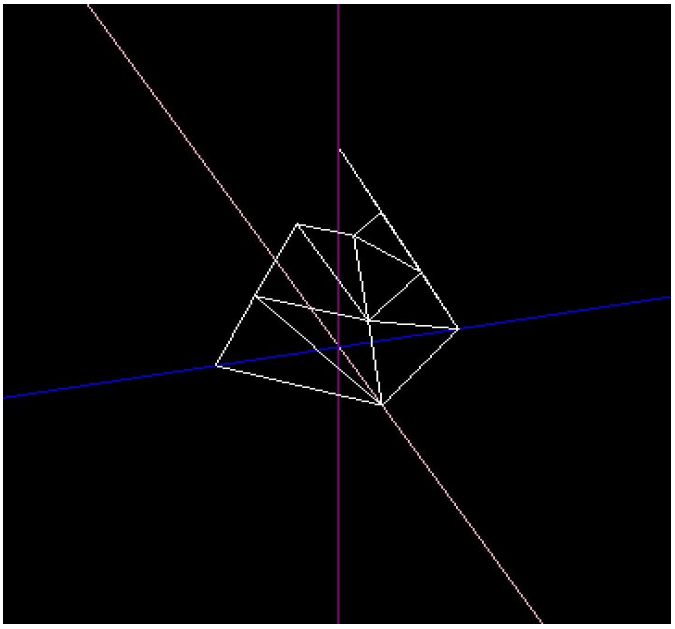

Figura 10 - Resultado do teste para o cone

Para a esfera:

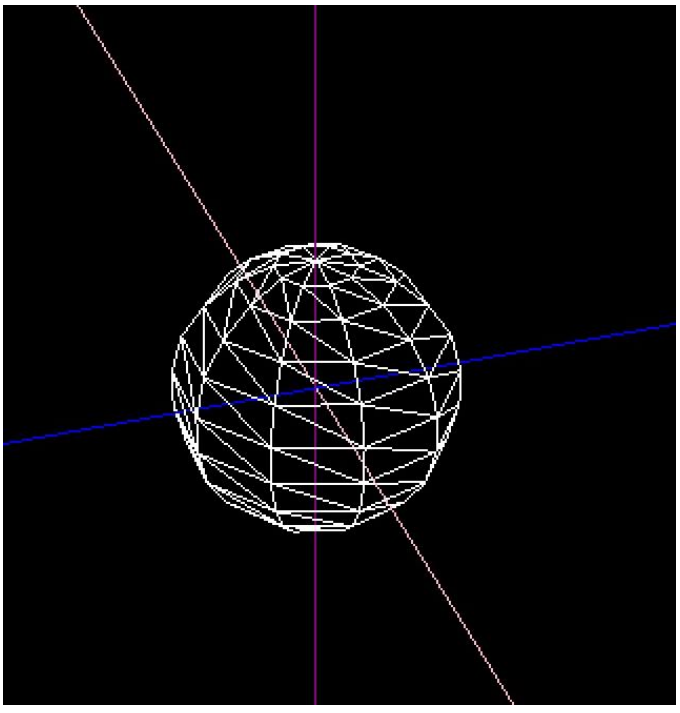

Figura 11 - Resultado do teste para a esfera

Conclusão

Na nossa perspetiva, este projeto ajudou-nos a perceber como começar na área da computação gráfica. Com o desenvolvimento deste sistema conseguimos aplicar os conhecimentos adquiridos nas aulas teóricas e implementar funcionalidades dadas nas aulas práticas, como por exemplo a interação do utilizador através do teclado.

Em suma, consideramos que o nosso trabalho, nesta primeira fase, está bem conseguido, uma vez que conseguimos desenvolver o que nos foi pedido pela equipa docente. Além disso, pensamos que já deixamos estruturas importantes que podem ser necessárias para as próximas fases do projeto.