Universidade do Minho
Departamento de Informática

Computação Gráfica

Fase 1 - Grupo 59

March 8, 2024

Carlos Ferreira
a89509

Gerson Junior
a88000

Pedro Sousa
a100823

Pedro Viana
a100701

Sumário

1 Introdução

Serve o seguinte relatório para documentar o processo de desenvolvimento da primeira fase do trabalho prático da unidade curricular de Computação Gráfica, do ano letivo 2023/2024.

O objetivo desta fase é a implementação de duas aplicações, o Generator e o Engine. O Generator gera arquivos ".3d" com os dados dos modelos necessários ao Engine para os desenhar. O Engine, por sua vez, lê um arquivo de configuração XML que oferece as configurações da camera e uma listagem dos arquivos ".3d" dos modelos que, por fim, este desenha.

2 Utils

Foram desenvolvidos alguns modulos com o objetivo de implementar conceitos abtratos como pontos, vectores e modelos.

2.1 Point3

O tipo de dados `Point3` representa um ponto no espaço tridimensional e é implementado como 3 floats, x, y e z.

Objetos do tipo `Point3` são imutáveis, mas é possível obter um novo `Point3` transladado somando estes com um `Vector3` (vetores) ou triplos de `floats`.

2.2 Vector3

O tipo de dados `Vector3` representa um vector no espaço tridimensional e é implementado como 3 `floats`, x, y e z.

Objetos do tipo `Vector3` são imutáveis, mas é possível obter um novo `Vector3` resultante da soma com outro `Vector3` ou da multiplicação por um escalar (`float`).

2.3 Model

O tipo de dados `Model` representa um modelo 3D composto por um conjunto de pontos. Cada três pontos representam um triângulo.

Este foi implementado como um `std::vector<Point3>`.

Objetos do tipo `Model` podem ser concatenados com outros `Model`, transladados dado um `Vector3`, rodados dados um `Vector3` e um ângulo (`float`), e escalados dado um triplo de `floats`.

É ainda possível desenhar o modelo, escrever para um ficheiro e ler de um ficheiro.

3 Generator

3.1 Modelos

Cada modelo é implementado como uma classe que extende a classe abstrata `Model` e, no seu construtor, calcula os seus vértices.

Nesta fase do projeto foram definidas 4 primitivas:

- Plano
- Caixa
- Esfera
- Cone

3.1.1 Plano

O plano é definido pelo comprimento do seu lado e o número de divisões. Cada divisão é composta por dois triângulos isósceles rectângulos.

Estratégia de cálculo dos vértices:

- Começamos por calcular o comprimento de cada divisão dividindo o comprimento do lado do plano pelo número de divisões.
- O plano é desenhado no plano xOz pelo que o y de todos os seus vértices é 0.
- Como é especificado apenas um comprimento do lado, o plano é quadrado e terá tantas divisões quanto o quadrado do número de divisões do seu lado. Assim, os triângulos destas são calculados em dois cilcos aninhados. Sendo n o número de divisões, o ciclo externo itera n vezes sobre o eixo dos z e o ciclo interno itera n vezes sobre o eixos dos x.
- A cada iteração calculámos as coordenadas dos vértices dos dois triângulos que compõe a divisão e adicionámos estes à lista de vértices do plano pela ordem ditada pela regra da mão direita de forma a que os triângulos estejam direcionados para cima.

A Figura 1 esquematiza a estratégia de cálculo dos vértices descrita, utilizando como variáveis as variáveis da própria implementação que se segue:

```
Plane::Plane(float size, int divisions)
{
    float originOffset = size / 2;
    float divisionSize = size / (float) divisions;

    for (int i = 0; i < divisions; i++) {
        for (int j = 0; j < divisions; j++) {
            // back left corner of the current square
            float x1 = (float) i * divisionSize - originOffset;
            float z1 = (float) j * divisionSize - originOffset;

            // front right corner of the current square
            float x2 = x1 + divisionSize;
            float z2 = z1 + divisionSize;

            // first (back right) triangle
            vertices.emplace_back(x1, 0, z1);
            vertices.emplace_back(x2, 0, z2);
            vertices.emplace_back(x2, 0, z1);

            // second (front left) triangle
            vertices.emplace_back(x1, 0, z1);
```

```
            vertices.emplace_back(x1, 0, z2);
            vertices.emplace_back(x2, 0, z2);
        }
    }
}
```

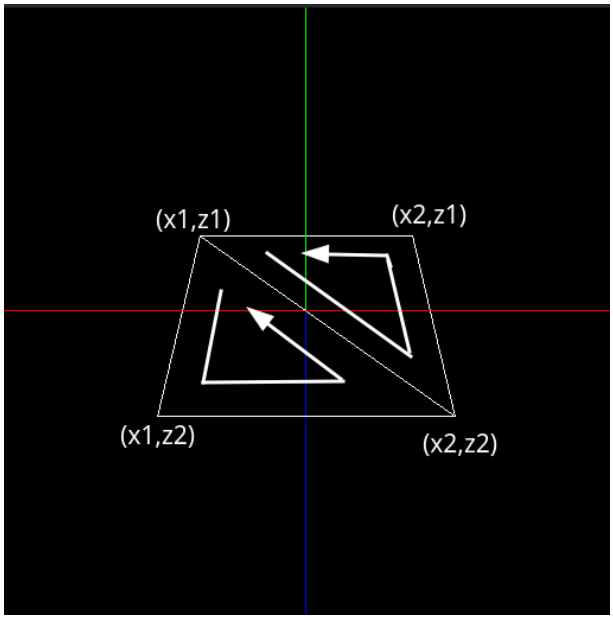

Figura 1: Plano com 1 divisão e 2 de comprimento com coordenadas no plano xOz (y=0).

3.1.2 Caixa

A caixa é definida pelo comprimento do seu lado e o número de divisões. Cada divisão é composta por dois triângulos isósceles rectângulos.

Estratégia de cálculo dos vértices:

- A caixa é definida como a concatenação de 6 planos rodados e transladados para o efeito.

Esta estratégia leva a uma implementação bastante declarativa:

```
vector<Model> createBox(float size, int divisions)
{
    float halfSize = size / 2;

    Plane top(size, divisions);
    top.translate({0, halfSize, 0});
    Plane bottom(size, divisions);
    bottom.rotate({0, 0, 1}, M_PI);
    bottom.translate({0, -halfSize, 0});
    Plane left(size, divisions);
    left.rotate({0, 0, 1}, M_PI_2);
    left.translate({-halfSize, 0, 0});
    Plane right(size, divisions);
    right.rotate({0, 0, 1}, -M_PI_2);
    right.translate({halfSize, 0, 0});
    Plane front(size, divisions);
    front.rotate({1, 0, 0}, M_PI_2);
    front.translate({0, 0, halfSize});
    Plane back(size, divisions);
    back.rotate({1, 0, 0}, -M_PI_2);
    back.translate({0, 0, -halfSize});
```

```
    return vector<Model>{top, bottom, left, right, front, back};
}
```

3.1.3 Esfera

A esfera é definida por um raio, o número de *slices* e o número de *stacks*.

Estratégia de cálculo dos vértices:

- Os vértices dos triângulos são calculados *stack* a *stack* e depois *slice* a *slice*, ou seja, a cada iteração são calculados os dois triângulos que formam o quadrilátero correspondente à interseção da *stack* e *slice* atual.
- Foi decidido iterar *stack* a *stack* de forma a calcular a altura (o y) dos vértices de uma *stack* apenas uma vez.
- A primeira e última *stack* da esfera são cálculadas separadamente do resto do modelo pois cada *slice* destas contém apenas um triângulo evitando incluir triângulos duplicados. Estas são calculadas no mesmo ciclo de forma a reaproveitar o cálculo dos ângulos.
- O cálculo de cada coordenada é feito convertendo coordenadas esféricas, (α, β, r) em coordenadas cartesianas do seguinte modo:

$$x = r \cdot \cos(\beta) \cdot \sin(\alpha)$$
$$y = r \cdot \sin(\beta)$$
$$z = r \cdot \cos(\beta) \cdot \cos(\alpha)$$

A Figura 2 esquematiza a estratégia de cálculo dos vértices descrita, utilizando como variáveis as variáveis da própria implementação simplificada que se segue. Esta generaliza a primeira e última *stack* como qualquer outra *stack* das esfera:

```
Sphere::Sphere(double radius, int slices, int stacks)
{
    double stackStep = 2 * M_PI / slices; // angle between each slice
    double sliceStep = M_PI / stacks; // angle between each stack

    // for each stack
    for (int i = 0; i < stacks; i++) {
        double beta1 = i * sliceStep - M_PI_2; // angle of current stack
        double beta2 = (i + 1) * sliceStep - M_PI_2; // angle of next stack

        double y1 = radius * sin(beta1); // height of current stack
        double y2 = radius * sin(beta2); // height of next stack

        // for each slice
        for (int j = 0; j < slices; j++) {
            double alpha1 = j * stackStep; // angle of current slice
            double alpha2 = (j + 1) * stackStep; // angle of next slice

            double x1 = radius * cos(beta1) * sin(alpha1);
            double z1 = radius * cos(beta1) * cos(alpha1);

            double x2 = radius * cos(beta1) * sin(alpha2);
            double z2 = radius * cos(beta1) * cos(alpha2);

            double x3 = radius * cos(beta2) * sin(alpha1);
            double z3 = radius * cos(beta2) * cos(alpha1);
```

```
            double x4 = radius * cos(beta2) * sin(alpha2);
            double z4 = radius * cos(beta2) * cos(alpha2);

            // bottom left triangle
            vertices.emplace_back(x1, y1, z1);
            vertices.emplace_back(x2, y1, z2);
            vertices.emplace_back(x3, y2, z3);

            // bottom right triangle
            vertices.emplace_back(x3, y2, z3);
            vertices.emplace_back(x2, y1, z2);
            vertices.emplace_back(x4, y2, z4);
        }
    }
}
```

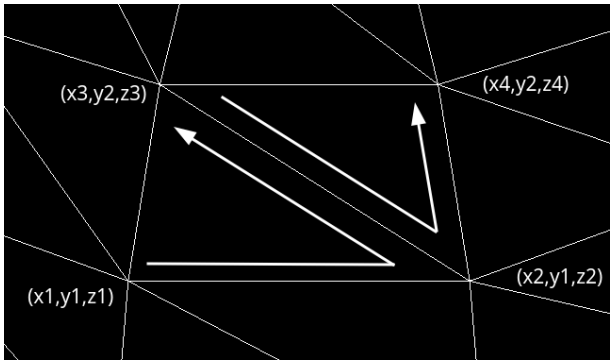

Figura 2: Secção de esfera com 1 de raio e 12 stacks e slices

3.1.4 Cone

O cone é definido pelo raio da base, a sua altura e o número de *slices* e *stacks* do próprio cone.

Estratégia de cálculo dos vértices:

- Similarmente à esfera, os vértices são calculados *stack* a *stack* e a ponta (última *stack*) é calculada separadamente de forma a evitar incluir triângulos duplicados.

Segue a implementação simplificada do cone generalizando a ponta (última *stack*) como qualquer outra *stack*:

```
Cone::Cone(float radius, float height, int slices, int stacks)
{
    double stackStep = 2 * M_PI / slices; // angle between each slice
    double sliceStep = height / (float) stacks; // height between each stack

    // circular base
    for (int i = 0; i < slices; i++) {
        double alpha1 = i * stackStep; // current angle
        double alpha2 = (i + 1) * stackStep; // next angle

        // bottom left vertex
        double x1 = radius * sin(alpha2);
        double z1 = radius * cos(alpha2);

        // bottom right vertex
        double x2 = radius * sin(alpha1);
```

```
        double z2 = radius * cos(alpha1);

        // triangle
        vertices.emplace_back(0, 0, 0);
        vertices.emplace_back(x1, 0, z1);
        vertices.emplace_back(x2, 0, z2);
    }

    // curved surface
    // for each stack
    for (int i = 0; i < stacks - 1; i++) {
        double y1 = i * sliceStep; // current height
        double y2 = (i + 1) * sliceStep; // next height

        double radius1 = (stacks - i) * radius / stacks; // current radius
        double radius2 = (stacks - i - 1) * radius / stacks; // next radius

        // for each slice
        for (int j = 0; j < slices; j++) {
            double alpha1 = j * stackStep;
            double alpha2 = (j + 1) * stackStep;

            double x1 = radius1 * sin(alpha1);
            double z1 = radius1 * cos(alpha1);

            double x2 = radius1 * sin(alpha2);
            double z2 = radius1 * cos(alpha2);

            double x3 = radius2 * sin(alpha1);
            double z3 = radius2 * cos(alpha1);

            double x4 = radius2 * sin(alpha2);
            double z4 = radius2 * cos(alpha2);

            // bottom left triangle
            vertices.emplace_back(x1, y1, z1);
            vertices.emplace_back(x2, y1, z2);
            vertices.emplace_back(x3, y2, z3);

            // top right triangle
            vertices.emplace_back(x3, y2, z3);
            vertices.emplace_back(x2, y1, z2);
            vertices.emplace_back(x4, y2, z4);
        }
    }
}
```

3.2 Sintaxe ficheiros .3d

Os ficheiros ".3d" são ficheiros binários de forma a minimizar o tamanaho e o tempo de leitura e escrita destes.

Estes contém apenas as coordenadas de cada vértice do modelo. Cada 3 vértices representam um triângulo.

As coordenadas são representadas por floats pelo que cada uma ocupa 4 bytes. Assim, um plano com uma divisão ocupa 72 bytes (2 triângulos $* 3$ vértices $* 3$ coordenadas $* 4$ bytes).

4 Engine

Nesta fase foi desenvolvido um motor 3D capaz de posicionar e orientar a camera e desenhar múltiplos modelos compostos por triângulos.

Este motor começa por ler um ficheiro XML cujo nome é passado como primeiro argumento do programa. Este ficheiro contém os dados da camera a definir e uma lista de modelos a desenhar.

São ainda desenhados os eixos x, y e z com cores vermelho, verde e azul, respetivamente, de forma a mais facilmente identificar a posição dos modelos desenhados.

Foram também definidos alguns controlos para controlar a camera.

4.1 Parsing de XML

Para o parsing do ficheiro *XML* foi usada a biblioteca TinyXML2

Nesta fase a implementação do parsing foi feita de forma mais simples possível deixando todo o código na função do *engine* e iniciando múltiplas variáveis globais, em particular, os modelos são carregados para um `vector<Model>`. Posteriormente, o parsing será encapsulado na seu próprio módulo.

4.2 Desenho dos Modelos

Para desenhar os modelos itera-se sobre o vetor de `Model` e chama-se o método `draw()` a cada modelo. Este método desenha um triângulo com cada 3 vértices no modelo.

4.3 Câmara

As propriedades da camera são definidas pelos dados contidos no ficheiro XML, porém também foi incluída a possibilidade de mover a camera no estilo "terceira pessoa" após esta ser definida com os seguintes controlos:

- W - subir
- S - descer
- A - esquerda
- D - direita
- J - zoom in
- K - zoom out

O shift é usado como modificador para executar as mesmas ações 10 vezes mais depressa.

Para a execução destas funcionalidades, o programa recorre a dois ângulos (`yaw` e `pitch`).

Estes são calculados logo após ao parse do ficheiro xml, utilizando os valores da posição da câmara e da posição para onde está a 'olhar'. O yaw é o ângulo em torno do eixo y, e o `pitch` é o ângulo em torno do eixo do z.

O movimento baseia-se na incrementação ou decrementação do valor destes ângulos, dependendo da tecla pressionada

5 Conclusão

Nesta fase do projeto foi possível identificar e desenvolver estratégias de cálculo e representação de simples modelos 3D compostos por triângulos.

Considera-se que nesta fase foi possível consolidar os básicos do desenvolvimento de software com Glut/OpenGL assim como o uso de trignometria e C++ para codificar o cálculo de pontos no espaço e a regra da mão direita para construir triângulos direcionados para o lado pretendido.

Assim, o grupo considera ser agora capaz de implementar outras primitivas como o torus.

Contudo, o trabalho feito nesta fase pode ainda ser desenvolvido, por exemplo, implementando uma abstração de coordenadas polares simplificando assim o cálculo de coordenadas na construção do cone e da esfera.

É ainda possível melhorar a performance do projeto, porém considerou-se que nesta fase era mais valioso apontar para um bom balanço entre simplicidade e performance, mas, em fases seguintes, performance será uma maior preocupação do grupo.