Computação Gráfica: Fase 1 – Primitivas Gráficas

Afonso Faria,
André Ferreira,
Bruno Campos e
Ricardo Gomes

Universidade do Minho, Braga, Portugal
{a83920,a93211,a93307,a93785}@alunos.uminho.pt

Resumo Para esta fase foram desenvolvidos um gerador de ficheiros contendo modelos de primitivas gráficas e um motor capaz de apresentar graficamente tais modelos a partir de um ficheiro de configuração no formato XML.

1 Introdução e Contextualização

O presente relatório diz respeito à primeira fase do projeto da Unidade Curricular de Computação Gráfica.

Nesta fase foi-nos proposta a implementação de dois programas:

- ***Generator***: Programa que gera ficheiros com a informação referente aos modelos de primitivas gráficas;
- ***Engine***: Programa capaz de ler um ficheiro de configuração XML e apresentar os modelos graficamente.

Nas seguintes secções do relatório iremos apresentar a abordagem tomada na resolução dos problemas e os obstáculos que surgiram durante a realização de cada um dos programas desta fase do projeto.

2 *Generator*

2.1 Primitivas

Os parâmetros fornecidos ao *generator* para a criação do modelo gráfico de cada primitiva são:

- `plane <comprimento> <#divisões ao longo de cada eixo>`
- `box <#unidades> <comprimento da grelha>`
- `sphere <raio> <#slices> <#stacks>`
- `cone <raio> <altura> <#slices> <#stacks>`

O *generator* é executado através do seguinte comando:

```
$ ./generator primitiva <argumentos>... <ficheiro de output>
```

Este comando irá gerar um ficheiro com o nome `<ficheiro de output>` que contém os dados necessários para a criação do modelo gráfico da primitiva especificada.

De seguida é explicado o processo de geração de cada primitiva.

Plano: Para o cálculo dos pontos referentes a um (sub)plano no plano xOz, começamos por calcular os valores mínimos e máximos das coordenadas x e z. Como valor mínimo, tanto para x como para z, escolhemos $-(comprimento/2)$; como valor máximo, escolhemos $comprimento/2$. Assim, o plano fica centrado na origem.

De seguida, dividimos o plano em quadrículas de lado *comprimento / divisões* segundo os eixos x e z, e representamos essas quadrículas numa matriz de *divisões* * *divisões*.

Iterando sobre essa matriz (primeiro por x no ciclo exterior e por z no ciclo interior), geramos dois triângulos para cada quadrícula,

Caixa: Para o cálculo dos pontos referentes à caixa, definimos uma função auxiliar para criar os pontos referentes a cada um dos seis lados da caixa, designada `geraPlanoBox`. Esta função gera o plano referente a cada um dois seis lados da caixa, tendo uma mecânica muito similar à criação do plano, dependendo dos valores recebidos. De maneira a saber quais os pontos a criar, e a que lado da caixa pertencem, a função recebe o comprimento de cada lado da caixa, o número de divisões a considerar em cada eixo, um caráter k a indicar que eixo do referencial é que o plano vai "cortar"e um valor v a indicar qual o valor nesse mesmo eixo, se negativo ou positivo.

Por exemplo:

- k = X com v = - 1 → plano que corta o eixo X na parte negativa (lado esquerdo da caixa)
- k = X com v = 1 → plano que corta o eixo X na parte positiva (lado direito da caixa)
- k = Y com v = 1 → plano que corta o eixo Y na parte positiva (lado de cima da caixa)
- k = Z com v = - 1 → plano que corta o eixo Z na parte negativa (lado de trás da caixa)

Esfera: No cálculo dos vértices de uma esfera com raio r, $\#stacks$ stacks e $\#slices$ slices, decidimos que o método apropriado seria recorrer ao uso de coordenadas esféricas. Deste modo, precisaremos de calcular o ângulo β que o ponto desejado tem em relação ao plano xOz ($-90^{o} \leq \beta \leq 90^{o}$) e o

ângulo α que este mesmo tem em relação ao eixo y. Assim, teremos as coordenadas esféricas (α, β, r) que podemos facilmente converter em coordenadas cartesianas através das seguintes expressões:

$$x = r \times cos(\beta) \times sin(\alpha)$$

$$y = r \times sin(\beta)$$

$$z = r \times cos(\beta) \times cos(\alpha)$$

Seguindo este método, resta apenas calcular o α e β de cada vértice. Para tal, tiramos partido do número de stacks e slices através das seguintes formulas:

$$rotation\_stack = \pi \div \#stacks$$

$$rotation\_slice = 2\pi \div \#slices$$

Usando estas fórmulas encontramos o ângulo de divisão entre cada stack ($rotation\_stack$) e de divisão entre cada slice ($rotation\_slice$), logo podemos facilmente calcular o α e β do vértice que se encontra na stack i e na slice j:

$$stack\_angle = \pi \div 2 - i \times rotation\_stack$$

$$slice\_angle = j \times rotation\_slice$$

Aplicando estas fórmulas num ciclo que itera por cada stack e, dentro deste ciclo, num outro ciclo que itera por cada slice, definimos uma função capaz de gerar os vértices da esfera especificada.

Cone: A construção de um cone com raio r, altura h, $\#slices$ slices e $\#stacks$ stacks, começa da base até ao apex, slice a slice. O ciclo exterior do programa itera sobre as *slices*, de $i = 0$ a $i = \#slices - 1$. Para cada *slice* i, começa por determinar os ângulos que delimitam a *slice*:

$$\alpha 1 = i \times 2\pi / \#slices$$

$$\alpha 2 = (i + 1) \times 2\pi / \#slices$$

De seguida, determina as coordenadas x e z dos dois vértices exteriores do triângulo da base (com $y = 0$):

$$x1 = r \times sin(\alpha 1)$$

$$z1 = r \times cos(\alpha 1)$$

$$x2 = r \times sin(\alpha 2)$$

$$z2 = r \times cos(\alpha 2)$$

Imprime então os 3 vértices do triângulo da base:

$$(0, 0, 0), (x1, 0, z1), (x2, 0, z2)$$

De seguida, o ciclo interior do programa itera sobre as *stacks*, de baixo ($j = 0$) para cima ($j = \#stacks - 1$). Para cada *stack* j, gera 2 triângulos (exceto na última *stack*, em que só gera o 1º triângulo), com as seguintes coordenadas:

$$(x1_j, y_j, z1_j), (x2_j, y_j, z2_j), (x2_{j+1}, y_{j+1}, z2_{j+1})$$

$$(x1_j, y_j, z1_j), (x1_{j+1}, y_{j+1}, z1_{j+1}), (x2_{j+1}, y_{j+1}, z2_{j+1})$$

em que

$$x1_k = x1 \times (\#stacks - k) \div \#stacks$$
$$z1_k = z1 \times (\#stacks - k) \div \#stacks$$
$$x2_k = x2 \times (\#stacks - k) \div \#stacks$$
$$z2_k = z2 \times (\#stacks - k) \div \#stacks$$
$$y_k = k \times h \div \#stacks$$

Para evitar cálculos repetidos o programa utiliza algumas variáveis auxiliares que não são apresentadas aqui.

2.2 Formato .3d

Por convenção, os ficheiros gerados usam a extensão `.3d`, mas o seu uso não é obrigatório. Estes ficheiros são formatados da seguinte forma:

- a primeira linha contém o número de vértices do modelo;
- cada uma das linhas seguintes contem as coordenadas cartesianas reais x, y e z de um vértice, separadas por espaços.

Por exemplo, o ficheiro seguinte representa um triângulo no plano XOZ:

`triangle.3d`

```
3
0 0 0
1 0 0
0 0 1
```

3 *Engine*

O engine está dividido em dois módulos: ***parse*** e ***render***.
O módulo parse é responsável pela validação das entidades descritas no ficheiro XML, e da sua leitura para memória.
O módulo *render* é responsável pela manipulação e exibição destas entidades no ecrã.

3.1 *Parse*

A leitura do ficheiro XML é feita no início do programa. Se nenhum ficheiro for fornecido, um *default world* é assumido. Se for fornecido um ficheiro XML, este terá de cumprir os seguintes requisitos:

- conter um nodo com o nome *world*;
- conter um sub-nodo do nodo *world* com o nome *camera*;
- conter um sub-nodo do nodo *world* com o nome *group*;
- o nodo *camera* terá de conter os sub-nodos *position*, *lookAt*, *up* e *projection*;
- se o nodo *group* tiver sub-nodos, estes terão o nome *transform*, *models* ou *group*;
- se o nodo *transform* tiver sub-nodos, estes terão o nome *translate*, *rotate* ou *scale*;
- se o nodo *models* tiver sub-nodos, estes terão o nome *model*;
- qualquer nodo *model* terá de conter um atributo com o nome *file* cujo valor é o caminho para um ficheiro existente, cuja formatação seja a dos ficheiros gerados pelo *generator*;

Quando um atributo numérico está ausente é assumido o valor 0.
Qualquer erro de sintaxe, *input/output* ou memória causa o término do programa com uma mensagem de erro.

Por exemplo, o ficheiro XML seguinte é um input válido:

world.xml

```
<world>
    <camera>
        <position x="10" y="10" /> <!-- valor 0 assumido para z -->
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
        <transform>
            <rotate angle="30" x="0" y="1" z="0" />
        </transform>
        <models>
            <model file="box.3d" />
            <model file="plane.3d" />
        </models>
    </group>
</world>
```

3.2 *Renderer*

World Layout Para representar um *world* em memória são usadas as seguinte estruturas, organizadas hierarquicamente de forma similar ao formato dos ficheiros XML:

```
struct world {
    struct camera camera;
    group root;
};

struct camera {
    glm::dvec3 pos;
    glm::dvec3 lookat;
    glm::dvec3 up;
    glm::dvec3 projection;
};

struct group {
    std::vector<transform> transforms;
    std::vector<model> models;
    std::vector<group> children;
};

struct transform {
    enum class kind {
        translate,
        rotate,
        scale
    } kind;

    union {
        glm::vec3 translate;
        glm::vec4 rotate;
        glm::vec3 scale;
    };
};

struct model {
    std::vector<float> coords;
};
```

O tipo `glm::dvec3` representa um *array* de três `double`s, o `glm::vec3` representa um *array* de três `float`s, e o `glm::vec4` representa um array de quatro `float`s.

Inicialmente era usado o tipo `float` para todos os números reais. No entanto, as funções da API GLUT que manipulam a câmara utilizam o tipo `double`. Por esta razão, todas as estruturas, à exceção da câmera, utilizam o tipo `float`, e a câmera utiliza o tipo `double`.

Um grupo é composto por três *arrays* de tamanho variável: um para *transforms*, outro para *models*, e outro para sub-grupos. Deste modo, um `group` pode ser considerado uma *rose tree*.

Um `transform` pode ser um `translate`, `rotate`, ou `scale`. Por motivos de eficiência de espaço, optamos por usar uma *union* e um discriminante (`transform::kind`) para o representar. Alternativamente, podíamos recorrer a herança e *dynamic dispatch*, mas essa solução implicaria uma maior quantidade de código *boilerplate* e possivelmente um custo no tempo de execução.

Um `model` é composto por todas as coordenadas cartesianas necessárias para o representar. Assim, o comprimento de `model::coords` será sempre um múltiplo de três.

Rendering Nesta etapa são usadas as funções da API GLUT. Estas funções operam com base em *callbacks*. Muitas destas *callbacks* não recebem argumentos, o que nos forçou a usar estado global. Este estado é composto por um *pointer* para o `world` que está a ser renderizado, pelo *aspect ratio* atual, e por um `keyboard`, que representa o estado das teclas (premidas ou não):

```
namespace engine::render::state {
    double aspect_ratio;
    world* world_ptr;
    keyboard kb;
}

class keyboard {
  private:
    std::bitset<std::numeric_limits<unsigned char>::max()> keys;

  public:
    auto pressed(unsigned char key) const -> bool {
        return this->keys[key];
    }

    auto press(unsigned char key) -> void {
        this->keys[key] = true;
    }

    auto release(unsigned char key) -> void {
        this->keys[key] = false;
    }
};
```

O `keyboard` é implementado com recurso a um *bitset* por motivos de eficiência de espaço.

A renderização consiste na chamada à função `gluLookAt` e `gluPerspective` com os parâmetros guardadas na `world::camera`, e na chamada às funções `glBegin` e `glEnd`. As coordenadas guardadas em cada modelo são recursivamente encaminhadas à função `glVertex3fv`:

```
auto render() -> void {
    // Algum código omitido por brevidade.
    auto const& camera = state::world_ptr->camera;
    gluLookAt(
        camera.pos.x,    camera.pos.y,    camera.pos.z,
        camera.lookat.x, camera.lookat.y, camera.lookat.z,
        camera.up.x,     camera.up.y,     camera.up.z
    );
    render_group(state::world_ptr->root);
}

auto render_group(group const& root) -> void {
    for (auto const& model : root.models) {
        float const* i = model.coords.data();
        float const* const end = i + model.coords.size();
        glBegin(GL_TRIANGLES);
        for ( ; i != end; i += 3) {
            glVertex3fv(i);
        }
        glEnd();
    }
    for (auto const& child_node : root.children) {
        render_group(child_node);
    }
}

auto resize(int width, int height) -> void {
    // Algum código omitido por brevidade.
    auto const& camera_proj = state::world_ptr->camera.projection;

    state::aspect_ratio
        = static_cast<double>(width) / static_cast<double>(height);

    gluPerspective(
        camera_proj[0], state::aspect_ratio, camera_proj[1], camera_proj[2]
    );
}
```

Finalmente, também foram implementados controlos para mover a câmera, através das funções `glutKeyboardFunc` e `glutKeyboardUpFunc`:

```
glutKeyboardFunc([](unsigned char key, int, int) {
    switch (key) {
        case KEY_MOVE_UP:
        case KEY_MOVE_LEFT:
        case KEY_MOVE_DOWN:
        case KEY_MOVE_RIGHT:
            state::kb.press(key);
            glutPostRedisplay();
            break;
        default:
            break;
    }
});

// Similar for glutKeyboardUpFunc.
```

Os eventos de *input* das teclas são tratados por uma função auxiliar `update_camera`, que é executada a cada 16 milissegundos, para obter um efeito de 60 FPS:

```
auto update_camera(int) -> void {
    auto& camera_pos = state::world_ptr->camera.pos;

    if (kb.pressed(KEY_MOVE_UP) and not kb.pressed(KEY_MOVE_DOWN)) {
        camera_pos.y += config::CAM_TRANSL_FACTOR;
    } else if (kb.pressed(KEY_MOVE_DOWN) and not kb.pressed(KEY_MOVE_UP)) {
        camera_pos.y -= config::CAM_TRANSL_FACTOR;
    }

    // Similar for KEY_MOVE_LEFT and KEY_MOVE_RIGHT.

    glutPostRedisplay();
    glutTimerFunc(config::RENDER_TICK_MILLIS, update_camera, 0);
}
```

4 Resultados

O projeto e todos os testes foram desenvolvidos em Linux. Várias funcionalidades implementadas tiram partido de bibliotecas standard de C++20 usando o compilador g++-11.

Utilizando os ficheiros de configuração XML fornecidos pelos docentes para fins de testes, bem como os modelos especificados nestes, obtemos os seguintes resultados:

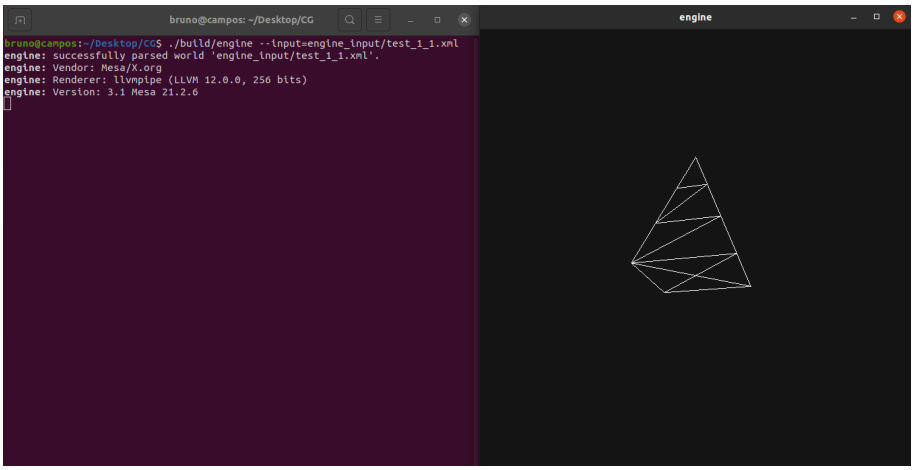

Figura 1. Resultado de test_1_1.xml

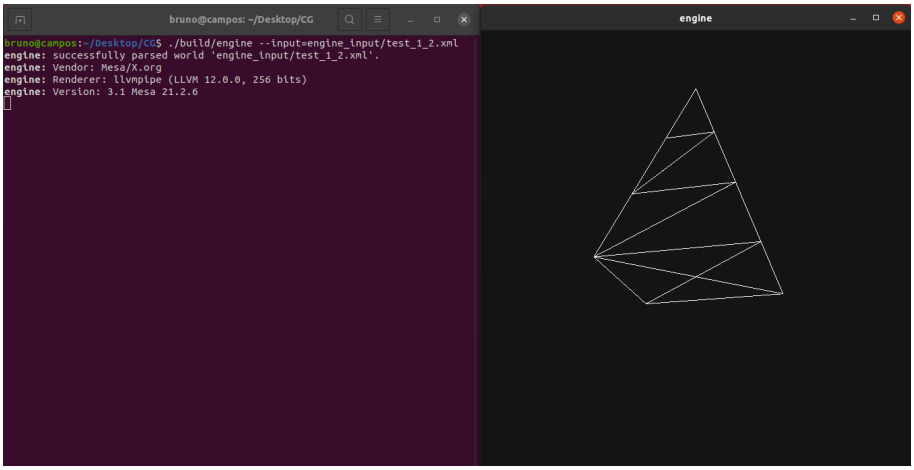

Figura 2. Resultado de test_1_2.xml

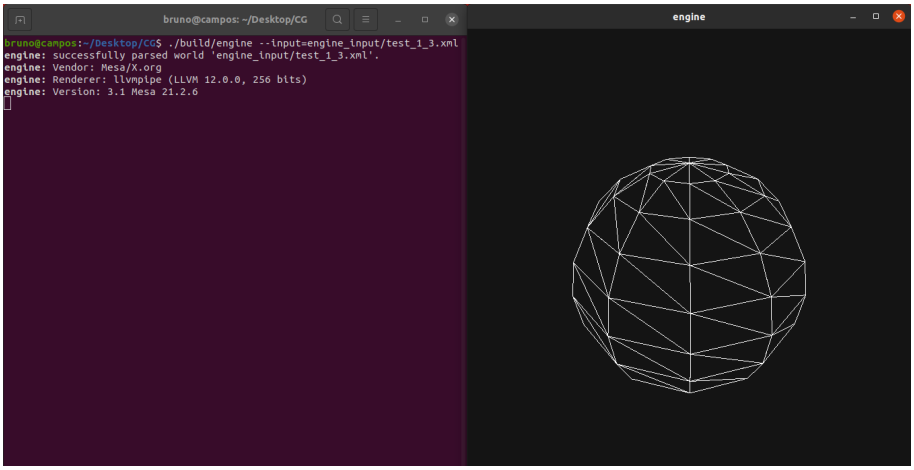

Figura 3. Resultado de test_1_3.xml

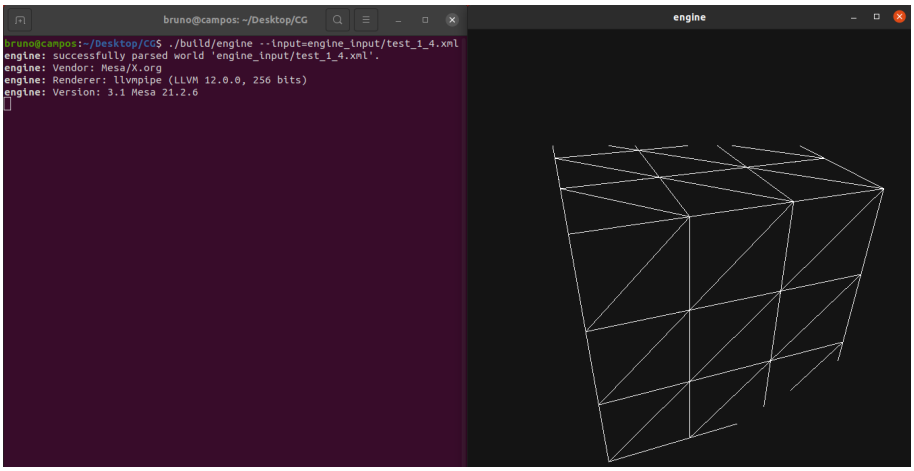

Figura 4. Resultado de test_1_4.xml

Comparando as imagens produzidas pelo programa com as imagens fornecidas juntamente com os ficheiros XML, podemos concluir que foram atingidos os resultados pretendidos.

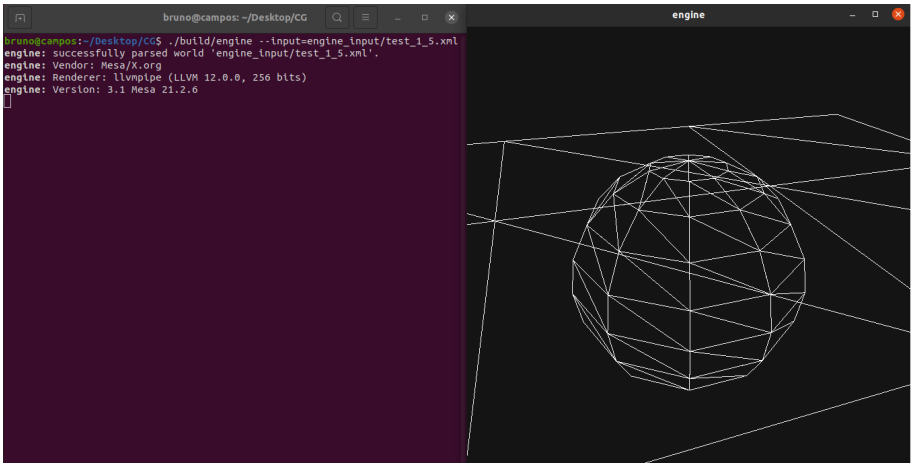

Figura 5. Resultado de test_1_5.xml

5 Conclusão

Finalizando este relatório, podemos afirmar que os problemas propostos para esta fase foram resolvidos com sucesso. Cremos também ter concebido uma boa estrutura de código, que servirá de base para a implementação das próximas fases.

Por outro lado, é de realçar que o formato escolhido para os ficheiros .3d, embora permita representar toda a informação necessária, causa alguma redundância, pelo que deverá ser objeto de otimização no futuro.

Como trabalho futuro, segue-se a implementação de transformações geométricas.