Universidade do Minho
Escola de Engenharia

Computação Gráfica
Fase 2 - Grupo 11
2022/2023

Afonso Xavier Cardoso Marques a94940
Ana Filipa da Cunha Rebelo a90234
Tomás Cardoso Francisco a93193
Simão Paulo da Gama Castel-Branco e Brito a89482

Abril 2023




Índice

List of Figures

1 Introdução

Esta segunda fase do projeto consistiu em desenvolver uma representação do sistema solar num cenário gráfico 3D através da leitura e interpretação de um ficheiro XML e da utilização da primitiva gráfica esfera. Para tal começamos por criar a esfera dado que esta não tinha sido desenvolvida na fase anterior. Para além, foi necessário atualizar o Engine, criado na fase anterior, de modo a conseguir interpretar e armazenar corretamente os novos constituintes do ficheiro XML, armazenando os dados lidos numa estrutura adequada e em seguida, desenhar todos os modelos e transformações em 3D.

2 Alterações à 1 Fase

Tal como referido no relatório da fase anterior, nesta fase começamos por concluir o que faltava da fase anterior nomeadamente a geração da esfera e a geração do cone já tendo em conta o parâmetro stacks.

2.1 Esfera

Para a geração da esfera foi criada a função generateSphere que tem como parâmetros o raio da esfera a ser gerada, o número de slices, número de stacks e o nome do arquivo que será gerado e onde os vértices da esfera serão escritos.

O parâmetro slices determina o número de divisões horizontais da esfera, isto é, o número de fatias que a esfera terá e o parâmetro stacks determina o número de divisões verticais da esfera, ou seja, o número de segmentos ao longo do eixo da esfera tal como é possível verificar através da figura 2.

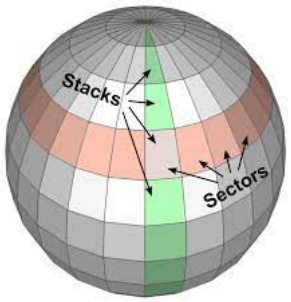

Figure 1: Stacks e Slices da esfera

Quanto maior for o número de slices e stacks mais redonda será a esfera.

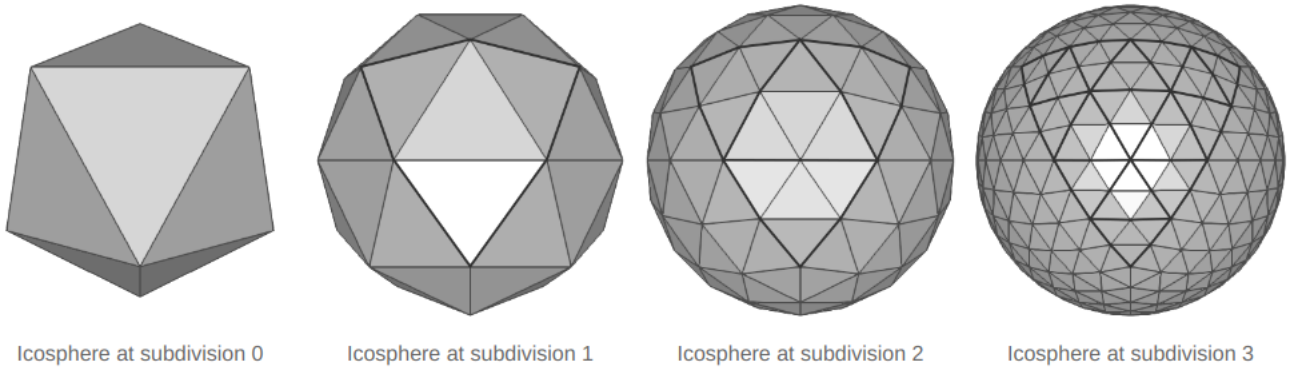

Figure 2: Modo como o número de triângulos afeta a forma da esfera

Através da figura 3 é possível saber a expressão genérica de cada uma das coordenadas de um vértice. Porém, em OpenGL os eixos são diferentes, isto é, o eixo do x da figura corresponde ao eixo z e o eixo z da figura corresponde ao eixo y e o eixo y corresponde ao x. Assim temos:

- x = r * sin(θ) * cos(ϕ)
- y = r * cos(θ)
- z = r * sin(θ) * sin(ϕ)

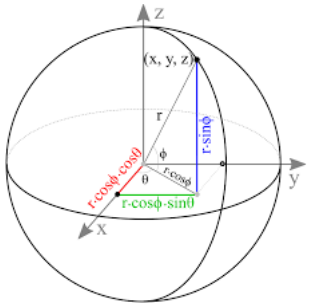

Figure 3: Ponto numa esfera utilizando os ângulos de slices e stacks

onde r é o raio da esfera, θ é o ângulo polar que varia de 0 a π e π é o ângulo azimutal que varia de 0 a 2 π.

A fórmula de indexação de vértices é usada para criar triângulos a partir dos vértices gerados e é também usada para indexar os vértices em um vetor de índices, que é usado para desenhar a esfera na ordem correta.

Para cada triângulo, a fórmula é:

- v1 = rowStart + j
- v2 = nextRowStart + j
- v3 = nextRowStart + j + 1

onde rowStart é o índice do primeiro vértice na linha atual e nextRowStart é o índice do primeiro vértice na próxima linha. As variáveis j e i são usadas para iterar sobre os vértices em cada linha e em cada coluna, respectivamente.

Essas fórmulas são usadas em conjunto para gerar os vértices da esfera e criar triângulos a partir deles.

O cálculo do número total de pontos da figura é feito através da seguinte expressão:

(slices + 1) * (stacks + 1) * 3

onde slices é o número de fatias da esfera (ou seja, o número de segmentos em torno da circunferência da esfera), stacks é o número de camadas da esfera (ou seja, o número de segmentos ao longo do eixo da esfera), e 3 é o número de coordenadas (x, y, z) necessárias para representar cada ponto da esfera.

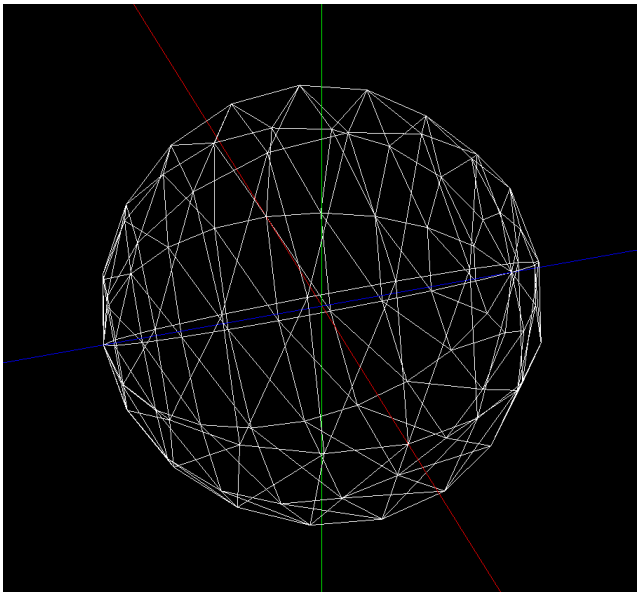

Figure 4: Esfera gerada

2.2 Engine

Nesta fase é nos pedida a alteração do engine de forma a permitir fazer transformações geométricas.Com esse fim,é necessário não só alterar criar uma nova estrutura de dados para guardar as informações do ficheiro como também o modo como o ficheiro XML é lido.

2.2.1 Estrutura de dados implementada

Um dos pontos que o grupo não abordou na primeira entrega refere ao facto que os membros do grupo deviam ter concebido estruturas para guardar as informações extraídas do ficheiro xml em memória, informações estas que serviam como informação sobre os vários ficheiros de modelos usados no desenho das figuras.

Nesta fase no entanto, foram construídas as estruturas para tal, que se passam a demonstrar na figura em baixo:

```
struct transformation {
    string type; // Qual o tipo de transformação?
    float alter_x,alter_y,alter_z,alter_angle; // Variáveis necessárias para transformação GLUT.
};

struct scene {
    vector<int> ordem; // Ordem pela qual são aplicadas os subgroups (1 = Transformation; 2 = Figuras; 3 = Cenas Adicionais)
    vector<vector<transformation>> trans; // Transformações possíveis.
    vector<vector<string>> figuras; // Figuras a desenhar neste group.
    vector<scene> cenas_adicionais; // Quando existe um group dentro de um group.
};

struct storage {
    float width, height; // Definições da Janela.
    float position_x, position_y, position_z; // Posição da Câmera.
    float lookat_x, lookat_y, lookat_z; // Para onde a Câmera está a olhar.
    float up_x, up_y, up_z; // Variável Up.
    float fov, near, far; // Projeção da Câmera.
    vector<scene> cenas; // Várias modelos e transformações a desenhar.
} xml_data;
```

Figure 5: Estrutura do XML

2.2.2 Leitura do Ficheiro XML

Para a leitura do Ficheiro XML foram criadas as funções *set_up_from_files* e *group_set_up*.

A função *set_up_from_files* foi criada como forma de guardar a informação do ficheiro xml fornecido numa variável global chamada de ***xml_data***. *xml_data* é uma variável global do tipo *transformation*(um dos structs mostrados anteriormente). A função *set_up_from_files* tem como objectivo central preencher os vários campos desta variável de forma a que informação fique sempre salvaguardada em memória.

Um pequeno detalhe que é preciso apontar é que o nosso engine agora faz parsing com um novo programa de parsing para ficheiros XML, o **tinyxml2**. A razão para esta troca foi pelo facto que, com a nova estrutura apresentada nos ficheiros XML, seria extremamente útil utilizar as variáveis do programa de parsing a nivel global. Mas, durante a tentativa de implementação, a equipa reparou que o programa **rapidXML** tinha os seus atributos

definidos como **Non-Copyable**. Isto significa que só poderiam ser usados na função em que eram invocados, e por isso, o grupo tomou a decisão de sacrificar alguma velocidade do trabalho para utilizar um parser mais adequado aos nossos objetivos.

Agora com um novo parser, a função acede ao ficheiro cujo o nome é fornecido como argumento, e procura no específico path para a diretoria onde se encontram os ficheiros xml de teste. Após se concluir que o ficheiro carregou com sucesso para a variável de **XMLDocument**(variável de tinyxml2 para criar o documento), acedemos aos vários **ChildElements** e **Attributes** dos vários nodos consecutivos do XMLDocument e guardaremos as informações da janela e da câmara facilmente. No entanto, quando se trata das figuras que se têm de desenhar, isto torna-se um pouco mais díficl. E para isso, usaremos a função *group_set_up*.

A função *group_set_up* recebe como argumento um ponteiro para um elemento XML, que irá sempre representar um grupo, e retorna uma *scene(struct)* que contém informações acerca da ordem dos elementos do grupo, transformações a serem aplicadas, modelos 3D a serem carregados e cenas adicionais caso exista um *group* dentro de um *group*. Desta forma, os grupos podem ser criados de forma não só iterativa, como também como recursiva, o que torna possível criar grupos dentro de grupos, representando a stack de matrizes e os próprios *nested groups* que agora podem existir no ficheiro xml.

Após a *scene* ter sido criada, ela retorna à função *set_up_from_files*, onde é adicionada a um vector que contém as outras possíveis cenas a serem desenhadas. Após o vector estar terminado, este é adicionado ao xml_data e agora, a nossa variável contém toda a informação sobre o ficheiro xml, sendo possível o fecharmos, automaticamente feito graças à biblioteca de tinyxml2.

Na secção dos anexos é possivel encontrar as funções referidas nesta porção do relatório.

3 Estrutura do Ficheiro XML

Para esta fase foi necessário criar um ficheiro XML mais complexo do que o da fase anterior. Neste foram adicionadas transformações a serem feitas às primitivas gráficas tais como, rotação, translação e escala.

As primitivas agora são organizadas em grupos(*group*) encadeados, em que cada um contém a transformação a ser aplicada, o que permite haver agrupamento de várias primitivas no XML. Assim, as transformações aplicadas ao grupo principal aplicam-se aos seus subgrupos. Na figura 6, referente à representação do Sol, é possível observar a hierarquia *groups* bem como a representação das diferentes transformações.

```
<group>
    <transform>
        <translate x="0" y="0" z="0" />
    </transform>
    <group>
        <transform>
            <rotate angle="0" x="0" y ="1" z = "0" />
            <translate x="0" y="0" z="0" />
            <scale x="1.2" y="1.2" z="1.2" />
        </transform>
                    <models>
                            <model file="sphere.3d" /> <!-- generator sphere 1 8 8 sphere.3d -->
                    </models>
    </group>
```

Figure 6: Excerto de um ficheiro XML

4 Sistema Solar

Para o sistema solar, o grupo tentou replicar os modelos do sistema solar que se encontram em livros escolares. A representação foi feita tendo em conta apenas o trabalho desenvolvido no decorrer das duas primeiras fases, pelo que as formas dos corpos celestes são esferas simples deprivadas de cor ou textura.

Os planetas são desenhados na ordem esperada e com um tamanho apropriado a uma distância pequena uns dos outros para permitir que todos estejam visíveis. Visto que se trata de um modelo estático onde não existe movimento da câmara, foi necessário aumentar os valores da matriz desta, bem como o tamanho da janela no XML criado, de forma a garantir que todos os planetas estão enquadrados na imagem gerada.

A seguinte imagem mostra o resultado obtido:

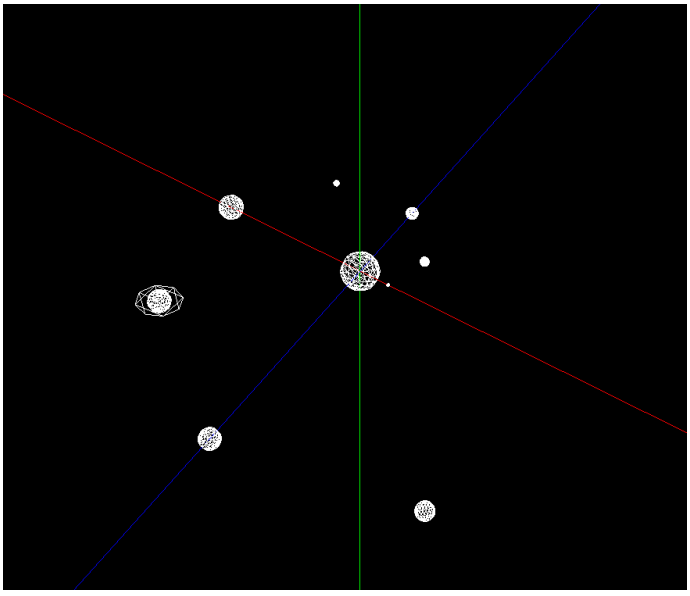

Figure 7: Modelo 3D do sistema solar

5 Conclusão

No decorrer desta fase foi possível melhorar o trabalho inicialmente concebido na primeira fase bem como adicionar as alterações necessárias para cumprir com os requisitos da segunda fase. Sentimos melhorias nas nossas capacidades de utilização do OpenGL e Glut e também na leitura de ficheiros XML.

Dada por concluída a segunda fase do projeto, consideramos importante realizar uma análise crítica, e ainda, realçar os pontos positivos e negativos. Durante o desenvolvimento do projeto surgiram algumas dificuldades, tal como descobrir como gerar a esfera que faltava concluir da primeira fase. Também foi dificil para o grupo alterar o funcionamento do engine de forma a que este pudesse lidar com as transformações das figuras.

Por outro lado, apesar das dificuldades encontradas o grupo conseguiu implementar a totalidade das funcionalidades propostas o que consideramos ser um ponto positivo do trabalho.

6 Anexos

```
void set_up_from_files(const char* xml_file) {
    XMLDocument doc;
    if (doc.LoadFile(xml_file) == XML_SUCCESS) {
        XMLElement* world = doc.RootElement();

        // Settings para Window
        XMLElement* janela = world->FirstChildElement("window");
        xml_data.width = stof(janela->Attribute("width"));
        xml_data.height = stof(janela->Attribute("height"));

        // Settings para Camera
        XMLElement* camera = world->FirstChildElement("camera");

        XMLElement* position = camera->FirstChildElement("position");
        xml_data.position_x = stof(position->Attribute("x"));
        xml_data.position_y = stof(position->Attribute("y"));
        xml_data.position_z = stof(position->Attribute("z"));

        XMLElement* lookat = camera->FirstChildElement("lookAt");
        xml_data.lookat_x = stof(lookat->Attribute("x"));
        xml_data.lookat_y = stof(lookat->Attribute("y"));
        xml_data.lookat_z = stof(lookat->Attribute("z"));

        XMLElement* up = camera->FirstChildElement("up");
        xml_data.up_x = stof(up->Attribute("x"));
        xml_data.up_y = stof(up->Attribute("y"));
        xml_data.up_z = stof(up->Attribute("z"));

        XMLElement* projection = camera->FirstChildElement("projection");
        xml_data.fov = stof(projection->Attribute("fov"));
        xml_data.near = stof(projection->Attribute("near"));
        xml_data.far = stof(projection->Attribute("far"));

        // Figuras para desenhar
        vector<scene> scenes;
        for (XMLElement* group = world->FirstChildElement("group"); group; group = group->NextSiblingElement()) {
            scene cena = group_set_up(group);
            scenes.push_back(cena);
        }
        xml_data.cenas = scenes;

    } else {
        cerr << "Erro: Ficheiro não foi aberto sucessivamente." << endl;
```

Figure 8: Função *set_up_from_files*

```
scene group_set_up(XMLElement* group) {
    scene result;
    vector<int> order;
    vector<vector<transformation>> transformations;
    vector<vector<string>> ficheiros;
    vector<scene> adicionais;

    for (XMLElement* subgroup = group->FirstChildElement(); subgroup; subgroup = subgroup->NextSiblingElement()) {
        if (strcmp(subgroup->Name(),"transform") == 0) {
            order.push_back(1);
            vector<transformation> trans;
            for (XMLElement* transform = subgroup->FirstChildElement(); transform; transform = transform->NextSiblingElement()) {
                transformation t;
                if (strcmp(transform->Name(),"translate") == 0) {
                    t.type = "translate";
                    t.alter_x = stof(transform->Attribute("x"));
                    t.alter_y = stof(transform->Attribute("y"));
                    t.alter_z = stof(transform->Attribute("z"));
                    trans.push_back(t);
                } else if (strcmp(transform->Name(),"rotate") == 0) {
                    t.type = "rotate";
                    t.alter_angle = stof(transform->Attribute("angle"));
                    t.alter_x = stof(transform->Attribute("x"));
                    t.alter_y = stof(transform->Attribute("y"));
                    t.alter_z = stof(transform->Attribute("z"));
                    trans.push_back(t);
                } else if (strcmp(transform->Name(),"scale") == 0) {
                    t.type = "scale";
                    t.alter_x = stof(transform->Attribute("x"));
                    t.alter_y = stof(transform->Attribute("y"));
                    t.alter_z = stof(transform->Attribute("z"));
                    trans.push_back(t);
                } else {
                    cerr << "Erro: Transformação inválida encontrada." << endl;
                }
            }
            transformations.push_back(trans);
        } else if (strcmp(subgroup->Name(),"models") == 0) {
            order.push_back(2);
            vector<string> files;
            for (XMLElement* model = subgroup->FirstChildElement(); model; model = model->NextSiblingElement()) {
                string s;
                files.push_back(s.append(dot_file_path).append(model->Attribute("file")));
```

Figure 9: Função *group_set_up*

```
                cerr << "Erro: Transformação inválida encontrada." << endl;
            }
        }
        transformations.push_back(trans);
    } else if (strcmp(subgroup->Name(),"models") == 0) {
        order.push_back(2);
        vector<string> files;
        for (XMLElement* model = subgroup->FirstChildElement(); model; model = model->NextSiblingElement()) {
            string s;
            files.push_back(s.append(dot_file_path).append(model->Attribute("file")));
        }
        ficheiros.push_back(files);
    } else if (strcmp(subgroup->Name(),"group") == 0) {
        order.push_back(3);
        scene cena = group_set_up(subgroup);
        adicionais.push_back(cena);
    } else {
        cerr << "Erro: Membro de grupo não existe." << endl;
    }
}

result.ordem = order; result.trans = transformations; result.figuras = ficheiros; result.cenas_adicionais = adicionais;
return result;
```

Figure 10: Continuação da *função group_set_up*