Computação Gráfica

LEI - 3º ano - 2º semestre
Universidade do Minho

Trabalho Prático

Fase 2

Grupo 10

Trabalho realizado por:	*Número*
Duarte Moreira	A93321
Lucas Carvalho	A93176
Manuel Novais	A89512

Braga, 4 de abril de 2022

Índice

1. Introdução

O presente relatório foi realizado no âmbito da Unidade Curricular de Computação Gráfica e tem como objetivo expor detalhadamente as decisões e abordagens tomadas pelo grupo durante a fase de desenvolvimento do projeto.

Este trabalho, dividido em 4 fases, tem como objetivo final a criação de um mini cenário baseado em gráficos 3D. Nesta segunda fase é-nos proposto que atualizemos a nossa primeira fase do trabalho de forma a conseguirmos lidar com a criação de cenários hierárquicos usando transformações geométricas, tais como, **translações**, **rotações** e **escalas**. Para tal, apenas foi necessário atualizar a aplicação **engine**.

2. Gerador

Apesar de não ser necessário alterar esta aplicação para o desenvolvimento da segunda fase o grupo descobriu uma pequena gralha, vinda da fase um, que corrigiu de imediato. Este erro baseava-se apenas na criação dos vértices da primitiva gráfica **caixa**, ou seja, as nossas caixas estavam a ser "desenhadas"tendo o plano *y=0* como base, enquanto que, as da equipa docente eram centradas nos eixos. Estas diferenças eram notórias na resolução dos ficheiros *xml* fornecidos.

3. Motor

Para esta fase, a estrutura dos ficheiros *xml* fornecidos foi alterada o que levou também a uma alteração drástica do código desta aplicação. Para lidar com o problema segui-se uma estratégia de *divide and conquer*, ou seja, dividiu-se o problema em partes mais pequenas para no final juntarmos todas.

Como podemos observar pela Figura 3.1, um ficheiro *xml* encontra-se divido em diferentes grupos, sendo eles: **<camera>**, **<group>**, **<transform>** e **<models>**.

```
<world>
    <camera>
        <position x="10" y="10" z="10" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
         <projection fov="60" near="1" far="1000" />
     </camera>
    <group>
        <transform>
            <translate x="4"  y="0" z="0" />
            <rotate angle="30" x="0" y="1" z="0" />
            <scale x="2" y="0.3" z="1" />
        </transform>
        <models>
            <model file="cone.3d" />
            <model file="plane.3d" />
        </models>
    </group>
</world>
```

Figura 3.1: Ficheiro *xml*.

O primeiro, **<camera>**, apresenta as variáveis a serem usadas nas configurações da câmara.

O **<group>**, é um conjunto que contém um grupo de **<transform>**, um de **<models>** e pode ou não apresentar mais conjuntos de **<group>**.

Já o grupo **<transform>** apresenta uma listagem das transformações geométricas a realizar assim como os seus parâmetros.

Por fim, o grupo **<models>** contém a listagem do nome dos modelos 3d a carregar.

3.1 Camera

Para o tratamento da câmara foram criadas, uma classe **Camera**, que contém como variàveis de instância os parâmetros encontrados no ficheiro *xml* e uma função de *get* e *set* para manipular os mesmos. Para auxiliar no *parsing* deste grupo foi também criada a função *parseCamera* que utiliza a mesma estratégia usada anteriormente na fase um, apenas com a diferença de que agora, o conteúdo lido será guardado numa instância da classe **Camera**.

```
class Camera {
        private:
                int x_pos, y_pos, z_pos;
                (...)
        public:
                Camera();
                int getXPos(); int getYPos(); int getZPos();
                (...)
                void setValues(int x_pos, (...));
};

void parseCamera(XMLElement* camera, Camera* c);
```

3.2 Model

De maneira similar o conjunto **Model** reutiliza código usado na primeira fase. No entanto agora, por cada modelo 3d lido é criado uma instância da classe **Model** que posteriormente são adicionados à instância que representa o **Group** em questão. Esta classe apresenta uma função *draw()* que usará os pontos carregados dos modelos, nas funções fornecidas pelo **OpendGl** para criar os triângulos necessários.

```
class Model {
        private:
                vector<Ponto> pontos;
        public:
                Model(vector<Ponto> pontos);
                void draw();
};

void parseModels(XMLElement* models, Group* group) {
    XMLElement* model = models->FirstChildElement("model");
    for (; model != NULL; model = model->NextSiblingElement("model")) {
                string file3d = model->Attribute("file");
                vector<Ponto> pontos = modelReader(pathxml + file3d);
                group->addModel(new Model(pontos));
        }
}
```

3.3 Transform

A classe **Transform** foi criada com o objetivo de armazenar uma de três transformações geométricas: translação, rotação e escala. Para além de funções de *set* para manipular as suas variáveis foi definida a função *run()* que é usada para aplicar a transformação em questão.

```
class Transform {
        private:
                Translate* translate;
                Rotate* rotate;
                Scale* scale;
        public:
                Transform();
                void setTranslate(Translate* t);
                void setRotate(Rotate* r);
                void setScale(Scale* s);
                void run();
};
```

Quanto ao *parsing* do conjunto de transformações geométicas este é realizado de maneira a que a ordem seja mantida, guardando cada transformação lida (instância da classe **Transform**) no vector de transformações do **Group** em questão.

```
void parseTransform(XMLElement* transform, Group* group) {
    XMLElement* t = transform->FirstChildElement();
    for (; t != NULL; t = t->NextSiblingElement()) {
        string type = t->Value();

                Transform* transform = new Transform();

        if (type.compare("translate") == 0) {
            float x = t->FloatAttribute("x");
            float y = t->FloatAttribute("y");
            float z = t->FloatAttribute("z");

                        transform->setTranslate(new Translate(x, y, z));
        }
            else if (type.compare("rotate") == 0) {
            (...)
            }
            else if (type.compare("scale") == 0) {
            (...)
            }
                else cout << "ERROR parsing transform " << type << endl;

                group->addTransform(transform);
        }
}
```

3.4 Group

A classe de maior dimensão - **Group** - irá conter um vector de **Transform**, um vector de **Model** e um vector de **Group** para lidar com os grupos aninhados, permitindo assim uma noção de hierarquização. A classe faz uso de uma função *draw()* que irá, primeiramente, aplicar as tranformações geométricas, de seguida desenhar os modelos lidos e por fim, de maneira recursiva, fazer *draw()* dos subgrupos, caso existam.

```
class Group {
        private:
                vector<Transform*> transforms;
                vector<Model*> models;
                vector<Group*> subgroups;
        public:
                (...);
                void draw();
};

(...)

void Group::draw() {
        glPushMatrix();
        for (Transform* transform : this->transforms)
                transform->run();
        for (Model* model : this->models)
                model->draw();
        for (Group* group : this->subgroups)
                group->draw();
        glPopMatrix();
}
```

Para fazer *parsing* deste conjunto do ficheiro *xml*, são apenas utilizados os parsers anteriores para os respetivos conjuntos, **Model** e **Transform** e, caso seja necessário, chama-se de forma recursiva a função *parseGroup()*.

```
void parseGroup(XMLElement* group, Group* grupo) {
    (...)
        if (type.compare("transform") == 0) {
                        parseTransform(elem, grupo);
        }
                else if (type.compare("models") == 0) {
                        parseModels(elem, grupo);
                }
                else if (type.compare("group") == 0) {
                        Group* subgroup = new Group();
                        grupo->addSubGroup(subgroup);
                        parseGroup(elem, subgroup);
                }
                else cout << "ERROR parsing group " << type << endl;
        }
}
```

4. Transformações Geométricas

4.1 Translação

As translações são essencialmente movimentações em linha reta para um determinado ponto a partir das coordenadas originais, não alterando a figura tanto nas suas dimensões como na sua forma. Para isso, é necessário saber o vetor de direção do movimento para calcular a posição final do objeto

Assim, foi criada a seguinte classe *Translate*:

```
class Translate {
    private:
        float x, y, z;
    public:
        Translate(float x, float y, float z);
        void run();
};
```

Por ultimo, para aplicar a transformação, é necessário o uso da primitiva do OpenGL através do seguinte método:

```
void Translate::run() {
    glTranslatef(this->x, this->y, this->z);
}
```

Como podemos observar pela imagem abaixo a translação do objeto esta a devolver o resultado esperado.

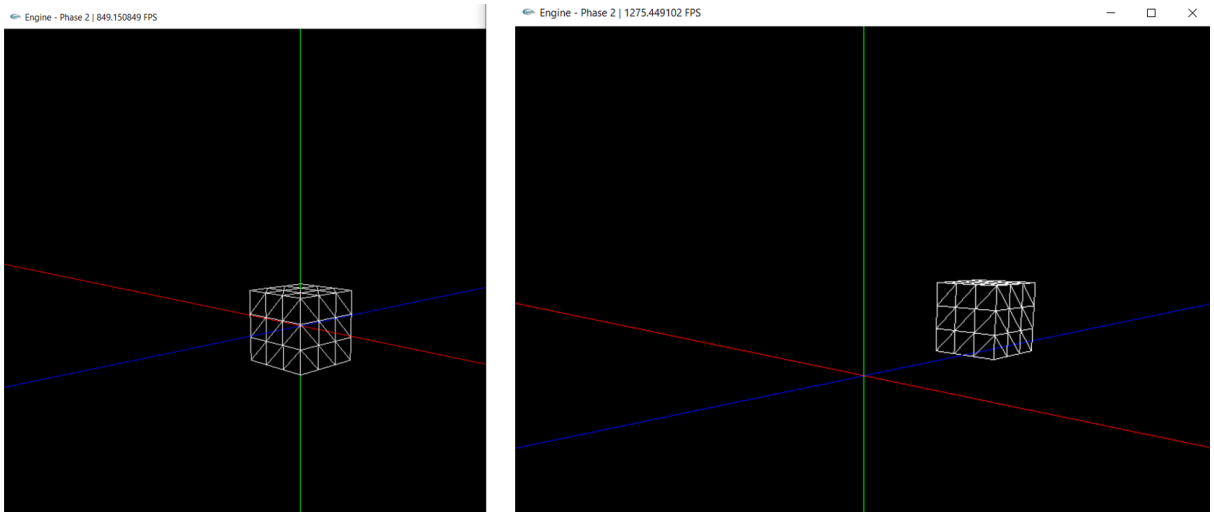

Figura 4.1: glTranslatef(0,1,-5)

4.2 Rotação

Para a rotação de um objeto é necessário definir o ângulo de rotação e as coordenadas do vetor que funciona como eixo do movimento. O objetivo é a rotação do objeto sem que este sofra quaisquer outras alterações

Para isso foi criada a classe:

```
class Rotate {
    private:
        float angulo, x, y, z;
    public:
        Rotate(float angulo, float x, float y, float z);
        void run();
};
```

Assim como para a translação foi também necessário usar a primitiva do OpenGL da seguinte forma:

```
void Rotate::run() {
    glRotatef(this->angulo, this->x, this->y, this->z);
}
```

Abaixo temos um teste que prova que a rotação foi corretamente implementada.

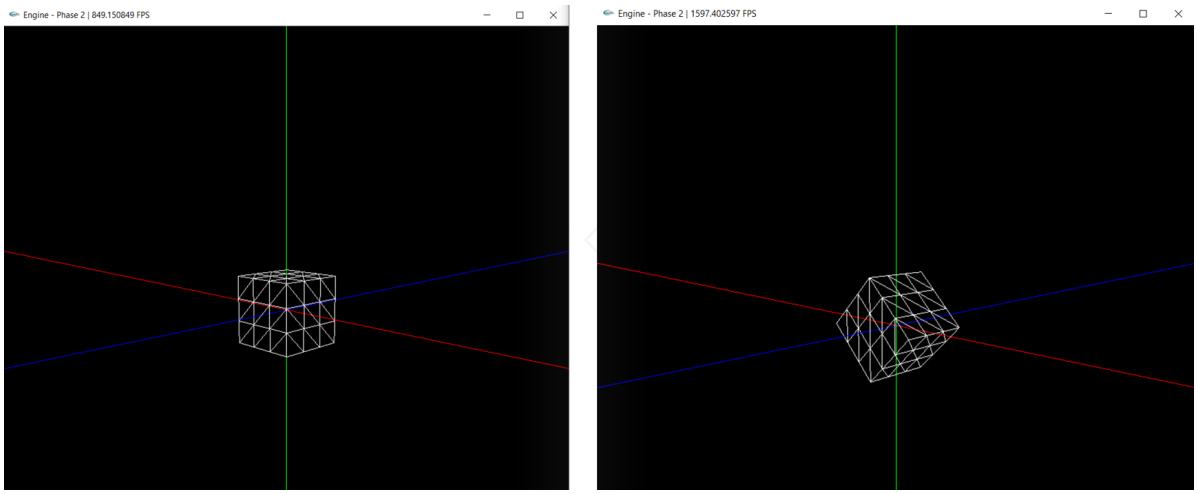

Figura 4.2: glRotatef(45,0,0,1)

4.3 Escala

A última das transformações geométricas é a escala. Escalar uma figura é essencialmente encolher ou expandir a mesma, enquanto que as transformações anteriores mantinham a forma do objeto, o propósito desta é alterá-la. Para isso é definido um fator de escala para cada eixo de forma a multiplicar as coordenadas do objeto por ele.

Foi então usada a seguinte classe *Scale*:

```
class Scale {
    private:
        float x, y, z;
    public:
        Scale(float x, float y, float z);
        void run();
};
```

Tal como as anteriores com o uso do OpenGL foi criado o seguinte método para a transformação:

```
void Scale::run() {
    glScalef(this->x, this->y, this->z);
}
```

O teste abaixo mostra que foi alcançado o resultado desejado

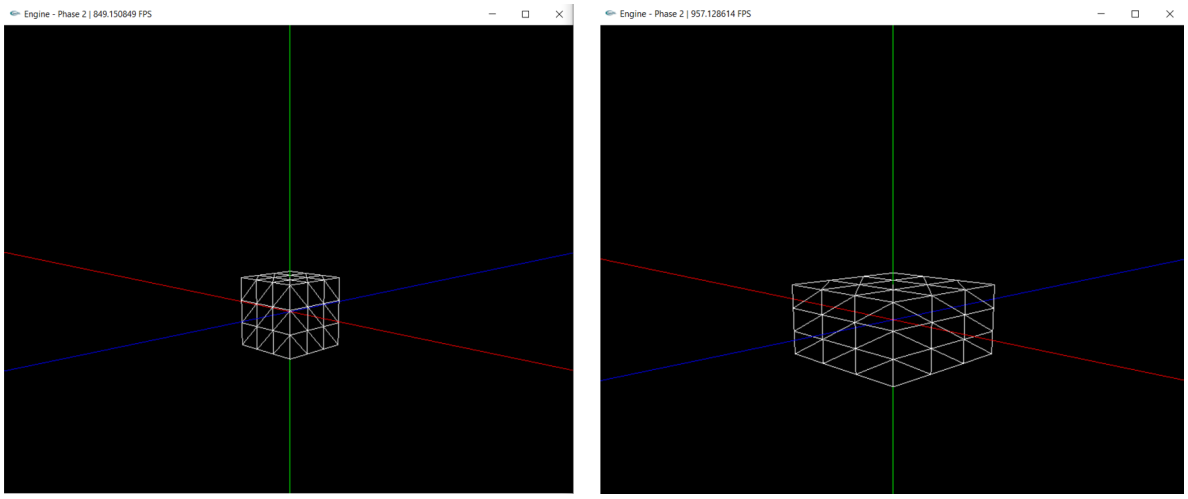

Figura 4.3: glScalef(2,1,2)

5. Resultados Obtidos

5.1 Testes

Apresentam-se a seguir os resultados obtidos com as nossas aplicações de acordo com os ficheiros de testes fornecidos pela equipa docente.

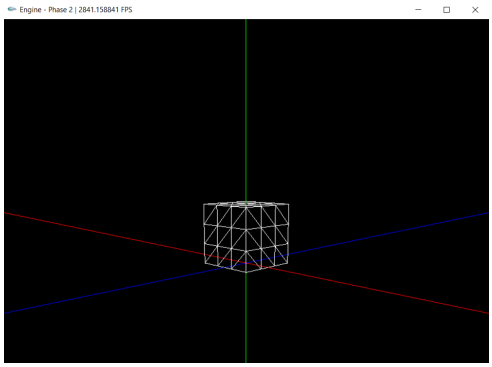

Figura 5.1: Resultado de *test_2_1.xml*

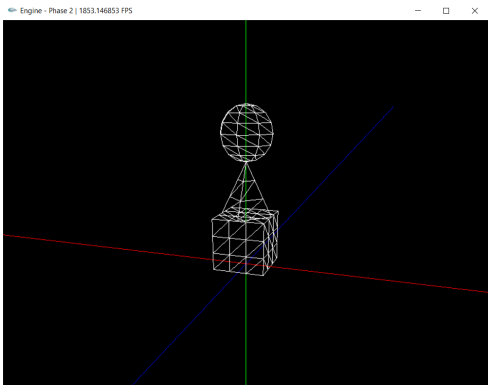

Figura 5.2: Resultado de *test_2_2.xml*

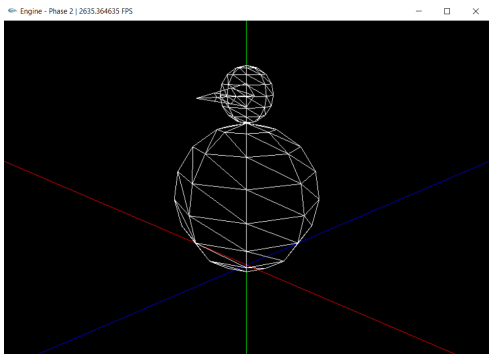

Figura 5.3: Resultado de *test_2_3.xml*

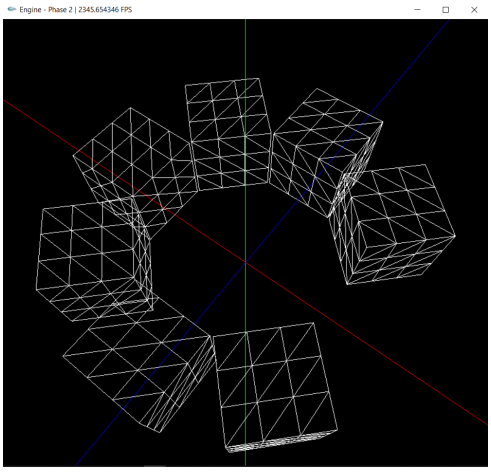

Figura 5.4: Resultado de *test_2_4.xml*

5.2 Cenários de demonstração

De seguida expõem-se o cenário de demonstração pretendido para esta fase: o sistema solar. Tomando isto como um exemplo académico o grupo decidiu construir um sistema solar à escala para os tamanhos mas não para as distâncias. Para além disso, damos ênfase ao facto de termos colocado tanto os planetas como as luas no plano *z=0* de forma a facilitar a visualização do resultado.

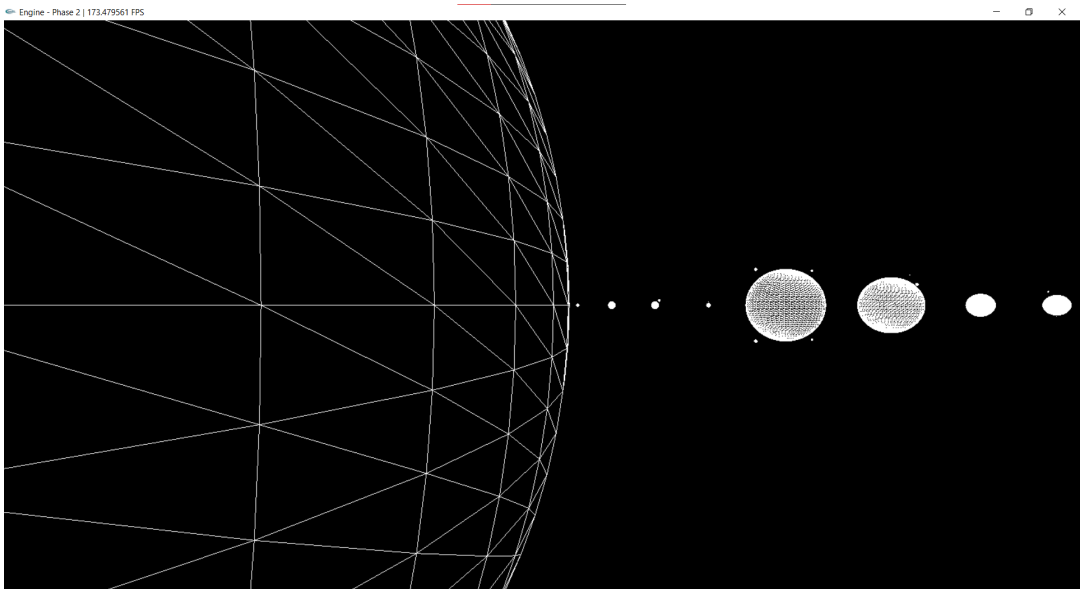

Figura 5.5: Sistema solar.

6. Extras

Com o objetivo de melhorar o aproveitamento da utilização da aplicação o grupo decidiu substituir a capacidade de controlar a movimentação da câmara com as teclas por um controlo auxiliado pelo rato, permitindo assim uma movimentação mais fluída e uma experiência mais proveitosa. O mesmo se aplica à capacidade de dar zoom in/out da imagem.

Para além disso e com o intuito de analisar a performance do programa adicionou-se um pequeno detalhe que permite ao utilizador consultar os *fps* - frames per second - da aplicação em tempo real.

7. Conclusão

Concluindo, o grupo ficou satisfeito com o resultado obtido uma vez que foi implementado tudo que nos foi pedido no enunciado.

Salienta-se a importância das transformações, dado que o seu uso facilitou bastante o processo de modelação do sistema solar.

Com um pouco mais de tempo a equipa gostaria de ter explorado mais funcionalidades extras, como modelação de cenários diferentes e a adição de leitura de cores no ficheiro *xml*, uma vez que essa foi a parte que deixou um bocado mais a desejar.

Por último, concluída esta fase, o objetivo é começar já a trabalhar na terceira fase, uma vez que se notou uma má gestão do tempo por parte do grupo.