Universidade do Minho
Licenciatura em Ciências da Computação

Unidade Curricular de Computação Gráfica

Ano Lectivo de 2021/2022

Phase 2 – Geometric Transforms

Filipe Azevedo A87969

João Nogueira A87973

Miguel Gonçalves A90416

Rui Baptista A87989

CG

Resumo

Nesta segunda fase iremos aplicar o conhecimento adquirido nas ultimas semanas, nomeadamente sobre a hierarquia das transformações geométricas, de forma a ser possível desenhar várias figuras geométricas e várias transformações.

Palavras-Chave: Computação Gráfica, Figuras Geométricas, XML, GLUT, Engine, Transformações.

Índice

Índice de Figuras

1. Introdução

No âmbito da unidade curricular de Computação Gráfica foi proposta, nesta segunda fase, se implementasse um sistema capaz de desenhar um conjunto, qualquer, de figuras, figuras estas que foram feitas anteriormente na primeira fase do trabalho, mas à base de translações, rotações e escalas.
Sendo assim, foi necessário alterar a forma como tinhamos definido o parse na primeira versão do XML realizado. Posto isto, foi necessária uma modificação neste aspeto e ainda na forma como se desenhava, pois agora temos de incluir as transformações geométricas, referidas em cima.

2. Estrutura do Projeto

A estrutura do projeto nesta segunda fase mantém-se igual à primeira fase. Como já foi visto, no generator geram-se os pontos, guardando-os num ficheiro .3d, que vão ser usados para desenhar as figuras através de ficheiros XML. No engine, criamos as classes auxiliar que facilitam a leitura de ficheiros.

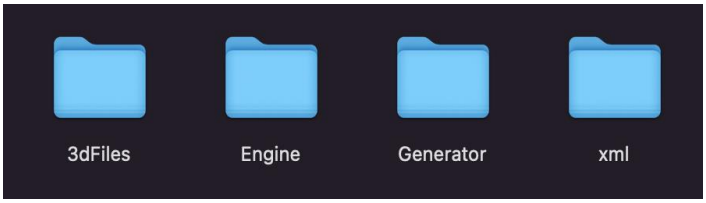


Figura 1 - Estrutura

3. Alterações no Engine

Nesta fase...

3.1 Classes

Em termos de alocação de memória da informação das transformações de cada grupo, decidimos implementar uma classe para cada uma das transformações.

3.1.1 Rotação

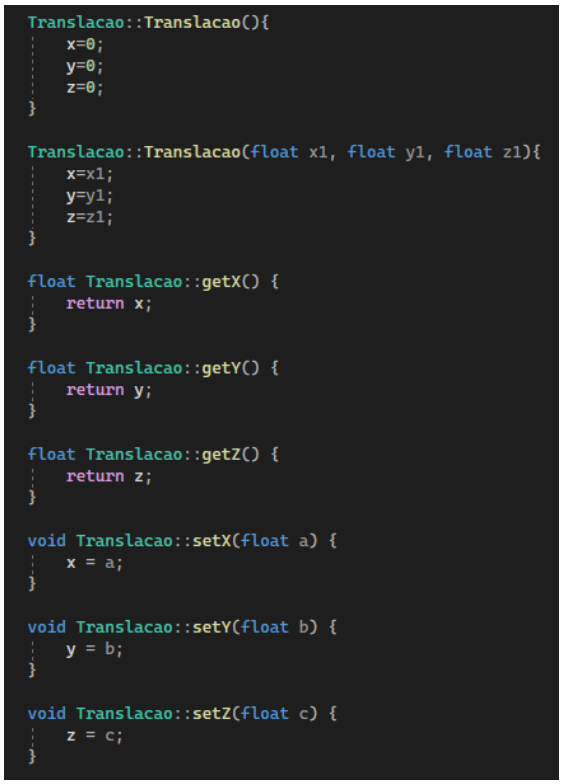
Cada rotação terá 4 atributos, sendo respetivamente: o angulo de rotacao, o eixo X, o eixo Y e por fim o Z. Serão também desenvolvidos os respetivos métodos para obter os valores de cada rotação, e para guardar
os valores. Temos também 2 construtores, um com argumentos e outro sem argumentos que iniciará todos os valores a 0.

```
Rotacao::Rotacao(){
    angulo=0;
    x=0;
    y=0;
    z=0;
}

Rotacao::Rotacao(float x1, float y1, float z1, float angulo1){
    angulo=angulo1;
    x=x1;
    y=y1;
    z=z1;
}

float Rotacao::getAngulo(){
    return angulo;
}

float Rotacao::getX() {
    return x;
}

float Rotacao::getY() {
    return y;
}

float Rotacao::getZ() {
    return z;
}

void Rotacao::setAngulo(float angulo1){
    angulo = angulo1;
}

void Rotacao::setX(float a) {
    x = a;
}

void Rotacao::setY(float b) {
    y = b;
}

void Rotacao::setZ(float c) {
    z = c;
}
```

Figura 2 - Classe Rotação

3.1.2 Translação

Cada translação terá 3 atributos, sendo eles, respetivamente: o valor a mover pelo eixo X, o valor a mover pelo eixo Y e por fim o valor a mover pelo eixo Z. Serão desenvolvidos os respetivos métodos para conseguirmos obter os valores de cada translação e para guardarmos os valores também. Temos também 2 construtores, um com argumentos e outro sem argumentos que iniciará todos os valores a 0.



```
Translacao::Translacao(){
    x=0;
    y=0;
    z=0;
}

Translacao::Translacao(float x1, float y1, float z1){
    x=x1;
    y=y1;
    z=z1;
}

float Translacao::getX() {
    return x;
}

float Translacao::getY() {
    return y;
}

float Translacao::getZ() {
    return z;
}

void Translacao::setX(float a) {
    x = a;
}

void Translacao::setY(float b) {
    y = b;
}

void Translacao::setZ(float c) {
    z = c;
}
```

Figura 3 - Classe Translação

3.1.3 Escala

Cada escala terá 3 atributos, sendo eles, respetivamente: o valor da escala para o eixo X, ovalor da escala para o eixo Y e por fim o valor da escala para o eixo Z. Serão também desenvolvidos os respetivos métodos para conseguirmos obter os valores de cada escala para guardarmos os valores também . Temos também 2 construtores, um com argumentos e outro sem argumentos que iniciará todos os valores a 1.

```
Escala::Escala(){
    x=1;
    y=1;
    z=1;
}

Escala::Escala(float x1, float y1, float z1){
    x=x1;
    y=y1;
    z=z1;
}

float Escala::getX() {
    return x;
}

float Escala::getY() {
    return y;
}

float Escala::getZ() {
    return z;
}

void Escala::setX(float a) {
    x = a;
}

void Escala::setY(float b) {
    y = b;
}

void Escala::setZ(float c) {
    z = c;
}
```

Figura 4 - Classe Escala

3.1.4 Ponto

Como na fase 1, precisamos de usar a estrutura “Ponto” que contém as coordenadas x, y e z de um Ponto, para guardarmos os pontos da figura em memória, sendo usada uma lista de pontos para cada modelo.

```
Ponto::Ponto() {
    x = 0;
    y = 0;
    z = 0;
}

Ponto::Ponto(float a, float b, float c) {
    x = a;
    y = b;
    z = c;
}

float Ponto:: getX() {
    return x;
}

float Ponto:: getY() {
    return y;
}

float Ponto:: getZ() {
    return z;
}

void Ponto:: setX(float a) {
    x = a;
}

void Ponto:: setY(float b) {
    y = b;
}

void Ponto:: setZ(float c) {
    z = c;
}
```

Figura 5 - Classe Ponto

3.1.5 Modelos

Esta classe recebe apenas um argumento que é uma lista com os pontos para desenhar de um dado modelo.

```
Modelo::Modelo(){
}

Modelo::Modelo(list<Ponto> pontosLista1) {
    pontosLista = pontosLista1;
}

list<Ponto> Modelo::getPontos() {
    return pontosLista;
}

void Modelo::setPontos(list<Ponto> pontosLista1) {
    pontosLista = pontosLista1;
}
```

Figura 6 - Classe Modelo

3.1.1 Grupo

Para a alocação em memória da informação dos grupos, decidimos criar uma classe Grupo, que vai conter as transformações desse grupo, a ordem, os modelos e os filhos, ou seja, os subgrupos.

Para as transformações foi criada uma classe para guardar a informação de cada uma delas, sendo as transformações do grupo dessa mesma classe.

Para a ordem das transformações temos uma String que vai conter 3 caracteres separados por um espaço sendo cada caracter representativo de uma transformação, sendo a ordem dos mesmos, a ordem das transformações.

Para os modelos usamos uma lista e para os filhos usamos uma lista de grupos.

Para finalizar, implementamosos respetivos construtores, getters, setters e as funções addFilho e addModelo para adicionar um filho e um modelo no fim da lista de filhos e modelos.

```
Grupo::Grupo() {
    esc = *new Escala();
    trans = *new Translacao();
    rot = *new Rotacao();
}


Escala Grupo::getEscala() {
    return esc;
}
void Grupo::setEscala(Escala e) {
    esc = e;
}

Translacao Grupo::getTranslacao() {
    return trans;
}

void Grupo::setTranslacao(Translacao t) {
    trans = t;
}

void Grupo::setRotacao(Rotacao r) {
    rot = r;
}

Rotacao Grupo::getRotacao() {
    return rot;
}

string Grupo::getOrdem() {
    return ordem;
}
```

```
void Grupo::setOrdem(string o) {
    ordem = o;
}

vector<Modelo> Grupo::getModelos() {
    return modelos;
}
void Grupo::setModelos(vector<Modelo> m) {
    modelos = m;
}

void Grupo::addModelo(Modelo m) {
    modelos.push_back(m);
}

vector<Grupo> Grupo::getFilhos() {
    return filhos;
}
void Grupo::setFilhos(vector<Grupo> lf) {
    filhos = lf;
}
void Grupo::addFilho(Grupo g) {
    filhos.push_back(g);
}
```

Figura 7 - Classe Grupo

3.2 Processo de leitura

3.2.1 ReadXML

Para ser possível a leitura do ficheiro xml usamos, como na fase 1, a livraria tinyxml2. Começamos com a função **readXML(String file)** a qual tem como argumento um nome de um ficheiro xml.

Após ser feito o carregamento avançamos até ao primeiro elemento “group” e chamamos a função **readGrupo(Grupo* grupo, XMLElement* elementoXml)** com esse elemento xml e com um novo objeto grupo passado por referência, sendo esse objeto modificado por essa função e depois adicionado à lista de grupos **(vector <Grupo> gruposLista)** que está globalmente definida.
Os grupos principais do xml serão lidos por esta mesma função no ciclo while, ao contrário dos filhos que serão chamados recursivamente na função readGrupo abordada posteriormente.

A leitura das informações da camera mantem-se inalterado, pois o elemento xml referente a essa parte mantem-se igual nesta fase 2.

```
void readXML(string file) {
    XMLDocument xml;
    XMLDocument xmltv;
    string s;
    if (!(xml.LoadFile(("C:\\Users\\Miguel\\Desktop\\CGF1\\xml\\" + file).c_str())) &&
        cout << "Ficheiro lido com sucesso" << endl;

        XMLElement* elemento = xml.FirstChildElement("world")->FirstChildElement("group
        while (elemento != nullptr) {                   //avança até ser null
            Grupo g = *new Grupo();
            readGrupo(&g, elemento);
            gruposLista.push_back(g);
            elemento = elemento->NextSiblingElement();     //avança para o proximo
        }

        //Camara
        XMLElement* tv = xmltv.FirstChildElement("world")->FirstChildElement("camera");
        XMLElement* tv2 = tv->FirstChildElement("position");
        XMLElement* tv3 = tv->FirstChildElement("lookAt");
        XMLElement* tv4 = tv->FirstChildElement("up");
        XMLElement* tv5 = tv->FirstChildElement("projection");

        camX = atof(tv2->Attribute("x"));
        camY = atof(tv2->Attribute("y"));
        camZ = atof(tv2->Attribute("z"));

        xInicial = camX;
        yInicial = camY;

        lookX = atof(tv3->Attribute("x"));
        lookY = atof(tv3->Attribute("y"));
        lookZ = atof(tv3->Attribute("z"));

        upX = atof(tv4->Attribute("x"));
        upY = atof(tv4->Attribute("y"));
        upZ = atof(tv4->Attribute("z"));

        fov = atof(tv5->Attribute("fov"));
        near = atof(tv5->Attribute("near"));
        far = atof(tv5->Attribute("far"));
```

Figura 8 - Função readXML

3.2.1 ReadGrupo

A função **readGrupo(Grupo* grupo, XMLElement* elementoXml)** lê as propriedades de um elementoXml de um grupo e guarda todas as informações no objeto grupo, também passado por parâmetro.

Começa por fazer a leitura das transformações (Translação, Rotação e Escala) desse mesmo grupo, criando um objeto do tipo da transformação e associando ao grupo. Guardamos ainda a linha onde está essa mesma transformação, pois posteriormente serão comparadas para ser possível saber a ordem das mesmas.

```
void readGrupo(Grupo* grupo, XMLElement* elementoXml) {


    int linhaTrans = -1, linhaRot = -1, linhaScale = -1;
    XMLElement* elementoAux = elementoXml->FirstChildElement("transform");

    XMLElement* translacaoElemento = elementoAux->FirstChildElement("translate");
    if (translacaoElemento != nullptr) {

        cout << "translate" << endl;
        linhaTrans = translacaoElemento->GetLineNum();

        float x = 0, y = 0, z = 0;
        if (translacaoElemento->Attribute("x") != nullptr) {
            x = stof(translacaoElemento->Attribute("x"));
        }
        if (translacaoElemento->Attribute("y") != nullptr) {
            y = stof(translacaoElemento->Attribute("y"));
        }
        if (translacaoElemento->Attribute("z") != nullptr) {
            z = stof(translacaoElemento->Attribute("z"));
        }
        cout << x << endl;
        cout << y << endl;
        cout << z << endl;
        Translacao t = *new Translacao(x, y, z);
        (*grupo).setTranslacao(t);
    }
```

Figura 9 - Função readGrupo translações

```
XMLElement* rotacaoElemento = elementoAux->FirstChildElement("rotate");
if (rotacaoElemento != nullptr) {


    linhaRot = rotacaoElemento->GetLineNum();

    float angulo = 0, x = 0, y = 0, z = 0;
    if (rotacaoElemento->Attribute("angle") != nullptr) {
        angulo = stof(rotacaoElemento->Attribute("angle"));
    }
    if (rotacaoElemento->Attribute("x") != nullptr) {
        x = stof(rotacaoElemento->Attribute("x"));
    }
    if (rotacaoElemento->Attribute("y") != nullptr) {
        y = stof(rotacaoElemento->Attribute("y"));
    }
    if (rotacaoElemento->Attribute("z") != nullptr) {
        z = stof(rotacaoElemento->Attribute("z"));
    }
    Rotacao r = *new Rotacao(x, y, z, angulo);
    (*grupo).setRotacao(r);

}
```

Figura 10 - Função readGrupo rotações

```
XMLElement* escaloElemento = elementoAux->FirstChildElement("scale");
if (escaloElemento != nullptr) {

    linhaScale = escaloElemento->GetLineNum();

    float x = 0, y = 0, z = 0;
    if (escaloElemento->Attribute("x") != nullptr) {
        x = stof(escaloElemento->Attribute("x"));
    }
    if (escaloElemento->Attribute("y") != nullptr) {
        y = stof(escaloElemento->Attribute("y"));
    }
    if (escaloElemento->Attribute("z") != nullptr) {
        z = stof(escaloElemento->Attribute("z"));
    }
    Escala e = *new Escala(x, y, z);
    (*grupo).setEscala(e);
```

Figura 11 - Função readGrupo escalas

Após termos as transformações guardadas vamos então ler os modelos através de um ciclo while até o modelo ser null, avançando sempre para o próximo modelo em cada iteração.

Criamos assim em cada iteração um novo objeto modelo, guardando os pontos e usando a mesma função readFile, que já tinha sido usada na fase 1, mas desta vez em vez da função guardar os pontos numa lista de pontos global, a lista de pontos lida é dada como return e associada ao modelo.

```
XMLElement* modelosXML = elementoXml->FirstChildElement("models");

if (modelosXML != nullptr) {
    XMLElement* modeloAtualXML = modelosXML->FirstChildElement("model");

    while (modeloAtualXML != nullptr) {

        Modelo modelAtual = *new Modelo();

        if (strcmp(modeloAtualXML->Attribute("file"), "sphere.3d") == 0) {
            cout << "Encontrei sphere" << endl;
            modelAtual.setPontos(readFile("C:\\Users\\Miguel\\Desktop\\CGF1\\3dFiles\\sphere.3d"));
            (*grupo).addModelo(modelAtual);
        }
        if (strcmp(modeloAtualXML->Attribute("file"), "cone.3d") == 0) {
            cout << "Encontrei cone" << endl;
            modelAtual.setPontos(readFile("C:\\Users\\Miguel\\Desktop\\CGF1\\3dFiles\\cone.3d"));
            (*grupo).addModelo(modelAtual);
        }

        if (strcmp(modeloAtualXML->Attribute("file"), "plane.3d") == 0) {
            cout << "Encontrei plane" << endl;
            modelAtual.setPontos(readFile("C:\\Users\\Miguel\\Desktop\\CGF1\\3dFiles\\plane.3d"));
            (*grupo).addModelo(modelAtual);

        }
        if (strcmp(modeloAtualXML->Attribute("file"), "box.3d") == 0) {
            cout << "Encontrei box" << endl;
            modelAtual.setPontos(readFile("C:\\Users\\Miguel\\Desktop\\CGF1\\3dFiles\\box.3d"));
            (*grupo).addModelo(modelAtual);
        }
        modeloAtualXML = modeloAtualXML->NextSiblingElement();
    }
}
```

Figura 12 - Função readGrupo modelos

Após lermos os modelos vamos passar a ver a ordem das transformações, para isso usamos uma série de comparações das linhas em que se encontram as mesmas e guardamos essa ordem no objeto grupo.

```
string o = "";

if (linhaTrans <= linhaRot && linhaTrans <= linhaScale) {
    if (linhaRot <= linhaScale) {
        o = "T R E";
    }
    else {
        o = "T E R";
    }
}

if (linhaRot <= linhaTrans && linhaRot <= linhaScale) {
    if (linhaTrans <= linhaScale) {
        o = "R T E";
    }
    else {
        o = "R E T";
    }
}

if (linhaScale <= linhaTrans && linhaScale <= linhaRot) {
    if (linhaRot <= linhaTrans) {
        o = "E R T";
    }
    else {
        o = "E T R";
    }
}

(*grupo).setOrdem(o);
```

Figura 13 - Função readGrupo ordem transformações

Por fim, a função readGrupo é chamada recursivamente para cada filho desse mesmo grupo criando um novo objeto Grupo para cada filho e adicionando à lista de filhos do grupo.

```
XMLElement* filhos = elementoXml->FirstChildElement("group");

while (filhos != nullptr) {

    Grupo filho = *new Grupo();
    readGrupo(&filho, filhos);
    (*grupo).addFilho(filho);

    filhos = filhos->NextSiblingElement();
}
```

Figura 14 - Função readGrupo recursiva

3.3 Processo de desenho

Para desenharmos começamos com a função **drawGrupos()**, que itera o vetor global, o qual tem alocado os grupos principais a ser desenhados, e cada grupo será individualmente dado como argumento para a função **drawGrupo(Grupo g)**.

```
void drawGrupos() {
    for (int i = 0; i < gruposLista.size(); i++) {
        drawGrupo(gruposLista[i]);
    }
}
```

Figura 15 - Função drawGrupos

A função **drawGrupo(Grupo g)** recebe como argumento o grupo a ser desenhado. Começamos por obter a informação alocada do grupo anterior como argumento com a utilização dos getters definidos previamente, informação que consiste nas transformações geométricas, o vetor dos modelos do grupo, o vetor dos subgrupos e a string que contém a ordem com que as transformações são executadas.

```
void drawGrupo(Grupo gR) {
    Translacao t = gR.getTranslacao();
    Rotacao r = gR.getRotacao();
    Escala e = gR.getEscala();
    string ordem = gR.getOrdem();
    vector<Grupo> filhos = gR.getFilhos();

    vector<Modelo> m = gR.getModelos();
```

Figura 16 - Função drawGrupo

Fazemos de seguida um **glPushMatrix()**, para depois ser possível fazer **glPopMatrix()** e evitarmos que as transformações efetuadas previamente afetem desenhos que não pretendemos que sejam afetados pelas mesmas.

Entre o push e o pop executamos as transformações pela ordem da string que obtivemos previamente, depois desenhamos todos os modelos desse grupo individualmente com a função **drawPontos(list listaPontos)** que através da lista de pontos vai percorrer a mesma usando um iterador, a cada 3 pontos desenha um triangulo desta forma. A cada iteração do ciclo for, 3 pontos e um triangulo são desenhados avançando assim o iterador em 3 posições.

```
void drawPontos(list<Ponto> pontosLista) {

    for (auto it = pontosLista.begin(); it != pontosLista.end(); ) {
        glBegin(GL_TRIANGLES);
        glVertex3f(it->getX(), it->getY(), it->getZ());
        ++it;
        glVertex3f(it->getX(), it->getY(), it->getZ());
        ++it;
        glVertex3f(it->getX(), it->getY(), it->getZ());
        ++it;
        glEnd();
    }
```

Figura 17 - Função drawPontos

De seguida vamos ao vetor dos filhos (subgrupos) e aplicamos a função drawGrupo antes do pop, de forma a que as transformações do grupo afetem os seus subgrupos.

```
glPushMatrix();

for (int i = 0; i < 5; i += 2) {
    switch (ordem[i]) {
    case 'T':
        glTranslatef(t.getX(), t.getY(), t.getZ());
        break;
    case 'R':
        glRotatef(r.getAngulo(), r.getX(), r.getY(), r.getZ());
        break;
    case 'E':
        glScalef(e.getX(), e.getY(), e.getZ());
        break;
    default:
        break;
    }
}


for (int i = 0; i < m.size(); i++) {
    glColor3f(v, g, b);
    drawPontos(m[i].getPontos());
}

for (int i = 0; i < filhos.size(); i++) {
    drawGrupo(filhos[i]);
}

glPopMatrix();
```

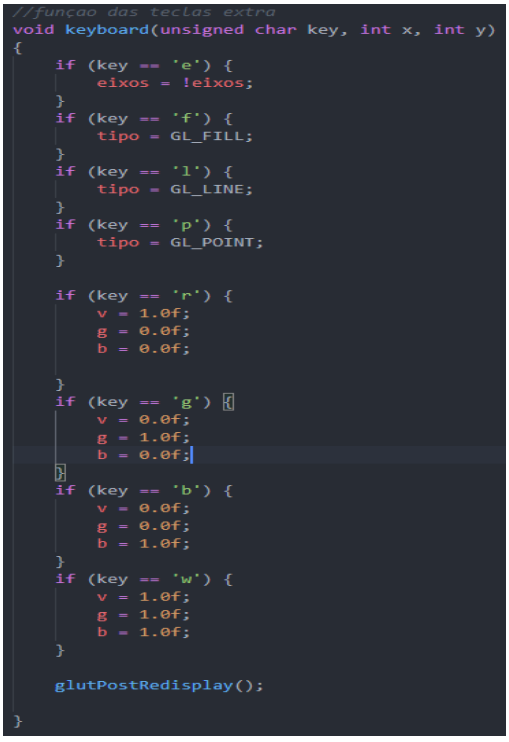
Figura 18 - Função drawGrupo II

4. Extras

Tal como na primeira fase, continuamos a ter os mesmos extras.

3.1 Função Keyboard

- **e**: Adiciona/Remove os eixos XYZ.
- **f**: Modo GL_FILL.
- **l**: Modo GL_LINE.
- **p**: Modo GL_POINT.
- **r**: Muda cor do polígono para vermelho.
- **g**: Muda cor do polígono para verde.
- **b**: Muda cor do polígono para azul.
- **w**: Muda cor do polígono para branco.



```
//funçao das teclas extra
void keyboard(unsigned char key, int x, int y)
{
    if (key == 'e') {
        eixos = !eixos;
    }
    if (key == 'f') {
        tipo = GL_FILL;
    }
    if (key == 'l') {
        tipo = GL_LINE;
    }
    if (key == 'p') {
        tipo = GL_POINT;
    }

    if (key == 'r') {
        r = 1.0f;
        g = 0.0f;
        b = 0.0f;

    }
    if (key == 'g') {
        r = 0.0f;
        g = 1.0f;
        b = 0.0f;
    }
    if (key == 'b') {
        r = 0.0f;
        g = 0.0f;
        b = 1.0f;
    }
    if (key == 'w') {
        r = 1.0f;
        g = 1.0f;
        b = 1.0f;
    }

    glutPostRedisplay();

}
```

Figura 19 - Função Keyboard

3.2 Câmera

As funções da câmera **processMouseButtons e processMouseMotion** mantem-se iguais à primeira fase. Já foi referido no relatório da fase 1 o seu funcionamento.

5. Testes

5.1 Teste 2_1

Translate x=0 y=0 z=0

Draw box

Posição da camera x=10 y=3 z=10

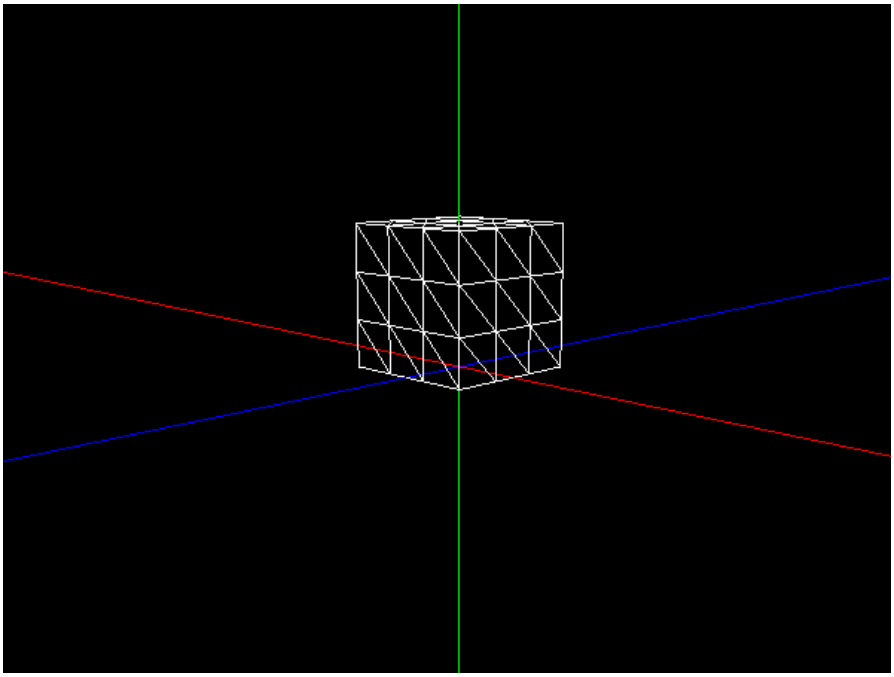


Figura 20 - Test_2_1.xml

5.2 Teste 2_2

Translate x=0 y=0 z=0

Draw box

Translate x=0 y=2 z=0

Draw cone

Translate x=0 y=3 z=0

Draw sphere

Posição da camera x=5 y=6 z=15

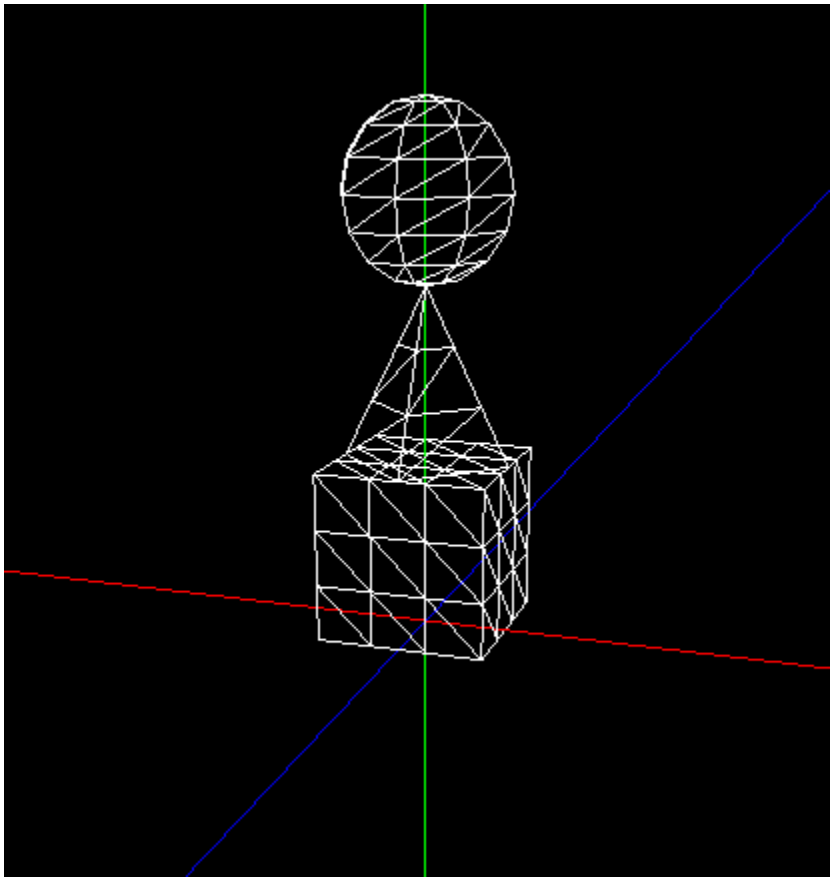


Figura 21 - Test_2_2.xml

5.3 Teste 2_3

Translate x=0 y=0.75 z=0

Scale x=0.75 y=0.75 z=0.75

Draw sphere

Translate x=0 y=1 z=0

Scale x=0.25 y=0.25 z=0.25

Draw sphere

Rotate alpha=90 x=1 y=0 z=0

Scale x=0.1 y=0.3 z=0.1

Draw cone

Posição da camera x=3 y=2 z=3

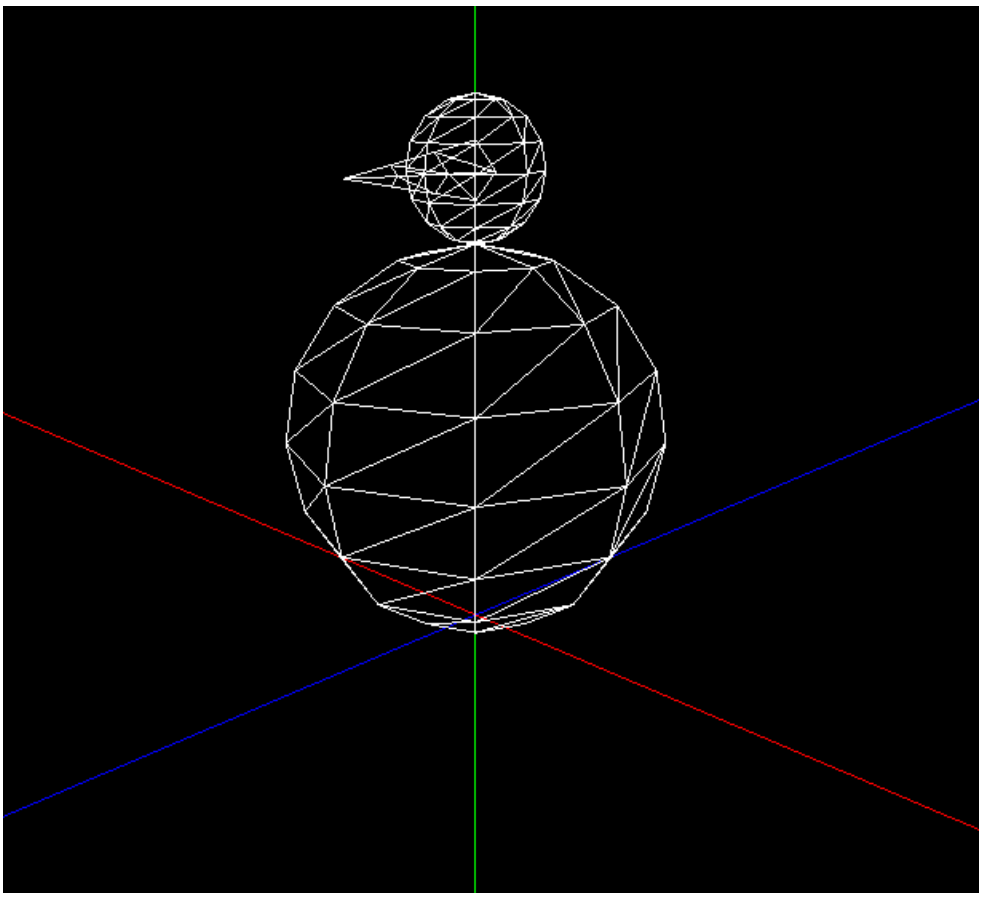


Figura 22 - Test_2_3.xml

5.4 Teste 2_4

Translate x=0 y=1 z=0

Rotate alpha=45 x=0 y=1 z=0

Translate x=3.5 y=0 z=0

Draw Box

Rotate alpha=90 x=0 y=1 z=0

Translate x=3.5 y=0 z=0

Draw Box

Rotate alpha=135 x=0 y=1 z=0

Translate x=3.5 y=0 z=0

Draw Box

Rotate alpha=180 x=0 y=1 z=0

Translate x=3.5 y=0 z=0

Draw Box

Rotate alpha=225 x=0 y=1 z=0

Translate x=3.5 y=0 z=0

Draw Box

Rotate alpha=270 x=0 y=1 z=0

Translate x=3.5 y=0 z=0

Draw Box

Rotate alpha=315 x=0 y=1 z=0

Translate x=3.5 y=0 z=0

Draw Box

Posição da camera x=3 y=10 z=4

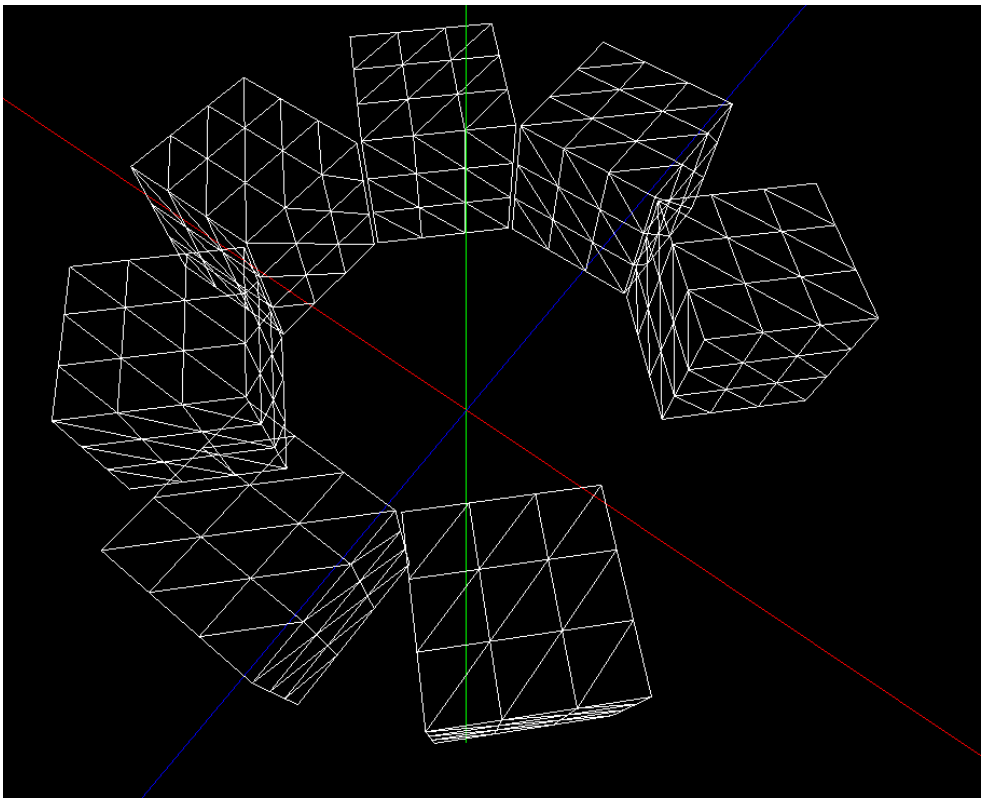


Figura 23 - Test_2_4.xml

6. Sistema Solar

Como é pedido no enunciado do trabalho prático, o grupo preparou dois ficheiros xml para representar um sistema solar, um em que os planetas estao em linha (demolinha.xml) e o segundo onde os planetas aparecem distanciados (demo.xml).

De forma a tentar obter uma escala e distância o mais perto possivel da realidade, o grupo baseou-se nas seguintes informações:

Distância Real (Milhões km)	**Escala**
50	2
y	x

Daqui resulta a fómula, usando a regra de 3 simples, usada para calcular a escala dos quatro primeiros planetas, tendo por base a distância em relação ao centro do sistema solar:

$$x = \frac{2 * y}{50}$$

Onde y é a distãncia real, retirada do Google, e o x a distãncia utilizada pelo grupo.

No que toca ao diâmetro dos planetas, foi utilizado o valor base 0.05, que corresponde ao diâmetro do planeta mercúrio estabelecido pelo grupo.

Tamanho real (km)	**Escala**
4880	0.05
z	w

Onde obtemos o seguinte resultado:

$$w = \frac{0.05 * z}{4880}$$

Onde w é a escala utilizada pelo grupo e z o diâmetro real do planeta, retidado do Google.

Para os quatro últimos planetas, o grupo teve por base o planeta júpiter, onde foi estabelecido uma escala de diâmetro de de 0.6.

Tamanho real (km)	**Escala**
143000	0.06
z	w

Assim obtemos:

$$w = \frac{0.6 * z}{143000}$$

Onde w é a escala do planeta e z o diâmetro real.

Para os satélites naturais dos planetes, o seu diâmetro é calculado pela seguinte fórmula:

$$x = \frac{d_{satelite}}{d_{planeta}}$$

6.1 Desenho do sistema solar

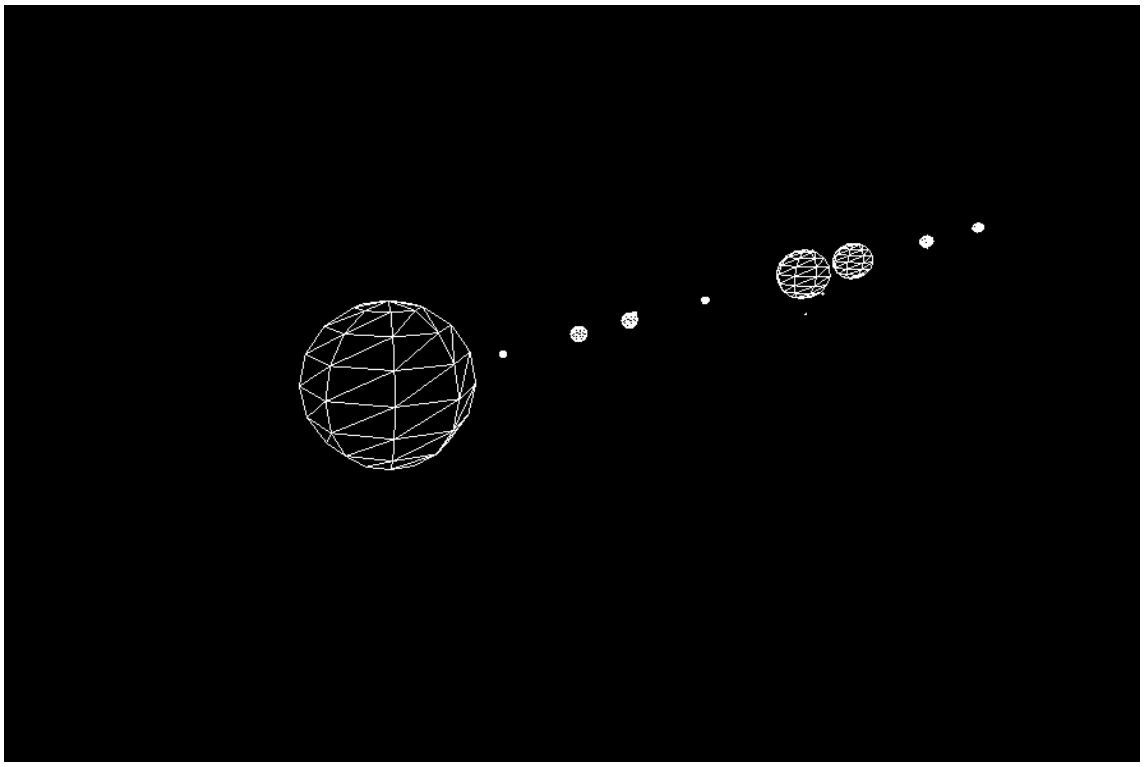


Figura 24 - demolinha.xml

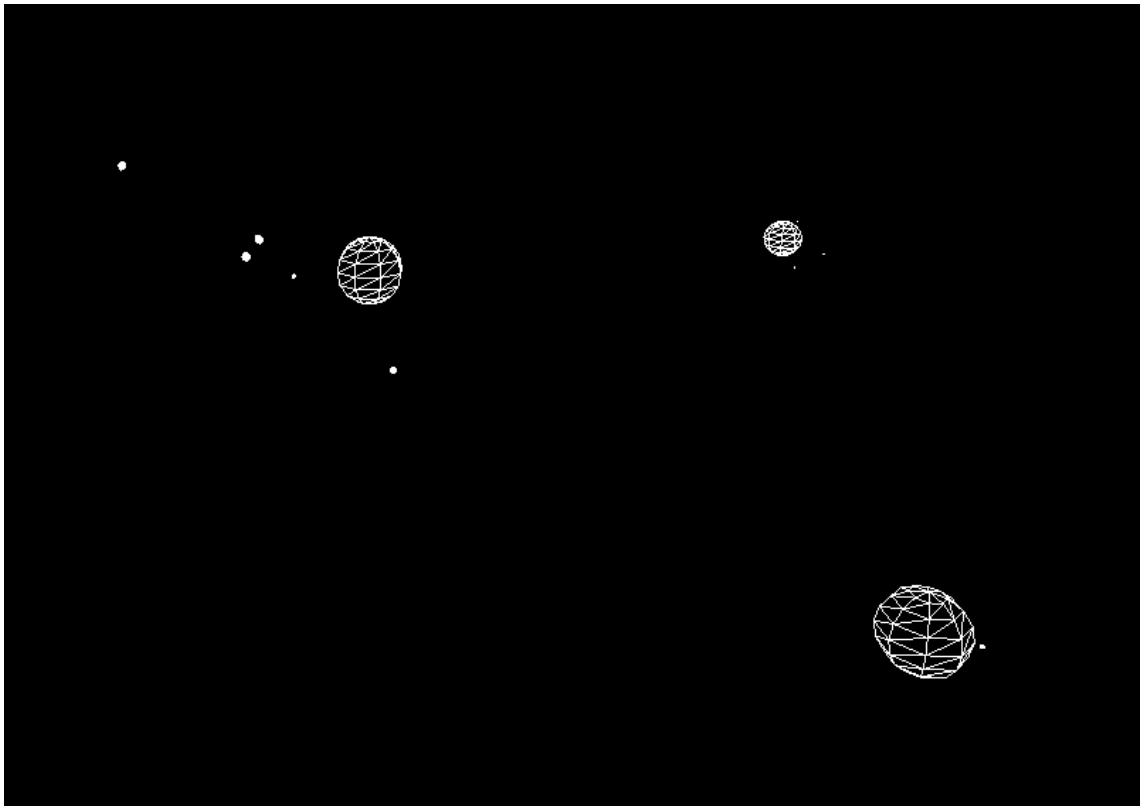


Figura 25 - demoxml

7. Conclusão

Para o desenvolvimento desta segunda fase de entrega, embora não tenha sido tão trabalhosa como a primeira, foi necessário mais conhecimento teórico acerca da disciplina, como por exemplo acerca da hierarquia das transformações geométricas.

Serviu como consolidação dessa parte teórica aprendida nas últimas semanas, dando ainda a oportunidade de colocar a mesma em prática.

Em suma, permitiu aplicar tudo o que foi aprendido até ao momento. Esperamos com ânsia pelas próximas entregas, de forma a termos mais oportunidades de pôr em prática toda a teoria dada.