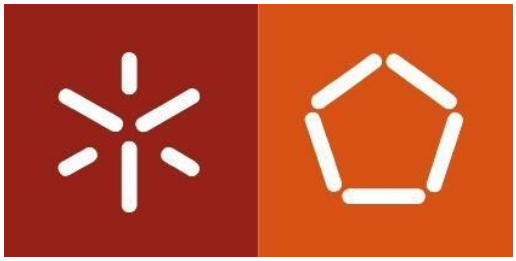
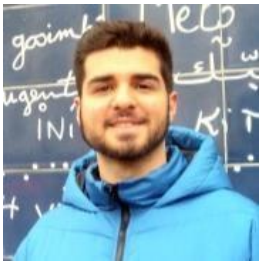


Computação Gráfica - CG 2019/2020

FASE 4

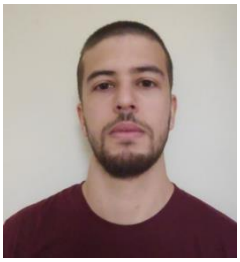
Mestrado Integrado em Engenharia Informática

José Pinto
A84590



Eduardo Costa
A85735

Luís Lopes
A85367



Ricardo Carvalho
A84261

Índice

Introdução 2

Alterações no *generator* 3

Plane *3*

Box *3*

Esfera *4*

Alterações das estruturas de dados 5

Light *5*

lightVector *5*

Color *5*

OperFile *6*

Alterações do *parser XML* 7

Novas funcionalidades da *Engine* 9

Aplicação de textura *9*

Aplicação de luz *11*

Aplicação de cor *12*

Sistema Solar, com textura, luz e cor 13

Conclusão 15

Introdução

No âmbito da Unidade Curricular de Computação Gráfica, foi-nos proposto o desenvolvimento de um mecanismo 3D baseado num cenário gráfico, que já vinha sendo desenvolvido nos três trabalhos práticos anteriores.

Desta forma, nesta quarta e última fase deste projeto pretendia-se, no que à *engine* diz respeito, a possibilidade de aplicação de cor, textura e luz. Quanto ao *generator*, o objetivo para esta fase focou-se no desenvolvimento de uma nova funcionalidade: era pretendido que este fosse capaz de gerar coordenadas de textura para as respetivas figuras geométricas.

Para tal, foram necessárias diversas alterações nas diferentes vertentes do trabalho em relação ao que havia sido desenvolvido na fase anterior, começando, desde logo, pelo *generator* e passando, depois, para as estruturas de dados, *parser* XML e funções de desenhar da *engine*. Para além disto e de tudo o que havia sido implementado até aqui, foram também aplicadas as novas funcionalidades à nossa representação do sistema solar. Todas estas alterações encontram-se, portanto, descritas no presente relatório.

Alterações no *generator*

Para esta fase do trabalho era pretendido que o programa fosse capaz de gerar, para além dos pontos referentes às formas geométricas, as respetivas coordenadas de textura dos mesmos pontos.

Plane

No caso do plano, para a obtenção das normais foi necessário verificar qual o plano desenhado. No nosso caso, desenhamos o plano y=0, logo o vetor normal em todos os pontos é (0,1,0). Repetindo o raciocínio para a face inferior, obtém-se a normal (0,-1,0). As coordenadas relativas às texturas são obtidas através da correspondência direta a cada vértice.

Box

A estratégia de obtenção das normais para a *box* é similar à do plano, no entanto, para esta forma geométrica, o processo é repetido 6 vezes, uma para cada uma das faces. As normais de cada face são as seguintes:

- Face frontal (0,0,1);
- Face direita (1,0,0);
- Face esquerda (-1,0,0);
- Face superior (0,1,0);
- Base (0,-1,0);
- Face traseira (0,0,-1);

De seguida apresenta-se uma imagem sobre o modelo que seguimos para o cálculo das coordenadas de textura, para este caso.

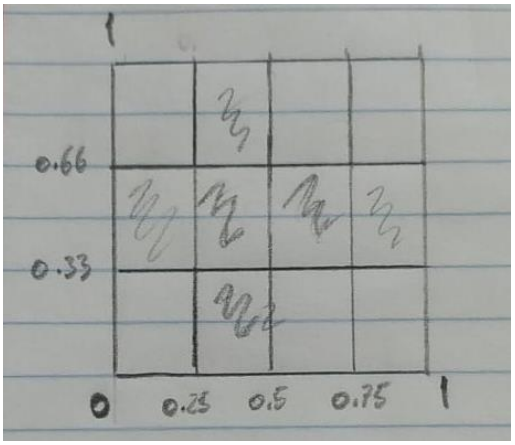


Figura 1 – Coordenadas de textura box

Esfera

Quanto às esferas, para o cálculo das normais, voltamos a utilizar os ângulos calculados para o desenho dos vértices, tendo em conta, desta feita, a orientação da origem do referencial até ao vértice em questão. Isto foi possível, uma vez que a nossa estratégia para gerar o ficheiro .3d passa por escrever, para cada ponto de um triângulo, as suas coordenadas, vetor normal e coordenadas de textura.

```
outFile << p1x << " " << p1y << " " << p1z << "\n";
outFile << p2x << " " << p2y << " " << p2z << "\n";
outFile << p4x << " " << p4y << " " << p4z << "\n";
outFile << "" << radius * sin(angle3)*sin(angle1) << " " << radius * cos(angle3) << " " << radius * cos(angle1) * sin(angle3) << "\n";
outFile << "" << radius * sin(angle3 + angle4) * sin(angle1) << " " << radius * cos(angle3 + angle4) << " " << radius * sin(angle3+angle4) * cos(angle1) << "\n";
outFile << "" << radius * sin(angle3 + angle4) * sin(angle1+angle2) << " " << radius * cos(angle3 + angle4) << " " << radius * sin(angle3 + angle4) * cos(angle1+angle2) << "\n";
outFile << s_factor << " " << t_factor << " 0\n";
outFile << s_factor << " " << t_factor - (1.d/stacks) << " 0\n";
outFile << s_factor + (1.d/slices) << " " << t_factor-(1.d/stacks) << " 0\n";
```

Figura 2 – Método de escrita para o ficheiro .3d

O processo de cálculo das coordenadas de textura passou por calcular o intervalo no qual aplicamos a textura. Obtemos este intervalo dividindo 1, que é o valor máximo que as texturas podem tomar, pelo número de *slices,* para calcular o valor de x e dividindo 1 pelo número de *stacks* para calcular o valor de y.

Alterações das estruturas de dados

Em relação às alterações que seriam necessárias realizar na *engine*, a nossa abordagem começou, tal como nas fases anteriores, pelas alterações às estruturas de dados e, simultaneamente, ao *parser* XML, que serão abordadas na secção seguinte. Estas alterações foram bastante simples e visaram a implementação da possibilidade de, agora, se poder aplicar textura, luz e cor aos cenários desenhados. Para tal, criamos duas novas estruturas de dados e aplicamo-las às previamente criadas.

Light

Necessitamos de criar, nesta fase, uma estrutura que armazenasse todas as informações acerca de cada luz a aplicar. Os ficheiros XML podem, agora, conter uma secção de luzes, no início do mesmo, tal como exemplificado no enunciado do projeto, com informação referente ao tipo de luz ("POINT", "DIRECTIONAL" e "SPOT") e às coordenadas da mesma.

```
typedef struct light {
    char* type;
    double x, y, z;
} Light;
```

Figura 3 – Nova estrutura Light

lightVector

Para armazenar as luzes criamos, então, um vetor de *Lights* (estrutura acima), com todas as luzes a aplicar no cenário, sendo este passado como argumento ao *parser XML*.

```
vector<Light*> lightVector;
```

Figura 4 – Vector lightVector

Color

Criamos, também, uma estrutura semelhante à referida acima, mas, desta feita, respetiva às cores. A cada cor estará associada a sua componente ("*diffuse*", "*specular*", "*emissive*" e "*ambient*") e os valores de R, G e B (RGB), para que seja possível aplicá-los posteriormente.

```
typedef struct color {
    char* component;
    double r, g, b;
} Color;
```

Figura 5 – Nova estrutura Color

OperFile

Para que cada ficheiro *.3d* tenha uma textura e uma cor associadas, alteramos a estrutura *OperFile* para que contenha, também, informação relativa a estes atributos, através do nome do ficheiro de textura e da estrutura *Color* descrita acima. Estas são sempre inicializadas como nulas, para o caso de o ficheiro não ter algum destes atributos.

```
typedef struct operFile {
    char* fileName;
    vector<Oper*> operations;
    char* texture;
    Color* color;
} OperFile;
```

Figura 6 – Alterações à estrutura OperFile

Alterações do *parser XML*

Juntamente com as alterações feitas às estruturas, adaptamos o nosso *parser* àquilo que era pedido no enunciado, à possibilidade do ficheiro XML conter informações relativas a luz, texturas e cores.. Assim sendo, na leitura/*parsing* do ficheiro XML recebido como argumento, tivemos que considerar, agora, a secção inicial de luzes, criando, para o efeito, um *parser* específico para este caso, que coloca cada *Light* no respetivo *vector* descrito acima.

```
void parseLights(const XMLElement* child, vector<Light*>& lightVector)
{
    for (const XMLElement* child2 = child->FirstChildElement(); child2; child2 = child2->NextSiblingElement()) {

        double x = 0, y = 0, z = 0;

        Light* lighting = (Light*)malloc(sizeof(struct light));
        lighting->type = (char*)malloc(sizeof(char) * 15);

        char* type = (char*)child2->Attribute("type");
        char* posX = (char*)child2->Attribute("posX");
        if (posX != NULL) x = atof(posX);
        char* posY = (char*)child2->Attribute("posY");
        if (posY != NULL) y = atof(posY);
        char* posZ = (char*)child2->Attribute("posZ");
        if (posZ != NULL) z = atof(posZ);

        strcpy(lighting->type, type);
        lighting->x = x;
        lighting->y = y;
        lighting->z = z;
        lightVector.push_back(lighting);
    }
}
```

Figura 7 – Parser XML de luzes

Tendo também em conta a possibilidade de cada ficheiro poder ter a si associadas uma cor ou uma textura, procedemos a alterações nesse sentido. Desta forma, aquando do *parsing* de cada secção *models*, inicializamos cada um destes atributos como nulos e verificamos, para cada caso, a existência dos mesmos no ficheiro XML, como podemos ver na figura abaixo, para o caso das texturas.

```
// textured model
char* texture = NULL;
if (child3->Attribute("texture")) {
    texture = (char*)child3->Attribute("texture");
}

// coloured model
Color* color = NULL;
char* r;
char* g;
char* b;
```

Figura 8 – Inicialização dos atributos de textura e cor

Para o caso das cores, a componente das mesmas é definida pelo prefixo colocado em cada um dos parâmetros RGB, como exemplificado no enunciado. Deste modo, consideramos os prefixos de 4 letras “diff”, “spec”, “emis” e “ambi” para representar as componentes difusa, especular, emissiva e ambiente, respetivamente. Sendo assim, o *parser*, na presença de um destes prefixos associados ao atributo R (por ser o primeiro a surgir), define a componente da cor com o respetivo nome na estrutura mostrada acima, e extrai cada um dos atributos RGB para a mesma.

```
if (child3->Attribute("diffR")) {
    color = (Color*)malloc(sizeof(struct color));
    color->component = (char*)malloc(sizeof(char) * 12);

    strcpy(color->component, "diffuse");

    r = (char*)child3->Attribute("diffR");
    color->r = atof(r);
    g = (char*)child3->Attribute("diffG");
    color->g = atof(g);
    b = (char*)child3->Attribute("diffB");
    color->b = atof(b);
}
else if (child3->Attribute("specR")) {
    color = (Color*)malloc(sizeof(struct color));
    color->component = (char*)malloc(sizeof(char) * 12);

    strcpy(color->component, "specular");

    r = (char*)child3->Attribute("specR");
    color->r = atof(r);
    g = (char*)child3->Attribute("specG");
    color->g = atof(g);
    b = (char*)child3->Attribute("specB");
    color->b = atof(b);
}
else if (child3->Attribute("emisR")) {
    color = (Color*)malloc(sizeof(struct color));
    color->component = (char*)malloc(sizeof(char) * 12);

    strcpy(color->component, "emissive");

    r = (char*)child3->Attribute("emisR");
    color->r = atof(r);
    g = (char*)child3->Attribute("emisG");
    color->g = atof(g);
    b = (char*)child3->Attribute("emisB");
    color->b = atof(b);
}
else if (child3->Attribute("ambiR")) {
    color = (Color*)malloc(sizeof(struct color));
    color->component = (char*)malloc(sizeof(char) * 12);

    strcpy(color->component, "ambient");

    r = (char*)child3->Attribute("ambiR");
    color->r = atof(r);
    g = (char*)child3->Attribute("ambiG");
    color->g = atof(g);
    b = (char*)child3->Attribute("ambiB");
    color->b = atof(b);
}
```

Figura 9 – Parser XML de cores

Novas funcionalidades da *Engine*

Como já foi referido até aqui, todas as alterações feitas ao *parser* e às estruturas tiveram como propósito possibilitar a aplicação de textura, cor e luz aos cenários produzidos. Desta forma, procedemos a algumas implementações nesse sentido, desde a criação de funções de processamento da luz, cor e textura, por exemplo, até a alterações à função de criação de VBOs, que passou a construir mais dois VBOs, um para guardar as normais e outro para guardar as coordenadas de textura.

```
glGenBuffers(files.size(), buffers);
glGenBuffers(files.size(), normals);
glGenBuffers(files.size(), texts);

glBindBuffer(GL_ARRAY_BUFFER, buffers[i]);
glBufferData(GL_ARRAY_BUFFER, 8 * vertexCount[i] * 3, vertexB[i], GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, normals[i]);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * vertexCount[i] * 3, normais[i], GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, texts[i]);
glBufferData(GL_ARRAY_BUFFER, 8 * vertexCount[i] * 2, textures[i], GL_DYNAMIC_DRAW);
```

Figura 10 – Alterações na função createVBO

Aplicação de textura

Para a aplicação de textura, após feita a leitura do ficheiro XML e respetiva associação entre o ficheiro de textura (colocado na diretoria “texturas” do projeto) lá indicado e o ficheiro .3d na estrutura *OperFile* descrita anteriormente, realizamos uma atribuição de um ID a cada uma delas através de um *array*.

```
void initTexturesByID() {

    int i = 0;
    texturesByID = (GLuint*)malloc(sizeof(GLuint) * files.size());
    for (int i = 0; i < files.size(); i++) texturesByID[i] = NULL;
    vector<OperFile*>::iterator it;
    for (it = files.begin(); it != files.end(); i++, it++) {
        OperFile* oper = *it;
        if (oper->texture != NULL) {
            char dir[60];//"../texturas/";
            char* textureFileName = oper->texture;
            string textureString(textureFileName);
            texturesByID[i] = loadTexture(textureString);
        }
    }
}
```

Figura 11 – Inicialização do array de IDs de texturas

Para que tal aconteça, e com recurso à livraria *DevIL,* construímos uma função *loadTexture*. Esta começa por carregar a textura, gerando assim um ID que, mais tarde, a identifica. Utilizando a função *glBindTexture* do *openGL*, aplicamos a textura ao desenhar o VBO do objeto correspondente.

```
int loadTexture(std::string s) {
    unsigned int t, tw, th;
    unsigned char* texData;
    unsigned int texID;

    ilInit();
    ilEnable(IL_ORIGIN_SET);
    ilOriginFunc(IL_ORIGIN_LOWER_LEFT);
    ilGenImages(1, &t);
    ilBindImage(t);
    ilLoadImage((ILstring)s.c_str());
    tw = ilGetInteger(IL_IMAGE_WIDTH);
    th = ilGetInteger(IL_IMAGE_HEIGHT);
    ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
    texData = ilGetData();

    glGenTextures(1, &texID);

    glBindTexture(GL_TEXTURE_2D, texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tw, th, 0, GL_RGBA, GL_UNSIGNED_BYTE, texData);
    glGenerateMipmap(GL_TEXTURE_2D);

    glBindTexture(GL_TEXTURE_2D, 0);
    return texID;
}
```

Figura 12 –Função loadTexture

Aplicação de luz

O processamento das luzes a aplicar no cenário passou a ser quase que direto após o preenchimento do *vector* por nós criado para o efeito, descrito acima. Para tal, quando a função *renderScene* chama a função da figura abaixo, esta, para cada *Light* presente no *vector*, verifica o seu tipo e aplica, nas devidas coordenadas, a respetiva luz.

```
void processLight() {
    luzesInt = 0;
    float position[4] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat ambi[4] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat diff[4] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat direction[3] = { 0.0, 0.0, -1.0 };


    vector<Light*>::iterator it;
    for (it = lightVector.begin(); it != lightVector.end(); it++) {

        if (luzesInt < 8) {

            Light* light = *it;

            if (strcmp(light->type, "POINT") == 0) {
                position[0] = static_cast<GLfloat>(light->x);
                position[1] = static_cast<GLfloat>(light->y);
                position[2] = static_cast<GLfloat>(light->z);
                position[3] = 1;
                glEnable(luzes[luzesInt]);
                glLightfv(luzes[luzesInt], GL_POSITION, position);
                glLightfv(luzes[luzesInt], GL_AMBIENT, ambi);
                glLightfv(luzes[luzesInt], GL_DIFFUSE, diff);
                luzesInt++;
                }

            else if (strcmp(light->type, "DIRECTIONAL") == 0) {
                position[0] = static_cast<GLfloat>(light->x);
                position[1] = static_cast<GLfloat>(light->y);
                position[2] = static_cast<GLfloat>(light->z);
                position[3] = 0;
                glEnable(luzes[luzesInt]);
                glLightfv(luzes[luzesInt], GL_POSITION, position);
                glLightfv(luzes[luzesInt], GL_AMBIENT, ambi);
                glLightfv(luzes[luzesInt], GL_DIFFUSE, diff);
                luzesInt++;
                }

            else if (strcmp(light->type, "SPOT") == 0) {
                position[0] = static_cast<GLfloat>(light->x);
                position[1] = static_cast<GLfloat>(light->y);
                position[2] = static_cast<GLfloat>(light->z);
                position[3] = 1;
                direction[0] = static_cast<GLfloat>(light->x);
                direction[1] = static_cast<GLfloat>(light->y);
                direction[2] = static_cast<GLfloat>(light->z);
                glEnable(luzes[luzesInt]);
                glLightfv(luzes[luzesInt], GL_POSITION, position);
                glLightfv(luzes[luzesInt], GL_DIFFUSE, diff);
                glLightfv(luzes[luzesInt], GL_SPOT_DIRECTION, direction);
                glLightf(luzes[luzesInt], GL_SPOT_CUTOFF, 45.0);
                glLightf(luzes[luzesInt], GL_SPOT_EXPONENT, 0.0);
                luzesInt++;
                }
        }
        else { cout << "Ultrapassado máximo de 8 luzes! \n"; break; }
    }
```

Figura 13 – Função de processamento de luz

Aplicação de cor

À semelhança da aplicação de luz descrita anteriormente, a aplicação de cor aos objetos do cenário revelou-se bastante direta através das estruturas criadas. Para tal, aquando do desenho dos objetos, caso a estrutura Color respetiva contenha informação relevante, esta é passada a uma função de processamento de cor.

```
if (op->color != NULL) {
    processColor(op->color);
}
```

Figura 14 – Chamada da função processColor

Por sua vez, a função *processColor* apenas verifica qual a componente da cor que deve ser desenhada e, para cada um dos 4 casos possíveis, atua em conformidade, através da função *glMaterialfv*, à qual são passados os devidos valores de RGBA (R, G e B presentes da estrutura e A sempre 1).

```
void processColor(Color* color)
{
    GLfloat colortype[4] = { 1.0, 1.0, 1.0, 1.0 };
    if (strcmp(color->component, "diffuse") == 0) {
        colortype[0] = static_cast<GLfloat>(color->r);
        colortype[1] = static_cast<GLfloat>(color->g);
        colortype[2] = static_cast<GLfloat>(color->b);
        colortype[3] = 1;
        glMaterialfv(GL_FRONT, GL_DIFFUSE, colortype);
    }
    if (strcmp(color->component, "specular") == 0) {
        colortype[0] = static_cast<GLfloat>(color->r);
        colortype[1] = static_cast<GLfloat>(color->g);
        colortype[2] = static_cast<GLfloat>(color->b);
        colortype[3] = 1;
        glMaterialfv(GL_FRONT, GL_SPECULAR, colortype);
    }
    if (strcmp(color->component, "emissive") == 0) {
        colortype[0] = static_cast<GLfloat>(color->r);
        colortype[1] = static_cast<GLfloat>(color->g);
        colortype[2] = static_cast<GLfloat>(color->b);
        colortype[3] = 1;
        glMaterialfv(GL_FRONT, GL_EMISSION, colortype);
    }
    if (strcmp(color->component, "ambient") == 0) {
        colortype[0] = static_cast<GLfloat>(color->r);
        colortype[1] = static_cast<GLfloat>(color->g);
        colortype[2] = static_cast<GLfloat>(color->b);
        colortype[3] = 1;
        glMaterialfv(GL_FRONT, GL_AMBIENT, colortype);
    }
}
```

Figura 15 – Função de processamento de cor

Sistema Solar, com textura, luz e cor

Nesta fase, de forma a produzirmos um sistema solar com textura, luz e cor, tivemos de proceder a algumas alterações. Primeiramente, introduzimos uma secção de luzes, tal como foi acima referido.

```
<lights>
<light type="POINT" posX="-90.0" posY="-1.0" posZ="15.0" />
<light type="POINT" posX="-90.0" posY="-1.0" posZ="-15.0" />
<light type="POINT" posX="-90.0" posY="10.0" posZ="15.0" />
<light type="POINT" posX="-90.0" posY="10.0" posZ="-15.0" />
<light type="POINT" posX="-96.0" posY="10.0" posZ="15.0" />
<light type="POINT" posX="-84.0" posY="10.0" posZ="-15.0" />
<light type="POINT" posX="-96.0" posY="-1.0" posZ="15.0" />
  </lights>
```

Figura 16 – Secção de luzes do nosso ficheiro XML do sistema solar

Para aplicação de texturas apenas é necessário associar o nome de um ficheiro desse tipo a um ficheiro .3d, da forma que se apresenta abaixo, respeitando as indicações das diretivas do trabalho.

```
<model file="sphere.3d"  texture="sol.jpg" />
```

Figura 17 – Referenciação de ficheiros de textura no sistema solar

Por último, para demonstrar a nossa aplicação de cores, utilizamo-la na representação de alguns planetas. De forma semelhante à das texturas, respeitando, também, o enunciado do trabalho, indicamos os valores de RGB pretendidos, juntamente com os devidos prefixos.

```
<model file = "sphere.3d" diffR="0.8" diffG="0.8" diffB="0.15" />
```

Figura 17 – Aplicação de cor no ficheiro XML do nosso sistema solar

Finalmente, como conclusão de todo o trabalho realizado, podemos verificar que conseguimos construir um sistema solar completo, tal como na fase anterior, com o Sol, todos os planetas e alguns dos principais satélites naturais mais importantes representados, assim como um cometa, mas, desta feita, para além do que já havia sido demonstrado, com cor, textura e luz. Como é óbvio, não é possível verificar a presença de movimento na imagem abaixo, mas, para obter uma representação mais realista do sistema solar, optamos por retirar o desenho das órbitas.

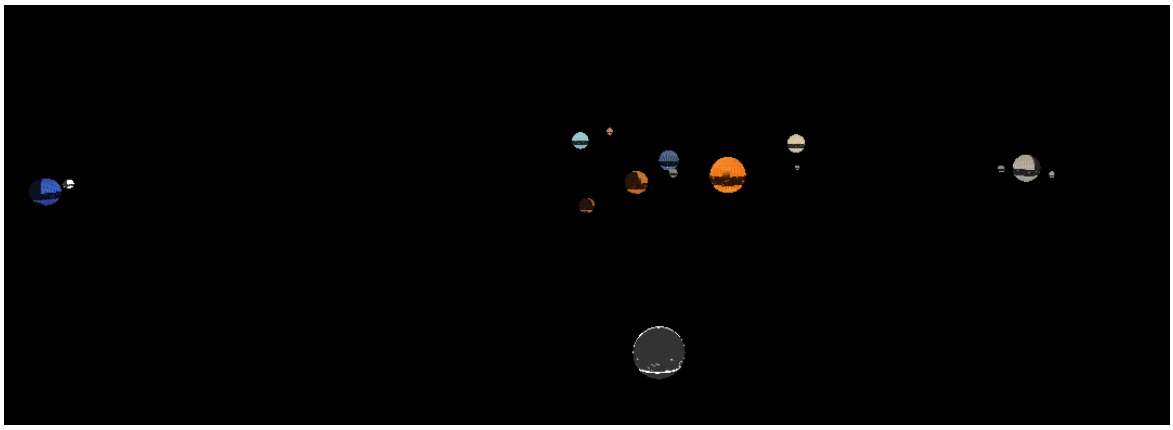


Figura 18 – Representação final do nosso sistema solar

Conclusão

Nesta quarta e última fase deste trabalho prático, foi necessário desenvolver e aplicar os nossos conhecimentos gerais acerca de *OpenGL* e da linguagem C++, assim como de toda a matéria da Unidade Curricular de Computação Gráfica, adquiridos nas aulas e nos guiões práticos, aprofundando-os, devido à ainda maior complexidade dos objetivos propostos.

De uma forma geral, tanto desta fase em específico como do projeto como um tudo, fazemos um balanço positivo, considerando que o trabalho realizado cumpre para com os objetivos propostos. Todo o esforço colocado no mesmo culminaram num resultado final do qual nos orgulhamos e pensamos representar esse mesmo esforço. Apesar disso, sabemos não estar perfeito, principalmente esta quarta fase do projeto, pois não obtivemos os resultados esperados na aplicação de texturas e iluminação, que se deveu, muito provavelmente, a erros nos cálculos das normais e pontos de textura.