Computação Gráfica

Trabalho prático – Fase 1 + 2 + 3 + 4

31 de maio de 2020

Grupo nº 56

Filipa Alves dos Santos (A83631)

Hugo André Coelho Cardoso (A85006)

João da Cunha e Costa (A84775)

Rui Alves dos Santos (A67656)




Mestrado Integrado em Engenharia Informática

Universidade do Minho

Índice de conteúdos

1. Introdução

O trabalho prático da Unidade Curricular Computação Gráfica tem por objetivo fundamental ajudar à consolidação experimental dos conhecimentos teórico-práticos lecionados na disciplina, bem como a aquisição de novas capacidades que não chegam a ser abordadas em contexto de sala de aula, nomeadamente a construção de um *engine*.

O projeto proposto consiste no desenvolvimento de um motor 3D baseado num cenário gráfico que explore e exiba as suas capacidades através de exemplos visuais e interativos, com recurso às ferramentas abordadas na disciplina – C++/C e OpenGL. O objetivo final é construir uma simulação realista do Sistema Solar, mantendo o melhor equilíbrio possível entre a componente gráfica (movimentos, texturas, iluminações) e a performance.

A fase inicial passou por desenvolver dois elementos fundamentais para a operabilidade e escalamento do trabalho, que virão a ser usados até à sua conclusão:

- Gerador: responsável por conceber ficheiros, em XML, com a informação dos modelos (no contexto desta primeira fase, apenas os vértices necessários para desenhar certas primitivas gráficas – plano, caixa, esfera e cone) a partir dos requisitos especificados (por exemplo, número de *stacks* e *slices*).
- Motor: responsável por analisar a informação dos ficheiros criados pelo gerador e exibir graficamente os modelos descritos nos mesmos, com recurso à biblioteca do GLUT.

Já na segunda fase, usamos os modelos gerados na fase anterior (neste caso, apenas necessitamos da esfera) para criar um modelo do sistema solar. Também implementamos estruturas e transformações recursivas no motor para obter o modelo 3d pretentido.

A fase 3 deste trabalho pode-se resumir nestes pontos distintos:

- A nível do gerador, é implementada a funcionalidade de gerar os triângulos de uma figura quando é fornecido um ficheiro .patch utilizando curvas de Bezier;
- Quanto ao motor, todas as figuras são agora desenhadas usando VBOs e é dada a possibilidade de novas transformações, como translações e rotações por tempo e usando pontos da curva de Camull-Rom.

Por fim, a última fase do trabalho pede a implementação de iluminação e texturas. Esta implementação necessitámos de adaptar o gerador para gerar tantos as normais como o mapeamento das texturas. Também foi necessário modificar o motor para este ser capaz de iluminar e desenhar as texturas.

2. Fase 1 – Primitivas Gráficas

2.1. Gerador

O gerador tem como objectivo crias os ficheiros que contém todos os pontos necessários para o desenho de um determinado sólido. Este será um executável independente logo, optamos por criar um projecto simples em C++ utilizando o Visual Studio. O executável poderá depois ser usado com variados argumentos de modo a gerar diferentes sólidos, sendo que o último argumento será o nome do ficheiro a ser criado.

2.1.1. Main

A main será apenas um simples switch, que por sua vez chamará a função que irá calcular os pontos do sólido correspondente.

2.1.2. Plano

O plano recebe como argumento dois inteiros, altura e comprimento. O plano irá estar assente no plano XoZ, centrado na origem. Visto que é apenas um quadrado, não foi preciso nenhum algoritmo, pois o desenho dos 2 triângulos é direto.

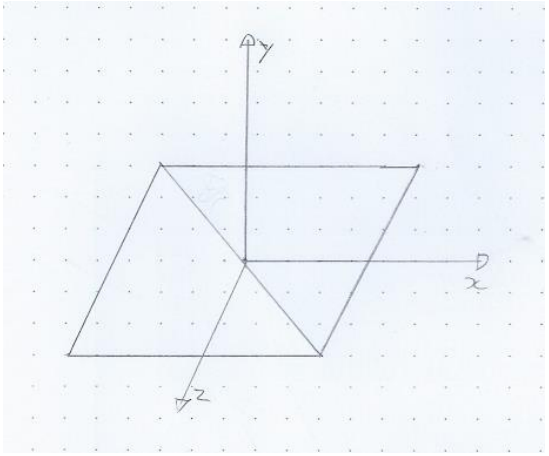

Figura 1 – Rascunho do Plano

Nota: Foi incluído o código para gerar o plano que possa ser visto dos dois lados, ou seja, com 4 triângulos.

2.1.3. Caixa

A caixa recebe quatro argumentos: comprimento, altura, largura e número de divisões, sendo que este último irá representar em quantas secções cada eixo será divido (por exemplo, para uma caixa com 3 secções cada face irá ser composta por 9 quadrados, ou 18 triângulos). O último campo pode ainda ser omitido, nesse caso cada face será apenas um quadrado. A caixa irá ser centrada na origem.

Visto que todas as faces são paralelas aos eixos, o algoritmo de determinação dos pontos é bastante simples. Para desenhar cada face iremos utilizar dois ciclos: o primeiro irá percorrer as camadas (cortes horizontais) e o segundo as fatias (cortes verticais). Ou seja, iremos começar numa esquina, calcular o quadrado (isto é, os dois triângulos que o compõe), e percorrer todos quadrados da linha até atingir o fim da mesma. De seguida, repetiremos o processo para a linha de cima, até todas as linhas estarem completas.

O algoritmo para determinar os pontos da face da frente seria:

```
for (int i = 0; i < divisões; i++) {      // percorre as linhas
   for (int j = 0; j < divisões; j++) {   // percorre cada quadrado na linha
      Escrever (p1,p2,p3);
      Escrever (p1,p3,p4);
   }
}
```

Em que os pontos p1, p2, p3, p4 (figura 2) são definidos da seguinte maneira:

```
p1 = (-xi + j*x     ,   -yi + i*fy       , zi)

p2 = (-xi + x + j*x ,   -yi + i*fy       , zi)

p3 = (-xi + x + j*x ,  -yi + y + i*fy    , zi)

p4 = (-xi +  j*x    ,  -yi + y + i*fy    , zi)
```

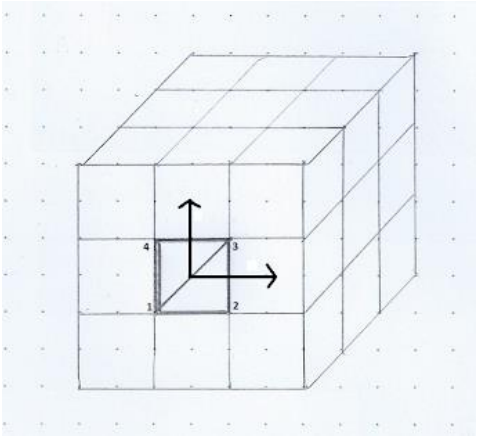

Figura 2 – Rascunho da Caixa

```
//dividir por 2 para centrar na origem
xi = largura / 2; yi = altura / 2; zi = compr / 2;


//incremento por iteração
x = largura / divs; y = altura / divs; z = compr / divs;
```

\Generatoror.exe box 3 3 3 caixa.3d

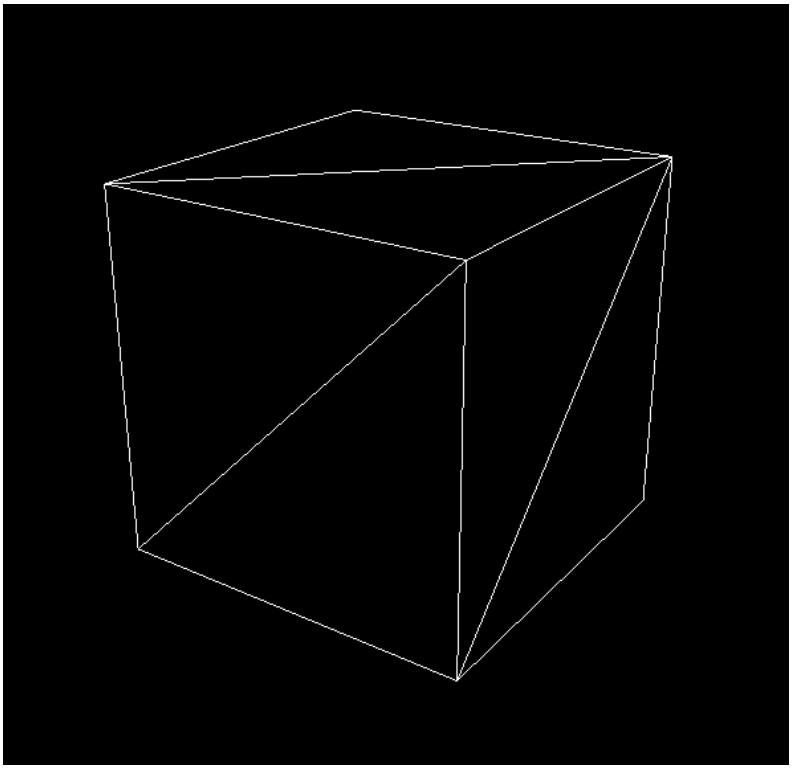

Figura 3 – Exemplo 1 de uma Caixa

.\Generator.exe box 3 3 3 5 caixa.3d

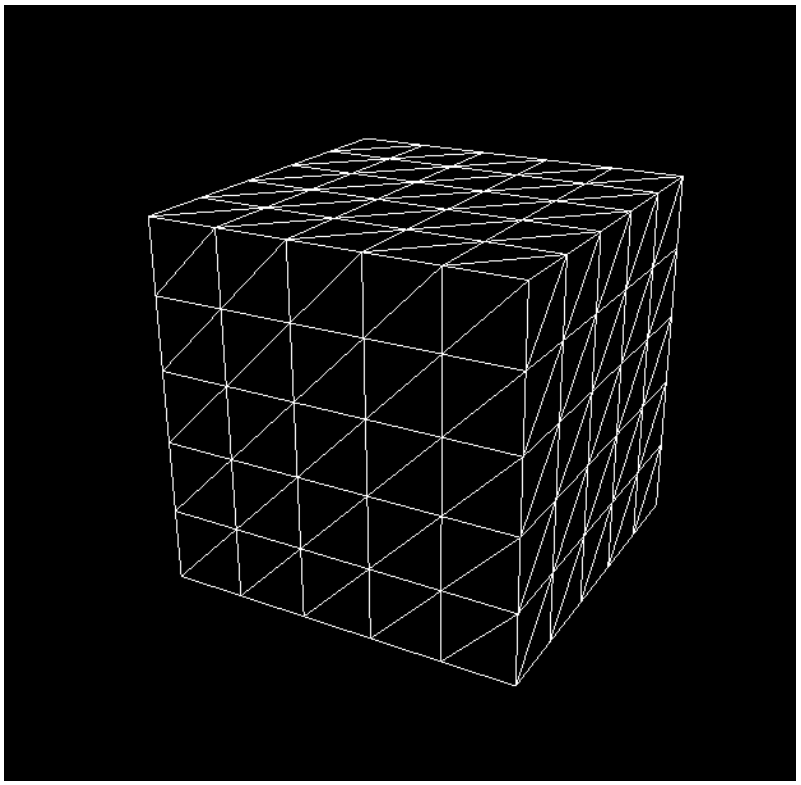

Figura 4 – Exemplo 2 de uma Caixa

2.1.4. Cone

O cone recebe 4 argumentos, raio da base, altura, camadas (cortes horizontais) e fatias (cortes verticais, neste caso cortes até ao eixo central, como fatias de um bolo, mas num cone). A base encontra-se no plano XoZ e o vértice no eixo Y.

O algoritmo para determinar os pontos será divido em duas partes: desenhar a base e desenhar as laterais.

A base terá um algoritmo simples, pois apenas teremos que usar coordenadas polares para determinar X e Z sendo que Y é sempre 0. Assim, vamos ter:

```
alfa=0;
alfainc=(2*pi)/fatias
for (int j = 0; j < fatias; j++){
      Escrever (p0,p1,p2);
      alfa+=alfainc
}
```

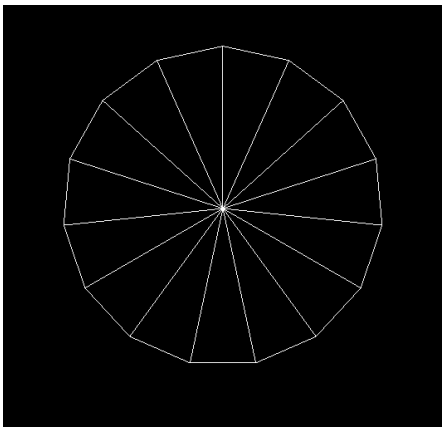

Figura 5 – Base de um Cone

Em que p0, p1, p2 são definidos por:

```
p0 = (           0            , 0 ,             0            )
p1 = ( raio*sin(alfa+alfainc) , 0 , raio * cos(alfa + alfainc))
p2 = ( raio*sin(alfa)         , 0 ,     raio * cos(alfa)      )
```

De modo a desenhar as laterais, o cone irá ser desenhado de baixo para cima, uma camada de cada vez. Podemos pensar em cada camada como um "disco" com um raio mais pequeno face de cima do que na base. Deste modo cada disco terá um número de quadrados igual ao número de fatias. Calculamos os pontos usando uma vez mais as coordenadas polares, como mostra a imagem ao lado.

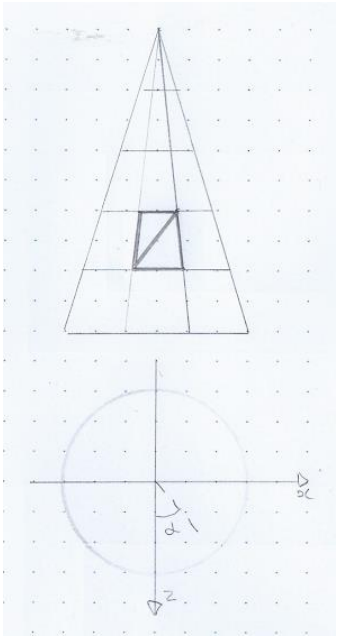

Figura 6 – Rascunho de um Cone

Deste modo o algoritmo para desenhar a lateral será:

```
altura = 0;
alfainc = (2*pi)/fatias;
alturainc = alturaTotal / camadas;
for (int i = 0; i < camadas; i++) { // uma iteração por camada
       alfa = 0;
       raioCima = (alturaTotal - altura - alturainc) / (alturaTotal / raio);
       raioBaixo = (alturaTotal - altura) / (alturaTotal / raio);
       for (int j = 0; j < slices; j++) { // desenhar os quadrados da camada
              Escrever(p1.p2.p3);
              Escrever(p2.p3.p4);
              alfa += alfainc;
       }
       altura += alturainc;
}
```

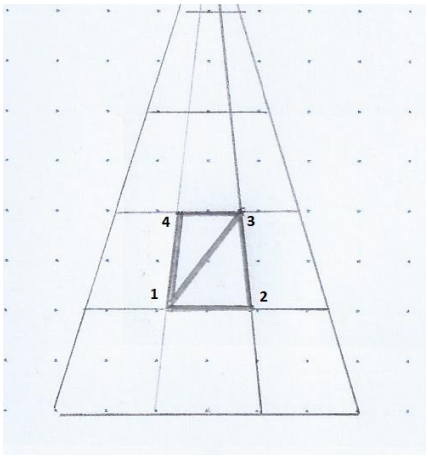

Figura 7 – Rascunho da lateral de um Cone

Em que p1, p2, p3 e p4 são definidos por:

```
p1 = (     raioBaixo * sin(alfa)      ,      altura      , raioBaixo *cos(alfa) );
p2 = ( raioBaixo * sin(alfa + alfainc),      altura      , raioBaixo *cos(alfa+ alfainc) );
p3 = ( raioCima * sin(alfa + alfainc) , altura+alturainc , raioCima  *cos(alfa+ alfainc) );
p4 = (     raioCima * sin(alfa)       , altura+alturainc , raioCima  *cos(alfa) );
```

.\Generator.exe cone 2 3 5 5 cone.3d

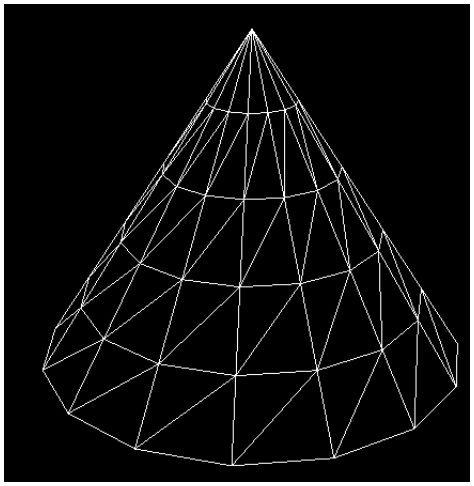

Figura 8 – Exemplo de um Cone

2.1.5. Esfera

Uma esfera recebe 3 argumentos, raio, número de camadas e número de fatias. Será centrada na origem.

De modo a determinar os pontos que constituem a esfera iremos usar um algoritmo semelhante ao do cone, mas utilizando coordenadas esféricas para maior facilidade de cálculo. A esfera será desenhada camada a camada, de baixo para cima. Cada camada irá ser percorrida utilizando o ângulo alfa e para determinarmos a altura das camadas, utilizaremos o ângulo beta, como demonstrado na figura. Deste modo, o algoritmo será:

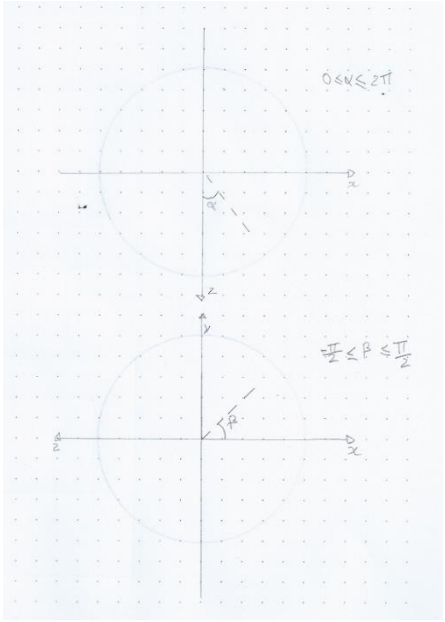

Figura 9 – Rascunho de uma camada da Esfera

```
ainc = (2 * piI) / fatias;
binc = (pi) / camadas;


float alfa = 0;
float beta = -pi/2;


for (int i = 0; i < camadas; i++) {
       alfa = 0;
       for (int j = 0; j < fatias; j++) {
              Escrever (p1,p2,p3);
              Escrever (p2,p3,p4)
               alfa += ainc
       }
       beta += binc
}
```

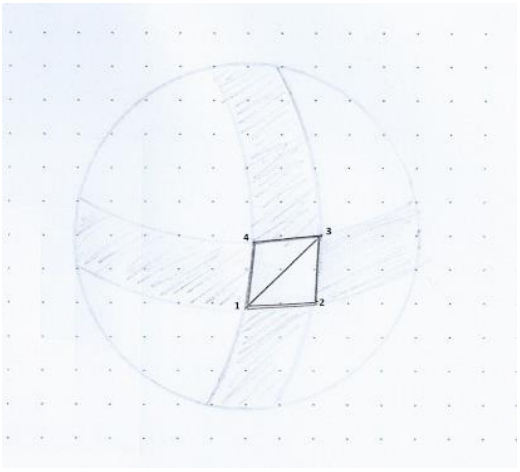

Figura 10 – Rascunho de uma secção da Esfera

Em que p1, p2, p3 e p4 são definidos por:

```
p1 = (     raio*sin(alfa)*cos(beta)      ,    raio*sin(beta)   , raio*cos(alfa)*sin(beta) )
p2 = (raio*sin(alfa+ainc)*cos(beta)      ,    raio*sin(beta)   , raio*cos(alfa+ainc)*sin(beta) )
p3 = (raio*sin(alfa+ainc)*cos(beta+binc) , raio*sin(beta+binc) , raio*cos(alfa+ainc)*sin(beta+binc) )
p4 = (  raio*sin(alfa)*cos(beta+binc)    , raio*sin(beta+binc) , raio*cos(alfa)*sin(beta+binc) )
```

.\Generator.exe sphere 2 5 5 esfera.3d

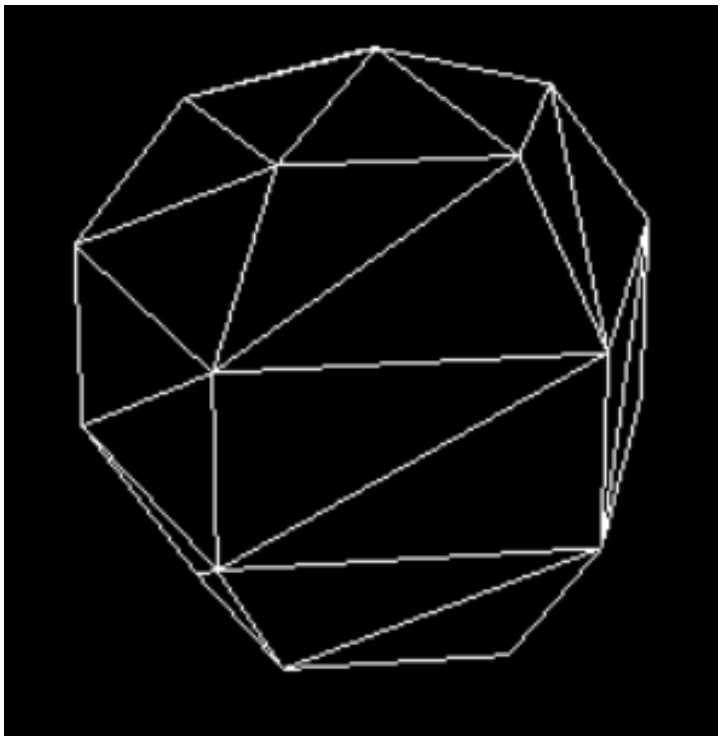

Figura 11 – Exemplo 1 de uma "Esfera"

.\Generator.exe sphere 2 20 20 esfera.3d

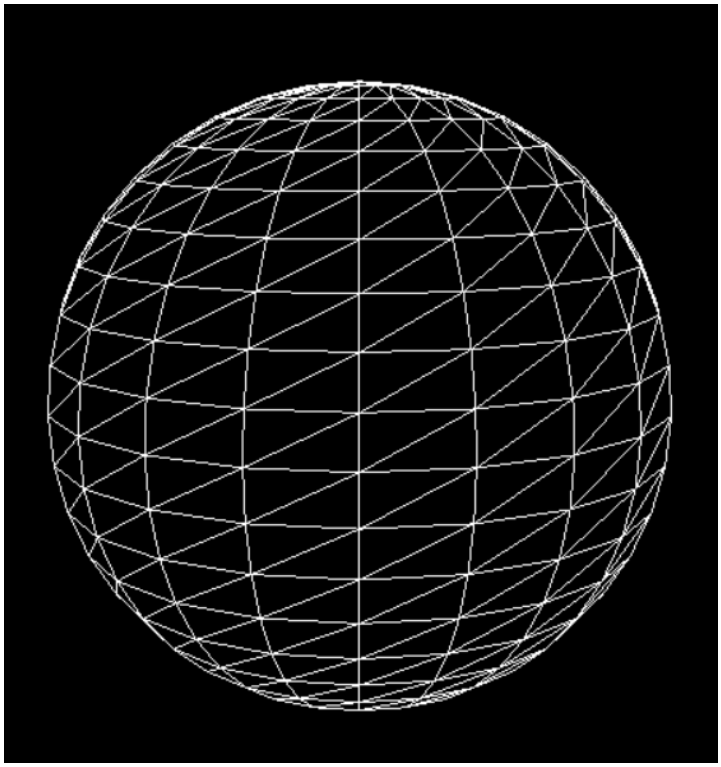

Figura 12 – Exemplo 2 de uma Esfera

2.2. Motor

O nosso motor será um programa bastante simples, responsável apenas por ler o ficheiro de config e os ficheiros escritos pelo gerador, criando depois o output 3d com a informação contida nos mesmos.

Numa primeira fase, irá ser lido o ficheiro "config.xml", ficheiro que contém uma lista de nomes de todos o ficheiros tipo .3d a ser carregados. Os nomes serão guardados num vector chamado files.

Seguidamente, a função parseFiles() é chamada, que irá iterar o vector criado anteriormente e, para cada nome, vai ser criada uma nova estrutura de dados, que irá de seguida ser povoada com os triângulos da mesma.

```
struct Triangle{

    float x1, y1, z1;

    float x2, y2, z2;

    float x3, y3, z3;

};

struct Model {

    string nome;

    vector<Triangle> triangles;

};
```

Numa fase final, a função drawModel() que itera o vector de modelos e, para cada modelo, irá iterar o seu vector de triângulos.

Embora os triângulos já se encontrem num buffer e fosse relativamente simples implementar VBO's, cada triângulo irá ser desenhado usando glBegin(GL_TRIANGLES), visto não ser um requisito desta fase, e como nos foi lecionado perto da entrega do trabalho, aumentaria muito a hipótese de introdução de bugs no nosso programa.

No nosso motor opengl, é possível movimentar a nossa câmara de explorador. W A S D irá mover a câmara para cima, baixo, esquerda e direita, correspondentemente. Q e E para zoom in e zoom out, e X para esconder ou mostrar os eixos.

Ao longo de toda a execução irá ser imprimida informação sobre os modelos na consola, bem como das teclas que podem ser usadas.

Correção da Fase 1

Ao executar o código da Fase 1 em modo Debug, deperámo-nos com um pequeno erro. Na linha responsável por apresentar os fps no topo da janela (exemplifcada em baixo), a string s não tinha sido inicializada.

```
sprintf(&s.front(), "%d", fps);
```

Como tal,

```
string s;
```

foi substituído por

```
string s = "d";
```

3. Fase 2 – Transformações Geométricas

3.1. Gerador

O gerador não necessitou de qualquer alteração em relação à primeira fase, visto que os sólidos usados são os mesmos.

3.2. Motor

Ao contrário do gerador, o motor necessitou de várias alterações. Como foram introduzidos três tipos de transformações, bem como uma hierarquia de objectos, consideramos vantajoso adaptar a nossa estrutura de dados.

```
struct Model {

	float tx = 0, ty = 0, tz = 0;

	float ra = 0, rx = 0, ry = 0, rz = 0;

	float sx = 1, sy = 1, sz = 1;

	string nome;

	vector<Triangle> triangles;

	vector<Model> submodules;

};
```

Como podemos verificar acima, acrescentamos 3 conjuntos de floats (um para cada uma das transformações), bem como um vector da própria estrutura para criamos a nossa hierarquia, tornando assim a estrutura recursiva. Deste modo, conseguimos níveis infinitos de hierarquia, embora o nível máximo no nosso caso seria 3 (Sol->Planetas->Luas), o nosso código permite níveis infinitos, como mostrado na figura abaixo.

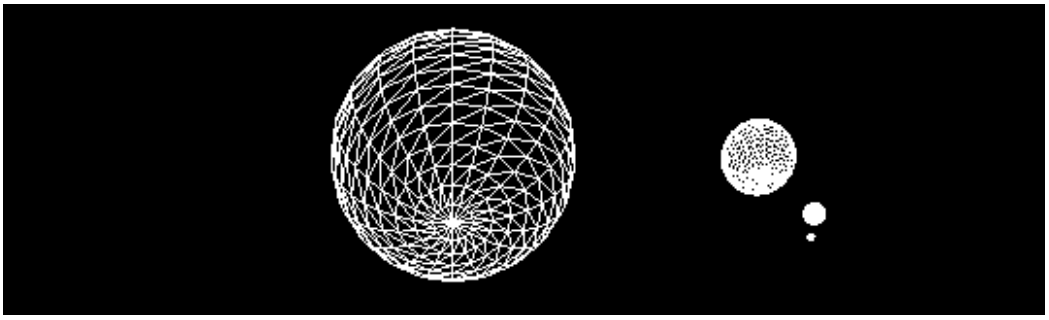

Figura 13 – 4 níveis de objetos

Num passo seguinte, tivemos também que alterar o nosso parser de XML. Este agora encontra-se pronto para ler os novos ficheiros de configuração. O parser irá guardar todos os dados de transformações da figura e de seguida, aplicar recursividade sobre sub-modelos existentes. É possível até criar vários sistemas solares, como na figura 14. No final, é apenas necessário dar parse aos ficheiros .3d e guarda-los no vector de triângulos.

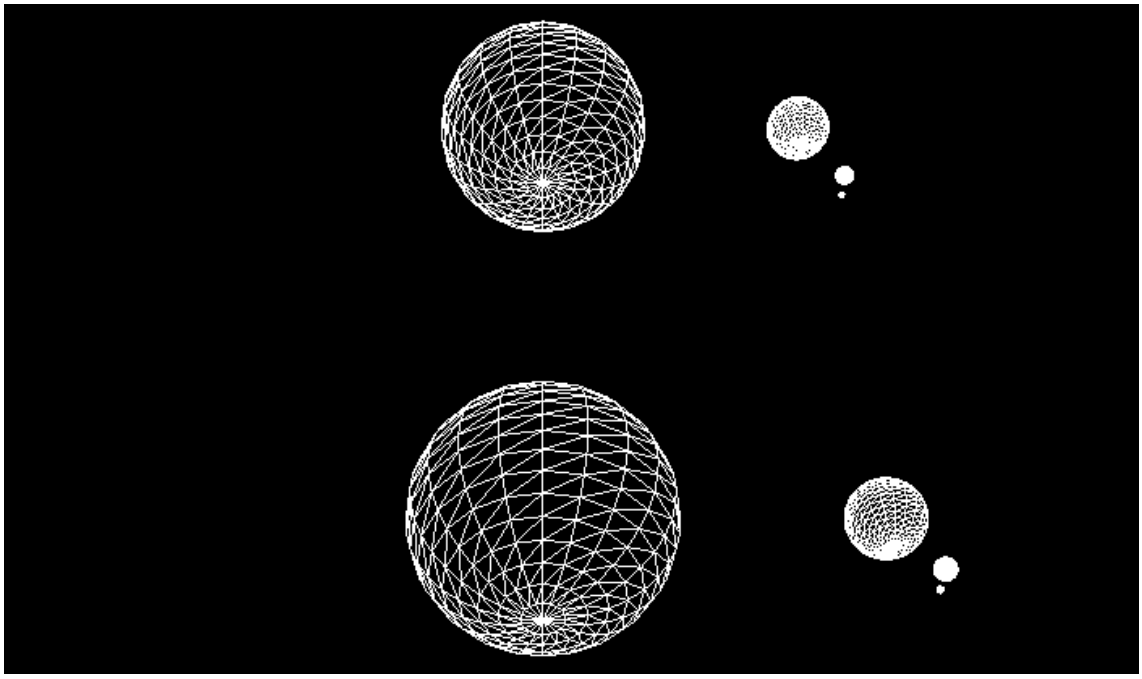

Figura 14 – Exemplos de vários sistemas

Tendo a nossa estrutura de dados atualizada e povoada pelo novo parser, faltará agora apenas atualizar a função de desenho para ter em conta tanto as transformações como as hierarquias. Da mesma maneira que construímos a estrutura e o parser, a função também foi feita recursivamente, permitindo assim desenho de vários sistemas solares, com níveis de hierarquia infinitos, em que os filhos herdam todas as transformações dos pais.

Assim dito, utilizando o ficheiro config.xml incluído, teremos um sistema solar. Podemos interagir com o mesmo rodando a câmara com WASD, fazendo Zoom-in ou Zoom-out com QE, e ainda podemos fazer os planetas a orbitar (duma maneira muito simplificada, longe das rotações reais) usando a tecla F.

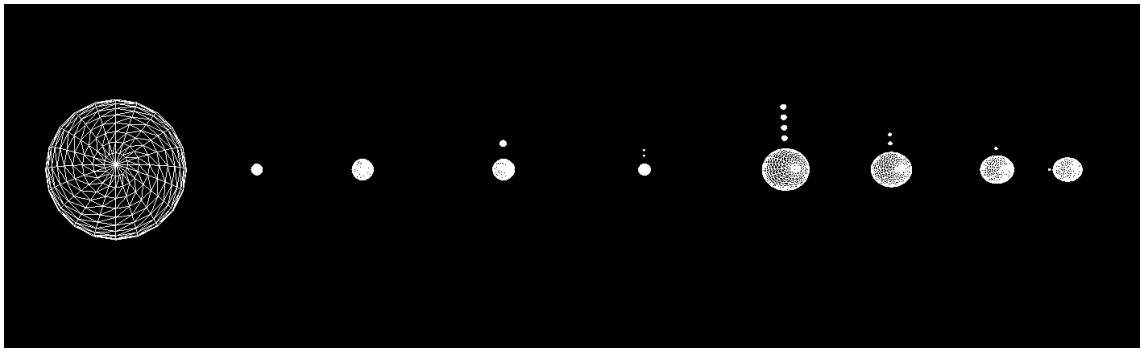

Figura 15 – Sistema Solar no instante inicial

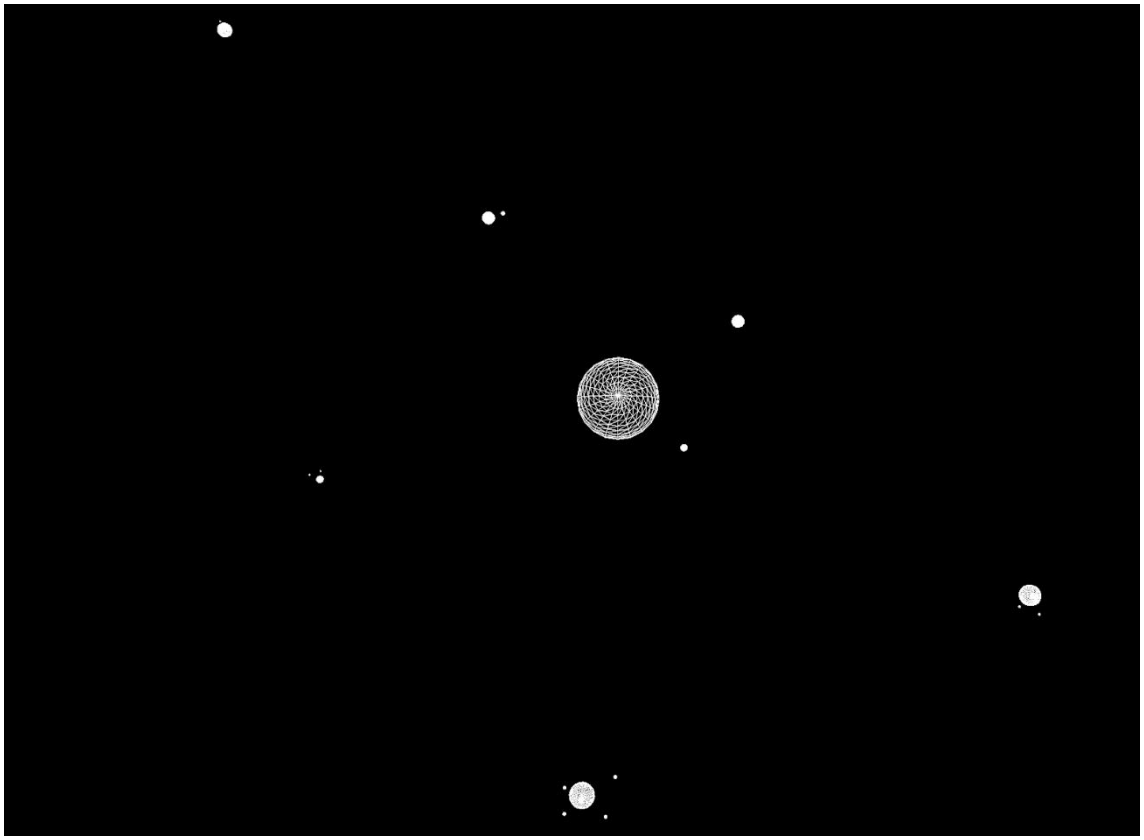

Figura 16 – Sistema Solar alguns “anos” depois

4. Fase 3 – Curvas, Superfícies Cúbicas e VBOs

4.1. Gerador e Curvas de Bezier

O primeiro passo de gerar a nossa figura por pontos de Bezier será ler o nosso ficheiro patch. Para tal, consideramos vantajoso criar uma pequena estrutura de dados auxiliar: usamos um vector para os pontos e outro vector 2D para os índices.

```
struct Point {
    float x;
    float y;
    float z;
};

struct PatchData {
    int nrPatches;
    vector<vector<int>> indices;
    int nrPontos;
    vector<Point> pontos;
};

PatchData patchDados;
```

Figura 17 – Estruturas de dados

Como tal, a primeira função a ser chamada será a `void loadPatch(char* patch)` que recebe um ficheiro e povoa a estrutura acima. Tem um algoritmo bastante simples: lê o número de patches, de seguida itera e guarda cada lista de indíces dos patches e por fim itera e guarda os pontos de controlo.

```
void loadPatch(char* patch) {
    char buffer[4096];   //talvez devessemos usar ifstream, mas nao sei muito bem, e ca
    char* temp;
    FILE* file = fopen(patch, "r");

    if (file == NULL) {
        printf("Fich n encontrado\n"); return;
    }

    fgets(buffer, 4096, file);//ler nr de patches
    temp = strtok(buffer, "\n");
    patchDados.nrPatches = atoi(temp);

    for (int a = 0; a < patchDados.nrPatches; a++) {
        patchDados.indices.push_back({});//needed para inicializar?!
        fgets(buffer, 4096, file);//ler uma linha dos indices

        temp = strtok(buffer, " ");
        while (temp[strlen(temp) - 1] != '\n') {
            patchDados.indices.at(a).push_back(atoi(temp));
            temp = strtok(NULL, " ");
        }
        temp[strlen(temp) - 1] == '\0';
        patchDados.indices.at(a).push_back(atoi(temp));
    }

    fgets(buffer, 4096, file);
    temp = strtok(buffer, "\n");
    patchDados.nrPontos = atoi(temp);

    for (int b = 0; b < patchDados.nrPontos; b++) {    //le um ponto, guarda um ponto
    }
```

Figura 18 – Função loadPatch

Com todos os dados carregados, podemos começar a gerar os nosso pontos através da função `void gerarPontoBezier(double u, double v, int a, double r[3])` utilizando a seguinte fórmula:

$$B(u,v) = [u^3 \quad u^2 \quad u \quad 1]M\begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Começamos por calcular res=U*M, depois calculamos cada coordenanda individualmente e como tal teremos resX=res*Px, resY=res*Py e resZ=res*Pz. O passo seguinte será multiplicar cada coordenada por Mt e por fim por V, obtendo assim as 3 coordenadas do ponto a desenhar.

```
void gerarPontoBezier(double u, double v,int a, double r[3])
{
    double up[1][4] = { {powf(u,3), powf(u,2), u, 1} };
    double m[4][4] = { {-1.0f,  3.0f, -3.0f,  1.0f}, ...
    double mT[4][4] = { {-1.0f,  3.0f, -3.0f, 1.0f}, ...
    double vH[4][1] = { {powf(v,3)}, ...
    double pX[4][4];
    double pY[4][4];
    double pZ[4][4];

    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            pX[i][j] = patchDados.pontos.at(patchDados.indices.at(a).at(i * 4 + j)).x;
            pY[i][j] = patchDados.pontos.at(patchDados.indices.at(a).at(i * 4 + j)).y;
            pZ[i][j] = patchDados.pontos.at(patchDados.indices.at(a).at(i * 4 + j)).z;
        }
    }

    double res[1][4]; double resX[1][4];
    double resY[1][4]; double resZ[1][4];

    double tX[1][4];double tY[1][4];double tZ[1][4];

    double x;double y;double z;
                                    /*   res = u * m    */
    multMatrix(up, m, res);
                                    /* resx = res * px,   y e z  */
    multMatrix(res, pX, resX);
    multMatrix(res, pY, resY);
    multMatrix(res, pZ, resZ);
                                    /* tx = resx * mt, y e z */
    multMatrix(resX, mT, tX);
    multMatrix(resY, mT, tY);
    multMatrix(resZ, mT, tZ);
                                    /*  x = transX * V       */
    multMatrix2(tX, vH, &x);
    multMatrix2(tY, vH, &y);
    multMatrix2(tZ, vH, &z);
                            /*  Colocar os resultados no vetor a dar "return"   */
    r[0] = x;
    r[1] = y;
    r[2] = z;
}
```

Figura 19 – Função gerarPontoBezier

Com a nossa função que gera pontos completa, podemos fazer um simples algoritmo para gerar a nossa figura e guardar no ficheiro.

Iremos iterar cada patch, e de seguida uma grelha, cujo tamanho irá depender apenas do tesselation. Para cada uma destas grelhas, iremos gerar sempre 4 pontos de cada vez. Estes representam um quadrado (2 triângulos na prática) e são de imediato escritos no ficheiro .3d, terminando assim o processo de gerar a figura.

```
void buildBezier(char* patch, int tess, char* outFile) {
    double uu, vv, uuu, vvv;
    double p1[3], p2[3],p3[3],p4[3];
    std::ofstream ofs(outFile, std::ofstream::out);

    for (int a = 0; a < patchDados.nrPatches; a++) {

        for (int u = 0 ; u < tess ; u++) {
            for (int v = 0 ; v < tess ; v++) {
                uu = (double)       u / tess;
                vv = (double)       v / tess;
                uuu = (double) (u+1) / tess;
                vvv = (double) (v+1) / tess;
                gerarPontoBezier(uu, vv, a, p1);
                gerarPontoBezier(uu, vvv, a, p2);
                gerarPontoBezier(uuu, vvv, a, p3);
                gerarPontoBezier(uuu, vv, a, p4);

                //1  2
                //4  3
                ofs << p1[0] << " " << p1[1] << " " << p1[2] << " " << p2[0] << " " << p2[1] << " " << p2[2] << " "
                ofs << p1[0] << " " << p1[1] << " " << p1[2] << " " << p3[0] << " " << p3[1] << " " << p3[2] << " "
            }
        }
    }
    ofs.close();
}
```

Figura 20 – Função buildBezier

4.2. VBOs

Visto que já estamos a guardar todos os pontos numa estrutura de dados, esta fase não forçou muitas mudanças. Optamos por guardar todos os pontos em

```
vector<vector<float>> pontos;
```

guardando apenas a posição do vector na estrutura principal.

De modo a inicializar as nossas VBOs e fazer o povoamento com data, utilizamos a função `void createVbo():`

```
void createVBO() {

    glGenBuffers(pontos.size(),buffers);

    for (int a = 0; a < pontos.size(); a++) {
        glBindBuffer(GL_ARRAY_BUFFER, buffers[a]);
        glBufferData(GL_ARRAY_BUFFER, pontos.at(a).size() * sizeof(float), &pontos.at(a).front(), GL_STATIC_DRAW);
    }
}
```

Figura 21 – Função createVBO

Mais tarde, a função `renderScene` irá chamar a `drawVBO`. Esta função será responsável por todas as transformações geométricas (que vão ser discutidas no capítulo seguinte) bem como o desenho das nossas figuras em si, como podemos ver no excerto seguinte:

```
int location = planetas.at(p).pos;

glBindBuffer(GL_ARRAY_BUFFER, buffers[location]);
glVertexPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, pontos.at(location).size() / 3);
```

Figura 22 – Função drawVBO (1)

De modo a poder avaliar o ganho de performance, introduzimos o cálculo de fps:

```
frame++;
time = glutGet(GLUT_ELAPSED_TIME);
if (time - timebase > 1000) {
    fps = frame * 1000.0 / (time - timebase);
    timebase = time;
    frame = 0;
}
```

Figura 23 – Cálculo de FPS

NOTA: Embora tenhamos tido um ganho claro de frames, não conseguimos concluir porque é que computadores diferentes executam o programa com um número de fps tão diferente, número este que é superior em algumas das piores máquinas.

4.3. Catmull-Rom e Transformações

A função `drawVBO` irá ser responsável por chamar as funções de cálculo de Catmull, todas as transformações e por fim o desenho.

4.3.1. Catmull-Rom

De modo a calcular pontos de Catmull, utilizamos a função utilizada na ficha 8 (apenas com a pequena adaptação de receber o nosso planeta como argumento) da qual obtemos a posição e a derivada.

```
void getCatmullRomPoint(float t, float* p0, float* p1, float* p2, float* p3, float* pos, float* deriv) {

    // catmull-rom matrix
    float m[4][4] = { {-0.5f,  1.5f, -1.5f,  0.5f},
                      { 1.0f, -2.5f,  2.0f, -0.5f},
                      {-0.5f,  0.0f,  0.5f,  0.0f},
                      { 0.0f,  1.0f,  0.0f,  0.0f} };


    float xx[4] = { p0[0],p1[0],p2[0],p3[0] };
    float yy[4] = { p0[1],p1[1],p2[1],p3[1] };
    float zz[4] = { p0[2],p1[2],p2[2],p3[2] };

    // Compute A=M*P
    float ax[4];
    float ay[4];
    float az[4];

    multMatrixVector((float*)m, (float*)xx, (float*)ax);
    multMatrixVector((float*)m, (float*)yy, (float*)ay);
    multMatrixVector((float*)m, (float*)zz, (float*)az);
    // Compute pos=T*A

    float T[4] = { pow(t,3), pow(t,2), t, 1 };
    pos[0] = T[0] * ax[0] + T[1] * ax[1] + T[2] * ax[2] + T[3] * ax[3];
    pos[1] = T[0] * ay[0] + T[1] * ay[1] + T[2] * ay[2] + T[3] * ay[3];
    pos[2] = T[0] * az[0] + T[1] * az[1] + T[2] * az[2] + T[3] * az[3];

    // compute deriv=T'*A
    float TT[4] = { 3 * pow(t,2), 2 * pow(t,1), 1, 0 };
    deriv[0] = TT[0] * ax[0] + TT[1] * ax[1] + TT[2] * ax[2] + TT[3] * ax[3];
    deriv[1] = TT[0] * ay[0] + TT[1] * ay[1] + TT[2] * ay[2] + TT[3] * ay[3];
    deriv[2] = TT[0] * az[0] + TT[1] * az[1] + TT[2] * az[2] + TT[3] * az[3];
}
```

Figura 24 – Função getCatmullRomPoint

4.3.2. Transformações

A função `drawVBO` irá aplicar as transformações, com o seguinte algoritmo:

Translação
- Catmull
- Simples, por tempo.

Rotação
- Por tempo
- Estática

Escala

O primeiro passo será, portanto, as translações. Se existirem pontos de Catmull, iremos entrar no seguinte if :

```
if (planetas.at(p).transtime != 0 && planetas.at(p).catPoints.size() > 0)  { //ca
    renderCatmullRomCurve(planetas.at(p));
    getGlobalCatmullRomPoint(tt, (float*)pos, (float*)derivada, planetas.at(p));
    glTranslatef(pos[0], pos[1], pos[2]);

    normalize((float*)derivada);

    cross((float*)derivada, (float*)planetas.at(p).up, (float*)z);
    normalize((float*)z);

    cross((float*)z, (float*)derivada, (float*)planetas.at(p).up);
    normalize((float*)planetas.at(p).up);

    buildRotMatrix(derivada, planetas.at(p).up, z, *m);

    transpose(m, mm);
    glMultMatrixf((float*)*mm);
```

Figura 25 – Função drawVBO (1)

Na função explicada anteriormente, começa-se por desenhar a órbita do planeta, seguido do cálculo da posição e derivada, que são imediatamente aplicadas. De modo a calcular a posição na órbita, relativamente ao tempo de translação utilizamos:

```
float tt = fmod((a / (planetas.at(p).transtime * 1000)), 1);
```

Caso não existam pontos de Catmull mas exista um tempo de órbita, utilizamos uma simples rotação e translação, assumindo uma órbita perfeitamente circular:

```
else if (planetas.at(p).transtime != 0) { //se nao tiver catmull é uma rotaçã
    glRotatef(ttt, planetas.at(p).rx, planetas.at(p).ry, planetas.at(p).rz);
    glTranslatef(planetas.at(p).tx, planetas.at(p).ty, planetas.at(p).tz);
        }
```

Figura 26 – Função drawVBO (2)

Para este caso, a nossa variável tempo seria:

```
float ttt = fmod((a / (planetas.at(p).transtime * 1000)), 1) * 360;
```

Com os planetas no local certo, resta agora aplicar as nossas rotações. Uma vez mais teremos duas hipóteses: se existir tempo de rotação iremos calcular o angulo de rotação usando

```
float tttt = fmod((a / (planetas.at(p).rotatetime * 1000)), 1) * 360;
```

Caso não exista tempo de rotação podemos ainda aplicar uma rotação estática.

```
if (planetas.at(p).rotatetime != 0) {
    glRotatef(tttt, planetas.at(p).rx, planetas.at(p).ry, planetas.at(p).rz); //rotaçao port tempo
}
else if (planetas.at(p).ra!=0) {
    glRotatef(planetas.at(p).ra, planetas.at(p).rx, planetas.at(p).ry, planetas.at(p).rz); //rotaçao estatica
}
```

Figura 27 – Função drawVBO (3)

Para terminar, bastará apenas aplicar a escala e desenhar a nossa figura:

```
glScalef(planetas.at(p).sx, planetas.at(p).sy, planetas.at(p).sz);
glColor3f(1.0f, 1.0f, 1.0f);

int location = planetas.at(p).pos;

glBindBuffer(GL_ARRAY_BUFFER, buffers[location]);
glVertexPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, pontos.at(location).size() / 3);
```

Figura 28 – Função drawVBO (4)

5. Fase 4 – Normais e Coordenadas de Textura

5.1. Gerador

O nosso gerador, responsável por criar os ficheiros .3d, irá ser adaptado para gerar os pontos de mapeamento de textura bem como as normais a cada ponto, de modo a iluminar correctamente.

Até agora escrevíamos 3 pontos (ou seja, 1 triângulo por linha). De modo a satisfazer os novos requisitos, criou-se grupos de 3 linhas: a primeira guarda um triângulo, a segunda as normais a esse triângulo e a terceira os pontos de mapeamento de texturas. Cada sólido necessitará de algoritmos próprios para calcular cada um destes pontos.

5.1.1. Normais

Cada tipo de sólido irá ter um algoritmo próprio para calcular as normais do mesmo. No caso da caixa e do plano não é preciso qualquer algoritmo, pois as faces são sempre paralelas aos planos dos eixos. Deste modo, os vectores normais serão sempre do tipo (1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1) ou (0,0,-1).

A esfera também se provou bastante simples de calcular as normais. O nosso algoritmo para calcular os pontos da superfície da esfera (explicado na parte 1) usava coordenadas esféricas para calcular um vector unitário que era de seguida multiplicado pelo raio. Logo, tudo o que tivemos que fazer foi não efetuar essa multiplicação, obtendo assim o vector normal ao ponto.

Por último, para calcular as normais do nosso teapot (ou qualquer outra figura gerada por pontos de bezier) iremos calcular o produto vectorial dos vectores tangentes ao ponto em questão.

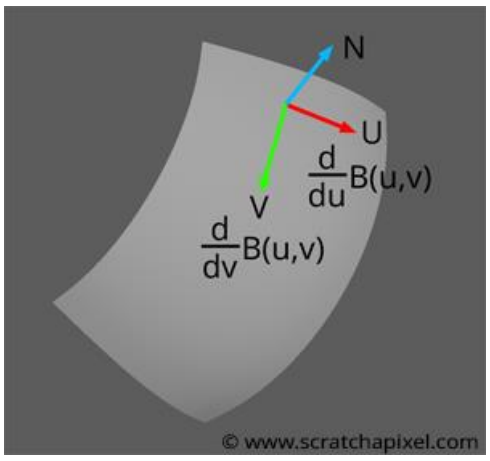

Figura 29 – Normais de uma figura gerada por pontos de bezier

Vector U:

$$\frac{\partial p(u,v)}{\partial u} = [3u^2 \quad 2u \quad 1 \quad 0]M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T V^T$$

Vector V:

$$\frac{\partial p(u,v)}{\partial v} = UM \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix}$$

O cálculo de cada um destes vectores será bastante semelhante ao cálculo de cada um dos pontos de bezier, e será feito nas funções `gerarUBezier` e `gerarVBezier`.

NOTA: No cáclulo do vector U é pedido que se use a transposta de V, o que nos levou a um erro de cálculo pois acabamos com duas matrizes não possíveis de multiplicar. Desse modo, usamos V em vez de V transposta e o resultado parece ser o pretendido.

No fim de calcular os dois vectores, apenas teremos de calcular o produto entre os mesmos, obtendo o seguinte resultado:

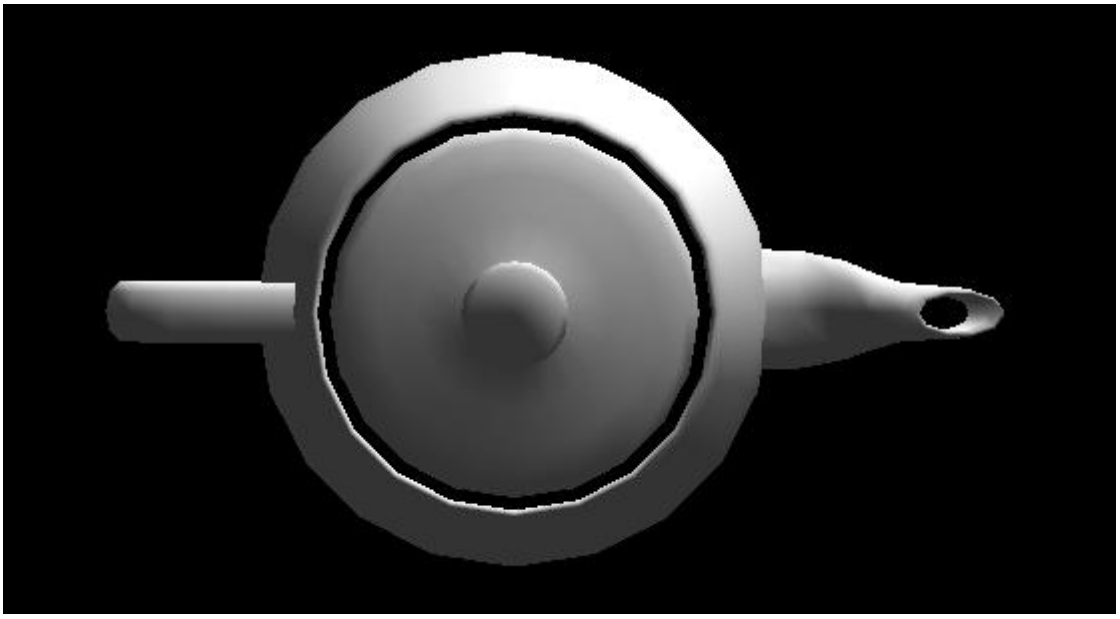

Figura 30 – Exemplo de Iluminação no teapot

5.1.2. Texturas

O cálculo do mapeamento de texturas foi apenas efectuado para a esfera. Em todos os outros sólidos inserimos apenas zeros, de modo a manter a consistência dos ficheiros .3d.

De modo a mapear imagens retangulares para a esfera, decidimos mapear cada stack da esfera, para um corte horizontal de imagem.

Figura 31 – Corte horizonatal de uma esfera

Este foi o algoritmo que nos pareceu mais viável, embora acabe a comprimir a imagem nas primeiras e últimas stacks (na realidade, a projeção também não seria um retângulo). Deste modo, o nosso algoritmo anula alguma da distorção criada ao mapear a superfície de uma esfera para um rectângulo.

O resultado final tem o seguinte aspecto:

Figura 32 – Planeta iluminado e com texturas

5.2. Motor

De modo a implementar as novas funcionalidades, começamos por adaptar a nossa estrutura de dados, atualizando os seguintes campos.

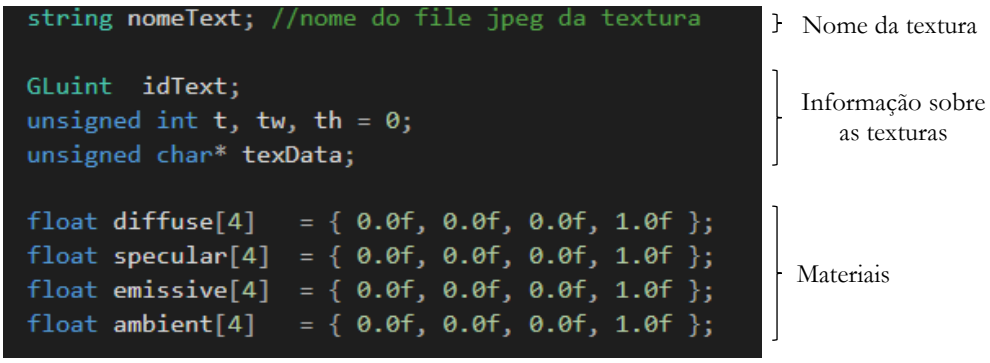

Figura 33 – Estrutura de dados atualizada

Ainda adicionamos novos vectores onde guardar os novos vectores e pontos, para buffers e, por fim, guardar as light sources.

```
vector<Light> luzes;

vector<vector<float>> pontos;     //so os pontos em si
vector<vector<float>> normais;    //normais para light
vector<vector<float>> texturas;   //text coords

GLuint buffers[128];              //buffers para as vbo's
GLuint buffersN[128];             //buffers para as normais
GLuint buffersT[128];             //buffers para as texCords
```

Figura 34 – Novos vectores

Da mesma maneira, alterámos a nossa função que dá parse ao XML de configuração (`parseConfig`), para ler estes novos campos e povoar a estrutura de dados. Foi também criada uma função nova, `parseLights`, com o objetivo de dar parse aos light sources.

Depois de dar parse no nosso ficheiro config, temos de dar load às nossas texturas. Isto será feito na função `loadTextures`, que irá iterar a estrutura de dados principal e carregar as texturas para todos os modelos que tenham um ficheiro de texturas associado. De seguida a função `prepareVBO` foi ligeiramente adaptada para preparar os novos buffers.

```
void prepareVBO() {

    glGenBuffers(pontos.size() ,buffers);
    glGenBuffers(normais.size(), buffersN);
    glGenBuffers(texturas.size(), buffersT);

    for (int a = 0; a < pontos.size(); a++) {

        glBindBuffer(GL_ARRAY_BUFFER, buffers[a]);
        glBufferData(GL_ARRAY_BUFFER, pontos.at(a).size() * sizeof(float), &pontos.at(a).front(), GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, buffersN[a]);
        glBufferData(GL_ARRAY_BUFFER, normais.at(a).size() * sizeof(float), &normais.at(a).front(), GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, buffersT[a]);
        glBufferData(GL_ARRAY_BUFFER, texturas.at(a).size() * sizeof(float), &texturas.at(a).front(), GL_STATIC_DRAW);
    }
}
```

Figura 35 – Função prepareVBO atualizada

Entrando agora no `glutMainLoop`, a primeira alteração que fizemos foi colocar os nossos light sources na render scene.

```
for (int l = 0; l < luzes.size(); l++) {
    glEnable(GL_LIGHT0+l);
    float dark[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
    float diffuse[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    float specular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    float ambient[4] = { 0.0f, 0.0f, 0.0f, 1.0f };

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, dark);//retirar todo a luz ambiente, not sure
    glLightfv(GL_LIGHT0 + l, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0 + l, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0 + l, GL_SPECULAR, specular);

    if (luzes.at(0).tipo == "POINT" || luzes.at(0).tipo == "DIRECTIONAL") {
        glLightfv(GL_LIGHT0 + l, GL_POSITION, luzes.at(0).pos);
    }
    if (luzes.at(0).tipo == "SPOT" ) {
        glLightfv(GL_LIGHT0 + l, GL_SPOT_DIRECTION, luzes.at(0).pos);
        glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
        glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 0.0);
    }

}
```

Figura 36 – glutMainLoop atualizado

A função itera o nosso vector de luzes e tipos diferentes de luz irão ter configurações diferentes.

NOTA: Utilizamos `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, dark)` para retirar toda a luz ambiente, tornando o modelo mais realista.

Finalmente, resta-nos adaptar a nossa função de desenho, `drawVBO()`. É um processo bastante simples: inicialmente, precisamos de dar bind na textura, bem como configurar os materiais:

```
if (planetas.at(p).nomeText != "") {  //so damos bind se t
    glBindTexture(GL_TEXTURE_2D, planetas.at(p).idText);
}

materiais(planetas.at(p));
```

Figura 37 – drawVBO atualizada (1)

De seguida, temos de dar bind ao buffer antes de desenhar, bem como limpar a textura, de modo a que esta não seja usada em modelos seguintes:

```
glBindBuffer(GL_ARRAY_BUFFER, buffersT[location]);
glTexCoordPointer(2, GL_FLOAT, 0, 0);

glDrawArrays(GL_TRIANGLES, 0, pontos.at(location).size() / 3);


glBindTexture(GL_TEXTURE_2D, 0);
```

Figura 38 – drawVBO atualizada (2)

No final obtemos um modelo do sistema solar "realista", iluminado apenas pelo sol. Também obtivemos uma diminuição dos fps de cerca de 20 a 30% (aprox. 3500 para 2700).

Assim, os resultados finais que obtivemos são os seguintes:

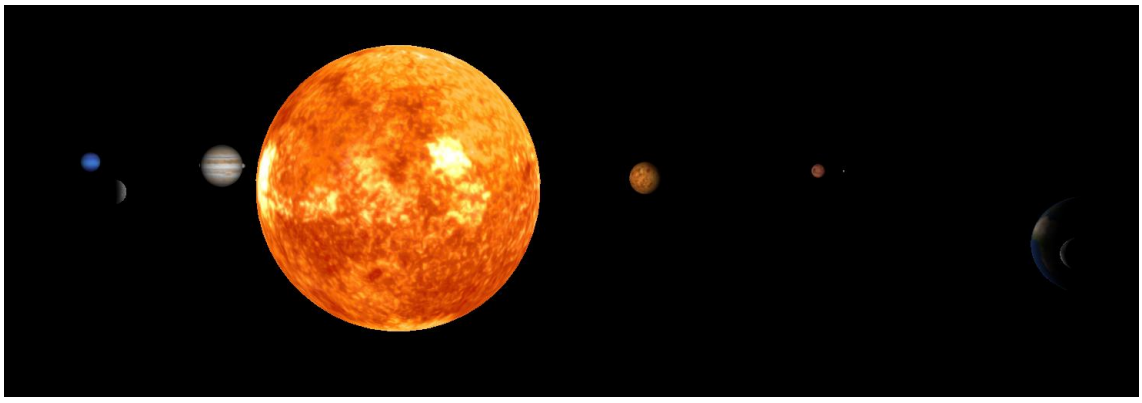

Figura 39 – Vista "horizontal" - No plano XoZ

Figura 40 – Vista do topo do eixo y