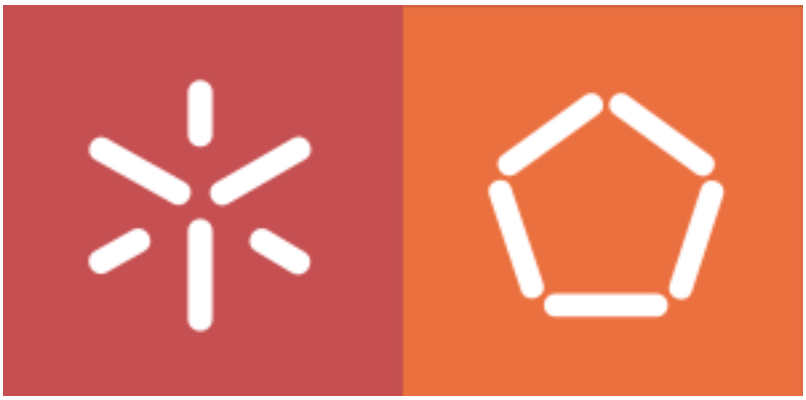


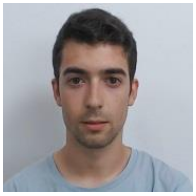
Universidade do Minho

Departamento de Informática

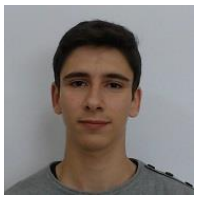
Computação Gráfica

Grupo nº 5

Carlos Preto (a89587)



Pedro Veloso (a89557)



Simão Monteiro (a85489)

Mafalda Costa (a83919)

Índice

Introdução

No âmbito da UC de Computação Gráfica da Universidade do Minho foi proposto o desenvolvimento de uma simulação de um Sistema Solar.

Nesta terceira fase, de modo a melhorar a performance e as funcionalidades da *engine*, bem como criar objetos 3D mais complexos, foi necessário alterar tanto o gerador, como o *engine*.

O gerador passa a poder receber como parâmetros o nome do ficheiro onde se encontram os pontos de controlo de *Bezier*, juntamente com um nível de *tesselação* (*tessellation level*). Com esta informação deve ser produzida um ficheiro com os todos triângulos a serem desenhados.

O ficheiro XML vai sofrer alterações aos seus parâmetros *translate* e *rotate*. O *translate* passa agora a poder receber pontos que vão ser usados como pontos de controlo para a construção das curvas de *Catmull-Rom*, e também um número que irá representar o número de segundos em que a curva será percorrida. O *rotate* recebe também um tempo que representará o tempo, em segundos, que demora uma rotação de 360º sobre um dado eixo. Isto irá servir para realizar animações no sistema solar.

Por fim, a *engine* irá começar a usar *VBO's* para desenhar os triângulos, o que permite uma melhoria da sua performance, para além de modificar a forma como realiza o *parsing* de ficheiros XML, de modo a ler e guardar todos os dados encontrados.

Gerador

Nesta fase acrescentou-se uma nova funcionalidade ao Gerador. Este pode agora receber um ficheiro do formato *patch* e um nível de *tesselação*. Com essa informação deve guardar os novos triângulos a serem desenhados num ficheiro .3d. De maneira a poder realizar tal ação, será necessário ao utilizador escrever no terminal o novo comando:

"*Gerador patch patchFile level saveFile*"

onde ***patchFile*** corresponde ao nome do ficheiro onde se encontram os *patches* e vértices necessários para posteriormente calcular os pontos de *Bezier*. Neste ficheiro encontram-se o número de *patches*, seguido desses mesmos *patches*, sendo que cada *patch* irá ser sempre definido por 16 pontos. Depois dos patches, aparece o número de vértices presentes, seguido desses vértices. O ***level*** corresponde ao nível de *tesselação* pretendido, sendo que quanto maior for este valor, mais detalhe terá a figura obtida.

Por fim, o ***saveFile*** corresponde ao nome do ficheiro .3d, onde se irá guardar o número de vértices a serem desenhados, seguido desses mesmos vértices.

Pontos de *Bezier*

Leitura e Armazenamento Informação

De maneira a poder calcular os pontos de *Bezier*, foi necessário ler e armazenar a informação presente no ficheiro fornecido (com o formato *patch*). Para armazenar a informação decidiu-se utilizar vetores, já definidos na biblioteca *vector*.

A primeira ação a realizar é abrir o ficheiro ***patchFile*** para leitura e, uma vez que o número de *patches* está presente na primeira linha do ficheiro, guarda-se sesse valor numa variável denominada ***numberPatches***. Para armazenar os índices de cada *patch*, criou-se o vector<vector<int>> ***patchesIndex***.

Uma vez que cada patch terá obrigatoriamente 16 pontos, cada vector<int> irá conter 16 índices, e o tamanho de **patchesIndex** será correspondente ao número de *patches* presentes no ficheiro.

Assim, para cada linha do ficheiro que for lida, vai-se percorrendo caracter a caracter, e sempre que se encontrar uma virgula, significa que encontramos um novo índice. Esse índice é adicionado a um vector<int> auxiliar, denominado **values**, que depois é adicionado ao vector **patchesIndex**.

```
fstream patchFile(file);

string numberPatches;
getline(patchFile, numberPatches);

vector<vector<int>> patchesIndex;

for (size_t i = 0; i < stoi(numberPatches); i++) {

    vector<int> values;
    string line;

    if (getline(patchFile, line)) {

        string number = "";
        int patches_lenght_counter = 0;

        for (char word : line) {

            if (word == ',' && patches_lenght_counter != line.size()-1) {
                values.push_back(stoi(number));
                number = "";
            }
            else {
                if (word != ' ' && patches_lenght_counter != line.size()-1) {
                    number = number + word;
                }
                else {
                    if (patches_lenght_counter == line.size()-1) {//ultimo numero nao estava a ser lido
                        number = number + word;
                        values.push_back(stoi(number));
                        number = "";
                    }
                }
            }
            patches_lenght_counter++;
        }
    }
    patchesIndex.push_back(values);
}
```

Figura 1: Leitura de Patches

Depois da leitura dos *patches*, é necessário armazenar os diferentes vértices. Uma vez mais, depois dos *patches*, é fornecido o número de vértices, por isso armazena-se esse valor numa variável, denominada **numberVertices**. Para armazenar cada vértice, criou-se o vector<vector<float>> **vertices.**

Cada vértice é constituído por 3 *float's*, correspondentes aos valores das coordenadas X, Y e Z, respetivamente, por isso, cada vector<float> irá ter 3 valores. O tamanho de **vertices** será correspondente ao número de vértices presentes no ficheiro.

Assim, para cada linha do ficheiro, vai-se lendo caracter a caracter, e sempre que se encontrar uma virgula, significa que encontramos uma nova coordenada. Essa coordenada é adicionada a um vector<float> **auxiliar**, denominado **verticeValue**, que depois é adicionado ao vector **vertices**.

```
string numberVertices;
getline(patchFile, numberVertices);

vector<vector<float>> vertices;

for (size_t j = 0; j < stoi(numberVertices); j++) {

    vector<float> verticeValue;
    string vertices_line;

    if (getline(patchFile, vertices_line)) {

        string point = "";
        int vertices_length_counter = 0;

        for (char word : vertices_line){

            if (word == ',' && vertices_length_counter != vertices_line.size() - 1) {
                verticeValue.push_back(stof(point));
                point = "";
            }
            else {
                if (word != ' ' && vertices_length_counter != vertices_line.size() - 1) {
                    point = point + word;
                }
                else {
                    if (vertices_length_counter == vertices_line.size() - 1) {//ultimo numero nao estava a ser lido
                        point = point + word;
                        verticeValue.push_back(stof(point));
                        point = "";
                    }
                }
            }
        vertices_length_counter++;
        }
    }
    vertices.push_back(verticeValue);
}
```

Figura 2: Leitura dos Vértices

Cálculo Pontos de Bezier

Depois de realizada a fase de leitura e armazenamento dos *patches* e dos vértices, foi então necessário calcular os pontos de *Bezier* para cada *patch*.

Como foi lecionado nas aulas, a expressão para calcular cada ponto de *Bezier* pode ser escrita como:

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

Figura 3: Expressão para calcular cada ponto de Bezier

A matriz M corresponde à matriz de *Bezier*. Esta matriz é simétrica, o que significa que a sua transposta é igual à matriz original.

$$M = M^T = \begin{bmatrix} -1 & -3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

```
for (int c1 = 0; c1 < 4; c1++) {//para cada patch, estamos a formar as matrizes x, y e z corretas
    for (int c2 = 0; c2 < 4; c2++) {
        matrizX[c1][c2] = vertices[patchesIndex[i][c1 * 4 + c2]][0];//x de um vertice num dado indice
        matrizY[c1][c2] = vertices[patchesIndex[i][c1 * 4 + c2]][1];//y de um vertice num dado indice
        matrizZ[c1][c2] = vertices[patchesIndex[i][c1 * 4 + c2]][2];//z de um vertice num dado indice
    }
}
```

Figura 4: Determinar Matrizes X, Y e Z

Cada *patch* tem 16 pontos, e cada um desses pontos tem coordenadas X, Y e Z, logo, para cada *patch*, é necessário calcular 3 matrizes distintas, todas de dimensão 4x4, denominadas **matrizX**, **matrizY** e **matrizZ**. Uma vez que em cada *patch*, cada um dos 16 pontos corresponde a um dado índice, é necessário ir ao vetor **vertices** adquirir o vértice que estará nesse índice.

Uma vez que, para um dado *patch*, as matrizes X, Y e Z não variam, pode-se concluir que o resultado de multiplicar cada uma dessas matrizes pela transposta da matriz de *Bezier* e em seguida multiplicar a matriz *Bezier* pelo resultado da multiplicação anterior, será uma matriz 4x4 constante. Assim, criou-se uma função denominada **multiplicarMatrizes**, que recebe as duas matrizes a multiplicar, e uma outra onde irá guardar o resultado da multiplicação. Posteriormente, é então possível calcular a **matrizConstanteX**, **matrizConstanteY** e a **matrizConstanteZ.**

```
multiplicarMatrizes(matrizX, matrizBezier, matrizBezier_X);
multiplicarMatrizes(matrizBezier, matrizBezier_X, matrizConstanteX);

multiplicarMatrizes(matrizY, matrizBezier, matrizBezier_Y);
multiplicarMatrizes(matrizBezier, matrizBezier_Y, matrizConstanteY);

multiplicarMatrizes(matrizZ, matrizBezier, matrizBezier_Z);
multiplicarMatrizes(matrizBezier, matrizBezier_Z, matrizConstanteZ);
```

Figura 5: Calcular Matrizes Constantes

Depois de calculadas as matrizes constantes, passa-se então para o cálculo dos pontos de Bezier. Definiu-se que:

$$float\ u\ = \frac{1}{level}$$

$$float\ v\ = \frac{1}{level}$$

Quanto maior for o *level*, mais quadrados vão ser desenhados, e por isso maior detalhe irá ter uma dada figura. Assim, para cada *patch*, é necessário saber quantos quadrados vão ser desenhados.

Criou-se dois ciclos que começam em 0 e vão até *level*. O primeiro ciclo vai incrementando v à variável **newV**, e o segundo vai incrementando u à variável **newU**, fazendo com que em cada iteração se tenha uma nova combinação para os diferentes valores de u e v possíveis.

Por último, basta multiplicar as diferentes matrizes constantes pelo vetor v e em seguida pelo vetor u. Para isso criou-se a função **valorPUV**, que recebe os vetores U e V e uma matriz constante, calculando assim o valor da coordenada para um determinado **newV** e **newU**.

Em cada iteração vai ser calculado p(newU, newV), p(newU, newV+v), p(newU+u, newV) e p(newU+u, newV+v), para assim se poder desenhar os diferentes quadrados, cada um constituído por 6 vértices.

```
    for (int w = 0; w < level; w++) {

        float newV = w * v;

        for (int j = 0; j < level; j++) {

            float newU = j * u;

            //calcular p(u,v)
            vetorU[0] = pow(newU, 3);
            vetorU[1] = pow(newU, 2);
            vetorU[2] = newU;
            vetorU[3] = 1;

            vetorV[0] = pow(newV, 3);
            vetorV[1] = pow(newV, 2);
            vetorV[2] = newV;
            vetorV[3] = 1;

            float px0 = valorPUV(vetorU, vetorV, matrizConstanteX);
            float py0 = valorPUV(vetorU, vetorV, matrizConstanteY);
            float pz0 = valorPUV(vetorU, vetorV, matrizConstanteZ);

            //calcular p(u,v+1)
            vetorV[0] = pow(newV + v, 3);
            vetorV[1] = pow(newV + v, 2);
            vetorV[2] = newV + v;

            float px1 = valorPUV(vetorU, vetorV, matrizConstanteX);
            float py1 = valorPUV(vetorU, vetorV, matrizConstanteY);
            float pz1 = valorPUV(vetorU, vetorV, matrizConstanteZ);

            //calcular p(u+1,v)
            vetorU[0] = pow(newU + u, 3);
            vetorU[1] = pow(newU + u, 2);
            vetorU[2] = newU + u;

            vetorV[0] = pow(newV, 3);
            vetorV[1] = pow(newV, 2);
            vetorV[2] = newV;

            float px2 = valorPUV(vetorU, vetorV, matrizConstanteX);
            float py2 = valorPUV(vetorU, vetorV, matrizConstanteY);
            float pz2 = valorPUV(vetorU, vetorV, matrizConstanteZ);

            //calcular p(u+1,v+1)
            vetorU[0] = pow(newU + u, 3);
            vetorU[1] = pow(newU + u, 2);
            vetorU[2] = newU + u;

            vetorV[0] = pow(newV + v, 3);
            vetorV[1] = pow(newV + v, 2);
            vetorV[2] = newV + v;

            float px3 = valorPUV(vetorU, vetorV, matrizConstanteX);
            float py3 = valorPUV(vetorU, vetorV, matrizConstanteY);
            float pz3 = valorPUV(vetorU, vetorV, matrizConstanteZ);

            write << px0 << " " << py0 << " " << pz0 << endl;
            write << px1 << " " << py1 << " " << pz1 << endl;
            write << px3 << " " << py3 << " " << pz3 << endl;

            write << px0 << " " << py0 << " " << pz0 << endl;
            write << px3 << " " << py3 << " " << pz3 << endl;
            write << px2 << " " << py2 << " " << pz2 << endl;
        }
    }
}
```

Figura 6: Calcular e escrever pontos Bezier

Engine

Como já foi dito anteriormente, nesta fase é necessário estender os *translate* e os *rotate*, uma vez que passa a ser possível realizar translações e rotações animadas. Para que as animações sejam possíveis teve-se então que desenvolver algum código para, por exemplo, construir curvas de *Catmull-Rom*. A construção destas curvas, levou à necessidade de realizar alterações, na estrutura de dados já desenvolvida e na forma em que se realiza o *parsing* dos dados, para que fosse possível passar e guardar os pontos de controlo contidos no ficheiro XML. Dada estas alterações teve-se também de realizar alterações no ficheiro XML.

Para o cálculo dos pontos de controlo, integrados no ficheiro XML desenvolvido, usamos as seguintes fórmulas:

$$X = cos(angulo) * raio \text{ e } Z = sin(angulo) * raio$$

Na fórmula acima apresentada, o raio é a distância entre o sol e planeta ou entre o planeta e a sua lua, enquanto o angulo é apenas um valor qualquer entre 0º e 360º para que se pudesse obter vários pontos de uma circunferência, neste caso, órbita do planeta/lua.

Para calcular o tempo de translação/rotação dos planetas pesquisou-se pelos valores reais na internet e tentou-se manter a proporcionalidade entre eles.

Parsing do XML

Na realização do *parsing* do ficheiro XML desta fase, o raciocínio manteve-se o mesmo da fase anterior, de modo a garantir a compatibilidade com os modelos estáticos já desenvolvidos, no entanto foi necessário fazer pequenas alterações nas classes desenvolvidas. Na classe "OperacoesGeometricas", para facilitar o registo da informação útil à criação de curvas de *Catmull-Rom* (pontos e controlo, tempo e eixo dos y da normal do objeto), criaram-se 3 vetores, um para cada eixo de coordenadas, onde são guardados os valores das coordenadas dos pontos de controlo, para além de três variáveis usadas para definir o eixo do y, do vetor normal, durante o cálculo da curva.

```
class OperacoesGeometricas {
public:

    char tipo;

    float angle;

    float x;
    float y;
    float z;

    float time;

    vector<float> p1;
    vector<float> p2;
    vector<float> p3;

    float yy1 = 0;
    float yy2 = 1;
    float yy3 = 0;
```

Figura 7: Classe OperacoesGeometricas

Observando a figura acima, visualiza-se o conteúdo da classe "OperacoesGeometricas", o que evidencia a existência dos vetores acrescentados, para esta fase, e das variáveis usadas para o eixo do y do vetor normal. Contudo verifica-se também a existência de uma variável denominada por "time" que corresponde ao tempo total que um objeto deve demorar a percorrer a curva na totalidade ou a realizar uma rotação de 360 graus sobre um determinado eixo.

Translação de um objeto

Uma vez que uma translação de um objeto é realizada através do cálculo de curvas de *Catmull-Rom*, é necessário a existência de pelo menos 4 pontos de controlo usados para definir uma curva. Caso existam mais de 4 pontos é necessário encontrar os 4 pontos que vão desenhar o segmento da curva para o instante de tempo atual.

```
int POINT_COUNT = (int) p1.size();

float tempo = glutGet(GLUT_ELAPSED_TIME) % (int)(time * 1000);
float a = tempo / (time * 1000);
float t = a * POINT_COUNT;
int index = floor(t);
t = t - index;

// indices store the points
int indices[4];
indices[0] = (index + POINT_COUNT - 1) % POINT_COUNT;
indices[1] = (indices[0] + 1) % POINT_COUNT;
indices[2] = (indices[1] + 1) % POINT_COUNT;
indices[3] = (indices[2] + 1) % POINT_COUNT;
```

Figura 8: Cálculo dos índices dos 4 pontos de controlo

Tal como se observa na figura acima, de modo a descobrir os 4 pontos de controlo a usar é necessário determinar o instante de tempo atual. Contudo uma vez que o tempo obtido através da função $glutGet(GLUT\_ELAPSED\_TIME)$ é ilimitado (a função dá o tempo, em milissegundos, desde o início da aplicação), é necessário pô-lo a variar entre 0 e tempo máximo de uma volta completa. De seguida é necessário, encontrar a porção da curva, que pretendemos obter no momento atual. Por fim basta obter os índices dos pontos pretendidos, baseado na informação obtida.

Tendo os quatro pontos de controlo obtidos e o tempo atual da curva, basta calcular os pontos atuais, bem como a sua derivada para todas as coordenadas, seguindo as seguintes formulas:

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$P'(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Figura 9: Fórmulas usadas para calcular os pontos e as suas derivadas

Por fim, com os pontos obtidos, na fórmula anterior, calcula-se os novos eixos normais do objeto, da seguinte forma:

```
getAtualCatmullRomPoint(it->time, it->p1, it->p2, it->p3, pos, xx);

float zz[3];
float yy[3];

yy[0] = it->yy1;
yy[1] = it->yy2;
yy[2] = it->yy3;

cross(xx, yy, zz);
cross(zz, xx, yy);

normalize(yy);
normalize(xx);
normalize(zz);
```

Figura 10: Cálculo do vetor normal

Como se observa na figura, o eixo dos do x para a normal é dado pelo cálculo da derivada, apresentada anteriormente, enquanto o eixo dos y é calculado pelo produto do vetor usado no eixo do z (da iteração atual) com o vetor usado para o eixo do x. O eixo do z é obtido pelo produto do vetor do eixo dos x com o vetor do eixo dos y da iteração anterior. Como é necessário utilizar o vetor do eixo y da iteração anterior, foi necessário registá-lo em memória, como referiu-se anteriormente.

Rotação de um objeto

Para que a rotação animada fosse possível apenas se teve que ter em conta o tempo que se recebe do ficheiro XML e ir atualizando os valores à medida que o tempo passa. Para isto fez-se uso da função $glutGet(GLUT\_ELAPSED\_TIME)$.

```
case 'R': {
    if (it->time == 0) {
        glRotatef(it->angle, it->x, it->y, it->z);
    }
    else {
        int t = glutGet(GLUT_ELAPSED_TIME);
        float angle = 360 / (it->time * 1000);
        glRotatef(t * angle, it->x, it->y, it->z);
    }
    break;
}
```

Figura 11 - Código para realizar a rotação de um planeta

Observando a figura acima, percebe-se os passos realizados para obter o ângulo de rotação a utilizar no momento atual. O ângulo a considerar é obtido pela divisão do ângulo giro (360º) a dividir pelo tempo que demora a realizar uma volta completa (em milissegundos) multiplicado pelo tempo atual de execução, dado pela função já referenciada.

Utilização dos *VBO's*

Uma vez que o uso de *VBO's* permite o aumento exponencial da performance do programa, nesta terceira fase do trabalho, realizou-se alterações na *engine* de modo a permitir o uso de tal ferramenta.

A principal alteração à *engine*, passa pela forma me que se regista a informação dos triângulos em memória. Assim, optou-se por criar duas novas classes, descritas na figura seguinte, para registar a informação dos pontos já lidos na memória da placa gráfica.

```
class ModeloVBO {
public:

    map<Cor, GLuint> triangulos;
    map<Cor, int> tam;

    ModeloVBO() {
        this->triangulos = map<Cor, GLuint>();
    }

    ModeloVBO(map<Cor, GLuint> triangulos, map<Cor, int> tam) {
        this->triangulos = triangulos;
        this->tam = tam;
    }
};
```

```
class GroupVBO {
public:

    list<OperacoesGeometricas> operacoes;

    list<ModeloVBO> modelos;

    list<GroupVBO> subGrupos;

    GroupVBO() {    }

    GroupVBO(list<ModeloVBO> m, list<GroupVBO> g, list<OperacoesGeometricas> ops) {

        this->operacoes = ops;

        this->modelos = m;

        this->subGrupos = g;
    }
};
```

Figura 12: Classes criadas para o uso de VBO's

Tal como se observa com as figuras acima, as novas classes seguem o mesmo raciocínio que as anteriormente desenvolvidas, contudo guarda-se os pontos num *GLuint* (na placa gráfica) em vez de numa lista de triângulos. Optou-se por acrescentar estas classes, em vez de alterar as já desenvolvidas, de modo a preservar a possibilidade de representar os pontos em memória central, possibilitando a troca de abordagem no desenho dos triângulos.

Uma vez que os pontos agora estão registados em memória da placa gráfica foi necessário modificar a forma como os mesmos são desenhados em cada *frame*.

```
for (list<ModeloVBO>::iterator it = g.modelos.begin(); it != g.modelos.end(); it++) {
    int j = 1;
    for (map<Cor, GLuint>::iterator it2 = it->triangulos.begin(); it2 != it->triangulos.end(); it2++) {
        Cor c(it2->first.r, it2->first.g, it2->first.b);
        glColor3f(c.r, c.g, c.b);
        int t = (it->tam)[c];
        glBindBuffer(GL_ARRAY_BUFFER, it2->second);
        glVertexPointer(3, GL_FLOAT, 0, 0);
        glDrawArrays(GL_TRIANGLES, 0, t);
    }
}
```

Figura 13: Desenho dos triângulos com VBO's

Ao observar a figura acima é possível verificar, que a forma de navegar pela estrutura de dados e obter os pontos é semelhante à desenvolvida nas fases anteriores. Contudo nesta fase, uma vez que a função "glDrawArrays" responsável por redesenhar os pontos no ecrã, necessita do número de total de pontos a desenhar, a partir de um outro ponto (0 se for desde o início do *array*), é necessário registar também essa informação em memória, para em cada iteração ter tal informação (este procedimento pode ser verificado no uso da variável inteira **t**).

Conclusão

Como forma de conclusão, consideramos que esta terceira fase do projeto foi a mais desafiante e trabalhosa até ao momento, o que, pelo lado mais positivo, nos permitiu aumentar os nossos conhecimentos em Computação Gráfica. Apesar do desafio, o grupo considera ter conseguido atingir todos os objetivos propostos para a mesma.

Numa fase inicial, houve algumas dificuldades em aplicar, na prática, o que foi lecionado nas aulas teóricas de CG, porém, após alguma persistência, obteve-se os resultados esperados. Houve também algumas dúvidas em aplicar as animações pretendidas, no entanto, com entreajuda por parte dos elementos do grupo conseguimos obter o resultado esperado, com sucesso.

Bibliografia

- TinyXML-2: *www.github.com/leethomason/tinyxml2/releases/tag/8.0.0*
- http://www1.ci.uc.pt/iguc/