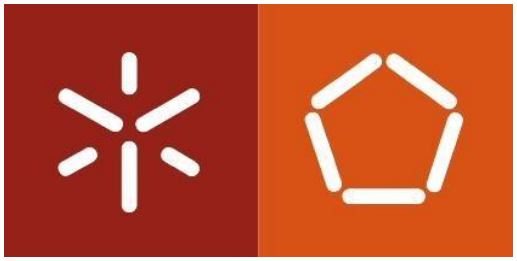
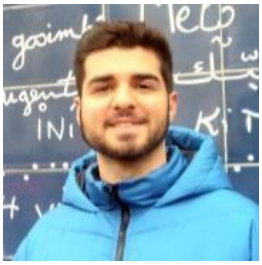


Computação Gráfica - CG 2019/2020

FASE 3

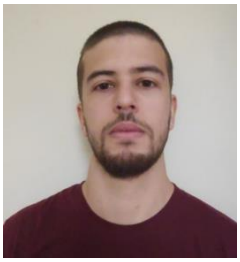
Mestrado Integrado em Engenharia Informática

José Pinto
A84590



Eduardo Costa
A85735

Luís Lopes
A85367



Ricardo Carvalho
A84261

Índice

Introdução

No âmbito da Unidade Curricular de Computação Gráfica, foi-nos proposto o desenvolvimento de um mecanismo 3D baseado num cenário gráfico com o auxílio de duas ferramentas utilizadas, também, nas aulas práticas, sendo elas *OpenGL* e a linguagem de programação C++, e que já vinha sendo desenvolvido nos dois trabalhos práticos anteriores.

Desta forma, nesta terceira fase pretendia-se, no que à *engine* diz respeito, uma evolução das transformações geométricas previamente implementadas, nomeadamente as translações e rotações, tornando-as, sempre que desejado, dinâmicas. As translações dinâmicas serão calculadas através de curvas *de Catmull-Rom* associadas a um tempo, enquanto que as rotações apenas dependem de um tempo, que descreve a duração de uma rotação completa de 360º. Quanto ao *generator*, o objetivo para esta fase focou-se no desenvolvimento de uma nova funcionalidade: a capacidade de criação de um novo tipo de modelo baseado em *Bezier patches*, que lhe permite, a partir de um ficheiro com uma estrutura definida (.*patch*), criar um modelo composto por triângulos que permita desenhar a superfície pretendida.

Para tal, foram necessárias diversas alterações em várias vertentes do projeto em relação às tarefas da 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, foi também aprimorada a anterior representação do sistema solar, tornando-a dinâmica, recorrendo às novas funcionalidades.

Alterações no *generator*

Em relação ao *generator,* nesta fase, adicionamos uma nova função requerida no enunciado, *Bezier*, que lê um ficheiro do tipo .*patch*, transformando a informação que nele se encontra, escrevendo-a no ficheiro de destino (último argumento passado ao *generator*).

Um *Patch* de *Bezier* consiste:

- Na primeira linha, que corresponde ao número total de *patches*.
- Nas restantes *n* linhas, que correspondem aos *patches*.
- Após as *n* linhas dos *patches*, num inteiro que representa o número total de pontos de controlo.
- Por fim, todas as restantes linhas representam os pontos de controlo.

Quanto à nossa função *Bezier,* podemos dizer que se encontra dividida em duas partes, uma parte para ler o *patch*, e outra para aplicar os cálculos das curvas de *bezier* e escrever no ficheiro.

Figura 1 – Função bezier

- ***readBezier***

A leitura do *patch* é realizada no início da função *Bezier*, chamando a função *readBezier* e passando-lhe como argumento apenas o nome do ficheiro do *Patch* (*patchfilename)*, sendo que nesta é aberto o *patch* que é recebido como argumento na função.

Em primeiro lugar, guardamos o número de *patches*, seguido de todos os índices dos pontos correspondentes a cada *patch*. Os índices de cada patch são guardados numa

estrutura do tipo *vector* que contem vetores de inteiros. De seguida guardamos os pontos de controlo, cada um numa célula da estrutura do tipo *vector*, formada por pontos.

```
void readBezier(std::string patchfilename) {
    std::fstream f;
    f.open(patchfilename, std::ios::in);
    if (f.is_open()) {
        ////////////////////// le numero de patches
        std::string line;
        if (getline(f, line)) sscanf(line.c_str(), "%d\n", &patches);
        for (int i = 0; i < patches; i++) {
            std::vector<int> iPatch;
            std::getline(f, line);
            std::istringstream str1(line);
            std::string str2;
            ////////////////////// patches
            while (std::getline(str1, str2, ',')) {
                /////////////// adiciona indices dos patchs
                iPatch.push_back(atoi(str2.c_str()));

            }

            indicesPatch.push_back(iPatch);
        }
        ////////////////////// cria os pontos e coloca no vetor points
        if (getline(f, line)) sscanf(line.c_str(), "%d\n", &vertices);
        for (int i = 0; i < vertices; i++) {
            float x = 0, y = 0, z = 0;
            if (getline(f, line)) sscanf(line.c_str(), "%f, %f, %f\n", &x, &y, &z);

            Point p;
            p.x = x; p.y = y; p.z = z;

            points.push_back(p);
        }

        f.close();
    }
    else { printf("Error: Not possible acess patch file..."); exit(0); }

}
```

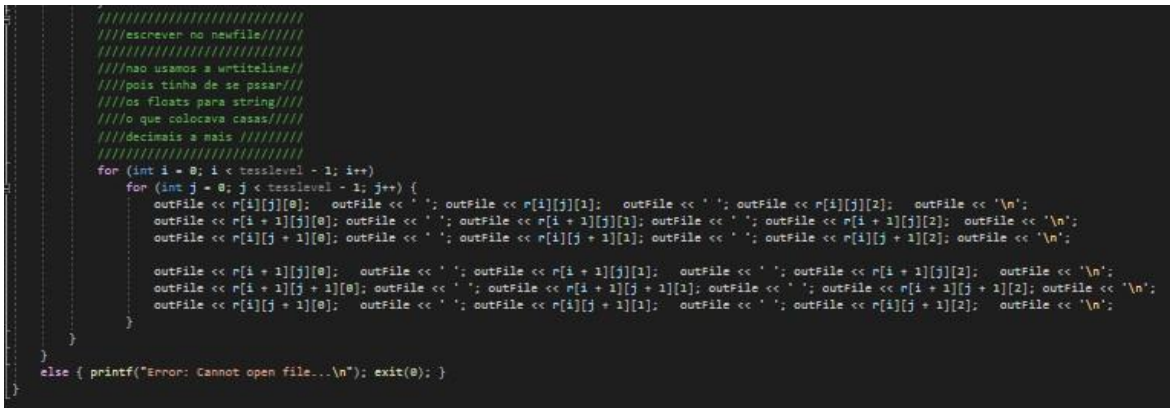
Figura 2 – Função readBezier

- **writeBezier**

Após concluir a leitura do *patch*, é feito o cálculo do número de pontos, segundo o algoritmo de *Bezier*, que vai definir as quatro curvas de *Bezier*, cada uma constituída por quatro vértices. A informação gerada vai ser guardada no ficheiro recebido como último argumento da *main*.

Quanto à função *writeBezier* propriamente dita, é guardado o número total de pontos elaborados e de seguida a lista de coordenadas de cada ponto. De seguida, temos o código da *writeBezier.*

```
if (open(newfile)) {
    ////////////////////////////
    ///escrever num de pontos////
    ////////////////////////////

    outFile << pontos; outFile << '\n';
    for (int n = 0; n < patches; n++) {
        ////////////////////////////////////////////////////
        /////recolher os vértices de points para x y z //////
        ////////////////////////////////////////////////////
        float p[16][3];
        for (int m = 0; m < 16; m++) {
            p[m][0] = points[indicesPatch[n][m]].x;
            p[m][1] = points[indicesPatch[n][m]].y;
            p[m][2] = points[indicesPatch[n][m]].z;
        }
        int j = 0, k = 0;
        ////////////////////////////////
////////////////////////////
///desenahr 4 curvas de Bezier///
////////////////////////////
for (int i = 0; i < 15; i += 4) {
    indices[0] = i;
    indices[1] = i + 1;
    indices[2] = i + 2;
    indices[3] = i + 3;
    ////////////////////////////////////////////////////////////
    ////////////calcular a curva/////////////////////////////////
    ///  [-t^3 + 3t^2 - 3t,3t^3-6t^2+3t, -3t^3+3t^2,t^3][P0] ////////////
    ////////////////////////////////////////////////////// [P1] ////////////
    ////////////////////////////////////////////////////// [P2] ////////////
    ////////////////////////////////////////////// [P3] ////////////
    ////////////////////////////////////////////////////////////
    for (int a = 0; a < tesslevel - 1; a++) {
        t = a * tess;
        index = floor(t);
        t = t - index;
        res[0] = (-p[indices[0]][0] + 3 * p[indices[1]][0] - 3 * p[indices[2]][0] + p[indices[3]][0]) * t * t * t
            + (3 * p[indices[0]][0] - 6 * p[indices[1]][0] + 3 * p[indices[2]][0]) * t * t + (-3 * p[indices[0]][0] + 3 * p[indices[1]][0]) * t + p[indices[0]][0];
        res[1] = (-p[indices[0]][1] + 3 * p[indices[1]][1] - 3 * p[indices[2]][1] + p[indices[3]][1]) * t * t * t
            + (3 * p[indices[0]][1] - 6 * p[indices[1]][1] + 3 * p[indices[2]][1]) * t * t + (-3 * p[indices[0]][1] + 3 * p[indices[1]][1]) * t + p[indices[0]][1];
        res[2] = (-p[indices[0]][2] + 3 * p[indices[1]][2] - 3 * p[indices[2]][2] + p[indices[3]][2]) * t * t * t
            + (3 * p[indices[0]][2] - 6 * p[indices[1]][2] + 3 * p[indices[2]][2]) * t * t + (-3 * p[indices[0]][2] + 3 * p[indices[1]][2]) * t + p[indices[0]][2];
        q[j][k][0] = res[0];
        q[j][k][1] = res[1];
        q[j][k][2] = res[2];
        k++;
    }

    t = 1;

    res[0] = (-p[indices[0]][0] + 3 * p[indices[1]][0] - 3 * p[indices[2]][0] + p[indices[3]][0]) * t * t * t
        + (3 * p[indices[0]][0] - 6 * p[indices[1]][0] + 3 * p[indices[2]][0]) * t * t + (-3 * p[indices[0]][0] + 3 * p[indices[1]][0]) * t + p[indices[0]][0];
    res[1] = (-p[indices[0]][1] + 3 * p[indices[1]][1] - 3 * p[indices[2]][1] + p[indices[3]][1]) * t * t * t
        + (3 * p[indices[0]][1] - 6 * p[indices[1]][1] + 3 * p[indices[2]][1]) * t * t + (-3 * p[indices[0]][1] + 3 * p[indices[1]][1]) * t + p[indices[0]][1];
    res[2] = (-p[indices[0]][2] + 3 * p[indices[1]][2] - 3 * p[indices[2]][2] + p[indices[3]][2]) * t * t * t
        + (3 * p[indices[0]][2] - 6 * p[indices[1]][2] + 3 * p[indices[2]][2]) * t * t + (-3 * p[indices[0]][2] + 3 * p[indices[1]][2]) * t + p[indices[0]][2];

    q[j][k][0] = res[0];
    q[j][k][1] = res[1];
    q[j][k][2] = res[2];
    j++;
    k = 0;
}

for (int j = 0; j < tesslevel; j++) {
    for (int a = 0; a < tesslevel - 1; a++) {
        t = a * tess;;
        index = floor(t);
        t = t - index;

        res[0] = (-q[0][j][0] + 3 * q[1][j][0] - 3 * q[2][j][0] + q[3][j][0]) * t * t * t
            + (3 * q[0][j][0] - 6 * q[1][j][0] + 3 * q[2][j][0]) * t * t + (-3 * q[0][j][0] + 3 * q[1][j][0]) * t + q[0][j][0];
        res[1] = (-q[0][j][1] + 3 * q[1][j][1] - 3 * q[2][j][1] + q[3][j][1]) * t * t * t
            + (3 * q[0][j][1] - 6 * q[1][j][1] + 3 * q[2][j][1]) * t * t + (-3 * q[0][j][1] + 3 * q[1][j][1]) * t + q[0][j][1];
        res[2] = (-q[0][j][2] + 3 * q[1][j][2] - 3 * q[2][j][2] + q[3][j][2]) * t * t * t
            + (3 * q[0][j][2] - 6 * q[1][j][2] + 3 * q[2][j][2]) * t * t + (-3 * q[0][j][2] + 3 * q[1][j][2]) * t + q[0][j][2];
        r[j][k][0] = res[0];
        r[j][k][1] = res[1];
        r[j][k][2] = res[2];
        k++;
    }

    t = 1;

    res[0] = (-q[0][j][0] + 3 * q[1][j][0] - 3 * q[2][j][0] + q[3][j][0]) * t * t * t
        + (3 * q[0][j][0] - 6 * q[1][j][0] + 3 * q[2][j][0]) * t * t + (-3 * q[0][j][0] + 3 * q[1][j][0]) * t + q[0][j][0];
    res[1] = (-q[0][j][1] + 3 * q[1][j][1] - 3 * q[2][j][1] + q[3][j][1]) * t * t * t
        + (3 * q[0][j][1] - 6 * q[1][j][1] + 3 * q[2][j][1]) * t * t + (-3 * q[0][j][1] + 3 * q[1][j][1]) * t + q[0][j][1];
    res[2] = (-q[0][j][2] + 3 * q[1][j][2] - 3 * q[2][j][2] + q[3][j][2]) * t * t * t
        + (3 * q[0][j][2] - 6 * q[1][j][2] + 3 * q[2][j][2]) * t * t + (-3 * q[0][j][2] + 3 * q[1][j][2]) * t + q[0][j][2];

    r[j][k][0] = res[0];
    r[j][k][1] = res[1];
    r[j][k][2] = res[2];
    k = 0;
}
```



```
        ////////////////////////////
        ////escrever no newfile//////
        ////////////////////////////
        ////nao usamos a wrtiteline//
        ////pois tinha de se pssar///
        ////os floats para string////
        ////o que colocava casas/////
        ////decimais a mais /////////
        ////////////////////////////
        for (int i = 0; i < tesslevel - 1; i++)
            for (int j = 0; j < tesslevel - 1; j++) {
                outFile << r[i][j][0];   outFile << ' '; outFile << r[i][j][1];   outFile << ' '; outFile << r[i][j][2];   outFile << '\n';
                outFile << r[i + 1][j][0]; outFile << ' '; outFile << r[i + 1][j][1]; outFile << ' '; outFile << r[i + 1][j][2];  outFile << '\n';
                outFile << r[i][j + 1][0]; outFile << ' '; outFile << r[i][j + 1][1]; outFile << ' '; outFile << r[i][j + 1][2]; outFile << '\n';

                outFile << r[i + 1][j][0];   outFile << ' '; outFile << r[i + 1][j][1];   outFile << ' '; outFile << r[i + 1][j][2];   outFile << '\n';
                outFile << r[i + 1][j + 1][0]; outFile << ' '; outFile << r[i + 1][j + 1][1]; outFile << ' '; outFile << r[i + 1][j + 1][2]; outFile << '\n';
                outFile << r[i][j + 1][0];   outFile << ' '; outFile << r[i][j + 1][1];   outFile << ' '; outFile << r[i][j + 1][2];   outFile << '\n';
            }
    }
    else { printf("Error: Cannot open file...\n"); exit(0); }
}
```

Figura 3 – Função writeBezier (por partes)

Alterações das estruturas de dados

Em relação à *engine*, a nossa abordagem começou pelas alterações às estruturas de dados e, simultaneamente, ao *parser* XML, que serão abordadas na secção seguinte. Estas alterações foram, de facto, mínimas, mas cruciais para que as transformações dinâmicas (*translate* e *rotate*) fossem possíveis. Para além da criação de uma nova estrutura de dados, as alterações que necessitamos aplicar nesta fase às já existentes apenas afetaram a *Oper* e *OperAux*.

- ***Transform***

A estrutura que necessitamos criar nesta fase, diz respeito às transformações dinâmicas a implementar. Nestes casos, os ficheiros XML podem, agora, conter referência a um tempo de translação ou rotação, e, no caso das translações, um conjunto de pontos da curva a elas referentes, armazenados num *vector* de Ponto(s), estrutura esta já previamente criada, aquando da realização da primeira fase do projeto.

```
typedef struct transform {
    double time;
    vector<Ponto> points;
} Transform;
```

Figura 4 – Nova estrutura Transform

- ***Oper***

A criação da estrutura referida acima implicou alterações na estrutura *Oper*, de forma a que esta incluí-se, caso fosse necessário, referência a transformações dinâmicas, passando a conter um apontador para uma estrutura *Transform*. De modo a ser possível continuar a realizar transformações estáticas, este apontador é inicializado com valor *NULL,* sendo sempre possível verificar que tipo de transformação queremos realizar.

```
typedef struct oper {
    char* operation;
    double x, y, z, angle;
    Transform* transform;
} Oper;
```

Figura 5 – Alterações na estrutura Oper.

- ***OperAux***

À semelhança do caso anterior, nesta estrutura de dados auxiliar ao *parser,* foi necessário acrescentar um apontador para uma estrutura *Transform*. Do mesmo modo que era inicializado a NULL na situação anterior, neste caso é usado o mesmo método.

```
typedef struct operationAux {
    char* operation;
    double x, y, z, angle;
    Transform* transform;
    struct operationAux* next;
} OperAux;
```

Figura 6 – Alterações na estrutura OperAux

Alterações do *parser XML*

Ainda em relação à *engine*, a nossa abordagem, de seguida, passou por adaptar o *parser* XML àquilo que era pedido no enunciado. Estas alterações foram, de facto, mínimas, mas cruciais para que as transformações dinâmicas fossem possíveis. Assim sendo, na leitura/*parsing* do ficheiro XML recebido como argumento, tivemos que considerar, agora, que algumas operações seriam dinâmicas. Para o efeito, sempre que surge um *translate* ou um *rotate*, verificamos agora se esse será dinâmico ou não.

Tanto no caso dos *translates*, como no caso da operação *rotate,* verificamos agora a existência do atributo "*time*", pois este fará a distinção entre serem estáticas ou dinâmicas.

Para a operação "translate", se esta for estática, funciona da mesma forma que na fase anterior, se for dinâmica, terá o referido atributo "*time*", da forma: <translate time="x">, que representa o número de segundos que um certo objeto demorará a dar uma volta completa à curva gerada posteriormente. Neste último caso, é construída a estrutura *Transform* com os devidos valores de tempo e os pontos a considerar.

```
if (strcmp((char*)child2->Value(), "translate") == 0) {

    char operation[15];
    strcpy(operation, (char*)child2->Value());

    double time = 0;
    double x = 0;
    double y = 0;
    double z = 0;
    vector<Ponto> points;

    char* timeStr;
    if ((timeStr = (char*)child2->Attribute("time"))) {
        time = atof(timeStr);

        for (const XMLElement* child3 = child2->FirstChildElement(); child3; child3 = child3->NextSiblingElement()) {

            char* pointX = (char*)child3->Attribute("X");
            if (pointX != NULL) x = atof(pointX);
            char* pointY = (char*)child3->Attribute("Y");
            if (pointY != NULL) y = atof(pointY);
            char* pointZ = (char*)child3->Attribute("Z");
            if (pointZ != NULL) z = atof(pointZ);

            addPoint(x, y, z, points);
        }

        transform = new Transform();
        transform->time = time;
        transform->points = points;

    }
    else
    {
        char* translateX = (char*)child2->Attribute("X");
        if (translateX != NULL) x = atof(translateX);
        char* translateY = (char*)child2->Attribute("Y");
        if (translateY != NULL) y = atof(translateY);
        char* translateZ = (char*)child2->Attribute("Z");
        if (translateZ != NULL) z = atof(translateZ);
    }

    addOpGroup(group, i, operation, x, y, z, -1, transform);

}
```

Figura 7 – Alterações no parser de translações

Já no caso das rotações, o atributo estático *angle* pode agora ser substituído pelo atributo dinâmico "*time*", que representa o número de segundos que um certo objeto demora a fazer uma rotação completa de 360º em torno do seu próprio eixo. Neste último caso, é criado um *vector* de pontos vazio, por não ser necessário, e recolhido o período de rotação, colocando-os numa estrutura *Transform*.

```
if (strcmp((char*)child2->Value(), "rotate") == 0) {

    char operation[15];
    strcpy(operation, (char*)child2->Value());

    double time = 0;
    double angleValue = 0;
    double x = 0;
    double y = 0;
    double z = 0;

    char* angle = (char*)child2->Attribute("angle");
    if (angle != NULL) angleValue = atof(angle);
    char* axisX = (char*)child2->Attribute("axisX");
    if (axisX != NULL) x = atof(axisX);
    char* axisY = (char*)child2->Attribute("axisY");
    if (axisY != NULL) y = atof(axisY);
    char* axisZ = (char*)child2->Attribute("axisZ");
    if (axisZ != NULL) z = atof(axisZ);

    char* timeStr;
    char* rotateAngle;
    if ((timeStr = (char*)child2->Attribute("time"))) {
        time = atoi(timeStr);
        vector<Ponto> points;

        transform = new Transform();
        transform->time = time;
        transform->points = points;
    }
    else if ((rotateAngle = (char*)child2->Attribute("angle"))) {
        angleValue = atof(rotateAngle);
    }

    addOpGroup(group, i, operation, x, y, z, angleValue, transform);
}
```

Figura 8 – Alterações no parser de rotações

Implementação dos *VBOs*

Um dos objetivos desta fase foi implementar *VBOs,* de forma a desenhar diferentes modelos. A sua utilização permite uma melhor performance do programa, visto que os pontos deixam de ser desenhados um a um e passam a estar guardados num *buffer*.

A nossa solução para esta implementação passou pela criação de uma função *createVBO*, que é chamada por uma outra função que se encontra na *main*. A função *createVBO* passa a informação de todos os pontos lidos para uma matriz *vertexB* (assinalada abaixo), que é usada para a transferência de informação para um *Glint buffer*.

```
void createVBO(int i) {
    int p = 0;  int vertex = 0;

    vector<Ponto>::iterator tri = triangles.begin();
    vector<OperFile*>::iterator itFile;

    vertexB[i] = (double*)malloc(sizeof(double) * triangles.size() * 3);

    while (tri != triangles.end()) {

        Ponto aux_1 = *tri; tri++;
        Ponto aux_2 = *tri; tri++;
        Ponto aux_3 = *tri; tri++;

        vertexB[i][p] = aux_1.x;
        vertexB[i][p + 1] = aux_1.y;
        vertexB[i][p + 2] = aux_1.z;
        vertex++;

        vertexB[i][p + 3] = aux_2.x;
        vertexB[i][p + 4] = aux_2.y;
        vertexB[i][p + 5] = aux_2.z;
        vertex++;

        vertexB[i][p + 6] = aux_3.x;
        vertexB[i][p + 7] = aux_3.y;
        vertexB[i][p + 8] = aux_3.z;
        vertex++;

        p += 9;
    }
    vertexCount = vertex;
}
```

Figura 9 – Função createVBO

Optamos por usar um VBO para cada ficheiro, de forma a realizar as transformações geométricas corretamente. Na figura abaixo exemplifica-se a forma como se desenham os pontos que colocamos em *vertexB*.

```
glColor3f(1, 1, 1);
glBindBuffer(GL_ARRAY_BUFFER, buffers[i]);
glBufferData(GL_ARRAY_BUFFER, vertexCount * sizeof(double) * 3, vertexB[i], GL_STATIC_DRAW);
glVertexPointer(3, GL_DOUBLE, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, vertexCount);
```

Figura 10 – Desenhar pontos a partir de um VBO

Novas funcionalidades da *Engine*

Como já foi referido até aqui, nesta terceira fase do projeto, foram introduzidas as transformações dinâmicas. Todas as alterações feitas ao *parser* e às estruturas tiveram como propósito possibilitar a implementação dessas transformações. Desta forma, aquando da análise das operações a realizar para cada ficheiro, passamos a considerar a possibilidade de estas serem dinâmicas, tanto para o caso das rotações como para o caso das translações.

```
while(it2 != op->operations.end())
{
    Oper* oper = *it2;
    it2++;
    if (strcmp(oper->operation, "translate")==0)
    {
        if (oper->transform == nullptr) {
            glTranslatef(oper->x, oper->y, oper->z);
        }
        else {
            dynamicTranslate(oper, i);
        }
    }
    if (strcmp(oper->operation, "rotate") == 0)
    {
        if (oper->transform == nullptr) {
            glRotatef(oper->angle, oper->x, oper->y, oper->z);
        }
        else {
            dynamicRotate(oper, i);
        }
    }
    if (strcmp(oper->operation, "scale") == 0) {
        glScalef(oper->x, oper->y, oper->z) ;
    }
}
```

Figura 11 – Alterações à função desenhar

Para as transformações normais, o processo mantém-se em relação à fase anterior, enquanto que para as transformações dinâmicas a abordagem difere. Para tal, foram criadas duas funções – *dynamicTranslate* e *dynamicRotate* – que tratam dos cálculos necessários para a aplicação de translações e rotações dinâmicas, respetivamente.

Quanto às rotações, a abordagem foi muito simples e direta, passando apenas pelo cálculo dos ângulos em função do tempo, aplicando, em cada instante, a rotação desejada, como demonstrado na figura abaixo.

```
void dynamicRotate(Oper* oper, int i)
{
    Transform* t = oper->transform;

    double angles[20];
    double time = t->time;

    angles[i] += 360 / (time * 1000);
    glRotated(angles[i], oper->x, oper->y, oper->y);
}
```

Figura 12 – Função dynamicRotate

Por sua vez, o caso das translações, por envolver cálculos de curvas de Catmull-Rom, necessitou de mais atenção aos pormenores e revelou-se mais complexo. Tal como na resolução do Guião 8, criamos uma matriz (*curve*) que armazena os pontos da curva, presentes, nesta altura, na estrutura previamente construída. A partir daqui, através da função getGlobalCatmullRomPoint, que por sua vez utiliza a função getCatmullRomPoint, calculamos os pontos da curva para cada instante, aplicando, de seguida a translação, como assinalado na figura. Todos os cálculos de matrizes aplicados foram, em tudo, semelhantes aos realizados no Guião 8, que nos facilitou a implementação neste caso, mas que se revelou, de qualquer forma, desafiante.

```
void dynamicTranslate(Oper* oper, int i)
{
    Transform* t = oper->transform;
    static double intervalos[20];
    static double checkpoints[20];
    static double time = 0;

    //criar matriz aqui
    int h = (t->points).size();
    int cols = 3;
    double** curve = new double* [h];

    for (int i = 0; i < h; ++i)
        curve[i] = new double[cols];


    int size = 0;
    vector<Ponto>::iterator it;
    for (it = t->points.begin(); it != t->points.end(); it++) {
        Ponto aux = *it;

        curve[size][0] = aux.x;
        curve[size][1] = aux.y;
        curve[size][2] = aux.z;

        size++;
    }

    double pos[4];
    float deriv[4];

    glBegin(GL_LINE_LOOP);
        int n = 100;
        for (int j = 1; j < n; j++) {
            getGlobalCatmullRomPoint((double) j / n, pos, deriv, curve, size);
            glVertex3d(pos[0], pos[1], pos[2]);
        }
    glEnd();

    getGlobalCatmullRomPoint(intervalos[i], pos, deriv, curve, size);

    //if (time < files.size()) {
        double ttt = glutGet(GLUT_ELAPSED_TIME);
        checkpoints[i] = 1 / (t->time * 1000);
        time++;
    //}
    intervalos[i] += checkpoints[i];

    //apagar matriz aqui
    for (int i = 0; i < h; ++i)
        delete[] curve[i];
    delete[] curve;


    glTranslated(pos[0], pos[1], pos[2]);
}
```

Figura 13 – Função dynamicTranslate

Sistema Solar, agora dinâmico

Nesta fase tivemos de alterar o nosso ficheiro XML de forma a obter um sistema solar dinâmico. Para tal, alteramos os *translates* que tínhamos previamente, para conseguirmos passar vários pontos das órbitas dos diferentes objetos estelares.

Chegamos à conclusão de que, para obter uma órbita com forma semelhante a uma circunferência, seriam necessários 12 pontos. Por esta razão, todas as órbitas do nosso ficheiro têm 12 pontos diferentes, exceto a órbita do cometa em que apenas passamos 4 de forma a obter uma órbita mais achatada.

Os cálculos para a obtenção das coordenadas dos diferentes pontos das órbitas passam pela lógica de que cada um dos pontos da órbita está distanciado num ângulo de 30 graus dos seus vizinhos. Desta forma, basta multiplicar as suas coordenadas por *cos* e *sin* de 30 graus, de forma a obter x e y, respetivamente.

```
<translate time="0.25">
    <point X="16.23" Y="0" Z="0" />
    <point X="14.05" Y="0" Z="8.115" />
    <point X="8.115" Y="0" Z="14.05" />
    <point X="0" Y="0" Z="16.3" />
    <point X="-8.115" Y="0" Z="14.05" />
    <point X="-14.05" Y="0" Z="8.115" />
    <point X="-16.23" Y="0" Z="0" />
    <point X="-14.05" Y="0" Z="-8.115" />
    <point X="-8.115" Y="0" Z="-14.05" />
    <point X="0" Y="0" Z="-16.3" />
    <point X="8.115" Y="0" Z="-14.05" />
    <point X="14.05" Y="0" Z="-8.115" />
</translate>
```

Figura 13 - Exemplo de translação dinâmica no nosso sistema solar

Para além da definição dos pontos, foi necessária também a correta escolha do "time", que representa o tempo que leva o objeto estelar a dar a volta à sua órbita, pelo que necessitamos de alguma pesquisa para tal. Foi ainda adicionado um cometa ao nosso modelo, desenhado através de um ficheiro Bezier.

```
<group>
    <translate time="1">
    <point X="200" Y="0" Z="0" />
    <point X="0" Y="0" Z="20" />
    <point X="-200" Y="0" Z="0" />
    <point X="0" Y="0" Z="-20" />
    </translate>
    <scale X="0.5" Y="0.55" Z="0.55" />
    <models>
        <model file = "bezi.3d" />
    </models>
</group>
```

Figura 14 – Cometa que incluímos no nosso sistema solar

No final de tudo isto, como confirmação de todo o trabalho realizado, podemos ver que conseguimos construir um sistema solar completo, com o Sol, todos os planetas e alguns dos principais satélites naturais mais importantes representados, assim como o cometa, e a órbita de todos. Como é óbvio, na imagem não é possível verificar, mas, após alguns cálculos conseguimos que o movimento dos satélites naturais acompanhasse o planeta a que correspondem.

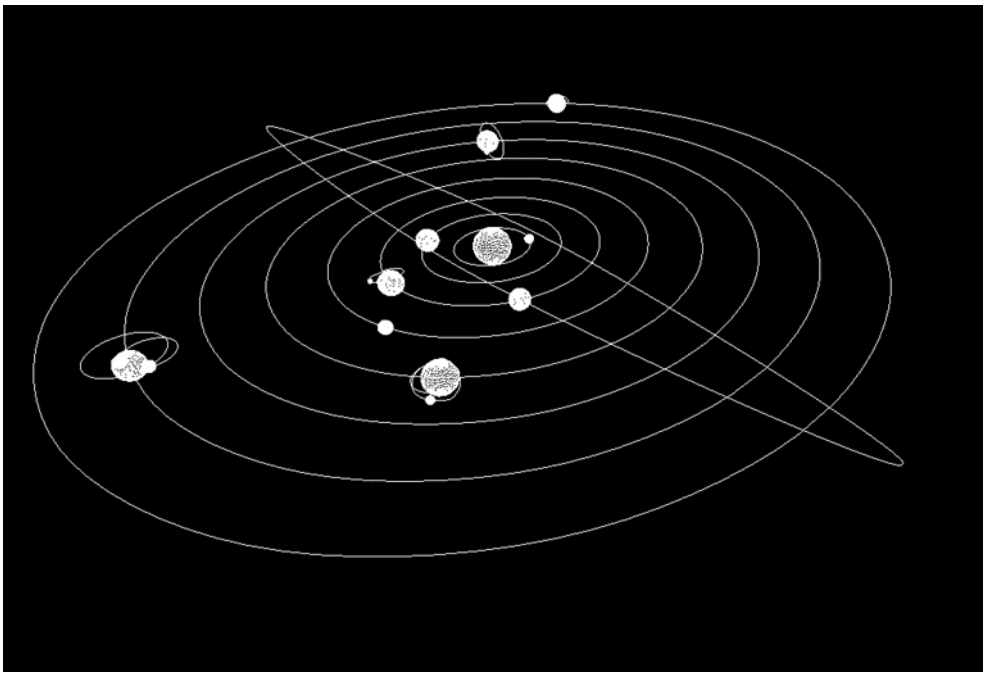


Figura 15 – Representação final do nosso sistema solar

Conclusão

Nesta terceira fase do trabalho prático, conseguimos desenvolver os conhecimentos sobre as transformações dinâmicas adquiridos nas aulas e nos guiões práticos, aprofundando-os, devido à maior complexidade dos objetivos propostos. Pudemos também aprofundar os nossos conhecimentos em *OpenGL* e na linguagem C++, que, de outra forma, não seria possível.

De uma forma geral, fazemos um balanço positivo do trabalho realizado e consideramos que os objetivos para esta fase foram alcançados com sucesso, pois pensamos ter cumprido com todos os requisitos, culminando num resultado final que nos orgulhamos e pensamos representar o nosso esforço no desenvolvimento do mesmo. Apesar disso, pensamos não estar perfeito, mas sabemos que, com o tempo e a prática, aperfeiçoaremos o projeto na fase final do mesmo.