# Motor

Nesta fase tivemos de alterar alguns aspetos do motor, nomeadamente os relativos às transformações geométricas. Para além disso e por forma a poder utilizar VBO’s tivemos também de alterar um pouco a classe Figura e também incluir algumas bibliotecas para os poder utilizar, a *GLEW e DevIL*.

## Translacao

Por forma a permitir definir as curvas de *Catmull-Rom* para as translações das figuras, decidimos que a classe translação teria um conjuntos de pontos de controlo. Para além disto, como esta translação é dependente de um tempo foi necessário guardar também o valor desse mesmo tempo. Por forma a não gerar todas as iterações os pontos da curva (de translação), e visto que a tínhamos de mostrar, então guardamos também um conjunto de todos os pontos da curva, geradas com a fórmula para as curvas de *Catmull-Rom* com um tempo a variar entre 0 e 1 com um delta igual a 0.01.

vector<Ponto\*> Translacao::getCurva(){

if(curva.size()==0){

float res[3];

float deriv[3];

for (float t = 0; t<1; t += 0.01){

getGlobalCatmullRomPoint(t, res, deriv);

Ponto\* p = new Ponto(res[0], res[1], res[2]);

curva.push\_back(p);

}

}

return curva;

}

Para além disto, era necessário ter um vetor referente ao *upVector* que é importante para a rotação do planeta segundo a trajetória consoante a derivada no ponto em questão.

class Translacao: public Operacao {

private:

float tempo;

vector<Ponto\*> pontos;

vector<Ponto\*> curva;

float upY[3];

void normalize(float\*);

void getCatmullRomPoint(float,int\*,float\*,float\*);

void getGlobalCatmullRomPoint(float,float\*,float\*);

void rodaElemento(float\*);

public:

Translacao();

Translacao(float,float,float,float,vector<Ponto\*>);

float getTempo();

void setTempo(float);

vector<Ponto\*> getPontos();

void setPontos(vector<Ponto\*>);

vector<Ponto\*> getCurva();

void aplicaOperacao();

void renderCatmullRomCurve();

string toString();

};

A única alteração em termos de algoritmos na classe foi a forma como realizamos a translação. Caso o tempo de translação seja 0 então é considerado que a translação é simples, como na fase anterior. Caso não o seja, então geramos os pontos de translação como o valor da curva de *Catmull-Rom* consoante o tempo passado (desde o início da execução) e depois aplicamos a translação consoante a posição dada pela fórmula. Por forma a aplicar a rotação segunda a derivada, temos também uma função que a aplica, invocando-a com o argumento da derivada recebida pela função que executa a fórmula das curvas de *Catmull-Rom*. Para além disso, desenhamos também a curva através da função *renderCatmullRomCurve()*

void Translacao::aplicaOperacao(){

if(tempo==0){

glTranslatef(getX(),getY(),getZ());

}

else{

renderCatmullRomCurve();

float tempoDecorrido = glutGet(GLUT\_ELAPSED\_TIME);

float percentagem = tempoDecorrido / (tempo \* 1000);

percentagem = percentagem – floor(percentagem);

float pos[3];

float deriv[3];

getGlobalCatmullRomPoint(percentagem,pos,deriv);

glTranslatef(pos[0],pos[1],pos[2]);

rodaElemento(deriv);

}

}

Após ter sido calculada a posição segundo a curva de *Catmull-Rom* e ter sido aplicada a translação para essa posição foi necessário aplicar uma rotação segundo a derivada calculada, para que o objeto acompanhasse o movimento da trajetória. Essa rotação foi realizada com o auxílio d afunção **rodaElemento**.

Esta função apenas normaliza o vetor das derivadas, aplica um produto vetorial a esse vetor e ao *upVector* (que é iniciado como 0,1,0). Depois normaliza o vetor resultante (não seria necessário pois os 2 vetores estão normalizados, mas por questões de segurança realizamos essa operação) e aplica um produto vetorial a esse vetor e ao das derivadas, guardando o resultado no *upVector*, voltando a normalizar o vetor resultado. Depois constrói a matriz de rotação segundo os vetores calculados através das produtos vetoriais e das derivadas, sendo essa matriz:

|  |  |  |  |
| --- | --- | --- | --- |
| Deriv(0) | Deriv(1) | Deriv(2) | 0 |
| upY(0) | upY(1) | upY(2) | 0 |
| vZ(0) | vZ(1) | vZ(2) | 0 |
| 0 | 0 | 0 | 1 |

upY corresponde ao *upVector* e vZ corresponde ao vetor resultante do primeiro produto vetorial.

void buildRotMatrix(float \*x, float \*y, float \*z, float \*m) {

m[0] = x[0]; m[1] = x[1]; m[2] = x[2]; m[3] = 0;

m[4] = y[0]; m[5] = y[1]; m[6] = y[2]; m[7] = 0;

m[8] = z[0]; m[9] = z[1]; m[10] = z[2]; m[11] = 0;

m[12] = 0; m[13] = 0; m[14] = 0; m[15] = 1;

}

Após ter calculado essa matriz de rotação, multiplica a matriz atual por essa matriz, estando de momento em condições de poder desenhar o objeto que se move na trajetória.

void Translacao::rodaElemento(float \*deriv){

float rotacao[4][4];

float vZ[3];

normalize(deriv);

cross(deriv,upY,vZ);

normalize(vZ);

cross(vZ,deriv,upY);

normalize(upY);

buildRotMatrix(deriv,upY,vZ,\*rotacao);

glMultMatrixf(\*rotacao);

}

void cross(float \*a, float \*b, float \*res) {

res[0] = a[1]\*b[2] - a[2]\*b[1];

res[1] = a[2]\*b[0] - a[0]\*b[2];

res[2] = a[0]\*b[1] - a[1]\*b[0];

}

void Translacao::normalize(float \*a) {

float l = sqrt(a[0]\*a[0] + a[1] \* a[1] + a[2] \* a[2]);

a[0] = a[0]/l;

a[1] = a[1]/l;

a[2] = a[2]/l;

}

Para desenhar a curva, apenas chamamos a função que nos dá os pontos da curva, caso estes ainda não tenham sido calculados (daí o tamanho ser 0) e depois para cada ponto desenhamos esse ponto. Usamos a opção **GL\_LINE\_LOOP** para poder desenhar a curva.

void Translacao::renderCatmullRomCurve() {

// desenhar a curva usando segmentos de reta - GL\_LINE\_LOOP

if(curva.size()==0){

getCurva();

}

glBegin(GL\_LINE\_LOOP);

for(int i = 0; i < curva.size(); i++){

Ponto\* p = curva.at(i);

glVertex3f(p->getX(),p->getY(),p->getZ());

}

glEnd();

}

A função que gera a curva varia o tempo com um delta de 0.01 obtendo para cada valor de t o ponto correspondente segundo a função **getGlobalCatmullRomPoint** e guarda esse ponto num conjunto de pontos correspondentes à curva.

vector<Ponto\*> Translacao::getCurva(){

if(curva.size()==0){

float res[3];

float deriv[3];

for (float t = 0; t<1; t += 0.01){

getGlobalCatmullRomPoint(t, res, deriv);

Ponto\* p = new Ponto(res[0], res[1], res[2]);

curva.push\_back(p);

}

}

return curva;

}

O algoritmo de seleção do tempo e da função de *Catmull-Rom* será detalhada numa seção mais à frente.

## Rotação

Da mesma forma que na translação adicionamos uma variável correspondente ao tempo.

class Rotacao: public Operacao {

private:

float angulo;

float tempo;

O algoritmo segue também o mesmo raciocínio, caso o tempo seja 0 então usa o ângulo definido, caso não seja usa uma fórmula para obter uma percentagem em relação ao tempo de rotação, consoante o tempo decorrido para multiplicar posteriormente por 360 graus (uma volta completa).

void Rotacao::aplicaOperacao(){

if(tempo==0){

glRotatef(angulo,getX(),getY(),getZ());

}

else{

float tempoDecorrido = glutGet(GLUT\_ELAPSED\_TIME);

float percentagem = tempoDecorrido / (tempo \* 1000);

percentagem = percentagem – floor(percentagem);

glRotatef(360\*percentagem,getX(),getY(),getZ());

}

}

### Algoritmo de Tempo

Para o algoritmo do cálculo do tempo tanto de translação como de rotação utilizamos a função **glutGet(GLUT\_ELAPSED\_TIME)** por forma a saber qual o tempo passado desde o início da execução do programa. Depois apenas tínhamos de retirar o valor da divisão pelo tempo de rotação/translação do **Grupo**. De seguida tivemos de retirar a parte fracionária do valor anteriormente calculado, para isso fizemos a operação: valor – valor inteiro arredondado para baixo (resultado da operação **floor**).

Apenas tivemos de multiplicar sempre o tempo de rotação/translação por 1000 pois o valor do tempo passado é dado em milissegundos.

float tempoDecorrido = glutGet(GLUT\_ELAPSED\_TIME);

float percentagem = tempoDecorrido / (tempo \* 1000);

percentagem = percentagem – floor(percentagem);

Inicialmente pensamos em retirar o resto da divisão inteira do valor do tempo passado pelo tempo de rotação/translação e posteriormente dividir pelo tempo de rotação/translação, no entanto como os números são em vírgula flutuante isso poderia trazer erros ao programa, daí utilizarmos a forma referida acima. Apresentamos esse algoritmo de seguida:

float tempoDecorrido = glutGet(GLUT\_ELAPSED\_TIME);

float percentagem = (int) tempoDecorrido % (int) (tempo \* 1000);

percentagem = percentagem / (tempo \* 1000);

## Figura

A alteração efetuada na classe **Figura** concentra-se no facto de agora imprimirmos as figuras com o auxílio de *VBO’s*. Para isso em vez de guardar um conjunto de pontos nesta classe, guardamos um *array* de *GLuint*, neste caso apenas com 1 posição pois nesse guardamos os pontos para desenhar as figuras com os *VBO’s*. Guardamos também o número de pontos presentes nesse *array*.

class Figura {

GLuint pontos[1];

int numeroPontos;

Quando iniciamos o objecto Figura com um conjunto de pontos, esta função chama uma auxiliar **adicionaPontos** que é responsável por passar aquele conjunto de pontos para a estrutura que possibilita o desenho com os *VBO’s*.

Figura::Figura(vector<Ponto\*> ps) {

this->adicionaPontos(ps);

}

Essa função de início calcula o número de pontos, e de seguida aloca memória para um *array* de *float’s* com espaço para todos os pontos, ou seja, número de pontos \* 3 posições. Depois preenche esse *array* com as coordenadas dos pontos presentes no conjunto passado como argumento. Depois gera o *buffer* que irá conter os pontos para serem desenhados com o mecanismo referido e associa a primeira posição desse *buffer* como sendo o *array* atual através das funções **glGenBuffers** e **glBindBuffer** respetivamente. Depois transfere os dados presentes no *array* qe contém os pontos para o *array* atual através da função **glBufferData** e liberta o espaço alocado anteriormente.

void Figura::adicionaPontos(vector<Ponto\*> ps){

numeroPontos = ps.size();

float\* vertices = (float\*) malloc(sizeof(float) \* ps.size() \* 3);

int i = 0;

for(Ponto\* p: ps){

vertices[i++] = p->getX();

vertices[i++] = p->getY();

vertices[i++] = p->getZ();

}

glGenBuffers(1, pontos);

glBindBuffer(GL\_ARRAY\_BUFFER, pontos[0]);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* numeroPontos \* 3, vertices, GL\_STATIC\_DRAW);

free(vertices);

}

A outra alteração é que em vez do **motor** “pedir” os pontos à figura e desenhar um a um, agora aplica a função desenha à figura e esta desenha os pontos todos através dos *VBO’s*. Essa função apenas define o *array* atual como o *array* de pontos presentes na classe, define que os vértices são compostos por 3 coordenadas do tipo *float* e desenha os triângulos desse *array*.

void Figura::desenha(){

glBindBuffer(GL\_ARRAY\_BUFFER, pontos[0]);

glVertexPointer(3, GL\_FLOAT, 0, 0);

glDrawArrays(GL\_TRIANGLES, 0, numeroPontos \* 3);

}

# Curvas Catmull-Rom

Para calcular as curvas de *Catmull-Rom* e as posições dos elementos nas curvas temos uma função **getGlobalCatmullRomPoint** que dado um tempo, um vetor para o ponto e um para as derivadas, preenche os 2 últimos com os valores das coordenadas do ponto e das derivadas segunda o algoritmo das curvas de *Catmull-Rom*.

Essa função calcula quais os pontos de controlo serão utilizados, ou seja qual será a parte da curva a que corresponde o tempo passado. Armazena os índices desses pontos num array **índices** de 4 posições, visto serem necessários 4 pontos de controlo. Depois invoca uma função **getCatmullRomPoint** que aplica o método de *Catmull-Rom* para os índices calculados, para o tempo em questão e preenche os vetores de coordenadas e derivadas.

void Translacao::getGlobalCatmullRomPoint(float gt, float \*pos, float \*deriv) {

int numeroPontos = pontos.size();

float t = gt \* numeroPontos; // this is the real global t

int index = floor(t); // which segment

t = t - index; // where within the segment

// indices store the points

int indices[4];

indices[0] = (index + numeroPontos-1)%numeroPontos;

indices[1] = (indices[0]+1)%numeroPontos;

indices[2] = (indices[1]+1)%numeroPontos;

indices[3] = (indices[2]+1)%numeroPontos;

getCatmullRomPoint(t, indices, pos, deriv);

}

A função **getCatmullRomPoint** utiliza a matriz de *CatmullRom:*

|  |  |  |  |
| --- | --- | --- | --- |
| -0.5 | 1.5 | -1.5 | 0.5 |
| 1 | -2.5 | 2 | -0.5 |
| -0.5 | 0 | 0.5 | 0 |
| 0 | 1 | 0 | 0 |

Depois apenas tivemos de multiplicar esta matriz pela coordenada x de cada ponto, para isso utilizamos uma função auxiliar **multMatrixVector** que multiplica a mariz pelo vetor de coordenadas. Após esta operação ficamos com uma matriz 4x1 e apenas tínhamos de multiplicar esse matriz pela matriz correspondente ao tempo. Ou seja:

|  |  |  |  |
| --- | --- | --- | --- |
| t3 | t2 | t | 1 |

\*

|  |
| --- |
| a1 |
| a2 |
| a3 |
| a4 |

Após este passo tínhamos a coordenada x do ponto pretendido. Para as restantes coordenadas (y e z) aplicamos o mesmo processo.

Relativamente às derivadas o algoritmo segue o mesmo raciocínio, diferindo apenas na matriz correspondente ao tempo, sendo que neste caso essa matriz corresponderia à matriz das derivadas dessa matriz, ou seja:

|  |  |  |  |
| --- | --- | --- | --- |
| 3t2 | 2t | 1 | 0 |

\*

|  |
| --- |
| a1 |
| a2 |
| a3 |
| a4 |

Na função apenas apresentamos o caso para a coordenada x, os restantes são análogos, como já foi referido.

void Translacao::getCatmullRomPoint(float t, int\* indices, 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}

};

Ponto\* p0 = pontos.at(indices[0]);

Ponto\* p1 = pontos.at(indices[1]);

Ponto\* p2 = pontos.at(indices[2]);

Ponto\* p3 = pontos.at(indices[3]);

// Compute A = M \* P

float aX[4];

float pX[4];

pX[0] = p0->getX();

pX[1] = p1->getX();

pX[2] = p2->getX();

pX[3] = p3->getX();

multMatrixVector(\*m,pX,aX);

// Compute pos = T \* A

float T[4] = {t\*t\*t,t\*t,t,1};

float posX = 0;

for(int i = 0; i < 4; i++){

posX += T[i]\*aX[i];

}

pos[0] = posX;

// compute deriv = T' \* A

float TD[4] = {3\*t\*t,2\*t,1,0};

float derX = 0;

for(int i = 0; i < 4; i++){

derX += TD[i]\*aX[i];

}

deriv[0] = derX;

}

void multMatrixVector(float \*m, float \*v, float \*res) {

for (int j = 0; j < 4; ++j) {

res[j] = 0;

for (int k = 0; k < 4; ++k) {

res[j] += v[k] \* m[j \* 4 + k];

}

}

}