Computação Gráfica

Trabalho Prático - Fase 3

Universidade do Minho, Departamento de Informática
José Correia, Diogo Abreu, e Rodrigo Monteiro
{100610, 100646, 100706}@alunos.uminho.pt

Grupo 69

1. Introdução

Para esta terceira fase do projeto tem-se como objetivo dar seguimento ao trabalho realizado nas fases anteriores, expandindo as funcionalidades do Generator e da Engine com a introdução de novos conceitos: curvas de Bezier, curvas de Catmull-Rom e **VBO's**.

O Generator desenvolve a capacidade de construir modelos baseados em *patches* de Bezier. Já a Engine aplica novos tipos de translações e rotações com base em curvas de Catmull-Rom. Para além disso, optimiza o desempenho no desenho dos modelos produzidos através do uso de **VBO's**.

2. Geração de modelos baseados em patches de Bezier

2.1. Leitura de ficheiros .patch

Para construir estes novos modelos baseados em *patches* de Bezier, contidos em ficheiros `.patch`, começamos por implementar um método no Generator, que lê estes ficheiros e armazena os pontos dos vários *patches* num `vetor<vetor<Point>>`.

1. Identificamos os índices dos pontos de cada *patch*.

```
vector<vector<Point>> read_patches_file(const char* file_path) {
  int num_patches;
  istringstream(line) >> num_patches;
  vector<vector<int>> idxs_per_patch;
  for (int i = 0; i < num_patches && getline(input_file, line); i++) {
        vector<int> idxs;
        ...
        idxs_per_patch.push_back(idxs);
  }
```

2. De seguida, obtemos os pontos de controlo.

```
  int num_pontos;
  istringstream(line) >> num_pontos;
  vector<Point> control_points;
  for (int i = 0; i < num_pontos && getline(input_file, line); i++) {
    ...
    control_points.push_back(Point(x, y, z));
  }
```

3. Por fim, contruimos os *patches*, percorrendo os vetores de índices e extraindo os pontos correspondentes ao *patch* atual.

```
  vector<vector<Point>> result;
  for (vector<int> idxs : idxs_per_patch) {
    vector<Point> patch;
    for (int idx : idxs) {
      patch.push_back(Point(control_points[idx]));
    }
    result.push_back(patch);
  }
  input_file.close();
  return result;
}
```

2.2. **Figuras de Bezier**

Decidimos criar uma nova figura para este tipo de modelos, esta figura recebe como argumentos *points_per_patch* (`vetor<vetor<Point>>`) e *tesselation* (nível de divisão).

Este método começa por inicializar os parâmetros $u = 0.0\text{f}$, $v = 0.0\text{f}$ e $\Delta = 1.0\text{f}/$ tessellation. Posteriormente, calcula os pontos de cada patch de acordo com o nível de divisão fornecido.

```
void Bezier::generate_points() {
    float u = 0.0f;
    float v = 0.0f;
    float delta = 1.0f / this->tesselation;

    for (vector<Point> patch : this->points_per_patch) {  // 16 pontos
        for (int i = 0; i < this->tesselation; i++, u += delta) {
            for (int j = 0; j < this->tesselation; j++, v += delta) {
                // Pontos para formar o quadrado
                Point a = Point::surface_point(u, v, patch);
                Point b = Point::surface_point(u, v + delta, patch);
                Point c = Point::surface_point(u + delta, v, patch);
                Point d = Point::surface_point(u + delta, v + delta,
patch);
                // Triangulacao
                this->points.push_back(c);
                this->points.push_back(a);
                this->points.push_back(b);
                this->points.push_back(b);
                this->points.push_back(d);
                this->points.push_back(c);
            }
            v = 0.0f;
        }
        u = v = 0.0f;
    }
```

```
}
```

Sendo estes pontos calculados utilizando o método `surface_point(u,v,patch)`, que implementa uma multiplicação de matrizes, onde os parâmetros `u` e `v` representam as coordenadas do patch, e o `patch` é a matriz que descreve o patch em questão.

$$p(u,v) = U \cdot M \cdot P \cdot M^T \cdot V^T$$

$$U = [u3 \;\; u2 \;\; u \;\; 1], V = [v3 \;\; v2 \;\; v \;\; 1], M = M^T = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

3. Translações e Rotações Animadas

3.1. Leitura dos grupos XML

Com a introdução destes dois novos tipos de transformações, reparamos que estes têm parâmetros de configuração diferentes das antigas transformações (Scale, Translate e Rotate) que possuem as três os mesmos parâmetros. Por isso decidimos modificar a nossa classe Transform de forma a conseguir agrupar todos os tipos de transformações durante a leitura dos ficheiros de configuração.

```
class Transform {
public:
    enum TransformType { TRANSLATE = 1, SCALE = 2, ROTATE = 3,
TRANSLATE_ANIMATION = 4, ROTATE_ANIMATION = 5 };
    Transform(float x, float y, float z, float angle, TransformType type);
    Transform(float x, float y, float z, float time);// ROTATE_ANIMATION
    Transform(float time, bool align, std::vector<Point>& points);//
TRANSLATE_ANIMATION
```

3.2. Desenho de Rotações Animadas

As rotações animadas tomam como parâmetro extra o tempo que o objeto demora a realizar uma rotação de 360 graus (t_r). Com este valor conseguimos deduzir o ângulo de rotação a cada instante (t), da seguinte forma:

$$\alpha(t) = (t - t_0) \times \frac{360^o}{t_r}$$

Esta fórmula só faz sentido para tempos superiores a 0. Ou seja, quando o tempo é 0 significa que estamos perante uma rotação estática. Por essa razão verificamos o tempo para saber se aplicamos a fórmula referida acima ou uma simples rotação estática.

```
void drawGroups(Group* group, int *index) {
 ...
  for (Transform* t : group->transforms) {
   case Transform::TransformType::ROTATE_ANIMATION:
```

```
    if (t->time > 0.0f) {
      glRotatef((((NOW - init_time) * 360.0f) / t->time), t->x,t->y,t->z);
    } else {
      glRotatef(t->angle, t->x, t->y, t->z);
    }
  }
```

3.3. Desenho de Translações Animadas

Esta transformação tem como parâmetros um conjunto de pontos de controlo de uma curva de Catmull-Rom, um tempo e um indicador se o objeto fica alinhado com a curva. A junção destes parâmetros vai permitir definir a trajetória da translação ao longo do tempo.

3.3.1. Pontos nas curvas de Catmull-Rom

Para definir os pontos da curva utilizamos dois métodos, o método **`get_catmull_rom_point`** que calcula o ponto de uma curva de Catmull-Rom dados quatro pontos de controlo e o método **`get_global_catmull_rom_point`** que calcula o ponto da curva dado o instante de tempo atual da animação.

Este primeiro método recebe quatro pontos de controlo e executa os seguintes calculos matriciais para determinar o ponto p:

$$p(t) = [t^3 \;\; t^2 \;\; t \;\; 1] \cdot \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Já o método `get_global_catmull_rom_point` identifica quatro pontos de controlo de acordo com o instante de tempo atual, e posteriormente passa estes mesmos pontos para o método `get_catmull_rom_point`.

```
void get_global_catmull_rom_point(float gt, vector<Point> controlPoints,
Point* pos, Point* deriv) {
  size_t POINT_COUNT = controlPoints.size();
  float t = gt * POINT_COUNT;
  int index = (int) floor(t);
  t = t - index;

  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;

  get_catmull_rom_point(t, controlPoints[indices[0]],
 controlPoints[indices[1]], controlPoints[indices[2]],
 controlPoints[indices[3]], pos, deriv);
}
```

De forma a determinar ao tempo que o objeto demora a percorrer a curva, dividimos o instante de tempo atual pelo tempo especificado no XML (`gt`). Ao realizar esta divisão, alteramos a taxa de variação da posição do objeto ao longo da curva, o que, por sua vez, influencia o tempo total necessário para o objeto percorrer toda a curva.

3.3.2. **Desenho**

Agora que temos como nos posicionar nas curvas ao longo do tempo, podemos passar para o desenho da translação animada, que foi definida da seguinte forma:

```
case Transform::TransformType::TRANSLATE_ANIMATION:
  if (t->time && t->time > 0.0f) {
    Point* pos = new Point();
    Point* deriv = new Point();
    get_global_catmull_rom_point(NOW / t->time, t->points, pos, deriv);
    if (show_curves) drawCatmullRomCurve(t->points);
    glTranslatef(pos->x, pos->y, pos->z);
    if (t->align) {
      deriv->normalize();
      Point* z = Point::cross(deriv, t->y_axis);
      z->normalize();
      Point* y = Point::cross(z, deriv);
      t->y_axis = y;
      y->normalize();

      float rotation_matrix[16];
      build_rotation_matrix(deriv, y, z, rotation_matrix);
      glMultMatrixf(rotation_matrix);
    }
  }
```

Nos casos em que a flag `align` está ativa temos de alinhar o objeto com a curva, para tal precisamos de redefinir a orientação dos eixos do seu sistema de coordenadas:

$$X_i = p'(t)$$
$$Z_i = X_i \times Y_{i-1}$$
$$Y_i = Z_i \times X_i$$

Sendo $p'(t)$ a derivada da curva no instante t e Y_{i-1} a orientação do eixo Y no instante anterior. Nós tomamos o vetor (0,1,0) como orientação padrão do eixo dos Y.

4. Point

4.1. cross

Esta função recebe dois objetos `Point` a e b, e calcula o produto vetorial deles, resultando em um novo vetor perpendicular a ambos a e b. O produto vetorial é calculado usando o determinante de uma matriz 3x3 formada pelos componentes dos vetores de entrada.

```
Point* Point::cross(Point* a, Point* b) {
  return new Point(
    a->y * b->z - a->z * b->y,
    a->z * b->x - a->x * b->z,
    a->x * b->y - a->y * b->x
  );
}
```

4.2. normalize

Esta função calcula o comprimento do vetor representado pelo objeto `Point` usando a fórmula da norma Euclidiana (raiz quadrada da soma dos quadrados de seus componentes), e então divide cada componente do vetor pelo seu comprimento para dimensioná-lo para um vetor unitário.

```
void Point::normalize() {
  float l = sqrt(this->x * this->x + this->y * this->y + this->z * this-
>z);
  this->x = this->x / l;
  this->y = this->y / l;
  this->z = this->z / l;
}
```

5. VBO's

Na inicialização do `Engine`, este começa por ler a configuração dos ficheiros XML e construir o vetor `stored_figures`. Estando este construído, através da função abaixo `loadBuffersData` carrega-se para os buffers os pontos de cada modelo presente no vetor referido acima.

```
void loadBuffersData(int* index) {
  for (Figure* f : stored_figures) {
    vector<float> figure_buffer = f->to_vector();
    glBindBuffer(GL_ARRAY_BUFFER, buffers[(*index)++]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * figure_buffer.size(),
figure_buffer.data(), GL_STATIC_DRAW);
    buffer_sizes.push_back(figure_buffer.size() / 3);
  }
}
```

Quando o `Engine` estiver a correr a animação, a função `renderScene`, que por sua vez executa a função `drawGroups`, será executada em cada frame.

```
for (unsigned long i = 0; i < group->models.size(); i++, (*index)++) {
      glBindBuffer(GL_ARRAY_BUFFER, buffers[*index]);
      glVertexPointer(3, GL_FLOAT, 0, 0);
      glDrawArrays(GL_TRIANGLES, 0, buffer_sizes[*index]);
    }

    for (Group* g : group->groups) {
      drawGroups(g, index);
    }
```

6. Output

6.1. Testes

O conjunto de testes disponibilizados foram testados com êxito. De seguida, seguem-se os resultados.

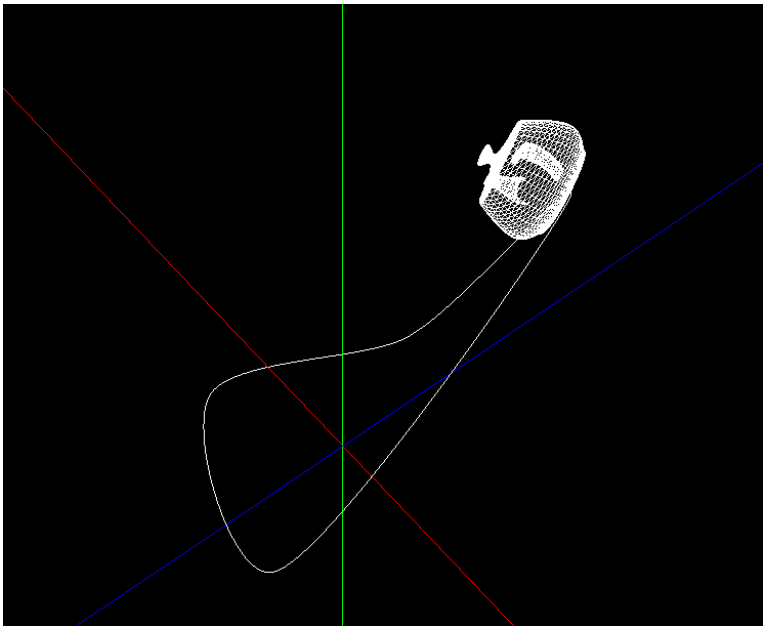


Figure 1: Chaleira e a sua trajetória

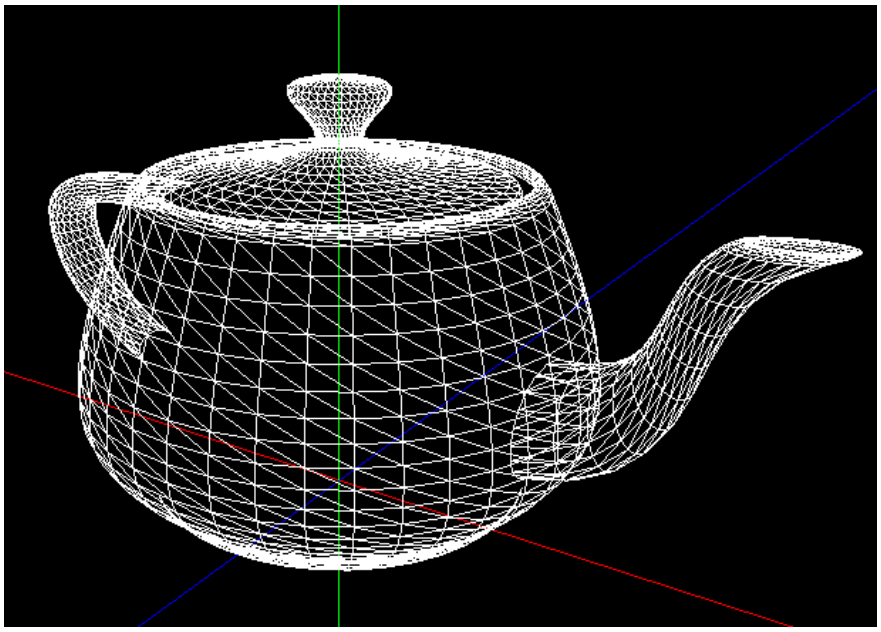
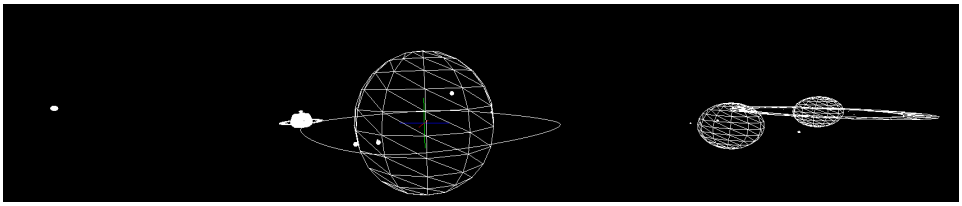


Figure 2: Chaleira originada pelo ficheiro patch

6.2. **Sistema Solar**

Para a demonstração do sistema solar, foi feito um script em python.



7. **Conclusão**

Ao longo do desenvolvimento desta fase do projeto, foi nos possível consolidar conhecimentos relativos a curvas de Bezier e curvas de Catmull-Rom, assim como também nos foi possível aprimorar as aplicações já existentes.

Quanto ao trabalho realizado, encontrámo-nos satisfeitos, dado que conseguimos concretizar todas as funcionalidades pretendidas.

References

1. Wikipedia: Rotation Matrix, https://en.wikipedia.org/wiki/Rotation_matrix
2. Swiftless: OpenGL Popping and Pushing Matrices, https://www.swiftless.com/tutorials/opengl/pop_and_push_matrices.html