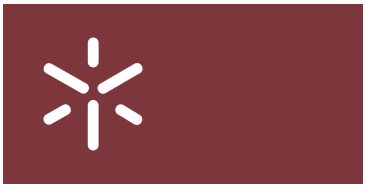

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2024/2025

VBOS, Bezier e Catmull-Rom - Fase 3

Afonso Pedreira A104537
André Pinto A104267
Fábio Magalhães A104365

Abril, 2025

CG

Índice

1. Introdução

Este relatório tem como objetivo descrever o trabalho desenvolvido na terceira fase do projeto, no âmbito desta unidade curricular. Nesta fase, foram introduzidas novas funcionalidades tanto no **engine** como no **generator**, alargando significativamente as suas capacidades.

No **engine**, implementou-se o suporte à utilização de Vertex Buffer Objects (VBOs), substituindo o modo imediato anteriormente utilizado para o desenho de modelos. Para além disso, foi integrada a possibilidade de realizar animações baseadas em curvas de **Catmull-Rom**, permitindo definir movimentos suaves e contínuos através de pontos de controlo, com suporte adicional para alinhamento do modelo à direção da curva. A transformação associada a rotações foi também expandida, passando a permitir rotações animadas baseadas no tempo.

No que diz respeito ao **generator**, foi adicionada a capacidade de gerar superfícies a partir de patches de **Bézier**. O utilizador fornece um ficheiro com os pontos de controlo e o nível de *tessellation* pretendido, sendo gerado um ficheiro com os triângulos necessários para desenhar a superfície correspondente.

2. Estrutura do Projeto

O projeto está dividido em dois programas principais: ***generator*** e ***engine***, que trabalham em conjunto para gerar e renderizar figuras gráficas. A estrutura do projeto é organizada de forma a facilitar a modularidade e a reutilização de código.

A arquitetura do projeto é composta pelas seguintes pastas principais:

- **engine**: Esta pasta contém os componentes responsáveis pela renderização das cenas. Inclui elementos essenciais como:
  - **camera**: Define a perspetiva e a posição da visualização da cena.
  - **configuration**: Contém as configurações gerais do sistema, como a `window`, as definições da câmara e informação sobre a cena.
  - **window**: Responsável pela gestão dos parâmetros da janela de visualização.
  - **ModelGroup**: Responsável por armazenar todos os modelos pertencentes a uma cena e respetivas transformações.
  - **Model**: Responsável pelo armazenamento dos pontos dos respetivos modelos, incluindo toda a lógica de **VBOs**
  - **filesParser**: Este módulo é responsável por fazer o `parsing` dos ficheiros de entrada (`.xml`) e configurar os dados necessários para a renderização.
  - **catmullCurves**: Responsável por armazenar toda a lógica relacionada com translações ao longo de uma curva, rotações, cálculo de posições na curva, entre outros.
- **generator**: Esta pasta contém os módulos responsáveis pela geração das figuras geométricas. Cada figura é gerada a partir de parâmetros específicos e guarda num ficheiro `.3d`. Estes ficheiros, posteriormente, serão embutidos em ficheiros `.xml`, que a *engine* utiliza para renderizar as cenas.
- **common**: Esta pasta contém componentes compartilhados entre os dois programas principais. Inclui definições essenciais, como a definição de **Ponto** bem como funções auxiliares para tarefas comuns, como:
  - **salvar em ficheiro**: Funções para escrever os dados gerados em ficheiros de diferentes formatos.
  - **ler de ficheiro**: Funções para ler dados de ficheiros de entrada, como `.xml`,`.obj` e `.patch`.
  - **criar ficheiro 3D**: Funções que permitem a criação e manipulação de ficheiros no formato `.3d`.

A organização modular do projeto permite uma gestão eficiente de cada componente, garantindo a escalabilidade e uma boa base para as próximas fases.

2.1. *Engine*

Nesta terceira fase, foi necessário proceder a algumas alterações no **engine**, de modo a ser possível responder às necessidades e desafios propostos. Para isso, foi criada uma classe individual para cada modelo, denominada ***Model***, e foi também realizada uma alteração na estrutura ***ModelGroup***.

2.1.1. *ModelGroup*

Para assegurar que toda a informação fosse devidamente organizada, foi adaptada a estrutura criada anteriormente, ***ModelGroup***. Ao contrário do que foi apresentado na fase anterior, agora cada modelo é guardado individualmente, assim como a informação referente a subgrupos, à matriz de transformações, à ordem das transformações (temporais), rotações e translações que devem ser aplicadas a cada grupo e correspondente sub grupos.

```
ModelGroup::ModelGroup(std::vector<Model> models,
                       std::vector<ModelGroup> modelSubGroup,
                       glm::mat4 transformations,
                       std::vector<Rotations> rotations,
                       std::vector<Translations> translations,
                       std::vector<TimeTransform> orderOfTransformations) {
  this->models = models;
  this->subgroups = modelGroup;
  this->transformations = transformations;
  this->rotations = rotations;
  this->translates = translations;
  this->order = orderOfTransformations;
}
```

Listing 1: Estrutura de dados: ModelGroup

2.1.2. Model

Conforme referido anteriormente, optámos por seguir uma abordagem de separação de responsabilidades, criando uma classe dedicada à representação de cada modelo utilizado na cena. Esta decisão teve como principal objetivo simplificar a implementação da lógica associada aos Vertex Buffer Objects (VBOs), tornando o desenvolvimento mais modular e facilitando tanto o carregamento como o desenho dos modelos, em conformidade com os objetivos desta fase.

A classe ***Model*** é, portanto, responsável por armazenar todos os pontos a serem enviados para a placa gráfica (*GPU*), os respetivos índices — de forma a evitar a duplicação desnecessária de vértices —, o estado de inicialização do modelo, o nome do ficheiro de origem, bem como todos os pontos que o compõem.

```
Model::Model(std::string modelFileName, std::vector<Point> controlPoints) {
  this->modelFileName = modelFileName;
  this->modelIdentifier = counter;
  this->vertexBufferObject = generateVBOs(controlPoints);
  this->indexBufferObject = generateIBOs(controlPoints, this->vertexBufferObject);
  this->isInitialized = false;
  this->controlPoints = controlPoints;
  counter++;
}
```

Listing 2: Estrutura de dados: Model

2.1.3. VBOs (Vertex Buffer Objects) & IBOs (Index Buffer Objects)

Como parte dos objetivos desta fase, após o processamento de cada modelo , é necessário criar os respetivos Vertex Buffer Objects (VBOs) e Index Buffer Objects (IBOs). Estes buffers são gerados automaticamente durante a construção do modelo, através das funções `generateVBOs()` e `generateIBOs()`, que alocam e preenchem os buffers com os dados correspondentes ao modelo.

A seguir, detalhamos o funcionamento de cada uma destas operações fundamentais:

2.1.3.1. Função `generateVBOs()`

Esta função é responsável por criar um Vertex Buffer Object (VBO), eliminando vértices duplicados do modelo . O VBO resultante contém apenas vértices únicos, o que é essencial para uma renderização eficiente e para evitar a sobrecarga na GPU.

Lógica de implementação:
1. Recebe um vetor de vértices como input
2. Utiliza uma tabela de *hash* para detetar vértices duplicados
3. Constrói um novo vetor contendo apenas vértices únicos
4. Preserva a ordem de ocorrência dos vértices originais

```
std::vector<Point> generateVBOs(const std::vector<Point>& input_vertices) {
  std::vector<Point> unique_vertices; // VBO resultante com vértices únicos
    std::unordered_map<Point, int, PointHash> vertex_to_index_map; // Mapeamento
vértice→índice

  for (const Point& vertex : input_vertices) {
    if (vertex_to_index_map.find(vertex) == vertex_to_index_map.end()) { // Vértice não
encontrado
      vertex_to_index_map[vertex] = unique_vertices.size(); // Regista novo vértice
      unique_vertices.push_back(vertex); // Adiciona ao buffer
    }
  }
  return unique_vertices;
}
```

Listing 3: Função generateVBOs

Se, por exemplo, o vetor de pontos que recebermos corresponder a:

```
[
-1, -0.5, 0.90   -----> (0)

1, 0.5,  1       -----> (1)

-1, -0.5, 0.90   -----> (já registado na tabela de hash)

0.60, -0.65, 1   -----> (2)

-0.5, 1, 2       -----> (3)

1, 0.5, 1        -----> (já registado na tabela de hash)

]
```

Após a aplicação da função ***generateVBOs***, o vetor de pontos é reduzido, removendo os pontos duplicados. A função mantém apenas a primeira ocorrência de cada ponto, resultando no seguinte vetor:

```
[
-1.0, -0.5, 0.90

1.0, 0.5, 1.0

0.60, -0.65, 1.0

-0.5, 1.0, 2.0

]
```

2.1.3.2. Função `generateIBOs()`

Esta função é responsável por criar um Index Buffer Object (IBO), que mapeia os vértices originais para seus índices no VBO. Isso permite reutilizar vértices compartilhados entre múltiplos triângulos ou faces.

Lógica de implementação:
1. Recebe:
   - O vetor original de vértices (`points`)
   - O *VBO* processado (com vértices únicos)
2. Cria um mapeamento vértice→índice baseado no *VBO*
3. Gera um vetor de índices que referencia os vértices no *VBO*

```
std::vector<unsigned int> generateIBO(const std::vector<Point>& originalPoints,
                                      const std::vector<Point>& uniqueVertices) {
  std::vector<unsigned int> indexBuffer;
  std::unordered_map<Point, int, PointHash> vertexToIndex;

  // Associa cada vértice único ao seu índice no VBO
  for (size_t i = 0; i < uniqueVertices.size(); ++i) {
    vertexToIndex[uniqueVertices[i]] = i;
  }

  // Constrói o IBO com base na ordem dos pontos originais
  for (const Point& vertex : originalPoints) {
    indexBuffer.push_back(vertexToIndex[vertex]);
  }

  return indexBuffer;
}
**
```

Listing 4: Função generateIBOs

Se, por exemplo, o vetor de pontos que recebermos corresponder a:

```
[
-1, -0.5, 0.90   -----> (0)

1, 0.5,  1       -----> (1)

-1, -0.5, 0.90   -----> (já registado na tabela de hash)

0.60, -0.65, 1   -----> (2)

-0.5, 1, 2       -----> (3)

1, 0.5, 1        -----> (já registado na tabela de hash)

]
```

Após a aplicação da função **generateIBOs**, o vetor de **IBO** (Index Buffer Object) passa a conter os índices correspondentes a cada vértice do modelo, referindo-se à respetiva posição desses vértices no **VBO** (Vertex Buffer Object). Ou seja, em vez de repetir vértices, o modelo é agora representado de forma mais eficiente através de índices que apontam para os vértices únicos armazenados no **VBO**. Com base nos pontos apresentados anteriormente, o vetor de **IBO** (**Index Buffer Object**) resultante é:

```
[0,1,0,2,3,1]
```

2.1.4. Desenho dos Models

Após a criação dos **VBOs** e **IBOs**, etapa essencial para a correta inicialização dos modelos, é necessário definir duas funções fundamentais: uma para o **setup** dos modelos e outra para o respetivo desenho.

Na fase de **setup**, procede-se à geração e ao **bind** dos **vertex buffers** e **index buffers**, de forma a enviar a informação necessária para a **GPU**. Já na fase de desenho, é necessário verificar se o **setup** do modelo já foi realizado.

Caso afirmativo, utiliza-se as funções definidas no **GLUT** para desenhar o modelo. Caso contrário, realiza-se o **setup** e só depois se procede ao desenho.

```
void Model::draw() {
  if (!this->isInitialized) {
    setupModel();
    this->isInitialized = true;
  }

  glColor3f(1.0f, 1.0f, 1.0f);
  glBindBuffer(GL_ARRAY_BUFFER, this->glVBO);
  glVertexPointer(3, GL_FLOAT, 0, 0);

  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->glIBO);
  glDrawElements(GL_TRIANGLES, this->indexBuffer.size(), GL_UNSIGNED_INT, 0);
}

void Model::setup() {
  std::vector<float> vertexData = vPointstoFloats(this->vertexBuffer);

  // Gerar e associar o Vertex Buffer Object (VBO)
  glGenBuffers(1, &this->glVBO);
  glBindBuffer(GL_ARRAY_BUFFER, this->glVBO);
  glBufferData(GL_ARRAY_BUFFER, sizeof(float) * vertexData.size(), vertexData.data(),
               GL_STATIC_DRAW);

  // Gerar e associar o Index Buffer Object (IBO)
  glGenBuffers(1, &this->glIBO);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->glIBO);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * this->indexBuffer.size(),
               this->indexBuffer.data(), GL_STATIC_DRAW);
}
```

Listing 5: Desenho e Setup

2.1.5. Transformações Temporais (Translações e Rotações)

Um dos requisitos obrigatórios desta fase consistia na correta interpretação e implementação de um novo tipo de transformações: transformações temporais, que ocorrem de forma contínua ao longo do tempo, num movimento definido por um período previamente especificado pelo utilizador. Estas transformações podem assumir a forma de rotações e/ou translações, permitindo criar animações dinâmicas e realistas nos modelos representados na cena. Os ficheiros de input `.xml` passaram a suportar, no campo **transform**, transformações temporais — nomeadamente rotações e translações cuja execução varia ao longo do tempo, conforme os parâmetros definidos pelo utilizador.

```xml
<world>
    <window width="512" height="512" />
    <camera>
   ...
    </camera>
    <group>
    <transform>
      <rotate time="3" x="0" y="1" z="0" />
    </transform>
    <group>
      <transform>
        <translate time = "10" align="true"> <!-- O campo align diz se o objecto deve ser
orientado na curva -->
          <point x = "0" y = "0" z = "4" />
          <point x = "4" y = "0" z = "0" />
          <point x = "0" y = "0" z = "-4" />
          <point x = "-4" y = "10" z = "0" />
        </translate>
        ...
      </transform>
      <models>
        ...
      </models>
    </group>
  </group>
</world>
```

Listing 6: Novo formato XML

Assim, ao contrário do que foi implementado na fase anterior, o grupo optou por adotar uma nova estratégia relativamente ao armazenamento das transformações (apenas temporais). Como a matriz de transformações associada a cada grupo já não é estática (devido à natureza temporal das transformações), tornou-se necessário armazenar cada transformação de forma individual. A cada renderização dos modelos, estas transformações são aplicadas sequencialmente à matriz atual, garantindo a correta animação em função do tempo. Ou seja, tal como é possível observar na classe ***ModelGroup***, apenas as transformações que não dependem do tempo são previamente aplicadas à matriz de transformações. Por outro lado, as transformações temporais são armazenadas individualmente, com indicação do respetivo tipo (rotação ou translação) e da ordem em que devem ser aplicadas, uma vez que a sequência destas operações é **relevante** para o resultado final da animação. Assim, o grupo procedeu à criação de duas classes, a ***Rotation*** e a ***Translation*** :

```
class Translation {
 public:
  float duration;                      // Tempo total da translação
  bool alignToPath;                    // Alinhamento ao longo da curva
  std::vector<Point> controlPoints;    // Pontos de controlo da curva
  Point upVector;                      // Vetor de orientação (normalmente eixo Y)
};
class Rotation {
 public:
  float duration;      // Tempo total da rotação
  float axisX;         // Componente X do eixo de rotação
  float axisY;         // Componente Y do eixo de rotação
  float axisZ;         // Componente Z do eixo de rotação
};
```

Listing 7: Transformações

2.1.5.1. Translações Temporais

As translações temporais revelaram-se, sem surpresa, o tipo de transformações mais complexas de implementar corretamente. Para determinar a posição do modelo ao longo do tempo, foi necessário utilizar os parâmetros definidos no construtor, nomeadamente os pontos de controlo da curva e o tempo total da animação. A fim de simplificar o processo, foi fundamental a correta definição das funções que manipulam as curvas de *Catmull-Rom*.

Para a implementação desta transformação, foi utilizado o algoritmo de interpolação *Catmull-Rom*, que permite calcular uma posição suave ao longo de uma curva definida por pontos de controlo. Esta técnica permite obter tanto a posição do ponto numa determinada fração do tempo, como a direção (derivada) da curva nesse instante.

Quando existem mais de quatro pontos de controlo, a trajetória é calculada dinamicamente a partir dos quatro pontos de controlo mais próximos do instante atual da animação. Isto permite que o percurso seja contínuo e suave ao longo de toda a curva.

A posição final do ponto e a sua derivada são obtidas através da aplicação da seguinte fórmula matricial:

$$\mathcal{C}(P_0, P_1, P_2, P_3) = \begin{pmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

$$\mathcal{P}(t, P_0, P_1, P_2, P_3) = \mathcal{C}(P_0, P_1, P_2, P_3)(t^3 \;\; t^2 \;\; t \;\; 1)$$

$$\mathcal{P}'(t, P_0, P_1, P_2, P_3) = \mathcal{C}(P_0, P_1, P_2, P_3)(3t^2 \;\; 2t \;\; 1 \;\; 0)$$

Sendo P_0, P_1, P_2, P_3 os pontos de controlo utilizados para a interpolação, e $t[0.0, 1.0]$ o valor de tempo normalizado correspondente à posição atual ao longo da curva, calculamos:

- $P(t)$: a posição do modelo nesse instante;
- $P'(t)$: a derivada da curva nesse ponto, que representa a direção da trajetória.

Com a posição $P(t)$ calculada, a transformação correspondente consiste numa simples matriz de translação, que posiciona o modelo ao longo da curva animada.

Opcionalmente, esta transformação pode incluir um parâmetro de alinhamento à curva. Quando esse parâmetro está ativo, a derivada $P'(t)$ é utilizada para orientar o modelo de forma a que este acompanhe a direção da trajetória. Este processo envolve:

- a normalização do vetor derivada (direção *forward*);
- o cálculo do vetor *right* através do produto vetorial com o eixo Y fornecido;
- a obtenção do vetor *up* também por produto vetorial, garantindo uma base ortonormal.

Esta base é então usada para construir uma matriz de rotação, que, quando o parâmetro de alinhamento está ativado, é aplicada em conjunto com a matriz de translação.

A matriz de rotação é construída a partir da base ortonormal formada pelos vetores `forward`, `right` e `up`, definidos com base na derivada da curva e no eixo Y fornecido. Esta matriz representa a orientação do modelo ao longo da trajetória.

Por fim, esta matriz de rotação é multiplicada pela matriz de translação previamente calculada, de forma a aplicar simultaneamente a posição e a orientação correta do modelo ao longo da curva, garantindo um movimento realista e coerente com o percurso definido.

2.1.5.1.1. Aplicação do Algoritmo de Catmull-Rom

Para a correta aplicação do algoritmo, o grupo implementou a função `applyTranslations`, que recebe o tempo decorrido desde o início da animação. Primeiro, calcula-se a fração de tempo (`global_t`) correspondente ao ponto atual da animação, dividindo o tempo decorrido pelo tempo total definido.

Com essa fração, usa-se a função `catmullRomPosition` para calcular a posição do modelo e a direção da tangente à curva nesse instante. Esta função aplica a matriz de Catmull-Rom sobre os pontos de controlo para gerar uma interpolação suave entre os mesmos, retornando dois vetores: a posição e a direção da curva naquele momento.

De seguida, é aplicada uma translação para mover o modelo até essa posição, com `glTranslatef`.

Caso a opção de alinhamento (`align`) esteja ativa, são realizados cálculos adicionais para orientar o modelo com base na direção da curva. Para isso:

- A tangente é normalizada para obter o vetor "forward".
- Calcula-se o vetor "right" como o produto vetorial entre o vetor "forward" e o vetor auxiliar fornecido (`y_axis`).
- Depois, calcula-se o vetor "up" como o produto vetorial entre "right" e "forward", formando assim um sistema de coordenadas ortogonal.

Desta forma, o modelo move-se suavemente ao longo da curva definida e, se necessário, orienta-se de forma coerente com a direção do movimento.

```
std::pair<Point, Point> catmullRomPosition(std::vector<Point> controlPoints, float
globalTime) {
  const std::array<std::array<float, 4>, 4> catmullMatrix = catmoll_rom_matrix;

  float scaledTime = globalTime * controlPoints.size();
  int segmentIndex = (int)floor(scaledTime);
  float localT = scaledTime - segmentIndex;

  int baseIndex = segmentIndex + controlPoints.size() - 1;
  Point p0 = controlPoints[(baseIndex + 0) % controlPoints.size()];
  Point p1 = controlPoints[(baseIndex + 1) % controlPoints.size()];
  Point p2 = controlPoints[(baseIndex + 2) % controlPoints.size()];
  Point p3 = controlPoints[(baseIndex + 3) % controlPoints.size()];

  const std::array<std::array<float, 4>, 3> points = {{
      {p0.x, p1.x, p2.x, p3.x},
      {p0.y, p1.y, p2.y, p3.y},
      {p0.z, p1.z, p2.z, p3.z},
  }};

  const std::array<float, 4> timeVec = {localT * localT * localT, localT * localT, localT,
1};
  const std::array<float, 4> derivVec = {3 * localT * localT, 2 * localT, 1, 0};

  std::array<float, 3> pos{};
  std::array<float, 3> deriv{};

  for (size_t axis = 0; axis < 3; ++axis) {
    std::array<float, 4> coeffs{};

    for (size_t row = 0; row < 4; ++row) {
      for (size_t col = 0; col < 4; ++col) {
        coeffs[row] += points[axis][col] * catmullMatrix[row][col];
      }
    }

    for (size_t i = 0; i < 4; ++i) {
      pos[axis] += timeVec[i] * coeffs[i];
      deriv[axis] += derivVec[i] * coeffs[i];
    }
  }

  return {Point(pos[0], pos[1], pos[2]), Point(deriv[0], deriv[1], deriv[2])};
}
void Translations::applyTranslations(float elapsedTime) {
  if (this->time == 0) return;

  float globalT = elapsedTime / this->time;
  auto [position, tangent] = catmullRomPosition(this->curvePoints, globalT);

  glTranslatef(position.x, position.y, position.z);

  if (this->align) {
    Point forward = tangent.normalize();
    Point right = Point(forward).cross(this->y_axis).normalize();
    Point up = Point(right).cross(forward).normalize();

    glMultMatrixf(rotationMatrix(forward, up, right).data());
  }
}
```

Listing 8: Algoritmo Catmull-Rom

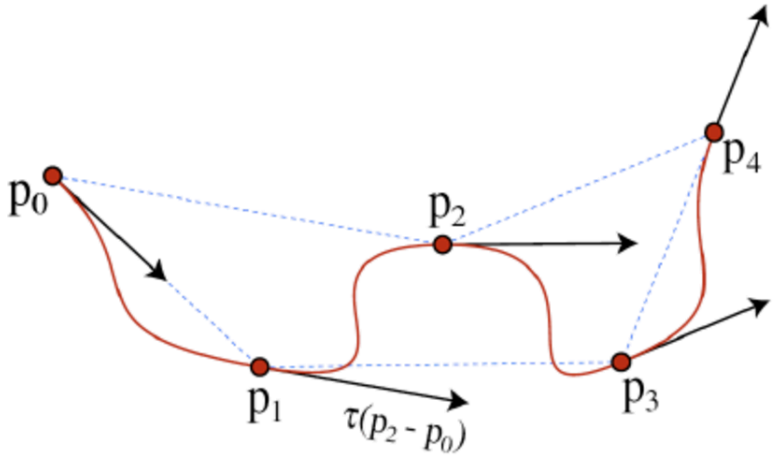

Figura 1: Catmull-Rom spline

2.1.5.2. Rotações Temporais

A outra transformação temporal adicionada foi a **rotação temporal**, que é significativamente mais simples do que a transformação de translação ao longo de curvas. Esta rotação permite que um grupo se mova em torno de um **eixo arbitrário**, completando uma rotação total num intervalo de tempo definido pelo utilizador.

A implementação foi relativamente direta, uma vez que reutilizámos a função de cálculo da matriz de rotação já desenvolvida na fase anterior. O único passo adicional foi calcular o ângulo de rotação atual com base no tempo decorrido.

A fórmula utilizada é a seguinte:

$$\mathcal{R}_{\mathcal{T}}(t, \Delta t, x, y, z) = \text{calculateRotation}\left(\frac{2\pi t}{\Delta t}, x, y, z\right)$$

Onde:

- Δt representa o tempo necessário para uma rotação completa,
- t é o tempo atual,
- x, y, z definem o eixo de rotação,
- calculateRotation é a função que calcula a matriz de rotação para um dado ângulo e eixo (já definida anteriormente).

Desta forma, garantimos uma rotação contínua, suave e parametrizável ao longo do tempo.

2.1.5.2.1. Aplicação do Algoritmo

A função *applyRotation* aplica uma rotação proporcional ao tempo decorrido. Se o tempo total de rotação (*time*) for zero, a função termina sem fazer nada. Caso contrário, calcula o ângulo de rotação em função do tempo passado e aplica essa rotação ao objeto, segundo o eixo definido pelos valores x, y e z.

```
void Rotations::applyRotation(float elapsed_time) {
  if (this->time == 0) return;
  float rotation_angle = 360 * (elapsed_time / this->time);
  glRotatef(rotation_angle, this->x, this->y, this->z);
}
```

Listing 9: Algoritmo Rotação Temporal

2.2. *Generator*

Um dos requisitos desta fase consistiu na introdução de uma nova primitiva gráfica. Para tal, foi implementado o suporte à interpretação de *patches* de superfícies de ***Bézier***, os quais são convertidos num ficheiro `.3d` contendo os triângulos correspondentes ao modelo. Este ficheiro pode depois ser utilizado para a respetiva representação gráfica.

2.2.1. *Patches Bézier*

Para gerar modelos a partir de ficheiros `.patch`, o processo é semelhante ao utilizado para outras primitivas gráficas. Assim, basta executar o *generator* com o comando: `./generator patch <ficheiro.patch> <número_divisões> <output.3d>`. O segundo parâmetro, que se refere ao número de divisões, define quantas subdivisões terá cada *Bezier Patch*. Quanto maior for esse número, maior será o nível de detalhe do modelo gerado. No entanto, é importante ter em mente que valores elevados de divisões podem gerar um número excessivo de vértices, o que pode prejudicar a performance de renderização sem oferecer um ganho significativo na qualidade visual.[1]

Em seguida, é apresentado um exemplo de como deve ser o ficheiro `.patch` passado pelo utilizador:

```
2 # numero de patches bezier
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 # indice dos pontos de cada patch
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31
32 # numero de pontos
1.0, 0.0, 2.0
2.0, 0.0, 2.0
2.0, 1.0, 2.0
1.0, 1.0, 2.0
...
1.5, 0.5, 2.5
```

Listing 10: Novo formato XML

2.2.1.1.1. Geração dos modelos no ficheiro *.3d*

A geração dos modelos ocorre ao nível de cada *patch*, onde, para cada um deles, calculamos os respetivos pontos com base na seguinte fórmula paramétrica:

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

Neste contexto, P_{ij} representa os pontos de controlo do dado *patch*, u, $v \in [0,1]$ e

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

Os pontos são calculados por cada componente x, y, z.

A matriz M utilizada na fórmula anterior resulta dos coeficientes dos polinómios de Bernstein de grau 3 — apropriados para curvas definidas por 4 pontos de controlo.

Dado que os parâmetros u e v variam no intervalo $[0,1]$, os pontos da superfície são gerados iterativamente com incrementos de $\frac{1}{\text{tesselation}}$, sendo este valor fornecido como argumento ao gerador. Ou seja, maior número de divisões implica uma malha mais densa e detalhada.

[1] Por exemplo, um valor acima de 10 pode causar esse efeito indesejado.

Importa referir que o cálculo de MAM^T, onde A representa a matriz dos pontos de controlo do *patch*, é realizado apenas uma vez por *patch*, uma vez que os seus pontos não se alteram ao longo da tesselação.

2.2.1.1.2. Algoritmo utilizado

Após o grupo tentar perceber qual seria a melhor abordagem para a geração de superfícies a partir de *Bezier Patches*, optámos por simplificar a lógica clássica baseada em multiplicações matriciais. Assim, recorremos à **interpolação de Bézier**, baseada diretamente nos polinómios de *Bernstein* de grau 3.

Esta opção revelou-se mais intuitiva e direta de implementar, permitindo-nos manter o foco na correta geração dos triângulos sem nos preocuparmos com detalhes mais matemáticos da álgebra linear.

Concretamente, a função `patchTriangles` é responsável por:

- Ler os dados do ficheiro `.patch`, incluindo o número de *patches*, os índices dos pontos de controlo de cada um e as suas coordenadas;
- Para cada *patch*, aplicar a função `evaluateBezierPatch` a uma grelha regular de valores (u, v), com incrementos definidos pelo parâmetro de tesselação;
- Com os pontos calculados, formar dois triângulos por cada célula da grelha, compondo assim a malha da superfície.

A função `evaluateBezierPatch` calcula os pontos da superfície aplicando diretamente a fórmula da curva de *Bézier*, com base em combinações dos polinómios de *Bernstein*.

Esta escolha permite evitar a complexidade da abordagem matricial, mantendo uma implementação de fácil compreensão.

```
  std::vector<Point> finalPoints;

  for (size_t patchIdx = 0; patchIdx < patchCount; ++patchIdx) {
    std::vector<Point> currentPatchPoints;
    for (size_t pointIdx = 0; pointIdx < 16; ++pointIdx) {
      currentPatchPoints.push_back(controlPoints[patchIndices[patchIdx][pointIdx]]);
    }

    for (int uStep = 0; uStep < tessellationLevel; ++uStep) {
      for (int vStep = 0; vStep < tessellationLevel; ++vStep) {
        float u1 = (float)uStep / tessellationLevel;
        float v1 = (float)vStep / tessellationLevel;
        float u2 = (float)(uStep + 1) / tessellationLevel;
        float v2 = (float)(vStep + 1) / tessellationLevel;

        Point p1 = evaluateBezierPatch(currentPatchPoints, u1, v1);
        Point p2 = evaluateBezierPatch(currentPatchPoints, u2, v1);
        Point p3 = evaluateBezierPatch(currentPatchPoints, u1, v2);
        Point p4 = evaluateBezierPatch(currentPatchPoints, u2, v2);

        finalPoints.push_back(p1);
        finalPoints.push_back(p3);
        finalPoints.push_back(p2);

        finalPoints.push_back(p2);
        finalPoints.push_back(p3);
        finalPoints.push_back(p4);
      }
    }
  }

  return finalPoints;
}

Point evaluateBezierPatch(const std::vector<Point>& patchControlPoints, float u, float v)
{
  Point result(0, 0, 0);
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
      float bernsteinCoeff = bernsteinPolynomial(i, u) * bernsteinPolynomial(j, v);
      result.x += patchControlPoints[i * 4 + j].x * bernsteinCoeff;
      result.y += patchControlPoints[i * 4 + j].y * bernsteinCoeff;
      result.z += patchControlPoints[i * 4 + j].z * bernsteinCoeff;
    }
  }

  return result;
}

float bernsteinPolynomial(int index, float t) {
  switch (index) {
    case 0:
      return pow(1 - t, 3);
    case 1:
      return 3 * t * pow(1 - t, 2);
    case 2:
      return 3 * pow(t, 2) * (1 - t);
    case 3:
      return pow(t, 3);
    default:
      return 0;
  }
}
```

Listing 11: Interpolação Bezier

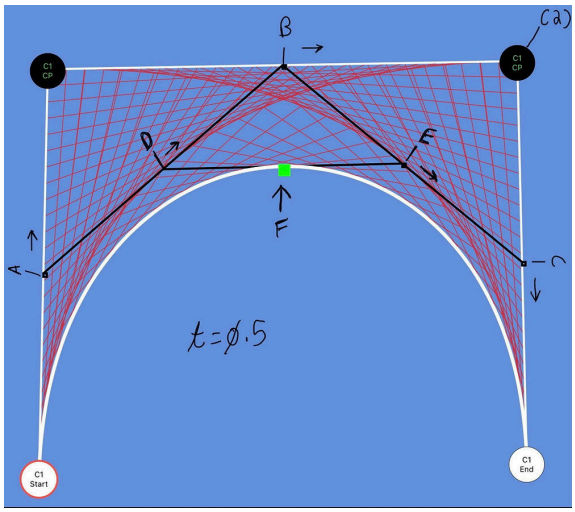

Figura 2: Curva de Bezier, com 0.5 de tesselação

2.3. Testes Realizados

Após as alterações necessárias no ***engine***, nomeadamente no que diz respeito à introdução de transformações dependentes do tempo, e no ***generator***, no que toca à geração de modelos provenientes de *patches*, procedemos à execução de uma série de testes disponibilizados pelos docentes com o objetivo de validar a correta interpretação e renderização dos modelos. Estes testes encontram-se disponíveis na pasta `scenes/testes`.

É possível encontrar também em `scenes/videos` um ficheiro `.mkv` que ilustra os resultados obtidos com as transformações temporais.

Em seguida, apresentam-se os resultados obtidos:

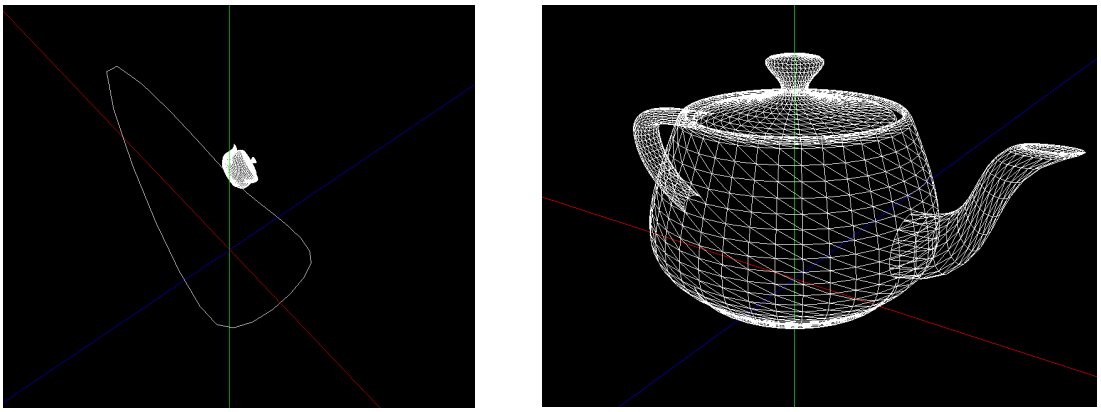
Figura 3: Resultados dos testes disponibilizados pelos docentes

3. Sistema Solar

Um dos requisitos para esta terceira fase foi dar *upgrade* à cena referente ao sistema solar apresentada na fase anterior. Desta vez, foi necessário escalar todo o Sistema Solar de forma a ser possível observar a translação dos planetas em torno do Sol, bem como a rotação do próprio Sol e dos planetas sobre si mesmos. Foi também adicionado um cometa que percorre uma trajetória elíptica, passando perto do Sol e aproximando-se da órbita de Úrano. A trajetória foi definida com base nas fórmulas de uma elipse, garantindo um movimento suave e natural.

3.1. Rotação, Translação e Inclinação dos Planetas

Com os *datasets* dos planetas do sistema solar usados na fase anterior (*Eyes on the Solar System* , *Devstronomy*), já tínhamos disponível a informação necessária para tempos de rotação e translação de todos os corpos celestes. Agora, com a introdução de transformações dependentes do tempo, conseguimos aplicar essas rotações e translações de forma dinâmica, fazendo com que os planetas e satélites se movam e rodem ao longo do tempo.

Para aplicar a **inclinação axial** de cada planeta, utilizámos uma abordagem simples: aplicámos uma rotação em torno do eixo x, com o ângulo de inclinação fornecido nos datasets, o que contribui para uma visualização ainda mais fiel do movimento orbital.

Para a **rotação** dos planetas em torno do seu eixo, aplicámos uma rotação temporal baseada no tempo de rotação real de cada planeta. No entanto, como estes tempos são demasiado longos, foi necessário convertê-los para valores mais agradáveis, fazendo a transformação de horas para segundos. No caso da Terra, por exemplo, como o tempo de rotação é de 24 horas, usamos 24 segundos como valor visual. Para os restantes planetas, a lógica foi a mesma.

Para a **translação** dos planetas em torno do Sol, recorremos também à informação presente nos *datasets* fornecidos, com o intuito de garantir um maior realismo.

Desta forma, ao contrário da fase anterior — onde a translação dos planetas em torno do Sol era determinada apenas pela respetiva distância —, recorremos agora a cerca de 40 pontos de controlo por planeta ou lua. Esta nova abordagem permite descrever trajetórias elípticas mais próximas das reais, resultando num movimento mais contínuo, natural e visualmente apelativo.

Os pontos de controlo foram gerados através de um *script* em *Python* que, tendo em conta a distância do planeta ou lua ao Sol, calcula automaticamente os 40 pontos pelos quais cada corpo celeste deverá transitar. Esta solução permitiu automatizar o processo e garantir consistência nas trajetórias elípticas definidas para cada órbita. O *script* pode ser encontrado em `scripts/controlPoints.py`

Esta abordagem permitiu definir trajetórias mais suaves e visivelmente mais apelativas, conferindo maior credibilidade e qualidade visual à simulação.

3.1.1. Cintura de Asteroides

Relativamente aos 1200 asteroides gerados na fase anterior, com o intuito de aumentar o realismo do sistema solar, optámos por aplicar um movimento de translação aos mesmos. Para simplificar a

implementação, todos os asteroides seguem a mesma trajetória e percorrem-na no mesmo intervalo de tempo.

3.2. Cometa de Halley

Um dos requisitos para esta fase foi também a adição de um cometa ao sistema. Para isso, incluímos o cometa de *Halley*, conhecido por ser visível a olho nu.

A sua trajetória foi gerada com base em 40 pontos de controlo, utilizando as fórmulas de uma elipse, de forma a representar o seu percurso característico em torno do Sol. Este movimento ajuda a enriquecer a representação do sistema solar, tornando-o mais completo e realista.

3.3. Resultados Obtidos

Após a incorporação de toda a informação adicional no ficheiro `scenes/solar_system.xml`, foi possível observar uma representação significativamente mais realista e dinâmica do sistema solar.

Como demonstração do resultado final, incluímos um vídeo disponível em `scenes/videos`, onde é possível visualizar o movimento completo do sistema solar em tempo real.

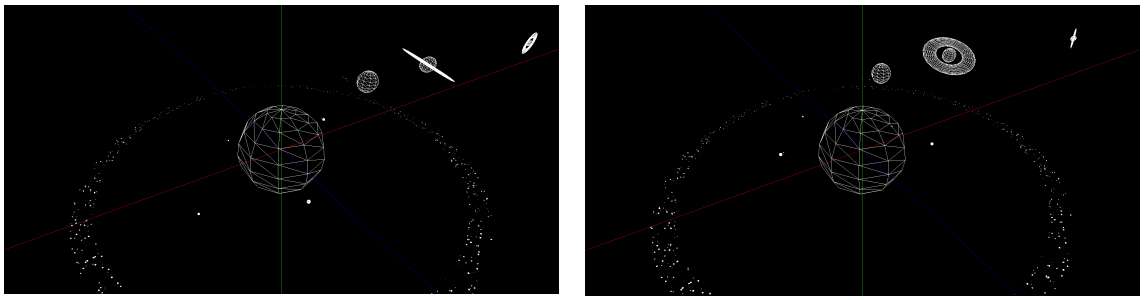

Figura 4: Sistema solar

4. Conclusões e Trabalho Futuro

O desenvolvimento desta terceira fase revelou-se bastante positivo, e o grupo conseguiu concluir com sucesso tudo o que tinha planeado. A introdução dos *VBOs* e *IBOs* permitiu um avanço significativo no desempenho, e de um modo geral, o projeto está a avançar de acordo com o planeado. A implementação das transformações temporais, a geração de modelos com *VBOs* com índices e a criação de modelos com *Bezier Patches* foram realizadas com sucesso. Assim, o grupo está muito ansioso para ver o resultado da fase final e tem adquirido bastante conhecimento, no que toca ao mundo da computação gráfica.

Bibliografia

- [1] «vcpkg - Open source C/C++ dependency manager from Microsoft». Disponível em: https://vcpkg.io/en/
- [2] «CMake». Disponível em: https://cmake.org/
- [3] «OpenGL - The Industry Standard for High Performance Graphics». Disponível em: https://www.opengl.org/
- [4] «GitHub - Build software better, together». Disponível em: https://github.com/
- [5] Song Ho Ahn, «OpenGL Tutorials». Disponível em: https://www.songho.ca/opengl/
- [6] «Eyes on the Solar System». NASA. Disponível em: https://eyes.nasa.gov/apps/solar-system/#/home
- [7] «Devstronomy - Planetary Data Reference». Disponível em: https://devstronomy.martinovo.net/
- [8] «Bezier Interpolation». Medium. Disponível em: https://medium.com/@adrian_cooney/bezier-interpolation-13b68563313a
- [9] «Catmull-Rom Spline Algorithm». MVPs. Disponível em: https://www.mvps.org/directx/articles/catmull/
- [10] «Catmull-Rom Splines in Plain English». Andrew Hung Blog. Disponível em: https://andrewhungblog.wordpress.com/2017/03/03/catmull-rom-splines-in-plain-english/

Definição de Ponto

A estrutura `Ponto` representa uma coordenada num espaço tridimensional e é definida como:

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

  Point(float x_val = 0.0f, float y_val = 0.0f, float z_val = 0.0f)
      : x(x_val), y(y_val), z(z_val) {}

} Point;
```

ModelGroup

```
ModelGroup::ModelGroup(std::vector<Model> models,
                       std::vector<ModelGroup> modelSubGroup,
                       glm::mat4 transformations,
                       std::vector<Rotations> rotations,
                       std::vector<Translations> translations,
                       std::vector<TimeTransform> orderOfTransformations) {
  this->models = models;
  this->subgroups = modelGroup;
  this->transformations = transformations;
  this->rotations = rotations;
  this->translates = translations;
  this->order = orderOfTransformations;
}
```

Model

```
Model::Model(std::string modelFileName, std::vector<Point> controlPoints) {
  this->modelFileName = modelFileName;
  this->modelIdentifier = counter;
  this->vertexBufferObject = generateVBOs(controlPoints);
  this->indexBufferObject = generateIBOs(controlPoints, this->vertexBufferObject);
  this->isInitialized = false;
  this->controlPoints = controlPoints;
  counter++;
}
```