Abril 23, 2024

Relatório CG - Fase 3 - Grupo 24

Guilherme Barbosa (a100695)
Ivan Ribeiro (a100538)
Nuno Aguiar (a100480)
Pedro Ferreira (a100709)

Sumário

1) Introdução

Este documento apresenta as decisões tomadas e a implementação seguida ao longo da terceira fase do projeto desta UC. São descritas as modificações efetuadas sobre a hierarquia de classes em relação à fase anterior, as alterações na componente *XML Parser*, a criação das *Bezier Patches*, de curvas de *Catmull-Rom* e de *VBOs*.

2) Mudanças em relação a fases anteriores

As curvas de *Catmull-Rom* tiveram peso significativo nas modificações observadas em relação à fase anterior, nomeadamente na forma de calcular as transformações aplicadas aos pontos de cada objeto da cena e na aparição de novas tags e atributos nos ficheiros xml.

2.1) Hierarquia de classes

Na fase 2, a hierarquia de classes seguia uma estrutura em árvore n-ária em que cada nodo possuía o conjunto de pontos das primitivas contidas numa tag *models* e a matriz pré-computada das transformações efetuadas sobre esse mesmo conjunto de pontos, sendo que esta solução assumia que as matrizes de transformação eram estáticas. Porém, com a introdução das curvas de *Catmull-Rom* as matrizes de transformação passaram a ser dependentes do tempo, pelo que a hierarquia de classes teve de ser atualizada, retirando-se a pré-computação das matrizes de transformação.

Desejando aproveitar a estrutura em árvore n-ária já desenvolvida, decidiu-se alterar o conteúdo de cada nodo, passando este a conter:

- um conjunto de matrizes de transformação (translações, rotações e escalas);
- um conjunto em que cada elemento é o conjunto de pontos de uma primitiva;

Em cada frame, para cada nodo, é calculada a matriz de transformação a aplicar ao conjunto de pontos das primitivas, multiplicando as matrizes do nodo e as do nodo pai. O novo aspeto da hierarquia de classes é o seguinte:

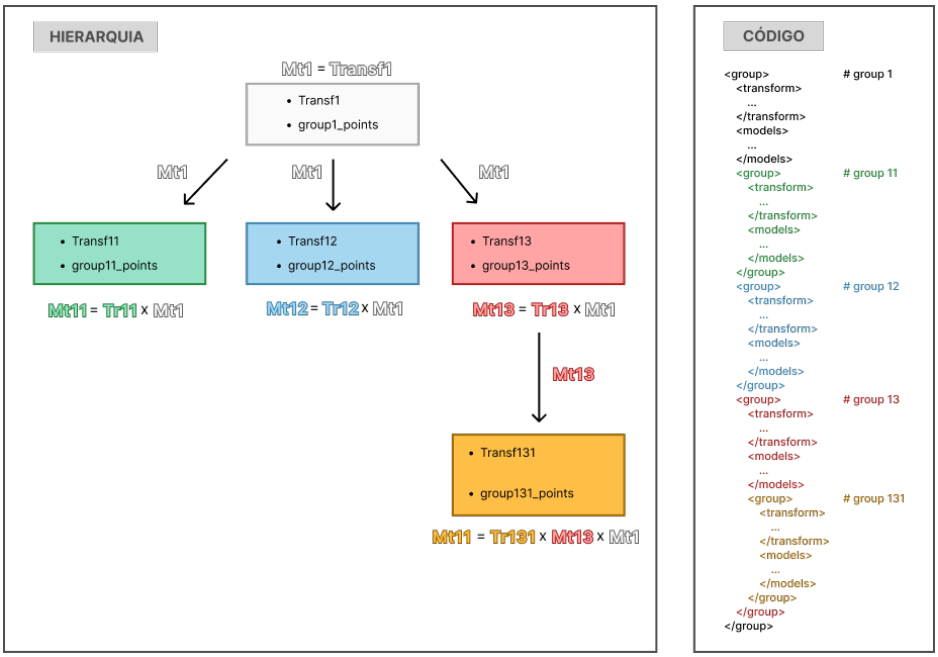

Figura 1: Árvore que representa a nova hierarquia de objetos

2.2) XML Parser

Com o aparecimento de novos atributos em tags xml já reconhecidas e a mudança na hierarquia de classes, foi necessário alterar ligeiramente esta componente. Em primeiro lugar, conforme já referido, as transformações de cada tag *group* deixaram de ser multiplicadas conforme o documento xml vai sendo lido, passando a ser guardadas num vetor no respetivo nodo da hierarquia de classes. Em segundo lugar, foi adicionado o reconhecimento de novas tags e atributos relativos a curvas de *Catmull-Rom*: a tag *Point*, que indica um ponto constituinte da curva e os atributos *time* e *aligned*, sendo este último encontrado apenas em tags *Translate*. Além disso, para permitir um alinhamento em relação aos eixos mais flexível, o grupo decidiu criar vários atributos para a tag *Translate* que indicam qual o vetor up e o vetor de alinhamento do objeto.

O aspeto atual da componente é descrito de seguida:

```
xml_doc = TinyXML2.read(xml_file)

# Base xml tags in each document
window_config = xml_doc.child_tag("window")
camera_config = xml_doc.child_tag("camera")
group = xml_doc.child_tag("groups")

# Read group tags (Transformation é uma abstração de uma matrix 4x4 'mat4' provida pelo
glm)
engine_obj = parseEngineObject(group, Transformation());

def parseEngineObject(group : XML_Tag, transformation : Transformation) -> EngineObject
  obj_points = [] # each element is an array of vertices
  child_objects = []

  for tag in group:
    if tag == "transformation":
      # read and construct matrix for rotation/translation/scale, appending it to group
transformations
      transformations.append(parseTransformation(tag))

    if tag == "models":
      obj_points.append(parseModels(tag))

    if tag == group:
      child_objects.append(parseEngineObject(tag, Transformation()))

  return EngineObject(transformation, obj_points, child_objects)

def parseModels(models):
  vertices = []

  for model in models:
    # read file indicated by model and retrieve its vertices
    vertices.append(parseVertex(model))

  return vertices
```

```
def parseTransformation(transformTag):

  if transformTag == translate:
    t = Translate(translateTime, alignCurve, translateVector, isCurve, controlPoints,
upVector, alignVector)

  else if transformTag == rotate:
    t = Rotate(angle, time, rotationVector)

  else:
    t = Scale(scaleVector)

  return t
```

3) Novas implementações

O foco da terceira fase é a implementação das *Bézier Patches*, das curvas de *Catmull-Rom* e a utilização de *VBOs*. Para tal, foram realizadas adições ao *generator* e à *engine*, as quais exigiram a tomada de decisões apresentadas de seguida.

3.1) Bézier Patches

As Bézier patches são conjuntos de 16 pontos de controlo dispostos no formato de grelha. Cada 4 pontos definem uma curva de Bézier e, em conjunto, as 4 curvas definem uma superfície de Bézier bicúbica. Combinando várias destas patches é possível criar primitivas tridimensionais complexas. Para a sua construção, utilizámos polinomiais de *Bernstein* (uma curva de Bézier de grau n é uma combinação de polinómios de Bernstein de grau n aplicados aos pontos de controlo). Após especificar o nível de detalhe do objeto final (nível de tesselação), são seguidos os passos:

- Percorrer as curvas tantas vezes quanto o nível de tesselação. Chamemos a esta variável *i*.
- Para cada uma das 4 curvas, calcular o ponto P_n, interpolando a curva em u = i / tesselation_level
- Com os 4 pontos obtidos anteriormente (1 de cada curva), construímos uma nova curva que é interpolada em v = i / tesselation_level, obtendo um ponto P(u, v).
- Com todos os pontos de um dado valor u calculados, imaginando que estes formam uma linha, podemos uni-los à linha anteriormente formada para construir os triângulos que formam a superfície final.

Para interpolar as coordenadas (x,y,z) de um ponto numa dada curva, considerando os pontos de controlo da curva P_0,P_1,P_2,P_3 e t o valor de interpolação da curva ($0 \leq t \leq 1$), podendo t assumir o valor de u ou v, dependendo da curva considerada na iteração, temos:

$$P(t) = (1-t)^3 * P_0 + 3.0 * (1-t)^2 * t * P_1 + 3.0 * (1-t) * t^2 * P^2 + (1-t)^3 * P_3$$

Uma possível alternativa para o cálculo de cada ponto P(u, v), sem recorrer a interpolação, envolve a utilização de multiplicações matriciais, seguindo, conforme lecionado, a fórmula:

$$\mathrm{P(u,\ v)} = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} * \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} P_{01} & P_{11} & P_{21} & P_{31} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{03} & P_{13} & P_{23} & P_{33} \\ P_{04} & P_{14} & P_{24} & P_{34} \end{pmatrix} * \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} v^3 \\ v^2 \\ v \\ 1 \end{pmatrix}$$

em que P_{i_j} representa o ponto j da curva i.

O problema que o grupo encontrou com este último método foi o de implementar a matriz:

$$\begin{pmatrix} P_{01} & P_{11} & P_{21} & P_{31} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{03} & P_{13} & P_{23} & P_{33} \\ P_{04} & P_{14} & P_{24} & P_{34} \end{pmatrix}.$$

Em aulas posteriores à implementação estar concluída foi exposta a possíbilidade de utilização desta matriz, porém o grupo optou pela abordagem já implementada e que segue a explicação teórica, ou seja, a interpolação dos pontos.

Estas patches foram utilizadas para a criação das primitivas do teapot e das primitivas que constituem o cometa.

3.2) Curvas Catmull-Rom

As curvas de *Catmull-Rom* são definidas por quatro pontos e funcionam de forma semelhante às curvas de *Bézier*. Para as calcular foi necessário desenvolver uma função que, dado um parâmetro t, devolve a posição atual do objeto e a sua derivada, sendo esta última essencial para alinhar os objetos. Primeiro calcula-se o resto da divisão de t pelo tempo em que se deve percorrer a curva completa e dividimos o resultado pelo mesmo tempo. De seguida, selecionamos os pontos que constituem a curva e obtemos o ponto atual e a sua derivada da seguinte forma:

$$\mathrm{P(t)} = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix} * \begin{pmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} P_{01} & P_{02} & P_{03} & P_{04} \\ P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{pmatrix}$$

$$\mathrm{P'(t)} = \begin{pmatrix} 3t^2 & 2t & 1 & 0 \end{pmatrix} * \begin{pmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} P_{01} & P_{02} & P_{03} & P_{04} \\ P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{pmatrix}$$

Para desenhar as curvas utiliza-se um nível de tesselação arbitrário (atualmente assume o valor 100) que simulará a passagem pelas várias posições das curvas, obtendo as suas coordenadas e derivadas. Estas coordenadas são depois desenhadas utilizando VBOs com o modo GL_LINE_LOOP e as derivadas da curva são desenhadas de forma semelhante, passando os valores das coordenadas e coordenadas + derivada para um array que será renderizado com o modo GL_LINES.

Por fim, para alinhar os diversos objetos, é necessário passar quais os vetores *up* e *align* a usar, sendo o valor padrão, respetivamente, (0, 1, 0) e (1, 0, 0), que corresponde ao alinhamento para x positivo. A leitura deste vetores é feita coordenada a coordenada. Caso o up.x seja 1, mas os restantes valores não sejam indicados, o valor de up final será (1, 1, 0), isto é, as coordenadas y e z continuam com o seu valor padrão. O mesmo ocorre no vetor *align*. Ao nível do xml é necessário indicar os atributos upX, upY, upZ, alignX, alignY, alignZ, respetivamente, os quais esperam valores do tipo float. Assume-se que os valores passados são válidos pelo que, se os vetores forem paralelos, o objeto simplesmente não é desenhado, pois a matriz de alinhamento é a matriz nula. A função de alinhamento tem o seguinte aspeto:

```
//valores iniciais
yVector = upVector

matrix alignObject(){

  zVector = normalize(deriv);

  xVector = normalize(cross(zVector, yVector));
```

```
  yVector = normalize(cross(xVector, zVector));

  // Calculate vector perpendicular to align and up vectors
  right = normalize(cross(upVector, alignVector));

  // Update vector up
  upVector = cross(alignVector, right);

  // Create the rotation matrix to align with the specified align and up vectors
  alignMatrix = {
      {   right.x    ,    right.y    ,    right.z    , 0.0f},
      {  upVector. x ,   upVector.y  ,   upVector.z  , 0.0f},
      {-alignVector.x, -alignVector.y, -alignVector.z, 0.0f},
      {     0.0f     ,      0.0f     ,      0.0f     , 1.0f}
    };

  rotMatrix = buildRotMatrix(xVector, yVector, zVector) * alignMatrix;
}


matrix buildRotMatrix(x_axis, y_axis, z_axis){
  return {
    {x_axis.x, y_axis.x, z_axis.x, 0},
    {x_axis.y, y_axis.y, z_axis.y, 0},
    {x_axis.z, y_axis.z, z_axis.z, 0},
    {       0,        0,        0, 1}
  };
}
```

Como é observável no pseudo-código, o cálculo da rotação do objeto por um vetor arbitrário possui duas partes. A primeira consiste em alinhar o objeto segundo o eixo do z, usando para tal a derivada da curva de *Catmull-Rom* naquele ponto. O vetor y inicial possui o mesmo valor do vetor up indicado.

A segunda parte deste processo consiste em calcular o vetor *right* e atualizar o vetor *up*. O vetor *right* é prependicular ao *up* e ao *align* e garante que o objeto se encontra alinhado com a curva. A atualização constante do vetor up garante que o objeto se encontra sempre alinhado verticalmente. Por fim, o vetor *align* é constante para garantir o eixo de rotação correto. A matriz final constrói-se como indicado no pseudo-código, sendo a última linha o padrão de transformações homogéneas, e necessita que o vetor *align* seja multiplicado por -1 ou corre-se o risco de inverter a orientação dos triângulos que constituem o objeto que se pretende alinhar.

Nota: Aquando da leitura do vetor *align*, é necessário multiplicar a componente x por -1 para obtermos um resultado correto

Além das translações, também é possível obter rotações baseadas em tempo. O vetor de rotação não é dependente de t, mas o ângulo é e pode ser obtido realizando as mesmas operações de módulo e divisão de t pelo tempo de rotação. O vetor de rotação é normalizado de forma a não afetar eventuais escalas aplicadas sobre o mesmo objeto.

3.3) VBOs

Para implementar os VBOs, decidimos definir os diferentes vértices como *structs*, permitindo maior legibilidade e flexibilidade. Assim, a struct definida possui o seguinte formato:

```
struct AxisVertex {
  vec4 coords;       // posicao, XYZW
  vec4 color;        // cor, RGBA
};
```

Estes vértices são usados para desenhar linhas como os eixos, mas também, em situações de debug, são usados para desenhar as normais dos vértices e das derivadas das curvas de *Catmull-Rom*. Guardam uma posição, bem como a cor do vértice.

A estrutura dos vértices dos triângulos do *viewport* é:

```
struct ViewportVertex {
  glm::vec4 coords;            // posição
  glm::vec2 tex_coord;         // coords textura, UV
};
```

Estes vértices são apenas usados para desenhar triângulos no *viewport* e obter efeitos como bloom (a ser explicado na próxima fase). Assim, apenas necessitam de ter coordenadas de posição e textura.

Os vértices dos objetos da cena possuem o formato:

```
struct Vertex {
  glm::vec4 coords;
  glm::vec3 normal;         // coordenadas do vetor normal
  glm::vec2 tex_coord;
  GLint object_id;          // índice do objeto
```

Como constituem os objetos da cena, têm coordenadas de posição e textura, vetor normal e um índice para o ID de objeto. A partir deste ID, no *shader*, a sua matriz de transformação pode ser lida num array (mencionado na fase anterior). O uso de um ID por objeto permitirá também suportar informações sobre o material a utilizar, implementado na próxima fase.

Com as estruturas definidas anteriormente, os VBOs podem ser criados muito facilmente, sendo o seu layout especificado a partir destes vértices. Por exemplo, para especificar a posição no espaço (coords) da struct Vertex temos:

```
glVertexAttribPointer(
  <numero do layout>, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
  (const void *)offsetof(Vertex, coords)
)
```

Para especificar o ID de um objeto:

```
glVertexAttribIPointer(
  <numero do layout>, 1, GL_INT, sizeof(Vertex),
  (const void *)offsetof(Vertex, object_id)
)
```

Como se pode observar, o uso de sizeof(Vertex) bem como de offsetof(Vertex, object_id) simplificam bastante o uso destas funções.

Não se faz uma gestão minuciosa da memória deste VBO, sendo todos os dados reenviados em cada frame.

4) Sistema Solar

No seguimento do modelo estático do Sistema Solar desenvolvido na fase anterior, foram adicionadas animações a todos os planetas, anéis e luas, e introduziu-se um cometa. O novo sistema

solar encontra-se no ficheiro "custom_test_files_phase3/solar_system.xml". Para demonstrar a movimentação ao longo do tempo, os planetas apresentam rotações em torno do vetor (0,1,0), para simular o movimento de translação em torno do Sol, e as luas e anéis descrevem diferentes rotações em torno dos seus respetivos planetas. Para demonstrar as curvas de Catmull-Rom, definimos uma curva que interseta parcialmente o cinturão de Asteróides. O objeto que segue esta curva é um cometa, alinhado com o eixo x e é construído com Bézier Patches, sendo dividido em duas partes: a cabeça e a cauda (definidos na pasta "custom_test_files_phase_3" com nomes "comet.patch" e comet_tail.patch"). Para definir as patches da cabeça, baseamo-nos na foto abaixo, retirada do site:

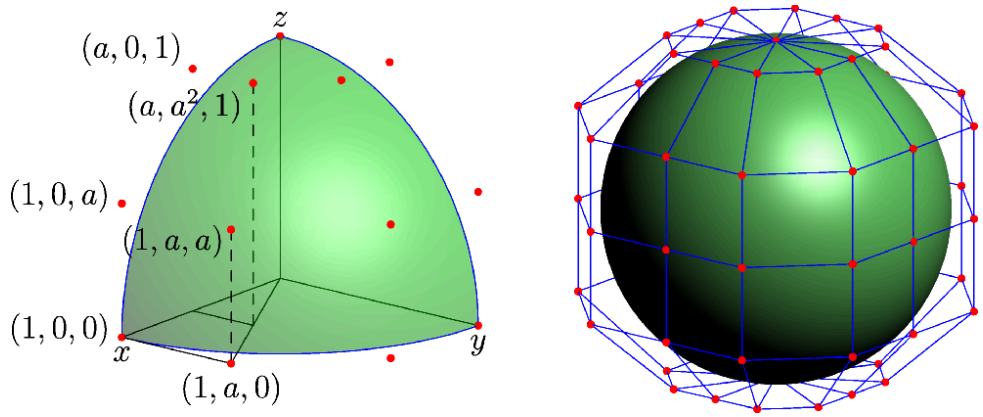

Figura 2: Octeto de esfera definido por pontos de controlo em Bézier Patch

Cada patch permite definir um octeto da esfera (que irá constituir a cabeça), assumindo um valor arbitrário de a = 0.5. Adaptando as coordenadas para os diferentes octantes, é possível obter a esfera completa. Por sua vez, a cauda é formada também com base nesta estrutura. Pegando nos 4 octetos da esfera com o valor de x $<$ 0 (cria uma semi-esfera) e escalando os pontos, é possível definir o rasto do cometa: multiplicamos as coordenadas x por 3, de y por 1.6 e de z por 1.5, obtendo uma forma semelhante a um cone, que se traduz na cauda do cometa.

5) Alterações da estrutura de threads

Como mencionado na primeira fase, usamos duas threads no programa: uma para físicas/animações e outra para renderização em si. Esta última tinha como principal objetivo ser determinística, executando a uma *framerate* especifica. Para tal, utilizava usleep() ou equivalente, com o objetivo de esperar um certo número de microssegundos caso acabasse a sua tarefa demasiado rápido. Infelizmente, no caso dos sistemas Windows e Mac, o sistema operativo não permite uma resolução temporal tão minuciosa ou simplesmente não aceita devolver o controlo da thread suficientemente rápido, pelo que a espera prolongava-se 10x ou mais o esperado. Assim, restaram algumas alternativas: usar sleep e sofrer um atraso significativo, recorrer a espera ativa e desperdiçar recursos, abandonar o conceito de threads independentes uma da outra e processar sequencialmente, ou permanecerem duas threads, mas com a thread de física permanentemente em sincronia com a thread de renderização, perdendo a independência que desejavamos. Encontramos algumas possíveis soluções para melhorar a espera ativa, utilizando instruções assembly de pausa do CPU, mas não conseguimos obter um impacto satisfatório.

No fim, decidimos manter esta estrutura, por simplicidade:

- Linux: mantém espera passiva com usleep() e tem determinismo.
- Windows e Mac: não utilizam duas threads, nem têm determinismo, sendo tudo processado na thread de renderização.

Ficou ainda implementada a espera ativa não determinística, contudo não é usada atualmente. Como não existe nenhum sistema de físicas, qualquer impacto será meramente visual, pelo que todas as soluções apresentam cenas semelhantes.

Achamos também interessante implementar uma funcionalidade que modifique a velocidade das animações (mantendo a velocidade da câmera). No entanto, surgiram vários problemas derivados do sistema básico de sincronização das threads: por vezes renderizar pode ser mais rápido do que calcular as animações, outras vezes ocorre o oposto, dando resultados inesperados. Decidimos que, se o cálculo de animações for muito mais rápido que a renderização, esse resultado será simplesmente ignorado. Outra alternativa seria 'saltar' os cálculos que nunca fossem ser mostrados, nem sequer os realizando, contudo, se eventualmente fosse implementado um sistema de físicas, poderiam surgir problemas. Não consideramos que estaria no âmbito deste trabalho um sistema complexo para sincronizar as threads e ter em conta todos os edge cases, tendo adaptado as abordagens que referimos anteriormente.

6) Conclusão

Nesta fase consolidamos conceitos relativos a curvas de Bézier, Bézier patches, e curvas de Catmull-Rom. Com recurso a estas últimas, definimos animações de objetos, aplicando transformações ao longo de intervalos de tempo definidos. Introduzimos VBOs para otimizar o desenho de objetos e, para ilustrar as várias funcionalidades, construímos um modelo dinâmico do Sistema Solar.

Implementamos também algumas funcionalidades de debug adicionais: modificar a velocidade das animações (da cena), velocidade da câmera, toggle para mostrar curvas e as suas normais. Separamos os FPS de renderização dos FPS de cálculo de animações e do *timestep* que as animações estão a considerar nos seus cálculos.

Aprendemos sobre diversos problemas que surgem ao separar as animações da renderização e como deveríamos ter estruturado toda a Engine de forma diferente para suportar as funcionalidades que implementámos, entendendo as vantagens e limitações de diversas abordagens. Enfrentámos também desafios quando decidimos implementar alinhamento das curvas de *Catmull-Rom* através de um vetor aleatório. Como trabalho futuro, o grupo planeia a implementação da fase 4 do projeto com a utilização de shaders.