Abril 5, 2024

Relatório CG - Fase 2 - Grupo 24

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

Sumário

1) Introdução 3
2) Transformações Geométricas 3
2.1) XML Parser 3
2.2) Hierarquia de objetos 4
2.3) Consts 5
2.4) Renderer 5
2.5) Modelo estático do sistema solar 6
2.6) Testes e debug 7
3) Conclusão 7

1) Introdução

Neste documento é caraterizado o processo de desenvolvimento da fase 2 do projeto da Unidade Curricular de Computação Gráfica. São descritas as transformações geométricas implementadas, a hierarquia de classes criada e as modificações realizadas entre a fase atual e a anterior.

2) Transformações Geométricas

A implementação das transformações conduziu a modificação nalguns componentes do programa Engine, nomeadamente:

- XML Parser, responsável por processar a informação contida nos ficheiros XML
- Consts, o qual fornece as funções que constroem as matrizes de transformação
- Renderer, responsável pela renderização das cenas

2.1) XML Parser

A interpretação das diversas tags XML ocorre neste componente pelo que, para suportar as novas tags relativas às diversas transformações (rotações, translações e escalas), foi necessário acrescentar o reconhecimento das tags `<transform>`, `<rotate>`, `<translate>` e `<scale>`.

Simultaneamente, para otimizar o programa, e com a hierarquia de classes implementada (e descrita mais à frente) em mente, o grupo decidiu realizar a pré-computação da matriz de transformação final multiplicando a matriz de transformação atual pelas matrizes já lidas. O objetivo desta otimização é o de, durante a renderização da cena, apenas realizar uma transformação por objeto e, simultaneamente, economizamar memória.

Atualmente, o XML Parser segue o seguinte formato:

```
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
      transformation *= 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
```

2.2) Hierarquia de objetos

A hierarquia dos objetos é das componentes prinicipais desenvolvidas nesta fase. Além de realizar a leitura e obtenção das diversas primitivas contidas no ficheiro XML, determinará a forma como os objetos resultantes do parsing serão enviados para o Renderer. Para além disso, possui fortes implicações na futura implementação das curvas de *Catmull-Rom*, as quais serão utilizadas para animar os vários objetos.

Analisando os ficheiros XML fornecidos pela equipa docente, os quais possuem uma estrutura em árvore, o grupo decidiu criar uma árvore n-ária em que cada nodo possui uma matriz de transformação (a qual pode ser interpretada como a multiplicação das transformações do nodo pai e das transformações do nodo atual), que é aplicada aos pontos das primitivas da respetiva tag `<group>`. Esta estrutura permite manter em cada nodo apenas a matriz de transformação correspondente a cada tag `<group>`, porém o grupo decidiu que em vez de multiplicar as matrizes ao percorrer a árvore, a multiplicação aconteceria durante o parsing. O resultado é idêntico, porém, para efeitos de debug, é mais conveniente saber qual a transformação aplicada a cada ponto sem precisar de saber quais as transformações resultantes do nodo pai.

Com a estrutura desenvolvida, o Renderer apenas precisa de invocar duas funções: a primeira devolve um array em que cada elemento é o conjunto dos objetos presentes numa tag `<models>` (cada objeto provém de uma tag `<model>`); a segunda devolve um array em que cada índice possui a transformação final aplicada aos objetos no mesmo índice do primeiro array. No futuro, este segundo array conterá ainda as informações relativas às cores e texturas dos diversos objetos, porém são necessárias algumas alterações mínimas nesta hierarquia que serão aplicadas na fase 4 - em vez de cada nodo possuir um vetor com os pontos dos objetos da tag `<models>`, teremos um vetor de pares (material, pontos do objeto), material este que indica a textura e as cores a aplicar nos pontos.

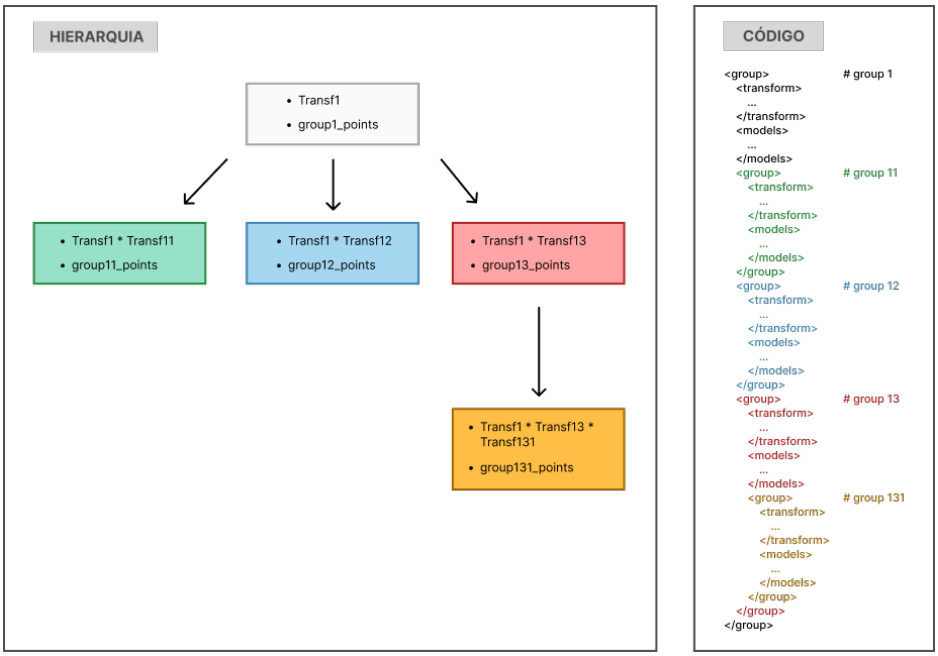

Figura 1: Árvore que representa a hierarquia de objetos

2.3) Consts

Para suportar as diferentes transformações, desenvolvemos nesta componente um conjunto de funções para obter matrizes de escala, rotação e translação. Destaca-se a função que devolve a matriz de rotação em torno de um vetor arbitrário que, em vez de decompor as rotações em torno de cada eixo, como lecionado nas aulas, explora outro método semelhante que utiliza a fórmula de rotação de Rodrigues, pelo facto de ser concisa e computacionalmente eficiente. Baseamo-nos no site http://www.songho.ca/opengl/gl_rotate.html, no qual consta uma demonstração extensa da fórmula.

2.4) Renderer

Decidimos que a estratégia mais prática para desenhar os objetos na posição correta seria aplicar as transformacões em shaders, preservando sempre a posição original dos vértices em memória ao invés de, por exemplo, aplicar as transformações aos vértices de imediato (no CPU) e depois desenhar os mesmos, possivelmente perdendo as posições originais.

Assim, com base nos arrays fornecidos pela hierarquia de objetos, associamos a cada vértice do primeiro array um ID de objeto com o qual pode ler a sua transformação a partir do segundo array. Como o macOS só tem suporte até openGL 4.1, o que impossibilita o uso de SSBOs, e não nos querendo limitar a usar *uniforms*, decidimos que os dados seriam colocados num texture buffer.

Como o tipo `mat4` se alinha perfeitamente com `vec4`, basta apenas aceder, no shader, a vetores consecutivos para construir a matriz de transformação, conforme descrito:

```
#define numero_vec4_em_transf 4
mat4 transf;
transf[0] = texelFetch(u_TextureBuffer, 0 + (ObjectID * numero_vec4_em_transf));
transf[1] = texelFetch(u_TextureBuffer, 1 + (ObjectID * numero_vec4_em_transf));
transf[2] = texelFetch(u_TextureBuffer, 2 + (ObjectID * numero_vec4_em_transf));
transf[3] = texelFetch(u_TextureBuffer, 3 + (ObjectID * numero_vec4_em_transf));
```

E depois aplicá-la como se fosse o model da MVP:

```
gl_Position = u_Projection * u_View * transf * PosicaoOriginal;
```

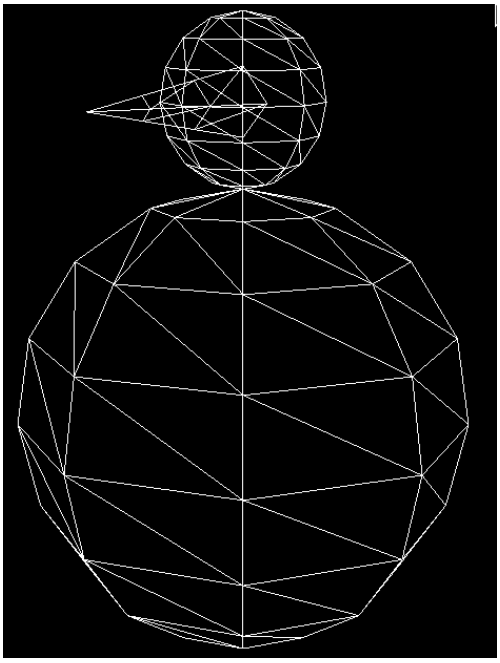

Figura 2: Exemplo de uma cena

2.5) Modelo estático do sistema solar

Para demonstrar a aplicação das transformações implementadas foi desenvolvida uma cena de um modelo do sistema solar num ficheiro xml que consta no caminho custom_test_files_phase_2/solar_system.xml. Não houve atenção para manter proporções de distâncias e tamanhos entre objetos realistas, porque dificultava a observação dos objetos da cena. Foram utilizadas esferas para representar os planetas e as suas respetivas luas e a primitiva do torus criada na fase 1 para representar os anéis de saturno e neptuno e o cinturão de asteróides.

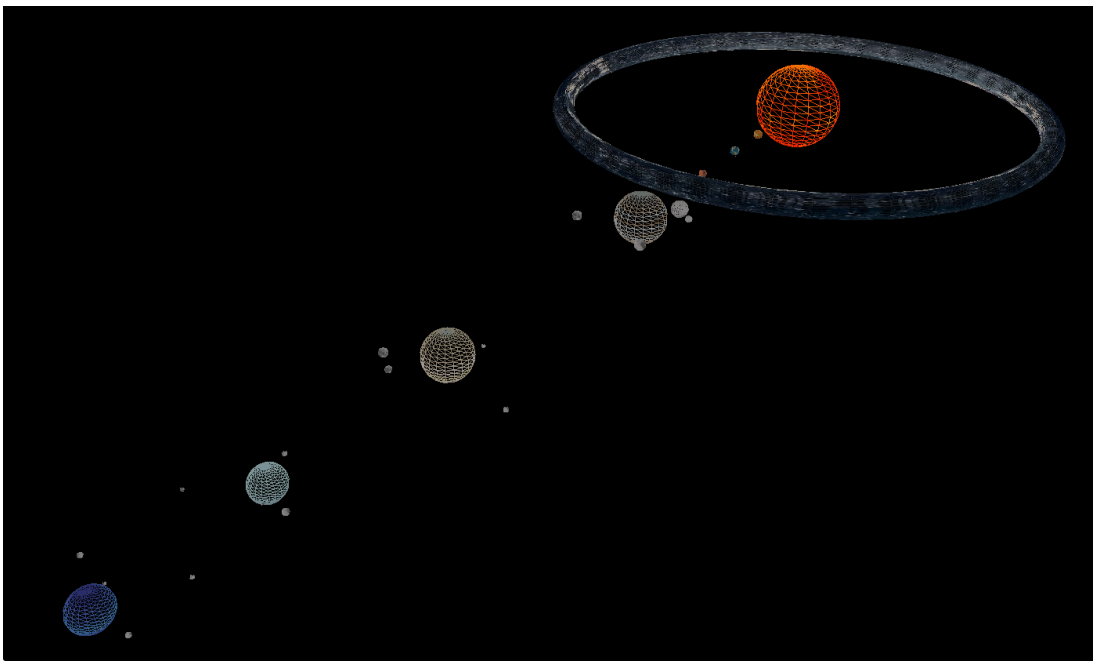

Figura 3: Cena do sistema solar

2.6) Testes e debug

Com base nos ficheiros XML de teste fornecidos pelos docentes, verificámos que todas as transformações de primitivas se encontram corretamente renderizadas.

Como preparação para as fases seguintes e para teste das funcionalidades atuais, desenvolvemos uma ferramenta de debug usando o ImGui. Disponibilizamos assim um menu onde podemos:

- ver informações como FPS, posição e orientação da câmera;
- alterar a posição da câmara e a sua velocidade;
- visualizar os eixos através de um toggle;
- limitar os fps atavés de um slider e uma checkbox.

3) Conclusão

Nesta fase consolidamos conceitos relativos à aplicação de transformações (translação, rotação, escala), à importância da ordem das transformações e estruturamos a hierarquia dos objetos das cenas. O grupo planeia, durante a implementação das fases seguintes do projeto, definir algumas cenas para ilustrar as funcionalidades do Engine.