Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

2ª Fase - Transformações

Diogo Matos
A100741

Júlio Pinto
A100742

Mário Rodrigues
A100109

Miguel Gramoso
A100835

April 05, 2024

CG

Índice

1. Introdução **1**

2. Engine **2**
2.1. Leitura do XML 2
2.2. Transformações 2
2.2.1. Translação 3
2.2.2. Rotação 4
2.2.3. Escala 4
2.3. Desenhar Grupos 6
2.4. Sistema Solar 7

3. Testes **8**

4. Extras **9**
4.1. Save Latest 9
4.2. Scripts 10

5. Conclusões e Trabalho Futuro **11**

1. Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular Computação Gráfica (CG) no ano letivo 2023/2024.

A segunda fase do projeto consistiu em adicionar novas funcionalidades ao **engine**, destacando-se a interpretação de *scenes* que incluam transformações geométricas definidas hierarquicamente. Para esta fase, foi também necessário desenvolver uma *scene* que represente o **Sistema Solar**.

2. Engine

2.1. Leitura do XML

De maneira ao nosso *engine* conseguir renderizar transformações, foi necessário fazer algumas reestruturações ao que tínhamos previamente. Para tal, tivemos que alterar a nossa classe `Configuration`. Agora, em vez de guardar vetores de pontos (*i.e.* modelos), guarda **grupos**.

Estes grupos são definidos pela classe **`Group`** que guarda as diversas *paths* para os ficheiros `.3d`, os subgrupos (`subgroups`), todos os pontos para o desenho do grupo (`points`) e ainda uma matriz de transformações (`arr`).

```
<group>
    <transform>
      ...
    </transform>
    <models>
      ...
    </models>
    <group>
              ...
    </group>
<group>
```

Listing 1: Estrutura de um grupo no ficheiro XML.

```
class Group {
 public:
  std::vector<std::string> models;
  std::vector<Group> subgroups;
  std::vector<Point> points;
  std::array<std::array<double, 4>, 4> arr = {
      {
      {1, 0, 0, 0},
      {0, 1, 0, 0},
      {0, 0, 1, 0},
      {0, 0, 0, 1}
    }};
```

Listing 2: Estrutura da Classe Group.

Esta classe `Group` irá permitir que as transformações sejam efetuadas de forma hierárquica e sem duplicação de informação, no que toca às transformações em si. De modo a também conseguirmos representar um grupo sem transformações, o grupo inclui uma matriz identidade, desta forma, caso um dos seus subgrupos contenha transformações, não haverá qualquer problema na multiplicação de matrizes.

2.2. Transformações

Considerando que a ordem pela qual se aplica as transformações nos modelos será relevante, como mencionado previamente, cada grupo tem uma matriz de transformações. Esta matriz representa todas as transformações que serão aplicadas naquele grupo e nos subgrupos do mesmo. Desta forma, um subgrupo não saberá quais foram as transformações previamente aplicadas ao seu grupo "pai".

Para conseguirmos obter estas transformações, multiplicamos a matriz associada a cada uma pela matriz prévia.

Por exemplo, dado o seguinte ficheiro `.xml`:

```
<group>
      <transform>
        <translate x="0" y="1" z="0" />
      </transform>
      <group>rre422e1qer
        <transform>
          <rotate angle="90" x="1" y="0" z="0" />
          <scale x="0.1" y="0.3" z="0.1" />
        </transform>
        <models>
          <model file="cone_1_2_4_3.3d" /> <!-- generator cone 1 2 4 3 -->
        </models>
      </group>
</group>
```

As transformações necessárias para o cone implicariam a seguinte ordem de multiplicação de matrizes:

$$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} * \begin{pmatrix}1&0&0&0\\0&1&0&1\\0&0&1&0\\0&0&0&1\end{pmatrix} = \begin{pmatrix}1&0&0&0\\0&1&0&1\\0&0&1&0\\0&0&0&1\end{pmatrix}$$

matriz base translação $(0,1,0)$ resultado

$$\begin{pmatrix}1&0&0&0\\0&1&0&1\\0&0&1&0\\0&0&0&1\end{pmatrix} * \begin{pmatrix}1&0&0&0\\0&0&-1&0\\0&1&0&0\\0&0&0&1\end{pmatrix} = \begin{pmatrix}1&0&0&0\\0&0&-1&1\\0&1&0&0\\0&0&0&1\end{pmatrix}$$

matriz base rotação 90º $(1,0,0)$ resultado

$$\begin{pmatrix}1&0&0&0\\0&0&-1&1\\0&1&0&0\\0&0&0&1\end{pmatrix} * \begin{pmatrix}0.1&0&0&0\\0&0.3&0&0\\0&0&0.1&0\\0&0&0&1\end{pmatrix} = \begin{pmatrix}0.1&0&0&0\\0&0&-0.1&1\\0&0.3&0&0\\0&0&0&1\end{pmatrix}$$

matriz base escala $(0.1,0.3,0.1)$ resultado

De seguida, exploramos cada uma destas transformações.

2.2.1. Translação

Esta transformação permite alterar a posição dos modelos.

Ao aplicar uma translação segundo um vetor a um grupo, estamos a multiplicar a sua matriz pela seguinte:

$$\begin{pmatrix}1&0&0&x\\0&1&0&y\\0&0&1&z\\0&0&0&1\end{pmatrix}$$

Figura 1: Matriz de translação associada a um vetor (x,y,z)

Após se aplicar a translação à matriz base, verificamos que os objetos desenhados movimentam-se no espaço segundo o vetor defenido. No exemplo seguinte é realizada uma translação na figura triangular.

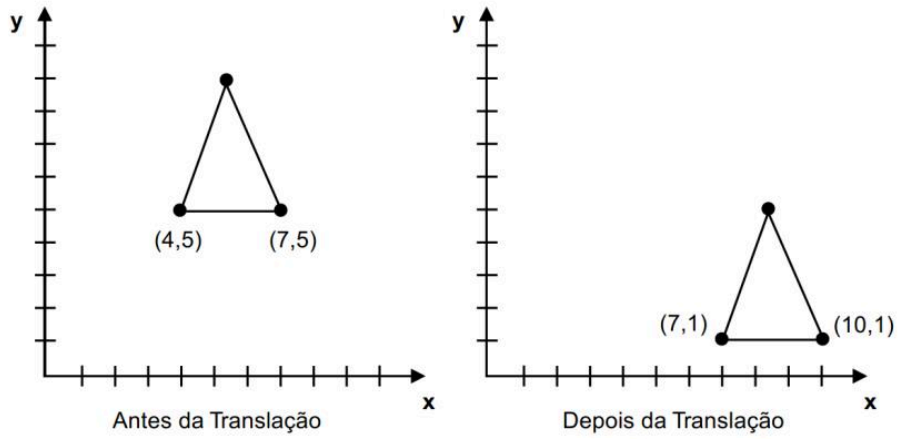

Figura 2: Aplicação de uma translação segundo o vetor $(3, -4, 0)$

2.2.2. Rotação

Aplicando uma rotação segundo um eixo arbitrário e um ângulo a um grupo, estamos a multiplicar a sua matriz pela seguinte:

$$\begin{pmatrix} x^2 + (1 - x^2) * \cos(\theta) & x * y * (1 - \cos(\theta)) - z * \sin(\theta) & x * z * (1 - \cos(\theta)) + y * \sin(\theta) & 0 \\ y * x * (1 - \cos(\theta)) + z * \sin(\theta) & y^2 + (1 - y^2) * \cos(\theta) & y * z * (1 - \cos(\theta)) - x * \sin(\theta) & 0 \\ x * z * (1 - \cos(\theta)) - y * \sin(\theta) & y * z * (1 - \cos(\theta)) + x * \sin(\theta) & z^2 + (1 - z^2) * \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 3: matriz de rotação associada a um vetor (x, y, z) e a um ângulo θ

Após se aplicar a rotação à matriz base, verificamos que os objetos desenhados são rodados no espaço segundo o vetor e ângulo defenidos. No exemplo que se segue foi realizada uma rotação na figura triangular.

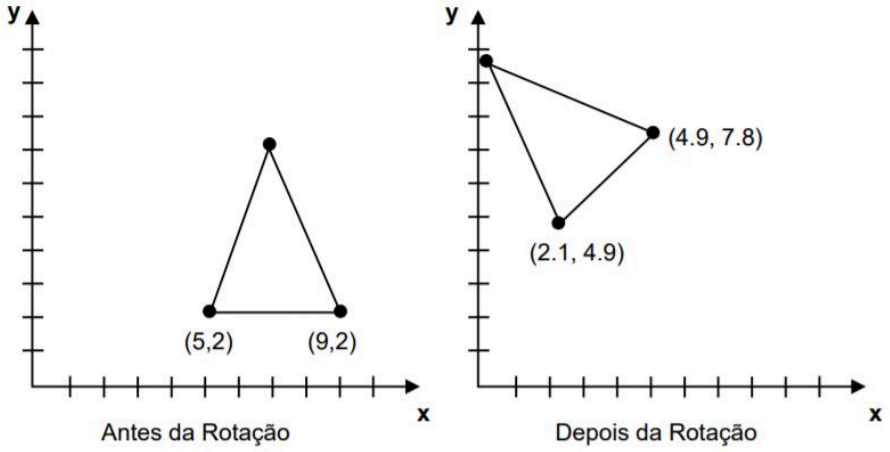

FIGURA 2.6. *A multiplicação das coordenadas por uma matriz de rotação pode resultar em uma translação.*

Figura 4: Aplicação de uma rotação segundo o vetor X e ângulo Y

2.2.3. Escala

Ao aplicar uma escala segundo um vetor a um grupo, estamos a multiplicar a sua matriz pela seguinte:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 5: Matriz de escala associada a um vetor (x, y, z)

Após se aplicar a escala à matriz base, verificamos que os objetos desenhados são escalados no espaço segundo o vetor defenido. No exemplo que se segue foi realizada uma escala na figura triangular.

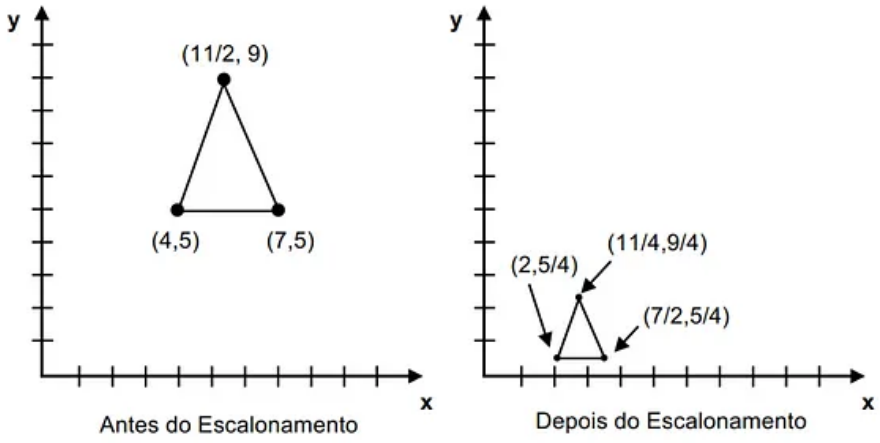

Figura 6: Aplicação de uma escala segundo o vetor $(0.5, 0.25, 0)$

2.3. Desenhar Grupos

De modo a aplicar as transformações aos grupos de menor hierarquia, optamos por usar uma função recursiva que desenha os pontos de cada grupo, `drawGroups()`. Esta função utiliza a função `glMultMatrix()` para calcular o produto da matriz atual de desenho pela matriz fornecida como argumento. Isto permite que no momento do desenho dos pontos de cada grupo, a matriz de desenho seja o produto de todas as transformações que estejam acima da sua hierarquia.

```
void drawGroups(const Group &group) {
  std::vector<Point> points = group.points;

  glPushMatrix();
  GLfloat matrix[16] = {
      group.arr[0][0], group.arr[1][0], group.arr[2][0], group.arr[3][0],
      group.arr[0][1], group.arr[1][1], group.arr[2][1], group.arr[3][1],
      group.arr[0][2], group.arr[1][2], group.arr[2][2], group.arr[3][2],
      group.arr[0][3], group.arr[1][3], group.arr[2][3], group.arr[3][3]};

  glMultMatrixf(matrix);

  glBegin(GL_TRIANGLES);
  glColor3f(1.0f, 1.0f, 1.0f);
  for (size_t i = 0; i < points.size(); i += 3) {
    // Draw each triangle
    glVertex3f(points[i].x, points[i].y, points[i].z);
    glVertex3f(points[i + 1].x, points[i + 1].y, points[i + 1].z);
    glVertex3f(points[i + 2].x, points[i + 2].y, points[i + 2].z);
  }
  glEnd();

  for (size_t i = 0; i < group.subgroups.size(); i++) {
    drawGroups(group.subgroups[i]);
  }

  glPopMatrix();
}
```

Listing 3: Função para o desenho dos grupos (`Group`)

2.4. Sistema Solar

Como teste para esta fase, adicionamos uma *scene* referente ao sistema solar, escalado de maneira a ser possível observar todos os elementos renderizados. Optamos por inserir o máximo de objetos de maneira a obter o máximo realismo possível, nomeadamente a inclusão de todos os satélites naturais, de múltiplos asteróides e do anel de Saturno.

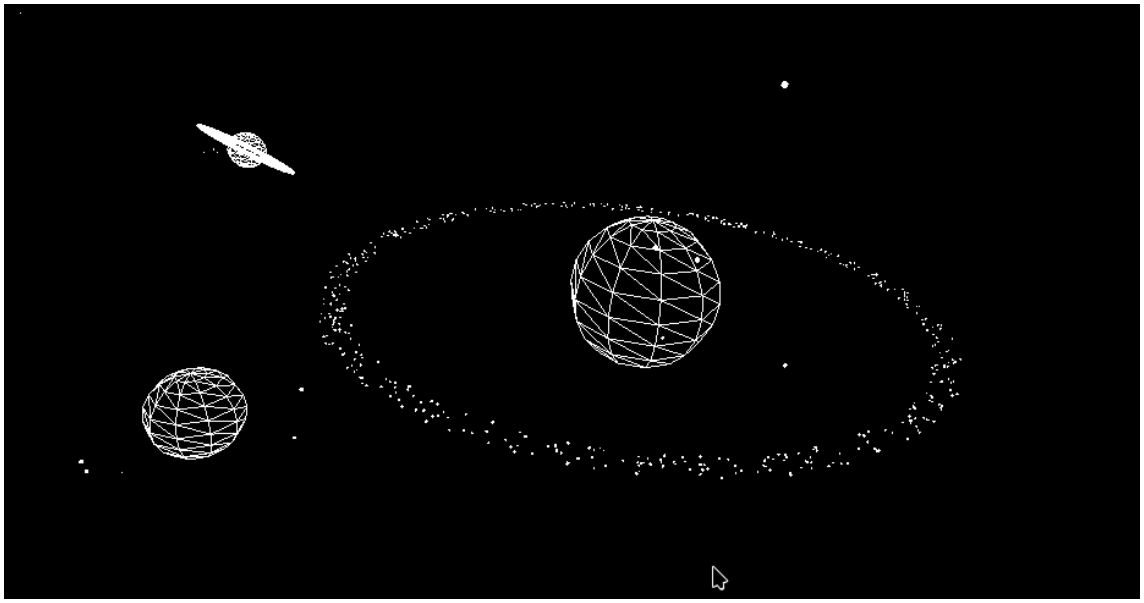

Figura 7: Renderização da *scene* do Sistema Solar

3. Testes

De modo a testar as diversas *features* implementadas nesta fase, foram realizados um conjunto de testes que se apresentam de seguida.

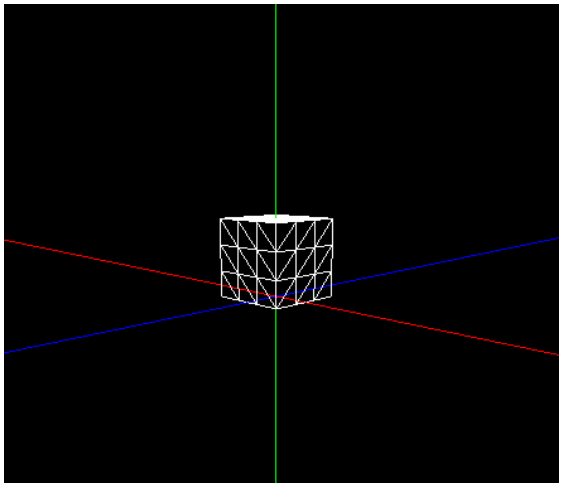

Figura 8: Teste nº1 - Cubo

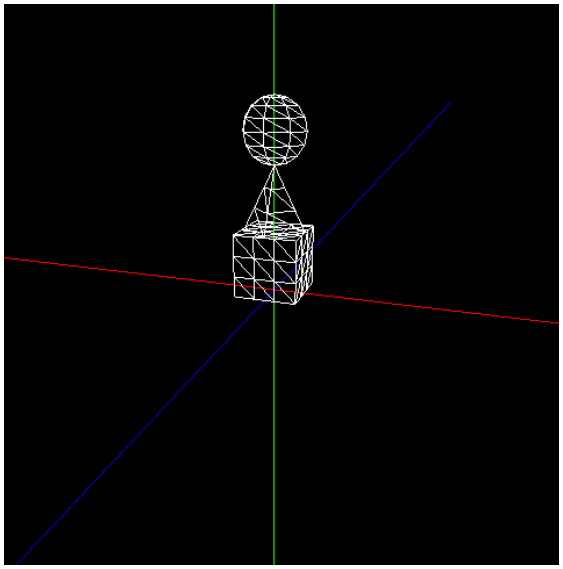

Figura 9: Teste nº2 - Esfera, Cone e Cubo

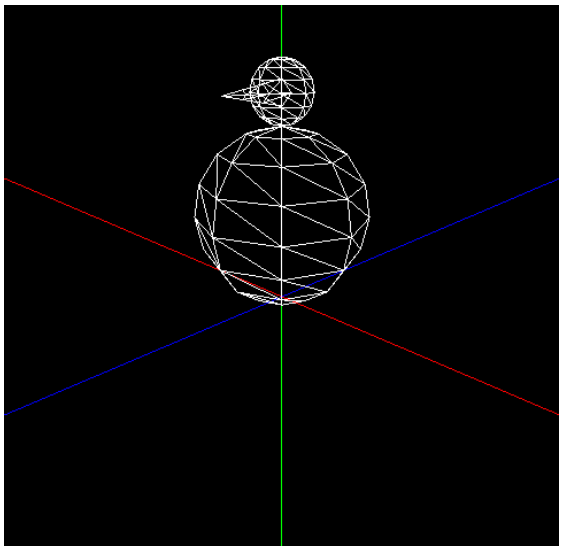

Figura 10: Teste nº3 - Boneco de neve

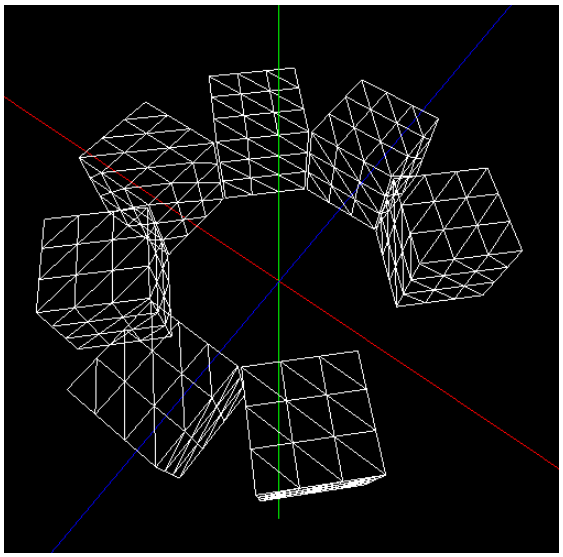

Figura 11: Teste nº4 - Cubos circundando eixo

4. Extras

Fomos motivados a ir mais longe e implementar algumas funcionalidades extra que exploramos de seguida.

4.1. Save Latest

Uma das ideias que tivemos foi guardar o estado atual da cena. Para tal, desenvolvemos algumas funções. Neste momento, realizamos os movimentos da câmara a através do uso da função `glRotatef()` e `glScalef()`. Através dos ângulos que lhe passamos e do fator de escala, tendo a posição de origem da câmara, conseguimos calcular a posição atual da câmara.

Para além disso, utilizamos a função `glutget()` com as váriaveis `GLUT_WINDOW_WIDTH` e `GLUT_WINDOW_HEIGHT` para obter as dimensões atuais da janela.

Tendo a posição atual da câmara e o tamanho atual da janela, utilizamos a biblioteca `rapidxml` para alterar o ficheiro `.xml` de origem de modo a conter essas informações.

```
// Accessing window node
rapidxml::xml_node<>* windowNode = root->first_node("window");

if (windowNode) {
  rapidxml::xml_attribute<>* widthAttr = windowNode->first_attribute("width");
  if (widthAttr)
    widthAttr->value(doc.allocate_string(std::to_string(w.width).data()));

  rapidxml::xml_attribute<>* heightAttr =
      windowNode->first_attribute("height");
  if (heightAttr)
    heightAttr->value(doc.allocate_string(std::to_string(w.height).data()));
}

// Accessing camera node
rapidxml::xml_node<>* cameraNode = root->first_node("camera");

if (cameraNode) {
  rapidxml::xml_node<>* positionNode = cameraNode->first_node("position");
  if (positionNode) {
    rapidxml::xml_attribute<>* xAttr = positionNode->first_attribute("x");
    if (xAttr)
      xAttr->value(doc.allocate_string(std::to_string(point.x).data()));

    rapidxml::xml_attribute<>* yAttr = positionNode->first_attribute("y");
    if (yAttr)
      yAttr->value(doc.allocate_string(std::to_string(point.y).data()));

    rapidxml::xml_attribute<>* zAttr = positionNode->first_attribute("z");
    if (zAttr)
      zAttr->value(doc.allocate_string(std::to_string(point.z).data()));
  }
}

// Similar updates for 'lookAt' and 'up' nodes in the future
```

O código acima demonstra como foi alterado o `xml_doc` inicial de modo a conter as configurações que lhe passamos (as configurações atuais).

4.2. Scripts

De maneira a não termos problemas com ficheiros de *cache* do `CMake`, criamos um pequeno *script* em *shell* (`clean.sh`) que elimina todos os ficheiros gerados pelo `CMake`.

5. Conclusões e Trabalho Futuro

Consideramos que esta fase foi realizada de acordo com os requisitos pretendidos e que o programa já se encontra com múltiplas funcionalidades de apoio ao projeto. Pretendemos adicionar mais funcionalidades extra, em especial a implementação de *frustsum culling*.

Relativamente à próxima fase, já começamos a modular o código para a implementação de *buffers* armazenados na placa gráfica (*VBOs*), de maneira a tornar as renderizações mais rápidas e eficientes.