Fase 2
Transformações Geométricas e *Grouping*

Grupo 42
Ângelo Sousa[A80813], Diogo Ribeiro[A84442], Rui Mendes[A83712], and Rui Reis[A84930]

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mail: {a80813,a84442,a83712,a84930}@alunos.uminho.pt

Resumo Extensão do projecto prático, no intuito de contemplar operações de agrupamento de elementos, aplicando transformações geométricas hierarquicamente por grupos.

1 Introdução

Para a segunda fase do trabalho prático de Computação Gráfica foram trabalhados essencialmente o novo formato dos ficheiros XML bem como um novo módulo para o movimento da câmara.

Relativamente ao novo formato dos ficheiros XML, as principais mudanças centram-se na nova hierarquia que nos permite agrupar diversas transformações geométricas.

Para o movimento da câmara implementámos as funcionalidades que nos permitem movimentar as câmaras em qualquer um dos sentidos bem como, pôr a câmara a apontar para qualquer direção. Também adicionamos as funcionalidades de aumentar e diminuir as velocidades das acções da câmara, e, reiniciar todos os parâmetros da câmara. Para o movimento da câmara ser mais natural também adicionámos a capacidade de conseguir pressionar várias teclas simultaneamente.

2 Adaptação à operação de *Grouping*

Para conseguirmos trabalhar com grupos e o seu simbolismo mecânico, criamos a classe `Group`, classe esta que visa englobar todo o comportamento expectável de um grupo.

Num grupo, podemos identificar duas necessidades primárias que estes devem encapsular, primeiramente é necessário que cada grupo tenha conhecimento das transformações geométricas que tem de efetuar sobre o sistema, bem como de todos os objectos que devem ser imediatamente desenhados dentro deste grupo. Então o que fizemos foi dotar o grupo de duas filas de espera, uma que armazena `TransformEvent`, que dizem respeito às transformações geométricas indicadas, e outra que armazena `DrawEvent` que dizem respeito aos objectos que devem ser desenhados dentro do grupo.

Para além disso, e por questões explicitadas mais à frente, é necessário que cada grupo saiba quantos sub-grupos possui. Sendo também necessário que cada grupo forneça também uma função capaz de comunicar com o exterior a informação sobre o objecto a desenhar, função esta à qual chamamos `publish` e que retorna também o número de sub-grupos do respectivo grupo.

Tendo tudo isto em conta é possível chegar ao seguinte API de grupos, que visa garantir que um grupo é capaz de responder a todos os requesitos.

```
1  class Group {
2  private:
3      vector<TransformEvent> transformations;
4      vector<DrawEvent> drawings;
5      int n_subgroups;
6      void popDraw(int idx, GLuint * buffers,
7                   GLuint * indexes);
8  public:
9      Group();
10     void pushTransformation(TransformEvent te);
11     void pushDraw(DrawEvent de);
12     int publish(GLuint * buffers, GLuint * indexes);
13     int addSubgroup();
14 };
```

2.1 Arquitetura do *Engine*

Com a definição de grupos já em mente, é nós possível abstrair e visualizar de que forma pretendemos que o nosso motor gráfica tenha em conta todos os possíveis grupos.

A nossa visão foi que um motor gráfico, é composto unicamente por 2 estruturas essenciais:

- Um motor para controlo de movimentos da câmara, analisado em maior profundidade na secção seguinte.
- Um *array* contendo todos os grupos, e respectivos subgrupos, que fazem parte do motor gráfico.

Desta forma, podemos pensar em grupos apenas como sendo elementos de uma cadeia. Com esta visão, podemos facilmente verificar que se um dado grupo possuir N sub-grupos, então esses N sub-grupos e todos os seus respectivos sub-grupos estarão em posições seguidas ao grupo inicial. Formando assim uma ordenação topológica de grupos, que, juntamente com o parâmetro `n_subgroups` da classe `Group`, permite especificar todos os grupos e respectivos sub-grupos.

Assumindo que já temos temos o nosso *array* preenchido, então podemos desenhar todos os grupos utilizando o seguinte algoritmo:

Algorithm 1: runGroups

Result: N, nº de grupos processados.
Require: $idx <$ `groups.size()`;
$N \leftarrow 1$;
`G` $\leftarrow$ `groups[`idx`]`;
`glPushMatrix()` ;
`sub_groups` $\leftarrow$ `G.publish(`$\cdots$`)`;
$i \leftarrow 0$;
while $i <$ *`sub_groups`* **do**
  $N \leftarrow N+$ `runGroups(`*`idx`* $+N$`)`
  $i \leftarrow i + 1$;
end

Que permite definir a função `runGroups`, responsável por recursivamente processar os grupos e todos os seus sub-grupos.

2.2 Adaptação XML

Nesta fase foi necessário alterar algumas coisas de modo a que o nosso *parser*, construído anteriormente, fosse capaz de lidar com o novo formato dos ficheiros XML.

Optamos por continuar a usar o tinyXML como ferramenta principal para lidar com este tipo de ficheiros. Para de facto construir o modelo foram utilizador várias funções do motor gráfico, que eram invocadas de acordo com a informação presente no ficheiro XML.

Para poder separar o programa de modo a ser mais legível e mais organizado foi criado um ficheiro `Txml.h` que contêm todas as funções necessárias para a transformação da informação em XML para a linguagem do motor gráfico. Essas funções são `loadGroup` (explicada mais à frente), `loadFile` (indica ao objecto `Txml` qual a directoria onde se encontra o ficheiro XML pretendido) e `loadEngine` (Para carregar um *engine* com a informação e modelos pretendidos fazendo uso da função `loadGroup`).

A função principal desta nova versão do *parser* é a função `loadGroup` que permite carregar toda a informação de um grupo para o motor gráfico, esta função é recursiva e utiliza índices para poder caracterizar os grupos e os seus respectivos sub-grupos, assim como um booleano que indica se o grupo que se pretende fazer *load* é um sub-grupo ou não para desta forma fazer a indexação correta, que permite agrupar os objetos pretendidos.

3 Suplemento de movimentos da Câmara

De modo a permitir a movimentação por todo o mundo que criámos e de modo a elaborar uma funcionalidade extra para o trabalho, implementámos uma câmara em primeiro pessoa.

Esta câmara permite que, usando as setas do teclado, se possa olhar em diversas direções e usando outras teclas do teclado, seja possível movimentar a

câmara em qualquer direção, bem como aumentar e diminuir a velocidade com que a movimentação é feita, bem como reiniciar todas os paramêtros iniciais.

Para representar tanto a posição da câmara(`p`), como a sua direção(`l`, ou *line of sight*), guardamos os valores das 3 variáveis em vetores tridimensionais. De modo a associar a cada tecla (setas, w, a, s, d, ...) a sua respetiva ação (função), efetuamos o mapeamento entre tecla e respetiva função.

Para atualizar o movimento da câmara temos 4 funções, uma para cada sentido, que atualizam o vetor p, sendo que as fórmulas que usamos para movimentar para frente, trás, esquerda e direita são respetivamente:

- **Frente:** Para recriar o movimento em frente, é somado à posição da câmara, `p`, o vetor de *line of sight* escalado com a velocidade do motor gráfico, *speed*. Originando:

$$\text{PosCamara} \longleftarrow \text{PosCamara} + \textit{speed} \cdot \textit{line of sight} \tag{1}$$

- **Trás:** Semelhante ao exemplo ao anterior, movimento no sentido para trás corresponde simplesmente a subtrair da posição atual o vetor da linha de visão.

$$\text{PosCamara} \longleftarrow \text{PosCamara} - \textit{speed} \cdot \textit{line of sight} \tag{2}$$

- **Esquerda:** Mais desafiante que os examplos anteriores, isto porque para criar este movimento foi necessário utilizar o operador de produto externo. Considerando o vetor $\overrightarrow{UP} = \begin{bmatrix}0 & 1 & 0\end{bmatrix}^T$, corresponde ao eixo y, desta forma o movimento na esquerda corresponde a subtrair da posição da câmara o vetor correspondente à linha de visão.

$$\text{PosCamara} \longleftarrow \text{PosCamara} - \textit{speed} \cdot (\overrightarrow{UP} \times \textit{line of sight}) \tag{3}$$

- **Direita:** De forma análoga, movimento para a direita correspondente ao mesmo processo, só que somar em vez de subtrair.

$$\text{PosCamara} \longleftarrow \text{PosCamara} + \textit{speed} \cdot (\overrightarrow{UP} \times \textit{line of sight}) \tag{4}$$

A variável *speed* é aquela que nos permite aumentar ou diminuir a velocidade com que a câmara muda de posição ou direção.

Para atualizar a direção para onde a câmara está a olhar, utilizamos a seguinte atribuição, sendo que α corresponde ao ângulo alterado quando olhamos para a direita ou esquerda, e β corresponde ao ângulo alterado quando olhamos para cima ou para baixo. Resultando em:

$$\textit{line of sight} = \begin{bmatrix} sin(\alpha) \\ -sin(\beta) \\ -cos(\alpha) \end{bmatrix} \tag{5}$$

No entanto, como estamos a trabalhar com 2 ângulos α e β, relativamente à rotação e inclinação, temos as seguintes atualizações para cada ação, associada a cada seta.

- **Up:** Como estamos a atualizar a linha de visão consoante o simétrico do seno de β, então ao diminuir β estamos a aumentar a inclinação. Utilizamos $\frac{1}{10}$ de forma a mitigar os efeitos da velocidade sobre a *line of sight*. Originando:

$$\beta \longleftarrow \beta - \frac{speed}{10} \tag{6}$$

- **Down:** De forma semelhante, olhar para baixo corresponde a incrementa o β.

$$\beta \longleftarrow \beta + \frac{speed}{10} \tag{7}$$

- **Left:** Quando olhamos para a direita estamos a diminuir o ângulo α, então:

$$\alpha \longleftarrow \alpha - \frac{speed}{10} \tag{8}$$

- **Right:** O que leva, de forma simétrica, a:

$$\alpha \longleftarrow \alpha + \frac{speed}{10} \tag{9}$$

Com estas funções e parâmetros somos capazes de definir a movimentação da câmara bem como a direção para onde está a "olhar", conseguindo controlar a velocidade com que são realizamos estas ações.

A posição para onde a câmara está a olhar corresponde a:

$$\text{PosFutura} = \text{PosCamara} + \textit{line of sight} \tag{10}$$

À função `gluLookAt` é passada a seguinte matrix, respectivamente de acordo com os argumentos. A primeira linha corresponde à posição do olho da câmara, a segunda corresponde ao ponto para onde a câmara está olhar, a última corresponde ao vetor vertical referencial, neste caso y.

$$\begin{bmatrix} \text{PosCamara}^T \\ \text{PosFutura}^T \\ \overrightarrow{UP}^T \end{bmatrix} \tag{11}$$

3.1 Movimento Natural

De forma a conseguir implementar um movimento natural, permitindo pressionar várias teclas simultâneamente dentro do nosso ecossistema, utilizamos várias estruturas de `unordered_set` especificamente, utilizamos quatro conjuntos, com os seguintes objetivos:

- **2 Conjuntos:** Visam representar todas teclas dispóniveis no sistema, tanto *ASCII keys* como *non-ASCII keys*.

- **2 Conjuntos:** Visam englobar todas as teclas que estão a ser pressionadas, quando uma tecla passa a estado UP, então abandona este conjunto, particionado em dois conjuntos, um para cada tipo de teclas.

Inicialmente, as teclas a mapear no *Engine* são indicadas utilizando as funções `build_key_mappers` e `build_special_mappers` presentes na classe *EngineMotion*. Preenchendo assim os primeiros dois conjuntos.

De seguida, preenchemos os seguintes conjuntos com as teclas pressionadas, utilizando o seguinte algoritmo, simplificado:

1. Se tecla pressionada e tecla está mapeada no *Engine*.
   – Então, adiciona a tecla ao conjunto das teclas pressionadas.
2. Se a tecla é levantada e tecla está mapeada no *Engine*
   – Então, tecla é removida do conjunto de teclas pressionadas.

Posteriormente, o conjunto de teclas pressionadas são processadas pelo `render Scene`, fornecendo assim o a ação composta do conjunto de teclas pressionadas.

4 Demonstração Tecnológica

Para a criação de uma Scene representativa do Sistema Solar foram utilizados dados reais e diferentes escalas aplicadas aos mesmos. Os dados utilizados foram os seguintes:

Corpo Celeste	Raio(Km)	Distância ao Sol (UA)
Sol	696,340	
Mercúrio	2,439.7	0.39
Vénus	6,051.8	0.723
Terra	6,371	1
Marte	3,389.5	1.524
Júpiter	69,911	5.203
Saturno	58,232	9.539
Urano	25,362	19.18
Neptuno	24,622	30.06
Plutão	1,188.3	39.53

Tabela 1. Valores de referência usados.

As escalas usadas aos planetas traduzem-se nestas duas formulas:

- Para o **tamanho**,

$$\ln\left(1 + \frac{\text{Raio do Planeta}}{\text{Raio da Terra}}\right) \tag{12}$$

- Para a **distância**,

$$30 \cdot \ln\left(1 + \text{Distância Ao Sol}\right) + \text{RaioSol} \tag{13}$$

Para o Sol foi usada a mesma fórmula para o tamanho, e depois multiplicado por dois o valor resultante. Para os planetas gasosos foi feito um ajuste ao tamanho, somando um ao valor de escala.

As luas representadas na Scene são as seguintes:

Corpo Celeste	Luas
Mercúrio	–
Vénus	–
Terra	Lua
Marte	*Phobos*, *Deimos*
Júpiter	*Europa*, *Callisto*, *IO*, *Ganymede*
Saturno	*Mimas*, *Enceladus*, *Tethys*, *Dione*, *Rhea*, *Titan*, *Lapetus*
Úrano	*Miranda*, *Ariel*, *Umbriel*, *Titania*, *Oberon*
Neptuno	*Proteus*, *Triton*
Plutão	*Charon*

Tabela 2. Luas consideradas na demonstração.

A escala usada para o tamanho das luas é a mesma usada para o tamanho dos planetas. A escala usada para a distância entre a lua e o planeta é conseguida através da seguinte fórmula:

$$\frac{\frac{\text{Distância ao Planeta}}{1000} + \text{Raio do Planeta}}{\text{Raio do Planeta}} \tag{14}$$

Para representar os corpos celestes na Scene, foi usado apenas o modelo de uma esfera, com raio de 1, 50 stacks e 50 slices.

5 Imagens do Protótipo

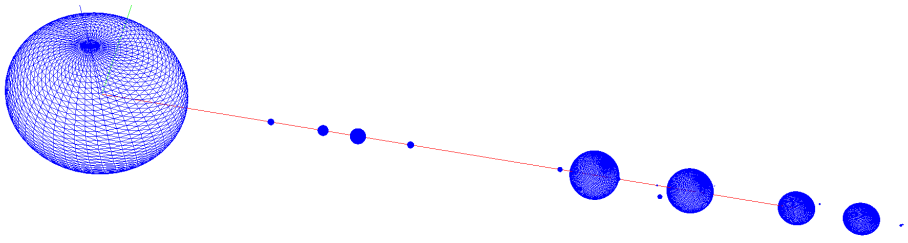

Figura 1. Primeira imagem da demonstração.

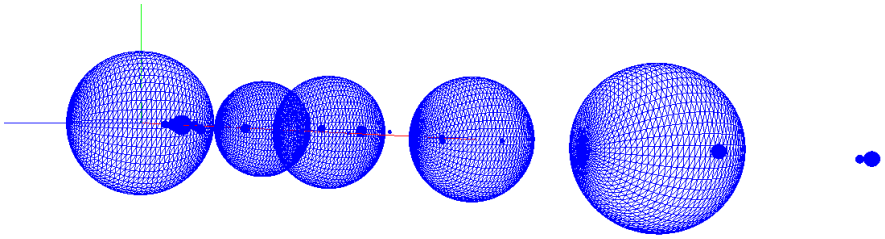

Figura 2. Segunda imagem da demonstração.

6 Conclusões e Trabalho Futuro

Tendo em conta todos os aspectos envolvidos no desenvolvimento do projecto, achamos que conseguimos alcançar todos os pontos necessários com precisão.

Em especial, tendo em conta que já tínhamos agilizado as VBOs com indexação na primeira fase, conseguimos facilmente organizar a arquitetura da nossa Aplicação.

Para além do mínimo pedido nesta 2ª fase, conseguimos com sucesso implementar um movimento da câmara semelhante a um movimento *first person*, disponível em qualquer tipo de jogo. Tirando também partido de uma peça de software que permite pressionar várias teclas simultâneamente.

Como trabalho futuro, achamos que estamos mais permeabilizados a fases seguintes, visto que conseguimos adiantar trabalho nesta fase, o que certamente nós permitirá futuramente desenvolver mais eficazmente peças de software necessárias.