Universidade do Minho

Escola de Engenharia

Mestrado Integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica

Relatório do Trabalho Prático

2016/2017

GRUPO 35

Pedro Cunha A73958, José Silva A74601, Luís Fernandes A74748, João Coelho A74859

31 de março de 2017

Índice

Introdução

A incidência desta segunda fase do trabalho prático são as transformações geométricas. Para pôr em prática este conceito, serão criadas cenas hierárquicas, isto é, cenas definidas como árvores, em que cada nodo poderá possuir, além de um conjunto de transformações geométricas associadas, um conjunto de nodos aninhados (*nodos-filho*).

Particularizando, o objetivo recai na criação de uma cena baseada no sistema solar, estático, com a representação do Sol, planetas e luas, definidas com hierarquia. Como requisito extra procurar-se-á adicionar à cena elementos representativos da cintura de asteroides.

Fase 2

1) Descrição do processo de leitura

A primeira fase do algoritmo de leitura passa por validar se o ficheiro xml a ler foi aberto com sucesso. Caso não seja possível abrir o ficheiro, é detetado o erro e o processo para. Considerando que o ficheiro foi corretamente carregado pelo motor, a sua interpretação começa pelo carregamento para a raiz da árvore de hierarquias do elemento na tag “scene”, uma vez que temos apenas uma cena.

A partir deste momento, começa o processo recursivo que opera sobre os grupos (tag “group”) da nossa cena. Entrando num grupo – e é claro que este processo se inicia no primeiro grupo que surge no xml da cena – todas as transformações neste grupo são carregadas pela ordem que aparecem no ficheiro. Dado que a figura a desenhar pode, na verdade, ser um conjunto de figuras, existe uma tag “models” que alberga todos os modelos (tag “model”) com a designação dos ficheiros com os vértices a desenhar, ao contrário do que acontecia no xml da primeira fase do trabalho. O parser entra nessa secção com os vários modelos e carrega os vértices da primeira figura. Depois, figura a figura, vai carregando todos os vértices a desenhar, até que não sobrem modelos e possa abandonar a tag “models”. Para cada um dos modelos são também carregadas as cores, se forem fornecidas no xml. Salienta-se que os vértices de cada modelo só são carregados uma vez, independentemente de quantas vezes o modelo é referido no xml, para uma maior eficiência de leitura e de memória.

Dado que o grupo de trabalho decidiu usar uma tag “random” para gerar n objetos num determinado raio interior e exterior e com um tamanho que varia entre dois valores dados, o parser também deteta essa tag quando está a carregar as transformações e carrega toda a informação necessária, fornecida por essa tag, para o “group” em questão. É importante referir que esta informação da tag “random” aplica-se a cada um dos “models” desse “group”, o que é explicado em mais detalhe quando se fala do ciclo de render.

Carregadas as transformações e os vértices, caso o grupo não contenha nodos aninhados, o parser procurará o próximo grupo “pai” e o processo repetir-se-á. Caso contrário, o processo repete-se, mas desta feita sobre o(s) subgrupo(s) do grupo. Cada subgrupo poderá ter também os seus subgrupos, portanto trata-se de um processo recursivo que leva o parser a ler os conteúdos descritos acima até que atinja o nodo final do último subgrupo. O processo de leitura termina quando não se encontram mais tags “group” como “pai”, que acontece após terem sido percorridos pelo parser todos os grupos.

2) Descrição das estruturas de dados

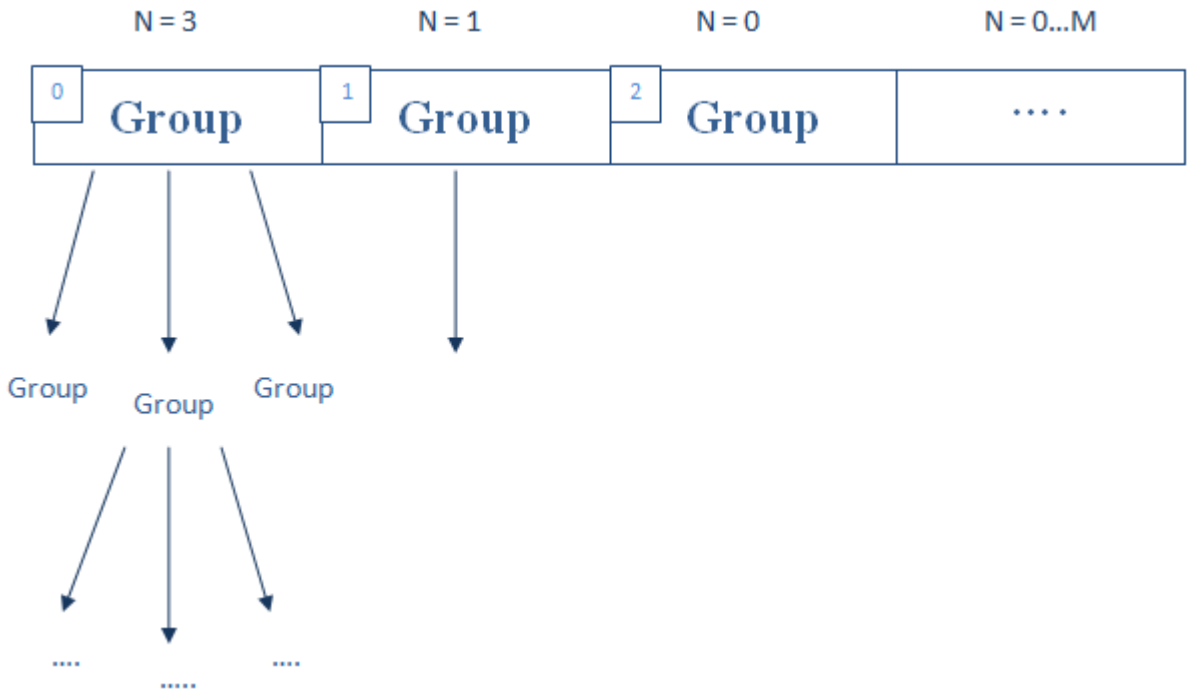
A estrutura de dados sobre a qual assenta a implementação é uma classe “Group”. Quando o parser percorre o ficheiro xml, todos os grupos “pai” são lidos e colocados num vetor desta classe. Por sua vez, um “Group” engloba:

1) o conjunto de transformações (rotates, translates e scales) associadas ao modelo a desenhar, guardadas num vetor da classe “Transformation”. Esta classe é caracterizada por um método virtual (transform()), cuja função principal é simplificar a codificação do algoritmo. Declarando uma subclasse, desta classe, por cada transformação e associando-lhe a função do OpenGL responsável pela sua aplicação, sempre que se pretenda realizar uma transformação aplica-se o método transform(), que saberá que função chamar consoante a subclasse que o invoca. Nota para o facto de as cores terem sido consideradas transformações, a fim de poderem ser aplicadas também através do método transform();

2) um vetor de pares , que constitui o conjunto dos vértices a desenhar. A sua inclusão num par deve-se ao facto de podermos desenhar diferentes modelos num grupo, pelo que podemos querer que tenham cores também diferentes. Assim, com um par é possível associar ao ficheiro com o modelo a desenhar a cor que se pretende para esse modelo. As coordenadas dos vértices estão, naturalmente, no ficheiro do modelo, também elas dentro de uma estrutura própria, que neste caso simplesmente guarda 3 floats, um por coordenada xyz;

3) os subgrupos, ou grupos “filho”, no caso de existirem. Por exemplo, no caso do xml do sistema solar, alguns planetas possuem luas, que são incluídas dentro do grupo do planeta sobre o qual orbitam, para que as transformações associadas a estas possam ser feitas relativamente ao referencial do planeta. Como podem ser vários subgrupos, existe um vetor para este efeito.

Assim, este vetor acaba por ser um conjunto de árvores da classe “Group”, que graficamente pode ser representado por:



3) Descrição do ciclo de *rendering*

O ciclo de rendering começa iterando todos os grupos “pai” obtidos através do parser do xml, que estão guardados num vector. Para cada um destes grupos é chamada uma função(renderGroup) recursiva. Na função renderGroup, a primeira ação é chamar a função glPushMatrix, com o objetivo das transformações do grupo atual não afetarem grupos “irmãos” deste. Pelo mesmo motivo, a última ação é chamar a função glPopMatrix. Entre estas duas funções, começa-se por aplicar as transformações do grupo. De seguida, para todos os files que este grupo contém, são desenhados os vértices com as cores correspondentes a cada file. Ainda entre estas funções, é chamada esta mesma função(renderGroup) para todos os subgrupos, num ciclo. Isto porque os subgrupos herdam todas as transformações do seu grupo “pai”.

Visto que um grupo pode conter informação para gerar de forma aleatória n figuras num determinado espaço, antes de aplicar transformações e desenhar, verifica-se se se trata do caso random. Se este grupo tiver essa informação, é então chamada a função renderRandom que trata de aplicar translações e scales aleatórios dentro dos parâmetros pretendidos(dados no xml) a cada figura desse mesmo grupo(para cada figura as transformações têm que estar entre glPushMatrix e glPopMatrix, para não afetar outras figuras). Destaca-se que os scale tem que ser aplicado depois do translate para não causar conflito.

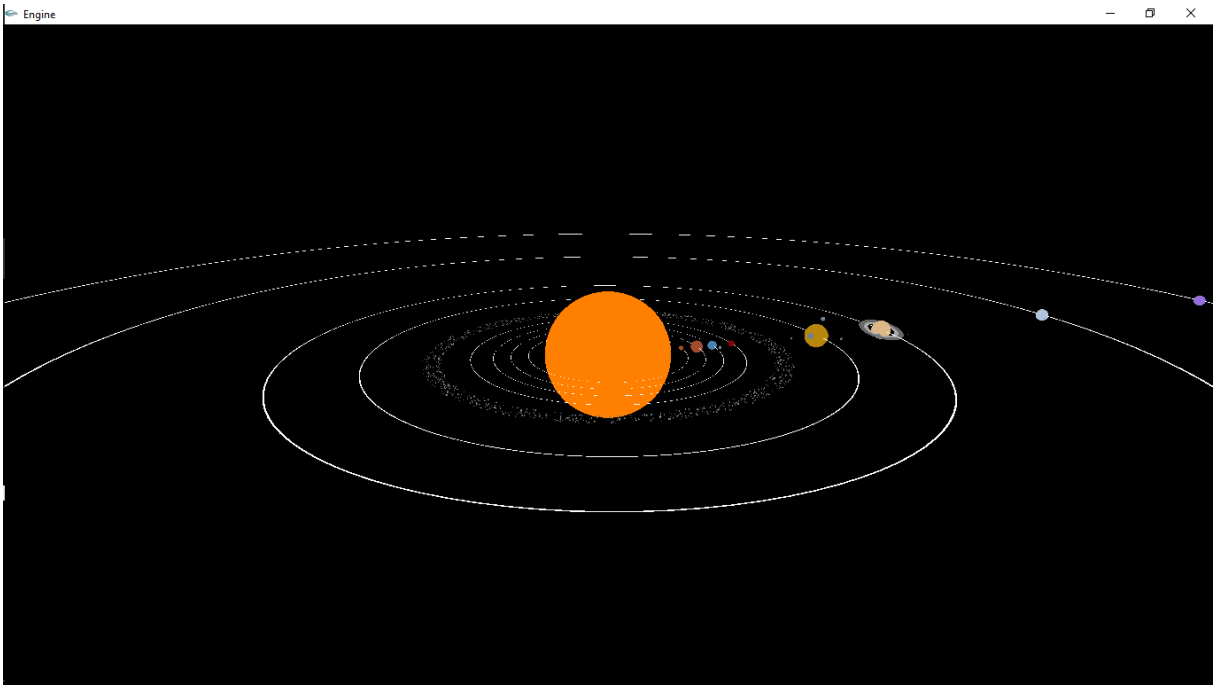
No final da função renderRandom, volta-se à renderGroup e já não se faz as habituais transformações de grupo e desenho, visto que foi tudo efetuado na função. Passa-se então diretamente para a iteração pelos subgrupos. Como já foi referido, a função renderGroup termina chamando glPopMatrix(). O ciclo de render termina quando renderGroup é chamada para todos os grupos “pai” no vector. Assim, todos os grupos foram renderizados recursivamente pela ordem necessária para herdar transformações de grupos acima dos mesmos na árvore.

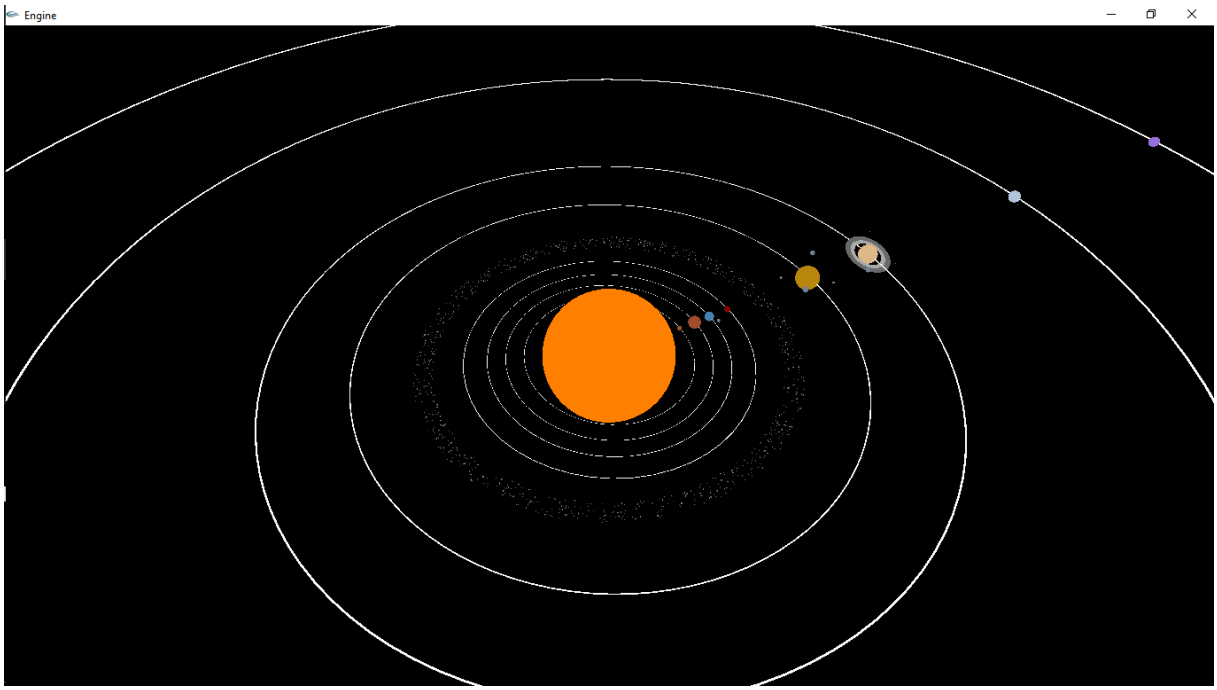
4) Informações adicionais

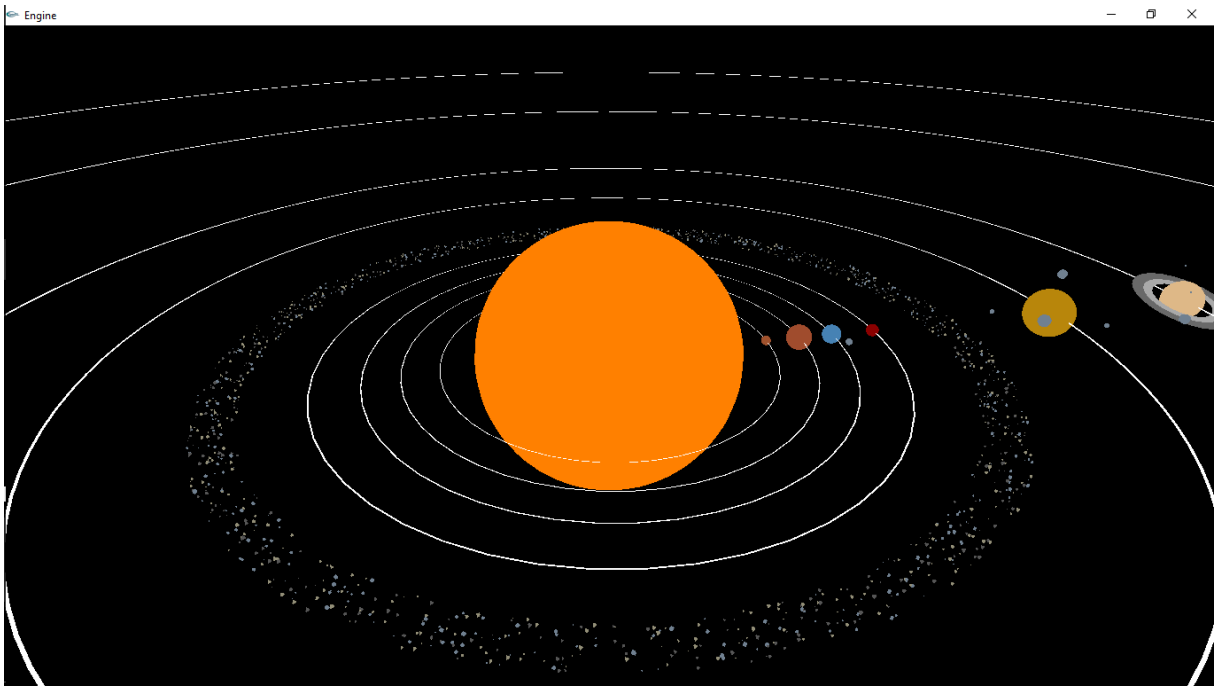
Importante será, também, referir que foi usada a tabela que se segue, nomeadamente a coluna das distâncias em cm, para a definição das distâncias entre os planetas no nosso xml.

Planeta	Tamanho dos Planetas	Distância Aproximada em Relação ao Sol	Distância Entre os Planetas	Dividindo pela escala de 1cm = 20.000.000 km	Distância entre os Planetas em cm
Mercúrio	8º maior	60.000.000 km	**60.000.000 km**	60.000.000 :20.000.000	**3.0 cm**
Vênus	6º maior	110.000.000 km	**40.000.000 km**	40.000.000 : 20.000.000	**2.5 cm**
Terra	5º maior	150.000.000 km	**50.000.000 km**	50.000.000 : 20.000.000	**2.0 cm**
Marte	7º maior	230.000.000 km	**80.000.000 km**	80.000.000 : 20.000.000	**4.0 cm**
Ceres		**(?)**	**(?)**	**(?)**	**(?)**
Júpiter	1º maior	780.000.000 km	**550.000.000 km**	550.000.000 : 20.000.000	**27.5 cm**
Saturno	2º maior	1.430.000.000 km	**650.000.000 km**	650.000.000 : 20.000.000	**32.5 cm**
Urano	3º maior	2.900.000.000 km	**1.470.000.000 km**	1.470.000.000 : 20.000.000	**73.5 cm**
Netuno	4º maior	4.500.000.000 km	**1.600.000.000 km**	1.600.000.000 : 20.000.000	**80.0 cm**
Plutão	Menor	5.900.000.000 km	**1.400.000.000 km**	**(?)**	**(?)**
Eris		**(?)**	**(?)**	**(?)**	**(?)**
Nevem de Oort		**(?)**	**(?)**	**(?)**	**(?)**

Por fim, apresentam-se agora algumas imagens do resultado final do trabalho elaborado:




Engine


Engine

Conclusão

Nesta segunda fase do projeto foi possível ao grupo desenvolver ou aprimorar a já adquirida prática no desenvolvimento de código em C++, bem como, e principalmente, ao nível das transformações geométricas, anteriormente faladas translate, rotate e scale. A realização dos guiões proposto nas aulas práticas da unidade curricular por parte de todos os elementos do grupo serviu de suporte e foi fundamental para que todos os requisitos associados a esta fase do projeto fossem cumpridos.

De um modo geral, fazemos uma apreciação positiva desta segunda fase do trabalho dado que o modelo desenvolvido do sistema solar, que serve de base às restantes fases do projeto, irá permitir, na nossa opinião, uma versão final bastante satisfatória.

É possível demonstrar pelos pormenores adicionais no projeto a satisfação e interesse do grupo pelo tema e pela unidade curricular em si.