Universidade do Minho
Escola de Engenharia

Computação Gráfica

Fase 2
Grupo 7

Braga, Abril de 2023

Bernardo Amado Pereira da Costa, A95052
Eduardo Miguel Pacheco Silva, A95345
José Carlos Gonçalves Braz, A96168

Índice

1. Introdução

Nesta segunda fase do trabalho foi-nos pedido para desenvolver cenários gráficos 3D através da leitura e interpretação dum ficheiro XML. Este terá presente várias transformações (*translate, rotate, scale*) que deverão ser aplicadas através dos nossos ficheiros *engine* e *engineMaterials*.

Comparando com a fase anterior, foi necessário criar dois novos *scripts* de código denominados de *engineMaterials.cpp* e *engineMaterials.h*. Mais à frente abordaremos as mudanças realizadas nos ficheiros, bem como a estrutura de dados que decidimos usar para armazenar os dados lidos.

Para além dos objetivos propostos, achamos por bem adicionar uma funcionalidade extra que é a mostragem dos FPS na renderização da cena. Provavelmente, em fases mais adiantadas adicionaremos mais funcionalidades de modo que o projeto esteja mais completo e apelativo.

2. Ficheiro XML

Os ficheiros XML de teste foram-nos providenciados pela equipa docente, mas é importante abordar a estrutura dos mesmos, pois serão parte importante no armazenamento e processamento de informação que retiramos dos mesmos. Em baixo, apresentamos um exemplo dum XML de teste (*test_2_4.xml*):

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="10" z="4" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" /> <!-- optional, use these values as default-->
        <projection fov="60" near="1" far="1000" /> <!-- optional, use these values as default-->
    </camera>

    <group>
        <transform>
            <translate x="0" y="1" z="0" />
        </transform>
        <group>
            <transform>
                <rotate angle="45" x="0" y ="1" z = "0" />
                <translate x="3.5" y="0" z="0" />
            </transform>
            <models>
                <model file="box_2_3.3d" /> <!-- generator box 2 3 box_2_3.3d -->
            </models>
        </group>
        <group>
            <transform>
                <rotate angle="90" x="0" y ="1" z = "0" />
                <translate x="3.5" y="0" z="0" />
            </transform>
            <models>
                <model file="box_2_3.3d" /> <!-- generator box 2 3 box_2_3.3d -->
            </models>
        </group>
        <group>
            <transform>
                <rotate angle="135" x="0" y ="1" z = "0" />
                <translate x="3.5" y="0" z="0" />
            </transform>
            <models>
                <model file="box_2_3.3d" /> <!-- generator box 2 3 box_2_3.3d -->
            </models>
        </group>
```

Figura 1-test_2_4.xml

Os ficheiros XML utilizados para representar uma cena neste âmbito têm na sua essência uma estrutura em árvore. Cada nodo contém um conjunto de transformações geométricas (*translate, rotate, scale*) e modelos (*file.3d*). Para além disso, cada nodo pode também ter nodos filhos aninhados que herdam transformações de nodos anteriores. Como se pode perceber, a ordem das transformações é importante e o armazenamento das mesmas terá que ser cuidadoso de modo a não alterar essa mesma ordem.

Através da figura verifica-se que cada nodo corresponde a um *group* que é definido por um conjunto de elementos (transformações e modelos). Também podemos observar a constituição de cada uma das 3 primitivas das transformações:

- A primitiva ***translate*** tem como função estabelecer a translação de um objeto através de um vetor (definido por x, y e z).
- A primitiva ***rotate*** permite estabelecer a rotação de um objeto através de um ângulo (angle) e um vetor de rotação (definido por x, y e z).
- A primitiva ***scale*** estabelece a escala de um objeto através de escalares (x, y e z).

3. Leitura e Interpretação do Ficheiro XML

Nesta fase foi necessário realizar algumas alterações e atualizações em ficheiros fulcrais como os responsáveis por realizar *parsing* dos ficheiros XML fornecidos e *Engine.*

3.1. ParseXML

Recorremos novamente à biblioteca *tinyxml-2* para realizar o *parsing* do ficheiro. As informações extraídas são armazenadas num objeto de classe *World* e num objeto de classe *Content. World* armazena informações sobre a janela e a câmera. *Content* armazena informações sobre as transformações e modelos que compõem o mundo.

A função *parseGroup* é responsável por ler informações sobre as transformações e modelos contidos num "*group*". Ela começa por inserir uma matriz de transformação *push* na *stack*. De seguida, procura nas transformações por "*translate*", "*rotate*" e "*scale*" e extrai as informações necessárias. Após isso, temos outra condição relativa a encontrar *models*. Continuamos a procura e desta vez é relativa a subgrupos. No caso de existirem, recorremos a recursividade. Após isto ainda fazemos a verificação da existência de elementos "*group"* ao mesmo nível e recorremos de novo a recursividade. Por fim, já que fornecemos a instrução "PUSH_MATRIX" inicialmente, chega a hora de enviar um "POP_MATRIX".
Por sua vez, a função *parseWorld* analisa primeiramente os constituintes do *world* como a janela e a câmera (posição e afins) para depois invocar a função *parseGroup.*
A função *parseGroup* recebe um ponteiro para um elemento *"group"* do arquivo XML e um ponteiro para um objeto de classe *"Content"* que é usado para armazenar as informações de transformação dos objetos da cena.
Adicionamos também funções relativas ao *parsing* das cores e uma relativa à aleatoriedade dos constituintes da Cintura de Kuiper, bem como dos modelos. Estas estão expressas como *_parse_color*, *_parse_3dCircRandObjPlac* e *_parse_models* respetivamente.

3.2. EngineMaterials

A classe *Content* é o cerne deste ficheiro. Aqui definimos as funções que usaremos no parseXML.cpp. Estas traduzem ações como rotações, translações, *scaling,* push matrix e pop matrix. Teremos também um método denominado de *applyContent* que aplica as operações anteriormente referidas numa ordem correta para a renderização de objetos tridimensionais. Teremos também, como seria previsível, classes referentes ao *World*, *Window* e *Camera*.

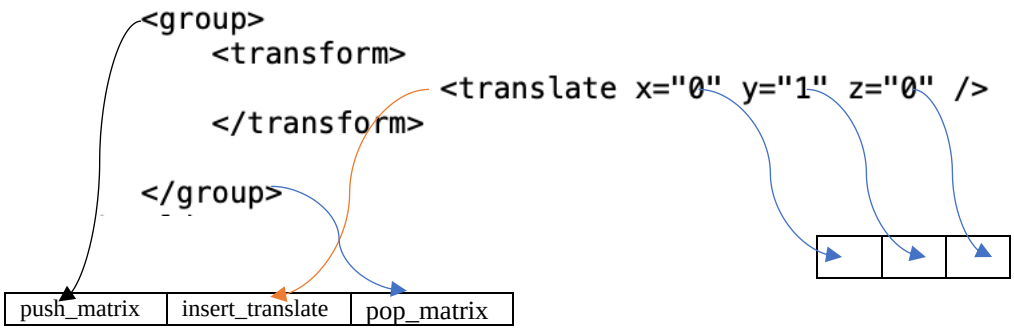
3.3. Engine

Neste ficheiro optamos por incluir a mostragem dos FPS. Para tal, construímos um método denominado de displayFrameRate(). Fizemo-lo porque consideramos que seria uma adição valiosa para o projeto no sentido de o tornar mais completo.

3.4. Arquitetura Utilizada

Acima explicamos o código e as alterações realizadas. No entanto, é necessário especificar a estrutura que utilizamos para ir processando e armazenando as transformações que são realizadas. Optamos pelo uso de dois vetores, sendo que um vai conter valores (relativos às transformações) e o outro vai conter as transformações bem como as instruções de push_matrix e pop_matrix e *models*.

A título de exemplo muito simples, apresentamos um excerto bastante simples dum XML e o que aconteceria consequentemente na estrutura:



Podemos observar pelo esquema acima que push_matrix e pop_matrix são ações sem valores associados e com propósito de orientação e sequencialidade de leitura do XML. Por sua vez, as células referentes a *insert_translate*, *insert_rotate* e *insert_scale* terão valores associados que estarão presentes no array values. Assim sendo, *scale* e translate teriam 3 valores associados (coordenadas x, y e z) e *rotate* teria 4 valores associados (o ângulo e as 3 coordenadas).

4. Sistema Solar

4.1. Implementação de valores e componentes

De modo a concretizar de maneira fiel a modelação do Sistema, recorremos (tal como já foi possível observar) a funções como *translate*, *rotate* e *scale*. Usamos também uma transformação denominada de *color* que pretende representar as cores de cada planeta. Assim, desenhamos cada planeta através de esferas, aplicando depois transformações que tiveram em conta diversos fatores. A considerar: *translate* refere-se às distâncias entre planetas, *rotate* a diferentes fases orbitais e *scale* às dimensões com que os mesmos são representados comumente.

Para uma representação adequada das cores, utilizamos códigos de cores RGB, adaptando os mesmos ao *OpenGL*.

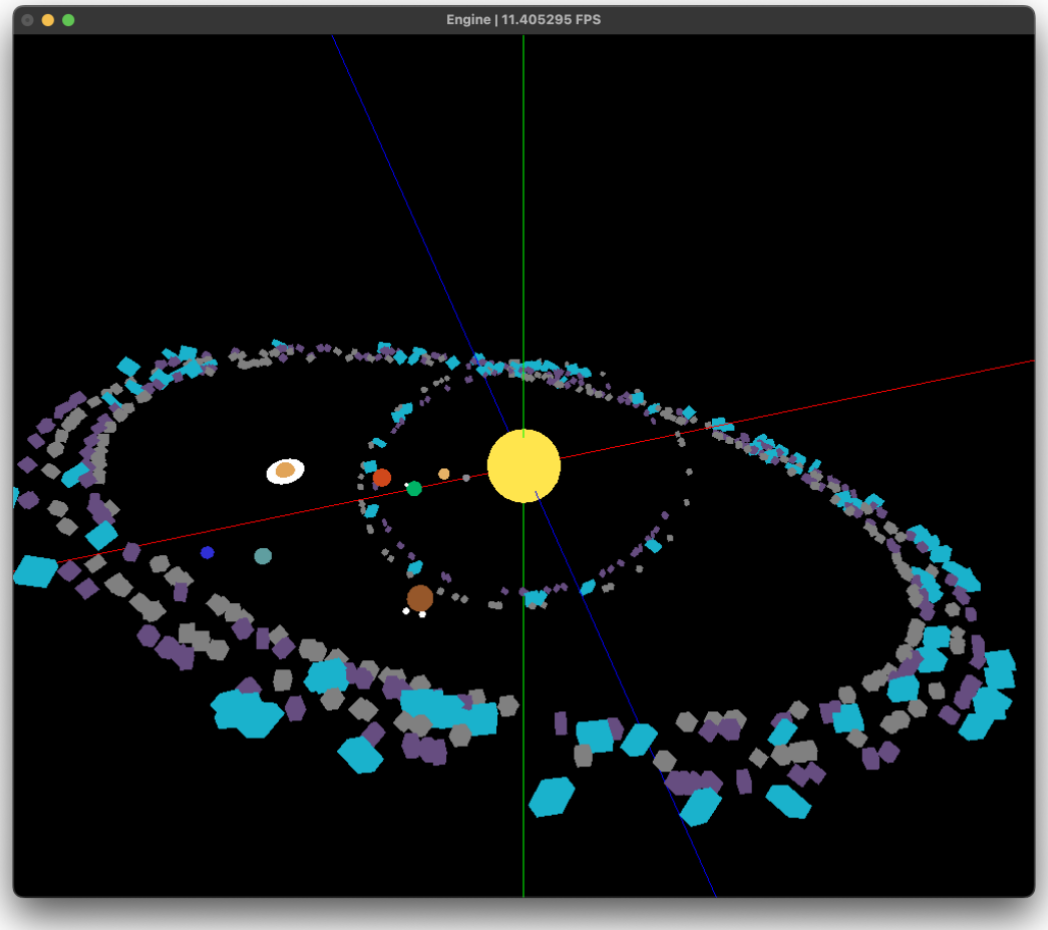
4.2. XML representativo do Sistema Solar

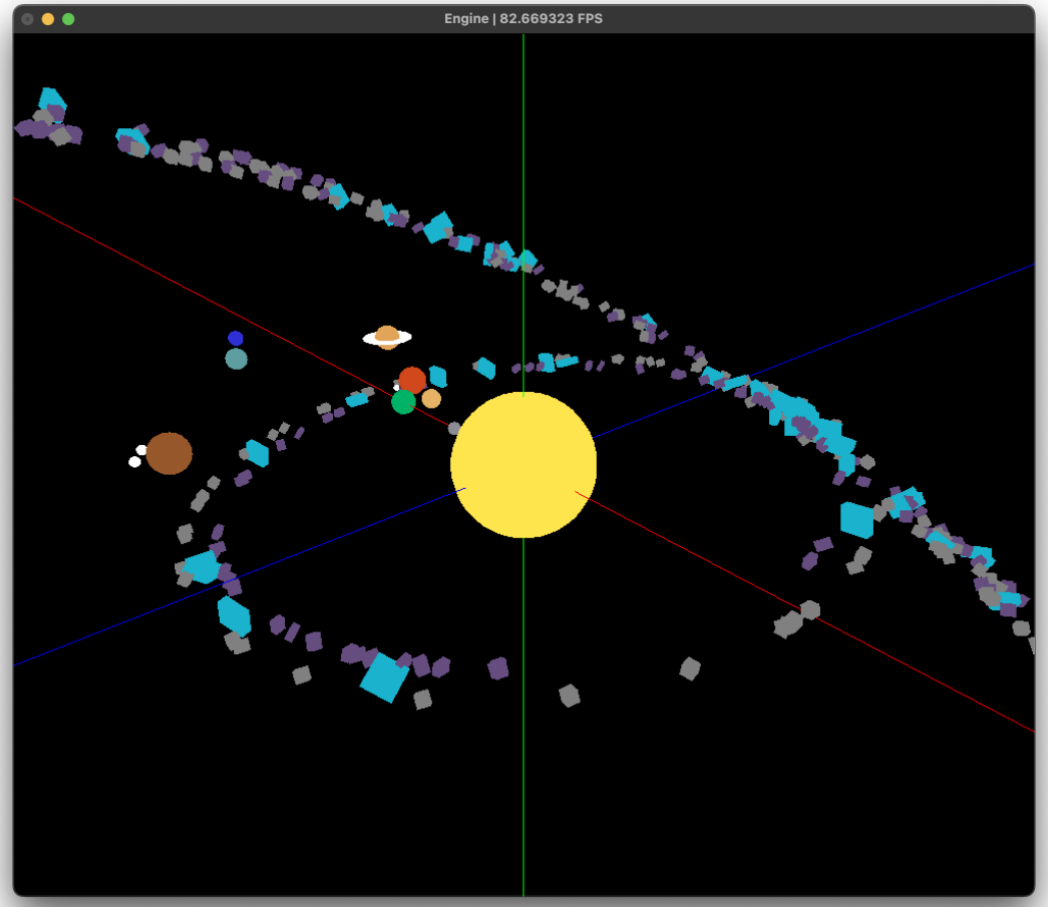
Para representar graficamente o Sistema Solar escrevemos um ficheiro XML tendo em atenção as escalas e cores que estão presentes nas representações do mesmo. A organização deste ficheiro rege-se pela distância dos planetas ao Sol, representando este primeiro, como se pode ver abaixo pelo excerto exemplificativo:

```
<group> <!-- SUN GROUP -->
    <transform>
        <scale x="5" y="5" z="5" />
    </transform>
    <models>
        <model file="sphere_1_20_20.3d">
            <color r="1" g="0.9" b="0.3" />
        </model>
    </models>
</group>
<group> <!-- MERCURY -->
    <transform>
        <translate x="8" y="0" z="0" />
        <scale x="0.5" y="0.5" z="0.5" />
    </transform>
    <models>
        <model file="sphere_1_20_20.3d">
            <color r="0.55" g="0.55" b="0.58" />
        </model>
    </models>
</group>
<group> <!-- VENUS GROUP -->
    <transform>
        <rotate x="0" y="0" z="1" angle="8" />
        <translate x="11" y="0" z="0" />
        <scale x="0.75" y="0.75" z="0.75" />
    </transform>
    <models>
        <model file="sphere_1_20_20.3d">
            <color r="0.9" g="0.7" b="0.4" />
        </model>
    </models>
</group>
<group> <!-- EARTH AND MOON GROUP -->
    <transform>
        <translate x="15" y="0" z="0" />
    </transform>
    <group> <!-- EARTH GROUP -->
        <models>
            <model file="sphere_1_20_20.3d">
                <color r="0" g="0.7" b="0.4" />
            </model>
        </models>
    </group>
    <group> <!-- MOON GROUP -->
        <transform>
            <rotate x="0" y="0" z="1" angle="45" />
            <translate x="1.35" y="0" z="0" />
            <scale x="0.25" y="0.25" z="0.25" />
        </transform>
        <models>
            <model file="sphere_1_20_20.3d" />
```

Figura 2- test_2_solar.xml

4.3. Representação do Sistema Solar





Como é possível observar, os planetas encontram-se em posições diferentes que representam uma trajetória orbital dos mesmos. O seu tamanho difere, bem como as suas cores. É importante ressalvar também que os FPS encontram-se representados também, sendo eles aproximadamente 82 e 11 nos momentos de captura de imagem.
A cintura de Kuiper apresenta-se através duma série de caixas geradas.

4.4. Extra – Cintura de Kuiper e Space Ring

Para conferir mais realismo e interesse à nossa *demo scene* decidimos integrar a Cintura de Kuiper e os anéis de Saturno. Para isso, recorreu-se às já mencionadas transformações geométricas, translações, escalas e rotações, de modo a posicionar cada primitiva corretamente. Para a cintura de Kuiper utilizamos o método *d3CircRandObjPlac*. Julgamos serem adições que valorizam o nosso trabalho e o completam. Adicionamos, de seguida, *screenshots* do XML onde se mostram os parâmetros passados para a renderização deste extra:

```
<group> <!-- SATURN GROUP -->
    <transform>
        <rotate x="0" y="1" z="0" angle="345" />
        <translate x="32" y="0" z="0" />
        <scale x="1.25" y="1.25" z="1.25" />
    </transform>
    <models>
        <model file="sphere_1_20_20.3d">
            <color r="0.882" g="0.643" b="0.345" />
        </model>
        <model file="ring_1_2_50_3.3d" disableCull="True"/>
    </models>
</group>
```

```
<group> <!-- ASTEROID BELT GROUP -->
    <transform>
        <rotate angle="-5" x="1" y="0" z="1"/>
    </transform>
    <d3CircRandObjPlac radius="22" n="50" isRandRotation="True" scaleX="0.3" scaleY="0.3" scaleZ="0.3">
        <models>
            <model file="box_2_3.3d">
                <color r="0.5" g="0.5" b="0.5"/>
            </model>
        </models>
    </d3CircRandObjPlac>
    <d3CircRandObjPlac radius="20" n="50" isRandRotation="True" scaleX="0.2" scaleY="0.3" scaleZ="0.4">
        <models>
            <model file="box_2_3.3d">
                <color r="0.4" g="0.3" b="0.5"/>
            </model>
        </models>
    </d3CircRandObjPlac>
    <d3CircRandObjPlac radius="21" n="20" isRandRotation="True" scaleX="0.8" scaleY="0.8" scaleZ="0.3">
        <models>
            <model file="box_2_3.3d">
                <color r="0.1" g="0.7" b="0.8"/>
            </model>
        </models>
    </d3CircRandObjPlac>
</group>
```

4.5. Interação com o utilizador

Optamos por recorrer à utilização do rato. De maneira bastante intuitiva é possível mudar as posições da câmera, girando-a para a direita ou para a esquerda com o *touchpad* e fazendo zoom-in e zoom-out com o botão direito do rato.

5. Informação destinada aos utilizadores de Mac OS

A invocação da biblioteca *glew* não é necessária num contexto Mac OS. Assim sendo, quem necessite de testar o projeto num ambiente Mac OS deverá retirar as linhas `#include <GL/glew.h>` que se encontram no engine.cpp bem como as linhas `glewInit()` e `glEnableClientState(GL_VERTEX_ARRAY)`.

6. Conclusão

Consideramos importante efetuar uma análise crítica do trabalho realizado.

O programa funciona corretamente e a implementação está intuitiva e em concordância com a estrutura do XML. Conseguimos implementar uma estrutura de dados simples de trabalhar e manipular que nos permitirá, possivelmente, mais tarde realizar alterações de escala se necessário.

Apesar de terem sido necessárias algumas adaptações relativamente à primeira fase, consideramos que o trabalho desenvolvido foi satisfatório, tendo sido superados todos os requisitos.