Universidade do Minho
Escola de Engenharia

Trabalho Prático - Grupo 32

Fase 2 - Transformações Geométricas

Computação Gráfica

Bruno Filipe de Sousa Dias A89583
Luís Enes Sousa A89597
Pedro Miguel de Soveral Pacheco Barbosa A89529

4 de abril de 2021

Conteúdo

1 Introdução

Nesta fase, de forma a dar continuidade à primeira fase do trabalho prático, tivemos que evoluir o nosso projeto, tornando possível criar um cenário, usando transformações geométricas (translações, rotações e escalamentos).

O objetivo final é apresentar um modelo estático do Sistema Solar, incluindo o Sol, planetas e luas.

2 Alterações na Estrutura do Projeto

Nesta secção, serão abordadas as alterações que foram efetuadas na estrutura do projeto, para alcançar os objetivos finais.

2.1 Generator

De uma forma geral, a única alteração no Generator foi a introdução de mais uma primitiva geométrica, o Toro (este será abordado mais detalhadamente na secção seguinte).

2.1.1 *generator.cpp*

Neste ficheiro, foi adicionada a opção de criar um Toro a partir dos argumentos passados ao programa, tal como acontece para as outras primitivas. O menu de ajuda também foi atualizado, agora mostrando as instruções para contruir um Toro.

2.1.2 *primitives.cpp*

Neste ficheiro, foram adicionadas duas funções. Uma delas cria os pontos necessários para desenhar um Toro e retorna-os num *vector*. A outra é, apenas, uma função auxiliar da função anterior.

2.2 Engine (Carregamento dos Modelos)

Nesta fase do projeto, tivemos que adaptar a leitura dos ficheiros *XML*, pois estes deixam de ter apenas informação relativamente aos modelos que devem ser carregados e passam a ter também as transformações geométricas aplicadas a estes.

Tendo isto em conta, não era possível continuar a guardar os modelos num *vector* de *vector* de pontos, pois seria impossível aplicar as transformações corretamente, a cada modelo. Desta forma, decidimos criar uma classe *Group*, capaz de guardar as transformações, os modelos e até outros grupos, relativos a um certo grupo.

Também decidimos criar uma classe *Object*, para guardar os pontos de cada objecto, permitindo-nos ter um código mais organizado.

2.2.1 *engine.cpp*

Neste ficheiro, e de forma a aplicar as alterações referidas acima, passamos a ter uma variável global *groups_vector* que guarda todos os grupos encontrados no ficheiro *XML* carregado. A função de *parsing* do ficheiro *XML* (*loadXMLFile*) também foi alterada, de forma a ler corretamente a nova estrutura do ficheiro *XML*.

2.2.2 *group.cpp*

Nestes ficheiros, podemos encontrar a implementação da classe *Group*, que guarda as informações relativas a cada grupo retirado do ficheiro *XML*. Esta classe guarda as transformações geométricas (pela ordem com que aparecem no ficheiro) e a cor dos objetos pertencentes ao dado grupo. Estes objetos são também guardados num *vector*. É ainda possível haver vários grupos dentro de um grupo.

2.2.3 *object.h*

Neste ficheiro, encontramos a declaração da classe *Object*. Esta guarda um *vector* de pontos e uma descrição do que representam esses pontos.

2.3 Engine (Câmara)

Devido à elevada complexidade do cenário requisitada para esta fase do projeto (um Sistema Solar estático), sentimos necessidade de facilitar a sua visualização, permitindo ao utilizador observar todos os seus detalhes.

Sendo assim, decidimos implementar duas câmaras: uma que olha sempre para a origem (no caso do Sistema Solar, o Sol); e outra mais livre, permitindo ao utilizador navegar pelo espaço.

2.3.1 *engine.cpp*

Neste ficheiro, a alteração mais relevante é o facto de todas as variáveis e funções que influenciam o comportamento da câmara terem sido removidas, sendo enviadas para os respetivos ficheiros. O *engine* apenas usa a interface da câmara, pois todos os cálculos são feitos pela classe correspondente.

2.3.2 Câmara estática: *staticCamera.cpp*

Este tipo de câmara foi usado na fase anterior. Contudo, nesta fase, todos os cálculos são efetuados na classe *staticCamera*, tal como todas as variáveis necessárias. Temos, ainda, a funcionalidade extra de guardar o estado atual da câmara para um ficheiro e carregá-lo no futuro.

Esta funcionalidade cria um ficheiro *.cfg*, para guardar os valores das variáveis relativas à câmara. Este pode, depois, ser lido pelo programa, carregando os valores para as respetivas variáveis.

2.3.3 Câmara livre: *fpsCamera.cpp*

Esta câmara permite ao utilizador navegar pelo espaço e observar melhor o cenário apresentado. O utilizador pode olhar à sua volta com rato, navegar paralelamente ao plano xOz e alterar a sua posição relativamente ao eixo Oy.

Todas estas funcionalidades tornam esta câmara mais complexa do que a anterior, pois é necessário calcular, não só a sua posição, como também o ponto para onde esta olha.

Para tal, são necessárias várias variáveis, para guardar o estado atual da câmara:

- **Ângulo α (*alpha*)**

  Este ângulo tem valor nulo quando a câmara aponta no sentido positivo do eixo Oz e cresce no sentido anti-horário, se olharmos a partir da metade positiva do eixo Oy. Em relação à movimentação da câmara, afeta as coordenadas x e z.

- **Ângulo β (*beta*)**

  Este ângulo tem valor nulo no sentido positiva do eixo Oy e cresce no sentido horário, se olharmos a partir da metade positiva do eixo Oz. Na nossa implementação, limitamos este ângulo da seguinte forma, de maneira a evitar complicações com a orientação da câmara:

  $$\pi/32 <= \beta <= \pi - \pi/32$$

  Não tem influência quanto à movimentação da câmara, pois esta move-se sempre paralelamente ao plano xOz.

- **Sensibilidade (*sensitivity*)**

  Permite regular o incremento dos valores *alpha* e *beta*, alterando a sensibilidade da câmara ao movimento do rato.

- **Posição no eixo Ox (*eyeX*)**

  Este valor representa a posição da câmara em relação ao eixo Ox.

- **Posição no eixo Oy (*eyeY*)**

  Este valor representa a posição da câmara em relação ao eixo Oy.

- **Posição no eixo Oz (*eyeZ*)**

  Este valor representa a posição da câmara em relação ao eixo Oz.

- **Velocidade (*speed*)**

  Permite regular o incremento dos valores $eyeX$, $eyeY$ e $eyeZ$, alterando a velocidade de translação da câmara.

- ***startX* e *startY***

  Guardam a posição anterior do rato, de forma a podermos calcular a distância percorrida pelo rato num dado movimento.

- **Array *tracking***

  Caso o utilizador clique no botão esquerdo do rato, a posição 0 passa a ter valor 1, indicando que devemos registar os movimentos do rato na horizontal e atualizar os valores de *alpha*. Quando o utilizador soltar o botão esquerdo, a variável passa a 0 e interrompemos este processo.

Caso este clique no botão direito, executamos o mesmo procedimento para a posição 1. Neste caso rastreamos os movimentos na vertical e atualizamos a variável *beta*.

O utilizador pode clicar nos dois botões ao mesmo tempo, movendo a visão da câmara em ambas as direções.

Tal como acontece com a câmara estática, este modo livre também permite guardar e carregar o seu estado atual.

3 Primitivas Geométricas Adicionadas

Embora não fosse requisitado, explicitamente, no enunciado desta fase, sentimos necessidade de adicionar uma primitiva geométrica ao nosso projeto, para obtermos uma representação mais realista do Sistema Solar.

3.1 Toro

Para a construção do Toro assumimos uma abordagem semelhante, em certa medida, à da Esfera, pois dividimos a figura em fatias e processamos cada fatia individualmente. Cada fatia, por sua vez, está dividida em camadas. Quando todas as camadas de uma dada fatia são processadas, continuamos para a fatia seguinte, seguindo o mesmo processo, até toda a figura ser processada.

No início, começamos por calcular o ângulo de cada fatia e de cada camada. Depois, começando o processamento de uma fatia, calculamos o ângulo inicial e final dessa fatia. Prosseguimos, então, para o processamento de todas as camadas da fatia. Mais uma vez, calculamos o ângulo inicial e final de cada camada.

No processamento de uma dada fatia e uma dada camada, temos que desenhar um trapézio. Para tal, calculamos os seus quatro pontos, usando os ângulos calculados anteriormente e, ainda, o raio interior e exterior, que são passados como argumentos da função. Usando os quatro pontos calculados, conseguimos desenhar dois triângulos, equivalentes ao trapézio.

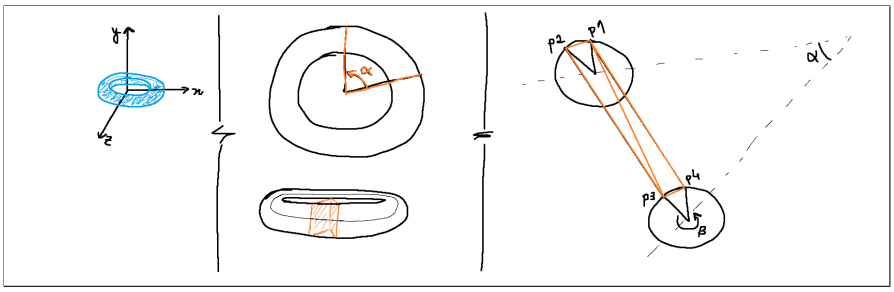

Figura 1: Esboço da construção do Toro

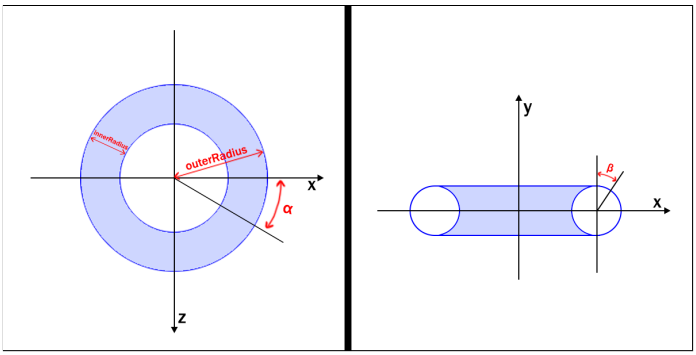

Figura 2: Explicação da construção do Toro

4 Estrutura e *Parsing* do Ficheiro *XML*

4.1 Estrutura

O ficheiro *XML* contém um único elemento *scene*. Este elemento pode conter vários elementos *group*.

Cada *group* é composto por:

- ***translate***

  Este elemento apresenta três atributos: X, Y e Z Estes atributos representam a translação dos modelos em relação ao eixo correspondente.

- ***rotate***

  Este elemento apresenta quatro atributos: *angle*, *axisX*, *axisY* e *axisZ*. O atributo *angle* representa a rotação, em graus, em torno do eixo formado pelos outros três atributos.

- ***scale***

  Este elemento apresenta três atributos: X, Y e Z. Estes atributos influenciam o escalamento dos modelos em relação ao eixo correspondente.

- ***color***

  Este elemento apresenta três atributos: R, G e B. O atributo R influencia a intensidade da cor vermelha, o atributo G a cor verde e o atributo B a cor azul, no formato RGB.

  Este elemento foi adicionado pelo nosso grupo, pois achamos relevante para representar o Sistema Solar e diferenciar os diferentes planetas e luas.

- ***models***

  Este elemento apresenta um conjunto de elementos *model*. Estes são os modelos que devem ser carregados para este grupo.

  O elemento *model* é composto por: um atributo *file*, que indica o nome do ficheiro com os pontos que devem ser carregados; um atributo *description*, que oferece uma descrição dos pontos que serão carregados do ficheiro.

  O atributo *description* do elemento *model* foi adicionado pelo nosso grupo, pois facilita o *debugging* do programa, permitindo perceber melhor o que representam dados pontos. Este é, no entanto, opcional, não influenciando a execução do programa caso não esteja presente.

- ***group***

  Este elemento permite inserir um grupo dentro de outro grupo.

Todos estes elementos podem aparecer nenhuma ou, no máximo, uma vez, exceto o elemento *group*, que pode aparecer um número ilimitado de vezes. Em relação aos seus atributos, também podem ser omitidos, assumindo um valor predefinido.

4.2 *Parsing*

Para o *parsing* do ficheiro *XML*, e tal como na primeira fase, usamos a biblioteca ***TinyXML2***.

Para começar o *parsing*, tentamos abrir o ficheiro. Em caso de fracasso, abortamos a operação.

Depois, tentamos obter o elemento *scene*. Mais uma vez, em caso de fracasso, abortamos a operação.

Caso consigamos abrir o ficheiro e obter o elemento *scene*, vamos percorrer os vários elementos *group* nele contidos, guardando-os, depois de tratados, num *vector* global.

4.2.1 *Parsing* de um elemento *group*

Para o *parsing* de um grupo, tentamos encontrar cada um dos elementos referidos anteriormente.

1. **Elemento *translate***

   Neste elemento, tentamos obter os seus três atributos. Caso um atributo não seja encontrado, assume o valor predefinido *0*. No final, a translação é adicionada ao grupo em questão.

   Caso este elemento não esteja presente, a translação é ignorada para este grupo.

2. **Elemento *rotate***

   Neste elemento, tentamos obter os seus quatro atributos. Caso um atributo não seja encontrado, assume o valor predefinido *0*. No final, a rotação é adicionada ao grupo em questão.

   Caso este elemento não esteja presente, a rotação é ignorada para este grupo.

3. **Elemento *scale***

   Neste elemento, tentamos obter os seus três atributos. Caso um atributo não seja encontrado, assume o valor predefinido *1*. No final, o escalamento é adicionado ao grupo em questão.

   Caso este elemento não esteja presente, o escalamento é ignorado para este grupo.

4. **Elemento *color***

   Neste elemento, tentamos obter os seus três atributos. Caso um atributo não seja encontrado, assume o valor predefinido *0*. No final, a cor é adicionada ao grupo em questão.

   Caso este elemento não esteja presente, a cor é definida com o valor predefinido *R=1, G=1 e B=1* (branco).

5. **Elemento *models***

   Neste elemento, tentamos obter os seus vários elementos *model*.

Para cada elemento, tentamos obter os seus dois atributos. Caso o atributo *file* não seja encontrado, não é carregado um ficheiro. Caso seja o atributo *description* em falta, este assume o valor predefinido *"Undefined Object"*.

Caso este elemento não esteja presente, não são carregados modelos para este grupo.

6. **Elemento *group***

   Em relação a este elemento, invocamos a própria função para processar todas as suas repetições, usando recursividade.

5 Construção do Cenário Final

5.1 Etapa 1

Inicialmente, criamos apenas o Sol e os oito planetas principais do Sistema Solar, alinhando-os no eixo *Ox*, com o Sol na origem. Para tal, usamos uma translação para cada planeta, de forma a criar espaço entre estes. Em relação ao tamanho das figuras, e sendo que todas elas usam o mesmo ficheiro de pontos (*sphere.3d*), foi necessário usar um escalamento para cada um.

O comando usado usado para gerar a esfera de referência foi:

./generator sphere 1 32 32 sphere.3d

De notar que, tanto o tamanho dos corpos celestes como a seu distanciamento não está feito à escala, pois seria muito difícil de observar todo o Sistema Solar assim. Decidimos, então, experimentar vários valores, até encontrarmos dimensões que consideramos ser um meio termo entre a representação fidedigna do Sistema Solar e facilidade de visualização deste.

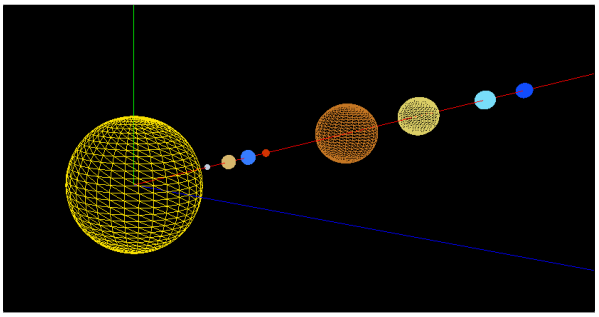

Figura 3: Cenário da Etapa 1

5.2 Etapa 2

Depois de termos uma base sólida, decidimos acrescentar alguns planetas secundários, mais especificamente, a Lua e as quatro Luas de Galileu (Io, Europa, Ganímedes e Calisto). Para a Lua, criamos um grupo dentro do grupo referente à Terra, fazendo com que a Lua fosse afetada pela translação aplicada à Terra. O mesmo se sucedeu para as outras quatro luas, desta vez, para Júpiter.

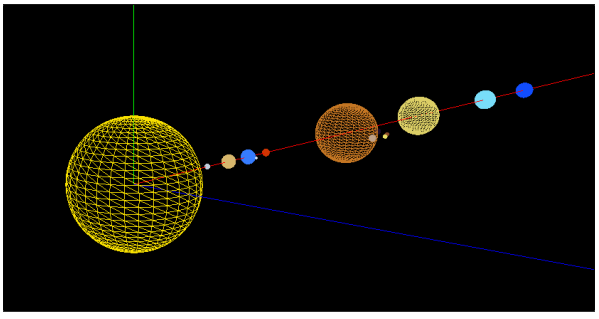

Figura 4: Cenário da Etapa 2

5.3 Etapa 3

Nesta fase do desenvolvimento do cenário, sentimos que era necessário representar os icónicos anéis de Saturno e de Urano. Foi então que decidimos desenvolver a criação da primitiva Toro. Quando concluída, procedemos para a criação dos anéis, usando os comandos apresentados abaixo. Para a sua disposição, seguimos o mesmo raciocínio e procedimentos como na inclusão das luas e criamos um grupo dentro dos grupos de Saturno e Urano.

./generator torus 0.6 1.7 32 32 saturn_ring.3d

./generator torus 0.01 1.7 32 32 uranus_ring.3d

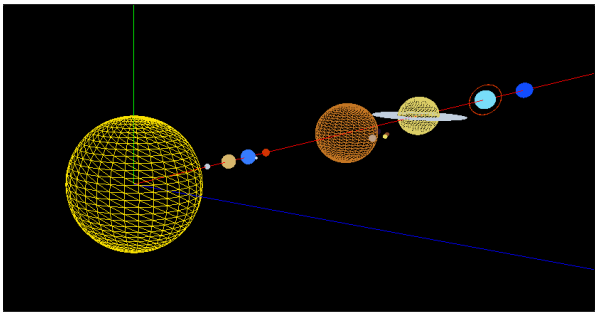

Figura 5: Cenário da Etapa 3

5.4 Etapa 4

Estando satisfeitos com a disposição e coloração de todos os modelos até este momento, achamos conveniente usar o Toro para desenhar as órbitas dos planetas. Criamos, então, uma órbita modelo e usamos o escalamento para a personlizar para cada planeta. Para a criação do ficheiro *orbit.3d*, usamos o seguinte comando:

./generator torus 0.01 1 2 128 orbit.3d

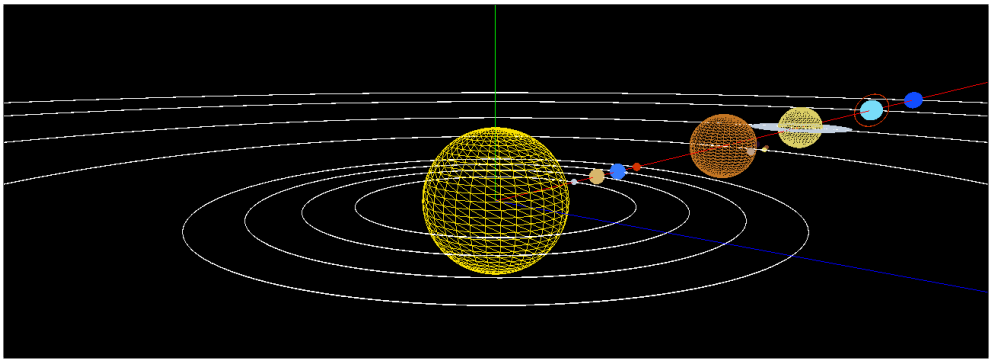

Figura 6: Cenário da Etapa 4

5.5 Etapa 5

Nesta etapa decidimos finalizar o nosso cenário com a disposição não linear dos planetas, espalhando-os pelo espaço. Escolhendo um ângulo e usando as regras da trigonometria, conseguimos perservar a distância de cada planeta ao Sol.

Finalmente, adicionamos a Cintura de Asteróides e ocultamos os eixos. O comando para gerar a Cintura de Asteróides foi o seguinte:

./generator torus 2 25 32 128 asteroid_belt.3d

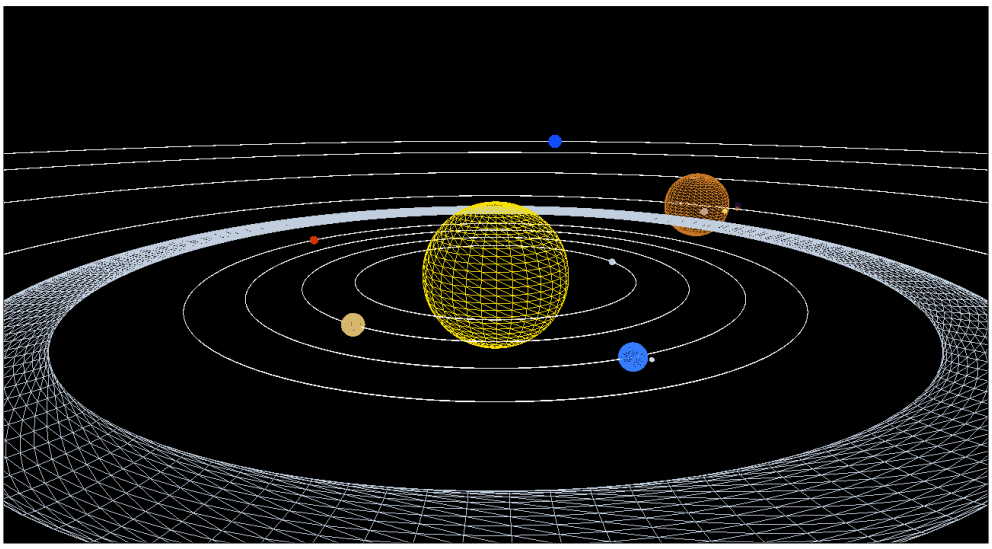

Figura 7: Cenário final com a câmara estática

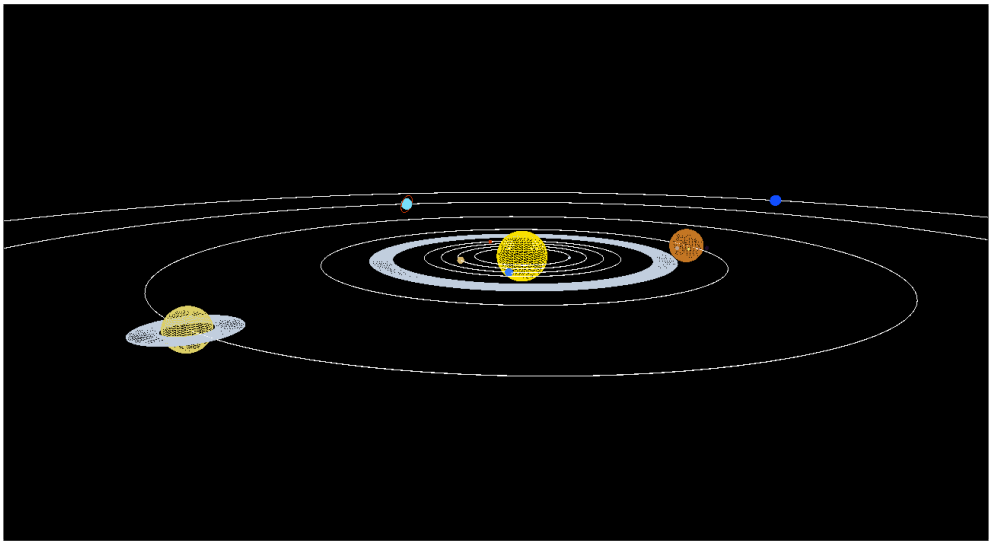

Figura 8: Cenário final com a câmara livre

6 Conclusão e Trabalho Futuro

Concluída a segunda fase do trabalho prático, podemos afirmar que consolidamos os conceitos sobre transformações geométricas que foram abordados nas aulas e sentimos que os nossos conhecimentos da trigonometria foram postos à prova. Também ganhamos mais destreza para trabalhar com a linguagem C++ e com o *GLUT*, devido às dificuldades que fomos ultrapassando ao longo desta fase de trabalho.

Em relação às próximas fases, esperamos ter o trabalho facilitado, pois sentimos que temos um código bem organizado e estruturado, pronto para receber as novas funcionalidades.