Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

Trabalho Prático
Fase 2 - Transformações geométricas

António Filipe Castro Silva(a100533) Diogo Rafael dos Santos Barros(a100600)
Duarte Machado Leitão(a100550) Pedro Emanuel Organista Silva(a100745)

5 de abril de 2024

CG

Índice

Lista de Figuras

1 Introdução

Em relação à primeira fase, inicialmente fizemos algumas melhorias, quer na geração dos planos, quer na geração da esfera, para estas ficarem melhor desenhadas nesta segunda fase, mas, como é algo proveniente da fase anterior, não entraremos em muito detalhe. Simplesmente houve a necessidade de alterar a ordem da escrita de alguns pontos e a adição de alguns pontos no plano para o vermos de várias perspetivas diferentes.

Já sobre esta segunda fase do projeto é nos pedido para continuar a trabalhar com o que formulamos na fase anterior (devido a isso é que fizemos as alterações), mas para acrescentar algumas novas funcionalidades, sendo estas maioritariamente focadas na **Engine** em si.

As novas funcionalidades têm a ver com o facto de que agora a **Engine** tem de conseguir aplicar **transformações geométricas**, que agora aparecem num formato específico nos ficheiros XML, de forma hierárquica aos derivados modelos utilizados.

De forma a conseguirmos ver que desenvolvemos corretamente estas funcionalidades, além de termos corrido os testes dados pelos professores, escrevemos um ficheiro XML que desenha o **Sistema Solar** utilizando as novas funcionalidades que colocamos na **Engine**.

Neste relatório, vamos começar por demonstrar que alterações fizemos na forma de leitura dos dados, quer na estrutura, quer nas transformações geométricas e como esta trata de as desenhar. Também vamos demonstrar como ficaram os nossos testes a partir dos XML dos professores.

Depois disso, mostraremos a forma como tratamos dos anéis para Saturno e Urano, todos os modelos que foram utilizados no Sistema Solar, como ficou o nosso ficheiro XML e o seu resultado final em duas perspetivas diferentes.

2 Mudanças na Engine e nas Utils

Nesta segunda fase, exceto a formulação do Anel, todas as alterações foram feitas na pasta da **Engine** e nas **Utils**, principalmente termos de alterar grande parte do ficheiro **Leitor** para tratarmos das **transformações geométricas** de forma hierárquica e a criação de um novo ficheiro **Groups** para alterar/criar estruturas de dados.

2.1 Novas estruturas de dados e o Group

Tal como na primeira fase, continuamos a ter uma estrutura que guarda os dados que lemos do XML, o **Leitor**. A alteração que aconteceu foi que, em vez de ter uma lista de **files**, passamos a ter um **Group**, pois agora em vez de existir só um group com models, podem existir vários groups dentro de groups, contendo estes variadas transformações, tendo assim em conta as **transformações geométricas e a hierarquia**. Todos os outros campos manteram-se inalterados, estando a sua explicação no relatório anterior.

```
struct leitor {
    float posicao[3];
    float lookAt[3];
    float up[3];
    float projection[3];
    Group group;
};
```

Figura 2.1: Struct Leitor

Logo para tratarmos dessa estrutura Group tivemos de criar um novo ficheiro **Groups**, pois este vai tratar de várias estruturas dentro dele e de muitas partes importantes de como tratamos das transformações geométricas e a sua hierarquia. Uma das alterações que se pode mencionar já sobre o **Group** é que no ficheiro **Engine** obviamente deixou-se de fazer getFiles e passou-se a fazer getNode (a antiga função getFiles, mas agora alterada) para ir buscar esse Group, pois nele é que passarão a estar os ficheiros com o resto dos dados. Claramente também tivemos de alterar a função **criarListaFiguras** para agora funcionar com Group, mas explicaremos isso nas **"Mudanças no Parsing do XML"**.

Para tratar então desses casos foi criado dentro do ficheiro **Groups** primeiramente a estrutura **Group** que tratará de guardar todas as transformações que aparecem num group, utilizando uma lista de Transform, outra estrutura que explicaremos a seguir; terá em conta todos os groups filhos dentro do principal, tendo assim dentro de cada Group uma lista de Group chamada childs (ou seja, os seus filhos diretos), sendo que obviamente estes filhos também puderão ter os seus filhos logicamente; e, por fim, que ficheiro pertence a esse grupo em concreto ou mais do que um, mantendo assim, como já mencionei, de alguma forma a lista de files que existia previamente.

```
struct group{
    std::list<Transform> transform;
    std::list<Group> childs;
    std::list<std::string> files;
};
```

Figura 2.2: Struct Group

A estrutura **Transform** é a estrutura que vai guardar os dados sobre todas as transformações geométricas e primeiramente o TransformType, que é um Enum, vai dizer qual é o tipo de transformação geométrica a fazer. Nesse Enum, existem três tipos: **SCALE**, **TRANSLATE** e **ROTATE**. Dentro da estrutura, temos o **angle** que fica a zero, a não ser que seja um **ROTATE**, onde temos de dar o ângulo de rotação. Por fim, temos o x, y e z, para sabermos em que coordenadas temos de fazer as alterações. Por exemplo, se o TransformType for **Translate** e o **y** for 3 e o resto tudo 0, significa que mudamos as coordenadas de todos os pontos, somente adicionando 3 ao y.

```
struct transform{
    TransformType type;
    float angle;
    float x;
    float y;
    float z;
};

enum class TransformType{
    SCALE,
    TRANSLATE,
    ROTATE
};
```

Figura 2.3: Struct Transform

De resto, no ficheiro group, visto que é um ficheiro totalmente novo, temos vários funções novas que temos de explicar:

- **novoGrupo** - criação da estrutura Group
- **getChild** - buscar uma child em específico de um group
- **getTransform** - buscar a lista de transforms de um grupo
- **getFiles** - buscar a lista de files de um grupo
- **add_transform** - adicionar uma lista de transform ao group
- **add_node** - adicionar uma nova child ao group
- **push_file** - adicionar um novo file ao group
- **novoTransform** - criação da estrutura Transform
- **add_transformType** - adicionar o TransformType ao transform
- **add_transformAngle** - adicionar o valor do angle ao transform
- **add_transformX/Y/Z** - adicionar o valor da coordenada x/y/z ao transform (existe uma função para cada coordenada)

2.2 Mudança no Parsing do XML

Em termos de alterações no parsing do XML, tivemos de alterar duas funções e criar duas novas. As duas novas funções foram criadas devido à existência de vários groups e das transformações geométricas de forma hierárquica. Sendo assim, estas foram chamadas: **extrair_grupo** e **extrair_transform**. As duas funções alteradas foram obviamente a **extrairXML**, pois agora tem de incluir estas duas novas funções e a **criarListaFiguras** como já mencionei previamente.

Primeiramente, vamos resumir como funcionam as duas novas funções e depois explicar as alterações que necessitaram de ser feitas nas outras duas funções:

- **Extrair_transform** - função que ao encontrar um **transform** num XML dentro de **group** começa por criar uma nova estrutura Transform, identifica qual é o TransformType para colocar na estrutura, se for um rotate coloca logo o **angle** na estrutura e, por fim, coloca o valor de todas as coordenadas na estrutura e adiciona este transform à lista de transforms do Group em questão. Esta função está toda dentro de um for para pegar em todas as transformações que se encontram dentro de este **Transform**.

- **Extrair_grupo** - função que primeiramente tenta dar extract a todos os groups dentro do group em questão, sendo assim recursiva em si mesma. Depois trata de extrair os **transforms**, utilizando a função anterior e, para finalizar, pega nos models (ou seja, os ficheiros) que se encontram dentro do grupo que se encontra a processar e que sofrem as transformações daquele group.

- **Extrair_XML** - antigamente esta função simplesmente processava um group que somente tinha um ou mais models(os files). Agora como pode conter vários groups, esta simplesmente encontra o group principal e utiliza a função **extrair_grupo** para tratar do resto, passando-lhe a informação desse group do XML à função e a parte Group da estrutura Leitor onde vai ficar guardada toda a informação processada.

- **criarListaFiguras** - esta função que continua a ser chamada no ficheiro **Engine** depois de termos extraido o XML sofreu algumas alterações, pois agora em vez de receber somente a lista da path dos files, recebe o Group que existe no Leitor depois da leitura do XML. Sendo assim, esta vai continuar a ter uma lista de Figuras, mas vai estar a ver recursivamente dentro das childs de cada Group para obter todos os files e todos os transforms que existiam no XML. Dentro de cada Group existente, ao pegar nesses transforms e nos files, vai criando uma Figura para cada file, mete-a dentro da lista-Figuras dessa child em específico, que posteriormente vai ser colocada na listaFiguras do group principal no final, mas esta já vai utilizar a função **applyTransforms** na lista desta child antes de terminar a função para lhe aplicar as transformações específicas para essa Figura.

2.3 Funcionalidades novas

2.3.1 applyTransforms

Como podemos ver na última função explicada na **Mudança no Parsing do XML**, uma das funcionalidades novas é a aplicação de **transformações geométricas** nas Figuras, que é base desta fase do projeto, sendo que a função que trata da maioria dessas transformações é a tal função **applyTransforms**.

Esta função pega numa lista de figuras e na lista de transformações que é suposto estas sofrerem e vai aplicando uma de cada vez. Antes de aplicar as transformações, tivemos de ir buscar os pontos da figura e colocá-los numa lista de pontos. Também tivemos de ter em conta se a lista de pontos tinha acabado de ser rodada, criando assim uma variável para verificar se esta tinha sido rodada e um Transform para guardar essa rotação. Vamos explicar cada um desses casos:

- **Scale sem rotate anterior**: Pega nos pontos e utiliza as novas funções setX, setY e setZ para alterar os pontos da figura de acordo com os parâmetros dados para cada coordenada, ou seja, se a scale for 0,25 no X, multiplica o valor atual da coordenada X

por 0,25 em todos os pontos e altera na figura (lista de pontos) para esse novo valor.

- **Translate sem rotate anterior**: Pega nos pontos e utiliza as novas funções setX, setY e setZ para alterar os pontos da figura de acordo com os parâmetros dados para cada coordenada, ou seja, se o translate for 3 no X, adiciona ao valor atual da coordenada X 3 em todos os pontos e altera na figura para esse novo valor.

- **Scale com rotação anterior**: se o scale verifica que aconteceu uma rotação, ele cria um vetor com as coordenadas do scale e roda esse vetor com a rotação que foi aplicada previamente utilizando a função **rodarPonto**. Depois de rodado o vetor, tal como quando não há rotação, multiplica o valor do ponto atual pelo valor desse vetor na mesma coordenada e altera os valores do ponto utilizando as funções set, fazendo isto para todos os pontos da figura.

- **Translate com rotate anterior**: se o translate verifica que aconteceu uma rotação, ele cria um vetor com as coordenadas do translate e roda esse vetor com a rotação que foi aplicada previamente utilizando a função **rodarPonto**. Depois de rodado o vetor, tal como quando não há rotação, adiciona ao valor do ponto atual o valor desse vetor na mesma coordenada e altera os valores do ponto utilizando as funções set, fazendo isto para todos os pontos da figura.

- **Rotate**: ao verificar que é um rotate, muda a variável que verifica se houve rotação para um e atualiza o Transform que guarda a rotação para este rotate. Por fim, roda todos os pontos da Figura, utilizando a função **rodarPonto**.

2.3.2 rodarPonto

Como já mencionamos vários vezes, existe a função **rodarPonto** que trata de todas as rotações de pontos. A função começa por pegar num ponto e transformar o ângulo para radianos, depois descobre o vetor de acordo com os parâmetros dados à função e os pontos atuais. Depois utilizando matrizes de rotação, descobre o vetor de rotação. As matrizes que utilizamos já são matrizes depois destas se multiplicarem por um ponto genérico para facilitar cálculos.

Se for para rodar no eixo do X, utiliza-se a seguinte matriz de rotação:

$$\begin{bmatrix} x \\ y * cos(\theta) - z * sin(\theta) \\ y * sin(\theta) + z * cos(\theta) \end{bmatrix}$$

Em código ficou algo deste gênero:

```
if(x > 0){
    vecRotX = vecX;
    vecRotY = vecY * cos(angleRad) - vecZ * sin(angleRad);
    vecRotZ = vecY * sin(angleRad) + vecZ * cos(angleRad);
    x = 0;
}
```

Como já tinha mencionado antes estes parâmetros são calculados deste modo:

```
    float angleRad = angle * M_PI/180;
    float pontox = getX(p);
    float pontoy = getY(p);
    float pontoz = getZ(p);

    float pontoRotX = 0;
    float pontoRotY = 0;
    float pontoRotZ = 0;

    if(x > 0){
        pontoRotX = x;
    }
    else if(y > 0){
        pontoRotY = y;
    }
    else if(z > 0){
        pontoRotZ = z;
    }

    float vecX = pontox - pontoRotX;
    float vecY = pontoy - pontoRotY;
    float vecZ = pontoz - pontoRotZ;
```

Sendo o angle, o x, o y e o z, os parâmetros recebidos pela função rodarPonto para determinar qual o tipo da rotação.

Do mesmo modo, tivemos de fazer as matrizes de rotação se fosse uma rotação no eixo do Y ou no eixo do Z.

Para o eixo do Y a matriz de rotação e o código ficaram deste modo:

$$\begin{bmatrix} x * cos(\theta) + z * sin(\theta) \\ y \\ -x * sin(\theta) + z * cos(\theta) \end{bmatrix}$$

```
else if(y > 0){
    vecRotX = vecX * cos(angleRad) + vecZ * sin(angleRad);
    vecRotY = vecY;
    vecRotZ = -vecX * sin(angleRad) + vecZ * cos(angleRad);
    y = 0;
}
```

Para o eixo do Z a matriz de rotação e o código ficaram deste modo:

$$\begin{bmatrix} x * cos(\theta) - y * sin(\theta) \\ x * sin(\theta) + y * cos(\theta) \\ z \end{bmatrix}$$

```
else if(z > 0){
    vecRotX = vecX * cos(angleRad) - vecY * sin(angleRad);
    vecRotY = vecX * sin(angleRad) + vecY * cos(angleRad);
    vecRotZ = vecZ;
    z = 0;
}
```

Por fim depois de calculado o vetor de rotação, calcula-se os novos pontos, utiliza-se as funções set para alterar esse ponto e verifica-se se já se fez todas as rotações pedidas e, se não for verdade, faz-se novamente a função **rodarPonto**. Para descobrir os novos valores do ponto faz-se estes cálculos:

```
    newPontoX = vecRotX + pontoRotX;
    newPontoY = vecRotY + pontoRotY;
    newPontoZ = vecRotZ + pontoRotZ;
```

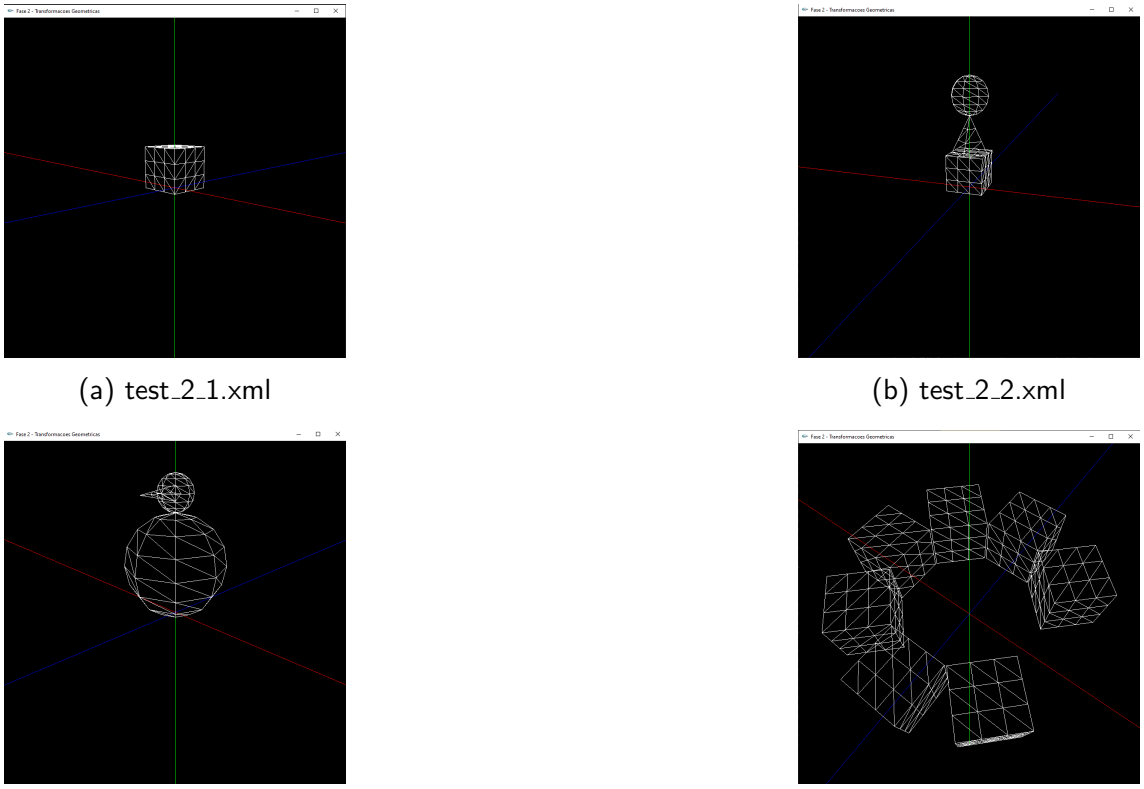
2.4 Demonstração dos Testes Fornecidos

Para verificarmos que o que estavamos a fazer estava correto, corremos o código e verificamos os testes fornecidos pelos professores. Verificamos que teriamos de criar uma nova **Sphere** e criamos uma nova makefile para a 2.ª Fase.

Além da compilação normal e da criação dessa nova sphere, também acrescentamos a criação do planet e do ring que serão necessário para o próximo capítulo do relatório: **Sistema Solar**.

```
build_f2:
    cmake -B FASE2/build -S FASE2/src -A Win32
    cmake --build FASE2/build/ --config Release
    @echo "Build␣FASE2␣done"
    ./FASE2/build/Release/generator.exe sphere 1 8 8
       FASE2/out/sphere_1_8_8.3d
    @echo "Sphere_1_8_8␣done"
    ./FASE2/build/Release/generator.exe sphere 1 25 25
       FASE2/out/planet.3d
    @echo "Planet␣done"
    ./FASE2/build/Release/generator.exe ring 1.5 2 40
       FASE2/out/ring.3d
    @echo "Ring␣done"
```

Nesta fase foram nos dados 4 testes e os nossos resultados foram os seguintes:



(a) test_2_1.xml

(b) test_2_2.xml

(c) test_2_3.xml

(d) test_2_4.xml

Figura 2.4: Teste fornecidos pelos professores

Comparando estes resultados, aos testes dos professores conseguimos ver que tratamos das transformações geométricas de forma hierárquica com sucesso.

3 Sistema Solar

De forma a demonstrarmos com mais extensão todas as funcionalidades desenvolvidas nesta fase do projeto, foi nos pedido que desenvolvessemos uma demo scene do **Sistema Solar**. Para fazermos isso, tivemos de fazer a **formulação do Anel** para colocarmos à volta de Saturno e Urano, criamos uns modelos mais precisos para os planetas e, para realmente demonstrar a demo scene, criamos um XML utilzando transformações geométricas de forma hierárquica para, por exemplo, manter a escala entre os planetas, sendo que este inclue o Sol, todos os planetas de Mercúrio até Neptuno, os anéis em Saturno e Urano e a Lua da Terra.

3.1 Formulação do Anel

Para criamos o Anel criamos uma nova função chamada **generateRing** no **generator** para tratar da criação dessa figura que começa por receber 3 parâmetros: **innerRadius**, **outerRadius** e **divisions**. Estes três parâmetros representam o raio interno do anel, o raio externo do anel e o número de segmentos em que este anel vai ser dividido, ou seja, quantos mais, mais preciso vai ser o anel, mas também mais pontos vai ter e mais complexo de criar vai ser.

Com esses parâmetros, começa por inicializar a Figura ring que vai receber os pontos todos e devolver no fim e depois calcula o angle de cada segmento fazendo o seguinte cálculo:

$$2 * \pi / divisions$$

Depois de calculado o angle, faz-se um ciclo for para adicionar todos os pontos para cada angle, ficando então com o seguinte código:

```
for (int i = 0; i < divisions; i++){
    float currentAngle = angle * i;

    float x1 = innerRadius * cos(currentAngle);
    float z1 = innerRadius * sin(currentAngle);
    float x2 = outerRadius * cos(currentAngle);
    float z2 = outerRadius * sin(currentAngle);

    currentAngle = (i + 1) * angle;


```

```
12      float x3 = innerRadius * cos(currentAngle);
13      float z3 = innerRadius * sin(currentAngle);
14      float x4 = outerRadius * cos(currentAngle);
15      float z4 = outerRadius * sin(currentAngle);
16
17      adicionarPonto(ring, novoPonto(x3, 0, z3)); // Interno
18      adicionarPonto(ring, novoPonto(x1, 0, z1)); // Interno
19      adicionarPonto(ring, novoPonto(x2, 0, z2)); // Externo
20
21      adicionarPonto(ring, novoPonto(x3, 0, z3)); // Interno
22      adicionarPonto(ring, novoPonto(x2, 0, z2)); // Externo
23      adicionarPonto(ring, novoPonto(x4, 0, z4)); // Interno
24
25      adicionarPonto(ring, novoPonto(x2, 0, z2)); // Externo
26      adicionarPonto(ring, novoPonto(x1, 0, z1)); // Interno
27      adicionarPonto(ring, novoPonto(x3, 0, z3)); // Interno
28
29      adicionarPonto(ring, novoPonto(x4, 0, z4)); // Interno
30      adicionarPonto(ring, novoPonto(x2, 0, z2)); // Externo
31      adicionarPonto(ring, novoPonto(x3, 0, z3)); // Interno
32  }
```

Inicialmente, com o valor do ângulo atual e os raios calculamos as coordenadas dos vários pontos com os quais vamos criar os triângulos. Depois com essas coordenadas criamos os triângulos para formular o anel. Os últimos 6 pontos são criados para podermos ver os anel de todas as perspetivas possíveis. Foi a partir destes pontos, que formularam os triângulos, que conseguimos com sucesso para criar o anel que depois utilizariamos para criar o Anel de Saturno e o Anel de Urano.

3.2 Modelos utilizados

Os modelos que utilizamos para a formulação do Sistema Solar foram gerados a partir do **generator**, tal como todos os modelos até agora.

Os parâmetros que foram utilizados para ambos este ficheiros podem ser observados no **makefile** que foi demonstrado previamente, mas vamos agora explicar com mais detalhe. A **sphere** utilizou os parâmetros: 1 de *radius*, 25 *slices* e 25 *stacks* para ficar com mais detalhe, ficando esta figura guardada no ficheiro **planet.3d**. O ring utilizou os parâmetros: 1,5 de *innerRadius*, 2 de *outerRadius* e 40 *divisions* para ficar o mais detalhado possível, mas para ainda se conseguir ver as *divisions* de forma legível, ficando este guardado no ficheiro **ring.3d**.

A **planet.3d** foi utilizada para desenhar o Sol, todos os planetas e a Lua e o **ring.3d** foi utilizado para desenhar os Anéis de Saturno e de Urano.

3.3 Criação do ficheiro XML

Para criar o ficheiro XML que representa o Sistema Solar começamos por, dentro do group principal, criar um group para o Sol, cada planeta e a Lua. Casos como Saturno e Urano vão ter dois groups dentro do seu group para representar o planeta e o anel.

O Sol ficou desenhado no centro e o resto dos planetas ficaram numa escala de tamanho x2, ou seja, como Mercúrio é 0.003x mais pequeno que o Sol, na scale poríamos todas as suas coordenadas a 0.006x de scale. Além disso, depois de Mercúrio que se encontra num translate de 1.25 do Sol, todos os outros planetas encontram-se a mais 0.25 de cada um(exceto a Lua que se encontra um pouco mais próxima da Terra) utilizando uma translação no eixo do X, pois se colocassemos o Sistema Solar à escala nas distâncias entre cada planeta, ficaria pouco percetível.

```
<group>
    <models>
        <model file="../out/planet.3d" />
    </models>
</group>

<group>
    <transform>
        <scale x="0.006" y="0.006" z="0.006" />
        <translate x="1.25" y="0" z="0" />
        <rotate angle="21" x="0" y="1" z="0" />
    </transform>
    <models>
        <model file="../out/planet.3d" />
    </models>
</group>
```

Figura 3.1: Desenho do Sol e de Mercúrio

Além disso, colocamos um angle para que os planetas não se alinhassem e dessem uma melhor perspetiva de que se encontram em órbita. Depois criamos outro ficheiro XML com uma perspetiva em linha em que simplesmente removemos esse angle, alteramos a posição da câmara e não incluimos a Lua.

Para desenharmos Saturno, por exemplo, criamos dois subgrupos dentro do grupo de Saturno em que ambos o planeta e o anel sofrem a mesma escala, translação e angle, mas depois o anel recebe outro rotate no angle em Z para este ficar com a inclinação que tem sobre Saturno na vida real. Respeitamos assim a forma hierárquica que é suposto estes dois elementos sofrerem.

```
<group>
    <transform>
        <scale x="0.16" y="0.16" z="0.16" />
        <translate x="2.5" y="0" z="0" />
        <rotate angle="217" x="0" y="1" z="0" />
    </transform>
    <group>
        <models>
            <model file="../out/planet.3d" />
        </models>
    </group>
    <group>
        <transform>
            <rotate angle="23" x="0" y="" z="1" />
        </transform>
        <models>
            <model file="../out/ring.3d" />
        </models>
    </group>
</group>
```

Figura 3.2: Desenho de Saturno e do seu Anel

3.4 Resultado final do Sistema Solar

Depois de compilado o programa com a Makefile, corremos a **engine** usando os ficheiros da pasta tests_custom: **solar_system.xml** e **solar_side.xml** para visualizarmos o Sistema Solar em órbita e em linha, respetivamente.

3.4.1 Perspetiva orbital

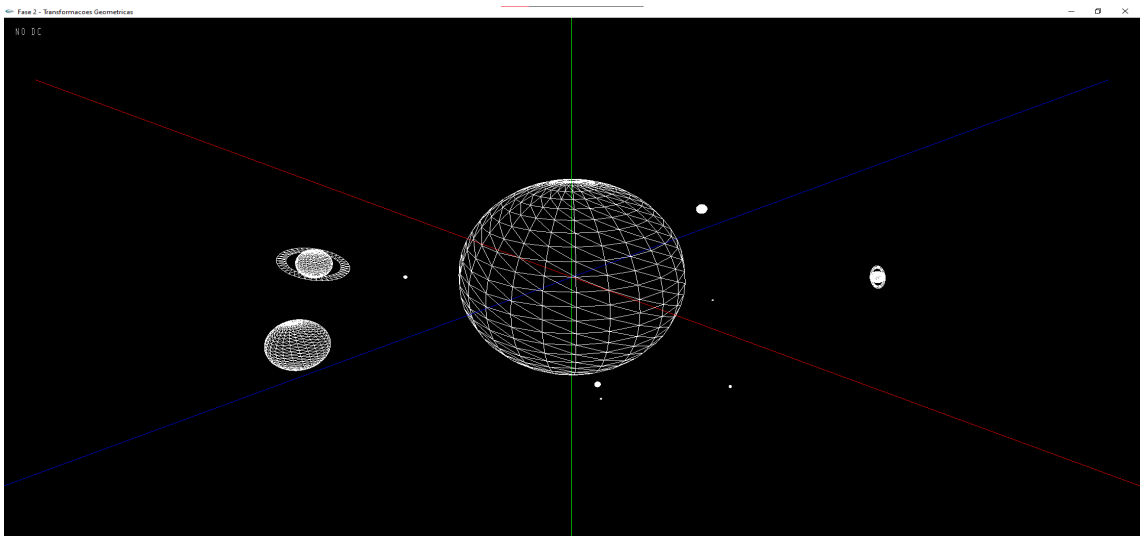


Figura 3.3: Desenho do Sistema Solar em órbita

3.4.2 Perspetiva em linha

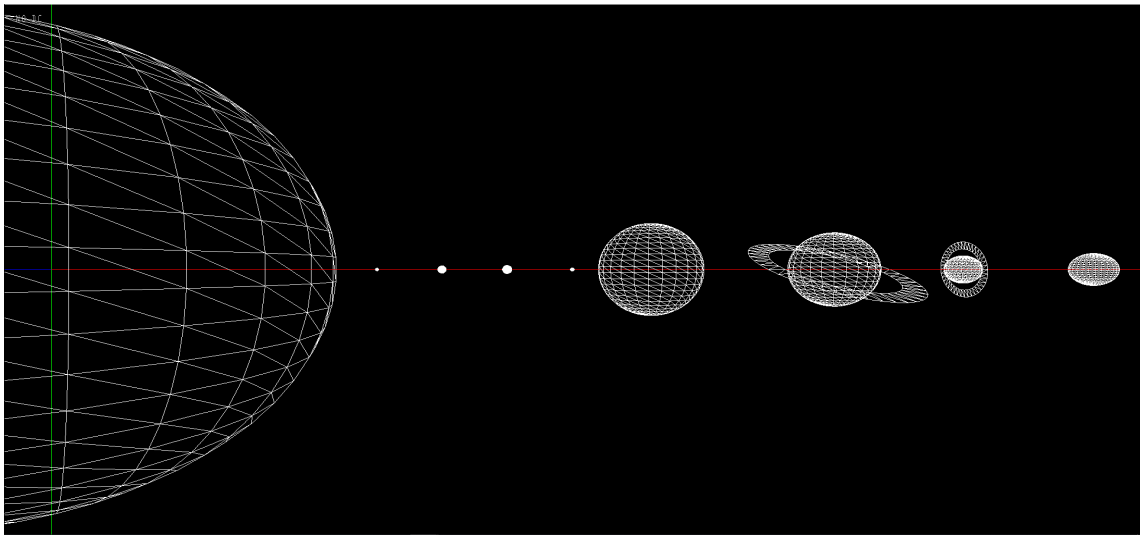


Figura 3.4: Desenho do Sistema Solar em linha

4 Conclusão

Com a resolução desta segunda fase do trabalho prático conseguimos consolidar de que forma funcionam as transformações geométricas de forma hierárquica e como utilizar matrizes de rotação em pontos. Também aprendemos a criar os nossos próprios ficheiros XML tendo por base os dos testes fornecidos pelos professores e a criar uma estrutura (o Ring) mais abstrata que nós necessitamos de forma autónoma.

Em termos de satisfação sobre a produção desta fase, achamos que foi um trabalho bem conseguido, visto que conseguimos implementar todas as funcionalidades pedidas, criamos o anel para o Sistema Solar, visto que não nos fazia muito sentido fazer um Sistema Solar sem os anéis de Saturno, pelo menos. Continuamos a puder rodar a câmara tal como na Fase 1 e fizemos duas perspetivas diferentes do Sistema Solar: em órbita e alinhado. No de órbita, colocamos a Lua da Terra para termos uma das luas pedidas.

Concluindo, tendo em conta o que foi produzido nesta fase, achamos que nos encontramos com tudo o que é necessário para iniciarmos a terceira fase do trabalho prático.