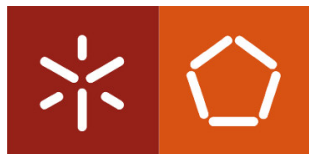


UNIVERSIDADE DO MINHO

ESCOLA DE ENGENHARIA



Computação Gráfica

Licenciatura em Engenharia Informática

Fase 2 - Geometric Transforms

Grupo 12

Alexandre Fernandes - [A94154]

Henrique Vaz - [A95533]

Abril, 2023

Conteúdo

1	Introdução	2
2	Reestruturação do projeto	3
2.1	Parser XML	3
2.2	Reorganização dos módulos	4
3	Nova Primitiva Gráfica	5
3.1	Torus	5
4	Sistema Solar	7
5	Câmara	8
6	Dificuldades e aspetos a melhorar	9
7	Conclusão	10

1. Introdução

Nesta nova fase do desenvolvimento foi proposto a alteração do *engine*, de modo a que este fosse capaz de suportar um mecanismo de criação hierárquica de cenas, recorrendo, para isso, a transformações geométricas.

Resumidamente, isto implica que cada cena poderá agora conter um N número de sub-cenas, podendo estas ter também outro M número de sub-camadas associadas, sendo que em todas elas terão de estar presentes os efeitos produzidos pelas transformações realizadas na cena presente no nível hierárquico imediatamente anterior.

É também importante esclarecer que a ordem pela qual as transformações são realizadas é de extrema relevância, já que o incumprimento deste requisito poderá condicionar fortemente a obtenção dos resultados esperados.

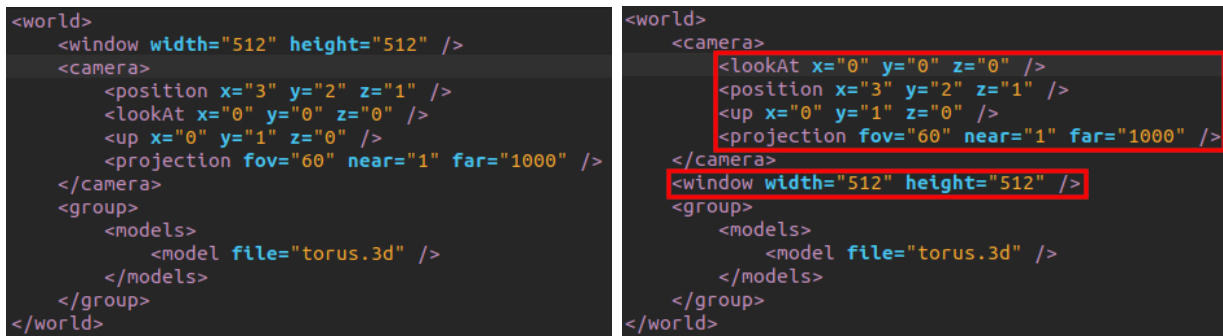
Para que tal objetivo fosse cumprido, foi necessário realizar diversas alterações ao código base da anterior fase de trabalho. Alterações essas que vão desde a reformulação da estratégia de parsing utilizada até à adição de uma nova primitiva geométrica "*Torus*".

2. Reestruturação do projeto

2.1 Parser XML

Tendo em conta os requisitos impostos nesta nova fase de desenvolvimento do projeto, foi possível constatar de imediato um número considerável de deficiências presente na estratégia utilizada no parsing de ficheiros XML na fase 1.

Um dos exemplos mais relevantes de uma dessas deficiências é o facto de que a estratégia de parsing usada era apenas capaz de reconhecer ficheiros com uma estruturação muito rígida dos seus elementos. Isto devia-se ao facto de que a ordem pela qual os elementos iam sendo avaliados pelo parser estava *hardcoded* no código fonte do projeto, tornando assim o processo de validação dos ficheiros de configuração muito pouco flexível.



```
<world>
  <window width="512" height="512" />
  <camera>
    <position x="3" y="2" z="1" />
    <lookAt x="0" y="0" z="0" />
    <up x="0" y="1" z="0" />
    <projection fov="60" near="1" far="1000" />
  </camera>
  <group>
    <models>
      <model file="torus.3d" />
    </models>
  </group>
</world>
```

```
<world>
  <camera>
    <lookAt x="0" y="0" z="0" />
    <position x="3" y="2" z="1" />
    <up x="0" y="1" z="0" />
    <projection fov="60" near="1" far="1000" />
  </camera>
  <window width="512" height="512" />
  <group>
    <models>
      <model file="torus.3d" />
    </models>
  </group>
</world>
```

Figura 2.1: Ficheiro XML válido vs. Ficheiro XML inválido

Como é fácil de imaginar, esta abordagem não era adequada aos novos problemas impostos, tais como a relevância da ordem das transformações a realizar.

Com isto em mente, foi então criada uma nova classe de objetos capaz de armazenar a informação referente a um determinado grupo da cena atual, juntamente com duas novas estruturas com os dados relativos à camera e à janela.

Fazendo recurso a estas novas estruturas e objetos, a nova e reformulada estratégia de parsing de ficheiros XML passa por, num primeiro momento, verificar se o respetivo ficheiro cumpre ou não os requisitos mínimos para se proceder ao processo de análise. Isto é, se o ficheiro em questão tem como primeiro elemento *world*, visto que sem ele não faria sentido realizar mais nenhuma ação de análise.

Posteriormente, verificamos de forma cíclica o próximo *child element* disponível. Caso se trate da *camera* ou *window* realizamos as respetivas funções de parsing de elementos e/ou atributos do mesmo modo cíclico referido anteriormente. No entanto, caso estejamos perante um elemento do tipo *group* procedemos à criação de um novo objeto, cujo construtor será então responsável pelo parsing de todos os componentes válidos do mesmo, tal como transformações, modelos e respetivos sub-grupos, os quais são gerados de forma recursiva.

2.2 Reorganização dos módulos

Devido a todas as alterações necessárias para abranger a nova e melhorada estratégia de parsing, foi então decidido proceder a uma reorganização dos módulos desenvolvidos até ao momento, de modo a facilitar não só a procura e leitura dos mesmos, como também a sua modularidade e escalabilidade para futuras fases de desenvolvimento.

Para isso foram então criadas duas novas diretorias, uma para cada uma das funcionalidades principais do programa (*generator* e *engine*), assim como uma outra para ficheiros comuns a ambas.

Foram também criados dois novos módulos, juntamente com dois outros responsáveis pela implementação das classes referentes às transformações geométricas e grupos da cena a serem desenhados, que processam todos os níveis de parsing e renderização, removendo, então, estas responsabilidades do anterior módulo *engine*.

Finalmente, procedeu-se também à documentação dos módulos, de modo a garantir um maior entendimento e coesão de ideias durante a atual e futuras edições do conteúdo do projeto a ser desenvolvido.

3. Nova Primitiva Gráfica

3.1 Torus

De forma a ser possível gerar um torus, são necessários 4 parâmetros: float rad1, float rad2, int slice, int stack

- **Raio interno** - O raio do centro até ao centro do torus (R na figura 4.2);
- **Raio secção** - O raio de uma secção do torus (r na figura 4.2);
- **Slices** - Como o torus é dividido verticalmente, em fatias;
- **Stacks** - Como o torus é dividido horizontalmente, em pilhas.

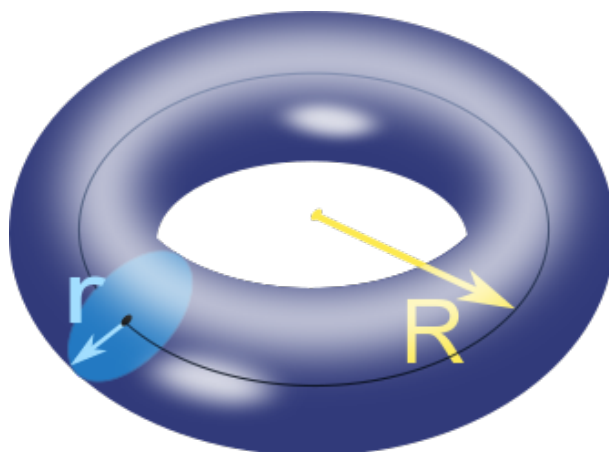


Figura 3.1: Representação dos raios de um torus

Apos termos toda a informação necessária temos um ciclo que percorre as slices e um ciclo aninhado que percorre as stacks gerando assim o torus em "fatias" verticais desenhando em cada iteração um novo "quadrado" da superfície da figura.

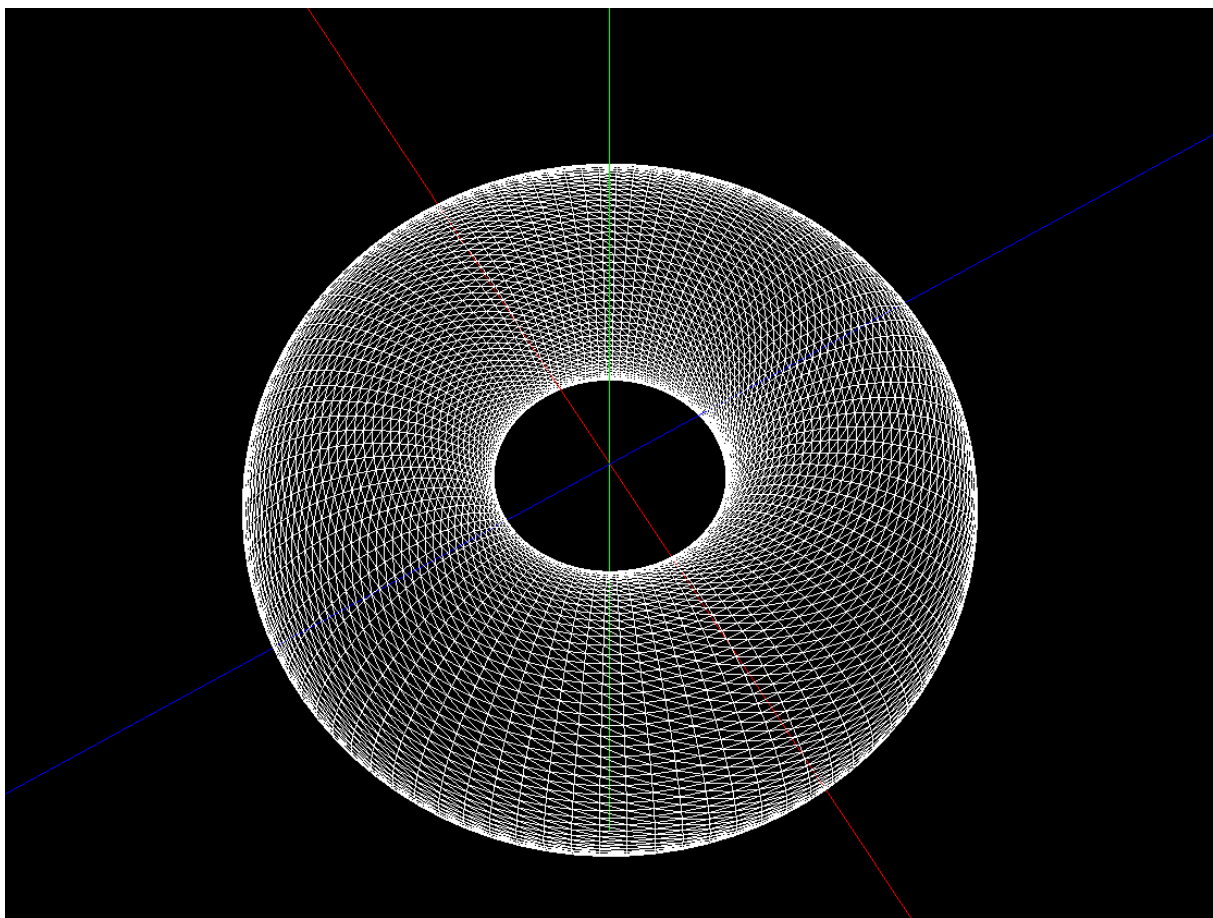


Figura 3.2: Torus com raio interno 2, raio secção 1, 100 slices e 100 stacks

4. Sistema Solar

De forma a obter um modelo do Sistema Solar que seja o mais próximo da realidade, as transformações têm em conta a escala real. No entanto, algumas das distâncias e escalas foram alteradas para termos uma visão mais agradável de todo o sistema solar.

No desenho dos planetas todos têm por base a mesma esfera (ou torus quando usado) que depois sofrem as transformações necessárias para obtermos a posição e escala desejada.

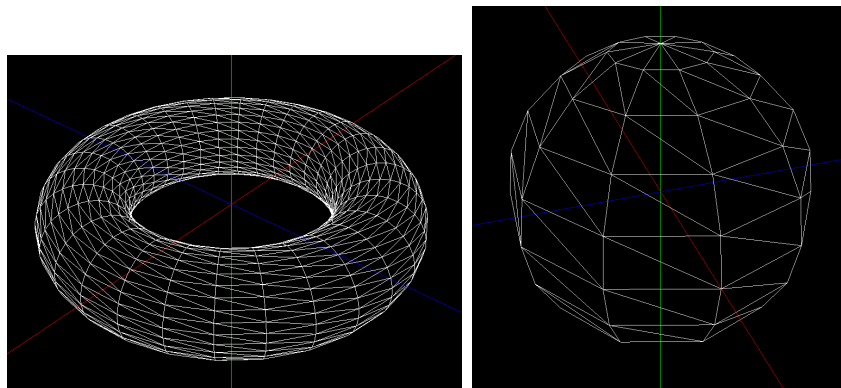


Figura 4.1: Torus e esfera originais (sem transformações)

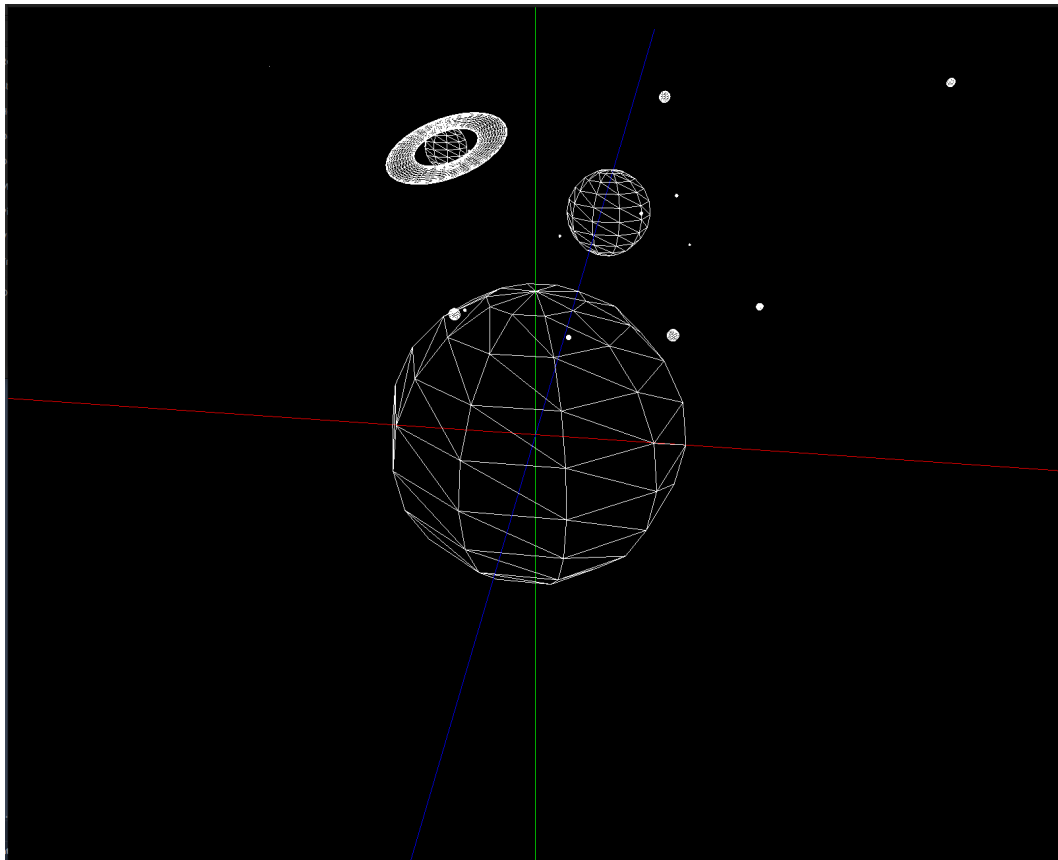


Figura 4.2: Sistema solar estático

5. Câmara

Para facilitar a visualização das figuras, foi implementada a possibilidade de mover tanto a câmara como as figuras, tanto através do uso do teclado ou rato. É também importante notar, que esta funcionalidade se destina maioritariamente ao auxílio em momentos de debug e exploração de cenas criadas.

As teclas associadas a cada movimento são:

- **W,S,A,D,J,K** : Alterar o ponto para o qual a câmara "olha" individualmente em cada eixo.
- **+, -** : Zoom in e zoom out.
- **F,L,P** : Modo *Fill*, *Line* e *Point*.
- **B,N,M** : `GL_BACK`, `GL_FRONT` e `GL_BACK_AND_FRONT`.
- **Botão esquerdo do rato + arrastar** : Rotação da câmara em relação a cada eixo.
- **Botão direito do rato + arrastar** : Zoom in e zoom out.

6. Dificuldades e aspetos a melhorar

Ao longo do desenvolvimento desta fase do projeto, foram encontrados alguns obstáculos que levaram à reorganização e reimplementação de várias secções do código. Um desses obstáculos passou pela implementação da organização da estrutura das diversas transformações.

Desde cedo, foi decidido que iria ser criado uma hierarquia de transformações, tendo como raiz uma classe abstrata *Transformation*, de modo a tornar mais fácil a expansão do módulo a novos tipos de transformações geométricas no futuro. No entanto, o comportamento de classes abstratas em C++ é ligeiramente diferente ao mesmo comportamento em Java, pelo que foi necessário recorrer à leitura de alguma documentação para auxiliar e agilizar o processo de geração de código fonte.

Apesar destes esforços, continuamos a encontrar algumas inconveniências ao longo de todo o processo, já que, à semelhança do C, a linguagem C++ não possui nenhum tipo de *garbage collection*. Isto por si só não é um problema, apenas um inconveniente (tal como havia sido referido), já que tal implica que sejam implementados os mecanismos de libertação da memória alocada, o que muitas vezes, pode passar um pouco despercebido a meio de uma reestruturação tão incisiva do projeto.

Consequentemente, procuramos descobrir técnicas / ferramentas capazes de realizar estas tarefas automaticamente, até que nos deparamos com o sistema de apontadores inteligentes presentes no namespace `std` da linguagem C++.

Tal como o seu nome indica, este sistema de apontadores é capaz de realizar alocação e libertação de memória de forma automática através do uso de `std::unique_ptr` e `std::make_unique`, retirando-nos assim a responsabilidade de arquitetar uma solução eficiente para a gestão de memória.

Por outro lado, um dos aspetos que pode e, com certeza, será melhorado no futuro é o teste e especificação de casos de erro no parsing dos ficheiros XML, já que por várias vezes a falta de documentação de erros levou à perda de tempo precioso no debugging de problemas que poderiam ter sido trivializados caso os sistemas adequados estivessem implementados.

7. Conclusão

Nesta fase do trabalho prático, foi elaborado um modelo inicial do sistema solar. Além disso, foi criado um polígono adicional, o Torus, para representar os anéis de alguns planetas. O restante código foi também reorganizado e melhorado.

Com o desenvolvimento desta fase, foi possível consolidar conhecimentos relacionados com a manipulação de rotações, translações e escalas para tornar o modelo do sistema solar mais próximo da realidade.

Além dos conhecimentos mencionados, esta fase do trabalho permitiu expandir ainda mais o conhecimento na linguagem de programação C++ e a manipulação de arquivos XML.