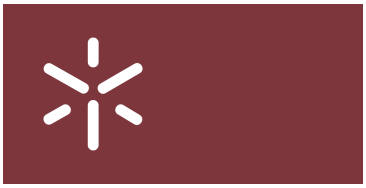


Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

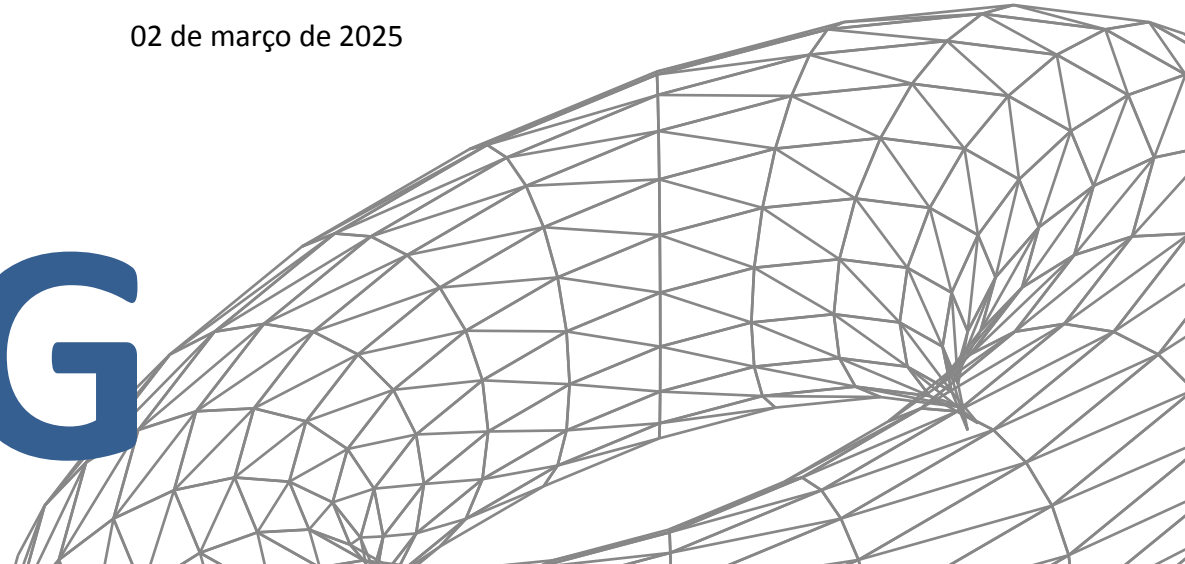
Ano Letivo de 2024/2025

Trabalho Prático - Fase 1

Eduardo Faria	**Hélder Gomes**	**Nuno Silva**	**Pedro Pereira**
a104353	a104100	a104089	a104082

02 de março de 2025

CG



Data da Receção	
Responsável	
Avaliação	
Observações	

Trabalho Prático - Fase 1

Eduardo Faria	**Hélder Gomes**	**Nuno Silva**	**Pedro Pereira**
a104353	a104100	a104089	a104082

02 de março de 2025

Resumo

Nesta primeira fase do projeto, foi desenvolvido um gerador de modelos (*generator*) e um motor gráfico (*engine*) capaz de interpretar e renderizar primitivas gráficas tridimensionais. O *generator* permite criar ficheiros contendo a informação dos vértices e associações. O *engine*, por sua vez, lê um ficheiro de configuração em XML, onde estão especificadas definições de cena e a lista de modelos a carregar, renderiza os objetos especificados e fornece uma interface gráfica para a sua visualização.

Área de Aplicação: Computação Gráfica, Modelação Tridimensional e Motores de Renderização

Palavras-Chave:Computação Gráfica, OpenGL, Primitivas Gráficas, Modelação 3D, XML

Índice

1. Arquitetura Geral

A arquitetura do projeto é dividida em dois módulos principais: o *generator* e o *engine*. Ambos são projetados para oferecer uma solução modular e escalável para a criação e visualização de modelos tridimensionais, com ênfase na separação de responsabilidades e otimização de recursos computacionais.

Generator

O módulo *generator* é responsável pela geração de ficheiros contendo os dados dos vértices das primitivas gráficas (como planos, esferas, cones, etc.) disponibilizadas ao utilizador. Estes ficheiros seguem um formato padronizado, otimizado para reduzir a sobrecarga computacional no lado do *engine*. Os dados gerados são armazenados em ficheiros (ex.: `plane.3d` e `sphere.3d`) e integrados no ficheiro de configuração da cena (`config.xml`), que define a disposição espacial e as propriedades visuais dos modelos. Este processo permite ao *engine* focar-se exclusivamente na renderização, melhorando a performance e a modularidade do sistema.

Engine

O módulo *engine* é responsável pela interpretação dos ficheiros de configuração (`config.xml`) e pela renderização dos modelos tridimensionais. Utiliza os dados gerados pelo *generator* para construir a cena e aplicar técnicas de visualização. A arquitetura do *engine* é baseada em componentes reutilizáveis (ex.: `draw.hpp` e `xml_parser.hpp`), facilitando a manutenção e a extensão do sistema.

2. Generator

O *generator* permite ao utilizador automatizar a criação das primitivas plano, cubo, esfera, cone, cilindro e *toro*. Para tal, pode fazer uso do comando

```
./generator <shape_name> <param1> ... <paramN> <output_file>
```

para gerar as diferentes estruturas, assegurando que introduz os parâmetros esperados:

```
./generator plane <dimension> <divisions> <output_file>
./generator box <dimension> <divisions> <output_file>
./generator sphere <radius> <slices> <stacks> <output_file>
./generator cone <radius> <height> <slices> <stacks> <output_file>
./generator cylinder <radius> <height> <slices> <stacks> <output_file>
./generator torus <radius> <minor_radius <slices> <stacks> <output_file>
```

2.1. Estrutura dos ficheiros

O formato pardronizado dos ficheiros `.3d` permite ao nosso *engine* rapidamente ler e interpretar os vértices e faces associadas. A estrutura é definida por um número n inicial, representativo do número de pontos da primitiva, seguido de n pontos com coordenadas x, y e z, e um número k de associações entre pontos, seguido de k associações entre os índices de 3 pontos.

```
16
1 0 -1
1 0 -0.3333
1 0 0.3333
1 0 1
0.3333 0 -1
...
18
1 2 5
2 3 6
3 4 7
...
```

Figura 1: Exemplo da estrutura de um ficheiro `.3d`.

2.2. Primitivas

2.2.1. Plano

A geração de planos tridimensionais consiste em subdividir um plano quadrado, definido pelos limites $-\text{maxwidth} \leq x \leq \text{maxwidth}$ e $-\text{maxwidth} \leq z \leq \text{maxwidth}$ ($\text{maxwidth} = \frac{\text{dimension}}{2}$) em várias secções, de acordo com o número de divisões especificado.

Algoritmo de geração de Planos

O processo de criação de planos é realizado pela função `createPlane()`, que segue os seguintes passos:

1. Definição dos limites do plano:
   - O plano está centrado na origem, estendendo-se desde $-\text{maxwidth}$ até maxwidth ao longo dos eixos x e z
2. Cálculo das coordenadas dos pontos:
   - O número total de pontos ao longo de cada aresta é igual a `divPerEdge + 1` **(?? talvez dizer melhor o que é divPerEdge)**
   - As coordenadas (x, y e z) dos pontos são calculadas iterando sobre as divisões:
     - $x = \text{maxwidth} - \frac{\text{dimension} \times i}{\text{divPerEdge}} \forall i \in \{0, 1, ..., \text{divPerEdge}\}$
     - $y = 0$, assumindo um plano apoiado no plano xz
     - $z = \text{maxwidth} - \frac{\text{dimension} \times j}{\text{divPerEdge}} \forall j \in \{0, 1, ..., \text{divPerEdge}$
3. Construção dos triângulos:
   - Para cada célula definida pelos pontos vizinhos, são formados dois triângulos:
     - Um triângulo "norte" (orientado positivamente ao longo de z)
     - Um triângulo "sul" (orientado negativamente ao longo de z)
   - As associações entre pontos são armazenadas através das posições relativas na estrutura

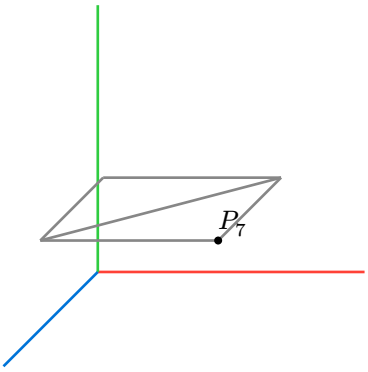


Figura 2: Representação de um conjunto de associações, na iteração 7, na execução do comando `generator plane 2 3 plane.3d`[1].

4. Escrita para ficheiro.

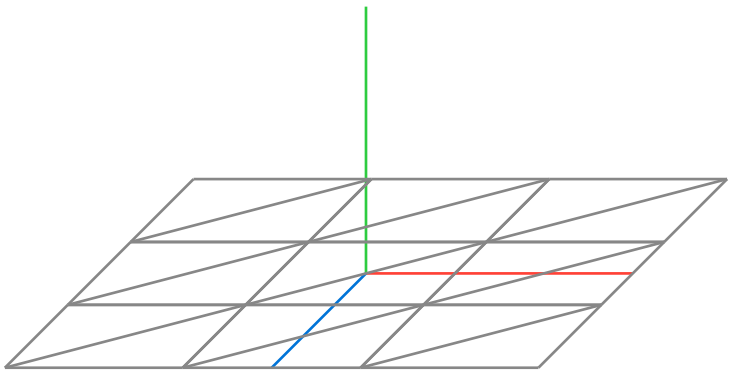


Figura 3: Representação da primitiva com a execução do comando `generator plane 2 3 plane.3d`[1].

[1]Os eixos seguem a seguinte coloração: (x, y, z)

2.2.2. Cubo

O processo de criação da primitiva cubo envolve a subdivisão de cada uma das seis faces em múltiplas secções triangulares e a sua posição relacional correta.

Cada face é gerada ao longo dos planos $x = \pm\frac{d}{2}$, $y = \pm\frac{d}{2}$ e $z = \pm\frac{d}{2}$. Para cada ponto (i, j) da estrutura de subdivisão n, as coordenadas (x, y, z) são determinadas por:

$$\begin{cases} x = \pm\frac{d}{2} \text{ , se a face for perpendicular ao eixo } x \\ x = \pm\frac{d}{2} - \frac{2di}{n} \text{ , caso contrário} \\ y = \pm\frac{d}{2} \text{ , se a face for perpendicular ao eixo } y \\ x = \pm\frac{d}{2} - \frac{2(i\cdot|u|+j\cdot|w|)}{n} \text{ , caso contrário} \\ z = \pm\frac{d}{2} \text{ , se a face for perpendicular ao eixo } z \\ z = \pm\frac{d}{2} - \frac{2dj}{n} \text{ , caso contrário} \end{cases}$$

onde:

- d é a dimensão total do cubo
- n é o número de divisões por aresta
- u, v, w representam as normais unitárias das faces, que podem assumir valores em $\{-1, 0, 1\}$

Algoritmo de de Geração de Cubos

A função `createBox()` executa os seguintes passos para gerar os pontos de um cubo:

1. Configuração das faces:
   - Itera sobre as 6 faces possíveis do cubo
   - Determina as normais u, v, w para cada face
2. Geração de pontos:
   - Para cada ponto da estrutura de subdivisão (i, j), calcula as coordenadas (x, y, z) de acordo com as equações acima
3. Formação dos triângulos:
   - Para cada artefacto da estrutura, gera dois triângulos (ver 2.2.1.3), ajustando a ordem dos vértices para garantir a orientação correta da face
4. Escrita para ficheiro.

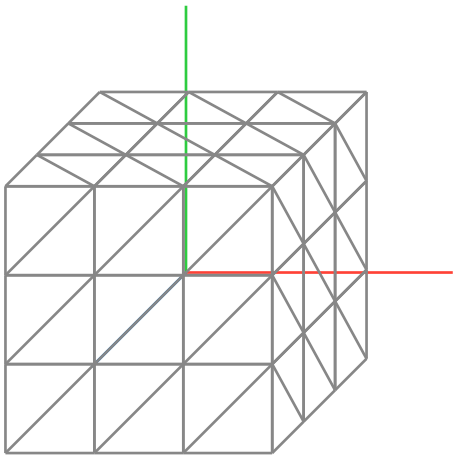


Figura 4: Representação da primitiva com a execução do comando `generator box 2 3 box.3d`.

2.2.3. Esfera

A criação de esferas tridimensionais é baseada na discretização das suas superfícies usando coordenadas esféricas. A esfera é definida pelo raio r, número de fatias s (*slices*) e número de camadas t (*stacks*), fundamentando-se nas seguintes formulas matemáticas:

$$\begin{cases} x = r \cdot \cos(\alpha) \cdot \cos(\beta) \\ y = r \cdot \sin(\beta) \\ z = r \cdot \sin(\alpha) \cdot \cos(\beta) \end{cases}$$

onde:

- r é o raio da esfera
- $\alpha = \frac{2\pi i}{s}$ é o ângulo azimutal, $0 \leq \alpha \leq 2\pi$
- $\beta = -\frac{\pi}{2} + \frac{\pi j}{t}$ é o ângulo polar, $-\frac{\pi}{2} \leq \beta \leq \frac{\pi}{2}$

Algoritmo de Geração de Esferas

A função `createSphere()` segue os seguintes passos:

1. Definição dos ângulos:
   - Calcula o incremento de cada fatia e camada:
     - $\alpha = \frac{2\pi}{s}$
     - $\beta = \frac{\pi}{t}$
2. Geração de pontos:
   - Itera por cada combinação de i e j, usando as equações paramétricas para determinar (x, y, z)
3. Formação dos triângulos:
   - Associa os pontos para formar triângulos, lidando separadamente com os polos (quando $j = 1$ ou $j = t - 1$)

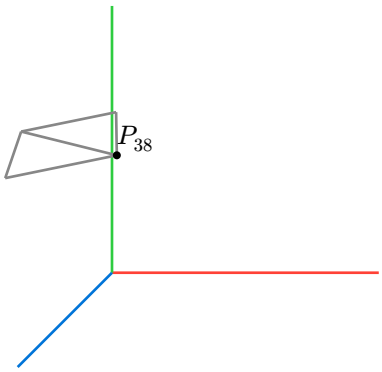


Figura 5: Representação de um conjunto de associações, na iteração 38, na execução do comando `generator sphere 1 10 10 sphere.3d`.

4. Escrita para ficheiro.

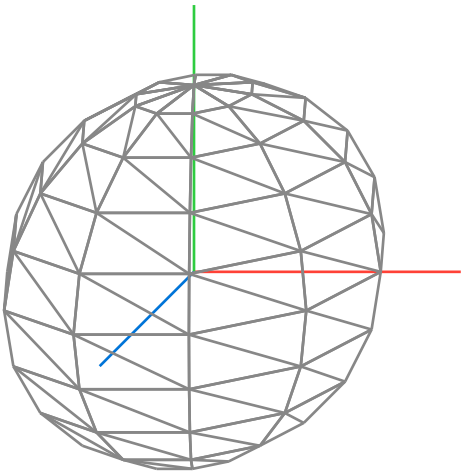


Figura 6: Representação da primitiva com a execução do comando `generator sphere 1 10 10 sphere.3d`.

2.2.4. Cone

A criação de cones tridimensionais é feito através de um processo iterativo pelos seus anéis. Este processo inicia-se no anel respetivo à base do cone e termina no anel respetivo ao vértice do topo do mesmo, sendo este simplesmente um anel/circunferência mas com raio nulo.

Por cada um dos anéis do cone, que representam os limites (horizontais) de cada *stack* do mesmo, itera-se pelos vários limites (verticais) que surgem da subdivisão do cone em *slices*. Desta forma, cria-se, em cada iteração, um ponto (x, y, z) que resulta da interseção entre um limite vertical e um limite horizontal respetivos à iteração atual, tendo por base as seguintes expressões matemáticas:

$$\begin{cases} x = r_a \cdot \cos(\alpha) \\ y = h_a \\ z = r_a \cdot \sin(\alpha) \end{cases}$$

onde:

- r_a é o raio do anel/circunferência que será percorrido na iteração atual
- h_a é a altura atual a que se encontra o anel a ser percorrido
- $\alpha = \frac{2\pi j}{\text{slices}}$ é o ângulo entre o eixo z o limite da *slice* respetiva à iteração atual pelo anel, $0 \leq \alpha \leq 2\pi$ e $0 \leq j < \text{slices}$

Algoritmo de Geração de Cones

A função `createCone()` segue os seguintes passos:

1. Definição do ângulo de uma *slice* e da distância entre *stacks*:
   - O ângulo de uma *slice* (ângulo entre os limites verticais de uma *slice*):
     - $\Delta\alpha = \frac{2\pi}{\text{slices}}$
   - O distância entres os limites horizontais de uma *stack*:
     - $\Delta h = \frac{\text{altura}}{\text{stacks}}$
2. Geração de pontos:
   - Itera-se por cada combinação de i e j para determinar as coordenadas (x, y e z) de um ponto
   - Por cada i calcula-se $h_a = i \cdot \Delta h$ e calcula-se $r_a = r \cdot \left(1 - \frac{i}{\text{stacks}}\right)$
   - Por cada j calcula-se $\alpha = j \cdot \Delta\alpha$ e determinam as coordenadas x, y e z de um ponto com base nas expressões matemáticas definidas acima
   - O ponto $(0,0,0)$ é criado separadamente, após todas as iterações serem finalizadas

3. Formação dos triângulos:
    - Associam-se os pontos para formar triângulos, lidando separadamente com os triângulos da base do cone

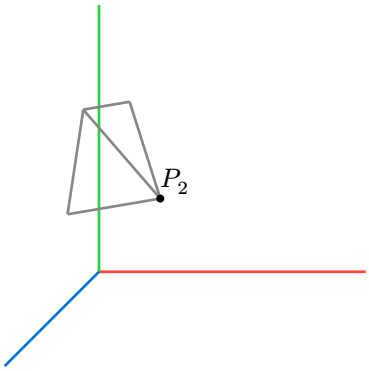


Figura 7: Representação de um conjunto de associações, na iteração 2, na execução do comando `generator cone 1 2 6 3 cone.3d`.

4. Escrita para ficheiro.

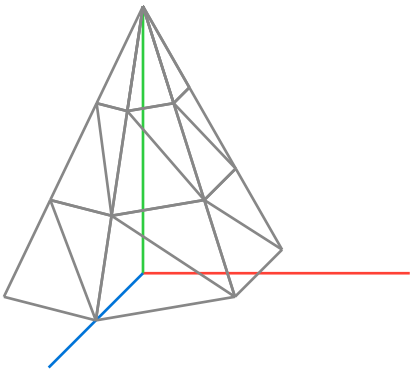


Figura 8: Representação da primitiva com a execução do comando `generator cone 1 2 6 3 cone.3d`.

2.2.5. Cilindro

A geração de cilindros tridimensionais é realizada através de um processo iterativo pelos seus anéis. Este processo inicia-se pelo anel corresponde à base situada no plano xz e termina no anel corresponde à base com maior cota.

$$\begin{cases} x = r \cdot \cos(\alpha) \\ y = h_a \\ z = r \cdot \sin(\alpha) \end{cases}$$

onde:

- r é o raio do anel/circunferência
- h_a é a altura atual a que se encontra o anel a ser percorrido
- $\alpha = \frac{2\pi j}{\text{slices}}$ é o ângulo entre o eixo z o limite da *slice* respetiva à iteração atual pelo anel, $0 \leq \alpha \leq 2\pi$ e $0 \leq j < \text{slices}$

Algoritmo de Geração de Cilindros

A função `createCylinder()` segue os seguintes passos:

1. Definição do ângulo de uma *slice* e da distância entre *stacks*:

- O ângulo de uma *slice*:
  - $\Delta\alpha = \frac{2\pi}{\text{slices}}$
- O distância entre uma *stack*:
  - $\Delta h = \frac{\text{altura}}{\text{stacks}}$

2. Geração de pontos:
   - Itera-se por cada combinação de i e j para determinar todos os pontos do cilindro
   - Por cada i calcula-se $h_a = i \cdot \Delta h$
   - Por cada j calcula-se $\alpha = j \cdot \Delta\alpha$ e determinam as coordenadas x, y e z de um ponto com base nas expressões matemáticas definidas acima
   - O ponto $(0,0,0)$ e $(0,h,0)$ são criados separadamente, após todas as iterações serem finalizadas
3. Formação dos triângulos:
   - Associam-se os pontos para formar triângulos, lidando separadamente com os triângulos do topo e da base do cone

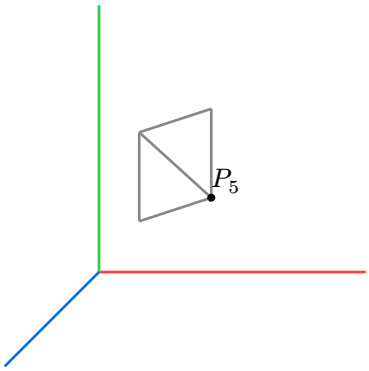


Figura 9: Representação de um conjunto de associações, na iteração 5, na execução do comando `generator cylinder 1 10 10 cylinder.3d`.

4. Escrita para ficheiro.

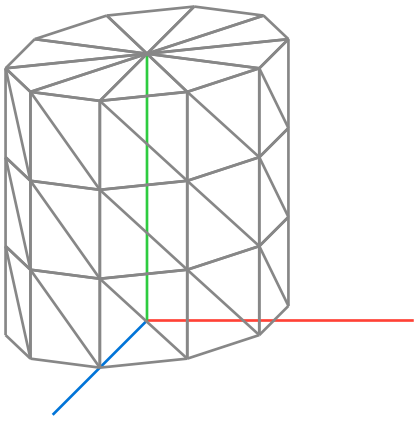


Figura 10: Representação de um conjunto de associações, numa dada iteração, na execução do comando `generator cylinder 1 2 10 3 cylinder.3d`.

2.2.6. Toro

A geração de toros tridimensionais é realizada através da parametrização matemática da superfície de um toro, definida pelas seguintes equações paramétricas:

$$\begin{cases} x = (R + r\cos(\beta))\cos(\alpha) \\ y = r\sin(\beta) \\ z = (R + r\cos(\beta))\sin(\alpha) \end{cases}$$

onde:

- R é o raio maior (distância do centro do toro ao centro da estrutura)
- r é o raio menor (raio da estrutura)
- α é o ângulo que percorre as divisões ao longo da circunferência principal (*slices*)
- β é o ângulo que percorre as divisões ao longo da circunferência do tubo (*stacks*)

Algoritmo de Geração de Toros

A função `createTorus()` implementa o seguinte processo:

1. Definição dos ângulos incrementais:
   - O ângulo por *slice* (divisão ao longo da circunferência maior) é dado por:
     - $\Delta\alpha = \frac{2\pi}{\text{slices}}$
   - O ângulo por *stack* (divisão ao longo da circunferência da estrutura) é dado por:
     - $\Delta\beta = \frac{2\pi}{\text{stacks}}$
2. Cálculo das coordenadas dos pontos:
   - Itera-se sobre os *slices* e *stacks*, calculando as coordenadas (x, y, z) para cada ponto do sólido
3. Construção dos triângulos:
   - Para cada artefacto gerado pelos pontos adjacentes, são definidos dois triângulos

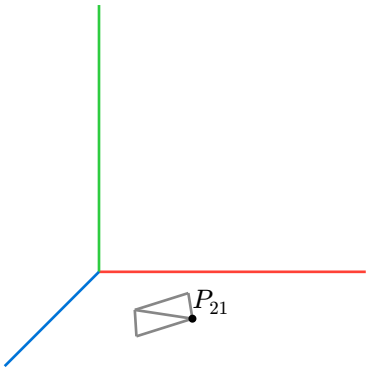


Figura 11: Representação de um conjunto de associações, na iteração 21, na execução do comando `generator torus 1 0.3 20 10 torus.3d`.

4. Escrita para ficheiro.

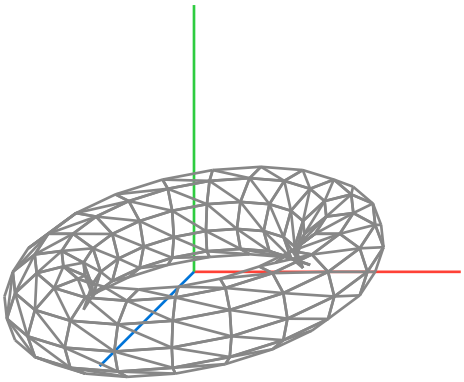


Figura 12: Representação da primitiva com a execução do comando `generator torus 1 0.3 20 10 torus.3d`.

3. Engine

3.1. Ficheiros de cena .xml

O ficheiros `.xml` desempenham um papel fundamental no projeto, atuando como configuração centralizada para a cena tridimensional. Estes ficheiros fornecem uma forma estruturada e legível de definir todos os elementos necessários para a renderização, promovendo independência entre o conteúdo visual e o código.

Estrutura do ficheiro `.xml`

A estrutura de um ficheiro `.xml`, no contexto do nosso projeto, é hierárquica e compreensível, permitindo uma organização lógica dos componentes da cena. O elemento raiz é `<world>`, que engloba todas as configurações e objetos da cena. Dentro destes, destacam-se os seguintes blocos:

- `<window>`: Especifica as dimensões da janela gráfica. Os atributos *width* e *height* determinam as dimensões em *px*
- `<camera>`: Contém os parâmetros de posicionamento e projeção da câmara. O elemento `<position>` define a localização da câmara no espaço 3D. A tag `<lookAt>` indica o ponto focal, enquanto `<up>` estabelece o vetor vertical. A secção `<projection>` configura parâmetros como o campo de visão (*fov*) e as distâncias de visualização *near* e *far*
- `<group>`: Agrupa os modelos a serem renderizados. Dentro deste grupo, encontram-se elementos `<model>` que fazem referência aos ficheiros *.3d* ou *.obj* a serem interpretados pelo *engine*

Exemplo de `config.xml`

```
<world>
    <window width="1000" height="600" />
    <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="../../objects/sphere.3d" />
            <model file="../../objects/ferrari.obj" />
        </models>
    </group>
</world>
```

Figura 13: Exemplo da estrutura de um ficheiro `.3d`.

Leitura e processamento

O ficheiro `.xml` introduzido na inicialização do *engine* é assimilado através da classe *XMLParser*. Utilizando a biblioteca *TinyXML2*, os elementos e atributos são interpretados e mapeados para estruturas internas (*WorldConfig*, *CameraConfig*, *ModelConfig*). Esta abordagem oferece flexibilidade, permitindo a fácil adição de novos parâmetros ou modelos sem alterar o código. Finalmente, a configuração carregada

é disponibilizada globalmente para configuração dos parâmetros das várias funções como definição do tamanho da janela, disposição da posição da câmara, entre outras.

3.2. Renderização

3.2.1. Controlos

O nosso *engine* fornece ao utilizador uma forma intuitiva de movimentar a câmara. Utilizando a convenção *click and drag*, o utilizador pode controlar a posição da câmara clicando e arrastando o cursor. Este movimento induz uma alteração da posição da câmara em torno do objeto em foco.

O utilizador dispõe ainda da funcionalidade de aproximar ou afastar a câmara fazendo *scroll* para cima ou para baixo. O GLUT não fornece suporte para rastrear o *scroll* do rato. Para tal, fez uso da função `glutMouseWheelFunc()` da alternativa ao GLUT — *freeGlut*.

3.2.2. VBOs

A implementação de *Vertex Buffer Objects* (VBOs) teve um impacto direto no desempenho da aplicação. No contexto da primeira fase, decidiu-se seguir a metodologia de **VBOs sem índice**, com múltiplos *buffers*. Desta forma, as estruturas de dados que, antes da implementação, alojavam um conjunto de estruturas do tipo `struct Point {float x, y, z;};`, passaram apenas a conter elementos do tipo `float` — 3 valores por cada coordenada de cada ponto. Segue um exemplo de um *buffer* preenchido com os pontos do ficheiro de exemplo.

1.0f	0.0f	-1.0f	1.0f	0.0f	-0.3333f	1.0f	0.0f	0.3333f	...

Figura 14: Exemplo dos elementos de um *buffer*.

O uso de VBOs destaca-se em situações onde existe um elevado número de triângulos. Para tal, realizaram-se testes de *performance*[2] para comparar o desempenho do programa com e sem VBOs implementados. Criaram-se e renderizaram-se esferas com números de triângulos variáveis e alternou-se entre o uso ou não de VBOs sem índice (ver 4.1.2).

Frames por segundo		
Número de triângulos	**Sem VBOs**	**VBOs sem índices**
16[c]	6805	7253
1012[d]	2701	6812
262812[e]	32	1437
1049800[f]	8	487

Tabela 1: Resultados do teste de desempenho com e sem VBOs.

[2] Testes realizados num *MacBook Air* M1 2020

[c] `./generator sphere 1 4 3 sphere.3d`

[d] `./generator sphere 1 23 23 sphere.3d`

[e] `./generator sphere 1 363 363 sphere.3d`

[f] `./generator sphere 1 725 725 sphere.3d`

4. Extras

4.1. Menu na Interface Gráfica

Para maximizar o aproveitamento das funcionalidades do nosso *engine*, desenvolveu-se uma interface gráfica utilizando a biblioteca *Dear ImGui*. Esta interface proporciona ao utilizador um controlo intuitivo sobre as configurações da cena, dos modelos, da câmara e da renderização.

O menu está estruturado em categorias expansíveis, permitindo uma navegação organizada entre diferentes opções, como ajustes da câmara, personalização da cena e gestão dos modelos. Além disso, exibe em tempo real informações de desempenho, incluindo a taxa de FPS e o número de triângulos renderizados.

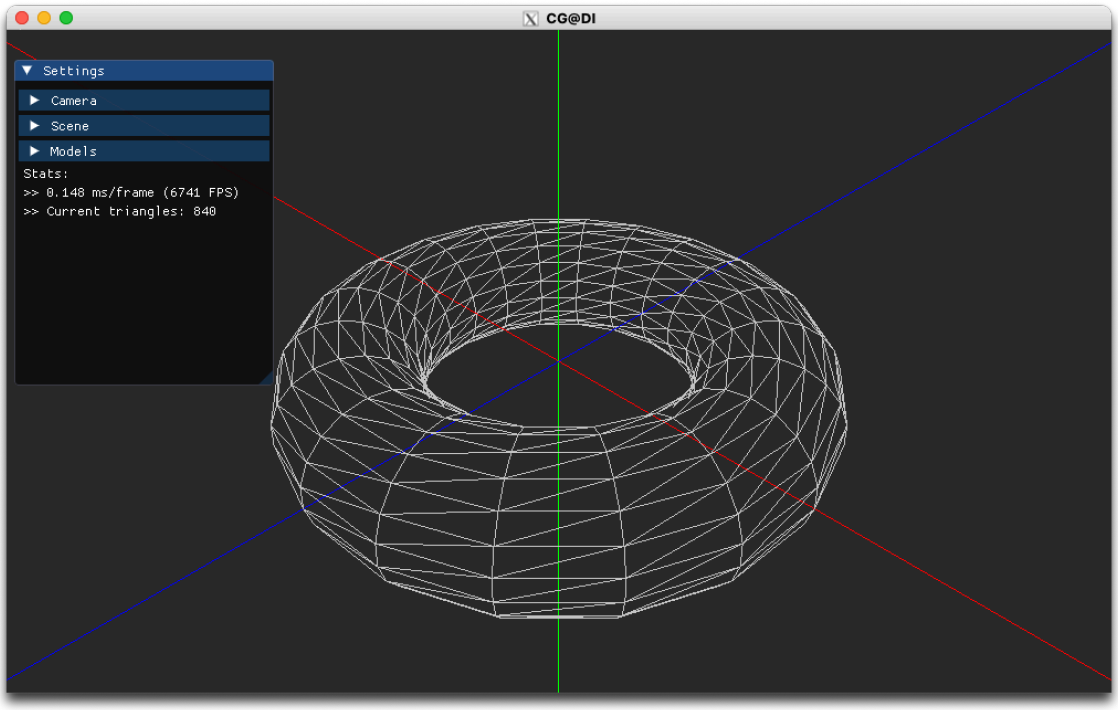


Figura 15: Visão geral do menu.

4.1.1. Definições da Câmara

Na aba das definições da câmara, o utilizador detém do controlo total da posição da câmara e para onde olha. Pode ainda alterar a sensibilidade das funcionalidades de *click and drag* e *srcoll*.

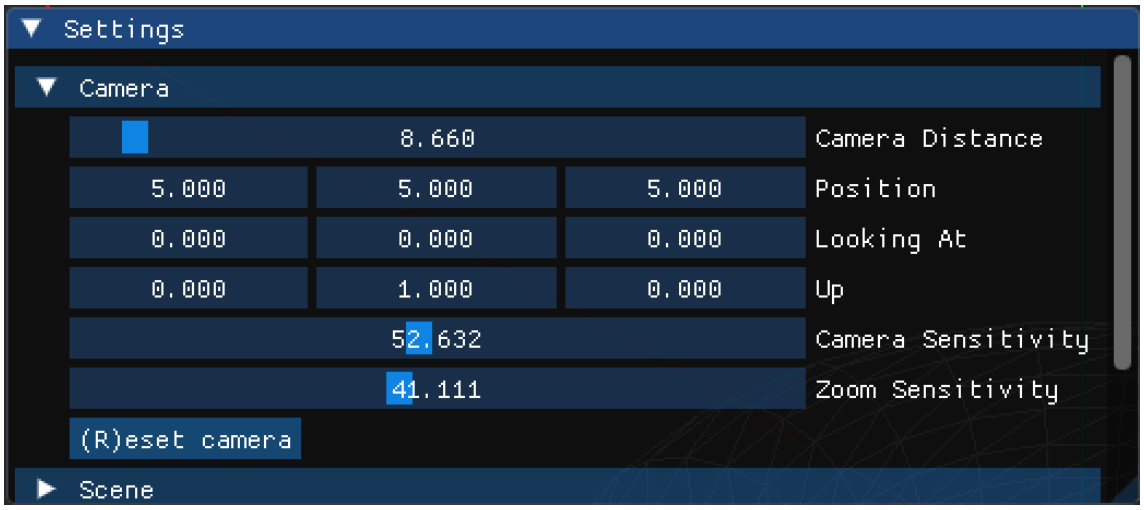


Figura 16: Definições da câmara.

4.1.2. Definições de Cena

Nesta secção, o utilizador consegue alterar parâmetros globais sobre a renderização no *engine*. É possível ativar ou desativar opções como *culling* das faces, modo de *wireframe*, desenho dos eixos e o uso de VBOs[7]. Além disso, permite ajustar a cor de fundo da cena, modificando os valores de vermelho (R), verde (G) e azul (B).

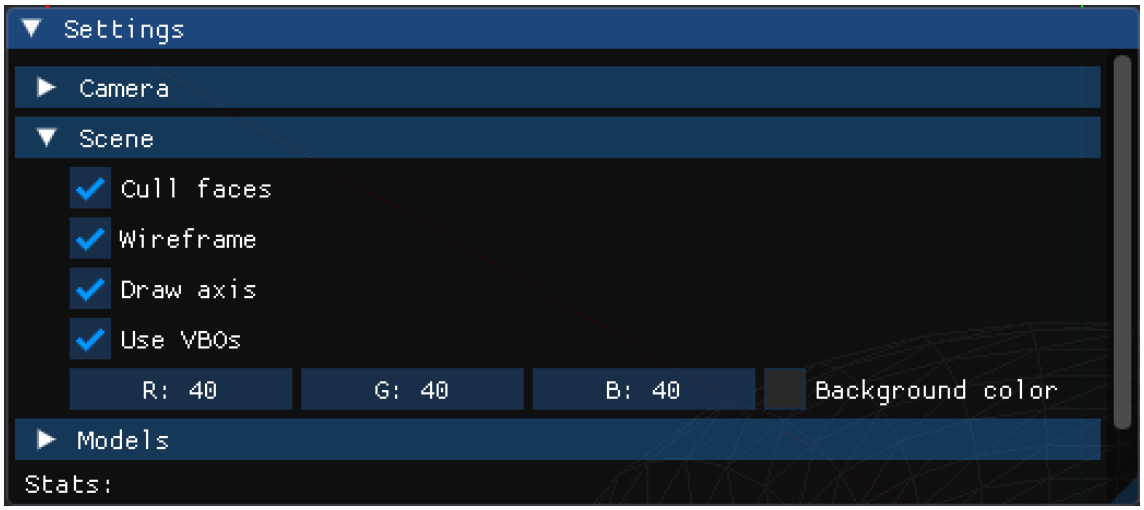


Figura 17: Definições da cena.

4.1.3. Definições dos Modelos

Para o contexto da primeira fase, o utilizador consegue ainda alterar a cor dos modelos em cena, modificando os valores de vermelho (R), verde (G) e azul (B).

[7]A existência de uma opção para ativar ou desativir o uso de VBOs implica ter duas estruturas de dados distintas para armazenar a mesma informação. Modelos com um elevado número de triângulos podem impactar a memória utilizada mais do que o esperado.

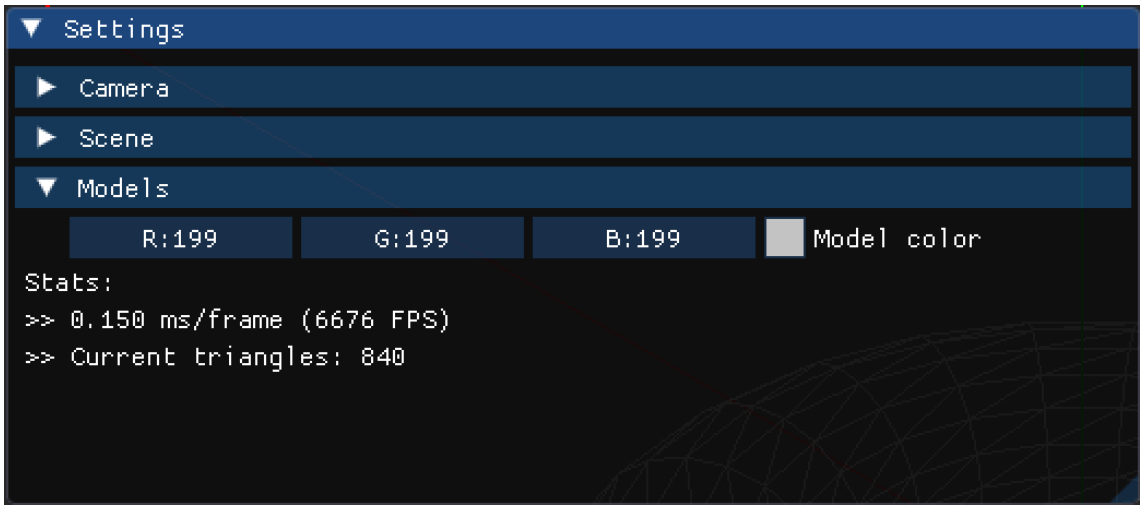


Figura 18: Definições dos modelos.

4.2. Suporte para ficheiros .obj

O *engine* desenvolvido suporta a leitura de ficheiros `.obj`, permitindo a integração de modelos 3D mais complexos e gerados por software industrial. Contudo, numa fase inicial, foram descartadas as coordenadas de textura e as normais dos vértices, centrando-nos apenas na geometria da estrutura.

Exemplo de `.obj`

```
# Vértices
v -1.0 -1.0 -1.0
v -1.0 -1.0 1.0
v 1.0 -1.0 1.0
v 1.0 -1.0 -1.0
v -1.0 1.0 -1.0
v -1.0 1.0 1.0
v 1.0 1.0 1.0
v 1.0 1.0 -1.0

# Faces
f 1/1/1 2/2/2 3/3/3
f 3/3/3 4/4/4 1/5/5
```

Figura 19: Exemplo da estrutura de um ficheiro `.obj`.

Leitura e Processamento

A função `parseFile()`, responsável pelo parsing dos ficheiros, reconhece automaticamente o formato através da extensão do ficheiro. Para ficheiros `.obj`:

1. Leitura de vértices: Linhas iniciadas por 'v' são interpretadas como pontos 3D (x, y, z).
2. Leitura de faces: Linhas iniciadas por 'f' especificam triângulos utilizando índices dos vértices. Utilizam-se apenas os índices de vértices (ignorando texturas e normais), embora estas estejam habitualmente presentes na linha.
3. Integração na estrutura: Os pontos obtidos do ficheiro são introduzidos na mesma estrutura de dados que os pontos dos ficheiros `.3d`, uma vez que esta é totalmente modular e apenas armazena os pontos a desenhar, independentemente da sua origem.

Como referido nesta secção, o grupo optou por descartar parte das informações características dos ficheiros `.obj`, no entanto, prevê que numa próxima fase venha a complementar este trabalho.

5. Conclusão

O grupo completou os pontos requeridos para a primeira fase do projeto. Com a boa execução do pretendido e a inclusão de pontos extra relevantes, estabelecemos uma base sólida para futuras fases do projeto.

Referências

CMake. (2025). CMake. https://cmake.org/

The Khronos Group. (2025). OpenGL. https://www.opengl.org/

John F. Fay, John Tsiombikas, and Diederick C. Niehorster. (2025). freeGlut. https://freeglut.sourceforge.net/

Lee Thomason. (2025). TinyXML2 [Source code]. GitHub. https://github.com/leethomason/tinyxml2

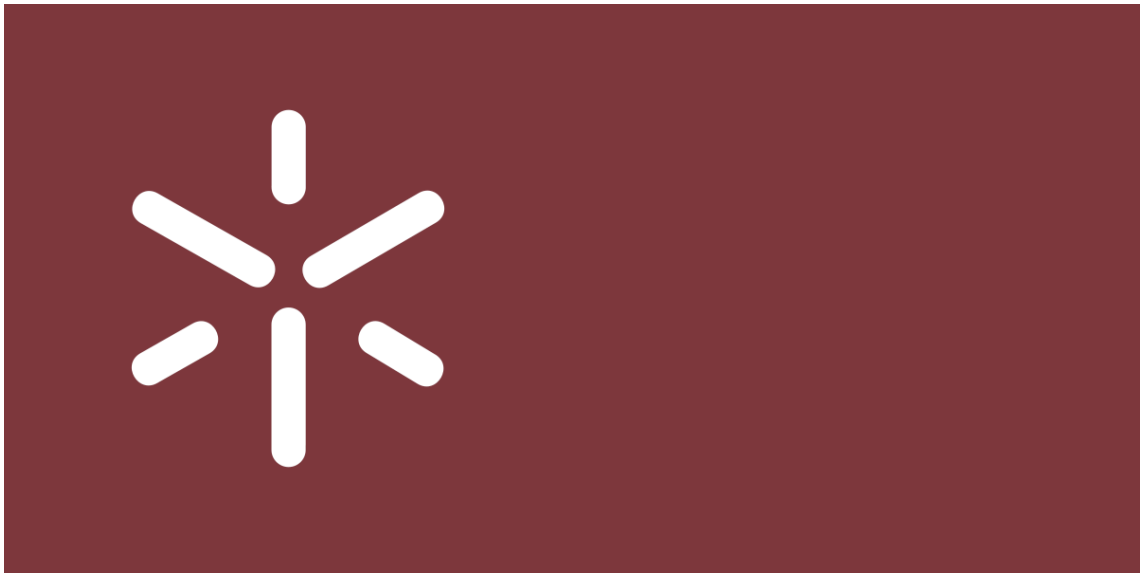
orconut. (2025). Dear imGui [Source code]. GitHub. https://github.com/ocornut/imgui

Lista de Siglas e Acrónimos

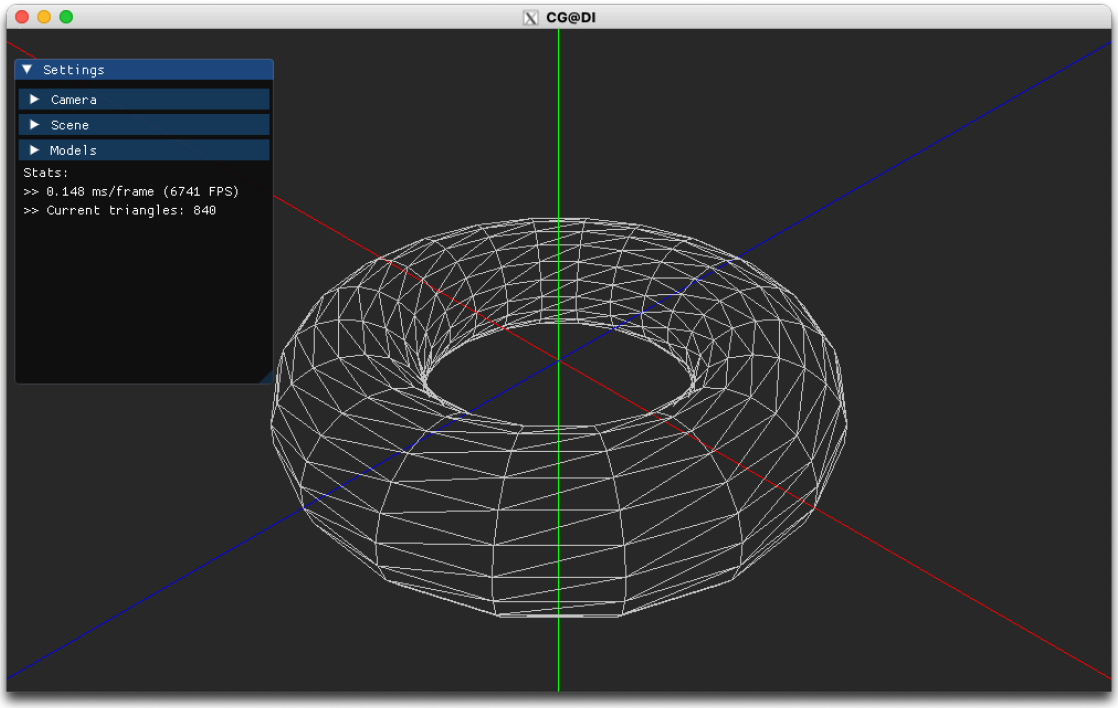
CG Computação Gráfica
XML *Extensible Markup Language*
GLUT *OpenGL Utility Toolkit*
VBO *Vertex Buffer Object*
FPS *Frames per Second*

Anexos

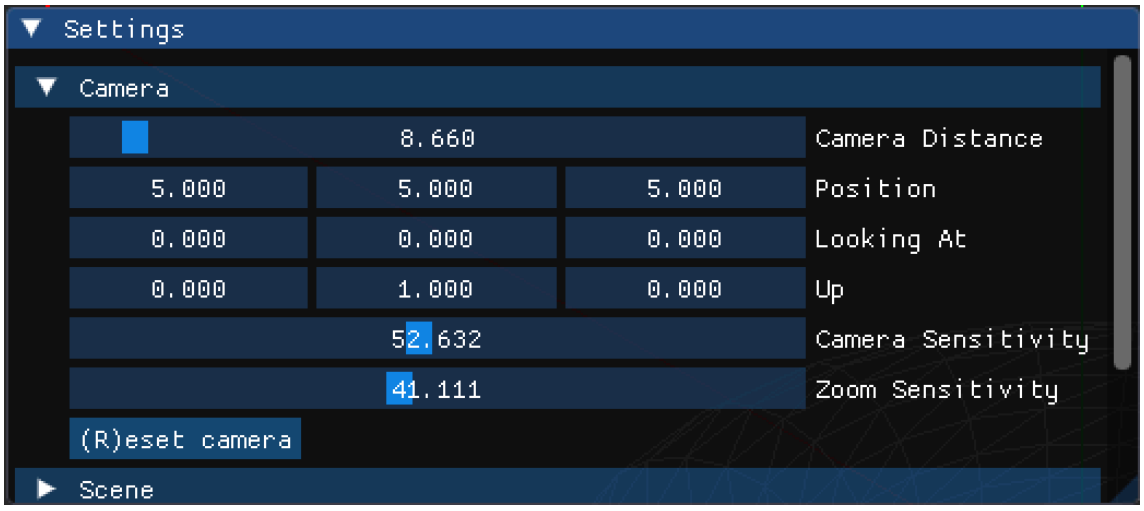
Anexo 1: Logo da Universidade do Minho



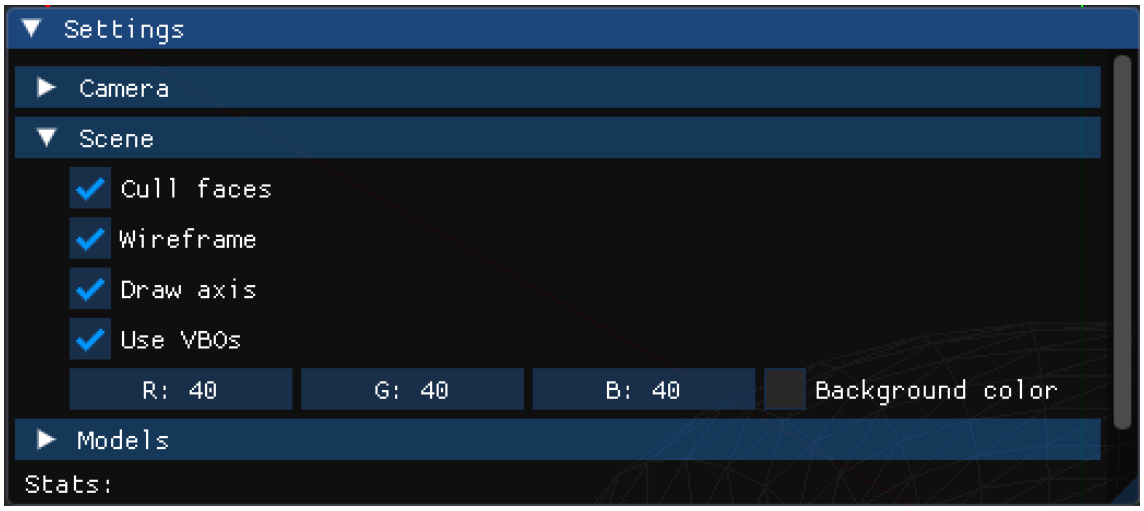
Anexo 2: Visão geral do menu



Anexo 3: Definições da câmara



Anexo 4: Definições da cena



Anexo 5: Definições dos modelos