Computação Gráfica - Fase 1

5 de Março de 2024

Grupo 2

Daniel Pereira	**Duarte Ribeiro**	**Francisco Ferreira**	**Rui Lopes**
A100545	A100764	A100660	A100643

Introdução

Este relatório tem como objetivo apresentar o trabalho prático desenvolvido para a primeira fase da unidade curricular de Computação Gráfica. O trabalho consiste no desenvolvimento de uma *engine* que carrega cenas com modelos a partir de um ficheiro e a mostra no ecrã, usando várias técnicas sobre o tópico da UC.

1. Visão geral da arquitetura do projeto

O projeto encontra-se dividido em dois programas principais: gerador e *engine*. Este primeiro tem como propósito a geração de figuras primitivas gráficas, resultando em ficheiros `.3d`. Esses ficheiros são, depois, embutidos em ficheiros `.xml`, lidos e renderizados pela *engine*. O funcionamento de cada um destes programas é detalhado ao pormenor nas secções seguintes.

2. Geração de figuras primitivas gráficas

Como requisito deste trabalho prático, foi necessário fazer um gerador de primitivas gráficas. Isto consiste num programa que, conforme os argumentos passados para o mesmo, escreve num ficheiro de texto informações necessárias para desenhar a primitiva pretendida no ecrã.

O nosso programa gera ficheiros como resultado, num formato idealizado pelo grupo, que nós apelidamos de ficheiros `.3d`. Estes ficheiros em formato textual seguem o seguinte formato: a primeira linha contém o número de vértices do modelo, n, seguido de n vértices tridimensionais com coordenadas separadas por espaço seguindo a ordem x, y e z, que formam a figura geométrica em questão. Um exemplo de um retângulo a ocupar o ecrã completamente[1] guardado neste formato seria:

```
6
-1 1 0
-1 -1 0
1 1 0
1 1 0
-1 -1 0
1 -1 0
```

A forma como este ficheiro é lido para memória no programa da *engine* é explicada no próximo capítulo.

[1] Assumindo que não há transformações na câmara (coordenada z ignorada), o centro do ecrã tem as coordenadas $(0, 0)$, o canto inferior esquerdo tem as coordenadas $(-1, -1)$ e o canto superior direito tem as coordenadas $(1, 1)$.

Foi optado pelo formato textual e não por um formato binário, que poderia trazer benefícios de rapidez de leitura e melhor utilização de espaço de disco, devido à conveniência de leitura e alteração de dados por parte de um utilizador, que assim, não tem de recorrer a programas que interpretariam o formato binário para o editar. Desta forma, um utilizador poderá apenas editar o ficheiro em forma de texto e poderá alterar ou adicionar novos vértices muito facilmente.

Nas fases seguintes, este formato será estendido e alterado de modo a guardar índices de faces, vetores normais, informação de texturas, entre outros possíveis requisitos dessas. Para já, achamos que temos um formato adequado para esta fase.

2.1. As figuras

Nesta fase houve como requisito o desenvolvimento da geração das seguintes primitivas: Plano, Esfera, Cone e Caixa. Como extra, também desenvolvemos a geração da primitiva Cilindro.

A forma como um utilizador pode gerar estas figuras está auto-descrita na mensagem de *help* do gerador:

```
Usage: generator <command> <args> <output file>
Commands:
        generator plane <length> <divisions> <output file>
        generator sphere <radius> <slices> <stacks> <output file>
        generator cone <radius> <height> <slices> <stacks> <output file>
        generator box <length> <divisions> <output file>
        generator cylinder <radius> <height> <slices> <output file>
```

Segue-se a descrição de como foram trianguladas cada uma dessas figuras.

2.1.1. Geração do plano

- Parâmetros: Tamanho (decimal), Divisões (inteiro);

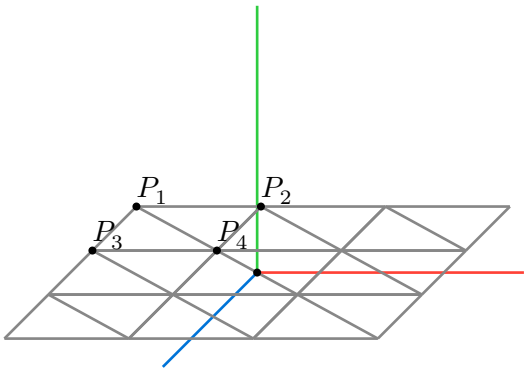

Figura 1: Primeira iteração da geração de planos[2]

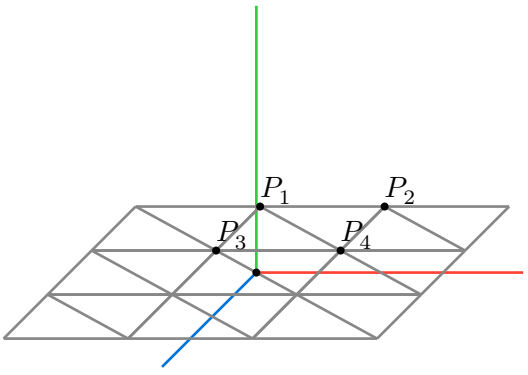

Figura 2: Segunda iteração da geração de planos

As triangulações são feitas da forma como estão descritas nas figuras acima. O tamanho é dividido pelo número de divisões, e assim obtemos o espaçamento entre os pontos. Os pontos P_1, P_2, P_3 e P_4 são obtidos da seguinte forma:

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

$$P_1 = \left(-\frac{l}{2} + x \times s, 0, -\frac{l}{2} + z \times s\right), \forall x, z \in \{0, 1, ..., d\}$$
$$P_2 = \left(-\frac{l}{2} + x \times s, 0, -\frac{l}{2} + z \times s\right), \forall x, z \in \{0, 1, ..., d\}$$
$$P_3 = \left(-\frac{l}{2} + x \times s, 0, -\frac{l}{2} + z \times s\right), \forall x, z \in \{0, 1, ..., d\}$$
$$P_4 = \left(-\frac{l}{2} + x \times s, 0, -\frac{l}{2} + z \times s\right), \forall x, z \in \{0, 1, ..., d\}$$

Sendo, l o tamanho do plano, s o tamanho de cada subdivisão (igual a $\frac{l}{d}$) e d o número de divisões.

A cada iteração, estes pontos são colocados como resultado pela ordem: $\{P_1, P_3, P_4\} + \{P_1, P_4, P_2\}$, formando os dois triângulos de cada subdivisão com normal paralela ao eixo y (para ser vista de cima).

2.1.2. Geração da Esfera

- Parâmetros: Raio (decimal), *Slices* (inteiro), *Stacks* (inteiro);

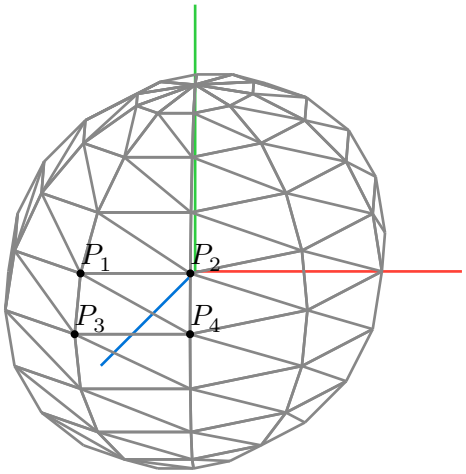

Figura 3: Uma iteração da geração de esferas

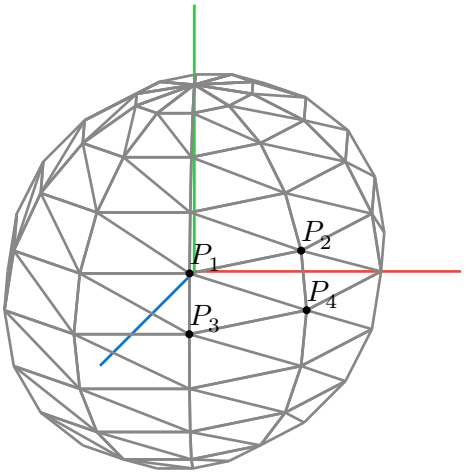

Figura 4: Próxima iteração da geração de esferas

A geração de esferas tem o mesmo princípio da geração dos planos, mas, neste caso, é usado coordenadas esféricas para este efeito. No caso, a cada iteração é calculado um novo ângulo para a posição dos pontos.

A conversão de coordenadas esféricas para coordenadas cartesianas é dada pela fórmula:

$$\mathcal{S}(r, \alpha, \beta) = (r\cos(\beta)\sin(\alpha), r\sin(\beta), r\cos(\beta)\cos(\alpha)) \quad (1)$$

Com isto, para gerar a esfera, podemos fazer com que α incremente $\frac{2\pi}{\text{slices}}$ a cada iteração e que β incremente $\frac{\pi}{\text{stacks}}$ a cada iteração do α. O número de iterações totais é dado por slices $\times$ stacks vezes. O r é dado pelo parâmetro "Raio" da esfera.

Assim, os pontos P_1, P_2, P_3 e P_4 são obtidos da seguinte forma:

$$P_1 = \mathcal{S}\left(r, \text{slice} \times \frac{2\pi}{\text{slices}}, (\text{stack}+1) \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$
$$P_2 = \mathcal{S}\left(r, (\text{slice}+1) \times \frac{2\pi}{\text{slices}}, (\text{stack}+1) \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$
$$P_3 = \mathcal{S}\left(r, \text{slice} \times \frac{2\pi}{\text{slices}}, \text{stack} \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$
$$P_4 = \mathcal{S}\left(r, (\text{slice}+1) \times \frac{2\pi}{\text{slices}}, \text{stack} \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$

Sendo slice a iteração do *slice* atual, stack a iteração da *stack* atual e $\mathcal{S}$ a função em Equação 1.

A razão de se subtrair $\frac{\pi}{2}$ é relacionado à necessidade de "rodar" a esfera, sobre o eixo do z, noventa graus, devido à nossa definição do $\mathcal{S}$ (Equação 1).

Os pontos são adicionados ao resultado na mesma ordem do que o plano: $\{P_1, P_3, P_4\} + \{P_1, P_4, P_2\}$.

2.1.2.1. Problema dos polos da esfera

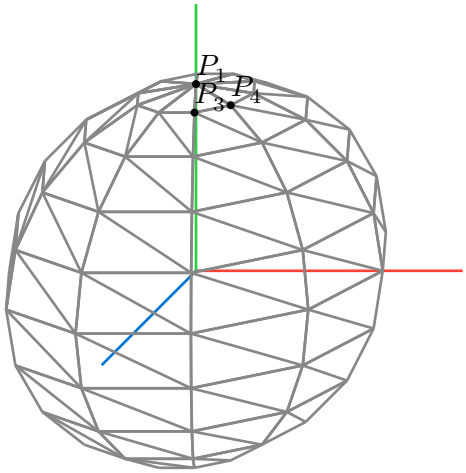

Figura 5: Iteração no polo superior na geração da esfera

No caso demonstrado na figura acima, P_1 e P_2 partilham as mesmas coordenadas devido a estarem num polo da esfera. Usando a mesma lógica de fazer a triangulação da forma $\{P_1, P_3, P_4\} + \{P_1, P_4, P_2\}$, o segundo triângulo vai ser apenas uma linha. Para evitar este triângulo inválido, apenas a primeira triangulação é adicionada, quando a iteração está num dos polos.

Este problema também acontece na Geração do Cone.

2.1.3. Geração do Cone

- Parâmetros: Raio (decimal), Altura (decimal), *Slices* (inteiro), *Stacks* (inteiro);

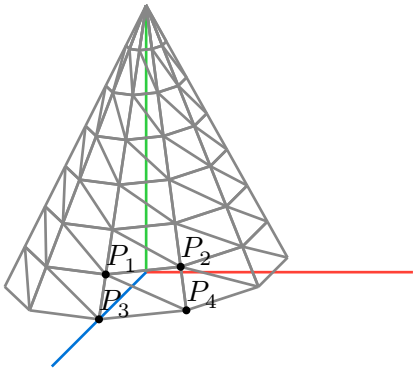

Figura 6: Primeira iteração da geração de cones

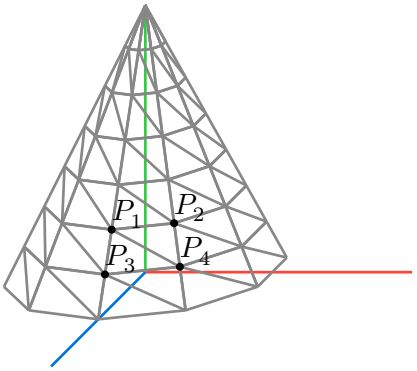

Figura 7: Segunda iteração da geração de cones

Similarmente às figuras anteriores, o cone usa o mesmo método de iterações sobre *slices* e *stacks* da esfera, mas a cada *stack* (divisão horizontal), calcula os seus pontos a partir de coordenadas polares. O componente do raio faz uma interpolação linear desde a base até ao topo, tendo aí o valor 0, dando o efeito do cone.

A conversão de coordenadas polares para coordenadas cartesianas é dada pela fórmula:

$$\mathcal{P}(r, \alpha, y) = (r\sin(\alpha), y, r\cos(\alpha)) \tag{2}$$

Usando isto, pode-se facilmente gerar cada *stack* do cone como um círculo, unindo com a *stack* da camada superior.

Com a mesma lógica e ordem dos pontos das figuras geométricas anteriores temos:

$$\begin{aligned}
l_{\text{slice}} &= \frac{2\pi}{\text{slices}} \\
l_{\text{stack}} &= \frac{h}{\text{stacks}} \\
r_i &= r - \text{stack} \times l_{\text{stack}} \\
r_{i+1} &= r - (\text{stack} + 1) \times l_{\text{stack}} \\
P_1 &= \mathcal{P}(r_{i+1}, \text{slice} \times l_{\text{slice}}, (\text{stack} + 1) \times l_{\text{stack}}) \\
P_2 &= \mathcal{P}(r_{i+1}, (\text{slice} + 1) \times l_{\text{slice}}, (\text{stack} + 1) \times l_{\text{stack}}) \\
P_3 &= \mathcal{P}(r_i, \text{slice} \times l_{\text{slice}}, \text{stack} \times l_{\text{stack}}) \\
P_4 &= \mathcal{P}(r_i, (\text{slice} + 1) \times l_{\text{slice}}, \text{stack} \times l_{\text{stack}})
\end{aligned}$$

Sendo slice o *slice* da iteração atual, stack o *stack* da iteração atual, h a altura do cone, stacks o número de *stacks* do cone, slices o número de *slices* do cone e r o raio do cone.

Os pontos são adicionados ao resultado na mesma ordem do que a esfera: $\{P_1, P_3, P_4\} + \{P_1, P_4, P_2\}$.

2.1.3.1. Base do cone

A geração descrita não é suficiente para completar o cone, visto que a base do cone não é tratada. Para resolver isto, por cada *slice* há um triângulo a mais que liga deste os vértices da camada da base até ao ponto da origem, que é o centro da base do cone.

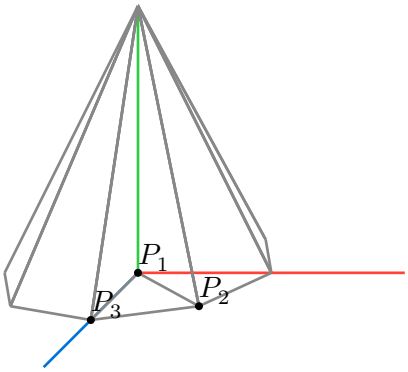

Figura 8: Iteração para a geração da base do cone

Estes pontos são adicionados ao resultado pela ordem: $\{P_1, P_2, P_3\}$. Assim, o triângulo fica com normal com vetor "para baixo", mostrando assim a base, apenas quando a câmara o vê por baixo.

2.1.3.2. Problema dos polos do cone

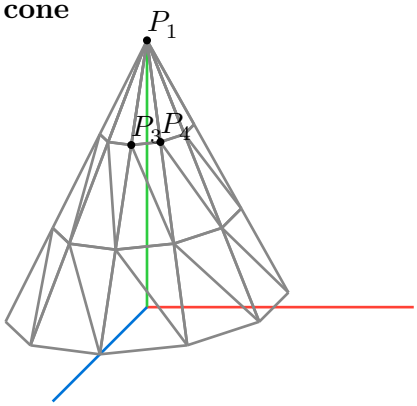

Figura 9: Iteração no polo superior na geração do cone

Na mesma forma de ideias da esfera (Secção 2.1.2.1), o cone também partilha o mesmo problema de, na iteração no polo superior, o ponto P_1 partilhar as mesmas coordenadas do P_2. Isto é resolvido da mesma forma, em que não é adicionado o segundo triângulo, $\{P_1, P_4, P_2\}$, à lista de vértices resultado, neste caso.

2.1.4. Geração da Caixa

- Parâmetros: Tamanho (decimal), Divisões (inteiro);

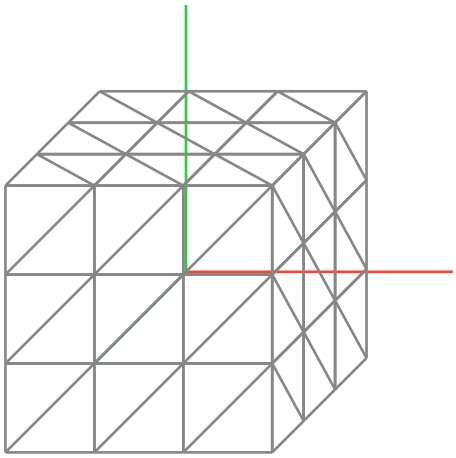

Figura 10: Geração de uma caixa

Conseguimos notar facilmente que a caixa é composta com várias réplicas (6 faces) do plano. Usando esta noção e tendo o conhecimento de multiplicação de matrizes e como elas conseguem fazer rotações e translações em vetores, podemos chegar à caixa muito facilmente. Podemos aproveitar o código feito para a Geração do Plano e fazer as tais transformações nos pontos com as matrizes adequadas. Isto já aproveita a lógica de subdivisão dos planos e não temos de lidar com isso.

Iremos mostrar as matrizes que fazem as tais rotações e translações necessárias, que foram aplicadas a todos os pontos gerados pela Geração do Plano.

De notar, que foi adicionado um quarto componente a mais com valor 1 a todos os pontos, para se conseguirem fazer translações a partir de matrizes.

Face superior:

Esta face é simples, visto que apenas é necessário deslocar o plano $\frac{l}{2}$ unidades sobre o eixo y, sendo l o tamanho da caixa.

$$\mathcal{F}_{\text{up}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \frac{l}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Face inferior:

A face inferior passa por uma translação de $-\frac{l}{2}$ unidades sobre o eixo y e uma rotação de 180º sobre o eixo x, para que face fique virada para baixo (corrigindo assim a sua normal).

$$\mathcal{F}_{\text{down}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -\frac{l}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\pi) & -\sin(\pi) & 0 \\ 0 & \sin(\pi) & \cos(\pi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Face frontal:

Obtém-se a face frontal aplicando uma translação de $\frac{l}{2}$ no eixo do z, e uma rotação de 90º sobre o eixo do x.

$$\mathcal{F}_{\text{front}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \frac{l}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) & 0 \\ 0 & \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Face da direita:

A face da direita é calculada fazendo uma translação de $\frac{l}{2}$ no eixo do x e uma rotação de -90^{o} sobre o eixo do z.

$$\mathcal{F}_{\text{right}} = \begin{bmatrix} 1 & 0 & 0 & \frac{l}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(-\frac{\pi}{2}) & -\sin(-\frac{\pi}{2}) & 0 & 0 \\ \sin(-\frac{\pi}{2}) & \cos(-\frac{\pi}{2}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Face da esquerda:

Já a face da esquerda é obtida pelo inverso da face da direita, ou seja, uma translação de $-\frac{l}{2}$ no eixo do x e uma rotação de 90^{o} sobre o eixo do z.

$$\mathcal{F}_{\text{left}} = \begin{bmatrix} 1 & 0 & 0 & -\frac{l}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) & 0 & 0 \\ \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Face traseira:

Por fim, a face traseira é gerada pelo inverso da face da frente, isto é, uma translação de $-\frac{l}{2}$ no eixo do z e uma rotação de -90^{o} sobre o eixo do x.

$$\mathcal{F}_{\text{back}} = \begin{bmatrix} 1 & 0 & 0 & -\frac{l}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) & 0 \\ 0 & \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Com isto, os pontos gerados por todas estas faces formam a caixa.

2.2. Geração do Cilindro

- Parâmetros: Raio (decimal), Altura (decimal), *Slices* (inteiro);

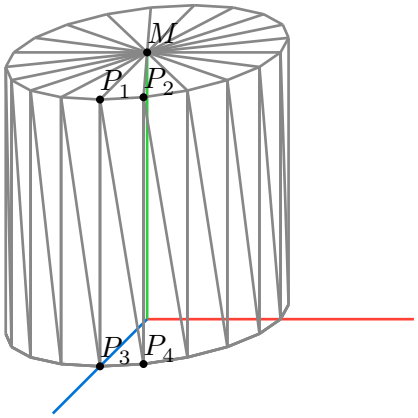

Figura 11: Primeira iteração da geração de cilindros

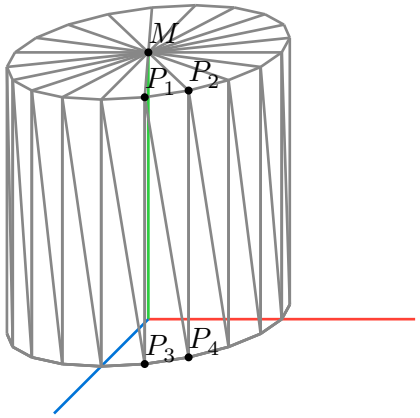

Figura 12: Segunda iteração da geração de cilindros

A geração de cilindros é muito similar à Geração do Cone mas sem a interpolação do raio (mantendo-se sempre igual), tendo uma base a mais no topo e sem divisão de *stacks*.

Com isto, os pontos P_1, P_2, P_3 e P_4 são obtidos da seguinte forma:

$$P_1 = \mathcal{S}\left(r, \text{slice} \times \frac{2\pi}{\text{slices}}, h\right)$$
$$P_2 = \mathcal{S}\left(r, (\text{slice}+1) \times \frac{2\pi}{\text{slices}}, h\right)$$
$$P_3 = \mathcal{S}\left(r, \text{slice} \times \frac{2\pi}{\text{slices}}, 0\right)$$
$$P_4 = \mathcal{S}\left(r, (\text{slice}+1) \times \frac{2\pi}{\text{slices}}, 0\right)$$

Sendo slice a iteração atual, slices o número de *slices* e h a altura do cilindro.

Com a mesma ordem do cone, os triângulos são adicionados pela ordem: $\{P_1, P_3, P_4\} + \{P_1, P_4, P_2\}$.

Também são adicionados os triângulos das bases pela ordem: $\{M, P_1, P_2\} + \{M_{\text{base}}, P_3, P_4\}$ a cada iteração.

3. *Parsing* dos arquivos de cena `.xml`

A *engine* começa a ler o arquivo `.xml`, passado como parâmetro ao programa, fazendo recurso à biblioteca *TinyXML2* [1]. A estrutura dos arquivos de cena `.xml` está de acordo com a que foi requerida pelo trabalho prático e os seus exemplos. Para já, só é configurável o tamanho da janela da *engine*, posição, olhar, vetor *up*, *FOV*[3], *z-near* e *z-far* da câmara e modelos dentro de grupos (podendo um grupo estar dentro de outro).

O programa lê essas informações e coloca-as em memória para rápido acesso na renderização. Informações extra são simplesmente ignoradas. O programa só dá erro quando o formato dos parâmetros configuráveis não está correto.

[3] *Field of View*

3.1. *Parsing* dos Modelos

Aquando o *parser* de arquivos de cena `.xml` encontra um elemento com o nome `<model>`, o ficheiro com nome do atributo `file` desse elemento é lido de acordo com a sua extensão, resultado num `Model`, uma estrutura de dados que guarda uma lista de vértices que será posteriormente renderizada.

3.1.1. Parsing de `.3d`

Caso o ficheiro encontrado na fase descrita no parágrafo anterior tenha como extensão `.3d` é chamado o *parser* de modelos `.3d`. Esse *parser* começa por ler a primeira linha[4], e aloca um `std::vector` com esse tamanho, n.

Após isso, são lidas n linhas, que são transformadas em vértices e adicionadas ao *vector* criado inicialmente pela ordem de leitura.

No fim, é criado a estrutura de dados `Model` com a lista desses vértices, dentro do `World`, que contém os dados para a *engine* renderizar o mundo.

Atenção a ter para as próximas fases

Neste formato não há informação sobre índices de vértices, havendo vértices repetidos. Para a implementação de ***VBO with indexing*** será necessário a alteração deste formato para guardar tal. Nas fases futuras também será necessário guardar informação relativa a vetores normais e posições de texturas. Estes são aspetos que o grupo teve em consideração mas optou por escolher um formato mais simples de acordo com os requisitos para esta fase.

3.1.2. Parsing de `.obj`

Como forma de expansão do projeto, para além do nosso formato `.3d`, também suportamos a leitura de `Wavefront .obj files`.

O formato está especificado aqui: https://en.wikipedia.org/wiki/Wavefront_.obj_file.

Este formato de ficheiro já tem suporte a indexação de vértices, posições de texturas e normais, pelo que, poderá ser um formato a considerar para se tornar o *standard* do projeto, nas fases futuras.

Para já, da parte do nosso programa, o ficheiro `.obj` é carregado de acordo com as faces nele registadas e os seus vértices correspondentes, gerando, novamente o `std::vector` de vértices que o `Model` necessita.

Com isto, devido à existência de vários programas de modelação com suporte a exportação para este formato, tornámos a nossa *engine* muito mais versátil, dando suporte a carregar modelos de terceiros.

[4]Recorde-se o exemplo do ficheiro: Ficheiro exemplo retângulo

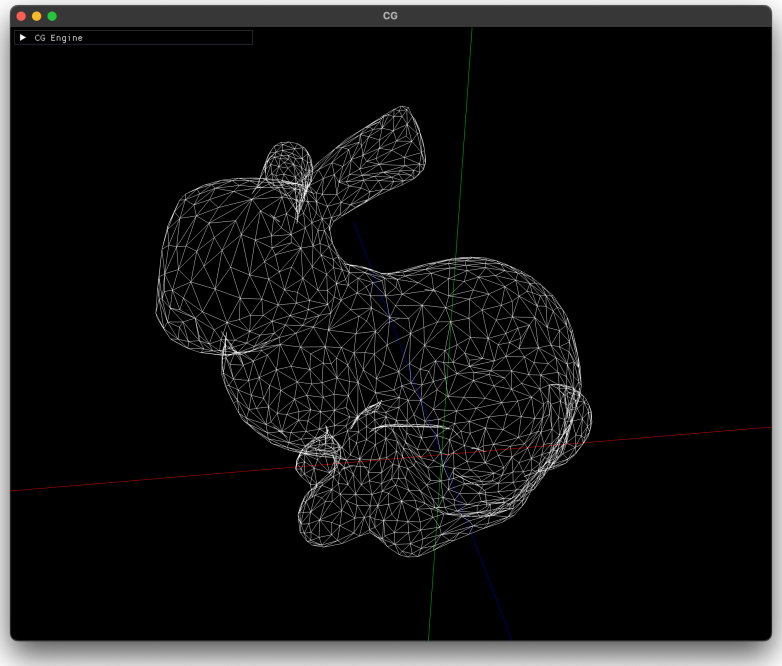

Figura 13: Renderização de um Stanford bunny de um modelo `.obj`

4. *Build* do Projeto

De forma a assegurar grande compatibilidade entre os vários sistemas operativos utilizados, recorremos ao *vcpkg* [2]. Com o *CMake* [3], o *vcpkg* é responsável por assegurar o *setup* das várias dependências nos vários sistemas operativos, cumprindo a necessidade de cada um tem para cada dependência.

Com compatibilidade ótima do *vcpkg* sobre as dependências usadas[5], o projeto compila sem qualquer problema sem precisar de *sources* ou *DLLs* dessas dependências no nosso código fonte.

O resultado da execução do *CMake* são dois executáveis: o gerador de primitivas e o programa da *engine*.

4.1. GitHub Actions

Para concretizar o objetivo citado nos parágrafos anteriores, estando o projeto hospedado num repositório privado do *GitHub* [8], demos uso à funcionalidade de *GitHub Actions* [9] para, a cada *commit* no repositório, testar a compilação nos sistemas operativos *Windows*, *MacOS* e *Linux (Ubuntu)*.

Com isto, a integração continua do projeto é mantida muito facilmente devido à facilidade do isolamento de pedaços de código de um *commit* que deram problemas num compilador de um sistema operativo diferente.

Esta funcionalidade já foi extremamente útil nesta primeira fase do projeto, com a deteção de algumas diferenças de *headers* nos diferentes compiladores.

[5]GLFW [4], OpenGL [5], TinyXML2 [1], ImGui [6], Glew [7].

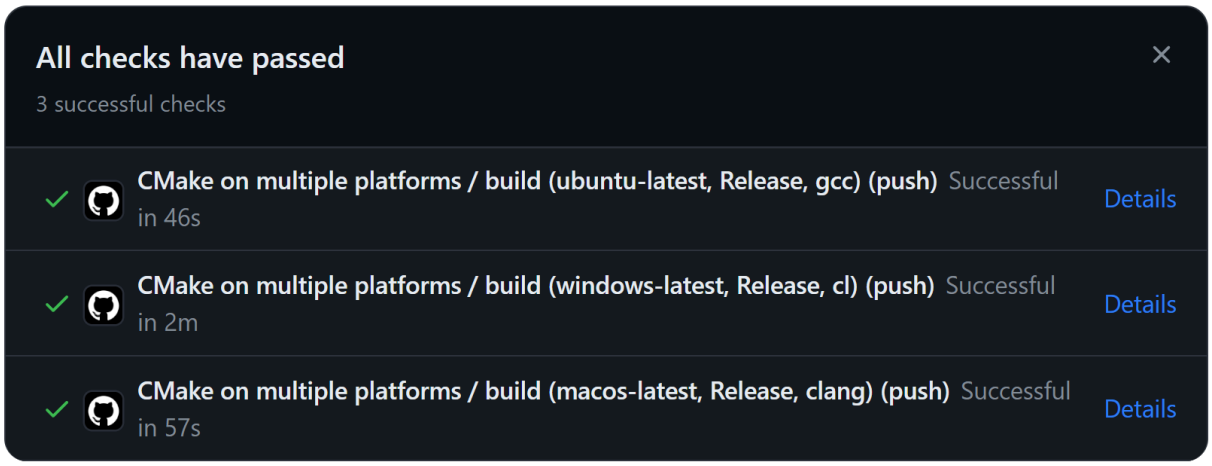

Figura 14: Resultado de um *commit* após serem rodadas todas as verificações da *Action*

Futuramente, também temos ideia em implementar alguns testes unitários, fazendo uso ainda mais desta ferramenta.

5. A *Engine*

De uma forma geral, a *engine* é o programa responsável por ler o ficheiro de cena `.xml` e renderizar o mundo descrito nesse ficheiro. A mesma é simplesmente um motor de renderização 3D, que além de desenhar os modelos, tem também suporte a interações com o utilizador, como alterar a posição da câmara, ativar/desativar funcionalidades do *OpenGL* e mostrar informações relevantes do mundo.

Essencialmente, o programa da *engine* começa por criar uma classe `World` a partir do ficheiro de cena `.xml` passado como argumento. Esta classe contém informações tais como o nome do mundo, informações da janela em que o programa está a correr, a câmara e os modelos em questão. Seguidamente, é criada a classe `Engine`, que implementa funcionalidades pretendidas do OpenGL. Esta contém métodos para inicializar a *engine* propriamente dita, renderizar o mundo, alterar definições do mundo e processar *input* do utilizador.

O ciclo de vida da *engine* pode ser descrito através do seguinte bloco de código:

```
void Engine::Run() {
  Init();

  while (!WindowShouldClose(...)) {
    PollEvents();
    ProcessInput();
    Render();
  }

  Shutdown();
}
```

A função `Init` é responsável por inicializar a *engine* e todo o contexto do OpenGL.

A função `WindowShouldClose`[6] tem como objetivo verificar se o utilizador fechou a janela. Em caso positivo, o ciclo é terminado e a *engine* é desligada.

[6] A função é referente à biblioteca *GLFW*.

Dentro do corpo do ciclo, a função `PollEvents`[6] é responsável por processar todos os eventos que ocorreram desde a última iteração do ciclo; a função `ProcessInput` é responsável por processar, desses eventos, os de *input* do utilizador, como por exemplo, teclas que na posição da câmara e a função `Render` é responsável por desenhar o mundo no ecrã em conformidade.

Muito importante é a função `Render`. Esta, itera sobre todos os modelos do mundo e desenha-os no ecrã, aplicando a função `glVertex3f` a cada um dos vértices[7].

5.1. *GLFW*

Para gestão de janelas, eventos e contextos OpenGL, utilizamos a biblioteca *GLFW* [4]. Esta biblioteca, para além da sua utilização ser muito simples e intuitiva (em relação ao *GLUT*), tem suporte a algumas funcionalidades que demos uso. Algumas destas funcionalidades como suporte a alternação ligar e desligar *VSync* em *runtime*, suporte a MSSA, foram implementadas, e serão explicitadas num próximo capítulo.

5.2. *GLEW*

Para carregar as funções de OpenGL, também utilizamos a biblioteca *GLEW*. Esta biblioteca é necessária para carregar as funções de OpenGL de versões mais recentes, que não são carregadas por defeito por alguns sistemas operativos.

5.3. Interações

O ponto mais diferencial da nossa *engine* é a sua interação com o utilizador através de uma interface gráfica e controlo de rato e teclado.

De modo intuitivo, o utilizador pode alterar a posição da câmara de forma a rodar em torno do ponto para onde a câmara está a olhar, clicando e arrastanto o rato. O *scroll* do rato também é suportado, e altera a distância da câmara ao ponto para onde está a olhar. Estas alterações são feitas em tempo real numa movimentação suave e contínua[8].

A implementação desta funcionalidade segue os princípios de coordenadas esféricas, que as coordenadas cartesianas da câmara são convertidas para coordenadas esféricas, os ângulos (ou raio no caso do *scroll*) são alterados conforme a interação do utilizador, e as coordenadas esféricas são convertidas de volta para coordenadas cartesianas, atualizando a posição da câmara.

[7] Sempre com `glBegin(GL_TRIANGLES)` e `glEnd` antes e depois, respetivamente.

[8] Não é de facto contínua, mas as alterações entre *frames* são tão pequenas que dá a sensação de ser contínua.

```
void Camera::ProcessInput(const float x_offset, const float y_offset, const float
scroll_offset)
{
    float radius, alpha, beta;
    (position - looking_at).ToSpherical(&radius, &alpha, &beta);

    alpha -= x_offset * sensitivity;
    beta -= y_offset * sensitivity;
    radius -= scroll_offset * scroll_sensitivity;

    // Verificação de limites...

    const auto after = Vec3fSpherical(radius, alpha, beta);
    position = after + looking_at;
}
```

5.3.1. *ImGui*

A parte mais interessante está de facto dentro do menu do *ImGui*.

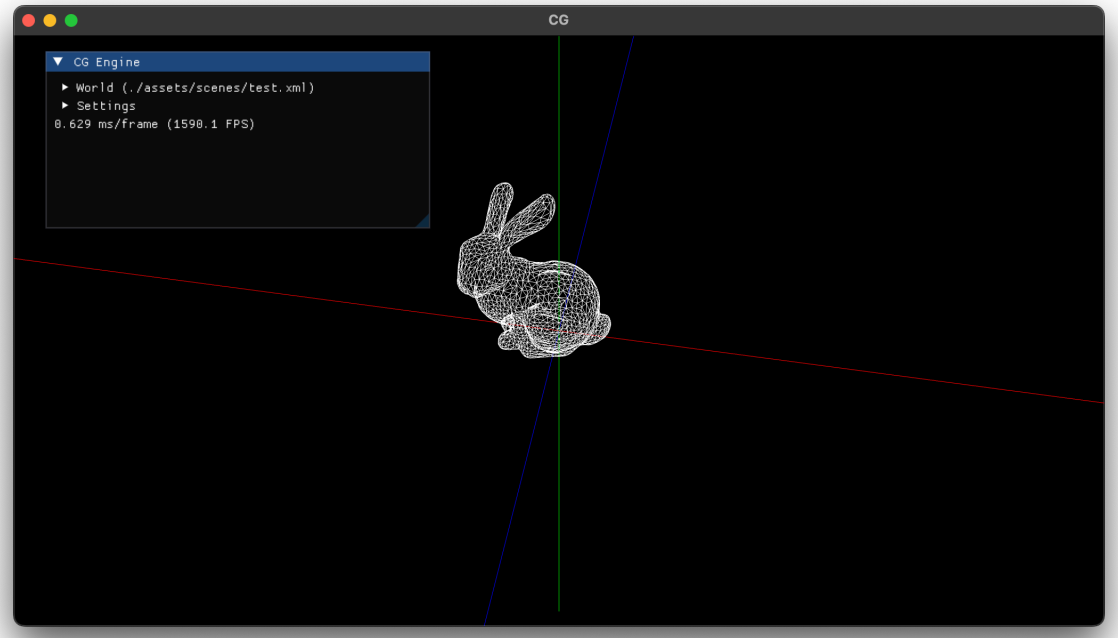

Figura 15: Vista principal do menu da interface

Nesta janela, podendo ser arrastada para qualquer lugar dentro do programa, numa forma de menus colapsados, temos a opção de visualizar opções do mundo, *settings* do programa e *FPS*[9].

[9] *Frames per Second*, dados pelo *ImGui* que usa o número de chamadas ao *render* da sua janela para calculá-los.

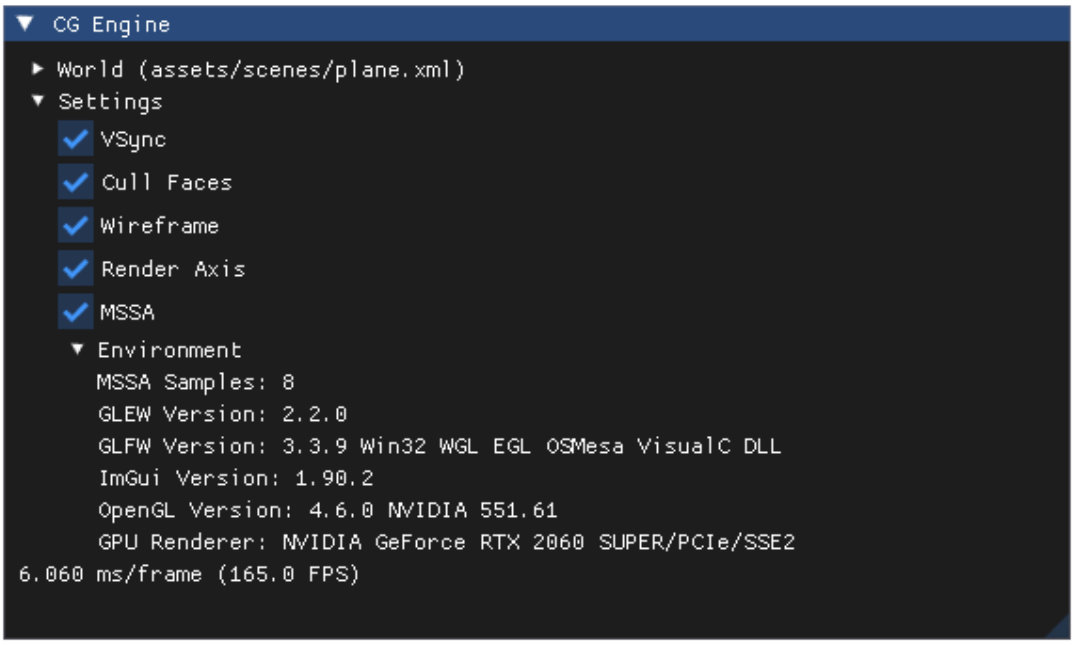

Figura 16: Vista *ImGui* com menu de *Settings* aberto

Abrindo a aba das *Settings*, é possível habilitar ou desabilitar funcionalidades como:

- ***VSync***, por parte do *GLFW*, que ativa/desativa a sincronização vertical;
- ***Cull Faces***, por parte do *OpenGL*, que mostra/esconde faces com normal não direcionadas para a câmara;
- ***Wireframe***, por parte do *OpenGL*, que alterna entre mostrar uma estrutura de arame entre os triângulos ou preenchê-los com cor completamente;
- ***Render Axis***, por parte da *engine* com *OpenGL*, que mostra/esconde os eixos do gráfico;
- ***MSSA***, por parte do *OpenGL* + *GLFW*, que liga ou desliga o MSSA[10].

A nossa implementação do *MSSA*, toma vantagem de não termos implementado o nosso próprio *framebuffer*, e faz uso de um *buffer multisample*, que os sistemas conseguem fornecer na criação de janelas.

Abrindo a aba *Environment* dentro das *Settings*, informações do sistema operativo e versões de bibliotecas são mostradas.

[10] *Multisample Anti-Aliasing*

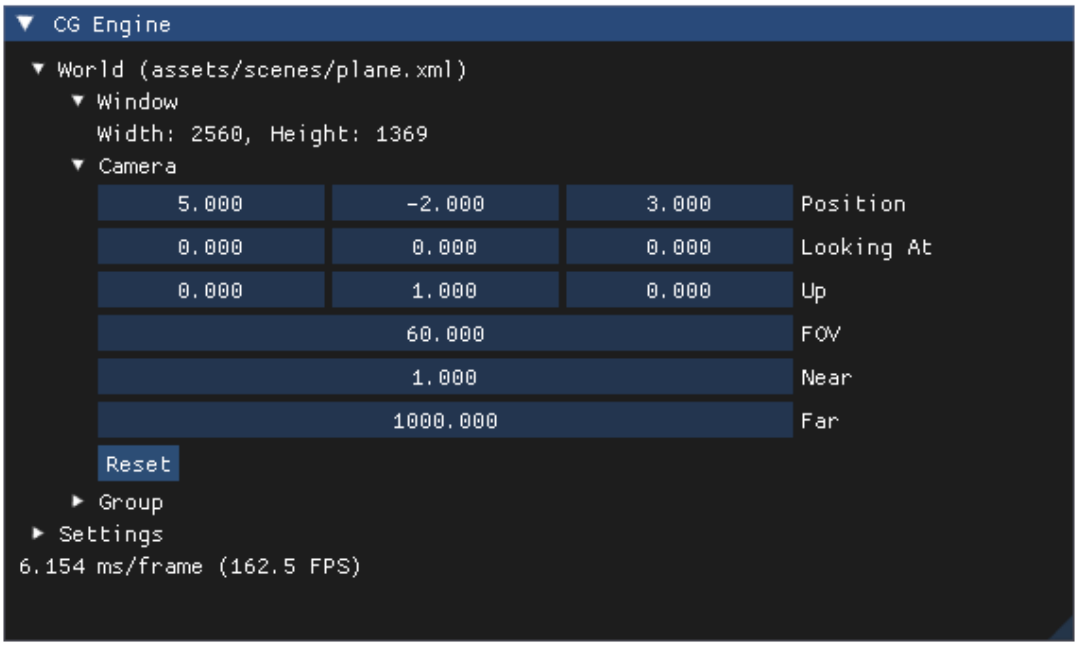

Figura 17: Vista *ImGui* com menu de *World* aberto

Passando para a aba *World* é possível ver informações do tamanho da janela da *engine* (a janela é redimensionável) e é possível alterar os parâmetros da câmara durante a execução do programa, arrastando como um slider ou clicando duas vezes com o rato e escrever o valor que queremos. Também é possível restabelecer a posição inicial da câmara clicando no botão *Reset*.

Abrindo a aba *World*, temos informação do mundo carregado a partir da cena `.xml`.

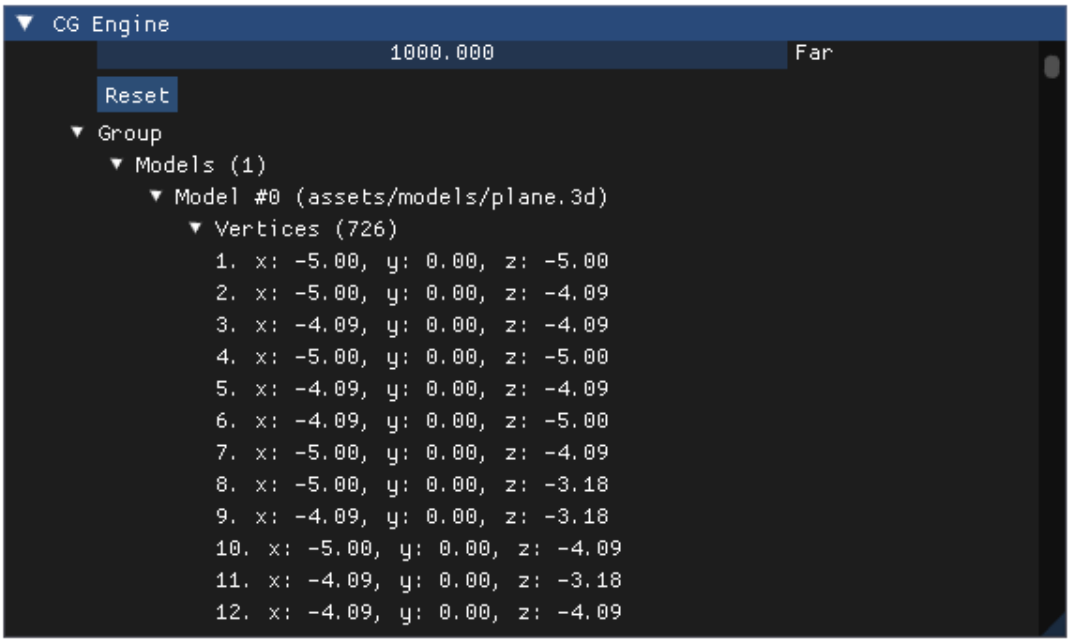

Figura 18: Vista *ImGui* com menu de *World Group* aberto

Nesta aba é possível ver informações de todos os modelos, como as posições dos seus vértices (a única informação neles presente até agora). Futuramente queremos expandir esta funcionalidade para, por exemplo, carregar novos modelos durante a execução, editá-los e ou removê-los.

6. Conclusão

Para concluir, o grupo está muito satisfeito com o resultado final desta primeira etapa do projeto. Acreditamos ter uma base muito sólida para realizar as próximas etapas, também, com sucesso.

Bibliografia

[1] L. Thomason, «TinyXML-2». [Em linha]. Disponível em: https://github.com/leethomason/tinyxml2

[2] «vcpkg - Open source C/C++ dependency manager from Microsoft». Acedido: 7 de março de 2024. [Em linha]. Disponível em: https://vcpkg.io/en/

[3] «CMake». [Em linha]. Disponível em: https://cmake.org/

[4] «An OpenGL library». [Em linha]. Disponível em: https://www.glfw.org/

[5] «OpenGL - The Industry Standard for High Performance Graphics». [Em linha]. Disponível em: https://www.opengl.org/

[6] ocornut, «ocornut/imgui». [Em linha]. Disponível em: https://github.com/ocornut/imgui

[7] «GLEW: The OpenGL Extension Wrangler Library». [Em linha]. Disponível em: https://glew.sourceforge.net/

[8] Github, «Build software better, together». [Em linha]. Disponível em: https://github.com/

[9] GitHub, «Features • GitHub Actions». [Em linha]. Disponível em: https://github.com/features/actions