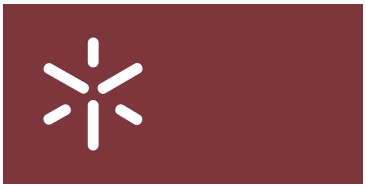

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2024/2025

Motor de Mini Cenas 3D Baseado em Grafos

Alex Sá a104257
José Vasconcelos a100763
Paulo Ferreira a96268
Rafael Fernandes a104271

2 de Março, 2025

CG

Resumo

A elevada utilização de dispositivos eletrónicos e computacionais na atualidade obriga cada vez mais à necessidade de implementação de software e hardware cada vez mais sofisticados. Uma das áreas onde podemos observar isso a acontecer é na área de computação gráfica, que já se demonstra presente em diversos aspetos do quotidiana, seja em aplicações CAD/CAM, filmes, jogos, interfaces e muito mais. Neste relatório será apresentado o processo de desenvolvimento de uma *engine* utilizando os conceitos mais básicos que existem por de trás destas aplicações. Aqui será descrito a primeira de quatro fases, sendo que esta terá maior incidência na definição das primitivas e os seus geradores, assim como no inicio do desenvolvimento do motor de cenas 3D(*engine*). Na fase final, já será apresentada uma demo, com iluminação, texturas e animações implementadas.

Área de Aplicação: Computação Gráfica

Palavras-Chave: OpenGL; C++; 3D Engine

Índice

1. Introdução

No âmbito da unidade curricular de Computação Gráfica, foi proposta a realização de um projeto prático que visa o desenvolvimento de um motor de mini cenas em três dimensões (3D), baseado em grafos. Este projeto, devido ao seu caráter progressivo, foi estruturado em quatro fases distintas, cada uma com objetivos específicos e complementares. A primeira fase concentrou-se na definição e implementação de **primitivas gráficas**, bem como no desenvolvimento inicial da ***engine*** (motor gráfico). As fases subsequentes dedicaram-se, respetivamente, às **transformações geométricas;** à modelação de **curvas**, **superfícies cúbicas** e à utilização de Vertex Buffer Objects (**VBOs**); e, por fim, à implementação de **normais** e coordenadas de **texturas**, elementos essenciais para a criação de cenas visualmente apelativas.

O presente relatório tem como objetivo descrever e analisar o processo de desenvolvimento da primeira fase do projeto, desde a sua conceção teórica até à concretização prática. Nesta fase, foi dada particular atenção ao desenvolvimento dos geradores de primitivas gráficas, os quais serão detalhados na **Secção 2**, e à estruturação da *engine*, cuja arquitetura e funcionalidades serão exploradas na **Secção 3**. A compreensão destes componentes é fundamental para o sucesso das fases posteriores, uma vez que constituem a base sobre a qual todo o sistema gráfico será construído.

Através deste trabalho, pretende-se não apenas consolidar os conceitos teóricos abordados no âmbito da unidade curricular, mas também demonstrar a sua aplicação prática no desenvolvimento de ferramentas gráficas robustas e eficientes. A execução deste projeto permitiu, ainda, uma maior familiarização com tecnologias e técnicas essenciais no domínio da computação gráfica, tais como o **OpenGL**, que desempenha um papel central na renderização de cenas 3D.

2. Gerador

O gerador é um programa desenvolvido integralmente em C++, tal como a *engine*, e tem como função principal a criação de sete primitivas gráficas. A implementação do código responsável pela geração de cada uma dessas primitivas baseou-se em fórmulas matemáticas que permitem, através da utilização de parâmetros específicos, determinar as coordenadas dos vértices que compõem as formas geométricas. Dessa forma, torna-se possível desenhar as primitivas de maneira precisa e eficiente. A transcrição dessas fórmulas matemáticas para código será detalhada nas subsecções seguintes.

Dentre os objetos gerados, quatro eram considerados essenciais para o desenvolvimento do projeto: **o plano**, **a caixa**, **a esfera** e **o cone**. No entanto, para proporcionar maior flexibilidade e variedade na construção da cena final, foram também implementados geradores para **o cilindro**, **o torus**, **um flatRing** (anel achatado) e um **icosaedro** que permite inclusivamente criar uma *icosphere*. Outros objetos, como a pirâmide, o octaedro e o paralelepípedo, foram considerados para implementação, mas, após análise, verificou-se que não eram estritamente necessários, uma vez que podem ser obtidos através de alterações às medidas, e de transformações geométricas específicas, aplicadas ao cone e ao cilindro, respetivamente. Por outro lado, formas como o elipsoide e a cápsula permanecem como objetos de estudo, podendo vir a ser implementados em fases posteriores do projeto, caso se revele pertinente.

Nas subsecções seguintes, será detalhado o desenvolvimento das primitivas consideradas essenciais para o cumprimento dos requisitos propostos, com enfoque na sua fundamentação matemática e na respetiva implementação em código.

2.1. Primitivas Essenciais

2.1.1. Plano

Para a construção do plano, são necessários dois parâmetros de construção: o comprimento de cada aresta, doravante denominado len, e o número de divisões dessas mesmas arestas, doravante denominado $\mathrm{sdCount}$. Conjuntamente, estes parâmetros permitem a definição de uma grelha com $\mathrm{sdCount}$ X $\mathrm{sdCount}$ segmentos, cada um com comprimento $\frac{\mathrm{len}}{\mathrm{sdCount}}$. Cada segmento é composto por dois triângulos isósceles unidos pelas suas hipotenusas e com catetos congruentes de medida $\frac{\mathrm{len}}{\mathrm{sdCount}}$.

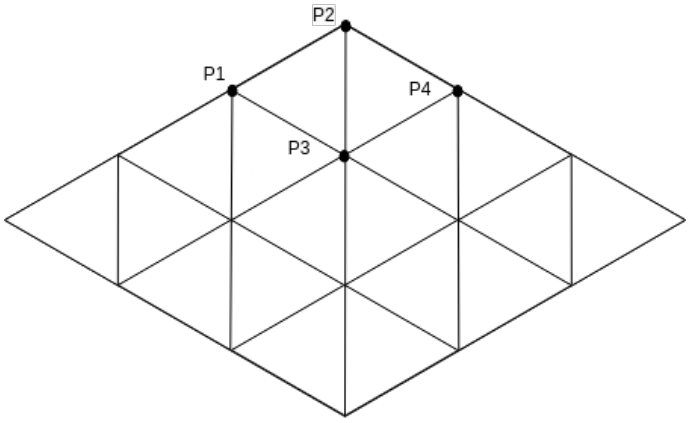

Figura 1: Plano de grelha de segmentos 3x3.

Como ilustrado na Figura 1, o segmento localizado no canto superior direto é composto por dois triângulos, (P3 P4 P2) e (P3 P2 P1). Todos os segmentos são construídos segundo a mesma forma, definida pelas equações:

$$\mathrm{P1} = (-\mathrm{offset} + \mathrm{sdCount} * j, \mathrm{offset}, -\mathrm{offset} + \mathrm{sdCount} * (i+1))$$
$$\mathrm{P2} = (-\mathrm{offset} + \mathrm{sdCount} * j, \mathrm{offset}, -\mathrm{offset} + \mathrm{sdCount} * i)$$
$$\mathrm{P3} = (-\mathrm{offset} + \mathrm{sdCount} * (j+1), \mathrm{offset}, -\mathrm{offset} + \mathrm{sdCount} * (i+1))$$
$$\mathrm{P4} = (-\mathrm{offset} + \mathrm{sdCount} * (j+1), \mathrm{offset}, -\mathrm{offset} + \mathrm{sdCount} * i)$$

onde:

$$\mathrm{offset} = \frac{\mathrm{len}}{2},$$

i é o índice do segmento no eixo X,

j é o índice do segmento no eixo Y

2.1.2. Box

Para a construção da caixa, são necessários dois parâmetros fundamentais, à semelhança do que ocorre na definição de um plano: o comprimento de cada aresta e o número de divisões em cada lado. Estas divisões, por sua vez, resultam na geração de faces organizadas segundo uma *grid* de dimensão NxN, onde N representa o número de divisões. É importante destacar que a caixa poderia ter sido construída a partir de um plano, aplicando sucessivas transformações e rotações por meio de matrizes, porém optamos por uma abordagem baseada na definição dos vértices diretamente nas coordenadas desejadas, de forma a minimizar qualquer impacto no desempenho computacional, ainda que este seja praticamente nulo no contexto do sistema que estamos a desenvolver.

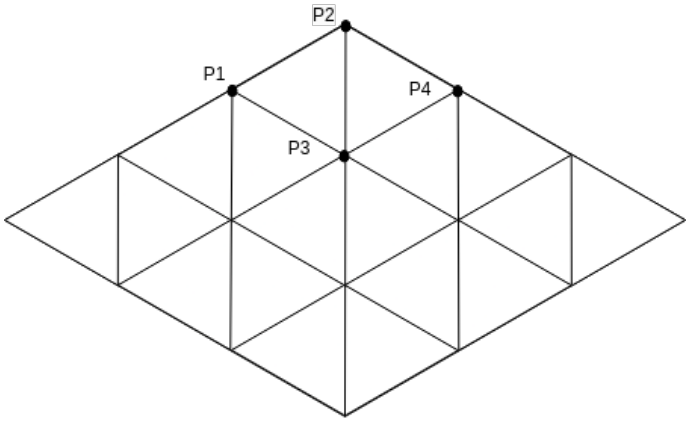

Figura 2: Face superior da caixa de grid 3x3.

Deste modo, iniciamos o desenvolvimento da caixa através da definição da face superior, situada no plano $y = \frac{\text{len}}{2}$, onde len corresponde ao comprimento de cada aresta. Cada face da caixa possui subdivisões formadas a partir de dois triângulos, tendo sido, neste caso, começada a sua definição pela seguinte sequência de vértices que permite a sua visualização desde o topo:

1. Primeiro Triângulo: P4, P2, P3
2. Segundo Triângulo: P2, P1, P3

Para finalizar a construção desta face, necessitamos de realizar NxN iterações, nas quais vamos definindo cada subdivisão da face, sendo as coordenadas dos vértices de cada subdivisão calculadas pelas seguintes expressões:

$$\text{P1} = \left(-\frac{\text{len}}{2} + \text{sbLen} * j, \frac{\text{len}}{2}, -\frac{\text{len}}{2} + \text{sbLen} * (i+1)\right)$$

$$\text{P2} = \left(-\frac{\text{len}}{2} + \text{sbLen} * j, \frac{\text{len}}{2}, -\frac{\text{len}}{2} + \text{sbLen} * i\right)$$

$$\text{P3} = \left(-\frac{\text{len}}{2} + \text{sbLen} * (j+1), \frac{\text{len}}{2}, -\frac{\text{len}}{2} + \text{sbLen} * (i+1)\right)$$

$$\text{P4} = \left(-\frac{\text{len}}{2} + \text{sbLen} * (j+1), \frac{\text{len}}{2}, -\frac{\text{len}}{2} + \text{sbLen} * i\right)$$

$$\text{len equivale ao tamanho de cada lado}$$

$$\text{dv equivale ao número de divisões}$$

$$\text{sbLen equivale ao tamanho de cada subdivisão } \left(\frac{\text{len}}{\text{dv}}\right)$$

$$\forall i, j \in \{0, 1, ..., \text{dv} - 1\}$$

A face inferior da caixa é definida de forma análoga, diferindo apenas na ordem dos vértices devido à orientação da normal, que aponta para baixo. Assim sendo, esta face encontra-se no plano y = -len/2, cujos triângulos de cada subdivisão são gerados da seguinte forma:

1. Primeiro Triângulo: P4, P3, P2
2. Segundo Triângulo: P2, P3, P1

Para a definição das faces laterais da caixa, aplica-se um processo iterativo semelhante. Cada uma destas faces mantém uma coordenada constante, seja no eixo X ou Z, enquanto as outras duas vão tendo o seu valor modificado entre iterações. Desta forma, as quatro faces laterais da caixa são definidas nos seguintes planos:

1. Face esquerda: $Z = \frac{\text{len}}{2}$;
2. Face frontal: $X = \frac{\text{len}}{2}$;
3. Face direita: $Z = -\frac{\text{len}}{2}$;
4. Face traseira: $X = -\frac{\text{len}}{2}$.

2.1.3. Esfera

Para construirmos a esfera, necessitamos de receber informações relevantes, tais como o raio, o número de *slices* (ou *sectors*) e *stacks*. Os *slices* representam as divisões verticais da esfera, semelhantes aos meridianos de um globo, enquanto os *stacks* correspondem às divisões horizontais, semelhantes às linhas de latitude. Cada *slice* é dividido por um número de *stacks*, e quanto maior for o número dessas subdivisões, mais suave será a aparência da esfera.

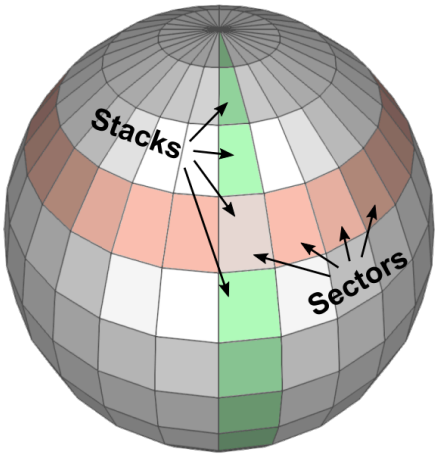

Figura 3: Esquema da subdivisão da esfera.

Após isto, necessitamos de calcular as coordenadas dos vértices dos triângulos. Tendo em vista este objetivo, começamos por determinar as coordenadas esféricas, para depois poderem ser convertidas em coordenadas cartesianas.

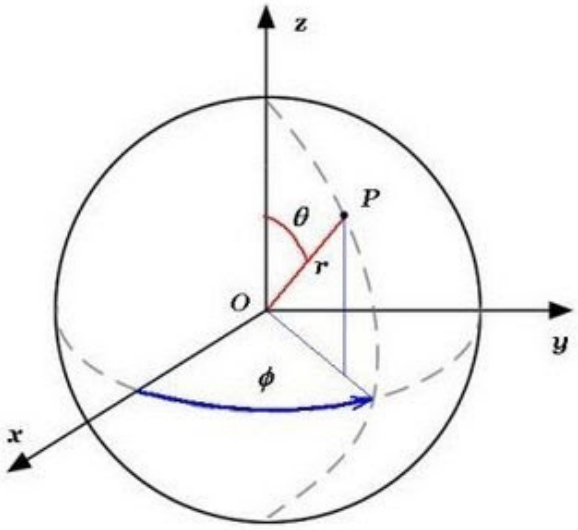

Figura 4: Coodernadas esféricas.

$$\varphi = \frac{\text{SliceAtual} * \pi}{\text{Slices}}$$
$$\theta = \frac{\text{StackAtual} * 2\pi}{\text{Stacks}}$$

Tendo calculado as coordenadas polares, é possível convertê-las em coordenadas cartesianas através das seguintes fórmulas:

$$x = r * \sin(\theta) * \cos(\varphi)$$
$$y = r * \sin(\theta) * \sin(\varphi)$$
$$z = r * \cos(\theta)$$

Utilizando estas fórmulas, conseguimos calcular a posição de cada vértice necessário para desenhar a esfera. Para simplificar o processo, em cada parte formada por uma *stack* e um *slice*, calculamos quatro pontos (dois na *stack* atual e dois na próxima). Com esses pontos, desenhamos dois triângulos que juntos formam a superfície da esfera. Após isto, é possível gerar esferas, como na **Figura 5**, de raio 2, 20 *slices* e 20 *stacks*.

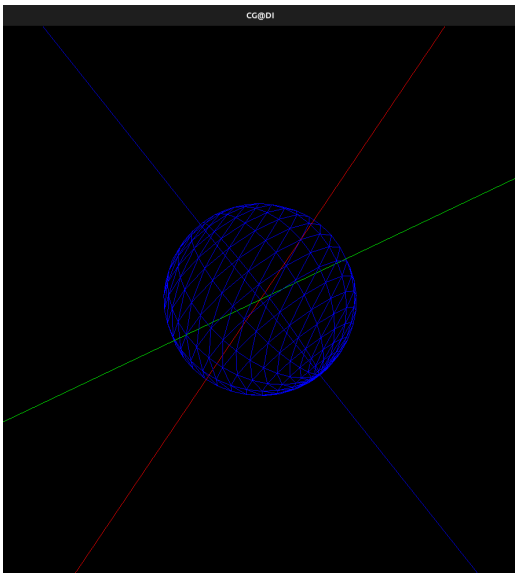

Figura 5: Esfera gerada pelo gerador.

2.1.4. Cone

Para gerar o cone, tal como ocorreu com as outras formas, foi necessário definir, em primeiro lugar, os parâmetros essenciais para o seu desenho. Após uma breve pesquisa, decidiu-se que esses parâmetros seriam: o raio da base, a altura, o número de *slices* (fatias) e o número de *stacks* (níveis). Com essas informações, foi possível avançar para a fase de cálculo, que permitiu determinar as coordenadas exatas de cada vértice.

O primeiro passo consistiu em calcular o ângulo **alpha** (α), ilustrado na **Figura 6**, e a altura de cada *stack*. Para isso, dividiu-se o ângulo total (360 graus) pelo número de *slices* e a altura total pelo número de *stacks*.

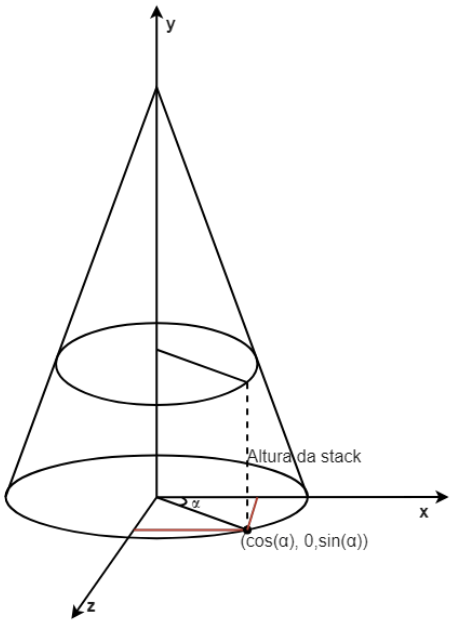

Figura 6: Cálculo dos vértices do cone

As fórmulas utilizadas para esses cálculos são as seguintes:

$$\alpha = \frac{360}{\text{slices}}$$

$$\text{alturaDaStack} = \frac{\text{alturaTotal}}{\text{NumeroStacks}}$$

Após esses cálculos iniciais, procedeu-se à criação de um ciclo que itera sobre cada *slice* e, dentro deste, sobre cada *stack*. Dessa forma, foi possível definir todos os vértices presentes em cada *slice*. Durante esse processo, verificou-se que a coordenada **y** de cada vértice varia em função da altura da *stack* atual, enquanto as coordenadas **x** e **z** podem ser calculadas com base no vértice da base, multiplicado por um fator que depende da *stack* atual e do número total de *stacks*. Para melhor compreensão, seguem-se as fórmulas utilizadas:

Para os vértices da base, as coordenadas são definidas da seguinte forma:

$$x = \text{raio} * \cos(\alpha)$$

$$z = \text{raio} * \sin(\alpha)$$

$$y = 0$$

Para os vértices ao longo das *stacks*, as coordenadas são ajustadas da seguinte maneira:

$$x' = x * \left(1 - \frac{\text{stackAtual}}{\text{NumeroStacks}}\right)$$

$$z' = z * \left(1 - \frac{\text{stackAtual}}{\text{NumeroStacks}}\right)$$

$$y' = j * \text{alturaDaStack}$$

O resultado final desse processo está ilustrado na figura abaixo:

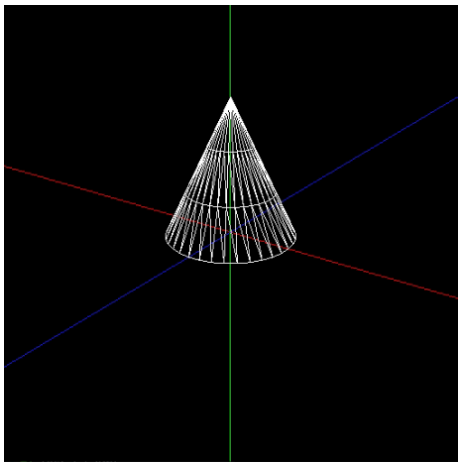

Figura 7: Cone gerado pelo gerador

2.2. Primitivas Extra

2.2.1. Cilindro

Olhando agora para as primitivas que foram acrescentadas ao projeto, começou-se por definir o cilindro. Este foi apresentado pelos docentes em contexto académico e, por isso, a sua implementação foi rápida e direta. Mesmo tendo isso em conta, as etapas seguidas foram as seguintes: perceber os parâmetros fornecidos, relacioná-los com as fórmulas matemáticas necessárias e aplicá-las em código para obter os vértices necessários à construção do cilindro.

Para implementar esta primitiva, aplicou-se a mesma lógica utilizada no cone: calcular os vértices da base inferior e, em seguida, os vértices da base superior. Como o cilindro mantém o mesmo diâmetro ao longo da sua altura, não é necessário realizar cálculos adicionais além do valor do ângulo **alpha**. Na **Figura 8**, podemos observar como calcular um ponto da base do cilindro. Esse cálculo, juntamente com o número de *slices*, permite determinar a localização exata de cada vértice. As fórmulas utilizadas foram as seguintes:

Para determinar o número exato de vértices nas bases, primeiro calculou-se o ângulo **alpha**:

$$\alpha = \frac{360}{\text{slices}}$$

Após esse cálculo, iterou-se pelo número de *slices*, aplicando as seguintes fórmulas para a base inferior:

$$x = \text{raio} * \cos(\alpha')$$
$$z = \text{raio} * \sin(\alpha')$$
$$y = 0$$

E para a base superior:

$$x = \text{raio} * \cos(\alpha')$$
$$z = \text{raio} * \sin(\alpha')$$
$$y = \text{altura}$$

Este processo assume que, a cada iteração, o valor do ângulo inicial é somado ao **alpha** atual, criando assim todos os pontos necessários.

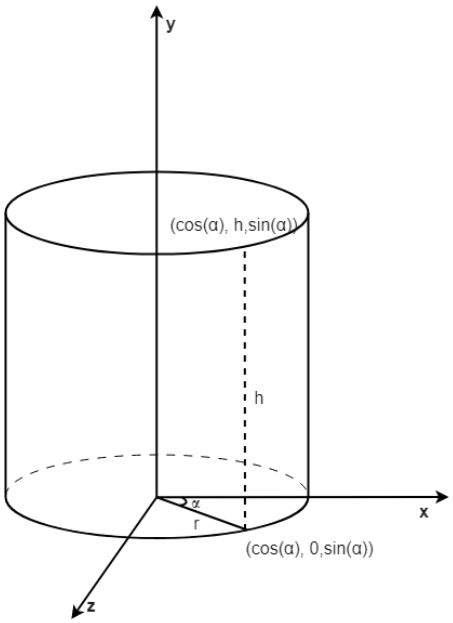

Figura 8: Cálculo dos vértices do cone

O resultado final desse processo está ilustrado na figura abaixo:

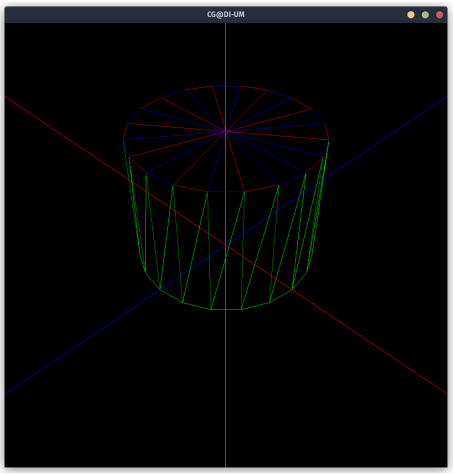

Figura 9: Cilindro gerado pelo gerador

2.2.2. Torus

O **torus** foi uma das primitivas adicionadas ao projeto para aumentar a variedade de formas geométricas disponíveis. A sua implementação baseou-se na definição de dois raios principais: o raio maior (*majorRadius*), que define a distância do centro do torus até o centro do "tubo", e o raio menor (*minorRadius*), que define o raio do próprio "tubo". Além disso, foram utilizados os parâmetros *slices* e *sides* para controlar o nível de detalhe da primitiva, representando o número de divisões ao redor do raio maior e do raio menor, respetivamente.

A lógica de implementação do torus envolveu a criação de vértices em dois ciclos aninhados: um para iterar sobre as *slices* (divisões ao redor do raio maior) e outro para iterar sobre as *sides* (divisões ao redor do raio menor). Para cada combinação de *slice* e *side*, foram calculados quatro pontos que formam dois triângulos, os quais, juntos, compõem uma pequena face do torus. As coordenadas desses pontos foram determinadas utilizando funções trigonométricas, como seno e cosseno, para garantir a forma circular do torus.

O cálculo dos vértices foi realizado da seguinte forma:

1. Para cada *slice* (índice `i`), calculou-se o ângulo **beta**:

$$\beta = \text{slicesJump} * i$$
$$\text{slicesJump} = (2 * \pi)\ \text{slices}$$

2. Para cada *side* (índice `j`), calculou-se o ângulo **alpha**:

$$\alpha = \text{sidesJump} * j$$
$$\text{sidesJump} = \frac{2 * \pi}{\text{sides}}$$

3. Com base nesses ângulos, calcularam-se as coordenadas dos quatro pontos que formam dois triângulos que foram posteriormente adicionados à lista de vértices do torus, permitindo a sua renderização correta. Para saber a localização exata dos pontos, as formulas que serviram de base foram as seguintes:

$$x = \text{majorRadius} + \text{minorRadius} * \cos(\alpha)) * \cos(\beta)$$
$$y = \text{minorRadius} * \sin(\alpha)$$
$$z = (\text{majorRadius} + \text{minorRadius} * \cos(\alpha)) * \sin(\beta)$$

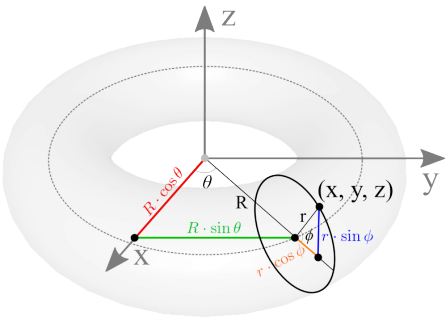

Figura 10: Cálculo dos vértices do torus

O resultado final desse processo está ilustrado na figura abaixo:

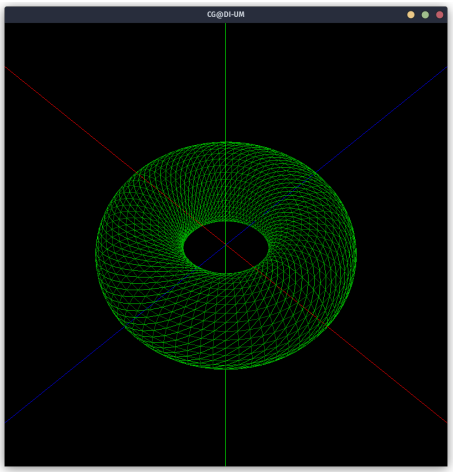

Figura 11: Torus gerado pelo gerador

2.2.3. flatRing

Utilizando o **torus** como base, e removendo unicamente o fator da altura/*sides* (que define o número de pontos ao longo da circunferência do "tubo"), foi possível desenhar um anel raso. Este anel pode ser útil em diversas implementações, como, por exemplo, a representação dos

anéis de Saturno numa simulação do sistema solar. Para esta implementação, utilizaram-se as fórmulas especificadas no torus, com as devidas alterações.

Para o círculo exterior, as coordenadas dos vértices são dadas por:

$$x = \text{majorRadius} + \text{minorRadius} * \cos(\alpha)$$
$$y = 0$$
$$z = \text{majorRadius} + \text{minorRadius} * \sin(\alpha)$$

Para o círculo interior, as coordenadas dos vértices são calculadas da seguinte forma:

$$x = \text{majorRadius} - \text{minorRadius} * \cos(\alpha)$$
$$y = 0$$
$$z = \text{majorRadius} - \text{minorRadius} * \sin(\alpha)$$

Neste caso, o ângulo `alpha` é calculado dentro de um ciclo, utilizando:

$$\alpha = \text{slicesJump} * i$$
$$\text{slicesJump} = \frac{2 * \pi}{\text{slices}}$$

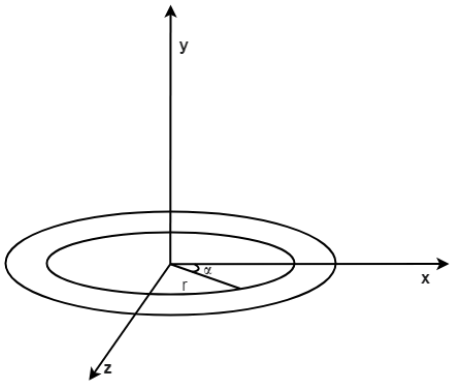

Figura 12: Cálculo dos vértices do flatRing

O resultado final desse processo está ilustrado na figura abaixo:

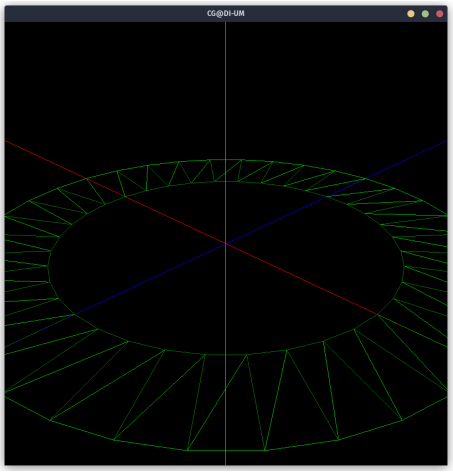

Figura 13: FlatRing gerado pelo gerador

2.2.4. Icosaédro/Icoesfera

O **icosaedro** é um poliedro regular composto por 20 faces triangulares, 30 arestas e 12 vértices. Para a geração da **icoesfera**, partimos do icosaedro e aplicamos um processo de

subdivisão recursiva das suas faces, seguido de uma normalização dos vértices para garantir que todos os pontos estejam sobre a superfície de uma esfera. Este processo permite criar uma esfera com um nível de detalhe controlado pelo número de subdivisões aplicadas. Os parâmetros utilizados para este gerador foram:

- **Raio**: Define o tamanho da esfera gerada. Todos os vértices são normalizados para estarem à distância do raio especificado em relação ao centro.
- **Subdivisões**: Controla o número de vezes que as faces do icosaedro são subdivididas. Cada subdivisão aumenta o número de faces e vértices, resultando em uma esfera mais suave e detalhada.

O processo de geração passa por três etapas principais, sendo elas a criação do Icosaedro, a subdivisão e a normalização. Quando analisado com mais detalhe o processo é o seguinte:

1. **Criação do Icosaedro Base**:
   - O icosaedro base é criado a partir de 12 vértices, que são definidos com base na **proporção áurea** (φ = (1 + $\sqrt{5}$) / 2). Estes vértices são posicionados de forma a garantir que o icosaedro seja regular.
   - As 20 faces do icosaedro são definidas através de índices que conectam os vértices. A utilização de índices facilita a organização dos vértices, especialmente durante o processo de subdivisão, onde a ordem correta dos vértices é crucial para a renderização adequada.
2. **Subdivisão das Faces**:
   - Cada face triangular do icosaedro é subdividida em quatro faces menores. Para isso, cada aresta da face é dividida ao meio, criando novos vértices no ponto médio de cada aresta.
   - Os novos vértices são então normalizados para garantir que estejam sobre a superfície de uma esfera com o raio especificado. A normalização é realizada através da seguinte fórmula:

$$v_{\text{normalizado}} = \left(\frac{v_x}{\text{length}}, \frac{v_y}{\text{length}}, \frac{v_z}{\text{length}} \right) * \text{raio}$$

Onde:
- (v_x, v_y, v_z) são as coordenadas do vértice.
- $(\text{length} = \sqrt{v_x^2 + v_y^2 + v_z^2}$ é a distância do vértice ao centro.

3. **Normalização dos Vértices**:
   - Após cada subdivisão, todos os vértices são normalizados para garantir que estejam sobre a superfície de uma esfera. Este passo é essencial para manter a forma esférica da icoesfera, independentemente do número de subdivisões aplicadas.
4. **Repetição do Processo**:
   - O processo de subdivisão e normalização é repetido um número de vezes igual ao número de subdivisões especificado. Cada subdivisão aumenta o número de faces e vértices, resultando em uma esfera mais suave e detalhada.

As figuras abaixo ilustram a icoesfera gerada pelo gerador para diferentes níveis de subdivisão:

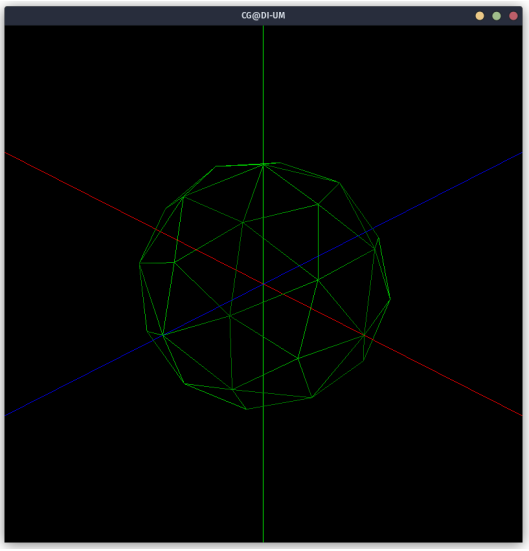

Figura 14: Icoesfera gerada com 1 subdivisão

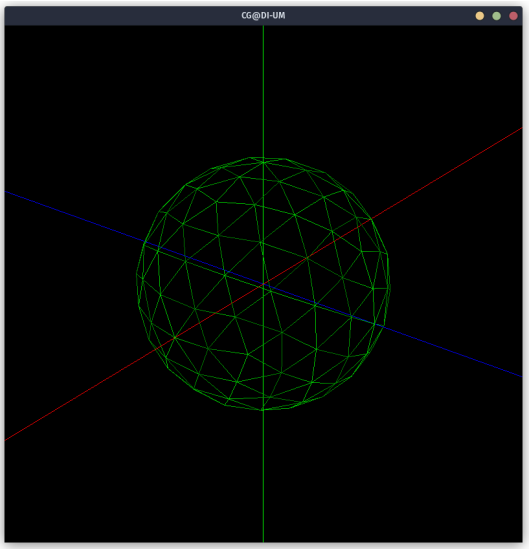

Figura 15: Icoesfera gerada com 2 subdivisões

Como pode ser observado, à medida que o número de subdivisões aumenta, a icoesfera torna-se mais suave e aproxima-se cada vez mais de uma esfera perfeita.

2.3. Armazenamento e Estruturação de Dados

Após o desenvolvimento dos geradores de primitivas, foi necessário definir uma forma eficiente de armazenar a informação gerada. Para isso, optou-se por utilizar ficheiros **binários**, que oferecem vantagens significativas em termos de velocidade de leitura e escrita, especialmente quando comparados com ficheiros de texto. A estrutura escolhida para estes ficheiros foi a seguinte:

- **1 byte** para a versão do ficheiro, permitindo futuras atualizações e compatibilidade.
- **1 byte** para o tipo de geometria, permitindo que a *engine* identifique a primitiva.
- **2 bytes** para o número de triângulos presentes na figura, o que também permite calcular o número de vértices, caso necessário.
- **12 bytes** para cada vértice, uma vez que cada vértice é composto por três coordenadas (x, y, z), sendo cada uma representada por um valor do tipo `float` (4 bytes).

Esta estrutura permite uma leitura rápida e direta dos dados, essencial para a renderização eficiente das cenas 3D. Além disso, a utilização de ficheiros binários reduz também o tamanho dos ficheiros em comparação com formatos de texto, o que é particularmente útil em projetos com um grande número de primitivas.

Os ficheiros são posteriormente guardados numa pasta designada `models`, com a extensão `.3d`, para facilitar a sua identificação e posterior carregamento pela *engine*.

3. Engine

Após o desenvolvimento dos geradores, começou-se por implementar uma versão rudimentar da *engine*, na qual se fez um carregamento direto de cada modelo para se poder verificar a consistência dos mesmos. Depois desta primeira versão, começou-se por desconstruir as várias etapas necessárias para cumprir os requisitos estabelecidos para esta parte do projeto, passando estes por ser a leitura de um ficheiro XML contento os modelos a ser carregados, juntamente com algumas definições da câmara. Para tal, criaram-se várias classes que permitem gerir estes mesmos objetos, sendo nas secções seguintes descrito todo o desenvolvimento por detrás de cada uma destas fases. O núcleo do *engine* é a classe ***Scene***, que detém o controlo sobre o estado do mundo em cada momento, e é responsável por diversos comportamentos, nomeadamente a captura de *input* por parte do utilizador, a organização dos objetos presentes em cena e a sua renderização, e o carregamento e processamento dos modelos de cada objeto.

3.1. Parsing do XML

O parsing do XML foi realizado com assistência da biblioteca **TinyXML2**, de forma a facilitar a leitura e tratamento de cada campo presente nas várias *tags* do ficheiro XML, proporcionando ao grupo mais tempo para tratar de primitivas geométricas, bem como da própria *engine*. Esta biblioteca é utilizada pelas classes ***Scene***, ***Window***, ***Camera***, ***Group*** e ***Model***, para a leitura dos parâmetros necessários á inicialização das instâncias de cada uma. A classe ***Scene*** é a única classe que lê um ficheiro XML, ao passo que as restantes operam exclusivamente sobre o documento carregado e passado pela ***Scene*** á fábrica de cada uma.

3.2. Carregar os modelos

Na inicialização de cada objeto ***Model***, é utilizado um método construtor para ler do documento XML os parâmetros necessários á sua inicialização, nomeadamente o identificador do ficheiro contendo a geometria associada a esse modelo. Após a leitura dos mesmos, o modelo tenta então obter a geometria associada ao seu identificador a partir de uma *cache* global, partilhada entre todos os modelos de igual identificador de modo a reduzir a memória utilizada. Caso a geometria associada ao modelo ainda não se encontre carregada, o ficheiro de geometria associado será carregado para a memória e é retornado um ponteiro para a geometria.

3.3. Exploração da cena

Após a extração das informações da câmara pelo objeto ***Scene***, é criado um objeto responsável por controlar todos os valores associados à mesma. Esses valores incluem informação

como a sua posição no espaço 3D, o vetor que define a sua orientação, o seu campo de visão (FOV) e os modos de controlo disponíveis. Atualmente, a câmara pode operar em três modos distintos:

- **Exploração (EX)**, que se encontra completamente funcional;
- **First Person (FP)**, que se encontra parcialmente implementado, estando em falta os movimentos de translação e a correção da sua rotação;
- **Third Person (TP)**, que se encontra em fase inicial de implementação.

3.3.1. Modo Exploração (EX)

No modo **Exploração**, a câmara é posicionada a uma distância fixa do centro da cena, permitindo que o utilizador circunde o objeto nos eixos horizontal e vertical. Esta implementação foi realizada utilizando coordenadas polares, onde a distância da câmara ao centro é tratada como o raio. O ângulo **beta** (denominado *yaw* internamente) controla a rotação horizontal, e um ângulo **alfa** (denominado *pitch* internamente) controla a rotação vertical.

3.3.2. Modo First Person (FP)

O modo **First Person** assemelha-se ao utilizado em jogos do género *first-person shooter* (FPS), onde o utilizador possuí cinco graus de liberdade no seu movimento, caracterizado por três envelopes de translação (***surge:*** frente, trás; ***heave:*** cima, baixo; ***sway:*** esquerda, direita) e dois envelopes de rotação (***pitch:*** cima, baixo; ***yaw:*** esquerda, direita). Atualmente encontram-se implementados os envelopes de rotação e encontram-se em falta os envelopes de translação.

3.3.3. Modo Third Person (TP)

O modo **Third Person** é uma fusão do **Modo Exploração**, na medida que todos os controlos associados ao mesmo são virtualmente idênticos, com o modo **First Person**, através da adição dos envelopes de translação associados ao mesmo (***surge:*** frente, trás; ***heave:*** cima, baixo; ***sway:*** esquerda, direita). Atualmente encontra-se em fase inicial de implementação e ainda não se apresenta funcional.

4. Scripts e Debug

Para agilizar o processo de desenvolvimento e garantir uma construção (*build*) mais eficiente do projeto, foram criados scripts que automatizam tarefas comuns, como a configuração do ambiente, a compilação do código e a limpeza dos ficheiros gerados. Estes scripts, `setup.sh`, `build.sh` e `clean.sh`, permitem gerir o ciclo de construção do projeto de forma rápida e consistente, fornecendo também uma opção de *debug* que disponibiliza informação mais detalhada sobre o programa através da alteração do ambiente de desenvolvimento. Esta alteração é realizada adicionando a *flag* ***DEBUG_MODE*** ao processo de compilação, permitindo a adição de informação relevante apenas durante o processo de desenvolvimento, podendo ser excluída do artefacto final.

A utilização destes scripts trouxe várias vantagens:

- **Eficiência**: Reduz o tempo necessário para configurar, compilar e limpar o projeto.
- **Consistência**: Garante que todos os membros da equipa utilizam o mesmo processo de construção.
- **Facilidade na Deteção de Erros**: A build de debug fornece ferramentas poderosas para identificar e corrigir erros de forma mais rápida e precisa.

5. Conclusão e trabalho futuro

Para concluir esta primeira fase do projeto, o grupo ficou satisfeito com os diversos componentes desenvolvidos, embora alguns elementos adicionais, como o carregamento de ficheiros *.obj* e a criação de mais algumas primitivas geométricas, não tenham sido incluídos nesta parte. Vale ressaltar que todas as decisões tomadas pelo grupo foram orientadas no sentido de otimizar o desempenho computacional, garantindo também a melhor experiência possível para o utilizador. Antes de terminar, certas componentes extra, tais como o carregamento de ficheiros *.obj*, a câmara ***First-Person View*** e a alteração de modos de visualização através de um menu já possuem toda a sua base desenvolvida, apenas restando fazer algumas alterações para serem lançadas no projeto. Assim sendo, consideramos que esta primeira fase está robusta, facilitando a implementação das próximas etapas do projeto, bem como a adição de mais componentes extra.