Março 05, 2024

Relatório CG - Fase 1 - Grupo 24

Guilherme Barbosa (a100695)
Ivan Ribeiro (a100538)
Pedro Ferreira (a100709)
Nuno Aguiar (a100480)

Sumário

1) Introdução

Neste documento é descrito o processo de desenvolvimento da primeira fase do projeto da Unidade Curricular de Computação Gráfica. O tema abordado é um motor de renderização de gráficos 3D. Para demonstrar o potencial do motor desenvolvido, foram desenvolvidas pequenas cenas. Os objetos que participam nestas cenas são obtidos a partir de primitivas fornecidas por um programa Gerador.

2) Fase 1 - Primitivas gráficas

A primeira fase cobre as fases iniciais de desenvolvimento de duas aplicações: um gerador de primitivas, denominado Generator, e um motor de renderização 3D, com o nome de Engine. O Generator cria ficheiros com a informação necessária à renderização de primitivas - nesta fase, o grupo adotou a abordagem de apenas guardar em ficheiro os vértices das primitivas. Por sua vez, a Engine - principal foco do projeto - dado um ficheiro de configuração em XML, configura parâmetros da janela e câmara, e renderiza primitivas a partir de ficheiros criados pelo Generator.

2.1) Generator

O Generator, responsável pela criação dos diversos objetos, suporta as várias primitivas gráficas que constavam no enunciado - plano, cubo, cone e esfera - e, além destas, permite a criação de cilindros e torus. O cilindro surgiu da ideia de uma possível renderização de caudas de cometas, enquanto que o toro poderá ser usado para representar os anéis de Saturno e Júpiter na cena do sistema solar, além do Cinturão de Asteróides.

Cada uma das primitivas possui um processo diferente de criação dos seus vértices. No entanto, todas coincidem no facto de que, como resultado final, apenas fornecem os pontos que constituem os triângulos necessários à renderização do objeto pretendido.

2.1.1) Plano

O plano é a única primitiva de duas dimensões. Localiza-se sobre o plano XZ, centrado na origem do referencial, e é definido por um tamanho e número de divisões. Para o construir, começamos por definir o salto entre cada dois pontos: distância = tamanho / divisões ("step" na figura 1). Com base nessa distância são gerados todos os pontos necessários para a estrutura, ao mesmo tempo que centramos o plano na origem, aplicando a transformação (*-length/2*, 0, *-length/2*) sobre todos os pontos criados. Para gerar os vértices da primitiva, constatamos que cada 2 pontos consecutivos de 2 linhas diferentes formam um quadrado que pode ser dividido em dois triângulos (representados com as setas vermelhas e amarelas na figura 1). Para a construção dos triângulos, os quais devem ficar com a face voltada para cima ($y > 0$), recorreu-se à regra da mão direita para definir a ordem em que os vértices dos triângulos devem ser desenhados. O pseudo-código de geração da primitiva é apresentado de seguida:

```
func plane (length, divisions) {
  points = [] // pontos intermédios que a estrutura utiliza
  triangles = [] // pontos finais que constituem a primitiva

  step = length / divisions
  shift = length / 2 // translação para centrar pontos de plano

  for col in divisions:
    for line in divisions:
      x = col * step - shift
      y = 0
```

```
      z = line * step - shift
      points_add(x, y, z)

  for columns in divisions:
    for point in columns:
        if point has (points_below and points_right):
          triangles_add(point, point_down, point_right_down)
          triangles_add(point, point_right_down, point_right)

  return triangles
}
```

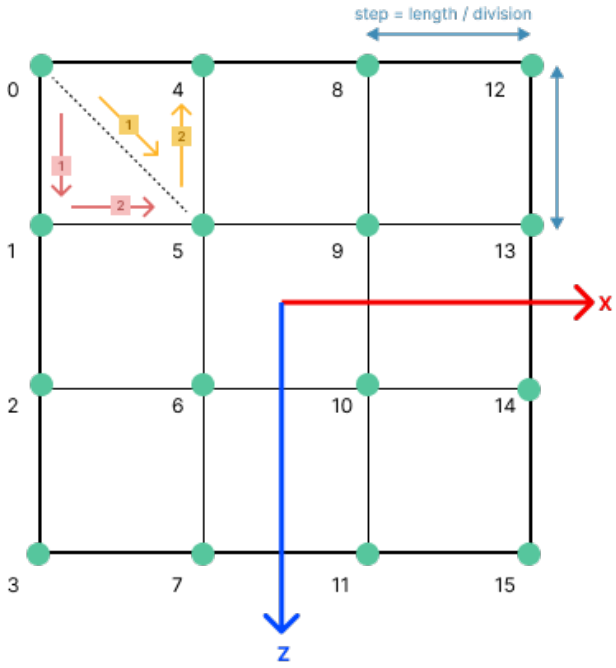
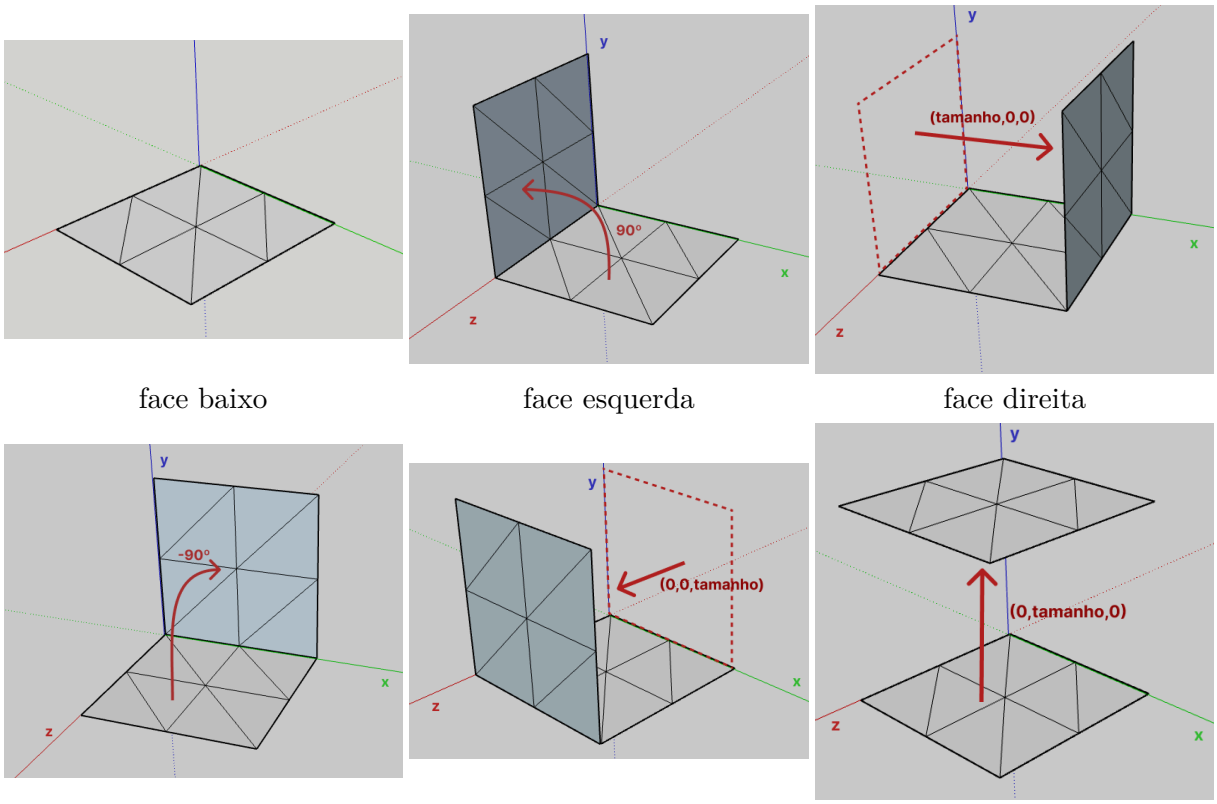


Figura 1: Construção do plano

2.1.2) Caixa

A primitiva da caixa é centrada na origem e, tal como o plano, é necessário fornecer o tamanho de cada aresta e o número de divisões nesta. A construção desta primitiva requer a implementação prévia do plano, pois envolve a aplicação de rotações e translações sobre o mesmo (conforme descrito na figura 2), sendo as dimensões de cada plano as indicadas como argumento. O plano de baixo não requer transformações ou rotações sobre o plano original. O plano da esquerda requer uma rotação de 90º sobre o eixo Z, enquanto o da direita requer a mesma rotação seguida da translação (*tamanho*, *0*, *0*). O plano de trás requer um rotação de $-90º$ sobre o eixo X, ao passo que o da frente requer a mesma rotação seguida da translação (*0*,*0*,*tamanho*). Por fim, o plano de cima requer a translação (*0*,*tamanho*,*0*) sobre o plano original. Note-se que, para representar alguns dos planos do cubo com as faces dos triângulos orientadas corretamente, foi necessário partir de uma primitiva plano com os triângulos virados para $y < 0$, de modo a, depois das rotações, estarem corretamente dispostos. Por fim, para centrar o cubo na origem, aplicamos a translação (*-length/2*, *-length/2*, *-length/2*) conforme os pontos são criados. O pseudo-código de geração da primitiva é apresentado de seguida:

```
func box (length, divisions) {
  triangles = [] // array that accumulates resulting vertices

  planeFacingUp =[vertex] // plane with triangles facing upwards, points are all x>=0,
y=0, z>=0

  planeFacingDown = reverse of planeFacingUp // plane with triangles facing downwards,
points are all x>=0, y=0, z>=0

  shiftVector = (length/2, length/2, length/2) // vector to center cube

  translateVector_x = (length, 0, 0)
  translateVector_y = (0, length, 0)
  translateVector_z = (0, 0, length)

  // Down face
    downVertices = planeFacingDown - shiftVector
    triangles_add(downVertices)

  // Left face
    leftVertices = rotate_z_matrix(90) * planeFacingUp - shiftVector
    triangles_add(leftVertices)

  // Right face
    rightVertices = rotate_z_matrix(90) * planeFacingDown -
      shiftVector + translateVector_x
    triangles_add(rightVertices)

  // Back face
    backVertices = rotate_x_matrix(-90) * planeFacingUp - shiftVector
    triangles_add(backVertices)

  // Front face
    translateVector = (0,0,length)
    frontVertices = rotate_x_matrix(-90) * planeFacingDown -
      shiftVector + translateVector_z
    triangles_add(frontVertices)

    // Up face
    translateVector = (0,length,0)
    upVertices =  planeFacingUp - shiftVector + translateVector_y
    triangles_add(upVertices)

  return triangles
}
```



face baixo face esquerda face direita

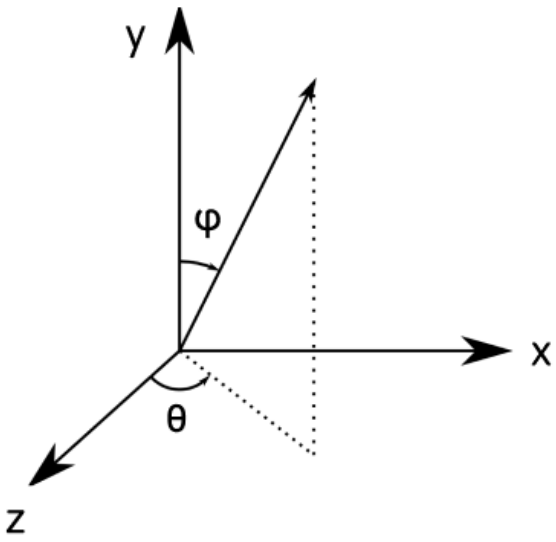
face trás face frente face cima

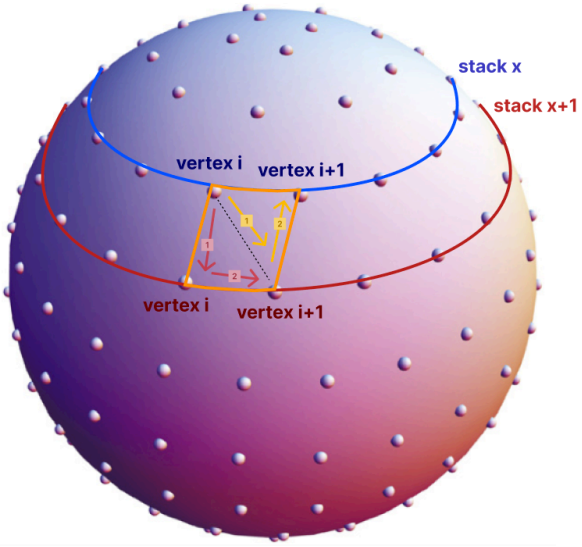
Figura 2: Passos de construção das faces do cubo

2.1.3) Esfera

O centro da esfera localiza-se na origem e define-se com um raio, número de fatias e de pilhas. Para construir esta primitiva, utilizámos coordenadas esféricas. Começamos por definir o ângulo de cada tira vertical e camada horizontal. O valor de phi varia entre 90^o e -90^o, logo o salto de cada camada horizontal será pi / pilhas, e o de theta varia entre 0^o e 360^o, logo o salto entre tiras verticais será 2pi / fatias. Iterando sobre os vários ângulos de phi (camadas horizontais) e theta (tiras verticais), obtemos todos os pontos, de todas as camadas, que constituem a esfera.

Entre cada dois pontos consecutivos de duas camadas consecutivas, criam-se dois triângulos de forma idêntica à descrita no plano, obtendo-se todos os vértices necessários (conforme descrito na figura 3). Excecionalmente, no código, os extremos superior e inferior da esfera são conectados ao resto da estrutura separadamente, com o intuito de evitar que fossem criados pontos extremamente próximos, porém diferentes, nos polos, resultantes de erros de aproximação nos cálculos efetuados com pontos flutuantes, o que causava defeitos visuais na primitiva.





Esquema de coordenadas esféricas Construção de quadrado lateral da esfera

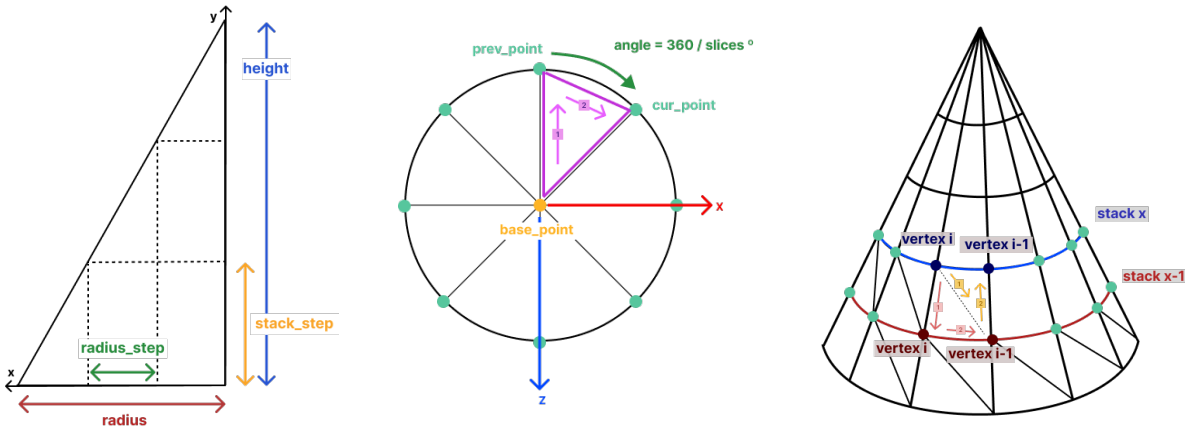
Figura 3: Passos de construção das faces da esfera

```
func sphere (radius, slices, stacks) {
  points = [] // pontos intermédios que a estrutura utiliza
  triangles = [] // pontos finais que constituem a estrutura

  sector_step = 2PI / slices
  stack_step = PI / stacks

  // obter pontos que constituem a esfera
  for stack in stacks:
    curr_phi = PI / 2 - stack_step * stack
    y = radius * sin(curr_phi)

    for slice in slices:
    curr_theta = sector_step * slice
    z = radius * cos(curr_phi) * cos(curr_theta)
    x = radius * cos(curr_phi) * sin(curr_theta)

    // construir os triângulos
    for slices in stacks:
      for point in slices:
        if point has (points_below and points_right):
        triangles_add(point, point_down, point_right_down)
        triangles_add(point, point_right_down, point_right)

  return triangles
}
```

2.1.4) Cone

O cone é centrado no eixo Y, com base sobre o plano XZ, e define-se com um raio, altura, número de fatias e pilhas. Para construir esta primitiva, começamos por definir o ângulo alfa da base e a diferença entre raios de camadas horizontais consecutivas, além da distância vertical entre diferentes camadas (conforme descrito na 1ª imagem da figura 4). Em cada camada, após estabelecer um ponto base, aplica-se um rotação de ângulo alfa (em torno do eixo Y) ao ponto anteriormente calculado, obtendo assim o próximo ponto da camada. Se a camada em questão for a base do cone, então une-se o ponto atual, o anterior e o ponto base para desenhar os

triângulos da mesma (conforme descrito na 2ª imagem da figura 1). Caso contrário, usa-se o ponto atual, o ponto anterior, e os seus equivalentes na camada imediatamente abaixo para criar a lateral do cone (conforme descrito na 3ª imagem da figura 1). Por fim, para criar o seu topo, considera-se apenas um único ponto na camada mais alta e une-se o mesmo com os pontos da penúltima camada, formando os triângulos do topo.



Medidas do cone | Construção da base | Construção de quadrado lateral do cone

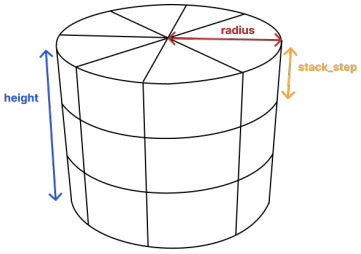
Figura 4: Passos de construção do cone

```
func cone (radius, height, slices, stacks) {
  // ângulo entre pontos da base adjacentes
  angle = 360 / slices

  // diferença de altura entre camadas horizontais
  stack_step = height / stacks

  // diminuição de raio entre camadas horizontais
  radius_step = radius / stacks

  triangles = []

  for stack in stacks:
    prev_point = (radius - stack * radius_step, stack * stack_step, 0)

    for slice in slices:
      cur_point = rotate_y_matrix(angle) * prev_point

      // adicionar triângulos que vão constituir a base
      if stack == base_stack:
        triangles_add(base_point, prev_point, cur_point) // base_point = (0, 0, 0)

      // adicionar triângulos que constituem a lateral
      else:
        triangles_add(right_down_point, right_point, cur_point)

        triangles_add(cur_point, right_point, down_point)

      cur_point = prev_point

  // adicionar triângulos que vão constituir o chapéu do cone
```

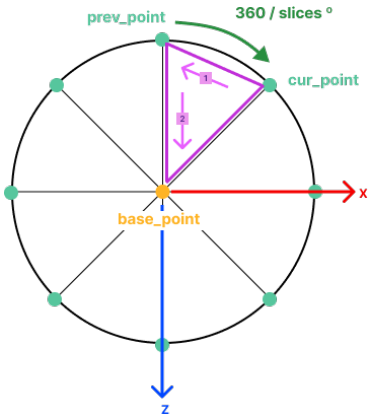
```
  highest_point = (0, height, 0)

  for cur_point in last_stack_points:
    triangles_add(cur_point, highest_point, left_point)

  return triangles
}
```

2.1.5) Cilindro

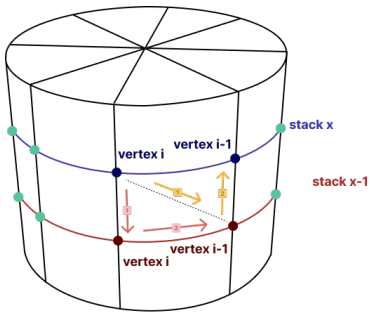
O cilindro é centrado na origem do referencial e define-se com um raio, altura, número de fatias e pilhas. Para construir esta primitiva, o processo é idêntico ao do cone, com a exceção da construção da base superior. Ao invés de ser constituída por um único ponto, esta base é formada de forma semelhante à base inferior.



Medidas do cilindro



Construção da base superior



Construção de quadrado lateral do cilindro

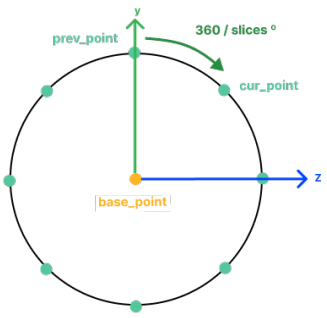
Figura 5: Passos de construção do cone

```
func cylinder (radius, height, slices, stacks) {
  // ângulo entre pontos da base adjacentes
  angle = 360 / slices

  // diferença de altura entre camadas horizontais
  stack_step = height / stacks

  triangles = []

  for stack in stacks:
    prev_point = (radius, stack * stack_step, 0)

    for slice in slices:
      cur_point = rotate_y_matrix(angle) * prev_point

      // adicionar triângulos que vão constituir a base inferior
      if stack == first_stack:
        triangles_add(base_point, prev_point, cur_point) // base_point = (0, 0, 0)

      // adicionar triângulos que vão constituir a base superior
      if stack == last_stack:
        triangles_add(cur_point, prev_point, base_point) // base_point = (0, height,
0)

      // adicionar triângulos que constituem a lateral
```

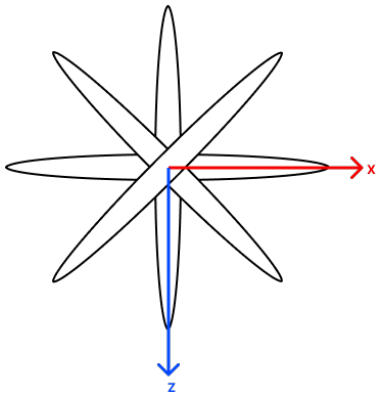
```
      if stack != first_stack:
        triangles_add(right_down_point, right_point, cur_point)

        triangles_add(cur_point, right_point, down_point)

      cur_point = prev_point

  return triangles
}
```

2.1.6) Torus

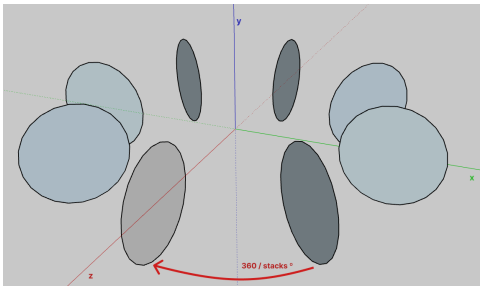
O torus é centrado na origem do referencial e é definido por raios interno e externo, número de fatias e pilhas. Começamos por definir uma circunferência no plano XZ com raio = *(Raio Exterior - Raio Interior)/2* e com número de pontos igual ao número de fatias. Esta circunferência, posteriormente, sofre uma rotação de 90º sobre o eixo Z, para constar no plano YZ (conforme descrito na 1ª imagem da figura 6). Depois desta rotação, a circunferência é replicada tantas vezes quanto o número das pilhas e cada uma das circunferências criadas é sujeita a uma rotação no eixo Y, com ângulo mútiplo de 360/pilhas (conforme descrito na 2ª imagem da figura 6). De seguida, cada uma das circunferências sofre uma translação, sendo assim colocada na distância e orientação pretendidas para definir o Torus (conforme descrito na 3ª imagem da figura 6). Por fim, são criados os triângulos que conectam as diversas circunferências já posicionadas corretamente (conforme descrito na 4ª imagem da figura 6).



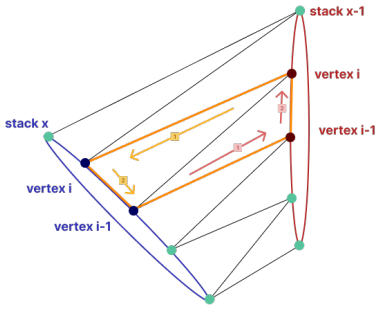
Circunferência base



Visão superior pós-replicação da circunferência

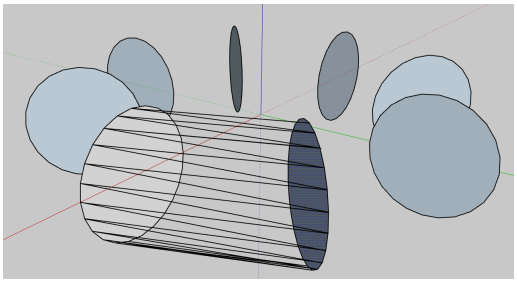


Posicionamento das várias circunferências

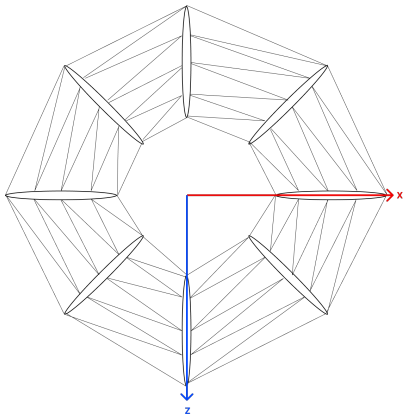


Construção de quadrado entre circunferências

Figura 6: Passos de construção do toro



Vista 3D da construção de quadrados



Vista áerea da forma final do toro

Figura 7: Passos de construção do toro

```
func Torus(raioInterior, raioExterior, slices, stacks) {

    // raio de cada circunferência que constitui o Toro
    float raioTorus = (raioExterior - raioInterior) / 2

    // distância horizontal entre origem e centro de cada circunferência na posição
final
    float distPonto = raioInterior + raioTorus

    // ângulo entre as circunferências que constituem o Torus
    float angleSlices = 360 / slices

    // ângulo entre os pontos de cada circunferência
    float angleStacks = 360 / stacks

    circunferenciaXZ = [] // pontos de circunferência no plano XZ centrada na origem
    circunferenciaYZ = [] // pontos de circunferência no plano YZ centrada na origem
    fatiasTorus = [] // pontos de todas as circunferências na posição final
    triangles = []

    // pontos da circunferência no plano XZ
    prev_point = (raioTorus, 0, 0)
    for slice in slices:
        curr_point = rotate_y_matrix(angleSlices) * prev_point
        prev_point = curr_point
        circunferenciaXZ_add(curr_point)

    // pontos da circunferência no plano YZ
    for point in circunferenciaXZ:
        curr_point = rotate_z_matrix(90) * point
        circunferenciaYZ_add(curr_point)

    // colocar as circunferências na posição correta
    for stack in stacks:

        for curr_point in circunferenciaXZ:

            // Matriz de translação que move o ponto para as coordenadas finais
            x = distPonto * sin(angleStacks * i)
            y = 0
            z = distPonto * cos(angleStacks * i)
```

```
            // Mover ponto para a posição correta e com a rotação pretendida
             posicaoFinal = translate_matrix(x,y,z) * rotate_y_matrix(angleStacks) *
curr_point

            fatiasTorus_add(posicaoFina)

    // construir os triângulos
    for slices in stacks:
      for point in slices:
        if point has (points_below and points_right):
           triangles_add(point, point_down, point_right_down)
           triangles_add(point, point_right_down, point_right)

    return triangles
}
```

2.2) Engine

O programa Engine é responsável por ler um ficheiro XML, recorrendo à biblioteca *TinyXML2*. A escolha desta biblioteca deriva do facto de ser eficiente, simples de utilizar e de fornecer todas as funcionalidades expectáveis, como o acesso aos diversos elementos e aos seus atributos. Com recurso a esta, obtêm-se diversos parâmetros de configuração da câmara e da janela de visualização, além de saber quais os ficheiros a ler para desenhar as estruturas pretendidas.

Para poder desenhar as diversas primitivas recorre-se ao *OpenGL* e à biblioteca *GLM*. Esta última facilita a utilização de matrizes, vetores, e operações tais como calcular matrizes de projeção. Também decidimos utilizar a biblioteca *GLFW*, por oposição ao *GLUT*, por sentirmos necessidade de uma API com maior controlo sobre diversos eventos, principalmente relacionados com inputs. Para além disso, é uma biblioteca mais consistente entre diversas plataformas e alguns membros já possuíam experiência prévia com a mesma.

De forma a facilitar a implementação deste programa, o mesmo foi dividido em módulos: o módulo XML Parser é o responsável por ler o XML; o módulo camera possui as funções necessárias ao movimento e posicionamento da mesma; o renderer possui as funções de callback requiridas para o desenho das figuras indicadas no XML; o módulo input handler é responsável por processar o input do utilizador, proveniente do teclado e rato. Cada um destes módulos é, de seguida, descrito com maior pormenor.

2.2.1) XML Parser

O módulo XML Parser recorre à biblioteca *TinyXML2*, já anteriormente referida, para ler o ficheiro XML passado como argumento e, a partir daí, fornecer as informações necessárias ao funcionamento dos restantes módulos. Estas informações, nesta fase, referem-se apenas às configurações da janela, como largura e altura, à posição da câmara e ao vetor look up, às informações relativas à perspetiva observada, nomeadamente os valores do fov, z_near e z_far, e os nomes dos ficheiros a ler, dos quais são obtidos os pontos necessários ao desenho de cada uma das primitivas.

O código necessário à leitura dos ficheiros pode ser abstraído como:

```
xml_doc = TinyXML2.read(xml_file)


// Base xml tags in each document
window_config = xml_doc.child_tag("window")
```

```
camera_config = xml_doc.child_tag("camera")
groups_config = xml_doc.child_tag("groups")

// Read model tags

engine_obj = Engine_Obj();

for model in groups:
  engine_obj.add(model.file.read_points())
```

2.2.2) Camera

O módulo Camera é baseada na implementação do learnopengl.com

São guardados vetores para posição, up, right, front, worldUp e ângulos pitch e yaw, permitindo facilmente implementar um sistema para mover a posição da câmara e olhar em volta da cena: para mover basta alterar a posição, para olhar basta alterar o pitch e o yaw, sendo depois os vetores necessários recalculados. A matriz resultante pode ser obtida através de GetViewMatrix().

O sistema de movimentos foi simplificado de modo a abstrair o gestor de inputs. Por exemplo, *ProcessKeyboard(FRONT, tempo)* irá mover a câmara para a frente. Para utilizar o ponto lookAt dos testes, foi criado um novo construtor, em que os ângulos de pitch e yaw são obtidos através de:

```
vetor = lookAtPoint - position;
vetor.normalize();

yaw = atan2(vetor.z, vetor.x);
pitch = atan2(vetor.y, sqrt((vetor.z * vetor.z) + (vetor.x * vetor.x)));
```

2.2.3) Input Handler

O módulo InputHandler processa qualquer callback de input e armazena o estado necessário para o processar. É guardada a última e a atual posição do rato e um array com o estado de todas as teclas, permitindo, por exemplo, verificar se estas estão a ser pressionadas ou não.

Como mencionado, a câmara tem os seus inputs abstraídos, pelo que será esta classe a responsável por processar os inputs "reais" e aplicá-los à câmara.

Implementamos os seguintes controlos:

- Mover rato: altera para onde a câmara está a olhar
- WASD: move a posição da câmara, mas apenas num eixo horizontal
- Espaco e Alt: move para cima e para baixo, num eixo vertical

NOTA: Usando o ESC, o cursor passa a ser separado da aplicação, e os callbacks são desativados (usar o ESC novamente reenquadra o cursor na aplicação e ativa novamente as callbacks)

2.2.4) Renderer

O módulo Renderer, neste momento, apenas possui a função de desenhar o conjunto de vértices das primitivas, bem como os eixos XYZ, tendo em conta a matriz fornecida pela câmera.

2.2.5) Loops

Implementados os módulos que processam inputs e renderizam a cena, tornou-se necessário decidir a estrutura em que se deve desenvolver cada frame. Para tal, existem dois objetivos que servem de orientação:

1. A renderização e os movimentos na cena devem ser fluidos, adaptando-se a qualquer refresh rate
2. Os cálculos, tais como as animações e as físicas, devem ser determinísticos, qualquer que seja o ritmo de renderização

Estes objetivos permitem que todos os elementos do grupo tenham exatamente os mesmos resultados usufruindo, se possível, das refresh rates mais altas.

Assim, Loops representa a agregação de dois ciclos que correm em threads diferentes, além de dois buffers que contêm vértices:

- Ciclo de cálculos: executa num timestep fixo (PHYS_STEP), por agora igual a 1/60. Altera os vértices consoante quaisquer cálculos e, após terminar, copia os mesmos para o buffer de desenho, onde poderão ser renderizados assincronamente.
- Ciclo de renderização: executa num timestep dinâmico, neste caso tenta manter a mesma refresh rate que o monitor através de glfwSwapInterval(1). Responsável por renderizar os vértices do buffer de desenho e por processar os inputs, aplicando-os à câmara.

Como os inputs são processados no ciclo de renderização, independentemente da velocidade das animações, podemos mover-nos livremente pela cena, observando sempre resultados determinísticos.

3) Testes

Com base nos ficheiro XML de teste fornecidos pelos docentes, verificámos que todas as configurações e primitivas se encontram corretamente renderizadas.

Para realizar alguns testes adicionais, como renderizar primitivas de maiores dimensões ou primitivas extra desenvolvidas pelo grupo, foram ainda desenvolvidos alguns ficheiros de teste. Este ficheiros constam na diretoria *custom_test_files_phase_1*.

4) Conclusão

Esta primeira fase do projeto permitiu, sobretudo, consolidar conceitos relativos ao desenho de vértices e à aplicação de transformações (rotações e translações) aos mesmos. Aprendemos também a integrar diferentes parâmetros de configuração da câmara e janela de visualização no OPEN_GL, suportando movimentos, redimensionamento de janelas, etc. Foi ainda possível reforçar conhecimentos transmitidos em aulas, nomedamente o desenvolvimento de primitivas. Por fim, esta primeira fase constituiu uma oportunidade de exploração de conceitos relativos ao processamento de inputs e refresh de frames, de forma a construir um motor de renderização fluido e consistente.

Como trabalho futuro, o grupo planeia, além do indicado no enunciado, explorar conceitos de shaders e implementar a iluminação e texturas nos mesmos. Planeamos ainda desenvolver uma câmara com perspetiva na 3ª pessoa e, se oportuno, desenvolver um primitiva de tubo.