Universidade do Minho - Escola de Engenharia

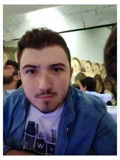
Mestrado Integrado em Engenharia Informática

Computação Gráfica

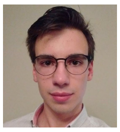
Relatório do Projeto Prático - Parte 1

Autores :
Diana Costa (A78985)

José Oliveira (A78806)



Marcos Pereira(A79116)



Vitor Castro(A77870)

23 de Março de 2018

Resumo

Perante a proposta de realizar duas aplicações que gerassem e interpretassem vértices para a criação de figuras, houve um impasse inicial devido à necessidade de uma boa estruturação dos problemas. Tudo isto requeriu a escolha acertada de algoritmos de geração de vértices e técnicas de parsing, de forma a que a resolução destes fosse a mais clara e simples possível.

Depois de algum tempo e trabalho, o resultado encontrado foi satisfatório, e os objetivos e respostas às questões do enunciado proposto cumpridos.

Conteúdo

1 Introdução

Este projeto surge no âmbito da unidade curricular de Computação Gráfica, do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Era requerido, como primeira fase, que se realizassem duas aplicações, um *Generator* e um *Engine*, com o objetivo de produzir graficamente um conjunto de figuras.

O *Generator*, através de determinadas características de uma figura e de primitivas gráficas, é responsável por gerar um ficheiro com os vértices dos triângulos necessários para formar a mesma.

O *Engine* tem como objetivo ler um ficheiro XML, que contém as referências para os documentos gerados pelo *Generator*, e dar como output os modelos pretendidos.

Em suma, a Secção 2 descreve a resolução da primeira parte do problema, com a resolução do **plano** na secção 2.2, da **caixa** em 2.3, do **cone** em 2.5 e da **esfera** na secção 2.4. Realizaram-se também duas figuras extras, o **cilindro** e a **pirâmide**, descritas nas secções 2.7 e 2.6. É feita uma breve consideração geral sobre todas as figuras geradas em 2.1, com o objetivo de melhor entender o problema e de evitar repetir informação.

Segue-se a explicação e raciocínio envolvente do Engine na secção 3, onde se explica, em primeiro lugar, o funcionamento da main (secção 3.1), do XML-Loader (secção 3.2), ao nível do formato do ficheiro .3d, XML e a leitura do último (secção 3.2.1, 3.2.2 e 3.2.3). Caracteriza-se também o Engine em si (secção 3.3), descrevendo as classes auxiliares na secção 3.3.1 e os métodos na 3.3.2. Termina-se esta secção com alguns testes que o grupo realizou, como forma de garantir a correta realização do projeto.

O relatório termina com uma breve conclusão na Secção 4, onde é feito um balanço do trabalho realizado, tendo em conta as dificuldades ao longo do desenvolvimento do mesmo. Também em "Anexos", na secção 5, apresentam-se os resultados finais de todas as figuras.

2 Generator

2.1 Considerações gerais

Independentemente da figura geométrica em questão, é sempre recebido como argumento a string *nameFile*, representativa do ficheiro que guardará os pontos constituintes dos triângulos que modelam a figura. Assim, cada ficheiro será constituído por um triplo de coordenadas por linha, na forma (x, y, z), e considera-se que um conjunto de três linhas representm um triângulo.

Tendo isto esclarecido, os restantes argumentos recebidos por cada função variam, dependendo da figura, e serão detalhados nas próximas secções.

2.2 Plane

Antes de mais, para a função *plane(float width, string fileName)*, o grupo considerou que o plano seria representado não na sua infinidade, mas restrito ao plano XZ=0 e como um quadrado, centrado em (x, y, z)=(0, 0, 0). Assim, apenas seria necessário fornecer como input a largura (width) da figura, da qual se deduziam os quatro pontos necesários à construção dos dois triângulos constituintes do quadrado.

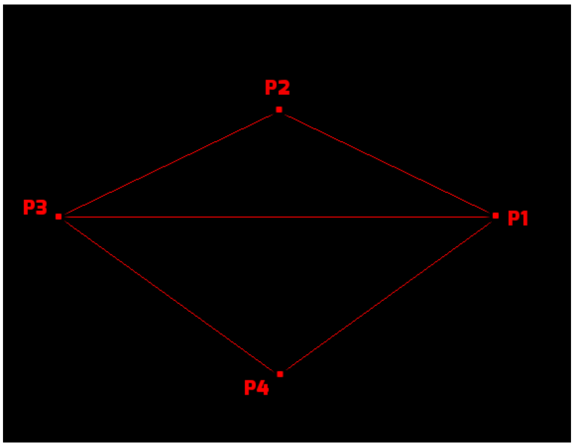


Figura 1: Vértices do plano

Deste modo, surge o seguinte algoritmo:

- Início do algoritmo: Uma vez que o quadrado está centrado em (x, y, z)=(0, 0, 0), é necessário entender que a variável width será usada na representação de coordenadas como width/2
- Representação do primeiro triângulo (de acordo com a regra da mão direita):
  - P1 (width/2, 0, -width/2)
  - P2 (-width/2, 0, -width/2)
  - P3 (-width/2, 0, width/2)
- Representação do segundo triângulo:
  - P3 (-width/2, 0, width/2)
  - P4 (width/2, 0, width/2)
  - P1 (width/2, 0, -width/2)
- Fim do algoritmo.

2.3 Box

Para a presente função *box(float x, float y, float z, int nDivisions, string fileName)*, o grupo considerou que a caixa seria representada em relação ao ponto (x, y, z)=(0, 0, 0), não ficando centrada no mesmo. O objetivo de não centrar a caixa no ponto (0, 0, 0) é construir algo de forma mais simples na fase de desenho do algoritmo, o que facilmente se verificou por comparação com as outras figuras. Assim, é necessário fornecer como input a largura (x), altura (y) e comprimento (z) da figura, para além do número de divisões a fazer. Claramente, no caso de uma caixa, não seria necessária a divisão para melhorar a qualidade do objeto reproduzido (como no caso da esfera, por exemplo).

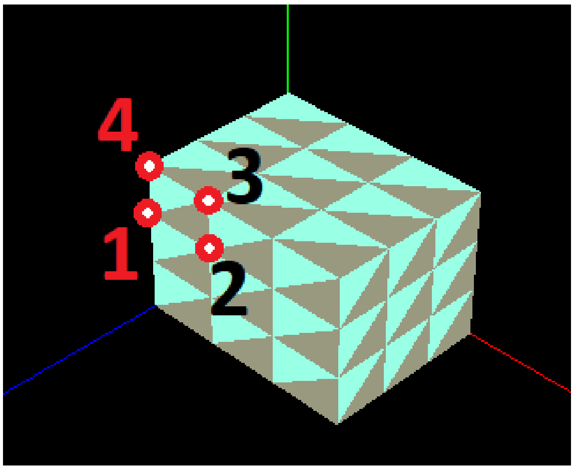


Figura 2: Representação da caixa, com as respetivas divisões

Deste modo, surge o seguinte algoritmo, tendo como base a figura acima:

- Início do algoritmo: Uma vez que a caixa tem diferentes tamanhos nos eixos xx, yy e zz, deve ser feita a divisão do tamanho especificado para cada um desses parâmetros pelo número de divisões especificadas (nDivisions). Assim, obtém-se os valores unitários:
  - unitx = x / nDivisions
  - unity = y / nDivisions
  - unitz = z / nDivisions

  Vão também ser usadas as variáveis *col*, *dep* e *row*, representando a coluna, profundidade e linha atual, para cada um dos triângulos a desenhar, a cada iteração do algoritmo.

- Representação do primeiro triângulo, formado pelos pontos 3-4-1 (de acordo com a regra da mão direita):
  - 3 = (unitx*(col+1), unity*(row+1), z)
  - 4 = (unitx*col, unity*(row+1), z)
  - 1 = (unitx*col, unity*row, z)
- Representação do segundo triângulo:
  - 3 = (unitx*(col+1), unity*(row+1), z)
  - 1 = (unitx*col, unity*row, z)
  - 2 = (unitx*(col+1), unity*row, z)
- Iterar desde 0 até nDivisions as variáveis col, row ou dep, dependendo da face a representar.

- Fim do algoritmo.

De facto, acima apenas se faz o raciocínio para aquele quadrado, representado por 1-2-3-4, mas seria exatamente o mesmo para qualquer outra face, à exceção da coordenada fixa Z, que poderá assumir o valor de X ou 0 (para o lado direito ou esquerdo, respetivamente), Y ou 0 (para a parte superior e inferior, respetivamente) e Z ou 0 (para a frente e trás, respetivamente).

2.4 Sphere

Para começar, na função *sphere(float radius, int slices, int stacks, string fileName)*, seria necessário entender o conceito de coordenadas esféricas e elaborar um raciocínio, com as slices e stacks, para construir a esfera.

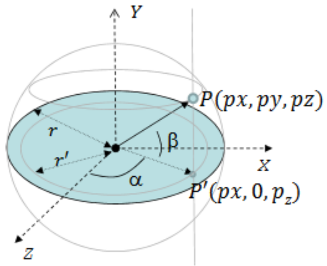


Figura 3: Coordenadas esféricas

Como está representado na figura anterior, o sistema de coordenadas passará, então, a ser representado por um raio e os ângulos alfa e beta. O raio ($0 \leq r < \infty$) indica a distância radial em relação à origem, beta ($-\pi \leq \beta < \pi$) é o ângulo entre o eixo y e a linha que une a origem e ponto P(x, y, z), e alfa ($-2\pi \leq \alpha < 2\pi$) é o ângulo entre o eixo x positivo e a linha que une a origem com a projeção do ponto P(x, y, z) np plano xz. Assim, passando para coordenadas cartesianas, as coordenadas necessárias à resolução deste problema resumem-se a:

- z = r * cos(β) * cos(α)
- x = r * cos(β) * sin(α)
- y = r * sin(β)

Posto isto, e agora considerando as slices e as stacks necessárias à construção da esfera neste contexto, é facil deduzir que alfa e beta serão definidos, na definição de cada ponto e em diferentes iterações da construção da esfera, pelas seguintes opções:

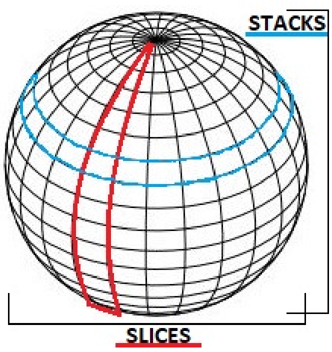


Figura 4: Representação da esfera, dividida em slices e stacks.

- α 1 = (2 * π / slices) * slice
- α 2 = (2 * π / slices) * (slice + 1)
- β 1 = (2 * π / stacks) * stack
- β 2 = (2 * π / stacks) * (stack + 1)

Em linguagem corrente e meramente ilucidativa, um ponto que seja representado por $\alpha 1$ e $\beta 1$ será o ponto atual da iteração. Um outro ponto que tenha exatamente as mesmas coordenadas do ponto atual, mas que seja representado por $\alpha 2$, ao invés de $\alpha 1$, será um ponto na slice exatamente ao lado. Por outro lado, um ponto que disponha das mesmas coordenadas do ponto atual, mas com $\beta 2$, em vez de $\beta 1$, será um ponto exatamente na stack imediatamente ao lado.

Para terminar a explicação, resume-se como se definirão os triângulos no algoritmo que o grupo usou, através da seguinte imagem exemplificativa:

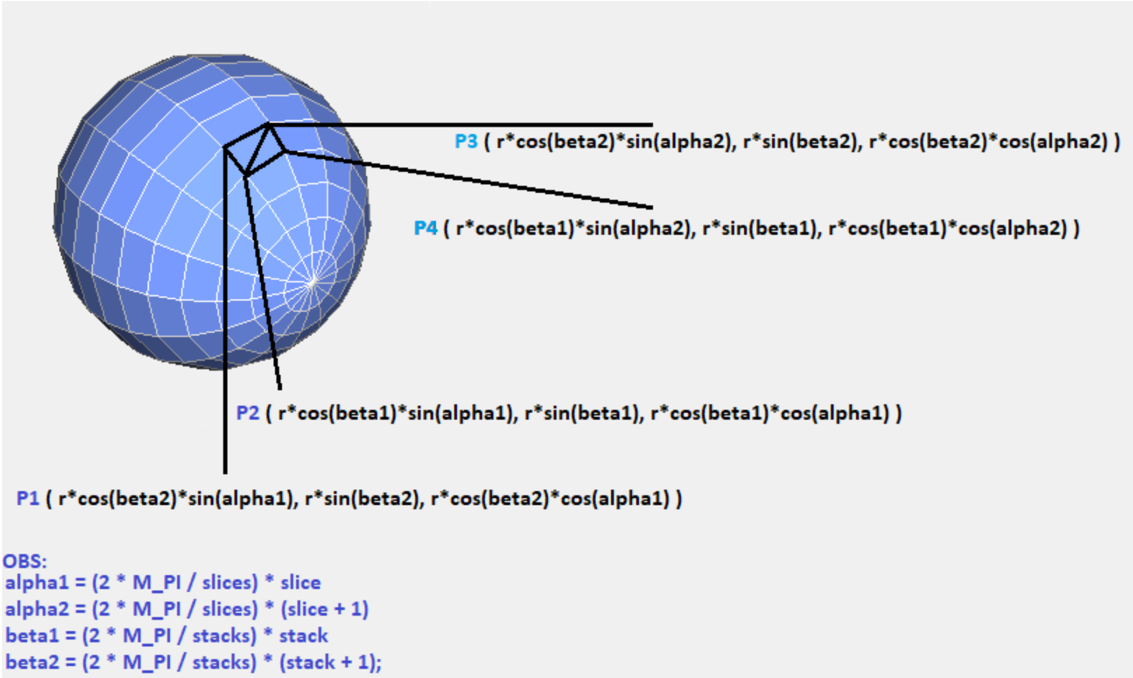


Figura 5: Exemplo sumário de representação das coordenadas na esfera.

Depois de todo este esclarecimento prévio, estamos em condições de resumir o algoritmo utilizado, para melhor compreensão e percepção do que é feito em cada iteração dos ciclos utilizados na função *sphere(float radius, int slices, int stacks, string fileName)*:

- Início do algoritmo: serão necessários dois ciclos aninhados, o mais exterior para controlo das slices sucessivas, e o mais interior para definir as stacks crescentes.
- Dentro dos dois ciclos:
  - Declaração das variáveis "alpha1", "alpha2", "beta1", "beta2", no formato já definido em cima
  - Representação do primeiro triângulo (regra mão direita):
    - P1 (r*cos(beta2)*sin(alpha1), r*sin(beta2), r*cos(beta2)*cos(alpha1))
    - P2 (r*cos(beta1)*sin(alpha1), r*sin(beta1), r*cos(beta1)*cos(alpha1))
    - P3 (r*cos(beta2)*sin(alpha2), r*sin(beta2), r*cos(beta2)*cos(alpha2))
  - Representação do segundo triângulo(regra mão direita):
    - P3 (r*cos(beta2)*sin(alpha2), r*sin(beta2), r*cos(beta2)*cos(alpha2))
    - P2 (r*cos(beta1)*sin(alpha1), r*sin(beta1), r*cos(beta1)*cos(alpha1))
    - P4 (r*cos(beta1)*sin(alpha2), r*sin(beta1), r*cos(beta1)*cos(alpha2))
- Fim do algoritmo.

2.5 Cone

Para construção da função *cone(float r, float height, int slices, int stacks, string fileName)*, foi usado um raciocínio análogo ao da esfera, sendo que a explicação das coordenadas ficará implícita. Deste modo, para construir um cone, o grupo achou que seria necessário, em cada iteração, um triângulo para a base (slice atual), e de dois triangulos para formar cada quadrado da superfície do cone, como mostra a seguinte imagem:

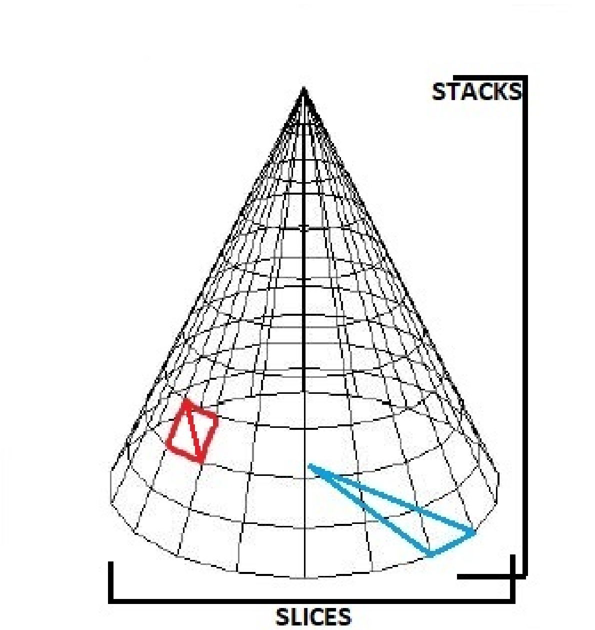


Figura 6: Cone com o efeito de uma slice na base, e de uma slice e stack na superfície lateral.

Começando pela base do cone, um ciclo mais exterior irá avançar de slice em slice, e determinará um triângulo em cada iteração. Já que, para esta figura não é mandatório o uso do ângulo β, apenas se usará os antigos $\alpha 1$ e $\alpha 2$, mas desta vez serão explicitamente desenvolvidos na fórmula das coordenadas. O triângulo terá, então, como coordenadas:

- P1 (0, 0, 0) -> centro
- P2 (r*sin(alpha*(slice+1)), 0, r*cos(alpha*(slice+1))) -> y=0
- P3 ((r*sin(alpha*slice), 0.0, r*cos(alpha*slice)) -> y=0

Ao mesmo tempo, em cada slice, um ciclo interior, percorrendo as stacks, vai ajudar a definir os triangulos laterais. É importante notar que, nesta figura, o argumento de input height sortirá de grande importância, uma vez que servirá para definir as alturas sucessivas das stacks, enquanto que o raio também terá que ser controlado.

Para descobrir quais valores da altura e do raio de cada stack e em cada iteração, em relação à altura e raio atuais, o grupo utilizou a regra da semelhança de triângulos, demonstrada abaixo:

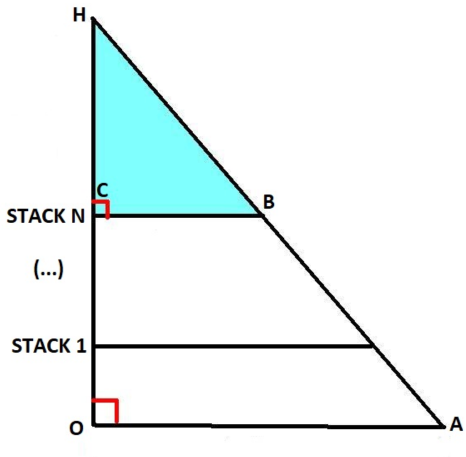


Figura 7: Regra de semelhança de triângulos.

$$\frac{HC}{HO} = \frac{CB}{OA} \Leftrightarrow$$

$$\Leftrightarrow \frac{Height - StackHeight}{Height} = \frac{StackRadius}{Radius} \Leftrightarrow$$

$$\Leftrightarrow StackRadius = \frac{(Height - StackHeight) * Radius}{Height} \Longrightarrow$$

$$\Longrightarrow StackHeight = \frac{Height}{Stacks} * nStack$$

Posto tudo isto, o algoritmo usado pelo grupo na elaboração dos vértices do cone apresenta-se em baixo, junto com uma imagem representativa:

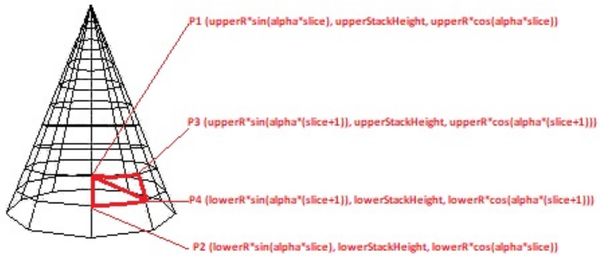


Figura 8: Exemplo sumário de representação das coordenadas no cone.

- **Início do algoritmo**: serão necessários dois ciclos aninhados, o mais exterior para controlo das slices sucessivas, e o mais interior para definir as stacks crescentes.
- Declaração da variável alpha = 2*pi / slices;
- Dentro do ciclo externo:
  - Representação do triângulo da base (regra mão direita):
    - P1 (0, 0, 0)
    - P2 (r*sin(alpha*(slice+1)), 0, r*cos(alpha*(slice+1)))
    - P3 ((r*sin(alpha*slice), 0.0, r*cos(alpha*slice))
  - Declaração da variável lowerR = r
  - Declaração da variável lowerStackHeight = 0
  - Ciclo interno:
    - Declaração da variável upperStackHeight = height/stacks*stack
    - Declaração da variável upperR = (height - upperStackHeight) * r / height
    - Representação de um triângulos da superfície (regra mão direita):
      - P2(lowerR*sin(alpha*slice),lowerStackHeight,lowerR*cos(alpha*slice))
      - P4(lowerR*sin(alpha*(slice+1)),lowerStackHeight,lowerR*cos(alpha*(slice+1)))
      - P1(upperR*sin(alpha*slice),upperStackHeight,upperR*cos(alpha*slice))
    - Representação de um triangulo da superfície (regra mão direita):
      - P4(lowerR*sin(alpha*(slice+1)),lowerStackHeight,lowerR*cos(alpha*(slice+1)))
      - P3(upperR*sin(alpha*(slice+1)),upperStackHeight,upperR*cos(alpha*(slice+1)))
      - P1(upperR*sin(alpha*slice),upperStackHeight,upperR*cos(alpha*slice))
    - lowerR=upperR
    - lowerStackHeight = upperStackHeight
- **Fim do algoritmo**.

2.6 Pyramid

A função *pyramid(float height, float width, float length, string fileName)* recebe como input três floats, representantes da altura (height), largura (width) e comprimento (length) da pirâmide. A partir destes valores, criam-se os cinco pontos para formar os seis triângulos (quatro laterais+dois da base) necessários à construção da pirâmide quadrangular/rectangular, seguindo o algoritmo:

- Início do algoritmo: Uma vez que o quadrado/retângulo está centrado em (x, y, z)=(0, 0, 0), é necessário entender que as variáveis "width"e "length"serão usadas na representação de coordenadas como width/2 e length/2.
- Representação dos dois triângulos da base (regra da mão direita):
  - P1 (length/2, 0, width/2)
  - P2 (-length/2, 0, width/2)
  - P3 (-length/2, 0, -width/2)

  - P3 (length/2, 0, width/2)
  - P4 (-length/2, 0, -width/2)
  - P1 (length/2, 0, -width/2)
- Representação dos quatro triângulos laterais (regra da mão direita):
  - P1 (0, height, 0)
  - P3 (-length/2, 0, width/2)
  - P4 (length/2, 0, width/2)

  - P1 (0, height, 0)
  - P2 (length/2, 0, width/2)
  - P3 (length/2, 0, -width/2)

  - P1 (0, height, 0)
  - P2 (-length/2, 0, -width/2)
  - P5 (-length/2, 0, width/2)

  - P1 (0, height, 0)
  - P5 (length/2, 0, -width/2)
  - P4 (-length/2, 0, -width/2)
- Fim do algoritmo.

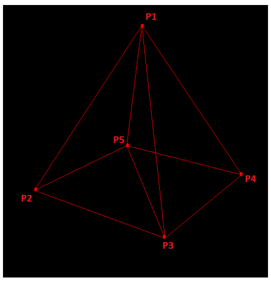


Figura 9: Vértices da pirâmide

2.7 Cylinder

O algoritmo utilizado para o cilindro (*cylinder(float r, float height, int stacks, int slices, string fileName)*) é muito semelhante ao do cone. Apenas não é necessário fazer a variação do raio, o que leva à necessidade de construir a parte superior e inferior de forma diferente.

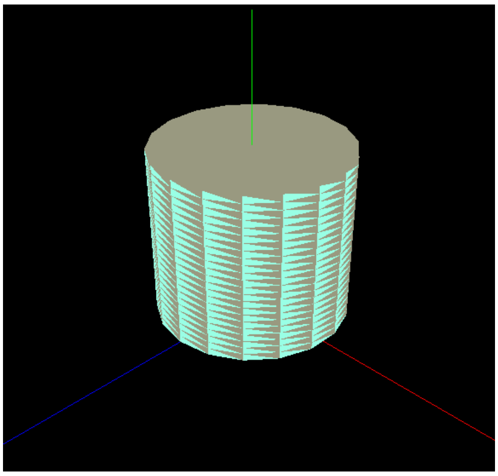


Figura 10: Representação do cilindro, onde se pode ver a construção da lateral e parte superior

- **Início do algoritmo**: declaração da variável alpha = 2*pi / slices.
- Serão necessários dois ciclos aninhados. O primeiro iterará pelas slices, sendo o segundo responsável pela iteração pelas stacks, sendo que as slides são construídas uma a uma. O código a seguir constrói os triângulos do topo e base, para a slice em iteração.
  - P1 (0, 0, 0)
  - P2 (r*sin(alpha*(slice + 1)), 0.0, r*cos(alpha*(slice+1)))
  - P3 (r*sin(alpha*slice), 0.0, r*cos(alpha*slice))

  e

  - P4 (0.0, height, 0.0)
  - P5 (r*sin(alpha*slice), height, r*cos(alpha*slice))
  - P6 (r*sin(alpha*(slice+1)), height, r*cos(alpha*(slice+1)))
- Declaração da variável upperStackHeight = height/stacks*stack;
- O segundo ciclo desenha as laterais, seguindo a mesma lógico do cone (com a referida exceção da necessidade de ajustar o raio):
  - Representação do primeiro triângulo, seguindo a lógica já demonstrada anteriormente:
    * P1 (r*sin(alpha*slice), lowerStackHeight, r*cos(alpha*slice))
    * P2 (r*sin(alpha*(slice+1)), lowerStackHeight, r*cos(alpha*(slice+1)))
    * P3 (r*sin(alpha*slice), upperStackHeight, r*cos(alpha*slice))
  - Representação do segundo triângulo:
    * P4 (r*sin(alpha*(slice+1)), lowerStackHeight, r*cos(alpha*(slice+1)))
    * P5 (r*sin(alpha*(slice+1)), upperStackHeight, r*cos(alpha*(slice+1)))
    * P6 (r*sin(alpha*slice), upperStackHeight, r*cos(alpha*slice))
- **Fim do algoritmo**.

3 Engine

Para o engine foram criados dois ficheiros (em adição ao main.cpp): engine.cpp e xml-loader.cpp (cada um com um ficheiro .h correspondente). Cada um destes ficheiros adicionais tem também o seu namespace, de modo a garantir boas práticas de programação em C++.

O xml-loader encarrega-se de ler o ficheiro XML correspondente a uma cena que se quer apresentar, carregando para a memória a informação relativa aos vértices dos modelos envolvidos.

O engine, por sua vez, trata de receber a informação relativa aos vértices, e a partir deles apresentar os modelos no ecrã.

3.1 main

O ficheiro principal, *main.cpp*, trata de correr o código de inicialização necessário, assim como de certificar que todas as partes do programa funcionam em conjunto. Isto inclui:

- Inicializar a biblioteca openGL através do GLUT.
- Carregar o documento XML através do *xml-loader*.
- Passar o vetor de modelos carregados ao *engine* de maneira a apresentá-los ao utilizador.

3.2 xml-loader

3.2.1 Formato do documento XML

O formato do ficheiro XML utilizado para representar uma cena é o seguinte:

```
<scene>
    <model file="circle.3d"/>
    <model file="circle.3d"/>
</scene>
```

Este consiste num elemento pai <scene>, que contém um ou mais elementos filho *<model>*, com um atributo *file="ficheiro.3d"* que indica o nome do ficheiro que contém os vértices correspondentes a um dado modelo.

3.2.2 Formato do ficheiro .3d

O formato de ficheiro escolhido para os ficheiros .3d foi o mais simples possível, contando apenas com as coordenadas X, Y e Z dos vértices, um vértice por linha.

```
1.000000 1.000000 0.000000
0.000000 1.000000 0.000000
0.000000 1.000000 1.000000
1.000000 1.000000 0.000000
```

3.2.3 Leitura do Ficheiro XML

Como sugerido pelos docentes, foi usada a biblioteca *tinyxml2* de maneira a facilitar a leitura do documento. Apenas foi criado um método para este efeito, declarado como *xmlLoader::loadSceneXML(const char* path)*, que recebe como único parâmetro o caminho até ao ficheiro XML.

Este método faz o seguinte:

- Começa por tentar carregar o documento no caminho recebido, terminando com um erro se não o tiver encontrado ou se tiver ocorrido um erro.
- Procura o elemento pai *<scene>*.
- Itera pelos elementos filho *<model>*, guardando o valor do atributo file num vetor *std::vector<string> modelPaths*.
- Tendo obtido todos os caminhos dos ficheiros *.3d*, o xml-loader itera por estes e para cada um:
  - Carrega o ficheiro *3d.* para uma *ifstream*.
  - Cria um *engine::model* (classe criada por nós) que contém um vetor onde são guardados os vértices que compõem o modelo.
  - Até ao end of file, lê linha a linha e por cada uma cria um *engine::vertex* (classe criada por nós) com as coordenadas correspondentes (modelFile » vertex.x » vertex.y » vertex.z). Uma vez criado o vértice, este é adicionado ao *engine::model* criado para o efeito.
  - Tendo terminado de percorrer o ficheiro *.3d*, adiciona o modelo resultante a um vetor *std::vector<engine::model>loadModels*.
- No fim, é retornado o vetor *loadedModels*.

3.3 engine

3.3.1 Classes auxiliares

Foram criadas duas classes auxiliares:
engine::vertex

```
struct vertex {
    float x, y, z;
};
```

Guarda a informação relativa a um vértice em 3 floats, cada uma corresponde a uma coordenada: X, Y e Z.

engine::model

```
struct model {
    std::vector<vertex> vertices;
};
```

Guarda a informação relativa a um modelo, sendo colocados num vetor os vértices que o compõem.

3.3.2 Métodos

O *engine* dispõe de dois métodos:

- *void engine::loadScene(vector<model> scene)*, que é usado para guardar a cena a ser representada na memória, para que esta seja acessível localmente;
- *void engine::drawFrame()*, que é chamado a cada frame e se encarrega de renderizar os modelos presentes na scene.

loadScene
O método *loadScene* limita-se a colocar o vetor *scene* recebido numa variável local, para que o método *drawFrame* lhe consiga aceder.

drawFrame

O método *drawFrame* itera pelo vetor *scene* e renderiza os modelos um a um. Começa por executar apenas *glBegin(GL TRIANGLES)*, uma vez que as outras configurações já foram executadas pelo ficheiro main.cpp. De seguida, por cada modelo, itera pelo vetor dos seus vértices e executa *glVertex3f(vertex->x, vertex->y, vertex->z)*.

Como os vértices são guardados na ordem certa, não é necessário qualquer cuidado adicional neste passo. Entre a renderização dos triângulos, são realizadas algumas chamadas ao método *glColor3f* de maneira a variar a cor destes, facilitando a sua identificação pelo utilizador. Por fim, é executado o método *glEnd*.

3.4 Debug

De maneira a verificar os resultados obtidos, foram adicionadas algumas funcionalidades à interface do programa. A tecla F1 foi definida para renderizar apenas as arestas das figuras, enquanto que a F2 define o modo de renderização que inclui as faces dos modelos. As teclas W, A, S e D rodam os modelos, enquanto que as setas movem-nos no espaço.

Com a ajuda destas funcionalidades foram descobertos alguns problemas ao longo do desenvolvimento do programa, que por terem sido identificados a tempo foram rapidamente resolvidos.

4 Conclusão

Esta primeira fase do projeto foi bastante importante e enriquecedora, pois permitiu aos membros do grupo perceber e interiorizar melhor os conceitos abordados nas aulas práticas da Unidade Curricular de Computação Gráfica. Deste modo, foram adquiridos conhecimentos básicos acerca da biblioteca de funcionalidades GLUT para OpenGL, bem como conhecimento relativo a um conjunto de primitivas gráficas, que certamente serão úteis para as fases seguintes do projeto prático.

No que diz respeito ao desenvolvimento da aplicação Generator, a principal dificuldade encontrada foi o desenvolvimento de algumas figuras e chegar ao algoritmo de geração de vértices das mesmas, uma vez que o grupo decidiu alargar o conjunto de primitivas gráficas requeridas no enunciado, na perspectiva de demonstrar o interesse e empenho no projeto desta UC. Posto isto, com algum trabalho e aproveitando as bases adquiridas nas aulas,a equipa foi capaz de produzir com sucesso os algoritmos responsáveis pela estruturação das figuras.

Quanto ao desenvolvimento do Engine, um dos grandes obstáculos foi a divisão do problema em partes de maneira a seguir a abordagem mais simples e eficaz. Depois de desenhada a solução, e tendo em conta que este foi o primeiro contacto que os elementos do grupo tiveram com a linguagem C++, a implementação foi relativamente simples.

Em suma, é feita uma apreciação positiva relativamente ao trabalho realizado, visto que a implementação de todas as funcionalidades propostas foram conseguidas com sucesso. O grupo conseguiu tirar partido dos conhecimentos adquiridos neste projeto, sentido-se capaz de, num contexto futuro, aplicar os conceitos subjacentes de forma eficaz.

5 Anexos

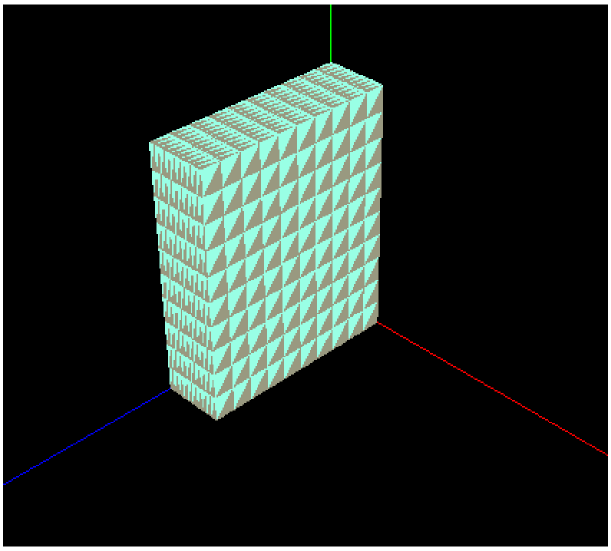


Figura 11: BOX - Resultado final.

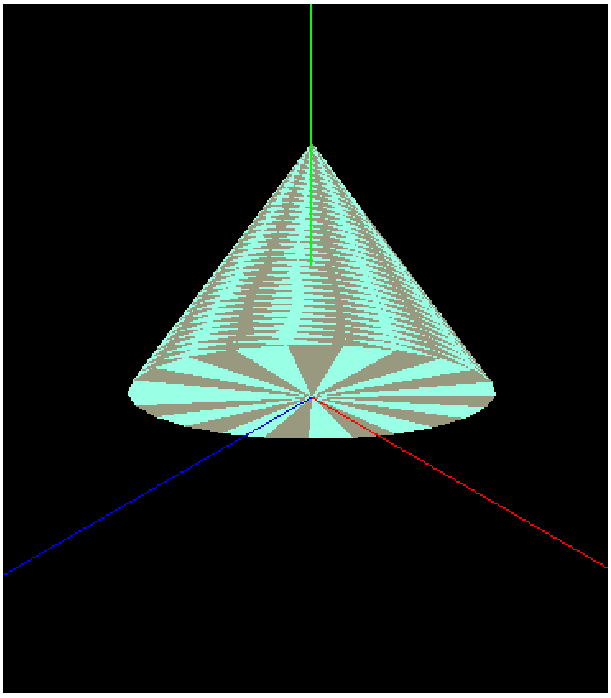


Figura 12: CONE - Resultado final1.

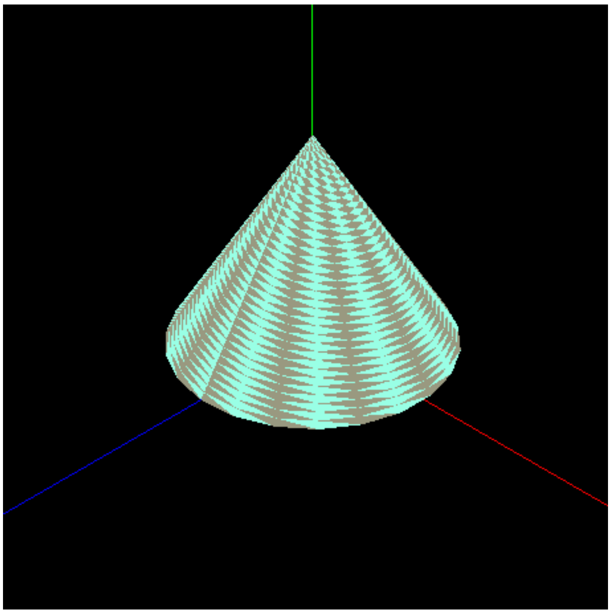


Figura 13: CONE - Resultado final2.

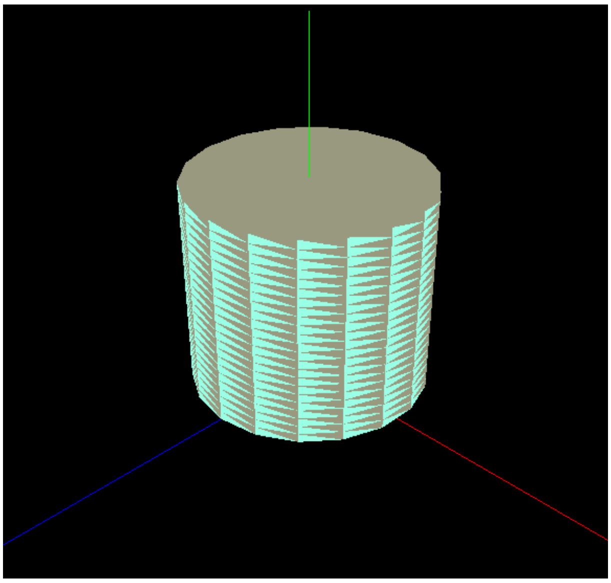


Figura 14: CYLINDER - Resultado final.

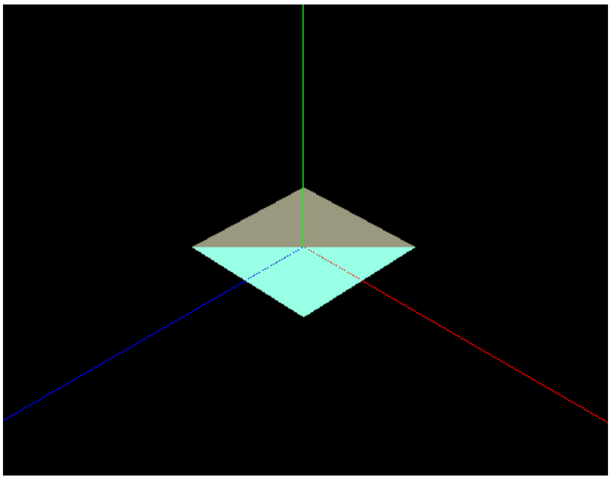


Figura 15: PLANE - Resultado final.

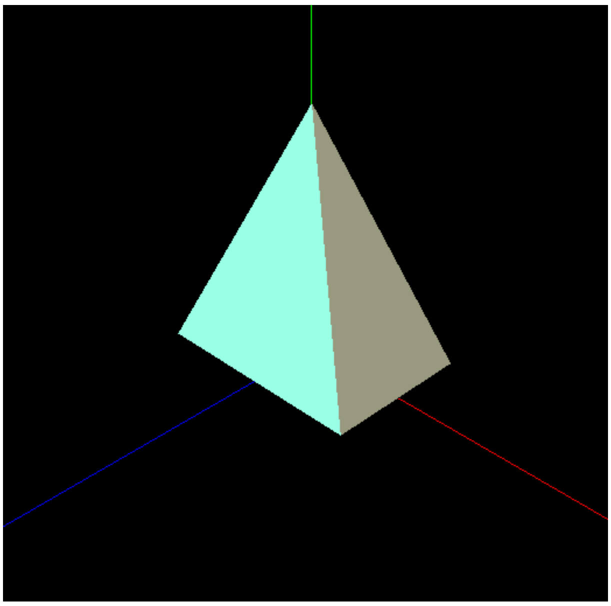


Figura 16: PYRAMID - Resultado final1.

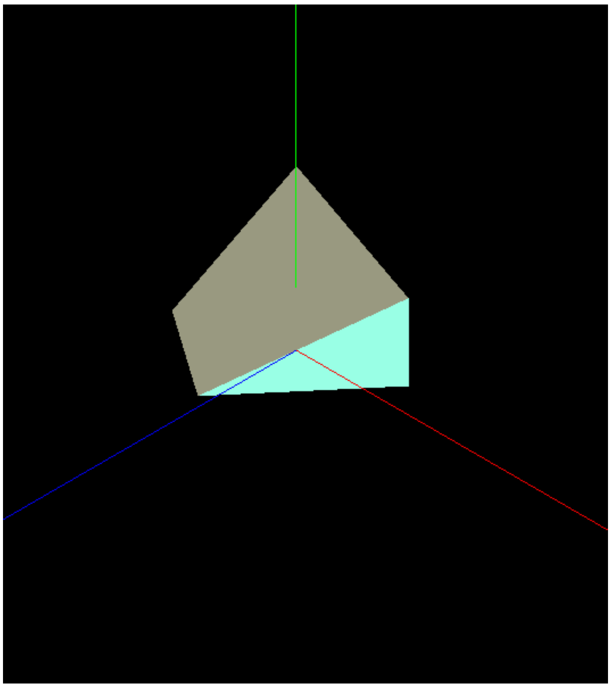


Figura 17: PYRAMID - Resultado final2.

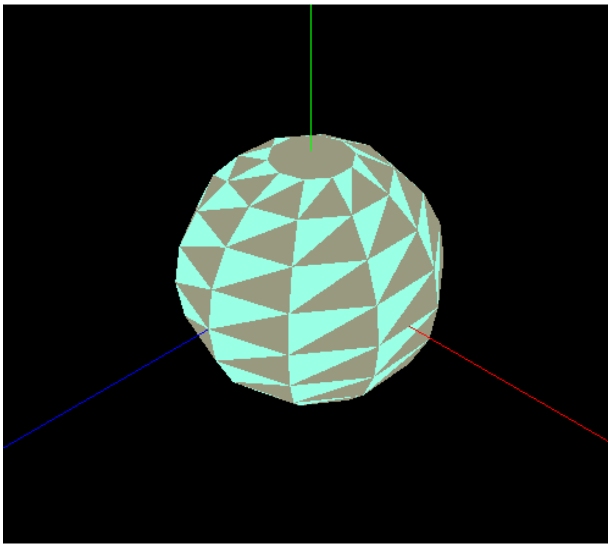


Figura 18: SPHERE - Resultado final.