Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

1ª Fase

Grupo 17

Daniela Fernandes	José Gomes	Ricardo Costa	Tiago Rodrigues
A73768	A82418	A85851	A84276

14 de março de 2021

Conteúdo

1 Introdução

Este documento, elaborado para a primeira fase do Projeto Prático na unidade curricular de Computação Gráfica, serve para retratar todas decisões tomadas para a resolução do problema desenvolvido em C/C++.

Para a vertente prática da unidade curricular utiliza-se como base a utilização do *OpenGL*, recorrendo à biblioteca *GLUT*, para a construção de modelos 3D.

Nesta fase foi-nos pedido a criação de um plano, uma caixa, uma esfera e um cone. Para além disso, foi-nos pedido um gerador para gerar ficheiros com os pontos de cada primitiva e um motor para ler um ficheiro XML, onde estão descritos os modelos que é suposto desenhar.

2 Descrição do Problema

Para esta fase do projeto foi-nos dado como objetivo a criação de dois programas: um **gerador** que gera um ficheiro com os pontos de um modelo específico e um **motor 3D** que lê um ficheiro de configuração, escrito em XML e representa graficamente os modelos.

Os requisitos básicos desta fase são:

- **Gerador** tem como parâmetros os tipos de primitiva gráfica, parâmetros relativos ao modelo e o nome do ficheiro onde vão ser guardados os vértices do modelo:
  - **plane:** Um quadrado no plano xz, centrado na origem e feito com 4 triângulos (frente e trás).
  - **box:** Requer o comprimento (x), a largura(z) e a altura(y) e o número de divisões.
  - **sphere:** Necessita do raio e o número de divisões verticais(*slices*) e horizontais(*stacks*).
  - **cone:** Necessita do raio da base, da altura e o número de divisões verticais(*slices*) e horizontais(*stacks*).
- **Motor 3D** lê de um ficheiro XML que contêm os modelos das primitivas, carrega e armazena os vértices dos modelos em memória. Depois de carregados os ficheiros, desenha, utilizando triângulos, as primitivas dos modelos.

3 Resolução

Para a simplificação da resolução do problema foi criada uma estrutura *Point* que envolve a representação de um vértice num plano de 3 dimensões.

```
struct Point {
    float x;
    float y;
    float z;
};
```

Vai armazenar as coordenadas x, y e z de um ponto.

3.1 Gerador

3.1.1 Plano/Plane

Para formar o **plano** vamos utilizar quatro triângulos, dois desenhados no sentido positivo do eixo y e dois no sentido negativo, centrado na origem e sobre os eixos coordenado x e z.

Deste modo os vértices que formam o plano, na coordenada y vão ser igual a 0, nas coordenadas x e z vão possuir valores iguais simétricos.

Tendo em conta que o plano é um quadrado que está centrado na origem, concluímos que a distância de qualquer ponto à origem é igual à metade do tamanho do lado. Depois, foi apenas necessário ter em consideração a ordem pela qual escrevemos os pontos para ficheiro para termos os 2 triângulos que vão estar virados para cima e os 2 que vão estar viradas para baixo, tendo em conta a regra da mão direita.

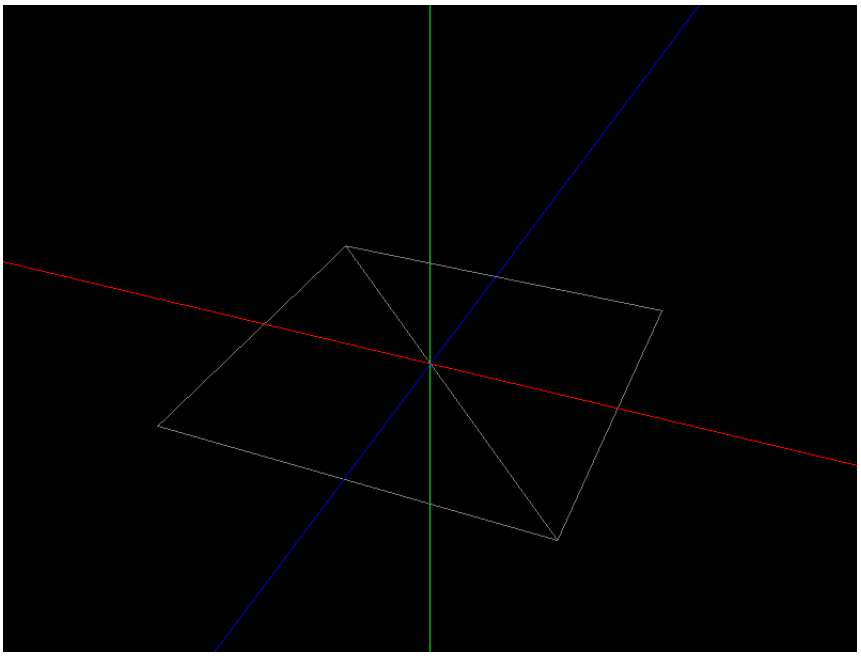

Figura 1: Plano com lado 4 visto de cima

3.1.2 Caixa/Box

Para a **box**, começou-se por calcular os pontos que formam as faces da caixa. Com estes valores conseguimos definir os triângulos que compõe cada uma delas. Começamos por apenas nos preocupar com desenhar a **box** com 2 triângulos para cada face, aplicando basicamente a mesma estratégia como no plano. De seguida, pensámos como poderiamos tratar das divisões e também o que seria exatamente cada divisão. Assim, decidimos que esta primeira forma inicial com apenas 2 triângulos corresponde a 0 divisões, já que é o mínimo necessário para desenhar um quadrado que corresponde à face da **box**.

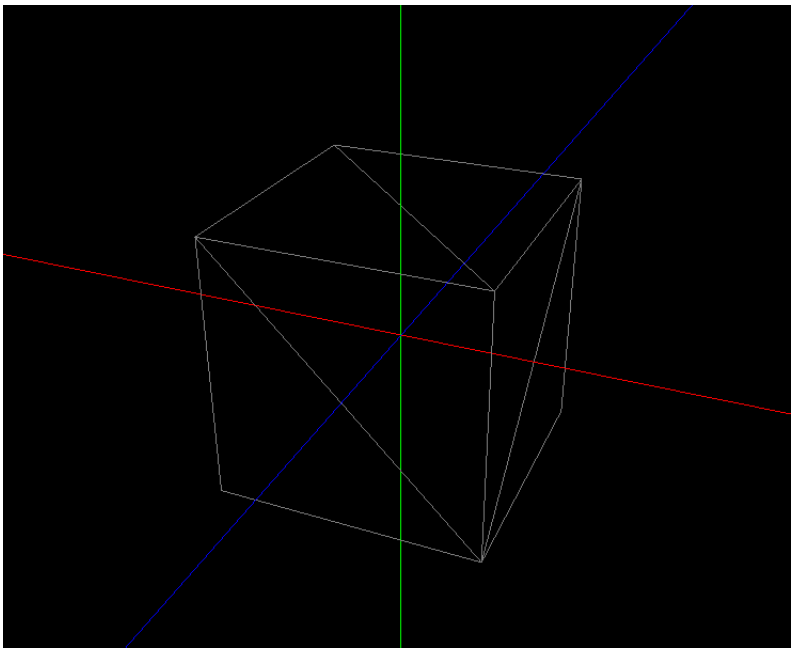

Figura 2: Caixa com 0 divisões, ou seja, apenas 2 triângulos por face

Posto isto, se quisessemos por exemplo 1 divisão seria dividir os dois triângulos que teríamos inicialmente (com 0 divisões). Para fazermos isto de maneira correta, temos de determinar qual é o maior lado de um dado triângulo (a hipotenusa) para podermos calcular um ponto no meio desse lado do triângulo, ou seja, entre os dois pontos cuja distância é a maior. Esse ponto é o que vai servir para formar os dois triângulos resultantes da divisão de um triângulo, juntamente com os outros pontos iniciais. Depois é executado o mesmo processo para o outro triângulo.

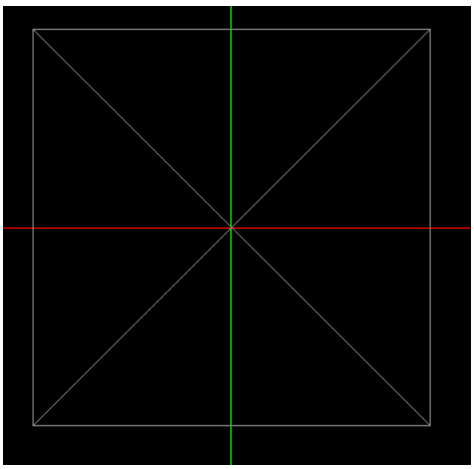

Figura 3: Resultado de 1 divisão visto de frente (eixo Y a verde e eixo X a vermelho

A função que definimos para isto (*divideTriangle*) é chamada recursivamente 2 vezes de cada vez com o número de divisões subtraído por 1, já que quando dividimos um triângulo, se o número de divisões subtraído por 1 não for 0, vamos ter de aplicar o processo a cada um dos triângulos formados pela divisão. Este processo para quando o número de divisões chegar a 0, altura em que os pontos são escritos para ficheiro.

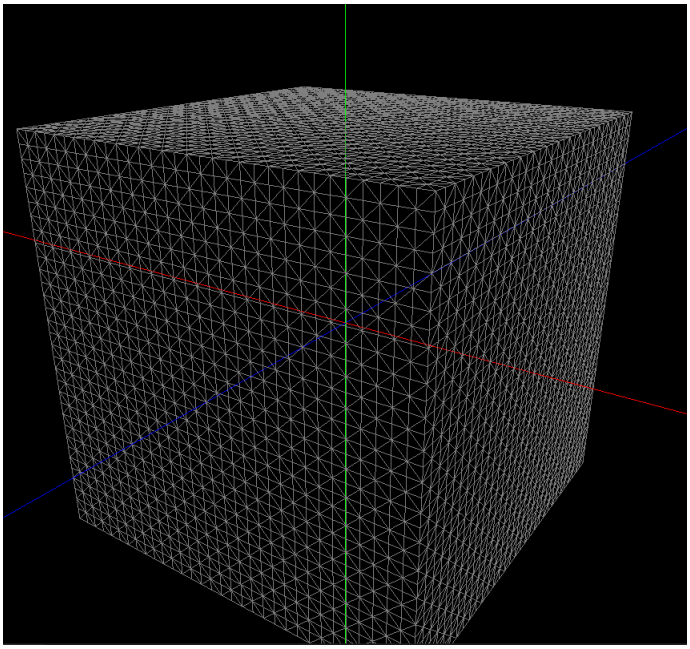

Figura 4: Caixa com 10 divisões

3.1.3 Esfera/Sphere

Para formar a **sphere** vamos precisar de utilizar coordenadas esféricas, através de dois ângulos α e β, e a distância do ponto à origem, raio, definem as coordenadas de um ponto.

O ângulo α vai variar entre 0 e 2π radianos, enquanto o ângulo β vai variar entre $\frac{-\pi}{2}$ e $\frac{\pi}{2}$. O α vai ser dividio em *slices* e o β em *stacks*.

Para a divisão da esfera, vamos percorrer *slices* divisões com o ângulo α e *stacks* divisões com o ângulo β. Sabemos que para α cada divisão vale $\frac{2\pi}{slices}$ e que iniciamos a divisão em 0 até à divisão *slices*. O ângulo β como inicia em $\frac{-\pi}{2}$ vai iniciar na divisão $\frac{-stacks}{2}$ e terminar em $\frac{stacks}{2}$, cada uma das divisões vai valer $\frac{\pi}{stacks}$.

Para cada ponto o que é alterado são os valores dos ângulos α e β, e utilizando as coordenadas esféricas conseguimos calcular todas as coordenadas de cada ponto.

$$x = radius \times sin(\alpha) \times cos(\beta) \tag{1}$$

$$y = radius \times sin(\beta) \tag{2}$$

$$z = radius \times cos(\alpha) \times cos(\beta) \tag{3}$$

Em cada iteração vamos desenhar dois triângulos (um quadrado), ou seja, vamos calcular 4 pontos através das coordenadas esféricas. Tivemos em consideração que se nos encontrarmos na primeira ou na última iteração das *stacks* apenas é necessário escrever 3 pontos para o ficheiro, já que nestas iterações que correspondem à parte de baixo e à parte de cima da esfera apenas há um triângulo. Podiamos ter optado por ignorar isto e escrever na mesma os 6 pontos para ficheiro, já que devido ao valor ângulo β com que os pontos são escritos há 2 pontos dos 3 pontos que formariam o segundo triângulo que são iguais e então na parte do *engine* não é desenhado nada. Assim, também evitámos que o ficheiro da esfera contenha pontos inúteis.

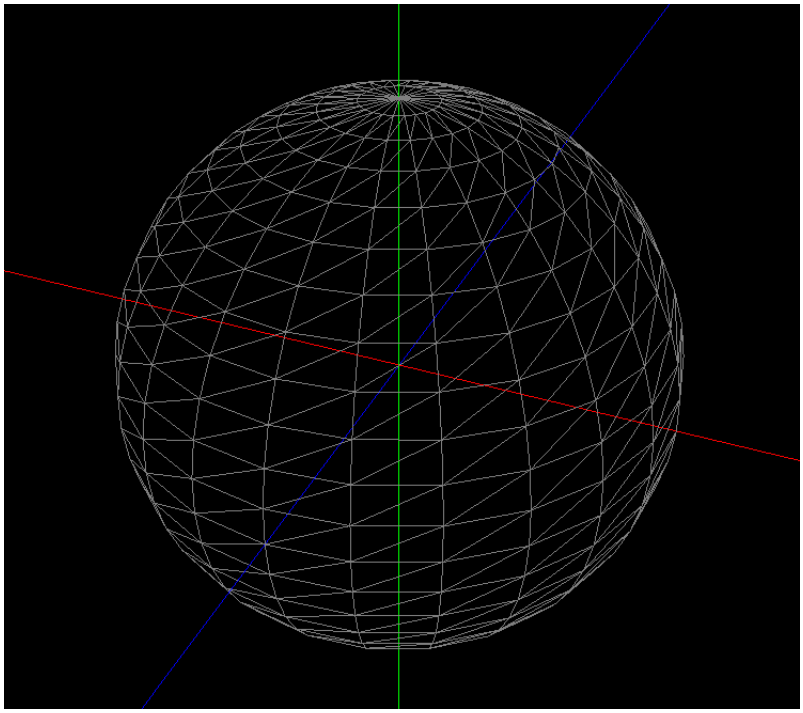

Figura 5: Esfera com 24 *slices* e 24 *stacks*

3.1.4 Cone

Em relação ao **cone** precisamos de receber quatro argumentos: o raio, a altura, o número de *slices* (divisões na horizontal) e o número de *stacks* (divisões na vertical). Os pontos para o **cone** são gerados utilizando apenas coordenadas polares, ou seja apenas um ângulo α que varia entre 0 e 2π radianos, tal como no caso da esfera. Este ângulo vai ser dividido pelo número de *slices* para obtermos o incremento horizontal de *slice* para *slice*. Além disso, o α também é na mesma inicializado com 0.

$$x = radius \times sin(\alpha) \tag{4}$$

$$z = radius \times cos(\alpha) \tag{5}$$

Podemos dividir a geração dos pontos em 2 fases:

- **Base do cone:** os pontos para a base são gerados através de um ciclo *for* que itera pelo número de *slices* e em cada iteração gera 3 pontos (1 deles sendo sempre o ponto (0, 0, 0)) que vão formar os triângulos da base através das coordenadas polares. A coordenada y é sempre 0 já que a base é desenhada no plano ZX.

- **Laterais do cone:** os pontos para esta parte são gerados iterando na mesma pelo número total de *slices* e para cada *slice* iterar pelo número total de *stacks*. Assim, são dois ciclos *for* aninhados. Em cada iteração das *stacks* são calculados 4 pontos (que vão formar 2 triângulos, ou seja, 1 quadrado) através na mesma das coordenadas polares. Um detalhe é que é necessário em cada iteração calcular um novo raio que vai ser usado no cálculo dos dois pontos de cima de cada quadrado e para o calcular é usada a semelhança de triângulos. Além disto, também é incrementada uma variável h por cada iteração que corresponde à altura dos pontos que vamos desenhar (apenas os 2 de baixo porque não esquecer que os 2 pontos de cima de cada quadrado tem esta altura mais o incremento da altura por iteração). Este incremento é calculado dividindo a altura dada do cone pelo número de *stacks*.

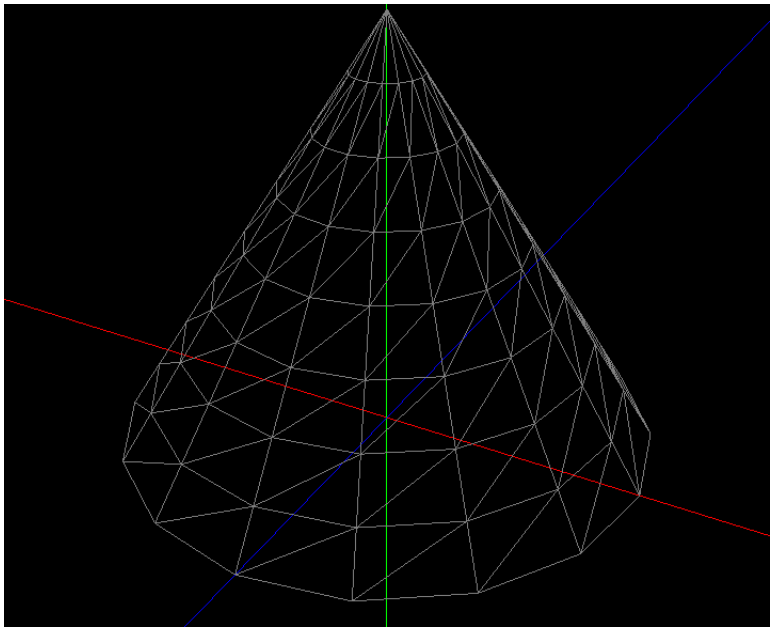

Figura 6: Cone com 16 *slices* e 8 *stacks* visto de cima

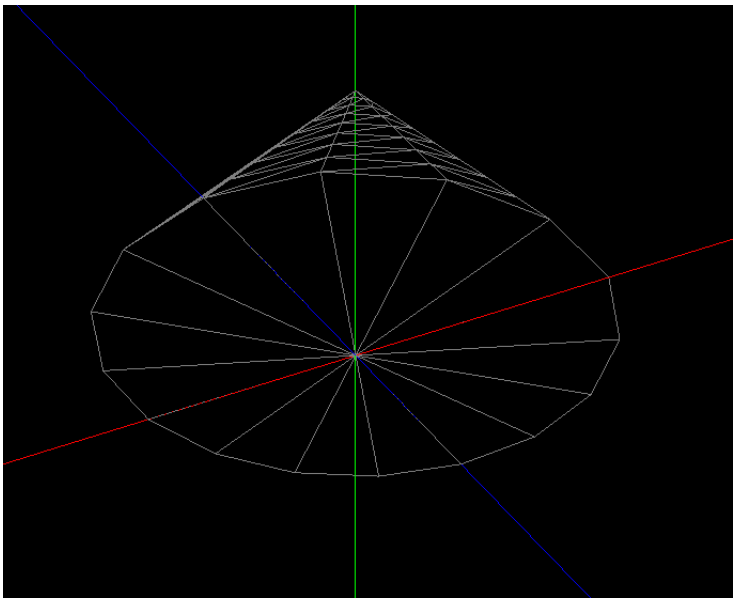

Figura 7: Base do mesmo cone

3.1.5 Escrita no ficheiro

A escrita no ficheiro cujo nome é passado como argumento de entrado ao **gerador**, realizada de cada vez que se calcula um ponto, ou seja , os pontos não vão ser armazenados em nenhuma estrutura de dados. Fazemos isto para poupar a memória, mantendo assim a eficiência do programa. Para cumprir este objetivo criamos uma função *point_write* que escreve no ficheiro um dado ponto, isto é, de cada vez que é decifrado um ponto escreve-se no seu respetivo ficheiro.

Foi utilizada uma *offstream* que abre o ficheiro com os modos **ios::binary** e **ios::out** ativos para podermos escever nele. Com isto, o ficheiro para o qual o programa vai escrever vai ser do tipo binário. Esta escolha foi tomada para tornar o processo mais económico, pois, para ficheiros com muitos vértices, além de compactar o tamanho do ficheiro, também é mais eficiente a sua escrita e leitura.

Também para facilitar a leitura do ficheiro vamos utilizar o caracter '\0' para delimitar o fim da informação de um ponto.

No ficheiro apenas estão os vértices necessários para desenhar cada primitiva. Cada um dos ficheiros contém a informação organizada com as três coordenadas de cada vértice, separadas por um espaço, e separadas de outro vértice utilizando '\0'.

3.2 Motor 3D

3.2.1 Leitura do Ficheiro XML

Como os ficheiros que devem ser carregados (ficheiros .3d) são ficheiros XML, vamos utilizar uma biblioteca *TinyXML-2* para fazer o *parsing* do documento, para decifrar qual o modelo a desenhar.

O ficheiro XML é passado como argumento ao **engine**. Para se ler o ficheiro criamos uma função **load_XML_file**, que percorre o ficheiro e à medida que vai encontrando um modelo, armazena-o em $vector < string > models$.

3.2.2 Modelos

Depois dos modelos estarem carregados, leu-se os mesmos. Para isto, percorre-se o vetor **models** que foi anteriormente preenchido e , para cada um dos modelos, é invocada a função **load_model_file**.

A função vai abrir o ficheiro, que é um ficheiro binário, relativo ao modelo que lhe é passado como argumento, e vai percorrê-lo de '\0' em '\0', até ao fim do ficheiro. Como a informação que se encontra entre '\0' representa um vértice, e a ordem pelo o qual está escrito permite desenhar os triângulos necessários para a formação de cada primitiva. Os pontos à medida que vão ser lidos vão ser armazenados num $vector < Point > vertices$.

3.2.3 Desenhar as primitivas

Para desenhar as primitivas, cujos vértices já foram carregados, utilizamos a função **renderScene**.

Vai percorrer o vetor **vertices** e desenha os triângulos necessários para a criação das primitivas. Para isso vai definir a opção **GL_TRIANGLES** e vai usar a função **glVertex3f** da biblioteca *GLUT*.

3.2.4 Câmara

A câmara é definida como já sabemos através da função *gluLookAt* da biblioteca *GLUT*. Decidimos definir a câmara através das coordenadas esféricas, no modo que se costuma designar de *Explorer*. Basicamente a câmara pode-se mover livremente pela superfície de uma esfera invisível, alterando o ângulo α e β, e está sempre a "olhar" para o centro, ou seja o ponto (0, 0, 0). Podemos dar *zoom in* ou *zoom out* diminuindo ou aumentado o raio da tal esfera invisível. Isto é feito definindo 3 variáveis globais: *alpha*, *beta* e *radius* que são alteradas através do teclado que vamos explicar a seguir. Assim, a função *gluLookAt* está definida da seguinte forma:

```
gluLookAt(radius * cos(beta) * sin(alpha), radius * sin(beta),radius * cos(beta) *
↪  cos(alpha),
              0.0,0.0,0.0,
              0.0f,1.0f,0.0f);
```

3.2.5 Teclado

A interação com o teclado permite como já falamos mover a câmara, dar *zoom in* ou *zoom out* e, além disso, alterar o modo de desenho entre *GL_FILL* e *GL_LINE* através da função *glPolygonMode*.

Para isto são usadas duas *callbacks* que permitem registar as teclas especiais (como as setas) através da função *glutSpecialFunc(special_keys)* e as teclas como o *P* através da função *glutKeyboardFunc(regular_keys)*, em que as funções *special_keys* e *regular_keys* são as que definimos para escolher o que faz cada tecla (especial e regular).

Assim, as setas do teclado alteram as variáveis globais α e β, as teclas *Page Up* e *Page Down* alteram a variável *radius* e, por fim, a tecla *P* altera o modo de desenho.

3.2.5.1 Eixo das coordenadas

Além das primitivas desenhadas é sempre desenhado um eixo de coordenadas para permitir perceber melhor se os modelos estão direitos. Para isso é usada a função *glBegin(GL_LINES)* e são dados dois pontos para cada eixo através da função *glVertex3f*. Para os eixos serem desenhados de cores diferentes para sabermos qual é qual é usada a função *glColor3f*.

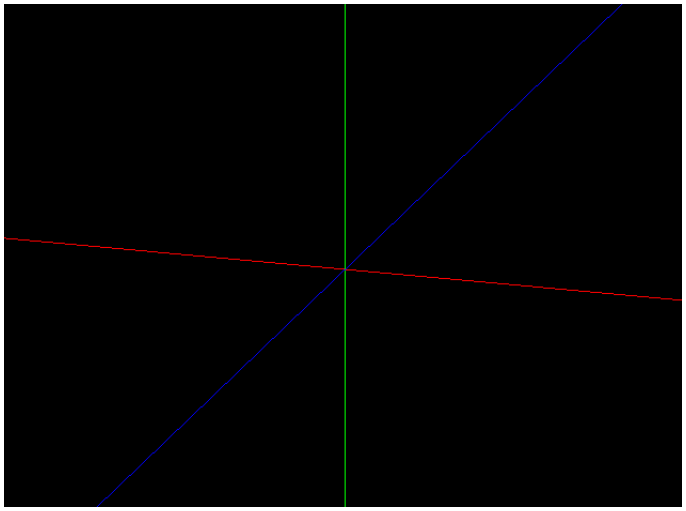

Figura 8: Eixo de coordenadas com o eixo do X a vermelho, o eixo do Z a azul e o eixo do Y a verde

4 Conclusão

Consideramos que a realização desta primeira fase foi bem sucedida e conseguimos cumprir os requisitos estabelecidos primeiramente.

Na implementação do **gerador** não utilizamos uma estrutura de dados para armazenamento dos pontos, mas optamos por escrever no ficheiro à medida que os pontos são calculados. Os ficheiros que são utilizados são do tipo binário, para a leitura e escrita ser facilitada. Estas decisões foram tomadas para haver uma melhoria no desempenho dos programas.

Concluímos que a esta fase do projeto é importante para a consolidação dos temas abordados nas aulas da Unidade Curricular.