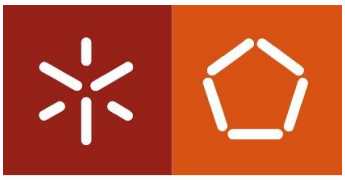

Universidade do Minho

Departamento de Informática

Computação Gráfica
Primitivas gráficas

Bárbara Cardoso (a80453) Marcio Sousa (a82400)
Pedro Mendes (a79003)

12 de março de 2022

Conteúdo

1 Introdução

Este trabalho foi proposto no âmbito da unidade curricular de Computação Gráfica, e tem como objetivo desenvolver um motor gráfico genérico para representar objetos a 3 dimensões.

O projeto está dividido em várias fases, sendo que nesta primeira, serão implementadas algumas primitivas gráficas, como o plano, o paralelepípedo, a esfera e o cone e também uma primeira implementação de dois modos de Câmera.

2 Arquitetura do Projecto

Após analisar o problema, foi decidido que o projeto estaria dividido em dois executáveis, o `generator` e o `engine`. Para além destes, temos também na nossa arquitetura, uma biblioteca comum aos dois.

2.1 Common

Aqui está apenas definido um ficheiro `Point.cpp/hpp` que representa um ponto no espaço 3D.

```
class Point {
    private:
    float _x, _y, _z;
    // -- snip -- /
}
```

2.2 Engine

Este módulo é composto pelos `model.cpp/hpp` que representa cada modelo, ou seja, o conjunto de pontos que define cada um dos modelos, e disponibilizando também um método para os desenhar, e a `main.cpp` que carrega todos os ficheiros especificados no ficheiro `config.xml` (ou o ficheiro passado como argumento do programa) e desenha-os no ecrã. Esta define também a lógica que controla a câmera.

O algoritmo de desenho é simples. Primeiro todos os modelos do ficheiro de configuração são carregados para memoria sobre a forma de `Models`. De seguida cada um destes modelos é renderizado no ecrã, chamando `glVertex3f` para cada um dos seus pontos, pela ordem em que foram lidos, e, por fim, inicia-se o *loop* do *glut*. Desta forma, os modelos são carregados apenas uma vez para memória.

2.3 Generator

Este módulo é composto pelo ficheiro `generator.cpp/hpp` que disponibiliza métodos para gerar os pontos necessários para desenhar as 4 primitivas geométricas e a `main.cpp` que interpreta os argumentos da linha de comandos para chamar estes métodos e produzir um ficheiro com estes argumentos.

O trabalho feito por este módulo é explicado em mais detalhe na secção 3.

2.4 Dependências

A nossa única dependência é o rapidxml (`http://rapidxml.sourceforge.net/manual.html`). Este permite-nos fazer rapidamente a interpretação do ficheiro de configuração, onde estão definidos os objetos a desenhar.

3 Primitivas

Para esta fase do projeto foram desenhadas quatro figuras, contando com um plano, um cubo, uma esfera e um cone. Serão, em seguida, apresentados os algoritmos para o cálculo dos pontos necessários para desenhar cada uma destas primitivas, bem como uma contextualização com as premissas utilizadas durante o raciocínio, quando necessário.

3.1 Plano

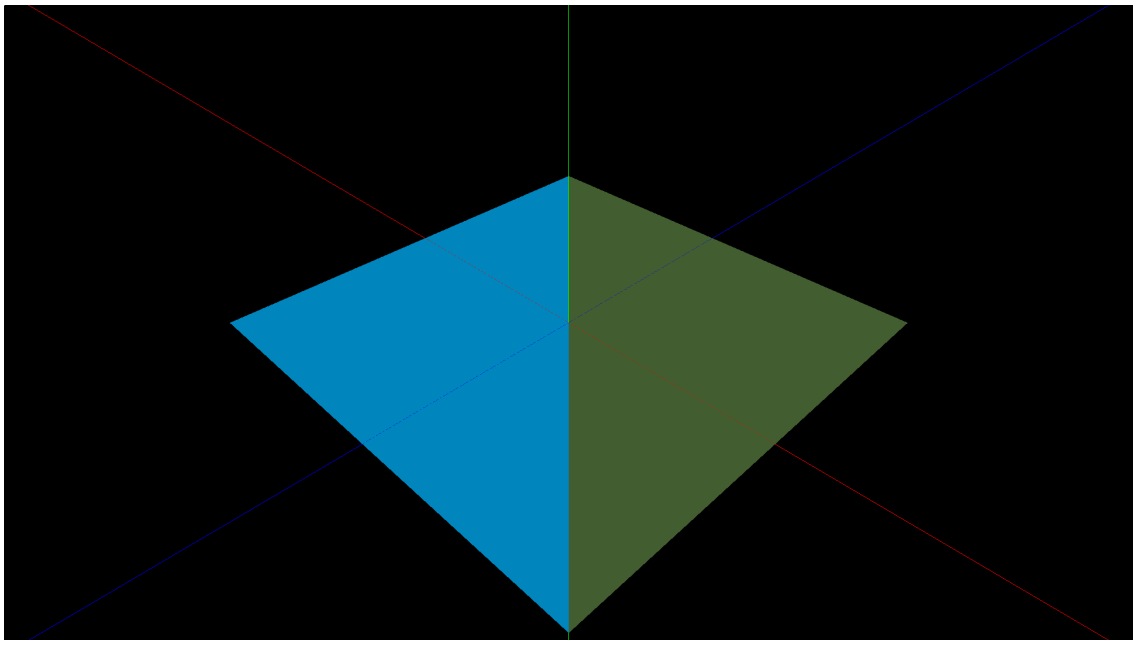

Figura 1: Plano: Comprimento de Lado: 10

O plano é uma figura de extrema simplicidade, composto apenas por 6 pontos que formam 2 triângulos, que por sua vez darão origem ao plano em XoZ.

É recebido como parâmetro o comprimento do lado do quadrado em questão, que será usado para calcular os pontos. Sendo que este plano está centrado na origem, cada um dos pontos é calculado da seguinte forma:

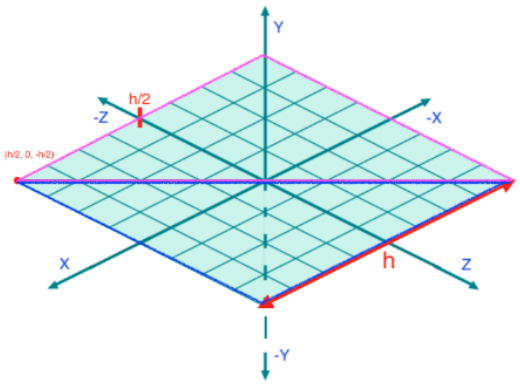

Figura 2: Esquema do Plano.

Para definir o triângulo superior:

$$(h/2, \quad 0, \quad h/2)$$
$$(-h/2, \quad 0, \quad -h/2)$$
$$(-h/2, \quad 0, \quad h/2)$$

Para definir o triângulo inferior:

$$(-h/2, \quad 0, \quad -h/2)$$
$$(h/2, \quad 0, \quad h/2)$$
$$(h/2, \quad 0, \quad -h/2)$$

3.2 Paralelepípedo

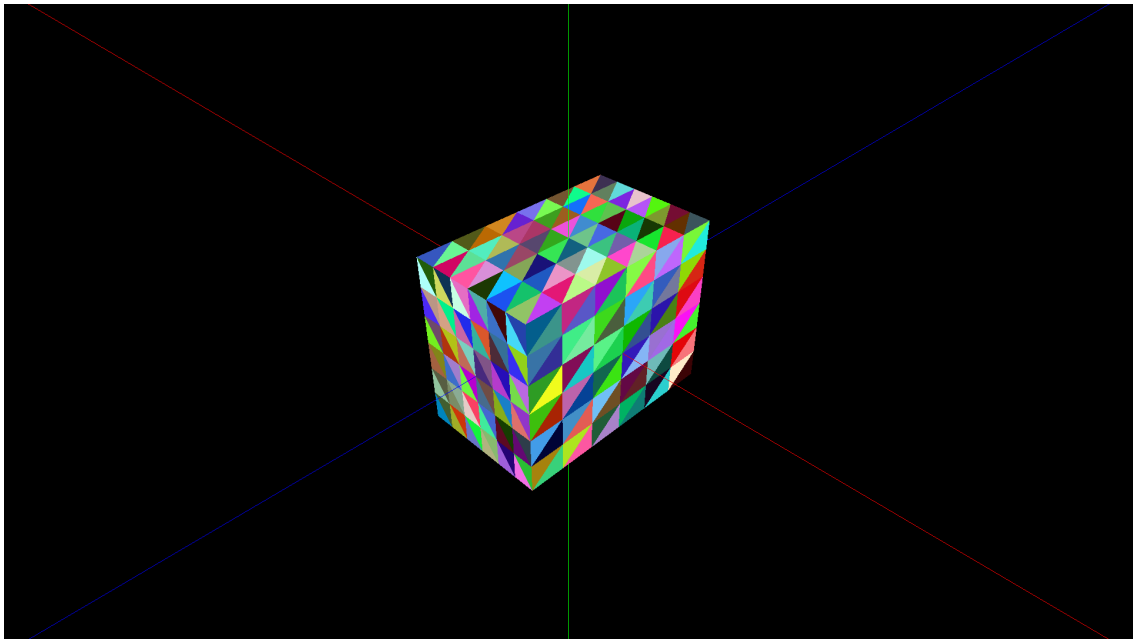

Figura 3: Paralelepípedo: x: 3, y: 4, z: 5, Divisões: 6

Um paralelepípedo é formado por 6 faces, sendo cada face formada por $divisions^2$ quadrados, sendo cada um desses quadrados formado por dois triângulos. Estas divisões são utilizadas para iterar sobre todas as superfícies do paralelepípedo, sendo esta distância entre divisões usada para calcular o próximo ponto baseado no ponto 'atual'.

Tendo o paralelepípedo centrado na origem, por motivos de facilidade de cálculos, calculamos:

$$X_{axis} = x/2$$
$$Y_{axis} = y/2$$
$$Z_{axis} = z/2$$

como 'valores máximos' que podem ser tomados por cada uma das coordenadas dos pontos que constituem o paralelepípedo, e

$$X_{spacing} = x/divisions$$

$$Y_{spacing} = y/divisions$$
$$Z_{spacing} = z/divisions$$

como sendo os valores que, tal como o nome indica, correspondem ao espaçamento entre divisões para cada eixo.

Assim, usando como exemplo um retângulo da face frontal de um paralelepípedo, teríamos as seguintes coordenadas, para o primeiro e segundo triângulo respetivamente:

$$\Big(-X_{axis} + X_{spacing} \times i, \quad -Y_{axis} + Y_{spacing} \times k, \quad Z_{axis} \Big)$$
$$\Big(-X_{axis} + X_{spacing} \times (i+1), \quad -Y_{axis} + Y_{spacing} \times k, \quad Z_{axis} \Big)$$
$$\Big(-X_{axis} + X_{spacing} \times i, \quad -Y_{axis} + Y_{spacing} \times (k+1), \quad Z_{axis} \Big)$$

$$\Big(-X_{axis} + X_{spacing} \times (i+1), \quad -Y_{axis} + Y_{spacing} \times k, \quad Z_{axis} \Big)$$
$$\Big(-X_{axis} + X_{spacing} \times (i+1), \quad -Y_{axis} + Y_{spacing} \times (k+1), \quad Z_{axis} \Big)$$
$$\Big(-X_{axis} + X_{spacing} \times i, \quad -Y_{axis} + Y_{spacing} \times (k+1), \quad Z_{axis} \Big)$$

3.3 Esfera

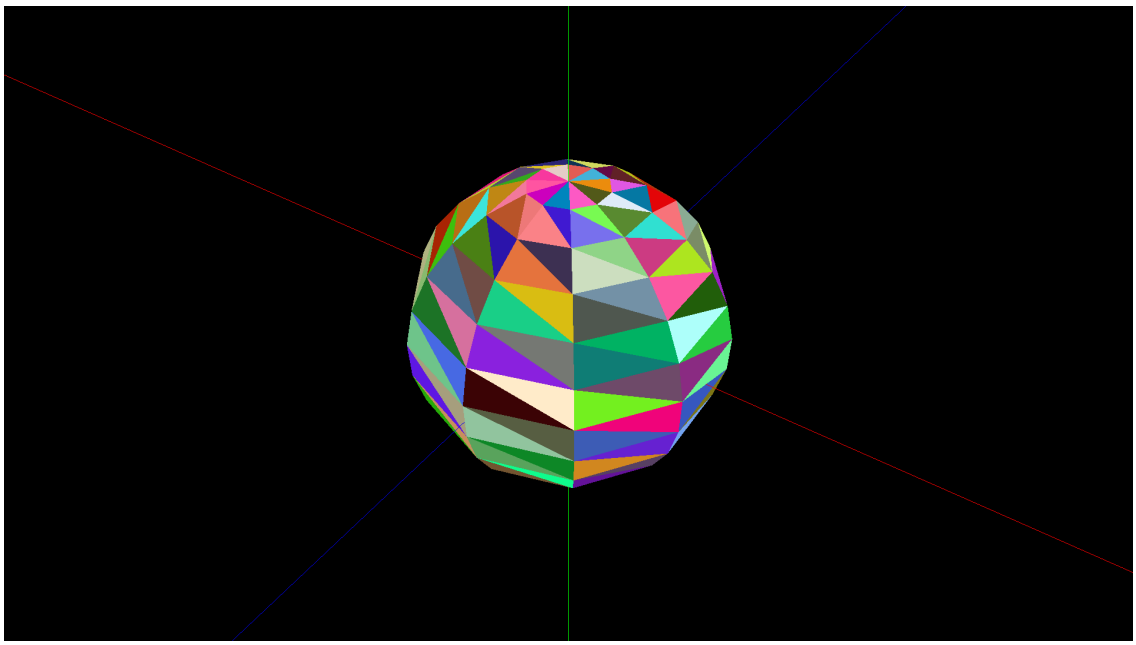

Figura 4: Esfera: Raio: 3, *Slices*: 10, *Stacks*: 13

Uma esfera é dividida em stacks e slices, que serão usadas para iterar a toda a volta da mesma. A interseção entre as slices e as stacks forma retângulos, e cada um destes será constituído por dois triângulos.

Para a criação da esfera serão usadas coordenadas esféricas devido à facilidade que proporcionam, no caso em questão. Assim sendo, são necessárias duas variáveis para representar os dois ângulos que constituem este tipo de coordenadas, ϕ e θ, utilizadas na iteração.

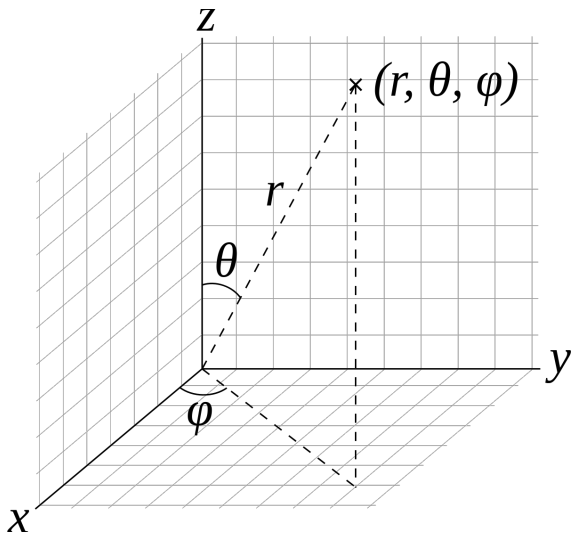

Figura 5: Coordenadas esféricas.

Para o cálculo das coordenadas de cada ponto será também necessário calcular a stack e slice em que se encontram atualmente as iterações, `currentStack` e `currentSlice` que serão calculadas da seguinte forma:

$$currentStack = \theta \times \theta_{Movement}$$

$$currentSlice = \phi \times \phi_{Movement}$$

As variáveis $\theta_{Movement}$ e $\phi_{Movement}$ correspondem, respetivamente, à distância entre Stacks e à distância entre Slices.

$$\phi_{Movement} = 2\pi/slices$$

$$\theta_{Movement} = \pi/stacks$$

Para utilizar coordenadas esféricas, o X, Y e Z devem ser alterados, passando a ser:

$$x = radius \times \sin(\theta) \times \sin(\phi)$$

$$y = radius \times \cos(\theta)$$

$$z = radius \times \sin(\theta) \times \cos(\phi)$$

3.4 Cone

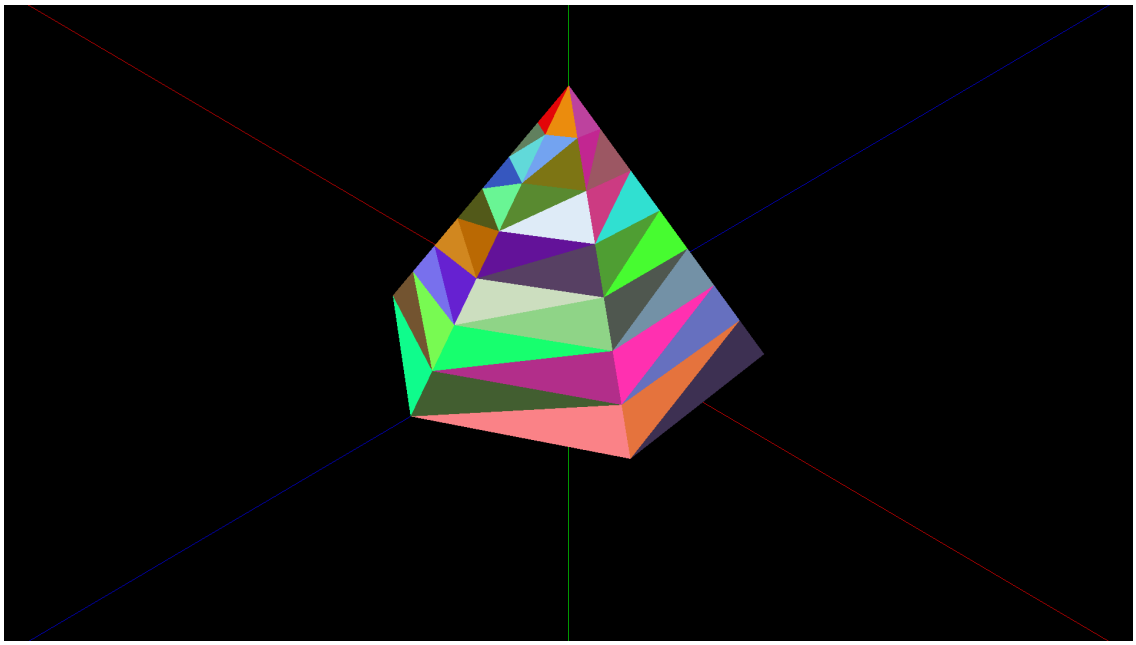

Figura 6: Cone: Raio: 4, Altura: 5, *Slices*: 6, *Stacks*: 7

O cone tem a base sobre o eixo XoZ, centrada na origem, e é dividido em stacks e slices. A função geradora dos pontos do cone recebe como parâmetros o raio, a altura, o número de slices e o número de stacks.

São declaradas três variáveis, ϕ, θ e $stackSpacing$.

A primeira corresponde à divisão equivalente da base por cada slice.

$$\phi = 2\pi/slices$$

A segunda, θ, é calculada dividindo o raio da base pelo número de stacks, o que se traduz na prática em calcular o recuo do raio à medida que se sobe nas stacks.

$$\theta = radius/stacks$$

Por fim, $stackSpacing$, tal como o nome indica, corresponde à distância entre cada stack, o "stack shift", e é calculado dividindo a altura pelo número de stacks.

$$stackSpacing = height/stacks$$

O desenho do cone em si foi dividido em 3 fases: base, topo e a zona em volta, e tal como na esfera, são utilizadas coordenadas esféricas.

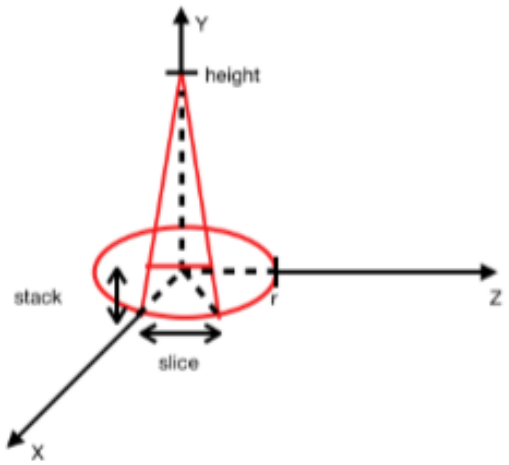

Figura 7: Esquema do cone.

Para a base, quando estamos na primeira stack, em cada iteração é desenhado um triângulo, e o conjunto destes triângulos desenhados formarão a base concêntrica na origem quando tiverem sido executadas todas as iterações. Utilizando coordenadas esféricas chegamos às seguintes coordenadas:

$$\Big(0, \quad 0, \quad 0\Big)$$

$$\Big(radius \times \sin(\phi(k+1)), \quad 0, \quad radius \times \cos(\phi(k+1))\Big)$$

$$\Big(radius \times \sin(\phi k), \quad 0, \quad radius \times \cos(\phi k)\Big)$$

sendo que k toma valores entre 0 e $slices - 1$ e $\phi(k+1)$ representa o próximo valor de ϕ.

O topo é desenhado no fim, para completar a "ponta" do cone, então, quando estamos na penúltima stack (a última é o vértice do cone), são desenhados triângulos a toda a volta que partilham um ponto em comum (o vértice), um por cada slice. Utilizando coordenadas esféricas chegamos às seguintes coordenadas:

$$\Big((radius - \theta i) \times \sin(\phi k), \quad i \times stackSpacing, \quad (radius - \theta i) \times \cos(\phi k)\Big)$$

$$\Big((radius - \theta i) \times \sin(\phi(k+1)), \quad i \times stackSpacing, \quad (radius - \theta i) \times \cos(\phi(k+1))\Big)$$

$$\Big(0, \quad stacks \times stackSpacing, \quad 0\Big)$$

Para os lados, com exceção do topo, serão desenhados retângulos formados por dois triângulos cada um, e serão esses retângulos que constituem o cone até chegar ao topo. Chegamos às seguintes coordenadas:

$$\Big(i(radius - \theta) \times \sin(\phi k), \quad i \times stackSpacing, \quad i(radius - \theta) \times \cos(\phi k)\Big)$$

$$\Big(i(radius - \theta) \times \sin(\phi(k+1)), \quad i \times stackSpacing, \quad (radius - \theta i) \times \cos(\phi(k+1))\Big)$$

$$\Big((radius - \theta(i+1)) \times \sin(\phi k), \quad (i+1) \times stackSpacing, \quad (radius - \theta(i+1)) \times \cos(\phi k)\Big)$$

4 Câmera

A câmera pode ser utilizada em dois modos diferentes, o modo *Explorer mode* e o modo *FPS mode*, que podem ser alternados durante a execução do `engine`.

4.1 Explorer Mode

Neste modo, o movimento da câmera é descrito por uma superfície esférica, na qual o utilizador se movimenta enquanto mantém, sempre, o seu olhar fixo na origem do referencial. O raio desta superfície esférica é que define a distância até ao centro, podendo ser aumentado para afastar o utilizador do centro do espaço 3d, ou diminuído para o aproximar.

4.1.1 Controlos *(key bindings)*

- `H`: Move no plano xz no sentido ao dos ponteiros do relógio.
- `J`: Desce.
- `K`: Sobe.
- `L`: Move no plano xz no sentido contrário ao dos ponteiros do relógio.
- `I`: Reduz o raio da esfera.
- `O`: Aumenta o raio da esfera.

4.2 FPS Mode

Neste modo a posição da câmera no espaço 3d é completamente livre, assim, o utilizador pode mover-se em linha reta em qualquer direção. Pode também alterar o ponto para o qual está a olhar, na horizontal e na vertical.

4.2.1 Controlos *(key bindings)*

- `W`: Move para a frente.
- `A`: Move para a esquerda.
- `S`: Move para trás.
- `D`: Move para a direita.
- `H`: Olha para a esquerda.
- `J`: Olha para baixo.
- `K`: Olha para cima.
- `L`: Olha para a direita.
- `G`: Desce.
- `Shift+G`: Sobe.

4.3 Transição de modos da câmera

Para que a transição seja o mais suave possível foi necessário recorrer à conversão de um sistema de coordenadas para o outro.

Quando a câmera se encontra no modo *Explorer*, a sua posição é determinada através do vetor definido pelos ângulos α e β e através da sua distância à origem. O ponto para onde a câmera olha é a origem. Ou seja, para efetuar a transição para o modo *FPS* foi necessário encontrar um vetor que quando aplicado na posição da câmera, esta fique a olhar para a origem.

$$\alpha_f = \begin{cases} \alpha_e + \pi & \text{se } \alpha_e < 0 \\ \alpha_e - \pi & \text{caso contrário} \end{cases}$$

$$\beta_f = -\beta_e$$

Figura 8: Transição de *Explorer* para *FPS*.

Quando a câmera se encontra no modo *FPS* é necessário encontrar os ângulos α e β e também o raio. Estes valores definem o vetor correspondente às coordenadas (`(x,y,z)`) da câmera no momento da transição.

$$\alpha_e = \begin{cases} \arctan(x/z) & \text{se } z < 0 \\ \arctan(x/z) + \pi & \text{caso contrario} \end{cases}$$

$$\beta_e = \arcsin\left(\frac{y}{\sqrt{x^2+y^2+z^2}}\right)$$

$$radius = \sqrt{x^2+y^2+z^2}$$

Figura 9: Transição de *FPS* para *Explorer*.

4.4 Outros controlos *(key bindings)*

- `+`: Aumenta a escala de todos os modelos.
- `-`: Diminui a escala de todos os modelos.
- `V`: Muda o modo da câmera.
- `Q`: Termina o programa.

5 Gestão de memória

Foi utilizada a ferramenta `Valgrind` para garantir que não havia *leak*s de memória.

Para o gerador foram testadas todas as primitivas:

```
==14866== Command: target/generator/generator plane.3d plane 10
==14866==
==14866==
==14866== HEAP SUMMARY:
==14866==     in use at exit: 0 bytes in 0 blocks
==14866==   total heap usage: 54 allocs, 54 frees, 83,619 bytes allocated
==14866==
==14866== All heap blocks were freed -- no leaks are possible
==14866==
```

```
==14866== For counts of detected and suppressed errors, rerun with: -v
==14866== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

==14867== Command: target/generator/generator box.3d box 20 30 40 6
==14867==
==14867==
==14867== HEAP SUMMARY:
==14867==     in use at exit: 0 bytes in 0 blocks
==14867==   total heap usage: 542 allocs, 542 frees, 263,139 bytes allocated
==14867==
==14867== All heap blocks were freed -- no leaks are possible
==14867==
==14867== For counts of detected and suppressed errors, rerun with: -v
==14867== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

==14868== Command: target/generator/generator sphere.3d sphere 10 100 200
==14868==
==14868==
==14868== HEAP SUMMARY:
==14868==     in use at exit: 0 bytes in 0 blocks
==14868==   total heap usage: 239,468 allocs, 239,468 frees, 68,346,015 bytes allocated
==14868==
==14868== All heap blocks were freed -- no leaks are possible
==14868==
==14868== For counts of detected and suppressed errors, rerun with: -v
==14868== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

==14871== Command: target/generator/generator cone.3d cone 10 6 100 200
==14871==
==14871==
==14871== HEAP SUMMARY:
==14871==     in use at exit: 0 bytes in 0 blocks
==14871==   total heap usage: 238,474 allocs, 238,474 frees, 68,075,587 bytes allocated
==14871==
==14871== All heap blocks were freed -- no leaks are possible
==14871==
==14871== For counts of detected and suppressed errors, rerun with: -v
==14871== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

E para o motor, foi testado todo o código à exceção das rotinas do OpenGl:

```
==18205== Command: target/engine/engine
==18205==
==18205==
==18205== HEAP SUMMARY:
==18205==     in use at exit: 0 bytes in 0 blocks
==18205==   total heap usage: 18,726,897 allocs, 18,726,897 frees,
                               1,353,099,887 bytes allocated
==18205==
==18205== All heap blocks were freed -- no leaks are possible
==18205==
==18205== For counts of detected and suppressed errors, rerun with: -v
==18205== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

6 Conclusões e Trabalho Futuro

Ao longo desta primeira fase foi possível perceber o modo de funcionamento de um motor gráfico 3D e como o OpenGL atua na criação das diferentes figuras geométricas.

Conseguimos fazer com que todas as primitivas sejam geradas corretamente e que o motor gráfico seja capaz de as representar genericamente.

Futuramente, em relação à câmera, esta será melhorada para que se possa utilizar o rato no modo *FPS*.

Fazendo uma análise geral ao trabalho desenvolvido ao longo desta primeira fase, aferimos que foram cumpridos todos os objetivos que nos foram propostos.