Projeto Fractais 3D

Cristiano Vagos 65169 Miguel Brás 49977

***Resumo* – O texto que se segue pretende apresentar a aplicação desenvolvida para a visualização de fractais em 3D.**

O artigo começa por fazer uma apresentação teórica sobre fractais, de seguida apresenta-se os fractais escolhidos para a aplicação e como foi feita a sua implementação.

I. INTRODUÇÃO

A aplicação Fractais 3D foi desenvolvida para a unidade curricular Computação Visual (4º ano, 1º semestre) do Mestrado Integrado em Engenharia de Computadores e Telemática. O objetivo da aplicação é permitir a visualização de alguns fractais simples 3D nomeadamente "Sierpinski Gasket", "Koch Snowflake", "Mosely Snowflake", "Menger Sponge", "Jerusalem Cube" e "Cantor Dust". A aplicação foi desenvolvida em WebGL (sigla para Web Graphics Library), que é uma API (Interface de Programação de Aplicações) desenvolvida em JavaScript para renderizar gráficos 2D e 3D em web browsers compatíveis. Actualmente, o WebGL encontra-se integrado em todos os web browsers permitindo a páginas web mostrarem imagens/objectos gerados pela GPU [2]. Como tal, o WebGL é uma ferramenta útil e bastante importante na área da Computação Visual, e permite uma maior facilidade no desenvolvimento comparado com o OpenGL (Open Graphics Library), API em que o WebGL é baseado.

Na natureza podemos observar certos objetos com repetições de padrões e simetria, como é o caso de um floco de neve (Fig. 1). Os fractais é por exemplo uma forma de podermos simular tais objetos, conforme podemos ver na figura acima descrita.

Os fractais têm, geralmente, um padrão '*self-similar*' [3], isto é, são objetos exatamente similares a uma parte mais pequena deles próprios [4]. Neste artigo vai-se descrever os fractais acima referidos, que descrevem este padrão. Cada parte do fractal sofre o mesmo tipo de padrão sucessivamente, conforme o número de iterações. Também iremos descrever como esses fractais foram implementados usando a tecnologia WebGL.

O objetivo deste trabalho é adquirir e pôr em prática conhecimentos nesta área em crescimento, adquirir conhecimentos de como criar e manipular fractais em 3D com iluminação em real time e aplicar esses conhecimentos numa aplicação que pode ser executada numa página web.

Fig. 1 - Floco de Neve um exemplo clássico de um Fractal Natural [9]

II. SIERPINSKI GASKET

Sierpinski Gasket ou triângulo de Sierpinski é um fractal clássico e simples. Este fractal tem a forma geral de um triângulo equilátero constituído por vários triângulos equiláteros mais pequenos. A ideia do fractal é dividir um triângulo equilátero em quatro triângulos equiláteros, removendo o triângulo central, sendo que depois volta a aplicar-se a mesma operação para cada triângulo gerado. Este é um exemplo de um objeto '*self-similar*'. Foi atribuído, a este fractal, o nome de um matemático Polaco Wacław Sierpiński. [5]

Fig. 2 - Triângulo de Sierpinski (4 iterações)

Este triângulo foi implementado nas aulas de Computação Visual da seguinte forma:

- para cada dois vértices do triangulo encontra-se o seu ponto intermédio;
- Estes novos vértices criados conseguem formar 4

triângulos equiláteros mais pequenos. Se o número de iterações desejado foi alcançado então criam-se os novos triângulos ignorando o central, caso contrário volta a repetir-se o passo anterior para cada conjunto de três vértices que geram os triângulos mais pequenos (ignorando o central).

Em 3D, podemos aplicar a mesma linha de pensamento, tendo simplesmente em atenção que agora a figura é definida por quatro vértices. E fazemos isto com a função com a seguinte assinatura:

```
function divideTetrahedron(v1, v2, v3, v4, recursionLevel)
```

Em que v1, v2, v3 e v4 são os vértices que definem o tetraedro e o recursionLevel indica o numero de iterações desejadas.

A primeira coisa que se verifica é se o nível de iterações chegou ao desejado, caso tenhamos chegado ao número pretendido, então definimos os tetraedros com o código:

```
var coordinatesToAdd = [].concat(v1, v2, v3,
            v3, v2, v4,
            v4, v2, v1,
            v1, v3, v4);
for (var i = 0; i < 36; i += 1) {
    vertices.push(coordinatesToAdd[i]);
}
```

Com este código cria-se um *array* que vai armazenar todas as coordenadas dos vértices que definem cada face, sendo depois estes adicionados ao *array* de vértices a ser desenhado.

Caso não tenhamos chegado ao nível de iterações então teremos que dividir o tetraedro em quatro mais pequenos, ou seja, descobrir os novos vértices que vão definir esses tetraedros. Para isso simplesmente descobrimos o ponto intermédio em cada aresta da figura com a função que tem a assinatura [1]:

```
function computeMidPoint( p1, p2 )
```

Obteremos então 6 novos vértices que irão, juntamente com os originais, definir os novos tetraedros. Diminuímos então o número de iterações restantes e voltamos a invocar a função '`divideTetrahedron`' para cada tetraedro gerado ignorando o tetraedro central.

Após a execução deste código obtemos então uma representação 3D do fractal *Sierpinski Gasket* (Fig. 3)

Figura 3 - *Sierpinski Gasket* 3D (2 iterações)

III. FLOCO DE NEVE KOCH (KOCH SNOWFLAKE)

Outro fractal 2D também realizado nas aulas práticas de Computação Visual foi o Floco de Neve de Koch (ou *Koch Snowflake*) [6], um dos primeiros fractais a ser descrito, como tal, um fractal clássico "obrigatório" a incluir neste trabalho.

Para se obter este fractal faz-se o seguinte:

- Divide-se cada linha em 3 lados iguais;
- Desenha-se um triângulo equilátero cujos vértices da base coincidem com os pontos obtidos no passo anterior.
- Remove-se a base dos novos triângulos.

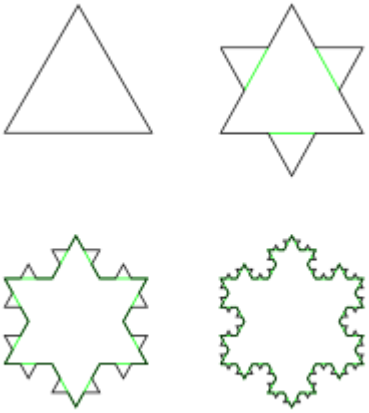

Figura 4 - Floco de Neve de Koch (3 iterações)

Para a passagem em 3D, o processo não é tão direto como o anterior. Há que ter em conta que para cada face é gerado um tetraedro, então para cada face do tetraedro pretendido invoca-se a função:

```
function divideFace( v1, v2, v3, n )
```

Em que v1, v2 e v3 são os vértices que definem a face e 'n' é o numero de iterações pretendidas. Ao inicio verifica-se o numero de iterações, se já chegámos ao pretendidas, então paramos a função.

Caso contrário temos que descobrir os novos vértices do tetraedro a ser gerado nesta face. Facilmente se descobre os vértices da base, visto que estes são descobertos usando a função 'computeMidPoint' para cada aresta. Sabendo que o triângulo gerado por estes novos três pontos é equilátero, então sabe-se que o quarto vértice que dará a

altura do tetraedro estará centrado em relação aos outros vértices, isto é, imaginando que o vértice va, vb e vc que definem o triângulo estão no plano Y = 0, então as coordenadas em X e em Z do vértice xH que define a altura também definem o centro do triângulo. Com isto em mento descobre-se essas novas coordenadas com a função [1]:

```
function computeCentroid( p1, p2, p3 )
```

Agora, saber a posição exata do vértice temos que descobrir a altura do tetraedro. Sabendo que o triângulo é equilátero [7] então criou-se a função:

```
function computeHeightofTriangle(side)
{
    var result = side / 2;
    result = result * Math.sqrt(3);
    return result;
}
```

Esta função dá-nos a altura do tetraedro. Agora para aplicar esta altura no vértice temos que descobrir a normal da face com a função:

```
function computeNormalVector( p0, p1, p2 )
```

Obtendo o vetor unitário normal à face, podemos então multiplicar a altura obtida anteriormente a este vetor. E após adicionarmos este vetor ao centro do triângulo vamos obter o quarto vértice do tetraedro.

Usando o mesmo código que se usou no *Sierpinski Gasket*, adiciona-se os vértices à lista de vértices a desenhar, e para cada face do novo tetraedro invoca-se a função 'divideFace' com o numero de iterações reduzido em 1.

Figura 5 - Floco de Neve de Koch (2 iterações)

IV. FLOCO DE NEVE DE MOSELY (VERSÃO LEVE)

Neste fractal 3D divide-se um cubo em 27 cubos mais pequenos e remove-se os cubos dos cantos. Existem duas versões [8] deste fractal: a versão leve (Fig. 6) – que remove o cubo central, e a versão pesada (Fig. 7) que deixa o cubo pesado. Para esta aplicação escolhemos a versão leve.

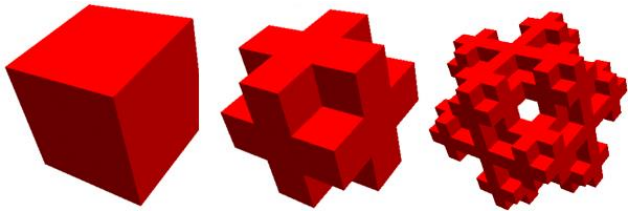

Fig. 6 - Floco de Neve de Mosely leve (2 iterações)

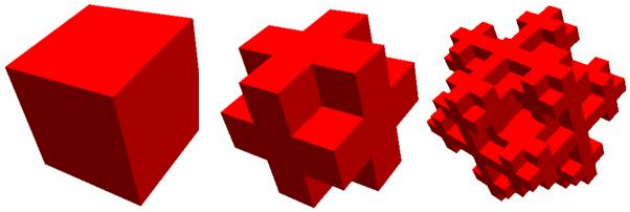

Figura 7 - Floco de Neve de Mosely pesado (2 iterações)

Para implementar este fractal, criou-se a função:

```
function generateMosely(v1, v2, v3, v4, v5,
v6, v7, v8, n)
```

Em que v1, v2, v3, v4, v6, v7 e v8 são os vértices que definem o cubo e 'n' é o numero de iterações desejadas.

Se 'n' for igual a 0, então chegámos ao numero de iterações desejadas e chamamos a função:

```
function defineCube(v1, v2, v3, v4, v5, v6,
v7, v8)
{
    var toAdd = [].concat(v1, v2, v3,
                          v1, v3, v4,
                          v5, v8, v7,
                          v5, v7, v6,
                          v8, v4, v3,
                          v8, v3, v7,
                          v5, v6, v2,
                          v5, v2, v1,
                          v6, v7, v3,
                          v6, v3, v2,
                          v5, v1, v4,
                          v5, v4, v8);
    for (var i = 0; i < toAdd.length; i +=
1){
        vertices.push(toAdd[i]);
    }
}
```

Este é um código análogo ao que foi feito anteriormente para os tetraedros. Adicionam-se todas as coordenadas dos vértices que definem os triângulos das faces dos quadrados e posteriormente adicionam-se esses vértices ao array dos vértices a desenhar.

Se ‘n’ for maior do que 0 então temos que descobrir os vértices que definem os 3x3x3= 27 cubos dentro do cubo original. Para isso usa-se a função interpolate[1]:

```
function interpolate( u, v, s )
```

Em que ‘u’ é o primeiro vértice, ‘v’ é o segundo e ‘s’ é o valor que desejamos interpolar. Escolhendo o valor ‘0.33’, então obteremos o ponto que está a um terço da distância entre ‘u’ e ‘v’. Fazendo isso para todas as combinações possíveis dos vértices do cubo obtemos as coordenadas dos vértices dos 27 cubos interiores.

Após isso decrementa-se o valor de ‘n’ e volta a invocar-se a função ‘generateMosely’ para cada cubo dos interiores excetuando os cubos nos cantos e o central.

V. MENGER SPONGE

O fractal Menger Sponge[9] foi criado por Karl Menger em 1926, e é um fractal 3D que consiste na sub divisão de cubos, retirando os cubos do meio em cada face. Este fractal é a versão 3D do fractal conhecido como carpete de Sierpinski.

Para compreender como este é implementado, temos que perceber como a carpete de Sierpinski é construída.

Começando com um quadrado, divide-se esse quadrado em 9 quadrados iguais, e remove-se o quadrado central. Depois aplica-se a mesma operação para cada quadrado novo.

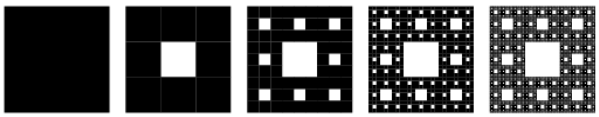

Figura 8 - Carpete de Sierpinski(4 iterações)

Em 3D este fractal é construído começando com um cubo. Chamando a função:

```
function mengerSponge (v1, v2, v3, v4, v5,
v6, v7, v8, recursionLevel)
```

Tal como no Mosely verifica-se se chegámos ao fim das iterações, e caso afirmativo criamos o array de vértices. Caso contrário, tal como a carpete de Sierpinski divide-se cada face em 9 quadrados de igual tamanho, totalizando 27 cubos. É então retirado o cubo do meio de cada face, incluindo o cubo do meio no interior.

Ficamos então com um total de 20 cubos, em que cada um irá seguir o mesmo procedimento de acordo com o número de iterações indicadas para o efeito.

Para obtermos este efeito sucessivo utilizamos uma função recursiva que aplica este método.

Calculamos cada coordenada do cubo pretendido com base no cubo dado, e chamamos a função que irá fazer o mesmo procedimento.

Figura 9 - Menger Sponge (2 iterações)

VI. JERUSALEM CUBE

O fractal Jerusalem Cube é um fractal bastante semelhante ao Menger Sponge. A sua principal diferença assenta no modo como é feito a subdivisão dos cubos. Enquanto que no fractal Menger Sponge todos os cubos têm a mesma dimensão, neste fractal existe um cubo de tamanho inferior no meio das extremidades.

Deste modo é obtida uma cruz em cada face, que não é preenchida. Então, de forma semelhante ao fractal Menger Sponge, começamos com um cubo, subdividimos esse mesmo cubo em 8 cubos, em que 4 deles têm tamanho inferior, ficando estes cubos de tamanho inferior no meio de cada extremidade da face. A seguir o procedimento é semelhante. Na sua forma original, este cubo pequeno tem um rácio de cerca de $(\sqrt{2}-1)^2$ do tamanho da aresta [10]. No desenvolvimento real, consideramos que o tamanho do cubo pequeno é cerca de $\frac{1}{6}$ do tamanho da aresta para facilitar os cálculos, uma vez que o tamanho para ambos os cubos (grande e pequeno) deve ser proporcional ao tamanho da aresta.

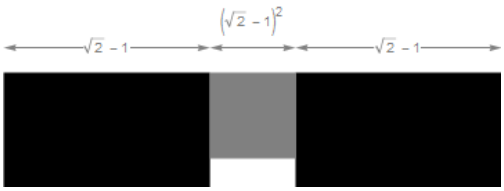

Figura 10 - Comparação de tamanhos Jerusalem Cube

Assim, é feito o mesmo método para cada um dos cubos obtidos a partir do cálculo dos mesmos, obtendo sempre uma cruz no meio de cada face dos cubos.

Tal como anteriormente, utilizamos uma função recursiva para aplicar este método para a construção do fractal. É feito o cálculo das coordenadas do cubo pretendido a partir do cubo dado, e é invocada a mesma função para cada cubo de acordo com o número de iterações pretendida.

Figura 11 - Jerusalem Cube (2 iterações)

VII. CANTOR DUST

O Cantor Dust é o fractal 3D inverso ao Floco de Neve Mosely versão pesada.

A implementação deste fractal é muito semelhante à dos anteriores. O cubo é dividido em 3x3x3 cubos mais pequenos, mas só nos interessa os cubos dos cantos.

Descobre-se os vértices desses quatro cubos e para cada um desses cubos voltamos a invocar a função até o numero de iterações chegar ao pretendido. O método de cálculo dos vértices é bastante semelhante ao dos fractais Menger Sponge e Jerusalem Cube, pelo que reutilizamos o mesmo método de cálculo.

Figura 12 - Fractal Cantor Dust (2 iterações)

VIII. USER INTERFACE

Por usarmos WebGL e outras tecnologias Web como o Javascript, o nosso projeto é executado a partir de uma página Web.

De forma a dar um aspeto agradável à página HTML usada para mostrar a execução, resolvemos usar uma framework conhecida, o Bootstrap [11], que nos permite desenvolver e apresentar uma interface limpa, simples e agradável para interação com o utilizador, através do uso de CSS para customização de estilos de apresentação da interface e Javascript.

Também usamos algumas bibliotecas externas de forma a poder melhorar o aspeto da interface e sua funcionalidade, nomeadamente "Bootstrap Toggle" [12] para personalização de um botão para ativar/desativar as rotações do objecto em torno dos eixos de coordenadas, e "jsColor" [13] para a introdução de um "color picker" que permite selecionar facilmente a cor do objecto gerado (para tal usamos a componente de luz difusa abordada nas aulas)[1].

Descrevendo detalhadamente a interface gráfica da página Web, no lado esquerdo da página encontramos alguns controlos principais a efetuar no objeto, mostrado à direita da página, na tela (canvas).

É possível controlar a rotação em cada um dos eixos disponíveis (XX, YY, ZZ) bem como definir a velocidade e direção dessa mesma rotação. Mais abaixo podemos selecionar o modo de renderização (triângulos, vértices ou arestas/wireframe), bem como o modo de visualização (ortogonal ou perspetiva).

Apresentamos também um botão de reset para retornar às configurações iniciais.

Já no lado direito é possível selecionar o tipo de fractal a ser mostrado na tela, bem como a cor do objeto, que pode ser selecionada facilmente.

Abaixo encontramos a tela que irá mostrar o objeto e executar o WebGL. Inicialmente nenhuma rotação do objeto está ativa, pois é possível mover o objeto através do rato com arrasto.

Também é possível modificar a escala do objeto, usando a roda do rato em cima da tela.

Por baixo da tela vemos o número de iterações do fractal, e botões que permitem aumentar ou diminuir esse mesmo número de iterações.

Achamos que o objetivo da interface foi cumprido, permitindo a qualquer utilizador manipular o objeto gerado de forma fácil e intuitiva.

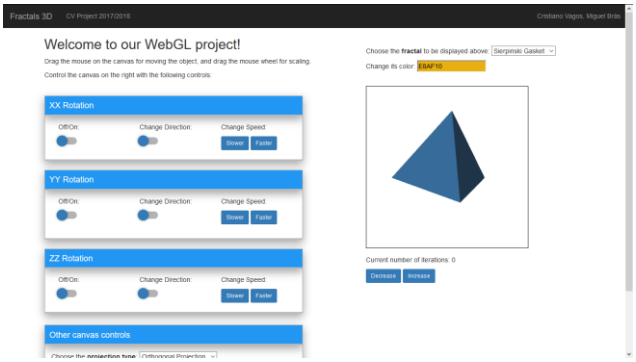

Figura 13 - Aplicação Fractais 3D

IX. CONCLUSÃO

Para todos os fractais aplicados, usou-se uma solução recursiva, que era o que se esperava devido à natureza 'self-similar' dos fractais.

Com isto dá-se por concluído a apresentação da nossa aplicação. Conseguimos recriar alguns fractais simples e ao fazê-lo aprofundar e treinar conhecimentos adquiridos na unidade curricular de Computação Visual, como a criação e manipulação de figuras 3D em runtime, criar uma interface capaz de interagir com os modelos.

Os fractais são em si objetos complexos, mas com utilidade em vários campos, nomeadamente para compreender certos objetos e padrões encontrados na natureza até podem ser usados em campos como filmes e jogos (por exemplo, na criação de um terreno).

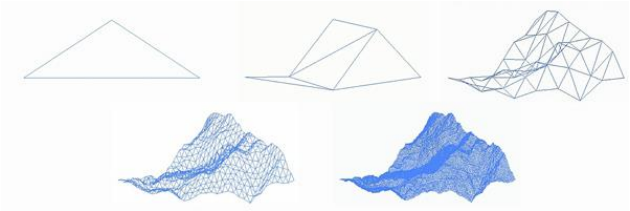

Figura 14 - Fractal a gerar uma Montanha [3]

REFERENCIAS

[1] Joaquim Madeira, http://sweet.ua.pt/jmadeira/WebGL/ em <http://sweet.ua.pt/jmadeira/WebGL/>.

[2] Wikipedia the free encyclopedia, "WebGL" em <https://en.wikipedia.org/wiki/WebGL/>.

[3] Wikipedia the free encyclopedia, "Fractal" em <https://en.wikipedia.org/wiki/Fractal/>.

[4] Wikipedia the free encyclopedia, "Self-similarity" em <https://en.wikipedia.org/wiki/Self-similarity/>.

[5] Wikipedia the free encyclopedia, "Sierpinski triangle" em <https://en.wikipedia.org/wiki/Sierpinski_triangle>.

[6] Wikipedia the free encyclopedia, "Koch Snowflake" em <https://en.wikipedia.org/wiki/Koch_snowflake/>.

[7] Alfa Vritual School, "Triângulos GEO03" em <http://objetoseducacionais2.mec.gov.br/bitstream/handle/mec/10396/geo0303.htm>.

[8] Wikipedia the free encyclopedia, "Mosely Snoflake" em <https://en.wikipedia.org/wiki/Mosely_snowflake/>

[9] Jess McNally, "Earth's Most Stunning Natural Fractal Patterns" em <https://www.wired.com/2010/09/fractal-patterns-in-nature/>

[10] Robert Dickau, "Cross Menger (Jerusalem) Cube Fractal" em <http://www.robertdickau.com/jerusalemcube.html>

[11] Twitter Bootstrap (versão 3.3), em <https://getbootstrap.com/docs/3.3/>

[12] Min Hur, New York Times Developers, "Bootstrap Toggle", em <http://www.bootstraptoggle.com/>

[13] East Desire, jsColor, em <http://jscolor.com/>