Computação Gráfica

Fase 3 - Curvas, Superfícies Cúbicas e *VBOs*

Grupo 10

Carlos Pereira (A61887)
João Barreira (A73831)
Rafael Costa (A61799)

Maio 2017

Índice

1 Introdução

Esta terceira fase consiste na ilustração de animações relativas a uma translação ou rotação.

Uma rotação completa (360 graus) passa a ser efetuada num determinada período de tempo, contrariamente à fase anterior. Já uma translação é definida à custa de uma curva de *Catmull-Rom*, também efetuada num determinado período de tempo.

Para além destes componentes, uma nova primitiva foi adicionada (*Teapot*), sendo esta baseada em *Bezier Patches*. Todas as outras primitivas sofreram alterações no cálculo dos seus vértices. Estas são desenhadas com recurso a *VBOs* com índices.

Neste relatório descrevem-se com detalhe cada uma das componentes da terceira fase. Começa-se por descrever as novas funcionalidades dos dois programas, *Generator* e *Engine*, bem como o modo de utilização das mesmas. De seguida apresentam-se todos os algoritmos efetuados no cálculo dos vértices e índices de todas as primitivas, bem como de uma rotação e uma translação. Finalmente é ilustrado um exemplo do novo modelo do sistema solar.

2 Novas Funcionalidades

2.1 *Generator*

O programa *Generator* passou a suportar novas primitivas com recurso a *Bezier Patches*. De maneira a que programa reconheça que irá receber uma primitiva deste tipo, deve ser executado com o seguinte formato de argumentos ("*Ficheiro_Input* Tesselagem *Ficheiro_Output*"). O ficheiro de *input* é o nome do ficheiro onde se encontram os *patches* necessários para o desenho da primitiva. A Tesselagem representa o valor da precisão com que a primitiva irá ser desenhada e o ficheiro de *output* representa o ficheiro onde serão guardados os vértices e índices calculados. Relativamente ao ficheiro de *input*, este deve seguir um formato específico. A primeira linha deve conter o número de *patches* e, as linhas seguintes, os índices de cada *patch*. Depois de todos os índices deve estar representada na linha seguinte o número de pontos de controlo. Nas restantes linhas devem ser descritos todos estes pontos (um por cada linha), representando-se as suas coordenadas *x, y* e *z*.

Este programa sofreu alterações no cálculo dos vértices de uma primitava. O número de vértices calculados passou a ser muito menor já que estes serão desenhados com recurso a *VBOs*. Além dos vértices, passaram a ser calculados todos os índices que representam a relação entre eles. Posto isto, o formato do ficheiro gerado pelo programa é diferente das fases anteriores. Novamente, cada vértice é representado numa única linha, sendo as suas coordenadas *x, y* e *z* separadas por espaços. No final da representação de todos os vértices, é guardada uma linha em branco no ficheiro, informando o início dos índices. Nas linhas seguintes são armazenados todos os índices calculados (um por linha). Na secção seguinte descrevem-se, com detalhe, o cálculo dos índices de uma primtiva.

2.2 *Engine*

O programa *Engine* passou a estar preprado para desenhar primitivas com recurso a *VBOs* com índices. Para que isso aconteça, o ficheiro relativo a um modelo deve possuir o formato descrito na secção anterior.

Uma rotação e uma translação passaram a ser representadas a partir de animações. Para que o program reconheça uma animação estas operações devem conter um atributo do tipo *time*, em que se o seu valor representa o tempo (em segundos) de uma animação. Numa rotação com animação deixa de ser necessário descrever o seu ângulo, já que esta irá representar um ângulo de 360 graus. Numa translação com animação deixa de ser necessário indicar os atributos *x, y* e *z*. Ao invés disso, deve-se fornecer uma lista de *tags* do tipo *point* (cada uma com um atributo *x, y* e *z*). Como uma translação com animação é definida à custa de uma curva de *Catmull-Rom* devem-se fornecer, no mínimo, quatro pontos.

Além destas novas funcionalidades passa a ser possível desenhar vários modelos do mesmo tipo aleatóriamente numa qualquer área. Para isto foram

acrescentados novos atributos do tipo *rand*, *xzMinR*, *xzMaxR*, *yMinR* e *yMaxR*. O seguinte exemplo representa o desenho de 1000 modelos do tipo *asteroid.3d* aleatóriamente, num raio entre 20 e 22 nos eixos x e z, e com um raio de uma unidade no eixo y.

```
<model file="demos/asteroid.3d" rand=1000
       xzMinR=20 xzMaxR=22 yMaxR=1/>
```

3 *VBOs* com Índices

3.1 *Plane*

Tal como já foi dito nas fases anteriores, o cálculo da primitivas *Plane* é feito a partir das dimensões segundo x e z, que são recebidas como parâmetro. Um plano é formado por dois triângulos e estes, por sua vez, são formados por quatro pontos diferentes. Nesta fase utilizam-se *VBOs* para desenhar o plano. Assim, as coordenadas de cada um dos pontos são calculadas apenas uma vez, sendo imediatamente guardadas no *vector* apropriado:

```
//Ponto A - canto superior direito
vertices.adicionar(Vertice(x, 0, -z))

//Ponto B - canto superior esquerdo
vertices.adicionar(Vertice(-x, 0, -z))

//Ponto C - canto inferior esquerdo
vertices.adicionar(Vertice(-x, 0, z))

//Ponto D - canto inferior direito
vertices.adicionar(Vertice(x, 0, z))
```

Depois definem-se, no *vector* dos índices, a ordem em que cada ponto deverá ser desenhado:

Triângulo defino pelos pontos A-B-C

```
indices.adicionar(A)
indices.adicionar(B)
indices.adicionar(C)
```

Triângulo defino pelos pontos A-C-D

```
indices.adicionar(A)
indices.adicionar(C)
indices.adicionar(D)
```

Isto significa que, para o primeiro triângulo, deverá ser desenhado primeiro o ponto A, depois o ponto B e por fim o ponto C. Para o segundo triângulo o raciocínio é o mesmo: o ponto A é desenhado primeiro, depois o ponto C e no fim o ponto D é desenhado. A ordem pela qual os pontos são desenhados respeita a regra da mão direita, de forma a que as faces da frente do triângulo (e por consequência a do plano) fiquem viradas para a câmara.

3.2 *Box*

A *box* é composta por seis faces diferentes. Por sua vez, cada uma destas faces é dividida num número de divisões, recebido como parâmetro. O desenho desta primitiva divide-se em três fases. Cada uma destas correspondende

ao cálculo dos pontos que constitutem as faces pertencentes ao mesmo plano. Começa-se por cálcular os pontos pertencentes às faces no plano XY, depois às que pertencem ao plano XZ e no fim às faces que se encontram no plano YZ.

O algoritmo aplicado no cálculo dos pontos é quase idêntico para todas as faces, havendo apenas variações no valor das coordenadas. Pode-se considerar uma face da *box* como sendo uma grelha, onde cada elemento dessa grelha resulta da junção de dois triângulos. As divisões recebidas como parâmetro determinam o número de linhas e colunas da grelha. Calculam-se, em primeiro lugar, os pontos pertencentes à linha inferior e vai-se subindo nas linhas à medida que as coordenadas são calculadas.

O método utilizado para calcular os pontos de cada face é igual ao método antes de se utilizarem os *VBOs*. A diferença é na não repetição de pontos, visto que quando se calculam os pontos, se insere, numa estrutura à parte, a ordem pelos quais eles devem ser desenhados.

Pontos que definem um elemento de uma face:

```
//Ponto A - canto superior direito
A = (xA, yA, zA)

//Ponto B - canto superior esquerdo
B = (xB, yB, zB)

//Ponto C - canto inferior esquerdo
C = (xC, yC, zC)

//Ponto D - canto inferior direito
D = (xD, yD, zD)

vertices.adicionar(A)
vertices.adicionar(B)
vertices.adicionar(C)
vertices.adicionar(D)
```

Cada elemento é composto por dois triângulos:

```
indices.adicionar(A)
indices.adicionar(D)
indices.adicionar(C)

indices.adicionar(A)
indices.adicionar(C)
indices.adicionar(B)
```

Aqui, por exemplo, as coordenadas dos pontos A e C apenas são calculadas uma vez e inseridas no *vector* apropriado uma vez. Ao contrário do que acontecia anteriormente, em que tínhamos no *vector* as coordenadas destes pontos o número de vezes que eles fossem precisos. Agora, estes apenas se repete no

vector dos índices, visto ser a partir deste que a *Engine* irá desenhar os pontos pela ordem correta.

3.3 *Cone*

O cálculo dos vértices de um cone pode ser divido em duas partes: cálculo dos vértices da base e cálculo dos vértices dos lados. O número de vértices distintos da base de um cone corresponde ao número de *slices* + 2. Todos os triângulos envolvidos na representação da base de um cone têm um vértice em comum (o vértice do centro da base). Em cada iteração basta apenas calcular-se um novo vértice, já que o vértice calculado na iteração anterior também pertence ao novo triângulo. O algoritmo é o seguinte:

```
// Adicionar o vertice inicial
vertices.adicionar(verticeInicial);

// Adicionar o centro
vertices.adicionar(centro);


// Para cada slice utilizada calculam-se as
//coordenadas dos pontos que a caracterizam.
Para i = 0 ate i < slices fazer i = i + 1 {

// Primeiro ponto pertencente ao triangulo
// (indice da iteracao anterior)
indeces.adicionar(index);

// Segundo ponto pertencente ao triangulo
// que e sempre o centro da base
indexes.adicionar(1)

// Calcula-se o novo vertice
calcularNovoPonto()
vertices.adicionar(novoVertice)

// Insere-se o terceiro ponto
indeces.adicionar(index + 1);
}

// Final dos indices da base
index = index + 1
```

O cálculo da base de um cone é relativamente diferente do cálculo efetuado para todas as outras primitivas. Já o cálculo dos vértices dos lados do cone

obedece à regra da grelha. Ou seja o número de vértices distintos dos lados de um cone é (*stacks* + 1) * (*slices* + 1). O algoritmo é o seguinte:

```
// Calculo dos vertices
Para i = 0 ate i <= stacks fazer i = i + 1 {
    r = calcularRaioStack(i)

   Para j = 0 ate j <= slices fazer j = j + 1 {
       angle = calcularAnguloSlice(j)
      v = calcularVertice(r, angle)
      vertices.adicionar(v)
   }
}

// Calculo dos indices
// A variavel index corresponde ao numero de
// indices calculados para o desenho da base
// cone
Para i = 0 ate i < stacks fazer i = i + 1 {
   Para j = 0 ate j < slices fazer j = j + 1 {
       indeces.adicionar(i * (slices + 1) + j + index)
      indeces.adicionar((i + 1) * (slices + 1) + j + 1 + index)
       indeces.adicionar(i * (slices + 1) + j + 1 + index)

      indeces.adicionar(i * (slices + 1) + j + index)
      indeces.adicionar((i + 1) * (slices + 1) + j + index)
      indeces.adicionar((i + 1) * (slices + 1) + j + 1 + index)
   }
}
```

3.4 *Sphere*

Os pontos que constituem a *sphere* são calculados a partir do ângulo que as normais fazem em relação ao centro. A esfera é dividida em secções de quatro pontos que, por sua vez, definem dois triângulos. Tal como nas primitivas anteriores, o cálculo das coordenadas dos pontos é igual ao das versões anteriores, a diferença reside no facto de se estar a inserir no *vector* dos índices a ordem pela qual estes deverão ser desenhados pela *Engine*.

A partir dos ângulos ao centro (segundo os planos XZ e XY), define-se o ponto de referência da secção a partir de onde se vão calcular as restantes coordenadas.

```
//Ponto A - canto inferior esquerdo
// e ponto de referencia
```

```
A = (xA, yA, zA)

//Ponto B - canto inferior direito
B = (xB, yB, zB)

//Ponto C - canto superior direito
C = (xC, yC, zC)

//Ponto D - canto superior esquerdo
D = (xD, yD, zD)
```

Tendo as coordenadas de todos os pontos da secção, estas são inseridas na estrutura apropriada:

```
vertices.adicionar(A)
vertices.adicionar(B)
vertices.adicionar(C)
vertices.adicionar(D)
```

Feito isto, é adicionado ao *vector* dos índices a ordem pela qual estes pontos deverão ser desenhados:

```
indices.adicionar(A)
indices.adicionar(B)
indices.adicionar(C)

indices.adicionar(A)
indices.adicionar(C)
indices.adicionar(D)
```

Visto que se estão a utilizar *VBOs*, ao contrário do que acontecia na versão anterior, as coordenadas dos pontos A e C já não se encontram repetidas na estrutura *vertices*. Apenas é necessário indicar na estrutura *indices* a ordem pela qual os pontos têm que ser desenhados.

3.5 *Teapot*

O *Teapot* é desenhado à custa de *Bezier Patches*. Através do ficheiro recebido como *input* guardam-se os pontos de controlo num *array* de vértices e os índices desses pontos num *array* de índices.

Para se calcular um ponto de *Bezier Patches* necessitam-se dos dezasseis pontos que definem um *patch* e do valor da tesselagem em duas variáveis distintas (chamemo-lhes u e v). Através destas duas variáveis pode-se concluir que para cada *patch* são calculados $(u + 1) * (v + 1)$ pontos distintos. Como se tratam de pontos com coordenadas x, y e z, são necessárias três matrizes (cada uma com quatro linhas e quatro colunas) para armazenarem, respetivamente,

as dezasseis coordenas em x, y e z. Tal como para as primitivas anteriores, os pontos de um *patch* podem ser vistos como uma grelha, em que $(u + 1)$ representa o número de linhas e $(v + 1)$ o número de colunas dessa grelha. O seguinte algoritmo traduz o ciclo necessário para o cálculo de todos os vértices de um modelo a ser desenhado com recurso a *Bezier Patches*.

```
t = Tesselagem

// Calculo dos vertices

// Cada patch possui 16 pontos
Para s = 0 ate s < numIndexes fazer s = s + 16 {
    Para i = 0 ate i <= t fazer i = i + 1 {

        // Calcula-se o valor de u
        u = i / t

        Para j = 0 ate j <= t fazer j = j + 1 {

            // Calcula-se o valor de v
            v = j / t

            // Criam-se as matrizes com as coordenadas
            // em x, y e z relativas a um certo patch
            criarMatrizX(s)
            criarMatrizY(s)
            criarMatrizZ(s)

            calcularPontoBezier(u, v, matrizX,
                                matrizY, matrizZ)

            vertices.adicionar(ponto)
        }
    }
}


// Calculo dos indices

Para s = 0 ate s < numIndexes / 16 fazer s = s + 1) {

    // Selecao dos indices correspondentes a um patch
   patch = (t + 1) * (t + 1) * s;

   Para i = 0 ate i < t fazer i = i + 1 {
```

```
        Para j = 0 ate j < t fazer j = j + 1 {

            // Ponto A - canto superior esquerdo
            indeces.adicionar(patch + ((t + 1) * i) + j)

            // Ponto B - canto inferior direito
            indeces.adicionar(patch + (t + 1) * (i + 1) + j + 1)

            // Ponto C - canto superior direito
            indeces.adicionar(patch + ((t + 1) * i) + j + 1)

            // Ponto A - canto superior esquerdo
            indeces.adicionar(patch + ((t + 1) * i) + j)

            // Ponto D - canto inferior esquerdo
            indeces.adicionar(patch + (t + 1) * (i + 1) + j)

            // Ponto B - canto inferior direito
            indeces.adicionar(patch + (t + 1) * (i + 1) + j + 1)
        }
    }
}
```

O algoritmo anterior representa as rotinas necessárias para o cálculo de todos os vértices e índices necessários para o desenho de uma primitiva com recurso a *Bezier Patches*. Resta, no entanto, referir os passos necessários para o cálculo de um ponto de uma superfície de *Bezier* através de um valor de u e de v. A seguinte fórmula representa os passos necessários para tais cálculos:

$$\mathrm{B(u,\ v)} = \mathrm{U} * \mathrm{M} * \mathrm{P} * M^T * \mathrm{V}$$

Em que U representa o vetor $\{\ u^3,\ u^2,\ \mathrm{u},\ 1\ \}$, M a matriz de *Bezier*, P a matriz com os dezasseis pontos de um *patch* e V o vetor $\{\ v^3,\ v^2,\ \mathrm{v},\ 1\ \}$. Os cálculos efetuados seguem a ordem da direita para a esquerda, ou seja, o primeiro cálculo consite na multiplicação da matriz M pelo vetor V ($M^T = M$ porque M é uma matriz simétrica). Todos os cálculos consistem na multiplicação de uma matriz por um vetor, à excessão do último em que se multiplica o resultado pelo vetor U, componente a componente. É importante referir que estes cálculos são efetuados três vezes, que correspondem aos cálculos das coordenadas x, y e z de um ponto de *Bezier*. O seguinde pseudo-código ilustra este algoritmo:

```
resMxV = multMatrizVetor(M, V)

// Calculo para a matriz com as coordenadas x
Px = multMatrizVetor(matrixX, resMvezesV)
```

```
// Calculo para a matriz com as coordenadas y
Py = multMatrizVetor(matrizY, resMvezesV)

// Calculo para a matriz com as coordenadas z
Pz = multMatrizVetor(matrizZ, resMvezesV)

// Calcular M * P (x, y e z)
Mx = multMatrizVetor(M, Px)
My = multMatrizVetor(M, Py)
Mz = multMatrizVetor(M, Pz)

// Calculo final das posicoes x, y e z
// de um ponto de Bezier
x = (U[0] * Mx[0]) + (U[1] * Mx[1]) +
    (U[2] * Mx[2]) + (U[3] * Mx[3])

y = (U[0] * My[0]) + (U[1] * My[1]) +
    (U[2] * My[2]) + (U[3] * My[3])

z = (U[0] * Mz[0]) + (U[1] * Mz[1]) +
    (U[2] * Mz[2]) + (U[3] * Mz[3])
```

4 Animações

4.1 Translação

Uma das animações implementadas é a translação que pode ser aplicada a um ou mais modelos. A *tag* referente a esta animação é a seguinte:

```
<translate time=VALOR>
```

Em que *VALOR* corresponde ao valor do tempo total da translação.

Ao nível da nossa implementação, uma translação possui os seguintes campos:

- **x** – Valor de uma translação no eixo dos xx
- **y** – Valor de uma translação no eixo dos yy
- **z** – Valor de uma translação no eixo dos zz
- **catmullPoints** – Lista de pontos para uma curva do tipo *Catmull-Rom*
- **elapsedTime** – Tempo decorrido desde o início da contagem do tempo
- **totalTime** – Tempo total da animação
- **timeAcc** – Tempo decorrida da animação

- **up** – Normal de um modelo

Uma translação foi implementada recorrendo às seguintes funções principais deste módulo:

- **getDeltaTime** – Calcula o tempo decorrido desde a última translação.
- **getGlobalCatmullRomPoint** – Para um instante de tempo, devolve as coordenadas globais de um ponto da curva Catmull-Rom.
- **execute** – Efetua, em *OpenGL*, a translação em x, y ou z com recurso aos pontos da curva *Catmull-Rom*, definida num determinado período de tempo.

Assim sendo, uma translação é efetuada dividindo o tempo total em tempos parciais de igual duração. Para cada um deles são calculadas as coordenadas globais do ponto pertencente à curva *Catmull-Rom* e feita a translação em *OpenGL* correspondente.

Quando o tempo acumulado dos tempos parciais for igual ao tempo total da animação, significa que se deu uma volta completa e, por isso, o valor do tempo acumulado é posto a zero de modo a que se dê mais uma volta.

Após cada uma desta translações em *OpenGL*, é utilizada a normal e os vetores z e *deriv* (todos perpendiculares entre si) para manter a orientação correta do modelo ao longo da animação.

4.2 Rotação

A outra animação implementada é a rotação que pode ser aplicada a um ou mais modelos. A *tag* referente a esta animação é a seguinte:

```
<rotate axisA=VALOR time=TEMPO/>
```

Em que *A* corresponde ao eixo sobre o qual a rotação deve ser feita, *VALOR* a 1 ou -1 consoante o sentido da rotação nesse eixo, e *TEMPO* ao valor, em segundos, que essa rotação deve demorar.

Ao nível da nossa implementação, uma rotação possui os seguintes campos:

- **angle** – Ângulo da rotação (calculado a partir do tempo total)
- **x** – Valor de uma rotação no eixo dos xx
- **y** – Valor de uma rotação no eixo dos yy
- **z** – Valor de uma rotação no eixo dos zz
- **totalTime** – Tempo total de uma rotação
- **timeAcc** – Acumulado de tempo de uma rotação

- **elapsedTime** – Tempo total decorrido de uma rotação

Uma rotação foi implementada recorrendo às seguintes funções principais deste módulo:

- **getDeltaTime** – Calcula o tempo decorrido desde a última rotação.
- **execute** – Efetua, em *OpenGL*, a rotação com um certo ângulo em torno dos eixos dos *xx, yy* ou *zz*.

Assim sendo, uma rotação é efetuada pela função *execute*, dividindo o tempo total da rotação em tempos parciais de igual duração. De seguida e para cada um destes tempos, é efetuada, em *OpenGL*, a rotação correspondente à divisão do tempo total pelo tempo parcial multiplado por 360 (para termos o ângulo).

Quando o tempo acumulado dos tempos parciais for igual ao tempo total da animação, significa que a rotação chegou ao fim e, por isso, colocam-se os tempos a zero.

5 O novo modelo do Sistema Solar

De modo a ilustrar as novas funcionalidades dos dois programas (*Generator* e *Engine*) é descrito com detalhe o novo modelo do Sistema Solar. A estrutura do modelo (os planetas do Sistema Solar) mantém-se inalterada desde as fases anteriores. No entanto, todos os planetas (incluindo o Sol) passam a ser animados com uma rotação. O Sol roda em torno do seu eixo e todos os planetas rodam em torno do Sol. Os tempos de rotação foram inspirados nos tempos de rotação dos planetas do Sistema Solar, tendo estes sido convertidos para uma escala adequada (um ano equivale a cinco segundos). Todos os planetas rodam no sentido contrário ao dos ponteiros do relógio, à excessão de Vénus. Esta rotação em torno do Sol é facilmente conseguida representando-a no ficheiro *XML* antes de qualquer translação que um planeta sofra.

Através da nova funcionalidade de desenho de múltiplos modelos espalhados aleatóriamente numa determinada área, o Sistema Solar passa a incluir a cintura de asteróides. O modelo de um asteróide é uma esfera com poucas *slices stacks*, ilustrando um objeto relativamente pontiagudo. Através deste método, conseguiu-se também representar os anéis de Saturno.

Finalmente, é representado neste modelo um cometa (com a forma de um *Teapot*) através de *Bezier Patches*. O cometa segue uma trajetória calculada a partir de curvas de *Catmull-Rom*. O modelo pode ser visualizado na seguinte figura:

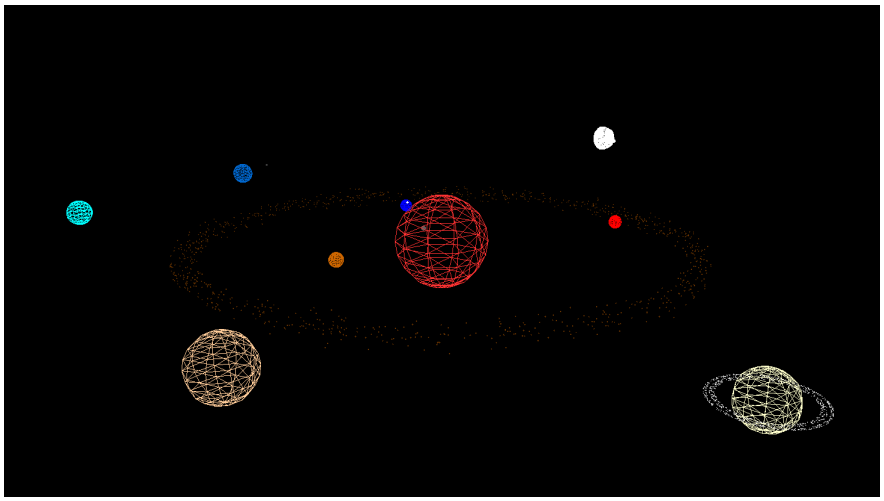

Figura 1 - Modelo do Sistema Solar

6 Conclusão

Nesta terceira fase conseguiu-se explorar os conceitos de animações em translações e rotações. As curvas de *Catmull-Rom* representaram um método simples para o cálculo de todos os pontos que traduzem uma animação de uma translação.

As superfícies cúbicas de *Bezier* permitiram a construção de primitivas com bastante complexidade (neste projeto a primitiva escolhida foi um *Teapot*) através de uma sequência de instruções fácies de concretizar.

Uma grande utilidade a nível de eficiência consistiu no desenho de uma primitiva com recurso a *VBOs*. Este método, especialmente quando acompanhado de uma boa escolha de índices permite um aumento significativo das *fps* de uma qualquer cena.

Finalmente, a hierarquia deste projeto (os dois programas *Generator* e *Engine*, bem como a leitura de ficheiros *XML*) permitiu que se acrescentassem novas funcionalidades bastante facilmente, já que esta hierarquia é bastante genérica.