PL 04- Vertex Buffer Objects

Notas para a componente prática de Computação Gráfica
Universidade do Minho
António Ramires

Com este guião pretende-se explorar os benefícios da utilização de Vertex Buffer Objects (VBOs) para desenhar geometria em OpenGL.

Um VBO pode ser visto como um array residente na memória da placa gráfica. Criar este array implica criar inicialmente um array na memória central com o conteúdo desejado utilizando os métodos típicos em C, e seguidamente copiar esse array para a memória da placa gráfica. Posteriormente é possível utilizar esse array, localizado na memória gráfica, para desenhar geometria com um número reduzido de instruções.

Vejamos um exemplo prático. Queremos desenhar um triângulo com os seguintes vértices: P0 = (-1, 1, 0), P1 = (0, 0, 0), P2 = (1, 1, 0).

Recorrendo ao modo imediato (glBegin – glEnd) poderíamos escrever o seguinte código na função `renderScene`:

```
glBegin(GL_TRIANGLES);
glVertex3f(-1.0f, 1.0f, 0.0f);
glVertex3f(0.0f, 0.0f, 0.0f);
glVertex3f(1.0f, 1.0f, 0.0f);
glEnd();
```

1. VBOs - Preparação

Com VBOs o processo é diferente. Ao iniciar a aplicação criamos o array/vector em C e copiamos para a placa gráfica. Este processo só é realizado uma vez durante a vida da aplicação (excepto se a geometria for dinâmica). Esta fase inicial pode ser realizada numa função chamada a partir da main.

```
void prepareData() {

    // criar um vector com as coordenadas dos pontos

    // criar o VBO

    // copiar o vector para a memória gráfica
}
```

O vector com os pontos ficará com o seguinte conteúdo:

-1.0f	1.0f	0.0f	0.0f	0.0f	0.0f	1.0f	1.0f	0.0f

A primeira parte da função `prepareData` ficará assim:

```
    // criar um vector com os dados dos pontos
    vector<float> p;

    // primeiro ponto
    p.push_back(-1.0f);
    p.push_back(1.0f);
    p.push_back(0.0f);

    // segundo ponto
    ...

    // terceiro ponto
    p.push_back(1.0f);
    p.push_back(1.0f);
    p.push_back(0.0f);

    verticeCount = p.size() / 3;
```

Nota: a utilização de vectores implica que o nome do ficheiro em que estamos a trabalhar tenha a extensão cpp, e ainda o seguinte include:

```
#include <vector>
```

A criação do VBO em OpenGL implica chamar a função `glGenBuffers`

```
    // criar o VBO
    glGenBuffers(1, &vertices);
```

A variável `vertices` deverá ser global já que será necessária na função `renderScene`. A variável `vertices` ficará com o valor 1 por ser o primeiro VBO a ser criado. Este valor representa o número do "slot" alocado e é usado sempre que nos quisermos referir ao VBO. A variável `verticeCount`, também global, irá guardar o número total de vértices. Iremos precisar de ambas estas variáveis para desenhar o triângulo na função `renderScene`.

```
GLuint vertices, verticeCount;
```

Para copiar o vector para o VBO necessitamos primeiro de especificar que o VBO 1 é o VBO activo. Seguidamente podemos copiar o vector para o VBO com a função `glBufferData`.

```
// copiar o vector para a memória gráfica
glBindBuffer(GL_ARRAY_BUFFER, vertices);
glBufferData(
    GL_ARRAY_BUFFER, // tipo do buffer, só é relevante na altura do desenho
    sizeof(float) * p.size(), // tamanho do vector em bytes
    p.data(),  // os dados do array associado ao vector
    GL_STATIC_DRAW); // indicativo da utilização (estático e para desenho)
```

A primeira linha define que o VBO com o índice `vertices` (1) é o VBO activo. A segunda instrução copia os dados do vector para a memória gráfica.

Após a execução da função `prepareData` temos uma cópia do vector na memória gráfica pronto para ser utilizado. A função `prepareData` só deve ser executada uma vez durante a vida da aplicação.

2. VBO – Desenho

Na função `renderScene`, para desenhar o triângulo podemos escrever o seguinte:

```
glBindBuffer(GL_ARRAY_BUFFER, vertices);
glVertexPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, verticeCount);
```

No excerto de código acima primeiro indicamos que `vertices` é o VBO activo. Aqui sim, o tipo do buffer é relevante. Para arrays com propriedades dos vértices, como as posições, o tipo deve ser `GL_ARRAY_BUFFER`.

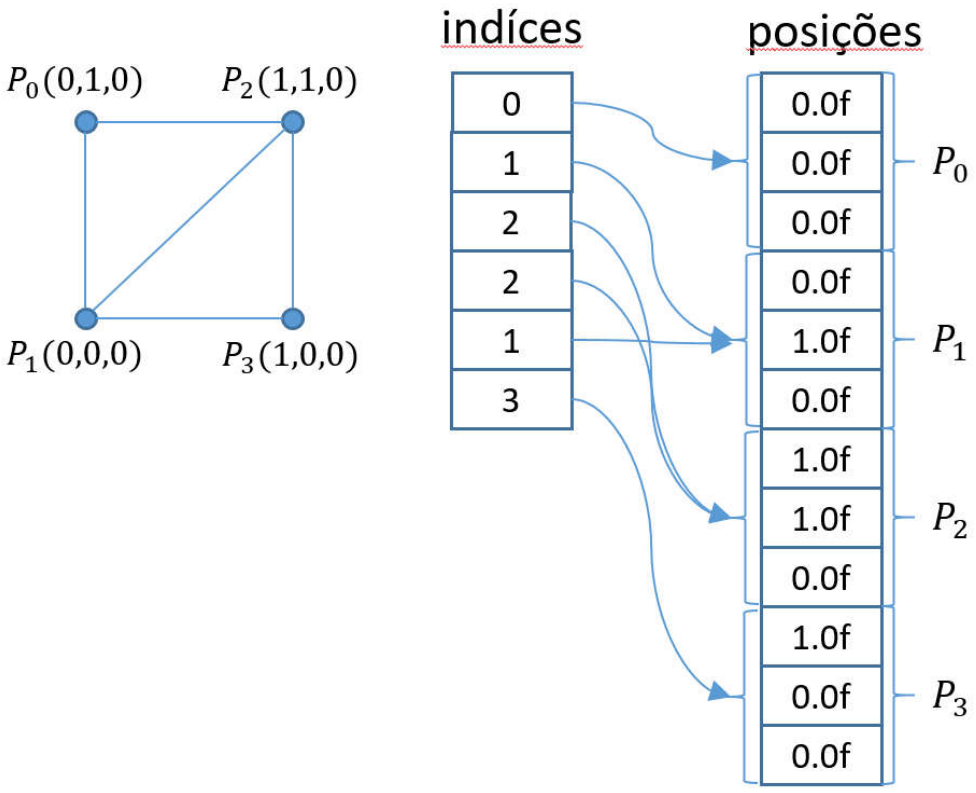
A segunda linha define a semântica dos dados no VBO. Ao copiarmos os dados, com a função `glBufferData`, não passámos nenhuma informação sobre a estrutura dos dados, neste caso que um vértice contem 3 floats. Poderiam ser 4 doubles, ou outra variante qualquer. A função `glBufferData` só copia bytes sem associar nenhuma semântica a esses bytes. A função `glVertexPointer` é usada para fornecer essa semântica (os dois últimos parâmetros não são relevantes para este exemplo). Neste caso estamos a dizer que um vértice é constituído por 3 floats.

Finalmente, com `glDrawArrays`, é realizado o desenho dos triângulos presentes no VBO. O segundo parâmetro indica o vértice inicial, e o último o número de vértices a desenhar.

3. Índices

No guião é também mencionada a utilização de índices. Os índices são utilizados para poupar memória e em certos casos optimizar o processamento, aumentando o desempenho.

Consideremos que se pretende desenhar dois triângulos como na figura seguinte. Pelo processo definido acima necessitaríamos de um array com 6 vértices (3 vértices x 2 triângulos). No entanto dois vértices são repetidos. Através da utilização de índices podemos criar um array só com 4 vértices, e criar em simultâneo um array de índices que define quais os vértices, e por que ordem, a utilizar para desenhar os dois triângulos. Neste caso o array de índices terá 6 posições (sendo o tipo de dados `unsigned int`). Os arrays de índices devem ter o número de elementos divisível por 3. Por exemplo, se criarmos um array de índices com 5 elementos, os últimos dois serão descartados.



Em termos de criação e cópia de arrays/vectores o processo é idêntico. Se considerarmos a definição de um vector i para os índices:

```
    vector<unsigned int> i;
```

depois de preenchido o vector com os índices, a geração do VBO para os índices faz-se do seguinte modo:

```
    glGenBuffers(1, &indices);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indices);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                    sizeof(unsigned int) * i.size(),
                    i.data(),
                    GL_STATIC_DRAW);
     indexCount = i.size();
```

Notem que a variável índices também terá de ser global. Necessitamos ainda de uma variável global com o número de índices a desenhar:

```
GLuint indices;
unsigned int indexCount;
```

Na função `renderScene` teremos de indicar qual o VBO activo para os índices. A instrução de desenho também é diferente:

```
    glBindBuffer(GL_ARRAY_BUFFER,vertices);
    glVertexPointer(3,GL_FLOAT,0,0);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indices);

    glDrawElements(GL_TRIANGLES,
         indexCount, // número de índices a desenhar
         GL_UNSIGNED_INT,  // tipo de dados dos índices
         0);// parâmetro não utilizado
```

A terceira instrução indica qual o VBO activo para os índices. Notem que agora o primeiro parâmetro é `GL_ELEMENT_ARRAY_BUFFER`, indicativo que o VBO será utilizado como índices. Para o desenho com índices usa-se a função `glDrawElements`.

4. Desempenho com e sem VBOs

O desempenho foi testado com um cilindro variando o número de faces.

	Frames por segundo		
Número de lados	Sem VBOS	VBOs sem índices	Com índices

16	8017	8135	8316
1024	3910	8056	8285
262144	30	4296	4370
1048576	8	1486	1386

Há medida que o número de triângulos (4 vezes o número de lados do cilindro) aumenta torna-se clara a vantagem dos VBOs. Os ganhos são muito significativos e resultam de dois factores. Primeiro no modo imediato (sem VBOs) é necessário chamar a função glVertex para cada vértice. Se tivermos em consideração que a chamada de uma função tem um custo, ao multiplicar esse custo por um número crescente de vértices esse custo torna-se significativo. O segundo ponto é que sem VBOs não tiramos partido do potencial paralelismo disponibilizado pelo GPU dado que estamos a desenhar cada triângulo imediatamente ao receber cada conjunto de três vértices.

Com VBOs a comunicação entre o CPU e o GPU é extremamente reduzida na função `renderScene`. Por outro lado, ao pedirmos para desenhar um array de vértices que já se encontra na memória gráfica, torna-se possível tirar real partido do paralelismo do GPU (a distribuição de carga pelos diversos cores do GPU é feita pelo driver).

Quanto à questão dos índices verifica-se que a utilização de índices traz algum benefício no início, mas no último caso o desempenho sem índices torna-se superior. Notem que estes resultados variam de placa para placa, e dependem do GPU e da memória gráfica.

Os dados acima foram obtidos com uma NVIDIA RTX 2080 Ti. Com um NIVIDA GTX 1070 nota-se uma melhoria significativa com a utilização de índices. Por um lado, podemos considerar que sem índices o processamento é sequencial, sem saltos na memória para ir buscar o próximo vértice. Por outro, a utilização de índices permite aproveitar o processamento de vértices anteriores (no entanto é sempre necessário calcular a cor dos pixels, e em regra esse cálculo é mais pesado que o cálculo dos vértices).

O fundamental é experimentar em cada caso concreto.