School of Computer Science and Engineering, Sun Yat-sen University

计算机图形学：作业 4

18364066 卢彦作

January 5, 2021

Contents

1 Introduction

- 使用 VBO 对作业 3 小球进行绘制
  - 使用足够的细分产生充足的顶点和三角面片，便于计算绘制时间
- 讨论是否使用 VBO 的效率区别
- 讨论是否使用 index array 的效率区别

2 Implementation

由于我在作业 3 中已经使用了外部的 obj 文件和 VBO 来绘制作业 3 的小球，根据老师在课堂上的说明，这种情况下我只需要对绘制的时间进行测试，但是我在先前是没有使用 index array 也即缓冲索引对象 (Element Buffer Object，EBO，也叫 Index Buffer Object，IBO) 来对绘制进行加速的，所以下面首先将会在初版上对每一次绘制的时间进行测试，测试在不使用 EBO 情况下程序的运行时间。

然后简单介绍一下 EBO 的使用，说明我在初版上做了哪些修改，最后在添加了 EBO 的程序上再进行测试，测试在使用 EBO 情况下程序的每一次绘制时间。其中测试部分的结果将放到 Result 小节部分，Implementation 小节部分主要讲解如何进行性能测试、EBO 本身和代码在初版上的改进。

2.1 Performance Test

因为绘制的画面比较简单，我所使用的小球 obj 中也只有 960 个顶点，经过测试发现两种方法的差距非常细微，如果是在 Release-x64 运行环境下，二者的 paintGL 函数运行时间几乎没有差别，但是在 Debug-x64 运行环境下还是能看出一些差别，具体结果可在 Result 小节中查看，这种情况的原因可能是因为二者的差别在 Release 中被优化掉了。

至于时间的测试方面，因为 paintGL 的运行时间是微秒级的，使用 ctime 库可能不够精确，所以我使用了 C++11 标准中的一种新计时库即 Chrono，头文件处要添加如下所示的代码：

```
1 #include <chrono>
2 using namespace std;
3 using namespace chrono;
```

然后在 paintGL 函数的头尾分别记录下时间戳，还要维护一个采样次数和平均时间用来计算到目前为止的所有采样平均时间，这样的平均数据更有说服力。

```
int sampling_times = 0;
double average_time = 0.0;
```

```
auto start = system_clock::now();
// ...paintGL
auto end = system_clock::now();
auto duration = duration_cast<microseconds>(end - start);
sampling_times += 1;
auto sec = double(duration.count()) * microseconds::period::num /
    ↪ microseconds::period::den;
average_time = average_time * (sampling_times - 1) / sampling_times + sec /
    ↪ sampling_times;
cout << "Average " << average_time << " seconds." << endl;
```

2.2 Element Buffer Object

前面在作业 3 中已经讲解过 VAO 和 VBO 的基本原理和使用方法，下面将着重讲解 EBO 的原理和使用，部分资料来自于 LearnOpenGL。

回想一下此前我们在课堂上讲到过的图元 GL_TRIANGLES 和 GL_TRIANGLE_STRIP，这两个图元的区别就在于，GL_TRIANGLES 每三个点就构成一个三角形，每一组点之间没有互相重叠，而 GL_TRIANGLE_STRIP 也是每三个点构成一个三角形，但两两组点之间都会重叠两个点，如果我们要构建一个顶点数比较多的物体，我们肯定想要尽可能地少写几行代码。

同理，我们现在就需要绘制一个小球，一个小球由多个三角面片构成，但是三角面片之间是重叠了很多顶点的，但是 obj 文件中还是存放了每一个三角面片的每一个顶点，那么有没有办法让 OpenGL 的渲染管线不要重复地来绘制这么多顶点呢，减少表示一个球所需要的定点数，一方面可以减少数据传输需要的时间，另一方面也减少了渲染管线渲染所有顶点需要的时间。

这个问题的解决方法就是索引缓冲对象 (Element Buffer Object，EBO，也叫 Index Buffer Object，IBO)，和顶点缓冲对象 VBO 一样，EBO 也是一个缓冲，它专门储存索引，OpenGL 调用这些顶点的索引来决定该绘制哪个顶点，所谓的索引绘制 (Indexed Drawing) 正是我们问题的解决方案。

假设现在我们要绘制一个矩形，首先，我们先要定义（不重复的）顶点，和绘制出矩形所需的索引：

```
float vertices[] = {
    0.5f, 0.5f, 0.0f,   // 右上角
    0.5f, -0.5f, 0.0f,  // 右下角
    -0.5f, -0.5f, 0.0f, // 左下角
    -0.5f, 0.5f, 0.0f   // 左上角
};

unsigned int indices[] = { // 注意索引从 0 开始!
    0, 1, 3, // 第一个三角形
    1, 2, 3  // 第二个三角形
};
```

你可以看到，当时用索引的时候，我们只定义了 4 个顶点，而不是 6 个。下一步我们需要创建索引缓冲对象：

```
unsigned int EBO;
glGenBuffers(1, &EBO);
```

与 VBO 类似，我们先绑定 EBO 然后用 glBufferData 把索引复制到缓冲里。同样，和 VBO 类似，我们会把这些函数调用放在绑定和解绑函数调用之间，只不过这次我们把缓冲的类型定义为 GL_ELEMENT_ARRAY_BUFFER。

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

要注意的是，我们传递了 GL_ELEMENT_ARRAY_BUFFER 当作缓冲目标。最后一件要做的事是用 glDrawElements 来替换 glDrawArrays 函数，来指明我们从索引缓冲渲染。使用 glDrawElements 时，我们会使用当前绑定的索引缓冲对象中的索引进行绘制：

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

第一个参数指定了我们绘制的模式，这个和 glDrawArrays 的一样。第二个参数是我们打算绘制顶点的个数，这里填 6，也就是说我们一共需要绘制 6 个顶点。第三个参数是索引的类型，这里是 GL_UNSIGNED_INT。最后一个参数里我们可以指定 EBO 中的偏移量（或者传递一个索引数组，但是这是当你不在使用索引缓冲对象的时候），但是我们会在这里填写 0。

2.3 Code Improvement

首先是要在载入 OBJ 文件的函数部分进行修改，先前该函数只会返回符合 GL_TRIANGLES 的若干三角形顶点，现在我们需要维护两个向量而不是一个了，第一个向量用于存储若干不同位置的顶点，第二个向量用于存储 GL_TRIANGLES 中若干三角形顶点在第一个向量中的索引位置，因此函数返回类型要从 vector<Vertex>，变为 pair< vector<Vertex>,vector<unsigned int> >，接着在存储顶点部分的代码要修改成如下所示：

```
auto iter = find(vertices.begin(), vertices.end(), vert);
if (iter == vertices.end()) {
    vertices.push_back(vert);
    indices.push_back(vertices.size() - 1);
}
else {
    indices.push_back(iter - vertices.begin());
}
```

除此之外，在对 OpenGL 有了更深入的了解以后，我发现 VAO 的绑定其实只需要做一次，而不需要每一次都在 paintGL 这个被反复调用的函数中完成，因此我将相关变量的名称进行修改，并调整了相关函数调用的顺序，让其和 LearnOpenGL 中调用的顺序保持一致，并将只需要完成一次的操作都放到 initializeGL 函数来。再加上一些在先前提到的 EBO 使用相关操作，initializeGL 函数的核心部分如下所示：

```
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

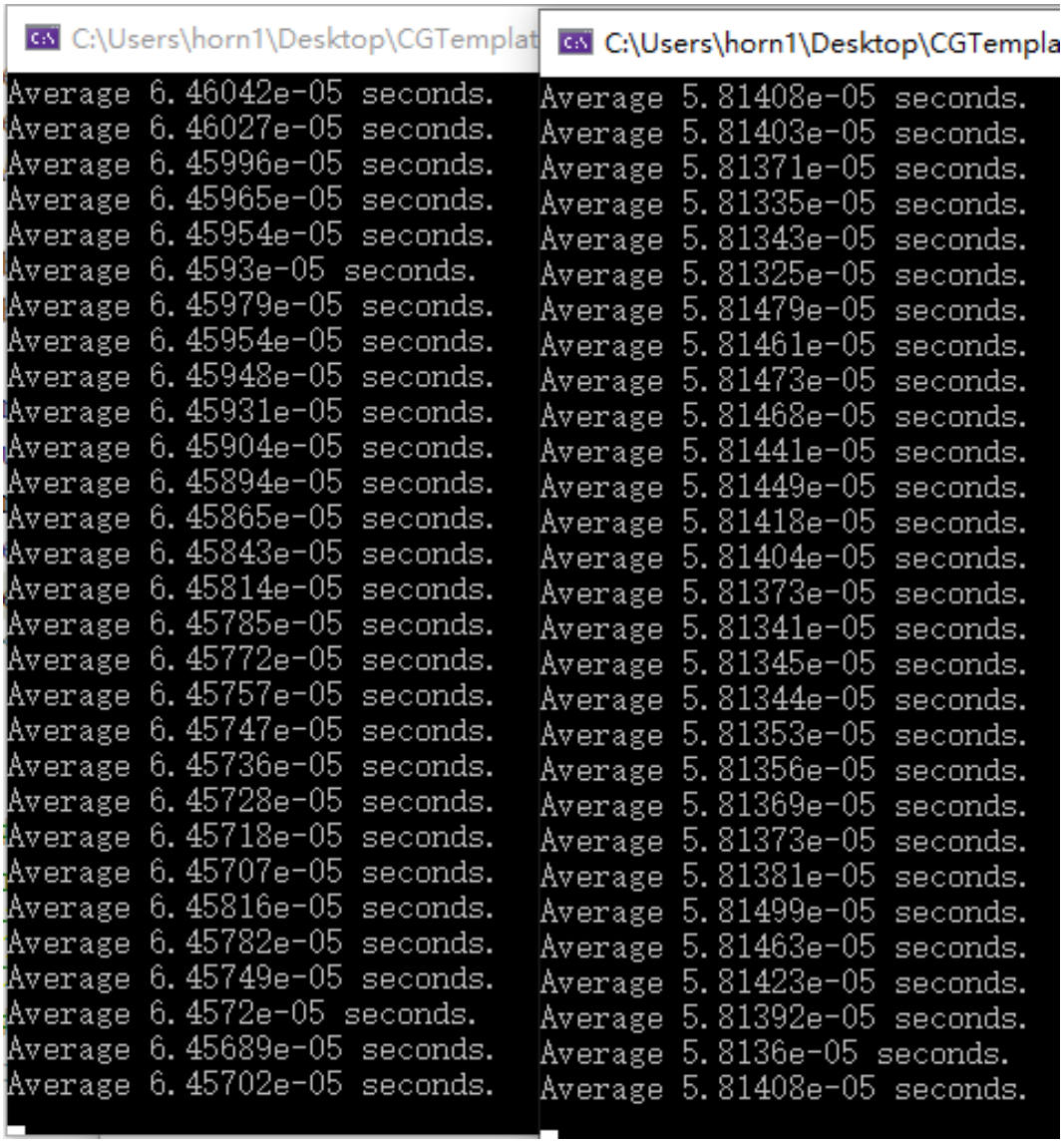
```
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0],
↪ GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
↪ &indices[0], GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)16);
glEnableVertexAttribArray(1);
```

最后在 paintGL 中的剩下操作就非常简单了，一个是使用我们在 initializeGL 已经完成绑定的 VAO，第二个是使用 glDrawElements 来利用 index array 索引数组来绘制画面：

```
// Drawing.
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

3 Result

下面实在 Debug-x64 运行环境下两个版本的 paintGL 函数每一次运行的时间，可以看到在代码改进后，使用了 index array 索引数组以后每一次的绘制时间有所改善，但是经过测试发现这个优势在 Release-x64 运行环境下并不存在，应该还是系统的优化比较给力，如果小球的细分程度更高应该在 Release-x64 环境下也能看出一些差距。


C:\Users\horn1\Desktop\CGTemplat
Average 6.46042e-05 seconds.
Average 6.46027e-05 seconds.
Average 6.45996e-05 seconds.
Average 6.45965e-05 seconds.
Average 6.45954e-05 seconds.
Average 6.4593e-05 seconds.
Average 6.45979e-05 seconds.
Average 6.45954e-05 seconds.
Average 6.45948e-05 seconds.
Average 6.45931e-05 seconds.
Average 6.45904e-05 seconds.
Average 6.45894e-05 seconds.
Average 6.45865e-05 seconds.
Average 6.45843e-05 seconds.
Average 6.45814e-05 seconds.
Average 6.45785e-05 seconds.
Average 6.45772e-05 seconds.
Average 6.45757e-05 seconds.
Average 6.45747e-05 seconds.
Average 6.45736e-05 seconds.
Average 6.45728e-05 seconds.
Average 6.45718e-05 seconds.
Average 6.45707e-05 seconds.
Average 6.45816e-05 seconds.
Average 6.45782e-05 seconds.
Average 6.45749e-05 seconds.
Average 6.4572e-05 seconds.
Average 6.45689e-05 seconds.
Average 6.45702e-05 seconds.
C:\Users\horn1\Desktop\CGTempla
Average 5.81408e-05 seconds.
Average 5.81403e-05 seconds.
Average 5.81371e-05 seconds.
Average 5.81335e-05 seconds.
Average 5.81343e-05 seconds.
Average 5.81325e-05 seconds.
Average 5.81479e-05 seconds.
Average 5.81461e-05 seconds.
Average 5.81473e-05 seconds.
Average 5.81468e-05 seconds.
Average 5.81441e-05 seconds.
Average 5.81449e-05 seconds.
Average 5.81418e-05 seconds.
Average 5.81404e-05 seconds.
Average 5.81373e-05 seconds.
Average 5.81341e-05 seconds.
Average 5.81345e-05 seconds.
Average 5.81344e-05 seconds.
Average 5.81353e-05 seconds.
Average 5.81356e-05 seconds.
Average 5.81369e-05 seconds.
Average 5.81373e-05 seconds.
Average 5.81381e-05 seconds.
Average 5.81499e-05 seconds.
Average 5.81463e-05 seconds.
Average 5.81423e-05 seconds.
Average 5.81392e-05 seconds.
Average 5.8136e-05 seconds.
Average 5.81408e-05 seconds.