游戏研发系列

Cocos2D-X 3.X 3D图形学渲染技术讲解

- 介绍Cocos2D-X 3.X以上版本使用的3D图形学渲染技术，以及3D引擎的架构和模型加密等
- 结合实际案例用最通俗的语言逐一讲解开发者所关心的3D问题
- 适合具备一定游戏开发经验的初学者和3D项目开发经验的游戏开发者阅读

姜雪伟 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

游戏研发系列

Cocos2D-X 3.X 3D图形学渲染技术讲解

姜雪伟 著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书主要介绍 Cocos2D - X 3. X 以上版本使用的 3D 图形学渲染技术，以及关于 3D 引擎的架构和模型加密等，全书共分 12 章，主要内容包括可编程流水线、OpenGL 编程、矩阵变换算法、3D 坐标系统、包围盒算法、3D 架构设计、3D 特效、Shader 渲染、3D 模型渲染、引擎的滤镜渲染、3D 骨骼动画、3D 模型加密。

本书重点介绍 3D 引擎架构设计、Shader 渲染、3D 特效、3D 模型渲染算法及模型骨骼动画。第 12 章介绍了 3D 模型加密算法，在游戏开发中对模型加密是必须要实现的。

本书适合具备一定游戏开发经验的初学者和 3D 项目开发经验的游戏开发者阅读。

图书在版编目（CIP）数据

Cocos2D - X 3. X 3D 图形学渲染技术讲解 / 姜雪伟著. —北京：电子工业出版社，2017. 7
（游戏研发系列）
ISBN 978-7-121-31745-3

Ⅰ. ①C… Ⅱ. ①姜… Ⅲ. ①三维动画软件 - 游戏程序 - 程序设计 Ⅳ. ①TP391. 414
中国版本图书馆 CIP 数据核字（2017）第 124128 号

策划编辑：张 迪
责任编辑：刘真平
印 刷：北京中新伟业印刷有限公司
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787 × 1 092 1/16 印张：16 字数：409. 6 千字
版 次：2017 年 7 月第 1 版
印 次：2017 年 7 月第 1 次印刷
定 价：49. 00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@ phei. com. cn，盗版侵权举报请发邮件至 dbqq@ phei. com. cn。

本书咨询联系方式：（010）88254469；zhangdi@ phei. com. cn。

前言

<<<<< PREFACE

当今，网络游戏和 VR 虚拟现实得到了蓬勃发展，越来越多的 IT 从业者加入到这个行业中。在 2D 游戏开发领域，Cocos2D 引擎已占据大半江山，而 3D 领域几乎已被 Unity 全部占领，Cocos2D - X 引擎在 3D 开发方面的投入明显不足，到现在 3D 引擎版本也只是个半成品，配套的编辑器还没有制作，如果想用它开发 3D 游戏，限制还是很大的。但 Cocos2D 最大的优势是代码开源，而且代码量相对来说比较小，对于想学习 3D 游戏开发的爱好者来说是一个优势。开发者可以在此基础上进行功能扩展，实用性非常强。它不像 UE4 引擎那样代码量很大，初学者不易学习，而且在移动端包体非常大，这也是 UE4 引擎的一个劣势，而 Unity 的包体相对 Cocos2D 包体也是比较大的，代码不开源，所以 Cocos2D 引擎的 3D 模块还是值得开发者学习和借鉴的。随着时间的推移，Cocos2D 中的 3D 模块肯定会独立于 Cocos2D 成为真正的 3D 引擎，因此开发者现在学习非常及时。

Cocos2D - X 在 2D 领域的参考资料非常多，有针对性的书籍也很多，但是关于 Cocos2D 在 3D 开发方面的书籍市面上还没有。本书并不是介绍 Cocos2D 的基础知识以及安装步骤的，而是针对 Cocos2D 中的 3D 模块进行讲解：包括 GPU 编程、3D 模型、3D 模型骨骼动画、3D 模型材质渲染、3D 模型加密等核心技术。衡量 3D 引擎最重要的指标就是渲染，本书围绕 3D 渲染进行讲解，对学习 3D 游戏开发的从业人员非常有帮助。

很多以前从事 2D 游戏开发的人员也在转向 3D 游戏开发，但是他们对 3D 的知识了解得比较少，经验就更无从谈起了。他们对学习 3D 知识非常渴望，就笔者的了解，他们也学习过许多关于 3D 方面知识的书，但是在技能方面提升不明显，笔者曾经也迷茫，不知道该如何学习才能快速提高自己的技能。经过多年的研发积累，笔者通过编写此书帮助初学者快速提升 3D 技能，借助 Cocos2D 中的 3D 模块给想从事 3D 项目开发的人员介绍关于 3D 模块的开发，书中知识点并没有做到面面俱到，但都是最实用的技术，可以直接应用到项目开发中。

任何跨平台 3D 引擎的渲染都是基于 OpenGL 库实现的，它们的底层都是对 OpenGL 的封装，通过对 Cocos2D 引擎中 3D 模块的剖析，让学习者更容易掌握 3D 渲染技术。尤其对于 GPU 编程，很多人对编写 Shader 望而却步，其实如果真正明白其实现原理，掌握起来并没有想象的那么难。本书针对开发者所关心的 3D 问题逐一讲解，用通俗的语言结合实际案例让读者真正明白 3D 渲染技术，本书在讲解中会将图形学算法运用到 Shader 编程中，最终实现非常绚丽的效果。

本书由姜雪伟著，参与本书编写的还有张燕华、姜翠香、吕凤珍、姜瑞庆、张晓宾、朱晓磊和张兴华，在此一并表示感谢。

关于本书的代码，读者可以登录华信教育资源网（http://www.hxedu.com.cn）免费注册后再进行下载。

著　者

目录

CONTENTS

可编程流水线

现今，可编程流水线已经深深地在3D引擎上打下了烙印，目前市面上大部分3D引擎都是基于可编程流水线开发的，也就是用GPU编程开发。提到可编程流水线，就不得不提固定流水线，其实它们的目的是一样的，就是把游戏虚拟场景在屏幕上显示出来，只是它们中间的处理过程是不同的。固定流水线通过名字可以看出它是按照固定的流程绘制出物体的，详情可查看笔者已出版的书籍《手把手教你架构3D游戏引擎》一书，在书里有对固定流水线的详细讲解，以及用固定流水线实现了一个比较弱小的3D游戏引擎。本章重点介绍的是可编程流水线，通过字面意思知道它是“可变”的流水线，意味着它的矩阵计算并不是在CPU中计算得到的，而是通过GPU编程多线程实现的。下面结合可编程流水线流程图，介绍可编程流水线的原理，可编程流水线流程图如图1-1所示。

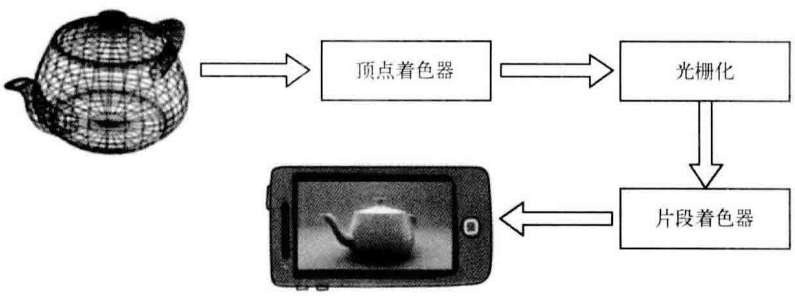

图1-1 可编程流水线流程图

在图1-1中，茶壶模型是美术人员利用Max工具建造的模型，将模型的顶点传输到GPU中的顶点着色器中，进行矩阵变换，并且通过光栅化插值操作，将计算的结果传输到片段着色器中绘制，最终将其输出到手机屏幕上，这整个流程就是可编程流水线。矩阵的变换操作都是在顶点着色器中进行的，为了帮助读者更好地理解可编程流水线，把固定流水线流程图给读者展示出来，如图1-2所示。

该图展示的是固定流水线，之所以称为“固定”，是因为模型在屏幕上显示的过程必须要经历这些固定的操作步骤。可变流水线把矩阵的换算放到GPU中进行了，从而把CPU解放出来，优化了运算效率。

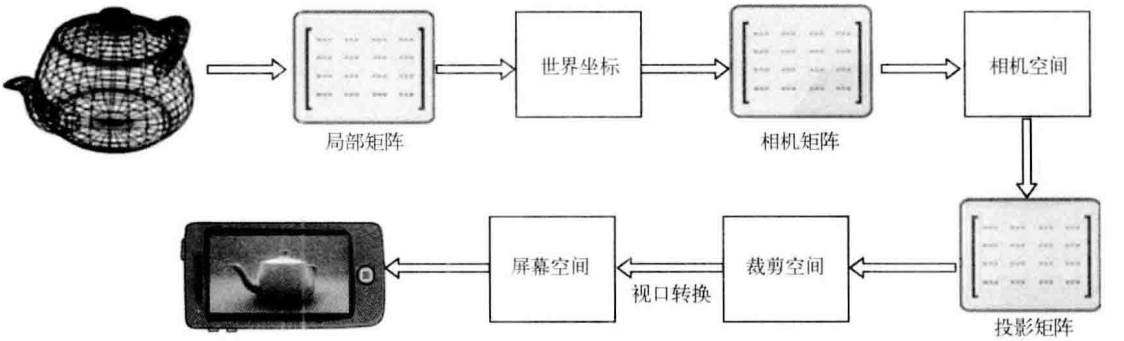

图 1-2　固定流水线

1.1 GPU 功能介绍

GPU（英文全称：Graphics Processing Unit）是嵌入在显卡的芯片处理器，是专门用于处理顶点着色器和片段着色器的。关于顶点着色器和片段着色器会在后面章节中讲解。以前计算机显卡没有 GPU 芯片时，大部分游戏都是基于 2D 游戏或像素游戏开发的，效果可想而知。随着硬件的飞速发展，显卡具有了自己的芯片处理器 GPU，3D 游戏引擎强大的渲染功能也随之发展起来，3D 游戏中各种绚丽多彩的效果都是通过 GPU 编程实现的。比如，玩家常说的次世代网络游戏渲染效果，充分利用了 GPU 的渲染功能，它非常适合对游戏的模型材质渲染和游戏的场景渲染，游戏场景的渲染被称为后处理，这些效果包括：Bloom 效果、Blur 效果、SSAO 效果、PSSM 效果、HDR 效果等。

不论在 PC 端还是在移动端，都可以利用 GPU 编程渲染出好的效果，引擎中最能体现 GPU 强大功能的渲染效果，比如海水的反射、折射、高光效果，笔者在自己研发的项目中已实现过，如图 1-3 所示。

图 1-3　海水的反射、折射效果

下面是笔者利用 Cocos2D－X 引擎中的 3D 渲染模块实现的模型材质渲染，模型的材质含有高光、法线、反射效果，如图 1-4 所示。

图 1-4 含有高光、法线渲染效果图

另外，利用 Cocos2D－X 引擎中的 3D 模块实现的高光、法线、环境映射效果如图 1-5 所示。

图 1-5 高光、法线、环境映射效果

无论是利用商业还是开源 3D 游戏引擎，通过 GPU 编程都可以实现比较绚丽的效果。GPU 的渲染功能还是非常强大的，图 1-4 和图 1-5 实现的 Shader 渲染效果在后面章节中会给读者介绍。

1.2 GPU 编程语言

GPU 功能在模型材质渲染方面这么强大，那如何才能实现材质的渲染，或者说是 GPU 是如何工作的？这就要从 GPU 的编程语言说起，GPU 的编程语言出现了三个派系，传统的图形学领域内使用的 GPU 编程语言有：基于 OpenGL 的 GLSL、基于 DirectX 的 HLSL，还有 Nvidia 公司的 CG。由于目前移动端硬件的快速发展，越来越多的产品在移动端运行，在 GPU 渲染方面也要求具有跨平台编程的 GPU 语言，跨平台的 GPU 语言就是 OpenGL 图形库使用的 GLSL 语言，它也是市面上各个跨平台引擎使用最多的 GPU 编程语言。

对于初学者来说，一提到 Shader 编程都有种望而生畏的感觉，或者说不知道如何下手，其实 GPU 说白了只是一种脚本语言，这种脚本语言类似于 C 语言的语法；另外，在 GPU 编程中使用了顶点着色器和片段着色器，其中在顶点着色器编程中使用了大量的矩阵计算，对于矩阵之间换算不清楚的读者，可以学习一下大学的《线性代数》课程，结合着固定流水线学习效果更佳。作为学习 GPU 编程的初学者来说，要想快速地学习 GPU 编程，笔者推荐先学习 GLSL 编程语言，因其结构类似于 C 语言，更容易上手。GLSL 底层是用 C 语言实现的，GLSL 着色器代码主要分为两部分：VertexShader（顶点着色器）和 Fragment（片段着色器），有时也会在代码中看到 Geometry Shader（几何着色器）。掌握了 GLSL 语言的编写，就可以很轻松地进行各种跨平台引擎的材质渲染和场景渲染。脚本语言都是相通的，CG 语言和 HLSL 语言的语法跟 GLSL 类似，下面会通过案例让大家认识 GLSL 编程语言。

1.3 GPU 编程案例

目前在移动端游戏开发中，大部分引擎的渲染都是用 GLSL 编写脚本语言。GLSL 是为图形算法量身定制的，先介绍 GLSL 编写着色器的内容：在着色器文件的第一行一般要写版本声明，接着是用 uniform 定义的输入和输出变量，以及主函数 main 的实现。每个着色器的入口点都是 main 函数，这跟 C 语言很类似。在 main 函数中处理所有的输入变量，并将结果传输给片段着色器。如果你不知道什么是 uniform 也不用担心，在后面的章节中会进行讲解。下面以顶点着色器代码和片段着色器代码为例，给大家介绍一下 OpenGL 中的 GLSL 脚本的顶点着色器代码编写，如下所示：

```
//position 变量的属性位置值为 0
layout (location =0) in vec3 position;
//为片段着色器指定一个颜色输出
out vec4 vertexColor;
void main()
{
```

```
    //注意如何把一个 vec3 作为 vec4 的构造器的参数
    gl_Position = vec4(position,1.0);
    //把输出变量设置为暗红色
    vertexColor = vec4(0.5f,0.0f,0.0f,1.0f);
}
```

与顶点着色器对应的是片段着色器代码，如下所示：

```
//从顶点着色器传来的输入变量(名称相同、类型相同)
in vec4 vertexColor;
//片段着色器输出的变量名可以任意命名,类型必须是 vec4
out vec4 color;vec4

void main()
{
    //输出颜色
    color = vertexColor;
}
```


下面再把代码给读者解释一下，以上代码也是最基本的 Shader 编程脚本。在顶点着色器中声明了一个 vertexColor 变量作为 vec4 输出，并在片段着色器中声明了一个与顶点着色器相同的变量 vertexColor，用于接收顶点着色器计算的数值，从而让片段着色器中的 vertexColor 和顶点着色器中的 vertexColor 链接起来，从顶点着色器成功地向片段着色器发送数据，这就是顶点着色器和片段着色器成对出现的原因，因为二者是互相关联的。

uniform 是一种从 CPU 中的应用向 GPU 中的着色器发送数据的方式，但 uniform 和顶点属性有些不同。首先，uniform 是全局的（Global）。全局意味着 uniform 变量必须在每个着色器程序对象中都是独一无二的，而且它可以被着色器程序的任意着色器在任意阶段访问。其次，无论你把 uniform 值设置成什么，uniform 会一直保存它们的数据，直到它们被重置或更新。将上面的代码修改成 uniform 修饰的变量，修改后的片段着色器代码如下：

```
out vec4 color;
//在 OpenGL 程序代码中设定这个变量
uniform vec4 ourColor;
void main()
{
    color = ourColor;
}
```

在片段着色器中声明了一个 uniform vec4 的 ourColor，并把片段着色器的输出颜色设置为 uniform 值的内容，因为 uniform 是全局变量，可以在任何着色器中定义它们，而无须通过顶点着色器作为中介。顶点着色器中不需要这个 uniform，所以不用在那里定义它。已在 Shader 中定义的参数需要通过 C++ 代码将参数传递给它，在 C++ 中的代码需要调用 OpenGL 库的接口函数，如下所示：

```
GLfloattimeValue = glfwGetTime( );
GLfloatgreenValue = ( sin( timeValue) / 2) +0. 5;
GLintvertexColorLocation = glGetUniformLocation( shaderProgram," ourColor" );
glUseProgram( shaderProgram);
glUniform4f( vertexColorLocation,0. 0f,greenValue,0. 0f,1. 0f);
```

再把上面函数代码段给读者解释一下：代码段中通过 glfwGetTime 函数获取运行的秒数，然后使用 sin 函数让颜色在 0.0 ~ 1.0 之间变换，最后将结果存储到 greenValue 里。

接着，用 glGetUniformLocation 函数查询 uniform ourColor 的位置值，为查询函数提供着色器程序和 uniform 的名字（这是希望获得的位置值的来源）。如果 glGetUniformLocation 函数返回 -1，则表示没有在着色器中找到这个位置值。否则，通过 glUniform4f 函数设置 uniform 的值。注意，查询 uniform 地址不要求你之前使用过着色器程序，但是更新一个 uniform 声明的变量之前你必须先使用程序（调用 glUseProgram），因为它是激活着色器脚本的。

GPU 编程也称为图形学编程，在程序优化、程序加速方面应用得比较多，因为 GPU 本身是多线程运行的，所以在 Shader 编程中，一般不要使用 if else、for、while 等条件判断或循环语句，这样不利于 GPU 的多线程运行，因为这些条件语句会打断程序的流畅运行，这是编写 Shader 代码时应该注意的问题。下面再说一下编程时的具体问题，如果你声明了一个 uniform 变量，却在 GLSL 代码中没用过，编译器会静默移除这个变量，导致最后编译出的版本中并不会包含它，这可能导致几个非常麻烦的错误。另外，OpenGL 在其核心是一个 C 库，所以它不支持类型重载，在函数参数不同时就要为其定义新的函数。学习 GLSL 编程先从其语法入手，至少保证能看懂其他人编写的代码，这样自己才能在别人编写的基础上做出修改，更进一步自己能够编写 GLSL 语言脚本。

OpenGL编程

市面上，跨平台引擎使用的底层图形库都是 OpenGL，很多人认为 OpenGL 是一个 API（Application Programming Interface，应用程序编程接口），因为它包含了一系列操作图形、图像的函数。实际上，OpenGL 本身并不是一个 API，它是一个由 Khronos 组织制定并维护的规范。OpenGL 规范规定了库中的每个函数如何执行，以及它们的输出值。本书介绍的 OpenGL 面向 OpenGL3.3 以上的版本。

2.1 OpenGL 库介绍

OpenGL 库是用 C 语言编写的，同时也支持多种语言的派生，但其内核仍是一个 C 库。由于 C 的一些语言结构不易被翻译到其他的高级语言，因此 OpenGL 开发的时候引入了一些抽象层。“对象（Object）”就是其中的一个。

在 OpenGL 中一个对象是指一些选项的集合，它代表 OpenGL 状态的一个子集。例如，可以用一个对象来代表绘图窗口的设置，之后就可以设置它的大小、支持的颜色位数等。可以把对象看作一个 C 风格的结构体（Struct）：

```
struct object_name {
    GLfloat   option1;
    GLuint    option2;
    GLchar[ ] name;
};
```

使用 OpenGL 时，建议使用 OpenGL 定义的基元类型。例如使用 float 时会加上前缀 GL（因此写作 GLfloat）、int 写成 GLInt 等。下面通过一段代码介绍如何理解 OpenGL 中的对象概念，如下所示：

```
//创建对象
GLuint objectId = 0;
glGenObject(1,&objectId);
//绑定对象至上下文
glBindObject(GL_WINDOW_TARGET,objectId);
//设置当前绑定到 GL_WINDOW_TARGET 的对象的一些选项
```

```
glSetObjectOption(GL_WINDOW_TARGET,GL_OPTION_WINDOW_WIDTH,800);
glSetObjectOption(GL_WINDOW_TARGET,GL_OPTION_WINDOW_HEIGHT,600);
//将上下文对象设回默认
glBindObject(GL_WINDOW_TARGET,0);
```

给大家解释一下代码含义：这一小段代码展现了使用 OpenGL 编写代码时常见的工作流。首先需要创建一个对象，然后用一个 id 保存它的引用（实际数据被存储在后台），然后将对象绑定至上下文的目标位置（实例中窗口对象目标的位置被定义成 GL_WINDOW_TARGET）。接下来设置窗口的选项，最后将目标位置的对象 id 设回 0，从而解绑这个对象。设置的选项将被保存在 objectId 所引用的对象中，一旦重新绑定这个对象到 GL_WINDOW_TARGET 位置，这些选项就会重新生效。如下语句：

```
glBindObject(GL_WINDOW_TARGET,objectId);
```

函数的功能是绑定对象至上下文，OpenGL 自身是一个巨大的状态机（State Machine）：一系列的变量描述 OpenGL 此刻应当如何运行，OpenGL 的状态通常被称为 OpenGL 上下文（Context）。换句话说，就是当更改 OpenGL 状态，比如设置某些选项后，用 OpenGL 上下文来渲染。下面再介绍一下在 OpenGL 的 Shader 编程中经常使用的着色器。

2.2 着色器介绍

着色器是使用一种叫作 GLSL 的类 C 语言写成的。GLSL 是为图形计算量身定制的，它包含一些针对向量和矩阵操作的特性。

着色器中定义了输入变量和输出变量：uniform、varying、attribute 及 main 函数编写。每个着色器的入口点都是 main 函数，在这个函数中处理所有的输入变量，并将结果输出到已定义的输出变量中。

着色器是各自独立的小程序，它们都是一个整体的一部分，每个着色器都有输入和输出，这样才能进行数据交流和传递。GLSL 定义了 in 和 out 关键字专门来实现这个目的，每个着色器使用这两个关键字设定输入和输出，只要一个输出变量与下一个着色器阶段的输入匹配，它就会传递下去。但在顶点和片段着色器中会有些不同。顶点着色器应该接收的是一种特殊形式的输入，否则就会效率低下。顶点着色器的输入特殊在它从顶点数据中直接接收输入。而片段着色器需要一个 vec4 颜色输出变量，因为片段着色器需要生成一个最终输出的颜色。如果没有在片段着色器中定义输出颜色，OpenGL 就会把物体渲染为默认的黑色（或白色）。

下面介绍顶点着色器和片段着色器是如何结合在一起的，如果开发者打算从一个着色器向另一个着色器发送数据，则必须在发送方着色器中声明一个输出，在接收方着色器中声明一个相同定义的输入。当类型和名字都一样时，OpenGL 就会把两个变量链接到一起，它们之间就能发送数据了（这是在链接程序对象时完成的）。以代码为例说明一下，首先给大家展示的是顶点着色器代码：

```
//position 变量的属性位置值为 0
layout (location = 0) in vec3 position;
//为片段着色器指定一个颜色输出
```

```
out vec4 vertexColor;

voidmain( )
{
    //注意如何把一个 vec3 作为 vec4 的构造器的参数
    gl_Position = vec4( position,1. 0) ;
    //把输出变量设置为暗红色
    vertexColor = vec4( 0. 5f,0. 0f,0. 0f,1. 0f) ;
}
```

顶点着色器中实现了位置及颜色的输出，下面是片段着色器代码：

```
//从顶点着色器传来的输入变量(名称相同、类型相同)
in vec4 vertexColor;
//片段着色器输出的变量名可以任意命名,类型必须是 vec4
out vec4 color;

voidmain( )
{
    color = vertexColor;
}
```


通过顶点着色器和片段着色器代码可以看到，在顶点着色器中声明了一个 vertexColor 变量作为 vec4 输出，并在片段着色器中声明了一个相同的 vertexColor。由于它们名字相同且类型相同，片段着色器中的 vertexColor 就和顶点着色器中的 vertexColor 链接了。它们的链接当然是在 GPU 内部实现的，在这里不需要再继续深入理解，只要知道原理就可以了。由于在顶点着色器中将颜色设置为深红色，最终的片段也是深红色的。实现的效果如图 2-1 所示。

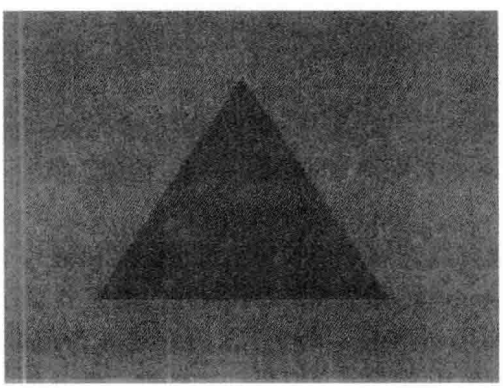

图 2-1 OpenGL 实现的三角形

2.3 OpenGL 属性

在 GPU 编程中，经常需要定义一些变量，这些变量需要一些修饰。通常 Shader 代码编

写中有三个变量修饰定义：uniform、varying、attribute，下面以 uniform 为例给大家介绍一下。uniform 是一种从 CPU 中的应用向 GPU 中的着色器发送数据的方式，但 uniform 和顶点属性有些不同。首先，uniform 是全局的（Global）。全局意味着 uniform 变量必须在每个着色器程序对象中都是独一无二的，而且它可以被着色器程序的任意着色器在任意阶段访问。无论你把 uniform 值设置成什么，uniform 都会一直保存它们的数据，直到它们被重置或更新。下面在一个着色器中添加 uniform 关键字，用来声明一个变量，以下面片段着色器代码为例给读者介绍一下：

```
out vec4 color;
//在 OpenGL 程序代码中设定这个变量
uniform vec4 ourColor;

voidmain( )
{
    color = ourColor;
}
```


在片段着色器中声明了一个 uniform vec4 的 ourColor，并把片段着色器的输出颜色设置为 uniform 值的内容。因为 uniform 是全局变量，可以在任何着色器中定义它们，而无须通过顶点着色器作为中介。在这里友情提示一下，如果声明了一个 uniform 却在 GLSL 代码中没用过，编译器会静默移除这个变量，导致最后编译出的版本中并不会包含它，这可能导致几个非常麻烦的错误。另外，在定义变量时，如果是在 Shader 内部使用的常量定义，就不要用任何声明修饰，直接编写就可以了，例如定义一个三维位置点代码如下：

```
vec3 pos = new vec3(100,100,100);
```

继续讲解上述片段着色器代码，目前 uniform 还是空的，还没有给它添加任何数据，下面的语句就做这件事，首先需要找到着色器中 uniform 属性的索引/位置值。当得到 uniform 的索引/位置值后，就可以更新它的值了。代码如下所示：

```
GLint vertexColorLocation = glGetUniformLocation( shaderProgram," ourColor" );
glUseProgram( shaderProgram);
glUniform4f( vertexColorLocation,0. 0f,1. 0f,0. 0f,1. 0f);
```

用 glGetUniformLocation 函数查询 uniform ourColor 的位置值。主要是为查询到的函数提供着色器程序和 uniform 的名字（这是我们希望获得的位置值的来源）。如果 glGetUniformLocation 返回 -1 就代表没有找到这个位置值。最后，可以通过 glUniform4f 函数设置 uniform 值。注意，查询 uniform 地址不要求你之前使用过着色器程序，但是更新一个 uniform 之前必须先使用程序（调用 glUseProgram），因为它是在当前激活的着色器程序中设置 uniform 的。uniform 变量一般用来表示变换矩阵、材质、光照参数和颜色等信息。有时在 Shader 变量声明中会用到 attribute 定义类型，它只能在顶点着色器中使用，不能在片段着色器中声明 attribute 变量，也不能使用。一般用 attribute 变量来表示一些顶点的数据，如顶点坐标、法线、纹理坐标、顶点颜色等。

在 OpenGL 代码中，一般用函数 glBindAttribLocation 来绑定每个 attribute 变量的位置，然后用函数 glVertexAttribPointer 为每个 attribute 变量赋值。以下是顶点着色器代码实例，给读者展示一下使用 uniform、attribute、varying 三个变量定义的 Shader 代码片段：