**着色语言（Shader Language），以及HLSL VS GLSL VS CG**  
  
摘抄“GPU Programming And Cg Language Primer 1rd Edition” 中文名“GPU编程与CG语言基础”

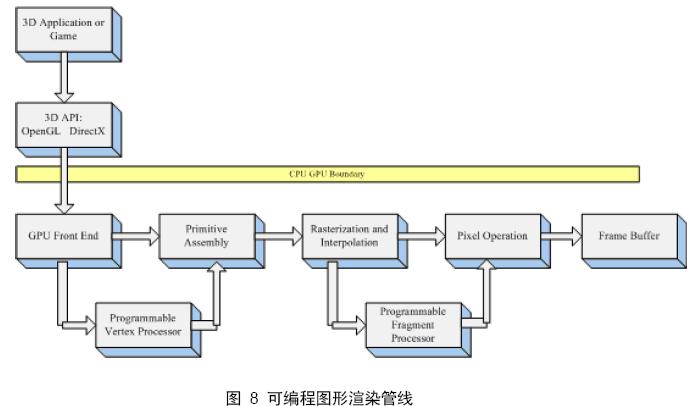
    In the last year I have never had to write a single HLSL/GLSL shader. Bottom line, I can’t think of any reason NOT to use CG.

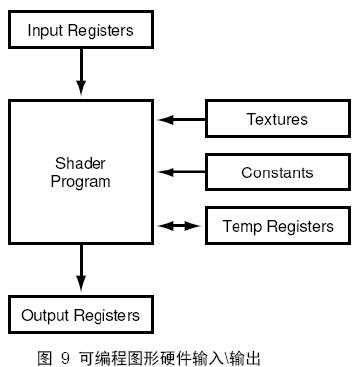
    shader language，称为着色语言，shade在英语是阴影、颜色深浅的意思，Wikipedia上对shader language的解释为“The job of a surface shading procedure is to choose a color for each pixel on a surface, incorporating any variations in color of the surface itself and the effects of lights that shine on the surface(Marc Olano)”，即，shader language基于物体本身属性和光照条件，计算每个像素的颜色值。

    实际上这种解释具有明显的时代局限性，在GPU编程发展的早期，shader language的提出目标是加强对图形处理算法的控制，所以对该语言的定义亦针对于此。但随着技术的进步，目前的shader language早已经用于通用计算研究。

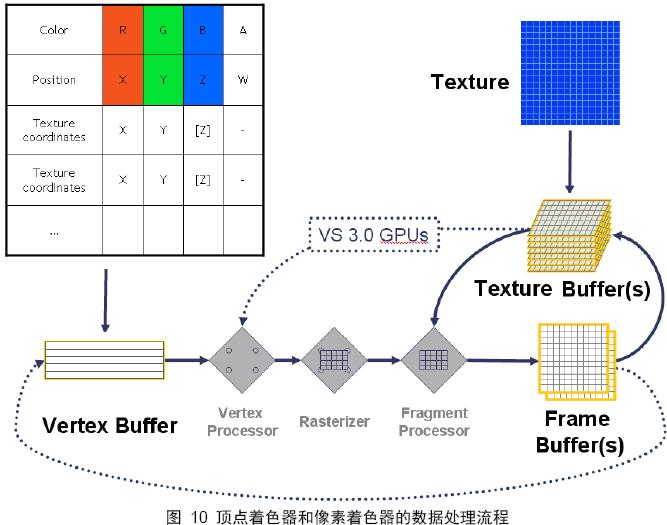
    shader language被定位为高级语言，如，GLSL的全称是“High Level Shading Language”，Cg语言的全称为“C for Graphic”，并且这两种shader language的语法设计非常类似于C语言。不过高级语言的一个重要特性是“独立于硬件”，在这一方面shader language暂时还做不到，shader language完全依赖于GPU构架，这一特征在现阶段是非常明显的！任意一种shader language都必须基于图形硬件，所以GPU编程技术的发展本质上还是图形硬件的发展。在shader language存在之前，展示基于图形硬件的编程能力只能靠低级的汇编语言。  
    目前，shader language的发展方向是设计出在便捷性方面可以和C++\JAVA相比的高级语言，“赋予程序员灵活而方便的编程方式”，并“尽可能的控制渲染过程”同时“利用图形硬件的并行性，提高算法的效率”。Shader language目前主要有3种语言：基于OpenGL的GLSL，基于Direct3D的HLSL，还有NVIDIA公司的Cg 语言。  
    本章的目的是阐述shader language的基本原理和运行流程，首先从硬件的角度对Programmable Vertex Processor（可编程顶点处理器，又称为顶点着色器）和 Programmable Fragment Processor（可编程片断处理器，又称为片断着色器）的作用进行阐述，然后在此基础上对vertex program和fragment program进行具体论述，最后对GLSL、HLSL和Cg进行比较。

3.1 Shader Language原理

    使用shader language编写的程序称之为shader program（着色程序）。着色程序分为两类：vertex shader program（顶点着色程序）和fragment shader program（片断着色程序）。为了清楚的解释顶点着色和片断着色的含义，我们首先从阐述GPU上的两个组件：Programmable Vertex Processor（可编程顶点处理器，又称为顶点着色器）和 Programmable Fragment Processor（可编程片断处理器，又称为片断着色器）。文献[2]第1.2.4节中论述到：  
    The vertex and Fragment processing broken out into programmable units. The Programmable vertex processor is the hardware unit that runs your Cg Vertex programs, whereas the programmable fragment processor is the unit that runs your Cg fragment programs.  
    这段话的含义是：顶点和片段处理器被分离成可编程单元，可编程顶点处理器是一个硬件单元，可以运行顶点程序，而可编程片段处理器则是一个可以运行片段程序的单元。  
顶点和片段处理器都拥有非常强大的并行计算能力，并且非常擅长于矩阵（不高于4阶）计算，片段处理器还可以高速查询纹理信息（目前顶点处理器还不行，这是顶点处理器的一个发展方向）。  
如上所述，顶点程序运行在顶点处理器上，片段程序运行在片段处理器上，哪么它们究竟控制了GPU渲染的哪个过程。图 8展示了可编程图形渲染管线。  


    对比上一章图 3。中的GPU渲染管线，可以看出，顶点着色器控制顶点坐标转换过程；片段着色器控制像素颜色计算过程。这样就区分出顶点着色程序和片段着色程序的各自分工：Vertex program负责顶点坐标变换；Fragment program负责像素颜色计算；前者的输出是后者的输入。  
    图 9展示了现阶段可编程图形硬件的输入\输出。输入寄存器存放输入的图元信息；输出寄存器存放处理后的图元信息；纹理buffer存放纹理数据，目前大多数的可编程图形硬件只支持片段处理器处理纹理；从外部宿主程序输入的常量放在常量寄存器中；临时寄存器存放着色程序在执行过程中产生的临时数据。  
  


3.2 Vertex Shader Program

    Vertex shader program（顶点着色程序）和Fragment shader program（片断着色程序）分别被Programmable Vertex Processor（可编程顶点处理器）和 Programmable Fragment Processo（可编程片断处理器）所执行。  
    顶点着色程序从GPU前端模块（寄存器）中提取图元信息（顶点位置、法向量、纹理坐标等），并完成顶点坐标空间转换、法向量空间转换、光照计算等操作，最后将计算好的数据传送到指定寄存器中；然后片断着色程序从中获取需要的数据，通常为“纹理坐标、光照信息等”，并根据这些信息以及从应用程序传递的纹理信息（如果有的话）进行每个片断的颜色计算，最后将处理后的数据送光栅操作模块。  
图 10展示了在顶点着色器和像素着色器的数据处理流程。在应用程序中设定的图元信息（顶点位置坐标、颜色、纹理坐标等）传递到vertex buffer中；纹理信息传递到texture buffer中。其中虚线表示目前还没有实现的数据传递。当前的顶点程序还不能处理纹理信息，纹理信息只能在片断程序中读入。  
顶点着色程序与片断着色程序通常是同时存在，相互配合，前者的输出作为后者的输入。不过，也可以只有顶点着色程序。如果只有顶点着色程序，那么只对输入的顶点进行[来源：GameRes.com]操作，而顶点内部的点则按照硬件默认的方式自动插值。例如，输入一个三角面片，顶点着色程序对其进行phong光照计算，只计算三个顶点的光照颜色，而三角面片内部点的颜色按照硬件默认的算法（Gourand明暗处理或者快速phong明暗处理）进行插值，如果图形硬件比较先进，默认的处理算法较好（快速phong明暗处理），则效果也会较好；如果图形硬件使用Gourand明暗处理算法，则会出现马赫带效应（条带化）。  
而片断着色程序是对每个片断进行独立的颜色计算，并且算法由自己编写，不但可控性好，而且可以达到更好的效果。  
    由于GPU对数据进行并行处理，所以每个数据都会执行一次shader程序程序。即，每个顶点数据都会执行一次顶点程序；每个片段都会执行一次片段程序。  


3.3 Fragment Shader Program

    片断着色程序对每个片断进行独立的颜色计算，最后输出颜色值的就是该片段最终显示的颜色。可以这样说，顶点着色程序主要进行几何方面的运算，而片段着色程序主要针对最终的颜色值进行计算。  
片段着色程序还有一个突出的特点是：拥有检索纹理的能力。对于GPU而言，纹理等价于数组，这意味着，如果要做通用计算，例如数组排序、字符串检索等，就必须使用到片段着色程序。让顶点着色器也拥有检索纹理的能力，是目前的一个研究方向。  
    附：什么是片断？片断和像素有什么不一样？所谓片断就是所有的三维顶点在光栅化之后的数据集合，这些数据还没有经过深度值比较，而屏幕显示的像素都是经过深度比较的。

3.4 CG VS GLSL VS HLSL

    Shader language目前有3种主流语言：基于OpenGL的GLSL（OpenGL Shading Language，也称为GLslang），基于Direct3D的HLSL（High Level Shading Language），还有NVIDIA公司的Cg （C for Graphic）语言。  
    GLSL与HLSL分别提基于OpenGL和Direct3D的接口，两者不能混用，事实上OpenGL和Direct3D一直都是冤家对头，曹操和刘备还有一段和平共处的甜美时光，但OpenGL和Direct3D各自的东家则从来都是争斗不休。争斗良久，既然没有分出胜负，那么必然是两败俱伤的局面。  
首先ATI系列显卡对OpenGL扩展支持不够，例如我在使用OSG（Open Scene Graphic）开源图形引擎时，由于该引擎完全基于OpenGL，导致其上编写的3D仿真程序在较老的显卡上常常出现纹理无法显示的问题。其次GLSL 的语法体系自成一家，而HLSL和Cg语言的语法基本相同，这就意味着，只要学习HLSL和Cg中的任何一种，就等同于学习了两种语言。不过OpenGL 毕竟图形API的曾经领袖，通常介绍OpenGL都会附加上一句“事实上的工业标准”，所以在其长期发展中积累下的用户群庞大，这些用户当然会选择 GLSL学习。此外，GLSL继承了OpenGL的良好移植性，一度在unix等操作系统上独领风骚（已是曾经的往事）。  
    微软的HLSL移植性较差，在windows平台上可谓一家独大，可一出自己的院子（还好院子够大），就是落地凤凰不如鸡。这一点在很大程度上限制了 HLSL的推广和发展。目前HLSL多半都是用于游戏领域。我可以负责任的断言，在Shader language领域，HLSL可以凭借微软的老本成为割据一方的诸侯，但，决不可能成为君临天下的霸主。这和微软现在的局面很像，就是一个被带刺鲜花簇拥着的大财主，富贵已极，寸步难行。  
    上面两个大佬打的很热烈，在这种情况下可以用一句俗话来形容，“鹬蚌相争，渔翁得利”。NVIDIA是现在当之无愧的显卡之王（尤其在AMD兼并ATI之后），是GPU编程理论的奠基者，GeForce系列显卡早已深入人心，它推出的Cg语言已经取得了巨大的成功，生生形成了三足鼎立之势。NVIDIA公司深通广告之道，目前最流行的GPU编程精粹一书就出自该公司，书中不但介绍了大量的GPU前沿知识，最重要的是大部分都用Cg语言实现。凭借该系列的书籍，NVIDIA不光确定了在青年学子间的学术地位，而且成功的推广了Cg语言。我本人就是使用Cg语言进行研发，基于如下理由：

    其一，Cg是一个可以被OpenGL和Direct3D广泛支持的图形处理器编程语言。 Cg语言和OpenGL、DirectX并不是同一层次的语言，而是OpenGL和DirectX的上层，即，Cg程序是运行在OpenGL和 DirectX标准顶点和像素着色的基础上的；  
    其二，Cg语言是Microsoft和NVIDIA相互协作在标准硬件光照语言的语法和语义上达成了一致，文献[1]在1.3.1节的标题就是 “Microsoft and NVIDIA’s Collaboration to Develop Cg and HLSL”，所以，HLSL和Cg其实是同一种语言（参见Cg教程\_可编程实时图形权威指南29页的致谢部分）。很多时候，你会发现用HLSL写的代码可以直接当中Cg代码使用。也就是说，cg基于知识联盟（Microsoft和NVIDIA），且拥有跨平台性，选择cg语言是大势所趋。有心的读者，可以注意市面上当前的GPU编程方面的书籍，大都是基于CG语言的。（附：Microsoft和NVIDIA联手推出Cg，应该是一种经济和技术上的双赢，通过这种方式联手打击GLSL）  
    此外，Cg，即C for graphics，用于图形的C语言，这其实说明了当时设计人员的一个初衷，就是“让基于图形硬件的编程变得和C语言编程一样方便，自由”。正如C++和 Java的语法是基于C的，cg语言本身也是基于C语言的。如果您使用过C、C++、Java其中任意一个，那么Cg的语法也是比较容易掌握的。Cg语言极力保留了C语言的大部分语义，力图让开发人员从硬件细节中解脱出来，Cg同时拥有高级语言的好处，如代码的易重用性，可读性提高等。使用cg还可以实现动画驱动、通用计算（排序、查找）等功能。  
在曾经的一段时间中有一种流言：NVIDIA将要抛弃Cg语言。并且在网上关于Cg、GLSL、HLSL的优劣讨论中，Cg的跨平台性也受到过广泛的质疑。我在2007年12月参加朱幼虹老师OSG培训班时，他曾专门对Cg、GLSL、HLSL进行了比较，说道：尽管目前还有一些关于Cg和GLSL之间的争议，不过主流的3D图形厂家都开始支持Cg语言。市场经济的选择可以说明一切，时间可以明辨真伪，到2009年末，Cg语言不但没有被抛弃，而且越来越受欢迎。

    我在OGRE官方论坛上，搜索过有关使用Cg和GLSL的讨论帖子，套用其中一个帖子的结尾语来结束本章：  
    In the last year I have never had to write a single HLSL/GLSL shader. Bottom line, I can’t think of any reason NOT to use CG.