HLSL 初级教程

作者：trcj

目录

前言

本教程针对 HLSL（High Level Shading Language）初学者，从应用的角度对 HLSL、顶点着色器、像素着色器和 Effect 效果框架进行了介绍，教程中去掉了对 HLSL 语法等一些细节内容的讨论，力求帮助读者尽可能快地理解 HLSL 编程的概念，掌握 HLSL 编程的方法。

教程中部分阐述直接引用了其他文档，这是因为这些文档表述之精要，已经达到了不能更改的地步，这里表示感谢。

1.HLSL入门

1.1 什么是着色器

DirectX 使用管道技术（pipeline）进行图形渲染，其构架如下：

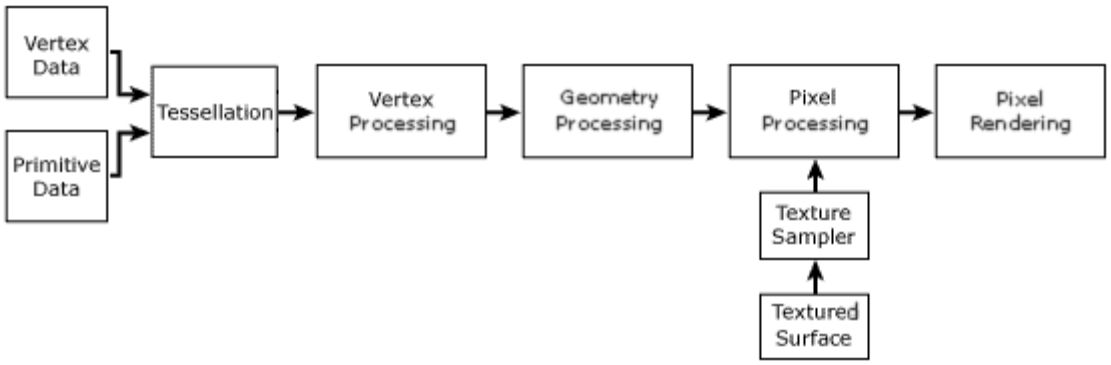


图 1.1 Direct3D Graphics Pipeline

之前我们使用管道的步骤如下：

1. 设定顶点、图元、纹理等数据信息；
2. 设定管道状态信息；

渲染状态

通过 SetRenderState 方法设定渲染状态；

另外，使用以下方法设置变换、材质和光照：

SetTransform

SetMaterial

SetLight

LightEnable

取样器状态

通过 SetSamplerState 方法设定取样器状态；

纹理层状态

通过 SetTextureStageState 设定纹理层状态；

3. 渲染；

这部分交由 D3D 管道按照之前的设定自行完成，这部分操作是 D3D 预先固定的，所以这种管道技术被称为固定功能管道(fixed function pipeline)；

固定功能管道给我们编程提供了一定的灵活性，但是仍有很多效果难以通过这种方式实现，比如：

1. 在渲染过程中，我们要求 y 坐标值大于 10 的顶点要被绘制到坐标值（0，0，0）的地方，在之前的固定功能管道中，顶点被绘制的位置是在第 1 步即被设定好的，不可能在渲染过程中进行改变，所以是不可行的；
2. 谋顶点在纹理贴图 1 上映射为点 A，在纹理贴图 2 上映射为点 B，我们要求该顶点颜色由 A、B 共同决定，即：

定点颜色 = A 点色彩值*0.7 + B 点色彩值*0.3

这在固定管道编程中也是不可行的。

以上两个问题都可以由可编程管道（pragrammable pipeline）来解决。

可编程管线允许用户自定义一段可以在 GPU 上执行的程序，代替固定管道技术中的 Vertex Processing 和 Pixel Processing 阶段（参照图 1.1），从而在使我们在编程中达到更大的灵活性。其中替换 Vertex Processing 的部分叫做 Vertex Shader（顶点着色器），替换 Pixel Proccessing 的部分叫做 Pixel Shader（像素着色器），这就是我们所说的着色器 Shader。

1.2 什么是 HLSL

Direct8.x 中，着色器是通过低级着色汇编语言来编写的，这样的程序更像是汇编式的指令集合，由于其效率低、可读性差、版本限制等缺点，迫切要求出现一门更高级的着色语言。到了 Direct3D9，HLSL（High Level Shading Language，高级渲染语言）应运而生了。

HLSL 的语法非常类似于 C 和 C++，学习起来是很方便的。

1.3 怎么写 HLSL 着色器

我们可以直接把 HLSL 着色器代码作为一长串字符串编写进我们的应用程序源文件中，但是，更加方便和模块化的方法是把着色器的代码从应用程序代码中分离出来。因此，我们将着色器代码单独保存为文本格式，然后在应用程序中使用特定函数将其加载进来。

下面是一个完整的 HLSL 着色器程序代码，我们把它保存在 BasicHLSL.txt 中。该着色器完成顶点的世界变换、观察变换和投影变幻，并将顶点颜色设定为指定的颜色。

```
//
// BasicHLSL.txt
//

//
// Global variable
//

matrix WVPMatrix;
vector color;

//
// Structures
//

struct VS_INPUT
{
    vector position : POSITION;
};

struct VS_OUTPUT
{
    vector position : POSITION;
    vector color : COLOR;
};

//
// Functions
```

```
//

VS_OUTPUT SetColor(VS_INPUT input)
{
    VS_OUTPUT output = (VS_OUTPUT)0;

    output.position = mul(input.position, WVPMatrix);
    output.color = color;

    return output;
}
```

下面就针对上述代码讲解一下 HLSL 着色器程序的编写：

1.3.1 全局变量

代码中声明了两个全局变量：

```
matrix WVPMatrix;
vector color;
```

变量 WVPMatrix 是一个矩阵类型，它包含了世界、观察、投影的合矩阵，用于对顶点进行坐标变换；

变量 color 是一个向量类型，它用于设定顶点颜色；

代码中并没有对全局变量进行初始化，这是因为我们对全局变量的初始化过程将在应用程序中进行，全局变量在应用程序中赋值而在着色器程序中使用，这是应用程序和着色器通信的关键所在。具体赋值过程将在后续部分讲述。

1.3.2 输入输出

输入输出结构

程序中定义了两个输入输出结构 VS_INPUT 和 VS_OUTPUT

```
struct VS_INPUT
{
    vector position : POSITION;
};

struct VS_OUTPUT
{
    vector position : POSITION;
    vector color : COLOR;
};
```

自定义的结构可以采用任意名称，结构不过是一种组织数据的方式，并不是强制的，你也可以不使用，而将本程序的输入改为：

```
vector position : POSITION;
```

标志符

用于输入输出的变量采用用一种特殊的声明方式：

```
Type VariableName : Semantic
```

这个特殊的冒号语法表示一个语义，冒号后面的标志符用来指定变量的用途，如

```
vector position : POSITION;
```

其中，POSITION 标志符表明该变量表示顶点位置，另外还有诸如 COLOR、NORMAL 等很多表示其他意义的标志符。

本节所说的输入输出其实是指着色器代码和编译器、GPU 之间的通信，和应用程序是无关的，所以这些变量不需要在应用程序中进行赋值，标志符告诉编译器各个输入输出变量的用途（顶点位置、法线、颜色等），这是着色器代码和编译器、GPU 之间通信的关键。

1.3.3 入口函数

程序中还定义了一个函数 SetColor：

```
OUTPUT SetColor(INPUT input)
{
    VS_OUTPUT output = (VS_OUTPUT)0;

    output.position = mul(input.position, WVPMatrix);
    output.color = color;

    return output;
}
```

1. 该函数以 input 和 output 类型作为输入输出；
2. 使全局变量 WVPMatrix 和 input.position 相乘，以完成顶点的世界、观察、投影变换，并把结果赋值到 output.position；

   ```
   output.position = mul(input.position, WVPMatrix);
   ```

3. 将全局变量 color 的值赋给 output.color；

   ```
   output.color = color;
   ```

4. 在同一个着色器代码文件中，可以有多个用户自定义函数，因此在应用程序中需要指定一个入口函数，相当于 windows 程序的 WinMain 函数，本程序只包含 SetColor 一个函数而且它将被做为入口函数使用。

1.3.4 总结

至此，一个 HLSL 着色器编写完毕，渲染过程中，当一个顶点被送到着色器时：

1. 全局变量 WVPMatrix、color 将在应用程序中被赋值；
2. 入口函数 SetColor 被调用编译器根据标志符将顶点信息填充到 VS_INPUT 中的各个字段；
3. SetColor 函数中，首先定义一个 VS_OUTPUT 信息，之后根据 WVPMatrix 和 color 变量完成顶点的坐标变换和颜色设定操作，最后函数返回 VS_OUTPUT 结构；
4. 编译器将会再次根据标志符把返回的 VS_OUTPUT 结构中的各字段映射为顶点相应的信

息。

5. 顶点被送往下一个流程接受进一步处理。

上述过程中，全局变量在应用程序中赋值而在着色器程序中使用，这是应用程序和着色器通信的关键所在；标志符告诉编译器各个输入输出变量的用途（顶点位置、法线、颜色等），这是着色器代码和编译器、GPU 之间通信的关键。个人认为这是着色器中最为精义的地方:)

1.4 怎么用 HLSL 着色器

应用程序中对 HLSL 着色器的使用分为以下步骤：

1. 加载（称为编译更为妥当）着色器代码；
2. 创建（顶点/像素）着色器；
3. 对着色器中的变量进行赋值，完成应用程序和着色器之间的通信。
4. 把着色器设定到渲染管道中；

本例使用的着色器是一个顶点着色器，因此我们将通过顶点着色器的使用来讲解着色器的使用过程，像素着色器的使用过程与此大同小异，二者之间仅有些微差别。

1.4.1 声明全局变量

```
IDirect3DVertexShader9* BasicShader = 0; //顶点着色器指针

ID3DXConstantTable* BasicConstTable = 0; //常量表指针
D3DXHANDLE WVPMatrixHandle          = 0;
D3DXHANDLE ColorHandle              = 0;

ID3DXMesh* Teapot                   = 0; //指向程序中 D3D 茶壶模型的指针
```

1.4.2 编译着色器

通过 D3DXCompileShaderFromFile 函数从应用程序外部的文本文件 BasicHLSL.txt 中编译一个着色器：

```
//编译后的着色器代码将被放在一个 buffer 中，可以通过 ID3DXBuffer 接口对其进行访问，之后的着色器将从这里创建
ID3DXBuffer* shaderBuffer = 0;
//用于接受错误信息
ID3DXBuffer* errorBuffer  = 0;

//编译着色器代码
D3DXCompileShaderFromFile("BasicHLSL.txt", //着色器代码文件名
                          0,
                          0,
                          "SetColor", //入口函数名称
                          "vs_1_1", //顶点着色器版本号
```

```
D3DXSHADER_DEBUG,// Debug 模式编译
&shaderBuffer, //指向编译后的着色器代码的指针
&errorBuffer,
&BasicConstTable); //常量表指针
```

1.4.3 创建着色器

应用程序通过 CreateVertexShader 创建一个顶点着色器，注意使用了上一步得到的 shaderBuffer：

```
g_pd3dDevice->CreateVertexShader((DWORD*)shaderBuffer->GetBufferPointer(), &BasicShader);
```

1.4.3 对着色器中的变量进行赋值

1.3.4 节说到着色器的全局变量在应用程序中赋值而在着色器程序中使用，这是应用程序和着色器通信的关键所在，这里就具体说明赋值过程。

着色器中的全局变量在编译后都被放在一个叫常量表的结构中，我们可以使用 ID3DXConstantTable 接口对其进行访问，参照 1.4.1 中编译着色器函数 D3DXCompileShaderFromFile 的最后一个参数，该参数即返回了指向常量表的指针。

对一个着色器中变量进行赋值的步骤如下：

1. 通过变量名称得到指向着色器变量的句柄；

还记得在 BasicHLSL.x 着色器文件中我们声明的两个全局变量吗：

```
matrix WVPMatrix;
vector color;
```

我们在应用程序中相应的声明两个句柄：

```
D3DXHANDLE WVPMatrixHandle = 0;
D3DXHANDLE ColorHandle = 0;
```

然后通过变量名得到分别得到对应的两个句柄：

```
WVPMatrixHandle = BasicConstTable->GetConstantByName(0, "WVPMatrix");
ColorHandle = BasicConstTable->GetConstantByName(0, "color");
```

2. 通过句柄对着色器变量进行赋值；

我们可以先设置各变量为默认值：

```
BasicConstTable->SetDefaults(g_pd3dDevice);
```

之后,可以使用 ID3DXConstantTable::SetXXX 函数对各个变量进行赋值：

```
HRESULT SetXXX(
  LPDIRECT3DDEVICE9 pDevice,
  D3DXHANDLE hConstant,
  XXX value
);
```

其中 XXX 代表变量类型，例如 Matrix 类型的变量就要使用 SetMatrix 函数赋值，而 Vector 类型的则要使用 SetVector 来赋值。

1.4.4 把着色器设定到渲染管道中

这里我们使用 SetVertexShader 方法把顶点着色器设定到渲染管道中：

```
g_pd3dDevice->SetVertexShader(BasicShader);
```

1.4.5 整个渲染过程如下

在渲染过程中，我们设定顶点的变换坐标和颜色值，渲染代码如下：

```
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                     D3DCOLOR_XRGB(153,153,153), 1.0f, 0 );
//开始渲染
g_pd3dDevice->BeginScene();

//得到世界矩阵、观察矩阵和投影矩阵
D3DXMATRIX matWorld, matView, matProj;
g_pd3dDevice->GetTransform(D3DTS_WORLD, &matWorld);
g_pd3dDevice->GetTransform(D3DTS_VIEW, &matView);
g_pd3dDevice->GetTransform(D3DTS_PROJECTION, &matProj);

D3DXMATRIX matWVP = matWorld * matView * matProj;
//通过句柄对着色器中的 WVPMatrix 变量进行赋值
BasicConstTable->SetMatrix(g_pd3dDevice, WVPMatrixHandle, &matWVP);

D3DXVECTOR4 color(1.0f, 1.0f, 0.0f, 1.0f);
//通过句柄对着色器中的 color 变量进行赋值，这里我们赋值为黄色
BasicConstTable->SetVector(g_pd3dDevice, ColorHandle, &color);

//把顶点着色器设定到渲染管道中
g_pd3dDevice->SetVertexShader(BasicShader);

//绘制模型子集
Teapot->DrawSubset(0);

//渲染完毕
g_pd3dDevice->EndScene();
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
```

编译运行程序，运行效果如图 1.2 所示，这里我们将顶点颜色设置为黄色，如果读者在渲染过程中不断变换对着色器变量 color 的赋值，你将会得到一个色彩不断变幻的 D3D 茶壶。

```
D3DXVECTOR4 color(1.0f, 1.0f, 0.0f, 1.0f); //读者可以尝试改变颜色值
BasicConstTable->SetVector(g_pd3dDevice, ColorHandle, &color);
```

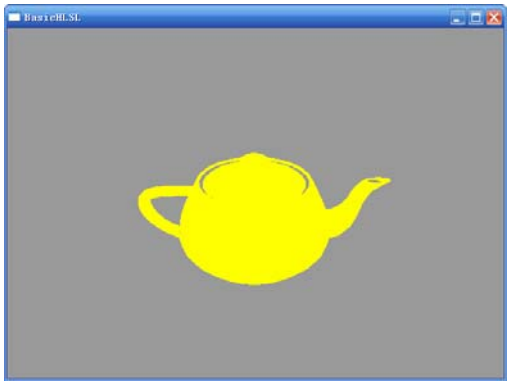


图 1.2 着色器效果

2.顶点着色器

顶点着色器（vertex shader）是一个在显卡的 GPU 上执行的程序，它替换了固定功能管道（fixed function pipeline）中的变换（transformation）和光照（lighting）阶段（这不是百分之百的正确，因为顶点着色器可以被 Direct3D 运行时（Direct3D runtime）以软件模拟，如果硬件不支持顶点着色器的话）。图 2.1 说明了管线中顶点着色器替换的部件。

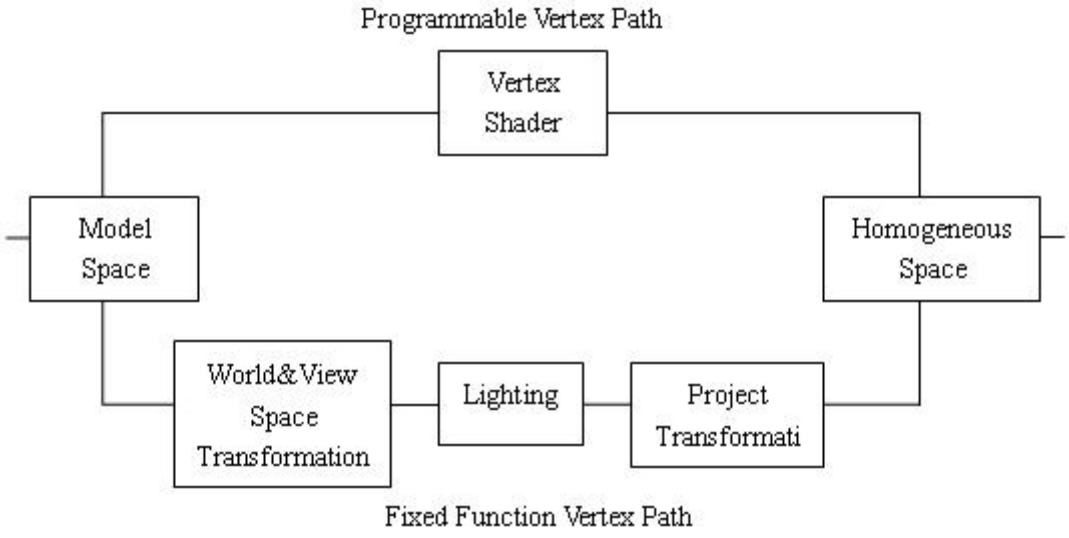


图 2.1

由于顶点着色器是我们（在 HLSL 中）写的一个自定义程序，因此我们在图形效果方面获得了极大的自由性。我们不再受限于 Direct3D 的固定光照算法。此外，应用程序操纵顶点位置的能力也有了多样性，例如：布料仿真，粒子系统的点大小操纵，还有顶点混合/变形。此外，我们的顶点数据结构更自由了，并且可以在可编程管线中包含比在固定功能管线中多的多的数据。

正如作者所在群的公告所说，“拍照不在于你对相机使用的熟练程度,而是在于你对艺术的把握。”之前的介绍使读者对着色器的编写和使用都有了一定的了解，下面我们将把重心从介绍如何使用着色器转到如何实现更高级的渲染效果上来。

2.1 可编程数据流模型

DirectX 8.0 引入了数据流的概念，可以这样理解数据流（图 2.2）：

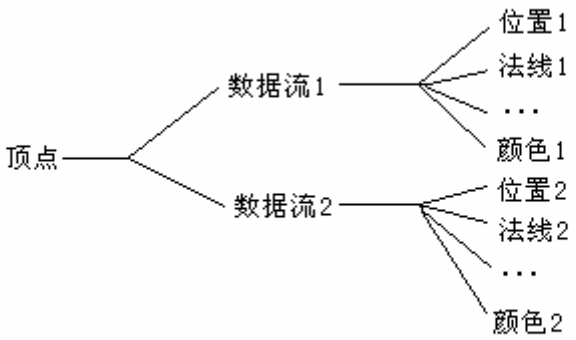


图 2.2

一个顶点由 n 个数据流组成。

一个数据流由 m 个元素组成。

一个元素是[位置、颜色、法向、纹理坐标]。

程序中使用 IDirect3DDevice9::SetStreamSource 方法把一个顶点缓存绑定到一个设备数据流。

2.2 顶点声明

该小节对顶点声明的描述绝大多数都取自翁云兵的《着色器和效果》，该文对顶点声明的描述是我所见到最详尽最透彻的，这里向作者表示敬意:)

到现在为止，我们已经使用自由顶点格式（flexible vertex format，FVF）来描述顶点结构中的各分量。但是，在可编程管线中，我们的顶点数据可以包含比用 FVF 所能表达的多的多的数据。因此，我们通常使用更具表达性的并且更强有力的顶点声明（vertex declaration）。

注意：我们仍然可以在可编程管线中使用 FVF——如果我们的顶点格式可以这样描述。不管怎样，这只是为了方便，因为 FVF 会在内部被转换为一个顶点声明。

2.2.1 描述顶点声明

我们将一个顶点声明描述为一个 D3DVERTEXELEMENT9 结构的数组。D3DVERTEXELEMENT9 数组中的每个元素描述了一个顶点的分量。所以，如果你的顶点结构有三个分量（例如：位置、法线、颜色），那么其相应的顶点声明将会被一个含 3 个元素的 D3DVERTEXELEMENT9 结构数组描述。

D3DVERTEXELEMENT9 结构定义如下：

```
typedef struct _D3DVERTEXELEMENT9 {
    BYTE Stream;
    BYTE Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

Stream——指定关联到顶点分量的流；

Offset——偏移，按字节，相对于顶点结构成员的顶点分量的开始。例如，如果顶点结构是：

```
struct Vertex
{
    D3DXVECTOR3 pos;
    D3DXVECTOR3 normal;
};
```

……pos 分量的偏移是 0，因为它是第一个分量；normal 分量的偏移是 12，因为 sizeof(pos) == 12。换句话说，normal 分量以 Vertex 的第 12 个字节为开始。

Type——指定数据类型。它可以是 D3DDECLTYPE 枚举类型的任意成员；完整列表请参见文档。常用类型如下：

D3DDECLTYPE_FLOAT1——浮点数值

D3DDECLTYPE_FLOAT2——2D 浮点向量

D3DDECLTYPE_FLOAT3——3D 浮点向量

D3DDECLTYPE_FLOAT4——4D 浮点向量

D3DDECLTYPE_D3DCOLOR—D3DCOLOR 类型，它扩展为 RGBA 浮点颜色向量(r, g, b, a)，其每一分量都是归一化到区间[0, 1]了的。

Method——指定网格化方法。我们认为这个参数是高级的，因此我们使用默认值，标识为 D3DDECLMETHOD_DEFAULT。

Usage——指定已计划的对顶点分量的使用。例如，它是否准备用于一个位置向量、法线向量、纹理坐标等,有效的用途标识符（usage identifier）是 D3DDECLUSAGE 枚举类型的：

```
typedef enum _D3DDECLUSAGE {
    D3DDECLUSAGE_POSITION     = 0,  // Position.
    D3DDECLUSAGE_BLENDWEIGHTS = 1,  // Blending weights.
    D3DDECLUSAGE_BLENDINDICES = 2,  // Blending indices.
    D3DDECLUSAGE_NORMAL       = 3,  // Normal vector.
    D3DDECLUSAGE_PSIZE        = 4,  // Vertex point size.
    D3DDECLUSAGE_TEXCOORD     = 5,  // Texture coordinates.
    D3DDECLUSAGE_TANGENT      = 6,  // Tangent vector.
    D3DDECLUSAGE_BINORMAL     = 7,  // Binormal vector.
    D3DDECLUSAGE_TESSFACTOR   = 8,  // Tessellation factor.
    D3DDECLUSAGE_POSITIONT    = 9,  // Transformed position.
    D3DDECLUSAGE_COLOR        = 10, // Color.
    D3DDECLUSAGE_FOG          = 11, // Fog blend value.
    D3DDECLUSAGE_DEPTH        = 12, // Depth value.
    D3DDECLUSAGE_SAMPLE       = 13  // Sampler data.
} D3DDECLUSAGE;
```

其中，D3DDECLUSAGE_PSIZE 类型用于指定一个顶点的点的大小。它用于点精灵，因此我们可以基于每个顶点控制其大小。一个 D3DDECLUSAGE_POSITION 成员的顶点声明意味着这个顶点已经被变换，它通知图形卡不要把这个顶点送到顶点处理阶段（变形和光照）。

UsageIndex——用于标识多个相同用途的顶点分量。这个用途索引是位于区间[0, 15]间的一个整数。例如，假设我们有三个用途为 D3DDECLUSAGE_NORMAL 的顶点分量。我们可以为第一个指

定用途索引为 0，为第二个指定用途索引为 1，并且为第三个指定用途索引为 2。按这种方式，我们可以通过其用途索引标识每个特定的法线。

例：假设我们想要描述的顶点格式由两个数据流组成，第一个数据流包含位置、法线、纹理坐标 3 个分量，第二个数据流包含位置和纹理坐标 2 个分量，顶点声明可以指定如下：

```
D3DVERTEXELEMENT9 decl[] =
{
    //第一个数据流，包含分量位置、法线、纹理坐标
    { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_
POSITION, 0 },
    { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
NORMAL, 0 },
    { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
TEXCOORD, 0 },

    //第一个数据流，包含分量位置、纹理坐标
    { 1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
POSITION, 1 },
    { 1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
NORMAL, 1 },
    D3DDECL_END()
};
```

D3DDECL_END 宏用于初始化 D3DVERTEXELEMENT9 数组的最后一个顶点元素。

2.2.2 创建顶点声明

CreateVertexDeclaration 函数用于创建顶点声明，decl 为指向上一小节定义的 D3DVERTEXELEMENT9 数组的指针，函数返回 IDirect3DVertexDeclaration9 指针 g_Decl；

```
IDirect3DVertexDeclaration9     *g_Decl = NULL;
g_pd3dDevice->CreateVertexDeclaration(decl ,&g_Decl);
```

2.2.3 设置顶点声明

```
g_pd3dDevice->SetVertexDeclaration(g_Decl);
```

至此，可编程数据流模型、顶点声明介绍完毕，在下面的例子中读者将会有更连贯的理解。

2.3 用顶点着色器实现渐变动画

2.3.1 渐变动画（Morphing）

Morphing 渐变是 20 世纪 90 年代出现的一种革命性的计算机图形技术，该技术使得动画序列平滑且易于处理，即使在低档配置的计算机系统上也能正常运行。

渐变是指随时间的变化把一个形状改变为另一个形状。对我们而言，这些形状就是 Mesh 网格模型。渐变网格模型的处理就是以时间轴为基准，逐渐地改变网格模型顶点的坐标，从一个网格模型的形状渐变到另外一个。请看图 2.3：

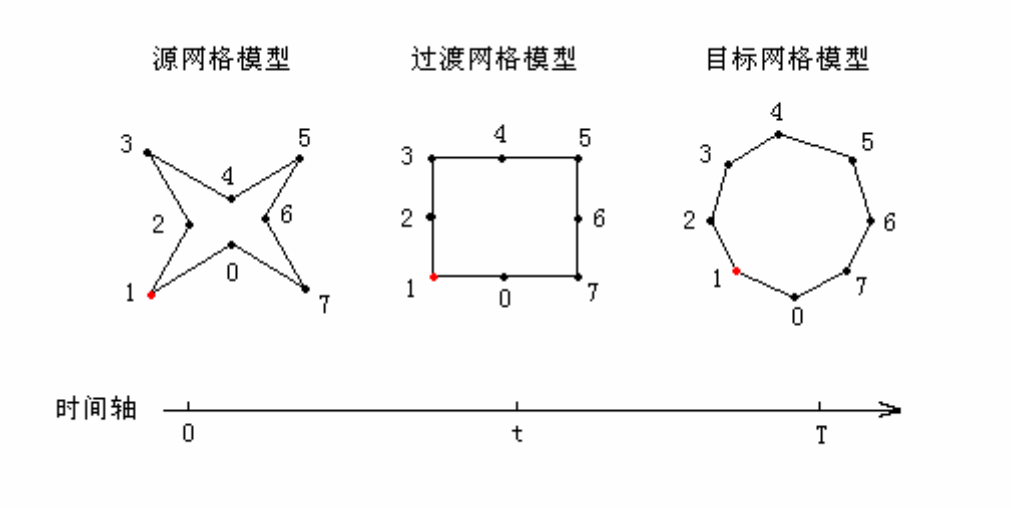


图 2.3

我们在程序中使用两个网格模型——源网格模型和目标网格模型，设源网格模型中顶点 1 的坐标为 A（A_x，A_y，A_z），目标网格模型中对应顶点 1 的坐标为 B（B_x，B_y，B_z），要计算渐变过程中时间点 t 所对应的顶点 1 的坐标 C（C_x，C_y，C_z），我们使用如下方法：

T 为源网格模型到目标网格模型渐变所花费的全部时间，得到时间点 t 占整个过程 T 的比例为：

S = t / T

那么顶点 1 在 t 时刻对应的坐标 C 为：

C = A * (1-S）+ B * S

这样，在渲染过程中我们根据时间不断调整 S 的值，就得到了从源网格模型（形状一）到目标网格模型（形状二）的平滑过渡。

接下来将在程序里使用顶点着色器实现我们的渐变动画。

2.3.2 渐变动画中的顶点声明

程序中，我们设定一个顶点对应两个数据流，这两个数据流分别包含了源网格模型的数据和目标网格模型的数据。渲染过程中，我们在着色器里根据两个数据流中的顶点数据以及时间值确定最终的顶点信息。

个数据流包含分量如下：

源网格模型数据流：顶点位置、顶点法线、纹理坐标；

目标网格模型数据流：顶点位置、顶点法线；

注意目标网格模型数据流没有包含纹理坐标，因为纹理对于两个网格模型都是一样的，所以仅使用源网格模型的纹理就可以了。

顶点声明指定如下：

```
D3DVERTEXELEMENT9 decl[] =
{
    //源网格模型数据流，包含分量位置、法线、纹理坐标
```

```
    { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_
POSITION, 0 },
    { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
NORMAL, 0 },
    { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
TEXCOORD, 0 },

    //目标网格模型数据流，包含分量位置、纹理坐标
    { 1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
POSITION, 1 },
    { 1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_
NORMAL, 1 },
    D3DDECL_END()
};
```

2.3.3 渐变动画中的顶点着色器

下面给出顶点着色器源码，代码存储于 vs.txt 中，该顶点着色器根据源网格模型数据流和目标网格模型数据流中的信息以及时间标尺值计算出顶点最终位置信息，并对顶点做了坐标变换和光照处理。代码中给出了详细的注释，帮助读者理解。

```
    //全局变量
    //世界矩阵、观察矩阵、投影矩阵的合矩阵，用于顶点的坐标变换
    matrix WVPMatrix;

    //光照方向
    vector LightDirection;
    //存储 2.3.1 小节提到的公式 S = t / T 中的时间标尺 S 值
    //注意到 Scalar 是一个 vector 类型，我们在 Scalar.x 中存储了 S 值，Scalar.y 中存储的则是（1-S）值
    vector Scalar;

    //输入
    struct VS_INPUT
    {
        //对应源网格模型数据流中的顶点分量：位置、法线、纹理坐标
         vector position : POSITION;
         vector normal      : NORMAL;
        float2 uvCoords : TEXCOORD;
        //对应目标网格模型数据流中的顶点分量：位置、法线
         vector position1 : POSITION1;
         vector normal1      : NORMAL1;
```

```
};

//输出
struct VS_OUTPUT
{
    vector position : POSITION;
    vector diffuse    : COLOR;
    float2 uvCoords : TEXCOORD;
};

//入口函数
VS_OUTPUT Main(VS_INPUT input)
{
    VS_OUTPUT output = (VS_OUTPUT)0;

    //顶点最终位置 output.position 取决于源网格模型数据流中位置信息 input.position 和目标网格模型数据流中位置信息 input.position1 以及时间标尺 Scalar 的值
    //对应 2.3.1 小节中的公式 C = A * (1-S）+ B * S
    output.position = input.position*Scalar.x + input.position1*Scalar.y;
    //顶点坐标变换操作
    output.position = mul(output.position, WVPMatrix);

    //计算顶点最终法线值
    vector normal = input.normal*Scalar.x + input.normal1*Scalar.y;
    //逆光方向与法线的点积，获得漫射色彩
    output.diffuse = dot((-LightDirection), normal);

    //存储纹理坐标
    output.uvCoords = input.uvCoords;

    return output;
}
```

以上是本例用到的顶点着色器，在接下来的应用程序中，我们将给三个着色器全局变量赋值：

WVPMatrix;

世界矩阵、观察矩阵、投影矩阵的合矩阵，用于顶点的坐标变换；

LightDirection

光照方向；

Scalar

存储 2.3.1 小节提到的公式 S = t / T 中的时间标尺 S 值；

注意到 Scalar 是一个 vector 类型，我们在 Scalar.x 中存储了 S 值，Scalar.y 中存储的则是（1-S）

值；

2.3.4 应用程序

我们在应用程序中执行以下操作：

加载两个两个 Mesh 模型：源网格模型，目标网格模型；

创建、设置顶点声明；

创建、设置顶点着色器；

为着色器全局赋值；

把两个 Mesh 模型数据分别绑定到两个数据流中；

渲染 Mesh 模型；

下面是应用程序代码：

```
…
/*********************声明变量*****************/
//两个指向 LPD3DXMESH 的指针，分别用于存储源网格模型和目标网格模型；
LPD3DXMESH                          g_SourceMesh;
LPD3DXMESH                          g_TargetMesh;

//顶点声明指针
IDirect3DVertexDeclaration9     *g_Decl = NULL;

//顶点着色器
IDirect3DVertexShader9              *g_VS     = NULL;
//常量表
ID3DXConstantTable* ConstTable = NULL;

//常量句柄
D3DXHANDLE WVPMatrixHandle              = 0;
D3DXHANDLE ScalarHandle                 = 0;
D3DXHANDLE LightDirHandle               = 0;
…
/***************程序初始化*****************/
//加载源、目标网格模型
Load_Meshes();

//顶点声明
D3DVERTEXELEMENT9 MorphMeshDecl[] =
{
    //1st stream is for source mesh - position, normal, texcoord
    {       0,          0,      D3DDECLTYPE_FLOAT3,     D3DDECLMETHOD_DEFAULT,
```

```
D3DDECLUSAGE_POSITION, 0 },
        {       0,      12,     D3DDECLTYPE_FLOAT3,     D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL,    0 },
        {       0,      24,     D3DDECLTYPE_FLOAT2,     D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_TEXCOORD, 0 },

        //2nd stream is for target mesh - position, normal
        {       1,      0,      D3DDECLTYPE_FLOAT3,     D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION, 1 },
        {       1,      12,     D3DDECLTYPE_FLOAT3,     D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL,    1 },
        D3DDECL_END()
    };

    //创建顶点着色器
    ID3DXBuffer* shader         = NULL;
    ID3DXBuffer* errorBuffer    = NULL;
    D3DXCompileShaderFromFile("vs.txt",
                              0,
                              0,
                               "Main", // entry point function name
                              "vs_1_1",
                              D3DXSHADER_DEBUG,
                               &shader,
                              &errorBuffer,
                              &ConstTable);

    if(errorBuffer)
    {
        ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
        ReleaseCOM(errorBuffer);
    }

    //创建顶点着色器
    g_pd3dDevice->CreateVertexShader((DWORD*)shader->GetBufferPointer(), &g_VS);

    //创建顶点声明
    g_pd3dDevice->CreateVertexDeclaration(MorphMeshDecl ,&g_Decl);

    //得到各常量句柄
```

```
WVPMatrixHandle = ConstTable->GetConstantByName(0, "WVPMatrix");
ScalarHandle = ConstTable->GetConstantByName(0, "Scalar");
LightDirHandle = ConstTable->GetConstantByName(0, "LightDirection");

//为着色器全局变量 LightDirection 赋值
ConstTable->SetVector(g_pd3dDevice, LightDirHandle, &D3DXVECTOR4(0.0f, -1.0f, 0.0f, 0.0f));
//设置各着色器变量为默认值
ConstTable->SetDefaults(g_pd3dDevice);
…
/*******************渲染*******************/
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                     D3DCOLOR_XRGB(153,153,153), 1.0f, 0 );
g_pd3dDevice->BeginScene();

//为着色器全局变量 WVPMatrix 赋值
D3DXMATRIX matWorld, matView, matProj;
g_pd3dDevice->GetTransform(D3DTS_WORLD, &matWorld);
g_pd3dDevice->GetTransform(D3DTS_VIEW, &matView);
g_pd3dDevice->GetTransform(D3DTS_PROJECTION, &matProj);
D3DXMATRIX matWVP;
matWVP = matWorld * matView * matProj;

ConstTable->SetMatrix(g_pd3dDevice, WVPMatrixHandle, &matWVP);

//为着色器全局变量 Scalar 赋值，注意程序中获取时间标尺值 Scalar 的方法
float DolphinTimeFactor = (float)(timeGetTime() % 501) / 250.0f;
float Scalar =
(DolphinTimeFactor<=1.0f)?DolphinTimeFactor:(2.0f-DolphinTimeFactor);
ConstTable->SetVector(g_pd3dDevice,ScalarHandle,&D3DXVECTOR4(1.0f-Scalar, Scalar, 0.0f, 0.0f));

//设置顶点着色器和顶点声明
g_pd3dDevice->SetVertexShader(g_VS);
g_pd3dDevice->SetVertexDeclaration(g_Decl);

//绑定目标网格模型的定点缓存到第二个数据流中
IDirect3DVertexBuffer9 *pVB = NULL;
g_TargetMesh->GetVertexBuffer(&pVB);
g_pd3dDevice->SetStreamSource(1, pVB, 0,
                              D3DXGetFVFVertexSize(g_TargetMesh->GetFVF()));
ReleaseCOM(pVB);
```

```
//绑定源网格模型的顶点缓存到第一个数据流中
g_SourceMesh->GetVertexBuffer(&pVB);
g_pd3dDevice->SetStreamSource(0, pVB, 0,
                              D3DXGetFVFVertexSize(g_TargetMesh->GetFVF()));
ReleaseCOM(pVB);

//绘制 Mesh 网格模型
DrawMesh(g_SourceMesh, g_pMeshTextures0, g_VS, g_Decl);

g_pd3dDevice->EndScene();
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
…
```

2.3.5 对应用程序的一点说明

程序中我们使用 SetStreamSource 方法把源网格模型和目标网格模型中的顶点缓存分别绑定到两个设备数据流，但是 Direct3D 对数据流中的数据的真正引用只有在调用诸如 DrawPrimitive、DrawIndexedPrimitive 之类的绘制方法时才发生，因此在绘制 Mesh 网格模型时我们不能再使用传统的 DrawSubmit 方法，而是使用了 DrawIndexedPrimitive，下面就如何调用 DrawIndexedPrimitive 绘制 Mesh 模型进行说明，该部分内容和 HLSL 着色器关系不大，在这里列出仅仅是为了大家理解程序的完整性，读者完全可以跳过本节不看。

使用 DrawIndexedPrimitive 绘制 Mesh 模型的步骤如下：

1. 加载网格模型后使用 OptimizeInPlace 方法对 Mesh 进行优化；

2. 一旦优化了网格模型，你就可以查询 ID3DXMesh 对象，得到一个 D3DXATTRIBUTERANGE 数据类型的数组，我们称之为属性列表，该数据类型被定义如下：

```
typedef struct_D3DXATTRIBUTERANGE{
    DWORD AttribId; //子集编号
    DWORD FaceStart; //这两个变量用于圈定本子集中的多边形
    DWORD FaceCount;
    DWORD VertexStart; //这两个变量用于圈定本子集中的顶点
    DWORD VertexCount;
} D3DXATTRIBUTERANGE;
```

我们属性列表中的每一项都代表一个被优化后 Mesh 的一个子集，D3DXATTRIBUTERANGE 结构的各字段描述了该子集的信息。

1. 得到属性数据后，我们就调用 DrawIndexedPrimitive 方法可以精美地渲染子集了。

下面是绘制 Mesh 模型的程序代码：

在 Load_Meshes()函数的最后，我们使用 OptimizeInPlace 方法对源网格模型和目标网格模型进行优化，其他加载材质和纹理的操作和之前一样，相信大家能够理解：

```
…
//优化源网格模型
g_SourceMesh->OptimizeInplace(D3DXMESHOPT_ATTRSORT, NULL, NULL, NULL, NULL);
…
//优化目标网格模型
g_TargetMesh->OptimizeInplace(D3DXMESHOPT_ATTRSORT, NULL, NULL, NULL, NULL);
…
```

在 Draw_Mesh()函数中，渲染模型，注意程序是如何配合属性表调用 DrawIndexedPrimitive 方法进行绘制的：

```
…

//分别得到指向 Mesh 模型顶点缓存区和索引缓存区的指针
IDirect3DVertexBuffer9 *pVB = NULL;
IDirect3DIndexBuffer9 *pIB    = NULL;
pMesh->GetVertexBuffer(&pVB);
pMesh->GetIndexBuffer(&pIB);

//得到 Mesh 模型的属性列表
DWORD NumAttributes;
D3DXATTRIBUTERANGE *pAttributes = NULL;
pMesh->GetAttributeTable(NULL, &NumAttributes);
pAttributes = new D3DXATTRIBUTERANGE[NumAttributes];
pMesh->GetAttributeTable(pAttributes, &NumAttributes);

//设置顶点着色器和顶点声明
g_pd3dDevice->SetVertexShader(pShader);
g_pd3dDevice->SetVertexDeclaration(pDecl);

//设置数据流
g_pd3dDevice->SetStreamSource(0, pVB, 0, D3DXGetFVFVertexSize(pMesh->GetFVF()));
g_pd3dDevice->SetIndices(pIB);

//遍历属性列表并配合其中的信息调用 DrawIndexPrimitive 绘制各个子集
for(DWORD i=0;i<NumAttributes;i++)
{
    if(pAttributes[i].FaceCount)
    {
        //Get material number
        DWORD MatNum = pAttributes[i].AttribId;
```

```
            //Set texture
            g_pd3dDevice->SetTexture(0, pTextures[MatNum]);

            //Draw the mesh subset
            g_pd3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0,
                                               pAttributes[i].VertexStart,
                                               pAttributes[i].VertexCount,
                                               pAttributes[i].FaceStart * 3,
                                               pAttributes[i].FaceCount);
        }
    }

    //Free resources
    ReleaseCOM(pVB);
    ReleaseCOM(pIB);
    delete [] pAttributes;

    …
```

编译运行程序，效果如图 2.4 所示，你将看到屏幕上白色的海豚上下翻腾，同时感受到顶点着色器为渲染效果所带来的巨大改善。

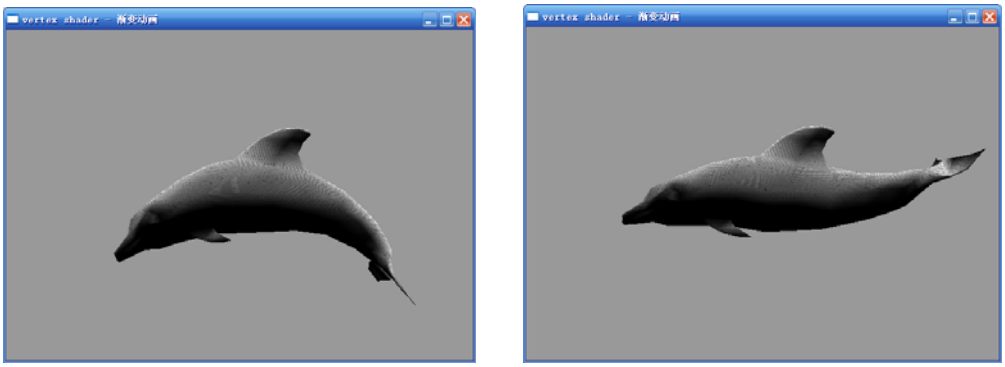


图 2.4

3.像素着色器

像素着色器是在对每个像素进行光栅化处理期间在图形卡的 GPU 上执行的程序。(不像顶点着色器，Direct3D 不会以软件模拟像素着色器的功能。）它实际上替换了固定功能管线的多纹理化阶段（the multitexturing stage），并赋予我们直接操纵单独的像素和访问每个像素的纹理坐标的能力。这种对像素和纹理坐标的直接访问使我们可以达成各种特效，例如：多纹理化（multitexturing）、每像素光照（per pixel lighting）、景深（depth of field）、云状物模拟（cloud simulation）、焰火模拟（fire simulation）、混杂阴影化技巧（sophisticated shadowing technique）。

像素着色器的编写、使用和顶点着色器大同小异，有了之前的基础，不用太过于详细的介绍相信读者也能理解，下面使用像素着色器实现多纹理化。

3.1 多纹理化

简单的说，多纹理化就是使用多个纹理贴图混合后进行渲染，如图 3.1，渲染过程中，从纹理 1 和纹理 2 中分别采样，得到的颜色值依据一定规则进行组合得到纹理 3，这就是多纹理化。

图 3.1

3.2 多纹理效果的像素着色器

下面是像素着色器的代码，该代码存储于 ps.txt 中，该像素着色器根据输入的两套纹理坐标对对应的纹理贴图进行采样，根据一定比例 Scalar 混合后输出像素颜色。

```
//全局变量

//存储颜色混合的比例值 s，其中
//Scalar.x = s
//Scalar.y = 1-s
vector Scalar;

//纹理
texture Tex0;
texture Tex1;

//纹理采样器
sampler Samp0 =
sampler_state
{
    Texture = <Tex0>;
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
};
```

```
sampler Samp1 =
sampler_state
{
      Texture = <Tex1>;
      MipFilter = LINEAR;
      MinFilter = LINEAR;
      MagFilter = LINEAR;
};

//输入两套纹理坐标
struct PS_INPUT
{
    float2 uvCoords0 : TEXCOORD0;
    float2 uvCoords1 : TEXCOORD1;
};

//输出像素颜色
struct PS_OUTPUT
{
    float4 Color : COLOR0;
};

//入口函数
PS_OUTPUT PS_Main(PS_INPUT input)
{
    PS_OUTPUT output = (PS_OUTPUT)0;
    //分别对两个纹理进行采样按照比例混合后输出颜色值
    output.Color = tex2D(Samp0, input.uvCoords0)*Scalar.x + tex2D(Samp1, input.uvCoords1)*Scalar.y;
    return output;
}
```

整个程序很容易理解，程序中涉及到着色器的纹理和采样，是我们第一次接触的内容，下面给于说明。

3.2.1HLSL采样器和纹理

和 vector、matrix 一样，采样器 sample 和纹理 texture 也是 HLSL 语言的一种类型，HLSL 着色器使用采样器对指定纹理进行采样，得到采样后的颜色值以供处理。

它们的用法如下：

```
//声明一个纹理变量
texture g_texture;
```

```
//定义采样器
sampler g_samp =
sampler_state
{
    //关联到纹理
    Texture = <g_texture>;
    //设置采样器状态
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
};

//调用 HLSL 内置函数 tex2D 取得颜色值，参数一为采样器，参数二为纹理坐标
vector Color = tex2D(g_samp, uvCoords);
```

更多 HLSL 采样器和纹理的内容请参见 DirectX 文档。

以上是本例用到的像素着色器，在接下来的应用程序中，我们将给三个着色器全局变量赋值：

Scalar
存储颜色混合的比例值 s，其中 Scalar.x = s, Scalar.y = 1-s;
Samp0
第一层纹理采样器；
Samp1
第二层纹理采样器；

像素着色器的输入结构中我们设定了一个顶点对应两套纹理坐标，读者可以留意一下应用程序中对应的顶点格式的定义。

3.3 应用程序

程序中我们首先创建一个四边形，然后使用像素着色器进行纹理混合后对其进行渲染。下面是应用程序代码：

```
…
/**********************顶点格式定义*****************/
struct CUSTOMVERTEX
{
    //定点位置坐标
    float x,y,z;
    //两套纹理坐标；
    float tu0, tv0;
    float tu1, tv1;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_TEX2)
```

```
…
/*********************声明变量*****************/
//顶点着色器
LPDIRECT3DPIXELSHADER9 pixelShader      = 0;
//常量表
ID3DXConstantTable* pixelConstTable    = 0;

//常量句柄
D3DXHANDLE ScalarHandle                        = 0;
D3DXHANDLE Samp0Handle                         = 0;
D3DXHANDLE Samp1Handle                         = 0;

//常量描述结构
D3DXCONSTANT_DESC Samp0Desc;
D3DXCONSTANT_DESC Samp1Desc;

//四边形顶点缓存
LPDIRECT3DVERTEXBUFFER9 quadVB    = NULL;
//两个纹理
LPDIRECT3DTEXTURE9 quadTexture0 = NULL;
LPDIRECT3DTEXTURE9 quadTexture1 = NULL;
…

/*********************初始化应用程序*****************/
//创建四边形顶点模型
CUSTOMVERTEX quad[] =
//    x          y          z       tu0      tv0      tu1      tv1
{{-3.0f, -3.0f, 10.0f, 0.0f, 1.0f, 0.0f, 1.0f},
{ -3.0f,    3.0f, 10.0f, 0.0f, 0.0f, 0.0f, 0.0f},
{    3.0f, -3.0f, 10.0f, 1.0f, 1.0f, 1.0f, 1.0f},
{    3.0f,    3.0f, 10.0f, 1.0f, 0.0f, 1.0f, 0.0f}};

//创建顶点缓存
void *ptr = NULL;
g_pd3dDevice->CreateVertexBuffer(sizeof(quad),
                                                D3DUSAGE_WRITEONLY,
                                                0,
                                                D3DPOOL_MANAGED,
                                                &quadVB,
                                                NULL);
```

```
quadVB->Lock(0, 0, (void**)&ptr, 0);
memcpy((void*)ptr, (void*)quad, sizeof(quad));
quadVB->Unlock();

//创建纹理
D3DXCreateTextureFromFile(g_pd3dDevice, "porpcart.jpg", &quadTexture0);
D3DXCreateTextureFromFile(g_pd3dDevice, "luoqi.jpg", &quadTexture1);

//检测系统是否支持像素着色器
D3DCAPS9 caps;
g_pd3dDevice->GetDeviceCaps(&caps);
if(caps.PixelShaderVersion < D3DPS_VERSION(1, 1))
{
    MessageBox(0, "NotSupport Pixel Shader - FAILED", 0, 0);
    exit(0);
}

//创建像素着色器
ID3DXBuffer* codeBuffer              = 0;
ID3DXBuffer* errorBuffer             = 0;

HRESULT hr = D3DXCompileShaderFromFile("ps.txt",
                                       0,
                                       0,
                                       "PS_Main", // entry point function name
                                       "ps_1_1",
                                       D3DXSHADER_DEBUG,
                                       &codeBuffer,
                                       &errorBuffer,
                                       &pixelConstTable);

// output any error messages
if(errorBuffer)
{
    MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
    ReleaseCOM(errorBuffer);
}

if(FAILED(hr))
{
```

```
    MessageBox(0, "D3DXCompileShaderFromFile() - FAILED", 0, 0);
    return false;
}


hr = g_pd3dDevice->CreatePixelShader((DWORD*)codeBuffer->GetBufferPointer(), &pixelShader);

if(FAILED(hr))
{
    MessageBox(0, "CreatePixelShader - FAILED", 0, 0);
    return false;
}

ReleaseCOM(codeBuffer);
ReleaseCOM(errorBuffer);

//得到各常量句柄
ScalarHandle = pixelConstTable->GetConstantByName(0, "Scalar");
Samp0Handle = pixelConstTable->GetConstantByName(0, "Samp0");
Samp1Handle = pixelConstTable->GetConstantByName(0, "Samp1");

//得到对着色器变量 Samp0、Samp0 的描述
UINT count;
pixelConstTable->GetConstantDesc(Samp0Handle, & Samp0Desc, &count);
pixelConstTable->GetConstantDesc(Samp1Handle, & Samp1Desc, &count);

//设定各着色器变量为初始值
pixelConstTable->SetDefaults(g_pd3dDevice);
…
/**********渲染**********/
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
D3DCOLOR_XRGB(153,153,153), 1.0f, 0 );
g_pd3dDevice->BeginScene();

//为着色器全局变量 Scalar 赋值
D3DXVECTOR4 scalar(0.5f, 0.5f, 0.0f, 1.0f);
pixelConstTable->SetVector(g_pd3dDevice, ScalarHandle, &scalar);

//设置像素着色器
g_pd3dDevice->SetPixelShader(pixelShader);
```

```
//设置定点格式、绑定数据流
g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
g_pd3dDevice->SetStreamSource(0, quadVB, 0, sizeof(CUSTOMVERTEX));

//设置第一、二层纹理
g_pd3dDevice->SetTexture(Samp0Desc.RegisterIndex, quadTexture0);
g_pd3dDevice->SetTexture(Samp1Desc.RegisterIndex, quadTexture1);

//绘制图形
g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

g_pd3dDevice->EndScene();
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
…
```

程序中像素着色器的使用和之前顶点着色器的使用无二，只是设置着色器中纹理采样器变量 Samp0、Samp1 和设定着色器其他变量稍有不同：

1. 首先通过变量名称得到变量句柄：

```
Tex0Handle = pixelConstTable->GetConstantByName(0, " Samp0");
Tex1Handle = pixelConstTable->GetConstantByName(0, " Samp1");
```

2. 然后通过句柄得到对变量的描述：

```
UINT count;
pixelConstTable->GetConstantDesc(Samp0Handle, & Samp0Desc, &count);
pixelConstTable->GetConstantDesc(Samp1Handle, & Samp1Desc, &count);
```

3.最后通过 SetTexture 配合所得到的描述信息设置纹理：

```
g_pd3dDevice->SetTexture(Samp0Desc.RegisterIndex, quadTexture0);
g_pd3dDevice->SetTexture(Samp1Desc.RegisterIndex, quadTexture1);
```

编译运行程序，运行效果如图 3.2，这里我们将颜色混合比例设置为 0.5，如果读者在渲染过程中不断变换对着色器变量 Scalar 的赋值，你将会得到一个混合度不断变换的多纹理效果。

```
D3DXVECTOR4 scalar(0.5f, 0.5f, 0.0f, 1.0f); //读者可以尝试改变混合采用的比例值
pixelConstTable->SetVector(g_pd3dDevice, ScalarHandle, &scalar);
```


纹理一

纹理二

混合后纹理三

图 3.2

4.HLSL Effect（效果框架）

进行到这里，读者可能会觉得使用着色器多少有些繁琐，Effect（效果框架）被提出以解决这些问题。作为一种方法，Effect 简化了使用着色器的操作；作为一个框架，Effect 把顶点着色器和像素着色器有机地组织了起来。

4.1Effect 代码结构

一个 Effect 效果代码的结构如下：

```
//effect
technique T0
{
    pass P0
    {
      ...
    }
}

technique T1
{
    pass P0
    {
      ...
    }
    pass P1
    {
      ...
    }
}

...
technique Tn
{
    pass P0
    {
      ...
    }
}
```

首先理解三个术语 effect（效果）、technique（技术）、pass（过程），所幸这三个术语从字面意思上就能得到很好的诠释。

要实现一种效果 effect，可以使用多种技术 technique，而每种技术中可能使用多个过程 pass 进行渲

染，这样就构成了上述 effect 包含多个 technique，technique 又包含多个 pass 的代码结构。

理解了代码结构，effect 知识就已经掌握了大半，下面我们直接使用一个程序实例对 effect 进行介绍。

4.2 用 Effect 实现多纹理化效果

前面我们介绍了一个使用像素着色器实现的多纹理化，这里用 Effect 框架重新给于实现，读者可以比较两者之间的异同，体会 Effect 框架给我们带来了哪些方面的改善。

4.2.1 着色器

下面是着色器代码，该代码存储于 Effect.txt 中，代码中包含了一个顶点着色器和一个像素着色器和一个 Effect 效果框架。

```
//-----------------------------------------
//                顶点着色器
//-----------------------------------------
matrix WVPMatrix;

struct VS_INPUT
{
    vector position : POSITION;
    float2 uvCoords0 : TEXCOORD0;
    float2 uvCoords1 : TEXCOORD1;
};

struct VS_OUTPUT
{
    vector position : POSITION;
    float2 uvCoords0 : TEXCOORD0;
    float2 uvCoords1 : TEXCOORD1;
};

VS_OUTPUT VS_Main(VS_INPUT input)
{
    VS_OUTPUT output = (VS_OUTPUT)0;

    output.position = mul(input.position, WVPMatrix);

    output.uvCoords0 = input.uvCoords0;
    output.uvCoords1 = input.uvCoords1;

    return output;
}
```

```
//-----------------------------------------
//                像素着色器
//-----------------------------------------
vector Scalar;

texture Tex0;
texture Tex1;

sampler Samp0 =
sampler_state
{
      Texture = <Tex0>;
      MipFilter = LINEAR;
      MinFilter = LINEAR;
      MagFilter = LINEAR;
};

sampler Samp1 =
sampler_state
{
      Texture = <Tex1>;
      MipFilter = LINEAR;
      MinFilter = LINEAR;
      MagFilter = LINEAR;
};

struct PS_INPUT
{
    float2 uvCoords0 : TEXCOORD0;
    float2 uvCoords1 : TEXCOORD1;
};

struct PS_OUTPUT
{
    float4 Color : COLOR0;
};

PS_OUTPUT PS_Main(PS_INPUT input)
{
```

```
    PS_OUTPUT output = (PS_OUTPUT)0;
    output.Color = tex2D(Samp0, input.uvCoords0)*Scalar.x + tex2D(Samp1, input.uvCoords1)*Scalar.y;
    return output;
}

//--------------------------------------------
//              效果框架
//--------------------------------------------
technique T0
{
    pass P0
    {
        vertexShader = compile vs_1_1 VS_Main();
        pixelShader = compile ps_1_1 PS_Main();
    }
}
```

注意程序中是如何使用效果框架将顶点着色器和像素着色器组织起来的：

```
    pass P0
    {
        //着色器类型            版本号  入口函数名称
        vertexShader = compile vs_1_1 VS_Main();
        pixelShader = compile ps_1_1 PS_Main();
    }
```

也可以直接将着色代码写在 pass 过程中，相关用法请读者参看 DirectX 文档：

```
    pass P0
    {
        //这里书写着色器代码
        …
    }
```

有了之前的基础，着色器代码读者应该很容易理解，下面具体介绍如何在应用程序中使用 Effect。

4.2.2 应用程序

```
…
/**********************顶点格式定义*****************/
struct CUSTOMVERTEX
{
    //定点位置坐标
    float x,y,z;
    //两套纹理坐标;
    float tu0, tv0;
```

```
    float tu1, tv1;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_TEX2)
…
/**********************声明变量*****************/
//Effect 效果指针
ID3DXEffect *g_pEffect                    = 0;

//常量句柄
D3DXHANDLE WVPMatrixHandle                = 0;
D3DXHANDLE ScalarHandle                   = 0;
D3DXHANDLE Tex0Handle                     = 0;
D3DXHANDLE Tex1Handle                     = 0;
D3DXHANDLE TechHandle                     = 0;

//四边形顶点缓存
LPDIRECT3DVERTEXBUFFER9 quadVB    = NULL;
//两个纹理
LPDIRECT3DTEXTURE9 quadTexture0 = NULL;
LPDIRECT3DTEXTURE9 quadTexture1 = NULL;
…

/**********************初始化应用程序*****************/
//定义四边顶点模型
CUSTOMVERTEX quad[] =
//    x          y          z       tu0     tv0     tu1     tv1
{{-3.0f, -3.0f, 10.0f, 0.0f, 1.0f, 0.0f, 1.0f},
{ -3.0f,    3.0f, 10.0f, 0.0f, 0.0f, 0.0f, 0.0f},
{    3.0f, -3.0f, 10.0f, 1.0f, 1.0f, 1.0f, 1.0f},
{    3.0f,    3.0f, 10.0f, 1.0f, 0.0f, 1.0f, 0.0f}};

//设置顶点缓存
void *ptr = NULL;
g_pd3dDevice->CreateVertexBuffer(sizeof(quad),
                                 D3DUSAGE_WRITEONLY,
                                 0,
                                 D3DPOOL_MANAGED,
                                 &quadVB,
                                 NULL);
quadVB->Lock(0, 0, (void**)&ptr, 0);
```

```
memcpy((void*)ptr, (void*)quad, sizeof(quad));
quadVB->Unlock();

//创建纹理
D3DXCreateTextureFromFile(g_pd3dDevice, "chopper.bmp", &quadTexture0);
D3DXCreateTextureFromFile(g_pd3dDevice, "Bleach.jpg", &quadTexture1);

//检测像素着色器是否被支持
D3DCAPS9 caps;
g_pd3dDevice->GetDeviceCaps(&caps);
if(caps.PixelShaderVersion < D3DPS_VERSION(1, 1))
{
    MessageBox(0, "NotSupport Pixel Shader - FAILED", 0, 0);
    exit(0);
}

//创建 Effect 效果
ID3DXBuffer* errorBuffer            = 0;

HRESULT hr = D3DXCreateEffectFromFile(g_pd3dDevice,
                                      "Effect.txt",
                                      0,
                                      0,
                                      D3DXSHADER_DEBUG,
                                      0,
                                      &g_pEffect,
                                      &errorBuffer);

// output any error messages
if(errorBuffer)
{
    MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
    ReleaseCOM(errorBuffer);
    exit(0);
}

if(FAILED(hr))
{
    MessageBox(0, "D3DXCreateEffectFromFile() - FAILED", 0, 0);
    return false;
```

```
}

//得到各常量句柄
WVPMatrixHandle = g_pEffect->GetParameterByName(0, "WVPMatrix");
ScalarHandle = g_pEffect->GetParameterByName(0, "Scalar");
Tex0Handle = g_pEffect->GetParameterByName(0, "Tex0");
Tex1Handle = g_pEffect->GetParameterByName(0, "Tex1");

//得到技术 technique T0 的句柄
TechHandle = g_pEffect->GetTechniqueByName("T0");

//设置纹理，注意这里设置纹理的方式比之前像素着色器简便很多
g_pEffect->SetTexture(Tex0Handle, quadTexture0);
g_pEffect->SetTexture(Tex1Handle, quadTexture1);
…
/*********************渲染*****************/
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
D3DCOLOR_XRGB(153,153,153), 1.0f, 0 );
g_pd3dDevice->BeginScene();

//为着色器变量 WVPMatrix 赋值
D3DXMATRIX matWorld, matView, matProj;
g_pd3dDevice->GetTransform(D3DTS_WORLD, &matWorld);
g_pd3dDevice->GetTransform(D3DTS_VIEW, &matView);
g_pd3dDevice->GetTransform(D3DTS_PROJECTION, &matProj);

D3DXMATRIX matWVP = matWorld * matView * matProj;
g_pEffect->SetMatrix(WVPMatrixHandle, &matWVP);

//为着色器全局变量 Scalar 赋值
D3DXVECTOR4 scalar(0.5f, 0.5f, 0.0f, 1.0f);
g_pEffect->SetVector(ScalarHandle, &scalar);

//设置定点格式、绑定数据流
g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
g_pd3dDevice->SetStreamSource(0, quadVB, 0, sizeof(CUSTOMVERTEX));

//注意下面使用 effect 框架进行渲染的方法
//设置要使用的技术
g_pEffect->SetTechnique(TechHandle);
```

```
//遍历技术中包含的所有过程进行多次渲染
UINT numPasses = 0;
g_pEffect->Begin(&numPasses, 0);
for(UINT i = 0; i<numPasses; ++i)
{
    //开始过程
    g_pEffect->BeginPass(i);
    //绘制图形
    g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
    //结束过程
    g_pEffect->EndPass();
}
//结束使用技术
g_pEffect->End();

g_pd3dDevice->EndScene();
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
…
```

以上是应用程序中使用 Effect 框架的代码，可以看到 Effect 在着色器加载、着色器变量赋值、顶点着色器和像素着色器配合使用等方面做出了简化，这里只是个简单的例子，当读者深入学习 Effect 的时候，会了解到更多 Effect 框架为着色器编程提供的方便。

编译运行程序，运行效果如图 4.1 所示，这和第三章使用像素着色器实现的多纹理化效果是一样的。


纹理一

纹理二

混合后纹理三

图 4.1

结语

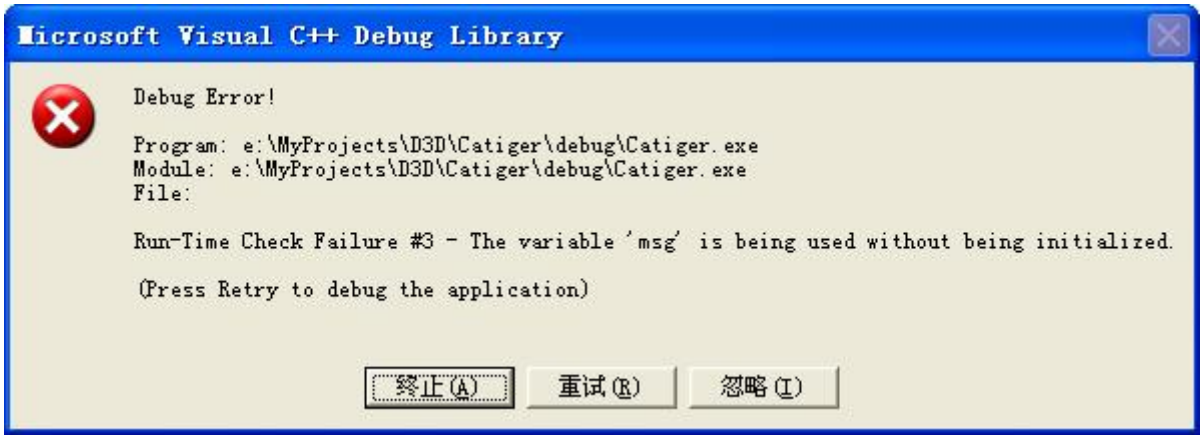
至此，HLSL初级内容介绍完毕，相信读者已经对HLSL、着色器、Effect等概念有了比较深入的理解，并且掌握了HLSL编程的基本方法，文章中裁去了对HLSL语法等细节的讨论，相关内容请查阅DirectX文档，教程配套程序源码放在http://bbs.gameres.com/upload/sf_200721615320.rar，读者可自行下载。

希望本教程能够帮助大家穿越 HLSL 之门，进入丰富多彩的 Direct3D 渲染世界。

参考资料

[1] Microsoft DirectX SDK (April 2006) Documentation；
[2] 翁云兵《3D 游戏程序设计入门》；
[3] Jim Adams《DirectX 高级动画制作》；
[4] Developer Relations http://www.gesoftfactory.com/developer/；
[5] 刘飞熊《Direct3D 中的 HLSL》；

源程序在 vc6 下调试，在 vs2003/2005 下编译运行可能会出现如下问题：



解决方法：

源程序主消息循环前加一句 PeekMessage(&msg, NULL, 0, 0, PM_REMOVE); 即可：

```
// The message loop
PeekMessage(&msg, NULL, 0, 0, PM_REMOVE); //添加该句
while(msg.message != WM_QUIT)
{
 if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
 {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
 }
 Game_Main();
}
```

可能是编译器的差异，给读者造成不便，请大家谅解:)