CHAPTER 10

Vertex and Pixel Shaders

"To know a thing well, know its limits. Only when pushed beyond its tolerances will true nature be seen."
——Paul Mu'adib, from the movie Dune

In this chapter, you are going to learn how to program *shaders*—specifically, vertex and pixel shaders. In essence, Direct3D allows you to code little functions that operate on each vertex of your world or finished pixel of your scene, using a special Direct3D minilanguage that looks a lot like assembly language. This enables you to create spectacular effects and still be sure that your code will run as fast as possible on all graphics cards. Behind the scenes, your vertex and pixel shader code can run on different hardware, depending on the graphics card used. For advanced cards, your code can run *on the card itself* if the card has a little CPU just for running shader code. On less advanced cards, Direct3D can push the shader code through your main CPU or do a combination, pushing some code through the CPU and the rest through the graphics card.

You don't have to worry about where your code actually runs. All you need to know is that on a given system configuration, Direct3D can set up things so that the vertex and pixel shader code runs as fast as possible. This means that any effects you create using the shaders will run as fast as possible(!).

Prepare yourself—shaders are weird little creatures. You might not immediately see the benefits to using a pixel or vertex shader, but stick with them. You will soon appreciate the power they possess.

Why Shaders?

In the beginning of 3D graphics programming, the graphics cards were simple, which meant that the API controlling them could also be simple. For example, if the state of the art in graphics cards does one texture stage and maybe a handful of blending operations, it makes sense to control that card by using a set of modes. You make API calls to set the texture mode and the blend mode, and you're done. Your API is simple, and it can do everything your card does.

Now let's crank up the complexity of the card. All of a sudden, you need more and more mode API calls to deal with the card's various features. You have multiple texture stages, different color- and alpha-blending operations for each stage, and more addressing modes, bump mapping, and so on. As you can see by the size of the Direct3D API, you need a huge number of API functions to deal with all those modes.

Because graphics cards continue to become more complex, Microsoft implemented a far better approach. Rather than use API calls to set modes on the card, you just write code for a virtual machine in an assembly-like programming language. With a code paradigm in place, you no longer have to worry about carefully setting each of the card's dozens of available modes to achieve the effect you're going for. Instead, you learn the architecture of the virtual machine and write code for that architecture to accomplish your task.

Vertex and pixel shaders make it much easier for you to create unique special effects for your applications. They're a much cleaner, more elegant solution than the texture stage states and rendering states, and they also enable you to do much more interesting things.

Now you will see how they work, starting with vertex shaders.

Vertex Shaders

First, I will nail down what exactly a vertex shader is.

Vertex shaders replace the transformation and lighting pipeline you learned about in Chapter 5, "3D Concepts" (see Figure 10.1).

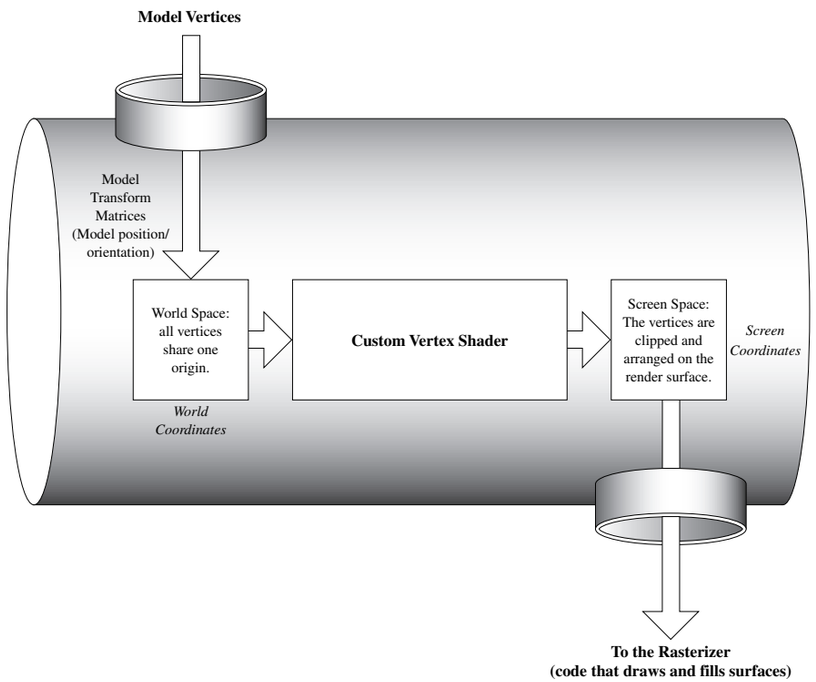

Figure 10.1

Vertex shaders replace most of the transformation and lighting pipeline.

A *vertex shader* takes a vertex as input and outputs a transformed, lit vertex. Now, depending on how you define your vertex format, a vertex can consist of many things: diffuse color, texture coordinates, a normal, you name it. This means that your vertex shader can have several inputs and outputs. In essence, though, a vertex shader is nothing more than a function (see Figure 10.2). It has a specific set of inputs, it runs some code against those inputs, and it outputs a specific set of things.

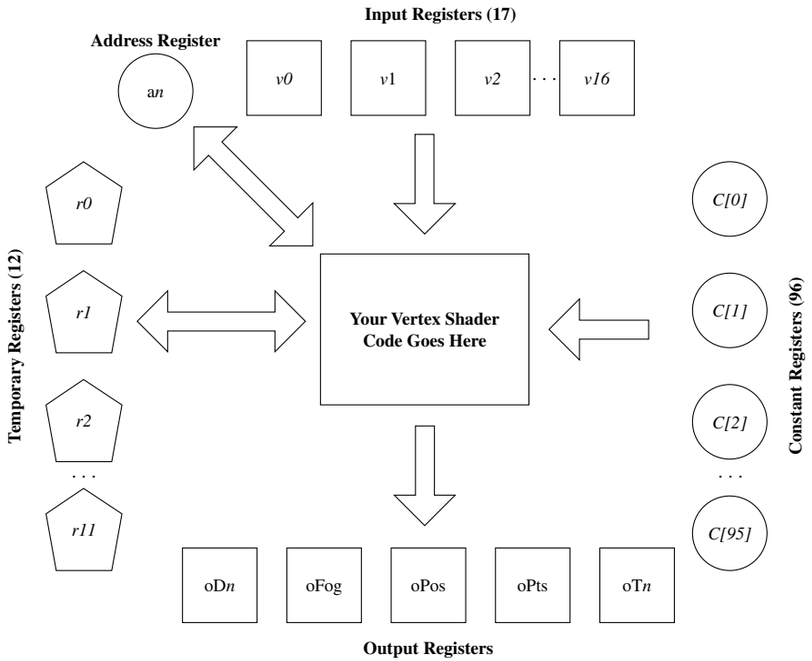

Figure 10.2

The vertex shader virtual machine.

To use a vertex shader, you first write the code for it and store the code in a text file, usually with the extension `.s`. After that's done, inside your main program, you use `D3DX` functions to load that file and compile it. When the `D3DX` functions succeed, they give you back a handle to your vertex shader. You then give this handle to the `SetVertexShader` method of `IDirect3DDevice8`, and that sets your vertex shader active.

Wait a minute, though—you have seen `SetVertexShader` before, in the first 3D sample program. You might not realize it, but you have been using a vertex shader ever since you rendered your first 3D scene. Direct3D comes with a default vertex shader. To use that default vertex shader, you call `SetVertexShader`, but rather than give it a vertex shader handle, you give it a combination of flexible vertex flags—which is exactly what you have been doing.

The default shader is special because various Direct3D interfaces can control its calculations. For example, you influenced the calculations of the default vertex shader when you set up different light types using `IDirect3DDevice8` methods such as `SetLight` or `LightEnable`, in Chapter 7 "Lighting".

You can't control your custom vertex shaders the same way, but that's okay—you're not losing any functionality. Your custom vertex shaders can still do everything the default shader does, only now, rather than call methods, you get your vertex shader to do your bidding by manipulating its inputs directly. In addition to the required vertex inputs, vertex shaders also take constants as inputs. These constant values can be set inside your main program, before you start pumping vertices. This means that you can communicate with your shader code by passing values from your main program into its constants.

Effects You Can Make Using Vertex Shaders

The ability to write your own vertex-processing code blows the doors open for many interesting effects. Here are just a couple examples, taken from the DirectX SDK help file:

- **Waves**. As your vertex shader processes each vertex of your landscape, it can manipulate the vertex's `z` value to create a wave or ripple effect. This is a killer technique for generating realistic water. You use the constants to pass in the parameters of the wave you want the shader to create.
- **Muscles**. As your vertex shader processes each vertex of your model, it can distort some of the vertex's coordinates, based on the shape and position of a sphere. You can use this to simulate some guy's ripped bicep moving as he lifts something. You can also use this technique to achieve the "There's something crawling under my skin!" effect, such as when Neo dives into the agent at the end of *The Matrix*. You can pass the coordinates and shape of the perturbing sphere into your shader through the constant registers, so by specifying different spheres, you can use the same shader for several effects.
- **Bones**. Your shader can move the vertices of your model, based on the position and weight of your model's "bones," enabling you to create a skeletal animation system for the characters in your game.

That should be enough to sell you on the power and flexibility of vertex shaders. Now you will get technical and start learning how they work.

Determining Whether a Device Supports Vertex Shaders

First things first—all this discussion is no good if the device you're on doesn't support vertex shaders. Use the members of the `D3DCAPS8` structure to answer your questions (see Table 10.1).

Table 10.1 D3DCAP8 Shader Members

Member	Description
`MaxPrimitiveCount`	The maximum number of primitives you can specify in each call to `DrawPrimitive`.
`MaxVertexIndex`	The maximum size of the vertex indices you can use for hardware vertex processing.
`MaxStreams`	The maximum number of concurrent data streams you can have.
`MaxStreamStride`	The maximum data stream for one data stream stride.
`MaxVertexShaderConst`	The maximum number of vertex shader constants you can use.
`VertexShaderVersion`	The level of support for vertex shaders.

`VertexShaderVersion` indicates the following levels of support:

- **0: DirectX 7.0.** This device does not support vertex shaders.
- **1.0: DirectX 8.0.** You can use vertex shaders, but you do not have the address register A0.
- **1.1: DirectX 8.0.** You can use vertex shaders, and you have the A0 address register. 1.1 is what you're hoping for, but the only difference between 1.0 and 1.1 is the missing address register A0.

Most of the cards out there now support vertex shaders, but it never hurts to check.

Specifying the Inputs to a Vertex Shader

After you verify that your card supports vertex shaders, the next order of business is to define exactly the inputs your vertex shader function will take.

You specify the inputs as an array of `DWORD`s. Direct3D deduces your vertex shader's inputs from an array of `DWORD`s, filled with certain flags and numbers. You don't usually specify these flags and numbers directly. Instead, you use the `D3DVSD_` macros, which save you the headache of making sure that you flip the right bits in the array.

Here is an example, which I will decipher. Say that your vertex structure looks something like this:

```
struct Vertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Normal;
    D3DCOLOR    Diffuse;
    D3DXVECTOR2 TexCoord0;
};
```

As you can see, you have a position (specified by a three-element vector), a normal, a diffuse color, and one set of (u,v) texture coordinates (specified by a two-element vector). That's typical, so, given what you learned in Chapter 6, "An Introduction to DirectGraphics," your flexible vertex flags would look like this:

```
DWORD dwFvf = D3DFVF_POSITION | D3DFVF_NORMAL | D3DFVF_DIFFUSE |
              D3DFVF_TEX0 | D3DFVF_TEXCOORDSIZE2(0);
```

Now you will learn how to tell Direct3D that your vertex shader takes these parameters as inputs. It all begins, of course, with an array of `DWORD`s. You don't know (or care) how long this array will be, so you define it like this:

```
DWORD dwVertexInputDecl[] =
{
};
```

To specify your inputs, you rely on the macros shown in Table 10.2.

For the first parameter of the `D3DVSD_REG` macro, you use any of the following values:

- `D3DVSDE_POSITION`
- `D3DVSDE_BLENDWEIGHT`
- `D3DVSDE_NORMAL`
- `D3DVSDE_PSIZE`
- `D3DVSDE_DIFFUSE`
- `D3DVSDE_SPECULAR`
- `D3DVSDE_TEXCOORD0`
- `D3DVSDE_TEXCOORD1`
- `D3DVSDE_TEXCOORD2`
- `D3DVSDE_TEXCOORD3`
- `D3DVSDE_TEXCOORD4`

Table 10.2 D3DVSD Macros

Macro	Example	Description
D3DVSD_STREAM	D3DVSD_STREAM(0)	Tells Direct3D from which stream to get the vertex data. This is usually the first macro you use.
D3DVSD_END	D3DVSD_END()	Tells Direct3D that this is the end of the DWORD array.
D3DVSD_REG	D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3) D3DVSD_REG(D3DVSDE_DIFFUSE, D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3) D3DVSD_REG(D3DVSDE_DIFFUSE, D3DVSDT_D3DCOLOR)	Specifies a data element of your vertex structure, along with that element's data type. You will use this macro the most often. The first parameter is the ID of the vertex component (color, position, and the like). All these IDs are #defines that start with D3DVSDE_ (for *Direct3D Vertex Shader Data Element*). The second parameter is the type of the data element. All these IDs start with D3DVSDT_ (for *Direct3D Vertex Shader Data Type*). Remember, the order in which you specify data elements must be the same as the order of the data members in your vertex structure so that things line up inside the vertex buffer.
D3DVSD_CONST	D3DVSD_CONST(8, 1), *(DWORD*)&diffuse[0], *(DWORD*)&diffuse[1], *(DWORD*)&diffuse[2], *(DWORD*)&diffuse[3], ...	Specifies a constant value. The first parameter tells Direct3D which constant register to begin filling with data, the second parameter tells Direct3D the number of constant *vectors* (not bytes—vectors are four DWORDs each!) to load.

Table 10.2 Continued

Macro	Example	Description
D3DVSD_NOP	D3DVSD_NOP()	Generates a NOP (*no operation*) token.
D3DVSD_SKIP	D3DVSD_SKIP(2)	Tells Direct3D to skip the specified number of DWORDs in the vertex. This can be useful if you have additional information in your vertex structure that you don't want to send to your shader.
D3DVSD_STREAM_TESS	D3DVSD_STREAM_TESS()	Sets the tesselator stream (for advanced vertex shader operations).
D3DVSD_TESSNORMAL	D3DVSD_TESSNORMAL(0,1)	Specifies that you want to enable tesselator-generated normals. The first parameter specifies the input stream; the second parameter specifies the stream where the normals will be written to. This command is used for advanced shader techniques.
D3DVSD_TESSUV	D3DVSD_TESSUV(5)	Specifies that you want to enable tesselator-generated surface parameters. The parameter specifies the output stream where you'd like the parameters to go.

- D3DVSDE_TEXCOORD5
- D3DVSDE_TEXCOORD6
- D3DVSDE_TEXCOORD7

For the second parameter of the D3DVSD_REG macro, you use any of the following values:

- **D3DVSDT_D3DCOLOR.** A D3DCOLOR
- **D3DVSDT_FLOAT1.** One float
- **D3DVSDT_FLOAT2.** Two floats
- **D3DVSDT_FLOAT3.** Three floats
- **D3DVSDT_FLOAT4.** Four floats
- **D3DVSDT_UBYTE4.** 4 bytes

As you can see, you probably won't use some of the D3DVSD_ macros. The most important ones are D3DVSD_REG, which registers an element of your structure, D3DVSD_STREAM, which tells Direct3D which vertex stream you want to use, and D3DVSD_END, which ends the array.

Given this simple vertex structure, you can now define the shader inputs:

```
struct Vertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Normal;
    D3DCOLOR    Diffuse;
    D3DXVECTOR2 TexCoord0;
};
DWORD dwDecl[] =
{
    D3DVSD_STREAM( 0 ),
    D3DVSD_REG( D3DVSDE_POSITION,  D3DVSDT_FLOAT3 ),
    D3DVSD_REG( D3DVSDE_NORMAL,    D3DVSDT_FLOAT3 ),
    D3DVSD_REG( D3DVSDE_DIFFUSE,   D3DVSDT_D3DCOLOR
),
    D3DVSD_REG( D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2 ),
    D3DVSD_END()
};
```

See how your vertex structure and input declaration fit together? For each element in the vertex structure, you put a corresponding D3DVSD_REG command inside the input declaration.

CAUTION

You must specify the inputs to your shader in the exact order you declare them in the structure. Otherwise, your vertex data will not line up correctly in memory. If you have a member of your vertex structure that you don't want to input to your shader function, use the D3DVSD_SKIP macro to skip over it.

Vertex Shader Assembly Language

After you specify your inputs, you can write the code for your vertex shader. You do this using a language that resembles assembly language. To learn this language, you have to know the registers (variables) with which you can work and the operations you can perform on those variables.

You will start with the registers.

Vertex Shader Registers

Table 10.3 summarizes the registers available for vertex shader programming. Keep in mind that when you see an *n* in the register name, it stands for an integer—for example, vn stands for v0, v1, v2, and so on.

Table 10.3 Vertex Shader Registers

Register	Description
an	Address registers. Currently, there is only one address register, a0, that only has one element, x (so a0.x). You can use the address register as a relative address into the array of constants, as in c[a0.x + n]. In this line, you're using the a0.x address register as in index into the array of constants. You can also do the following: c[a0.x + 5]. That is, you can add a number onto the address register and use that sum as an index into the array of constants.
c[n]	Constant registers. The vertex shader language supports at least 96 constant registers—the D3DCAPS8 structure gives the exact number of constants supported. If you try to read a constant whose index is out of range, you get back (0.0, 0.0, 0.0, 0.0). Your program can put values into the constant registers using the SetVertexShaderConstant method of IDirect3DDevice8.

Table 10.3 Continued

Register	Description
`rn`	Temporary registers you can use for calculation. The vertex shader language supplies 12 temporary registers—each register is four floating-point values. You can use up to three temporary registers per operation. Note that Direct3D destroys the contents of the temporary registers each time your shader finishes executing. This means that your vertex shader code must write to a temporary register before it reads from it—failure to do so results in a compile error.
`vn`	The input registers. The vertex shader language supplies 17 input registers.
`oDn`	Output data registers used to output vertex color data. These data registers are interpolated by the system and then sent into a pixel shader. You use them most often for color information.
`oFog`	The output fog factor. Direct3D uses only the x component of this 4D float.
`oPos`	The output position. Arguably the most important thing you calculate, this register contains the final position in homogenous clipping space.
`oPts`	The output point-size registers. Direct3D uses only the x component of this 4D float.
`oTn`	The output texture coordinate registers. Direct3D uses these values to determine which texels correspond to this vertex.

The vertex shader language provides several different inputs, detailed here:

- **`v0`.** Position
- **`v1`.** Blend Weight
- **`v2`.** Blend Indices
- **`v3`.** Normal
- **`v4`.** Point Size
- **`v5`.** Diffuse Color
- **`v6`.** Specular Color
- **`v7`.** Texture Coordinates 0
- **`v8`.** Texture Coordinates 1
- **`v9`.** Texture Coordinates 2
- **`v10`.** Texture Coordinates 3
- **`v11`.** Texture Coordinates 4
- **`v12`.** Texture Coordinates 5
- **`v13`.** Texture Coordinates 6
- **`v13`.** Texture Coordinates 7
- **`v13`.** Position 2
- **`v13`.** Normal 2

To recap in more detail, the purpose of a vertex shader is to take the inputted `vn` registers, perform calculations on them (using the `cn` constant registers, the `an` address register, and the `rn` temporary registers if needed), and finally output the `o` registers your program uses: `oDn`, `oFog`, `oPos`, `oPts`, and `oTn`—the color, fog factor, position, point size, and texture coordinates of the inputted vertex.

Now you will look at the operations you can perform on these registers.

HELP REFERENCE

The DirectX SDK help file has more detailed descriptions of each register. See DirectX 8.0\DirectX Graphics\Direct3DX Shader Assemblers Reference\Vertex Shader Assembler Reference\Registers.

Vertex Shader Instructions

Table 10.4 summarizes the instructions available in the vertex shader language.

Note that the `lit` instruction assumes that the source vector `vSrc0` contains the following information:

- `vSrc0.x` = N*L (the dot product between the vertex's normal and its direction to the light)
- `vSrc0.y` = N*H (the dot product between the vertex's normal and its half vector)
- `vSrc0.z` = ignored
- `vSrc0.w` = The power, in the range of –128 to +128

The Direct3DX team has also included the macro instructions shown in Table 10.5. These macros wrap up several instructions into one macro instruction, making for more readable and less error-prone coding.

Table 10.4 Vertex Shader Instructions

Instruction	Syntax	Description
add	add vDest, vSrc0, vSrc1	Adds vSrc0 and vSrc1 together and puts the result in vDest.
dp3	dp3 vDest, vSrc0, vSrc1	Calculates the three-component dot product of vSrc0 and vSrc1 and puts the result in vDest.
dp4	dp4 vDest, vSrc0, vSrc1	Calculates the four-component dot product of vSrc0 and vSrc1 and puts the result in vDest.
dst	dst vDest, vSrc0, vSrc1	Calculates the distance vector and puts the result in vDest. This instruction assumes that vSrc0's components are (*ignored*, d*d, d*d, *ignored*), and vSrc1's components are (*ignored*, 1/d, *ignored*, 1/d). When the instruction is finished, vDest will contain (1, d, d*d, 1/d).
expp	expp vDest, vSrc0	The expp instruction allows you to calculate "2 to the power of vSrc0," with partial precision. This instruction stores the answer in vDest.z. Note that this operation is undefined if you pass in a negative number.
lit	lit vDest, vSrc0	The lit instruction allows you to calculate lighting coefficients from two dot products and a power.
log	log vDest, vSrc0	Provides $\log_2(x)$ calculations with partial precision. This instruction stores the answer in vDest.z.
mad	mad vDest, vSrc0, vSrc1, vSrc2	Multiplies vSrc0 and vSrc1, adds vSrc2 onto the product, and puts the results in vDest.
max	max vDest, vSrc0, vSrc1	Fills vDest with the largest components of vSrc0 and vSrc1. For example, if vSrc0.x = 0.5 and vSrc1.x = 0.75, then vDest.x = 0.75.

Table 10.4 Continued

Instruction	Syntax	Description
min	max vDest, vSrc0, vSrc1	Same as max but uses the smallest components. Fills vDest with the smallest components of vSrc0 and vSrc1. For example, if vSrc0.x = 0.5 and vSrc1.x = 0.75, then vDest.x = 0.5.
mov	mov vDest, vSrc0	This instruction simply moves the contents of vSrc0 into vDest.
mul	mul vDest, vSrc0, vSrc1	Multiplies vSrc0 and vSrc1 and puts the result in vDest.
rcp	rcp vDest, vSrc0	Computes the reciprocal (1/x) of the source scalar and puts it in vDest.
rsq	rsq vDest, vSrc0	Computes the reciprocal square root of the source scalar and puts it in vDest.
sge	sge vDest, vSrc0, vSrc1	If vSrc0 >= vSrc1, then vDest = 1.0. Otherwise, vDest = 0.0. Even though the vertex shader programming language doesn't support conditional statements per se, you can use this function to achieve some primitive if statement functionality.
slt	slt vDest, vSrc0, vSrc1	Just like Store Greater Than/Equal (sge), only it's Store Less Than (slt). If vSrc0 < vSrc1, then vDest = 1.0. Otherwise, vDest = 0.0. Even though the vertex shader programming language doesn't support conditional statements per se, you can use this function to achieve some primitive if statement functionality.
sub	sub tDest, tSrc0, tSrc1	Subtracts two sources. When this instruction finishes, tDest = tSrc0 – tSrc1.

Table 10.4 Continued

Instruction	Syntax	Description
def	def vDest, fVal0, fVal1, fVal2, fVal3	Allows you to define a constant register by putting four floating-point values into it. You can accomplish the same thing by calling SetVertexShaderConstant; this is just another means to the same end.
vs	vs.MainVer.SubVer	Allows you to specify a version number for this shader. Version numbers can consist of a main version, followed by a sub version, that is, 1.0. Note that Direct3D requires this instruction to be at the beginning of all vertex shaders.

Table 10.5 Vertex Shader Macro Instructions

Instruction	Syntax	Description
exp	exp vDest, vSrc0	Provides "two to the power of vSrc0" with full precision. This takes 12 instructions to do, but hey, when you need the precision, those 12 instructions are a small price to pay. This operation takes its input from the w channel of vSrc0 (vSrc0.w).
frc	frc vDest, vSrc0	Puts the fractional component of vSrc0 into vDest. Each component of vDest will be in the range of 0.0–1.0. This takes three instructions to do, and it only writes the x and y components.

Table 10.5 Continued

Instruction	Syntax	Description
log	log vDest, vSrc0	Provides $\log_2(x)$ calculations, with full precision. This macro also expands into 12 instructions.
m3x2	m3x2 rDest, vSrc0, mSrc1	Computes the product of vSrc0 and the 3×2 matrix mSrc1. This macro expands into two instructions.
m3x3	m3x3 rDest, vSrc0, mSrc1	Computes the product of vSrc0 and the 3×3 matrix mSrc1. This macro expands into three instructions.
m3x4	m3x4 rDest, vSrc0, mSrc1	Computes the product of vSrc0 and the 3×4 matrix mSrc1. This macro expands into four instructions.
m4x3	m4x3 rDest, vSrc0, mSrc1	Computes the product of vSrc0 and the 4×3 matrix mSrc1. This macro expands into three instructions.
m4x4	m4x4 rDest, vSrc0, mSrc1	Computes the product of vSrc0 and the 4×4 matrix mSrc1. This macro expands into four instructions.

As you can see, the language supports essentially all the math tasks you will likely need to do while calculating vertices.

HELP REFERENCE

You can learn more about the instructions supported by the vertex shader language by looking in the DirectX documentation at DirectX 8.0\DirectX Graphics\Direct3DX Shader Assemblers Reference\Vertex Shader Assembler Reference\Instructions.

Vertex Shader Instruction Modifiers

The vertex shader language supports the component modifiers listed in Table 10.6. You can apply these modifiers to any of the `an`, `c[n]`, `rn`, or `vn` registers or any of the output registers.

Table 10.6 Vertex Shader Instruction Modifiers

Modifier	Description
`r.{x}{y}{z}{w}`	Destination mask. This allows you to mask off all but one of a vector's four components.
`r.[xyzw][xyzw][xyzw][xyzw]`	Source swizzle.
`-r`	Source negation.

Creating a Vertex Shader in Your Program

You can write all the vertex shader assembly code you want, but it won't do you a lick of good unless you can load and use the functions you've written inside your 3D application. Luckily, the process of loading and using a vertex shader is simple:

1. Write your vertex shader code, and store it in a text file, typically with the extension `VSH` (for *Vertex SHader*). This is the hard part.
2. Inside your application, call the `D3DXAssembleShaderFromFile` function, which loads and assembles your shader file. `D3DXAssembleShaderFromFile` gives you back an `ID3DXBuffer` interface containing the token stream of your assembled shader function.
3. Create the actual vertex shader by calling the `CreateVertexShader` method of `IDirect3DDevice8`, giving it the contents of the buffer you got in step 2. `CreateVertexShader` gives you back a `DWORD` handle to your shader.
4. Release the `ID3DXBuffer` interface you got in step 2.

In code, this process looks like the following:

```
LPD3DXBUFFER pCode;
DWORD dwShader;
DWORD dwDecl[] = { // the declaration for your shader
  D3DVSD_STREAM(0),
  D3DVSD_REG(D3DVSDE_POSITION,  D3DVSDT_FLOAT2),
  D3DVSD_END()
};
// Assemble the vertex shader from the file
if (FAILED(D3DXAssembleShaderFromFile( "MyCoolShader.vsh", 0,
  NULL, &pCode, NULL))) { /* handle errors! */ }
// Create the vertex shader
if (FAILED(m_pd3dDevice->CreateVertexShader( dwDecl,
  (DWORD*)pCode->GetBufferPointer(), &dwShader, 0 )) {
    /* handle errors! */
}
// Release the token buffer
pCode->Release();
```

CAUTION

When you're done with the vertex shader, remember to call `DeleteVertexShader`, giving it the `DWORD` of the vertex shader you want to delete.

Rendering, Using the Shader

The logical next step after loading a vertex shader is to tell DirectGraphics that you'd like to use this shader when you render. This is surprisingly easy. When you have your vertex shader's `DWORD`, you can pass that `DWORD` to the `SetVertexShader` method of `IDirect3DDevice8`, which activates your vertex shader. Here's a code snippet showing how that might look:

```
// Begin the scene
g_pd3dDevice->BeginScene();

// Make our quad the source for the vertex data
g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );

// set our vertex shader active
g_pd3dDevice->SetVertexShader( g_dwShader );

// draw some triangles
g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2 );

// End the scene
g_pd3dDevice->EndScene();
```

It really is that easy. Simply set your shader active, and then draw your triangles.

Passing Data to Your Shader, Using Constant Registers

You must do one last thing before your vertex shader will work correctly. Even the simplest vertex shader needs data above and beyond what it's getting from the geometry. For example, to transform, clip, and project your geometry properly, your shader has to know what the world, view, and projection matrices are. As you've seen, the vertex shader gets its data from the constant registers, but you haven't yet looked at how the constant registers are filled up in the first place.

The way you do this is by calling the `SetVertexShaderConstant` method of `IDirect3DDevice8`, explained in Table 10.7:

```
HRESULT SetVertexShaderConstant(
  DWORD Register,
  CONST void* pConstantData,
  DWORD  ConstantCount
);
```

For example, say that your vertex shader is going to color all your vertices. It expects you to put the color you'd like it to use into constant register four (`c4`). Therefore, you use `SetVertexShaderConstant` to load up four floating-point values (corresponding to red, green, blue, and alpha) into `c5`. This translates into code that looks like the following:

Table 10.7 SetVertexShaderConstant Parameters

Parameter	Description
`Register`	The number of the constant register you want to load up, or, if you're loading many constant registers at once, the number of the constant register at which you want to start.
`pConstantData`	The data you want to plug in to the constant register(s). Make sure that the size of this data is four floats for every constant register you're setting. For example, if you're setting three registers, you must make sure that the data size is `12*sizeof(float)`.
`ConstantCount`	The number of registers you're setting.

```
// plug the diffuse color into constant register 5
float color[4] = { 1.0f, 1.0f, 0.0f, 1.0f };
g_pd3dDevice->SetVertexShaderConstant(5, color, 1);
```

In this code, you are setting the color to RGBA(1,1,0,1)—opaque yellow—and calling `SetVertexShaderConstant` to push those four floats into constant register four.

CAUTION

A common mistake involves incorrectly setting the constant count (the last parameter to `SetVertexShaderConstant`). Remember, each constant register is four floats, so if you're pushing four floats into one register, make sure that this is one, not four. Incorrectly specifying the number of values (instead of the number of registers) can trash your constants behind your back and lead to really hard-to-find bugs.

NOTE

You can also use the `def` command to set constant registers in a vertex shader. If the constants you're using aren't changing frame by frame, it might be better to set them inside the shader code itself, using `def`, rather than bother with setting them inside your actual C code.

A Simple Vertex Shader Example

You now have a complete understanding of how to use vertex shaders in your programs, but I'll bet that you're still a little confused about how to write the shader code itself. Let me walk you through a series of progressively more complex shaders and show you how these work.

Here is a very simple vertex shader. Fire up the `Ch10p1_SimpleVertexShader` program. Run it, and you will see nothing terribly interesting, just a yellow rotating square on a blue background (see Figure 10.3). However, plenty of new techniques are going on under the surface. Rather than use the fixed function pipeline to generate this image, you are using your own vertex shader.

Because we don't care about texturing or anything fancy, the shader behind this image is very small. Ready? Here it is:

```
vs.1.0
m4x4 oPos , v0, c0
mov  oD0, c4
```

That's it! Three lines, and they're not even long lines. The first line is the mandatory `vs` command, which DirectX requires at the beginning of all shaders. It simply tells the system that you will be using vertex shader 1.0 commands.

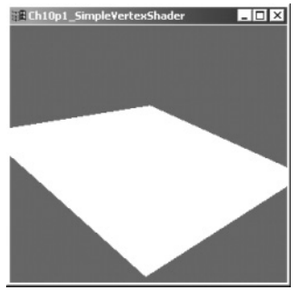

Figure 10.3

A simple vertex shader in action.

The next line is a matrix multiply. This line takes the incoming vertex position data (`v0`) and multiplies it by the matrix you've put in constant register zero. It stores the result of this multiplication in `oPos`, which is the position output register—what DirectGraphics looks at after the shader is done to determine the final position of the vertex.

"Wait a minute," you're probably saying, "What happened to the world, view, and projection matrices?" In the fixed function pipeline, the vertex position was first multiplied with the world matrix to get its position in world space. Then, the result of that was multiplied by the view matrix to get its position, as seen from a certain point in 3D space (the camera). Finally, that result was multiplied again, this time by the projection matrix to determine its position within the defined viewport.

As you know, you can combine several matrix multiplications into a single matrix. Multiply a matrix that scales something with a matrix that rotates something, and you end up with a matrix that does both the scale and the rotation at once. That's what's going on here. This code has multiplied the world, view, and projection matrices together and stuffed the combined world/view/projection matrix into `c0`. Here's the code that does this:

```
D3DXMATRIX mat;
D3DXMatrixMultiply( &mat, &g_matWorld, &g_matView );
D3DXMatrixMultiply( &mat, &mat, &g_matProj );
D3DXMatrixTranspose( &mat, &mat );
g_pd3dDevice->SetVertexShaderConstant(0, &mat,  4);
```

Ah-ha! Here you can see the multiplication fusing the three matrices together, followed by the call to `SetVertexShaderConstant` that places the fused matrix into constant register zero (`c0`).

You should now see how that second line of this little shader works. The `m4x4` macro multiplies the vector position and the fused world/view/projection matrix and puts the result in `oPos`. The result of all this is a vertex position equivalent to what you would end up with using the fixed function pipeline. The differences lie in how you get there and in the amount of freedom you have along the way. You could choose to leave the three matrices separate and let the shader combine them. You could use a different matrix for different vertices (for example, you could store the number of the matrix you would like to use for each vertex inside that vertex's data structure). Also, you could get fancy and use dozens of matrices, along with some vertex weight values, to calculate the final vertex position (as programmers do for bone calculations). The key idea here is that the shaders give you the freedom to do whatever you need to do.

That takes care of the vertex position. Now there's the third and final line, which sets the vertex color. It does this by copying constant register four (`c4`) into output data register zero (`oD0`).

As you might have guessed, you use `SetVertexShaderConstant` to stuff the vertex color you want (opaque yellow) into constant register four. The shader simply grabs that constant and uses it as the final color. Again, it doesn't have to be this way. If you want to do something bizarre, see what happens when you change `c4` to `c0`, causing the vertex shader to make the vertex's diffuse color dependent on the matrix you pass it.

Here's another simple shader:

```
vs.1.0
m4x4 oPos , v0, c0
mov  oD0, v5
```

In this code, you replace `c4` with `v5`. This subtle change makes the vertex shader obtain the color information from the diffuse color stored within the vertex structure. You can see this shader in action by making the `Ch10p1_SimpleVertexShader` program load `SimpleShader2.vsh` instead of `SimpleShader1.vsh`.

NOTE

It might seem strange to you that `oD0` isn't named `oColor` or something similar. Instead, it's named *output data register zero*, a much less precise name. This is because `oD0` doesn't necessarily have to be the color. The default DirectGraphics pixel shader uses `oD0` as the diffuse color, but if you write your own pixel shaders, you can use `oD0` for whatever you want—usually the color but occasionally not.

A Complex Vertex Shader Example

Are you starting to see how all this works together? Now take a look at a more complex vertex shader—one that does texturing and lighting, in addition to position and diffuse color. Here it is:

```
vs.1.0
m4x4 oPos, v0, c0

// lighting calculations
dp3 r0, v3, c4
mul oD0, r0.x, v5

// texture pass-through
mov oT0.xy, v7
```

Sample Program Reference

To see this shader in action, run the `Ch10p2_TexturingVertexShader` sample program.

The only thing terribly interesting in this new shader is the lighting calculation. A quick and easy way to calculate the amount of light falling on a vertex is to take that vertex's normal and dot it with the light vector (the vector specifying the direction in which the light is shining). Then, multiply that value by the vertex diffuse color to arrive at the final, lit color.

That's essentially what the shader does, in the two instructions under the light heading. The `c4` register contains the light vector, which you set up using the following code:

```
// set up the light
D3DXMatrixInverse(&matinv, NULL, &g_matWorld);
D3DXVECTOR3 vLight(0.0f, 0.0f, 1.0f);
D3DXVec3TransformNormal(&vLight, &vLight, &matinv);
D3DXVec3Normalize((D3DXVECTOR3*)&vLight, (D3DXVECTOR3*)&vLight);
vLight = -vLight;
g_pd3dDevice->SetVertexShaderConstant(4, &vLight, 1);
```

The light vector starts out as (0,0,1), which makes the light point directly into the scene. You then have to transform the light vector's normal by the inverse of the world matrix and normalize that result (remember, *normalization* means making the length of the vector equal to 1).

Don't worry about the math here. The important thing is that, in the last line of code, you plug a correct light vector in to register `c4`.

Now, back to the first lighting line in your shader. You are taking the dot product of `v3` and `c4` and putting the result in `r0`. You know where `c4` comes from, so look at where you get `v3`.

In the fixed function pipeline, `v3` is the normal, just as `v0` is the position and `v5` is the diffuse color. However, to use the normal in the first place, you must first add it to your vertex data structure:

```
struct CUSTOMVERTEX
{
  D3DXVECTOR3 position; // The position
  D3DXVECTOR3 norm;     // normal
  D3DCOLOR    color;    // The color
  FLOAT       tu, tv;   // The texture coordinates
};
```

Here you can see the new element, `norm`, sandwiched right in between `position` and `color`.

While you're at it, you should also change the custom FVF specification to match your structure. This is always a good habit to get into.

```
#define D3DFVF_CUSTOMVERTEX
  (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_DIFFUSE|D3DFVF_TEX1)
```


Here you can see the added `D3DFVF_NORMAL` tag.

Next, you tell DirectGraphics where inside your vertex structure it can find the normal. You do this by adding a line to your vertex shader declaration:

```
DWORD dwDecl[] =  {
  D3DVSD_STREAM(0),
  D3DVSD_REG(D3DVSDE_POSITION,  D3DVSDT_FLOAT3),
  D3DVSD_REG(D3DVSDE_NORMAL,    D3DVSDT_FLOAT3),
  D3DVSD_REG(D3DVSDE_DIFFUSE,   D3DVSDT_D3DCOLOR),
  D3DVSD_REG(D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2),
  D3DVSD_END()
};
```

In this code, notice that you add an additional binding for `D3DVSDE_NORMAL`. You tell DirectGraphics that the normal comes immediately after the position in your structure and that you consider a normal to be three floating-point values (the x, y, and z components of your vector).

CAUTION

I'll say it again here. It's very important to keep your declaration and your actual vertex structure in sync. If they're out of sync, your shader won't run, or it will run against incorrect `vn` input registers.

The only order of business left is to set the normals for each vertex. You do this at the same time you set the rest of the vertex data:

```
pVertices[0].position = D3DXVECTOR3(-2.0f, -2.0f, 0.0f);
pVertices[0].norm     = D3DXVECTOR3(0.0f, 0.0f, 1.0f);
pVertices[0].color    = D3DCOLOR_ARGB(255, 255, 0, 0);
pVertices[0].tu       = 0.0f;
pVertices[0].tv       = 0.0f;
```

The next line of the vertex shader lighting calculations multiplies the result of the dot product with your diffuse color and puts the result of that multiplication in the `oD0` output register. Now you have a vertex shader that calculates light.

The last line of the shader passes your texture coordinates unchanged from the input register (`v7`) to the output register (`oT0`). Note that you use the `.xy` modifier on `oT0` so that you store only the x and y components of the texture (because that's all you need—you're not using 3D textures).

That's it for the more complex shader.

Vertex Shader Wrap-Up

This section should give you at least a rudimentary understanding of how to write your own vertex shader. I hope that you will take what you learned and try to write more complex shaders. Vertex shaders enable you to create amazing effects (some of which are discussed in the 3D effects section of this book), so you should spend time learning and playing with them.

HELP REFERENCE

For more information on vertex shaders, go to DirectX 8.0\DirectX Graphics\Advanced Topics in DirectX Graphics\Vertex Shaders. Also, be sure to check out the .vsh files in the Samples\MultiMedia\Media directory of your SDK installation. Those are sample vertex shader files that show you how to implement many effects.

Pixel Shaders

Pixel shaders are similar to vertex shaders in that they are little snippets of code resembling assembly language. However, vertex shaders deal with vertex information—pixel shaders allow you to specify how different pixels and texels are blended together to create a final color value.

Effects You Can Make Using Pixel Shaders

Pixel shaders allow you to create and augment, using the pixel shader assembly language, all the texture stage setups discussed in Chapter 9, "Advanced Texturing." This includes the texture stage setups for light mapping, glow mapping, detail mapping, and multiple-texture blending. Additionally, pixel shaders allow you to do all sorts of neat things with light, including dot product lighting. What I'd like for you to do is to forget all that Chapter 8 and 9 stuff about texture stages and such because you're going to replace it with something even more powerful.

Pixel Shader Assembly Language

The pixel shader assembly language is very similar to the vertex shader assembly language, only it has different registers and instructions (see Figure 10.4 and Table 10.7).

Pixel Shader Registers

Table 10.8 summarizes the registers available in a pixel shader.

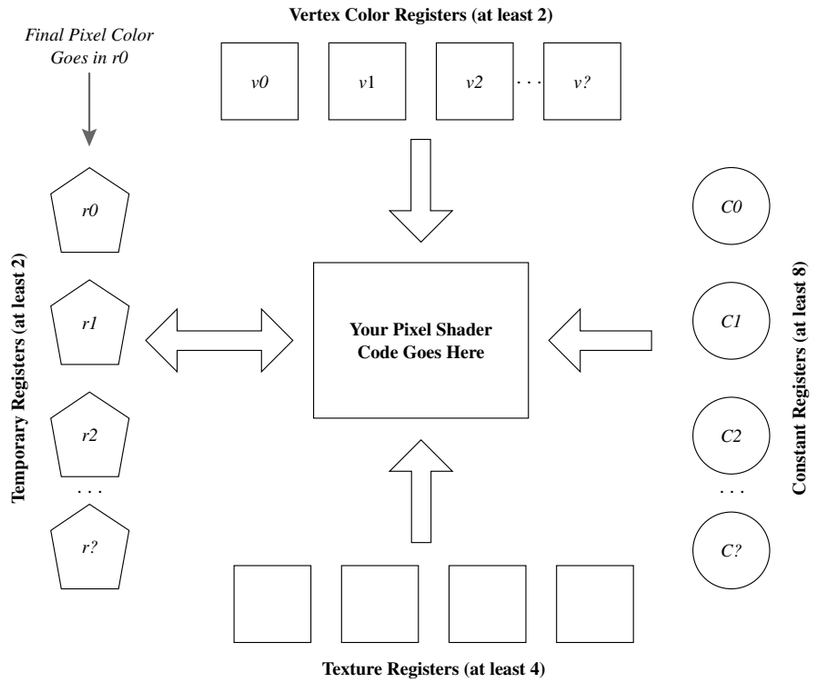

Figure 10.4

The pixel shader virtual machine.

In essence, all the color values come in through the `tn` and `vn` registers, and the job of the pixel shader is to calculate the final color value and put it in `r0`.

HELP REFERENCE

The DirectX SDK help file has more detailed descriptions of each register. See DirectX 8.0\DirectX Graphics\Direct3DX Shader Assemblers Reference\Pixel Shader Assembler Reference\Registers.

Generic Pixel Shader Instructions

To accomplish your goal, you can use any of the generic instructions listed in Table 10.9.

Table 10.8 Pixel Shader Registers

Register	Description
`cn`	Constant registers, which you set by calling the `SetPixelShaderConstant` method of `IDirect3DDevice8`. Constant registers are read-only. Direct3D supports at least eight constants, but you may only use up to two in each instruction.
`rn`	Temporary registers, which you can use to store intermediate results of a calculation. Temporary registers are read/write, and you can use up to two of them in a single instruction. Just like vertex shader temporary registers, however, these lose their values when the pixel shader completes, and it's illegal for you to read one without first writing something into it. Note that Direct3D takes `r0` as the final output value of the pixel shader. That's the color Direct3D eventually puts on the screen.
`tn`	Texture registers. You will have the same number of texture registers as maximum simultaneous supported textures. For each pixel, Direct3D initializes the texture registers so that they contain the texture colors of the appropriate texels from the various textures you've specified using the `SetTexture` command.
`vn`	Vertex color registers. Direct3D guarantees that you will have at least two input color registers, maybe more, depending on your 3D card. Direct3D initializes these registers with the color values obtained by iterating the color values output by the vertex shader (in other words, the vertex colors).

Table 10.9 Generic Pixel Shader Instructions

Instruction	Syntax	Description
`add`	`add tDest, tSrc0, tSrc1`	Puts the sum of `tSrc0` and `tSrc1` into `tDest`.
`cnd`	`cnd tDest, r0.a, tSrc0, tSrc1`	Compares the alpha value in `r0.a` to see whether it's greater than 0.5. If so, it puts `tSrc0` into `tDest`; otherwise, it puts `tSrc1` into `tDest`.
`dp3`	`dp3 tDest, tSrc0, tSrc1`	Puts the three component vector dot product of `tSrc0` and `tSrc1` into `tDest`.
`lrp`	`lrp tDest, tSrc0, tSrc1, tSrc2`	Linearly interpolates between `tSrc1` and `tSrc2` by the amount specified in `tSrc0`. Puts the result in `tDest`.
`mad`	`mad tDest, tSrc0, tSrc1, tSrc2`	Multiplies `tSrc1` by `tSrc2`, then adds `tSrc0`, and puts the result in `tDest`.
`mov`	`mov tDest, tSrc0`	A simple move operation. Puts `tSrc0` into `tDest`.
`mul`	`mul tDest, tSrc0, tSrc1`	Multiplies `tSrc0` and `tSrc1` and puts the result in `tDest`.
`sub`	`sub tDest, tSrc0, tSrc1`	Subtracts `tSrc1` from `tSrc0` and puts the result in `tDest`.
`def`	`def vDest, fVal0, fVal1, fVal2, fVal3`	Defines a constant, `vDest`, using four floating-point values.
`ps`	`ps.MainVer.SubVer`	Allows you to specify a version number for this shader. Version numbers can consist of a main version, followed by a subversion (that is, 1.0). Note that Direct3D requires this instruction to be at the beginning of all pixel shaders.

Texture-Addressing Instructions

Additionally, you can use the texture-addressing instructions and macros shown in Table 10.10.

Table 10.10 Texture-Addressing Instructions

Instruction	Syntax	Description
`tex`	`tex tDest`	Takes the current texel from the texture set at texture stage zero and puts it in `tDest`.
`texbem`	`texbem tDest, tSrc0`	Takes `tSrc0` as DuDv perturbation data and calculates the result in `tDest`. Typically used for bump mapping. Going into more detail on this would be beyond the scope of the book; however, I encourage you to crack open the online help and the DX SDK sample programs to see how this instruction works. This is a macro that expands into two instructions.
`texbeml`	`texbeml tDest, tSrc0`	Takes `tSrc0` as DuDv perturbation data with luminance information and calculates the result in `tDest`. Typically used for bump mapping. This is a macro that expands into two instructions.
`texcoord`	`texcoord tDest`	Puts the iterated texture coordinates for this stage into `tDest` as a color. You use this when declaring the texture registers you're using. For example, the line `texcoord t0` declares register `t0` as a color derived from its texture coordinates.
`texkill`	`texkill tDest`	Masks out the pixel if any texture coordinates are less than 0.
`texm3x2pad`	`texm3x2pad tDest, tSrc0`	Partially performs a 2×3 matrix multiply.

Table 10.10 Continued

Instruction	Syntax	Description
`texm3x2tex`	`texm3x2tex tDest, tSrc0`	Performs a 3×2 matrix multiply on the `tSrc0` color vector.
`texm3x3pad`	`texm3x3pad tDest, tSrc0`	Partially performs a 3×3 matrix multiply.
`texm3x3spec`	`texm3x3spec tDest, tSrc0, tSrc1`	Performs specular reflection and environment mapping using a 3×3 matrix.
`texm3x3tex`	`texm3x3tex tDest, tSrc0`	Performs a 3×3 matrix multiply on the `tSrc0` color vector.
`texm3x3vspec`	`texm3x3vspec tDest, tSrc0`	Performs specular reflection and environment mapping where the eye vector is not constant, using a 3×3 matrix.
`texreg2ar`	`texreg2ar tDest, tSrc0`	Samples this stage's texture at the 2D coordinates specified by the alpha and red components of `tSrc0`. In other words, it uses `tSrc0.a` as the u coordinate and `tSrc0.r` as the v coordinate and returns the color there.
`texreg2gb`	`texreg2gb tDest, tSrc0`	Samples this stage's texture at the 2D coordinates specified by the green and blue components of `tSrc0`. In other words, it uses `tSrc0.g` as the u coordinate and `tSrc0.b` as the v coordinate and returns the color there.

Pixel Shader Modifiers

The pixel shader language supports several modifiers:

- **Alpha-replicate**. You can replicate the alpha channel of a certain register across all channels before performing an instruction, by putting a `.a` after the register name. For example, the command `mul r0, r0, r1.a` replicates the alpha channel of `r1` into all four color channels, then multiplies using those channels, effectively modulating with the alpha value.

- **Invert**. You can invert the color components in a certain register by putting `1-` in front of that register. For example, the command `mul r0, r0, 1-r1` multiplies `r0` by the inverse of the color in `r1`.
- **Negate**. You can negate the color components in a register by putting a minus sign (–) in front of that register. For example, the command `mul r0, r0, -v1` multiplies `r0` by the negation of the color in `v1`.
- **Bias**. You can shift each channel in a register down by 0.5 by putting `_bias` after the register name. For example, the command `add r0, r0, t0_bias` shifts `t0` down 0.5 before adding it to `r0`.
- **Signed Scaling**. You can subtract 0.5 from each channel in a register and then scale the result by 2, by putting `_bx2` after the register name. For example, the command `dp3_sat r0, t1_bx2, v0_bx2` shifts the `t1` and `v0` registers down by 0.5 and then scales them by 2.

Creating and Using Pixel Shaders in Your Program

The process of setting up and using a pixel shader is, in my opinion, easier than what you must do for vertex shaders, so I'm not going to go through every step in mind-numbing detail.

Sample Program Reference

The `Ch10p3_PixelShader` sample program illustrates the process of loading and using a simple pixel shader.

You create a pixel shader the same way you create a vertex shader. First, you assemble the shader, via a call to `D3DXAssembleShaderFromFile` (or something). Then, you create the pixel shader by calling the `CreatePixelShader` of `IDirect3DDevice8`. You don't have to worry about passing `CreatePixelShader` a declaration, as you do for a vertex shader. Just pass it your buffer pointer and the address of the integer where you'd like it to put the pixel shader handle. You delete a pixel shader by calling the `DeletePixelShader` method of `IDirect3DDevice8`.

After you have a pixel shader loaded, you set it active by calling the `SetPixelShader` method of `IDirect3DDevice8`. Again, this is exactly how you use vertex shaders (only you call `SetVertexShader` instead). Keep in mind that you must still set your textures to point to the correct texture interfaces. If you need to set up any constant registers, call the `SetPixelShaderConstant` method of `IDirect3DDevice8`. Otherwise, just draw your primitives, end your scene, and DirectGraphics will execute your pixel shader code when the time comes.

A Simple Pixel Shader Example

Here's a really easy example, just to give you an idea of how to write a pixel shader. Later in this book you will be writing more advanced shaders.

The little pixel shader contained in the `Ch10p3_PixelShader` sample program contains three action-packed lines:

```
ps.1.0
tex t0
mov r0, t0
```

The first line is the obligatory header line, telling DirectGraphics that what follows is a pixel shader based on version 1.0 of the pixel shader language.

The next line declares a texture. This is something unique to pixel shaders. Essentially, declaring a texture gives you an opportunity to change the addressing mode for that texture. The texture declaration is the only place where you can manipulate the addressing for the texture (as opposed to the texel colors themselves). For example, it's in the texture declaration that you tell DirectGraphics that a certain texture is a bump map and should therefore be used to influence the colors of a different texture.

Yes, I'm being intentionally vague here because it's difficult to illustrate without going into a lot of graphics theory. The bottom line is, there are several texture-addressing instructions, but you use one more than all the others. The `tex` instruction tells DirectGraphics that what you're working with is a plain old texture. In other words, when you say `tex t0`, you're really saying, "DirectGraphics, I want you to figure out which texel we're on, given the (u,v) coordinates and the pixel we're currently processing. Then I want you to copy the color of that texel into `t0`."

After you have the appropriate color in `t0`, the next line copies it into `r0`. Recall that `r0` is the special register that DirectGraphics takes as the final pixel color. When you combine this line with the lines above it, all you are doing is taking the color of the texel to which this pixel corresponds, putting that color in `t0`, and copying `t0` into `r0`. This means that the output color (`r0`) is nothing more than the color of the texel this pixel corresponds to (`t0`).

This is texturing at its simplest. If you were writing a more complex pixel shader, you could play with `t0`—perhaps multiply it or add it to another texture color before you finally place it into `r0`. Additionally, you could choose to use a different texture-addressing instruction and change the way you calculate the appropriate texel or what you do with the texel color after you have found it. The sky is the limit. It just takes time and experimentation to understand all the instructions.

Sample Program Reference

As part of the DirectX SDK, Microsoft includes a great tool for writing pixel shaders and seeing the results of your shader code in real time. The program is named `MFCPixelShader` and can be found inside your DirectX SDK samples directory (<SDK Root>\Samples\Multimedia\Direct3D\MFCPixelShader).

Pixel Shader Wrap-Up

Using pixel shaders can greatly enhance the quality of your scenes, so take time to become familiar with their operations. The time you spend learning them will pay off big-time in the long run.

The `PixelShader` sample program on your CD (`Ch10p3_PixelShader`) illustrates how to set up and use a pixel shader. The program uses a pixel shader to texture a quad, instead of using the traditional fixed-function pipeline.

HELP REFERENCE

For more information on Pixel Shaders, go to DirectX 8.0\DirectX Graphics\Advanced Topics in DirectX Graphics\Pixel Shaders.

Chapter Wrap-Up

Wow! Talk about an amazing journey. Just a few hundred pages ago, you were learning how to write a Win32 program. Now you're programming graphics hardware on an assembly language level and have covered everything in between.

At this point, you should feel that you have all the programming and 3D knowledge necessary to make a 3D game. Of course, this doesn't mean that you *do* feel as though you know enough to write the next killer 3D game, but you certainly know much more about the internals of 3D programming (and Windows and DirectX programming in general) than when you began reading this book.

From here on, I'm going to talk about the special effects most games use, dissecting each one so that you can learn how to code it. I wouldn't suggest that you dive right into the effects just yet—you have come a long way, so take a breather and spend some time experimenting.

NOTE

The nVidia Web site contains many additional resources designed to help you understand and implement vertex and pixel shaders. On the Web site at http://developer.nvidia.com you'll find a virtual cornucopia of shader knowledge, in the form of white papers, sample programs, and tutorials.

About the Sample Programs

No enhancements to the coding style this chapter. Here are the programs:

- **`Ch10p1_SimpleVertexShader`.** Demonstrates a simple vertex shader in action. You walked through segments of this sample program in the section on vertex shaders.

- **`Ch10p2_TexturingVertexShader`.** Demonstrates a slightly more complex vertex shader—one that calculates texture coordinates for the vertices it processes. Most of this program is exactly the same as `Ch10p1_SimpleVertexShader`. The only difference is that the program loads a different vertex shader file (`VSH`) and manages a texture handle.
- **`Ch10p3_PixelShader`.** Demonstrates a simple pixel shader in action. You walked briefly through segments of this sample program in the section on pixel shaders.

Exercises

These are tough exercises, but you can tackle them.

Write vertex and pixel shaders that demonstrate some of the sample effects listed at the beginning of the chapter (waves, muscles, and bones). For vertex shaders, this means that you

1. Write a vertex shader that perturbs an incoming vertex stream into waves (for water).
2. Write a vertex shader that distorts an incoming vertex based on a sphere (for muscles).

For pixel shaders, this means that you

1. Write a pixel shader that performs a texture-blending operation of your choice.

Also, find something you can accomplish with a pixel shader that you can't do using texture stages.