Chapter 9

Vertex Shaders

"Great trees are good for nothing but shade."
George Herbert: *Outlandish Proverbs*, 1640

9.1 Overview

In chapter 6 and chapter 7, we saw how the fixed-function pipeline processes vertices from a model coordinate system into the coordinate system of the screen. While the fixed-function pipeline is versatile with many ways we can change the end result, its overall functionality is still limited. If we have an exotic lighting model, an exotic transformation method, an exotic method of computing fog, point sizes, etc., we would have to forego hardware acceleration with fixed-function processing and compute everything on the CPU using transformed vertices.

With vertex shaders we can have our cake and eat it too: we replace the fixed-function vertex processing with a small program we load into the hardware. The program receives our model space vertices as input and produces a vertex in homogeneous clip space as output, complete with per-vertex diffuse and specular colors, fog, transparency, texture coordinates and a point size.

A vertex shader exposes the vertex processing hardware as a vector-oriented CPU with an instruction set and sets of registers used to carry out the instructions. Different hardware may implement different levels of support for vertex shaders and this support is grouped roughly into a version of the shader architecture.

In this chapter we will explain the assembly language programming model available for vertex shaders with a complete discussion of the instruction sets and the available registers. Using high level shader language for vertex shaders is discussed in chapter 18. Following that we will look at how we can perform the elements of fixed-function vertex processing using a verion 1.1 vertex shader. We will discuss some examples of vertex shaders from the SDK as well

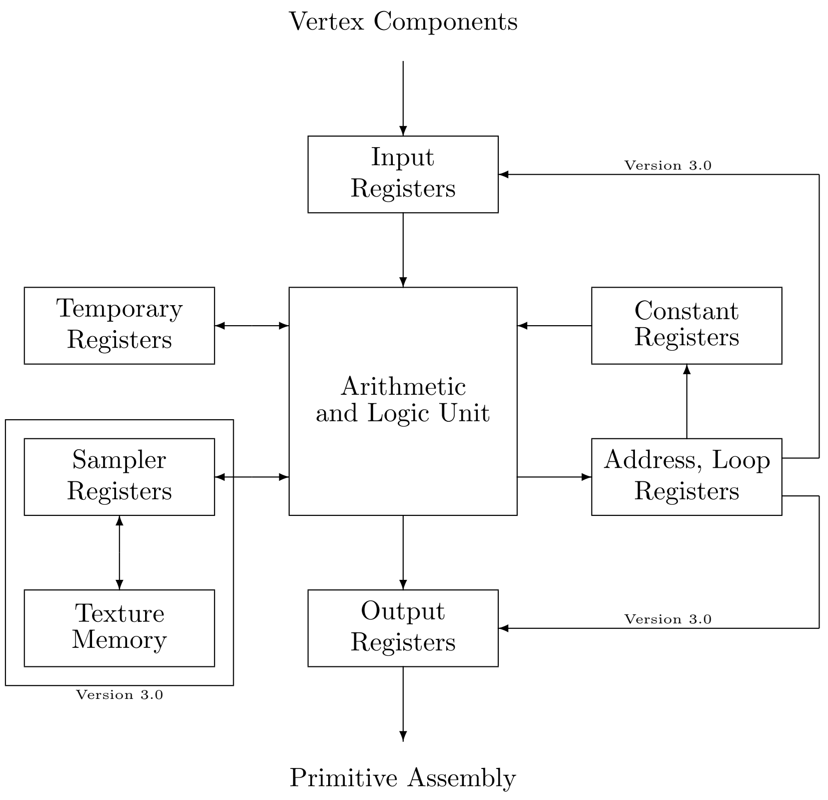

Figure 9.1: Vertex shader architecture.

as some examples of vertex processing that is possible with a vertex shader, but impossible with the fixed-function pipeline.

9.2 Vertex Shader Architecture

Direct3D exposes the details of different graphics processors as different shader architecture versions. Each version of the architecture can have different numbers and kinds of registers and different instruction sets. For the most part, the higher versions are an evolution of the lower versions, providing more instructions and fewer limits on the execution model. We'll look at shader version 1.1 in the most detail and cover the additions to the architecture of each succesive version.

DirectX 9.0c supports the following vertex shader architecture versions: 1.1, 2.0, 2.x and 3.0. The assembly language syntax for identifying these versions is `vs_1_1`, `vs_2_0`, `vs_2_x`, and `vs_3_0`, so you may see reference to the shader

	Shader Version			
	`vs_1_1`	`vs_2_0`	`vs_2_x`	`vs_3_0`
Instruction Count	128	256	≥ 256	≥ 512

Table 9.1: Vertex shader version instruction counts.

architecture by either syntax. Sometimes the different architecture versions are referred to as shader profiles or shader models; all these terms describe the same concept. Older versions of the SDK and documentation may also refer to the `vs_2_a` and `vs_2_b` shader versions; these versions have been incorporated into the `vs_2_x` shader version.

When you install the SDK, special software versions of the vertex shader architecture are available. These versions are designated as `vs_2_sw` and `vs_3_sw` and they are only available with software vertex processing on the reference rasterizer. Because of this restriction, they are slow and only available when the SDK is installed. They are only suitable for development purposes and should not be used in a shipping product. Each software version exposes all the features of the 2.x or 3.0 architecture and most of the shader validation requirements are relaxed for these software implementations.

All the architecture versions share a common execution model, shown in figure 9.1. A program, called a **shader**, is executed once for each vertex drawn in the scene. The shader contains one or more **instructions** and each instruction consists of an **opcode** and zero or more **operands**. The shader has access to five distinct groups of registers: input registers for vertex data, constant registers for rendering parameters, an address register for array lookups into the constant registers, temporary registers for storing intermediate results, sampler registers for sampling textures and output registers for the result of the shader. The instruction count limits for different shader versions are shown in table 9.1. The number of each kind of register is shown in table 9.2.

Each temporary register stores a four dimensional vector value and most instructions operate on four dimensional vector values. Each value is a floating-point quantity with a range and precision comparable to an IEEE single precision floating-point number, or about 6 decimal digits. Instructions are provided for the usual arithmetic operations, such as addition and multiplication, as well as vector arithmetic operations, such as dot product and vector matrix multiplication. Unlike a typical CPU, some versions of the shader architecture support no control flow with only linear sequencing. This restriction makes vertex shaders simpler and easier to accelerate with hardware.

9.2.1 Input Registers

The input registers supply the shader with the vertex data in the scene. The vertex components are mapped to semantics through a suitable vertex declaration, as discussed in section 5.8. The semantics are associated with input registers in the shader through the use of the `dcl_`*usage* instructions. The input registers are read-only and can only be used as the source of data in the vertex shader

Shader	Registers									
Version	`a0`	`aL`	`b`n	`c`n	`i`n	`o`n	`p0`	`r`n	`s`n	`v`n
`vs_1_1`	1	0	0	≥ 96	0	13	0	12	0	16
`vs_2_0`	1	1	16	≥ 256	16	13	0	12	0	16
`vs_2_x`	1	1	16	≥ 256	16	13	1	≥ 12	0	16
`vs_3_0`	1	1	16	≥ 256	16	12	1	32	4	16

Table 9.2: Vertex shader version register counts.

instructions. Each instruction can reference a single input register, although different source operands can apply different modifiers.

In shader version 3.0, the input registers can be indexed by the address register. This allows a loop to iterate over vertex elements and address them by the loop counter.

9.2.2 Constant Registers and the Address Register

Parameters that do not vary per-vertex are supplied to the shader through the constant registers. Floating-point constants are available in all shader versions, while integer and boolean constants are available in shader version 2.0 or higher. Each instruction can access only a single constant at a time, but multiple source operands may access the same constant register with different modifiers. The values in the constant registers may be defined in the shader with the `def`, `defb` and `defi` instructions. They can also be loaded from the device with the `SetVertexShaderConstantF`, `SetVertexShaderConstantB` and `SetVertexShaderConstantI` methods as described in section 9.14. You can think of values defined by shader instructions as "local constants" and those defined by the API as "global constants", with the local constants having precedence in any given shader.

The address register is a signed integer offset from a base constant register, addressing the constants as an array. The constant registers are read-only and the address register is write-only. When the register addressed is out of range for the constant registers, the returned value is always $\langle 0, 0, 0, 0\rangle$. The address register must be initialized before it can be used.

Shader version 1.1 provides a limited form of the address register. Only the x component of the register is available for indexing. The address register can only be set as the destination of a `mov` instruction. The value is rounded to the nearest integer value when it is loaded.

Shader versions 2.0 and higher provide a more general form of the address register. All four components of the register are available for indexing, allowing for addressing of different portions of the constant arrays simultaneously. The `mova` instruction is used to set the address register values.

9.2.3 Output Registers

The output registers are used to store the results of the shader computation that are passed to the rasterizer for source pixel generation. The output registers are write-only. The output of a vertex shader consists of a position in homogeneous clip space and per-vertex data to be interpolated by the rasterizer, such as colors and texture coordinates.

In shader versions before 3.0, the output registers are individually named. The available registers are the position register `oPos`, the color registers `oD0` and `oD1`, the fog register `oFog`, the point size register `oPts` and the texture coordinate registers `oT0` through `oT7`. Every vertex shader must write all four components of the position register `oPos`. The scalar values of the fog factor and point size are extracted from the x component of the `oFog` and `oPts` registers, respectively. The `oFog` and `oPts` values are clamped to the interval $[0, 1]$ before being passed on to the rasterizer.

In shader version 3.0, the output registers are declared with a `dcl_`*usage* instruction. Unlike previous shader versions, there is no requirement that any particular register be written. The output values of the shader are interpolated and presented as inputs to pixel processing. If the fixed-function pixel processing is used, then the vertex shader should declare outputs as appropriate for the fixed-function pipeline. As with the input registers in version 3.0, the output registers can be indexed with the address register for processing vertex elements from within a loop.

9.2.4 Temporary Registers

A significant amount of work generally takes place in a vertex shader. A bank of read-write temporary registers provide a vertex shader with a small scratchpad for storing intermediate results. The shader generally moves data from the inputs to the temporary registers, performs computations on the temporary registers and then writes the results to the output registers. The other register types can only be used once in a given instruction, but the temporary registers can be used multiple times. Up to three different temporary registers can be read and a one register written during a single instruction. Any attempt to read a temporary register before its value is written results in an error when the shader is created.

9.2.5 The Loop Counter Register

Shader versions 2.0 and higher provide looping flow control with the `loop` and `endloop` instructions. The loop counter register `aL` contains the current value of the counter that is incremented by these instructions. Its value is undefined outside the body of a loop. Within the body of a loop its value can be used as an offset into the constant array. In shader version 3.0 the loop counter register can be used to index the output registers as well as the constant registers.

Destination register write mask	*r*`.xyzw`
Source register multiplex	*r*`.[xyzw][xyzw][xyzw][xyzw]`
Source register negation	`-`*r*
Absolute value	*r*`_abs`
Logical negation	`!`*r*

Table 9.3: Vertex shader register modifiers. The `[]` notation indicates an optional vertex component specifier. See the text for details on their meaning.

9.2.6 The Predicate Register

Shader versions 2.x and higher provide a predicate register that contains a four dimensional vector of boolean values. The boolean values can be used to perform conditional flow control. The `setp_`*comp* instruction is the only instruction that can write to the predicate register. The boolean values in the predicate register are used to control the `if`, `callnz` and `breakp` instructions.

9.2.7 Sampler Registers

Shader version 3.0 provides access to textures with sampler registers. The sampler registers themselves are only used with the `texldl` instruction to sample a texture. The sampler registers must be declared with the `dcl_`*texture* instruction before they can be used. With sampler registers, vertex shaders can perform texture lookups for data driven displacement effects and efficient table lookup operations.

9.2.8 Register Modifiers

The source and destination operands to each instruction operate on the entire four dimensional vector value by default. To increase the flexibility of a vertex shader while keeping the instruction count low, each operand can contain modifiers that change the way the operand is used. For vertex shader instructions, there are four kinds of modifiers: a destination operand write mask modifier, a source operand multiplex, or "swizzle", modifier, a source operand negation modifier and an absolute value modifier. The syntax of the modifiers is summarized in table 9.3.

The write mask modifier restricts the output of the instruction to a subset of the four components in the destination register. The components to be written are appended to the register name with a separating dot, such as `.xy` to write only the first two components. Omitting the destination register write mask is equivalent to specifying *r*`.xyzw`. Only those components in the write mask will be written during processing of the instruction. In this way, any of the four components of a temporary register can be changed without changing the remaining components.

The multiplex modifier allows a four-dimensional vector to be constructed from any of the four components in a source register. The components are denoted by r.`xyzw`, respectively. Individual components may be replicated to multiple components in the resulting vector. You can think of this as if a multiplexor were used for each component in the resulting vector to select one of the four components from the source register.

For example, the source register specification `r0.yyxx` results in a four dimensional vector whose first two components are taken from the second component of register 0 and whose second two components are taken from the first component of register 0. In this way the components of source registers can be arbitrarily moved around, or "swizzled".

When fewer than four components are present in the modifier, the last component used is implicitly repeated for the remaining components. The source operand `r0.x` is equivalent to `r0.xxxx`, `r0.xy` is equivalent to `r0.xyyy`, and `r0.xyz` is equivalent to `r0.xyzz`. Specifying no source register multiplexing modifier is equivalent to specifying r.`xyzw`.

The negation modifier performs an arithmetic negation on the vector, changing the sign of all the components of the vector. The negation of a source register vector happens after any multiplexing has been applied. Shader version 3.0 provides the `abs` modifier. It returns the absolute value of the register.

The boolean constant registers and the predicate register contain boolean values. The logical negation operator, `!`, can be used with the boolean constant registers and the predicate register to change its logical value before it is used in a conditional test.

Multiple register modifiers can be combined in a single instruction. For instance, a source register can be swizzled and negated while the destination register is write masked.

9.3 Vertex Shader 1.1 Architecture

Vertex shader 1.1 architecture is the simplest architecture. It provides for no conditional branching and no flow control. A minimum of 96 vertex shader constant registers are required for the vertex shader version 1.1. An implementation may provide more registers, however, and a DirectX 8 or higher level driver will report the maximum number of constant registers available in the `MaxVertexShaderConst` member of `D3DCAPS9`. The constant registers can only be indexed by the x component of the address register.

Instructions are provided for declarations, basic arithmetic, matrix arithmetic, simple comparisons and basic lighting calculations. These basic instructions are generally used unmodified in subsequent shader architecture versions. The instructions are described in detail in section 9.10.

The higher versions of the shader architecture include the 1.1 version instructions and registers, with minor variations in some instructions. To fully understand the later version architectures, it is important to understand the instructions in version 1.1 of the architecture.

9.4 Vertex Shader 2.0 Architecture

The version 2.0 architecture incorporates all of the instructions and registers of version 1.1 and extends it with additional features. The instructions added by version 2.0 are discussed in detail in section 9.11. The main improvement with version 2.0 is the addition of static flow control to the execution model. Subroutines, branching and looping instructions are introduced with static conditions. In static flow control, all the conditional expressions for evaluating branch points refer to values that are constant for the duration of the shader. With static flow control, loops execute a fixed number of times and conditional execution always follows the same path for all primitives drawn with the same set of constants. Different batches of primitives can have different flow control behavior by changing the constants between batches. All flow control instructions are issued in pairs and surround a block of instructions.

New constant register files are provided for defining the constants used to govern the flow control. With static flow control, you can write a single shader that applies to different kinds of vertices and make choices on a primitive-by-primitive basis. The constants defining the flow control conditions can be updated between successive calls to draw primitives.

Versions 2.0 and higher enhance the use of the address register and provide new boolean and integer constant register files. All four components of the address register `a0` can be used to index the floating-point constant register files; the boolean and integer register files cannot be indexed. Any component of the address register can be used as an index, but all source operands in an instruction must use the same component and base register.

The address register can only be set with the new `mova` instruction. The `mov` instruction is still used to write values into the temporary and output registers. New arithmetic operations are provided with the `abs`, `crs`, `lrp`, `nrm`, `pow`, `sgn` and `sincos` instructions.

The boolean constant registers `b`*n* are used for conditional branching with the `if`, `else` and `endif` instructions. Each register has one component containing a boolean value. Values in the boolean register file can be defined with the `defb` instruction. Unconditional subroutine calls are made with the `call` instruction and the target of the subroutine call is a block between the `label` and `ret` instructions. Conditional subroutine calls with the boolean registers are made with the `callnz` instruction.

The integer constant register file `i`*n* has four components per register, but the fourth component must always be zero. The registers control the execution count of loops defined with the `rep`, `endrep`, `loop`, and `endloop`, instructions. The `rep` instruction defines a simple loop with a repeat count and no access to the internal count register during the loop.

The `loop` instruction defines a loop that exposes its internal counter through the `aL` loop counter register. The loop counter register holds a scalar value that is initialized before the loop starts and is incremented every time the loop is repeated. The loop counter register can be used to index the floating-point constant registers like the address register. The starting value, increment and number of loop repititions are encoded into the integer constant register file. Values in the integer register file can be defined in the shader with the `defi` instructions or through the API with the `SetVertexShaderConstantI` method on the device.

9.5 Vertex Shader 2.x Architecture

Version 2.x introduces optional extensions of the version 2.0 architecture, with support indicated in the `D3DCAPS9` for a device. Version 2.0 is extended by adding predication, deeper nesting of static flow control instructions and dynamic flow control instructions. The instructions added by version 2.x are discussed in detail in section 9.12.

The optional support is described by the `VS20Caps` member of `D3DCAPS9`, which is a `D3DVSHADERCAPS2_0` structure. The optional support in 2.x includes the predicate register, support for dynamic flow control, support for more than 12 temporary registers and support for deeper nesting of static flow control.

```
typedef struct _D3DVSHADERCAPS2_0
{
    DWORD Caps;
    INT DynamicFlowControlDepth;
    INT NumTemps;
    INT StaticFlowControlDepth;
} D3DVSHADERCAPS2_0;
```

If the `D3DVS20CAPS_PREDICATION` bit of the `Caps` member is set, then the device supports the predicate register `p0` and its associated instructions `setp_`*comp*, `if`, `callnz` and `breakp`. The predicate register is a four-dimensional vector of boolean values and can only be initialized by the `setp_`*comp* instruction.

```
#define D3DVS20CAPS_PREDICATION (1<<0)
```

Predication is a form of conditional execution that can be applied to individual instructions without branching. One of the components from the predicate register can be applied to the entire instruction, or individual components from the register can be applied to the four components of the result, acting as a dynamic per-channel write mask. The syntax for predication is shown in section 9.11.

The `NumTemps` gives the number of `r`*n* temporary registers and will be at least 12, as in version 2.0. The actual value will be within the range $[12, 32]$ as defined by the `D3DVS20_MIN_NUMTEMPS` and `D3DVS20_MAX_NUMTEMPS` constants.

```
#define D3DVS20_MIN_NUMTEMPS 12
#define D3DVS20_MAX_NUMTEMPS 32
```

Dynamic flow control is provided by the `if_`*comp* and `break_`*comp* instructions. If the `DynamicFlowControlDepth` member is non-zero, then dynamic flow control is supported. A more restricted form of dynamic flow control can also be performed with predication.

9.6 Vertex Shader 3.0 Architecture

Version 3.0 of the vertex shader architecture relaxes architectural limits, generalizes the input and output register files, adds the saturate instruction modifier and introduces texture sampling with new sampler registers and associated instructions. The number of temporary registers is increased to 32, as shown in table 9.2. The minimum number of available instruction slots is increased to 512, as shown in table 9.1. The new instructions, the saturate instruction modifier and details of the flow control limits and nesting are described in section 9.13.

The input and output registers are generalized so that they can be indexed similarly to the floating-point constant registers by any component of the address register or by the scalar loop count register. This allows a shader that traverses its input data in a loop to generate output data. The output registers are not given specific names as they are in previous shader versions; they are named `o`*n*, like the input registers. To associate an output semantic with a particular register, the `dcl_`*usage* instructions are used. This allows the runtime to map shader outputs to the corresponding fixed-function pipeline rasterization inputs or the appropriate pixel shader input semantic.

Texture sampling in a vertex shader allows for advanced sample based displacement mapping of base geometry. The `s`*n* sampler registers are associated with textures with the `dcl_`*texture* instruction. Once declared, the sampler registers can be used in the `texldl` instruction to obtain sampled values from the bound texture.

9.7 Shader Instruction Syntax

Internally Direct3D uses an array of `DWORD`s to encode a shader program. This encoding can be thought of as the “machine language” for a shader program. Because it is difficult to write programs by creating arrays of `DWORD`s directly, the SDK includes programs and functions for assembling and compiling shader programs from a text syntax into `DWORD`s. D3DX provides the functions for assembling shader text into a `DWORD` array as described in chapter 15. D3DX also provides functions for compiling high-level shader language text, as described in chapter 18. The SDK includes a command-line assemblers and compilers as described in appendix A. This section describes the text syntax for shader assembly programs, both vertex shaders and pixel shaders.

The syntax for shader instructions is like most CPU assembly languages, with the instruction **opcode** appearing first, followed by the **operands** for the instruction. Shader program text is first parsed into a stream of parsing tokens. White space and comments are ignored during parsing, but may serve as a separator between two parsing tokens. Unlike some assembly languages,

```
Opcode   Operands       Comments
mov      oPos, r1       ; write oPos
add      r0, r1, r2     // r0 = r1+r2
add      r0, r0, r0     /* r0 = 2*r0 */
```

Figure 9.2: Shader assembly language syntax illustrating opcodes, operands and supported comment styles.

it is not necessary for each instruction to appear on a line of text by itself; multiple instructions may appear on a single line of text input provided that the instructions are properly separated by white space or comments. Typical syntax is illustrated in figure 9.2.

Each shader instruction consists of an opcode and uses zero or more operands. Each opcode is represented by a case sensitive reserved word, such as `mov`. Once an opcode has been parsed, parsing continues by consuming the operands for the instruction from the shader input. Since each instruction uses a fixed number of operands, no terminating token is necessary to indicate the end of an instruction's operands. The destination operand of an instruction always appears first, followed by zero or more source operands depending on the instruction. The operands are separated by commas (`,`).

Operands generally consist of case sensitive reserved words indicating specific registers, such as `r0` or `oPos`, or a limited form of arithmetic expression. The arithmetic expressions allowed by an operand are specific to the shader model and the particular instruction. For instance, the source operand `r0.xyyz` uses the source operand multiplex modifier to rearrange the components of the register `r0` before it is used in the execution of the instruction.

In general, constant register source operands are referred to by the name of the register, such as `c0`. However, a vertex shader allows the constant register to be indexed as an array by the address register `a0`. In this case C-style array syntax is used to indicate the use of the index register and the base register offset, such as `c[16 + a0.x]`. A variant of this syntax is to index the base register directly as in `c16[a0.x]`. The constant register array syntax, i.e. `c[16]` instead of `c16` can also be used when the address register is not used.

Comments are supported with three different styles: assembly comment style, C comment style and C++ comment style. An assembly comment begins with a semi-colon (`;`) and ignores all characters from the semi-colon to the end of the input line. A C++ comment is similar and begins with two slashes (`//`) and also ignores the remainder of the input line. A C comment begins with a slash-asterisk character combination (`/*`) and causes all characters in the input stream to be ignored until the asterisk-slash character combination (`*/`) is parsed. This allows multiple lines of a shader, or only a portion of a line, to be commented out.

The D3DX functions that assemble shader source text have provisions for a limited form of preprocessing. If you wish a fully compliant C++ style preprocessor, you can invoke this as a separate program before passing your source code to the assembler. The shader assembler does not support any preprocessing on the shader file. To use the C preprocessor on shader source you must invoke the preprocessor on the source before using the assembler on it.

In our discussions, shader instructions are shown in their assembly syntax. The effect of each instruction is depicted by a formula in terms of the operands from the assembly syntax. Individual components of a vector operands are designated by a subscript, such as s_w indicating the w component of the operand s. Flow control instructions are shown manipulating named quantities like *pc* and *loop*. These names are only used to explain the behavior of the instructions and do not represent any particular implementation.

9.8 Execution Model

The execution model for a vertex shader is fairly simple. Each instruction is executed in the order it appears within the shader function `DWORD` array. Each instruction is executed by first performing any source register multiplexing modifiers and then performing any source register negation modifiers. This results in vector operands that are used as inputs to the instruction. The instruction is performed and the result is written to the destination with any write mask modifiers. This sequence continues until the last shader instruction has been executed.

Every vertex shader begins with a `vs` version instruction that defines the vertex shader architecture version used by the shader. For versions before 3.0, every vertex shader must store a value in the `oPos` output register. For version 3.0, the output register associated with the position semantic must be written. Except for these two requirements and architectural restrictions governing the number of registers and instructions available, a vertex shader can contain any valid instruction sequence.

Care must be taken in multipass algorithms that require vertices to line up identically on the screen. If one pass is executed using the fixed-function pipeline and another pass is executed using a vertex shader, the vertices may not line up identically since they travel through different code paths. One way around this is to eliminate the use of the vertex shader effects in this situation, falling back to a simpler fixed-function effect. Another alternative is to code the fixed-function passes as vertex shader passes that use vertex shader instructions to implement the fixed-function processing. In this case, care should be taken to process the vertex position component identically in all passes.

9.9 Software Vertex Processing

A device created with software or mixed vertex processing can execute vertex shader programs on the CPU. A device using software vertex processing always supports all versions of the vertex shader architecture. When a vertex shader is used with software vertex processing, the Direct3D runtime compiles the shader program into native CPU instructions. If the CPU supports the Intel SSE or the AMD 3DNow! instruction set extensions, then the runtime will use those instruction set extensions in the native code version of the vertex shader program.

This means that even on a card that does not support vertex shaders in hardware, they can still perform at an acceptable rate. Of course, the more complicated the vertex shader program, the more work will have to be done on the CPU, so while vertex shader execution in software is reasonable, it isn't as fast as executing vertex shaders in hardware.

9.10 Vertex Shader 1.1 Instructions

Vertex shader instructions are grouped into two categories: simple instructions and complex[1] instructions. All simple instructions execute in one slot, while complex instructions may execute in one or more slots. In text form, each instruction is given in the form of an operation code, or opcode, followed by a destination operand and finally source operands. Not every instruction takes the same number of operands. The vertex shader version 1.1 instruction set is summarized in table 9.4.

Before we take a look at the instructions in detail, let's take a look at a simple vertex shader. This shader simply moves the input vertex data to the appropriate vertex output registers. Like all vertex shaders, it begins with a vertex shader version instruction defining the vertex shader architecture version used by the shader and declaration instructions mapping vertex element semantics to the input registers used by the shader.

```
vs_1_1
dcl_position    v0
dcl_color0      v1
dcl_color1      v2
dcl_fog         v2.w
dcl_texcoord0   v3
dcl_texcoord1   v4
dcl_texcoord2   v5
dcl_texcoord3   v6
mov oPos, v0
mov oD0, v1
mov oD1, v2.xyz
mov oFog.x, v2.w
mov oT0, v3
mov oT1, v4
mov oT2, v5
```

[1]The SDK documentation refers to the complex instructions as "macros", but this is a misnomer. They are not macros in the C++ sense. To avoid confusion, we will refer to these instructions as complex instructions.

Instruction	Slots	Function
`add` d, s_0, s_1	1	add
`dcl`*_usage* d	—	declare input registers
`def` d, v_0, v_1, v_2, v_3	—	constant definition
`dp3` d, s_0, s_1	1	3D dot product
`dp4` d, s_0, s_1	1	4D dot product
`dst` d, s_0, s_1	1	distance
`exp` d, s	≤ 10	full-precision exponentiate
`expp` d, s	1	partial-precision exponentiate
`frc` d, s	≤ 3	fractional part
`lit` d, s	1	lighting
`log` d, s	≤ 10	full-precision logarithm
`logp` d, s	1	partial-precision logarithm
`m3x2` d, s_0, s_1	≤ 2	vector, 3×2 matrix product
`m3x3` d, s_0, s_1	≤ 3	vector, 3×3 matrix product
`m3x4` d, s_0, s_1	≤ 4	vector, 3×4 matrix product
`m4x3` d, s_0, s_1	≤ 3	vector, 4×3 matrix product
`m4x4` d, s_0, s_1	≤ 4	vector, 4×4 matrix product
`mad` d, s_0, s_1, s_2	1	multiply accumulate
`max` d, s_0, s_1	1	maximum
`min` d, s_0, s_1	1	minimum
`mov` d, s	1	copy
`mul` d, s_0, s_1	1	multiply
`nop`	1	no operation
`rcp` d, s	≥ 1	reciprocal
`rsq` d, s	≥ 1	reciprocal square root
`sge` d, s_0, s_1	1	$\geq$ compare
`slt` d, s_0, s_1	1	$<$ compare
`sub` d, s_0, s_1	1	subtract
`vs` *_major_minor*	—	shader version

Table 9.4: Vertex shader 1.1 instruction summary. The instructions are shown in assembly syntax. Simple instructions execute in a single slot, complex instructions execute in one or more slots, up to a maximum of the slot count shown.

```
mov oT3, v6
```

9.10.1 Declaration Instructions

Every vertex shader must declare the version of the architecture that will be used with the shader with the `vs` instruction. This instruction must be the first instruction in the shader. The *major* and *minor* operands give the major and minor version numbers, respectively, of the architecture used by the shader.

Vertex shader constant registers may also be bound at the time that a vertex shader is bound to the device with `SetVertexShader`. The `def` instruction defines the contents of a constant register with four scalar floating-point values v_0, v_1, v_2, and v_3.

`vs_`*major*_*minor*
`def` d, v_0, v_1, v_2, v_3
$d \leftarrow \langle v_0, v_1, v_2, v_3 \rangle$

The `def` instruction must appear after the version instruction and before any computation instructions. When D3DX parses a vertex shader definition, the `def` instruction results in a vertex shader declaration fragment which contains the constant definitions. In this sense, the `def` instruction is not a true instruction and merely a convenience for the programmer to define constants and code that uses those constants in the same file. Therefore, the `def` instruction does not count against the instruction count for a vertex shader definition.

To map the vertex input registers to the corresponding components in a vertex and the elements in its declaration, the `dcl_`*usage* instruction is used. The *usage* is one of the `D3DDECLUSAGE` enumerations followed by a usage index. If the usage index is omitted, then an index of zero is assumed. The vertex declaration instructions are listed below. Note that there is no instruction corresponding to `D3DDECLUSAGE_POSITIONT`, since that declaration instructs the API that the vertices are already in transformed homogeneous clip space and require no vertex processing.

`dcl_position`n	s
`dcl_blendweight`n	s
`dcl_blendindices`n	s
`dcl_normal`n	s
`dcl_psize`n	s
`dcl_texcoord`n	s
`dcl_tangent`n	s
`dcl_binormal`n	s
`dcl_tessfactor`n	s
`dcl_color`n	s
`dcl_fog`n	s
`dcl_depth`n	s
`dcl_sample`n	s

The declaration of input registers can omit vertex components not needed by the shader and no harm will result. Any of the available input registers can be declared in any order; there is no requirement that the registers used by the shader begin at any particular register or be contiguous in the register file. Of course, any undeclared vertex data will not be accessible by the shader.

9.10.2 Basic Arithmetic Instructions

The `mov` instruction is used to copy data from a source operand to a destination operand. In its simplest form, it simply copies the source vector to the destination. With the source negation, absolute value, source multiplex, and destination write mask operand modifiers the `mov` instruction can permute vector components and change the sign of its inputs.

Basic arithmetic is performed with the `add`, `sub`, `mul`, and `mad` instructions. Vector addition and subtraction are handled with the `add` and `sub` instructions.

`mov` d, s
    $d \leftarrow s$
`add` d, s_0, s_1
    $d \leftarrow \langle s_{0x} + s_{1x},\ s_{0y} + s_{1y},\ s_{0z} + s_{1z},\ s_{0w} + s_{1w} \rangle$
`sub` d, s_0, s_1
    $d \leftarrow \langle s_{0x} - s_{1x},\ s_{0y} - s_{1y},\ s_{0z} - s_{1z},\ s_{0w} - s_{1w} \rangle$

The `mul` instruction performs the component-wise multiplication of its two source operands. Note carefully that this is not the dot-product of two vectors, but simply the product of their respective components. The `mad` instruction performs a component-wise multiplication as in `mul` and then adds the components of the third operand.

`mul` d, s_0, s_1
    $d \leftarrow \langle s_{0x}s_{1x},\ s_{0y}s_{1y},\ s_{0z}s_{1z},\ s_{0w}s_{1w} \rangle$
`mad` d, s_0, s_1, s_2
    $d \leftarrow \langle s_{0x}s_{1x} + s_{2x},\ s_{0y}s_{1y} + s_{2y},\ s_{0z}s_{1z} + s_{2z},\ s_{0w}s_{1w} + s_{2w} \rangle$

A division instruction is not supplied directly, but the reciprocal instruction `rcp` computes the scalar reciprocal of the w component of the source operand. The other components of the source operand are ignored. The instruction guarantees that a reciprocal of one is exactly one and a reciprocal of zero is $+\infty$. Similarly, a reciprocal square-root can be computed with the `rsq` instruction. The `rsq` instruction uses the absolute value of the w component of the source operand to compute the result. Some hardware may stall if the result of the reciprocal instructions are used in the next instruction.[2]

[2]This is why the slot count in table 9.4 is listed as ≥ 1.

`rcp` d, s

$$d \leftarrow \begin{cases} \langle 1,\ 1,\ 1,\ 1\rangle, & s_w = 1 \\ \langle +\infty,\ +\infty,\ +\infty,\ +\infty\rangle, & s_w = 0 \\ \langle f,\ f,\ f,\ f\rangle, \quad f = \frac{1}{s_w}, & \text{otherwise} \end{cases}$$

`rsq` d, s

$$d \leftarrow \begin{cases} \langle 1,\ 1,\ 1,\ 1\rangle, & |s_w| = 1 \\ \langle +\infty,\ +\infty,\ +\infty,\ +\infty\rangle, & |s_w| = 0 \\ \langle f,\ f,\ f,\ f\rangle, \quad f = \frac{1}{\sqrt{|s_w|}}, & \text{otherwise} \end{cases}$$

The `dp3` and `dp4` instructions perform dot products between two vectors. The `dp3` instruction computes a dot product of the first three components of the two source vectors, while the `dp4` instruction computes the dot product of all four components. In each case, the scalar result is replicated to all four components of the destination.

`dp3` d, s_0, s_1

$$d \leftarrow \langle f,\ f,\ f,\ f\rangle, \quad f = s_{0x}s_{1x} + s_{0y}s_{1y} + s_{0z}s_{1z}$$

`dp4` d, s_0, s_1

$$d \leftarrow \langle f,\ f,\ f,\ f\rangle, \quad f = s_{0x}s_{1x} + s_{0y}s_{1y} + s_{0z}s_{1z} + s_{0w}s_{1w}$$

Component-wise minima and maxima of two vectors can be computed with the `min` and `max` instructions.

`min` d, s_0, s_1

$$d \leftarrow \langle\, \min(s_{0x}, s_{1x}),\ \min(s_{0y}, s_{1y}),\ \min(s_{0z}, s_{1z}),\ \min(s_{0w}, s_{1w})\rangle$$

`max` d, s_0, s_1

$$d \leftarrow \langle\, \max(s_{0x}, s_{1x}),\ \max(s_{0y}, s_{1y}),\ \max(s_{0z}, s_{1z}),\ \max(s_{0w}, s_{1w})\rangle$$

Base two logarithms and exponentials can be computed with the `log`, `logp`, `exp`, and `expp` instructions. The simple instructions `logp` and `expp` compute a 10-bit precision result in a single slot. The complex instructions `log` and `exp` compute an full precision result in at most ten slots.

`exp` d, s

$$d \leftarrow \langle f,\ f,\ f,\ f\rangle, \quad f = 2^{s_w}$$

`expp` d, s

$$d \leftarrow \langle 2^{\lfloor s_w \rfloor},\ s_w - \lfloor s_w \rfloor,\ 2^{s_w},\ 1\rangle$$

`log` d, s

$$d \leftarrow \begin{cases} \langle -\infty,\ -\infty,\ -\infty,\ -\infty\rangle, & |s_w| = 0 \\ \langle f,\ f,\ f,\ f\rangle, \quad f = \log_2(|s_w|), & \text{otherwise} \end{cases}$$

`logp` d, s

$$d \leftarrow \begin{cases} \langle -\infty,\ 1,\ -\infty,\ 1\rangle, & |s_w| = 0 \\ \langle f,\ |s_w| - 2^f,\ \log_2(|s_w|),\ 1\rangle, \quad f = \lfloor \log_2(|s_w|)\rfloor, & \text{otherwise} \end{cases}$$

9.10.3 Matrix Instructions

The `m3x2`, `m3x3`, `m3x4`, `m4x3` and `m4x4` instructions are complex instructions that implement the product of a vector by a matrix. The first source operand gives the vector and the second source operand gives the matrix. The matrix is assumed to be stored in consecutively numbered registers beginning with s_1 and in the same register file. In the following formulas, we use the notation s', s'', and s''' to denote the consecutive rows of the matrix stored in the register file. For instance, if s_1 is the register `c[5]`, then s_1' is the register `c[6]`, s_1'' is `c[7]` and s_1''' is the register `c[8]`. Typically the matrix will be stored in the constant register file.

`m3x2` d, s_0, s_1

$$d_{xy} \leftarrow \begin{bmatrix} s_{0x} \\ s_{0y} \\ s_{0z} \end{bmatrix} \begin{bmatrix} s_{1x} & s_{1y} & s_{1z} \\ s'_{1x} & s'_{1y} & s'_{1z} \end{bmatrix}$$

`m3x3` d, s_0, s_1

$$d_{xyz} \leftarrow \begin{bmatrix} s_{0x} \\ s_{0y} \\ s_{0z} \end{bmatrix} \begin{bmatrix} s_{1x} & s_{1y} & s_{1z} \\ s'_{1x} & s'_{1y} & s'_{1z} \\ s''_{1x} & s''_{1y} & s''_{1z} \end{bmatrix}$$

`m3x4` d, s_0, s_1

$$d \leftarrow \begin{bmatrix} s_{0x} \\ s_{0y} \\ s_{0z} \end{bmatrix} \begin{bmatrix} s_{1x} & s_{1y} & s_{1z} \\ s'_{1x} & s'_{1y} & s'_{1z} \\ s''_{1x} & s''_{1y} & s''_{1z} \\ s'''_{1x} & s'''_{1y} & s'''_{1z} \end{bmatrix}$$

`m4x3` d, s_0, s_1

$$d_{xyz} \leftarrow \begin{bmatrix} s_{0x} \\ s_{0y} \\ s_{0z} \\ s_{0w} \end{bmatrix} \begin{bmatrix} s_{1x} & s_{1y} & s_{1z} & s_{1w} \\ s'_{1x} & s'_{1y} & s'_{1z} & s'_{1w} \\ s''_{1x} & s''_{1y} & s''_{1z} & s''_{1w} \end{bmatrix}$$

`m4x4` d, s_0, s_1

$$d \leftarrow \begin{bmatrix} s_{0x} \\ s_{0y} \\ s_{0z} \\ s_{0w} \end{bmatrix} \begin{bmatrix} s_{1x} & s_{1y} & s_{1z} & s_{1w} \\ s'_{1x} & s'_{1y} & s'_{1z} & s'_{1w} \\ s''_{1x} & s''_{1y} & s''_{1z} & s''_{1w} \\ s'''_{1x} & s'''_{1y} & s'''_{1z} & s'''_{1w} \end{bmatrix}$$

Only the `m4x4` and `m3x4` variants of these instructions modify all four components of the destination register. The `m3x2` instruction modifies only the x and y components of the destination, leaving the z and w components unchanged. Similarly, the `m3x3` and `m4x3` instructions modify only the x, y, and z components of the destination register, leaving the w component unchanged.

The source operand multiplex and negation modifiers are not allowed with these instructions. These complex instructions can be implemented using simple instructions, but it is recommended that applications use the complex instructions to more clearly indicate their intentions to the device. These complex instructions correspond to the following simple instruction sequences.

```
; m3x2 r1, r0, c0
dp3 r1.x, r0, c0
dp3 r1.y, r0, c1

; m3x3 r1, r0, c0
dp3 r1.x, r0, c0
dp3 r1.y, r0, c1
dp3 r1.z, r0, c2

; m3x4 r1, r0, c0
dp3 r1.x, r0, c0
dp3 r1.y, r0, c1
dp3 r1.z, r0, c2
dp3 r1.w, r0, c3
```

```
; m4x3 r1, r0, c0
dp4 r1.x, r0, c0
dp4 r1.y, r0, c1
dp4 r1.z, r0, c2

; m4x4 r1, r0, c0
dp4 r1.x, r0, c0
dp4 r1.y, r0, c1
dp4 r1.z, r0, c2
dp4 r1.w, r0, c3
```

9.10.4 Comparison Instructions

Although no branching is allowed within a version 1.1 vertex shader, it is possible to perform a limited form of comparison. Suppose you need to sometimes add a secondary color C_s to the diffuse color C_d of a vertex. Since you can't perform a branch to add C_s, you might think that you need to create two distinct shaders: one that adds C_s and one that doesn't. However, you may be able to perform everything in a single shader if you can create a color S that is zero when you don't want the secondary color added and C_s when the secondary color is to be added. Then you can compute $C_d + S$ as the diffuse color of the vertex and everything will be fine.

The `sge` and `slt` comparison instructions allow you to perform this trick. They perform a comparison between two source operands and store a value in each component for the result of the comparison. The component will be 1.0 when the comparison is true and 0.0 when the comparison is false. The `sge` instruction tests if the first operand is greather than or equal to the second operand and the `slt` instruction tests if the first operand is less than the second operand. These two comparisons are logical opposites.

`sge` d, s_0, s_1
$$d \leftarrow \langle s_{0x} \geq s_{1x},\ s_{0y} \geq s_{1y},\ s_{0z} \geq s_{1z},\ s_{0w} \geq s_{1w} \rangle$$
`slt` d, s_0, s_1
$$d \leftarrow \langle s_{0x} < s_{1x},\ s_{0y} < s_{1y},\ s_{0z} < s_{1z},\ s_{0w} < s_{1w} \rangle$$

If you need to compute $x > y$, remember that this can be rewritten as $y < x$ which is directly available via the `slt` instruction. Similarly, $x \leq y$ can be rewritten as $y \geq x$. At times it can also be useful to note that if $x < y$ this implies $-x > -y$ and if $x \geq y$ this implies $-x \leq -y$. The source operand negation modifier can be used directly to implement these comparisons. For instance, the following two instructions are identical in their result.

```
slt r0, r1, r2      ; r0 = (r1 < r2)
slt r0, -r2, -r1    ; r0 = (-r2 < -r1) = (-r1 > -r2) = (r1 < r2)
```

A logical "and" of two comparisons can be performed by computing the product of the results from two comparisons or the minimum of the two results. A logical "or" of two comparisons can be performed by computing the maximum of the two results. To invert the result of a comparison test, subtract the result of the test from the constant 1. To perform other comparison tests, rearrange the desired comparison in terms of the two available comparisons. For example, a test for equality between `r0` and `r1` can be performed by the following instruction sequence:

```
sge r2, r0, r1      ; r2 = (r0 >= r1)
sge r3, r1, r0      ; r3 = (r1 >= r0)
min r2, r2, r3      ; r2 = (r2 && r3) = (r0 == r1)
```

After the first instruction, `r2` will contain a 1 for each component of `r0` that is greater than or equal to the corresponding component of `r1`. After the second instruction, `r3` will contain a 1 for each component of `r1` that is greater than or equal to the component of `r0`. With the third instruction, we combine the results of both tests into `r2`. The conditions $x \geq y$ and $y \geq x$ can both be true only when $x = y$.

As mentioned in chapter 1, floating-point representations are approximations and care should be taken when comparing between two such values. An epsilon equality comparison can be performed by subtracting the two values and comparing the result to an accuracy criteria, ϵ. Other epsilon-based comparison tests are similar.

```
; c0.x = epsilon
def c0, 1e-6, 0, 0, 0

; perform epsilon comparison of r0 and r1
sub r2, r0, r1      ; r2 = r0 - r1
max r2, r2, -r2     ; r2 = abs(r2)
slt r2, r2, c0.x    ; r2 = (r2 < epsilon)
```

9.10.5 Lighting Instructions

The `dst` and `lit` instructions aid in the computation of lighting effects. The `dst` instruction computes a distance vector given $s_0 = \langle *, k^2, k^2, * \rangle$, and $s_1 = \langle *, \frac{1}{k}, *, \frac{1}{k} \rangle$, where the component values shown as $*$ are ignored.

The `lit` instruction computes lighting coefficients given two dot products and an exponent. The x component of the source register contains the dot product of the vertex normal and the light vector, $\vec{n} \cdot \vec{l}$, the y component contains the dot product of the vertex normal and the halfway vector $\vec{n} \cdot \vec{h}$, and the w component contains the exponent. The exponent must be in the interval $[-128, 128]$.

`dst` $d,\ s_0,\ s_1$

$$d \leftarrow \langle 1,\ k,\ k^2,\ \tfrac{1}{k}\rangle,\ s_0 = \langle *,\ k^2,\ k^2,\ *\rangle,\ s_1 = \langle *,\ \tfrac{1}{k},\ *,\ \tfrac{1}{k}\rangle$$

`lit` $d,\ s$

$$d \leftarrow \begin{cases} \langle 1,\ s_x,\ {s_y}^{s_w},\ 1\rangle, & s_x > 0 \text{ and } s_y > 0 \\ \langle 1,\ s_x,\ 0,\ 1\rangle, & s_x > 0 \text{ and } s_y \le 0 \\ \langle 1,\ 0,\ 0,\ 1\rangle, & \text{otherwise} \end{cases}$$

9.11 Vertex Shader 2.0 Instructions

Similar to the `def` instruction in version 1.1, version 2.0 provides the `defb` and `defi` instructions to define values for the boolean and integer constant registers. The `mova` instruction is used to write the four components of th eaddress register. The `mov` instruction cannot be used to write the address register in version 2.0.

`defb` $d,\ v$

$$d \leftarrow v$$

`defi` $d,\ i_0,\ i_1,\ i_2,\ i_3$

$$d \leftarrow \langle i_0,\ i_1,\ i_2,\ i_3\rangle$$

`mova` $d,\ s$

$$d \leftarrow \langle \mathrm{round}(s_x),\ \mathrm{round}(s_y),\ \mathrm{round}(s_z),\ \mathrm{round}(s_w)\rangle$$

New mathematical operations are introduced for absolute value, sign of operand, vector cross product, linear interpolation, vector normalization and the trigonometric functions cosine and sine. The `abs` instruction writes the absolute value of its source operand to its destination operand. The `sgn` instructions writes the sign function of its source operand to its destination.

`abs` $d,\ s$

$$d \leftarrow \langle |s_x|,\ |s_y|,\ |s_z|,\ |s_w|\rangle$$

`sgn` $d,\ s_0,\ s_1,\ s_2$

$$d \leftarrow \langle f(s_{0x}),\ f(s_{0y}),\ f(s_{0z}),\ f(s_{0w})\rangle,$$

$$f(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

The `crs` instruction writes the vector cross product of its source operands to its destination. The cross product is often computed in lighting computations and to determine a vector that is normal to two other vectors. The `lrp` instruction performs linear interpolation between

`crs` $d,\ s_0,\ s_1$

$$d \leftarrow \vec{s_0} \otimes \vec{s_1}$$

`lrp` $d,\ s_0,\ s_1,\ s_2$

$$d \leftarrow \langle f(s_{0x}, s_{1x}, s_{2x}),\ f(s_{0y}, s_{1y}, s_{2y}),\ f(s_{0z}, s_{1z}, s_{2z}),\ f(s_{0w}, s_{1w}, s_{2w})\rangle,$$

$$f(s_0, s_1, s_2) = s_0 s_1 + (1 - s_0) s_2$$

`nrm` $d,\ s$

$$d \leftarrow \frac{\vec{s}}{\|\vec{s}\|}$$

Instruction	Slots	Function
`abs` d, s	1	absolute value
`call` l	2	call a subroutine
`callnz` l, b	3	conditionally call a subroutine
`crs` d, s_0, s_1	2	vector cross product
`defb` d, v_0	—	boolean constant definition
`defi` d, v_0, v_1, v_2, v_3	—	integer constant definition
`else`	1	start an else block
`endif`	1	end an if or else block
`endloop`	2	end a loop block
`endrep`	2	end a repeat block
`if` b	3	start an if block
`label` l	—	start a subroutine block
`loop` `aL`, i	3	start a loop block
`lrp` d, s_0, s_1, s_2	2	linear interpolation
`mova` d, s	1	write address register
`nrm` d, s	3	vector normalization
`pow` d, s_0, s_1	3	full precision x^y
`rep` i	3	start a repeat block
`ret`	1	end a subroutine block
`sgn` d, s	3	sign function
`sincos` d, s_0, s_1, s_2	8	sine and cosine

Table 9.5: Vertex shader 2.0 instruction summary. The instructions are shown in assembly syntax. Simple instructions execute in a single slot, complex instructions execute in one or more slot, up to a maximum of the slot count shown.

Write Mask	Resulting Vector
`.x`	$\langle\cos(s_{0c}), ?, ?, *\rangle$
`.y`	$\langle ?, \sin(s_{0c}), ?, *\rangle$
`.xy`	$\langle\cos(s_{0c}), \sin(s_{0c}), ?, *\rangle$

Table 9.6: `sincos` results based on write mask. ? indicates a component that has an undefined value after the instruction executes. $*$ indicates a component that is not changed by the instruction.

The `pow` instruction computes a full-precision power function. The two source arguments must use one of the replicate register modifiers `.x`, `.y`, `.z` or `.w` to select the scalar components of the source operands to the function. The resulting vector replicates the scalar result to all four components and has at least 15 bits of precision. The destination register should be a different register than s_1.

```
pow $d$, $s_0$, $s_1$
    $d \leftarrow |s_0|^{s_1}$
```

The `sincos` instruction computes the sine and cosine of the angle in source operand s_0. The source angle must use one of the replicate register modifiers `.x`, `.y`, `.z` and `.w` and must be in the interval $[-\pi, \pi]$. The source operands s_1 and s_2 are floating-point constant registers that contain the constants `D3DSINCOS-CONST1` and `D3DSINCOSCONST2`, respectively. The constants can be defined with a `def` instruction or through the API with the `SetVertexShaderConstantF` method.

```
sincos $d$, $s_0$, $s_1$, $s_2$
    $d \leftarrow \langle\cos(s_{0c}), \sin(s_{0c}), ?, ?\rangle$
```

```
#define D3DSINCOSCONST1 \
    -1.5500992e-6f, -2.1701389e-5f, 0.0026041667f, 0.00026041668f
#define D3DSINCOSCONST2 \
    -0.020833334f,  -0.12500000f,   1.0f,          0.5f
```

The destination operand to `sincos` must be a temporary register and must use one of the write masks `.x`, `.y` or `.xy`. The components of the destination operand vary slightly, depending on the write mask used. The combinations are summarized in table 9.6.

The simplest form of looping is a repeat block. The block of instructions to be repeated is enclosed in a `rep` and `endrep` instruction pair. In the pseudocode for these instructions the value *pc* refers to the value of the program counter for the instruction. The expression $pc + 1$ referring to the address of the next instruction. The `rep` instruction uses an integer constant register as its operand. The operand's x component contains the number of times the block will be repeated and is in the range $[0, 255]$. A repeat count of zero will not execute the block. The counter used for the repeat block is not available for use within the block of instructions, so its only use is to perform a sequence of operations a certain number of times.

```
rep     $i$
        $count \leftarrow i_x$
        $loop \leftarrow pc + 1$
        if $count = 0$ then $pc \leftarrow endloop$
endrep
        $endloop \leftarrow pc + 1$
        $count \leftarrow count - 1$
        if $count > 0$ then $pc \leftarrow loop$
```

A loop block repeatedly executes the block between the `loop` and `endloop` instructions. A loop block provides access to the counter used to control the loop through the `aL` register. This register is always given as the destination operand to the `loop` instruction. The source operand is the integer register that contains the constants defining the loop execution.

```
loop     aL, $i$
         aL $\leftarrow i_y$
         $count \leftarrow i_x$
         $loop \leftarrow pc + 1$
         if $count = 0$ then $pc \leftarrow endloop$
endloop
         $endloop \leftarrow pc + 1$
         $count \leftarrow count - 1$
         aL $\leftarrow$ aL $+ i_z$
         if $count > 0$ then $pc \leftarrow loop$
```

A conditional block evaluates a boolean register and executes the instructions in the block if the value is true. An optional block can be included to execute instructions if the value is false. The `if` and `endif` instructions surround the block to be executed when the value is true. To provide alternate blocks to be executed when the value is true or false, use `if`, `else` and `endif` to surround the true and false blocks. The logical negation operator, `!`, can be used on the source operand to invert the sense of the test.

```
if      $b$
        if $b$ = false then $pc \leftarrow else$
else
        $else \leftarrow pc + 1$
        if $b$ = true then $pc \leftarrow endif$
endif
        $endif \leftarrow pc + 1$
```

A subroutine block is surrounded by a `label` and `ret` instruction pair. In order to have a subroutine, you must first end the main routine at the start of the shader with a `ret` instruction. If your main shader routine has no subroutines, then the `ret` instruction at the end of the main routine is optional. The `call` instruction invokes a subroutine and its target must be a forward reference.

```
label $l$
    $l \leftarrow pc + 1$
ret
    $pc \leftarrow \text{pop}(pc)$
call  $l$
    $\text{push}(pc, pc + 1)$
    $pc \leftarrow l$
```

Conditional subroutine calls can be made with the `callnz` instruction. The subroutine call to the label is made if boolean value is true. The logical negation operator, `!`, can be used on the boolean operand to logically negate the value before it is used.

```
callnz $l, b$
    if $b$ = true then
        $\text{push}(pc, pc + 1)$
        $pc \leftarrow l$
    endif
```

9.11.1 Version 2.0 Flow Control Nesting Limits

The limited resources of a GPU imposes limits on the amount of flow control that can be used within a shader. Each type of flow control instruction (looping, branching, and subroutines) has its own set of nesting limits. When one instruction block is enclosed in another block, the enclosed block is referred to as a nested block. The nesting limits and static flow count for the different vertex shader versions are listed in table 9.7.

In addition to the nesting limits, there is also a limit on the total number of control flow instructions that can appear in the shader, regardless of nesting level. The total number of control flow instructions is called the static flow count. Generally, each control flow instruction that introduces a decision point adds one to the static flow count for the shader. For instance, the `if` and `else` instructions add to the flow count, but the `endif` instruction does not because `if` and `else` involve a potential branch in control flow, but the `endif` only marks the target for such a branch. The `if`, `else`, `rep`, `loop`, `call` and `callnz` instructions count towards the static flow count for a 1.1 shader.

The static conditional instructions in version 2.0 have a nesting limit of one. This means that an `if` block can appear in a 2.0 shader, but only when it is located at the top-level of a routine. While you cannot nest one `if` block inside another, you can still utilize the comparison instructions `sge` and `slt` and the conditional evaluation tricks described in section 9.17.

Similarly, the `call` and `callnz` instructions have a nesting limit of one. This means that you can call a subroutine, but that subroutine cannot call another subroutine. You can have multiple subroutines as long as they are called from

	Shader Version		
Feature	2.0	2.x	3.0
Call Nesting	1	1-4	4
Static Conditions	16	16	24
Dynamic Conditions	—	0-24	24
Loop Nesting	1	1-4	4
Static Flow Count	16	16	∞

Table 9.7: Vertex shader version nesting limits.

Instruction		Slots	Function
`break`		1	break out of a loop
`break_`*comp*	s_0, s_1	3	conditionally break out of a loop
`break`	p	3	conditionally break out of a loop
`callnz`	l, p	3	conditionally call a subroutine
`if_`*comp*	s_0, s_1	3	start a dynamic if block
`if`	p	3	start a dynamic if block
`setp_`*comp*	d, s_0, s_1	1	set predicate register

Table 9.8: Vertex shader 2.x instruction summary. The instructions are shown in assembly syntax. Simple instructions execute in a single slot, complex instructions execute in one or more slot, up to a maximum of the slot count shown.

the main routine. The subroutine call nesting limit is separate from the static condition nesting limit, so a subroutine can contain an `if` instruction as long as it is not called from inside an `if` block in the main routine.

The `loop` and `rep` looping instructions have a nesting limit of one. Both forms of loop count towards the limit, so while a `rep` can be nested inside an `if`, it cannot be nested inside a `loop` and vice-versa. This nesting limit is separate from the subroutine call and static condition nesting limits.

9.12 Vertex Shader 2.x Instructions

Version 2.x of the shader architecture extends version 2.0 with instructions for predication and dynamic flow control. The new instructions are summarized in table 9.8. Dynamic control flow can be provided either through the use of predication on instructions or through dynamic conditional expressions. Predication of instructions is exposed through an instruction modifier syntax on each instruction. The instructions that support the predicate instruction modifier are listed in table 9.9.

An instruction is marked for predication by placing the predicate register in parenthesis before the instruction. The boolean negation operator can be used on the predicate register to invert the value from the predicate register before using it. The replicate source register modifiers `.x`, `.y`, `.z` and `.w` can be

`abs`	`dst`	`log`	`mad`	`nrm`	`slt`
`add`	`exp`	`logp`	`max`	`pow`	`sgn`
`crs`	`expp`	`lrp`	`min`	`rcp`	`sincos`
`dp3`	`frc`	`m3x4`	`mov`	`rsq`	`sub`
`dp4`	`lit`	`m4x4`	`mul`	`sge`	

Table 9.9: Vertex shader instructions supporting predication

used on the predicate register to select a single boolean value that operates on all channels of the destination register. If no multiplex modifier or the `.xyzw` modifier are used on the predicate register then each of its four boolean values applies individually to the four channels of the destination register.

The instructions listed in table 9.9 can be marked with the predication instruction modifier. Here are some examples of the predicate instruction modifier.

```
// p0 = <true, false, false, true>
(p0.x)  add     r3, r1, r2      // r3 = r1 + r2
(p0)    add     r4, r1, r2      // r4.x = r1.x + r2.x,
                                // r4.w = r1.w + r2.w
(!p0.x) add     r5, r1, r2      // r5 unchanged
```

The `setp_`*comp* instructions are the only instructions that can write to the predicate register. Each instruction applies the appropriate test and writes the components of the destination register specified by its write mask. If no write mask is specified, then all four channels of the predicate register are written.

`setp_eq` d, s_0, s_1
$$d \leftarrow \langle s_{0x} = s_{1x},\ s_{0y} = s_{1y},\ s_{0z} = s_{1z},\ s_{0w} = s_{1w} \rangle$$
`setp_ne` d, s_0, s_1
$$d \leftarrow \langle s_{0x} \neq s_{1x},\ s_{0y} \neq s_{1y},\ s_{0z} \neq s_{1z},\ s_{0w} \neq s_{1w} \rangle$$
`setp_ge` d, s_0, s_1
$$d \leftarrow \langle s_{0x} \geq s_{1x},\ s_{0y} \geq s_{1y},\ s_{0z} \geq s_{1z},\ s_{0w} \geq s_{1w} \rangle$$
`setp_gt` d, s_0, s_1
$$d \leftarrow \langle s_{0x} > s_{1x},\ s_{0y} > s_{1y},\ s_{0z} > s_{1z},\ s_{0w} > s_{1w} \rangle$$
`setp_le` d, s_0, s_1
$$d \leftarrow \langle s_{0x} \leq s_{1x},\ s_{0y} \leq s_{1y},\ s_{0z} \leq s_{1z},\ s_{0w} \leq s_{1w} \rangle$$
`setp_lt` d, s_0, s_1
$$d \leftarrow \langle s_{0x} < s_{1x},\ s_{0y} < s_{1y},\ s_{0z} < s_{1z},\ s_{0w} < s_{1w} \rangle$$

The `if` instruction can be used with a component of the predicate register for dynamic conditional execution. The source replicate modifier must be used on the predicate register to select one of the four boolean values for the conditional expression.

```
if     $p.c$
       if $p_c$ = false then $pc \leftarrow else$
else
       $else \leftarrow pc + 1$
       if $p_c$ = true then $pc \leftarrow endif$
endif
       $endif \leftarrow pc + 1$
```

The `break`, `breakp` and `break_`*comp* instructions allow static and dynamic exits from inside the body of a loop to resume execution after the loop body. The break instructions work with both `rep` and `loop` blocks. The `break` instruction terminates the loop unconditionally and can be used even when predication and dynamic control flow is not provided.

```
break
       $pc \leftarrow endloop$
breakp $p.c$
       if $p_c$ = true then $pc \leftarrow endloop$
```

The `breakp` instruction provides dynamic conditional termination of a loop block based on the value in a component of the predicate register. The `break_`*comp* instructions dynamically terminate a loop based on a conditional expression computed from two scalar values. The source registers in the `break_`*comp* instructions must be specified with a replicate register modifier to specify the scalar value to be used in the comparison.

```
break_eq $s_0.c$, $s_1.c$
         if $s_{0c} = s_{1c}$ then $pc \leftarrow endloop$
break_ne $s_0.c$, $s_1.c$
         if $s_{0c} \neq s_{1c}$ then $pc \leftarrow endloop$
break_ge $s_0.c$, $s_1.c$
         if $s_{0c} \geq s_{1c}$ then $pc \leftarrow endloop$
break_gt $s_0.c$, $s_1.c$
         if $s_{0c} > s_{1c}$ then $pc \leftarrow endloop$
break_le $s_0.c$, $s_1.c$
         if $s_{0c} \leq s_{1c}$ then $pc \leftarrow endloop$
break_lt $s_0.c$, $s_1.c$
         if $s_{0c} < s_{1c}$ then $pc \leftarrow endloop$
```

Although the predication instruction modifier cannot be used with the `call` instruction, you could surround the subroutine call with an `if` instruction and the predicate register to conditionally call a subroutine. The `callnz` instruction conditionally calls a subroutine in a single instruction. The first source operand identifies the subroutine label to be called and the second source operand gives the component of the predicate register to be used for the condition.

```
callnz $l$, $p.c$
    if $p_c = $ true then
        push($pc, pc + 1$)
        $pc \leftarrow l$
    endif
```

Dynamic conditional execution can be performed without predication by using the `if_`*comp* instructions. The two source operands must both use the replicate register modifier to specify the two scalar values to be used in the comparison.

```
if_eq $s_0.c$, $s_1.c$
    if $s_{0c} = s_{1c}$ then ...
if_ne $s_0.c$, $s_1.c$
    if $s_{0c} \neq s_{1c}$ then ...
if_ge $s_0.c$, $s_1.c$
    if $s_{0c} \geq s_{1c}$ then ...
if_gt $s_0.c$, $s_1.c$
    if $s_{0c} > s_{1c}$ then ...
if_le $s_0.c$, $s_1.c$
    if $s_{0c} \leq s_{1c}$ then ...
if_lt $s_0.c$, $s_1.c$
    if $s_{0c} < s_{1c}$ then ...
```

9.12.1 Version 2.x Flow Control Nesting Limits

The general mechanism of flow control limits described in subsection 9.11.1 applies to version 2.x as well. Version 2.x has at least the same limits as version 2.0, but some of the limits may be increased as summarized in table 9.7. The static flow count described in subsection 9.11.1 applies to version 2.x as well.

The `VS20Caps` member of `D3DCAPS9` is a `D3DVSHADERCAPS2_0` structure that gives the exact value for the static call nesting depth, the dynamic condition nesting depth and the loop nesting depth. The `StaticFlowControlDepth` and `DynamicFlowControlDepth` members give the amount of nesting allowed for static and dynamic flow control, respectively. The static flow control depth is always at least one, as in version 2.0. The static flow control depth applies also to the maximum loop nesting depth; the two values are always the same but each control structure maintains its own nesting depth.

```
#define D3DVS20_MIN_STATICFLOWCONTROLDEPTH   1
#define D3DVS20_MAX_STATICFLOWCONTROLDEPTH   4
#define D3DVS20_MIN_DYNAMICFLOWCONTROLDEPTH  0
#define D3DVS20_MAX_DYNAMICFLOWCONTROLDEPTH  24
```

Instruction	Slots	Function
`dcl_position`n d	—	declare a position output
`dcl_blendweight`n d	—	declare a blend weight output
`dcl_blendindices`n d	—	declare a blend indices output
`dcl_psize`n d	—	declare a point size output
`dcl_fog`n d	—	declare a fog factor output
`dcl_normal`n d	—	declare a normal vector output
`dcl_texcoord`n d	—	declare a texture coordinate output
`dcl_tangent`n d	—	declare a tangent vector output
`dcl_binormal`n d	—	declare a binormal vector output
`dcl_tessfactor`n d	—	declare a tessellation factor output
`dcl_depth`n d	—	declare a depth output
`dcl_2d` s	—	declare a 2D texture sampler
`dcl_cube` s	—	declare a cube texture sampler
`dcl_volume` s	—	declare a volume texture sampler
`texldl` s	2 or 5	sample texture

Table 9.10: Vertex shader 3.0 instruction summary. The instructions are shown in assembly syntax.

9.13 Vertex Shader 3.0 Instructions

In version 3.0 of the shader architecture, all output registers must be declared. The declaration syntax is similar to that for declaring input registers and consists of a declaration instruction that associates a semantic usage name and index with the register. The output register semantics for the vertex shader are matched with corresponding input register semantics on the pixel shader for use in rendering.

The address register can be used to index the input and output registers in addition to the constant registers.

Version 3.0 of the vertex shader architecture introduces the ability to sample a texture during vertex processing. The topology of the source texture is declared with one of the `dcl_2d`, `dcl_cube` or `dcl_volume` instructions. Each of these instructions takes a single operand that associates a sampling register `s`n with a texture of the corresponding topology.

```
dcl_2d     s
dcl_cube   s
dcl_volume s
```

Once a sampling register has been declared, the `texldl` instruction is used to sample the texture into a temporary register. The semantics of this instruction are fairly complicated, so first we'll go over the operands and then we'll explain the execution behavior. The destination operand d will receive the filtered texture sample. The destination register d must be a temporary register. A write mask can be used on the destination register to control which texture values are written. The s_0 source operand provides the texture coordinates at which the texture will be sampled. The s_1 source operand is the sampler register indicating which texture is to be sampled. A source replicate modifier can be used on the sampler register, which will permute the sample values before the write mask on the destination register is applied.

```
texldl $d$, $s_0$, $s_1$
    $L \leftarrow s_{0w} + \mathit{SSLODBias}$
    if $(L \leq 0)$ then
        $L \leftarrow \max(\mathit{SSMaxMipLevel}, 0)$
        $\mathit{filter} \leftarrow \mathit{SSMagFilter}$
        $q \leftarrow \text{lookup}(s_0, s_1, L, \mathit{filter})$
    else
        $L \leftarrow \max(\mathit{SSMaxMipLevel}, L)$
        $\mathit{filter} \leftarrow \mathit{SSMinFilter}$
        $q \leftarrow \text{lookup}(s_0, s_1, \lfloor L \rfloor, \mathit{filter})$
        if $(\mathit{SSMipFilter} = \text{Linear})$ then
            $r \leftarrow \text{lookup}(s_0, s_1, \lceil L \rceil, \mathit{filter})$
            $f \leftarrow s_{0w} - \lfloor s_{0w} \rfloor$
            $q \leftarrow (1 - f)q + fr$
        endif
    endif
    $d \leftarrow q$
```

The lookup function samples the texture bound to the sampler at the appropriate coordinates. If the texture is mipmapped, then a mipmap level must be chosen before addressing a texture level. The w component of s_0 is used to select the mipmap level of detail. If this value is negative, then the effect is to select the most detailed level of the texture with a magnification filter. The fractional part of this value may be used to interpolate between levels of detail if SS Mip Filter is set to D3DTEXF_LINEAR on the sampler. The sampler states SS Mip Map LOD Bias and SS Max Mip Level are honored for textures bound to the vertex shader sampler registers.

For a 2D texture, the x and y components of s_0 are used as the coordinates. For a cube or volume texture the x, y and z components of s_0 are taken as the texture coordinates. A value of $\langle 0, 0, 0, 1 \rangle$ is returned when a texture stage is sampled that has no bound texture.

The pseudocode for this instruction handles three basic scenarios: magnification, minification and mipmapping. In magnification, the most detailed texture mipmap level is used for the texture, influenced by the SS Max Mip Level sampler state and the filter used is the magnification filter. In minification, the appropriate texture level is sampled using the minification filter. Again, the SS Max Mip Level sampler state influences the level chosen. If linear mipmap filtering is enabled and the texture is minified, then a sample on the adjacent mipmap level is blended with the existing minified sample. Conceputally, the lookup function performs the sampling of a texture level using a set of texture coordinates, a sampler, a level number and a sampling filter. This is purely an attempt to illustrate the behavior of this instruction and is not meant to imply any particular implementation or design.

9.13.1 Version 3.0 Flow Control Nesting Limits

The general mechanism of flow control limits described in subsection 9.11.1 applies to version 3.0 as well. Version 3.0 meets or exceeds the limits for version 2.x in every respect as summarized in table 9.7. The static flow count described in subsection 9.11.1 no longer applies in version 3.0; the only limit to the number of flow control instructions is the total number of instructions allowed in a shader.

9.14 Manipulating Shaders

In section 5.11, we described the `CreateVertexShader`, `GetVertexShader` and `SetVertexShader` methods on the device for manipulating vertex shaders. Your application will use D3DX to assemble or compile vertex shader source code into an appropriate `DWORD` array for use with `CreateVertexShader`.

If you need to construct shaders dynamically at run-time, an easy way to do this is to construct the shader function as a string and use D3DX to assemble the string into a `DWORD` array. The assembly is fast and a cache of already assembled shaders can be used to avoid re-assembling shaders unnecessarily. Dynamic construction of high-level shaders can also be performed in this way, although compiling from high-level shader language is more CPU intensive than assembly language.

The vertex shader constant register file properties of the device are directly manipulated through the `GetVertexShaderConstant` and `SetVertexShader-Constant`, methods. Each register file has its own device methods with the corresponding data type on its arguments.

```
HRESULT GetVertexShaderConstantB(DWORD start,
            BOOL *value,
            DWORD count);
HRESULT GetVertexShaderConstantF(DWORD start,
            float *value,
            DWORD count);
HRESULT GetVertexShaderConstantI(DWORD start,
            int *value,
            DWORD count);
HRESULT SetVertexShaderConstantB(DWORD start,
            const BOOL *value,
            DWORD count);
HRESULT SetVertexShaderConstantF(DWORD start,
            const float *value,
            DWORD count);
HRESULT SetVertexShaderConstantI(DWORD start,
            const int *value,
            DWORD count);
```

The `start` argument gives the number of the first register retrieved or stored and the `count` argument gives the number of four dimensional vector values to retrieve or store. The `value` parameter should point to enough storage for all the data stored or retrieved. For example, the following code snippet stores a single four dimensional vector value in register `c15`.

```
const float data[4] = { 1.f, 0.f, 0.f, 0.f };
device->SetVertexShaderConstantF(15, &data[0], 1);
```

Vertex shader constants can also be changed implicitly by the vertex shader declaration, or when a vertex shader is set on the device if the shader includes `vsDef` instructions. The contents of the constant registers persist until the device is reset. If several vertex shaders use the same constant register layout, then the constant registers can be loaded once and the application can switch back and forth between several vertex shaders without reloading the constant registers each time. The maximum number of constant registers supported by the device is given by the `MaxVertexShaderConst` member of `D3DCAPS9`. This value will always be at least 96 for vertex shader architecture version 1.1 as shown in table 9.2.

A device supports vertex shaders if the `VertexShaderVersion` member of `D3DCAPS9` is non-zero. The least significant `BYTE` contains the minor version number and the next most significant `BYTE` contains the major version number. For instance, a device that supports vertex shader version 1.1 would correspond to the value `0x0101`. The macros `D3DSHADER_VERSION_MAJOR` and `D3DSHADER_-VERSION_MINOR` can be used to extract the major and minor version numbers, respectively, from the `VertexShaderVersion` member.

```
DWORD D3DSHADER_VERSION_MAJOR(DWORD version);
DWORD D3DSHADER_VERSION_MINOR(DWORD version);
DWORD D3DVS_VERSION(DWORD major, DWORD minor);
```

The high `WORD` of the version member contains `0xFFFE`, so care should be taken when performing version comparisons without the provided macros. The `D3DVS_VERSION` macro constructs a `DWORD` value that can be used in a direct comparison with the `VertexShaderVersion` member. For example, the following code snippet checks for version 1.1 vertex shader architecture support:

```
bool
vs1_1_supported(const D3DCAPS *caps)
{
    return caps->VertexShaderVersion >= D3DVS_VERSION(1, 1);
}
```

A device in software vertex processing mode always supports version 3.0 of the vertex shader architecture. Therefore it also supports versions 1.1, 2.0, and 2.x. To check for 2.x architecture support, check for a base support of shader level 2.0 or higher and then examine the `VS20Caps` member of `D3DCAPS9`.

When w-buffering is used, a proper projection matrix must still be set on the device, even though the projection matrix will not be used with a vertex shader. The Direct3D runtime uses the projection matrix with w buffering to determine some auxiliary values.

9.15 Drawing Multiple Instances

Version 3.0 of the vertex shader architecture introduces the ability to sample different vertex streams at different rates. This allows you to draw multiple instances of a model with data that varies per-instance as well as per-vertex. The scene data is split into at least two streams, with one stream containing per-vertex data and a second stream containing per-instance data. The sampling frequency of the source streams is set with the `SetStreamSourceFreq` method.

TODO: explain stream frequency as counters operating on each stream

```
HRESULT SetStreamSourceFreq(UINT stream, UINT frequency);
```

The `frequency` parameter tells the runtime how many times to reuse each set of vertex components before advancing to the next set of components within the stream. Flags telling the runtime whether to interpret the stream as per-vertex data or per-instance data are logically orred to the repeat count. Hardware vertex processing with vertex shader version 3.0 supports indexed primitives with instancing. Instancing with non-indexed primitives is only supported with software vertex processing and a version 3.0 vertex shader. Additionally, the per-vertex streams must be bound to consecutive streams beginning with stream zero. The per-instance streams are bound to higher numbered streams.

TODO: Diagram showing indexed data case

In the simplest case of drawing n instances with identical geometry and differing per-instance data, the geometry would repeat n times and the per-instance data would repeat once for each primitive. With differing amounts of repetition on the two types of streams, you can achieve various levels of reuse of vertex buffer data with a single `DrawIndexedPrimitive` call.

For indexed primitives, set the geometry data stream frequencies to `D3DSTREAMSOURCE_INDEXEDDATA` combined with the number of times the stream data should be reused. Set the per-instance data stream frequencies to `D3DSTREAMSOURCE_INSTANCEDATA` combined with the reuse count.

```
#define D3DSTREAMSOURCE_INDEXEDDATA  (1<<30)
#define D3DSTREAMSOURCE_INSTANCEDATA (2<<30)
```

The following code snippet demonstrates how to draw 20 instances with identical geometry on stream zero and different per-instance data on stream one. In this simple example, the per-vertex data consists of a position and normal and the per-instance data holds a position offset and a diffuse color. The per-instance offset would be used within the vertex shader to reposition each instance within the scene and the diffuse color used in a typical lighting calculation with the position and normal to compute the colors fed to the rasterizer.

```
// Structure for geometry vertex data
struct GeometryVertex
{
    D3DVECTOR m_position;
    D3DVECTOR m_normal;
};

// Structure for instance vertex data
struct InstanceVertex
{
    D3DVECTOR m_offset;
    D3DCOLOR m_diffuse;
};

// Fill vertex buffers holding the data
IDirect3DVertexBuffer9 *geometry = fill_geometry();
IDirect3DVertexBuffer9 *instances = fill_instances();

// Bind the geometry stream and repeat it 20 times per DIP
THR(device->SetStreamSource(0,
    geometry, 0, sizeof(GeometryVertex));
THR(device->SetStreamSourceFreq(0,
    D3DSTREAMSOURCE_INDEXEDDATA | 20));

// Bind the instance stream and repeat it 1 time per DIP
THR(device->SetStreamSource(1,
    instances, 0, sizeof(InstanceVertex));
THR(device->SetStreamSourcFreq(0,
    D3DSTREAMSOURCE_INSTANCEDATA | 1));
```

For non-indexed primitives, the geometry data streams must contain a complete copy of all the vertices for each instance drawn. The geometry streams have their `frequency` set to the number of instances to be drawn. The instance streams have their `frequency` set to the number of vertices in an instance. This allows each instance component to be repeated for every vertex in the geometry for the instance.

9.16 Shader Design

Given the large range of functionality available in fixed-function vertex processing, designing a shader to perform similar operations can be a daunting task.

Presumably, the reason you want a vertex shader instead of fixed-function processing is because you wish to perform some custom processing not supported by the fixed-function pipeline. However, it is likely that you will still need to duplicate some of the functionality of the fixed-function pipeline for portions of your vertex processing. You cannot use fixed-function processing and a vertex shader program simultaneously; you must choose one or the other. The next section gives vertex shader fragments that implement each of the stages of fixed-function processing. In this section we outline a strategy for implementing a shader from scratch.

Start with algorithm for what you want to accomplish. Next, write down the equations for the necessary operations in each step of the algorithm. Divide algorithm into steps that can be implemented and debugged individually and combined to product the final result. For instance, first write a shader that incorporates only the position component of your algorithm and get that working before moving on to computing color, fog, point size and texture coordinates.

For each step in your algorithm, write down the shader instructions that implement the equations for that step. As shader instructions are very simple, it is likely that you will need more than one instruction for all but the most simplest of formulas. Implement these instructions in your shader. Edit and revise the implementation until you're confident that it is working properly.

Next, write shader instructions that combine any previously computed results for the final algorithm using the results computed from the step you just implemented. Repeat this process for all the steps in your algorithm.

Finally, look for ways to reduce the number of instructions and constant registers used by your shader. The presence of any `mov` instructions in your shader is a good place to start. Usually a `mov` instruction can be eliminated by the use of an appropriate source or destination operand in another instruction. Look for places in your shader where you can replace a `mul` and `add` instruction pair with a single `mad` instruction. Multiple scalar constants can be combined into a single constant register. Scalars can be replicated to multiple channels to create a vector constant using the source operand multiplexing modifier. If you are using only part of the results of a complex instruction, consider expanding the complex instruction to a series of simple instructions that only compute what you need. For example, if only the x component resulting from an `m4x4` instruction is used, replace it with a single `dp4` instruction that transforms the x component.

9.17 Common Computations

You may need to compute values in your vertex shader that cannot be directly realized by a single instruction. Of course, multiple instructions can be combined to compute results. This section outlines a few common operations that can be used in your vertex shaders and will serve as examples to inspire you in coding other operations.[3]

9.17.1 Constant Generation

Since each instruction can reference only a single constant register, using multiple constant registers as source operands is illegal. You can use a `mov` instruction to copy a constant to a temporary register. However, if the constant is one or zero you can also use the conditional instructions to generate the value without consuming a constant register. An implementation can recognize this as always generating the constants one and zero, regardless of the contents of the constant register file, and may be able to perform additional optimizations with this knowledge.

```
slt r0, r0, r0      ; r0 = (r0 < r0) = <0, 0, 0, 0>
sge r0, r0, r0      ; r0 = (r0 >= r0) = <1, 1, 1, 1>
```

9.17.2 Fractional Part

The simple instruction `expp` calculates $s_w - \lfloor s_w \rfloor$ in the y component of its result, which is the fractional portion of the w component of the source operand. The complex instruction `frc` calculates the fractional portion of all components in its source operand in multiple slots. However, sometimes you only need the fractional portion of one, two, or three components of the source operand. Using portions of the code below you can compute the fractional parts of only the components you need. Use the `frc` instruction when you need the fractional part of all four components.

```
; compute fractional part of r0 in r1
expp r1.y, r0.x     ; r1.y = r0.x - floor(r0.x)
mov r1.x, r1.y
expp r1.y, r0.z     ; r1.y = r0.z - floor(r0.z)
mov r1.z, r1.y
expp r1.y, r0.w     ; r1.y = r0.w - floor(r0.w)
mov r1.w, r1.y
expp r1.y, r0.y     ; r1.y = r0.y - floor(r0.y)
```

9.17.3 Absolute Value

The most common way of thinking of the absolute value function $|x|$ would be in terms of a conditional expression:

$$|x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$$

[3]This section is based on the nVidia white paper "Where Is That Instruction? How To Implement "Missing" Vertex Shader Instructions", by Matthias Wloka. Used with permission.

Another equivalent formulation of $|x|$ is to take the maximum of x and $-x$, while $-|x|$ becomes the minimum of x and $-x$.

$$|x| = \max(x, -x) \qquad\qquad -|x| = \min(x, -x)$$

This latter formulation can be directly computed in vertex shader instructions.

```
; |r0|: absolute value of r0's components
max r0, r0, -r0

; -|r0|: negative of absolute value of r0's components
min r0, r0, -r0
```

9.17.4 Division

The instruction set does not provide a scalar division instruction directly, but you can compute x/y as the product of x and $1/y$.

```
; r2 = r0.x / r1.x
rcp r2.x, r1.x          ; 1/r1.x
mul r2.x, r2.x, r0.x    ; r0.x/r1.x
```

9.17.5 Square Root

Similar to division, there is no scalar square root instruction. You can compute the square root as a product of x and $1/\sqrt{x}$: $\sqrt{x} = x/\sqrt{x}$. You might be tempted to compute the reciprocal of the reciprocal square root, but that results in a loss of precision.

```
; r1.x = square root of r0.x
rsq r1.x, r0.x          ; 1/sqrt(r0.x)
mul r1.x, r1.x, r0.x    ; r0.x/sqrt(r0.x) = sqrt(r0.x)
```

9.17.6 Conditional Selection

Occasionally you'll want to select one of two values, v_1 or v_2 based on a condition c. Without actual branching in the instruction set, you can simulate this as the following expression.

$$x = cv_1 + (1 - c)v_2$$

Where c is 1 when value v_1 is to be selected and 0 when the value v_2 is selected. The expression above is readily computed using the `mad` instruction after we compute c and $1 - c$. If we are computing c from a conditional test, we can compute $1 - c$ by simply inverting the test used to compute c.

```
; r0 = (r1 < r2) ? r3 : r4
slt r0, r1, r2          ; c = (r1 < r2)
mul r3, r3, r0          ; r3 = r3*c
sge r0, r1, r2          ; c = (r1 >= r2)
mad r0, r0, r4, r3      ; r0 = c*r4 + r3
```

If we can compute the value $v_1 - v_2$ without any loss of precision, we can shave off another instruction here by rewriting the above formula.

$$x = c(v_1 - v_2) + v_2$$

```
; r0 = (r1 < r2) ? r3 : r4
slt r0, r1, r2          ; c = (r1 < r2)
sub r1, r3, r4
mad r0, r0, r1, r4      ; r0 = c*(r3 - r4) + r4
```

9.17.7 Clamping To An Interval

We can clamp a scalar value to the interval $[A, B]$ using the `min` and `max` instructions with the interval endpoints stored in a constant register. All the components of a vector can be clamped in this manner.

$$\text{clamp}(x, A, B) = \begin{cases} A, & x < A \\ x, & A \le x \le B \\ B, & x > B \end{cases}$$

```
; interval [A, B]
def c0, A, B, 0, 0

; r0 = clamp(r0, A, B)
max r0, r0, c0.x
min r0, r0, c0.y
```

9.17.8 Floor and Ceiling

The `expp` instruction computes $s_w - \lfloor s_w \rfloor$ which we can use to compute the floor and ceiling functions of any scalar value.

$$\begin{aligned} \lfloor x \rfloor &= x - (x - \lfloor x \rfloor) \\ \lceil x \rceil &= x + ((-x) - \lfloor -x \rfloor) \end{aligned}$$

```
; r1.y = floor(r0.x)
expp r1.y, r0.x
add r1.y, r0.x, -r1.y

; r1.y = ceil(r0.x)
expp r1.y, -r0.x
add r1.y, r0.x, r1.y
```

9.17.9 Vector Cross Product

We can use the multiplex source operand modifier to compute a vector cross product. Each component in the result is a difference of two terms, with each term being the product of two components from the input vectors.

$$\vec{c} = \vec{a} \times \vec{b} = \langle a_y b_z - a_z b_y,\, a_z b_x - a_x b_z,\, a_x b_y - a_y b_x \rangle$$

The straightforward implementation is to use two `mul` instructions to compute the component product terms and then `add` or `sub` to compute the difference. However, the `mad` instruction can be used to combine one of the multiply operations with the subtraction. With three-dimensional vectors, the w component of the registers is not needed, so we use the multiplex modifier to compute $a_w b_w - a_w b_w$, resulting in a value of zero for the w channel. Write masks can be used on the `mad` instruction to avoid overwriting the w channel if so desired.

```
; r2 = cross(r0, r1)
mul r2, r0.yzxw, r1.zxyw
mad r2, r0.zxyw, -r1.yzxw, r2
```

9.17.10 Vector Normalization

A three dimensional vector is normalized by scaling the vector by the reciprocal of its magnitude. The square of the magnitude of a vector can be computed as a dot product of the vector with itself. It is then a simple matter to compute the reciprocal square root and multiply this through the vector to normalize it.

```
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w
```

9.17.11 Transposed Matrix Multiplication

The matrix multiply instructions, such as `m4x4`, assume that the matrix has been stored in the constant register file in a transposed form, with matrix columns stored in each constant register. If you have a matrix stored in the constant register file with one row per register, you can still perform a matrix multiplication with multiple simple vertex shader instructions.

```
; c0-c3 are the 4 rows of the 4x4 matrix
; r0 is the 4D point to be transformed
mul r0, c0, r1.x
mad r0, c1, r1.y, r0
mad r0, c2, r1.z, r0
mad r0, c3, r1.w, r0
```

The method for a smaller matrix (2×3, 3×3, etc.), or a smaller dimension point, is similar.

9.17.12 Signum Function

The signum function returns the sign of its argument:

$$\text{signum}(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

To compute this, we can perform two exclusive comparisons and subtract the result of one from the other. We get the correct result because a scalar value cannot simultaneously be positive and negative. If the value is positive, then we compute $1 - 0 = 1$. If the value is negative then we compute $0 - 1 = -1$.

```
; constant for zero
def c0, 0, 0, 0, 0

; r1 = signum(r0)
slt r1, r0, c0        ; r1 = (r0 < 0)
slt r2, c0, r0        ; r2 = (0 < r0)
sub r1, r2, r1
```

9.17.13 Minimum and Maximum Vector Component

We can compute a vector with a 1 in the component that has the minimum magnitude of all the components in the vector. Similarly, we can compute a vector with a 1 in the component tha thas the maximum magnitude of all components in the vector. This can be useful as a modulating factor to extract the minimum or maximum vector component.

The minimum component is obtained by performing two comparisons on the source register and combining their results. The comparisons use the multiplexing source operand modifier to perform three component comparisons in a single slot.

$$\text{mincomp}(\vec{a}) = \begin{cases} \langle 1, 0, 0, 0\rangle, & |a_x| \le |a_y| \text{ and } |a_x| < |a_z| \\ \langle 0, 1, 0, 0\rangle, & |a_y| < |a_x| \text{ and } |a_y| \le |a_z| \\ \langle 0, 0, 1, 0\rangle, & |a_z| \le |a_x| \text{ and } |a_z| < |a_y| \end{cases}$$

```
; compute r1 = mincomp(r0)
max r1, r0, -r0         ; r1 = |r0|
slt r2, r1, r1.zxyw     ; x < z, y < x, z < y
sge r1, r1.yzxw, r1     ; y >= x, z >= y, x >= z
min r1, r2, r1          ; r1 = r1 && r2
```

The maximum component is obtained in the same manner.

$$\text{maxcomp}(\vec{a}) = \begin{cases} \langle 1, 0, 0, 0\rangle, & |a_x| \ge |a_y| \text{ and } |a_x| > |a_z| \\ \langle 0, 1, 0, 0\rangle, & |a_y| > |a_x| \text{ and } |a_y| \ge |a_z| \\ \langle 0, 0, 1, 0\rangle, & |a_z| \ge |a_x| \text{ and } |a_z| > |a_y| \end{cases}$$

```
; compute r1 = maxcomp(r0)
max r1, r0, -r0         ; r1 = |r0|
slt r2, r1.zxy, r1      ; z < x, x < y, y < z
sge r1, r1, r1.yzx      ; x <= y, y <= z, z <= x
min r1, r2, r1          ; r2 = r1 && r2
```

9.17.14 Trigonometric Functions

While vector and digital signal processing operations are well supported by the instruction set, trigonometric functions are one area where you must create your own functions from scratch. Several different approaches are outlined here, but the basic idea behind most of them is to perform some sort of polynomial approximation the function and compute the value of the polynomial function. Another approach is to use a table lookup scheme and interpolate between values in the table.

Power Series Approximation of Cosine and Sine

The Taylor series expansion for $\cos(x)$ and $\sin(x)$ are:

$$\begin{aligned}
\cos(x) &= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \\
&= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \\
\sin(x) &= \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \\
&= \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots
\end{aligned}$$

The following vertex shader snippet computes the power terms of x in a temporary register. The first four coefficients of each power series expansion are stored in the constant register file. By using a 4D dot product, the coefficients are multiplied by the powers and summed.

```
; scalar r0.x = cos(r1.x), r0.y = sin(r1.x)

def c0, PI,   1/2, 2*PI,   1/(2*PI)
def c1, 1.0, -1/2, 1/24,  -1/720
def c2, 1.0, -1/6, 1/120, -1/5040

; normalize argument into [-pi, +pi]
mad  r0.x,  r1.x, c0.w,  c0.y
expp r0.y,  r0.x
mad  r0.x,  r0.y, c0.z, -c0.x
```

```
; generate 1, (r0.x)^2, .. (r0.x)^6
dst  r2.xy, r0.x, r0.x
mul  r2.z,  r2.y, r2.y
mul  r2.w,  r2.y, r2.z

; generate r0.x, (r0.x)^3, .., (r0.x)^7
mul  r0,    r2,   r0.x
dp4  r0.y,  r0,   c2        ; compute sin(r0.x)
dp4  r0.x,  r2,   c1        ; compute cos(r0.x)
```

Table Lookup Cosine and Sine

A common way to evaluate complex functions is to build a data table of function values and linearly interpolate between the two closest values.[4] Computing trigonometric functions by table lookup is quite fast, provided you have enough constant registers to devote to the table. This snippet uses only the two components of the constant registers for the table values, so additional function tables can be overlapped with this table for $\sin x$ and $\cos x$. For simplicity, the example shown here uses a table with only 6 entries for the entire range of $[0, 2\pi]$. In practice, a table with 16 to 32 entries should be used for better accuracy. Another alternative is to use a table with fewer entries and exploit the symmetry of the trigonometric functions:

$$\begin{aligned}\sin(x) &= -\sin(-x), & 0 \le x \le \pi\\ \cos(x) &= \cos(-x), & 0 \le x \le \pi\end{aligned}$$

at the cost of a few additional instructions.

```
def c0,  1, 5, 0, 1/(2*PI)

; the table values, shown symbolicly
def c1, SIN(0.0 * 2*PI), COS(0.0 * 2*PI), 0, 0
def c2, SIN(0.2 * 2*PI), COS(0.2 * 2*PI), 0, 0
def c3, SIN(0.4 * 2*PI), COS(0.4 * 2*PI), 0, 0
def c4, SIN(0.6 * 2*PI), COS(0.6 * 2*PI), 0, 0
def c5, SIN(0.8 * 2*PI), COS(0.8 * 2*PI), 0, 0
def c6, SIN(1.0 * 2*PI), COS(1.0 * 2*PI), 0, 0

; scalar r0.x = cos(r1.x), r0.y = sin(r1.x)
mul r0.x, c0.w, r1.x        ; normalize input
expp r1.y, r0.x             ; to [0..1] range

; scale to table-size and add table-base
mad r0.w, c0.y, r1.y, c0.x
```

[4]This shader snippet originally provided by Phil Teschner.

```
mov a0.x, r0.w
expp r1.yw, r0.w          ; use fractional part for lerp
add r1.x, r1.w, -r1.y     ; use (1 - fractional part) for lerp

; linear interpolation
mul r0.xy, c[a0.x].xy, r1.x
mad r0.xy, c[a0.x + 1].xy, r1.y, r0.xy
```

Arctangent

The following vertex-shader fragment computes $\tan^{-1}$ to a precision of 1e-5 using a remodulated Pade/Chebychev expansion[5]. The arc tangent is evaluated in the interval $[-1, 1)$ and extended towards $-\infty$ or $+\infty$ using the formulae $\tan^{-1}(x) = \pi/2 - \tan^{-1}(1/x), x > 1$ and $\tan^{-1}(x) = -\tan^{-1}(-x)$.

```
; c0,c1,c2,c3 and r0,r1,r2,r3,r4 are used
def c0,  1.98888994, 2.13982195,  0.49868736, 0.01307137
def c1,  1.98888995, 2.80278416,  1.03518917, 0.08158508
def c2, -1.0,        1.0,        -1.0,        0.0
def c3, -PI/2.0,     0.0,         PI/2.0,     0.0

; compute r0.x = arctan(r0.x)
slt r1.xy, r0.xx, c2.xy     ; range check: -inf..-1, -inf..1
sge r1.zw, r0.xxxx, c2.xyxy ; range check: -1..+inf, 1..+inf
mul r2.xyz, r1.xyw, r1.xzw  ; && conditions
rcp r0.yz, r0.x             ; argument's reciprocal: (1/x, x, 1/x)
dp3 r0, r2.xyz, r0.yxz      ; (t, t, t, t)
dst r0, r0, r0              ; (1, t^2, t, t)
mul r4, r0.xyyy, r0.xxzy    ; (1, t^2, t^3, t^4)
mul r3, r4.xxyw, r0.xyyy    ; (1, t^2, t^4, t^6)
mul r4, r3.xyzw, r0.zzzz    ; (t, t^3, t^5, t^7)
dp4 r3, r3, c1              ; denominator: e + ft^2 + gt^4 + ht^6
dp4 r4, r4, c0              ; numerator: at + bt^3 + ct^5 + dt^7
rcp r3, r3
mul r3, r3, r4              ; arctan(t) between [-1, 1)
dp3 r0, r2, c2              ; selecting correct sign
dp3 r1, r2, c3              ; selecting correct offset
mad r0.x, r0, r3, r1        ; extending domain to (-inf, +inf)
```

9.17.15 Exponential and Logarithmic Functions

The simple intructions `expp` and `logp` compute a 10 bit precision result in a single slot. The complex instructions `exp` and `log` compute a full 21 bits of precision in 10 slots or less. If you need more precision than that provided by the simple instructions, but less than that provided by the complex instructions,

[5]This shader snippet by Marco Salvi

you may be able to achieve a satisfactory result in less slots than the complex instructions. As with the trigonometric functions, power series approximations to the exponential and logarithm functions may be used in combination with some space from the constant register file. (The complex instructions `exp` and `log` do not require any constant registers.)

Exponential

The following snippet computes a full-precision base 2 exponential of its argument by a power-series approximation.

```
; power series coefficients
; (shown on multiple lines for space reasons)
def c0, 1.00000000,    -6.93147182e-1,
        2.40226462e-1, -5.55036440e-2
def c1, 9.61597636e-3, -1.32823968e-3,
        1.47491097e-4, -1.08635004e-5

; compute scalar r0.z = exp2(r1.z)
expp r0.xy, r1.z
dst r1, r0.y, r0.y
mul r1, r1.xzyw, r1.xxxy    ; 1,x,x^2,x^3
dp4 r0.z, r1, c0
dp4 r0.w, r1, c1
mul r1.y, r1.z, r1.z        ; compute x^4
mad r0.w, r0.w, r1.y, r0.z  ; add the first to the last 4 terms
rcp r0.w, r0.w
mul r0.z, r0.w, r0.x        ; multiply by 2^I
```

Logarithm

The following snippet computes a full-precision base 2 logarithm of its argument by a power-series approximation.

```
def c0, 1.44268966,  -0.721165776, 0.478684813,   -0.347305417
def c1, 0.241873696, -0.137531206, 5.20646796e-2, -9.31049418e-3
def c2, 1.0,          0.0,         0.0,            0.0

; scalar r0.x = log2(r1.x)
logp r0.x, r1.x
add r0.y, r0.x, -c2.x       ; subtract 1.0
dst r1, r0.y, r0.y
mul r1, r1.xzyw, r1.xxxy    ; 1, x, x^2, x^3
dp4 r0.z, r1, c0
dp4 r0.w, r1, c1
mul r1.y, r1.z, r1.z        ; compute x^4
mad r0.w, r0.w, r1.y, r0.z  ; add the first to the last 4 terms
```

`r0`	scratch
`r1`	scratch
`r2`	scratch
`r3` (`Rd`)	scratch, light vector
`r4` (`Rr`, `Rf`, `Rl`)	scratch, reflectance, eye or light vectors
`r5` (`Rs`, `Rv`)	scratch, sphere vector, eye vector
`r6` (`Rx`)	scratch, specular color
`r7` (`Rc`)	scratch, diffuse color
`r8` (`RH`)	scratch, half-angle vector
`r9` (`Rh`)	homogeneous eye space position
`r10` (`Re`)	cartesian eye space position
`r11` (`Rn`)	eye space normal

Table 9.11: Register layout for fixed-function emulation.

```
mad r0.x, r0.w, r0.y, r0.x  ; exponent add
```

9.18 Fixed-Function Processing

This section presents the fixed-function vertex processing covered in the previous chapters as vertex shader program elements[6]. You can create any combination of fixed-function processing by joining these shader program elements together and resolving the register references.

9.18.1 Register Layout

A layout of the temporary registers used in the fixed-function pipeline shader elements is given in table 9.11. Most of the registers are used as temporary scratch storage space for storing intermediate results. Alternate names for some of the registers are used to aid understanding in the shader elements following this subsection. The alternate names are shown in parenthesis.

The input register layout to the shader is arbitrary and depends on the application's organization of data. In the following shader elements, the input position and normal data are designated by the symbols `vPOSITION` and `vNORMAL`, respectively.

9.18.2 Constant Register Layout

The following table gives the layout of the constant registers used in the fixed-function pipeline shader elements.

Register	**Meaning or Value**

[6]This section is based on a presentation by Erik Lindholm of NVIDIA Corporation, used with permission.

`c[NORMAL0_MATRIX_X]`	inverse transpose world × view matrix 0
`c[NORMAL0_MATRIX_Y]`	
`c[NORMAL0_MATRIX_Z]`	
`c[NORMAL1_MATRIX_X]`	inverse transpose world × view matrix 1
`c[NORMAL1_MATRIX_Y]`	
`c[NORMAL1_MATRIX_Z]`	
`c[NORMAL2_MATRIX_X]`	inverse transpose world × view matrix 2
`c[NORMAL2_MATRIX_Y]`	
`c[NORMAL2_MATRIX_Z]`	
`c[NORMAL3_MATRIX_X]`	inverse transpose world × view matrix 3
`c[NORMAL3_MATRIX_Y]`	
`c[NORMAL3_MATRIX_Z]`	
`c[WORLDVIEW0_MATRIX_X]`	world × view matrix 0
`c[WORLDVIEW0_MATRIX_Y]`	
`c[WORLDVIEW0_MATRIX_Z]`	
`c[WORLDVIEW1_MATRIX_X]`	world × view matrix 1
`c[WORLDVIEW1_MATRIX_Y]`	
`c[WORLDVIEW1_MATRIX_Z]`	
`c[WORLDVIEW2_MATRIX_X]`	world × view matrix 2
`c[WORLDVIEW2_MATRIX_Y]`	
`c[WORLDVIEW2_MATRIX_Z]`	
`c[WORLDVIEW3_MATRIX_X]`	world × view matrix 3
`c[WORLDVIEW3_MATRIX_Y]`	
`c[WORLDVIEW3_MATRIX_Z]`	
`c[PROJECTION_MATRIX_X]`	projection matrix
`c[PROJECTION_MATRIX_Y]`	
`c[PROJECTION_MATRIX_Z]`	
`c[PROJECTION_MATRIX_W]`	
`c[COMPOSITE_MATRIX_X]`	world × view × projection matrix
`c[COMPOSITE_MATRIX_Y]`	
`c[COMPOSITE_MATRIX_Z`	
`c[COMPOSITE_MATRIX_W]`	
`c[TEXTURE_MATRIX_X]`	texture matrix
`c[TEXTURE_MATRIX_Y]`	
`c[TEXTURE_MATRIX_Z]`	
`c[TEXTURE_MATRIX_W]`	
`c[GLOBAL_ILLUMINATION]`	RGBA, emission + global ambient
`c[LIGHT_POSITION]`	x, y, z
`c[LIGHT_HALF_ANGLE]`	x, y, z, for infinite light w/local viewer
`c[LIGHT_AMBIENT]`	RGB, light × material
`c[LIGHT_DIFFUSE]`	RGB, light × material
`c[LIGHT_SPECULAR]`	RGB, light × material, specular power
`c[LIGHT_ATTENUATION]`	a0, a1, a2, spot power
`c[LIGHT_SPOT_DIRECTION]`	x, y, z, cos(CUTOFF)

c[POINT_PARAMETER]	size, max, min
c[POINT_ATTENUATION]	a0, a1, a2
c[TEXTURE_OBJECT_PLANE_X]	x, y, z, w
c[TEXTURE_OBJECT_PLANE_Y]	x, y, z, w
c[TEXTURE_OBJECT_PLANE_Z]	x, y, z, w
c[TEXTURE_OBJECT_PLANE_W]	x, y, z, w
c[TEXTURE_EYE_PLANE_X]	x, y, z, w
c[TEXTURE_EYE_PLANE_Y]	x, y, z, w
c[TEXTURE_EYE_PLANE_Z]	x, y, z, w
c[TEXTURE_EYE_PLANE_W]	x, y, z, w
c[EYE_POSITION]	x, y, z, w
c[CONSTANT0]	-1, 0, 1, 0.5

9.18.3 Coordinate Transformation

Vertex positions and normals are transformed by a concatenated world and view matrix. If skinning is being used, the position and normal must be transformed by each skinning matrix.

```
; position
m4x4 Rh, vPOSITION, c[MODELVIEW0_MATRIX_X]
m4x4 r3, vPOSITION, c[MODELVIEW1_MATRIX_X]
m4x4 r5, vPOSITION, c[MODELVIEW2_MATRIX_X]
m4x4 r7, vPOSITION, c[MODELVIEW3_MATRIX_X]
; normal
m3x3 Rn, vNORMAL, c[NORMAL0_MATRIX_X]
m3x3 r2, vNORMAL, c[NORMAL1_MATRIX_X]
m3x3 r4, vNORMAL, c[NORMAL2_MATRIX_X]
m3x3 r6, vNORMAL, c[NORMAL3_MATRIX_X]
```

9.18.4 Vertex Blending

If vertex blending is used, the transformed positions and normals must be combined through the vertex blend weights to get the final position and normal.

```
; weight preparation
mov r0, c[WEIGHT]
mov r0.w, c[CONSTANT0].z          ; if need new weight
dp4 r0.y, r0, c[CONSTANT0].xyyz   ; or new 2nd weight
dp4 r0.z, r0, c[CONSTANT0].xxyz   ; or new 3rd weight
dp4 r0.w, r0, c[CONSTANT0].xxxz   ; or new 4th weight

; position/normal blend
mul Rh, r0.x, Rh                  ; 1st weight
mul Rn, r0.x, Rn
```

```
mad Rh, r0.y, r3, Rh              ; 2nd weight
mad Rn, r0.y, r2, Rn
mad Rh, r0.z, r5, Rh              ; 3rd weight
mad Rn, r0.z, r4, Rn
mad Rh, r0.w, r7, Rh              ; 4th weight
mad Rn, r0.w, r6, Rn
```

9.18.5 Position Output

Finally, the blended vertex position must be transformed to homogeneous clip space and output to the `oPos` register. The blended normal is used in further lighting calculations but not output by the shader.

```
m4x4 oPos, Rh, c[PROJECTION_MATRIX_X]
```

If skinning is not being used, the shader can be simplified slightly by using a composite matrix containing the world, view and projection transformations. In this case, the `oPos` output can be computed directly from the vertex input.

```
m4x4 oPos, vPOSITION, c[COMPOSITE_MATRIX_X]
```

9.18.6 Normalize Eye Normal

If scaling transformations are present in the world or view matrices, then the eye-space vertex normal may need renormalizing. To normalize a vector, we need to divide the components of a vector by its magnitude. We can compute the magnitude of the vector by computing a dot-product of the vector with itself.

```
dp3 Rn.w, Rn, Rn
rsq Rn.w, Rn.w
mul Rn, Rn, Rn.w
```

9.18.7 Non-Homogeneous Eye Position

To compute the cartesian eye-space coordinate for the position, we need to divide the homogeneous position by its w component.

```
rcp r0.w, Rh.w
mul Re, Rh, r0.w
```

9.18.8 Eye Space Vectors

The distance between the vertex and the eye position is calculated.

```
add r0, -Re, c[EYE_POSITION]
dp3 r0.w, r0, r0
rsq r1.w, r0.w
mul Rv, r0, r1.w
dst Rf, r0.w, r1.w
```

9.18.9 Fog Output

If you are using fog, you'll need to compute a value for the `oFog` register. You can compute the fog in any manner you wish. Shown here are two examples for computing fog.

```
; radial fog
mov oFog.x, Rf.y

; linear Z fog
mov oFog.x, -Re.z
```

9.18.10 Point Parameters

For point sprite primitives, the vertex shader can compute the point size into the `oPts` output register.

```
dp3 r0.w, Rf, c[POINT_PARAMETER_ATTENUATION]
rsq r0.w, r0.w
mul r0.w, r0.w, c[POINT_PARAMETER].x
mul r0.w, r0.w, c[POINT_PARAMETER].y
max oPts.x, r0.w, c[POINT_PARAMETER].z
```

9.18.11 Lighting

Computing the lighting for a vertex is probably where your vertex shaders will most differ from each other. Recall that the fixed-function pipeline computes the light for each vertex as a sum of all the light reflected at that vertex. Each of these terms is represented in a vertex shader by a group of instructions that computes the reflected light at the vertex for a given source of light within the scene. All of these terms are accumulated to give the final diffuse and specular color components for the vertex.

For instance, if there are 3 lights enabled in the scene, then the constant register file is loaded with the corresponding data for the definition of the three lights. The vertex shader includes three groups of instructions that computes the reflected light for each enabled light and adds it to a running total. The final total reflected light is output in the `oD0` and `oD1` registers for the total diffuse and specular reflected light, respectively.

Initialization

```
; diffuse only
mov Rc, c[GLOBAL_ILLUMINATION]

; diffuse and specular
mov Rc, c[GLOBAL_ILLUMINATION]
mov Rx, c[CONSTANT0].y
```

Infinite Light or Infinite Viewer

```
dp3 r0.x, Rn, c[LIGHT_POSITION]
dp3 r0.y, Rn, c[LIGHT_HALF_ANGLE_VECTOR]
mov r0.w, c[LIGHT_SPECULAR].w
lit r0, r0
mad Rc.xyz, r0.x, c[LIGHT_AMBIENT], Rc
mad Rc.xyz, r0.y, c[LIGHT_DIFFUSE], Rc
; use Rc here for a single color
mad Rc.xyz, r0.z, c[LIGHT_SPECULAR], Rx
```

Spotlight, Local Viewer

```
; light direction/distance vectors
add r0, -Re, c[LIGHT_POSITION]
dp3 r0.w, r0, r0
rsq r1.w, r0.w
mul r1, r0, r1.w            ; direction
dst Rd, r0.w, r1.w          ; distance

; half-angle vector
add Rh, Rv, Rl
; normalize
dp3 Rh.w, Rh, Rh
rsq Rh.w, Rh.w
mul Rh, Rh, Rh.w

; distance attenuation
dp3 Rd.w, Rd, c[LIGHT_ATTENUATION]
rcp Rd.w, Rd.w

; spotlight cone attenuation
dp3 r0.y, Rl, -c[LIGHT_SPOT_DIRECTION]
add r0.x, r0.y, -c[LIGHT_SPOT_DIRECTION].w
mov r0.w, c[LIGHT_ATTENUATION].w
lit r0, r0
mul Rd, Rd.w, r0.z

dp3 r0.x, Rn, Rl
dp3 r0.y, Rn, RH
mov r0.w, c[LIGHT_SPECULAR].w
lit r0, r0
mul r0, r0, Rd.w
mad Rc.xyz, r0.x, c[LIGHT_AMBIENT], Rc
mad Rc.xyz, r0.y, c[LIGHT_DIFFUSE], Rc
; use Rc here for a single color
```

```
mad Rc.xyz, r0.z, c[LIGHT_SPECULAR], Rx
```

Lighting Output

```
; diffuse only
mov oD0, Rc

; diffuse and specular
mov oD0, Rc
mov oD1, Rx
```

9.18.12 Texture Coordinate Generation

Texture coordinates can be generated directly from the vertex data or can be passed to the shader. Here we show shader instructions for generating texture coordinates similar to the fixed-function pipeline.

```
; pass-thru
mov r0, v[TEX0]
```

Initialization

```
; reflection vector
mul r0, Rn, c[EYE_POSITION].w
dp3 Rr.w, Rn, Rv
mad Rr, Rr.w, R0, -Rv

; sphere map vector
add r0, c[CONSTANT0].yyzy, Rr
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0.xyz, r0, c[CONSTANT0].wwyy
mad Rs, r0.w, r0, c[CONSTANT0].wwyy
```

Texture Coordinate Generation

The texture coordinates can be generated from a sphere mapping, the vertex normal

```
; object space plane
m4x4 r0, vPOSITION, c[TEXTURE_OBJECT_PLANE_X]

; eye space plane
m4x4 r0, Rh, c[TEXTURE_EYE_PLANE_X]

; sphere map
mov r0.xy, Rs
```

```
; normal vector
mov r0.xyz, Rn

; reflection vector
mov r0.xyz, Rr
```

Texture Coordinate Transform

The texture coordinates can be transformed by a matrix before writing to the appropriate output register.

```
m4x4 oT0, r0, c[TEXTURE_MATRIX_X]
```

9.19 Beyond Fixed-Function Processing

With a vertex shader program, we are not limited to the processing provided by the fixed-function pipeline. We are only limited by the instruction set, the size of the constant register file and the maximum size of a vertex shader program.

For instance, a vertex shader could provide for a large number of lights compared to the fixed-function pipeline. Each light's definition is stored in the constant register file and the vertex shader computes each light's contribution to a vertex and adds it to a running total.

A vertex shader can provide for a much wider range of fog effects when the fixed-function fog factor formulas are overly restrictive. Fog can be created based on vertex height, or any other formula of the vertex data.

A vertex shader can implement vertex-based color keying. The transparency of each vertex can be computed from its color, position or other vertex data. Since the vertex shader also computes lighting, a color key effect can be applied to the vertex based on the lit color or the unlit color.

A common approach in scientific visualization applications is to apply a false color map to vertices based on their position. This can be achieved per-vertex with a vertex shader that computes the color of a vertex from its position.

Although a vertex shader has access to only a single vertex, the constant register file can be used to pass information about the current frame being rendered as well as information about previous and future frames. For instance, the constant registers can be loaded with the transformation matrix for the previous, current, and subsequent frames. The vertex shader can use this information to create special temporal effects on the vertex data.

Vertex shader programs are more powerful when used in conjunction with pixel shader programs. In this scenario, the vertex shader program computes parameters to the pixel shader on a per-vertex basis. Usually these rendering parameters are passed through to the pixel shader through the texture registers. The texture output registers have a wider dynamic range than the color registers, which are clamped to the interval $[0, 1]$ on output from the vertex shader. The rasterizer will interpolate texture coordinates in a perspective correct manner, reducing distortion in perspective projections. The pixel shader will then use the interpolated texture coordinates either for direct texture lookup, or as additional inputs to a computation for per-pixel effects.

Vertex shaders can also be used to compress the size of a vertex. Remember that vertex declarations allow us to declare the data type of a vertex component. While the fixed-function pipeline expects vertices in a particular format, vertex shaders can use other representations for vertex components. For instance, with the fixed-function pipeline vertex normals are always encoded as a triplet of `float`s, while with a vertex shader you could use `D3DDECLTYPE_SHORT4` to encode the normal as a signed short integer. This would reduce the size for a surface normal vertex component from 12 bytes to 8 bytes at the expense of some shader instructions for providing the appropriate scaling of the encoded normal value. Similar tradeoffs can be made for other vertex components to reduce the size of a vertex further.

9.20 `rt_VertexShader` Sample Application

The sample code accompanying this book includes a program called `rt_Vertex-Shader` that allows you to interactively explore vertex shaders applied to a mesh. With this program you can load a mesh from a .x file and interactively write and assemble a vertex shader to be used with the mesh.

9.21 Vertex Shader SDK Samples

Several of the samples provided in the SDK use vertex shaders. In this section we'll briefly describe the vertex shaders used in the SDK samples. All of the vertex shaders discussed here are listed after this section at the end of the chapter. The vertex shaders in the SDK samples are not necessarily optimized, they are only intended to demonstrate a particular technique or effect.

9.21.1 BumpSelfShadow

This advanced bump-mapping sample uses four different vertex shaders. The sample also uses pixel shaders when they are supported by the device. For reference while examining the vertex shaders, the constant registers used by this sample are given in table 9.13.

The shaders in listing 9.1, listing 9.2 and listing 9.3 are used when no pixel shaders are supported. The first shader is used to render the diffuse color into a secondary render target and is stored in the member variable `m_dwShadow-Shader` in the source code. Lines 18–19 of this shader use the world matrix to transform the normal and binormal (or tangent) of the vertex into the same coordinate system as the light direction vector. Lines 23–24 compute a cross product of the normal and binormal to create the third axis of a coordinate system. This coordinate system is then used in lines 27–29 to transform the light direction vector. These computations are typical of tangent space bump

Register(s)	Meaning
0, 1, 2, 3	World matrix
8, 9, 10, 11	Composite of World, View and Projection matrices.
12	Normalized light direction vector
16, 17, 18, 19	Composite of World and View matrices.
32	$\langle 1, 1, 1, 1\rangle$
33	$\langle 0.5, 0.5, 0.5, 0.5\rangle$

Table 9.13: BumpSelfShadow vertex shader constants.

mapping, which will be covered in more detail in chapter 12. The z component of the transformed vector is then used as the diffuse color of the object on line 31, while the vector itself is used as the texture coordinate for stage 1.

The second shader is used to compute the dot product of the horizon textures and the basis function textures into an additional render target and is stored in the member variable `m_dwBasisShader` in the source code. This shader is very similar to the shadow shader. On lines 34–38 the transformed light vector is biased and scaled before being used as the texture coordinates.

The third shader is used when pixel shaders are not supported as well as when they are supported. It is stored in the member variable `m_dwBumpShader`. This shader is very similar to the basis shader except that the sense of the light vector is reversed before it is transformed on lines 28–30. In other words, the vector that is transformed is from the object to the light, and not from the light to the object.

The fourth vertex shader in the BumpSelfShadow sample is used only when pixel shaders are supported by the device and is shown in listing 9.4. It is stored in the member variable `m_dwShadowShader` in the sample code.

9.21.2 BumpWaves

In this sample, the vertex shader is used to compute the rippling water bump mapping effect when this effect is not supported by the fixed-function pipeline. The shader appears in listing 9.5. Here, the shader computes the texture coordinates from a the vertex position. First, lines 6–8 transform the input position by the composite World and View matrices. Then, lines 12–14 compute the (u, v) texture coordinates by dividing the x and y components by the z component and adding an offset.

9.21.3 DolphinVS

In the DolphinVS sample, vertex shaders are used to perform tweening animations between three poses of the dolphin. Note that this cannot be performed directly with the fixed-function pipeline, which supports tweening between two poses. The sample also provides caustic effects with a second rendering pass. Four vertex shaders are used: two for rendering the dolphin and two for rendering the sea floor. Each pair of shaders provides the effects for the two-pass

Register	Usage
`v0`	Position for dolphin pose 1
`v1`	Position for dolphin pose 2
`v2`	Position for dolphin pose 3
`v3`	Normal for dolphin pose 1
`v4`	Normal for dolphin pose 2
`v5`	Normal for dolphin pose 3
`v6`	Texture coordinates for all poses

Table 9.14: Input registers used by the DolphinVS sample

rendering: one for normal appearance and a second for the caustic effect. We will only discuss the shaders for the dolphin model; the sea floor shaders are similar.

The dolphin is first tweened between the three poses and rendered normally in listing 9.6. The input registers used by the shaders for both passes are given in table 9.14. Lines 24–28 perform the tweening on the input position before it is transformed into homogeneous clip space by line 31. Similarly, the three normals are tweened in lines 41–45. Lines 48–52 perform a simple lighting calculation between the tweened surface normal $\vec{n}$ and the light direction $\vec{l}$ by the formula $A + D\vec{n} \cdot \vec{l}$, where D is the diffuse color and A is the global ambient color. The input texture coordinates are copied directly to the output. The camera space position computed on line 34 is used to compute the fog factor in lines 67–70.

The caustic effect shader is shown in listing 9.7. The only difference from the first shader is that the texture coordinates for stage 0 are computed from the x and z components of the camera space position on line 52.

9.21.4 SkinnedMesh

The SkinnedMesh sample can perform skinning in a variety of methods. When using vertex shader based skinning with four blend matrix indices per vertex, the shader shown in listing 9.8 is used. Three other vertex shaders are used by the sample, corresponding to the case where 1, 2, or 3 blend matrix indices per vertex are used. We will discuss the most complicated case of 4 blend matrix indices per vertex, the other shaders are similar.

The blending of the vertex position and normal are similar to the DolphinVS sample. Each vertex includes three blend weights and four blend matrix indices. The fourth weight is computed by subtracting the sum of the other three weights from 1 in lines 35–36. As mentioned in section 5.8, a card may not support the `UBYTE4` vertex component data type. This shader passes the blend matrix indices as a color, which works on all cards. However, the components of a color are expanded to floating-point values in the range $[0, 1]$, while a `UBYTE4` component would be expanded to integers in the range $[0, 255]$. Line 32 scales the value from the $[0, 1]$ range to the proper range for indexing an array of matrices stored in the constant register file.

The blending of the position and normal is computed in lines 39–72. For each set of blend weight and index, computation is similar. First, the index is moved into the address register and used to access the constant register file to obtain the appropriate transformation matrix. In this sample, the matrices are stored beginning at register `c9`, so the offset of 9 is added to the address register in each case. The position and normal are transformed into temporary registers. The values in the registers are weighted by the appropriate blend weight and added to the running total. (The first set is simply stored to initialize the running total.)

After the weighted sum of the position and normal has been computed, the position is transformed into homogeneous clip space in line 76. Line 75 ensures that the position has a valid w component before being transformed by the 4×4 projection matrix. Finally, the skinned normal is normalized and used in a standard diffuse lighting model to compute the output colors. Texture coordinates are copied through unchanged.

9.21.5 SphereMap

Sphere mapping is a common environment mapping technique, but it is not supported directly by the fixed-function pipeline in Direct3D 8.1. However, sphere mapping can be performed in a vertex shader as is demonstrated by this sample. The details of sphere mapping are discussed in chapter 12, we will only discuss the shader here. The shader appears in listing 9.9. After computing the camera space reflection and view vector, the texture coordinates are computed in lines 27–30.

9.21.6 VertexBlend

The VertexBlend sample in the SDK performs a very simple form of vertex skinning, or blending. The vertex shader used by this sample appears in listing 9.10. The position is transformed by two world matrices and a linear interpolation between the two transformed positions is computed in lines 25–39. Lines 42–45 transform the interpolated position into homogeneous clip space to produce the output position. Similarly, the normals are transformed and interpolated in lines 50–64. The normals are assumed to be normalized in the input data, so they can be used directly in computing the diffuse lighting in lines 67–71. The texture coordinates are copied directly to the output without modification.

9.21.7 VertexShader

The VertexShader sample uses a vertex shader to procedurally displace the input geometry. This sample also demonstrates compressed vertex components by declaring position component as a two floating-point values. Lines 17–20 "decompress" the position component by using the input values as the x and z coordinate of the vertex, while the y and w coordinates of the vertex are taken from a constant register.

Lines 23–35 calculate the value of θ used for the trigonometric functions used to calculate the surface displacement. Lines 38–51 compute the value of $\cos\theta$ and $\sin\theta$. The diffuse color is computed from $\cos\theta$ in lines 54–55 and the value of $\sin\theta$ is used to scale the height of the input vertex in line 58. Finally, the position is transformed into homogeneous clip space as the output position.

Listing 9.1: `BumpShader.vsh`: BumpSelfShadow shader 1.

```
vs.1.0

; Constants
;
; c0-c3 Object
; c4-c7 Projection
; c8-c11 Total matrix
; c12 - Light Direction (In World Space)
;
; Input
;
; V0 - Position
; V7 - Texture
; V3 - Normal
; V8 - Tangnet

; Take normal and binormal into texture space first
m3x3 r7, v8, c0
m3x3 r8, v3, c0

; Cross product orientation flip
; is content dependent
mul r0, r7.zxyw, -r8.yzxw
mad r5, r7.yzxw, -r8.zxyw, -r0

; transform the light vector
dp3 r6.x, r7, c12
dp3 r6.y, r5, c12
dp3 r6.z, r8, c12

mov oD0.xyzw, -r6.z

; this is also our texture coordinate
; on our basis
mul oT1.xyz, -r6.xyz, c33
; mov the z value into all the values of the color

; mov oT1, c33

; transform into projection space
m4x4 oPos, v0, c8
mov oT0.xy, v7
```

Listing 9.2: `BumpShader2.vsh`: BumpSelfShadow shader 2.

```
vs.1.0

; Vertex Shader for DX7 class hardware
; Constants
;
; c0-c3 Object
; c4-c7 Projection
; c8-c11 Total matrix
; c12 - Light Direction (In World Space)
; c33 - .5, .5, .5, .5
;
; Input
;
; V0 - Position
; V7 - Texture
; V3 - Normal
; V8 - Tangnet

; Take normal and binormal into worldspace first
m3x3 r7, v8, c0
m3x3 r8, v3, c0

; Cross product, orientation flip here
; content dependent
mul r0, r7.zxyw, -r8.yzxw
mad r5, r7.yzxw, -r8.zxyw, -r0

; transform the light vector
dp3 r6.x, r7, c12
dp3 r6.y, r5, c12
dp3 r6.z, r8, c12

; bias around 128
add r6.xyz, -r6.xyz, c32

; this is also our texture coordinate
; on our basis
mul oT1.xy, r6.xy, c33

; transform into projection space
m4x4 oPos, v0, c8
mov oT0.xy, v7
```

Listing 9.3: `BumpShader3.vsh`: BumpSelfShadow shader 3.

```
vs.1.0

; Constants
;
; c0-c3 Object
; c4-c7 Projection
; c8-c11 Total matrix
; c12 - Light Direction (In World Space)
;
; Input
;
; V0 - Position
; V7 - Texture
; V3 - Normal
; V8 - Tangnet

; Take normal and binormal into worldspace first
m3x3 r7, v8, c0
m3x3 r8, v3, c0

; Cross product, flip orienation
; may or may not be neccisary here
; depending on the content
mul r0, r7.zxyw, -r8.yzxw
mad r5, r7.yzxw, -r8.zxyw, -r0

; transform the light vector
dp3 r6.x, r7, -c12
dp3 r6.y, r5, -c12
dp3 r6.z, r8, -c12

; bias around 128
add r6.xyz, r6.xyz, c32
mul oD0.xyz, r6.xyz, c33

; transform into projection space
m4x4 oPos, v0, c8
mov oT0.xy, v7
mov oT1.xy, v7
```

Listing 9.4: `BumpShader4.vsh`: BumpSelfShadow shader 4.

```
vs.1.0

; Constants
;
; c0-c3 Object
; c4-c7 Projection
; c8-c11 Total matrix
; c12 - Light Direction (In World Space)
;
; Input
;
; V0 - Position
; V7 - Texture
; V3 - Normal
; V8 - Tangnet

; Take normal and tangnet into texture space first
m3x3 r7, v8, c0
m3x3 r8, v3, c0

; Cross product
mul r0, r7.zxyw, -r8.yzxw
mad r5, r7.yzxw, -r8.zxyw, -r0

; transform the light vector
dp3 r6.x, r7, c12
dp3 r6.y, r5, c12
dp3 r6.z, r8, c12

; bias around 128
mad r6.xyz, -r6.xyz, c33, c33

; this is also our texture coordinate
; on our basis
mov oT1.xy, r6
mov oT3.xy, r6

mov oD0.xyzw, r6.z

; transform into projection space
m4x4 oPos, v0, c8
mov oT0.xy, v7
mov oT2.xy, v7
```

Listing 9.5: BumpWaves vertex shader.

```
vs.1.1
m4x4 oPos, v0, c3 ; transform position to the projection space

; Compute vertex position in the camera space
; - this is our texture coordinates
dp4 r0.x, v0, c0
dp4 r0.y, v0, c1
dp4 r0.z, v0, c2

; Do the rest of texture transform (first part was combined
; with the camera matrix)
rcp r0.z, r0.z
mad oT1.x, r0.x, r0.z, c8.x
mad oT1.y, r0.y, r0.z, c8.y

; Copy input texture coordinates for the stage 0
mov oT0.xy, v1
```

Listing 9.6: `DolphinTween.vsh`: DolphinVS shader 1.

```
;
; Constants specified by the app
;    c0      = ( 0, 0, 0, 0 )
;    c1      = ( 1, 0.5, 2, 4 )
;    c2      = ( fWeight1, fWeight2, fWeight3, 0 )
;    c4-c7   = matWorldViewProjection
;    c8-c11  = matWorldView
;    c19     = light direction (in model space)
;    c21     = material diffuse color * light diffuse color
;    c22     = material ambient color
;
; Vertex components (as specified in the vertex DECL)
;    v0    = Position
;    v3    = Normal
;    v6    = Texcoords
;
vs.1.1

;
; Vertex transformation
;

; Tween the 3 positions (v0, v1, v2) into one position
mul r0, v0, c2.x
mul r1, v1, c2.y
mul r2, v2, c2.z
add r3, r0, r1
add r3, r3, r2

; Transform position to the clipping space
m4x4 oPos, r3, c4

; Transform position to the camera space
m4x4 r9, r3, c8

;
; Lighting calculation
;

; Tween the 3 normals (v3, v4, v5) into one normal
mul r0, v3, c2.x
mul r1, v4, c2.y
mul r2, v5, c2.z
add r3, r0, r1
add r3, r3, r2

; Do the lighting calculation
dp3 r1.x, r3, c19    ; r1 = normal dot light
max r1.x, r1.x, c0.x   ; if dot < 0 then dot = 0
mul r0, r1.x, c21    ; Multiply with diffuse
add r0, r0, c22      ; Add in ambient
min oD0, r0, c1.x    ; clamp if > 1

;
; Texture coordinates
;

; Copy tex coords
mov oT0.xy, v6

;
; Fog calculation
;

; compute fog factor
; f = (fog_end - dist)*(1/(fog_end-fog_start))
add r0.x, -r9.z, c23.y
mul r0.x, r0.x, c23.z
max r0.x, r0.x, c0.x       ; clamp fog to > 0.0
min oFog.x, r0.x, c1.x     ; clamp fog to < 1.0
```

Listing 9.7: `DolphinTween2.vsh`: DolphinVS shader 2.

```
; Constants specified by the app
;    c0      = ( 0, 0, 0, 0 )
;    c1      = ( 1, 0.5, 2, 4 )
;    c2      = ( fWeight1, fWeight2, fWeight3, 0 )
;    c4-c7   = matWorldViewProjection
;    c8-c11  = matWorldView
;    c19     = light direction (in model space)
;    c21     = material diffuse color * light diffuse color
;    c22     = material ambient color
;
; Vertex components (as specified in the vertex DECL)
;    v0    = Position
;    v3    = Normal
;    v6    = Texcoords
;
vs.1.1

; Vertex transformation
;
; Tween the 3 positions (v0, v1, v2) into one position
mul r0, v0, c2.x
mul r1, v1, c2.y
mul r2, v2, c2.z
add r3, r0, r1
add r3, r3, r2

; Transform position to the clipping space
m4x4 oPos, r3, c4

; Transform position to the camera space
m4x4 r9, r3, c8

; Lighting calculation
;
; Tween the 3 normals (v3, v4, v5) into one normal
mul r0, v3, c2.x
mul r1, v4, c2.y
mul r2, v5, c2.z
add r3, r0, r1
add r3, r3, r2

; Do the lighting calculation
dp3 r1.x, r3, c19    ; r1 = normal dot light
max r1.x, r1.x, c0.x   ; if dot < 0 then dot = 0
```

```
mul r0, r1.x, c21    ; Multiply with diffuse
add r0, r0, c22      ; Add in ambient
min oD0, r0, c1.x    ; clamp if > 1

; Texture coordinates
;
; Gen tex coords from vertex xz position
mul oT0.xy, c1.y, r9.xz

; Fog calculation
;
; compute fog factor
; f = (fog_end - dist)*(1/(fog_end-fog_start))
add r0.x, -r9.z, c23.y
mul r0.x, r0.x, c23.z
max r0.x, r0.x, c0.x       ; clamp fog to > 0.0
min oFog.x, r0.x, c1.x     ; clamp fog to < 1.0
```

Listing 9.8: `SkinnedMesh4.vsh`: SkinnedMesh shader 4.

```
vs.1.1
;
; v0 = position
; v1 = blend weights
; v2 = blend indices
; v3 = normal
; v4 = texture coordinates
;
; r0.w = Last blend weight
; r1 = Blend indices
; r2 = Temp position
; r3 = Temp normal
; r4 = Blended position in camera space
; r5 = Blended normal in camera space
;
; Constants specified by the app:
;
; c9-c95 = world-view matrix palette
; c8     = diffuse * light.diffuse
; c7     = ambient color
; c2-c5  = projection matrix
; c1     = light direction
; c0     = {1, power, 0, 1020.01}
;
; oPos   = Output position
; oD0    = Diffuse
; oD1    = Specular
; oT0    = texture coordinates
;

// Compensate for lack of UBYTE4 on Geforce3
mul r1, v2.zyxw, c0.wwww

// first compute the last blending weight
dp3 r0.w, v1.xyz, c0.xxx;
add r0.w, -r0.w, c0.x

// Set 1
mov a0.x, r1.x
m4x3 r4, v0, c[a0.x + 9]
m3x3 r5, v3, c[a0.x + 9];

// blend them
mul r4, r4, v1.xxxx
mul r5, r5, v1.xxxx

// Set 2
mov a0.x, r1.y
m4x3 r2, v0, c[a0.x + 9]
m3x3 r3, v3, c[a0.x + 9]

// add them in
mad r4, r2, v1.yyyy, r4
mad r5, r3, v1.yyyy, r5

// Set 3
mov a0.x, r1.z
m4x3 r2, v0, c[a0.x + 9]
m3x3 r3, v3, c[a0.x + 9]

// add them in
mad r4, r2, v1.zzzz, r4
mad r5, r3, v1.zzzz, r5

// Set 4
mov a0.x, r1.w
m4x3 r2, v0, c[a0.x + 9]
m3x3 r3, v3, c[a0.x + 9]

// add them in
mad r4, r2, r0.wwww, r4
mad r5, r3, r0.wwww, r5

// compute position
mov r4.w, c0.x
m4x4 oPos, r4, c2

// normalize normals
dp3 r5.w, r5, r5
rsq r5.w, r5.w
mul r5, r5, r5.w

; Do the lighting calculation
dp3 r1.x, r5, c1      ; normal dot light
lit r1, r1
mul r0, r1.y, c8      ; Multiply with diffuse
add r0, r0, c7        ; Add in ambient
min oD0, r0, c0.x     ; clamp if > 1
mov oD1, c0.zzzz     ; output specular

```

```
; Copy texture coordinate
mov oT0, v4
```

Listing 9.9: SphereMap shader

```
vs.1.0
def c64, 0.25f, 0.5f, 1.0f, -1.0f

// r0: camera-space position
// r1: camera-space normal
// r2: camera-space vertex-eye vector
// r3: camera-space reflection vector
// r4: texture coordinates

// Transform position and normal into camera-space
m4x4 r0, v0, c0
m3x3 r1, v1, c0

// Compute normalized view vector
mov r2, -r0
dp3 r3, r2, r2
rsq r3, r3
mul r2, r2, r3

// Compute camera-space reflection vector
dp3 r3, r1, r2
mul r1, r1, r3
add r1, r1, r1
add r3, r1, -r2

// Compute sphere-map texture coords
mad r4.w, -r3.z, c64.y, c64.y
rsq r4, r4
mul r4, r3, r4
mad r4, r4, c64.x, c64.y

// Project position
m4x4 oPos, r0, c4
mul oT0.xy, r4.xy, c64.zw
mov oT0.zw, c64.z
```

Listing 9.10: `Blend.vsh`: VertexBlend shader

```
vs_1_1
//
// Constants specified by the app
//    c0      = (0, 0, 0, 0)
//    c1      = (1, 1, 1, 1)
//    c2      = (0, 1, 2, 3)
//    c3      = (4, 5, 6, 7)
//    c4-c7   = matWorld0
//    c8-c11  = matWorld1
//    c12-c15 = matViewProj
//    c20     = light direction
//    c21     = material diffuse color * light diffuse color
//    c22     = material ambient color
//
// Vertex components (as specified in the vertex DECL)
//    v0    = Position
//    v1.x  = Blend weight
//    v3    = Normal
//    v7    = Texcoords
//

dcl_position v0
dcl_blendweight v1
dcl_normal v3
dcl_texcoord v7

// Vertex blending
//
// Transform position for world0 matrix
dp4 r0.x, v0, c4
dp4 r0.y, v0, c5
dp4 r0.z, v0, c6
dp4 r0.w, v0, c7

// Transform position for world1 matrix
dp4 r1.x, v0, c8
dp4 r1.y, v0, c9
dp4 r1.z, v0, c10
dp4 r1.w, v0, c11

// Lerp the two positions r0 and r1 into r2
mul r0, r0, v1.x     // v0 * weight
add r2, c1.x, -v1.x  // r2 = 1 - weight
mad r2, r1, r2, r0   // pos = (1-weight)*v1 + v0*weight
```

```

// Transform to projection space
dp4 oPos.x, r2, c12
dp4 oPos.y, r2, c13
dp4 oPos.z, r2, c14
dp4 oPos.w, r2, c15

// Lighting calculation
//
// Transform normal for world0 matrix
dp4 r0.x, v3, c4
dp4 r0.y, v3, c5
dp4 r0.z, v3, c6
dp4 r0.w, v3, c7

// Transform normal for world1 matrix
dp4 r1.x, v3, c8
dp4 r1.y, v3, c9
dp4 r1.z, v3, c10
dp4 r1.w, v3, c11

// Lerp the two normals r0 and r1 into r2
mul r0, r0, v1.x     // v0 * weight
add r2, c1.x, -v1.x  // r2 = 1 - weight
mad r2, r1, r2, r0   // normal = (1-weight)*v1 + v0*weight

// Do the lighting calculation
dp3 r1.x, r2, c20    // r1 = normal dot light
max r1, r1.x, c0     // if dot < 0 then dot = 0
mul r0, r1.x, c21    // Multiply with diffuse
add r0, r0, c22      // Add in ambient
min oD0, r0, c1.x    // clamp if > 1

// Texture coordinates
//
// Just copy the texture coordinates
mov oT0,  v7
```

Listing 9.11: `Ripple.vsh`: VertexShader shader

```
vs.1.0
; Constants:
;
;  c0-c3  - View+Projection matrix
;  c4.x   - time
;  c4.y   - 0
;  c4.z   - 0.5
;  c4.w   - 1.0
;  c7.x   - pi
;  c7.y   - 1/2pi
;  c7.z   - 2pi
;  c7.w   - 0.05
;  c10    - first 4 taylor coefficients for sin(x)
;  c11    - first 4 taylor coefficients for cos(x)

; Decompress position
mov r0.x, v0.x
mov r0.y, c4.w       ; 1
mov r0.z, v0.y
mov r0.w, c4.w       ; 1

; Compute theta from distance and time
mov r4.xz, r0        ; xz
mov r4.y, c4.y       ; y = 0
dp3 r4.x, r4, r4     ; d2
rsq r4.x, r4.x
rcp r4.x, r4.x       ; d
mul r4.xyz, r4, c4.x     ; scale by time

; Clamp theta to -pi..pi
add r4.x, r4.x, c7.x
mul r4.x, r4.x, c7.y
frc r4.xy, r4.x
mul r4.x, r4.x, c7.z
add r4.x, r4.x, -c7.x

; Compute first 4 values in sin and cos series
mov r5.x, c4.w       ; d^0
mov r4.x, r4.x       ; d^1
mul r5.y, r4.x, r4.x ; d^2
mul r4.y, r4.x, r5.y ; d^3
mul r5.z, r5.y, r5.y ; d^4
mul r4.z, r4.x, r5.z ; d^5
mul r5.w, r5.y, r5.z ; d^6
mul r4.w, r4.x, r5.w ; d^7

mul r4, r4, c10      ; sin
dp4 r4.x, r4, c4.w

mul r5, r5, c11      ; cos
dp4 r5.x, r5, c4.w

; Set color
add r5.x, -r5.x, c4.w ; + 1.0
mul oD0, r5.x, c4.z   ; * 0.5

; Scale height
mul r0.y, r4.x, c7.w

; Transform position
dp4 oPos.x, r0, c0
dp4 oPos.y, r0, c1
dp4 oPos.z, r0, c2
dp4 oPos.w, r0, c3
```

Listing 9.12: FogShader.vsh: VolumeFog shader 1

```
vs_1_1
def     c40, 0.0f, 0.0f, 0.0f, 0.0f;

// transform into projection space
m4x4    r0, v0, c8
max     r0.z, c40.z, r0.z //clamp to 0
max     r0.w, c12.x, r0.w //clamp to near clip plane
mov     oPos, r0
add     r0.w, r0.w, -c12.x
mul     r0.w, r0.w, c12.y

// Load into diffuse
mov     oD0.xyzw, r0.w

// load into texture
mov     oT0.x, r0.w
mov     oT0.y, c12.x
```