Vertex Shader Introduction

Chris Maughan (cmaughan@nvidia.com) and Matthias Wloka (mwloka@nvidia.com)

NVIDIA Corporation

Introduction

The concept of shaders in the graphics industry is nothing new. Pixar, for example, has been using them for years to create their amazing films ("Luxo Jr.," "Geri's Game," "Toy Story," "A Bug's Life," etc.). The basic concept is pretty simple – run a program for each fragment in the graphics pipeline to generate the required color and position for rendering the final image.

DirectX8 (DX8), Microsoft's next-generation API, while some way off from being able to render "A Bug's Life," gives the first nod in the direction of programmable graphics hardware. Because gamers expect real-time performance from their games, moving to a completely programmable graphics solution is impractical – at least for now.

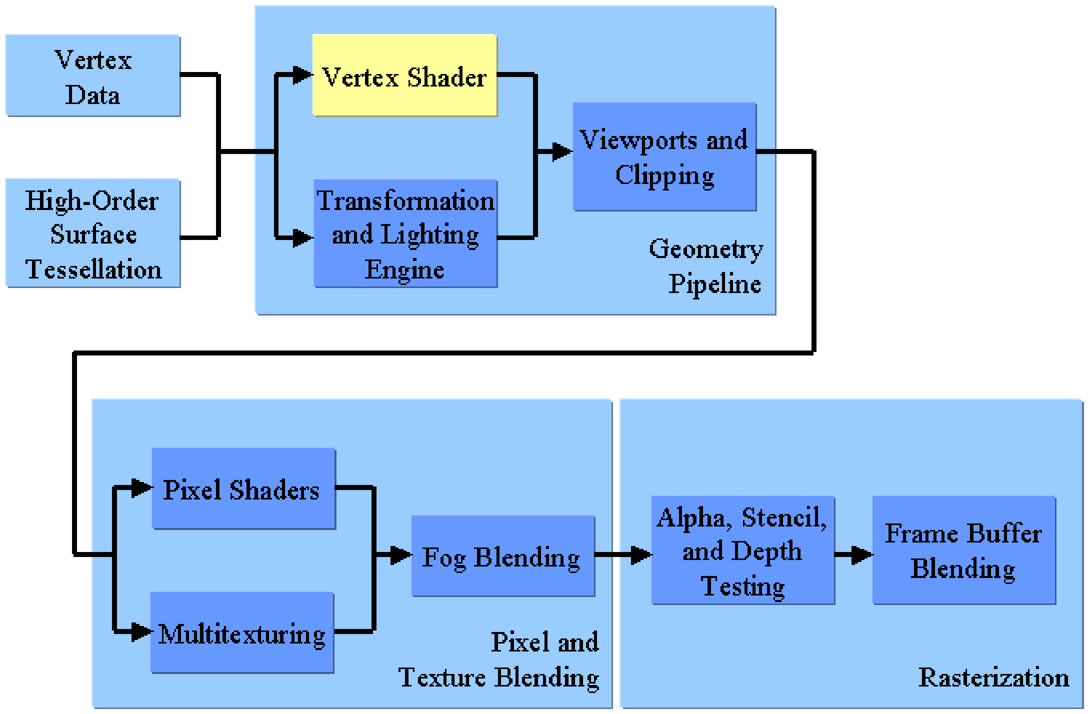
DX8's solution is to enable programmability in two critical areas, while leaving the rest of the graphics pipeline running as it always has. All video memory access, texture reads, polygon management, and many other complex operations in the graphics-processing unit (GPU) are fixed-function, i.e., non-programmable, just as they were in DirectX7, retaining the giga-texel fill-rates gamers have come to expect. The only two sections of the 3D pipeline that are now programmable are 'vertex shaders' and 'pixel shaders'. This article only covers the vertex-shader section of the DX8 pipeline and does not cover the pixel-shader operations.

Vertex shaders enable developers to get their 'hands dirty' and directly control the operations the GPU performs for each vertex; in essence, a vertex shader replaces the per-vertex T&L section of the pipeline. Previously, developing applications involved setting rendering-states, thus controlling how the GPU performs operations, but also restricting the possible effects that are achievable. Vertex shaders allow full access to the vertex pipeline and enable effects that were previously impossible.

This article explains programmable vertex shaders, provides an overview of the achievable effects, and shows how the programmable vertex shader integrates with the rest of the pipeline – and to some extent the pixel shaders further down the graphics pipeline. It not only shows how to achieve the new effects which vertex shaders offer, but also how to achieve well-known effects, such as point lights or cubic reflection mapping.

From unusual lighting models to fish-eye distortions, vertex shaders offer previously unimagined possibilities, enabling a whole new generation of games with exciting effects.

What Is a Vertex Shader?

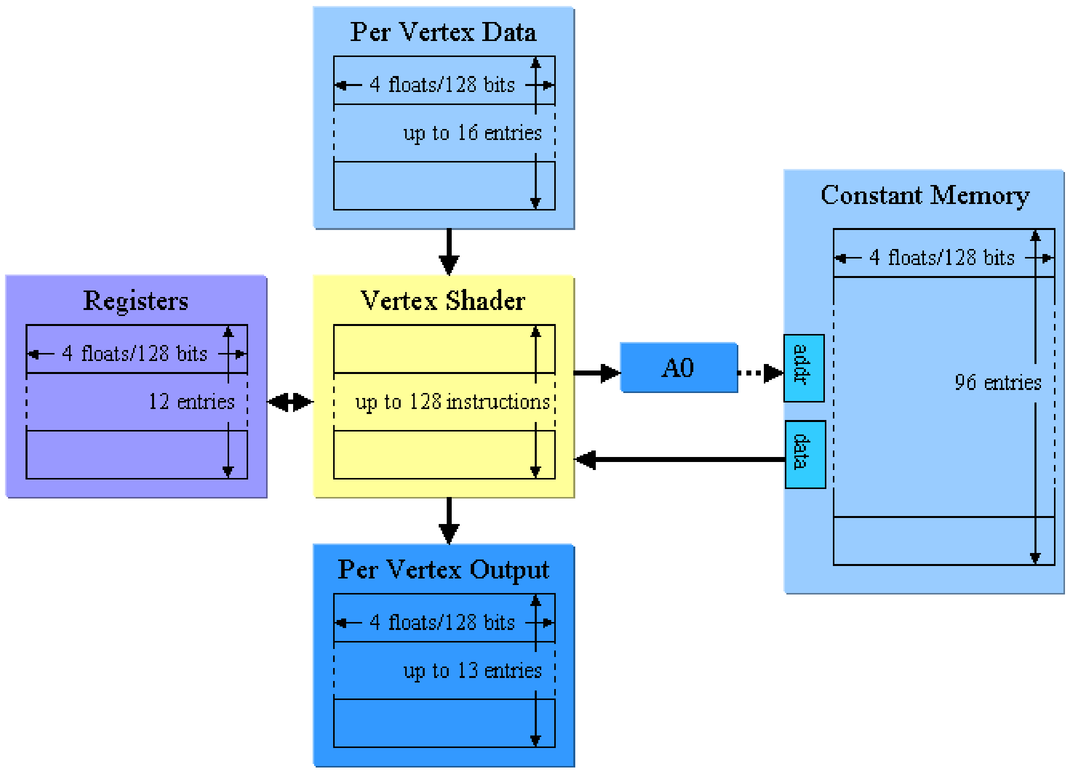


A vertex shader is a small assembly-language program. When enabled, it replaces the transform- and lighting-computations of the fixed-function pipeline for vertices (see Figure 1). Because it replaces the fixed-function, it is responsible for computing all desired effects, including, but not limited to, transforming vertex coordinates to clip-space, lighting, and generating texture-coordinates.

A vertex shader operates on a single vertex at a time. It does not operate on a primitive, such as a triangle. It is unable to generate additional or new vertices. Vertex shaders do apply, however, to the vertices that the higher-order surfaces tessellation generates in the GPU (see Figure 1).

The computation-model of vertex shaders is straightforward. For every vertex to be processed, the vertex shader executes its program. The instructions of that program have access to four different types of memory locations: the per-vertex data of an incoming vertex, “constant memory,” temporary registers, and per-vertex output-registers (see Figure 2). The following section (The Ins, Outs, and Registers) describes these in more detail.

This computation-model is stateless: the execution-results of a vertex shader depend only on the instructions executed, the incoming per-vertex data, and constant memory (which a vertex shader cannot write to). Vertex shaders therefore cannot generate data



that persists longer than their execution. In particular, they cannot generate data to be used when processing the next vertex, or the next time the same vertex is processed.

Regardless, vertex shaders are general enough to be able to express all the functionality (and more, see examples below) of the fixed-function vertex-pipeline. Indeed, NVidia has published the outline of a vertex shader that could be used to fully emulate all functionality of the fixed-function vertex-pipeline (see **Vertex Programs for the Fixed Function Pipeline** on http://www.nvidia.com/Marketing/Developer/DevRel.nsf/TechnicalPresentationsFrame?OpenPage).

And not to worry, vertex-shader performance is on par with the fixed-function vertex-pipeline: a vertex shader emulating fixed-function functionality runs just as fast as the fixed-function pipeline itself.

If using only fixed-function functionality, however, it is less work to simply employ the fixed-function pipeline: it is still available in DX8. Switching between the fixed-function vertex-pipeline and a vertex shader, or indeed between different vertex shaders, is possible, although there is some overhead associated with it. This overhead is smaller than, say texture-changes, and indeed is small enough to switch shaders on a per-object basis.

Finally, only a single vertex shader (or only the fixed-function vertex-pipeline) is active at one time. Combining vertex shaders to, say, have one compute the transform and the next one to compute the lighting of a vertex is impossible: the currently active vertex shader must compute all required per-vertex output-data.

The Ins, Outs, and Registers

As mentioned above, a vertex shader accesses up to four different types of memory locations: the per-vertex data of an incoming vertex, constant memory, temporary registers, and per-vertex output-registers. Each one of these memory locations stores a collection of four-dimensional vectors; each component of such a vector is a 32bit, signed floating-point number. These float-vectors represent a combination of positional data (xyzw), texture coordinates (uvwq), colors (rgba), or simply a collection of four scalars (abcd).

The per-vertex data associated with a vertex are read-only to a vertex shader. They typically contain per-vertex data such as model-space vertex-coordinates, vertex color, and texture coordinates. A vertex may contain up to 16 vector-registers, named `v0` thru `v15`. Each register is a one-, two-, three-, or four-dimensional vector of floats. The application declares to a vertex shader which per-vertex registers are available and of what type these data are. Thus, the runtime generates an error if a vertex shader refers to an undeclared per-vertex vector-register.

Constant memory is also read-only to vertex shaders. The application, however, is able to write to constant memory. Thus, constant memory typically stores world and per-object data that changes per-frame and/or per-object, for example, light properties, transformation matrices, or material properties. Constant memory contains up to 96 four-dimensional float-vectors.

Temporary registers are read/write. Vertex shaders use them to store all temporary results during execution. All temporary registers are reset to initial values before a vertex shader starts execution. There are 12 general-purpose registers, `r0-r11`. Each register is a vector of four floats.

The address register, `a0`, is an additional temporary register that is use/write only. It offsets data-reads from the constant memory area, i.e., it is an indirect addressing operator. This address register is scalar only, i.e., `a0.x` is the only valid component. A vertex shader may write `a0.x` only via the `mov` instruction. It is unreadable, and its use is restricted to offsetting constant memory, e.g., using `c[a0.x + 3]`.

Vertex shaders write the final results of their computation to the per-vertex output-registers. Clip-space vertex-coordinates, diffuse and specular color, and texture coordinates are typical outputs. These per-vertex output registers are write-only and feed into the rest of the graphics pipeline. The output registers are named (see Table 2) to suggest their use—when using pixel shaders, however, this use-pattern may be bent.

The output-register `oPos` specifies a vertex's clip-space position. It is special in that every vertex shader must write to `oPos`. While the pipeline explicitly clips `oPos` to the view-frustum (if clipping is enabled), several of the other output registers clamp implicitly: each vector-component of `oD0` and `oD1` clamps to between zero and one, inclusively. Similarly, `oFog.x` and `oPts.x` clamp to [0.0, 1.0] and [`D3DRS_POINTSIZE_MIN`, `D3DRS_POINTSIZE_MAX`], respectively. While for `oFog` and `oPts` only the first component is functional, i.e., they are scalar, all other output-registers are four-dimensional float-vectors.

Tables 1-3 summarize the available memory locations. The “Format” column specifies whether a memory location is a four-dimensional float-vector or scalar. “Access” describes the allowed access pattern for vertex shaders, while “Written by” shows what entity has write access to that type of memory. A single instruction of a vertex shader may access only a limited number of different memory locations; for example, it can refer to at most one constant memory location. Thus, `add r0, c[0].x, c[1].y` is illegal, while `add r0, c[0].x, c[0].y` is legal since it refers to exactly one constant memory location (see also Section “The Instruction Set”). The column “Max. Refs/Instr.” reveals that limit. Finally, the “Typical Use” column describes the typical use of a memory location.

Name	Format	Access	Written by	Max. Refs/Instr	Typical Use
`V0-v15`	vector	Read Only	Application/ Vertex-Stream	1	Per-vertex data as specified in the vertex-shader declaration.
`C[0]-c[95]`	vector	Read Only	Application	1	Constant data specified by the application.

Table 1: Vertex Shader Inputs

Name	Format	Access	Written by	Typical Use
`oPos`	vector	Write Only	Vertex Shader	Transformed clip-space coordinates of vertex.
`oD0`	vector	Write Only	Vertex Shader	Diffuse color component of vertex.
`oD1`	vector	Write Only	Vertex Shader	Specular color component of vertex.
`oT0–oT3`	vector	Write Only	Vertex Shader	Texture coordinates of vertex for texture stages 0 through 3.
`oFog.x`	scalar	Write Only	Vertex Shader	Fog value for vertex.
`oPts.x`	scalar	Write Only	Vertex Shader	Sprite-size for vertex.

Table 2: Vertex Shader Outputs

Name	Format	Access	Written by	Max. Refs/Instr	Typical Use
`R0-r11`	vector	Read/Write	Vertex Shader	3	Temporary registers
`A0.x`	scalar	Use /Write	Vertex Shader	1	Indirect addressing

Table 3: Vertex Shader Temporary Registers

The Instruction Set

A vertex-shader program consists of up to 128 instructions. The size of the instruction set is 17. While the length of a vertex-shader program and its instruction-set size may appear small, keep in mind that every instruction operates on four-dimensional float-vectors, and that the instruction set is specialized for graphics. As a result, vertex shaders are powerful enough to emulate the fixed-function vertex-transform and -lighting in its entirety plus then some.

The instruction set contains no branching, jumping, or looping instructions. Similarly, there is no early exit. That means that every instruction in a vertex-shader program executes exactly once for every vertex that the vertex shader processes.

If-then-else constructs are nonetheless possible. For example, a vertex shader would first compute both the "then-" as well as the "else-result," then generate 1.0 or 0.0 according to the "if-test," and multiply the "then-result" with the outcome of the test-result and multiply the "else-result" with (1.0 – test-result). Finally, adding the results of those multiplications produces either the "then-result" or the "else-result." This trick and many others like it or described in detail in the document "Where is That Instruction" (see **Implementation of "Missing" Vertex Shader Instructions** on http://www.nvidia.com/Marketing/Developer/DevRel.nsf/WhitepapersFrame?OpenPage).

Table 4 (Vertex Shader Instructions) lists all instructions and describes their effects. While DX8 also supports instruction macros, these macros do not appear in Table 4: macros tend to obscure optimization possibilities and their use is thus inadvisable.

The instructions `dst` and `lit` are graphics-specific and thus warrant further explanation. The instruction `dst` typically computes an attenuation vector `(1, d, d*d, 1/d)` from inputs `(n/a, d*d, d*d, n/a)` and `(n/a, 1/d, n/a, 1/d)`. The `lit` instruction is handy for lighting calculations as it computes specular power and clamps some components to zero if the argument's .x component is negative.

Every instruction may also use modifiers on all its destination- and source-registers. The destination-register modifier is a write mask, while the source-argument modifier is an arbitrary component swizzle with an optional negation.

The destination-register write-mask lists all vector-components of the destination register that the current instruction overwrites. Not specifying any write-mask at all, however, is equivalent to .xyzw, i.e., overwrites all four vector-components. Always using the destination write-mask is commendable (unless using all four vector-components of an instruction-result), since the destination mask clarifies which instruction-results are (and more importantly, are not) subsequently used, i.e., it is an effective form of code documentation. Furthermore, the use of destination masks may make optimization opportunities more apparent, both to the author and the driver.

The source-argument modifier is an arbitrary component swizzle. Any vector component (x, y, z, or w) may swizzle and/or replicate into any other component. Not specifying a source-argument modifier is equivalent to .xyzw, i.e., the identity mapping. Specifying .x (.y, .z, .w) is the same as specifying .xxxx (.yyyy, .zzzz, .wwww, respectively), i.e., replicating the x-component (y-, z-, w-component) into all four components. All 256 permutations are legal. Negating all four components of the result of the source-argument swizzle is optional. Table 5 (Vertex-Shader Instruction-Modifiers) lists all possible modifiers.

Every instruction executes in a single tick. Thus, vertex-shader performance is directly proportional to the number of instructions of a vertex shader: a vertex shader that is half the length of another vertex shader executes in half the time. Performance tuning for vertex shader therefore consists of eliminating unnecessary instructions. The `dst` and `lit` instructions (see above) are helpful in that task. Taking advantage of the vector-nature of all instructions and making heavy use of the source-argument swizzles also help in paring instructions. The questioning of every use of any `mov` instructions in a vertex shader may further reduce vertex-shader program-length. Check the document "Where is That Instruction" for more ideas (see **Implementation of "Missing" Vertex Shader Instructions** on http://www.nvidia.com/Marketing/Developer/DevRel.nsf/WhitepapersFrame?OpenPage).

Table 4: Vertex-Shader Instructions

Mnemonic	Description

Add d, s0, s1	d.x = s0.x + s1.x
	d.y = s0.y + s1.y
	d.z = s0.z + s1.z
	d.w = s0.w + s1.w

Dp3 d, s0, s1	d.x = d.y = d.z = d.w = s0.x*s1.x +s0.y*s1.y + s0.z*s1.z
Dp4 d, s0, s1	d.x = d.y = d.z = d.w = s0.x*s1.x +s0.y*s1.y + s0.z*s1.z + s0.w*s1.w

Dst d, s0, s1	d.x = 1;
	d.y = s0.y * s1.y
	d.z = s0.z
	d.w = s1.w

Expp d, s0	d.x = 2^floor(s0.w)
	d.y = s0.w – floor(s0.w)
	d.z = 2^(s0.w)
	d.w = 1

Lit d, s0	d.x = 1
	d.y = (s0.x > 0) ? S0.x : 0
	d.z = (s0.x > 0 && s0.y > 0) ? s0.y ^ s0.w : 0
	d.w = 1

Logp d, s0	d.x = d.y = d.z = d.w = (s0.w != 0) ? log(abs(s0.w))/log(2) : -∞

Mad d, s0, s1, s2	d.x = s0.x*s1.x + s2.x
	d.y = s0.y*s1.y + s2.y
	d.z = s0.z*s1.z + s2.z
	d.w = s0.w*s1.w + s2.w

Max d, s0, s1	d.x = (s0.x >= s1.x) ? s0.x : s1.x
	d.y = (s0.y >= s1.y) ? s0.y : s1.y
	d.z = (s0.z >= s1.z) ? s0.z : s1.z
	d.w = (s0.w >= s1.w) ? s0.w : s1.w

Min d, s0, s1	d.x = (s0.x < s1.x) ? s0.x : s1.x
	d.y = (s0.y < s1.y) ? s0.y : s1.y
	d.z = (s0.z < s1.z) ? s0.z : s1.z
	d.w = (s0.w < s1.w) ? s0.w : s1.w

Mov d, s0	d.x = s0.x
	d.y = s0.y
	d.z = s0.z
	d.w = s0.w

Mul d, s0, s1	d.x = s0.x * s1.x
	d.y = s0.y * s1.y
	d.z = s0.z * s1.z
	d.w = s0.w * s1.w

Nop	
Rcp d, s0	d.x = d.y = d.z = d.w = (s0.w == 0) ? ∞ : 1/s0.w
Rsq d, s0	d.x = d.y = d.z = d.w = (s0.w == 0) ? ∞ : 1/sqrt(abs(s0.w))

Sge d, s0, s1	d.x = (s0.x >= s1.x) ? 1 : 0
	d.y = (s0.y >= s1.y) ? 1 : 0
	d.z = (s0.z >= s1.z) ? 1 : 0
	d.w = (s0.w >= s1.w) ? 1 : 0

Slt d, s0, s1	d.x = (s0.x < s1.x) ? 1 : 0
	d.y = (s0.y < s1.y) ? 1 : 0
	d.z = (s0.z < s1.z) ? 1 : 0
	d.w = (s0.w < s1.w) ? 1 : 0

Sub d, s0, s1	d.x = s0.x - s1.x
	d.y = s0.y - s1.y
	d.z = s0.z - s1.z
	d.w = s0.w - s1.w

Table 5: Vertex-Shader Instruction-Modifiers

Name	Format	Description
Destination Mask	Dmask = .{x}{y}{z}{w}	If x ε Dmask then write x-component of destination vector If y ε Dmask then write y-component of destination vector If z ε Dmask then write z-component of destination vector If w ε Dmask then write w-component of destination vector
Source Swizzle	.[xyzw][xyzw][xyzw][xyzw]	The x, y, z, and w components of the input swizzle and/or replicate as indicated.
Source Negation	- s	Negates source operand for operation.

Enough Theory: Creating and Using Vertex Shaders

Let's get our hands dirty. All the following code-snippets assume familiarity with C++ and DX8. To create and apply a vertex shader one needs to 1) create a vertex-shading program, 2) compile the vertex-shading program into a vertex-shader object-file, 3) create a vertex shader in the application from the vertex-shader object-file and a matching per-vertex data-declaration, 4) switch to that newly created vertex shader, 5) write constants to constant memory, and 6) create primitives that conform to the expected vertex-shading format and draw them.

First, with any text-editor one creates a vertex-shading program using the instructions explained in the above Section "The Instruction Set." Listing1_SimpleVertexShader.nvv is one such vertex-shading program. All comments start with the ';' character—C++-style '//' comments and other niceties are available when using NVidia's vertex-shader assembler (see below).

The first line of any vertex shader that is not a comment has to declare the vertex-shader version that is in use. Both `vs.1.0` and `vs.1.1` are legal. Version 1.1 is a strict superset of version 1.0 and additionally includes the availability of the address register a0 (see Section "The Ins, Outs, and Registers").

Other than the comments, all following lines are vertex instructions, one instruction per line. The first four instructions perform a 4x4-matrix multiplication. Each `dp4` instruction computes one of the result-vector's components and stores its result -- the vertex's clip-space position -- directly into the per-vertex output `oPos`.

The next two instructions compute the per-vertex diffuse lighting. The instruction `dp3` computes a three-dimensional dot product between a vertex's normal and a light-direction from constant memory. Multiplying this dot product with a material color yields the diffuse color of this vertex. The program stores this diffuse vertex-color directly in the diffuse color output of the vertex shader, `oD0`. Even though the multiplication result may lie outside the valid interval [0.0, 1.0], explicitly clamping the result is unnecessary as `oD0` automatically clamps each of its components to [0.0, 1.0] (see "The Ins, Outs, and Registers").

The final instruction of this vertex shader simply copies the per-vertex texture-coordinates to the first texture stage.

To create a vertex-shader object-file from this vertex-shader program, one must compile it with a vertex-shader assembler. While run-time assembly is possible (see D3DXAssembleShaderFromFileW()), compile-time assembly is more reasonable. Microsoft's vertex-shader assembler vsa.exe comes with the DX8 Software development kit. Alternatively, NVidia's vertex-shader assembler nvasm.exe provides all features of vsa.exe plus many other niceties, such as more detailed error-messages, full Visual Studio integration, and a C++-style preprocessor (thus supporting '//' comments, #include, #define etc.). It is available for free (**NVASM** at http://www.nvidia.com/Marketing/Developer/DevRel.nsf/ProgrammingResourcesFrame?OpenPage).

The next step is to create a vertex shader in the application. The file Listing2_SetUp.cpp provides source code for this step. It first opens and reads the vertex-shader object-file into memory. Then it declares the per-vertex inputs that the vertex shader expects.

The use of STL-vectors simplifies the construction of this declaration. The first declaration token (“`D3DVSD_STREAM(0)`”) announces the data-source of the per-vertex inputs. In this case, all data comes from vertex-stream zero. The second token (“`D3DVSD_REG(0, D3DVSDT_FLOAT3)`”) specifies that the first three floats in the vertex-stream map to `v0`; they constitute the vertex’s model-space position for this vertex shader. To assign the first three floats in the vertex-stream to an arbitrary per-vertex input `vN`, specify the token `D3DVSD_REG(N, D3DVSDT_FLOAT3).`

The following two tokens specify that the next three floats in the vertex stream copy into `v1` and that the last two floats copy into `v2`. The use of `v0`, `v1`, and `v2` is arbitrary: any per-vertex input is usable instead; the per-vertex inputs must, however, match the vertex-shader program (see Listing1_SimpleVertexShader.nvv). The final token of all vertex-shader declarations is “`D3DVSD_END()`.”

Since the per-vertex inputs are four-dimensional float-vectors, the above declarations leave some of the vector’s component undefined. Floats always map first into the x-component, then the y-, z-, and w-components of a vector. Since at least one float always maps to a per-vertex input, the x-component is always defined. If the y- or z-components are undefined their values default to 0.0. Similarly, if the w-component is undefined, it defaults to 1.0. See the DX8 help-files for more information on vertex-shader declarations.

A call to `CreateVertexShader()` takes as parameters a pointer to the compiled vertex-shader in memory and to the vertex-shader declaration. This call actually creates the vertex shader and returns a handle to it via the last parameter. Passing this handle to `SetVertexShader()` switches that vertex shader on. All these calls return error-codes that are worth checking. Also see the DX8 help-files for more information on these function-calls.

Before rendering primitives to see the effects of our vertex shader, we must initialize constant memory with the values that the vertex shader expects to find there. The vertex shader of Listing1_SimpleVertexShader.nvv expects a model-space to clip-space transformation-matrix in `c0-c3`, a model-space light-direction vector in `c4`, and an RGB-material color in `c5`.

To initialize these values, Listing2_SetUp.cpp first queries the current render-state for the current world-, view-, and projection-matrices. To be valid, these render-states need to have been previously set via the corresponding SetTransform() calls. In general, not querying render-state is more efficient and thus preferable. Combining these three matrices into a single transform yields the desired model-space to clip-space transformation-matrix. Since D3DX operates in a row-vector format and the vertex shader of Listing1_SimpleVertexShader.nvv uses column-vectors, the format of this matrix is wrong. Transposing the model-space to clip-space matrix fixes this incompatibility.

The call to SetVertexShaderConstant() writes the four-by-four matrix to constant memory. The first parameter specifies where in constant-memory to write the data. The second parameter is the main-memory location from where to copy the data. The third and last parameter indicates how many float-vectors to copy. Thus, if the third parameter is one, one float-vector, i.e., four floating-point numbers, copies from main-memory to constant memory. To copy the four-by-four matrix, four float-vectors need to transfer from main-memory to constant-memory.

The code that follows defines a light-direction in world-space. Before writing this light-direction into constant memory, the inverse model- to world-space matrix transforms it to model-space.

Finally, a four-vector specifies the material color. The last call to SetVertexShaderConstant() writes it into constant memory at location c5.

These code-snippets do not show the final step of creating and drawing primitives. Instead, it is now time to download functional and complete example-code. The NVidia Effects Browser (see **NVEffectsBrowser** on http://www.nvidia.com/marketing/developer/devrel.nsf/TechnicalDemosFrame?OpenPage) is a well-documented and easy-to-use framework that makes playing with vertex shaders easy. Follow the instructions for "To come up to speed on DX8 quickly" on http://www.nvidia.com/Developer.nsf to install all necessary components to run it. Then download and install the effects browser and at least one of the effects, say, **Simple Quad Vertex Shader** (http://www.nvidia.com/marketing/developer/devrel.nsf/TechnicalDemosFrame?OpenPage). Remember to also read the document NVEffectsExplained.pdf.

The NVEffectsBrowser only requires Microsoft's DX8 Runtime: no DX8 drivers or hardware is needed to run and experiment! Vertex shaders run without DX8 driver- or hardware-support, because the DX8 run-time is able to emulate vertex shaders on the CPU. Once DX8 hardware and drivers are available, however, all these examples execute faster on the GPU (where they belong).

Go ahead and look at the source code for the "Simple Quad Vertex Shader" effect. The source code for the vertex-shading program of this effect is in Quad.nvv. The C++-code for creating and initializing the vertex shader is in shader_Quad.cpp. These files are very similar to the code-snippets explained line-by-line above.

As you start to experiment and change code, you may have the need to debug vertex shaders, i.e., step through the vertex-shader program and examine the contents of memory and registers. Luckily, NVidia provides a vertex-shader debugger for free (**NVIDIA Shader Debugger**
at
http://www.nvidia.com/Marketing/Developer/DevRel.nsf/ProgrammingResourcesFrame?OpenPage).

Now you know everything you need for programming with vertex shaders. Go to it! If you would like to know more inside tips and tricks, though, read on.

Advanced Vertex-Shader Examples

This section delves deeper into the vertex-shading lore. It presents some of the flexibility available with vertex shaders. Each of the subsections exposes various effects achievable via modification of the fixed-function algorithm to compute the per-vertex outputs. First, transform-effects modify the standard per-vertex clip-position computations. Second, lighting-effects modify how they compute the final per-vertex color. Third, texture-effects modify how they compute and use texture coordinates. Fourth and finally, sprite-point size- and fog-effects make use of the `oPts` and `oFog` per-vertex outputs. All examples discussed here are available for download (with source code) from http://www.nvidia.com/marketing/developer/devrel.nsf/TechnicalDemosFrame?OpenPage.

Transform Effects

Vertex processing typically transforms a per-vertex model-space position (provided via a per-vertex input) to clip-space and outputs the result to the per-vertex output `oPos` (see Listing1_SimpleVertexShader.nvv above). The following examples apply more complicated algorithms to compute the final clip-space position of a vertex.

The first example (see Figure 3) shows how to implement indexed matrix-palette skinning. Indexed matrix-palette skinning is useful in animating models. To animate models realistically, animators do not animate a model's individual vertices, but rather animate a bone-structure of a model. Associating a list of bones and a weight per used bone with every vertex then translates the bone-structure animation into a per-vertex animation: a vertex's transformation is the weighted sum of the transformations of the bones associated with that vertex.

Implementation of this technique in a vertex shader requires the following data. Constant memory stores the transformation matrices of all bones. Since transformation matrices are three-by-three, storing of up to 32 (constant memory size divided by three float-vectors per matrix) transformation matrices in constant memory is possible, if the vertex shader required no other data.

Every vertex contains its model-space position, a list of indices referring to the bone transformation-matrices in constant memory, and a list of weights associated with each index. Since this example limits itself to two-matrix skinning, each vertex contains only two indices and one weight (the second weight derives from the first one, since the sum of all weights equals one). Expanding this example to three-, four-, etc. matrix skinning is straightforward. In addition, this example also stores an S-, T-, and SxT-vector with every vertex. These vectors describe a local coordinate system for each vertex. Applying matrix-palette skinning to these vectors and then constructing a matrix from them yields a transformation that maps vectors from world-space to the vertex's texture-space. This transformation-matrix is useful for lighting calculations, as explained below.

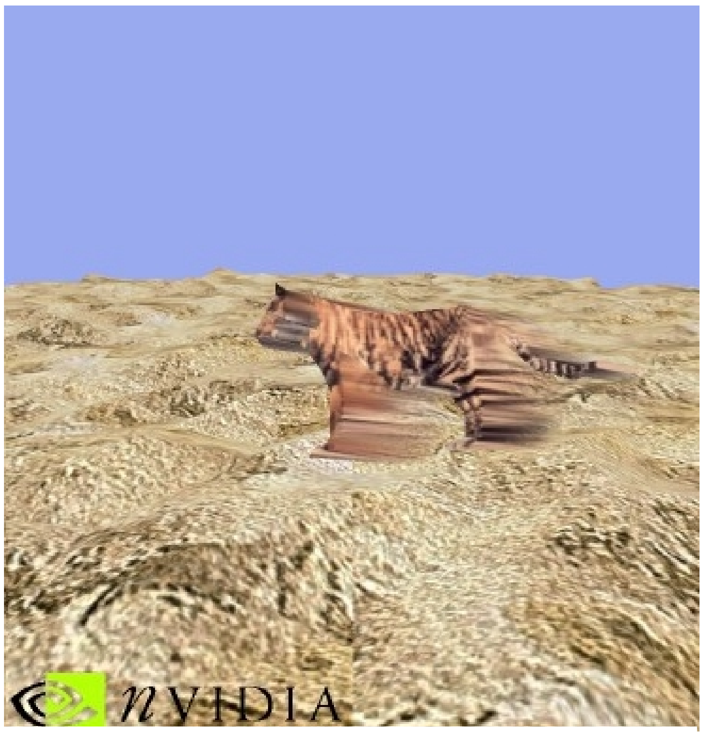
The vertex-shading program (shown in Listing3_MatrixPaletteSkiing.nvv) first calculates the second weight as (1.0 – first-weight). Then it transforms the per-vertex indices to actual constant-memory addresses. After loading the address-register, it transforms the vertex's position with the first index's bone-transformation. It then applies the same transformation to the vertex's S-, T-, and SxT vectors.

After loading the address-register with the address of the second bone's transformation-matrix, the following matrix-multiplications generate the vertex's position, S-, T-, and SxT-vectors transformed by the second bone. Using the per-vertex weights, blending between these results provides the final position, as well as final S-, T-, and SxT-vectors. Since this final position is still in model-space, the vertex-shading program multiplies this position with a matrix that takes the skinned vertex-position from model-space to clip-space.

The remaining instructions copy texture coordinates, as well as transform a light-direction vector to local texture-space and output it to `oD0`. Because the example sets the render-state so as to perform a dot product between `oD0` and a bump texture-map later on, it performs per-pixel diffuse lighting using the transformed light-vector stored in `oD0`.

The second transform-effect adapts the motion-blur technique described in Wloka, M. and Zeleznik, R.C. "Interactive Real-Time Motion Blur," *Visual Computer,* Springer Verlag, 1996, for use in vertex shaders. Listing4_MotionBlur.nvv lists the vertex shader program, while Figure 4 shows its result.

This motion-blur effect operates in two passes. The first pass renders the object that is to be motion-blurred fully opaque and in a normal mode, i.e., it simply transforms model-space vertex positions to clip-space with a four-by-four matrix (see above). This first pass is necessary to avoid rendering a wholly transparent object, when the object moves away from the camera (see below).



The second pass transforms a model-space vertex-position into view-space using both the current frame's model- to view-space transformation, as well as the previous frame's model- to view-space transformation. The difference between these two positions is a per-vertex motion vector. The next multiplication optionally shortens (or lengthens) this motion-vector to artificially shorten (or lengthen) the generated blur.

If the dot product of the motion-vector with the vertex-normal (in view-space) is positive, then the vertex faces into the direction of motion and the vertex shader transforms the vertex derived from the current frame's transformation to clip-space by multiplying it with the projection matrix. If the dot product is negative, however, then the vertex faces away from the direction of motion and the vertex shader uses the position derived from the previous frame's transformation.

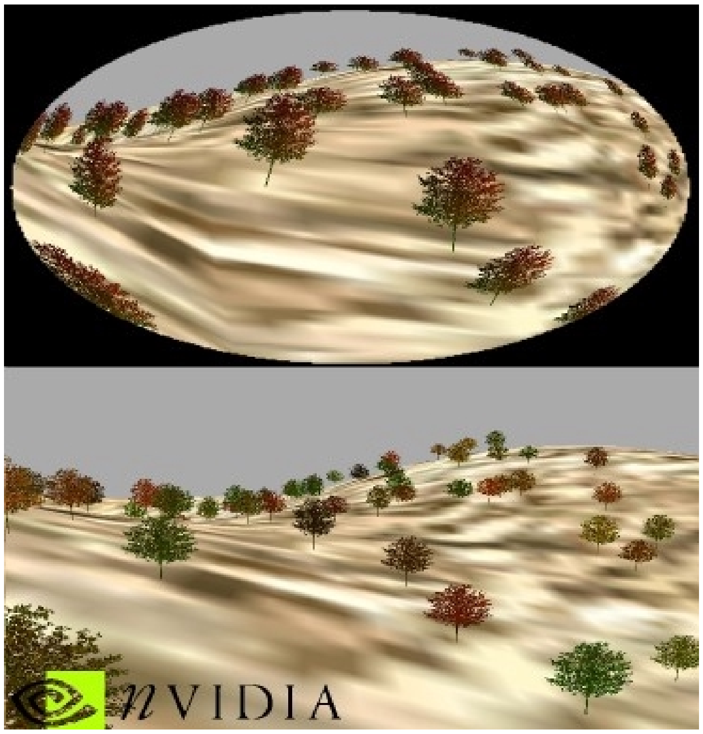
The vertex shader therefore stretches the model from its previous frame's position to its current frame's position on the screen. The faster an object moves, the bigger the difference between the object's position from frame to frame, and thus the longer the stretch.

To enhance the motion-blur look, the vertex shader then modifies the alpha-component of the diffuse color of the trailing vertices. The alpha component is proportional to the ratio of the length of the motion-vector over the extent of the object. Because the alpha component is proportional to the length of the motion vector, slow-moving objects do not become overly transparent. Since the hardware automatically interpolates the per-pixel alpha-values between vertices, mixing fully opaque leading vertices with semi-transparent trailing vertices results in the desired streaking seen in Figure 4.

Because trailing vertices are transparent, and because only trailing vertices may be visible, i.e., all leading vertices are part of camera back-facing triangles, e.g., when an object moves away from the camera, the second pass may render a mostly transparent object. The first rendering pass of this technique counteracts objects becoming intermittently invisible.

As the vertex shader creates semi-transparent triangles, it is also necessary to render triangles roughly back-to-front to achieve proper alpha blending. Luckily, it suffices to sort an object's triangles with respect to its six principal directions (+/-x, +/-y, and +/-z) at load time and store the result as six vertex-index lists. At render-time the principal direction closest to the camera's view-direction determines which vertex-index list to use.

The motion-blur effect is, in a sense, a variation on matrix skinning. All vertices use two matrices and two weights, except that weights are either zero or one and derive from the per-vertex normal and position.



The fisheye-lens effect, shown in Figure 5, applies a non-linear deformation to vertices in view-space (this technique is based on http://www-cad.eecs.berkeley.edu/Respep/Research/interact/playpen/fish/). The deformation maps a vertex position (x, y, z, w) to a new vertex position (x', y', z', w') in view-space. The following formulae use the constants `scalefactor` and `k` to transform (x, y, z, w) to (x', y', z', w'):

$d = |\, \text{scalefactor} * \text{sqrt}(x^2 + y^2)/z \,|$

$x' = x * (k + 1)/(k*d +1)$

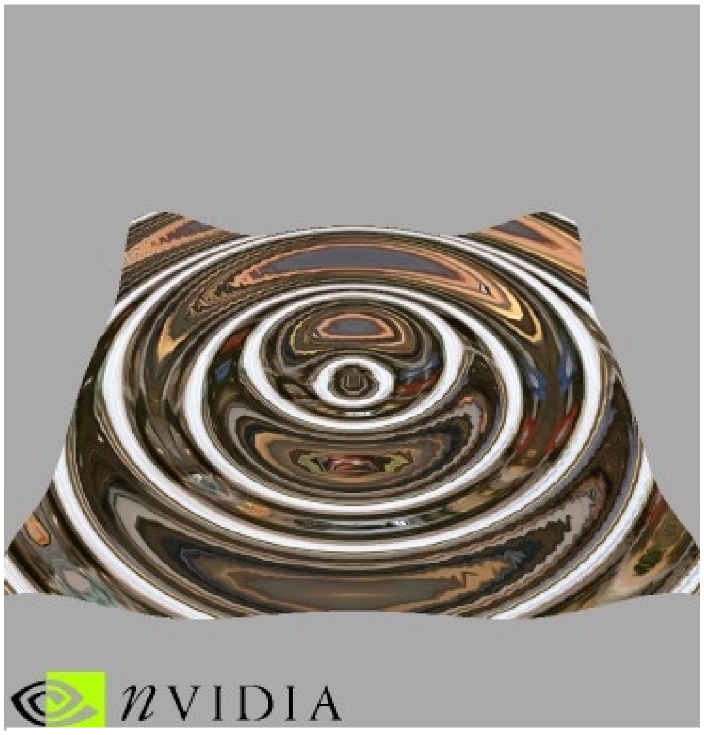
$y' = y * (k + 1)/(k*d +1)$

$z' = z$

$w' = w$

Since the distance d above is not normalized and generally larger than one, this formulation differs from the above referenced web page. Using the reciprocal of the formula provided on the referenced web page compensates for this difference. Finally, removing d from the nominator delocalizes and reduces the deformation-effect and makes it more applicable for our purposes.

The vertex shader (shown in Listing5_Fisheye.nvv) first transforms the per-vertex model-space position to view space. Then it deforms the x- and y-coordinates according to the above formulae, before finally projecting the result of that deformation to clip-space. The clip-space coordinates output to `oPos`. The final two instructions simply copy per-vertex color and texture-coordinates inputs to the corresponding per-vertex outputs.



The fourth and final transform effect (see Figure 6) applies a time-dependent non-linear deformation to vertices. It differs from all previous effects in that none of the processed vertices provide model-space positions. The vertex shader instead generates these data from other per-vertex inputs. The advantage of this effect is its ability to non-linearly animate objects without the CPU having to access vertex data—a generally costly operation—while at the same time minimizing vertex-size.

The only available per-vertex inputs are a vertex's uv-coordinates in the range [-1, 1]. From this input the vertex-shader computes a vertex's model-space position, its normal, the eye-direction vector, and the reflection of the eye-direction vector. This reflection-vector is the vertex-shader's primary output: the render-state uses it to look-up color-values from a surrounding cube-map, giving the appearance of a highly reflective surface (see Figure 6).

The vertex's model-space position is (u, v, h * sin(t * sqrt(u^2 + v^2), 1), where h and t are constants from constant-memory corresponding to height and time, respectively. Every frame the application increments the time-parameter t, thus animating the deformation.

The per-vertex normal is the cross product of the tangent vectors along the u- and v-directions at the vertex position. I.e.,

a = sqrt($u^2 + v^2$)

Position(u, v, t) = (u, v, h * sin(t * a))

S(u, v, t) = d/du Position(u, v, t) = (1, 0, h * cos(t*a) * t * u/a)

T(u, v, t) = d/dv Position(u, v, t) = (0, 1, h * cos(t*a) * t * v/a)

Normal(u, v, t) = S(u, v, t) x T(u, v, t) = (-h * cos(t*a) * t * u/a,
-h * cos(t*a) * t * v/a,
1)

The eye-direction E points from the camera position to the vertex position:

E(u, v, t) = cameraPosition – Position(u, v, t)

And finally, the reflection R of the eye-direction vector mirrors the eye-direction around the per-vertex normal:

R(u, v, t) = E(u, v, t) – 2 * (E(u, v, t) dot Normal(u, v, t)) * Normal(u, v, t)

Listing6_NonlinearAnimation.nvv provides the code for this effect. First, it computes t * a = t * sqrt($u^2 + v^2$). Because there is no sine or cosine vertex-shader instruction, the vertex shader then approximates these functions using a Taylor-series expansion:

f(x) ~= f(0) + x * f'(0)/1! + x^2 * f''(0)/2! + … + x^n* f^n(0)/n!

→ sin(x) ~= x - x^3 /6 + x^5 /120 – x^7 /5040

→ cos(x) ~= x - x^3 /6 + x^5 /120 – x^7 /5040

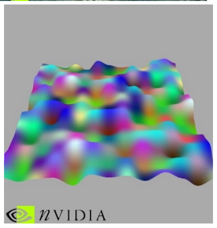
The code first limits the argument x = t*a to the range –PI to +PI. Since sine and cosine are periodic, this operation does not change the theoretical result, but it much improves the precision of the Taylor-series approximation. The following multiply statements compute all the required powers of x, before multiplying them with the Taylor-series coefficient stored in constant memory. The dot-product statements efficiently compute the final sums. The next step transforms the computed position to clip-space and output it.

The last part of this vertex-shader computes the reflection of the vector from the eye to the vertex. These computations happen in view-space, since this reflection vector R is expected in that space. First, the vector from the eye to the vertex (call it E) is the same as the vertex-position transformed to view-space, since in view-space the eye resides at the origin. Then, the per-vertex normal N is computed as given by the formula listed above. After transformation from model-space to view-space, N has to be normalized. The reflection vector R then computes as

R = E - 2*(E dot N)*N

And is stored in oT0.

The mesh-blending effect (see **Mesh Blending**, not shown here) is similar to the indexed matrix-palette skinning effect explained above. Instead of storing a single model-space position per vertex, it stores two model-space positions in every vertex. Weights from constant memory govern the blending between these two positions and vary over time to animate this morph.



The Perlin-noise effect (see **Perlin Noise**, not shown here) is similar to the nonlinear animation effect explained above. It uses bi-linearly-interpolated table-lookups to generate final vertex-position and –color from original per-vertex positions.

Conclusion

If after tinkering with vertex shaders you come up with a really cool effect, please feel free to submit it to demosub@NVIDIA.com for display on NVIDIA's web page.