Shading Languages

GeForce3 and DirectX8

Michael Oswald

Introduction

Shading Languages have become an important part of computer graphics. Sophisticated effects like reflections without a complicated and slow ray-tracing algorithm are possible with less effort. The RenderMan [1] Shading Language for example is massively used in films and animations including reflections and transparent objects, though it is in principal a scan-line renderer. The ISL (Interactive Shading Language) [2] for example is another approach that leads more towards real-time rendering. The shader is parsed into OpenGL-calls that, of course, can be accelerated by hardware. Still a problem is the complexity of ISL, for example the frame buffer is read out some times, a very expensive operation on custom graphic-cards.
So the goal was to have a hardware-implementation of a shading language to achieve maximum performance on real-time graphics. The first step was the PixelFlow System [3] that consists of many rendering nodes and a shading language called pfman that is first compiled into C++ classes with OpenGL calls and then in a second step compiled to machine code for the PixelFlow. The point is that PixelFlow is a very expensive system with a lot of hardware.
For the consumer-section Nvidia made the first step with the GeForce3 graphic-card with an own assembler like language. In close work with Microsoft, the GeForce3 in the moment represents the only graphics-card that supports the DirectX8 shading language, which in term is built on the assembler language of Nvidia. I will now cover the basic functionality from DirectX8 [4] (and therefore the GeForce3).

The Graphics Pipeline

The standard graphics-pipeline for common graphics looks the following way:

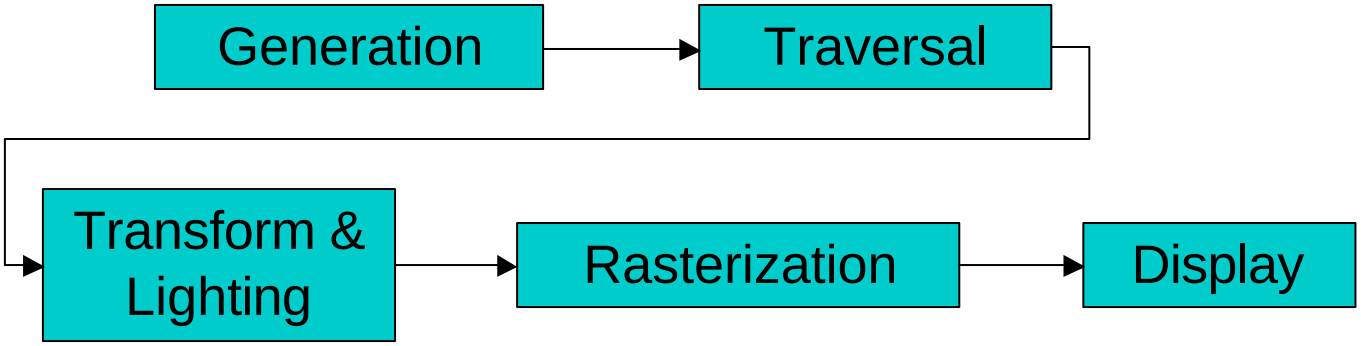


Figure 1.

- ***Generation:*** in the generation-stage the objects in a scene are generated and divided into triangles. This is a full software-step in responsibility of the application.
- ***Traversal:*** the tree of the scene is traversed and for each object the triangles are passed to transform & lighting stage.
- ***Transform & Lighting:*** in this stage the triangles are transformed in space, lighted with a fixed lighting model (e.g. Phong lighting for diffuse and specular reflections), checked for visibility, typically with backface-culling and the visible triangles are clipped against the viewing frustum. For texture-mapping the texture coordinates are generated. The triangle is then passed down to the rasterization-stage. In common hardware, transform & lighting is partly supported and accelerated, unimplemented functions have to be done in software.
- ***Rasterization:*** in this stage the triangles are rendered into the frame-buffer. For each pixel normally a z-buffer test is performed and the pixel's colour value is calculated according to a shading model (e.g. Gouraud Shading, Phong Shading) and written into the frame buffer. This is mostly done in hardware.
- ***Display:*** in the display-stage the frame buffer is sent to the graphics-display

DirectX8 still supports this rendering pipeline, which is called a fixed-function pipeline for backward-compatibility. Additionally there is the possibility to extend the Transform & Lighting and the Rasterization stages with user-defined code:

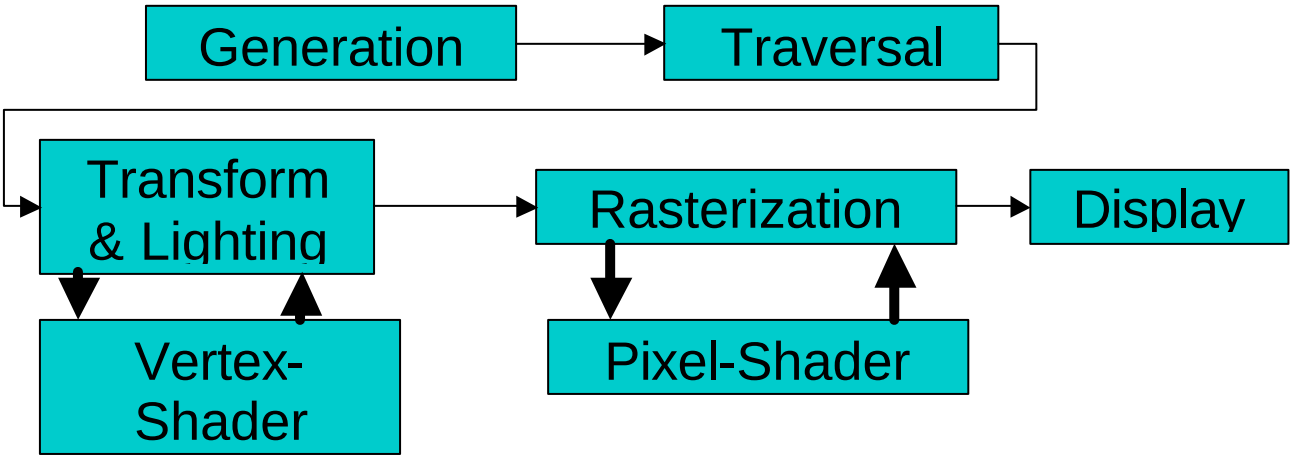


Figure 2.

- ***Generation and Traversal:*** this stages are the same as in the common pipeline
- ***Transform & Lighting:*** here each vertex is passed to a user-definable assembler program that processes the vertex and generates an output vertex for further processing. After the user-defined shader (called the vertex-shader) backface-culling and clipping against the viewing-frustum is performed.
- ***Rasterization:*** the rasterizer calls a user-definable assembler program per pixel, which generates the output-colour. After the pixel-shader a fixed fog-computation is performed and the pixel is put into the frame buffer
- ***Display:*** same as in a normal pipeline.

We will now have a closer look on vertex and pixel-shaders.

Vertex Shaders

As you recall, vertex shaders are small assembler programs executed for each vertex. This means, that a vertex is passed as an input to the program and the program must process an output of this vertex. There is no possibility to create a new vertex or to delete an existing vertex. The structure of a vertex shader looks as follows:

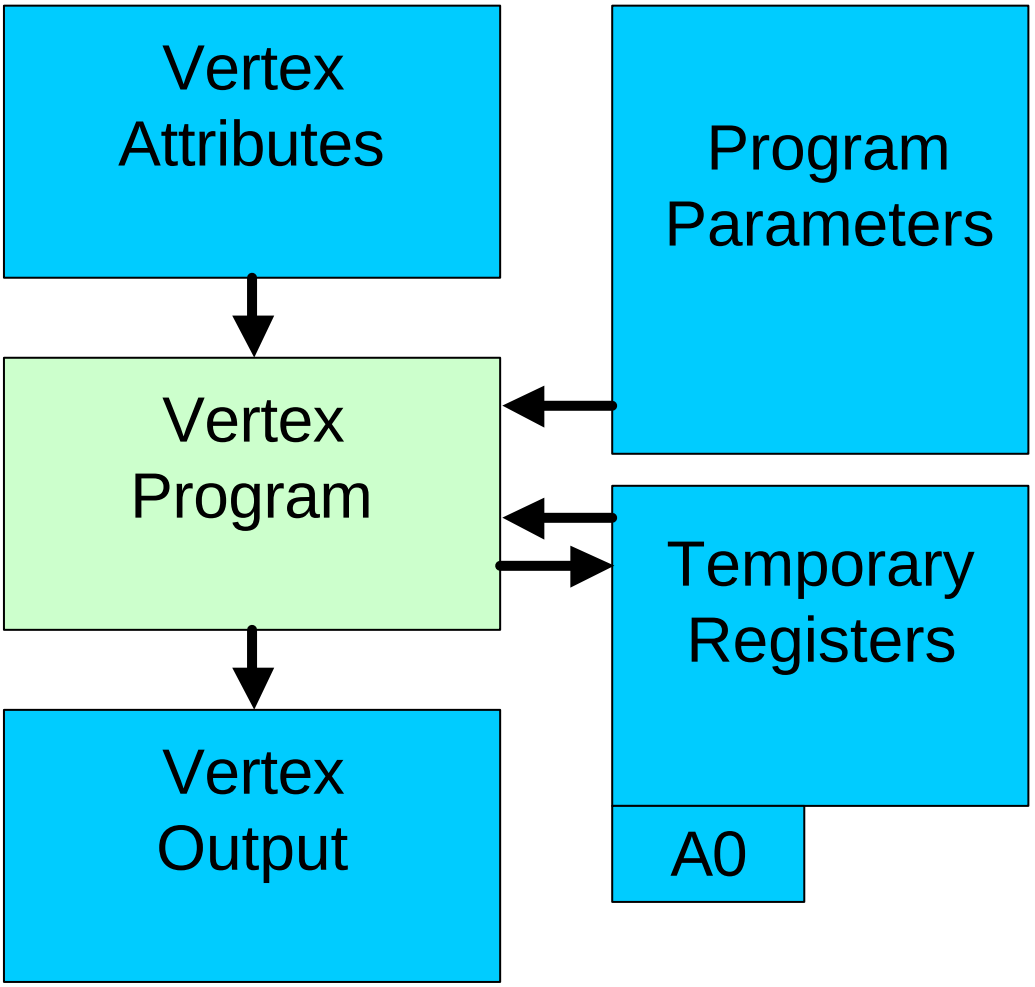


Figure 3.

Data are in a 4-channel format. These values can be treated as homogenous vector coordinates (x, y, z, w), colour values (r, g, b, a), texture coordinates (s, t, r, q) or as four independent scalars.
On input there are 16 of this 4-channel vectors, which can be freely assigned. Of course, one must be the position of the vertex. An example mapping could look like the following:

Name	Description	Format
V0	Vertex position	X, Y, Z, W
V1	Vertex weight	W, 0, 0, 1
V2	Normal	X, Y, Z, 1
V3	Primary colour	R, G, B, A
V4	Secondary colour	R, G, B, A

V5	Fog coefficient	FC, 0, 0, 1
V6	Texture coordinate 0	S, T, R, Q
V7	Texture coordinate 1	S, T, R, Q

This mapping is coded into a D3D-Structure, which then is passed to the input of the vertex-shader. There is no access to information of other vertices, no face or edge information.

The vertex-program itself consists of up to 128 instructions; constants can be read out of a constant-memory (96 4-channel vectors). This memory can contain matrices, vectors, colours, weights and other values, which stay constant to the object. The address-register A0 can be used to perform indirect addressing modes like in other assemblers. Note that the constant-memory is written by the application and is read-only for the vertex-shader.
For intermediate results there are 12 4-channel temporary registers that can be freely used.

After processing, the calculated values must be written into fixed registers, which are used for further processing in the pipeline. The output-registers are the following:

Register	**Description**	**Format**
Opos	Object (Vertex)-Position in homogenous clip-space coordinates	X, Y, Z, W
OD0	Diffuse colour-value	R, G, B, A
OD1	Specular colour-value	R, G, B, A
OT0	Texture coordinate 1	S, T, R, Q
OT1	Texture coordinate 2	S, T, R, Q
OT2	Texture coordinate 3	S, T, R, Q
OT3	Texture coordinate 4	S, T, R, Q
Ofog	Fog coefficient for fog-equation	FC, 0, 0, 1
OptSize	Point size	PS, 0, 0, 1

The complete vertex-shader (program + constant memory) is dynamically loadable for every D3DdrawPrimitve call.

Vertex Shaders Language

The language for vertex shaders consists of 17 instructions, mostly for vector-operations. Each instruction is executed in one clock-cycle no matter how complicated they are. So performance is direct proportional to the length of the program.
All operands of an instruction can be modified in the same step. The following modifications can be applied to operands: Negation, Swizzeling, Smearing and Masking.
Negation simply negates all indicated components of the vector, Swizzeling exchanges components, smearing puts the value of one component into more components and with masking only indicated components are written into a destination-register.

Command Overview

Command	**Description**
`MOV D,S`	Simply moves S to D
`ADD D,S1,S2`	Adds S1 and S2 component-wise and stores result in D (D = S1 + S2)
`MUL D,S1,S2`	Multiplies S1 and S2 component-wise and stores result in D (D = S1 * S2)

`MIN D,S1,S2`	Puts the component-wise minimum of S1 and S2 into D (D=min(S1,S2)
`MAX D,S1,S2`	Puts the component-wise maximum of S1 and S2 into D (D=max(S1,S2)
`MAD D,S1,S2,S3`	Multiplies S1 with S2 then adds S3 and puts result in D (D=S1*S2+S3)
`RCP D,S`	Calculates the reciprocal of S (D=1/S)
`RSQ D,S`	Calculates the reciprocal square-root of S (D=1/sqrt(S))
`EXPP D,S`	Calculates exp(S)
`LOGP D,S`	Calculates log(s)
`DP3 D,S1,S2`	Calculates the dot-product of S1 and S2 for 3-dimensional vectors (D=S1 • S2)
`DP4 D,S1,S2`	Calculates the dot-product of S1 and S2 for 4-dimensional vectors (D=S1 • S2)
`SGE D,S1,S2`	For each component: D = (S1 >= S2) then 1 else 0
`SLT D,S1,S2`	For each component: D = (S1 < S2) then 1 else 0
`LIT D,S`	Calculates lighting-coefficients for Phong-lighting
`DST D,S1,S2`	Calculates the distance-attenuation vector (1, d, d², 1/d)

Some of the instructions are explained here in detail:

RSQ calculates the reciprocal of the square root, which is extremely useful for vector-normalisation.

SGE and **SLT** are for conditional evaluations, because there are no branch-instructions. Branches make a program unpredictable and can force endless-loops. In a vertex-shader program this is not intended and so the execution time of the vertex shader is direct proportional to the instruction count. Because SGE and SLT return 1 or 0, the result can be used for accumulation and multiplication and so the problem of missing branches can be out masked.

LIT calculates the coefficients for the Phong-lighting model. As you recall the equation for intensities in the Phong-model:

$$I = k_a I_a + k_d I_l (N \bullet L) + k_s I_l (N \bullet H)^m$$

The inputs to LIT are the two dot products N•L, N•H and the power m. The following output is generated:

Dest.x = 1.0	ambient coefficient
Dest.y = clamp(N•L, 0, 1)	diffuse coefficient
Dest.z = if (N•L > 0. 0)	specular coefficient
dest. z = (MAX(N • H, 0))^m	
otherwise,	
dest. z = 0.0	

DST calculates the „distance attenuation vector“. It needs as input: d² and 1/d, where d is e.g. |eye-position – vertex position| and calculates the Vector (1, d, d², 1/d).

Some Operations are not implemented, but can be achieved with the following operations:

Absolute Value: **`MAX R1, -R1;`**

Division: **`RCP; MUL`**

Matrix Transform: **`DP4; DP4; DP4; DP4;`**

Cross- Product:

```
MUL r0, r1.yzxw, r2.zxyw;
MAD r0, -r2.yzxw, r1.zxyw, r0;
```

For the cross product, the swizzeling is useful. R1.yzxw means, that the components are “reordered” from xyzw to yzxw. So only two instructions are needed.

Vector-Normalisation:

```
DP3 R0.w, R5, R5;
RSQ R1.w, R0.w;
MUL R5.xyz, R5, R1.w;
```

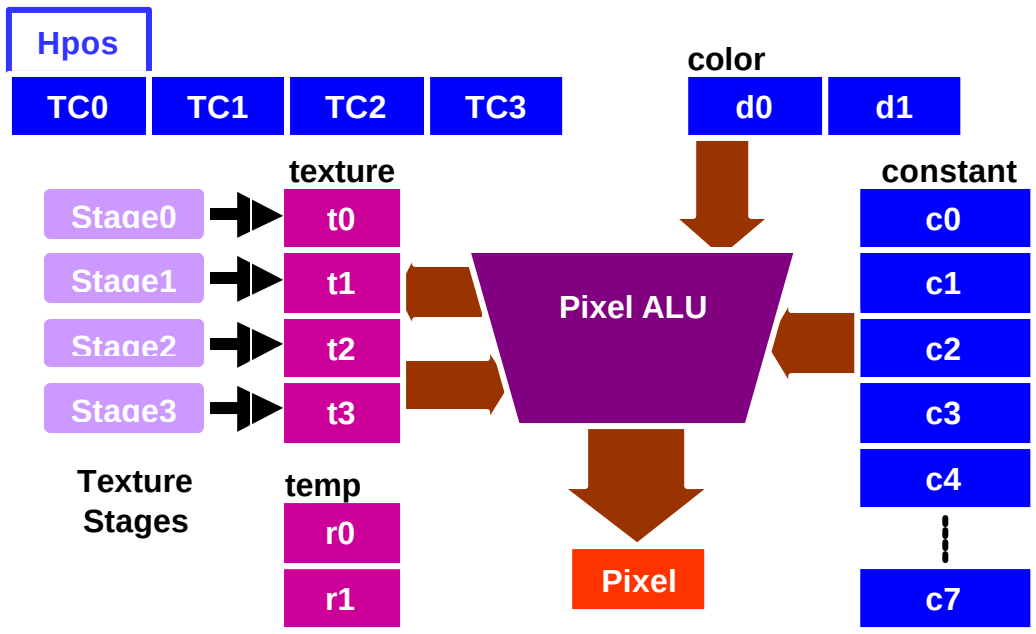
Vector Normalisation is done with only three instructions. In the third line masking and smearing is performed: R5.xyz masks out the w-component, R1.w “smears” the value from w above the x, y and z-component.

DirectX8 supports some Macros, which are expanded into more instructions:

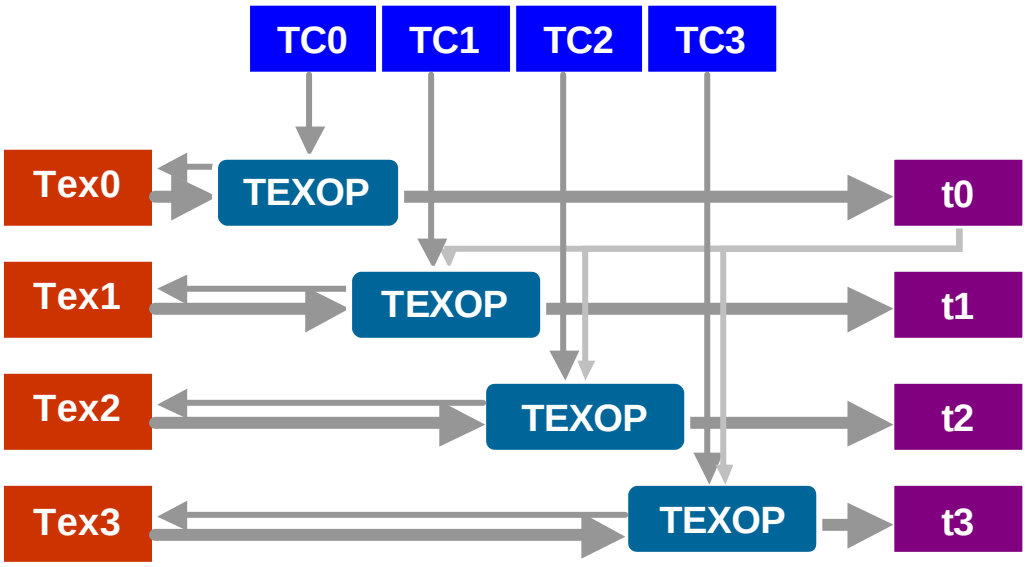
Macro	Description
EXP	Full-precision EXP-function but slow
FRC	Returns the fractional part of a component
LOG	Full precision Log function
M3x2	3x2 vector-matrix multiplication
M3x3	3x3 vector-matrix multiplication
M3x4	3x4 vector-matrix multiplication
M4x3	4x3 vector-matrix multiplication
M4x4	4x4 vector-matrix multiplication

Pixel Shaders

Similar to Vertex-Shaders Pixel-Shaders are small assembler-programs but the program is executed for each pixel. The architecture for a pixel-shader is as follows:



On the input there are the registers TC0 – TC3 that contain the texture-coordinates from the Vertex Shader and the registers d0 and d1 that contain the diffuse and the specular colour. All this registers serve as an input to the pixel-ALU that can process 8 blending-instructions and 4 texture address operations. Up to 7 constants can be used, this constants can be defined from the program with the def-directive. There are only two temporary registers, where R0 serves as the final output of the pixel. The registers are in principal the same 4-channel registers as in a Vertex Shader, but the components are named as rgba, which is the pixel-colour and the alpha-factor for transparency.
The texture addressing operations are coupled with the 4 texture-stages and the four temporary texture-registers t0 – t3, where t0 can be used as further input to the other three registers. The structure is as follows:



For each texture addressing operation there is a “slot” which is assigned to a certain texture-stage, a certain texture coordinate register and a certain temporary register. Each texture-stage has certain properties that can be set from the application (e.g. the texture itself, coefficients for a 2x2 matrix for Environmental Bump Mapping, …).

Pixel Shader Language

The texture addressing operations are the following:

Instruction	Description
TEX	Simply fetches a colour for the texture coordinates in the slot. 2D, 3D textures, normal maps, offset maps and cube maps are possible.
TEXCOORD	Changes the texture-coordinate in a colour. Useful for passing vectors.
TEXKILL	Kills the pixel if one of the coordinates is less than 0. Useful for clip-planes
TEXBEM, TEXBEML	Performs Environmental Bump Mapping with and without luminance. Really needs two instructions.
TEXM3X2PAD	Padding-instruction for TEXM3X2TEX. Performs the first dot product of a 3x2 multiplication and passes output to TEXM3X2TEX.
TEXM3X2TEX	Performs second dot-product and returns the result of the 3x2 multiplication

TEXREG2AR	Performs a dependent texture-lookup. The Alpha and the Red-component of the colour of the first lookup are taken as S and T coordinates for a second lookup. The colour of the second lookup is returned.
TEXREG2GB	Same as TEXREG2AR except that the Green and Blue-components are used as S and T coordinates for the second lookup.
TEXM3X3PAD	Same as TEXM3X2PAD only for a 3x3 multiplication
TEXM3X3SPEC	3x3 calculation for specular reflections with the viewer in the infinite
TEXM3X3VSPEC	3x3 calculation for specular reflections with a given viewer. The viewer coordinates are the three Q-components of the 2 padding and one vspec operation.
TEXM3X3TEX	Rotates a vector through a 3x3 matrix (two padding-instructions are needed) and performs a cube-map or 3D-texture lookup

Some operations are explained more in detail here:

Texbem, texbeml: the first texture is used as an offset map, which returns a vector (ds, dt). This 2D-vector is multiplied with a 2x2-matrix whose constants are set from the application in the certain texture-stage and added to the (S, T) coordinates of the following stage. The output of this calculation is another 2D-Vector (S', T') with which a normal 2D-texture lookup is performed. For texbeml additionally luminance (which is set in the texture-stage from the application) is applied and the colour is returned:

1. Stage $(S_1, T_1, R_1, Q_1) \longrightarrow$ lookup $\longrightarrow$ (ds, dt)
2. Stage $(S_2, T_2, R_2, Q_2) \longrightarrow$ $S' = S_2 + k_{11} * ds + k_{12} * dt$
   $T' = T_2 + k_{21} * ds + k_{22} * dt \longrightarrow (S', T') \longrightarrow$ (RGBA)

Texm3x2pad performs a dot product with the actual texture-coordinates and the source-register and produces the new S-component. **Texm3x2tex** then performs the second dot product (T-component) and uses these new (S, T) coordinates for a texture lookup.
Texm3x3pad is the same as Texm3x2pad. The only difference is the 3x3 calculation (so 2 padding-instructions are needed). The third instruction is **Texm3x3spec**, **Texm3x3vspec** or **Texm3x3tex**, which generates the third coordinate so that a triple (S, T, R) is used for a lookup in a cube-map (the third texture-stage). For Texm3x3spec the (S, T, R) vector is computed as the reflection of the eye-vector about the normal and then used for the cube-map lookup. The viewer is in the infinite. Texm3x3vspec does the same but the coordinates of the viewer are the three Q-coordinates of the texture coordinates. A sequence of this could look like:

Stage	Coordinates	Instruction	Description
0	TC0 : (S_0,T_0,R_0,Q_0)	`Tex t0`	Samples normal-map in t0
1	TC1 : $(S_1,T_1,R_1,Q_1 = v_x)$	`Texm3x3pad t1,t0`	First dot-product of normal with (S_1,T_1,R_1), x-coordinate of eye-vector
2	TC2 : $(S_2,T_2,R_2,Q_2 = v_y)$	`Texm3x3pad t2,t0`	Second dot-product of normal with (S_2,T_2,R_2), y-coordinate of eye-vector
3	TC3 : $(S_3,T_3,R_3,Q_3 = v_z)$	`Texm3x3vspec t3,t0`	Third dot product of normal with (S_3,T_3,R_3), z-coordinate of eye-vector. The

			transformed normal is used to reflect the eye-vector, these reflected vector is used to perform the cube-map-lookup
		`Mov r0,t3`	Move result colour to output

Texm3x3vspec is used for bumpy, reflective surfaces :

After the texture addressing operations there can be up to 8 blending-operations. The blending-operations are:

Instruction	Description
`Add D,S1,S2`	Adds two values (D = S1 + S2)
`Sub D,S1,S2`	Subtracts two values (D = S1 – S2)
`Mul D,S1,S2`	Multiplies two values (D = S1 * S2)
`Mad D,S1,S2,S3`	Multiplies two values and adds a third value (D = S1 * S2 + S3)
`Lrp D,S1,S2,S3`	Linear interpolation (D = S1 * S2 + (1 - S1) * S3
`Dp3 D,S1,S2`	3-component dot-product (D = S1.r*S2.r + S1.g*S2.g + S1.b*S2.b)
`Mov D,S`	Moves s to d
`Cnd D,R0.a,S1,S2`	Condition-check: if R0.a > 0.5 then D = S1 else D=S2

In addition the arguments can be modified:

- R0.a : alpha replicate
- -R0 : negate
- 1-R0 : invert
- _bias : biasing (subtract 0.5)
- _bx2 : unsigned-signed conversion, then bias and scale with factor 2

and the result can be modified:

- _x2 : double result
- _x4 : quadruple result
- _d2 : halve result

- _sat : saturate result (clamp to [0, 1])

where _sat can be used together with the scaling-modifiers.

Example:

```
Dp3_x2_sat r1, r0_bx2, t0_bx2
```

In this example r0 and t0 are changed to signed values, biased and scaled with 2. Then the dot product is performed (r0 • t0) and the result is scaled with 2 and then clamped to the interval [0,1].

The blending operations can be grouped for performance reasons. There exists a vector pipeline and an alpha pipeline, which run in parallel. The vector-pipeline processes 3-component-vectors (RGB value), the alpha-pipeline a single scalar (the A-value). A vector and an alpha-instruction can be grouped to be executed in parallel. For example:

```
  add r0.rgb, t1.rgb t0.rgb
+ mul r0.a, r1.a, t1.a
```

are executed in parallel.

Some technical data from the GeForce 3:

Graphics Core:	256-bit
Memory Interface:	128-bit DDR
Fill Rate:	3.2 Billion AA samples/s
Operations Per Second:	800 Billion
Memory Bandwidth:	7.36 GB/s

Conclusion

Shading Languages have become increasingly fast and are therefore well suited for real-time-computer graphics. The effort is by far not so high as, for example, ray tracing, with highly identical result. Especially in real-time graphics with a lot of motion there is no need for highly photo-realistic graphics, since motion is always a little bit unsharp, which led to the introduction of motion blur. So there is no need for highly detailed photo-realistic graphics in games for example. For this purpose Shading Languages serve as very good approximations. The integration of shaders into hardware pushed the possibilities of real time graphics to the next higher level. As seen in the chapters above, very complex and sophisticated surfaces and motions can be programmed. An example for the combination of Vertex- and Pixel Shader would be the demonstration video “Zoltar” [5] on the Nvidia homepage. The motion of the face of Zoltar with the wrinkles in face and forehead are done with a Vertex Shader, the reflection in the eyes, the beard stubbles and wrinkles in the skin with a Pixel-Shader. Shadowed bump maps, motion blur, cloth-simulation, anisotropic lighting and some simple ray-tracing algorithms are some of the possible effects for which implementations can be found at [6]. There are also some good examples in the MSDN Library [4] for DirectX8.
Further steps will be faster hardware, more memory for textures, matrices, vectors and constants, longer possible programs, new instructions and fewer restrictions, than in DirectX8

are. This includes the use of faster buses, because the limit of today's buses (PCI and AGP) is reached. The next step could involve hardware-real-time ray tracing although, with the use of Shaders, it seems not necessary.

References

[1] Upstill, Steve: *The RenderMan Companion*, Addison-Wesley, 1989
[2] SGI: *Interactive Multi-Pass Programmable Shading* ACM SIGGRAPH 2000
[3] Marc Olano, Anselmo Lastra: A Shading Language on Graphics Hardware:
The PixelFlow Shading System ACM SIGGRAPH 1998
[4] Microsoft: DirectX8 http://msdn.microsoft.com/directx
[5] Nvidia : « Zoltar » http://nvidia.com/Products/GeForce3.nsf/action.html
[6] Nvidia: Technichal Publications Library:
http://www.nvidia.com/Marketing/Developer/DevRel.nsf/TechnicalPresentationsFrame?OpenPage