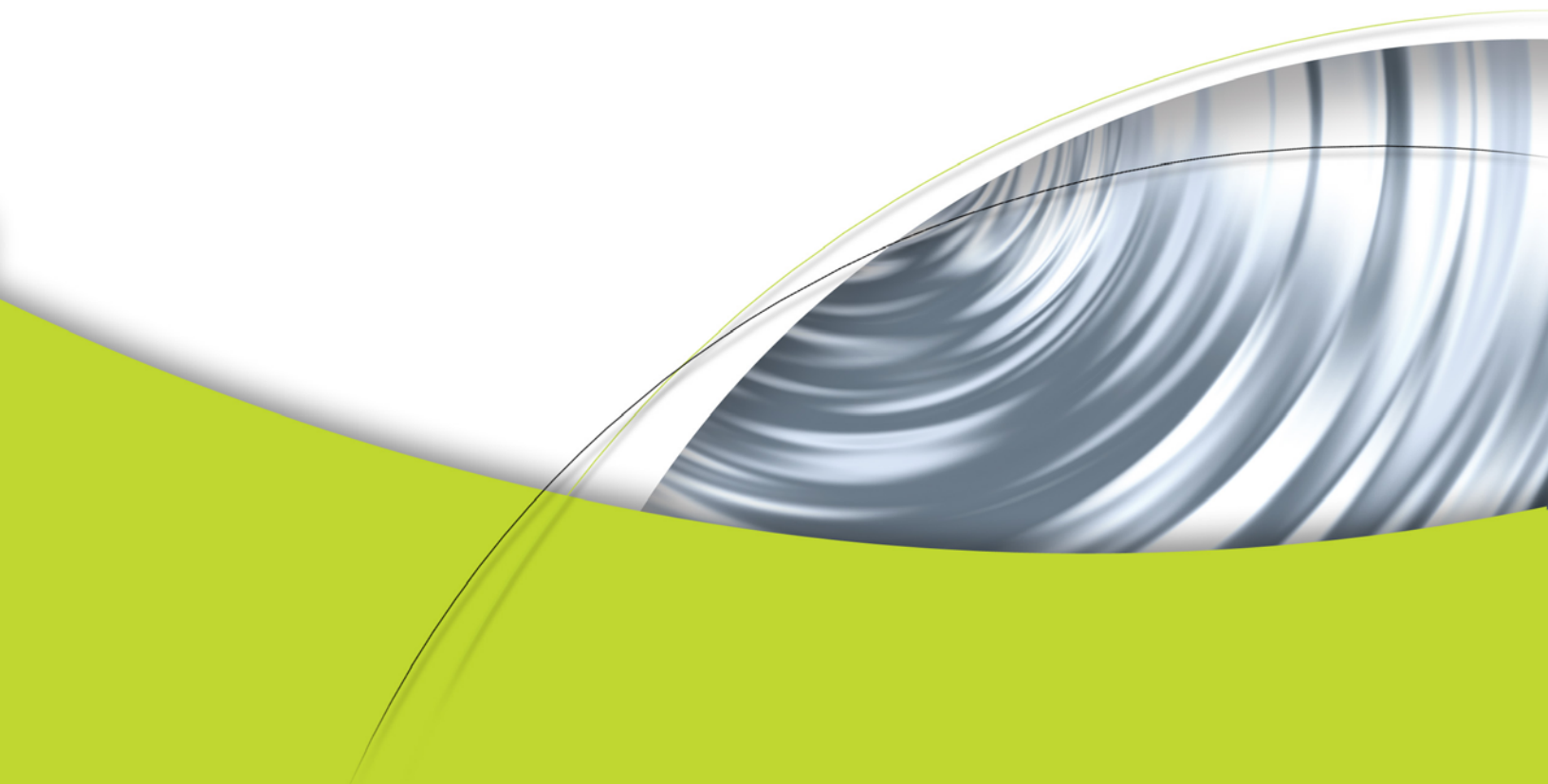




Technical Brief

NVIDIA CineFX Shaders

Cinematic Programmability for
Amazing Visual Effects

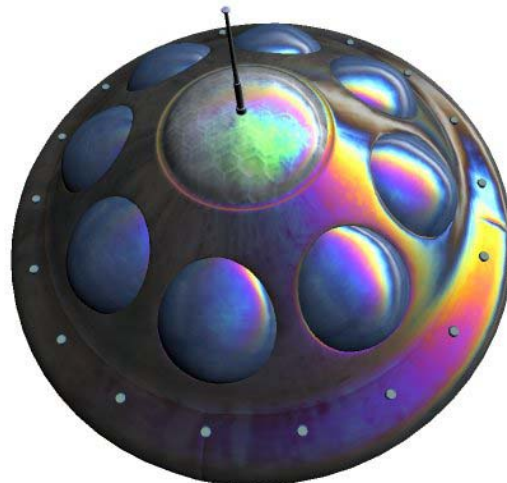


Expanding the World of Pixel and Vertex Shaders

The latest generation of NVIDIA® graphics processing units (GPUs) ushers in a new age of cinematic visual effects. The power and precision of these new GPUs deliver real-time cinematic graphics and, for the first time, bring true 128-bit color to the desktop. These dramatic advances shift the focus from simple pixel fill rate to sophisticated pixel shading. And, by taking advantage of the new NVIDIA CineFX™ engine, programmers gain:

- Increased levels of precision for true 128-bit color, switching between the various precision modes within a single shader program.
- The ability to write longer shader programs incorporating many more effects.
- Dynamic branching and looping for greatly enhanced control flow.
- Shorter shader development cycles using the Cg graphics language.

This technical brief explains the new level of precision and changes to both pixel and vertex shaders. These innovations enable more realistic shaders and lighting for characters; more lifelike character animation; and sophisticated effects and looks that can be applied to the entire scene in real-time.



(© 2002 NVIDIA Corporation)

Figure 1. New shaders simplify the creation of special effects such as this thin-film refraction technique.

Increased Precision

“The advent of 32-bit floating-point pixel precision makes it possible to create high-quality images. Efficient volumetric effects—ground fog, spherical fog, sprites that fade out smoothly rather than getting clipped by world geometry—can be based on buffering and reusing per-pixel 32-bit floating-point z. Precise per-pixel lighting attenuation formulas can be based on passing in light-source positions in 32-bit floating-point vector registers. Much higher-quality bump-mapping is also possible with 16- and 32-bit floating point. With just 8 bits per component, there were noticeable artifacts and un-normalized bump maps.”

Tim Sweeney, Epic Games, Inc.

The inherent 16- and 32-bit floating point formats (FP16 and FP32) of the NVIDIA CineFX engine give developers the flexibility to create the highest-quality graphics. FP32 offers the ultimate image quality, bringing full 128-bit precision to the entire graphics pipeline and delivering true 128-bit color in pixel shaders. FP16 provides an optimal balance of image quality and performance. In fact, the FP16 format offers exactly the same format and precision that Industrial Light & Magic and Pixar Animation Studios use for production of their feature films and effects.



(Image taken from the Elder Scrolls III: Morrowind game, courtesy of Bethesda Softworks, Inc.)

Figure 2. Rippling water with reflections and refractions needs high precision for better visual quality and freedom from constraints and limitations.

With FP32 and FP16, developers are free to move back and forth between these formats within a single shader program, using the format that is best suited to each particular computation. For instance, some actions such as indexing into a high-resolution texture can only be optimally accomplished using a 32-bit floating-points format. If the texture is larger than 1024 x 1024 ($2^{10} \times 2^{10}$, requiring at least 10 mantissa bits per texture coordinate), the developer needs FP32 to access all of the data. Other computations can be accurately accomplished using FP16, and can benefit from the maximized execution speed afforded by this level of precision.

For a complete overview of precision, and the problems that can result from lack of precision, please review the NVIDIA technical brief titled “High-Precision Graphics: Studio-Quality Color on the PC.”

Vertex Processing

By delivering greatly expanded vertex processing capabilities, the NVIDIA CineFX engine gives programmers the power they need to achieve literally any effect they can imagine—with greatly simplified programming techniques. For vertex processing, the NVIDIA CineFX engine:

- ❑ **Shatters existing limitations:** the number of instructions supported is increased from 128 to 65,536 through the use of data-dependent branching and more instructions, registers, and constants.
- ❑ **Provides greater control flow:** dynamic loops and branches provide for forward and backward changes in flow; call and return functions have been introduced; and vertex processing can also invoke an early exit on program termination.
- ❑ **Introduces new capabilities:** per-component condition codes and write masks.
- ❑ **Evolves to an advanced instruction set:** new instructions and capabilities including branching (BRA), high-precision trigonometric functions (COS, SIN), and high-precision exponentiation and logarithm functions (EX2, LG2, and others).

One simple example of the incredible power of this highly flexible programming model is matrix palettes skinning for character animation. With the Microsoft® DirectX® 8 (DX8) model, character animation required writing multiple shaders, one for each kind of skinning involved. For example, if a model used vertices that could be affected by one to four bones, the developer would need to write up to four separate shaders, one for each combination of bones (one bone, two bones, three bones or four bones that affect a vertex). In order to do this in DX8, the model would have to be segmented into polygons that used the same number of bones, with all sections of the model using the same number of bones rendered in the same pass (see Figure 3). Alternatively, the developer could always use the four-bone shader, for a simpler but slower solution.



(© 2002 NVIDIA Corporation)

Figure 3. Developing Polygon-based Models

With DirectX 8, characters (such as the one on the left) are created by developing polygon-based models and then writing shaders to calculate the effect of the bones on each vertex of the model. To create complex life-like movement, many of the vertices need to be influenced by a variable number of bones. Complex model segmentation (which is not model based) and multiple shader programs are required to create this result in DirectX 8. The image on the right shows the portions of the model that use a two-bone vertex shader program. Separate shaders are required to render the other parts of the model. (See Appendix D.)

By using the latest API functionality enabled by the newest generations of competitive hardware, this situation somewhat improves. One shader can now be written to represent the example involving up to four bones, but since the latest API only supports a very limited notion of branching, it can only be performed *per object*, which means that the model must still be broken up and drawn separately.

The NVIDIA CineFX engine, with its fully generalized loops and branches that can be data-dependent, has a much more straightforward programming methodology. One shader is written to encompass all the skinning methods and operations, and since the shader can branch on a *per-vertex* basis, it is not required to break up the model. By performing the loop conditionally on a per-vertex basis, segmentation of the model is not necessary, dramatically improving both application performance as well as developer productivity. The CineFX program listing for this example is provided in Appendix D. The code for the section that performs the conditional loop is as follows:

```

for (i = 0; i < IN.numBones.x; i = i+1)
{
    // transform the offset by bone i
    position = position + weight.x * float4(
        dot(boneMatrices[index.x+0], IN.position),
        dot(boneMatrices[index.x+1], IN.position),
        dot(boneMatrices[index.x+2], IN.position), 1);

    normal = normal + weight.x * float3(
        dot(boneMatrices[index.x+0].xyz,
IN.normal.xyz),
        dot(boneMatrices[index.x+1].xyz,
IN.normal.xyz),
        dot(boneMatrices[index.x+2].xyz,
IN.normal.xyz));

    // shift over the index variable
    index = index.yzwx;
    weight = weight.yzwx;
}

```

CineFX Vertex Processing

With the new engine, vertex shaders can take advantage of the following:

- ❑ ***Up to 65,536 vertex instructions executed per vertex (up to 256 static instructions per shader)***

The CineFX shading engine exposes an unprecedented amount of vertex processing capabilities. In addition to doubling the instruction storage, the addition of control flow dramatically increases the amount of actual computation that can occur for each vertex. This flexibility reduces the total number of vertex shaders required by an application.

- ❑ ***Up to 256 vector constants***

The number of constant registers available in the CineFX vertex shader has more than doubled—from 96 up to 256 quad words! This increase allows for substantially more bone matrices for matrix palette skinning and lots more simultaneous light sources.

- ❑ ***Sixteen temporary vector registers***

Temporary register storage has increased by 33% from 12 to 16. This temporary storage is particularly helpful with the larger programs supported by the CineFX engine.

- ❑ ***Up to 64 separate loops***

The CineFX vertex shading engine makes for simpler programs by supporting fully dependent looping and branching (including nested loops and branches) with up to 64 unique branch targets in a single shader program. Looping over all light sources and then branching to the appropriate light type is now a breeze.

The new engine introduces several new vertex shader features:

- ❑ ***Per-component conditional codes and write masks***
Condition codes are the machinery behind data-dependent branching, but they can also improve the performance and simplify the code for conditional assignments.
- ❑ ***Call and return (subroutines)***
In addition to the CineFX branching capabilities, the vertex processor supports full subroutine CALL/RETURN semantics, with an up-to-4-deep call stack.
- ❑ ***Loops and branching for both static and dynamic control flow***
Fully general looping and branching (along with dependent data reference) are what make the CineFX vertex shading engine so flexible and powerful.

Because of the increased capabilities and new features, the CineFX engine makes it much easier to write vertex shaders. For example, it becomes possible to use only one simple draw call for a complete animated character. Enabling and disabling lights no longer requires multiple shader programs; a simple change to a loop register will accomplish the task.

CineFX Vertex Instruction Set

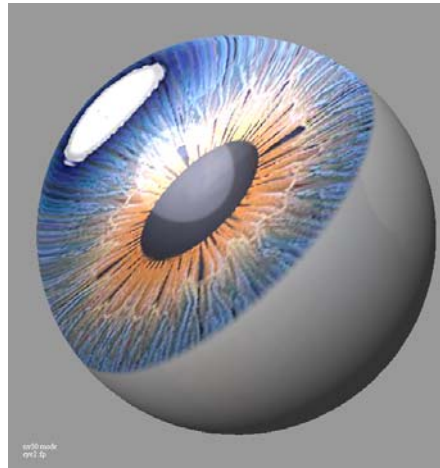
Table 1 shows the new vertex processing instruction set (details in Appendix A).

Table 1. NVIDIA vertex processing instruction set.

Categories	Instructions*
Add and multiply	ADD, DP3, DP4, DPH, MAD, MOV, SUB
Math	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
Set On	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Branching	BRA, CAL, NOP
Address Registers	ARL, ARR
Graphics-oriented	DST, LIT
Minimum/maximum	MAX, MIN

Pixel Processing

The NVIDIA CineFX engine raises pixel shading to a first-class programmable citizen of the graphics pipeline, and gives developers a host of new capabilities for controlling pixels and producing effects that are only limited by the imagination (see Figure 4). With the latest-generation capabilities, users gain GeForce4 vertex program capabilities—at the pixel level and beyond.



(© 2002 NVIDIA Corporation)

Figure 4. Raytraced-in-the-Pixel Shader.

The pixel shader evaluates a refracted view vector, and intersects it with a mathematically defined plane aligned to the model. The intersection point is used as a texture lookup for the iris. Rays that miss the texture are set to an appropriate alternative color (the background). The specular highlight has been processed with a `smoothstep()` function to create the illusion of a larger circular source, making the surface look wetter.

With the CineFX engine, developers can write a single-pass shader that can create a mixed appearance. Previously, multiple bidirectional reflectance distribution function (BRDF) shaders were required, with the results blended using a mask texture or chained using IF statements. The example shown in Figure 5 illustrates the result of a single-pass shader that integrates all of the key parts of the BRDFs as multiple painted textures; only one pass through the shader is required to create the mixed appearance. This permits a single-pass shader containing diffuse, specular, and environmental lighting effects in a compact, fast-executing package.

The shader code for this peeling paint on the metallic surface is included in Appendix E. A section of that code includes:

```

PixelOut main(
    MultiPaintV2F IN,
    uniform sampler2D  ColorMap : texunit0, // color
    uniform sampler2D  MaterialMap : texunit1, // encodes
{specStrength,metalness,normalized_specExpon,0)
    uniform sampler2D  NormalMap : texunit3, // tangent-space
normals
    uniform samplerCUBE EnvMap : texunit2, // environment
skybox
    uniform float4 SpecData, // components: {minpower,
maxPower,maxSpecStr,??}
    uniform float4 ReflData, // components: {fresMin,
fresMax,fresExpon,reflStrength}
    uniform float4 BumpData // components:
{bumpScale,?,?,??}
) {
    PixelOut OUT;
    float4 surfCol = f4tex2D(ColorMap,IN.TexCoords.xy);
    float4 material = f4tex2D(MaterialMap,IN.TexCoords.xy);
    float3 Nt = f3tex2D(NormalMap,IN.TexCoords.xy) -
float3(0.5f,0.5f,0.5f);
    float specStr = material.SPEC_STR * SpecData.MAXSPEC;
    float specPower = SpecData.MINPOWER + material.NORM_SPEC_EXPON
* (SpecData.MAXPOWER-SpecData.MINPOWER);

    float3 Vn = -normalize(IN.VPosition - IN.OPosition);
    float3 Ln = normalize(IN.LightVecO).xyz;
    float3 Nb = normalize(BumpData.BUMP_SCALE * (Nt.x * IN.T +
Nt.y * IN.B) + (Nt.z * IN.N));
    float diff = max(0.0f,-dot(Ln,Nb));
    float4 diffResult = diff * surfCol;
    float isLit = (diff > 0.0f) ? 1.0f : 0.0f;
    float3 Hn = normalize(Vn+Ln);
    float spec = pow(abs(dot(Hn,Nb)),specPower) * specStr;
    float4 WHITE = float4(1f,1f,1f,1f);
    float4 specCol = lerp(WHITE,surfCol,material.METALNESS);
    float4 specResult = (spec * isLit) * specCol;
    float3 reflVect = reflect(Vn,Nb);
    float4 reflColor = f4texCUBE(EnvMap,reflVect);
    float fakeFresnel = ReflData.FRESNEL_MIN +
ReflData.FRESNEL_MAX * pow((1.0f-dot(-
Vn,IN.N)),ReflData.FRESNEL_EXPON);
    float4 paintShine = fakeFresnel * reflColor;
    float4 metalShine = surfCol * reflColor;
    float4 shineCol = ReflData.REFL_STRENGTH *
lerp(paintShine,metalShine,material.METALNESS);
    float4 finalColor = diffResult + shineCol;
    // finalColor = vector_as_color(Ln);
    finalColor = specResult + diffResult + shineCol;
    finalColor.w = 1.0f;
    OUT.COL = finalColor;
    return OUT;
}

```



(© 2002 NVIDIA Corporation)

Figure 5. Multiple Textures In A Single, One-Pass Shader

Next-generation pixel shaders can combine multiple textures in a single, one-pass shader for optimized execution. This example demonstrates handling multiple surface effects (peeling paint on a metallic surface). See Appendix E.

Highlights of the new pixel processing engine are:

- ❑ **Introduces new instruction set for pixel shading:** instructions previously reserved for vertex processing are now available for pixel shading, extended with instructions necessary for pixel processing.
- ❑ **Removes existing limitations:** programs can be longer (up to 1,024 instructions) with up to 16 textures per pixel and unlimited levels of dependent texture lookups.
- ❑ **Vastly expands the number of pixel operations:** up to 1,024 pixel operations; per-component swizzling; per-component conditional write masks; arbitrary texture filters; and other advanced instructions. DirectX 8 supported eight instructions. Using the latest API functionality enabled by the newest generations of competitive hardware, this improves somewhat with support for up to 64 instructions. The NVIDIA CineFX engine, with up to 1,024 instructions, supports truly long shader programs to achieve stunning effects and shader possibilities.
- ❑ **Enhances fragment program storage:** stored in video memory, unlike vertex programs, bringing costs down for managing lots of fragment programs.

CineFX Pixel Instruction Set

The new pixel processing instruction set is provided in Table 2. For details about changes to the pixel shader instruction set, refer to Appendix B.

Table 2. NVIDIA Pixel Processing Instruction Set.

Categories	Instructions
Add and multiply	ADD, LRP, DP3, DP4, LRP, MAD, MOV, SUB
Texturing	TEX, TXD, TXP
Partial Derivatives	DDX, DDY
Math	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, NRM, POW, RCP, RSQ, SIN
Set On	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Graphics-oriented	DST, LIT, RFL
Minimum/maximum	MAX, MIN
Macros (mimic vertex shader functionality)	SINCOS, CRS
Pack	PK2H, PK2US, PK4B, PK4UB, PK4UBG, PK16
Unpack	UP2H, UP2US, UP4B, UP4UB, UP4UBG, UP16
Kill	KIL

New Pixel Operations

Highlights of the new CineFX pixel operations include:

- **Up to 16 texture maps.** The NVIDIA CineFX engine allows fetching from up to 16 unique texture maps in a single pixel shader program. These textures can be anything that defines the underlying surface properties; examples include bump maps, gloss/specularity maps, environment maps, shadow maps, and albedo maps.
- **Up to 1024 texture instructions per shader.** Previous architectures tightly bound the number of unique texture maps with the number of texture fetches available. The NVIDIA CineFX engine relaxes this restriction, and allows up to 1024 texture fetch instructions in a single shader, sourcing up to 16 unique textures. This enables a host of new effects which rely on multiple texture accesses:
 - **Soft shadows.** Soft shadows can now be created by taking an arbitrarily large number of samples from a shadow map and using them to generate a filtered shadow result.
 - **Framebuffer post-processing effects.** A number of interesting effects can be created by taking multiple texture samples from the framebuffer. Blurs, halos, and non-photorealistic rendering effects such as toon shading and painterly effects are now possible.

- **Complex filters.** Higher-quality filtering can be performed on texture lookups. A bicubic filter, for example, requires 16 samples from the same texture.
- **Look-up tables.** Certain functions can be encoded in textures and looked up from the pixel shader. Functions for vector normalization, noise, and lighting can all be encoded in textures, and samples taken from a texture each time the function is to be evaluated.
- ❑ **A maximum of 1024 color instructions.** Since true cinematic rendering often requires a large amount of computation at the per-pixel level, the NVIDIA CineFX engine supports up to 1024 arbitrary color instructions in a pixel shader, thus obviating the need for programmers to worry about instruction count limits. A sampling of effects that require large numbers of instructions includes:
 - **Volume rendering.** Performing ray-marching algorithms to step through volumes for effects such as smoke, fire, fur, and grass requires many instructions. The number of instructions is often increased further when ray-marching effects are recursive. For example, consider the case of shooting rays at each sample point towards a light to gather shadowing terms. Hundreds of instructions are not uncommon in these cases.
 - **Procedural texturing.** Generating procedural textures in real time often uses large numbers of instructions. Analytically antialiasing these procedural textures adds even more instructions that must be computed per-pixel.
 - **Complex lighting/multiple lights.** More complex lighting models have the potential to dramatically improve the realism of rendered images, and they frequently use many more instructions than traditional, simpler lighting models. For example, the Oren-Nayar lighting model, which accurately models rough diffuse surfaces, takes *ten times* as many instructions as traditional Lambertian diffuse lighting. The increased program length now gives the ability to do multiple lights in a single pixel shader, when it used to take multiple passes over the scene to accomplish the same effect.
- ❑ **Up to 64 temporary storage locations.** The ability to perform large numbers of instructions (both color and texture) necessitates a large number of temporary registers to hold intermediate results. Up to 32 FP32 registers or 64 FP16 registers are supported for temporary storage (these are 4-component registers).
- ❑ **Swizzling.** The NVIDIA CineFX engine supports completely flexible swizzling of components for operands to instructions. This enables greater flexibility and better optimization opportunities. For example, the common mathematical operation known as the cross-product takes only two instructions with a bit of clever swizzling:
 - ❑ MUL r0, r1.yzxw, r2.zxyw
 - ❑ MAD r0, -r2.yzxw, r1.zxyw, r0
- ❑ **Conditional write masks.** Flexible conditional write masking enables simple predicated branching, where both paths of a branch are computed and only one is chosen. The flexibility of the CineFX engine in this regard allows a number of optimization opportunities for the programmer, or for the compiler in the case of a high-level language such as Cg.

The NVIDIA CineFX engine supports a number of advanced pixel shader instructions:

- ❑ **DDX, DDY:** For computing the screen-space derivatives of an arbitrary value with respect to x and y, respectively, these instructions provide a measure of access to information about neighboring pixels and are invaluable for a number of effects such as toon shading and antialiasing.
- ❑ **TXD:** This instruction allows fetching from a texture and providing custom values for the partial derivatives of the texture coordinates with respect to x and y window coordinates. These partials are then used in the usual LOD calculations, thus giving unprecedented control over LOD.
- ❑ **SIN, COS:** High-precision trigonometric functionality didn't exist at the vertex level in previous generations of hardware. With the CineFX engine, the functionality is available per-pixel.
- ❑ **PK, UP (and variants):** These instructions allow programmers to “pack” and “unpack” smaller datatypes into larger ones. For example, 16 8-bit values can be packed into a single 128-bit output (which would normally store four 32-bit values). Later, these 16 values can be read back in and unpacked into registers. This capability can be used to store more than four per-pixel attributes in cases such as handling a vector normal, diffuse texture color, specular texture color, and high-precision depth.



(© 2002 NVIDIA Corporation)

Figure 6. Alien model.

This long pixel shader implements Lafortune Phong-nodes based on real-world measurements from aluminum, modulated with fresnel to create a vinyl appearance. (Source code: see Appendix C.)

Conclusion

The CineFX engine represents a major leap forward in pixel and vertex shader capabilities and is resulting in a new generation of cinematic effects in real time. Table 3 shows a comparison between current- and new-generation platform capabilities.

Table 3. GeForce4 Ti and GeForce FX Comparison

	GeForce4 Ti	GeForce FX
Higher Order Surfaces		
Geometry Displacement Mapping	-	✓
Vertex Shaders	1.1	2.0+
Max Instructions	128	65536
Max Static Instructions	128	256
Max Constants	96	256
Temporary Registers	12	16
Max Loops	0	256
Static Control flow	-	✓
Dynamic Control flow	-	✓
Pixel Shaders	1.1	2.0+
Texture Maps	4	16
Max Texture Instructions	4	1024
Max Color Instructions	8	1024
Max Temp Storage	-	64
Data Type	INT	FP
Data Precision	32-bit	128-bit

Appendix A.

Vertex Instruction Set Changes

Table 4. NVIDIA vertex processing instruction set.

Categories	Instructions*
Add and multiply	ADD, DP3, DP4, DPH, MAD, MOV, SUB
Math	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
Set On	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Branching	BRA, CAL, NOP
Address Registers	ARL, ARR
Graphics-oriented	DST, LIT
Minimum/maximum	MAX, MIN

Changes to Vertex Shader Numbers

- ❑ 256 instructions of stored program (was 128)
- ❑ 256 constants (was 96)
- ❑ Vector address register (was a scalar)
- ❑ Maximum number of instructions that can be executed per shader is now 65,536.

Appendix B.

Pixel Instruction Set Changes

Table 5. NVIDIA pixel processing instruction set.

Categories	Instructions
Add and multiply	ADD, LRP, DP3, DP4, LRP, MAD, MOV, SUB
Texturing	TEX, TXD, TXP
Partial Derivatives	DDX, DDY
Math	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, NRM, POW, RCP, RSQ, SIN
Set On	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Graphics-oriented	DST, LIT, RFL
Minimum/maximum	MAX, MIN
Pack	PK2H, PK2US, PK4B, PK4UB, PK4UBG, PK16
Unpack	UP2H, UP2US, UP4B, UP4UB, UP4UBG, UP16
Kill	KIL

Pixel Shader Changes

- ❑ Registers and instructions can be 12-bit fixed point, 16-bit floats, or 32-bit floats.
- ❑ Any number of texture fetches from up to 16 unique textures.
- ❑ 1,024 instructions per rendering pass.
- ❑ 8 texture coordinates (up to 16 active textures).
- ❑ If the target is a float surface, then the float value gets “width converted” to match the render target, and then gets stored. (No blending is allowed to float surfaces.)
- ❑ If the source is a float surface, no filtering can be performed on the way into the pixel unit (no bi-linear filter float values).
- ❑ Type and width conversions are all free.
- ❑ All pixels in a batch execute in the same number of clock cycles.

Texture Samplers

CineFX pixel shaders can take advantage of 16 texture samplers. Programmers can designate which sampler and which coordinate to pair up to perform a texture fetch. Each texture coordinate is no longer paired with the corresponding specific texture. Therefore, fetches can be performed from 16 different textures using just one texture coordinate. Similarly, fetches could be performed from eight different unmodified coordinates within one texture.

Appendix C. Pixel Shader Code Samples

Sample Program: Cg Version

The pixel shader used to create the vinyl-like texture for the alien was written in both assembly language and the Cg programming language. The Cg program is listed here:



```
/******NVMH3*****/
Copyright NVIDIA Corporation 2002
TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THIS SOFTWARE IS PROVIDED
*AS IS* AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE.  IN NO EVENT SHALL NVIDIA OR ITS
SUPPLIERS
BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES
WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS
PROFITS,
BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY
LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF
NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Comments:
* "lafortune" physically- based shading with parameters for aluminum and one
*   or two integrated lights. Because aluminum shading relies
*   on a mirror term, a cube map has also been added.
*****/
#define color float3
#define vector float3
//
// COMPILE-TIME OPTIONS
//
// undefine this to see the results with a single lightsource
#define TWO_LIGHTS
// To correct for differences between the color sensitivities used in the
//   original measurements and typical RGB display component colors,
//   enable CORRECTED_COLOR_SAMPLES to use dot product with rows of
matrix
// #define CORRECTED_COLOR_SAMPLES

//
// PROGRAM BEGINS
//
myFo main(vf30 IN,
```

```

        uniform texobjCUBE env_map,
        uniform texobj3D noise_map)
{
    /* parameters are currently hard-wired */
    /* Default BRDF: aluminum */
    /* this would actually be black, but let's pretend it's a little tiny
bit dirty */
    color defaultColor = float3(0.09,0.08,0.15);
    /* float3(cxy=direction, cz=scale, n=exponent) */
    color lobe0R = float3(-1.11854,1.01272,15.8708);
    color lobe0G = float3(-1.11845,1.01469,15.6489);
    color lobe0B = float3(-1.11999,1.01942,15.4571);
    color lobe1R = float3(-1.05334,0.69541,111.267);
    color lobe1G = float3(-1.06409,0.662178,88.9222);
    color lobe1B = float3(-1.08378,0.626672,65.2179);
    color lobe2R = float3(-1.01684,1.00132,180.181);
    color lobe2G = float3(-1.01635,1.00112,184.152);
    color lobe2B = float3(-1.01529,1.00108,195.773);

#ifdef CORRECTED_COLOR_SAMPLES
    color colorMatrixR = float3(1,0,0);
    color colorMatrixG = float3(0,1,0);
    color colorMatrixB = float3(0,0,1);
#endif /* CORRECTED_COLOR_SAMPLES */

    /* tangent space expressed in current coordinate system */
    /* Get unit vector in "u" parameter direction */
    /* Use to get a local coordinate system with local_z as the normal.
*/
    vector STDir = {dxd(IN.TEX0.x)+ddy(IN.TEX0.x),
                    ddx(IN.TEX0.y)+ddy(IN.TEX0.y),0};
    vector local_x = normalize(STDir);
    // vector local_x = normalize ( dPdu );
    vector V = normalize (eye_coords);
    vector local_z = faceforward(normalize(normal), V,normalize(normal));
    vector local_y = cross(local_z,local_x);

    /* The first term is the diffuse component. This should be the
diffuse component in the Lafortune model multiplied by pi. */

    color ambientLight = float3(.1,.1,.1);
    // if defaultColor is black, we should skip terms containing it
    color finalColor = ambientLight * defaultColor;
    // color finalColor = float3(0,0,0);

    vector L = float3(1,1,2);
    vector Ln = normalize(L);
    color LightColor = float3(1,1,1);
    // diffuse term
    finalColor = finalColor + defaultColor * (max(0,dot(local_z,
Ln))*.5).xxx;
    // lafortune specular terms
    // Compute the terms
    // x = x_in * x_view + y_in * y_view
    // z = z_in * z_view
    float xt = dot(local_x,V) * dot(local_x,Ln) + dot(local_y,V) *
dot(local_y,Ln);
    float zt = -(dot(local_z,V) * dot(local_z,Ln));

    float tmp = 0;

```

```

#define DO_LOBE(var,lobeVect) float var = 0.0; \
    tmp = -xt*lobeVect.x + zt*lobeVect.y; \
    if (tmp > (0.001*lobeVect.z)) var = pow(tmp,lobeVect.z) ;

    DO_LOBE(fr0,lobe0R)
    DO_LOBE(fg0,lobe0G)
    DO_LOBE(fb0,lobe0B)
    DO_LOBE(fr1,lobe1R)
    DO_LOBE(fg1,lobe1G)
    DO_LOBE(fb1,lobe1B)
    DO_LOBE(fr2,lobe2R)
    DO_LOBE(fg2,lobe2G)
    DO_LOBE(fb2,lobe2B)
    float atten = dot(local_z,Ln);
    finalColor = finalColor + (LightColor *
float3(fr0+fr1+fr2,fg0+fg1+fg2,fb0+fb1+fb2)) * atten;

    // second lightsource
#ifdef TWO_LIGHTS
    L = float3(-2,-.5,5);
    Ln = normalize(L);
    LightColor = float3(1,0.9,0.6);
    finalColor = finalColor + defaultColor * (max(0,dot(local_z,
Ln))*0.5).xxx;
    xt = dot(local_x,V) * dot(local_x,Ln) + dot(local_y,V) *
dot(local_y,Ln);
    zt = -(dot(local_z,V) * dot(local_z,Ln));
// exactly the same as DO_LOBE() but without the declaration of "var"
#define DO_LOBE_NEXT(var,lobeVect) var = 0.0; \
    tmp = -xt*lobeVect.x + zt*lobeVect.y; \
    if (tmp > (0.001*lobeVect.z)) var = pow(tmp,lobeVect.z) ;
    DO_LOBE_NEXT(fr0,lobe0R)
    DO_LOBE_NEXT(fg0,lobe0G)
    DO_LOBE_NEXT(fb0,lobe0B)
    DO_LOBE_NEXT(fr1,lobe1R)
    DO_LOBE_NEXT(fg1,lobe1G)
    DO_LOBE_NEXT(fb1,lobe1B)
    DO_LOBE_NEXT(fr2,lobe2R)
    DO_LOBE_NEXT(fg2,lobe2G)
    DO_LOBE_NEXT(fb2,lobe2B)
    atten = dot(local_z,Ln);
    finalColor = finalColor + (LightColor *
float3(fr0+fr1+fr2,fg0+fg1+fg2,fb0+fb1+fb2)) * atten;
#endif /* TWO_LIGHTS */

    //
    // mirror term (cube map)
    //
    vector R = reflect(V, local_z); // "local_z" is the same as a
    typical"Nf"
    // let's soften this up a little -- it's not a perfect mirror
    vector dxr = ddx(R)*6.0;
    vector dyr = ddy(R)*6.0;
    color mirrorColor = f3texCUBE(env_map, R,dxr,dyr);

    float eta = 1/1.5; // refraction index ratio
    float f = schlick_fresnel(V, local_z, eta);
    finalColor = finalColor + (mirrorColor * f); // not quite
    right... but maybe okay for lafortune?

```

```
// final output to framebuffer

    myFo O;
#ifdef CORRECTED_COLOR_SAMPLES
    float cr = dot(colorMatrixR,finalColor);
    float cg = dot(colorMatrixG,finalColor);
    float cb = dot(colorMatrixB,finalColor);
    O.COL = float4( cr, cg, cb, 1 );
#else /* !CORRECTED_COLOR_SAMPLES */
    O.COL = float4( finalColor.x, finalColor.y, finalColor.z, 1 );
#endif /* !CORRECTED_COLOR_SAMPLES */
    return O;
}

// eof
```

Sample: Assembly Code

In comparison with the Cg version, the assembly-language program for the same shader is more than 500 instructions in length. The portion of the program shown here is only the first 100 lines:

```
!!FP1.0
# NV_fragment_program generated by NVIDIA Cg compiler
# cgc version 1.1.0000 NDA Release, build date Jul 30 2002 15:13:03
# command line args: -profile fp30 -o vinyl.fp vinyl.cg
#vendor NVIDIA Corporation
#version 1.0.1
#profile fp30
#program main
#semantic main.env_map
#semantic main.noise_map
#var float4 IN.WPOS : $vin.WPOS : WPOS : 0 : 1
#var float4 IN.COL0 : $vin.COL0 : COL0 : 0 : 1
#var float4 IN.COL1 : $vin.COL1 : COL1 : 0 : 1
#var float4 IN.TEX0 : $vin.TEX0 : TEX0 : 0 : 1
#var float4 IN.TEX1 : $vin.TEX1 : TEX1 : 0 : 1
#var float4 IN.TEX2 : $vin.TEX2 : TEX2 : 0 : 1
#var float4 IN.TEX3 : $vin.TEX3 : TEX3 : 0 : 1
#var float4 IN.TEX4 : $vin.TEX4 : TEX4 : 0 : 1
#var float4 IN.TEX5 : $vin.TEX5 : TEX5 : 0 : 1
#var float4 IN.TEX6 : $vin.TEX6 : TEX6 : 0 : 1
#var float4 IN.TEX7 : $vin.TEX7 : TEX7 : 0 : 1
#var texobjCUBE env_map : : texunit 0 : 1 : 1
#var texobj3D noise_map : : texunit 1 : 2 : 1
#var float4 COL : $vout.COL : COL : -1 : 1
MOVR R1.xyz, f[TEX2].xyzz;
DP3R R0.x, R1.xyzz, R1.xyzz;
RSQR R0.x, R0.x;
MULR R0.xyz, R0.xxxx, R1.xyzz;
MOVR R3.xyz, -R0.xyzz;
MOVR R1.xyz, f[TEX7].xyzz;
DP3R R0.x, R1.xyzz, R1.xyzz;
RSQR R0.x, R0.x;
MULR R13.xyz, R0.xxxx, R1.xyzz;
MOVR R1.xyz, f[TEX2].xyzz;
DP3R R0.x, R1.xyzz, R1.xyzz;
RSQR R0.x, R0.x;
```

```

MULR R0.xyz, R0.xxxx, R1.xyzz;
DP3R R1.x, R0.xyzz, R13.xyzz;
MINR R2.x, R1.x, {0}.x;
ADDR R1.x, {1}.x, -R2.x;
MULR R1.xyz, R1.xxxx, R3.xyzz;
MULR R0.xyz, R0.xyzz, R2.xxxx;
ADDR R0.xyz, R0.xyzz, R1.xyzz;
DP3R R1.x, R13.xyzz, R0.xyzz;
ADDR R1.x, {1}.x, -R1.x;
LG2R R1.x, R1.x;
MULR R1.x, {5}.x, R1.x;
EX2R R3.x, R1.x;
ADDR R1.x, {0.66666669}.x, {1}.x;
RCPR R2.x, R1.x;
ADDR R1.x, {0.66666669}.x, -{1}.x;
MULR R1.x, R1.x, R2.x;
LG2R R1.x, R1.x;
MULR R1.x, {2}.x, R1.x;
EX2R R1.x, R1.x;
ADDR R2.x, {1}.x, -R1.x;
MULR R2.x, R2.x, R3.x;
ADDR R2.x, R1.x, R2.x;
DP3R R3.x, R0.xyzz, R13.xyzz;
MULR R1.xyz, {2}.xxxx, R0.xyzz;
MULR R1.xyz, R1.xyzz, R3.xxxx;
ADDR R1.xyz, R13.xyzz, -R1.xyzz;
DDYR R3.xyz, R1.xyzz;
MULR R4.xyz, R3.xyzz, {6}.xxxx;
DDXR R3.xyz, R1.xyzz;
MULR R3.xyz, R3.xyzz, {6}.xxxx;
TXD R1.xyz, R1.xyzz, R3, R4, TEX0, CUBE;
MULR R1.xyz, R1.xyzz, R2.xxxx;
DP3R R2.x, {-2, -0.5, 5}.xyzz, {-2, -0.5, 5}.xyzz;
RSQR R2.x, R2.x;
MULR R3.xyz, R2.xxxx, {-2, -0.5, 5}.xyzz;
DP3R R5.x, R0.xyzz, R3.xyzz;
MOVR R6.x, {0}.x;
MOVR R2.x, {-1.01684, 1.00132, 180.181}.y;
DP3R R7.x, R0.xyzz, R3.xyzz;
DP3R R4.x, R0.xyzz, R13.xyzz;
MULR R4.x, R4.x, R7.x;
MOVR R11.x, -R4.x;
MULR R7.x, R11.x, R2.x;
MOVR R8.x, {-1.01684, 1.00132, 180.181}.x;
MOVR R2.x, f[TEX0].x;
DDYR R4.x, R2.x;
MOVR R2.x, f[TEX0].x;
DDXR R2.x, R2.x;
ADDR R4.x, R2.x, R4.x;
MOVR R2.x, f[TEX0].y;
DDYR R9.x, R2.x;
MOVR R2.x, f[TEX0].y;
DDXR R2.x, R2.x;
ADDR R2.x, R2.x, R9.x;
MOVR R4.y, R2.xxxx;
MOVR R4.z, {0}.xxxx;
DP3R R2.x, R4.xyzz, R4.xyzz;
RSQR R2.x, R2.x;
MULR R4.xyz, R2.xxxx, R4.xyzz;
MOVR R9.xyz, R4.yzxx;

```

```
MOVR R2.xyz, R0.zxyz;  
MULR R9.xyz, R2.xyz, R9.xyz;  
MOVR R10.xyz, R4.zxyz;  
MOVR R2.xyz, R0.yzxx;
```

Appendix D. Vertex Shader Code Sample

Sample Program: Cg Listing

A single vertex shader, written using the Cg programming language, can be written for the four-bone shader example discussed earlier in this paper. The listing follows:



```
/******NVMH3*****/
Copyright NVIDIA Corporation 2002
TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THIS SOFTWARE IS PROVIDED
*AS IS* AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE.  IN NO EVENT SHALL NVIDIA OR ITS
SUPPLIERS
BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES
WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS
PROFITS,
BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY
LOSS)
ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF NVIDIA HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
*****/

struct vert2frag
{
    float4 hPosition      : HPOS;
    float4 color          : COL0;
};

struct app2vert
{
    float4 position       : ATTR0;
    float4 weights        : ATTR1;
    float4 normal         : ATTR2;
    float4 matrixIndices  : ATTR5;
    float4 numBones       : ATTR4;
};

vert2frag main(
                app2vert IN,
```

```
        uniform float4x4 modelViewProj : C0,
            const uniform float4 boneMatrices[90],
        uniform float4 color,
        uniform float4 lightPos)
{
    vert2frag OUT;

    float4 index = IN.matrixIndices;
    float4 weight = IN.weights;

    float4 position;
    float3 normal;

    float i;
    for (i = 0; i < IN.numBones.x; i = i+1)
    {
        // transform the offset by bone i
        position = position + weight.x * float4(
            dot(boneMatrices[index.x+0], IN.position),
            dot(boneMatrices[index.x+1], IN.position),
            dot(boneMatrices[index.x+2], IN.position), 1);

        normal = normal + weight.x * float3(
            dot(boneMatrices[index.x+0].xyz, IN.normal.xyz),
            dot(boneMatrices[index.x+1].xyz, IN.normal.xyz),
            dot(boneMatrices[index.x+2].xyz, IN.normal.xyz));

        // shift over the index variable
        index = index.yzwx;
        weight = weight.yzwx;
    }

    normal = normalize(normal);

    OUT.hPosition = mul(modelViewProj, position);
    OUT.color = dot(normal, lightPos.xyz) * color;
    return OUT;
}
```

Appendix E. Combining Multiple Textures

Sample Code

Shaders can handle multiple textures in one pass. The listings for the example discussed previously in this paper follow:



```
/******NVMH3*****/
Path: E:\nvidia\devrel\NVSDK\Common\media\programs
File: cg_multipaint.cg

Copyright NVIDIA Corporation 2002
TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THIS SOFTWARE IS PROVIDED
*AS IS* AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL NVIDIA OR ITS
SUPPLIERS
BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES
WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS
PROFITS,
BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY
LOSS)
ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF NVIDIA HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Comments:
  A simple method to encode multiple surface models encoded as maps.
  Besides "typical" values that are stored as maps (e.g., color and specular
  intensity), ALL parts of the BRDF are stored as maps, or as grayscale
  map channels, and sometimes along with extra data to define a range of
  possible values between "black" and "white" in that channel (so that we
  can cram the maximum number of values into the effective contrast range of
  an 8-bit mapi -- or even less! Control-channel maps are particularly
  well-suited to being palettized)

*****/
```

```

// FRAGMENT PROGRAM

// input -- same struct is the output from "cg_multipaintVP.cg"
struct MultiPaintV2F {
    float4 HPosition      : POSITION;      // clip space pos for rasterizer. Not
readable in frag prog
    float4 TexCoords      : TEXCOORD0; // base ST coordinates
    float3 OPosition      : TEXCOORD1; // Obj-coords location
    float3 Normal         : TEXCOORD2; // Eye-space normal
    float3 VPosition      : TEXCOORD3; // viewer pos in obj coordinates
    float3 T              : TEXCOORD4; // tangent in obj coordinates
    float3 B              : TEXCOORD5; // binormal in obj coordinates
    float3 N              : TEXCOORD6; // normal in obj coordinates
    float4 LightVec0      : TEXCOORD7; // light firection in obj coords
    float4 Color0         : COLOR0; // Color potentially passed from vertices
};

struct PixelOut {
    float4 COL;
    float DEPR;
};

//
// funcs //////////////////////////////////
//

//
// A handy way to visualize vectors at the surface, for debugging purposes..
// Not normally actually used in this program
//
float4 vector_as_color(float4 theVector)
{
    float4 nv = 0.5f+(0.5f*theVector);
    return nv;
}
// overloaded version for float3 vectors
float4 vector_as_color(float3 theVector)
{
    float4 nv = 0.5f+(0.5f*float4(theVector.x,theVector.y,theVector.z,0.0f));
    return nv;
}

////////////////////////////////////
// Actual Fragment Program Here //
////////////////////////////////////

// channels in our material map:
#define SPEC_STR x
#define METALNESS y
#define NORM_SPEC_EXPON z

// subfields in "SpecData"
#define MINPOWER x
#define MAXPOWER y
#define MAXSPEC z

```

```

// subfields in "ReflData"
#define FRESNEL_MIN x
#define FRESNEL_MAX y
#define FRESNEL_EXPON z
#define REFL_STRENGTH w

// subfields in "BumpData"
#define BUMP_SCALE x

PixelOut main(
    MultiPaintV2F IN,
    uniform sampler2D ColorMap : texunit0, // color
    uniform sampler2D MaterialMap : texunit1, // encodes
{specStrength,metalness,normalized_specExpon,0)
    uniform sampler2D NormalMap: texunit3, // tangent-space normals
    uniform samplerCUBE EnvMap : texunit2, // environment skybox
    uniform float4 SpecData, // components: {minpower,
maxPower,maxSpecStr,??}
    uniform float4 ReflData, // components: {fresMin,
fresMax,fresExpon,reflStrength}
    uniform float4 BumpData // components: {bumpScale,?,?,??}
) {
    PixelOut OUT;
    float4 surfCol = f4tex2D(ColorMap,IN.TexCoords.xy);
    float4 material = f4tex2D(MaterialMap,IN.TexCoords.xy);
    float3 Nt = f3tex2D(NormalMap,IN.TexCoords.xy) - float3(0.5f,0.5f,0.5f);
    float specStr = material.SPEC_STR * SpecData.MAXSPEC;
    float specPower = SpecData.MINPOWER + material.NORM_SPEC_EXPON *
(SpecData.MAXPOWER-SpecData.MINPOWER);

    float3 Vn = -normalize(IN.VPosition - IN.OPosition);
    float3 Ln = normalize(IN.LightVecO).xyz;
    float3 Nb = normalize(BumpData.BUMP_SCALE * (Nt.x * IN.T + Nt.y * IN.B) +
(Nt.z * IN.N));
    float diff = max(0.0f,-dot(Ln,Nb));
    float4 diffResult = diff * surfCol;
    float isLit = (diff > 0.0f) ? 1.0f : 0.0f;
    float3 Hn = normalize(Vn+Ln);
    float spec = pow(abs(dot(Hn,Nb)),specPower) * specStr;
    float4 WHITE = float4(1f,1f,1f,1f);
    float4 specCol = lerp(WHITE,surfCol,material.METALNESS);
    float4 specResult = (spec * isLit) * specCol;
    float3 reflVect = reflect(Vn,Nb);
    float4 reflColor = f4texCUBE(EnvMap,reflVect);
    float fakeFresnel = ReflData.FRESNEL_MIN + ReflData.FRESNEL_MAX *
pow((1.0f-dot(-Vn,IN.N)),ReflData.FRESNEL_EXPON);
    float4 paintShine = fakeFresnel * reflColor;
    float4 metalShine = surfCol * reflColor;
    float4 shineCol = ReflData.REFL_STRENGTH *
lerp(paintShine,metalShine,material.METALNESS);
    float4 finalColor = diffResult + shineCol;
    // finalColor = vector_as_color(Ln);
    finalColor = specResult + diffResult + shineCol;
    finalColor.w = 1.0f;
    OUT.COL = finalColor;
    return OUT;
}

```

```

/*****NVMH3****
Path:  NVSDK\Common\media\programs
File:  cg_multipaintVP.cg

Copyright NVIDIA Corporation 2002
TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THIS SOFTWARE IS PROVIDED
*AS IS* AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE.  IN NO EVENT SHALL NVIDIA OR ITS
SUPPLIERS
BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES
WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS
PROFITS,
BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY
LOSS)
ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF NVIDIA HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Comments:
    Based on cg_fp30setup.cg

*****/

// define inputs from vertex buffer
struct appin : application2vertex
{
    float4 Position      : POSITION;
    float4 UV            : TEXCOORD0;
    float4 Tangent       : BLENDWEIGHT;
    float4 Binormal      : DIFFUSE;
    float4 Normal        : NORMAL;
};

// output -- same struct is the input to "cg_multipaint.cg"
struct MultiPaintV2F {
    float4 HPosition     : POSITION; // clip space pos for rasterizer. Not
readable in frag prog
    float4 TexCoords     : TEXCOORD0; // base ST coordinates
    float3 OPosition     : TEXCOORD1; // Obj-coords location
    float3 Normal        : TEXCOORD2; // Eye-space normal
    float3 VPosition     : TEXCOORD3; // viewer pos in obj coordinates
    float3 T             : TEXCOORD4; // tangent in obj coordinates
    float3 B             : TEXCOORD5; // binormal in obj coordinates
    float3 N             : TEXCOORD6; // normal in obj coordinates
    float4 LightVec0     : TEXCOORD7; // light firection in obj coords
    float4 Color0        : COLOR0; // Color potentially passed from vertices
};

```

```
MultiPaintV2F main(appin IN,
    uniform float4x4 ModelViewProj : C0,
    uniform float4x4 ModelViewIT   : C4,
    uniform float4x4 ModelView     : C8,
    uniform float4x4 ModelViewI    : C12,
    uniform float4  TexRepeats,
    uniform float4  ViewerPos,
    uniform float4  LightVec)    // in EYE coords
{
    MultiPaintV2F OUT;
    OUT.HPosition = mul(ModelViewProj, IN.Position);    // clip space for
rasterizer use
    OUT.OPosition = IN.Position.xyz;    // obj space -- just pass through
    OUT.Normal = normalize(mul(ModelViewIT, IN.Normal).xyz); // xf to view-
space
    OUT.TexCoords = IN.UV * TexRepeats;
    OUT.N = normalize(IN.Normal.xyz);    // obj space
    OUT.T = IN.Tangent.xyz; // obj space
    OUT.B = IN.Binormal.xyz;    // obj space
    OUT.VPosition = mul(ModelViewI, float4(0,0,0,1)).xyz; // xfrom from eye
coords to obj coords
    OUT.LightVecO = mul(ModelViewI, LightVec);    // eye to obj space
    // OUT.Color0 = IN.Color;
    return OUT;
}
```

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are registered trademarks and CineFX is a trademark of NVIDIA Corporation.

Microsoft, DirectX, Windows, and the Windows logo are registered trademarks of Microsoft Corporation. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

Copyright NVIDIA Corporation 2002.



NVIDIA.

**NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com**