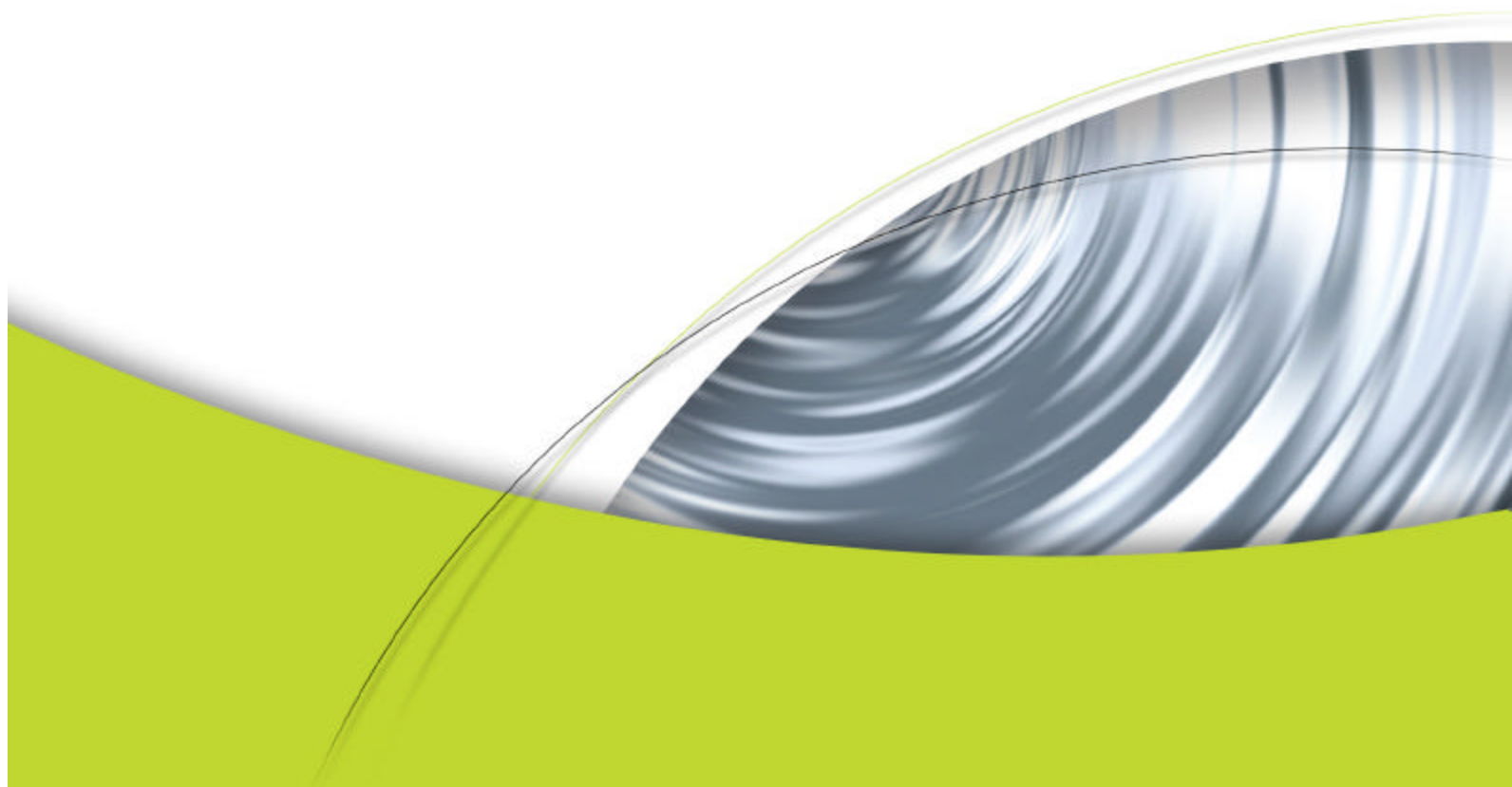




## Descrizione tecnica

### **Shader CineFX di NVIDIA**

Programmabilità cinematografica  
per effetti visivi stupefacenti



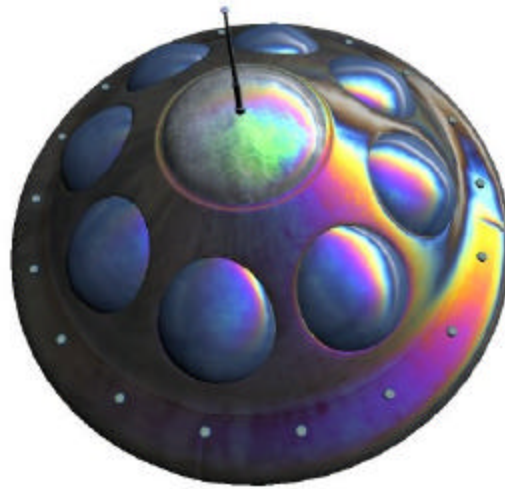


## Espansione del mondo dei shader per pixel e vertex

L'ultima generazione di GPU NVIDIA® ha introdotto una nuova era di effetti visivi di tipo cinematografico. La potenza e la precisione di queste nuove GPU offre la possibilità di creare grafica cinematografica in tempo reale e di portare sul desktop 128 bit di colore effettivo per la prima volta in assoluto. Queste importanti innovazioni spostano l'accento dal semplice fill rate dei pixel al più sofisticato pixel shading. Sfruttando il nuovo motore NVIDIA CineFX™, i programmatori possono ottenere:

- Un maggiore livello di precisione e 128 bit di colore effettivi, passando da una modalità di precisione all'altra all'interno di un solo programma di shading.
- La possibilità di scrivere programmi di shading più lunghi che incorporano numerosi effetti supplementari.
- Branching e looping dinamico per un controllo del flusso notevolmente migliorato.
- Cicli di sviluppo degli shader più brevi con il linguaggio di grafica Cg.

Questo documento tecnico illustra il nuovo livello di precisione e le modifiche apportate agli shader per pixel e vertex. Queste innovazioni permettono shading e illuminazione più realistiche per i personaggi, un'animazione dei personaggi più simile al vero ed effetti e visuali sofisticati applicabili all'intera scena in tempo reale.



(© 2002 NVIDIA Corporation)

**Figura 1. I nuovi shader semplificano la creazione degli effetti speciali come questa tecnica di rifrazione a pellicola sottile.**

## Miglioramento della precisione

“L'avvento della precisione in virgola mobile a 32 bit dei pixel rende possibile la creazione di immagini di altissima qualità. Effetti volumetrici efficaci — nebbia a livello del suolo, foschia sferica, sprite che si dissolvono in modo fluido invece di essere tagliati dalla geometria del mondo virtuale — si possono basare sul buffering z e il riutilizzo per pixel in virgola mobile a 32 bit. Le formule precise di attenuazione dell'illuminazione per pixel si possono basare sulla trasmissione delle posizioni delle sorgenti luminose a registri vettoriali in virgola mobile a 32 bit. Inoltre, le modalità in virgola mobile a 16 e 32 bit consentono un bump mapping di qualità molto superiore. Con soli 8 bit per componente, erano visibili difetti evidenti e mappe di protuberanze non normalizzate.” *Tim Sweeney, Epic Games, Inc*

I formati in virgola mobile a 16 e 32 bit (FP16 ed FP32) incorporati nelle GPU GeForce FX offrono agli sviluppatori la flessibilità necessaria per creare immagini di qualità sensazionale. La modalità FP32 offre il massimo di qualità dell'immagine, fornendo una precisione a 128 bit sull'intera pipeline grafica e il colore a 128 bit effettivi per i pixel shader. La modalità FP16 fornisce un equilibrio ottimale di qualità dell'immagine e prestazioni. In effetti, il formato FP16 offre esattamente lo stesso formato e precisione con cui gli studi cinematografici Industrial Light & Magic e Pixar Animation producono i propri film ed effetti speciali.



(Immagine presa da Elder Scrolls III: Morrowind, per cortesia di Bethesda Softworks, Inc.)

**Figura 2. L'acqua increspata con riflessi e rifrazioni necessita di una precisione elevata per una migliore qualità visiva e di una maggiore libertà da limiti e vincoli.**

Grazie alle modalità FP32 ed FP16, gli sviluppatori sono liberi di passare da un formato all'altro all'interno di un unico programma di shading, con la possibilità di utilizzare il formato più adatto a ogni particolare calcolo. Per esempio, alcune azioni quali l'indicizzazione di una texture ad alta risoluzione possono essere conseguite solo in modo ottimale usando un formato in virgola mobile a 32 bit. Se la texture è più grande di  $1024 \times 1024$  ( $210 \times 210$ , che richiede almeno 10 bit di mantissa per coordinata della texture), lo sviluppatore deve usare la FP32 per accedere a tutti i dati. Altri calcoli possono essere eseguiti in modo accurato usando la FP16, e possono beneficiare dalla velocità di esecuzione massimizzata garantita da questo livello di precisione.

Per una panoramica completa della precisione, e dei problemi risultanti da una carenza di precisione, riesaminare il documento tecnico di NVIDIA denominato "Grafica di alta precisione: Qualità cinematografica del colore sul PC."

---

## Elaborazione Vertex

Grazie all'espansione della gamma di capacità di elaborazione dei vertex, il motore NVIDIA CineFX offre ai programmatori la potenza necessaria a sviluppare letteralmente qualsiasi tipo di effetto immaginabile — con tecniche di programmazione decisamente semplificate. Per l'elaborazione vertex, il motore NVIDIA CineFX:

- ❑ **Infrange le limitazioni esistenti:** il numero di istruzioni supportate è aumentato da 128 a 65.536 grazie all'uso di branching dipendente dai dati e di un numero maggiore di istruzioni, registri e costanti.
- ❑ **Offre un maggiore controllo del flusso:** loop e branch dinamici che offrono la possibilità di variazioni in qualsiasi fase del flusso; introduzione di funzioni di chiamata e ritorno; possibilità di invocazione di uscita anticipata dall'elaborazione di vertex in caso di terminazione del programma.
- ❑ **Introduce nuove capacità:** codici di condizione per componente e maschere di programmazione.
- ❑ **Evoluzione a un gruppo di istruzioni avanzate:** nuove istruzioni e capacità includono branching (BRA), funzioni trigonometriche ad alta precisione (COS, SIN) e funzioni di esponenziazione e logaritmiche ad alta precisione (EX2, LG2 e altre).

Un esempio semplice della potenza incredibile di questo flessibile modello di programmazione è costituito dalle tavolozze a matrice per lo skinning dell'animazione dei personaggi. Con il modello Microsoft® DirectX® 8 (DX8), l'animazione dei personaggi richiedeva la programmazione di più shader, uno per ogni tipo di skinning previsto. Per esempio, se un modello faceva uso di vertici che potevano essere influenzati da un numero massimo di quattro ossa, lo sviluppatore avrebbe dovuto programmare sino a quattro shader separati, uno per ogni combinazione di ossa (vertice interessato da uno, due, tre o quattro ossa). Il modello DX8 esigeva una segmentazione in poligoni con lo stesso numero di ossa e il rendering nello stesso passaggio di tutte le sezioni del modello con uno stesso numero di ossa (vedere la figura 3). Alternativamente, lo sviluppatore avrebbe potuto utilizzare sempre lo shader a quattro ossa, per una soluzione lenta ma semplice.



(© 2002 NVIDIA Corporation)

### Figura 3. Sviluppo di modelli basati su poligoni

Con le API DirectX 8, i personaggi (come quello sulla sinistra) venivano creati sviluppando modelli basati su poligoni e quindi scrivendo shader per calcolare l'effetto delle ossa su ogni vertice del modello. Per conferire il massimo realismo ai movimenti complessi, numerosi vertici devono essere influenzati da un numero di ossa variabile. Per creare questo risultato con le API DirectX 8 si richiedono la segmentazione complessa del modello (non basata sul modello) e diversi programmi shader. L'immagine sulla destra mostra le porzioni del modello che fanno uso di un programma shader a due ossa. Sono richiesti shader separati per il rendering di altre parti del modello. (Vedere l'appendice D).

Utilizzando le più recenti funzionalità API permesse dalle generazioni più nuove di hardware della concorrenza, questa situazione migliora, sia pure non di molto. Ora è possibile sviluppare uno shader per rappresentare l'esempio che prevede sino a quattro ossa, ma dato che le API più recenti supportano solo una nozione molto limitata di branching, è possibile eseguirlo solo *per oggetto*, il che implica che il modello deve essere ancora suddiviso e disegnato in modo separato.

Il motore NVIDIA CineFX, con i suoi loop e branch pienamente generalizzati e la possibilità della dipendenza dai dati, dispone di una metodologia di programmazione molto più diretta. Viene sviluppato un unico shader per tutti i metodi e le operazioni di skinning, e dato che lo shader può eseguire il branching su base *per singolo vertice*, non è richiesta alcuna suddivisione del modello. L'esecuzione condizionale del loop per singolo vertice, rende inutile la segmentazione del modello, migliorando nettamente sia le prestazioni dell'applicazione che la produttività dello sviluppatore. Il listato del programma CineFX per questo esempio viene fornito nell'appendice D. Il codice per la sezione che esegue il loop condizionale è il seguente:

```

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;
}

```

## Elaborazione di vertici CineFX

Con il nuovo motore, gli shader vertex possono sfruttare quanto segue:

- **Sino a 65.536 istruzioni vertice eseguite per vertice (sino a 256 istruzioni statiche per shader)**

Il motore di shading CineFX offre una quantità di capacità di elaborazione vertex senza precedenti. Oltre al raddoppio della capacità di archiviazione delle istruzioni, l'aggiunta del controllo del flusso incrementa nettamente la quantità di calcolo effettivo che può verificarsi per ogni vertice. Questa flessibilità riduce il numero totale di vertex shader richiesto da un'applicazione.

- **Sino a 256 costanti vettoriali**

Il numero di registri di costanti disponibile nel vertex shader CineFX è più che raddoppiato — da 96 a 256 istruzioni quad. Questo incremento permette un numero sostanzialmente maggiore di matrici di ossatura per lo skinning e un numero molto più elevato di sorgenti luminose simultanee.

- **Sedici registri di vettori temporanei**

L'archiviazione temporanea dei registri è incrementata del 33%, da 12 a 16 unità. Questa archiviazione temporanea è particolarmente utile con i programmi di dimensioni maggiori supportati dal motore CineFX.

- **Sino a 64 loop separati**

Il motore di vertex shading CineFX permette di realizzare programmi più semplici grazie al supporto di looping e branching pienamente dipendente (che include loop e branch nidificati) con un massimo di 64 destinazioni di branching univoche in un solo programma di shading. Il looping su tutte le sorgenti luminose, e di conseguenza anche il branching sul tipo di luce appropriato, ora sono davvero semplici.

Il nuovo motore introduce diverse nuove funzionalità del vertex shader:

- ❑ **Codici condizionali per componente e maschere di scrittura**  
 I codici condizionali sono la parte macchina su cui poggia il branching dipendente dai dati, ma possono anche migliorare le prestazioni e semplificare il codice per le assegnazioni condizionali.
- ❑ **Chiamata e ritorno (subroutine)**  
 Oltre alle capacità di branching di CineFX, il processore di vertex offre il pieno supporto delle subroutine con semantica CALL/RETURN, con uno stack di chiamata di profondità massima di 4 unità.
- ❑ **Loop e branching per il controllo del flusso statico e dinamico**  
 Il looping e il branching completo generale (oltre ai riferimenti sui dati dipendenti) sono i fattori che rendono così flessibile e potente il motore di shading di vertex CineFX.

Grazie alle capacità supplementari e alle nuove funzionalità, il motore CineFX semplifica notevolmente lo sviluppo di vertex shader. Per esempio, permette l'uso di una sola chiamata di disegno per ogni personaggio animato. L'attivazione e la disattivazione delle luci non richiede più diversi programmi di shading; una semplice variazione di un registro di loop permette di completare l'operazione.

## Gruppo di istruzioni CineFX Vertex

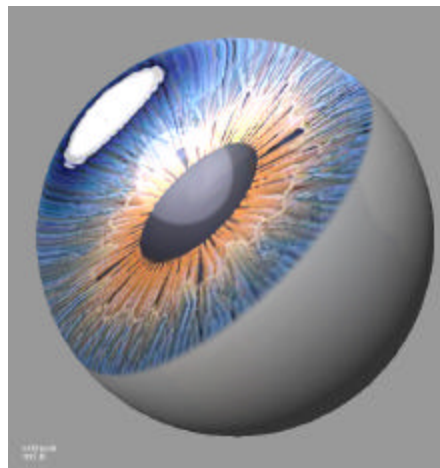
La tabella 1 illustra il nuovo gruppo di istruzioni di elaborazione dei vertici (dettagli nell'appendice A).

**Tabella 1. Gruppo di istruzioni di elaborazione vertex NVIDIA.**

Categorie	Istruzioni*
Somma e moltiplicazione	ADD, DP3, DP4, DPH, MAD, MOV, SUB
Matematica	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
Configurazione	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Branching	BRA, CAL, NOP
Registri degli indirizzi	ARL, ARR
Grafica	DST, LIT
Minimo/massimo	MAX, MIN

## Elaborazione pixel

Il motore NVIDIA CineFX porta il pixel shading al ruolo di protagonista della pipeline grafica e offre agli sviluppatori una vasta gamma di nuove funzioni per il controllo dei pixel e la produzione di effetti limitati solo dalla loro immaginazione (vedere la figura 4). Con l'ultima generazione di GeForce4, gli utenti ottengono capacità di programmazione dei vertici a livello di pixel e oltre.



(© 2002 NVIDIA Corporation)

### Figura 4. Raytracing nel pixel shader.

Il pixel shader valuta un settore di visualizzazione rifratto e lo interseca con un piano definito matematicamente allineato al modello. Il punto di intersezione viene utilizzato come lookup della texture per l'iride. I raggi che mancano la texture sono configurati su un colore alternativo appropriato (quello dello sfondo). La luce speculare è stata elaborata con una funzione `smoothstep()` per creare l'illusione di una sorgente circolare di dimensioni maggiori, e conferire un aspetto bagnato alla superficie.

Con il motore CineFX, gli sviluppatori possono programmare uno shader a passaggio singolo in grado di creare un aspetto misto. Precedentemente, erano necessari shader per la funzione di distribuzione di riflettanza multipla bidirezionale (BRDF), con la conseguente fusione dei risultati mediante una texture maschera o una concatenazione di istruzioni IF. L'esempio illustrato nella figura 5 mostra i risultati di uno shader a passaggio singolo che integra tutte le parti principali delle BRDF come diverse texture colorate. Si richiede un solo passaggio nello shader per creare l'aspetto miscelato. Questo permette a uno shader a passaggio singolo di contenere effetti di illuminazione ambientale, diffusa e speculare in un pacchetto compatto e di rapida esecuzione.

Il codice dello shader per la scrostatura della vernice da una superficie metallica è incluso nell'appendice E. Una sezione del codice include:

```

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)

**Figura 5. Texture multiple in uno shader a passaggio singolo**  
 pixel shader di nuova generazione permettono di unire più texture in un solo passaggio per un'esecuzione ottimizzata. Questo esempio dimostra la gestione di più effetti superficiali (scrostatura della vernice su una superficie metallica). (Vedere l'appendice E).

I principali vantaggi del nuovo motore di elaborazione pixel sono:

- ❑ **Introduzione di un nuovo gruppo di istruzioni per il pixel shading:** le istruzioni precedentemente riservate all'elaborazione vertex ora sono disponibili per il pixel shading, ampliate con le istruzioni necessarie all'elaborazione pixel.
- ❑ **Rimozione delle limitazioni esistenti:** i programmi possono ora essere più lunghi (sino a 1.024 istruzioni) con un massimo di 16 texture per pixel e livelli illimitati di lookup di texture dipendenti.
- ❑ **Notevole espansione del numero di operazioni pixel:** sino a 1.024 operazioni pixel; definizione per componente; maschere di programmazione condizionale per componente; filtri di texture arbitrari e altre istruzioni avanzate. Supporto di otto istruzioni DirectX 8. L'uso delle più recenti funzionalità API concesso dalle generazioni più nuove di hardware della concorrenza permette un certo miglioramento grazie al supporto di un massimo di 64 istruzioni. Il motore NVIDIA CineFX, tuttavia, grazie ad un tetto massimo di 1.024 istruzioni, supporta programmi di shading di lunghezza estrema, sviluppati appositamente per realizzare effetti stupefacenti e possibilità di shading davvero sensazionali.

- **Incremento dell'archiviazione di programmi frammentati:**  
 l'archiviazione nella memoria video, a differenza dei programmi vertex, permette di ridurre i costi di gestione di innumerevoli frammenti di programma.

## Gruppo di istruzioni per pixel di CineFX

Il nuovo gruppo di istruzioni di elaborazione dei pixel è fornito nella tabella 2. Per dettagli in merito alle variazioni del gruppo di istruzioni del pixel shader, fare riferimento all'appendice B.

**Tabella 2. Gruppo di istruzioni di elaborazione dei pixel NVIDIA.**

Categorie	Istruzioni
Aggiunta e moltiplicazione	ADD, LRP, DP3, DP4, LRP, MAD, MOV, SUB
Texturing	TEX, TXD, TXP
Derivate parziali	DDX, DDY
Calcoli matematici	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, NRM, POW, RCP, RSQ, SIN
Configurazione	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Grafica	DST, LIT, RFL
Minimo/massimo	MAX, MIN
Macro (imitazione delle funzionalità del vertex shader)	SINCOS, CRS
Compressione	PK2H, PK2US, PK4B, PK4UB, PK4UBG, PK16
Decompressione	UP2H, UP2US, UP4B, UP4UB, UP4UBG, UP16
Eliminazione	KIL

## Nuove operazioni sui pixel

Le principali nuove operazioni sui pixel di CineFX includono:

- **Massimo di 16 mappe texture.** Il motore NVIDIA CineFX permette il fetching di un massimo di 16 mappe di texture univoche in un singolo programma di shader per pixel. Queste texture possono essere una qualsiasi definizione di proprietà di superficie o secondaria quali le mappe di protuberanze, le mappe di spostamenti, le mappe di lucidità/specularità, le mappe ambiente, le mappe ombre e le mappe albedo.
- **Massimo di 1024 istruzioni texture per shader.** Le precedenti architetture limitano rigidamente il numero di mappe texture univoche con il numero di fetch texture disponibili. Il motore NVIDIA CineFX supera questa limitazione e permette un massimo di 1024 istruzioni di fetch texture in un singolo shader, con un sourcing massimo di 16 texture uniche. Questo consente di creare una vasta gamma di nuovi effetti che si fondano sull'uso di più texture:
  - **Ombre morbide.** Ora è possibile creare ombre morbide prelevando un numero arbitrariamente grande di campioni da una mappa ombre e usandoli per generare un risultato ombre filtrato.

- **Effetti di post-elaborazione del frame buffer.** Si possono creare diversi effetti interessanti prelevando più campioni dal frame buffer. Ora sono possibili sfocature, aloni ed effetti di rendering non fotorealistico quali l'ombreggiatura da cartone animato e gli stili pittorici.
- **Filtri complessi.** È possibile un filtraggio di qualità superiore sui lookup delle texture. Un filtro bicubico, per esempio, richiede 16 campioni dalla stessa texture.
- **Tablelle di lookup.** Certe funzioni possono essere codificate in texture e cercate dal pixel shader. Le funzioni per la normalizzazione dei vettori, il disturbo e l'illuminazione possono tutte essere codificate nelle texture ed è possibile prelevare campioni dalla texture ogni volta che si deve valutare la funzione.
- **Un massimo di 1024 istruzioni di colore.** Dato che il vero rendering cinematografico spesso richiede una grande quantità di calcoli a livello di singolo pixel, il motore NVIDIA CineFX supporta sino a 1024 istruzioni di colore arbitrarie in un pixel shader, ovviando in tal modo ai problemi legati al numero massimo di istruzioni. Alcuni degli effetti che richiedono un gran numero di istruzioni:
  - **Rendering del volume.** L'esecuzione di algoritmi di ray-tracing per il calcolo di effetti quali fumo, fuoco, pellicce e prati erbosi richiedono numerose istruzioni. Il numero di istruzioni viene spesso incrementato ulteriormente quando gli effetti di ray-marching sono ricorsivi. Per esempio, si consideri il caso dell'estroffessione di raggi da ciascun punto di campionamento verso una luce per raccogliere termini di ombreggiatura. In tal caso non è inusuale che si rendano necessarie centinaia di istruzioni.
  - **Texturing procedurale.** La generazione di texture procedurali in tempo reale spesso si avvale di un numero piuttosto ampio di istruzioni. L'antialias analitico di queste texture procedurali aggiunge altre istruzioni da calcolare per singolo pixel.
  - **Illuminazione complessa/più luci** I modelli di illuminazione più complessi possono perfezionare drasticamente il realismo delle immagini renderizzate, e fanno frequentemente uso di un numero elevatissimo di istruzioni, molto maggiore di quello dei più semplici modelli tradizionali. Per esempio, il modello di illuminazione Oren-Nayar, che modella con precisione le superfici ruvide diffuse, impiega un numero di istruzioni *dieci volte* superiore a quello del modello tradizionale Lambertian di illuminazione diffusa. La lunghezza incrementata del programma permette di inserire più luci in un singolo pixel shader, piuttosto di eseguire diversi passaggi sulla scena per ottenere lo stesso effetto come era necessario fare in precedenza.
- **Sino a 64 posizioni di archiviazione temporanea.** La possibilità di eseguire una vasta quantità di istruzioni (sia relative al colore che alle texture) necessita di un ampio numero di registri temporanei per contenere i risultati intermedi. Sino a 32 registri FP32 o 64 registri FP16 sono supportati per l'archiviazione temporanea (si tratta di registri a 4 componenti).

- **Swizzling** Il motore NVIDIA CineFX supporta lo swizzling completamente flessibile di componenti per gli operandi delle istruzioni. Questo permette una maggiore flessibilità e migliori opportunità di ottimizzazione. Per esempio, la comune operazione matematica nota come prodotto vettoriale prevede solo due istruzioni grazie a un pizzico di swizzling ben applicato:
- MUL r0, r1.yzxw, r2.zxyw  
MAD r0, -r2.yzxw, r1.zxyw, r0
- **Maschere di programmazione condizionale.** La mascheratura flessibile di programmazione condizionale permette il branching di semplici predicati, che prevedono il calcolo di entrambi i percorsi di un branch e la scelta di una sola delle alternative. La flessibilità del motore CineFX a tale scopo offre diverse opportunità di ottimizzazione al programmatore, o al compilatore, nel caso di un linguaggio di alto livello come il Cg.

Il motore NVIDIA CineFX supporta un gran numero di istruzioni pixel shader avanzate:

- **DDX, DDY:** Per il computing delle derivate dello spazio su schermo di un valore arbitrario in relazione a x ed y, rispettivamente, queste istruzioni forniscono la possibilità di accedere a informazioni sui pixel circostanti e sono utilissime per diversi effetti quali lo shading per cartoon e l'antialias.
- **TXD:** Questa istruzione permette il fetching da una texture e fornisce valori personalizzati per le derivate parziali delle coordinate della texture in relazione alle coordinate della finestra x e y. Questi parziali sono quindi utilizzati nei normali calcoli LOD, fornendo un controllo senza precedenti sul LOD.
- **SIN, COS:** Funzionalità trigonometrica ad alta precisione che non esisteva a livello di vertice nelle precedenti generazioni dell'hardware. Con il motore CineFX, la funzionalità è disponibile per singolo pixel.
- **PK, UP (e varianti):** Queste istruzioni permettono ai programmatori di “comprimere” e “decomprimere” tipi dati più ingombranti in tipi più compatti. Per esempio, 16 valori a 8 bit possono essere compressi in un singolo output a 128 bit (che normalmente archivierebbe quattro valori da 32 bit). In seguito, questi 16 valori possono essere nuovamente letti e decompressi nei registri. Questa capacità può essere utilizzata per archiviare più di quattro attributi per singolo pixel nei casi quali la gestione di una normale di vettore, di un colore di texture diffuso, di un colore di texture speculare, e di profondità ad alta precisione.



(© 2002 NVIDIA Corporation)

**Figura 6. Modello di alieno.**

Questo lungo pixel shader implementa nodi Lafortune Phong basati su misurazioni effettive di alluminio, modulate con effetto fresnel per creare l'aspetto del vinile. (Codice sorgente: vedere l'appendice C).

## Conclusione

Il motore CineFX rappresenta un importante passo avanti in termini di capacità degli shader pixel e vertex e permette la produzione di una nuova generazione di effetti cinematografici in tempo reale. La tabella 3 mostra un confronto fra le capacità delle piattaforme precedenti e della nuova generazione.

**Tabella 3. Confronto fra GeForce4 Ti e GeForce FX**

	GeForce4 Ti	GeForce FX
<b>Superfici di ordine più elevato</b>		
Mappatura dello scostamento della geometria	-	✓
<b>Vertex Shader</b>	1.1	2.0+
Istruzioni max	128	65536
Max istruzioni statiche	128	256
Max. costanti	96	256
Registri temporanei	12	16
Max. loop	0	256
Controllo del flusso statico	-	✓
Controllo del flusso dinamico	-	✓
<b>Pixel Shader</b>	1.1	2.0+
Mappe texture	4	16
Max. istruzioni texture	4	1024
Max. istruzioni colore	8	1024
Max archiviazione temp	-	64
Tipo dati	INT	FP
Precisione dei dati	32 bit	128 bit

# Appendice A.

## Variazioni del gruppo di istruzioni Vertex

**Tabella 4. Gruppo di istruzioni di elaborazione dei vertex NVIDIA.**

Categorie	Istruzioni*
Aggiunta e moltiplicazione	ADD, DP3, DP4, DPH, MAD, MOV, SUB
Calcoli matematici	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
Configurazione	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Branching	BRA, CAL, NOP
Registri degli indirizzi	ARL, ARR
Grafica	DST, LIT
Minimo/massimo	MAX, MIN

### Variazioni dei valori del vertex shader

- ❑ Archiviazione di 256 istruzioni di programma (invece di 128)
- ❑ 256 costanti (invece di 96)
- ❑ Registro indirizzo vettoriale (era scalare)
- ❑ Numero massimo di istruzioni eseguibili per shader ampliato a 65.536.

## Appendice B. Variazioni del gruppo di istruzioni pixel

**Tabella 5. Gruppo di istruzioni di elaborazione dei pixel NVIDIA.**

Categorie	Istruzioni
Aggiunta e moltiplicazione	ADD, LRP, DP3, DP4, LRP, MAD, MOV, SUB
Texturing	TEX, TXD, TXP
Derivate parziali	DDX, DDY
Calcoli matematici	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, NRM, POW, RCP, RSQ, SIN
Configurazione	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Grafica	DST, LIT, RFL
Minimo/massimo	MAX, MIN
Compressione	PK2H, PK2US, PK4B, PK4UB, PK4UBG, PK16
Decompressione	UP2H, UP2US, UP4B, UP4UB, UP4UBG, UP16
Eliminazione	KIL

### Pixel Shader 2.0:

- ❑ Registri e istruzioni possono essere 12 bit a punto fisso, 16 bit in virgola mobile, o 32 bit in virgola mobile.
- ❑ Qualsiasi numero di fetch di texture da un massimo di 16 texture uniche.
- ❑ 1.024 istruzioni per passaggio di rendering.
- ❑ 8 coordinate texture (sino a 16 texture attive).
- ❑ Se la destinazione è una superficie mobile, allora il valore mobile viene “convertito in larghezza” per corrispondere alla destinazione di rendering, e quindi viene archiviato. (Non è ammessa alcuna fusione di superfici mobili).
- ❑ Se la sorgente è una superficie mobile, non può essere eseguito alcun filtraggio all'interno dell'unità pixel (nessun valore mobile per filtro bi-lineare).
- ❑ Le conversioni di tipo e larghezza non hanno alcun costo in termini di prestazioni.
- ❑ Tutti i pixel di un batch sono eseguiti nello stesso numero di cicli di clock.

---

## Campionatori di texture

Gli shader pixel CineFX possono sfruttare i 16 campionatori di texture. I programmatori possono designare quale campionario e quali coordinate accoppiare per eseguire un fetch di texture. Ogni coordinata di texture non è più accoppiata con la texture specifica corrispondente. Pertanto, i fetch possono essere eseguiti da 16 texture differenti usando solo una coordinata di texture. In modo analogo, i fetch possono essere eseguiti da otto diverse coordinate non inalterate di una texture.

# Appendice C.

## Esempi di codice di pixel shader

---

### Esempio di programma: Versione Cg



Il pixel shader usato per creare la texture simile al vinile per l'alieno è stato programmato sia in linguaggio assembly che nel linguaggio di programmazione Cg. Segue il listato del programma Cg:

```
/******NVMH3*****/
Copyright NVIDIA Corporation 2002
SINO AL MASSIMO PREVISTO DALLA LEGGE APPLICABILE, QUESTO SOFTWARE È FORNITO
*NELLO STATO IN CUI SI TROVA* E NVIDIA E I SUOI FORNITORI RIFIUTANO QUALSIASI
GARANZIA, ESPlicita O IMPLICITa, AD INCLUSIONE MA SENZA LIMITARSI AD ESSE, DELLE
GARANZIE IMPLICITe DI COMMERCIALIZZABILITÀ E IDONEITÀ A UN PARTICOLARE SCOPO. IN
NESSUN CASO NVIDIA O I SUOI FORNITORI SARANNO RESPONSABILI PER NESSUN DANNO
SPECIALE, INCIDENTALE, INDIRETTO, O CONSEQUENZIALE DI QUALSIVOGLIA NATURA (AD
INCLUSIONE, SENZA ALCUNA LIMITAZIONE, DEI DANNI PER LA PERDITE DI PROFITTI
COMMERCIALI, PER INTERRUZIONE DELLE ATTIVITÀ, PER LA PERDITA DI INFORMAZIONI
COMMERCIALI O QUALSIASI ALTRA PERDITA PECUNIARIA) INSORGENTI DALL'USO O
DALL'IMPOSSIBILITÀ DI AVVALERSI DEL PRESENTE SOFTWARE, ANCHE OVE
NVIDIA SIA STATA AVVERTITA DELLA EVENTUALE POSSIBILITÀ DI TALI DANNI.

Commenti:
* "lafortune" shading a base fisica con parametri per alluminio e una
* o due luci integrate. Dato che lo shading per l'alluminio si affida
* a un termine di riflessione, è stata aggiunta anche una mappa cubica.
*****/
#define color float3
#define vector float3
//
// OPZIONI DI COMPILAZIONE
//
// annullare la definizione per vedere i risultati con una sola sorgente
luminosa
#define TWO_LIGHTS
// Per correggere le differenze tra le sensibilità di colore nelle
// misure originarie e i tipici componenti di colore RGB,
// attivare CORRECTED_COLOR_SAMPLES per usare il prodotto del punto
con le righe della matrice
```

```

// #define CORRECTED_COLOR_SAMPLES

//
// INIZIO DEL PROGRAMMA
//
myFo main(vf30 IN,
          uniform texobjCUBE env_map,
          uniform texobj3D noise_map)
{
    /* parametri attualmente hard-coded */
    /* BRDF predefinito: alluminio */
    /* questo in effetti sarebbe nero, ma fingiamo che sia un colore
lievemente meno definito */
    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 */

    /* spazio tangente espresso nel sorrente sistema di coordinate */
    /* Caricare vettore unitario nella direzione del parametro "u" */
    /* Usare per caricare un sistema di coordinate locale con local_z
come la normale. */
    vector STDir = {ddx(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);

    /* Il primo termine è il componente diffuso. Questo dovrebbe essere
il componente diffuso nel modello Lafortune moltiplicato per pi. */

    color ambientLight = float3(.1,.1,.1);
    // se defaultColor è nero, si dovrebbero saltare i termini che lo
contengono
    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);
    // termine diffuso
    finalColor = finalColor + defaultColor * (max(0,dot(local_z,
Ln))*0.5).xxx;
    // termini speculari lafortune

```

```

// Calcola i termini
// 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;

// seconda sorgente luminosa
#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));
// esattamente lo stesso di DO_LOBE() ma senza la dichiarazione di "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 */

//
// termine riflesso (mappa cubica)
//
vector R = reflect(V, local_z); // "local_z" è lo stesso di un
tipico "Nf"
// ammorbidiamolo un pochino -- non è uno specchio perfetto

```

```

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; // rapporto indice di rifrazione
float f = schlick_fresnel(V, local_z, eta);
finalColor = finalColor + (mirrorColor * f); // non proprio
corretto... ma forse è okay per lafortune?

// output finale al frame buffer

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

```

## Esempio: di codice Assembly

Rispetto alla versione in Cg, il programma in linguaggio assembly dello stesso shader è decisamente più lungo e sorpassa le 500 istruzioni. La porzione che segue contiene solo le prime 100 righe del programma:

```

!!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.xyz, R1.xyz;
RSQR R0.x, R0.x;
MULR R0.xyz, R0.xxxx, R1.xyz;
MOVR R3.xyz, -R0.xyz;
MOVR R1.xyz, f[TEX7].xyz;
DP3R R0.x, R1.xyz, R1.xyz;
RSQR R0.x, R0.x;
MULR R13.xyz, R0.xxxx, R1.xyz;
MOVR R1.xyz, f[TEX2].xyz;
DP3R R0.x, R1.xyz, R1.xyz;
RSQR R0.x, R0.x;
MULR R0.xyz, R0.xxxx, R1.xyz;
DP3R R1.x, R0.xyz, R13.xyz;
MINR R2.x, R1.x, {0}.x;
ADDR R1.x, {1}.x, -R2.x;
MULR R1.xyz, R1.xxxx, R3.xyz;
MULR R0.xyz, R0.xyz, R2.xxxx;
ADDR R0.xyz, R0.xyz, R1.xyz;
DP3R R1.x, R13.xyz, R0.xyz;
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.xyz, R13.xyz;
MULR R1.xyz, {2}.xxxx, R0.xyz;
MULR R1.xyz, R1.xyz, R3.xxxx;
ADDR R1.xyz, R13.xyz, -R1.xyz;
DDYR R3.xyz, R1.xyz;
MULR R4.xyz, R3.xyz, {6}.xxxx;
DDXR R3.xyz, R1.xyz;
MULR R3.xyz, R3.xyz, {6}.xxxx;
TXD R1.xyz, R1.xyz, R3, R4, TEX0, CUBE;
MULR R1.xyz, R1.xyz, R2.xxxx;
DP3R R2.x, {-2, -0.5, 5}.xyz, {-2, -0.5, 5}.xyz;
RSQR R2.x, R2.x;
MULR R3.xyz, R2.xxxx, {-2, -0.5, 5}.xyz;
DP3R R5.x, R0.xyz, R3.xyz;
MOVR R6.x, {0}.x;
MOVR R2.x, {-1.01684, 1.00132, 180.181}.y;
DP3R R7.x, R0.xyz, R3.xyz;
DP3R R4.x, R0.xyz, R13.xyz;
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.xyzz, R9.xyzz;
MOVR  R10.xyz, R4.zxyz;
MOVR  R2.xyz, R0.yzxx;

```

# Appendice D.

## Esempio di codice del vertex shader

---

### Esempio di programma: listato in Cg



Un singolo vertex shader, programmato con il linguaggio Cg, per l'esempio di shader per quattro ossa discusso in precedenza in questo documento. Segue il listato:

```
/******NVMH3*****/
Copyright NVIDIA Corporation 2002
SINO AL MASSIMO PREVISTO DALLA LEGGE APPLICABILE, QUESTO SOFTWARE È FORNITO
*NELLO STATO IN CUI SI TROVA* E NVIDIA E I SUOI FORNITORI RIFIUTANO QUALSIASI
GARANZIA, ESPLICITA O IMPLICITA, AD INCLUSIONE MA SENZA LIMITARSI AD ESSE, DELLE
GARANZIE IMPLICITE DI COMMERCIALIZZABILITÀ E IDONEITÀ A UN PARTICOLARE SCOPO. IN
NESSUN CASO NVIDIA O I SUOI FORNITORI SARANNO RESPONSABILI PER NESSUN DANNO
SPECIALE, INCIDENTALE, INDIRETTO, O CONSEGUENZIALE DI QUALSIVOGLIA NATURA (AD
INCLUSIONE, SENZA ALCUNA LIMITAZIONE, DEI DANNI PER LA PERDITE DI PROFITTI
COMMERCIALI, PER INTERRUZIONE DELLE ATTIVITÀ, PER LA PERDITA DI INFORMAZIONI
COMMERCIALI O QUALSIASI ALTRA PERDITA PECUNIARIA) INSORGENTI DALL'USO O
DALL'IMPOSSIBILITÀ DI AVVALERSI DEL PRESENTE SOFTWARE, ANCHE OVE
NVIDIA SIA STATA AVVERTITA DELLA EVENTUALE POSSIBILITÀ DI TALI DANNI.
*****/

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;
}

```

# Appendice E.

## Combinazione di più texture



### Esempio di codice

Gli shader possono gestire più texture in un solo passaggio. Seguono i listati dell'esempio discusso in precedenza in questo documento:

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

Copyright NVIDIA Corporation 2002

SINO AL MASSIMO PREVISTO DALLA LEGGE APPLICABILE, QUESTO SOFTWARE È FORNITO \*NELLO STATO IN CUI SI TROVA\* E NVIDIA E I SUOI FORNITORI RIFIUTANO QUALSIASI GARANZIA, ESPLICITA O IMPLICITA, AD INCLUSIONE MA SENZA LIMITARSI AD ESSE, DELLE GARANZIE IMPLICITE DI COMMERCIALIZZABILITÀ E IDONEITÀ A UN PARTICOLARE SCOPO. IN NESSUN CASO NVIDIA O I SUOI FORNITORI SARANNO RESPONSABILI PER NESSUN DANNO SPECIALE, INCIDENTALE, INDIRETTO, O CONSEGUENZIALE DI QUALSIVOGLIA NATURA (AD INCLUSIONE, SENZA ALCUNA LIMITAZIONE, DEI DANNI PER LA PERDITE DI PROFITTI COMMERCIALI, PER INTERRUZIONE DELLE ATTIVITÀ, PER LA PERDITA DI INFORMAZIONI COMMERCIALI O QUALSIASI ALTRA PERDITA PECUNIARIA) INSORGENTI DALL'USO O DALL'IMPOSSIBILITÀ DI AVVALERSI DEL PRESENTE SOFTWARE, ANCHE OVE NVIDIA SIA STATA AVVERTITA DELLA EVENTUALE POSSIBILITÀ DI TALI DANNI.

Commenti:

un semplice metodo per codificare più modelli di superfici come mappe. Oltre ai valori "consueti" archiviati come mappe (per es., colore e intensità speculare), TUTTE le parti del BRDF sono archiviate come mappe, o come canali di mappa in scala di grigi, e talvolta assieme a dati supplementari per definire una gamma di valori possibili compresi tra il "nero" e il bianco di quel canale (in modo da poter inserire il massimo numero dei valori nella gamma di contrasto effettiva di una mappa a 8 bit - o anche meno! Le mappe dei canali di controllo sono particolarmente adatte alla trasformazione in tavolozze)

```
*****/
```

```

// FRAMMENTO DI PROGRAMMA

// input -- stessa struttura dell'output di "cg_multipaintVP.cg"
struct MultiPaintV2F {
    float4 HPosition      : POSITION;      // pos dello spazio di clip del
rasterizzatore. Illeggibile nel framm di prog
    float4 TexCoords      : TEXCOORD0; // coordinate base ST
    float3 OPosition      : TEXCOORD1; // posizione coordinate oggetto
    float3 Normal          : TEXCOORD2; // Normale spazio occhio
    float3 VPosition      : TEXCOORD3; // Pos vis nelle coordinate oggetto
    float3 T               : TEXCOORD4; // tangente nelle coordinate oggetto
    float3 B               : TEXCOORD5; // binormale nelle coordinate oggetto
    float3 N               : TEXCOORD6; // normale nelle coordinate oggetto
    float4 LightVec0      : TEXCOORD7; // direzione luce nelle coordinate oggetto
    float4 Color0         : COLOR0; // Colore potenzialmente passato dai vertici
};

struct PixelOut {
    float4 COL;
    float DEPR;
};

//
// funzioni //////////////////////////////////
//

//
// Utile metodo per visualizzare vettori come superfici, a fini di debugging...
// Solitamente non usato in questo programma
//
float4 vector_as_color(float4 theVector)
{
    float4 nv = 0.5f+(0.5f*theVector);
    return nv;
}
// versione sovraccaricata di vettori float3
float4 vector_as_color(float3 theVector)
{
    float4 nv = 0.5f+(0.5f*float4(theVector.x,theVector.y,theVector.z,0.0f));
    return nv;
}

////////////////////////////////////
// Frammento di programma inserito qui //
////////////////////////////////////

// canali nella mappa materiale:
#define SPEC_STR x
#define METALNESS y
#define NORM_SPEC_EXPON z

// sottocampi di "SpecData"
#define MINPOWER x
#define MAXPOWER y
#define MAXSPEC z

```

```

// sottocampi di "ReflData"
#define FRESNEL_MIN x
#define FRESNEL_MAX y
#define FRESNEL_EXPON z
#define REFL_STRENGTH w

// sottocampi di "BumpData"
#define BUMP_SCALE x

PixelOut main(
    MultiPaintV2F IN,
    uniform sampler2D ColorMap : texunit0, // colore
    uniform sampler2D MaterialMap : texunit1, // codifica
    {specStrength,metalness,normalized_specExpon,0}
    uniform sampler2D NormalMap : texunit3, // normali spazio tangente
    uniform samplerCUBE EnvMap : texunit2, // skybox ambiente
    uniform float4 SpecData, // componenti: {minpower,
maxPower,maxSpecStr,??}
    uniform float4 ReflData, // componenti: {fresMin,
fresMax,fresExpon,reflStrength}
    uniform float4 BumpData // componenti: {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****
Percorso: NVSDK\Common\media\programs
File: cg_multipaintVP.cg

```

Copyright NVIDIA Corporation 2002  
SINO AL MASSIMO PREVISTO DALLA LEGGE APPLICABILE, QUESTO SOFTWARE È FORNITO \*NELLO STATO IN CUI SI TROVA\* E NVIDIA E I SUOI FORNITORI RIFIUTANO QUALSIASI GARANZIA, ESPLICITA O IMPLICITA, AD INCLUSIONE MA SENZA LIMITARSI AD ESSE, DELLE GARANZIE IMPLICITE DI COMMERCIALIZZABILITÀ E IDONEITÀ A UN PARTICOLARE SCOPO. IN NESSUN CASO NVIDIA O I SUOI FORNITORI SARANNO RESPONSABILI PER NESSUN DANNO SPECIALE, INCIDENTALE, INDIRETTO, O CONSEGUENZIALE DI QUALSIVOGLIA NATURA (AD INCLUSIONE, SENZA ALCUNA LIMITAZIONE, DEI DANNI PER LA PERDITE DI PROFITTI COMMERCIALI, PER INTERRUZIONE DELLE ATTIVITÀ, PER LA PERDITA DI INFORMAZIONI COMMERCIALI O QUALSIASI ALTRA PERDITA PECUNIARIA) INSORGENTI DALL'USO O DALL'IMPOSSIBILITÀ DI AVVALERSI DEL PRESENTE SOFTWARE, ANCHE OVE NVIDIA SIA STATA AVVERTITA DELLA EVENTUALE POSSIBILITÀ DI TALI DANNI.

Commenti:  
Basato su cg\_fp30setup.cg

```


*****/

// definire gli input dal vertex buffer
struct appin : application2vertex
{
    float4 Position      : POSITION;
    float4 UV            : TEXCOORD0;
    float4 Tangent       : BLENDWEIGHT;
    float4 Binormal      : DIFFUSE;
    float4 Normal        : NORMAL;
};

// output -- stessa struttura dell'input a "cg_multipaint.cg"
struct MultiPaintV2F {
    float4 HPosition     : POSITION; // pos dello spazio di clip del
rasterizzatore. Illeggibile nel frammento di prog
    float4 TexCoords    : TEXCOORD0; // coordinate base ST
    float3 OPosition    : TEXCOORD1; // posizione coordinate oggetto
    float3 Normal       : TEXCOORD2; // Normale spazio occhio
    float3 VPosition    : TEXCOORD3; // Pos vis nelle coordinate oggetto
    float3 T            : TEXCOORD4; // tangente nelle coordinate oggetto
    float3 B            : TEXCOORD5; // binormale nelle coordinate oggetto
    float3 N            : TEXCOORD6; // normale nelle coordinate oggetto
    float4 LightVec0    : TEXCOORD7; // direzione luce nelle coordinate oggetto
    float4 Color0       : COLOR0; // Colore potenzialmente passato dai vertici
};

```

```
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)    // nelle coordinate OCCHIO
{
    MultiPaintV2F OUT;
    OUT.HPosition = mul(ModelViewProj, IN.Position);    // spazio di clip per
l'uso da parte del rasterizzatore
    OUT.OPosition = IN.Position.xyz;    // spazio ogg -- solo per
l'attraversamento
    OUT.Normal = normalize(mul(ModelViewIT, IN.Normal).xyz); // xf per lo spazio
di visualizzazione
    OUT.TexCoords = IN.UV * TexRepeats;
    OUT.N = normalize(IN.Normal.xyz);    // spazio ogg
    OUT.T = IN.Tangent.xyz; // spazio ogg
    OUT.B = IN.Binormal.xyz;    // spazio ogg
    OUT.VPosition = mul(ModelViewI, float4(0,0,0,1)).xyz; // xfrom dalle coord
occhio alle coord ogg
    OUT.LightVec0 = mul(ModelViewI, LightVec);    // spazio da occhio a ogg
    // OUT.Color0 = IN.Color;
    return OUT;
}
```



Le informazioni fornite sono ritenute accurate e affidabili. Tuttavia, NVIDIA Corporation non si assume alcuna responsabilità per le eventuali conseguenze derivanti dall'uso di tali informazioni o da qualsiasi violazione di brevetti o altri diritti di terze parti che possono conseguire dal loro uso. Non viene concessa alcuna licenza implicita o in altro modo in base a nessun brevetto o diritto di autore di proprietà di NVIDIA Corporation. Le specifiche tecniche menzionate nella presente pubblicazione sono soggette a modifica senza preavviso. Questa pubblicazione rimpiazza e sostituisce tutte le informazioni precedentemente fornite. Non si autorizza l'impiego dei prodotti di NVIDIA Corporation come componenti cruciali di dispositivi per il supporto vitale o per sistemi che non abbiano ricevuto l'espressa approvazione scritta di NVIDIA Corporation.

#### **Marchi**

NVIDIA e il logo NVIDIA sono marchi registrati e CineFX è un marchio di NVIDIA Corporation.

Microsoft, DirectX, Windows e il logo di Windows sono marchi registrati di Microsoft Corporation. Altri nomi di società e di prodotti possono essere marchi o marchi registrati dei rispettivi detentori.

#### **Diritti di autore**

Copyright NVIDIA Corporation 2002



**NVIDIA.**

**NVIDIA Corporation  
2701 San Tomas Expressway  
Santa Clara, CA 95050  
[www.nvidia.com](http://www.nvidia.com)**