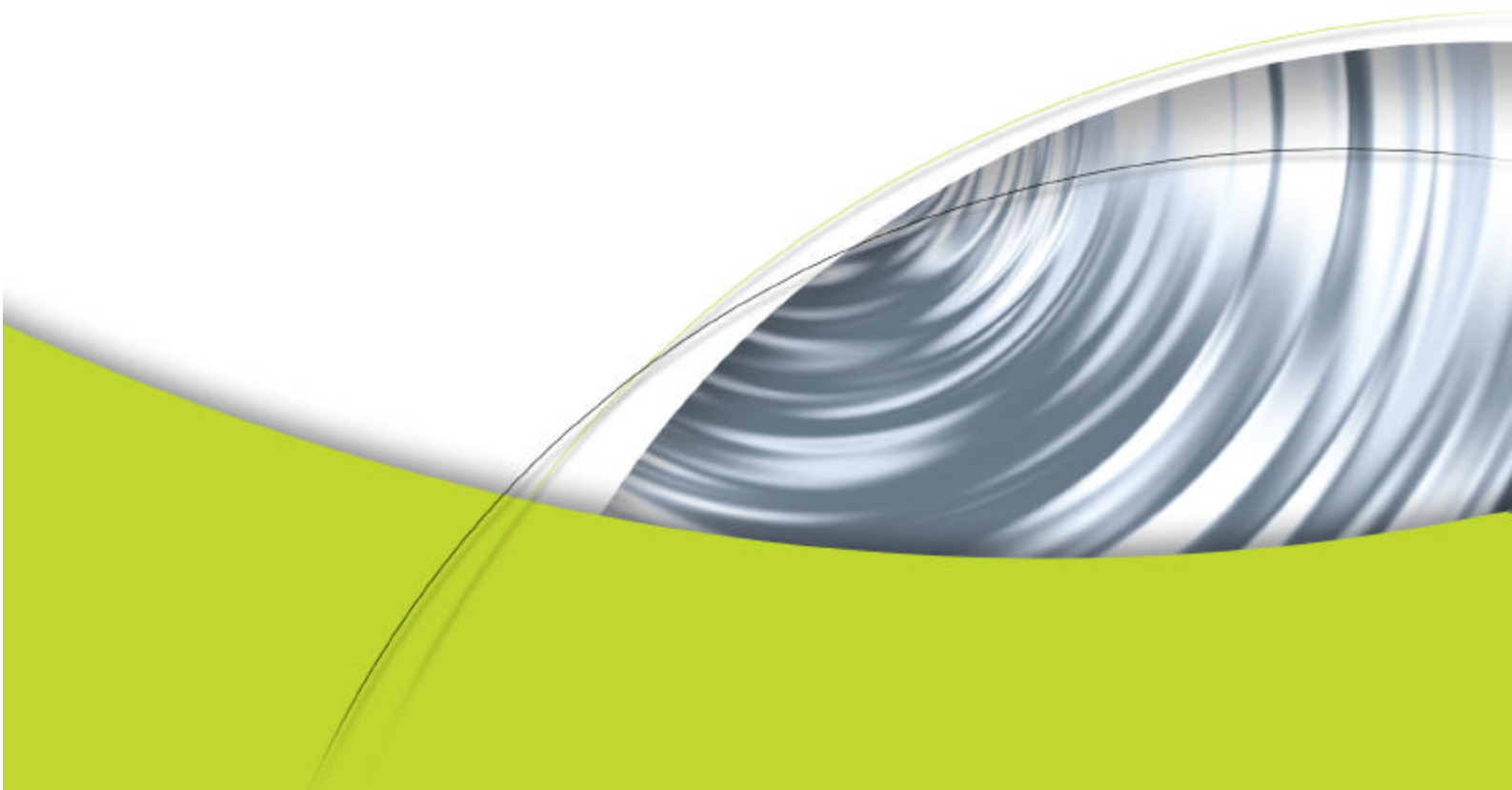




# Condensé technique

## **Shaders CineFX NVIDIA**

Une programmabilité  
cinématographique pour des effets  
visuels spectaculaires



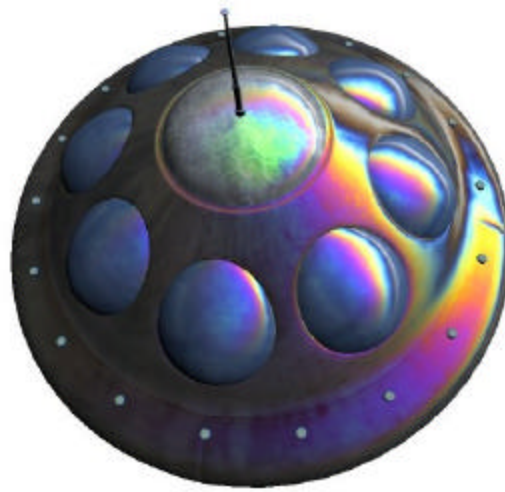


# Le monde des pixel shaders et des vertex shaders s'agrandit

La dernière génération des processeurs graphiques (GPU) NVIDIA® inaugure une nouvelle ère d'effets visuels cinématographiques. La puissance et la précision de ces nouveaux GPU permettent d'obtenir des graphiques cinématographiques en temps réel et, pour la première fois, dotent l'ordinateur de bureau de la couleur 128 bits. Ces avancées considérables amènent les professionnels à privilégier l'utilisation d'un ombrage de pixels sophistiqué au détriment du taux de remplissage de pixels conventionnel. En outre, en tirant parti du nouveau moteur CineFX™ de NVIDIA, les programmeurs bénéficient de :

- Niveaux de précision améliorés pour une couleur 128 bits vraie, en alternant les différents modes de précision dans un même shader.
- La possibilité d'écrire des shaders plus longs intégrant davantage d'effets.
- L'emploi de sauts et boucles dynamiques pour un flux de commande nettement amélioré.
- Cycles de mise au point de shaders plus courts grâce au langage graphique Cg.

Ce condensé technique décrit le nouveau niveau de précision et les modifications apportées aux spécifications des pixel shaders et des vertex shaders. Ces innovations permettent d'obtenir des ombres et un éclairage plus réaliste des personnages, des animations plus vraies ainsi que des effets et des aspects sophistiqués pouvant être appliqués en temps réel à l'ensemble de la scène.



(© 2002 NVIDIA Corporation)

**Figure 1. Les nouveaux shaders facilitent la création d'effets spéciaux, tels que cette technique de réfraction à couche mince.**

## Une précision accrue

« L'arrivée de la précision de pixel en virgule flottante de 32 bits permet de créer des images de haute qualité. Des effets de volume saisissants — nappes de brouillard, nuées d'insectes, images-objets qui s'effacent progressivement au lieu d'être brutalement découpées — peuvent être réalisés grâce à l'enregistrement dans un tampon et à la réutilisation des valeurs de profondeur par pixel adoptant le format en virgule flottante de 32 bits. Vous pouvez élaborer des formules d'atténuation de lumière par pixel précises qui réagissent à la position des sources de lumière dans des registres vectoriels en virgule flottante en 32 bits. Sans compter qu'une précision en virgule flottante en 16 et 32 bits autorise un bump mapping de qualité supérieure. Avec seulement 8 bits par composant, vous aviez des artefacts qui ne passaient pas inaperçus et des bump maps non-normalisées ». *Tim Sweeney, Epic Games, Inc.*

Les formats en virgule flottante en 16 et 32 bits (FP16 et FP32) inhérents au moteur CineFX de NVIDIA donnent aux développeurs la souplesse indispensable pour créer des images de la plus haute qualité. Le FP32 garantit une image d'une qualité extrême en amenant la précision 128 bits sur l'ensemble du pipeline graphique et en fournissant une vraie couleur 128 bits dans les pixel shaders. Le FP16 associe de manière optimale qualité d'image et performance. En fait, le FP16 correspond exactement au format et à la précision utilisés par les studios Industrial Light & Magic et Pixar Animation Studios pour réaliser leurs effets spéciaux et longs métrages.



(Image tirée du jeu Elder Scrolls III : Morrowind, reproduite avec la permission de Bethesda Softworks, Inc.)

**Figure 2. Les ondulations de la surface de l'eau riches de réflexions et réfractions exigent une précision élevée pour atteindre une qualité d'image de haut niveau et ne plus devoir obéir à des limites contraignantes.**

Avec le FP32 et le FP16, les développeurs sont libres d'alterner ces formats dans leurs applications en utilisant pour chaque calcul celui qui s'y prête le mieux. Par exemple, certaines actions telles que l'indexation dans une texture haute résolution ne peuvent être réalisées de façon optimale qu'en utilisant un format en virgule flottante en 32 bits. Si la texture dépasse  $1024 \times 1024$  ( $2^{10} \times 2^{10}$ , ce qui requiert au moins 10 bits de mantisse par coordonnée de texture), un développeur aura besoin du FP32 pour accéder à toutes les données. D'autres calculs en revanche pourront être effectués avec précision avec le FP16 et bénéficier ainsi de la vitesse d'exécution maximale rendue possible par ce niveau de précision.

Pour une vue d'ensemble complète de la précision et des problèmes susceptibles de résulter d'un manque de précision, veuillez consulter le document technique de NVIDIA intitulé « Graphiques haute précision : la couleur de qualité cinématographique sur les ordinateurs de bureau ».

---

## Le traitement des sommets

Grâce à ses fonctionnalités de traitement de sommets nettement améliorées, le moteur CineFX de NVIDIA donne aux programmeurs la puissance qui leur permettra de créer pratiquement tous les effets qui leur passeront par la tête, avec des techniques de programmation nettement simplifiées. Pour le traitement des sommets, le moteur CineFX de NVIDIA :

- ❑ **Élimine les limitations existantes** : le nombre des instructions prises en charge passe de 128 à 65 536 par le biais de l'utilisation de sauts dépendants des données et d'un plus grand nombre d'instructions, de registres et de constantes.
- ❑ **Offre un contrôle étendu** : les boucles et les sauts dynamiques permettent d'effectuer des modifications en avant et en arrière dans le flux ; des fonctions d'appel et de retour ont été introduites et le vertex shader peut également invoquer une fin anticipée.
- ❑ **Intègre de nouvelles fonctionnalités** : codes de conditions et masques d'écriture par composant.
- ❑ **Évolue vers un jeu d'instructions avancé** : de nouvelles instructions et fonctionnalités ont été ajoutées dont les sauts (BRA), les fonctions trigonométriques en haute précision (COS, SIN), les fonctions d'exponentiation et logarithmiques en haute précision (EX2, LG2 et autres).

Le « skinning » (formation de peau) par palette matricielle pour l'animation des personnages est un exemple simple de l'incroyable puissance de ce modèle de programmation extrêmement souple. Avec le modèle DirectX® 8 (DX8) de Microsoft®, l'animation des personnages nécessitait l'écriture de nombreux shaders, un par type de skinning. Par exemple, si un modèle utilisait des sommets pouvant être affectés par un à quatre os, le développeur devait écrire jusqu'à quatre shaders distincts, un pour chaque combinaison d'os (un os, deux os, trois os ou quatre os affectant un sommet). Afin de pouvoir réaliser cette opération dans DX8, le modèle devait être segmenté en polygones qui utilisaient le même nombre d'os, toutes les sections du modèle devant utiliser le même nombre d'os rendus dans le même passage (voir figure 3). Le développeur pouvait également recourir systématiquement au shader à quatre os qui offrait une solution plus simple mais plus lente.



(© 2002 NVIDIA Corporation)

### Figure 3. Développement de modèles basés sur des polygones

Avec DirectX 8, les personnages (tels que celui de gauche) sont créés en mettant au point des modèles basés sur des polygones et en écrivant ensuite des shaders pour calculer l'effet des os sur chaque sommet du modèle. Pour créer un mouvement complexe et réaliste, la plupart des sommets doivent être influencés par un nombre d'os variable. Une segmentation complexe du modèle (pas basée sur un modèle) ainsi que plusieurs shaders sont nécessaires pour obtenir ce résultat dans DirectX 8. L'image de droite montre les parties du modèle qui utilisent un vertex shader à deux os. Des shaders distincts sont requis pour obtenir le rendu des autres parties du modèle (voir annexe D).

Avec la dernière fonctionnalité API implémentée par le matériel concurrent de la dernière génération, cette situation s'améliore quelque peu. Un shader peut désormais être écrit pour représenter l'exemple impliquant jusqu'à quatre os, mais cette dernière API ne prenant en charge qu'une notion très limitée du branchement, qui ne peut être réalisé que *par objet*, le modèle doit être toujours découpé et dessiné séparément.

Le moteur CineFX de NVIDIA, grâce à ses boucles généralisées et à ses sauts qui peuvent être basés sur des données, offre une méthodologie de programmation beaucoup plus simple. Un shader peut désormais englober toutes les méthodes et opérations de skinning, et puisqu'il peut effectuer des sauts *par vertex*, il n'est plus nécessaire de démanteler le modèle. En exécutant la boucle de manière conditionnelle au niveau du sommet, la segmentation du modèle n'est plus obligatoire, ce qui améliore considérablement les performances de l'application ainsi que la productivité du développeur. La liste des programmes CineFX relatifs à cet exemple est fournie dans l'annexe D. Le code de la section exécutant la boucle conditionnelle est le suivant :

```

for (i = 0; i < IN.numBones.x; i = i+1)
{
    // transformer le décalage par position i de l'os
    = 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));

    // Déplacer la variable d'indice
    index = index.yzwx;
    weight = weight.yzwx;
}

```

## Le traitement des sommets avec CineFX

Grâce au nouveau moteur, les vertex shaders bénéficient des éléments suivants :

- ❑ **Jusqu'à 65 536 instructions exécutées par sommet (jusqu'à 256 instructions statiques par shader)**  
 Le moteur d'ombrage CineFX offre un nombre impressionnant de fonctions de traitement de sommets. Outre le doublage de la capacité de stockage d'instructions, le flux de commande supplémentaire augmente considérablement la quantité de calculs réels possible pour chaque sommet. Cette souplesse permet de réduire le nombre total des vertex shaders requis par une application.
- ❑ **Jusqu'à 256 constantes vectorielles**  
 Le nombre de registres de constantes disponibles dans le vertex shader CineFX a plus que doublé, passant de 96 à 256 mots quadruples ! Cette augmentation permet l'intégration d'un plus grand nombre de matrices d'os pour le skinning par palette matricielle et bien davantage de sources de lumière simultanées.
- ❑ **Seize registres vectoriels temporaires**  
 La capacité de stockage des registres temporaires est passée de 12 à 16, soit une augmentation de 33 %. Ce stockage temporaire s'avère particulièrement utile pour les programmes plus complexes pris en charge par le moteur CineFX.
- ❑ **Jusqu'à 64 boucles distinctes**  
 Le moteur d'ombrage de sommets CineFX permet également des programmes plus simples grâce à la prise en charge des boucles et des sauts entièrement dépendants (boucles et sauts imbriqués compris) avec jusqu'à 64 cibles de branchement distinctes dans un même shader. Effectuer un bouclage sur toutes les sources de lumière puis les aiguiller sur le type d'éclairage approprié s'effectue maintenant en un clin d'œil.

Le nouveau moteur intègre plusieurs nouvelles fonctionnalités de traitement des sommets :

- **Codes conditionnels et des masques d'écriture par composant**  
 Les codes conditionnels constituent la mécanique qui se cache derrière les sauts dépendant des données, mais ils peuvent également améliorer les performances et simplifier le code des tâches conditionnelles.
- **Appel et retour (sous-routines)**  
 Outre les fonctionnalités de branchement de CineFX, le processeur de sommets prend en charge l'ensemble de la sémantique APPEL/RETOUR de sous-routine, avec une pile d'une profondeur maximale de 4 appels.
- **Boucles et sauts pour un flux de commande statique et dynamique**  
 Le bouclage généralisé et le branchement (ainsi que la dépendance vis-vis des données) sont à l'origine de la souplesse et de la puissance du moteur d'ombrage de sommets CineFX.

Grâce à ses capacités améliorées et ses nouvelles fonctionnalités, le moteur CineFX facilite nettement l'écriture des vertex shaders. Par exemple, il devient possible de n'utiliser qu'un seul dessin pour obtenir un personnage entièrement animé. L'utilisation de plusieurs programmes d'ombrage n'est plus nécessaire pour l'activation et la désactivation des lumières : une simple modification de registre de boucle suffit.

## Jeu d'instructions des vertex shaders CineFX

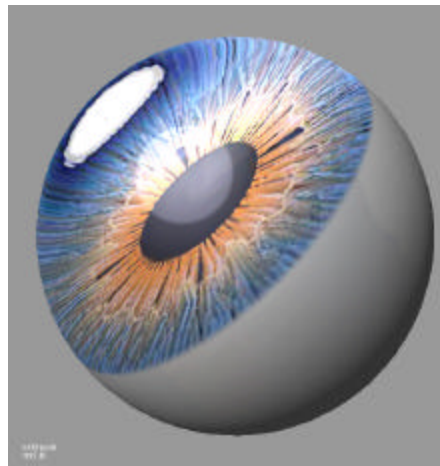
Le tableau 1 illustre le nouveau jeu d'instructions de traitement des sommets (détails disponibles dans l'annexe A).

**Tableau 1. Jeu d'instructions de traitement des sommets NVIDIA**

Catégories	Instructions*
Ajout et multiplication	ADD, DP3, DP4, DPH, MAD, MOV, SUB
Mathématiques	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
Paramétrage	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Sauts	BRA, CAL, NOP
Registres d'adresses	ARL, ARR
Orientée graphisme	DST, LIT
Minimum/maximum	MAX, MIN

## Le traitement des pixels

Le moteur CineFX de NVIDIA élève l'ombrage de pixels au rang d'élément programmable de choix du pipeline graphique et offre aux développeurs toute une panoplie de nouvelles fonctionnalités permettant de contrôler les pixels et de produire tous les effets possibles et imaginables (voir figure 4). Grâce à ces fonctionnalités de pointe, les utilisateurs bénéficient des capacités de programmation de sommets du GeForce4, au niveau du pixel et au-delà.



(© 2002 NVIDIA Corporation)

### Figure 4. Lancé de rayons et pixel shader

Le pixel shader évalue un vecteur de vue réfracté et en détermine le point d'intersection avec un plan défini mathématiquement aligné sur le modèle. Le point d'intersection est utilisé pour déterminer la couleur de l'iris. Si aucun objet n'intercepte le rayon la couleur est celle du fond. Ici, la mise en évidence spéculaire a été traitée avec une fonction smoothstep() pour créer l'illusion d'une source circulaire plus large, procurant à la surface cet aspect humide.

Grâce au moteur CineFX, les développeurs peuvent écrire un shader à un passage en mesure de créer un aspect mixte. Auparavant, plusieurs shaders à fonction de distribution de réflexion bidirectionnelle (bidirectional reflectance distribution function - BRDF) étaient nécessaires, et les résultats étaient combinés à l'aide d'une texture de masque ou enchaînés à l'aide d'instructions IF. L'exemple présenté dans la figure 5 illustre le résultat d'un shader à un passage intégrant l'ensemble des parties clés des BRDF telles que plusieurs textures peintes ; un seul passage dans le shader suffit à créer l'aspect mixte. Ceci permet d'obtenir un shader à un passage contenant des effets d'éclairage diffus, spéculaires et ambiants dans un ensemble compact et rapide à exécuter.

Le code du shader de cette peinture écaillée sur surface métallique est fourni dans l'annexe E. La section A de ce code inclut :

```

PixelOut main(
    MultiPaintV2F IN,
    uniform sampler2D ColorMap : texunit0, // couleur
    uniform sampler2D MaterialMap : texunit1, // codifiée
    {specStrength,metalness,normalized_specExpon,0)
    uniform sampler2D NormalMap : texunit3, // normales de
l'espace tangent
    uniform samplerCUBE EnvMap : texunit2, //
environnement skybox
    uniform float4 SpecData, // composants : {minpower,
maxPower,maxSpecStr,??}
    uniform float4 ReflData, // composants : {fresMin,
fresMax,fresExpon,reflStrength}
    uniform float4 BumpData // composants :
{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 = vecteur_comme_couleur(Ln);
    finalColor = specResult + diffResult + shineCol;
    finalColor.w = 1.0f;
    OUT.COL = finalColor;
    return OUT;
}

```



(© 2002 NVIDIA Corporation)

### Figure 5. Textures multiples dans un shader à un passage

Les pixels shaders de la future génération peuvent combiner plusieurs textures dans un seul shader à un passage pour une exécution optimisée. Cet exemple illustre le traitement d'effets de surface multiples (peinture s'écaillant sur une surface métallique). Voir annexe E.

Les principales caractéristiques du nouveau moteur de traitement de pixels sont les suivantes :

- ❑ **Introduction d'un nouveau jeu d'instructions pour l'ombrage des pixels** : les instructions auparavant réservées au traitement des sommets sont maintenant disponibles pour les pixels et s'ajoutent à celles spécifiques du traitement de pixels.
- ❑ **Suppression des limites existantes** : les programmes peuvent être plus longs (jusqu'à 1024 instructions) et proposer jusqu'à 16 textures par pixel ainsi que des niveaux illimités de valeurs de texture dépendantes.
- ❑ **Explosion du nombre d'opérations de pixels** : jusqu'à 1 024 opérations de pixel, mutation de pointeurs par composant, masques d'écriture par composant, filtres de texture arbitraires et autres instructions avancées. DirectX 8 prenait en charge huit instructions. Utiliser la dernière API mise en œuvre par le nouveau matériel de la concurrence, améliorerait un peu les choses puisque 64 instructions étaient prises en charge. Mais le moteur CineFX de NVIDIA et ses 1024 instructions sont les premiers à améliorer nettement la situation en prenant en charge des shaders extrêmement longs qui permettent d'obtenir des effets spectaculaires.
- ❑ **Amélioration de la capacité de stockage de fragments de programmes** : ces derniers sont stockés dans la mémoire vidéo, contrairement aux

vertex shaders, ce qui réduit les frais liés à la gestion d'un grand nombre de fragments de programmes.

## Jeu d'instructions des pixel shaders CineFX

Le nouveau jeu d'instructions de traitement des pixels est fourni dans le tableau 2. Pour plus de détails sur les modifications du jeu d'instructions des pixel shaders, reportez-vous à l'annexe B.

**Tableau 2. Jeu d'instructions du traitement de pixels NVIDIA**

Catégories	Instructions
Addition et multiplication	ADD, LRP, DP3, DP4, LRP, MAD, MOV, SUB
Texturation	TEX, TXD, TXP
Dérivées partielles	DDX, DDY
Mathématiques	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, NRM, POW, RCP, RSQ, SIN
Paramétrage	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Orientée graphisme	DST, LIT, RFL
Minimum/maximum	MAX, MIN
Macros (fonctionnalité de vertex shader synoptique)	SIN, COS, CRS
Empaquetage	PK2H, PK2US, PK4B, PK4UB, PK4UBG, PK16
Déempaquetage	UP2H, UP2US, UP4B, UP4UB, UP4UBG, UP16
Suppression	KIL

## Nouvelles opérations des pixel shaders

Les principales nouvelles opérations des pixel shaders CineFX sont les suivantes :

- **Jusqu'à 16 mappes de textures.** Le moteur CineFX de NVIDIA permet de charger jusqu'à 16 mappes de textures distinctes dans un même pixel shader. Ces textures peuvent être tout élément définissant des propriétés de surface sous-jacentes, comme des mappes de relief, des mappes de brillance/spécularité, des mappes d'environnement, des mappes d'ombres et des mappes d'albédo.
- **Jusqu'à 1 024 instructions de texture par shader.** Les architectures précédentes liaient étroitement le nombre de mappes de textures au nombre de chargements de texture disponibles. Le moteur CineFX de NVIDIA élimine cette restriction et permet d'inclure jusqu'à 1 024 instructions de chargement de textures dans un seul shader, fournissant jusqu'à 16 textures distinctes. Cela permet d'obtenir une myriade de nouveaux effets qui reposent sur l'accès à plusieurs textures :
  - **Ombres floues.** Des ombres floues peuvent à présent être créées en prenant un nombre arbitrairement grand d'échantillons à partir d'une mappe d'ombres et en les utilisant pour générer l'ombre filtrée résultante.
  - **Effets de post-traitement du tampon graphique.** Un certain nombre d'effets intéressants peuvent être créés en puisant divers échantillons de textures dans un tampon graphique. Des effets de flous, de halos et d'autres effets de rendu non photo-réalistes comme l'ombrage de dessins animés et des effets de peinture sont maintenant possibles.
  - **Filtres complexes.** Un filtrage de haute qualité peut être effectué sur les valeurs de texture. Un filtre bicubique requiert, par exemple, 16 échantillons de la même texture.

- **Tables de correspondance (LUT).** Certaines fonctions peuvent être codées sous forme de textures et recherchées depuis le pixel shader. Les fonctions employées pour la normalisation de vecteurs, le bruit et l'éclairage peuvent être codées sous forme de textures, et les échantillons pris d'une texture chaque fois que la fonction doit être évaluée.
- **Jusqu'à 1 024 instructions de couleur.** Le rendu cinématographique exigeant souvent une grande quantité de calculs au niveau du pixel, le moteur CineFX de NVIDIA prend en charge jusqu'à 1 024 instructions de couleur arbitraires dans un pixel shader, les programmeurs n'ont ainsi plus à se soucier du nombre d'instructions limite. Un échantillonnage d'effets exigeant un volume d'instructions important inclut :
  - **Rendu du volume.** L'exécution d'algorithmes d'adaptation de rayons pour créer des effets tels que la fumée, le feu, le poil ou l'herbe, requiert de nombreuses instructions. Le nombre d'instructions augmente encore lorsque les effets d'adaptation de rayons sont récursifs. Prenez à titre d'exemple le cas qui consiste à lancer des rayons au niveau de chaque point d'échantillon en direction d'une lumière pour recueillir des informations d'ombrage. Il n'est pas rare dans ce cas de devoir gérer des centaines d'instructions.
  - **Texturation procédurale.** La génération de textures procédurales en temps réel utilise souvent un grand nombre d'instructions. L'anticrénelage analytique de ces textures procédurales accroît encore le nombre d'instructions à calculer pour chaque pixel.
  - **Éclairage complexe/sources multiples.** Les modèles d'éclairage plus complexes peuvent améliorer considérablement le réalisme des images rendues et utilisent fréquemment un nombre d'instructions plus important que les modèles d'éclairage traditionnels plus simples. Par exemple, le modèle d'éclairage d'Oren-Nayar qui modélise de manière concise des surfaces diffuses vierges requiert *dix fois* plus d'instructions que l'éclairage diffus de Lambert traditionnel. La longueur accrue des programmes donne maintenant la possibilité de créer de nombreuses lumières dans un seul pixel shader, alors qu'il fallait auparavant plusieurs passages sur la scène pour obtenir le même effet.
- **Jusqu'à 64 emplacements de stockage temporaire.** Pour pouvoir exécuter des volumes importants d'instructions (couleur et texture), un grand nombre de registres temporaires destinés à conserver des résultats intermédiaires est nécessaire. Jusqu'à 32 registres FP32 ou 64 FP16 sont pris en charge pour le stockage temporaire (il s'agit de registres à 4 composants).
- **Mutation de pointeurs (swizzling).** Le moteur CineFX de NVIDIA prend en charge un swizzling extrêmement souple des composants pour les opérandes sur les instructions. Cette technique accroît la souplesse et élargit les possibilités d'optimisation. Par exemple, l'opération mathématique courante connue sous le nom de produit en croix ne nécessite que deux instructions avec un swizzling éclairé :
  - MUL r0, r1.yzxw, r2.zxyw
  - MAD r0, -r2.yzxw, r1.zxyw, r0
- **Masques d'écriture conditionnels.** Un masquage d'écriture conditionnel souple permet des sauts prévus simples, dans lequel les deux parties d'un branchement sont calculées et une seule sélectionnée. La souplesse du moteur CineFX offre à cet égard au programmeur ou au compilateur, dans le cas d'un langage de haut niveau comme Cg, un grand nombre de possibilités d'optimisation.

Le moteur CineFX de NVIDIA prend en charge un certain nombre d'instructions de pixel shader avancées :

- **DDX, DDY** : pour le calcul des dérivées de l'espace écran d'une valeur arbitraire par rapport à x et y respectivement, ces instructions permettent de mesurer l'accès aux informations relatives aux pixels environnants et sont très précieuses pour l'obtention de toute une série d'effets tels que l'ombrage de dessins animés et l'anticrénelage.
- **TXD** : cette instruction permet d'effectuer un chargement à partir d'une texture et d'attribuer des valeurs personnalisées aux dérivées partielles des coordonnées de texture pour les coordonnées de fenêtre x et y. Ces dérivées partielles sont ensuite utilisées dans les calculs LOD traditionnels, offrant ainsi un contrôle sans précédent sur LOD.
- **SIN, COS** : la fonctionnalité trigonométrique en haute précision n'existait pas au niveau des sommets dans les précédentes générations de matériel. Grâce au moteur CineFX, cette fonctionnalité est disponible pour chaque pixel.
- **PK, UP (et variantes)** : ces instructions permettent aux programmeurs d'« empaqueter » et de « déempaqueter » des types de données de plus petite taille dans des types plus volumineux. Par exemple, 16 valeurs de 8 bits peuvent être empaquétées dans une seule sortie 128 bits (qui stocke normalement quatre valeurs de 32 bits). Ultérieurement, ces 16 valeurs peuvent être de nouveau lues et déempaquetées dans des registres. Cette fonctionnalité peut être utilisée pour stocker plus de quatre attributs par pixel dans les cas du traitement d'une normale de vecteur, d'une couleur à la texture diffuse, d'une couleur de texture diffusée et d'une profondeur en haute précision.



(© 2002 NVIDIA Corporation)

**Figure 6. Modèle d'extra-terrestre.**


Ce long pixel shader met en œuvre les nœuds de Phong Lafortune basés sur des mesures réelles de l'aluminium, modulées avec Fresnel pour créer l'aspect du vinyle (code source : voir annexe C).

## Conclusion

Le moteur CineFX représente une avancée majeure dans le domaine des vertex shaders et des pixels shaders, et annonce une nouvelle génération d'effets cinématographiques en temps réel. Le tableau 3 présente une comparaison des caractéristiques des plates-formes actuelles et futures.

**Tableau 3. Comparaison du GeForce4 Ti et du GeForce FX**

	GeForce4 Ti	GeForce FX
<b>Surface d'ordre supérieur</b>		
Displacement mapping géométrique	-	✓
<b>Vertex shaders</b>	1.1	2.0+
Nbre max. d'instructions	128	65536
Instructions statiques max.	128	256
Nbre max. de constantes	96	256
Registres temporaires	12	16
Nbre max. de boucles.	0	256
Flux de commande statique	-	✓
Flux de commande dynamique	-	✓
<b>Pixels shaders</b>	1.1	2.0+
Mappes de texture	4	16
Nbre max. d'instructions de texture	4	1024
Nbre max. d'instructions de couleur	8	1024
Stockage temp. max.	-	64
Type de données	ENT	VF
Précision des données	32 bits	128 bits



## Annexe A. Modification du jeu d'instruction des vertex

**Tableau 4. Jeu d'instructions de traitement des vertex NVIDIA**

Catégories	Instructions*
Addition et multiplication	ADD, DP3, DP4, DPH, MAD, MOV, SUB
Mathématiques	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
Paramétrage	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Sauts	BRA, CAL, NOP
Registres d'adresses	ARL, ARR
Orientée graphisme	DST, LIT
Minimum/maximum	MAX, MIN

---

### Modification des chiffres des vertex shaders

- ❑ 256 instructions de programme stocké (contre 128)
- ❑ 256 constantes (contre 96)
- ❑ Registre d'adresses vectoriel (ils était scalaire auparavant)
- ❑ Le nombre maximal d'instructions pouvant être exécutées pour chaque shader est maintenant de 65 536.



## Annexe B. Modification du jeu d'instructions des pixels

**Tableau 5. Jeu d'instructions de traitement des pixels NVIDIA**

Catégories	Instructions
Addition et multiplication	ADD, LRP, DP3, DP4, LRP, MAD, MOV, SUB
Texturation	TEX, TXD, TXP
Dérivées partielles	DDX, DDY
Mathématiques	ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, NRM, POW, RCP, RSQ, SIN
Paramétrage	SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
Orientée graphisme	DST, LIT, RFL
Minimum/maximum	MAX, MIN
Empaquetage	PK2H, PK2US, PK4B, PK4UB, PK4UBG, PK16
Déempaquetage	UP2H, UP2US, UP4B, UP4UB, UP4UBG, UP16
Suppression	KIL

---

### Modifications des pixel shaders

- ❑ Les registres et les instructions peuvent adopter un format à virgule fixe 12 bits, virgule flottante 16 bits ou virgule flottante 32 bits.
- ❑ Nombre illimité de charges de texture jusqu'à 16 textures uniques.
- ❑ 1 024 instructions par passage de rendu.
- ❑ 8 coordonnées de texture (jusqu'à 16 textures actives).
- ❑ Si la cible est une surface flottante, la valeur flottante est alors « convertie en largeur » pour correspondre à la cible de rendu, puis stockée (aucun mélange n'est possible pour les surfaces flottantes).
- ❑ Si la source est une surface flottante, aucun filtrage ne peut être effectué au niveau des pixels (pas de valeurs flottantes de filtrage bilinéaire).
- ❑ Les conversions de type et de largeur sont libres.
- ❑ Tous les pixels d'un lot sont exécutés dans le même nombre de cycles d'horloge.

---

## Les échantillonneurs de texture

Les shaders CineFX bénéficient de 16 échantillonneurs de texture. Les programmeurs peuvent déterminer quel échantillonneur et quelle coordonnée associer pour charger une texture. Chaque coordonnée de texture n'est désormais plus associée à la texture spécifique correspondante. Par conséquent, les chargements peuvent être effectués à partir de 16 textures différentes en utilisant une coordonnée de texture. De la même manière, ils peuvent être obtenus à partir de huit coordonnées intactes différentes au sein d'une seule texture.

# Annexe C. Exemples de code d'un pixel shader

## Exemple de programme : version Cg



Le pixel shader utilisé pour créer la texture vinyle de l'extraterrestre a été écrit dans en langage assembleur et en Cg.  
Le programme en Cg est le suivant :

```
/******NVMH3*****/
Copyright NVIDIA Corporation 2002
DANS LES LIMITES DES TERMES PRÉVUS PAR LA LOI APPLICABLE, CE LOGICIEL EST
FOURNI *EN L'ÉTAT* ET NVIDIA AINSI QUE SES FOURNISSEURS DÉCLINENT TOUTE
GARANTIE, EXPRESSE OU TACITE, Y COMPRIS MAIS NON LIMITÉ À, TOUTE GARANTIE
IMPLICITE DE VALEUR MARCHANDE ET D'ADAPTABILITÉ À TOUT OBJECTIF PARTICULIER. EN
AUCUN CAS, NVIDIA OU SES FOURNISSEURS NE SERONT TENUS RESPONSABLES DE QUELQUE
DOMMAGE SPÉCIAL, FORTUIT OU INDIRECT QUE CE SOIT (Y COMPRIS, ET SANS RÉSERVE,
LES DOMMAGES CAUSÉS PAR LA PERTE DE PROFITS, L'INTERRUPTION DE L'ACTIVITÉ, LA
PERTE DES INFORMATIONS COMMERCIALES OU TOUTE AUTRE PERTE FINANCIÈRE) POUVANT
RÉSULTER DE L'UTILISATION DE OU DE L'INCAPACITÉ À UTILISER CE LOGICIEL, MÊME SI
NVIDIA A ÉTÉ PRÉVENUE DE LA POSSIBILITÉ DE TELS DOMMAGES.

Commentaires :
* ombrage physiquement basé sur la technique "lafortune" avec des paramètres
pour l'aluminium et une
*   ou deux lumières intégrées. Parce que l'ombrage de l'aluminium repose
*   sur un terme miroir, une mappe cubique a également été ajoutée.
*****/
#define color float3
#define vector float3
//
// OPTIONS DE TEMPS DE COMPILATION
//
// annulez cette définition afin de connaître les résultats avec une
simple lumière source
#define TWO_LIGHTS
// Pour résoudre les différences entre les sensibilités de couleurs
utilisées dans les mesures d'origine et les couleurs des composants
d'affichage
//   RVB classiques,
```

```

// activez CORRECTED_SAMPLE_COLORS pour utiliser les produits
scalaires avec des lignes de matrice
// #define CORRECTED_COLOR_SAMPLES

//
// DÉBUT DU PROGRAMME
//
myFo main(vf30 IN,
          uniform texobjCUBE env_map,
          uniform texobj3D noise_map)
{
    /* les paramètres sont actuellement câblés */
    /* BRDF par défaut : aluminium */
    /* il devrait en fait être noir mais disons que l'objet est un peu
sale */
    color defaultColor = float3(0.09,0.08,0.15);
    /* float3(cxy=direction, cz=échelle, 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_SAMPLE_COLORS
    color colorMatrixR = float3(1,0,0);
    color colorMatrixG = float3(0,1,0);
    color colorMatrixB = float3(0,0,1);
#endif /* CORRECTED_SAMPLE_COLORS */

    /* espace de la tangente exprimé dans le système de coordonnées
actuel */
    /* Obtenez le vecteur de l'unité dans la direction de paramètre "u"
*/
    /* Utilisez-le pour obtenir un système de coordonnées local en
choisissant local_z comme 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);

    /* Le premier terme est le composant diffus. Il s'agit du composant
diffus du modèle Lafortune multiplié par pi. */

    color ambientLight = float3(.1,.1,.1);
    // si la couleur par défaut est noire, nous devons ignorer les termes
la contenant
    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);
    // terme diffus

```

```

    finalColor = finalColor + defaultColor * (max(0,dot(local_z,
Ln))*0.5).xxx;
    // termes spéculaires de Lafortune
    // Calculez les termes
    // 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;

    // seconde source de lumière
#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));
    // exactement la même que DO_LOBE() mais sans l'instruction "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 */

    //
    // terme miroir (mappe cubique)
    //

```

```

    vector R = reflect(V, local_z); // "local_z" est identique au"Nf"
standard
    // lissons ceci un peu plus, ce n'est pas un miroir parfait
    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; // ratio d'indice de réfraction
    float f = schlick_fresnel(V, local_z, eta);
    finalColor = finalColor + (mirrorColor * f); // pas tout à fait
juste...mais peut-être OK pour Lafortune ?

// résultat final dans le tampon graphique

    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

```

## Exemple : code d'assemblage

Par rapport à la version en Cg, le programme en langage assembleur utilisé pour le même shader contient plus de 500 instructions. La partie du programme ici présentée ne correspond qu'aux 100 premières lignes :

```

!!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].xyz;
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;

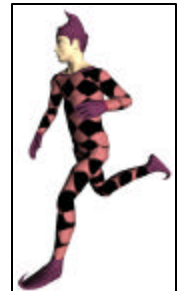
```

```
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;
```

## Annexe D. Exemple de code d'un vertex shader

### Exemple de programme : listes Cg

Un vertex shader simple, écrit en langage de programmation Cg, peut être écrit pour l'exemple de shader à quatre os abordé plus haut dans ce document. La liste est la suivante :



```
/******NVMH3*****/
Copyright NVIDIA Corporation 2002
DANS LES LIMITES DES TERMES PRÉVUS PAR LA LOI APPLICABLE, CE LOGICIEL EST
FOURNI *EN L'ÉTAT* ET NVIDIA AINSI QUE SES FOURNISSEURS DÉCLINENT TOUTE
GARANTIE, EXPRESSE OU TACITE, Y COMPRIS MAIS NON LIMITÉ À, TOUTE GARANTIE
IMPLICITE DE VALEUR MARCHANDE ET D'ADAPTABILITÉ À TOUT OBJECTIF PARTICULIER. EN
AUCUN CAS, NVIDIA OU SES FOURNISSEURS NE SERONT TENUS RESPONSABLES DE QUELQUE
DOMMAGE SPÉCIAL, FORTUIT OU INDIRECT QUE CE SOIT (Y COMPRIS, ET SANS RÉSERVE,
LES DOMMAGES CAUSÉS PAR LA PERTE DE PROFITS, L'INTERRUPTION DE L'ACTIVITÉ, LA
PERTE DES INFORMATIONS COMMERCIALES OU TOUTE AUTRE PERTE FINANCIÈRE) POUVANT
RÉSULTER DE L'UTILISATION DE OU DE L'INCAPACITÉ À UTILISER CE LOGICIEL, MÊME SI
NVIDIA A ÉTÉ PRÉVENUE DE LA POSSIBILITÉ DE TELS DOMMAGES.
*****/

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)
    {
        // transformez le décalage par la position de l'os i = 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));

        // déplacez la variable d'indice
        index = index.yzwx;
        weight = weight.yzwx;
    }

    normal = normalize(normal);

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

```

# Annexe E. Combinaison de textures multiples

## Échantillon de code

Les shaders peuvent traiter de nombreuses textures en un passage. Les listes pour l'exemple précédemment évoqué dans ce document sont les suivantes :



```
*****NVMH3****
Chemin d'accès : E:\nvidia\devrel\NVSDK\Common\media\programs
Fichier : cg_multipaint.cg

Copyright NVIDIA Corporation 2002
DANS LES LIMITES DES TERMES PRÉVUS PAR LA LOI APPLICABLE, CE LOGICIEL EST
FOURNI
*EN L'ÉTAT* ET NVIDIA AINSI QUE SES FOURNISSEURS DÉCLINENT TOUTE GARANTIE,
EXPRESSE OU TACITE, Y COMPRIS MAIS NON LIMITÉ A, TOUTE GARANTIE IMPLICITE DE
VALEUR MARCHANDE ET D'ADAPTABILITÉ À TOUT OBJECTIF PARTICULIER. EN AUCUN CAS,
NVIDIA OU SES FOURNISSEURS NE SERONT TENUS RESPONSABLES DE QUELQUE DOMMAGE
SPÉCIAL, FORTUIT OU INDIRECT QUE CE SOIT (Y COMPRIS, ET SANS RÉSERVE, LES
DOMMAGES CAUSÉS PAR LA PERTE DE PROFITS, L'INTERRUPTION DE L'ACTIVITÉ, LA PERTE
DES INFORMATIONS COMMERCIALES OU TOUTE AUTRE PERTE FINANCIÈRE) POUVANT RÉsulTER
DE L'UTILISATION DE OU DE L'INCAPACITÉ À UTILISER CE LOGICIEL, MÊME SI NVIDIA A
ÉTÉ PRÉVENUE DE LA POSSIBILITÉ DE TELS DOMMAGES.

Commentaires :
Manière simple de codifier plusieurs modèles de surface codés comme des
mappes.
Outre les valeurs "classiques" stockées comme des mappes (ex.: couleur et
intensité spéculaire), TOUTES les parties du BRDF sont stockées comme des
mappes ou des canaux de mappes à échelles de gris, auxquelles viennent parfois
s'ajouter des données supplémentaires pour définir une plage de valeurs
possible entre le "noir" et le "blanc" dans ce canal (afin que nous puissions
insérer le nombre maximal de valeurs dans la gamme de contrastes réelle d'un
mapi 8 bits, voire moins ! Les mappes du canal de contrôle sont
particulièrement adaptées à la palletisation)

*****/
```

```

// FRAGMENT DE PROGRAMME

// entrée -- structure identique à la sortie de "cg_multipaintVP.cg"
struct MultiPaintV2F {
    float4 HPosition      : POSITION;      // rognez la position de l'espace pour
le convertisseur d'ombrage. Non lisible dans le fragment de programme
    float4 TexCoords      : TEXCOORD0; // coordonnées ST de base
    float3 OPosition      : TEXCOORD1; // Emplacement des coordonnées de l'objet
    float3 Normal         : TEXCOORD2; // Normale de l'espace visuel
    float3 VPosition      : TEXCOORD3; // position du visualiseur dans les
coordonnées de l'objet
    float3 T              : TEXCOORD4; // tangente dans les coordonnées de l'objet
    float3 B              : TEXCOORD5; // binormale dans les coordonnées de l'objet
    float3 N              : TEXCOORD6; // normale dans les coordonnées de l'objet
    float4 LightVec0      : TEXCOORD7; // orientation de la lumière dans les
coordonnées de l'objet
    float4 Color0         : COLOR0; // Couleur provenant potentiellement des
sommets
};

struct PixelOut {
    float4 COL;
    float DEPR;
};

//
// fonctions //////////////////////////////////
//

//
// Une manière simple de visualiser des vecteurs à la surface, à des fins de
débogage.
// Pas utilisée habituellement dans ce programme
//
float4 vector_as_color(float4 theVector)
{
    float4 nv = 0.5f+(0.5f*theVector);
    return nv;
}
// version surchargée pour les vecteurs float3
float4 vector_as_color(float3 theVector)
{
    float4 nv = 0.5f+(0.5f*float4(theVector.x,theVector.y,theVector.z,0.0f));
    return nv;
}

////////////////////////////////////
// Fragment de programme réel ici //
////////////////////////////////////

// canaux dans notre mappe matérielle :
#define SPEC_STR x
#define METALNESS y
#define NORM_SPEC_EXPON z

// sous-champs dans "SpecData"
#define MINPOWER x
#define MAXPOWER y
#define MAXSPEC z

```

```

// sous-champs dans "ReflData"
#define FRESNEL_MIN x
#define FRESNEL_MAX y
#define FRESNEL_EXPON z
#define REFL_STRENGTH w

// sous-champs dans "BumpData"
#define BUMP_SCALE x

PixelOut main(
    MultiPaintV2F IN,
    uniform sampler2D ColorMap : texunit0, // couleur
    uniform sampler2D MaterialMap : texunit1, // codifie
    {specStrength,metalness,normalized_specExpon,0}
    uniform sampler2D NormalMap : texunit3, // normales de l'espace de la
    tangente
    uniform samplerCUBE EnvMap : texunit2, // environnement skybox
    uniform float4 SpecData, // composants : {minpower,
    maxPower,maxSpecStr,??}
    uniform float4 ReflData, // composants : {fresMin,
    fresMax,fresExpon,reflStrength}
    uniform float4 BumpData // composants : {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.LightVec0).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****
Chemin d'accès : NVSDK\Common\media\programs
Fichier : cg_multipaintVP.cg

Copyright NVIDIA Corporation 2002
DANS LES LIMITES DES TERMES PRÉVUS PAR LA LOI APPLICABLE, CE LOGICIEL EST
FOURNI *EN L'ÉTAT* ET NVIDIA AINSI QUE SES FOURNISSEURS DÉCLINENT TOUTE
GARANTIE, EXPRESSE OU TACITE, Y COMPRIS MAIS NON LIMITÉ À, TOUTE GARANTIE
IMPLICITE DE VALEUR MARCHANDE ET D'ADAPTABILITÉ À TOUT OBJECTIF PARTICULIER.
EN AUCUN CAS, NVIDIA OU SES FOURNISSEURS NE SERONT TENUS RESPONSABLES DE
QUELQUE DOMMAGE SPÉCIAL, FORTUIT OU INDIRECT QUE CE SOIT (Y COMPRIS, ET SANS
RÉSERVE, LES DOMMAGES CAUSÉS PAR LA PERTE DE PROFITS, L'INTERRUPTION DE
L'ACTIVITÉ, LA PERTE DES INFORMATIONS COMMERCIALES OU TOUTE AUTRE PERTE
FINANCIÈRE) POUVANT RÉSULTER DE L'UTILISATION DE OU DE L'INCAPACITÉ À UTILISER
CE LOGICIEL, MÊME SI NVIDIA A ÉTÉ PRÉVENUE DE LA POSSIBILITÉ DE TELS DOMMAGES.

Commentaires :
    Basé sur cg_fp30setup.cg

*****/

// définit les entrées du tampon du vertex
struct appin : application2vertex
{
    float4 Position      : POSITION;
    float4 UV            : TEXCOORD0;
    float4 Tangent       : BLENDWEIGHT;
    float4 Binormal      : DIFFUSE;
    float4 Normal        : NORMAL;
};


// sortie -- structure identique à l'entrée dans "cg_multipaint.cg"
struct MultiPaintV2F {
    float4 HPosition     : POSITION; // rognez la position de l'espace pour le
    convertisseur d'ombrage. Non lisible dans le fragment de programme
    float4 TexCoords     : TEXCOORD0; // coordonnées ST de base
    float3 OPosition     : TEXCOORD1; // Emplacement des coordonnées de l'objet
    float3 Normal        : TEXCOORD2; // Normale de l'espace visuel
    float3 VPosition     : TEXCOORD3; // position du visualiseur dans les
    coordonnées de l'objet
    float3 T             : TEXCOORD4; // tangente dans les coordonnées de
    l'objet
    float3 B             : TEXCOORD5; // binormale dans les coordonnées de
    l'objet
    float3 N             : TEXCOORD6; // normale dans les coordonnées de l'objet
    float4 LightVec0     : TEXCOORD7; // orientation de la lumière dans les
    coordonnées de l'objet
    float4 Color0       : COLOR0; // Couleur provenant potentiellement des
    sommets
};

```

```

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)    // dans les coordonnées EYE
{
    MultiPaintV2F OUT;
    OUT.HPosition = mul(ModelViewProj, IN.Position); // rognez l'espace pour
l'utilisation du convertisseur d'ombrage
    OUT.OPosition = IN.Position.xyz; // espace de l'objet-- passez
simplement dans
    OUT.Normal = normalize(mul(ModelViewIT, IN.Normal).xyz); // xf vers
l'espace de visualisation
    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 des
coordonnées Eye vers les coordonnées de l'objet
    OUT.LightVecO = mul(ModelViewI, LightVec); // Eye vers l'espace de
l'objet
    // OUT.Color0 = IN.Color;
    return OUT;
}

```



Les informations fournies sont réputées précises et fiables. Toutefois, NVIDIA Corporation décline toute responsabilité quant aux conséquences de l'utilisation qui pourrait en être faite ou de la contrefaçon de brevets ou autres droits de tierces parties pouvant résulter de leur utilisation. Aucune licence n'est octroyée implicitement ou de quelque autre manière sous quelque brevet ou droit de brevet de NVIDIA Corporation. Les caractéristiques techniques mentionnées dans ce document peuvent être modifiées sans préavis. Cette publication annule et remplace toute information diffusée antérieurement. Les produits de NVIDIA Corporation ne peuvent en aucun cas être utilisés en tant que composants critiques pour des systèmes de survie sans l'accord préalable écrit de NVIDIA Corporation.

**Marques commerciales**

NVIDIA et le logo NVIDIA sont des marques déposées et CineFX est une marque commerciale de NVIDIA Corporation.

Microsoft, DirectX, Windows et le logo de Windows sont des marques déposées de Microsoft Corporation. Les autres noms de sociétés ou produits peuvent être des marques commerciales des sociétés auxquelles ils sont associés.

**Copyright**

Copyright NVIDIA Corporation 2002



**NVIDIA.**

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