IMN638 – CHAPITRE 2 MODULE 2

RENDU PROGRAMMABLE

Automne 2013 – Université de Sherbrooke

Sommaire

- Notion de rendu programmable
- Étude des différents types de nuanceurs
- Étude du modèle de programmation
- Langages de programmation de nuanceurs
- Annexe : Exemple de nuanceur en GLSL

NOTION DE RENDU PROGRAMMABLE

Notion de rendu programmable

Historiquement, le matériel informatique pour le rendu d'images n'effectuait que l'étape de rastérisation du pipeline de rendu. L'étape de géométrie était alors effectuée par le processeur central lui-même.

Photo d'un processeur GeForce 256 de Nvidia (wikipedia.org)

La première carte vidéo à avoir supporté les transformations géométriques sur le processeur graphique lui-même fut la GeForce 256, introduite en 1999 par la compagnie Nvidia. C'est d'ailleurs à ce moment que le terme **processeur graphique** fut introduit, afin de différencier ce produit de ce qu'on appelait précédemment un **processeur vidéo**.

Notion de rendu programmable

Depuis le début des années 2000, les processeurs graphiques n'ont cessés de croître en fonctionnalité et flexibilité.

Les processeurs graphiques consistaient initialement en un pipeline graphique plus ou moins configurable à partir du processeur central de l'ordinateur. Ce pipeline graphique était alors appelé **pipeline fixe**, car il n'était pas possible de le modifier au-delà des options de configurations fixes qu'il proposait. Par exemple, il n'était pas possible d'intégrer du traitement logique dans le rendu.

De nos jours, le pipeline graphique a grandement évolué et est maintenant caractérisé de **pipeline programmable.** Le pipeline graphique programmable est en quelque sorte une toile vide sur laquelle le développeur en infographie peut programmer de toute pièce l'algorithme de rendu qu'il désire utiliser. La grande majorité du rendu est donc maintenant complètement laissée entre les mains du développeur plutôt qu'effectuée à travers l'électronique fixe du processeur graphique.

Notion de rendu programmable

Dans le module précédent, nous avons vu le pipeline de rendu général, s'appliquant à la très grande majorité des architectures graphiques. Les étapes fonctionnelles de ce dernier étaient données comme suit :

Étapes fonctionnelles d'un pipeline graphique standard.

Notion de rendu programmable

Pour un pipeline programmable, certaines étapes du pipelines sont combinées (ou ajoutées) et remplacées par un **programme (du code programmé par le développeur)**. Ce programme est appelé **nuanceur (shader)**, et possède des caractéristiques particulières en fonction de l'étape fonctionnelle qu'il remplace. Un type de nuanceur différent est utilisé pour chaque étape programmable du pipeline. Le code du programme en soit est écrit par le développeur, compilé par le pilote vidéo et/ou l'API graphique, puis exécuté sur le processeur graphique.

Pipeline standard comparé au pipeline programmable tel que nous l'étudierons dans le cadre du cours.

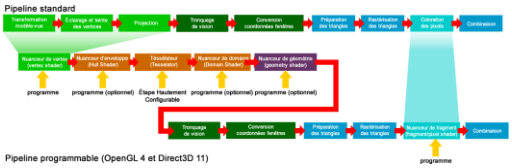
Dans le cadre du cours, nous étudierons en détail le nuanceur de vertex et le nuanceur de fragment, et aborderons aussi certains éléments du nuanceur de géométrie.

7

RENDU PROGRAMMABLE

Notion de rendu programmable

Avant de poursuivre, rappelons que nous concentrerons nos efforts sur les **nuanceurs de géométrie, de vertex et de fragment**. Ceci étant dit, plusieurs nouveaux types de nuanceurs ont été ajoutés depuis le début de l'année 2010 avec l'arrivée de Direct3D 11 et OpenGL 4. Bien qu'il ne soit pas encore majoritairement répandu, le pipeline programmable des cartes vidéos récentes (2e moitié de 2010 et plus récent) ressemble plutôt à ceci :



Pipeline programmable (OpenGL 4 et Direct3D 11)

8

RENDU PROGRAMMABLE

Notion de rendu programmable

Pour le cours, on dénote 3 types de nuanceurs différents. Dans l'ordre du pipeline :

- **Nuanceur de vertex** (modifie les propriétés d'un vertex)
- **Nuanceur de géométrie** (crée/supprime des primitives et modifie leurs propriétés)
- **Nuanceur de fragment** (modifie les propriétés d'un fragment)

De manière générale, **les sorties du nuanceur de vertex correspondent aux entrées du nuanceur de géométrie et les sorties du nuanceur de géométrie aux entrées du nuanceur de fragment**. Si aucun nuanceur de géométrie n'est utilisé, les sorties du nuanceurs de vertex deviennent les entrées du nuanceur de fragment.

Par exemple, si notre système possède un nuanceur de vertex et un nuanceur de fragment, et que le nuanceur de vertex retourne une normale, une position et une couleur, le nuanceur de fragment recevra automatiquement une normale, une position et une couleur.

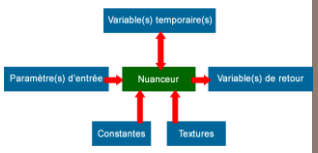
9

RENDU PROGRAMMABLE

Notion de rendu programmable

Un nuanceur, en sa qualité de programme servant au rendu, possède une série d'entrées et de sorties similaires aux entrées et sorties génériques des logiciels que nous programmons habituellement. Ces entrées/sorties possèdent différentes permissions de lecture/écritures :

- Entrées (lecture seule) :
  - Paramètres d'entrée
  - Variables constantes du programmes
  - Données de texture
- Sorties (écriture seule) :
  - Variables de retour
- Variables locales (lecture et écriture) :
  - Variables temporaires initialisées dans le nuanceur lui-même.



ÉTUDE DES DIFFÉRENTS TYPES DE NUANCEURS

Étude des différents types de nuanceurs

Tel que vu précédemment, il existe trois types de nuanceurs principaux que nous étudierons. Soit : le **nuanceur de vertex**, le **nuanceur de géométrie** et le **nuanceur de fragment**.

Lors du rendu, si le développeur choisi d'utiliser un rendu programmable (donc d'utiliser des nuanceurs plutôt que le pipeline fixe), il doit obligatoirement fournir au moins le nuanceur de vertex et le nuanceur de fragment.

Cette contrainte est justifiée par le fait qu'un nuanceur de vertex possède des variables d'entrées définies par le développeur au niveau de l'API graphique et des variables de retour définies par le développeur encore une fois, via le nuanceur. Ces dernières doivent donc être compatibles avec les entrées du nuanceur de fragment (on se souvient que ce sont deux programmes séparés). **Il n'est donc plus possible pour le processeur graphique de traiter sans nuanceur les entrées du nuanceur de vertex ou du nuanceur de fragment car ces dernières ne suivent plus les entrées/sorties du pipeline standard**.

Étude des différents types de nuanceurs

On se souvient que les nuanceurs de vertex et de fragment correspondent à des étapes obligatoires du pipeline graphique. Simplement ne pas implémenter un de ces nuanceurs n'est pas une option car leur non-implémentation impliquerait des étapes manquantes au niveau du pipeline graphique.

Pour terminer, notons que **l'utilisation d'un nuanceur de géométrie est optionnelle** au niveau du pipeline. En effet, le nuanceur de géométrie est une étape ajoutée au pipeline traditionnel, cette dernière n'est donc pas obligatoire pour que le rendu soit possible.

En pratique, l'utilisation d'un nuanceur de géométrie est plus rare et moins répandue que l'utilisation des nuanceurs de vertex et de fragment.

Nuanceur de vertex

Le nuanceur de vertex est la toute première étape d'un pipeline programmable et consiste en un programme qui remplace les étapes suivantes du pipeline standard :

- Transformation modèle-vue
- Éclairage et teinte des vertices
- Projection

Le nuanceur de vertex traite chaque vertex envoyé à la carte vidéo de façon indépendante. Autrement dit, il n'est pas possible d'accéder aux autres vertexes connectés à un vertex lorsqu'un nuanceur de vertex est exécuté.

En pratique, le nuanceur de vertex reçoit en entrée toutes les informations associées à un vertex en particulier. Ces informations sont transmises via la librairie graphique (OpenGL ou Direct3D). **Ces informations sont habituellement des informations géométriques, par exemple la position du vertex, son vecteur normal, sa couleur, ses coordonnées de texture, etc.**

Nuanceur de vertex

Ceci étant dit, puisque nous ne sommes plus dans un pipeline fixe, **n'importe quelle donnée personnalisée peut être envoyée au nuanceur de vertex**. Il est donc possible d'envoyer des données de vertex propre à notre application.

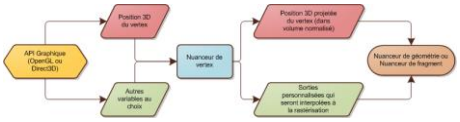
Par exemple, si notre application sert à modéliser un phénomène thermodynamique, on peut envoyer, en plus des informations standard de position et de normale, un nombre réel représentant la chaleur au vertex courant afin de colorer le vertex d'une façon particulière en fonction de la température.

Nuanceur de vertex

Le nuanceur de vertex possède des données d'entrée et de sortie obligatoires pour bien fonctionner avec le reste du pipeline non-programmable. Soit :

• **Entrée obligatoire :** Position 3D du vertex.
• **Sortie obligatoire :** Position 3D du vertex projetée du vertex (dans le volume de vision canonique)

Autrement dit, minimalement, une transformation modèle-vue et la projection doivent être effectuées au sein du nuanceur. Notons finalement que **le nuanceur de vertex peut aussi produire des variables de sorties personnalisées**. (Par exemple il pourrait retourner l'indice de chaleur de notre exemple précédent en sortie.)



16

RENDU PROGRAMMABLE

Nuanceur de vertex

Exemples d'utilisation du nuanceur de vertex :

• **Déformation de modèle 3D :** Déformer, plier, tordre un modèle 3D en déplaçant chaque vertex.

• **Placement de géométrie :** Générer un maillage en forme de grille et aller lire les données d'altitude de chaque point de la grille dans un texture à 1 canal de couleur où chaque texel de texture représente l'altitude en un point. On crée ainsi un relief 3D.

Exemples de nuanceur de vertex. **Gauche :** Nuanceur de vertex standard ne faisant que transformer les vertexes pour les emmener en espace projeté canonique (reproduit le pipeline fixe). **Milieu :** Déformation de cisaillement appliquée au sein du nuanceur de vertex en transformant chaque vertex du modèle. **Droite :** Application d'une fonction de bruit sur chaque vertex du maillage afin de le déformer. (RTR3,p.40)

17

RENDU PROGRAMMABLE

Nuanceur de géométrie

Le nuanceur de géométrie est un nuanceur relativement récent ajouté au pipeline graphique programmable. Tel qu'expliqué précédemment, il est complètement optionnel et n'est donc pas obligatoirement implémenté.

Le travail principal du nuanceur de géométrie est de modifier les primitives géométriques. Il peut modifier les attributs de ces primitives, ajouter ou enlever des attributs, créer de nouvelles primitives ou en enlever.

Un nuanceur de géométrie **reçoit toujours une seule primitive en entrée.** Le **nombre de primitive qu'il retourne en sortie peut être supérieur à un**. Le type de primitive reçu en entrée n'a pas non plus besoin de concorder avec le type de primitives émis en sortie. (Par exemple, un nuanceur de géométrie peut recevoir un point en entrée et retourner un ruban (strip) de 4 triangles en sortie.)

Comme pour le nuanceur de vertex, le nuanceur de géométrie doit produire des vertexes projetés dans un espace normalisé. **Dans cette optique, il est possible d'émettre des vertexes non-transformés du nuanceur de vertex et terminer leur projection dans le nuanceur de géométrie, ce, seulement s'il y a un nuanceur de géométrie implémenté.**

18

RENDU PROGRAMMABLE

Nuanceur de géométrie

En entrée, un nuanceur de géométrie peut recevoir trois types de primitives de base. Notons que la primitive d'entrée n'est pas obligée de correspondre à la primitive envoyée par l'API graphique. Par exemple, on peut envoyer des rubans de triangle avec un *glBegin(TRIANGLE_STRIP)* mais configurer le nuanceur de géométrie pour recevoir des points en entrées. Il recevra alors les points des triangles du ruban.

Les primitives d'entrées de base sont :

- Un point
- Un segment de poly-ligne
- Un triangle (3 vertexes connectés)

Le nuanceur de géométrie peut aussi recevoir des primitives spéciales, dites « primitives étendues » :

- Un segment de droite accompagné du points formant le segment précédent et du point formant le segment suivant.
- Un triangle et les trois vertexes formant les triangles adjacents.

Nuanceur de géométrie

Représentation des diverses entrées possibles d'un nuanceur de géométrie :

De gauche à droite : Point, segment de poly-ligne, triangle, segment de poly-ligne étendu et triangle étendu. (RTR3, p.41)

Nuanceur de géométrie

En sortie, le nuanceur de géométrie peut émettre trois types de primitives, soit :

- **Une liste de points** (point list)
- **Une poly-ligne** (line strip)
- **Un ruban de triangles** (triangle strip)

Notons que **le type d'entrée et de sortie du nuanceur est habituellement spécifié au niveau de l'API graphique ou au niveau du nuanceur lui-même**.

Notons aussi que l'étape du **nuanceur de géométrie**, bien qu'exécutée sur des multiples instances parallèles, **doit absolument produire ses sorties dans le même ordre qu'elle les reçoit pour chaque primitive fournies par l'API graphique** (une primitive = ce qui est envoyé entre un *glBegin* et un *glEnd*). Si un groupe d'instances de nuanceur A reçoit ses primitives avant un groupe d'instances B, le groupe A doit absolument avoir terminé son exécution avant que B puisse retourner ses résultats, ce, même si B était beaucoup plus rapide que A. Il faut donc s'assurer qu'un groupe d'exécution d'un nuanceur de géométrie ne devienne pas un goulot d'étranglement pour le reste du système.

Nuanceur de géométrie

Rappelons que le nuanceur de géométrie **peut émettre en sortie un nombre supérieur de primitives et de points qu'il n'en a reçu en entrée**. Il est donc possible, à partir d'un seul point par exemple, d'émettre un cube composé de plusieurs triangles en sortie.

Cette propriété d'une étape du pipeline à augmenter la quantité de géométrie du rendu est appelé la **capacité d'expansion. Le nuanceur de géométrie a une capacité d'expansion relativement réduite de l'ordre de 1 pour 10**. Autrement dit, pour un vertex en entrée, le nuanceur peut générer jusqu'à 10 points en sortie sans affecter outre mesure les performances. C'est donc dire qu'il n'est pas possible de générer des modèles trop complexes à partir du **nuanceur de géométrie, qui n'est pas conçu pour l'expansion de la géométrie mais plutôt pour la manipulation de primitive**.

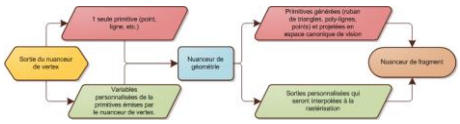
Une étape conçue pour l'expansion géométrique est **l'étape de téssélation ajoutée récemment, permettant des niveaux d'expansion excédant le 1 pour 1000**.

22

RENDU PROGRAMMABLE

Nuanceur de géométrie

Nous pouvons résumer les entrées/sorties du nuanceur de géométrie, soit :



En résumé, le nuanceur de géométrie permet de :

- **Modifier les informations de vertex** (enlever, modifier, ajouter des paramètres personnalisés.)
- **Produire de la nouvelle géométrie** (d'un type potentiellement différent du type d'entrée.)
- **Supprimer de la géométrie** (en produisant moins d'information que la quantité reçu en entrée.)

23

RENDU PROGRAMMABLE

Nuanceur de géométrie

Pour terminer, notons que le nuanceur de géométrie peut virtuellement faire exactement les mêmes opérations de vertex que le nuanceur de vertex. Ceci étant dit, **la contrainte d'ordre du nuanceur de géométrie crée un certain couplage dans l'exécution des instances du nuanceur, il est donc important de garder cette étape aussi courte que possible pour éviter les goulots d'étranglements**. Dans cette optique, la majeure partie du travaille effectué sur les vertexes devraient idéalement être effectué dans le nuanceur de vertex malgré le fait que le nuanceur de géométrie possède les structures pour effectuer ce dit travail.

Il est donc important de voir le nuanceur de géométrie comme une étape permettant d'altérer la structure géométrique plutôt que ses propriétés.

24

RENDU PROGRAMMABLE

Nuanceur de géométrie

Quelques exemples de nuanceurs de géométrie :

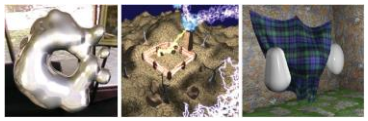
• À partir de l'API graphique, passer 1 seul point 3D et générer une particule carrée affichée en « billboard » composée de 2 triangles à partir de ce point. (Réduction d'environ 75% de la bande passante entre l'ordinateur et la carte graphique, on envoie 1 vertex plutôt que 4.)

•À partir d'un segment de ligne droite et de ses 2 voisins, générer une poly-ligne de plusieurs sous-segments afin de transformer le segment de droite en une courbe de bézier ou un spline, les voisins étant utilisés comme points de contrôle.

•Diminuer dynamiquement le niveau de détail d'une géométrie selon la distance de la caméra à cette dernière pour éviter de rendre inutilement des détails qui ne sont plus visibles.

Nuanceur de géométrie

Exemples de nuanceurs de géométrie (suite) :



Gauche : Génération d'un modèle 3D de surface implicite. **Milieu :** Les éclairs sont générés en subdivisant des segments de droite et en changeant leur angle à l'aide d'un bruit fractal. **Droite :** Calcul d'un réseau de masse-ressort simulant un tissus à l'aide d'un nuanceur de géométrie et des informations d'adjacence des primitives (RTR3, p.42)

Nuanceur de fragment

Le **nuanceur de fragment correspond à l'étape de coloration des pixels** du pipeline standard. Son travail principal est d'**utiliser les informations qui lui sont transmises en entrées par le nuanceur de vertex et/ou le nuanceur de géométrie, et de générer la couleur et la profondeur finale du fragment**.

Notons que le test pochoir est souvent effectué avant le nuanceur de fragment. Le résultat du test pochoir n'est cependant pas accessible par le nuanceur de fragment et la valeur au pochoir ne peut être modifiée par celui-ci.

Comme pour les autres nuanceurs, le nuanceur de fragment ne travaille que sur un seul élément à la fois, un fragment, et n'a pas accès à l'information des autres fragments. Plusieurs instances de nuanceurs de fragment s'exécutent d'ailleurs simultanément lors du rendu.

Nuanceur de fragment

Un fragment est le résultat de la rastérisation entre les vertexes des triangles: **la valeur d'un fragment en particulier n'est pas déterminée par un seul vertex**. Pour déterminer cette valeur, le processeur graphique doit interpoler les valeurs en sortie du nuanceur de vertex (ou de géométrie) et fournir la valeur interpolée en entrée au nuanceur de fragment. **Le nuanceur de fragment reçoit donc en entrées toutes les valeurs interpolées des sorties du nuanceur de géométrie ou de vertex, comprenant la position en espace image**.

Nous verrons aussi plus tard qu'**il est possible d'envoyer des valeurs dont la valeur demeure uniforme sur la totalité des fragments d'une primitive**. Ces valeurs ne seront pas interpolées entre les fragments puisqu'elles sont les mêmes pour tous les vertexes d'une même primitive. (Par exemple, un vecteur 3D spécifiant la position d'une lumière pour l'éclairage demeurera constante tout au long du rendu de l'image.)

Nuanceur de fragment

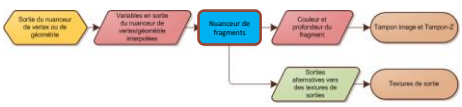
Le nuanceur de fragment doit au moins émettre la profondeur et la couleur finale du fragment. Notons que la profondeur peut rester inchangée. Dans un tel cas, elle n'a pas besoin d'être explicitement retournée du nuanceur, ce qui permet certains optimisations au niveau du rendu (notamment de procéder à un early-z test).

Le nuanceur de fragment peut émettre ses valeurs dans plusieurs tampons (buffers) de mémoire. Par défaut, les tampons en sortie sont le tampon de profondeur et le tampon de couleur de l'image. **Il est cependant possible d'ajouter des tampons pour émettre d'autres valeurs utiles** dans des « textures de sortie» séparées qui ne seront pas rendues directement à l'écran.

Par exemple, on pourrait vouloir émettre une image correspondant à l'image rendue dans le tampon de couleur, et une image colorée selon la direction des vecteurs normaux à chaque fragment dans un second tampon, qui serait sauvegardé sur le disque dur en bitmap et visionné pour le débogage.

Nuanceur de fragment

Schématiquement, le nuanceur de fragment se décrit ainsi : (remarquez spécialement les sorties possibles)



Nuanceur de fragment

Exemples d'utilisation du nuanceur de fragment :

• Simuler des effets optiques comme de la réflexion ou de la réfraction sur une surface.

• Générer des textures procédurales pixel par pixel sur une surface.

• Générer des effets d'illumination complexes comme de l'embossement afin de donner du relief supplémentaire à une surface.

Gauche : Texture procédurale de bois générée à l'aide de bruit perlin. **Milieu :** Effet de réflexion réalisé dans le nuanceur de fragment à l'aide d'une carte d'environnement cubique. **Droite :** Effet d'embossement (bump mapping) réalisé à l'aide de multiples textures. L'éclairage est calculé pixel par pixel au sein du nuanceur de fragment et tient compte du relief donné par une texture de relief, et de la couleur donnée par une texture de couleur.

MODÈLE DE PROGRAMMATION

Modèle de programmation

Un modèle de programmation est une façon de définir la programmation propre à un type de système en particulier en faisant abstraction des langages de programmation compatibles avec celui-ci.

Les nuanceurs étant des programmes, ils sont tôt ou tard convertis en code machine et exécutés sur un système ayant un fonctionnement propre, en l'occurrence, un processeur graphique.

Dans notre cas, le processeur graphique a comme objectif la réalisation de rendus graphiques. Le système est donc conçu expressément pour le rendu. Il possède donc certaines particularités qui lui sont propres et favorisent les performances pour l'accomplissement de tâches reliées au calcul géométrique. Ces particularités définissent le modèle de programmation propre aux nuanceurs qui sera étudié dans cette section du chapitre.

Modèle de programmation

Il existe une multitude de processeurs graphiques et d'architectures de processeurs graphiques différents. Il est donc légitime de se demander comment la **programmation de nuanceurs** peut être **définie** par **un seul modèle de programmation**, **malgré la très grande diversité de processeurs disponibles**.

Pour éviter une disparité dans les différentes façons de programmer un nuanceur, les différents acteurs du monde de l'infographie en temps réel (constructeurs de processeurs, développeurs d'API graphiques, chercheurs, etc.) ont mis en place un standard évoluant sous le nom de *Shader Model.*

Le shader model défini les capacités requises d'une carte vidéo par rapport à la programmation des nuanceurs. Ces capacités définissent les types d'entrées/sortie que doivent supporter les processeurs graphiques, le comportement général de ces derniers en regard d'un pipeline standardisé, la quantité de mémoire locale que doit posséder une unité de calcul, etc. Le modèle défini aussi une série d'instructions d'assembleur que doivent supporter les processeurs graphiques. **Ceci garanti un modèle de programmation homogène à travers les différents types de processeurs graphiques.**

Modèle de programmation

Avant de se pencher sur le modèle de programmation des nuanceurs, notons qu'il existe différentes versions des *Shader Models. Voici un tableau résumant certaines des contraintes des différents shader models :*

	SM 2.0/2.X	SM 3.0	SM 4.0
Introduced	DX 9.0, 2002	DX 9.0c, 2004	DX 10, 2007
VS Instruction Slots	256	$\geq 512^a$	4096
VS Max. Steps Executed	65536	65536	∞
PS Instruction Slots	$\geq 96^b$	$\geq 512^a$	$\geq 65536^a$
PS Max. Steps Executed	$\geq 96^b$	65536	∞
Temp. Registers	$\geq 12^a$	32	4096
VS Constant Registers	$\geq 256^a$	$\geq 256^a$	14×4096^c
PS Constant Registers	32	224	14×4096^c
Flow Control, Predication	Optional[d]	Yes	Yes
VS Textures	None	4^e	128×512^f
PS Textures	16	16	128×512^f
Integer Support	No	No	Yes
VS Input Registers	16	16	16
Interpolator Registers	8^g	10	$16/32^h$
PS Output Registers	4	4	8

Comparaison des différentes capacités des *Shader Models* entre la version 2.0 et la version 4.0 (RTR3, p.38, 2008)

Variables

Le modèle de programmation des nuanceurs supporte certaines variables de bases. Plusieurs sont similaire à celles qu'on retrouve en C mais d'autres types sont particuliers aux nuanceurs. Tout nuanceur supporte nativement les types de base suivants :

Types de variable :

- **Scalaire :** entiers, nombre à virgule flottante, booléens
- **Vecteurs :** vecteur de 2 à 4 composantes (entier, float, booléen)
- **Matrices :** matrices 2x2, 3x3 ou 4x4 de nombres en virgules flottantes
- **Samplers :** permet de lire une texture 1D, 2D, 3D ou Cubique, est associé à une texture en particulier au niveau de l'API graphique.

Notons que les samplers sont un type opaque. Nous observerons plus tard comment utiliser un sampler. Remarquons aussi l'absence de types tel que les caractères ou chaines de caractères, qui ne sont pas utiles dans un contexte de rendu 3D.

Variables

Qualificatifs des variables :

Selon le cas, il est possible d'utiliser des qualificatifs de variables, ce qui leur donne un comportement et un type d'accès différent au sein du pipeline programmable.

Les qualificatifs sont :

• **Attribute :** Variable très dynamique pouvant changer jusqu'à 1 fois par vertex. Passée par l'application vers le nuanceur de vertex uniquement. La position du vertex, sa normale, sa couleur, etc. seraient des exemples d'attributs.

• **Uniform :** Variable peu dynamique pouvant changer jusqu'à une fois par primitive. (glBegin/glEnd). Passée par l'application vers le nuanceur de vertex, de géométrie ou de fragment directement. Cette variable n'est pas interpolée car sa valeur est la même partout sur une primitive, et donc sur les triangles qui la composent. La position d'une lumière ou un sampler d'une texture à appliquer sur un maillage complet seraient des exemples de variable uniforme.

Variables

Qualificatifs des variables (suite) :

• **Varying :** Permet de faire passer une variable d'un nuanceur vers un autre. Par exemple, permet de faire passer une variable du nuanceur de vertex vers le nuanceur de fragment (si aucun nuanceur de géométrie n'est défini). Les variables de ce type sont interpolées entre les vertexes pour être envoyées au nuanceur de fragment. (Passée du nuanceur de vertex ou de géométrie vers le nuanceur de fragment.) Ces variables sont définies et déclarées au sein du nuanceur de géométrie ou de vertex et constituent leurs variables de sorties. Les variables *varying* sont aussi les variables d'entrées du nuanceur de fragment.

• **Const :** Variables non-inscriptibles résolues à la compilation. Déclarées et définies dans n'importe quel type de nuanceur. (Comportement identique aux variables constantes en C/C++.)

Notons qu'**une variable de type sampler est toujours uniforme** puisqu'il n'est pas possible de changer de texture entre deux vertexes au sein d'une même primitive dans un processeur graphique.

Contrôle d'exécution

Les nuanceurs, à partir du shader model 2.0, supportent le contrôle du flot d'exécution. Autrement dit, il est possible de mettre des structures de contrôle logiques (if, else, for,while, etc.) au sein d'un nuanceur.

On note aussi une structure de contrôle logique particulière, le **discard**, qui peut être appelé à n'importe quel moment dans un nuanceur de fragment. Cette structure de contrôle arrête automatiquement le nuanceur de fragment et le fragment ne sera pas écrit dans le tampon image.

Notons aussi que les structures de contrôle de type **goto** ou **switch** n'existent pas dans la programmation de nuanceurs.

De manière générale, il est recommandé d'éviter d'ajouter des structures de contrôle inutilement. En effet, **un nuanceur ne contenant aucune structure de contrôle peut être grandement optimisé par le compilateur, conférant au nuanceur un gain de performance très important**.

Opérations mathématiques et fonctions standards

Les nuanceurs supportent habituellement une série de fonctions mathématiques standards et des fonctions génériques standards. Ces fonctions sont extrêmement performantes au sein d'un nuanceur et dépassent en fonctionnalité et en performance les fonctions mathématiques habituellement intégrées à un processeur de calcul général.

Des fonctions mathématiques standards simple sont bien sûr présentes (sin, cos, ceil, floor, sqrt, etc.). On y ajoute aussi des fonctions mathématiques dédiées aux opérations de vecteur (dotproduct, crossproduct, multiplication de matrice, etc.) Et certaines opérations plus spécifiques (smoothstep, step, inversesqrt, etc.)

Toutes ces opérations mathématiques sont directement supportées par le processeur graphique. Autrement dit, l'appel de ces fonctions nécessite souvent seulement 3 ou 4 cycles de calcul alors qu'elle en nécessiteraient parfois plusieurs centaines sur un processeur normal

Opérations SIMD

Les opérations effectuées par le processeur graphique sur un vecteur ou une matrice au sein d'un nuanceur sont habituellement des opérations SIMD (single instruction, multiple data). Autrement dit, les opérations sont effectuées en parallèles sur toutes les composantes du vecteur ou de la matrice au même moment plutôt qu'en séquence. Une addition entre deux vecteurs à 4 composantes verra donc ses 4 additions effectuées au même moment plutôt qu'à tour de rôle, ce qui rend le calcul 4 fois plus rapide. Ce processus est appelé la vectorisation d'un calcul.

De même manières, plusieurs instances de nuanceur qui démarrent en même temps sont habituellement groupées et s'exécutent en parallèle de manière à vectoriser les calculs entre les instances du nuanceur.

Il est donc préférable d'effectuer des opérations sur les vecteurs plutôt que de séparer les vecteurs en composantes puis d'effectuer les calculs indépendamment sur chacune d'elles.

Lecture dans une texture

La lecture dans une texture s'effectue à l'aide d'une variable de type **sampler.** Il existe différents types de samplers, correspondant à différents types de texture. Par exemple, un Sampler2D pourra lire une texture à 2 dimensions.

Dans une instance d'exécution d'un nuanceur, un sampler n'est associé qu'à une seule texture. La texture associée à un certain sampler est définie au niveau de l'API. Bien entendu, il est possible d'avoir plusieurs samplers différents qui sont chacun associés à une texture différentes si plusieurs textures sont nécessaires dans l'exécution d'un nuanceur.

Lorsqu'on souhaite aller lire la texture, on utilise habituellement un appel de fonction prenant le sampler et la coordonnée de texture à lire en paramètre. La coordonnée est un vecteur dont les composantes sont normalisées entre 0 et 1. Le nombre de composantes dépend du nombre de dimensions de la texture.

Lecture dans une texture

Exemple de lecture de texture dans un nuanceur :

Sous GLSL (langage de nuanceur d'OpenGL), les différentes étapes de lecture dans une texture 2D s'effectuent comme suit :

Lors de la programmation du nuanceur, on crée notre variable « sampler » dans le code. (On se souvient, une variable de type sampler doit toujours être uniforme)

```
uniform sampler2D gazon;
```

On peut ensuite lire dans la texture à la coordonnée voulue avec la fonction « texture » :

```
vec4 couleurTexture = texture(gazon,coord);
```


Passage de variable entre l'API et le nuanceur

Les variables envoyées au nuanceur proviennent initialement de l'application. Le nuanceur reçoit en paramètre des données de géométrie qui sont définies via les différentes données chargées en mémoire vidéo ou les appels de fonction de l'API graphique.

Les variables définissant un vertex sont des attributs (attributes) et sont envoyées via les fonctions de l'API graphique ou chargées en mémoires.

Les variables définissants des propriétés du rendu, qui sont propres à un groupe de géométrie (ou toute la scène) sont quant à elles envoyées via l'API graphique sous la forme de variables uniformes (uniform).

Chaque API possède une méthode différente pour transmettre les valeurs des variables de l'application vers le nuanceur. La transmission demande habituellement de spécifier le nom de la variable dans le nuanceur pour laquelle on souhaite envoyer une valeur dans le nuanceur, puis de spécifier la valeur à envoyer vers celui-ci.

LANGAGES DE PROGRAMMATION DE NUANCEUR

Langages de programmation de nuanceur

Un nuanceur, lors de son exécution sur le processeur graphique, est un programme composé de code machine, au même titre que tout autre programme compilé nativement (comme du C ou du C++). Pour chaque série de processeur graphique, la famille d'instruction assembleur supportée par le processeur est différente. Programmer un nuanceur directement en assembleur est donc possible mais plutôt difficile car il faudrait prévoir autant de versions d'un même nuanceur qu'il existe de familles d'instructions différentes.

Pour faciliter la tâche du développeur, les API graphiques importants définissent un langage dit **de haut niveau** permettant de programmer des nuanceurs dans un langage plus générique (souvent similaire à du C/C++ en syntaxe), sans avoir à se soucier de la conversion en code machine.

Langages de programmation de nuanceur

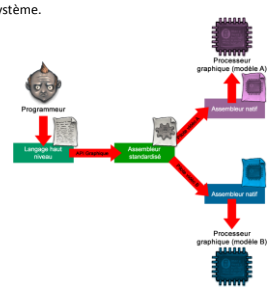
Lorsqu'un nuanceur est écrit dans un langage haut niveau, il doit être compilé et une édition des liens (linking) doit être effectuée par l'API graphique, sur le nuanceur, de manière à ce qu'il puisse être compris par le processeur graphique.

Encore là, une compilation directe de l'API vers le processeur graphique demanderait aux programmeurs des API graphiques de pouvoir créer des compilateurs fonctionnant avec tous les différents modèles de processeur graphiques, ce qui seraient bien sûr trop exigeant.

La tâche de produire un assembleur compréhensible par toutes les cartes graphiques est donc partiellement divisée entre les développeurs d'API graphiques et les constructeurs de carte graphique.

Langages de programmation de nuanceur

Globalement, les API graphiques se chargent de compiler leur code haut-niveau en un code assembleur standardisé à un Shader Model mais indépendant du processeur graphique. Le pilote vidéo prend ensuite le code assembleur standardisé et le traduit en un code assembleur dédié au modèle de processeur graphique du système.



Lors de la compilation, le code est optimisée une première fois par le compilateur de l'API graphique, puis une seconde fois par le traducteur d'assembleur du pilote vidéo qui peut tirer profit des petites optimisations du modèle de processeur graphique qui fera le rendu.

Langages de programmation de nuanceur

Les langages de programmation haut niveau les plus populaires pour les nuanceurs se ressemblent tous d'une certaine façon, puisqu'ils doivent suivre le modèle de programmation propre aux nuanceurs. Les langages sont appellés « C-Like » car leur syntaxe ressemble énormément à la syntaxe du C.

Parmi les langages haut niveau de programmation de nuanceurs, on note :

• **GLSL :** Langage haut niveau de nuanceur OpenGL. Il est extrêmement générique et offre une syntaxe très rapproche à celle du C. Son avantage principal est qu'il est non propriétaire et multiplateforme. Notons aussi que dans le cas de GLSL, la compilation vers l'assembleur standardisé est aussi effectuée par le pilote vidéo. Cette responsabilité incombe donc au fabricant du processeur graphique. (La nature ouverte et non propriétaire d'OpenGL justifie cette méthode de fonctionnement.)

• **HLSL :** Langage haut niveau de nuanceur pour Direct3D (Microsoft). Le langage est un « C-Like » mais possède néanmoins plus de particularités syntaxiques que GLSL. Il n'est pas multiplateforme et demeure propriétaire. Il fonctionne uniquement sur les plateformes Microsoft. (Ce qui peut devenir un avantage pour le développement de jeu vidéo sur PC/Xbox 360.)

Langages de programmations de nuanceurs

Langages de programmation haut niveau (suite) :

• **Cg :** HLSL a été développé par Microsoft en tandem avec Nvidia. Cg est la version multiplateforme de HLSL. Sa syntaxe est donc pratiquement identique.

De manière générale, les langages **Cg** et **HLSL** sont considérés comme plus rapprochés des concepts de shader model. L'abstraction entre le langage et le modèle de programmation des nuanceurs est donc moins poussée que pour **GLSL.**

Ceci confère un certain avantage au développeur qui possède un peu plus de contrôle sur le nuanceur au niveau du code, mais rend le code plus difficilement adaptable lors de la mise à jour vers un nouveau shader model ou advenant un changement important dans le modèle de programmation des nuanceurs.

ANNEXE : EXEMPLE DE NUANCEUR COMPLET

Annexe : Introduction

Dans le cadre de cette annexe, nous étudierons toutes les étapes de programmation, de construction, de configuration et d'utilisation d'un nuanceur simple. Le nuanceur sera écrit en GLSL (OpenGL) et utilisera un nuanceur de vertex en tandem avec un nuanceur de fragment.

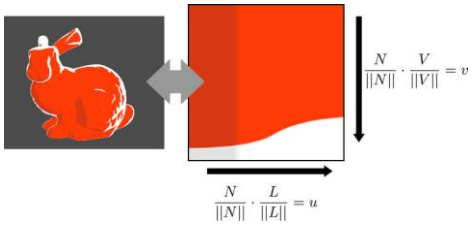
Annexe : Présentation du nuanceur

Pour cet exemple, **le nuanceur démontré sera un « Barla Shader »** (ou « X-Toon shader »). Globalement, le nuanceur nécessite diverses étapes :

1. (Dans le nuanceur de vertex.) Effectuer un produit scalaire entre le vecteur normal normalisé et le vecteur de lumière incidente normalisé, ce qui donne le terme « u » situé entre 0 et 1.
2. (Dans le nuanceur de vertex.) Effectuer un produit scalaire entre le négatif du vecteur de vision normalisé et le vecteur normal normalisé, ce qui donne un terme « v » situé entre 0 et 1.
3. (Dans le nuanceur de fragment) Prendre la valeur « u » et la valeur « v », effectuer une lecture à la coordonnée (u,v) d'une texture de référence et appliquer la couleur obtenue de la texture au fragment.

Annexe : Présentation du nuanceur

Schématiquement, on obtient :



Barla Shadin. Où N est le vecteur normal à un point, V le vecteur de vision négatif et L le vecteur entre la coordonnée courante et la source d'éclairage.

(Notons que la texture de référence doit être créée par un artiste dans une application de dessin quelconque. Différentes textures permettent d'obtenir différents effets.)

Annexe : Code source du nuanceur de vertex

Le code source du nuanceur de vertex, en GLSL est :

```
//Barla vertex shader (computes the NdotL and NdotV dot products)

varying float NdotL;
varying float NdotV;

void main(void)
{
          vec3 ecPosition     = vec3(gl_ModelViewMatrix * gl_Vertex);
          vec3 tnorm          = normalize(gl_NormalMatrix * gl_Normal);
          vec3 lightVec       = normalize(vec3(gl_LightSource[0].position) -
ecPosition);
          vec3 viewVec        = normalize(-ecPosition);

          NdotL = max(dot(lightVec,tnorm),0.0);
          NdotV = max(dot(viewVec,tnorm),0.0);

          gl_Position = ftransform();
}
```


Annexe : Code source du nuanceur de vertex

Variables standards :

`gl_ModelViewMatrix`: Variable par défaut de GLSL correspondant à la matrice modèle-vue courante d'OpenGL.

`gl_Vertex`: Variable par défaut de GLSL correspondant à la position courante du vertex traité dans le nuanceur.

`gl_LightSource`: Vecteur contenant des structures « lumière ». (La structure donne les informations sur la lumière comme la position, la couleur, l'angle d'ouverture, etc. Ces informations sont les informations configurées dans OpenGL.)

`gl_Position`: Valeur de retour du nuanceur de vertex, correspond à la position du vertex une fois transformé et projeté. (On se souvient que le nuanceur de vertex doit au moins retourner cette valeur.)

Annexe : Code source du nuanceur de vertex

`gl_NormalMatrix` et **`gl_Normal`** : Respectivement la matrice de transformation modèle → vue de la normale. Et la normale elle-même. (Variables par défauts).

Fonctions et variables en sortie :

Remarquons aussi l'utilisation de fonctions standards de GLSL, par exemple `max`, `dot` ou `normalize`.

Notons aussi la fonction `ftransform()`, fonction standard de GLSL qui retourne le vertex transformé dans l'espace normalisé.

Finalement, on remarque 2 variables de qualificatif « varying » (`NdotL` et `NdotV`). Ces variables sont envoyées à notre nuanceur de fragment qui recevra les valeurs interpolées de ces variables. Elles sont donc des sorties de notre nuanceur de vertex et, au même titre que la variable `gl_Position`, seront interpolées pour devenir les entrées de notre nuanceur de fragment.

Annexe : Code source du nuanceur de fragment

Le nuanceur de fragment est quant à lui très simple :

```
//Barla fragment shader (performs the texture lookup)

uniform sampler2D LookupSampler;

varying float NdotL;
varying float NdotV;

void main(void)
{
        gl_FragColor = texture(LookupSampler,vec2(NdotL,1.0-pow(NdotV,2.0)));
}
```


Annexe : Code source du nuanceur de fragment

Variables standards :

La seule variable particulière utilisée ici est la variable `gl_FragColor`. Cette dernière sert à indiquer la couleur finale du fragment. On se souvient qu'un nuanceur de fragment doit minimalement retourner cette valeur pour être valide. Par défaut, GLSL fixe la valeur à noir.

Variable provenant de l'application :

Dans le nuanceur, on remarque que la variable `LookupSampler`, une variable de type `Sampler2D` permettant d'aller lire dans une texture à deux dimensions. Cette variable est définie au niveau de l'application et est par la suite utilisée pour aller lire notre valeur de couleur dans la texture de référence.

Nous verrons plus tard comment associer une texture à un sampler au niveau d'OpenGL. Pour l'instant, notons l'utilisation de la fonction `texture` qui permet d'aller lire dans une texture.

Annexe : Compilation et édition des liens du nuanceur

Puisque les nuanceurs sont des programmes, ils doivent être compilés afin d'être convertis en code machine. Comme pour un programme habituel, une étape d'édition des liens doit aussi être effectuée. Ce n'est que lorsque la compilation et l'édition des liens du nuanceur est complétée qu'un nuanceur peut être utilisé.

Pour compiler et éditer les liens d'un nuanceur sous OpenGL, **on effectue les étapes suivantes dans l'application, en c++** :

1. Stocker le code source du nuanceur dans une chaîne de caractères. (Le code source peut provenir d'un fichier qui est lu et placé dans une chaîne de caractères, ou provenir d'une chaîne de caractères directement générée dans le code.)
2. Créer un objet « shader » (un objet par nuanceur, donc 2 objets shaders pour un nuanceur de vertex et un nuanceur de fragment)
3. Associer le code source à chaque objet shader.
4. Compiler chaque objet shader.
5. Créer un objet « program » et lui assigner les nuanceurs.
6. Procéder à l'édition des liens.

Annexe : Compilation et édition des liens du nuanceur

Le code C++ pour compiler et éditer les liens d'un nuanceur, via OpenGL :

```
//On crée des objets "shader" pour notre nuanceur de vertex et de fragment
unsigned int VxShaderHandle = glCreateShader(GL_VERTEX_SHADER);
unsigned int PxShaderHandle = glCreateShader(GL_FRAGMENT_SHADER);

//On associe le code source à nos objets "shader". (Le code source est un
// pointeur const de char. (const char*))
glShaderSource(VxShaderHandle,1,&VxShaderSource,NULL);
glShaderSource(PxShaderHandle,1,&PxShaderSource,NULL);

//On compile le nuanceur de vertex
glCompileShader(VxShaderHandle);
glGetShaderiv(VxShaderHandle,GL_COMPILE_STATUS,&CompilationStatus);
assert(CompilationStatus == GL_TRUE);

//On compile le nuanceur de fragment
glCompileShader(PxShaderHandle);
glGetShaderiv(PxShaderHandle,GL_COMPILE_STATUS,&CompilationStatus);
assert(CompilationStatus == GL_TRUE);

//On crée le programme de nuanceur et on lui associe les nuanceurs
unsigned int ShaderProgramHandle = glCreateProgram();
glAttachShader(ShaderProgramHandle,VxShaderHandle);
glAttachShader(ShaderProgramHandle,PxShaderHandle);

//On procède à l'édition des liens du programme de nuanceurs
glLinkProgram(ShaderProgramHandle);
glGetProgramiv(ShaderProgramHandle,GL_LINK_STATUS,&CompilationStatus);
assert(CompilationStatus == GL_TRUE);
```


Annexe: Activation du nuanceur

Notons que la compilation/édition des liens est normalement effectuée une fois au début de l'exécution du logiciel. Il n'est pas nécessaire de recompiler le nuanceur à chaque image, une fois compilé, le nuanceur est simplement activé lorsque nécessaire.

Pour indiquer qu'une certaine géométrie doit être rendue avec un nuanceur particulier, il faut simplement activer le dit nuanceur avant d'envoyer la géométrie au processeur graphique. Sous OpenGL, on utilise la fonction « `glUseProgram` ». Une fois la nuanceur activé, toute la géométrie envoyée pour le rendu utilisera ce nuanceur en particulier.

Dans notre exemple, on activerait le nuanceur en écrivant :

```
glUseProgram(ShaderProgramHandle);
```

Pour désactiver les nuanceurs et revenir fixe, on écrit :

```
glUseProgram(NULL);
```


Annexe : Passage de variable à partir d'OpenGL

Sous OpenGL, le passage d'une variable uniforme au nuanceur s'effectue en 2 étapes :

1. On demande à l'API de trouver la position mémoire de la variable dans le nuanceur, ce qui nous retourne un identifiant unique identifiant la variable.
2. On utilise l'identifiant unique pour configurer la valeur.

Par exemple, si on veut envoyer un nombre réel de valeur « 8.25 » à la variable uniforme `MaVariable` se trouvant dans un nuanceur, on ferait :

```
GLint TmpVariableLocation = glGetUniformLocation(ShaderProgramHandle,"MaVariable");
glUniform1f(TmpVariableLocation,8.25f);
```

Dans le cas d'un attribut de vertex, on procède similairement, mais le nom des fonctions change :

```
Glint TmpAttributeLocation = glGetAttribLocation(ShaderProgramHandle,"MonAttribut");
glVertexAttrib1f(TmpAttributeLocation,8.25f);
```

Notons que la position d'une variable du nuanceur ne change pas après l'édition des liens, il n'est donc pas nécessaire d'aller la rechercher à chaque fois qu'on désire l'utiliser.

Annexe : Passage de variable à partir d'OpenGL

Si nous revenons à notre exemple, nous avons une variable uniforme `LookupSampler` de type Sampler2D. Sous OpenGL, nous devons premièrement charger une texture dans un emplacement de texture, puis ensuite envoyer à notre nuanceur l'identifiant de l'emplacement de texture afin que notre Sampler2D soit associé au bon emplacement de texture :

```
//On met l'emplacement de texture 0 comme emplacement actif.
glActiveTexture(GL_TEXTURE0);

//On charge la texture de lookup (CurrentLookup) dans l'emplacement actif.
glBindTexture(GL_TEXTURE_2D,CurrentLookup);

//On indique que la texture associée à LookupSampler se trouve à l'emplacement 0
glUniform1i(glGetUniformLocation(ShaderProgramHandle,"LookupSampler"),0);
```


Annexe: Conclusion

À partir des informations contenues dans cette annexe, vous êtes maintenant en mesure de réaliser un nuanceur simple construit à partir d'un nuanceur de vertex et d'un nuanceur de fragment.

Avant de terminer, certains éléments sont intéressants à garder en tête lors de l'écriture et de l'utilisation de nuanceurs :

- Un nuanceur est exécuté seulement sur la géométrie qui est envoyée au processeur graphique. Si aucune géométrie n'est envoyée, le nuanceur n'est pas exécuté.
- Il est habituellement possible de récupérer les erreurs de compilation lors de l'écriture de nuanceur. (Sous OpenGL, on utilise la fonction `glGetShaderInfoLog`)
- La compilation des nuanceurs est relativement rapidement à effectuer et s'effectue alors que le programme est en exécution. Il est donc possible de faire une touche rapide pour recompiler un nuanceur pendant l'exécution, ce qui permet d'écrire un nuanceur et de le tester itérativement sans avoir à recompiler toute l'application

Références recommandées

[Livre] T. Akenine-Möller, E. Haines et N. Hoffman, **« Real-Time Rendering »,3rd Edition,** A.K.Peters, Ltd., Natick, MA, USA, 2008 (chapitre 3)

[Livre] Randi J. Rost, **« OpenGL Shading Language »**, Addison-Wesley, 2004

[Article] P. Barlat, J. Thollot, L. Markosian, **« X-toon, an extended toon shader »,**NPAR '06 : Proceedings of the 4th international symposium on non-photorealistic animation and rendering, Annecy, France, ACM, 2006