

GeForce 8800 : Programmation OpenGL

Christophe [Groove] Riccio

www.g-truc.net

Création : 12 Novembre 2006

Sommaire

Introduction

1. Rappel historique
 2. La programmation GPU chez nVidia
 3. Variables
 - 3.1. Catégorie de données
 - 3.2. Evolution de l'approche
 - 3.3. Bénéfices
 4. Texture
 - 4.1. Unsigned float textures
 - 4.2. Texture buffers
 - 4.3. Représentation entière
 - 4.4. Texture arrays
 - 4.5. Texture compression
 5. Framebuffer
 - 5.1. Framebuffer sRGB
 - 5.2. Z-Buffer flottant 32 bits
 - 5.3. Coverage Sample AntiAliasing
 - 5.4. Render targets blending
 6. Géométrie
 7. GLSL
 - 7.1. Geometry shader
 - 7.2. Mise à jour du langage
- Conclusion
Références

Introduction

Le GeForce 8800 est annoncé et pour une fois les cartes seront disponibles de manière imminente. Cette carte s'impose directement comme la nouvelle référence tant au niveau performance qu'au niveau fonctionnalité qu'au niveau qualité du rendu. Les performances sont impressionnantes et l'architecture raffinée mais ce n'est pas notre sujet. Pour cela, je vous conseille l'excellent article de Hardware.fr. Pour notre part, nous allons discuter de ce qui intéressent le programmeur, les nouvelles fonctionnalités dans le cadre de la programmation avec OpenGL. Cette carte est la première à supporter Direct3D 10 mais toutes les nouvelles possibilités qu'elles apportent sont également disponibles avec OpenGL par le biais d'extensions propriétaires, sous Windows XP également.

1. Rappel historique

Certaine nouvelle génération de carte graphique apporte un nouveau tournant à la programmation graphique, nous sommes en face d'un de ces tournants qui sous OpenGL va bien plus loin que le très médiatique geometry shader. Si nous revenons dans l'histoire des GPUs de nVidia, le GeForce 256 et 2 a permis l'accélération puis plus tard la programmation des sommets, les GeForce 3/4 ont énormément apporté pour la combinaison des textures mais ont également diversifié les formats de textures, cube map, shadows, texture 3d, etc, le GeForce 5 a permis une véritable programmation des traitements sur les fragments, les GeForce 6/7 ont marqués leur temps par le développement du format interne flottant pour des textures de tout type, texture 2D, frame buffer, non power of two textures ce qui a conduit à une certaine hystérie sur le High Dynamic Range (HDR). A présent, l'esprit du GeForce 8800 semble se tourner vers la manipulation des données et leur stockage. Bindable uniform, texture array, texture integer, texture buffer, transform feedback autant de nouvelles notions qui convergent vers cette objectif et qui nous permettra d'entrée vraiment dans l'air des GPGPUs (General Purpose Graphic Processing Unit).

Le GeForce 8800 est accompagné de quelques 21 nouvelles extensions OpenGL ce qui portent le total des extensions OpenGL à plus de 150 sur cette carte. En voici, la liste:

- EXT_bindable_uniform,
- EXT_draw_buffers2,
- EXT_draw_instanced,
- EXT_framebuffer_sRGB,
- EXT_geometry_shader4,
- EXT_gpu_shader4,
- EXT_packed_float,
- EXT_texture_array,
- EXT_texture_buffer_object,
- EXT_texture_compression_latc,
- EXT_texture_compression_rgtc,
- EXT_texture_integer,
- EXT_texture_shared_exponent,
- NV_depth_buffer_float,
- NV_fragment_program4,
- NV_framebuffer_multisample_coverage,
- NV_geometry_program4,
- NV_gpu_program4,
- NV_parameter_buffer_object,
- NV_transform_feedback
- NV_vertex_program4

Ouf ! Contrairement à ce que l'on pourrait croire toutes ces extensions sont exclusivement propriétaires et développées par nVidia. Il y a bien deux exceptions avec une certaine contribution de S3 pour EXT_texture_compression_latc et EXT_texture_compression_rgtc.

2. La programmation GPU chez nVidia

Commençons par expliquer un principe chez nVidia au niveau des langages de programmation GPUs. Bien que l'ARB est décidé de déléster le "langage assembleur" pour la programmation GPU, nVidia continue de le faire évoluer à mesure que de nouvelles fonctionnalités apparaissent. D'un autre coté, GLSL s'est imposé, il était donc naturel de faire évoluer le langage avec cette nouvelle carte. Nous nous retrouvons donc avec un certain nombre d'extensions en doublons parmi lesquelles la moitié ne servira qu'à une poignée de téméraire mais peut-être aussi pour la compatibilité avec Cg. Dans la première catégorie nous retrouvons NV_geometry_program4, NV_vertex_program4, NV_fragment_program4, NV_gpu_program4, NV_parameter_buffer_object et dans la seconde nous avons EXT_bindable_uniform, EXT_geometry_shader4 et EXT_gpu_shader4.

3. Variables

3.1. Catégorie de données

NV_parameter_buffer_object et EXT_bindable_uniform proposent la même nouvelle fonctionnalité mais pas pour le même langage. En terme GLSL, il existe 4 sources de données qu'un shader peut traiter : Les attributs, par exemple la position du sommet, sa normale, ses coordonnées de textures ; les textures avec toute la diversité que cela comprend surtout avec le GeForce 8800 ; les varying variables qui permettent de passer des données d'un shader à un autre et enfin les uniform variables qui sont des données valables pour tous les sommets, tous les triangles et tous les fragments d'un programme GPU. Parmi ces données nous retrouvons habituellement les samplers et autres propriétés des matériaux ou encore les lampes. D'idée derrière ces extensions est de sauver le contenu des variables uniformes dans la mémoire de la carte graphique puis pouvoir inter changer ces variables pour un même programme. Pour profiter de ce comportement, il n'est pas possible de passer par les fonctions du fixed pipeline ce qui enterre un peu plus cette ancienne approche.

3.2. Evolution de l'approche

Concrètement, une scène peut contenir un certain nombre de lampes différentes avec des propriétés très différentes hors si un objet se déplace dans la scène il sera alors soumis à différentes lampes suivant sa position, il suffira alors de changer les bindable uniform buffer objects utilisés par le programme. Plus concret encore, une scène peut être composée d'un certain

nombre d'objets statiques dont chaque objet dispose d'un matériau différent mais le même programme est applicable pour chaque objet. De même pour chaque objet, nous n'avons qu'à changer le bindable uniform buffer object plutôt que de changer la valeur de chaque objet une à une ce qui nous permet de réduire le coup au niveau du CPU. Enfin, un seul bindable uniform buffer object peut-être appliqué à plusieurs programmes différents. Pour un simple rendu d'éclairage multi passes, nous pouvons supposer que plusieurs passes auront besoin de connaître les données sur les lampes. Autre apport très intéressant qui va changer l'approche de la programmation des shaders, c'est que binder un buffer ne signifie pas assigner une valeur. Les valeurs peuvent être assignées à l'initialisation puis sélectionner quand c'est vraiment nécessaire. EXT_bindable_uniform utilise l'excellente API des buffers objects proposés par OpenGL 1.5 pour les vertex buffer objects.

3.3. Bénéfices

Cette extension me ravit particulièrement car je pense qu'elle ajoute de la flexibilité à la programmation GPU qui peut permettre de réduire le nombre de shader utilisés dans un projet tout en réduisant le coût sur le CPU et le coût en mémoire. Cette fonctionnalité sera assurément présente dans la prochaine version d'OpenGL officiellement d'autant qu'elle est actuellement étudiée par l'ARB.

4. Texture

4.1. Unsigned float textures

Parmi les améliorations apportées par le GeForce 8800, il y a également tout ce qui concerne les textures. L'extension EXT_packed_float permet de stocker une couleur RGB sur 32 bits au lieu de 96 bits avec des single floats ou 48 bits avec des half floats. Pour ce faire, 6 bits sont dédiés aux composants rouge et verte mais seulement 5 bits pour le bleu aux niveaux des mentisses et un total de 11 bits pour le rouge et le verte mais seulement 10 bits pour le bleu. Les composantes ne peuvent être signées, il s'agit uniquement de nombres positifs. Nous pouvons considérer que cette extension à porte la notion unsigned float à OpenGL. EXT_texture_shared_exponent poursuit un but similaire avec également un stockage sur 32 bits. Cependant, dans ce cas chaque composant utilise 9 bits pour leur mentisse et partage 5 bits pour les exposants. Il s'agit là encore d'unsigned float

4.2. Texture buffers

Autre nouveau type de texture, les buffer textures via l'extension EXT_texture_buffer_object. Vertex Buffer Object, Pixel Buffer Object, Bindable Uniform Buffer Object et maintenant Texture Buffer Object ? Oui et se sera tout pour le GeForce 8800. Ce nouvel objet est destiné à stocker des données qui seront par la suite accessibles seulement en utilisant des coordonnées de texture non normalisées. Les mipmaps, le filtrage ne sont pas disponibles. Il est également possible de récupérer les varying variables des vertex et geometry shaders dans ce buffer par une action conjoint avec l'extension NV_transform_feedback. Ce flux de sortie, au niveau du vertex shader ou du geometry shader, peut bien sûr être copié dans d'autres buffer objects comme le vertex buffer object ce qui introduit la notion d'itération au niveau des vertex et geometry shader c'est d'ailleurs ainsi que la subdivision de la géométrie est rendu possible pour un nombre N d'itérations.

4.3. Représentation entière

Proche de l'extension EXT_texture_buffer_object mais plus encore des textures traditionnelles d'OpenGL, l'extension EXT_texture_integer permet d'utiliser des textures de tout type dont la représentation des données dans les shaders se fait via des nombres entiers. Bien que dans la mémoire une texture 2D soit stocké la plus part du temps sous forme d'entiers, dans les shaders chaque composante s'affiche comme un nombre flottant entre 0 et 1. Avec cette extension ce n'est plus le cas, nous pouvons accéder aux données comme des nombres entiers signés ou non. Tout ce passe au niveau du format de stockage interne des textures du côté de l'API OpenGL GL_RGB8UI_EXT, par exemple et via de nouvelles fonctions de texture lookup dans GLSL.

4.4. Texture arrays

Comme son nom l'indique, EXT_texture_array apporte la notion de tableau de textures. Un tableau de texture à une dimension correspond à une collection de texture 2D et un tableau de texture à deux dimensions correspond à une collection de texture à 3 dimensions. Chaque texture de chaque tableau devant avoir la même taille. Pour accéder à chacune des entrées du tableau, nous pouvons utiliser les coordonnées de texture, la coordonnée correspondant à l'entrée du tableau étant

envoyée dans le shader sous forme d'un nombre flottant non normalisé. Excepté ceci, les tableaux de textures se comportent comme des textures 2D ou 3D au détail près qu'il n'y a pas de filtrage entre deux entrées du tableau mais cette opération peut-être réalisée dans le shader.

4.5. Texture compression

Deux nouvelles extensions apportent deux nouveaux formats de compressions ... ou un seul. D'un coté, nous avons EXT_texture_compression_rgtc (Red-Green Texture Compression) qui permet de stocker de manière compressée sous forme de block (à la manière du S3TC) une texture composée de une ou deux composantes chromatiques. D'un autre coté, nous avons GL_EXT_texture_compression_latc (Luminance-Alpha Texture Compression) qui en fait exactement de même et avec le même taux de compression (2x). A vrai dire, je suppose que la présence des deux extensions est d'avantage une histoire de sémantiques ou de compatibilité avec Direct3D 10. Ces formats sont appropriés pour le stockage de normalmaps car une normal est normalisé, nous pouvons donc facilement retrouver la dernière composante : $B = 1 - R - G$.

5. Framebuffer

5.1. Framebuffer sRGB

Nous arrivons à un nouveau chapitre des nouveautés de cette carte, tout ce qui concerne le framebuffer object. Bien que cette notion ne soit pas encore intégrée à la spécification d'OpenGL, c'est devenu une fonctionnalité fétiche pour les programmeurs qui souhaitent faire du rendu sur texture car c'est par ce moyen que sont garanties les meilleures performances. Le format du color buffer peut à présent être du sRGB grâce à l'extension EXT_framebuffer_sRGB. Pour rappel, le format sRGB est un format normalisé IEC qui caractérise la nature non linéaire des écrans particulièrement dans les salles mal éclairées, fondamentalement il s'agit d'une correction de gamma 2.2.

5.2. Z-Buffer flottant 32 bits

EXT_packed_depth_stencil est une extension qui permet un stockage sur 32 bits de 24 bits pour le Z-Buffer et 8 bits pour le stencil buffer dans une texture. Avec le GeForce 8800 vient s'ajouter NV_depth_buffer_float qui contient un nouveau format permettant de stocker sur 64 bits, un Z-Buffer 32 bits et un stencil buffer de 8 bit, soit 24 bits non utilisés. Précisons que l'intérêt de l'extension, mise à part le fait de gaspiller la mémoire, est d'avoir un Z-Buffer codé sur des nombres flottants.

5.3. Coverage Sample AntiAliasing

Le GeForce 8800 apporte un nouveau mode d'antialiasing appelé CSAA (Coverage Sample AntiAliasing) qui utilise un tableau de booléens (par exemple 16 par pixel pour de CSAA 16x) pour déterminer un pourcentage de recouvrement du pixel final par des triangles. Pour supporter cette fonctionnalité en rendu sur texture, nous disposons de l'extension NV_framebuffer_multisample_coverage.

5.4. Render targets blending

Pour en terminer avec cette partie sur le rendu sur texture, nous avons l'extension GL_EXT_draw_buffers2 qui vient compléter GL_ARB_draw_buffers. Il s'agit de permettre l'écriture dans plusieurs color buffer depuis un fragment shader. C'est fonctionnalité est souvent connu sous le nom render to target (RTT) qui est le fondement de l'approche des « deferred shaders ». GL_EXT_draw_buffers2 permet d'assigner des masques de couleurs différents pour chacun des tampons chromatiques. De plus, elle permet d'activer ou non le blending pour chacune des cibles bien que la fonction et l'équation du blending demeure les même pour tous les tampons chromatiques. Parler de tampon chromatique est abusif car l'idée des deferred shaders est par exemple d'utiliser l'une des cibles supplémentaires pour stocker les normales. L'intension est louable avec cette extension mais il y a encore beaucoup à faire mais un blending programmable à l'avenir reste d'actualité.

6. Géométrie

Dans un registre totalement différent, nous voici à réfléchir sur la géométrie avec GL_EXT_draw_instanced. Imaginons que nous souhaitions afficher plusieurs meshes identiques mais à des positions différentes. Habituellement, nous ferions autant de batches (glDraw*) que le nombre de fois que l'on souhait dessiner ce mesh. Avec cette extension, un seul batch suffit.

Un identifiant unique est passé pour chaque instance (`gl_InstanceID`) ce qui permet d'appliquer des traitements individuels dans le vertex shader, comme une transformation. Une nouvelle initiative qui va soulager le CPU.

7. GLSL

7.1. Geometry shader

Un bon nombre des extensions décrites précédemment ont des implications au niveau de GLSL ainsi qu'avec l'API permettant l'interaction en le CPU et le GPU. L'extension `GL_EXT_gpu_shader4` gère ces changements et `GL_EXT_geometry_shader4` s'occupe d'intégrer la notion de geometry shader à OpenGL qui, pour rappel, devrait s'appeler primitive shader selon la terminologie OpenGL. Les geometry shaders apportent de nouvelles formes géométriques pour décrire le voisinage de chaque primitive, ce qui permet d'appliquer la subdivision des triangles, le Trueform d'ATI si l'on veut mais aussi d'autres méthodes. Contrairement à Direct3D 10, la syntaxe des geometry shaders n'est pas tordue, en fait il s'agit de la même jusqu'alors utilisée pour les vertex shaders et les fragment shaders avec cette fameuse notion d'unification, c'est-à-dire que toutes les fonctionnalités sont disponibles pour tous les shaders. Il existe cependant deux fonctions uniquement disponibles sur les geometry shader, `EmitVertex()` et `EndPrimitive()` dont le fonctionnement se comprend facilement lorsque l'on se refait au mode immédiat d'OpenGL. `EmitVertex()` est alors l'équivalent de `glVertex*()` et `EndPrimitive()` l'équivalent de `glEnd()`. `glBegin()` est ici implicite, lorsque l'on rentre dans un geometry shader on peut considéré qu'un appel de ce type est lancé ainsi qu'après chaque appel de `EndPrimitive()`. Les données attribuées à chaque vertex créé correspondent les valeurs attribuées aux varying variables.

7.2. Mise à jour du langage

Au niveau de GLSL, il faut compter sur deux nouvelles fonctions, `round` et `truncate`, ainsi qu'une revalorisation des nombres entiers. Ils sont maintenant codés sur 32 bits, les opérateurs bits à bits sont supportés, les fonctions `abs`, `sign`, `min`, `max` et `clamp` supportent les nombres entiers, de nouveaux types d'entier non signé sont disponibles : `unsigned int`, `uvec2`, `uvec3` et `uvec4` et un grand nombre de nouveau sampler pour la représentation en nombre entier du contenu des textures. Au niveau des fonctions pour les textures c'est une vraie hémorragie, entre celle pour le texture fetch, pour récupérer les dimensions, accéder au tableau de textures mais c'est sans compter deux nouvelles notions, celle de gradient (dérivée partielle sur chaque composante) ou celle d'offset, qui permet de décaler via une somme les coordonnées de texture.

Conclusion

Nous avons survolé les nouvelles possibilités qu'apportent le GeForce 8800 mais il nous reste une question que vous vous posez sûrement : Est-ce que cette carte, voir cette génération de cartes, va rendre les graphismes plus beau ou apporter de nouveaux effets ? Honnêtement, mon point de vue est non mais je suppose que les gars du marketing de nVidia ne sont pas de mon avis. Ils pourront argumenter avec l'idée suivant : La plus part des nouvelles fonctionnalités du GeForce 8800 permettent d'accélérer le rendu, réduire l'espace mémoire utilisé ou baisser la charge sur le CPU, ce qui permet d'augmenter le niveau des détails dans les jeux. Rentrant dans cette case, il y a le fameux (fumeux ?) geometry instancing qui permet par exemple de créer de vraies particules sans pré-allouer de larges tableaux, ou la possibilité d'écriture dans plusieurs layers d'un cube map dans un geometry shader en une seule passe. Il reste aussi un dernier point important, le confort pour le programmeur ! Toutes les évolutions sur les nombres entiers et sur les sources de données permettent enfin de manipuler les textures naturellement avec des gains niveaux performances et espace mémoire une nouvelle fois. Il s'agit peut-être d'une voie vers plus de post processing du rendu, plus de physiques dans le GPU ou de manière générale plus de traitements non directement liés au rendu graphique. En fait, il s'agit peut-être d'une voie vers la GPGPU.

Références

NVIDIA, *NVIDIA OpenGL Extension Specification*, 2006
http://developer.nvidia.com/object/nvidia_opengl_specs.html

Damien Tricot, *nVidia GeForce 8800 GTX & 8800 GTS*, Hardware.fr, 2006,
<http://www.hardware.fr/articles/644-1/nvidia-geforce-8800-gtx-8800-gts.html>