HAL
open science

Rendu sur GPU pour écrans autostéréoscopiques

François de Sorbier, Vincent Nozick, Venceslas Biri

To cite this version:

François de Sorbier, Vincent Nozick, Venceslas Biri. Rendu sur GPU pour écrans autostéréoscopiques. 21èmes journées de l'Association Française d'Informatique Graphique (AFIG'08), Nov 2008, France. pp.245-252. hal-00733831

HAL Id: hal-00733831

https://hal.science/hal-00733831

Submitted on 19 Sep 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AFIG 2008 / Toulouse
IRIT Presse

Rendu sur GPU pour écrans autostéréoscopiques

François de Sorbier et Vincent Nozick et Venceslas Biri

Université Paris-Est
Laboratoire d'Informatique de l'Institut Gaspard Monge
UMR CNRS 8049
5 bd Descartes, 77454 Marne la Vallée Cedex 2, France
{fdesorbi,vnozick,biri}@univ-mlv.fr

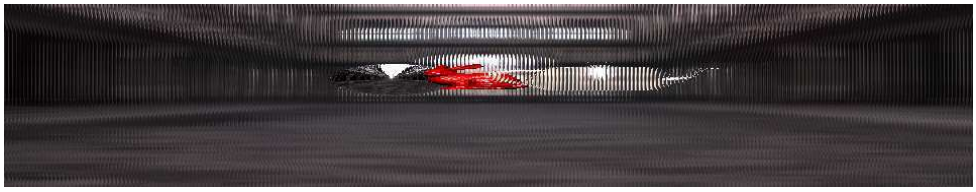

Figure 1: *Résultat pour écran autostéréoscopique composé de 4 images obtenues en temps réel et en une seule passe.*

Abstract

Ces dernières années, le développement des interfaces stéréoscopiques a permis l'apparition des écrans autostéréoscopiques. Cette catégorie de matériel permet de se passer du port de lunettes dédiées, mais nécessite d'afficher plusieurs vues d'une même scène. Dans le cas de la synthèse d'images en temps-réel, la méthode standard repose sur le rendu depuis des points de vue distincts. Ce papier présente une approche alternative permettant de générer les diverses vues en une seule passe. Notre méthode est fondée sur le fait que les sommets les mêmes propriétés d'une vue à l'autre. En se servant de la dernière version des GPU, nous proposons d'améliorer les temps de calcul en utilisant les geometry shaders pour dupliquer et transformer les primitives pour chacun des points de vue. Notre méthode ne nécessite que de très peu de modifications pour être utilisée avec un système de stéréoscopie classique.

Keywords: autostéréoscopie, GPU, geometry shaders, temps réel

1 Introduction

L'utilisation de la stéréovision dans le contexte de la réalité virtuelle est une technique classique pour accroître l'immersion d'un utilisateur. Le principal bénéfice de l'ajout du relief est qu'il facilite la perception de l'environnement virtuel qui est affiché. Récemment, il a été possible de constater la prolifération de nouveaux systèmes stéréoscopiques appelés écrans autostéréoscopiques. Contrairement à la plupart des solutions existantes, ce type de matériel permet de se passer de l'utilisation d'interfaces spécifiques comme des lunettes. En plus de cela, certaines catégories d'écrans autostéréoscopiques permettent d'observer une même scène depuis différentes positions, ce qui est avantageux pour des applications multi-utilisateurs. Ce nouveau principe stéréoscopique implique de nouvelles contraintes à prendre en considération comme le rendu, le transfert et l'acquisition des données. Concernant ce dernier point, nous pouvons le diviser en deux catégories : soit l'information peut être récupérée depuis des caméras vidéos, soit générée en utilisant des méthodes de l'informatique graphique. Nous nous focaliserons plus particulièrement sur la seconde catégorie qui nécessite qu'une scène soit rendue plusieurs fois. Cette phase implique de nombreuses redondances dans les calculs et ne permet pas toujours un rendu en temps réel.

Cet article présente une nouvelle méthode de synthèse d'images profitant des geometry shaders pour effectuer plusieurs rendus d'une scène depuis différents points de vue en une seule passe. Comme la seule différence, d'une image à l'autre, est la position de la « caméra », nous pouvons en déduire que certaines propriétés liées à la scène sont conservées. Par exemple, la géométrie, le calcul de la couleur des objets ou une partie de l'illumination vont rester inchangés. Nous pouvons donc bénéficier des avantages du GPU pour évaluer les attributs des sommets une unique fois, puis « cloner » les données pour obtenir les différentes vues

© Association Fran?aise d'Informatique Graphique 2008.

destinées à un écran autostéréoscopique. Le but de notre méthode est donc d'améliorer les temps de rendu en n'ayant plus qu'une seule passe regroupant une partie des calculs. Nous commencerons par présenter la technologie des écrans autostéréoscopiques avec les méthodes existantes permettant de réduire les temps de rendu lors de la génération de plusieurs vues. Puis, nous décrirons notre méthode fondée sur l'utilisation du GPU et, avant de conclure, nous discuterons des résultats obtenus.

2 Le rendu pour écrans autostéréoscopiques

Dans cette partie, nous présentons un bref aperçu des technologies autostéréoscopiques existantes avant de décrire les solutions de rendu multi-vues existantes.

2.1 Les écrans autostéréoscopiques

Le principe de la stéréovision est de produire deux images d'une même scène, depuis deux points de vue légèrement décalés, dont l'une est déstinée à l'œil droit et l'autre à l'œil gauche. La technique des anaglyphes, de la polarisation de la lumière ou de l'obturation comptent parmis les plus répandues [Oko77], mais oblige l'observateur à porter une paire de lunettes dédiée et qui, dans certains cas, peut se révéler contraignant. Cependant, les dernières recherches sur l'autostéréoscopie, facilité par le développement des écrans LCD, a apporté lse bénéfices suivants à la stéréovision :

- utilisation aisée pour des application multi-utilisateurs
- pas besoin de porter une quelconque interface
- différentes paires d'images stéréoscopiques sont observables

Dodgson [Dod05] présente une grande variété de procédés de restitution autostéréoscopiques. Parmi ceux-ci, seulement deux techniques ont été rendues disponibles sur le marché par le biais de constructeurs tels que Philips [Phi06] ou Sharp [Sha06]. La première technique décrite par [Ive28], dite à barrière de parallaxe, se fonde sur l'utilisation de fines lamelles positionnées avec précision sur la surface d'un écran LCD. Disposés de cette manière, ces obturateurs définissent un ensemble de pixels qui ne sont visibles que par l'œil droit ou par l'œil gauche depuis une position d'observation donnée. La seconde technique, dont le détail est donné dans [SSAE99], repose sur l'apposition d'un réseau lenticulaire sur la surface d'un écran LCD. Chaque lentille va diriger la lumière émise par un pixel dans une direction bien précise. En considérant que qu'un individu est correctement positionné face à l'écran, ses yeux vont pouvoir observer deux images avec un point de vue légèrement décalé. En augmentant le nombre de pixels présents sous chaque lentille, il devient possible d'accroître le nombre de points de vue stéréoscopiques (Takaki [Tak06] peut en obtenir jusqu'à 64). De plus, contrairement aux barrières parallaxes qui souffrent d'une diminution de la luminosité en raison de la présence des lamelles, l'autostéréoscopie à réseau lenticulaire permet de conserver la totalité de la luminosité.

2.2 Génération de plusieurs vues

Comme cela a été évoqué précédemment, l'acquisition des données peut se faire via une caméra vidéo ou alors en utilisant la synthèse d'images. Quel que soit le procédé employé, la création d'un contenu destiné à un écran autostéreoscopique peut être problématique. Dans le cas d'un flux vidéo, l'utilisation de plusieurs caméras sur un même ordinateur peut entraîner une saturation lors de la récupération des données. Une solution contournant cette limite a donc été proposée par Nozick et Saito [NS07]. Elle repose sur la technique de rendu à base d'images qui offre la possibilité de générer de nouveaux points de vue d'une scène à partir d'un nombre limité de caméras.

Dans le cas de la synthèse d'images, la méthode traditionnelle consiste à effectuer plusieurs rendus d'une même scène depuis différents points de vue. Si cela ne pose pas de problème dans le cadre d'un rendu stéréoscopique simple, la création d'un nombre plus important d'images peut se révéler coûteuse dans l'optique d'obtenir un affichage en temps réel. Par exemple, les écrans de la firme Philips nécessite de générer 9 vues. Diverses méthodes ont été proposées pour permettre le rendu multi-vues en temps réel à partir de données 3D. La première, décrite par Hübner *et al.* [HZP06], est dédiée au contexte du rendu à base de points. Elle tire avantage du GPU pour calculer le splatting multi-vues, les intersections paramétriques des splats et, enfin, les intersections entre les disques et les rayons issus des points de vue, le tout en une seule passe. Cette technique permet d'atteindre 10 images par seconde pour générer 8 vues d'une scène composée de 137000 points. Une seconde méthode a été présentée par Hübner et Pajarola [HP07], mais se destine au rendu en volume direct. Le GPU est pleinement utilisé pour calculer les différentes vues à partir de textures 3D en une seule passe.

Les deux précédentes méthodes permettent effectivement d'obtenir un rendu multi-vues en temps réel, mais ne s'applique toutefois pas à des données fondées sur des polygones. Pour cette catégorie, Morvan *et al.* [MFdW07] ont proposé de tirer parti des informations issues d'une carte de profondeur pour obtenir les vues supplémentaires souhaitées. Le principal avantage de cette technique est de diminuer la quantité de données envoyée vers l'écran autostéréoscopique. Néanmoins, le processus de génération des vues, fondée sur une estimation du résultat pour les parties dont il manque des données, provoque l'apparition d'artefacts.

3 Notre méthode de rendu multi-vues

Cette partie commence par une présentation des *geometry shaders*, pour ensuite montrer comment cette technolo-

© AFIG 2008.

gie a été exploitée pour améliorer le rendu multi-vues. Nous décrivons egelment la marche à suivre pour intégrer notre méthode au processus de rendu classique.

3.1 Les geometry shaders

Ces dernières années, les *vertex* et *fragment shaders* ont bénéficié d'un fort engouement de la part de la communauté graphique. En comparaison des CPU ou des anciennes versions des GPU, ils ont l'avantages de permettre d'accélérer les temps de rendu et d'améliorer la qualité du rendu.

La quatrième version du *shader model* a introduit un nouveau genre de traitement nommé *geometry shaders* [LB07]. Dans la chaîne d'opérations d'OpenGL, ce dernier prend place entre le *vertex shader* et la phase de *clipping*, comme illustré sur la figure 2. L'objectif de ce *shader* est de pouvoir manipuler, sur GPU, certaines primitives en offrant la possibilité de les transformer, dupliquer ou bien d'agir sur les sommets les composant. Étant donné un ensemble d'éléments en entrée, le *geometry shader* permet d'obtenir des primitives comme des points, des lignes ou des triangles.

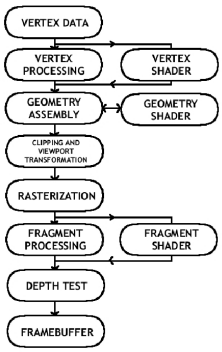

Figure 2: *Le pipeline OpenGL avec l'introduction de la version 4 du shader model.*

Les sommets de la primitive qui va être produite par le *geometry shader* sont déterminés via le mot clé `EmitVertex`. La primitive est finallement réintroduite dans la chaîne de traitement OpenGL par la fonction `EndPrimitive`.

3.2 Notre algorithme

Le principal objectif de notre méthode, présentée dans la figure 3, est d'accélérer le rendu des différentes vues d'une même scène. Pour cela, nous nous servons du fait que les proprietés des sommets sont conservées d'un point de vue à l'autre. Ainsi, nous pouvons constater que la position de chaque sommet, sa normale, sa couleur, ses coordonnées de texture vont rester inchangées. Nous proposons donc de ne calculer qu'une seule fois ces attributs dans le *vertex shader* ce qui va permettre de réduire les temps de calcul.

Une fois l'ensemble des sommets traités et assemblés

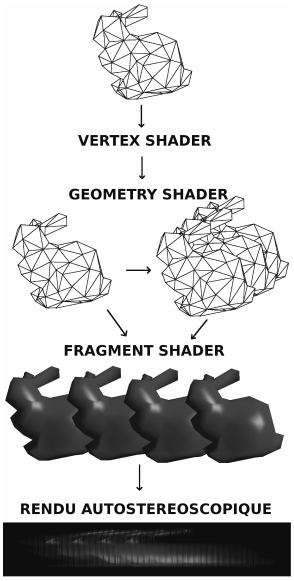

Figure 3: *Illustration du principe de notre méthode.*

pour former des primitives, ils sont envoyés vers le *geometry shader*. Le principe consiste alors à y dupliquer chaque primitive en autant de vues souhaitées. Chacun de ces « clones » subit ensuite une transformation propre, dépendant du point de vue auquel il est associé. Les différents résultats sont finalement transmis au *fragment shader* pour procéder à la rasterization puis au rendu.

Pour obtenir N vues, il est nécessaire de produire N buffers de rendu. Le rôle du *pixel shader* est alors d'affecter les fragments qu'il reçoit dans le bon buffer, c'est à dire en tenant compte de la vue à laquelle il est associé.

L'algorithme peut être résumé de la manière suivante :

Algorithm 3.1: MULTIVIEW(N)

comment: N le nombre de vues souhaitées
comment: Précalculer les N matrices projection
Calculer les N matrices modelview
Définir les propiétés des sommets
for each *primitive* $\in$ *scene*
do { **for each** *vue* $\in N$
  do { $p \leftarrow$ *primitive*
    Transformer p en fonction de *vue*
    output (p) }
}
Affecter les fragments aux N buffers

© AFIG 2008.

3.3 Détails de notre méthode

Cette partie détaille le processus qui permet d'effectué un rendu multi-vues reposant sur l'utilisation intensive des *geometry shaders*.

3.3.1 La duplication de la géometrie

Comme cela a déjà été evoqué, notre méthode repose sur le « clonage » des primitives dans le *geometry shader*. *N* nouvelles primitives sont générées en concordance avec les matrices de transformation permettant d'obtenir une projection et un positionnement en fonction du point de vue désiré.

Le code GLSL [Ros06] suivant décrit ce processus :

Listing 1: *Code GLSL du geometry shader*

```
#version 120
#extension GL_EXT_geometry_shader4 : enable
#extension GL_EXT_gpu_shader4 : enable

// maximum number of views
const int MAX_VIEW = 8;
// id transmitted to the fragment program
flat varying out float flag;

// number of rendered view
uniform int nview;
// reference view id
uniform int ref;
// views' projection matrices
uniform mat4 matrix[MAX_VIEW];

void main(void)
{
 // reference view processing
 flag = float(ref);
 // for each vertex of the triangle
 for(int i=0; i < 3; ++i){
  gl_Position =
   gl_ModelViewProjectionMatrix *
   gl_PositionIn[i];
  EmitVertex();
 }
 EndPrimitive();

 // additional views processing
 for(int v=0; v<nview; ++v){
  if(v!=ref){
   flag = float(v);
   for(int i=0; i<3; ++i){
    gl_Position = matrix[v] *
     gl_PositionIn[i];
    EmitVertex();
   }
   EndPrimitive();
  }
 }
}
```

Pour éviter d'avoir à multiplier la matrice de projection avec la matrice modelview, nous effectuons cette opération dans le programme OpenGL principal. ed cette façons nous préservons l'utilisation de calculs redondants pour chacune des primitives en passant directement le résultat des multiplications au *geometry shader*.

Dans l'étape de traitement suivante, il est impossible, en l'état, de savoir à quelle vue est déstiné un fragment. Nous avons donc mis en place en système d'identification dans le *geometry shader* qui s'illustre par la variable `flag` dans les listings 1 et 2. Avant d'emettre la nouvelle primitive, nous lui associons la variable `flag` en lui attribuant le numéro de la vue en cours de traitement. La principale contrainte est d'outrepasser la phase d'interpolation des valeurs transmisent. Cela est rendu possible par l'utilisation du mot clé `flat`.

3.3.2 Le rendu multi-vues

En se référant à la valeur de la variable `flag`, les fragments sont envoyés vers le buffer image correspondant au point de vue auquel il est destiné. Pour pouvoir faire le rendu dans plusieurs buffers, il est nécessaire d'utiliser l'extension *Multiple Render Target* (*MRT*). Cependant, son emploi introduit une limite au niveau du test de profondeur sur les fragments. Bien qu'il y ait plusieurs buffers à disposition, le test ne s'applique qu'au final que sur le premier. Différentes incohérences visuelles vont donc apparaître sur les autres vues. Une solution consiste à appliquer notre propre test de profondeur avec, par exemple, l'« algorithme du peintre » même si ce dernier est bien connu pour ne pas être optimal.

Une autre contrainte, imposée par l'extension *MRT*, oblige à envoyer chacun des fragments vers l'ensemble des buffers de rendu, sans quoi des artefacts apparaissent lors de l'affichage. Pour résoudre cette limite, nous nous servons du canal alpha des fragments pour les rendre invisibles lorsqu'ils sont dirigés vers le mauvais buffer. Dans ce cas, la valeur alpha est définie à 0 et à 1 si le fragment est envoyé vers le bon buffer. Finalement l'utilisation de l'*alpha test* va permettre de supprimer les fragments non voulus. Cela n'empêche bien sûr pas d'utiliser la valeur alpha pour effectuer des effets de transparence.

Le code GLSL suivant décrit l'étape de rendu dans le *fragment shader* :

Listing 2: *Code GLSL du fragment shader*

```
#version 120
#extension GL_ARB_draw_buffers : enable

flat varying in float flag;
uniform int nview;

void main()
{
```

© AFIG 2008.

```
  // Any fragment shader source code
  // like illumination, texture...

  for(int i=0;i<nview;++i){
   if(float(i)==flag)
    gl_FragData[i]=vec4(colorfrag.rgb,1.0);
   else
    gl_FragData[i]=vec4(0.0);
  }
}
```

Malgré les contraintes liées à l'extension *MRT*, l'ensemble des opérations usuelles comme le calcul de l'illumination par pixel ou l'application d'un environment mapping. Un résultat obtenu pour 8 vues est présenté dans la figure 4.

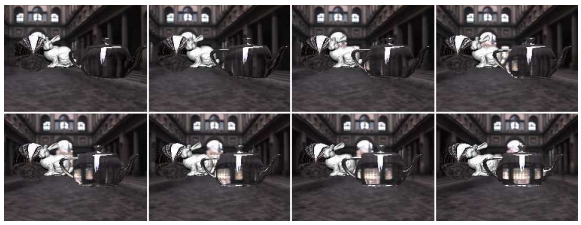

Figure 4: *8 vues d'une même scène, générées en une seule passe avec notre méthode.*

3.3.3 Autres pistes

Pour remédier aux problèmes liés à l'extension *MRT*, il existe d'autres possibilités. La première consiste à ne produire qu'une seule image qui va contenir, consécutivement, l'ensemble des vues de la scène. Dans ce cas il n'y a plus de contrainte lié à la destination des fragments, ni au test de profondeur car il n'y a plus qu'un seul buffer de rendu. Par contre, il est nécessaire d'appliquer un décalage sur les triangles afin, qu'après la rasterization, les fragments soient associés à la bonne zone de l'image. Cela implique également l'utilisation d'un *clipping* sur les triangles qui sont à cheval sur deux vues. Toutefois, une suppression pure et simple d'un tel triangle va produire un effet d'apparition et disparition brutale . Enfin, il faut également considérer que la taille d'un buffer de rendu est limité, ce qui induit que le nombre de vues est aussi limité.

Une seconde solution est d'utiliser notre propre buffer de profondeur pour effectuer les tests de visibilité. Dans les shaders, cela passe nécessairement par la lecture et l'écriture simultanées dans une texture active dédiée. Cette spécificité des cartes graphiques n'est malheureusement pas disponible pour le moment.

3.3.4 Génération de l'image finale

Les écrans autostéréoscopiques nécessitent chacun une manière spécifique de composer les images des différentes vues pour pouvoir observer l'effet de relief. La figure 5 présente deux versions d'agencement des pixels sur l'image finale transmise à l'écran. Cela implique donc de récupérer l'ensemble des vues obtenues lors de l'étape précédent, et de leur appliquer le traitement adéquat en fonction du matériel utilisé. Pour cela, nous utilisons extension *FrameBuffer Object* (*FBO*) permettant de spécifier la destination du rendu. En définissant que le résultat du rendu doit se faire vers des textures, il devient possible de les réutiliser directement pour les fusionner dans un *shader* par exemple.

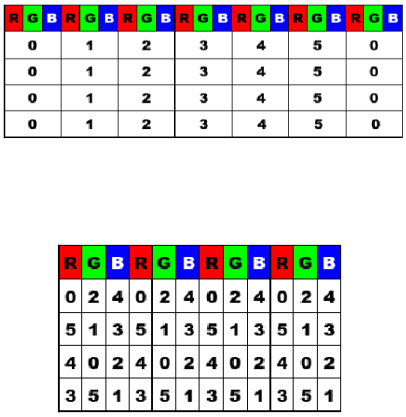

Figure 5: *Deux répartitions possibles des pixels issus des différentes images.*

Pour le moment, le nombre maximal de textures utilisables en même temps est limité, par la carte graphique, à 8. Par conséquent, le nombre maximal de vues que l'on peut générer est restreint à la même valeur. Dans l'attente de l'amélioration des drivers ou du matériel graphique, nous proposons d'augmenter simplement le nombre de vues en ajoutant des passes supplémentaires ayant chacune un *FBO* propre. Même en utilisant deux passes pour obtenir, par exemple, un total de 12 vues, notre solution permet de conserver un affichage en temps réel.

Concernant le mélange des pixels pour obtenir l'image finale, nous proposons pour le moment qu'une approche simple fondée sur le parcours des différentes textures. En attendant de proposer une solution faisant pleinement usage des *shaders*, quelques optimisations peuvent être obtenus dans [KPG*07]. Deux exemples sont présentés dans la figure 1 et la figure 6

4 Résultats et discussion

Pour évaluer les performances, nous avons appliqué notre méthode avec l'OS Linux sur une machine PC Intel core 2 DUO 2,40GHz incluant une carte graphique nVidia GeForce

© AFIG 2008.

8800 GTX. La scène, composée d'environ 80 000 triangles, fait appel à différents effets visuels comme l'environment mapping, l'illumination par pixels ou des réfractions. Une résolution possible d'un écran stéréoscopique étant de 1920×1080 et en se fondant sur une répartition en bandes des vues, on peut en déduire que pour les 8 images à générer ont une résolution de 240×180 (conservation d'un ratio 4 :3). Notre méthode peut bien sûr être utilisée avec des résolutions plus élevées.

Nombre de vues	1	2	4	8
Rendu traditionnel (img\s)	157	80	38	19
Notre méthode (img\s)	157	141	70	24

Table 1: *Comparaison de notre méthode avec un rendu traditionnel*

La table 1 présente les résultats obtenus avec notre nouvelle approche et sont comparés à ceux de la méthode traditionnelle. Il est possible d'observer que notre méthode accélère les calculs de rendu et est particulièrement efficace lorsque le nombre de vues est peu élevé. Pour expliquer la faible différence des résultats lorsque le nombre de vue est important, on peut se référer à l'opinion de la communauté d'informatique graphique qui tend à dire que la version actuelle des *geometry shaders* n'est pas encore optimale. Ainsi, lorsque le nombre de vues est supérieur à 4, il se produit un goulet d'étranglement dans le traitement des primitives. Une autre remarque concerne le passage de valeurs du *geometry shader* au *fragment shader*. Au cours de nos différents tests, nous avons pu observer que transmettre 13 valeurs au lieu de 14 pouvait quasiment diviser par 2 les temps de calcul lors de la génération de 4 vues.

Nous avons également appliqué notre méthode en faisant varier les types d'illuminations pour privilégier l'utilisation d'un *shader*, plutôt qu'un autre. Les résultats sont présentés dans la table 2.

Nombre de vues / (ips)	2	4	6	8
Sans illumination	326	168	80	60
Textures	326	167	80	60
Illumination par sommet	160	83	28	23
Illumination par pixel	160	83	28	24

Table 2: *Performances obtenues en fonction de différents types de colorisation des objets*

Le constat le plus important concerne les deux dernieres lignes qui illustre un calcul par sommet et un calcul par pixel. En effet, on peut observer que les résultats sont quasiment similaires alors qu'on pouvait s'attendre à des performances élevées lors du traitement des sommets. La raison est liée à l'architecture unifiée des dernières cartes graphiques qui permet de répartir les charges de calcul en attribuant plus ou moins d'unités à un *shader*.

Nos dernières évaluations ont porté sur l'étude des performances en tentant de générer plus de 8 vues. Cela signifie donc qu'il est nécessaire d'effectuer plus d'une passe en raison de la contrainte imposée par l'extension *FBO*. Les résultats sont présentés dans la table 3.

Nombre de passes	1	2	3	4	6
Vues par passe	12	6	4	3	2
Vitesse de rendu (ips)	N/A	14	15	23	21

Table 3: *Génération de douze rendus depuis douze points de vue en modifiant le nombre de passes et le nombre de vues par passe*

Outre la possibilité de générer plus de 8 vues en conservant toujours le temps réel, nous pouvons observer que les meilleures performances sont obtenues en effectuant trois rendus par passe. Cela tend à confirmer qu'il existe un goulet d'étranglement au niveau des *geometry shaders* qui prend de l'importance au-delà de la génération de 3 vues par passe. *A contrario*, nous pouvons qu'avec la génération de seulement deux vues, nous ne tirons pas complètement avantage des bénéfices des *geometry shaders*. Il est donc important de noter que l'efficacité de l'algorithme dépend alors d'un ratio entre le nombre de passes et le nombre de vues générées dans chacune de ces passes.

Lors de la création d'un maximum de 8 vues, nous pouvons dire que notre méthode, qui ne requiert que très peu de modifications dans le processus de rendu, dispense différents apports :

- $nview - 1 \times nb_primitives \times nb_vertex_shader_op$ operations économisées sur le calcul des sommets
- les données ne sont rendues qu'une seule fois
- Moins d'échanges de données entre le CPU et le GPU

En ce qui concerne la limite introduite par l'extension *MRT*, qui empêche l'utilisation du test de profondeur, nous attendons l'arrivée d'une prochaine amélioration des cartes graphiques qui permettrait, par exemple, la lecture et l'écriture dans une texture active.

5 Conclusion

Cet article présente une nouvelle approche pour générer de multiples vues en temps réel en utilisant le GPU. Destinée aux écrans autostéréoscopiques, notre méthode repose sur l'utilisation d'une seule passe rendue possible par l'arrivée des *geometry shaders*. Ces derniers permettent de n'appliquer qu'une seule fois les différents traitements appliqués aux sommets car ils rendent possible la duplication et la transformation de primitives juste avant l'étape de rasterization. Malgré des contraintes imposées par les extensions FBO et MRT qui limite le nombre de vues et retire la possibilité d'utiliser le buffer de profondeur, notre méthode offre des performances supérieures comparé au rendu multi-vues

© AFIG 2008.

traditionnel. Les prochaines évolutions apportées à ces deux extensions permettront d'améliorer les résultats tout en rendant l'implantation encore plus aisée. Outre la possibilité de destiner notre méthode à n'importe quel support autostéréoscopique, il est également possible de l'utiliser dans un contexte stéréoscopique simple.

Notre volonté pour la suite est d'essayer de trouver un moins permettant de fusionner le rendu multi-vues à l'étape de création de l'image finale déstinée à être affichée. Nous envisageons également de d'étudier les possiblités d'utilisation des écrans autostéréoscopiques en réalité virtuelle.

References

[Dod05] DODGSON N. A. : Autostereoscopic 3d displays. *IEEE Computer 38*, 8 (2005), 31–36.

[HP07] HÜBNER T., PAJAROLA R. : Single-pass multi-view volume rendering. In *IADIS* (2007).

[HZP06] HÜBNER T., ZHANG Y., PAJAROLA R. : Multi-view point splatting. In *GRAPHITE* (2006), pp. 285–294.

[Ive28] IVES H. E. : A camera for making parallax panoramagrams. In *Journal of the Optical Society of America* (1928), no. 17, pp. 435–439.

[KPG*07] KOOIMA R., PETERKA T., GIRADO J., JINGHUA G., SANDIN D., DEFANTI T. : A gpu sub-pixel algorithm for autostereoscopic virtual reality. In *IEEE Virtual Reality 2007* (2007), pp. 131–138.

[LB07] LICHTENBELT B., BROWN P. : *EXT_gpu_shader4 Extensions Specifications*. NVIDIA, 2007.

[MFdW07] MORVAN Y., FARIN D., DE WITH P. H. N. : Joint depth/texture bit-allocation for multi-view video compression. In *Picture Coding Symposium (PCS), to appear* (2007).

[NS07] NOZICK V., SAITO H. : Multiple view computation for multi-stereoscopic display. In *IEEE Pacific-Rim Symposium on image amd video Technology (PSIVT 2007)* (2007), vol. 4872, pp. 399–412.

[Oko77] OKOSHI T. : *Three-dimensional imaging techniques*. Academic Press, 1977.

[Phi06] PHILIPS : Wowvx for amazing viewing experiences. *Philips 3D solutions* (2006).

[Ros06] ROST R. J. : *OpenGL Shading Language*. Addison-Wesley Professional, 2006.

[Sha06] 3d lcds. *http ://sharp-world.com/products/device/about/technology/lcd-03.html* (2006).

[SSAE99] SAITOH G., SUZUKI T., ABE T., EBINA K. : Lenticular sheet, rear-projection screen or tv using the same, and fabrication method for said lenticualr sheet. *U.S. Patent 5870224* (1999).

[Tak06] TAKAKI Y. : High-density directional display for generating natural three-dimensional images. In *Proceedings of the IEEE* (2006), vol. 94, pp. 654–663.

© AFIG 2008.

Figure 6: *Exemple de composition de 4 images avec une repartition en biais des pixels (écran Philips).*

© AFIG 2008.