DENISSALEM

Zeste de savoir

Introduction aux compute shaders

12 août 2019

Table des matières

1. Rappels 1
2. À quoi servent les computer shaders ? 2
3. Aperçu de l'example 3
4. Création de la texture 4
5. Création du compute shader 5
6. Exécution du shader 8
7. Détail du compute shader 9
8. Résultat 11

Avant de se lancer dans ce tutoriel il convient d'être à l'aise avec la notion de shaders et les opérations basiques d'OpenGL. Si vous ne connaissez pas OpenGL, il est sans doute trop tôt pour se lancer dans les *compute shaders*. Il existe une pléthore de très bons tutoriels sur l'OpenGL moderne qui vous permettront de vous familiariser avec l'API. J'en retiens notamment trois :

— http ://www.opengl-tutorial.org/fr/
— https ://openclassrooms.com/courses/developpez-vos-applications-3d-avec-opengl-3-3
— http ://learnopengl.com/ (Ce dernier lien est très complet, mais en anglais seulement)

Pour comprendre cette introduction aux *compute shaders* vous n'avez pas besoin d'aller trop loin dans les tutoriels OpenGL ci-dessus. Le minimum étant de savoir rendre à l'écran un simple triangle, auquel on applique une texture, en OpenGL moderne, avec les vertex/fragment shaders minimaux qu'il convient d'avoir.

Ceci étant dit, nous allons voir ici une fonctionnalité introduite depuis la version 4.3 de notre *API* favorite qui va nous permettre d'étendre nos possibilités.

1. Rappels

OpenGL définit un processus de rendu bien précis dont certaines étapes (*stages*) sont programmables en *GLSL* *(OpenGL Shading Language)*; Ces étapes programmables sont appelées *shaders* et s'exécutent dans un ordre bien précis à l'intérieur de ce qu'on appel la *Render Pipeline*. Un schéma très simplifié de cette *render pipeline* est présenté ci-dessous, à gauche.

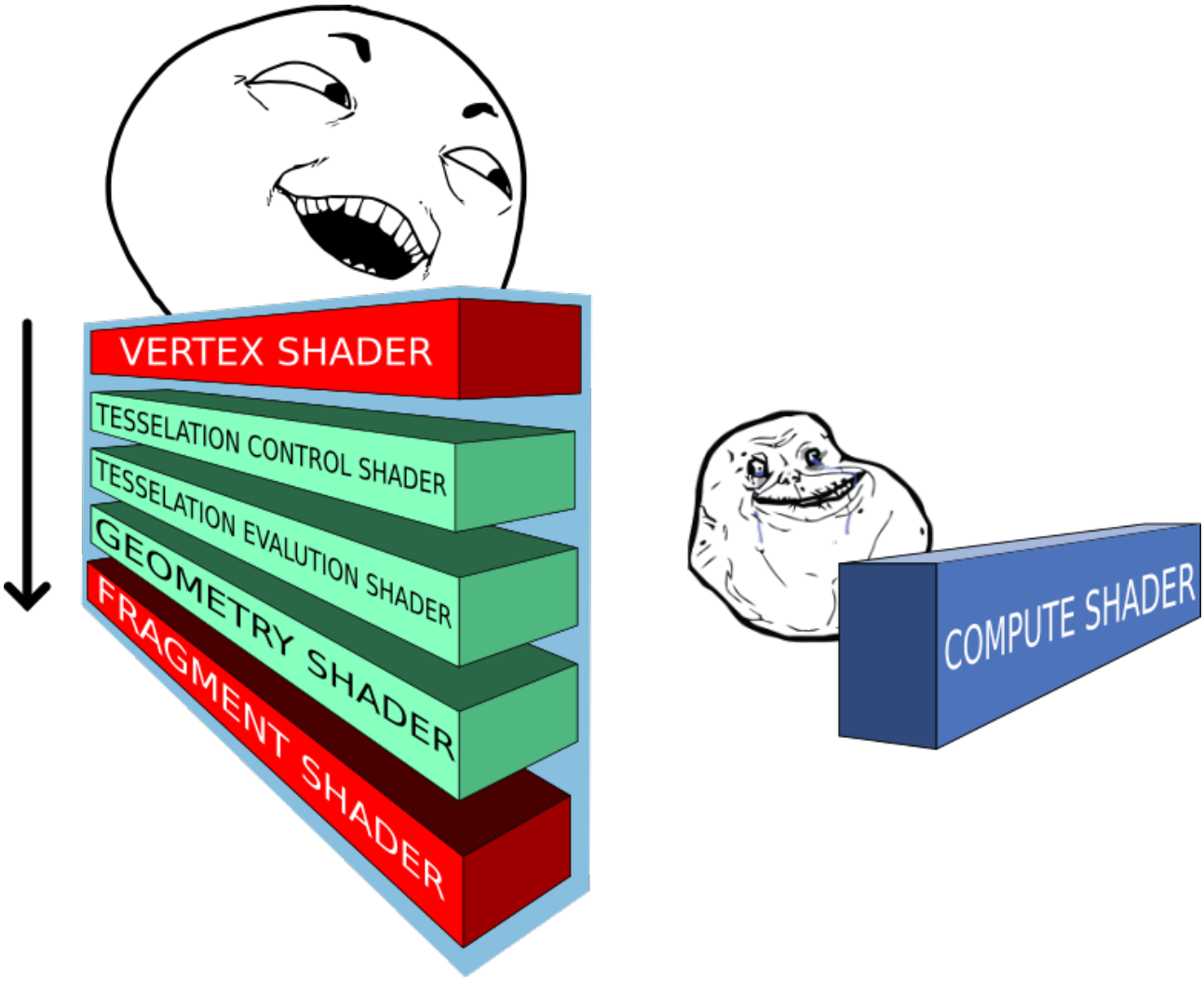

FIGURE 1. – Fixed Shader Pipeline VS Compute Shader

On connaissait déjà les *vertex shaders* qui permettent de manipuler les vertices et les *fragments shaders* qui permettent de réaliser et d'appliquer des effets graphiques à notre image de sortie. Ces deux types de *shaders* sont obligatoires en OpenGL moderne. Les trois autres sont optionnels et nous n'en parlerons pas ici car ce n'est pas du tout le propos de ce tutoriel.

Lorsque l'on manipule les *vertex shaders* ou les *fragment shaders*, on se rend compte que chaque appel est indépendant et ignorent les autres. Ces *shaders*, s'inscrivant dans une *pipeline* de rendu bien définit, leurs entrées/sorties sont elles aussi bien définies, tout comme leur fréquence d'exécution (par exemple un *vertex shader* donné s'exécutera pour chaque vertex qu'on lui donne).

2. À quoi servent les computer shaders ?

Avec les *compute shaders* il est maintenant possible de manipuler arbitrairement des données de la façon dont on le souhaite **sans passer par la pipeline de rendu**. En effet, avant les *compute shaders*, les développeurs utilisaient des *hacks* et des astuces plus ou moins efficaces pour faire des choses qui ressemblent à du *GPU computing*.

Car oui, ce que nous voulons faire avec les *compute shaders*, c'est du *GPU computing* **avec** l'*API* OpenGL.

Le *GPU computing* consiste en fait à faire du calcul dit *parallèle* sur la carte graphique. Certains algorithmes sont plus commodes et plus efficaces quand ils sont effectués sur ce type de support. En effet le CPU est très rapide mais a entre autre l'inconvénient majeur de ne pouvoir faire qu'une seule chose à la fois dans la majorité des situations. La carte graphique, elle, est beaucoup moins rapide mais peut effectuer un nombre astronomique de tâches simultanément. C'est cette aptitude qui confère à la carte graphique toute sa puissance.

Il faut cependant insister sur un point très important. L'architecture de la carte graphique étant conçue pour exécuter parallèlement des instructions, cela impose nécessairement au programmeur de prendre en considération cet état de fait lors de l'implémentation d'un algorithme sur ce type de support.
La programmation *multi-threadé*/parallèle est quelque chose de très différent et de substantiellement plus difficile que l'approche traditionnel ; celle qui consiste donc à penser un programme séquentiellement.
La conception d'un algorithme est très différent de son implémentation. Tous les algorithmes ne sont donc pas adapté à des implémentations sur *GPU* ou de façon générale à des architectures parallèles.

Les *compute shaders* permettent donc maintenant de faire tout ça proprement. Ce gain en flexibilité représente un grand intérêt puisque l'on peut maintenant tirer parti de la puissance de la *GPU* plus librement pour par exemple :

— créer à la volée des textures de synthèses
— générer procéduralement des objets
— effectuer des calculs parallèles qui ne sont pas forcément en rapport avec de l'imagerie 3D sans dépendre des APIs comme CUDA ou OpenCL .

CUDA ou OpenCL servent à ne faire **que** du calcul parallèle. En effet, embrayer dans un même programme d'OpenGL vers l'une des APIs mentionnées à l'instant (ou vis versa) est très chronophage, les *compute shaders* constituent une solution commode à ce problème.

Comme on l'a déjà dit les *compute shaders* sont indépendant de la pipeline de rendu. Lorsque ceux là sont invoqués les entrées/sorties sont définies par le programmeur et non plus par la *pipeline*. C'est également le programmeur qui détermine la charge de travail à effectuer en définissant des groupes de travail (*workgroups*) que nous verrons plus bas.

3. Aperçu de l'example

Pour illustrer le fonctionnement des *compute shaders* on se donne le programme exemple suivant :

damnSimpleComputeShader

Ce code source sert de support à ce tutoriel mais relire le code d'un autre peut être laborieux et/ou peut-être n'utilisez vous pas les mêmes librairies que moi. En réalité, il est très simple d'intégrer le *compute shader* rudimentaire que je présente ici dans un petit programme OpenGL tout aussi rudimentaire. N'ayez donc pas peur d'écrire votre propre programme et d'intégrer au fur et à mesure les éléments manquants qui nous intéressent.

Pour se représenter un peu le tableau de loin, on se propose donc à l'aide d'un *compute shader* de générer le contenu d'une texture vide pour réaliser un damier, et l'afficher.

En fait, c'est assez simple ; dans notre exemple, tout se passe comme si on allait créer une texture vierge et qu'on l'affichait normalement sur un quad (deux triangles formant un carré ou un rectangle) aux dimensions du *viewport* (la surface du rendu).

La différence étant qu'avant d'entrer dans la boucle de rendu, on génère le damier en appelant notre *compute shader* et en s'assurant que cette boucle n'interfère pas avec la génération des données.

En effet, sans explicitement bloquer le fil d'exécution OpenGL va écrire dans la texture et simultanément essayer de l'afficher, parallélisme oblige, ce qui n'est évidemment pas très commode.

4. Création de la texture

Maintenant que l'on a un aperçu de la façon dont se déroule notre programme, allons un peu plus dans le détail. La première chose à faire est de créer la texture qui nous servira de support.

```
1  glGenTextures(1, &quadTextureID);
```

On configure ensuite la texture

```
1  glBindTexture(GL_TEXTURE_2D, quadTextureID);
2    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
         GL_CLAMP_TO_EDGE);
3    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
         GL_CLAMP_TO_EDGE);
4    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
         GL_LINEAR);
5    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
         GL_LINEAR);
6    glGenerateMipmap(GL_TEXTURE_2D);
7    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, 640, 480, 0,
         GL_RGBA, GL_FLOAT, NULL);
8  glBindTexture(GL_TEXTURE_2D, 0);
```

Je ne vais pas m'attarder sur la configuration de la texture, mais à titre de rappel, on peut tout de même rapidement parler de ce qu'il se passe ici :

— *glTexParameteri* sert à configurer la texture courante, c'est à dire celle que l'on a *bindé* avec *glBindTexture*. Ici la configuration choisi est très généraliste. Inutile pour ce turoriel d'aller plus dans le détail.
— *glTexImage2D* permet de définir le type de texture que l'on utilise (*GL_TEXTURE_2D*) ainsi que le format interne des données de la texture côté *GPU* et côté *CPU*. C'est également avec cette commande qu'on envoie les données brutes de la texture à la carte graphique.

Remarquons qu'en général on passe à *glTexImage2D* un pointeur vers un tableau de données pour initialiser la texture avec celui-ci. Comme on alloue la mémoire côté GPU et qu'on génère ces données toujours sur la GPU ce pointeur peut-être NULL.

5. Création du compute shader

Un peu plus bas dans le programme, on arrive au bloc d'instructions où l'on crée notre *compute shader*.

```
1   PrintWorkGroupsCapabilities();
2
3   GLuint computeShaderID;
4   GLuint csProgramID;
5   char * computeShader = 0;
6
7   GLint Result = GL_FALSE;
8   int InfoLogLength = 1024;
9   char ProgramErrorMessage[1024] = {0};
10
11  computeShaderID = glCreateShader(GL_COMPUTE_SHADER);
12
13  loadShader(&computeShader, "compute.shader");
14  compileShader(computeShaderID, computeShader);
15
16  csProgramID = glCreateProgram();
17
18  glAttachShader(csProgramID, computeShaderID);
19  glLinkProgram(csProgramID);
20  glDeleteShader(computeShaderID);
```

Ici, rien de nouveau. On voit que le *compute shader* se construit de la même façon qu'un *vertex/fragment shader*. La seule différence réside au moment de l'invocation de la commande *glCreateShader*. En effet le type de *shader* est maintenant *GL_COMPUTE_SHADER*

La première chose à faire, bien que ce ne soit pas obligatoire ici, c'est de connaître la capacité des groupes de travail offerte par la *GPU*. Ça ne mange pas de pain et ça va être l'occasion de détailler comment les groupes de travail sont organisés.

Les groupes de travail contiennent un certain nombre d'invocations du *compute shader*, définit à l'intérieur du *shader* lui même comme nous le verrons plus bas. Ce nombre est appelé **taille locale du groupe de travail**. On définit également au moment de l'exécution du *compute shader* le nombre de groupes de travail à lancer.

On peut schématiser ce qui vient d'être dit ainsi :

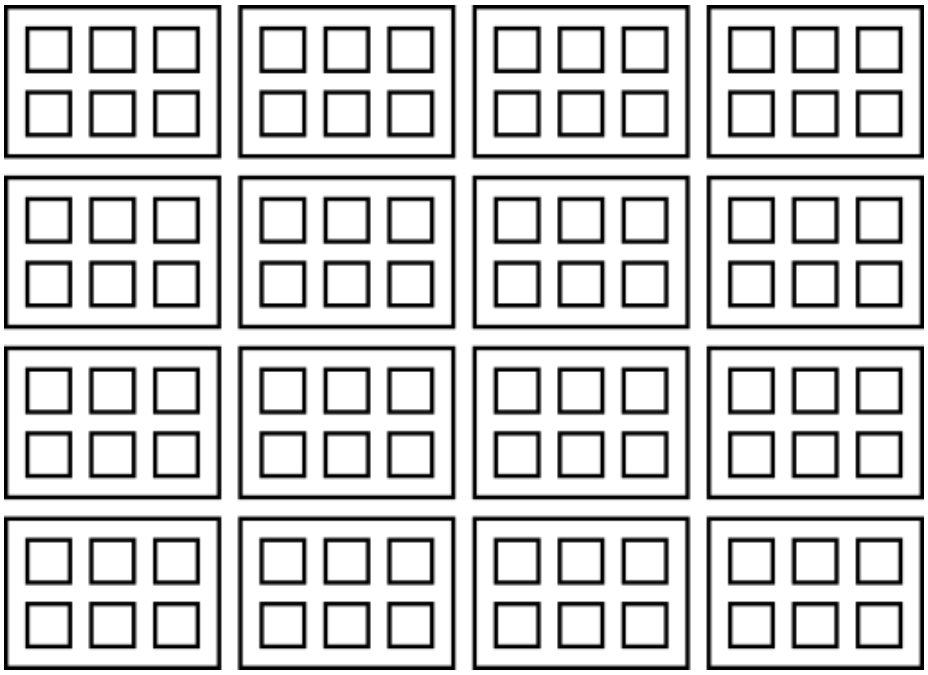

FIGURE 5. – Groupes de travail

Où chaque petit carré représente une invocation de *shader* contenu dans un groupe de travail.

Ceci étant dit on devine que la carte graphique a une capacité limitée de groupes de travail et de nombre d'invocations locales. OpenGL permet de déterminer la capacité de la carte graphique en la matière. En effet dans la fonction *printWorkGroupsCapabilities* on récupère ces informations et on les affiches dans la sortie standard :

```
1  void printWorkGroupsCapabilities() {
2    int workgroup_count[3];
3    int workgroup_size[3];
4    int workgroup_invocations;
5
6    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 0,
         &workgroup_count[0]);
7    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 1,
         &workgroup_count[1]);
8    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 2,
         &workgroup_count[2]);
9
10   printf
         ("Taille maximale des workgroups:\n\tx:%u\n\ty:%u\n\tz:%u\n",
11   workgroup_size[0], workgroup_size[1], workgroup_size[2]);
```

```
12
13    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 0,
          &workgroup_size[0]);
14    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 1,
          &workgroup_size[1]);
15    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 2,
          &workgroup_size[2]);
16
17    printf
          ("Nombre maximal d'invocation locale:\n\tx:%u\n\ty:%u\n\tz:%u\n",
18    workgroup_size[0], workgroup_size[1], workgroup_size[2]);
19
20    glGetIntegerv (GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS,
          &workgroup_invocations);
21    printf ("Nombre maximum d'invocation de workgroups:\n\t%u\n",
          workgroup_invocations);
22  }
```

Ce qu'on peut remarquer, et c'est très important, c'est que la taille locale d'un groupe de travail ainsi que le nombre de groupes de travail est définit en trois dimensions. Autrement dit les *compute shaders* travaillent dans l'espace. C'est très pratique, évidement, pour faire du traitement d'image, ou travailler en volume. Nous verrons plus bas comment organiser notre *shader* en fonction de ces informations.

Précisons également que ces groupes de travail sont lancés en parallèles selon la capacité et la disponibilité de la carte graphique. Il n'est pas possible de connaître l'ordre d'exécution des invocations et dans certains cas il est absolument vital de composer avec cet état de fait sous peine d'obtenir des résultats pour le moins inattendus ou pire, de planter le système.

Heureusement, dans notre cas :

— La question de la synchronisation n'est pas le sujet ici, et le programme que l'on étudie ne nécessite pas que l'on soit attentif à l'ordre d'exécution des groupes de travail et des invocations du *compute shader*.
— Notre exemple n'étant pas non plus gourmand en ressource il ne nécessite pas que l'on implémente une vérification particulière de la capacité des groupes de travail.

La fonction *printWorkGroupsCapabilities* est donc ici purement informative.

Rappelons tout de même que l'on crée un programme spécifiquement POUR le *compute shader*. Un programme qui n'est donc PAS celui où sera lié le *vertex shader* et le *fragment shader* qui servent au rendu. On le voit bien, plus bas dans le code ; la création du programme de rendu est indépendante et ne fait intervenir à aucun moment le *compute shader*.

6. Exécution du shader

Une fois que tout est prêt on peut enfin lancer notre *compute shader*.

```
glUseProgram(csProgramID);
  glBindTexture(GL_TEXTURE_2D, quadTextureID);
    glBindImageTexture (0, quadTextureID, 0, GL_FALSE, 0,
        GL_WRITE_ONLY, GL_RGBA32F);
      glDispatchCompute(40,30,1);
      glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
    glBindImageTexture (0, 0, 0, GL_FALSE, 0, GL_WRITE_ONLY,
        GL_RGBA32F);
  glBindTexture(GL_TEXTURE_2D, 0);
glUseProgram(0);
```

OpenGL étant une machine à état on *bind* la texture qui est associée au shader, exactement comme on l'a fait au moment de la création de la texture. On oublie pas également de *binder* notre *shader program*.

Par ailleurs, pour écrire dans la texture on utilise ce qu'on appelle une *image unit*. Ce qui signifie dans la pratique que l'on va pouvoir travailler dans un tableau avec des indices entiers, et non plus dans un *sampler2D* où les coordonnées sont normalisées. En utilisant la commande *glBindImageTexture* le *shader* peut accéder directement au contenu de la texture.

L'*image unit* associée à la texture est en *WRITE_ONLY*. En effet pour ce que l'on se propose de faire on a pas besoin de lire l'image depuis le *compute shader*, mais seulement d'y écrire.

C'est finalement avec *glDispatchCompute* que l'on lance l'exécution du *compute shader* et plus exactement que l'on lance l'exécution d'un certain nombre de groupes de travail. En effet *glDispatchCompute* prend trois paramètres x, y et z spécifiant chacun les dimensions de l'espace des *workgroups* à lancer.

! Dans notre example, on travaille sur une image, donc dans un espace en deux dimensions, le troisième paramètre est naturellement fixé à 1.

Dans le programme on voit que j'ai choisi pour l'espace des groupes de travail une largeur de 40 et une hauteur de 30 (j'expliquerai le choix de ces valeurs plus bas), ce qui signifie qu'OpenGL va exécuter 40 x 30 = 1200 groupes de travail. Cela peut paraître beaucoup mais c'est en fait relativement peu.

? Après l'appel de *glDispatchCompute*, OpenGL se met au travail et le fil d'exécution continue côté CPU ?

Non ! Programmeur vigilant que l'on est, on s'assure que le fil d'exécution de notre programme s'interrompe le temps qu'OpenGL termine le traitement de tous les groupes de travail à l'aide de la commande *glMemoryBarrier* qui permet comme on le devine de bloquer l'exécution des

commandes OpenGL suivantes tant que la carte graphique n'a pas terminé ses transactions mémoires en cours.

Bloquer l'exécution des commandes OpenGL suivantes ?

Oui, en fait, le CPU et la GPU travaillent tous les deux de façon asynchrones la plupart du temps. En réalité, lorsque l'on appelle une commande OpenGL, celle-ci est mise en attente et est réellement exécutée côté *GPU* quand arrive son tour. Ce qui signifie que du côté du *CPU*, le programme peut continuer d'avancer tout seul. *glMemorieBarrier* s'assure donc que lorsqu'une commande OpenGL est lancé, toutes les transactions mémoires définies sont bien terminé.

En particulier, cela permet de bloquer l'exécution en fonction du type de mémoire que l'on veut. Ici, comme nous travaillons sur une image on utilise la valeur prédéfinie *GL_SHADER_IMAGE_ACCESS_BARRIER_BIT*.

On pourrait également, si cela était nécessaire, combiner plusieurs valeurs prédéfinies par OpenGL pour bloquer l'exécution du programme en fonction de plusieurs zones mémoire. Si *GL_ALL_BARRIER_BITS* est utilisé alors le fil d'exécution des tâches à venir côté *GPU* se met en pause en fonction de toutes les zones mémoire précédemment sollicitées. Pour bloquer réellement le fil d'exécution côté CPU, il faudrait utiliser la commande *glFinish()*. On a ainsi la garantie absolue qu'OpenGL à bien finit tout son travail après cette commande.
C'est bien pratiquer pour débuguer...

Ceci étant fait, les *workgroups* étant tous exécutés, on peut reprendre le fil d'exécution du programme et *unbind* nos états OpenGL.

Après quoi, on rentre tout simplement dans la boucle de rendu qui ne fait absolument rien de nouveau.

Doit on réappeler la commande *glBindImageTexture* dans la boucle de rendu ?

Bonne question. Non, ce n'est pas nécessaire. En effet, le *fragment shader* n'a besoin à aucun moment d'une *image unit* puisque ce qu'on fait c'est *sampler* la texture. On n'y accède donc pas directement.

7. Détail du compute shader

C'est donc le moment de voir à quoi ressemble le compute shader que l'on a exécuté.

```
1 #version 430
2
3 layout (local_size_x = 16, local_size_y = 16) in;
```

```
4
5  layout (rgba32f, binding = 0) uniform image2D img_output;
6
7  void main() {
8    // Aucun tableau de donnée n'étant passé au moment de la création
         de la texture,
9    // c'est le compute shader qui va dessiner à l'intérieur de
         l'image associé
10   // à la texture.
11
12   // gl_LocalInvocationID.xy * gl_WorkGroupID.xy ==
         gl_GlobalInvocationID
13   ivec2 coords = ivec2(gl_GlobalInvocationID);
14
15   // Pour mettre en evidence. Les groupes de travail locaux on
         dessine un damier.
16   vec4 pixel;
17   if ( ((gl_WorkGroupID.x & 1u) != 1u) != ((gl_WorkGroupID.y & 1u)
         == 1u)) {
18     pixel = vec4(1.0,.5,.0,1.0);
19   }
20   else {
21     pixel = vec4(.0,.5,1.0,1.0);
22   }
23
24   imageStore(img_output, coords, pixel);
25 }
```

Un *compute shader* se présente en fait de façon très similaire aux *shaders* classiques. La nouveauté c'est que l'on définit à l'intérieur du *shader* lui même la taille des groupes de travail. Comme on le voit, la taille locale d'un groupe de travail est de 16 par 16. Rappelons que les dimensions de l'espace de travail des *workgroups* étaient de 40 par 30. On aura donc 16^2 x 1200 invocations du compute shader. On peut également remarquer que 16 $40 = 640$ *et* 16 $30 = 480$. Et ça tombe bien parce que c'est justement les dimensions de notre image et de notre *viewport*.

Quelques explications s'imposent. Nous pourrions effectivement générer un damier dans une image de façon itérative dans un seul *thread*. Mais ça ne serait pas vraiment optimal si on tient compte du fait que l'on peut faire la même chose en tirant parti de l'aptitude du *GPU* à faire du calcul parallèle.

Comme on l'a dit plus haut, la raison pour laquelle le nombre de groupes de travail et la taille de ceux là sont formulés en terme d'espace est que les *compute shaders* travaillent dans l'espace. Pour se repérer dans l'espace de la structure de donnée dans laquelle on travaille on utilise des variables introduites avec les *compute shaders* qui nous renseignent sur l'identifiant de l'invocation courante du *compute shader*. Cet identifiant étant tridimensionnel, on sait donc où l'on se trouve au moment d'une invocation donnée. Comme les invocations sont parallèles (mais également dans un ordre arbitraire décidé par OpenGL) on peut donc ici travailler simultanément à différents endroits de notre image, ce qui fait substantiellement gagner du temps.

Du coup, la seule façon de savoir où l'on se trouve est d'interroger ces variables spéciales qui

identifient l'invocation courante.

> Ces variables sont des *vec3*. À titre de rappel, OpenGL travail le plus souvent avec des vecteurs de deux, trois ou quatre composantes ; respectivement *vec2*, *vec3* et *vec4*. Les vecteurs peuvent contenir des entiers signés ou non signés ou, comme c'est le cas la plupart du temps, des nombres à virgules flottantes. Pour en savoir plus rendez vous sur le wiki d'OpenGL.

- *gl_LocalInvocationID* est un *vec3* qui nous dit où l'on se trouve relativement au groupe de travail courant. Les coordonnées xyz ainsi retournées ne peuvent donc pas être en dehors du volume décrit par la taille du groupe locale, à savoir ici : 16 par 16 par 1.
- *gl_WorkGroupID* est un *vec3* qui nous renseigne sur le groupe de travail courant, à ne pas confondre avec l'invocation courante donc. Les coordonnées xyz ainsi retournées ne peuvent pas être en dehors du volume décrit par la taille de l'espace des groupes de travail, à savoir ici : 40 par 30 par 1.
- *gl_GlobalInvocationID* est un *vec3* qui est en fait le produit des deux variables vues précédemment. Il repère l'invocation courante non plus relativement au groupe de travail courant mais dans l'espace global des invocations. Autrement dit, dans notre programme, cette variable nous dit où l'on se trouve dans l'image.

Finalement pour dessiner notre damier, on détermine la couleur de sortie en fonction de *gl_WorkGroupID* ; si sa composante x est paire et que sa composante y ne l'est pas alors la couleur de sortie est orange, sinon, elle est bleue. Toutes les invocations à l'intérieurs d'un groupe de travail seront donc soit oranges, soit bleues.

Pour terminer, on utilise la fonction GLSL *imageStore* pour écrire dans notre image. Remarquez qu'en fait on n'enregistre qu'un seul et unique pixel pour une invocation donnée du *compute shader*.

8. Résultat

En compilant et en exécutant le programme exemple (en vous assurant préalablement de satisfaire les dépendances) on obtient l'image ci-dessous.

FIGURE 8. – Damier de 40x30 secteurs de 16 pixels^2

Comme on pouvait s'y attendre notre damier comporte des secteurs de 16 pixels^2 au nombre de 40 en largeur et 30 en hauteur. On illustre ainsi sans ambiguïté l'utilisation d'un compute shader et des groupes de travail.

Pour terminer ce cours introductif vous pouvez jouer avec le *compute shader* présenté ici et modifier la taille du groupe de travail locale et adapter l'espace des groupes en conséquence pour changer la taille des cases du damier ou pour rendre à l'écran des formes géométriques et les positionner comme bon vous semble.

Pour terminer voici les ressources qui m'ont permis de réaliser ce tutoriel :

- http ://antongerdelan.net/opengl/compute.html
- https ://www.opengl.org/wiki/Compute_Shader
- http ://malideveloper.arm.com/resources/sample-code/introduction-compute-shaders-2/
- https ://www.panda3d.org/manual/index.php/Compute_Shaders

Ce tutoriel a été rédigé dans le cadre d'un stage à l'INRIA, un grand merci à Sylvain Lefebvre et son équipe pour leur aide et leur disponibilité.

Un très grand merci également à Ge0 et Shenzyn, ainsi qu'Arius et Anto59290 pour leurs conseils et leurs relectures.

Merci enfin à Glordim pour la relecture finale !