

# Apprendre OpenGL moderne

## Quatrième partie : OpenGL avancé

Par Joey de Vries - Alexandre Laurent (traducteur) 

Date de publication : 24 janvier 2019

TOUT PUBLIC

Developpez.com a la joie d'accueillir une traduction en français du célèbre cours de grande qualité [Learn OpenGL](#). Avec ce cours, vous apprendrez à programmer des applications graphiques 3D grâce à la bibliothèque OpenGL.

Ces tutoriels sont accessibles aux débutants, mais les connaisseurs ne s'ennuieront pas non plus grâce aux chapitres avancés.

Cette page vous amène à la quatrième partie du tutoriel, c'est-à-dire : les techniques avancées à intégrer à l'aide d'OpenGL. Plus précisément, cela sera l'occasion de découvrir comment utiliser les tampons de profondeur et de pochoir, les *geometry shader*, les textures cubiques et les tampons d'image.

Vous pouvez retrouver les autres parties ci-dessous :

- [introduction](#) ;
- [éclairage](#) ;
- [chargement de modèle](#) ;
- [OpenGL avancé](#) ;
- éclairage avancé ;
- PBR ;
- [mise en pratique](#).

**Commentez**

|  |    |
|--|----|
| I - Test de profondeur (depth testing).....            | 4  |
| I-A - Fonction de test.....                            | 5  |
| I-B - Précision du test de profondeur.....             | 6  |
| I-C - Visualisation du tampon de profondeur.....       | 8  |
| I-D - Conflit de profondeur.....                       | 10 |
| I-D-1 - Éviter le conflit de profondeur.....           | 11 |
| I-E - Remerciements.....                               | 11 |
| II - Test de pochoir (stencil test).....               | 12 |
| II-A - Fonctions de test du pochoir.....               | 13 |
| II-B - Contour d'un objet.....                         | 14 |
| II-C - Remerciements.....                              | 17 |
| III - Mélange des couleurs (blending).....             | 18 |
| III-A - Rejeter des fragments.....                     | 19 |
| III-B - Mélange de couleurs.....                       | 22 |
| III-C - Affichage de textures semi-transparentes.....  | 27 |
| III-C-1 - Ne pas casser l'ordre.....                   | 28 |
| III-D - Remerciements.....                             | 30 |
| IV - Suppression de faces (culling).....               | 30 |
| IV-A - Ordonnancement des sommets.....                 | 30 |
| IV-B - Élimination des faces.....                      | 31 |
| IV-C - Exercices.....                                  | 33 |
| IV-D - Remerciements.....                              | 33 |
| V - Tampon d'image (frame buffers).....                | 33 |
| V-A - Créer un tampon d'image.....                     | 34 |
| V-A-1 - Attaché de type texture.....                   | 35 |
| V-A-2 - Attaché de type tampon de rendu.....           | 36 |
| V-B - Rendu dans une texture.....                      | 37 |
| V-C - Post-traitement.....                             | 39 |
| V-C-1 - Inversion.....                                 | 40 |
| V-C-2 - Niveau de gris.....                            | 40 |
| V-C-3 - Effet de noyau.....                            | 41 |
| V-C-3-a - Flou.....                                    | 43 |
| V-C-3-b - Détection des bordures.....                  | 43 |
| V-D - Remerciements.....                               | 44 |
| VI - Texture cubique (cubemap).....                    | 44 |
| VI-A - Créer une texture cubique.....                  | 45 |
| VI-B - Skybox.....                                     | 47 |
| VI-B-1 - Chargement d'une skybox.....                  | 48 |
| VI-B-2 - Afficher une skybox.....                      | 49 |
| VI-B-3 - Optimisation.....                             | 50 |
| VI-C - Application de l'environnement.....             | 51 |
| VI-C-1 - Réflexion.....                                | 51 |
| VI-C-2 - Réfraction.....                               | 54 |
| VI-C-3 - Application de l'environnement dynamique..... | 56 |
| VI-D - Exercices.....                                  | 56 |
| VI-E - Remerciements.....                              | 56 |
| VII - Données avancées.....                            | 56 |
| VII-A - Mise en lot des attributs de sommets.....      | 57 |
| VII-B - Copie des tampons.....                         | 58 |
| VII-C - Remerciements.....                             | 59 |
| VIII - GLSL Avancé.....                                | 59 |
| VIII-A - Variables GLSL.....                           | 59 |
| VIII-A-1 - Variables du vertex shader.....             | 59 |
| VIII-A-1-a - gl_PointSize.....                         | 59 |
| VIII-A-1-b - gl_VertexID.....                          | 60 |
| VIII-A-2 - Variables du fragment shader.....           | 60 |
| VIII-A-2-a - gl_FragCoord.....                         | 60 |
| VIII-A-2-b - gl_FrontFacing.....                       | 61 |

|  |     |
|--|-----|
| VIII-A-2-c - gl_FragDepth.....   | 62  |
| VIII-B - Blocs d'interface.....  | 63  |
| VIII-C - Tampon de variables uniformes.....                              | 64  |
| VIII-C-1 - Disposition du bloc de variables uniformes.....               | 65  |
| VIII-C-2 - Utiliser un tampon de variables uniformes.....                | 67  |
| VIII-C-3 - Un exemple simple.....  | 68  |
| VIII-D - Remerciements.....  | 70  |
| IX - Geometry shader.....  | 71  |
| IX-A - Utiliser les geometry shaders.....                                | 73  |
| IX-B - Construisons des maisons.....                                     | 75  |
| IX-C - Explorer les objets.....  | 80  |
| IX-D - Visualiser les normales.....                                      | 82  |
| IX-E - Remerciements.....  | 84  |
| X - Instanciation.....   | 84  |
| X-A - Tableaux instanciés.....   | 87  |
| X-B - Affichage d'un champ d'astéroïdes.....                             | 89  |
| X-C - Remerciements.....   | 93  |
| XI - Anticrénelage.....  | 93  |
| XI-A - Multiéchantillonnage.....   | 95  |
| XI-B - MSAA dans OpenGL.....   | 99  |
| XI-C - MSAA hors écran.....  | 100 |
| XI-C-1 - Attaché d'une texture suréchantillonnée.....                    | 100 |
| XI-C-2 - Tampon de rendu suréchantillonné.....                           | 100 |
| XI-C-3 - Rendu dans un tampon d'image suréchantillonné.....              | 100 |
| XI-D - Algorithme personnalisé de suppression de l'effet d'escalier..... | 102 |
| XI-E - Remerciements.....  | 103 |

## I - Test de profondeur (depth testing)

Dans le tutoriel sur les **systèmes de coordonnées**, nous avons affiché un conteneur 3D et nous avons utilisé le tampon de profondeur (*depth buffer*) pour éviter que les faces arrière ne s'affichent devant les autres. Dans ce tutoriel, nous allons approfondir cette notion de profondeur. Les valeurs de profondeur sont stockées dans un tampon spécifique, dit tampon de profondeur (ou *z-buffer*). Nous verrons aussi comment ce tampon est utilisé pour déterminer si un fragment est derrière les autres.

Le tampon de profondeur est un tampon qui, tout comme le tampon de couleurs (qui stocke les couleurs de tous les fragments : le résultat visuel), stocke une information par fragment et a (généralement) la même largeur et hauteur que le tampon de couleur. Le tampon de profondeur est automatiquement créé par le système de fenêtrage et stocke les valeurs de profondeur à l'aide de nombres à virgule flottante sur 16, 24 ou 32 bits. La plupart des systèmes l'utilisent avec une précision sur 24 bits.

Lorsque le test de profondeur est activé, OpenGL teste la valeur de la profondeur du fragment avec le contenu du tampon de profondeur. OpenGL effectue un test de profondeur et, si le test réussit, le tampon de profondeur est mis à jour avec la nouvelle valeur de profondeur. Si le test échoue, le fragment est abandonné.

Le test de profondeur est effectué dans l'espace écran après l'exécution du *fragment shader* (et après le test de pochoir (*stencil*) que nous allons voir dans le [tutoriel suivant](#)). Les coordonnées de l'espace écran correspondent directement à la zone de rendu définie avec la fonction `glViewport()` d'OpenGL et est accessible en GLSL dans le *fragment shader* avec la variable `gl_FragCoord`. Les composants `x` et `y` de `gl_FragCoord` représentent les coordonnées du fragment dans l'espace écran (avec `(0, 0)` en bas à gauche). La variable `gl_FragCoord` contient aussi un composant `z` qui détermine la valeur de profondeur du fragment. Ce `z` est la valeur utiliser dans la comparaison avec le contenu du tampon de profondeur.

Aujourd'hui, la plupart des GPU supportent une fonctionnalité appelée *test de profondeur anticipé* (*early depth testing*). Cette technique permet d'exécuter le test de profondeur avant l'exécution du *fragment shader*. Chaque fois qu'il est évident qu'un fragment ne sera pas visible (il est derrière d'autres objets), nous pouvons ignorer le fragment plus tôt.

 Il est préférable de ne pas exécuter un *fragment shader* autant que faire se peut. L'optimisation n'est applicable que si vous ne définissez pas la valeur de la profondeur dans le *fragment shader*. Dans le cas contraire, il n'est pas possible de deviner la valeur pour effectuer le test de profondeur.

Le test de profondeur est désactivé par défaut et vous devez l'activer en utilisant l'option `GL_DEPTH_TEST` :

```
glEnable(GL_DEPTH_TEST);
```

Une fois activé, OpenGL stocke automatiquement les valeurs de la composante `z` dans le tampon de profondeur suivant le résultat du test. Si le test échoue, le fragment est abandonné. Si vous avez activé le test de profondeur, vous devez nettoyer le tampon avant chaque rendu à l'aide de `GL_DEPTH_BUFFER_BIT`, sinon vous garderez les valeurs du rendu précédent :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Il existe certains scénarios où vous souhaitez effectuer le test de profondeur sur tous les fragments, mais sans mettre à jour le tampon de profondeur. En soi, vous avez un tampon de profondeur en lecture seule. OpenGL nous permet d'activer ou non l'écriture dans le tampon de profondeur en définissons le masque à `GL_FALSE` :

```
glDepthMask(GL_FALSE);
```

Évidemment, cela n'a d'effet que si le test de profondeur est actif.

## I-A - Fonction de test

OpenGL permet de modifier l'opérateur de comparaison utilisé dans le test de profondeur. Cela permet de contrôler dans quel cas le test de profondeur réussit et dans quel cas les fragments sont abandonnés et quand mettre à jour le tampon. Nous pouvons définir l'opérateur de comparaison (ou fonction de profondeur) en appelant `glDepthFunc()` :

```
glDepthFunc(GL_LESS);
```

Voici la liste des opérateurs de comparaison possibles :

| Fonction    | Description   |
|-------------|---|
| GL_ALWAYS   | Le test de profondeur réussit toujours  |
| GL_NEVER    | Le test de profondeur ne réussit jamais   |
| GL_LESS     | Le test réussit si la valeur du fragment est inférieure à la valeur dans le tampon          |
| GL_EQUAL    | Le test réussit si la valeur du fragment est égale à la valeur dans le tampon               |
| GL_LEQUAL   | Le test réussit si la valeur du fragment est inférieure ou égale à la valeur dans le tampon |
| GL_GREATER  | Le test réussit si la valeur du fragment est supérieure à la valeur dans le tampon          |
| GL_NOTEQUAL | Le test réussit si la valeur du fragment est différente de la valeur dans le tampon         |
| GL_GEQUAL   | Le test réussit si la valeur du fragment est supérieure ou égale à la valeur dans le tampon |

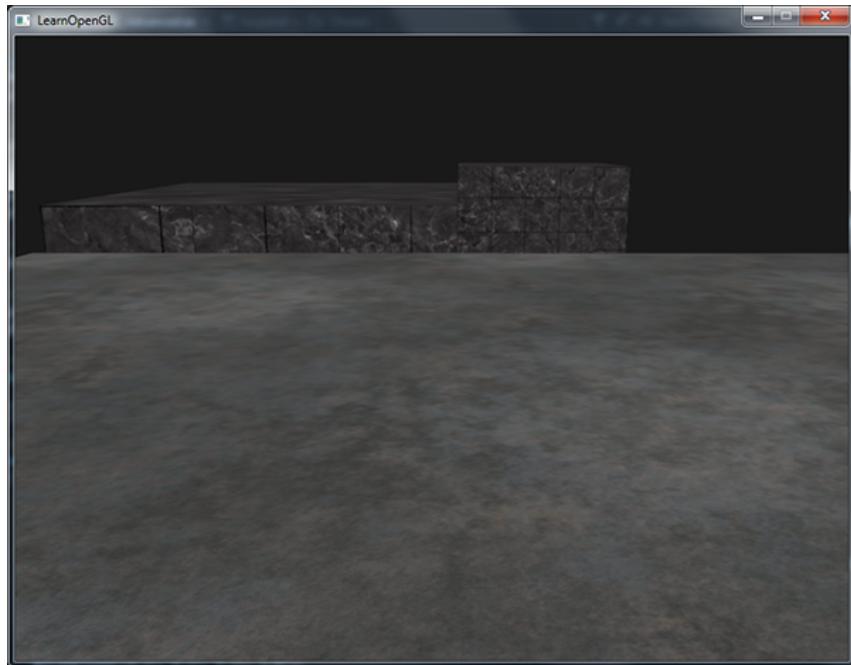
La fonction par défaut est `GL_LESS`. Cela signifie que tous les fragments dont la valeur de profondeur est supérieure ou égale à la valeur dans le tampon sont abandonnés.

Voyons l'impact de la fonction de profondeur. Nous allons utiliser une nouvelle configuration qui affiche une scène simple avec deux cubes texturés posés sur un sol texturé sans éclairage. Vous pouvez trouver le code source [Source ici](#).

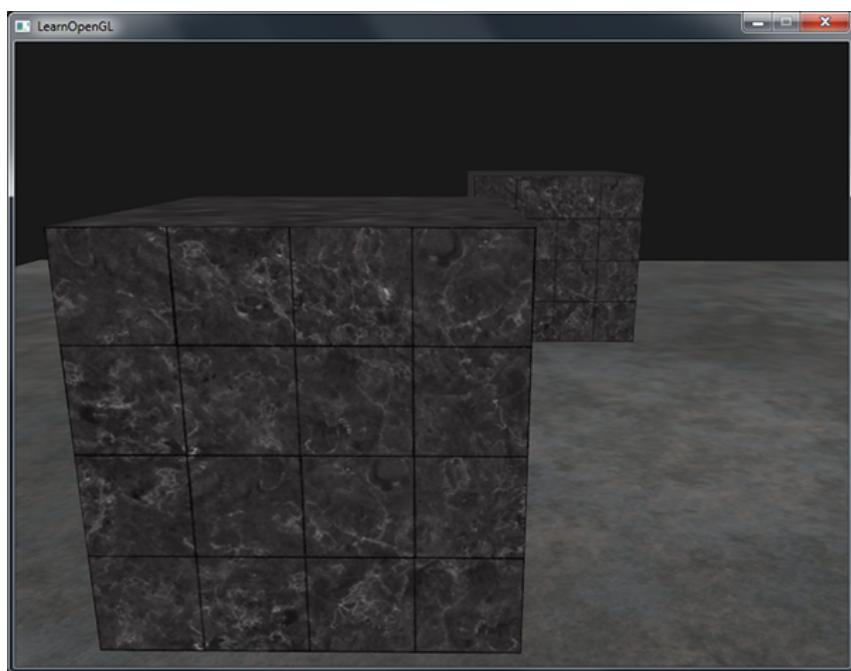
Dans le code, nous changeons la fonction de profondeur à `GL_ALWAYS` :

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_ALWAYS);
```

Cela revient à ne pas avoir activé le test de profondeur. Le test réussit toujours, donc les fragments qui sont affichés en dernier seront devant les fragments affichés avant, même s'ils auraient dû être derrière. Comme nous avons dessiné le sol à la fin, ses fragments recouvrent ceux des cubes :



En remettant `GL_LESS`, cela donne le type de scène auquel nous étions habitués :

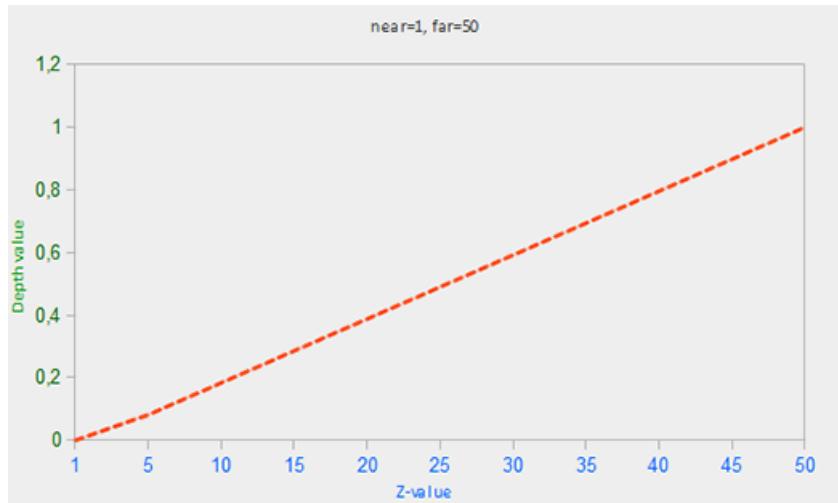


## I-B - Précision du test de profondeur

Le tampon de profondeur contient des valeurs entre 0.0 et 1.0. Ces valeurs sont comparées avec la composante z des objets de la scène comme vu par le spectateur. Ces valeurs z dans l'espace vue peuvent prendre n'importe quelle valeur entre le plan proche (*near*) et le plan lointain (*far*) déterminé par la projection. Nous devons donc trouver une méthode pour transformer ces valeurs provenant de l'espace vue vers l'échelle [0, 1]. Une méthode est de les transformer linéairement. L'équation (linéaire) suivante transforme la valeur z pour obtenir une valeur entre 0.0 et 1.0 :

$$F_{depth} = \frac{z - near}{far - near}$$

Ici, `near` et `far` sont les valeurs que nous avons utilisées pour créer la matrice de projection (voir le chapitre sur les **systèmes de coordonnées**). L'équation prend une valeur `z` présente dans le champ de vision et la transpose dans l'échelle [0, 1]. La relation entre la valeur `z` et la profondeur correspondante est présentée sur ce graphique suivant :



**i** Toutes les équations donnent une valeur proche de 0.0 lorsque l'objet est proche et une valeur de profondeur proche de 1.0 lorsque l'objet s'approche du plan lointain.

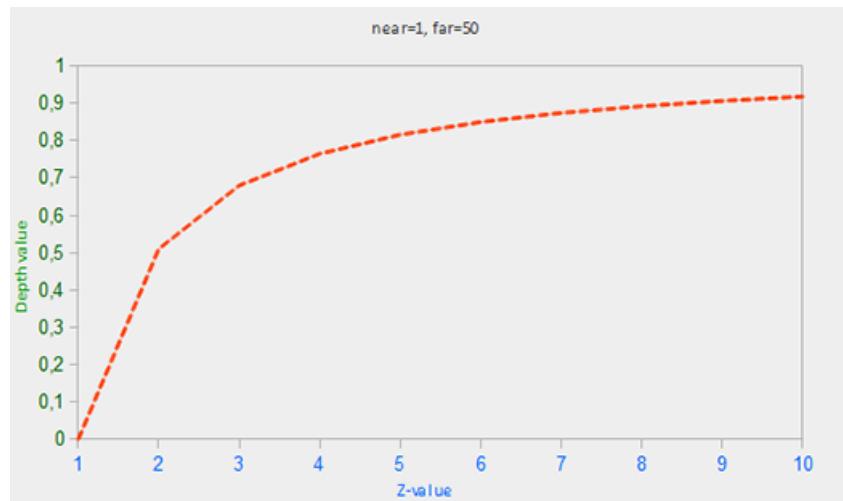
Toutefois, un tampon de profondeur linéaire n'est pratiquement jamais utilisé. Pour obtenir des propriétés de projection

correctes, une équation non linéaire proportionnelle à  $\frac{1}{z}$  est utilisée . Cela permet d'obtenir une très grande précision lorsque le `z` est petit et une plus faible précision lorsque l'objet est loin. Pensez-y une seconde : avons-nous vraiment besoin de la même précision pour un objet éloigné de 1000 unités et un objet proche ? L'équation linéaire ne fait pas ce genre de considérations.

Comme la fonction non linéaire est proportionnelle à  $\frac{1}{z}$ , les valeurs entre 1.0 et 2.0 donneront des valeurs entre 1,0 et 0,5, soit la moitié de la précision disponible. De même, des valeurs de `z` entre 50,0 et 100,0 ne prendront que 2 % de la précision disponible. Voici l'équation permettant de prendre en compte les distances `near` et `far` :

$$F_{depth} = \frac{\frac{1}{z} - \frac{1}{near}}{\frac{1}{far} - \frac{1}{near}}$$

Ne vous inquiétez pas si vous ne comprenez pas ce que fait l'équation. La chose à retenir est que les valeurs dans le tampon de profondeur ne sont pas linéaires dans l'espace écran (elles sont linéaires dans l'espace vue, avant que la matrice de projection ne soit appliquée). Une valeur de 0,5 dans le tampon de profondeur ne veut pas dire que la valeur `z` de l'objet est à la moitié du champ de vision. La valeur `z` du sommet est en réalité assez proche du spectateur ! Vous pouvez voir la relation entre les valeurs `z` et leur correspondance dans le graphique suivant :



Comme vous pouvez le voir, les valeurs de profondeur sont principalement déterminées par les petites valeurs de z, donnant ainsi une grande précision pour les objets proches. L'équation pour transformer les valeurs z (du point de vue du joueur) est embarquée dans la matrice de projection. Par conséquent, lors de la transformation des coordonnées de sommets de l'espace vue à l'espace de découpage, puis à l'espace écran, l'équation non linéaire est appliquée. Si vous êtes curieux de comprendre ce que fait réellement la matrice de projection, je vous suggère de lire [l'article de Song Ho Ahn](#).

L'effet de l'équation non linéaire devient très visible lorsque nous essayons d'afficher le tampon de profondeur.

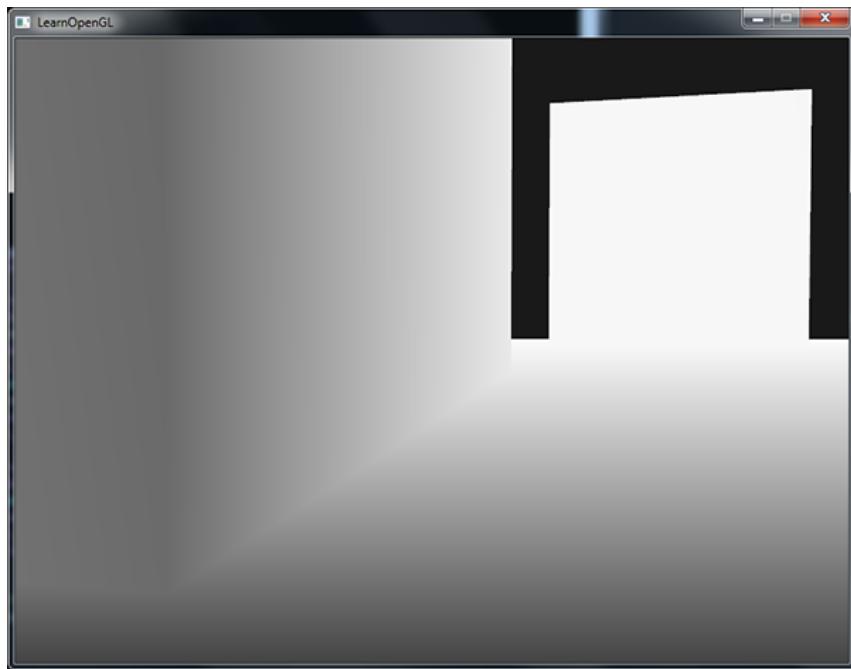
### I-C - Visualisation du tampon de profondeur

Nous savons que la valeur z de la variable du *fragment shader gl\_FragCoord* contient la valeur de profondeur pour un fragment en particulier. Si nous utilisons la valeur de profondeur du fragment comme une couleur, nous pouvons afficher les valeurs du tampon de profondeur de l'intégralité de la scène. Nous pouvons le faire en retournant une nuance de gris basée sur la valeur de profondeur :

```
void main()
{
    FragColor = vec4(vec3(gl_FragCoord.z), 1.0);
}
```

Si vous exécutez le même programme, vous allez sûrement voir que tout est blanc, comme si toutes les valeurs de profondeur sont à 1.0, soit la valeur de profondeur maximale. Pourquoi ?

Vous devez garder en tête ce que nous avons vu dans la précédente section : les valeurs de profondeur en espace écran ne sont pas linéaires, c'est-à-dire, que la précision est plus importante pour les petites valeurs de z. La valeur de profondeur augmente rapidement suivant la distance, tous les sommets sont donc proches d'une profondeur de 1,0. Si nous nous approchions des objets, nous pourrions éventuellement obtenir des couleurs plus sombres et donc des valeurs de z plus faibles :



Cela met en évidence la non-linéarité des valeurs de profondeur. Les objets proches ont un impact plus important sur les valeurs que les objets au loin. En se déplaçant seulement de quelques millimètres, les couleurs passent du noir au blanc.

Toutefois, nous pouvons transformer les valeurs non linéaires en leur correspondance linéaire. Pour ce faire, nous devons refaire à l'envers le procédé de la projection uniquement pour les valeurs de profondeur. Cela signifie que nous devons repasser les valeurs allant de 0 à 1 en des valeurs allant de -1 à 1 (espace de découpage). Ensuite, nous devons enlever l'aspect non linéaire (deuxième équation) provenant de la matrice de projection et appliquer l'inverse de l'équation à la valeur de profondeur finale. Ainsi, nous obtenons des valeurs de profondeur linéaire. Cela semble-t-il faisable ?

Premièrement, nous transformons les valeurs de profondeur en coordonnées normalisées :

```
float z = depth * 2.0 - 1.0;
```

Ensuite, nous prenons le résultat  $z$  et appliquons la transformation inverse pour obtenir une valeur de profondeur linéaire :

```
float linearDepth = (2.0 * near * far) / (far + near - z * (far - near));
```

Cette équation est déduite à partir de la matrice de projection qui utilise l'équation non linéaire permettant d'obtenir les valeurs de profondeur entre *near* et *far*. **Cet article mathématiquement imposant** explique en détails la matrice de projection. Il explique aussi d'où vient l'équation.

Le *fragment shader* complet qui transforme la profondeur non linéaire de l'espace écran en profondeur linéaire correspond à :

```
#version 330 core
out vec4 FragColor;

float near = 0.1;
float far = 100.0;

float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // back to NDC
    return (2.0 * near * far) / (far + near - z * (far - near));
```

```

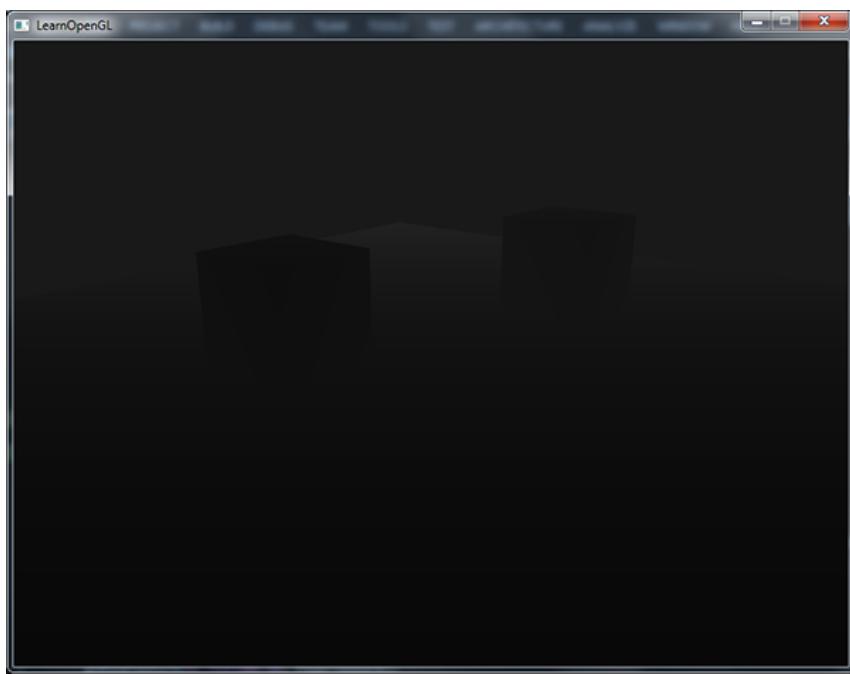
}

void main()
{
    float depth = LinearizeDepth(gl_FragCoord.z) / far; // division par far pour la démonstration
    FragColor = vec4(vec3(depth), 1.0);
}

```

Comme les valeurs entre `near` et `far` sont principalement supérieures à 1,0, elles sont affichées complètement en blanc. En divisant la profondeur linéaire par `far` dans la fonction `main()`, nous convertissons la profondeur pour obtenir des valeurs entre 0 et 1. Ainsi, nous pouvons voir les objets proches devenir progressivement plus clairs que les fragments proches du plan lointain.

Si nous exécutons l'application, nous obtenons des valeurs de profondeur linéaire sur la distance. Essayez d'explorer la scène pour voir les valeurs changer.



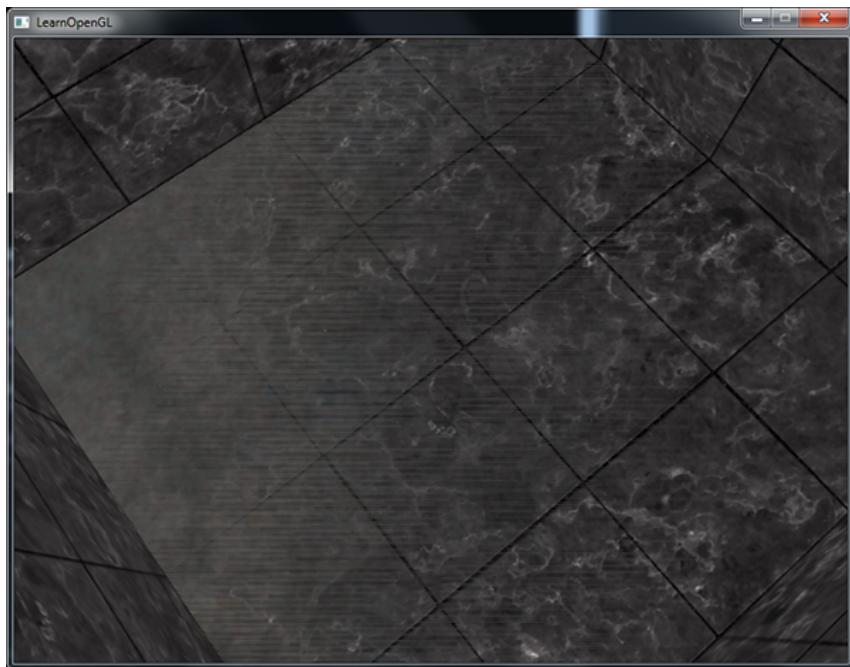
La scène est principalement noire, car le plan proche est à 0,1 et le plan lointain à 100, ce qui est très éloigné. Le résultat est que les objets sont plutôt proches du plan proche et donc les valeurs de profondeur sont faibles (plus sombres).

## I-D - Conflit de profondeur

Un problème visuel classique peut arriver lorsque deux plans ou triangles sont superposés et que le tampon de profondeur n'a pas assez de précision pour déterminer quel est celui devant l'autre. Les deux formes vont continuellement alterner et donner des artefacts visuels. Cela s'appelle un conflit de profondeur (*z-fighting*), car les formes semblent se battre pour être par-dessus l'autre.

Dans la scène de ce tutoriel, il y a quelques endroits où le conflit de profondeur peut arriver. Les conteneurs sont exactement placés à la même hauteur que le sol et donc leur face du dessous est coplanaire avec le plan du sol. Les valeurs de profondeur des deux plans sont les mêmes, ce qui fait que le test de profondeur n'a pas de méthode pour déterminer lequel est le bon.

Si vous déplacez la caméra dans l'un des conteneurs, vous allez vite en rendre compte : la face du dessous alterne constamment entre la face du conteneur et le sol :



Le conflit de profondeur est un problème courant avec les tampons de profondeur et il est généralement plus fort lorsque les objets sont lointains (car le tampon de profondeur a moins de précision pour ceux-ci). Le conflit de profondeur ne peut pas être complètement supprimé, mais il y a quelques astuces pour l'éviter.

### I-D-1 - Éviter le conflit de profondeur

La première et la plus importante astuce est de *ne jamais placer des objets trop proches des autres pour éviter le recouvrement des triangles*. En ajoutant un petit décalage entre deux objets, difficilement visible pour l'utilisateur, vous allez éviter complètement un conflit entre les deux objets. Dans le cas de cette scène, nous pouvions placer les conteneurs légèrement un peu plus haut. La différence est tellement faible qu'elle est difficilement constatable, mais elle évite les conflits de profondeur. Toutefois, cela nécessite une intervention manuelle sur chaque objet et des tests pour s'assurer que tous les cas sont évités.

Une deuxième astuce consiste à définir le plan proche aussi loin que possible. Dans l'une des précédentes sections, nous avons discuté de la précision qui est extrêmement grande pour les objets à proximité du plan proche. Si nous déplaçons ce plan un peu plus loin du joueur, nous allons grandement augmenter la précision sur l'intégralité du tronc (*frustum*). Toutefois, en définissant le plan proche trop loin, vous allez provoquer des coupures de vos objets proches. C'est donc un ajustement à faire avec précaution et justesse pour trouver la bonne distance pour le plan proche.

Une autre astuce coûteuse en termes de performance est d'utiliser un tampon de profondeur plus précis. La plupart des tampons de profondeur ont une précision de 24 bits, mais la plupart des cartes graphiques actuelles gèrent les tampons de profondeur sur 32 bits, ce qui augmente grandement la précision. En échange de performance, vous allez avoir une meilleure précision pour vos tests de profondeur et donc réduire les conflits.

Les trois techniques que nous avons vues sont les plus courantes et faciles à mettre en place pour éviter les conflits. Il en existe d'autres, nécessitant plus de travail et ne supprimant pas pour autant l'intégralité des cas de conflits. Les conflits de profondeur sont courants, mais si vous utilisez la bonne combinaison des techniques listées, vous allez probablement ne jamais vraiment avoir de vrais problèmes.

### I-E - Remerciements

Ce tutoriel est une traduction réalisée par **Alexandre Laurent** dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](http://learnopengl.com).

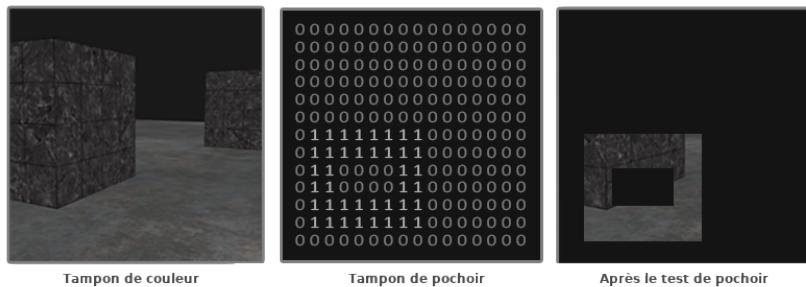
## II - Test de pochoir (stencil test)

Le test de pochoir (*stencil test*) est réalisé juste après l'exécution du *fragment shader*. Tout comme le test de profondeur, il permet d'ignorer des fragments. Les fragments non ignorés subissent ensuite le test de profondeur qui peut, à son tour, exclure encore plus de fragments. Le test de pochoir repose sur le contenu d'un autre tampon, appelé le tampon de pochoir (*stencil buffer*) que nous pouvons mettre à jour au cours du rendu pour obtenir des effets intéressants.

Un tampon de pochoir contient (généralement) 8 bits par valeur, ce qui permet un total de 256 valeurs différentes pour chaque pixel. Nous pouvons définir ces valeurs comme bon nous semble, puis abandonner les fragments suivant ces valeurs.

**i** Chaque bibliothèque de fenêtrage doit définir un tampon de pochoir. GLFW le fait automatiquement et vous n'avez pas besoin de lui dire d'en créer un, mais d'autres bibliothèques peuvent ne pas le créer par défaut. Vérifiez cet aspect dans la documentation de la bibliothèque que vous utilisez.

Voici un simple exemple de tampon de pochoir :



Le tampon de pochoir est d'abord initialisé avec des zéros, puis certaines valeurs du tampon sont passées à 1. Les fragments de la scène ne sont affichés que si la valeur de pochoir est à 1.

Les opérations sur le tampon de pochoir nous permettent de définir les valeurs du tampon lors du rendu des fragments. En changeant le contenu du tampon de pochoir au cours du rendu, nous écrivons dans celui-ci. Au cours du rendu d'une image (ou pour la prochaine image), nous pouvons lire ces valeurs pour ignorer certains fragments. Lors de l'utilisation des tampons de pochoir, vous pouvez faire ce que vous voulez, mais l'application suit souvent ce déroulement :

- activation en écriture du tampon de pochoir ;
- rendu des objets mettant à jour le tampon de pochoir ;
- désactivation en écriture du tampon de pochoir ;
- rendu des (autres) objets, dont certains fragments seront ignorés selon le contenu du tampon de pochoir.

En utilisant le tampon de pochoir, nous pouvons donc ignorer des fragments selon les fragments d'autres objets affichés dans la scène.

Vous pouvez activer le test de pochoir avec `GL_STENCIL_TEST`. À partir de ce moment, tous les appels de rendu vont influencer le tampon de pochoir :

```
glEnable(GL_STENCIL_TEST);
```

Remarquez que vous devez aussi nettoyer le tampon de pochoir avant chaque itération, tout comme pour le tampon de couleurs et celui de profondeur :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Aussi, tout comme pour la fonction `glDepthMask()` pour le test de profondeur, il y a une fonction équivalente pour le tampon de pochoir. La fonction `glStencilMask()` permet de définir un masque bit à bit utilisé. Ce masque est utilisé pour réaliser un test logique ET avec la valeur de pochoir qui va être inscrite dans le tampon. Par défaut, le masque est complètement à 1, c'est-à-dire qu'il n'affecte pas la sortie, mais si nous définissons un masque **0x00**, toutes les valeurs du tampon de pochoir finiront par être des 0. C'est équivalent à la désactivation du test de profondeur avec `glDepthMask(GL_FALSE)` :

```
glStencilMask(0xFF); // chaque bit est écrit dans le tampon de pochoir tel quel  
glStencilMask(0x00); // chaque bit devient un 0 dans le tampon de pochoir (désactivation de  
l'écriture)
```

Dans la plupart des cas, vous ne définirez le masque qu'à **0x00** ou **0xFF**, mais il est bon de savoir qu'il est possible de définir ce que l'on veut.

## II-A - Fonctions de test du pochoir

Tout comme pour le test de profondeur, vous pouvez configurer la fonction de test pour le pochoir et ainsi déterminer la manière de mettre à jour le tampon. Pour cela, il y a deux fonctions : `glStencilFunc` et `glStencilOp`.

La fonction `glStencilFunc(GLenum func, GLint ref, GLuint mask)` a trois paramètres :

- `func` : la fonction de test. Elle est utilisée pour comparer la valeur stockée dans le tampon avec la valeur `ref`. Les options sont : `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_GREATER`, `GL_NOTEQUAL`, `GL_ALWAYS`, `GL_EQUAL`, `GL_NOTEQUAL` et `GL_ALWAYS`. Elles sont équivalentes aux fonctions pour le test de profondeur ;
- `ref` : indique la valeur de référence du test de pochoir, dont le contenu est comparé à cette valeur ;
- `mask` : indique un masque utilisé dans un ET logique avec la valeur de référence et la valeur du tampon avant le test. Par défaut, le masque est complètement à 1.

Donc, dans le cas d'un exemple simple, nous aurions :

```
glStencilFunc(GL_EQUAL, 1, 0xFF)
```

Ce code indique à OpenGL que, pour n'importe quelle valeur de pochoir du fragment égale (`GL_EQUAL`) à la valeur de référence 1, le fragment passe le test et est dessiné. Sinon, il est ignoré.

Cependant, `glStencilFunc()` ne décrit que ce qu'OpenGL doit faire avec le contenu du tampon de pochoir, mais n'indique pas comment mettre à jour le tampon. C'est pour cela qu'il y a `glStencilOp()`.

La fonction `glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass)` contient trois options pour lesquelles nous pouvons indiquer l'action à prendre :

- `sfail` : l'action à faire lorsque le test échoue ;
- `dpfail` : l'action à prendre lorsque le test de pochoir réussit, mais pas le test de profondeur ;
- `dppass` : l'action à effectuer lorsque les tests de pochoir et de profondeur réussissent.

Pour chacune des options, vous pouvez indiquer les actions suivantes :

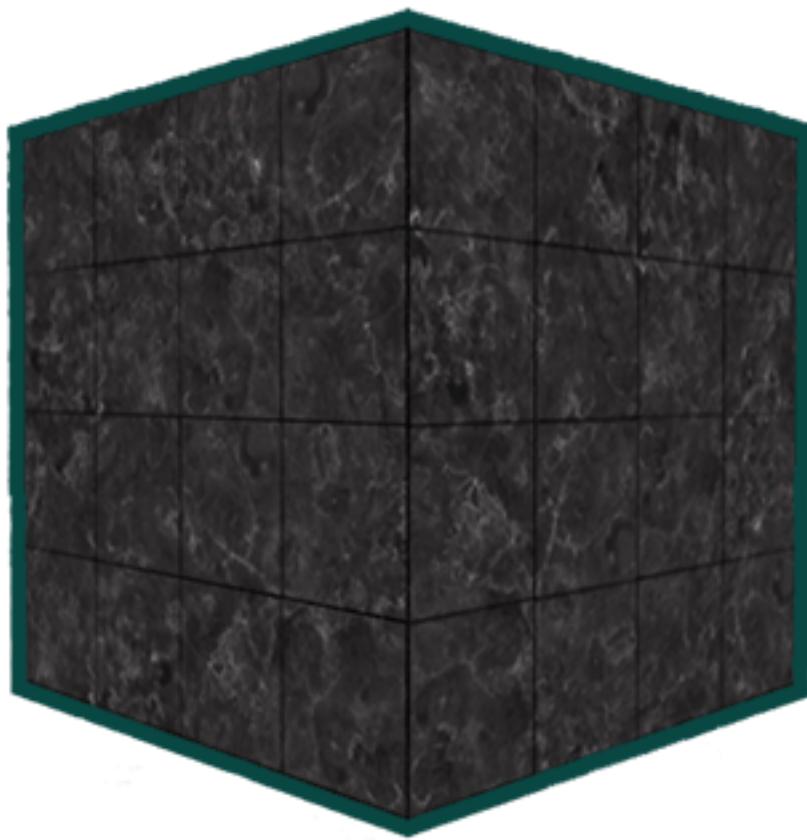
|              |   |
|--------------|---|
| GL_KEEP      | La valeur actuelle est gardée.  |
| GL_ZERO      | La valeur de pochoir est mise à 0.  |
| GL_REPLACE   | La valeur de pochoir est remplacée par la valeur de référence indiquée avec glStencilValue(). |
| GL_INCR      | La valeur de pochoir est incrémentée de 1, si elle est inférieure à la valeur maximale.       |
| GL_INCR_WRAP | Idem à GL_INCR, mais la valeur est remise à 0 si le maximum est dépassé.                      |
| GL_DECR      | La valeur de pochoir est décrémentée de 1 si elle est supérieure à la valeur minimale.        |
| GL_DECR_WRAP | Idem à GL_DECR, mais la valeur est remise à la valeur maximale lorsque 0 est dépassé.         |
| GL_INVERT    | La valeur est inversée binairement.   |

Par défaut, la fonction glStencilOp() est définie avec les options (GL\_KEEP, GL\_KEEP, GL\_KEEP). Quoiqu'il arrive, le tampon garde ses valeurs. Le comportement par défaut ne met pas à jour le tampon de pochoir. Si vous voulez écrire dans le tampon, vous devez spécifier au moins une action différente pour l'un de ces cas.

Par conséquent, en utilisant les fonctions glStencilFunc() et glStencilOp(), nous pouvons spécifier précisément quand et comment le tampon de pochoir doit être mis à jour et nous pouvons aussi indiquer lorsque le test réussit ou non, c'est-à-dire, quand le fragment doit être oublié ou non.

## II-B - Contour d'un objet

Il est difficile de comprendre le fonctionnement du test de pochoir rien qu'en lisant les sections précédentes. Voyons comment l'utiliser pour réaliser un effet mettant en avant le contour des objets.



Pour chaque objet (ici, seulement un), nous allons créer une petite bordure colorée autour des objets (combinés). Cet effet est particulièrement utile dans les jeux de stratégie, lorsque vous devez montrer les éléments sélectionnés par l'utilisateur. La procédure pour le réaliser est la suivante :

- définir la fonction de pochoir à `GL_ALWAYS` avant d'afficher les objets qui auront des bordures. Le tampon de pochoir sera rempli de 1 lors de l'affichage des objets ;
- afficher les objets ;
- désactiver l'écriture dans le tampon de pochoir et le test de profondeur ;
- agrandir légèrement les objets ;
- utiliser un autre *fragment shader* qui n'affiche que la couleur de bordure ;
- afficher les objets une nouvelle fois, mais seulement si la valeur de pochoir est différente de 1 ;
- réactiver l'écriture dans le tampon de pochoir et le test de profondeur.

Ce processus remplit le tampon de pochoir de 1 pour les fragments où les objets sont dessinés : lorsque nous souhaitons dessiner les bordures, nous dessinons une version plus grande des objets. Ainsi, lorsque le test de pochoir réussit, la version agrandie est affichée, ce qui correspond au contour de l'objet. Nous ignorons tous les fragments de l'objet plus grand qui sont compris dans l'objet original grâce au tampon de pochoir.

Nous allons d'abord créer un *fragment shader* basique qui affiche la couleur du contour. Nous définissons simplement une couleur en dur dans le *shader* :

```
shaderSingleColor
void main()
{
    FragColor = vec4(0.04, 0.28, 0.26, 1.0);
}
```

Nous allons ajouter une bordure aux deux conteneurs, mais pas au sol. Nous allons donc d'abord dessiner le sol, puis les deux conteneurs (en écrivant dans le tampon de pochoir), finalement une version plus grande des conteneurs (en ignorant les fragments qui recouvrent les conteneurs précédemment affichés).

Nous devons d'abord activer le test de pochoir et définir les actions à prendre lorsque le test réussit ou échoue :

```
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
```

Si l'un des tests échoue, nous ne faisons rien. Nous gardons la couleur du tampon de pochoir. Si les deux tests (profondeur et pochoir) réussissent, on remplace la valeur de pochoir par la valeur de référence (1) définie avec glStencilFunc().

Nous nettoyons le tampon de pochoir avec des 0 et les conteneurs remplissent le tampon de 1 :

```
glStencilFunc(GL_ALWAYS, 1, 0xFF); // tous les fragments doivent mettre à jour le tampon de pochoir
glStencilMask(0xFF); // active l'écriture dans le tampon de pochoir
normalShader.use();
DrawTwoContainers();
```

En utilisant la fonction `GL_ALWAYS`, nous nous assurons que tous les fragments des conteneurs vont mettre à jour le tampon de pochoir à 1. Comme les fragments réussissent toujours le test de pochoir, le tampon est mis à jour avec la valeur de référence à chaque fois que nous les dessinons.

Maintenant que le tampon de pochoir est à jour avec des 1, là où sont les conteneurs, nous pouvons dessiner la version agrandie des conteneurs, mais en désactivant l'écriture dans le tampon de pochoir :

```
glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
glStencilMask(0x00); // désactive l'écriture dans le tampon de pochoir
glDisable(GL_DEPTH_TEST);
shaderSingleColor.use();
DrawTwoScaledUpContainers();
```

Nous définissons la fonction de test à `GL_NOTEQUAL` pour s'assurer que nous ne dessinons que les parties du conteneur où le tampon de pochoir n'est pas à 1, donc que les parties qui ne recouvrent pas les conteneurs affichés précédemment. Notez que nous désaktivons aussi le test de profondeur, afin que les bordures ne soient pas recouverte par le sol.

Aussi, assurez-vous que vous réactivez le test de profondeur une fois les opérations de rendu finies.

La routine d'affichage des contours de la scène doit ressembler à ceci :

```
glEnable(GL_DEPTH_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

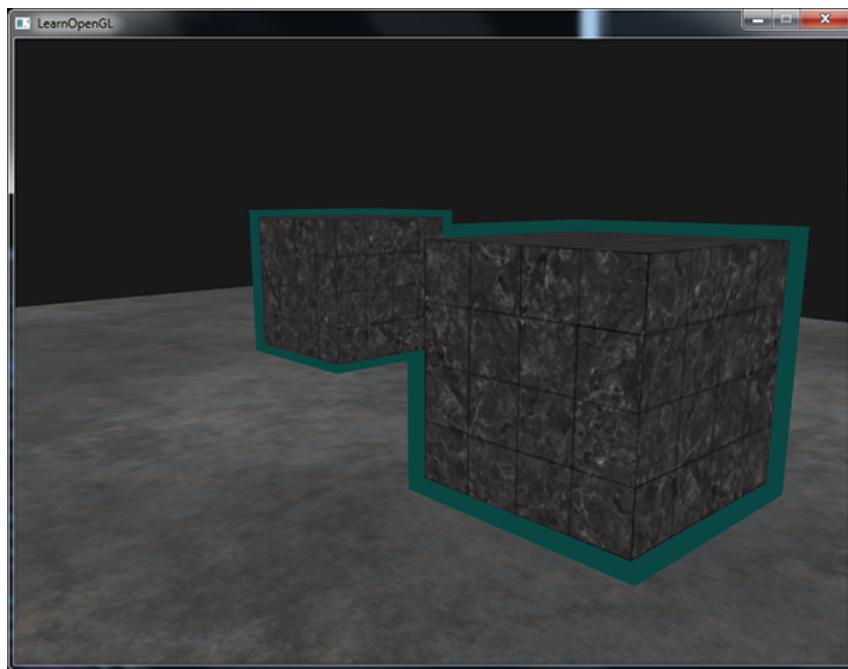
glStencilMask(0x00); // ne pas mettre à jour le tampon de pochoir lors du dessin du sol
normalShader.use();
DrawFloor()

glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilMask(0xFF);
DrawTwoContainers();

glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
glStencilMask(0x00);
glDisable(GL_DEPTH_TEST);
shaderSingleColor.use();
DrawTwoScaledUpContainers();
glStencilMask(0xFF);
glEnable(GL_DEPTH_TEST);
```

Si vous avez compris le fonctionnement du test de pochoir, ce fragment de code ne devrait pas être compliqué. Sinon, essayez de relire les sections précédentes afin de mieux comprendre ce que chaque fonction fait, maintenant que vous connaissez la mise en pratique.

Le résultat de cet algorithme, appliquée à la scène du [tutoriel du test de profondeur](#) ressemble à ceci :



Le code source complet de ce programme de contour est [Source disponible ici](#).

**i** Vous pouvez voir que les bordures se rejoignent entre les deux conteneurs. Généralement, c'est l'effet voulu (imaginez-vous sélectionner une dizaine d'unités dans un jeu de stratégie). Si vous souhaitez une bordure complète pour chaque objet, vous devez nettoyer le tampon de pochoir pour chaque objet et être astucieux avec le tampon de profondeur.

L'algorithme de contour que vous venez de voir est utilisé dans de nombreux jeux pour mettre en avant les objets sélectionnés (par exemple, dans les jeux de stratégie) et un tel algorithme peut être implémenté dans une classe. Vous pouvez simplement définir un booléen dans la classe modèle pour dessiner l'objet avec ou sans bordures. Si vous souhaitez être créatif, vous pouvez même donner un aspect plus naturel aux bordures avec un post-traitement comme un flou gaussien.

Les tests de pochoir ont plusieurs utilités, autres qu'afficher le contour d'un objet, telles qu'afficher une texture dans un miroir afin que la texture corresponde à la forme du miroir ou afficher des ombres en temps réel avec une technique appelée « *shadow volumes* ». Les tampons de pochoir nous offrent un outil sympa de plus pour notre boîte à outils OpenGL.

## II-C - Remerciements

Ce tutoriel est une traduction réalisée par [Alexandre Laurent](#) dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

### III - Mélange des couleurs (blending)

Le mélange des couleurs (*blending*) est une technique classique pour implémenter la transparence entre objets dans OpenGL. La transparence est le fait d'avoir une couleur non unie sur l'intégralité (ou une partie) d'un objet. À la place, l'objet sera teinté avec les couleurs des objets se plaçant derrière lui. Une fenêtre teintée est un objet transparent : le verre a une couleur propre, mais la couleur finale contient la couleur de tous les objets derrière la fenêtre. Cela explique l'origine du nom, sachant que nous mélangeons plusieurs couleurs (de différents objets) vers une unique couleur. La transparence nous permet de voir à travers les objets.



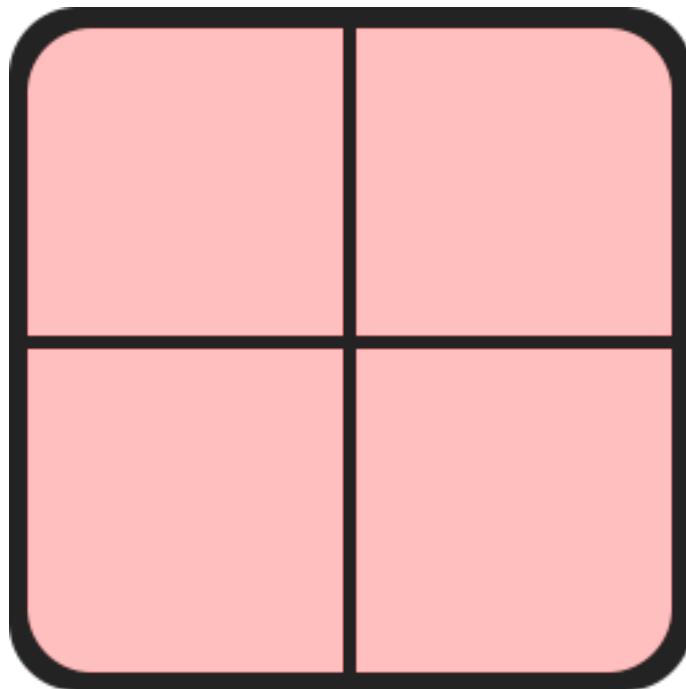
Full transparent window



Partially transparent window

Les objets transparents peuvent être complètement transparents (ils laissent passer toutes les couleurs) ou partiellement transparents (ils laissent les couleurs passer tout en donnant sa propre couleur). Le niveau de transparence d'un objet est défini par sa valeur alpha qui est la quatrième composante d'un vecteur de couleur. Jusqu'à présent, nous avons toujours donné une valeur de 1,0 à cette quatrième composante, pour avoir des objets opaques. Une valeur alpha de 0,0 rend un objet complètement transparent. Une valeur de 0,5 indique que la couleur de l'objet doit prendre 50 % de sa propre couleur et 50 % de l'objet derrière.

Les textures que nous avons utilisées jusqu'à présent ne contenait que trois composantes par *texel* : rouge, vert et bleu, mais certaines textures peuvent intégrer un canal alpha avec une valeur alpha en plus pour chaque *texel*. La valeur alpha indique précisément quelles parties de la texture doit être transparente. Par exemple, la texture suivante a une valeur alpha de 0,25 sur les parties représentant le verre (normalement, la texture devrait être complètement rouge, mais comme elle est 75 % transparente, elle montre le fond du site, et semble avoir perdu du rouge) une valeur de 0,0 dans ses coins.



Nous allons bientôt ajouter cette fenêtre à la scène, mais parlons d'abord d'une technique plus simple pour les textures qui sont soit totalement transparentes, soit totalement opaques.

### III-A - Rejeter des fragments

Certaines images n'ont pas de transparence partielle, elles représentent quelque chose ou rien suivant la couleur de la texture. Prenons l'exemple de l'herbe : pour créer de l'herbe avec peu d'effort, vous copiez une texture d'herbe sur un rectangle 2D et placez ce rectangle dans la scène. Toutefois, l'herbe n'est pas vraiment rectangulaire donc vous devez afficher certaines parties de la texture tout en ignorant le reste.

La texture suivante suit ce concept : elle est soit complètement opaque (une valeur alpha de 1,0), soit totalement transparente (une valeur alpha de 0,0). Il n'y a jamais une valeur intermédiaire. Vous pouvez voir que, lorsqu'il n'y a pas d'herbe, l'image affiche le fond du site à la place :



Ainsi, lorsque vous ajoutez de la végétation à votre scène, vous ne souhaitez pas voir le rectangle de l'image, mais juste l'herbe et voir à travers le reste de l'image. Nous souhaitons **rejeter** les fragments complètement transparents et ne pas les stocker dans le tampon d'image. Avant d'implémenter cette technique, nous devons apprendre à charger une texture transparente.

Pour charger une texture avec des valeurs alpha, il n'y a pas grand-chose à modifier. `stb_image` charge automatiquement le canal alpha lorsqu'il est présent, mais nous devons indiquer à OpenGL que notre texture utilise un canal alpha :

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

Aussi, assurez-vous que vous récupérez bien les quatre composantes de votre texture dans le *fragment shader* et non juste RGB :

```
void main()
{
    // FragColor = vec4(texture(texture1, TexCoords), 1.0);
    FragColor = texture(texture1, TexCoords);
}
```

Maintenant que vous savez comment charger des textures transparentes, il est temps d'ajouter quelques brins d'herbe dans notre scène du **test de profondeur**.

Nous créons un vecteur où nous ajoutons plusieurs `glm::vec3` pour indiquer la position des brins d'herbe :

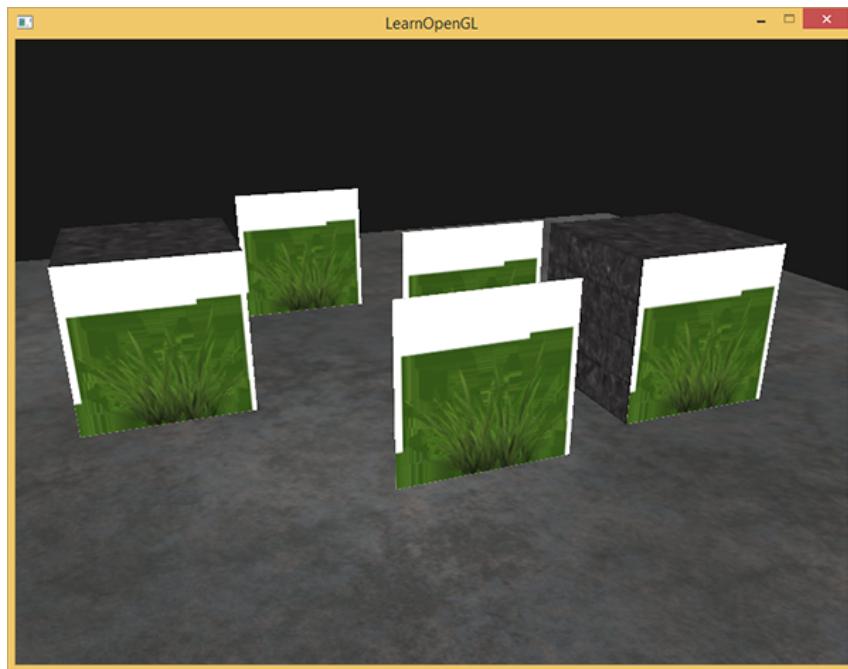
```
vector<glm::vec3> vegetation;
vegetation.push_back(glm::vec3(-1.5f, 0.0f, -0.48f));
vegetation.push_back(glm::vec3( 1.5f, 0.0f, 0.51f));
vegetation.push_back(glm::vec3( 0.0f, 0.0f, 0.7f));
vegetation.push_back(glm::vec3(-0.3f, 0.0f, -2.3f));
vegetation.push_back(glm::vec3( 0.5f, 0.0f, -0.6f));
```

Chaque brin d'herbe est affiché grâce à un simple rectangle avec la texture d'herbe. Ce n'est pas une représentation 3D parfaite de l'herbe, mais c'est bien plus performant que d'afficher des modèles complexes. Avec quelques astuces, notamment en ajoutant plusieurs brins d'herbe à une même position et leur donnant une rotation différente, vous pouvez obtenir de bons résultats.

Comme la texture d'herbe est appliquée à un objet rectangulaire, nous devons créer un autre VAO, remplir le VBO et définir les attributs de sommets. Après avoir dessiné le sol et les deux cubes, nous dessinons les brins :

```
glBindVertexArray(vegetationVAO);
glBindTexture(GL_TEXTURE_2D, grassTexture);
for(unsigned int i = 0; i < vegetation.size(); i++)
{
    model = glm::mat4();
    model = glm::translate(model, vegetation[i]);
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Si vous exécutez l'application, vous devriez obtenir ceci :



Le résultat est ainsi, car OpenGL ne sait pas quoi faire avec les valeurs alpha, ni même qu'il doit les ignorer. Nous devons le faire nous-même. Heureusement, c'est très facile avec les *shaders*. Le langage GLSL fournit la commande `discard` qui, une fois appelée, provoque l'abandon du fragment : par conséquent, il n'aura aucun impact sur le tampon de couleurs. Nous pouvons vérifier dans le *fragment shader* si le fragment a une valeur alpha inférieure à un seuil donné et, grâce à cette commande, nous pouvons l'ignorer :

```
#version 330 core
out vec4 FragColor;

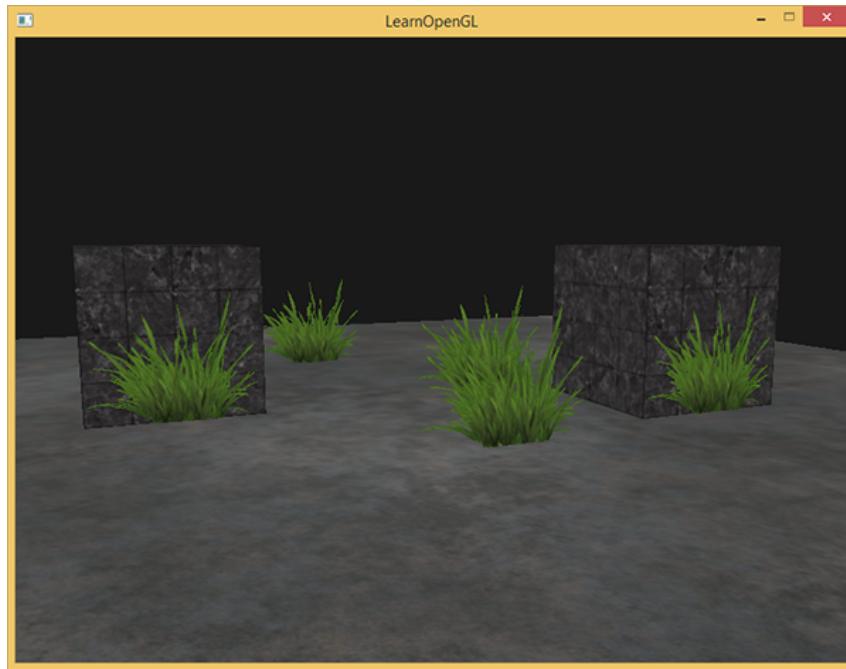
in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    FragColor = texColor;
```

{ }

Nous vérifions si la couleur de la texture échantillonnée a une valeur alpha inférieure à 0,1 et, dans ce cas, nous ignorons le fragment. Le *fragment shader* nous permet de n'afficher que les fragments qui sont opaques. Voici la scène corrigée :



**i** Remarquez que, lors de l'échantillonnage des bordures de la texture, OpenGL interpole les valeurs de la bordure avec la prochaine répétition de la texture (car nous avons utilisé le paramètre `GL_REPEAT`). Habituellement, c'est le comportement voulu, mais lorsque nous utilisons des valeurs transparentes, le haut de la texture est interpolé avec le bas de la texture, qui, lui, est opaque. Cela forme une bordure semi-transparente autour du rectangle. Pour éviter cela, définissez le wrapping à `GL_CLAMP_TO_EDGE` lorsque vous utilisez des textures avec transparence :

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Vous pouvez consulter le [Source code source ici](#).

### III-B - Mélange de couleurs

L'abandon des fragments est une bonne approche, mais elle ne permet pas d'afficher des images semi-transparentes ; nous ne pouvons que dessiner le fragment ou l'abandonner. Pour afficher des images avec différents niveaux de transparence, nous devons utiliser le **mélange de couleurs**. Comme pour la plupart des fonctionnalités d'OpenGL, nous pouvons activer le mélange de couleurs en activant `GL_BLEND` :

```
 glEnable(GL_BLEND);
```

Maintenant que nous avons activé le mélange de couleurs, nous devons indiquer à OpenGL **comment** l'utiliser.

Dans OpenGL, le mélange se fait avec l'équation suivante :

$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

- $\bar{C}_{source}$

: la couleur source. C'est la couleur provenant de la texture ;

$$\bar{C}_{destination}$$

: la couleur de destination. C'est la couleur qui est actuellement dans le tampon de couleurs ;

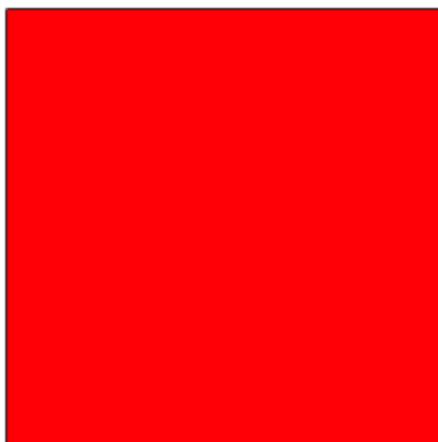
$$F_{source}$$

: le facteur de la source. Définit l'impact de la couleur source ;

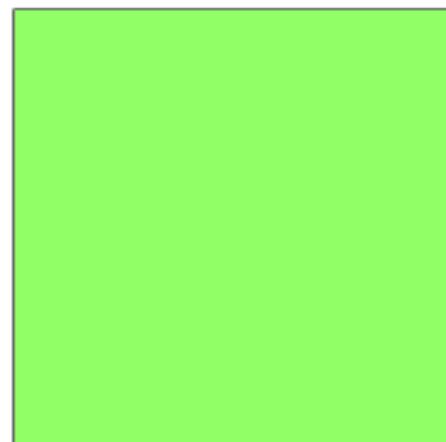
$$F_{destination}$$

: le facteur de destination. Définit l'impact de la couleur destination.

Après l'exécution du *fragment shader* et la réussite de tous les tests, l'équation de mélange est appliquée pour déterminer la couleur de sortie à partir de la couleur actuellement dans le tampon de couleurs (soit, la couleur du fragment précédemment sauvegardée). Les couleurs de la source et de la destination sont définies automatiquement par OpenGL, mais nous pouvons imposer le facteur de source et de destination. Voici un exemple simple :



(1.0, 0.0, 0.0, 1.0)



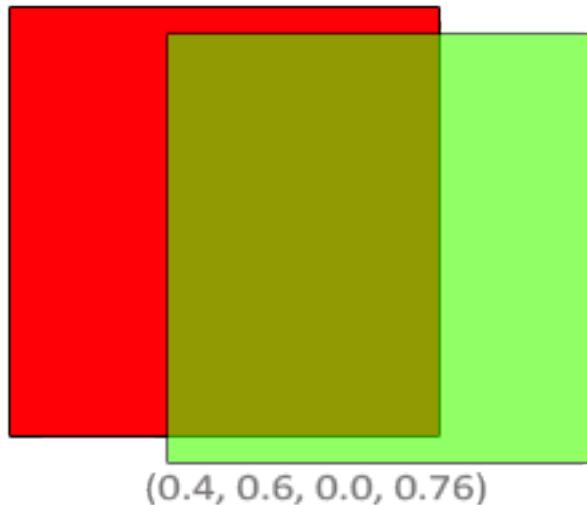
(0.0, 1.0, 0.0, 0.6)

Nous avons deux carrés et nous souhaitons superposer le vert sur le rouge. Le carré rouge sera la couleur de destination (et devra donc être la première couleur dans le tampon de couleurs). Ensuite, nous allons dessiner le carré vert au-dessus.

La question qui se pose est : quelle sont les valeurs des facteurs ? D'abord, nous souhaitons multiplier le carré vert avec sa valeur alpha, nous définissons donc  $F_{src}$  à la valeur alpha de la couleur source, soit 0.6. Ensuite, il est logique que le carré de destination ait une valeur alpha du reste de la transparence. Si le carré vert donne 60 % de la couleur finale, alors le carré rouge fournit les 40 % restant, soit 1.0 – 0.6. Ainsi, nous avons défini  $F_{destination}$  à la valeur alpha de la couleur source soustraite de 1. L'équation devient :

$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} * (1 - 0.6)$$

Le résultat donne une couleur 60 % verte et 40 % rouge :



La couleur résultante est alors stockée dans le tampon de couleurs, remplaçant l'ancienne.

Bien, mais comment faire cela avec OpenGL ? OpenGL fournit la fonction `glBlendFunc()` spécialement pour ça.

La fonction `glBlendFunc(GLenum sfactor, GLenum dfactor)` attend deux paramètres pour définir les facteurs source et destination. OpenGL définit quelques options dont nous décrirons ci-dessous les plus courantes. Notez que la constante  $\bar{C}_{constant}$  peut être définie avec la fonction `glBlendColor()`.

| Option                 | Valeur  |
|------------------------|---|
| GL_ZERO                | Le facteur est égal à 0.  |
| GL_ONE                 | Le facteur est égal à 1.  |
| GL_SRC_COLOR           | Le facteur est égal à la couleur source<br>$\bar{C}_{source}$                                     |
| GL_ONE_MINUS_SRC_COLOR | Le facteur est égal à 1 moins la couleur source<br>$1 - \bar{C}_{source}$                         |
| GL_DST_COLOR           | Le facteur est égal à la couleur de destination<br>$\bar{C}_{destination}$                        |
| GL_ONE_MINUS_DST_COLOR | Le facteur est égal à 1 moins la couleur de destination<br>$1 - \bar{C}_{destination}$            |
| GL_SRC_ALPHA           | Le facteur est égal à la composante alpha de la couleur source<br>$\bar{C}_{source}$              |
| GL_ONE_MINUS_SRC_ALPHA | Le facteur est égal à 1 moins la couleur source<br>$1 - \bar{C}_{source}$                         |
| GL_DST_ALPHA           | Le facteur est égal à la composante alpha de la couleur de destination<br>$\bar{C}_{destination}$ |
| GL_ONE_MINUS_DST_ALPHA | Le facteur est égal à 1 moins la couleur de destination<br>$1 - \bar{C}_{destination}$            |

|                             |   |
|-----------------------------|---|
|                             | de la couleur de destination<br>$\bar{C}_{destination}$                                   |
| GL_CONSTANT_COLOR           | Le facteur est égal à la couleur constante<br>$\bar{C}_{constant}$                        |
| GL_ONE_MINUS_CONSTANT_COLOR | Le facteur est égal à 1 moins la couleur constante<br>$\bar{C}_{constant}$                |
| GL_CONSTANT_ALPHA           | Le facteur est égal à la composante alpha de la couleur constante<br>$\bar{C}_{constant}$ |
| GL_ONE_MINUS_CONSTANT_ALPHA | Le facteur est égal à $1 - alpha$ de la couleur constante<br>$\bar{C}_{constant}$         |

Pour obtenir le même résultat que dans l'exemple précédent, nous devons prendre la composante *alpha* de la couleur source comme source et  $1 - alpha$  pour le facteur destination. Cela correspond à l'appel suivant :

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Il est aussi possible de définir des options différentes pour les canaux RGB et alpha avec la fonction `glBlendFuncSeparate()` :

```
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);
```

Cette fonction définit les composantes RGB aux mêmes valeurs que précédemment, mais fait en sorte que seule la couleur alpha résultante soit influencée par la valeur alpha source.

OpenGL nous offre encore plus de flexibilité en nous permettant de changer l'opérateur utilisé dans l'équation mêlant la source à la destination. Jusqu'à présent, la source et la destination sont ajoutées, mais nous pouvons les soustraire si nous le souhaitons. Cela est grâce à la fonction `glBlendEquation(GLenum mode)` qui nous permet trois options :

- `GL_FUNC_ADD` : l'opération par défaut, additionne les deux composantes :

$$\bar{C}_{result} = \textcolor{green}{Src} + \textcolor{red}{Dst}$$

- `GL_FUNC_SUBTRACT` : soustrait les deux composantes entre elles :

$$\bar{C}_{result} = \textcolor{green}{Src} - \textcolor{red}{Dst}$$

- `GL_FUNC_REVERSE_SUBTRACT` : soustrait les deux composantes, mais dans l'autre sens :

$$\bar{C}_{result} = Dst - Src$$

Généralement, `glBlendEquation()` n'est pas appelée, car l'option `GL_FUNC_ADD` correspond à la majorité des cas. Mais si vous voulez tout faire pour ne pas être dans le courant, alors n'importe quelle autre option vous conviendra.

### III-C - Affichage de textures semi-transparentes

Maintenant que nous savons comment OpenGL gère la transparence, nous pouvons ajouter plusieurs fenêtres semi-transparentes. Nous allons utiliser la même scène qu'au début du tutoriel, mais au lieu de la texture d'herbe, nous utilisons la texture de la **fenêtre transparente**.

Premièrement, lors de l'initialisation, nous activons la fusion des couleurs et définissons l'équation de fusion :

```
 glEnable(GL_BLEND);
 glEnable(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Comme nous activons la transparence, il n'y a plus besoin d'ignorer des fragments. Le *fragment shader* correspond à sa version originale :

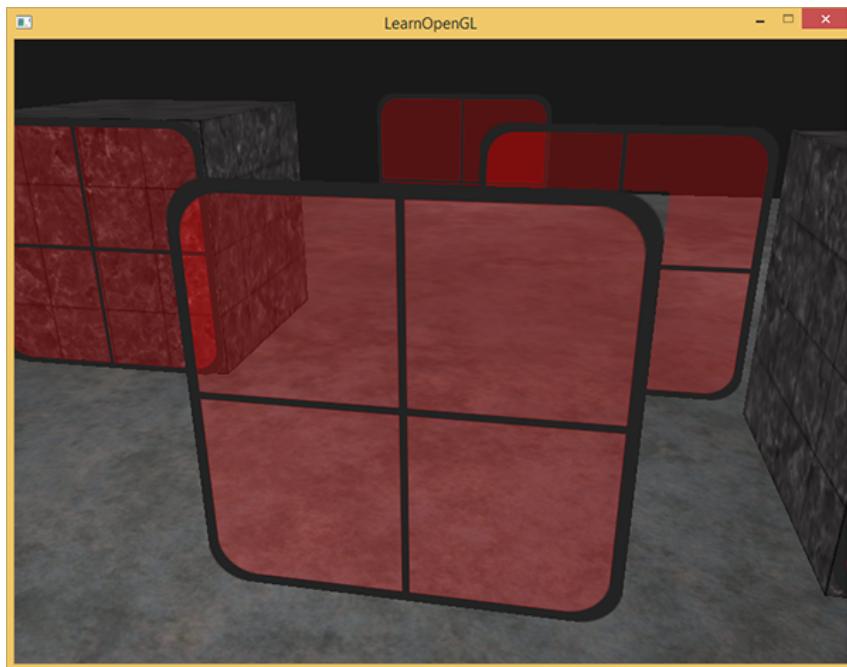
```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    FragColor = texture(texture1, TexCoords);
}
```

Cette fois (en fait, chaque fois qu'OpenGL affiche un fragment), la couleur du fragment est mélangée à la couleur du fragment présente dans le tampon de couleurs suivant sa valeur alpha. Comme le verre de la texture de la fenêtre est semi-transparent, nous pouvons voir le reste de la scène en regardant à travers.



Si vous regardez de plus près, vous allez voir que cela ne va pas. Les parties transparentes de la première fenêtre cachent les fenêtres derrière. Que se passe t-il ?

La raison est que le test de profondeur en conjonction avec la transparence ne font pas bon ménage. Lorsque vous écrivez dans le tampon de profondeur, le test de profondeur ne prend pas en compte la transparence du fragment, donc les parties transparentes sont écrasées dans le tampon de profondeur. Le résultat est que le rectangle complet de la fenêtre passe dans le test de profondeur, peu importe sa transparence. Même si la partie transparente fait que la fenêtre derrière soit visible, le test de profondeur ignore cela.

Par conséquent, nous ne pouvons pas simplement afficher les fenêtres comme nous le voulons et espérer que le tampon de profondeur règle les problèmes pour nous ; c'est aussi là que l'implémentation de la transparence devient un peu moins évidente. Pour s'assurer que les fenêtres à l'arrière de la scène s'affichent, nous devons dessiner les fenêtres en partant de la fin. Cela veut dire que nous devons trier les fenêtres de la plus lointaine à la plus proche selon la position de la caméra.

**i** Avec les objets complètement transparents, comme l'herbe, nous avons l'option de simplement ignorer les fragments transparents, ce qui nous évite des complications (pas de problème avec la profondeur).

### III-C-1 - Ne pas casser l'ordre

Pour que la transparence fonctionne avec plusieurs objets, nous devons dessiner les objets les plus éloignés avant ceux qui sont plus proches. Les objets normaux (non transparents) peuvent être dessinés comme à l'habitude, ils ne doivent donc pas être triés. Nous devons nous assurer qu'ils sont affichés avant les objets transparents (triés). Lors du dessin d'une scène avec des objets transparents et non transparents, l'algorithme d'affichage est le suivant :

- dessiner les objets opaques ;
- trier les objets transparents ;
- dessiner les objets transparents suivant le tri.

Une façon de trier les objets transparents est d'utiliser la distance entre l'objet et le spectateur. Cela peut se faire en prenant la distance entre le vecteur position de la caméra et le vecteur position de l'objet. Ensuite, nous stockons

la distance avec son vecteur position dans une table de hachage (`std::map`). La `std::map` trie automatiquement les objets suivant leur clé : une fois que nous avons ajouté toutes les positions avec leur distance comme clé, ils sont automatiquement triés suivant la distance. Cela donne le code suivant :

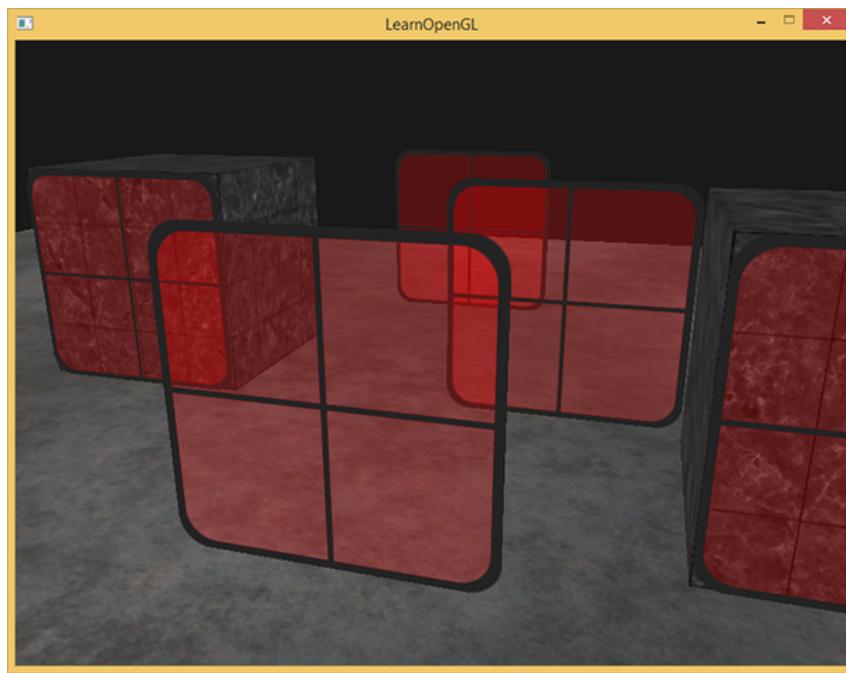
```
std::map<float, glm::vec3> sorted;
for (unsigned int i = 0; i < windows.size(); i++)
{
    float distance = glm::length(camera.Position - windows[i]);
    sorted[distance] = windows[i];
}
```

Le résultat est un conteneur d'objets triés qui stocke les positions de chaque fenêtre suivant leur `distance`, du plus proche au plus éloigné.

Ensuite, au moment du rendu, nous prenons chaque valeur de la table de hachage dans l'ordre inverse (du plus loin au plus proche) et nous dessinons la fenêtre correspondante :

```
for(std::map<float,glm::vec3>::reverse_iterator it = sorted.rbegin(); it != sorted.rend(); ++it)
{
    model = glm::mat4();
    model = glm::translate(model, it->second);
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Nous utilisons un itérateur inversé de la `std::map` pour parcourir les éléments et placer le rectangle des fenêtres à leur position. Cette approche simple permet de corriger le problème précédent et d'obtenir la scène suivante :



Vous pouvez trouver le code source complet [Source ici](#).

Même si le tri des objets selon leur distance fonctionne correctement pour certains scénarios, il ne prend pas en compte les transformations (rotations, redimensionnement) ni les objets ayant une forme plus complexe.

Le tri des objets dans la scène est compliqué (et coûteux en termes de performance) et dépend grandement du type de scène que vous avez. L'affichage d'une scène avec des objets solides et transparents n'est pas aussi simple. Il existe des techniques avancées comme la transparence indépendante de l'ordre d'affichage (*order independent transparency*), mais ceux-ci dépassent le cadre de ce tutoriel. Pour le moment, vous devez continuer ainsi, mais si

vous êtes prudent et que vous connaissez les limitations, vous pouvez déjà obtenir des implémentations décentes de la transparence.

### III-D - Remerciements

Ce tutoriel est une traduction réalisée par [Alexandre Laurent](#) dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

### IV - Suppression de faces (culling)

Essayez de visualiser mentalement un cube et comptez le nombre de faces que vous pouvez voir, et ce, dans n'importe quelle direction. Si votre imagination n'est pas trop débordante, vous devriez en compter au maximum trois. Pourquoi donc essayerons-nous de dessiner les trois autres faces alors qu'elles ne sont même pas visibles ? Si nous pouvions les ignorer d'une façon ou d'une autre, nous économiserions plus de 50 % du temps d'exécution du *fragment shader*.

**i** Nous disons plus de 50 % et non 50 %, car sous certains angles, seules deux, voire une seule face, sont visibles. Dans ce cas-là, nous évitons **plus** de 50 % du calcul.

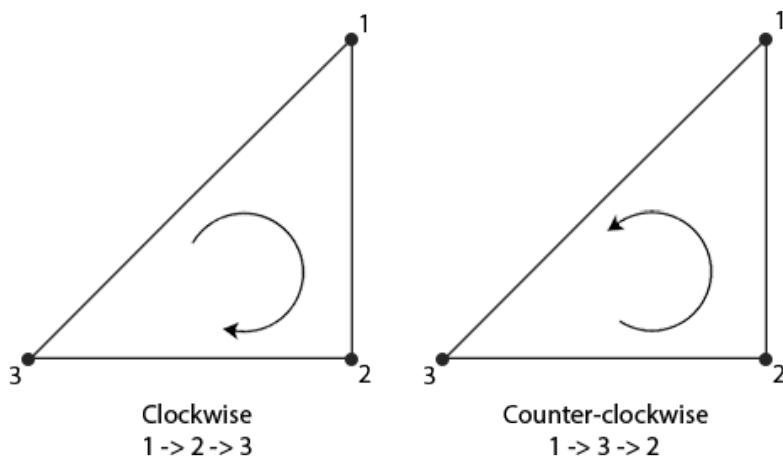
C'est une superbe idée, mais il y a un problème à résoudre : comment savoir si la face d'un objet est visible ou non pour le spectateur ?

En imaginant n'importe quelle forme fermée, chacune de ses faces possède deux côtés. Chaque côté sera soit face au spectateur, soit dans l'autre sens. Quid de ne dessiner que les faces qui sont réellement dirigées vers le spectateur ?

C'est exactement ce que fait la technique de suppression des faces arrière (*backface culling*). OpenGL vérifie si les faces avant sont orientées vers le spectateur et ne dessine que celles-ci tout en ignorant celles qui sont orientées dans l'autre sens. Ainsi, on évite beaucoup d'appels au *fragment shader* (qui sont coûteux !). Nous devons indiquer à OpenGL quelles sont les faces avant et celles qui sont des faces arrière. Pour cela, OpenGL utilise une astuce en analysant l'ordre des sommets (*winding order*).

### IV-A - Ordonnancement des sommets

Lorsque nous définissons les sommets d'un triangle, nous le faisons dans un certain ordre : soit dans le sens des aiguilles d'une montre, soit dans le sens inverse. Chaque triangle contient trois sommets et nous spécifions ces trois sommets dans l'un des sens suivant :



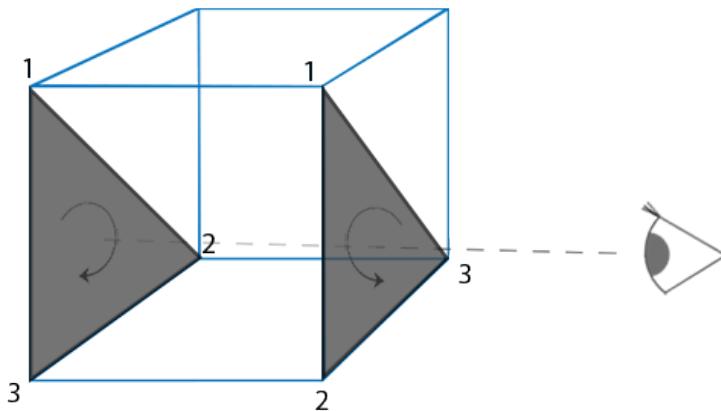
Comme vous pouvez le voir dans l'image, nous définissons d'abord le sommet 1 et nous pouvons définir ensuite soit le sommet 2, soit le 3. Ce choix détermine l'ordre des sommets du triangle. Le code suivant met en application ce concept :

```
float vertices[] = {
    // sens horaire (clockwise)
    vertices[0], // sommet 1
    vertices[1], // sommet 2
    vertices[2], // sommet 3
    // sens anti-horaire (counter-clockwise)
    vertices[0], // sommet 1
    vertices[2], // sommet 3
    vertices[1] // sommet 2
};
```

Chaque ensemble de trois sommets formant un triangle contient donc un ordre. OpenGL utilise cette information lors du rendu de la primitive pour déterminer si le triangle fait face ou non à la caméra. Par défaut, les triangles avec les sommets ordonnés dans le sens inverse des aiguilles d'une montre sont traités comme la face avant du triangle.

Lors de la définition de vos sommets, vous devez visualiser le triangle comme s'il vous faisait face, afin que chaque triangle ait ses sommets dans le sens inverse des aiguilles d'une montre. Ainsi, l'ordre des sommets est calculé pendant la *rasterization*, donc après exécution du *vertex shader*. Les sommets sont ainsi vus comme du point de vue du spectateur.

Tous les sommets du triangle auxquels le spectateur fait face sont évidemment dans le bon sens, mais les sommets des triangles de l'autre côté du cube sont maintenant affichés d'une telle manière que l'ordre des sommets est différent. Le résultat est que les triangles devant nous présentent leur face avant alors que les triangles à l'arrière présentent leur face arrière. Voici un schéma pour montrer cela :



Dans les données des sommets, nous aurions dû définir des triangles avec le sens anti-horaire (le triangle de devant dans le sens 1, 2, 3 et le triangle à l'arrière avec le sens 1, 2 et 3, si nous voulions voir le triangle depuis l'avant). Par contre, du point de vue du spectateur, le triangle de derrière est affiché dans le sens inverse des aiguilles d'une montre si nous le dessinons dans l'ordre 1, 2, 3. Même si nous avons spécifié le triangle de derrière dans le sens anti-horaire, il est maintenant affiché dans le sens horaire. C'est exactement ce que nous voulons : éliminer (*cull*) les faces non visibles.

## IV-B - Élimination des faces

Au début du tutoriel, nous avons dit qu'OpenGL peut éliminer la face arrière des triangles. Maintenant que nous savons comment définir l'ordre des sommets, nous pouvons commencer à utiliser l'option d'élimination des faces d'OpenGL, qui est désactivée par défaut.

Les données des sommets du cube du [tutoriel précédent](#) n'ont pas été définies en tenant compte de l'ordre des sommets. Du coup, j'ai mis en ligne les [données](#) définissant les sommets dans le sens inverse des aiguilles d'une montre. C'est une bonne pratique d'essayer de voir que ces sommets sont tous définis dans cet ordre.

Pour activer l'élimination des faces, nous devons juste activer l'option OpenGL GL\_CULL\_FACE :

```
glEnable(GL_CULL_FACE);
```

À partir de maintenant, toutes les faces qui ne sont pas des faces avant seront éliminées (essayez d'entrer dans le cube et voir l'intérieur des faces qui sont évidemment éliminées). Actuellement, on économise 50 % des calculs sur le rendu des fragments, mais notez que cela fonctionne uniquement avec les formes fermées, comme les cubes. Afin de voir les deux côtés, nous devons désactiver l'élimination des faces. C'est le cas pour les feuilles du [tutoriel précédent](#).

OpenGL permet de changer quelles faces à éliminer. Par exemple, si nous voulons éliminer les faces avant et non les faces arrière. Nous pouvons définir ce comportement avec la fonction glCullFace() :

```
glCullFace(GL_FRONT);
```

La fonction glCullFace() a trois options possibles :

- GL\_BACK : élimine les faces arrière ;
- GL\_FRONT : élimine les faces avant ;
- GL\_FRONT\_AND\_BACK : éliminent les faces avant et arrière.

La valeur initiale de glCullFace() est GL\_BACK. En plus de spécifier la face à éliminer, nous pouvons aussi indiquer à OpenGL que nous préférons considérer les faces définies dans le sens des aiguilles d'une montre comme face avant à la place des faces dans le sens anti-horaire avec la fonction glFrontFace() :

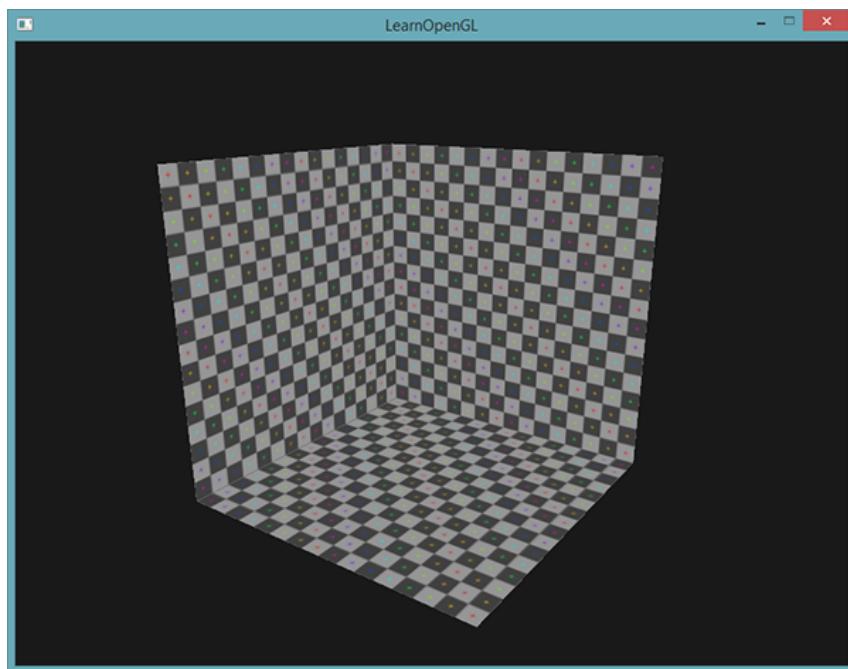
```
glFrontFace(GL_CCW);
```

La valeur par défaut est GL\_CCW pour les faces dans le sens anti-horaire. L'autre option étant GL\_CW.

Pour faire un simple test, nous pouvons inverser l'ordre en indiquant à OpenGL que les faces avant sont maintenant définie par l'ordre horaire à la place du sens anti-horaire.

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glFrontFace(GL_CW);
```

Le résultat est que seules les faces arrière sont affichées :



Notez que vous pouvez créer le même effet en éliminant les faces avant avec l'ordonnancement anti-horaire par défaut.

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

Comme vous pouvez le constater, l'élimination des faces est un outil puissant pour augmenter les performances de vos applications OpenGL avec un effort minimal. Vous devez auparavant déterminer les objets qui pourraient bénéficier de l'élimination des faces et ceux qui ne peuvent être optimisés.

## IV-C - Exercices

- Pouvez-vous redéfinir les données des sommets en spécifiant chaque triangle dans le sens horaire, puis afficher la scène en définissant les triangles horaires comme faces avant ? [Source](#) [Solution](#).

## IV-D - Remerciements

Ce tutoriel est une traduction réalisée par [Alexandre Laurent](#) dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

## V - Tampon d'image (frame buffers)

Jusqu'à présent, nous avons vu plusieurs composantes d'un tampon d'image : un tampon pour les couleurs, un autre pour la profondeur et finalement un dernier pour enlever certains fragments suivant une condition définie par l'utilisateur. La combinaison de ces tampons s'appelle un tampon d'image (*frame buffer*) et est stocké quelque part en mémoire. OpenGL nous donne la flexibilité de définir nos propres tampons d'image et donc de définir nos propres tampons de couleur, de profondeur et de pochoir.

Les opérations de rendu que nous avons effectuées jusqu'à présent ont été réalisées sur le tampon de rendu attaché au tampon d'image par défaut. Le tampon d'image par défaut est créé et configuré lors de la création de votre fenêtre (GLFW le fait pour nous). En créant votre propre tampon d'image, vous pouvez obtenir un nouvel endroit où faire le rendu.

L'utilité des tampons d'image peut ne pas être immédiat, mais le rendu de la scène dans un second tampon d'image permet de créer des miroirs ou d'effectuer de super effets de post-traitement (post-process). Premièrement, nous allons voir comment ils fonctionnent ; nous allons ensuite les utiliser pour implémenter ces super effets.

## V-A - Créer un tampon d'image

Comme pour tout autre objet OpenGL, nous devons créer un objet tampon d'image (FBO) en utilisant la fonction `glGenFramebuffers()` :

```
unsigned int fbo;  
 glGenFramebuffers(1, &fbo);
```

Cela a déjà été vu une dizaine de fois et la mise en place d'un FBO est similaire à la mise en place des autres objets OpenGL. Premièrement, on crée l'objet, on le lie en tant que FBO actif, on effectue des opérations et on le délie. Pour lier un FBO, on utilise :

```
 glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

En liant la cible `GL_FRAMEBUFFER`, toutes les opérations de lecture et d'écriture sur le tampon d'image seront effectuées sur le tampon d'image lié. Il est possible de lier un tampon d'image, pour la lecture, différent de celui pour l'écriture en utilisant `GL_READ_FRAMEBUFFER` et `GL_DRAW_FRAMEBUFFER`. Le tampon d'image lié à `GL_READ_FRAMEBUFFER` est utilisé pour les opérations de lecture telles que `glReadPixels()` et le tampon d'image lié à `GL_DRAW_FRAMEBUFFER` est utilisé comme destination pour le rendu, le nettoyage et toutes autres opérations d'écriture. La plupart du temps, vous n'avez pas besoin de différencier les opérations et vous utilisez donc `GL_FRAMEBUFFER`.

Malheureusement, nous ne pouvons pas utiliser notre tampon d'image pour le moment, car il n'est pas **complet**. Un tampon d'image est complet lorsque ces prérequis sont satisfaits :

- nous devons attacher au moins un tampon (couleur, profondeur ou stencil) ;
- il doit y avoir au moins une attache pour les couleurs ;
- toutes les attaches doivent être complètes (réservé en mémoire) ;
- chaque tampon doit avoir le même nombre d'échantillons.

Ne vous inquiétez pas, vous ne devez pas savoir ce que sont les échantillons. Nous verrons cela dans un **prochain tutoriel**.

À partir des prérequis, cela devient évident que nous devons créer une sorte d'attache pour le tampon d'image et l'attacher à celle-ci. Après que nous ayons rempli tous les prérequis, nous pouvons vérifier que nous avons réellement réussi à compléter le tampon d'image en appelant la fonction `glCheckFramebufferStatus()` avec `GL_FRAMEBUFFER`. La fonction vérifie le tampon d'image actuellement lié et retourne une de **ces valeurs**. Si c'est `GL_FRAMEBUFFER_COMPLETE`, alors nous l'avons bien configuré :

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) == GL_FRAMEBUFFER_COMPLETE)  
 // faire la danse de la victoire
```

Les opérations de rendu suivantes seront effectuées sur les attaches du tampon d'image lié et comme il n'est pas le tampon d'image par défaut, les commandes de rendu n'auront pas d'impact sur ce que vous verrez à l'écran. Le fait de faire un rendu qui n'est pas affiché à l'écran s'appelle un rendu hors écran (*off-screen rendering*). Pour s'assurer que toutes les opérations de rendu auront un impact visuel dans la fenêtre, il faut lier le tampon d'image par défaut, qui a l'identifiant 0 :

```
 glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Lorsque vous avez terminé avec vos opérations sur le tampon d'image, n'oubliez pas de supprimer l'objet :

```
glDeleteFramebuffers(1, &fbo);
```

Avant de vérifier la complétude du tampon d'image, nous devons attacher un ou plusieurs tampons au tampon d'image. Une attache est un emplacement mémoire qui peut agir comme un tampon pour le tampon d'image, telle une image. Lors de la création d'un attachement, nous avons deux options : des textures ou des **tampons de rendu** (render buffer).

## V-A-1 - Attache de type texture

Lorsque vous attachez une texture à un tampon d'image, toutes les commandes de rendu écriront dans la texture comme si c'était un tampon de couleurs, de profondeur ou de pochoir. L'avantage d'utiliser les textures est que le résultat de toutes les opérations de rendu seront stockées dans une image que nous pouvons réutiliser dans nos *shaders*.

Créer une texture pour un tampon d'image est globalement identique à la création d'une texture classique :

```
unsigned int texture;
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);

 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);

 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Les seules différences sont que nous définissons la dimension à celle de la taille de la fenêtre (même si ce n'est pas une nécessité) et nous passons `NULL` pour les données de la texture. Ainsi, nous ne faisons qu'allouer la mémoire. La définition de la texture s'effectuera lors du rendu dans le tampon d'image. Aussi, nous ne nous soucions pas de la méthode de dépassement (*wrapping*) ni du *mipmapping*, sachant que nous n'en avons pas besoin dans la plupart des cas.

 Si vous souhaitez afficher l'intégralité de votre écran dans une texture plus petite ou plus grande, vous devez appeler `glViewport()` (avant le rendu dans votre tampon d'image) avec les nouvelles dimensions de votre texture, sinon, seule une partie de la texture ou de l'écran sera dessinée dans la texture.

Maintenant que vous avez créé une texture, il ne reste plus qu'à l'attacher au tampon d'image :

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
```

La fonction `glFrameBufferTexture2D()` possède les paramètres suivants :

- **target** : le type de tampon d'image que nous utilisons (rendu, lecture ou les deux) ;
- **attachment** : le type d'attache que nous allons utiliser. Actuellement, nous attachons un tampon de couleurs. Notez que le 0 à la fin suggère que nous pouvons attacher plus d'un tampon de couleurs. Nous explorerons ceci dans un prochain tutoriel ;
- **textarget** : le type de texture que nous souhaitons attacher ;
- **texture** : la texture à attacher ;
- **level** : le niveau de *mipmap*. Nous le gardons à 0.

En plus de l'attache pour les couleurs, nous pouvons attacher une texture pour la profondeur ou le pochoir au tampon d'image. Pour attacher une texture pour la profondeur, nous devons spécifier le type `GL_DEPTH_ATTACHMENT`. Notez que le format de la texture et le **format interne** (*internalformat*) de la texture doivent devenir `GL_DEPTH_COMPONENT` pour correspondre au format de stockage du tampon de profondeur. Pour attacher

un tampon de pochoir, vous devez utiliser `GL_STENCIL_ATTACHMENT` comme deuxième argument et spécifier `GL_STENCIL_INDEX` comme format de texture.

Il est aussi possible d'attacher un tampon de profondeur et de pochoir à une unique texture. Chaque valeur sur 32 bits de la texture consistera en deux informations : 24 bits pour la profondeur et 8 bits pour le pochoir. Pour attacher une texture de ce type, nous devons utiliser le type `GL_DEPTH_STENCIL_ATTACHMENT` et configurer les formats de la texture pour contenir les deux valeurs. En voici un exemple :

```
glTexImage2D( GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, 800, 600, 0,
    GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL
);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_TEXTURE_2D, texture, 0);
```

## V-A-2 - Attache de type tampon de rendu

Les tampons de rendu ont été ajoutés dans OpenGL peu après les textures. À l'origine, seules les textures pouvaient être utilisées. Tout comme pour une texture, un tampon de rendu est un tampon, c'est-à-dire, un tableau d'octets, d'entiers, de pixels ou autre. Un tampon de rendu possède l'avantage d'utiliser le format natif d'OpenGL pour le stockage des données. Celui-ci est optimisé pour le rendu hors écran dans un tampon d'image.

Les tampons de rendu stockent les données de rendu directement dans leur tampon sans effectuer de conversion spécifique au format de la texture et sont donc plus rapides. Toutefois, les tampons de rendu ne sont généralement accessible qu'en écriture. Vous ne pouvez donc pas les lire dans un *shader* (contrairement aux textures). Il est possible de les lire avec `glReadPixels()`, qui retourne une zone provenant du tampon d'image actuellement lié, mais il n'existe pas de méthode pour obtenir les pixels de l'attache directement.

Comme leurs données sont déjà au format natif, ils sont plutôt rapides à l'écriture et lors de la copie dans d'autres tampons. Les opérations telles que le changement des tampons est donc rapide lors de l'utilisation d'un tampon de rendu. La fonction `glfwSwapBuffers()` que nous avons utilisée à la fin de chaque itération peut aussi être implémentée avec les objets de rendu : nous écrivons simplement dans le tampon de rendu, puis l'échangeons avec un autre à la fin. Les tampons de rendu sont parfaits pour ce type d'opération.

La création d'un tampon de rendu ressemble à la création d'un tampon d'image :

```
unsigned int rbo;
 glGenRenderbuffers(1, &rbo);
```

De même, nous voulons lier le tampon de rendu afin que les opérations suivantes l'affectent :

```
 glBindRenderbuffer(GL_RENDERBUFFER, rbo);
```

Comme les tampons de rendu sont généralement des tampons en écriture seule, nous les utilisons souvent comme attaches pour la profondeur et le pochoir. Habituellement, nous n'avons pas besoin de lire les valeurs de profondeur et de pochoir, juste de les tester lors du rendu. Lorsqu'il n'y a pas besoin d'échantillonner les tampons, les tampons de rendu sont préférables, car plus optimisés.

Créer un tampon de rendu pour la profondeur et le pochoir s'effectue ainsi :

```
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600);
```

Créer un tampon de rendu est similaire à la création des textures, la différence étant que l'objet est conçu pour être utilisé comme une image et non pour des données génériques. Ici, nous avons choisi le format interne `GL_DEPTH24_STENCIL8`, permettant de contenir les 24 bits de profondeur et les 8 bits de pochoir.

La dernière chose à faire est d'attacher le tampon de rendu :

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

Les tampons de rendu peuvent apporter des optimisations à vos tampons d'image, mais il est important de comprendre quand utiliser un tampon de rendu et quand utiliser une texture. La règle générale est que, si vous n'avez pas besoin d'échantillonner un tampon, alors il est préférable d'utiliser un tampon de rendu. Si vous devez échantillonner des données d'un tampon, telles des couleurs ou les valeurs de profondeur, vous devez utiliser une texture. Du côté des performances, l'impact n'est pas énorme.

## V-B - Rendu dans une texture

Maintenant que nous savons comment (en quelque sorte) les tampons d'image fonctionnent, nous pouvons les utiliser. Nous allons afficher la scène dans une texture de couleur attachée à notre tampon d'image et dessiner cette texture dans un rectangle couvrant l'intégralité de l'écran. Le résultat visuel sera exactement identique à celui sans tampon d'image, mais affiché sur un simple rectangle. Pourquoi est-ce donc utile ? Nous allons voir cela dans la prochaine section.

Premièrement, nous devons créer le tampon d'image et le lier :

```
unsigned int framebuffer;
 glGenFramebuffers(1, &framebuffer);
 glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

Ensuite, nous créons une texture que nous attachons comme tampon de couleurs au tampon d'image. Nous définissons les dimensions de la texture aux dimensions de la fenêtre et nous gardons ses données non initialisées :

```
// génère la texture
unsigned int texColorBuffer;
 glGenTextures(1, &texColorBuffer);
 glBindTexture(GL_TEXTURE_2D, texColorBuffer);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
 glBindTexture(GL_TEXTURE_2D, 0);

// attache la texture au tampon d'image
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texColorBuffer, 0);
```

Nous souhaitons aussi qu'OpenGL fasse ses tests de profondeur (et, si vous le voulez, les tests de pochoir), nous devons donc ajouter un tampon de profondeur (et de pochoir) au tampon d'image. Comme nous n'échantillonnons que le tampon des couleurs, nous pouvons créer un tampon de rendu pour cela. Vous rappelez-vous que c'est un bon choix lorsque vous n'échantillonnez pas ces tampons ?

Créer un tampon de rendu n'est pas très difficile. La seule chose à se rappeler est que nous le créons pour avoir un tampon de profondeur et de pochoir. Nous définissons son format interne à `GL_DEPTH24_STENCIL8` pour avoir assez de précision pour nos besoins.

```
unsigned int rbo;
 glGenRenderbuffers(1, &rbo);
 glBindRenderbuffer(GL_RENDERBUFFER, rbo);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600);
 glBindRenderbuffer(GL_RENDERBUFFER, 0);
```

Une fois que nous avons alloué assez de mémoire pour le tampon de rendu, nous pouvons le délier.

Ensuite, avant de pouvoir compléter le tampon d'image, nous devons attacher le tampon de rendu au tampon d'image :

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

Puis, par précaution, nous allons vérifier si le tampon d'image est complet, sinon nous affichons un message d'erreur.

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
 glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Finalement, assurez-vous de délier le tampon d'image afin de ne pas faire de rendu dedans par accident.

Maintenant que le tampon d'image est complet, il ne nous reste plus qu'à faire le rendu dans ce tampon d'image au lieu de celui par défaut, en le liant comme le tampon d'image. Toutes les commandes de rendu qui suivent vont influer sur le tampon d'image lié. Toutes les opérations sur la profondeur et le pochoir seront effectués sur les tampons de profondeur et de pochoir attachés au tampon d'image (si disponible). Si vous n'avez pas attaché de tampon de profondeur, les tests de profondeur ne fonctionneront pas, car il n'y a pas de tampon de profondeur dans le tampon d'image.

Donc, pour dessiner la scène dans une texture, nous devons suivre les étapes suivantes :

- dessiner la scène comme d'habitude avec le nouveau tampon d'image lié ;
- lier le tampon d'image par défaut ;
- dessiner un rectangle qui couvre l'écran et utiliser le tampon de couleurs du nouveau tampon d'image comme texture.

Nous allons dessiner la même scène que celle du tutoriel sur le **test de profondeur**, mais, cette fois, avec la **texture du conteneur**.

Pour dessiner le rectangle, nous allons créer un nouvel ensemble de *shaders*. Nous n'allons pas inclure de transformations de matrice, car nous allons juste fournir les **coordonnées de sommets dans l'espace de coordonnées de l'écran**. Ainsi, nous pouvons directement les utiliser comme sortie du *vertex shader*. Le *vertex shader* sera donc :

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    TexCoords = aTexCoords;
}
```

Rien de bien compliqué. Le *fragment shader* sera encore plus simple, il nous suffit juste d'échantillonner la texture :

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D screenTexture;

void main()
{
    FragColor = texture(screenTexture, TexCoords);
```

Il ne reste plus qu'à créer et configurer le VAO pour le rectangle. Une itération de rendu avec le tampon d'image aura la structure suivante :

```

// première passe
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // nous n'utilisons pas de tampon de profondeur
 glEnable(GL_DEPTH_TEST);
DrawScene();

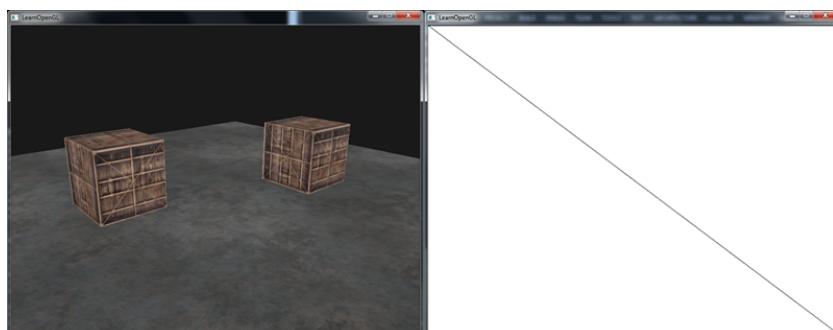
// deuxième passe
glBindFramebuffer(GL_FRAMEBUFFER, 0); // retour au framebuffer par défaut
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

screenShader.use();
 glBindVertexArray(quadVAO);
 glDisable(GL_DEPTH_TEST);
 glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
 glDrawArrays(GL_TRIANGLES, 0, 6);

```

Il y a quelques détails à prendre en compte. Premièrement, comme chaque tampon d'image possède ses propres tampons, nous souhaitons nettoyer ces tampons avec les options appropriées de la fonction `glClear()`. Deuxièmement, lorsque nous dessinons le rectangle, nous désactivons le test de profondeur, car nous n'en avons pas besoin pour un simple rectangle. Nous avons effectué les tests de profondeur lors du rendu classique de la scène.

Il y a plusieurs étapes qui peuvent être en défaut : si vous n'avez pas de résultat, essayez de déboguer là où cela est possible en relisant les sections appropriées du tutoriel. Si tout fonctionne, vous devriez obtenir :



À gauche vous avez exactement le résultat du tutoriel sur le **test de profondeur**, mais cette fois affiché dans un rectangle. Si nous dessinons la scène en mode fil de fer, il devient évident que ce n'est qu'un simple rectangle qui est affiché sur le tampon d'image par défaut.

Vous pouvez trouver le code source de cet exemple [Source ici](#).

Quelle pouvait bien être l'utilité de cela ? Eh bien, maintenant, nous avons un accès libre à tous les pixels de la scène à partir d'une texture : nous pouvons créer des effets intéressants à l'aide du *fragment shader*. La combinaison de ces effets se nomment des effets de post-traitement.

## V-C - Post-traitement

Maintenant que la scène est dessinée dans une texture, nous pouvons créer des effets intéressants en manipulant les données de la texture. Dans cette section, nous allons voir comment créer les effets les plus populaires et vous pourrez créer les vôtres avec un peu d'imagination.

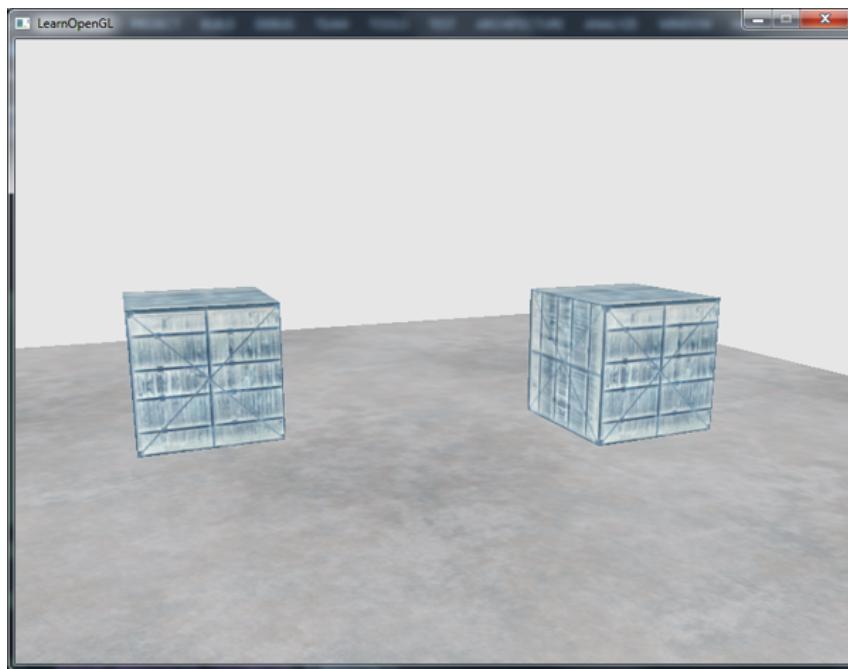
Commençons par l'effet le plus simple.

## V-C-1 - Inversion

Comme nous avons accès aux couleurs du rendu, il devient simple d'inverser les couleurs dans le *fragment shader*. Nous prenons donc la couleur de la texture et nous la soustrayons à 1.0 :

```
void main()
{
    FragColor = vec4(vec3(1.0 - texture(screenTexture, TexCoords)), 1.0);
}
```

Même si l'inversion est un effet très simple, il permet d'obtenir des résultats amusant :



La scène entière a maintenant ses couleurs inversées avec juste une ligne de code dans le *fragment shader*. Super, n'est-ce pas ?

## V-C-2 - Niveau de gris

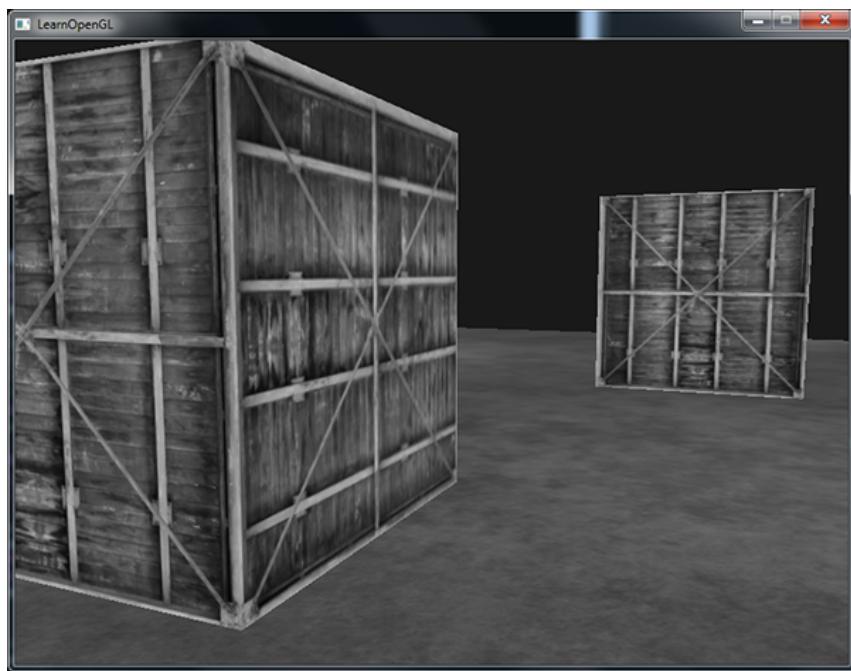
Un autre effet intéressant est de supprimer les couleurs à part le blanc, le gris et le noir afin d'avoir une image en nuances de gris. Une façon simple pour ce faire est de prendre chaque composante de la couleur et d'en faire la moyenne :

```
void main()
{
    FragColor = texture(screenTexture, TexCoords);
    float average = (FragColor.r + FragColor.g + FragColor.b) / 3.0;
    FragColor = vec4(average, average, average, 1.0);
}
```

Cela donne de bons résultats, mais l'œil humain est plus sensible aux teintes vertes et moins aux bleues. Pour obtenir un effet plus proche de la physique, nous devons ajouter des poids aux canaux de couleur :

```
void main()
{
    FragColor = texture(screenTexture, TexCoords);
    float average = 0.2126 * FragColor.r + 0.7152 * FragColor.g + 0.0722 * FragColor.b;
    FragColor = vec4(average, average, average, 1.0);
```

}



Vous pouvez ne pas voir la différence immédiatement, mais, avec des scènes plus complexes, l'ajout des poids permet d'avoir un résultat plus réaliste.

### V-C-3 - Effet de noyau

Un autre avantage d'effectuer un post-traitement sur une texture est que vous pouvez échantillonner les valeurs de couleurs des autres parties de la texture. Par exemple, nous pouvons obtenir les valeurs autour de la coordonnées de texture actuelle. Nous pouvons ainsi créer des effets intéressants en les combinant.

Un noyau (ou matrice de convolution) est une simple matrice de valeurs centrées sur le pixel actuel et qui permet de multiplier les pixels avec leur valeur correspondante dans le noyau puis d'en faire la somme afin d'obtenir une seule valeur finale. En résumé, en ajoutant un petit décalage aux coordonnées de texture du pixel actuel et ce dans toutes les directions, nous pouvons obtenir les valeurs de texture des alentours et les combiner grâce au noyau. Voici un exemple d'un noyau :

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & -15 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

Ce noyau prend les huit pixels alentour et les multiplie par 2 et le pixel courant par -15. Ce noyau permet donc d'ajouter un poids aux pixels à proximité et contre-balance le résultat en multipliant le pixel actuel par un poids négatif.

**i** La plupart des noyaux que vous trouverez sur Internet donne un résultat de 1, si vous additionnez tous les poids du noyau. Si cela ne donne pas 1, cela signifie que la couleur résultante sera plus lumineuse ou plus sombre que sa valeur d'origine.

Les noyaux sont très pratiques lors du post-traitement, car ils sont très faciles à utiliser et il y a beaucoup d'exemples en ligne. Nous devons légèrement adapter le *fragment shader* pour gérer les noyaux. Nous allons partir du principe que tous les noyaux ont une taille de 3x3 (comme la majorité) :

```

const float offset = 1.0 / 300.0;

void main()
{
    vec2 offsets[9] = vec2[] (
        vec2(-offset, offset), // haut gauche
        vec2( 0.0f, offset), // haut
        vec2( offset, offset), // haut droit
        vec2(-offset, 0.0f), // gauche
        vec2( 0.0f, 0.0f), // centre
        vec2( offset, 0.0f), // droit
        vec2(-offset, -offset), // bas gauche
        vec2( 0.0f, -offset), // bas
        vec2( offset, -offset) // bas droit
    );

    float kernel[9] = float[] (
        -1, -1, -1,
        -1,  9, -1,
        -1, -1, -1
    );
}

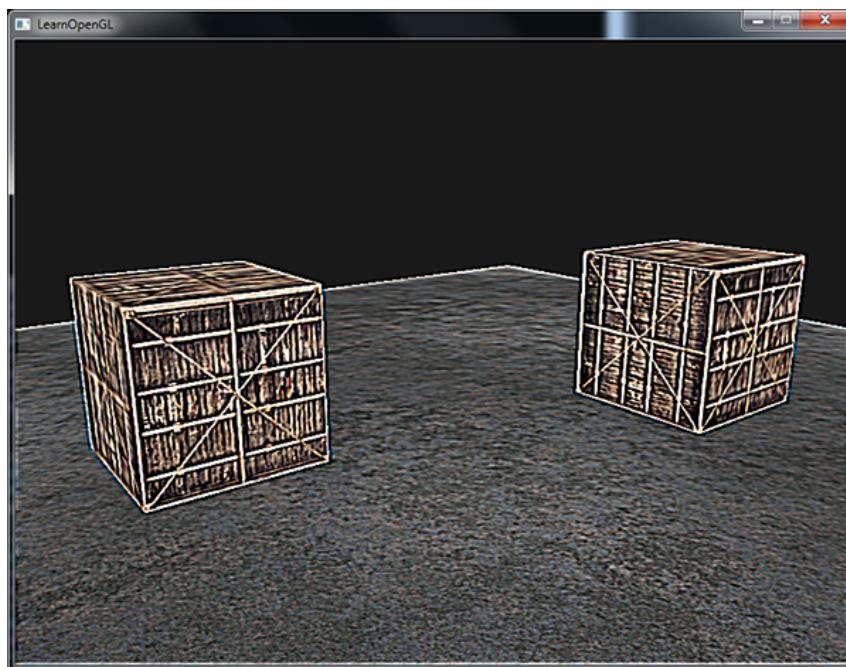
vec3 sampleTex[9];
for(int i = 0; i < 9; i++)
{
    sampleTex[i] = vec3(texture(screenTexture, TexCoords.st + offsets[i]));
}
vec3 col = vec3(0.0);
for(int i = 0; i < 9; i++)
    col += sampleTex[i] * kernel[i];

FragColor = vec4(col, 1.0);
}

```

Dans le *fragment shader*, nous créons un tableau de neuf vec2 pour le décalage de la coordonnées de texture afin d'obtenir les pixels autour du pixel actuel. Le décalage est simplement une valeur constante que vous pouvez modifier comme vous le souhaitez. Ensuite, nous définissons le noyau, qui dans ce cas permet d'exacerber chaque couleur en prenant en compte les pixels avoisinants. Finalement, nous ajoutons le décalage aux coordonnées de texture lors de l'échantillonnage, puis multiplions les valeurs de la texture avec celle du noyau et en effectuons la somme.

Ce noyau donne ce résultat :



Cela peut donner des effets intéressants, notamment si le joueur suit une aventure narcotique.

### V-C-3-a - Flou

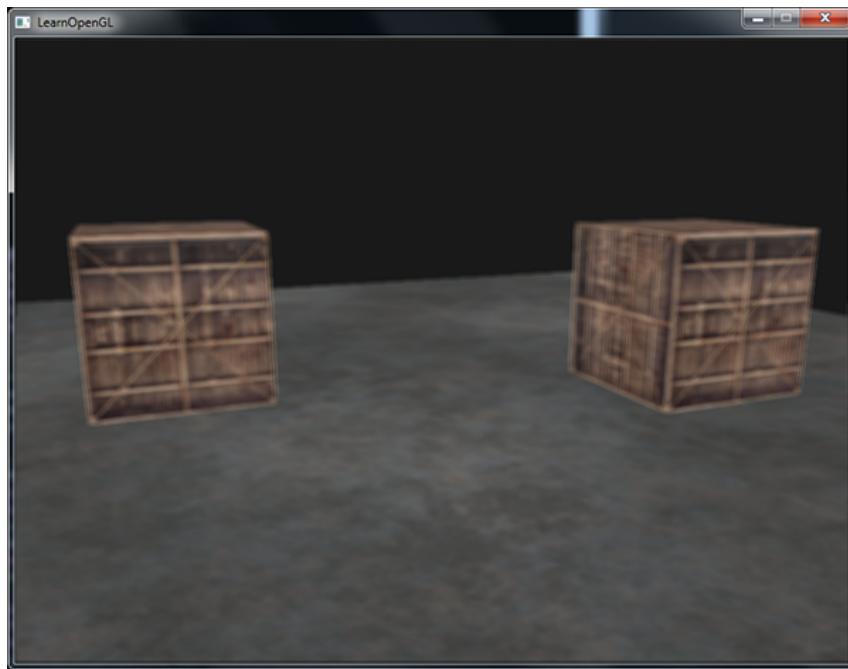
Un noyau permettant de flouter l'image est défini ainsi :

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} / 16$$

Comme la somme donne un résultat de 16, l'utilisation directe du résultat de l'application du noyau va donner une couleur trop claire et nous devons donc diviser par 16. Le noyau devient donc :

```
float kernel[9] = float[] {
    1.0 / 16, 2.0 / 16, 1.0 / 16,
    2.0 / 16, 4.0 / 16, 2.0 / 16,
    1.0 / 16, 2.0 / 16, 1.0 / 16
};
```

En changeant les valeurs du noyau, nous changeons complètement l'effet de post-traitement. Ainsi, on obtient :



Un tel effet crée des possibilités intéressantes. Il est possible de varier le floutage suivant le temps pour créer l'effet d'un personnage saoul ou lorsqu'il ne porte pas ses lunettes. Le floutage permet aussi d'adoucir les valeurs des couleurs, comme nous en aurons besoin dans les prochains tutoriels.

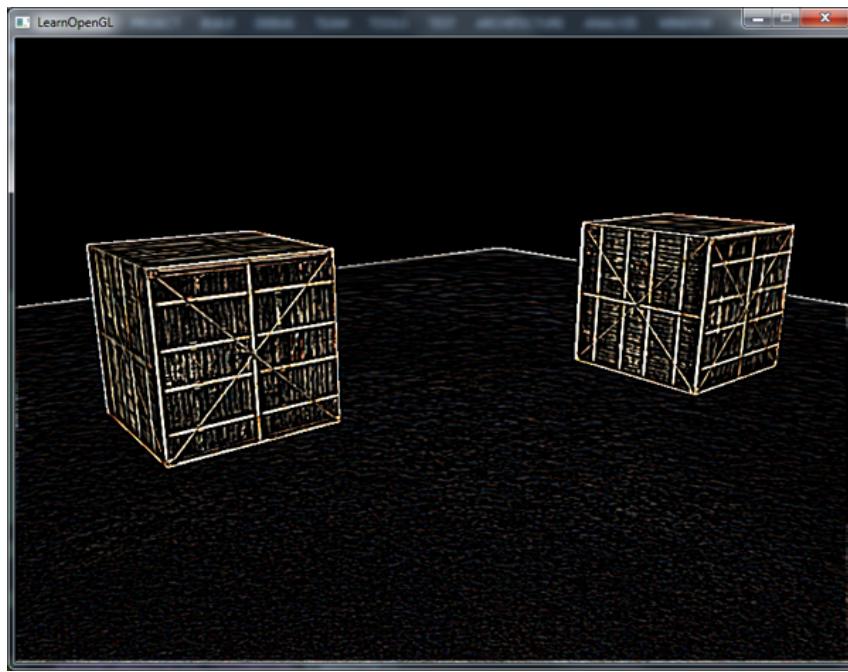
Vous pouvez voir que, avec une simple implémentation d'un noyau de convolution, il est possible d'obtenir des effets intéressants. Voyons voir un effet moins populaire pour finir.

### V-C-3-b - Détection des bordures

Ci-dessous, vous pouvez trouver un noyau permettant de détecter les bordures, similaire à celui pour exacerber les couleurs :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Le noyau met en avant les bordures et assombrit le reste, ce qui est très utile si nous voulons ne voir que les bordures d'une image.



Ce n'est pas une surprise si ce genre de noyau est utilisé dans les outils pour manipuler les images telles que Photoshop. Comme la carte graphique permet de traiter les fragments avec un haut niveau de parallélisme, nous pouvons manipuler les images pixel par pixel en temps réel avec une grande facilité. Les outils d'édition d'images utilisent de plus en plus les cartes graphiques pour les traitements.

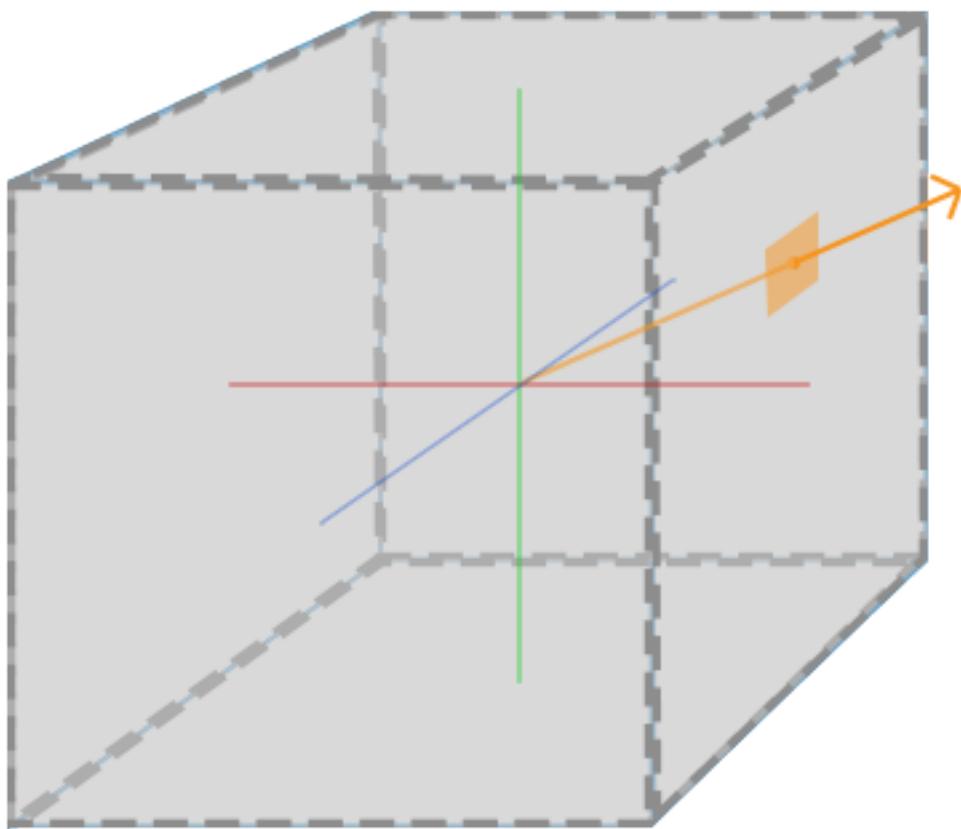
## V-D - Remerciements

Ce tutoriel est une traduction réalisée par **Alexandre Laurent** dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

## VI - Texture cubique (cubemap)

Nous utilisons les textures 2D depuis quelque temps déjà, mais il existe d'autres types de texture que nous n'avons pas encore vu. Ce tutoriel explore un de ces autres types, qui est en réalité une combinaison de plusieurs textures assemblées en une : une texture cubique (*cubemap*).

Une texture cubique est une texture qui contient un ensemble de six textures 2D pouvant être appliquée sur chaque face du cube : un cube texturé. Vous pouvez vous demander à quoi cela sert : en effet, pourquoi s'ennuyer à assembler six textures en une seule entité alors qu'il est possible d'utiliser six textures séparément ? Eh bien, les textures de cube ont la particularité d'être indexables/échantillonnable grâce à un vecteur direction. Imaginez que nous avons une texture cubique de  $1x1x1$  avec comme origine un vecteur direction en son centre. L'échantillonnage d'une valeur de la texture cubique avec le vecteur orange donnera ceci :



**i** La magnitude du vecteur direction n'est pas importante. Tant que la direction est fournie, OpenGL récupère le texel que le vecteur direction atteint (in fine) et retourne la valeur de la texture appropriée.

Si nous imaginons un cube et que nous y attachons une texture cubique, le vecteur direction qui échantillonnera la texture sera similaire à une position (interpolée) sur le cube. De cette façon, nous pouvons échantillonner la texture cubique à l'aide de la position dans le cube tant que celui-ci est centré en son origine. Nous pouvons obtenir les coordonnées de texture de tous les sommets juste en utilisant la position du sommet du cube. Le résultat est une coordonnée de texture qui accède à la face correspondante dans la texture cubique.

## VI-A - Créer une texture cubique

Une texture cubique est une texture comme toute autre, qu'il suffit de générer et de lier à la cible correspondante avant de faire plus d'opérations. Cette fois, nous lions au type `GL_TEXTURE_CUBE_MAP` :

```
unsigned int textureID;
 glGenTextures(1, &textureID);
 glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

Comme la texture cubique est composée de six textures, une par face, nous devons appeler six fois la fonction `glTexImage2D()` avec des paramètres similaires à ceux vus dans les précédents tutoriels. Cette fois, nous devons définir le paramètre de cible de la texture pour définir la face de la texture cubique. Cela indique à OpenGL sur quelle face nous souhaitons la texture que nous créons. Cela signifie que nous devons appeler `glTexImage2D()` pour chaque face de la texture cubique.

Comme nous avons six faces, OpenGL nous fournit six cibles :

| Cible de texture               | Orientation |
|--------------------------------|-------------|
| GL_TEXTURE_CUBE_MAP_POSITIVE_X | Droite      |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_X | Gauche      |
| GL_TEXTURE_CUBE_MAP_POSITIVE_Y | Haut        |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Y | Bas         |
| GL_TEXTURE_CUBE_MAP_POSITIVE_Z | Arrière     |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Z | Avant       |

Comme pour la majorité des énumérations OpenGL, c'est un `int` dont toutes les valeurs se suivent. Si nous devions avoir un tableau pour contenir les emplacements des textures, nous pourrions le parcourir en commençant par `GL_TEXTURE_CUBE_MAP_POSITIVE_X` et incrémenter l'index de 1 afin de boucler sur toutes les cibles de textures.

```
int width, height, nrChannels;
unsigned char *data;
for(GLuint i = 0; i < textures_faces.size(); i++)
{
    data = stbi_load(textures_faces[i].c_str(), &width, &height, &nrChannels, 0);
    glTexImage2D(
        GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
        0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
    );
}
```

Ici, nous avons un `std::vector` appelé `textures_faces` qui contient les emplacements de toutes les textures nécessaires pour la texture cubique. Cela génère une texture pour chaque face de la texture cubique liée.

Comme la texture cubique est une texture comme toute autre, nous devons aussi spécifier ses méthodes de filtrage :

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Ne soyez pas impressionnés par le `GL_TEXTURE_WRAP_R`. Il définit simplement la méthode de dépassement pour la coordonnée R qui correspond à la troisième dimension de la texture (comme le z pour la position). Nous définissons la méthode de dépassement à `GL_CLAMP_TO_EDGE` comme les coordonnées de texture entre deux faces ne vont pas toujours atteindre une même face (à cause des limitations matérielles). Donc, en utilisant `GL_CLAMP_TO_EDGE`, OpenGL retournera toujours les valeurs de bordure lorsque nous échantillonnerons entre deux faces.

Avant de dessiner des objets utilisant notre texture cubique, nous devons activer l'unité de texture correspondante et lier la texture cubique. En réalité, c'est comme pour les textures 2D.

Dans le *fragment shader*, nous devons aussi utiliser un nouveau type d'échantillonneur `samplerCube` que nous utilisons conjointement avec la fonction `texture()`. Évidemment, les coordonnées seront dans un vecteur de type `vec3` au lieu de `vec2`. Voici un exemple utilisant une texture cubique :

```
in vec3 textureDir; // vecteur direction représentant une coordonnée de texture 3D
uniform samplerCube cubemap; // échantillonneur pour une texture cubique

void main()
{
    FragColor = texture(cubemap, textureDir);
}
```

C'est bien, mais pourquoi s'en soucier ? Eh bien, il existe quelques techniques intéressantes qui sont plus faciles à implémenter avec un texture cubique. Une de ces techniques est la création d'une *skybox*.

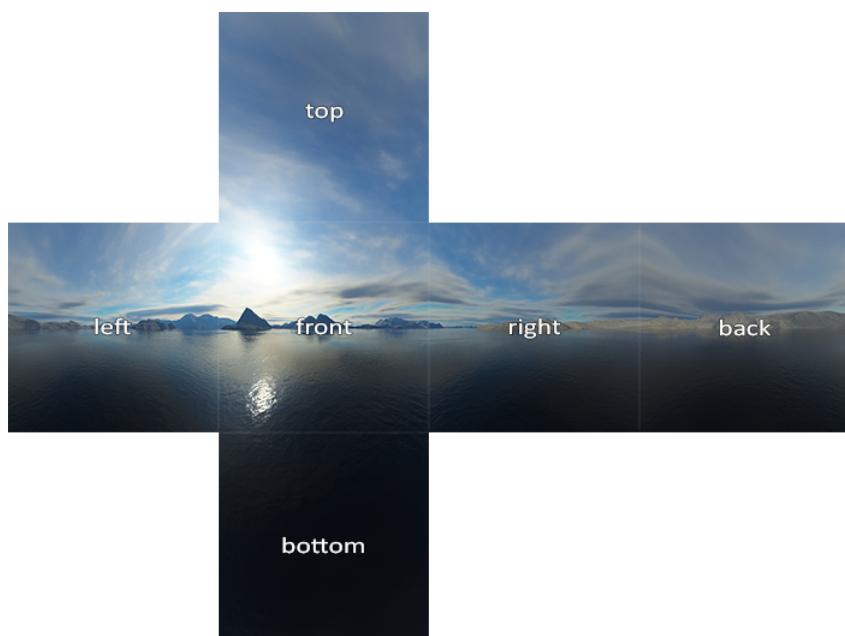
## VI-B - Skybox

Une *skybox* est un (grand) cube qui englobe l'intégralité de la scène et contient six images de l'environnement l'englobant. Ainsi, le joueur a l'illusion d'être dans un environnement bien plus grand que ce qu'il est vraiment. Certains exemples de *skybox* de jeux vidéos possèdent des montagnes, des nuages ou un ciel étoilé. Dans la capture suivante du troisième opus d'Elder Scrolls, vous pouvez y voir une *skybox* de ciel étoilé :



Vous le savez sûrement déjà, les textures cubiques sont la solution parfaite pour les *skybox* : nous avons un cube avec six faces qui a besoin d'une texture par face. Dans l'image précédente, les développeurs ont utilisé plusieurs images de ciel étoilé de nuit pour donner au joueur l'impression qu'il y a un univers bien plus grand que cette simple petite boîte.

Il est facile de trouver des *skybox* en ligne. Ce [site](#), par exemple, a un grand ensemble de *skybox*. Les images de *skybox* sont généralement conçues ainsi :



Si vous pliez ces six carrés, vous obtenez un cube complètement texturé simulant un grand paysage. Certains sites fournissent les *skybox* dans une image comme celle-ci et vous devez extraire manuellement les six faces, mais dans la majorité des cas, vous aurez six images.

Cette *skybox* (de haute qualité) est celle que nous allons utiliser dans notre scène et peut être téléchargée  [ici](#).

## VI-B-1 - Chargement d'une skybox

Comme la *skybox* n'est qu'une texture cubique, le chargement ne diffère pas de ce que nous avons vu précédemment. Pour charger la *skybox*, nous allons utiliser la fonction suivante qui accepte un `std::vector` des six emplacements de texture :

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &nChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                         0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
                         );
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

    return textureID;
}
```

La fonction ne doit pas être très surprenante. C'est grossièrement le code de chargement d'une texture cubique que nous avons vu dans la section précédente, mais intégré dans une unique fonction.

Avant d'appeler cette fonction, nous devons charger les chemins des textures dans le `std::vector` dans l'ordre de l'énumération OpenGL :

```
vector<std::string> faces;
{
    "right.jpg",
    "left.jpg",
    "top.jpg",
    "bottom.jpg",
    "front.jpg",
    "back.jpg"
};
unsigned int cubemapTexture = loadCubemap(faces);
```

Nous allons maintenant charger la *skybox* dans une texture cubique avec la fonction `cubemapTexture()`. Nous pouvons lier la texture à un cube et enfin remplacer la couleur de fond que nous utilisions depuis tout ce temps.

## VI-B-2 - Afficher une skybox

Comme la *skybox* est affichée sur un cube, nous devons créer d'autres VAO et VBO, ainsi qu'un nouvel ensemble de sommets. Vous pouvez obtenir les données des sommets [ici](#).

Utiliser une texture cubique pour texturer un cube 3D peut s'effectuer en utilisant la position du cube comme coordonnées de texture. Lorsque le cube est centré en (0, 0, 0), chaque point est aussi une direction depuis son origine. La direction est exactement ce dont nous avons besoin pour obtenir les valeurs correspondantes de la texture en cette position du cube. Pour cette raison, nous ne fournissons que les positions et nous ne fournissons pas de coordonnées de texture.

Pour afficher la *skybox*, nous allons avoir besoin d'un nouvel ensemble de *shaders* qui ne sont pas très compliqués. Comme nous n'avons que la position des sommets comme attribut, c'est plutôt simple :

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    gl_Position = projection * view * vec4(aPos, 1.0);
}
```

La partie intéressante du *shader* est que les positions en entrée sont copiées en sortie pour devenir des coordonnées de texture pour le *fragment shader*. Ce dernier les utilise pour échantillonner un *samplerCube*:

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
```

Le *fragment shader* est évident. Nous prenons la position des sommets comme vecteur de direction pour la texture et les utilisons pour échantillonner les valeurs de la texture cubique.

Le rendu de la *skybox* est simple maintenant que nous avons la texture cubique. Nous lions simplement la texture et l'échantillonneur *skybox* est automatiquement connecté avec la texture cubique. Pour dessiner la *skybox*, nous allons la dessiner en premier et en désactivant les tests de profondeur. De cette façon, nous sommes certains que la *skybox* sera toujours au fond, derrière tous les autres objets.

```
glDepthMask(GL_FALSE);
skyboxShader.use();
// ... définir la matrice de vue et projection
glBindVertexArray(skyboxVAO);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glDepthMask(GL_TRUE);
// ... dessiner le reste de la scène
```

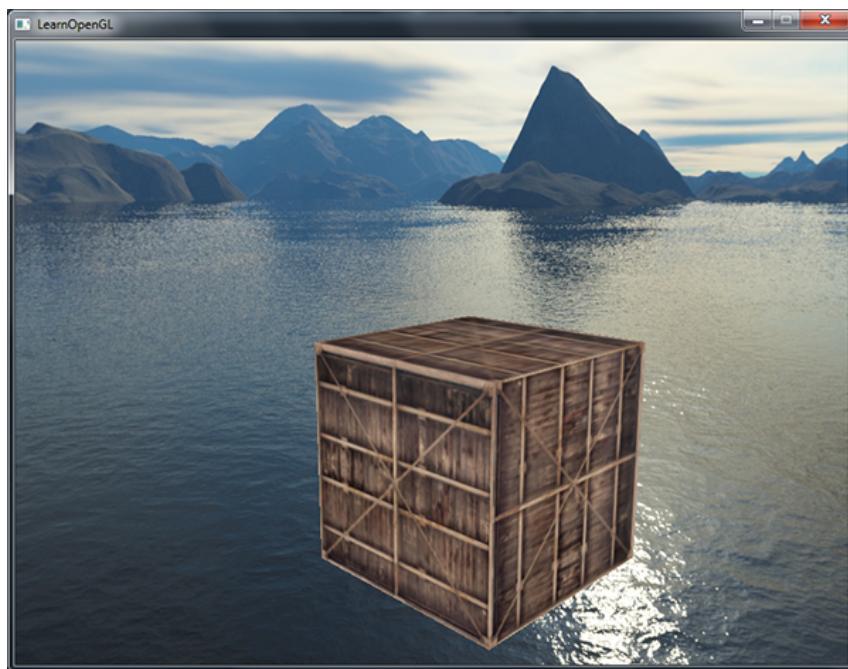
Si vous exécutez ce code en l'état, vous allez avoir des problèmes. Nous souhaitons avoir une *skybox* centrée autour du joueur, peu importe ses déplacements. La *skybox* ne s'approchera pas du joueur, ce qui donnera l'impression d'un environnement immense. La matrice de vue actuelle transforme les sommets de la *skybox* en les tournant, en les redimensionnant et en les déplaçant. Par conséquent, si le joueur se déplace, la *skybox* se déplace aussi ! Nous souhaitons supprimer le déplacement de la matrice de vue afin que le mouvement n'affecte pas la position de la *skybox*

Souvenez-vous du tutoriel [d'éclairage basique](#), où nous apprenions que l'on peut supprimer la translation en ne gardant que la partie 3x3 supérieure gauche d'une matrice 4x4. Nous pouvons faire cela en convertissant la matrice de vue en une matrice 3x3 (supprimant ainsi la translation) et en la reconvertissant en matrice 4x4 :

```
glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
```

Cela enlève toute translation, tout en gardant les autres transformations, ce qui permet ainsi au joueur de regarder autour de lui.

Le résultat rend la scène immense grâce à la *skybox*. Si vous naviguez autour du conteneur, vous avez une notion de sa taille, ce qui augmente le réalisme. Le résultat doit être celui-ci :



Essayez d'expérimenter avec différentes *skybox* et de voir leur impact sur l'aspect de la scène.

### VI-B-3 - Optimisation

Pour le moment, nous avons dessiné la *skybox* avant tous les autres objets de la scène. Cela fonctionne très bien, mais ce n'est pas efficace. Si nous dessinons la *skybox* en premier, nous exécutons le *fragment shader* pour chaque pixel de l'écran, même si certaines parties de la *skybox* seront cachées. Les fragments qui auraient pu être ignorés pourraient nous aider à gagner en bande passante.

Pour obtenir un léger gain de performance, nous allons donc dessiner la *skybox* en dernier. De cette façon, le tampon de profondeur est complètement replié avec les valeurs de profondeur de chaque objet. Ainsi, nous ne dessinerons que les pixels de la *skybox* qui seront réellement visibles, c'est-à-dire, ceux qui passent le test de profondeur. Le problème est que la *skybox* risque de ne pas s'afficher, car ce n'est qu'un cube 1x1x1 qui échouera à tous les tests de profondeur. Son affichage sans test de profondeur ne résout pas le problème, car la *skybox* recouvrera tous les objets de la scène. Nous devons tromper le tampon de profondeur afin de faire croire que la *skybox* a la profondeur

maximale (soit 1,0), de telle sorte qu'elle échouera à tous les tests où la profondeur est différente, c'est-à-dire lorsqu'il y a un autre objet devant.

Dans le tutoriel sur les **systèmes de coordonnées**, nous avons expliqué que la division de perspective est exécutée après l'exécution du **vertex shader**. Elle divise les coordonnées xyz de `gl_Position` par la composante w. Nous savons aussi, grâce au **tutoriel sur le test de profondeur**, que la composante z résultant de la division est égale à la valeur de la profondeur du sommet. Grâce à cette information, nous pouvons définir la composante z de la position de sortie égale à la composante w. Ainsi, la division de w/z donne toujours 1,0, car la division devient  $w/w = 1,0$ .

```
void main()
{
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

Le résultat obtenu en coordonnées normalisées sera toujours une valeur z égale à 1, soit la valeur de profondeur maximale. La *skybox* ne sera dessinée qu'aux emplacements où il n'y a pas d'objet visible (seuls ceux qui passent le test de profondeur, le reste du monde est devant la *skybox*).

Nous devons changer la fonction du test de profondeur pour le définir à `GL_EQUAL` au lieu de `GL_LESS`. Le tampon de profondeur sera toujours rempli avec la valeur 1,0 pour la *skybox*, nous devons donc nous assurer que la *skybox* réussisse le test de profondeur pour les valeurs plus petites ou égales à celle dans le tampon de profondeur.

Vous pouvez trouver cette version optimisée [Source ici](#).

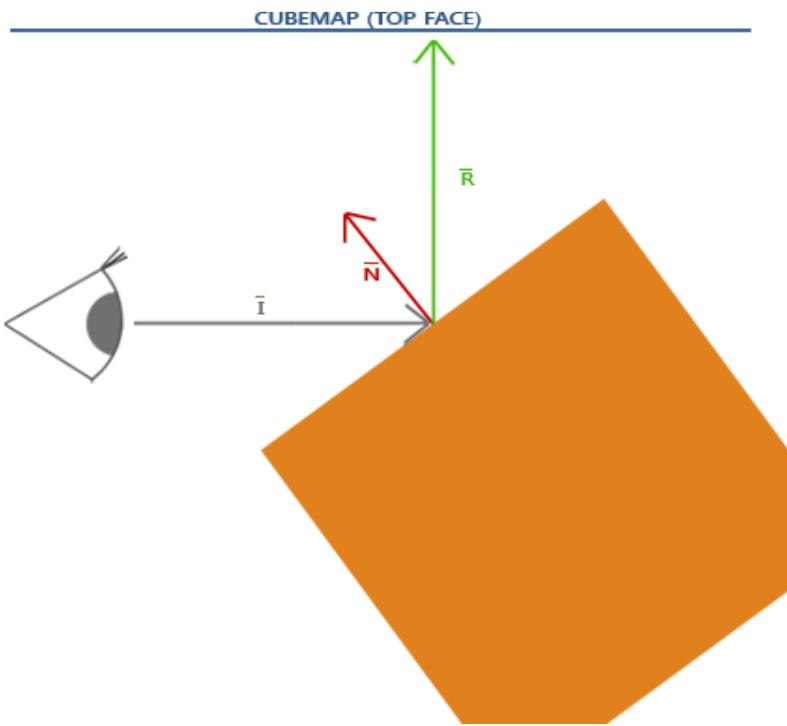
## VI-C - Application de l'environnement

Nous avons maintenant un environnement englobant grâce à l'application d'une simple texture. Nous pouvons aussi utiliser les textures cubiques pour d'autres effets. En particulier, elles nous permettent d'appliquer des propriétés de réflexion et de réfraction aux objets. Ces techniques qui utilisent une texture cubique d'environnement s'appellent « application de l'environnement » (*environment mapping*) et les deux plus courantes sont la réflexion et la réfraction.

### VI-C-1 - Réflexion

La propriété de réflexion fait qu'un objet (ou une partie d'un objet) reflète son environnement, c'est-à-dire, les couleurs de l'objet sont plus ou moins basées sur l'environnement l'englobant et l'angle du spectateur. Un miroir est un exemple d'objet reflétant : il reflète son environnement suivant la position du spectateur.

La réflexion de base n'est pas difficile. L'image suivante montre comment calculer un vecteur de réflexion et utilise ce vecteur pour échantillonner la texture cubique :



Nous calculons un vecteur réfléchissant  $\bar{R}$  grâce à la normale de l'objet  $\bar{N}$  et la direction du spectateur  $\bar{I}$ . Nous calculons le vecteur de réflexion grâce à la fonction GLSL `reflect()`. Le vecteur  $\bar{R}$  résultant est utilisé comme direction pour indexer/échantillonner la texture cubique et ainsi obtenir la couleur de l'environnement. L'effet final est que l'objet semble réfléchir la *skybox*.

Comme nous avons déjà configuré la *skybox* pour notre scène, la création des reflets n'est pas très compliquée. Nous changeons le *fragment shader* utilisé par le conteneur pour lui donner des propriétés réfléchissantes :

```
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 Position;

uniform vec3 cameraPos;
uniform samplerCube skybox;

void main()
{
    vec3 I = normalize(Position - cameraPos);
    vec3 R = reflect(I, normalize(Normal));
    FragColor = vec4(texture(skybox, R).rgb, 1.0);
}
```

D'abord, nous calculons le vecteur direction de la caméra  $I$ . Celui-ci permet d'obtenir le vecteur de réflexion  $R$ , qui est ensuite utilisé pour échantillonner la texture cubique de la *skybox*. Notez que les variables interpolées `Normal` et `Position` du *fragment shader* doivent être modifiées.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 Normal;
out vec3 Position;
```

```

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    Normal = mat3(transpose(inverse(model))) * aNormal;
    Position = vec3(model * vec4(aPos, 1.0));
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
    
```

Nous utilisons les normales, nous devons donc les transformer avec la matrice des normales. Le vecteur de sortie Position est un vecteur dans l'espace monde. Cette sortie est utilisée pour calculer la direction du spectateur dans le *fragment shader*.

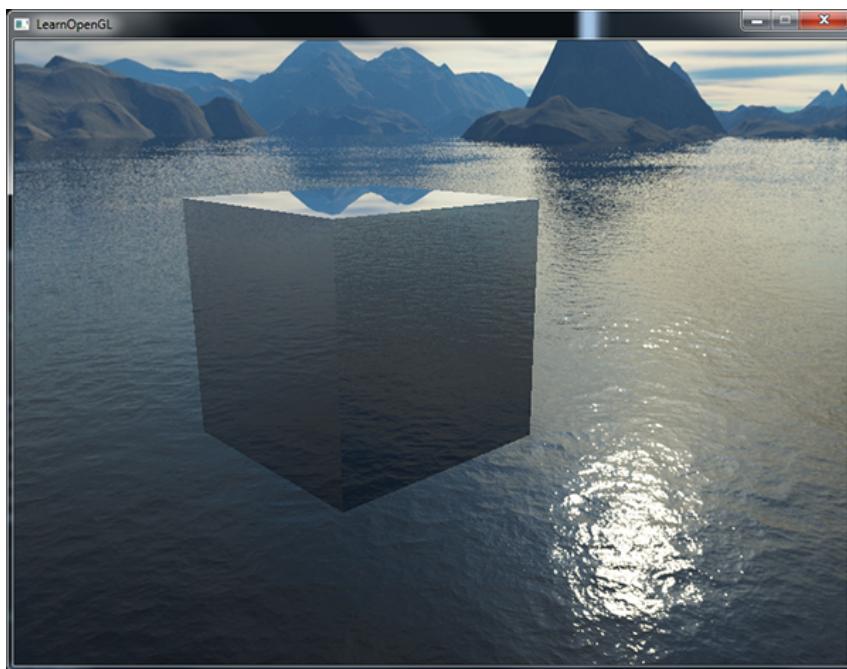
Comme nous utilisons les normales, nous souhaitons mettre à jour les **données des sommets** et les pointeurs d'attributs. Assurez-vous de définir la variable uniforme cameraPos.

Ensuite, nous souhaitons lier la texture cubique avant d'afficher le conteneur :

```

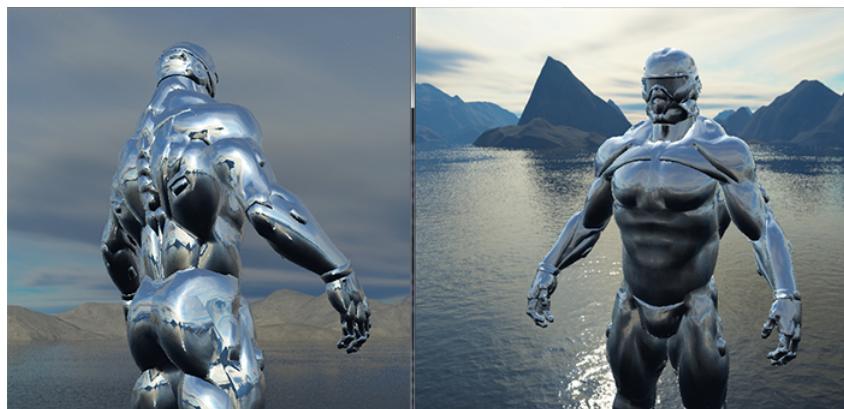
glBindVertexArray(cubeVAO);
glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
    
```

Compilez et exécutez votre code afin d'obtenir un conteneur réfléchissant comme un miroir. La *skybox* est parfaitement reflétée sur le conteneur :



Vous pouvez trouver le code source [Source ici](#).

Lorsque la réflexion est appliquée à l'intégralité d'un objet (comme le conteneur), l'objet semble être un matériau hautement réfléchissant comme de l'acier ou du chrome. Si nous chargeons le modèle *nanosuit* utilisé dans le [tutoriel sur le chargement d'objet 3D](#), nous aurions un effet faisant croire que la combinaison est en chrome :

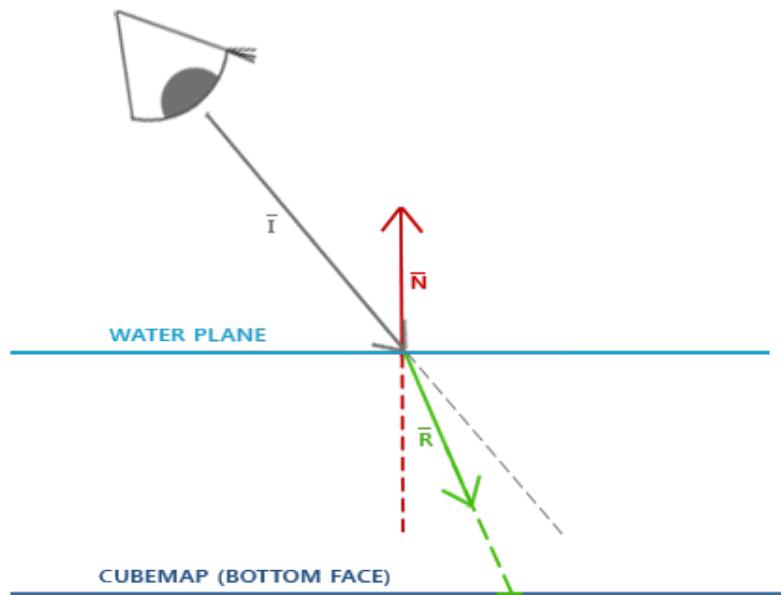


Cela est génial, mais, en réalité, la plupart des modèles ne sont pas complètement réfléchissants. Nous pouvons, par exemple, utiliser des textures de réflexion offrant des détails supplémentaires au modèle. Tout comme pour la texture diffuse et spéculaire, la texture de réflexion est une image qui est échantillonnée pour déterminer la propriété réfléchissante du fragment. Ainsi, nous pouvons déterminer quelle partie du modèle doit réfléchir l'environnement et avec quelle intensité. Dans les exercices de ce tutoriel, c'est à vous d'ajouter une texture de réflexion pour améliorer les détails de ce modèle

## VI-C-2 - Réfraction

Une autre technique d'application de l'environnement s'appelle la réfraction et s'apparente à la réflexion. La réfraction change la direction de la lumière, à cause du changement de matière traversée par celle-ci. La réfraction se voit généralement avec l'eau, où la lumière ne traverse pas le milieu aqueux en ligne droite. C'est comme regarder son bras lorsqu'il est à moitié dans l'eau.

La réfraction est décrite par la **loi de Snell**, qui ressemble à cela avec texture d'environnement :



Encore une fois, nous avons le vecteur du spectateur  $\bar{I}$ , et une normale  $\bar{N}$  qui permettent d'obtenir un vecteur de réfraction  $\bar{R}$ . Comme vous pouvez le voir, la direction du vecteur du spectateur est légèrement déviée. Le vecteur dévié  $\bar{R}$  est utilisé pour échantillonner la texture cubique.

La réfraction peut être facilement implémentée grâce à la fonction GLSL `refract()`, qui prend une normale, la direction du spectateur et un ratio entre les deux indices de réfraction des matériaux.

L'indice de réfraction est la quantité de lumière déviée par le matériau. Chaque matériau a son propre indice de réfraction. Voici une liste des indices les plus communs :

| Matériaux | Indice de réfraction |
|-----------|----------------------|
| Air       | 1,00                 |
| Eau       | 1,33                 |
| Glace     | 1,309                |
| Verre     | 1,52                 |
| Diamant   | 2,42                 |

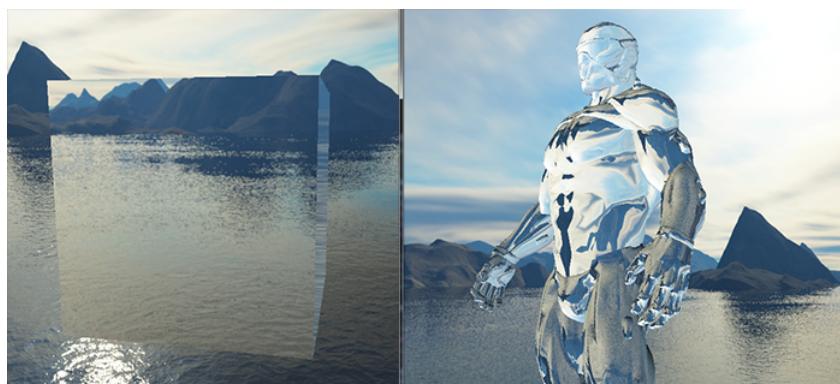
Nous utilisons ces indices pour calculer le ratio de lumière passant entre deux matériaux. Dans notre cas, la lumière/ rayon du spectateur passe de l'air au verre (si nous prenons un conteneur en verre). Le ratio est donc :

$$\frac{1.00}{1.52} = 0.658$$

Nous avons déjà la texture cubique liée, les données des sommets avec leurs normales et la définition de la caméra dans une variable uniforme. Il ne reste plus qu'à changer le *fragment shader* :

```
void main()
{
    float ratio = 1.00 / 1.52;
    vec3 I = normalize(Position - cameraPos);
    vec3 R = refract(I, normalize(Normal), ratio);
    FragColor = vec4(texture(skybox, R).rgb, 1.0);
}
```

En changeant l'indice de réfraction, vous pouvez créer des résultats très différents. La compilation et l'exécution de l'application n'a que peu d'intérêt, car nous n'utilisons qu'un simple conteneur. Il n'expose pas assez bien la réfraction qui n'agit que comme loupe. Pas contre, en utilisant les mêmes *shaders* et le modèle de *nanosuit*, nous obtenons bien un objet semblant être en verre.



Vous pouvez imaginer qu'avec la bonne combinaison de lumière, réflexion, réfraction ainsi qu'en déplaçant les sommets, vous pouvez créer un effet d'eau vraiment joli. Notez que, pour obtenir des résultats physiquement réalisistes, nous devrions refracter la lumière lorsqu'elle sort de l'objet. Pour le moment, nous utilisons juste une réfraction dans un seul sens, ce qui est suffisant pour la plupart des cas.

## VI-C-3 - Application de l'environnement dynamique

Jusqu'à présent, nous avons utilisé un ensemble d'images statiques pour la *skybox*. C'est bien, mais cela n'inclut pas les autres objets de la scène, qui peuvent se déplacer. Nous ne l'avons pas remarqué, car nous n'utilisons qu'un seul objet. Si nous avions plusieurs objets réfléchissant d'autres objets aux alentours, seule la *skybox* serait visible dans le miroir, comme si c'était l'unique objet de la scène.

Grâce au tampon d'image, il est possible de créer une texture de la scène avec les six différents angles aperçus de l'objet et de stocker cela dans une texture cubique à chaque itération. Nous pouvons ensuite l'utiliser (générée dynamiquement) pour créer des surfaces réfléchissantes et réfringentes qui incluent tous les objets. Cela s'appelle une texture d'environnement dynamique, car nous créons la texture cubique dynamiquement et nous l'utilisons comme texture d'environnement.

Même si c'est beau, cela a un gros désavantage : nous devons afficher la scène six fois par objet qui utilise une texture d'environnement. Les applications modernes essaient d'utiliser la *skybox* autant que possible et des textures cubiques pré-générées pour obtenir un effet similaire. Bien que les textures d'environnement dynamiques soient une bonne technique, cela nécessite beaucoup d'astuces et de *hacks* pour réussir à l'appliquer sans avoir de gros défaut de performances.

## VI-D - Exercices

- Essayez d'ajouter une texture de réflexion dans le chargement du modèle que nous avons créé au cours du [tutoriel sur le chargement](#). Vous pouvez trouver un modèle de *nanosuit* à jour, avec texture de réflexion,  [ici](#). Il y a quelques points à noter :
  - Assimp n'aime pas les textures de réflexion dans la plupart des formats de modèle 3D. Nous avons donc triché en l'implémentant comme une texture ambiaute. Vous pouvez charger la texture de réflexion en spécifiant `aiTextureType_AMBIENT` comme type de texture lors du chargement des matériaux ;
  - je me suis dépêché de créer les textures de réflexion à partir des textures spéculaires. Du coup, la texture de réflexion ne correspondra pas parfaitement au modèle :) ;
  - comme le chargement du modèle prend déjà trois unités de texture dans le *shader*, vous devez charger la *skybox* dans une quatrième unité, car vous devez échantillonner la *skybox* dans un même *shader* que les autres textures.
- Si vous avez réussi, cela devrait vous donner [ceci](#).

## VI-E - Remerciements

Ce tutoriel est une traduction réalisée par [Alexandre Laurent](#) dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

## VII - Données avancées

Depuis le début, nous avons intensément utilisé les tampons OpenGL pour stocker des données. Il existe d'autres façons de manipuler les tampons et d'autres méthodes pour transférer de nombreuses données aux *shaders*, notamment grâce aux textures. Dans ce tutoriel, nous allons voir quelques fonctions intéressantes à propos des tampons.

En OpenGL, un tampon n'est qu'un objet qui gère un morceau de mémoire et rien d'autre. Nous donnons un sens à ce tampon lorsque nous le lions à une cible. Un tampon n'est qu'un tableau de sommets lorsque vous le liez comme un `GL_ARRAY_BUFFER`, mais vous pouvez aussi le lier comme un `GL_ELEMENT_ARRAY_BUFFER`. En interne, OpenGL stocke un tampon par cible et, suivant la cible, traite le tampon différemment.

Jusqu'à présent, nous avons rempli la mémoire gérée par un objet tampon grâce à `glBufferData()` qui alloue un morceau de mémoire et ajoute ces données dans la mémoire. Si nous passons `NULL` comme argument pour les données, la fonction ne ferait que l'allocation de mémoire. C'est pratique si vous souhaitez d'abord réserver une certaine partie de la mémoire et la remplir plus tard, petit à petit.

Au lieu de remplir l'intégralité du tampon avec un seul appel, vous pouvez remplir des régions de ce tampon grâce à la fonction `glBufferSubData()`. Cette fonction attend une cible de tampon, un décalage, la taille et les données à insérer. Ainsi, avec cette fonction, il est possible de spécifier à partir de quel endroit nous souhaitons remplir le tampon. Cela nous permet d'insérer/mettre à jour certaines parties de la mémoire. Notez que le tampon doit être alloué avec assez de mémoire grâce à `glBufferData()` avant d'appeler la fonction `glBufferSubData()`.

```
glBufferSubData(GL_ARRAY_BUFFER, 24, sizeof(data), &data); // remplir la zone [24, 24 +  
sizeof(data)]
```

Vous pouvez aussi obtenir les données d'un tampon en demandant un pointeur sur sa mémoire et copier directement les données vers ce tampon vous-même. En appelant `glMapBuffer()`, OpenGL retourne un pointeur associé à la mémoire du tampon sur lequel vous pouvez travailler :

```
float data[] = {  
    0.5f, 1.0f, -0.35f  
    ...  
};  
glBindBuffer(GL_ARRAY_BUFFER, buffer);  
// obtenir le pointeur  
void *ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);  
// copier les données en mémoire  
memcpy(ptr, data, sizeof(data));  
// assurez-vous d'indiquer à OpenGL que vous avez terminé avec ce pointeur  
glUnmapBuffer(GL_ARRAY_BUFFER);
```

En indiquant à OpenGL que nos opérations sont finies avec `glUnmapBuffer()`, OpenGL peut reprendre possession de cette mémoire. Le pointeur devient invalide et la fonction retourne `GL_TRUE` si OpenGL a pu mettre vos données dans le tampon.

La fonction `glMapBuffer()` est pratique pour mettre vos données directement dans un tampon sans devoir les stocker temporairement. Ainsi, il est possible de lire vos données directement du disque pour les placer dans la mémoire du tampon.

## VII-A - Mise en lot des attributs de sommets

Grâce à la fonction `glVertexAttribPointer()` nous pouvons spécifier la disposition du contenu des tableaux de sommets. Dans celui-ci, nous intercalons les attributs, c'est-à-dire la position, la normale, la ou les coordonnées de texture pour chaque sommet. Maintenant que nous maîtrisons mieux les tampons, nous pouvons utiliser une autre approche.

Nous pouvons mettre en lot toutes les données dans un grand morceau de mémoire par type d'attribut au lieu de les intercaler. Ainsi, au lieu d'avoir un agencement 123123123123, nous aurions 111122223333.

Lors du chargement des données des sommets, nous récupérons un tableau de position, un tableau de normales et un ou plusieurs tableaux de coordonnées de texture. Cela peut être pénible de réarranger ces tableaux afin d'avoir des données agencées différemment. La mise en lot offre une solution plus simple à implémenter, grâce à la fonction `glBufferSubData()` :

```
float positions[] = { ... };  
float normals[] = { ... };  
float tex[] = { ... };  
// fill buffer  
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(positions), &positions);  
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions), sizeof(normals), &normals);  
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions) + sizeof(normals), sizeof(tex), &tex);
```

De cette façon, nous pouvons transférer directement les tableaux d'attributs en une seule fois sans avoir besoin de les traiter. Nous pouvons aussi remplir le tampon avec `glBufferData()`, mais la fonction `glBufferSubData()` correspond parfaitement à ce genre de tâches.

Nous devons aussi mettre à jour nos attributs de sommets ainsi :

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*) (sizeof(positions)));
glVertexAttribPointer(
    2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*) (sizeof(positions) + sizeof(normals)));
```

Remarquez que le paramètre `stride` est égal à la taille des attributs, car le prochain attribut se trouve directement après les 3 (ou 2) composantes.

Cela nous donne une nouvelle approche pour définir les attributs de sommets. L'utilisation d'une approche ou d'une autre n'a pas d'intérêt immédiat en OpenGL. C'est principalement une façon d'organiser les attributs de sommets. L'approche à utiliser repose uniquement sur vos préférences et le type d'application.

## VII-B - Copie des tampons

Une fois que vos tampons sont remplis avec vos données, vous pouvez souhaiter les partager avec d'autres tampons ou même copier un tampon dans un autre. La fonction `glCopyBufferSubData()` nous permet de copier les données d'un tampon vers un autre. Le prototype de la fonction est le suivant :

```
void glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr readoffset,
                        GLintptr writeoffset, GLsizeiptr size);
```

Les paramètres `readtarget` et `writetarget` définissent les tampons de source et de destination. Nous pouvons, par exemple, copier du tampon `VERTEX_ARRAY_BUFFER` vers un tampon `VERTEX_ELEMENT_ARRAY_BUFFER` en spécifiant ces cibles pour la lecture et l'écriture, respectivement. Les tampons actuellement liés à ces cibles seront alors affectés.

Comment faire si nous souhaitons lire et écrire dans deux tampons qui sont tous les deux des tableaux de sommets ? Nous ne pouvons pas lier deux tampons en même temps à la même cible. Pour cela, OpenGL nous donne deux autres cibles : `GL_COPY_READ_BUFFER` et `GL_COPY_WRITE_BUFFER`. Nous souhaitons alors lier les tampons de notre choix à ces deux nouvelles cibles en les passants en paramètre `readtarget` et `writetarget`.

`glCopyBufferSubData()` lit alors les `size` données à partir de `readoffset` pour les écrire dans le tampon indiqué par `writetargetbuffer` à partir de `writeoffset`. Voici un exemple montrant comment copier le contenu de deux tableaux de sommets :

```
float vertexData[] = { ... };
glBindBuffer(GL_COPY_READ_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0, sizeof(vertexData));
```

Nous pouvions aussi le faire en liant le tampon `writetarget` à l'un des nouveaux types :

```
float vertexData[] = { ... };
glBindBuffer(GL_ARRAY_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_ARRAY_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0, sizeof(vertexData));
```

Grâce à ces nouvelles connaissances sur la manipulation des tampons, nous sommes prêts à les utiliser de manière intéressante. Plus vous allez en profondeur dans OpenGL, plus ces nouvelles méthodes vont devenir utiles. Dans le prochain tutoriel, nous allons voir les objets de tampons uniformes (*uniform buffer objects*) et faire bon usage de `glBufferSubData()`.

## VII-C - Remerciements

Ce tutoriel est une traduction réalisée par **Alexandre Laurent** dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

## VIII - GLSL Avancé

Ce tutoriel ne présente pas vraiment de fonctionnalités avancées super cool qui apportent une énorme amélioration visuelle à votre scène. Ce tutoriel plonge dans certains aspects plus ou moins intéressant du GLSL et certaines astuces devraient vous aider dans vos projets futurs. Grossièrement, ce sont des fonctionnalités à connaître et qui peuvent rendre la vie plus simple lors de la création d'applications OpenGL et GLSL.

Nous allons voir quelques variables GLSL intéressantes, de nouvelles façons d'organiser les entrées et sorties des *shaders* et le superbe outil que sont les tampons uniformes.

### VIII-A - Variables GLSL

Les *shaders* ne contiennent que le minimum. Si nous avons besoin de données autres que les entrées du *shader*, nous devons passer ces données en argument. Nous avons appris à le faire avec les attributs de sommets, les variables uniformes et les échantillonneurs. En réalité, il existe d'autres variables proposées par GLSL et préfixées par `gl_` qui offrent de nouvelles façons pour récupérer ou écrire des données. Nous en avons déjà vus deux d'entre elles : `gl_Position`, le vecteur en sortie du *vertex shader*, et `gl_FragCoord`, la couleur en sortie du *fragment shader*.

Nous allons voir quelques variables intéressantes d'entrée et de sortie proposées par GLSL et qui peuvent nous être utiles. Notez que nous n'allons pas couvrir toutes les variables du GLSL. Vous pouvez accéder à une liste de celle-ci dans le [wiki d'OpenGL](#).

#### VIII-A-1 - Variables du vertex shader

Nous connaissons déjà `gl_Position`, représentant la position du sommet dans l'espace de découpage, calculée dans le *vertex shader*. Il est obligatoire de définir la variable `gl_Position` dans le *vertex shader* pour obtenir quelque chose à l'écran. Rien de nouveau.

##### VIII-A-1-a - `gl_PointSize`

Lors du rendu, nous pouvons choisir la primitive `GL_POINTS`. Dans ce cas, chaque sommet représente un point. Il est possible de définir la taille des points grâce à la fonction `glPointSize()`, mais il est aussi possible d'influer la valeur dans le *vertex shader*.

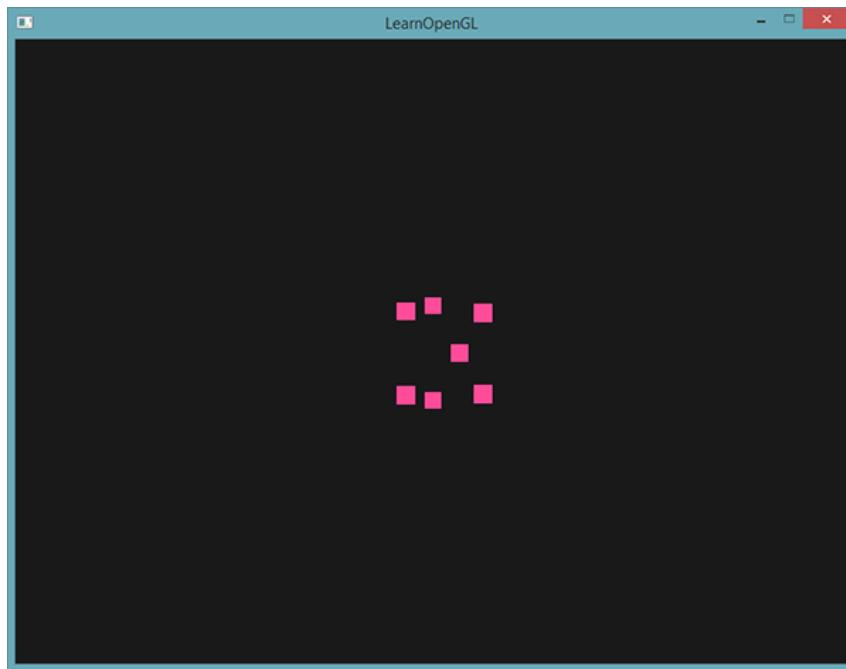
Par défaut, vous ne pouvez pas affecter la taille des points dans le *vertex shader*. Vous devez l'activer avec l'option `GL_PROGRAM_POINT_SIZE` :

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

Une méthode simple pour modifier la taille des points est d'utiliser la composante `z` de la position dans l'espace de découpage des sommets. Elle correspond à la distance entre le sommet et le spectateur. La taille des points doit augmenter plus ils sont loin du spectateur.

```
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    gl_PointSize = gl_Position.z;
}
```

Le résultat est que les points sont de plus en plus grand lorsqu'on s'éloigne d'eux :



Vous pouvez imaginer la taille des points pour chaque sommet dans votre système de particules.

### VIII-A-1-b - gl\_VertexID

Les variables `gl_Position` et `gl_PointSize` sont des sorties, car leur valeur est lue après l'exécution du *vertex shader* : ces variables nous permettent de modifier le rendu en leur donnant une valeur. Le *vertex shader* fournit aussi des variables d'entrée intéressantes et que nous pouvons lire, notamment `gl_VertexID`.

La variable `gl_VertexID` contient l'identifiant du sommet en cours de dessin. Lorsque vous faites un rendu indexé (avec la fonction `glDrawElements()`), cette variable contient l'indice du sommet en cours de dessin. Lors d'un rendu sans indices (avec la fonction `glDrawArrays()`), cette variable contient le numéro du sommet en cours d'affichage (en démarrant à partir du début de l'appel de dessin).

Bien que cela ne soit pas utile pour le moment, c'est toujours bon de savoir que nous avons accès à une telle information.

### VIII-A-2 - Variables du fragment shader

Le *fragment shader* fournit aussi des variables intéressantes. Le GLSL nous proposent deux variables d'entrées intéressantes : `gl_FragCoord` et `gl_FrontFacing`.

### VIII-A-2-a - gl\_FragCoord

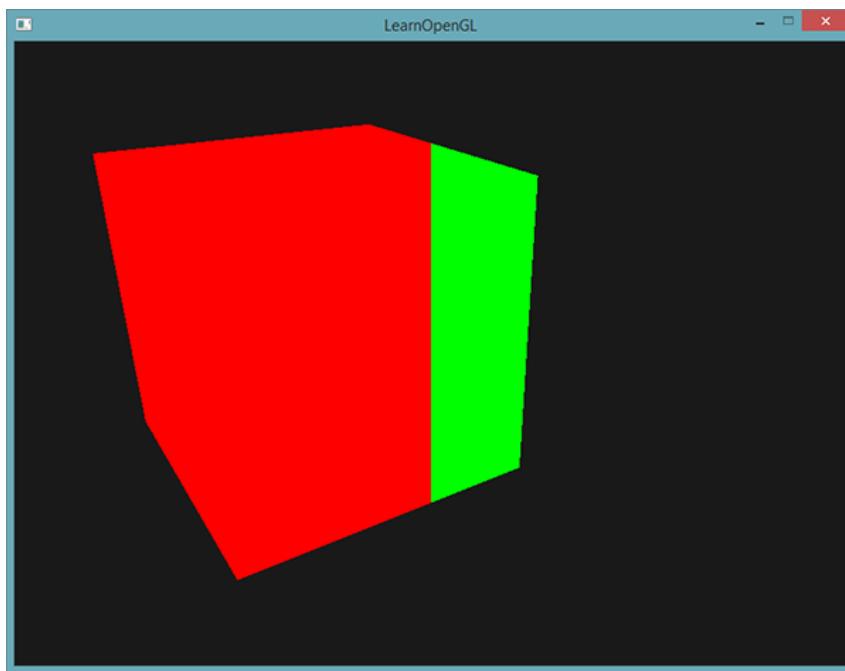
Nous avons déjà vu plusieurs fois la variable `gl_FragCoord` dans le tutoriel sur les tests de profondeur, car la composante `z` de la variable `gl_FragCoord` est égale à la valeur de la profondeur pour ce fragment. Toutefois, nous pouvons aussi utiliser les composants `x` et `y` du vecteur pour obtenir des effets intéressants.

Les composants `x` et `y` de la variable `gl_FragCoord` sont les coordonnées du fragment, dans l'espace écran, partant du bas gauche de la fenêtre. Nous avons défini une fenêtre de 800x600 avec `glViewport()`. Nous aurons donc des valeurs de 0 à 800 pour la composante `x` et de 0 à 600 pour la composante `y`.

En utilisant le *fragment shader*, nous pouvons générer une couleur différente suivant la position du fragment dans la fenêtre. Une utilisation classique est l'emploi de la variable `gl_FragCoord` en affichant une couleur différente suivant les calculs du fragment. Par exemple, nous pouvons découper l'écran en deux, en affichant une couleur pour la partie gauche et une autre pour la partie de droite. Voici un *fragment shader* qui le fait suivant les coordonnées du fragment dans la fenêtre :

```
void main()
{
    if(gl_FragCoord.x < 400)
        FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    else
        FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Comme la largeur de la fenêtre est égale à 800, chaque fois que la composante `x` du pixel est inférieure à 400, il doit être sur le côté gauche de la fenêtre et donne donc une couleur différente.



Nous pouvons maintenant utiliser deux *fragment shader* distincts qui vont afficher un résultat différent sur chaque partie de la fenêtre. C'est utile pour tester des techniques d'éclairage différentes.

## VIII-A-2-b - `gl_FrontFacing`

`gl_FrontFacing` est une autre variable d'entrée du *fragment shader*. Dans le [tutoriel sur l'élimination des faces](#), nous avons mentionné qu'OpenGL est capable de déterminer si la face est une face avant ou arrière grâce à l'ordre des sommets. Si nous n'utilisons pas l'élimination des faces (en activant `GL_FACE_CULL`), alors la variable `gl_FrontFacing` indique si le fragment actuel fait partie de la face avant ou arrière, nous pouvons alors décider de calculer une couleur différente pour l'une et l'autre.

La variable `gl_FrontFacing` est un booléen qui est à `true` si le fragment fait partie de la face avant. Sinon, la variable est à `false`. Par exemple, nous pouvons créer un cube ayant une texture différente à l'intérieur :

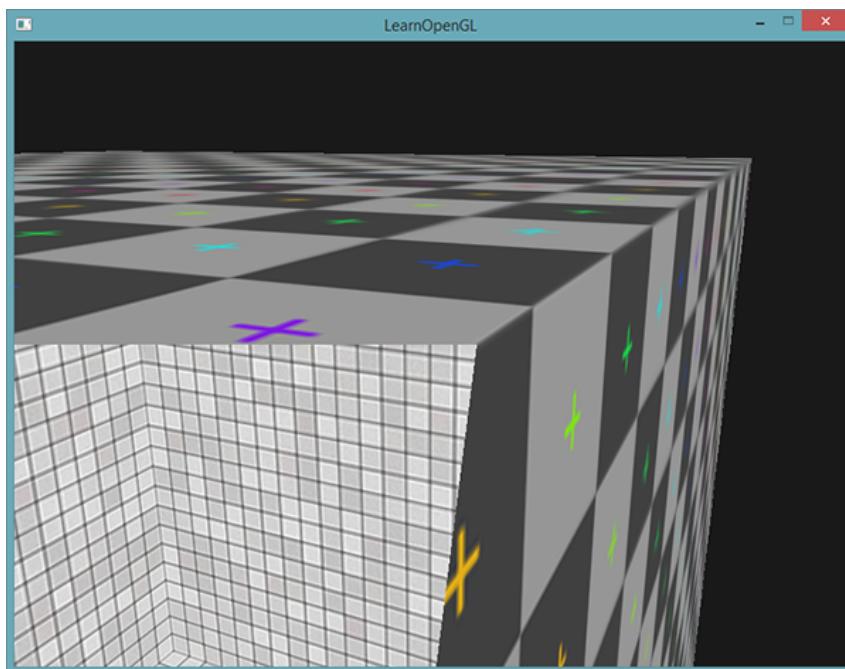
```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
```

```

uniform sampler2D frontTexture;
uniform sampler2D backTexture;

void main()
{
    if(gl_FrontFacing)
        FragColor = texture(frontTexture, TexCoords);
    else
        FragColor = texture(backTexture, TexCoords);
}
    
```

Si nous regardons à l'intérieur du conteneur, nous pouvons voir qu'une texture différente est utilisée :



Notez que, si vous activez l'élimination des faces, vous ne pourrez pas voir l'intérieur du conteneur et l'utilisation de `gl_FrontFacing` sera donc inutile.

### VIII-A-2-c - `gl_FragDepth`

La variable d'entrée `gl_FragCoord` permet de lire les coordonnées en espace écran et d'obtenir la valeur de profondeur du fragment actuel. Toutefois, c'est une variable en lecture seule. Nous ne pouvons pas influer sur les coordonnées en espace écran du fragment, mais il est possible de définir la profondeur du fragment. GLSL nous offre une variable de sortie appelée `gl_FragDepth` qui peut être utilisée pour modifier la profondeur du fragment dans le *shader*.

Pour définir la valeur de profondeur, il suffit d'écrire une valeur entre 0,0 et 1,0 dans la variable `gl_FragDepth` :

```
gl_FragDepth = 0.0; // le fragment a maintenant une profondeur de 0.0
```

Si le *shader* ne fournit pas de valeur à la variable `gl_FragDepth`, la valeur de `gl_FragCoord.z` sera utilisée.

Lorsque la profondeur est définie par nous-même, OpenGL désactive l'optimisation du test de profondeur anticipé (comme vu dans le [tutoriel sur les tests de profondeur](#)). OpenGL ne peut pas connaître la profondeur avant d'exécuter le *fragment shader*, comme ce dernier pourrait entièrement en changer la valeur.

En écrivant dans la variable `gl_FragDepth`, vous devez prendre en considération cet impact sur les performances. Toutefois, en OpenGL 4.2, nous pouvons obtenir un compromis en redéclarant la variable `gl_FragDepth` au début du *fragment shader* :

```
layout (depth_<condition>) out float gl_FragDepth;
```

La condition peut être une des valeurs suivantes :

| Condition | Description  |
|-----------|--|
| any       | La valeur par défaut. Le test de profondeur anticipé est désactivé et vous allez perdre en performance.        |
| greater   | Vous ne pouvez écrire qu'une valeur supérieure à <code>gl_FragCoord.z</code> .                                 |
| less      | Vous ne pouvez écrire qu'une valeur inférieure à <code>gl_FragCoord.z</code> .                                 |
| unchanged | Si vous écrivez dans la variable <code>gl_FragDepth</code> , vous allez utiliser <code>gl_FragCoord.z</code> . |

En indiquant `greater` ou `less` comme condition, OpenGL peut donc partir du principe que vous allez écrire une valeur plus grande ou inférieure que la valeur actuelle. De cette façon, OpenGL est encore capable de faire un test de profondeur anticipé dans les cas où la valeur de profondeur est inférieure ou supérieure à la valeur du fragment.

Voici un exemple où nous incrémentons la valeur de profondeur dans le fragment, mais où nous conservons le test de profondeur anticipé :

```
#version 420 core // notez la version GLSL !
out vec4 FragColor;
layout (depth_greater) out float gl_FragDepth;

void main()
{
    FragColor = vec4(1.0);
    gl_FragDepth = gl_FragCoord.z + 0.1;
}
```

Notez que cette fonctionnalité n'est disponible qu'à partir d'OpenGL 4.2.

## VIII-B - Blocs d'interface

Jusqu'à présent, chaque fois que nous voulions envoyer des données du *vertex shader* au *fragment shader*, nous déclarions plusieurs variables des deux côtés. les déclarer une par une est la méthode la plus simple pour transférer des données, mais, pour des applications de plus en plus imposantes, nous voudrions certainement envoyer plus de variables, dont des tableaux ou des structures.

Pour nous aider à organiser ces variables, GLSL nous propose des blocs d'interface qui nous permettent de regrouper ces variables. La déclaration d'un tel bloc ressemble beaucoup à la déclaration d'une structure, sauf que nous devons utiliser les mots clés `in` et `out` pour définir un bloc d'entrée ou de sortie.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
```

```

out VS_OUT
{
    vec2 TexCoords;
} vs_out;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    vs_out.TexCoords = aTexCoords;
}

```

Cette fois, nous avons déclaré un bloc d'interface appelé `vs_out` qui regroupe toutes les sorties que nous souhaitons envoyer au *shader* suivant. Cet exemple est trivial, mais vous pouvez imaginer combien cela peut aider à organiser les entrées sorties de vos *shaders*. C'est aussi pratique lorsque nous voulons regrouper des entrées/sorties dans des tableaux, comme nous allons le voir dans le prochain [tutoriel sur les geometry shaders](#).

Ensuite, nous allons déclarer un bloc d'entrée dans le *shader* suivant, soit, le *fragment shader*. Le nom du bloc (`VS_OUT`) doit être le même, mais l'instance (`vs_out` dans le *vertex shader*) peut être n'importe quoi d'autre. Évitez donc d'utiliser des noms apportant confusion comme `vs_out` alors que ce sont des variables d'entrée.

```

#version 330 core
out vec4 FragColor;

in VS_OUT
{
    vec2 TexCoords;
} fs_in;

uniform sampler2D texture;

void main()
{
    FragColor = texture(texture, fs_in.TexCoords);
}

```

Tant que les deux noms d'interface correspondent, l'entrée et la sortie seront connectées. C'est une autre fonctionnalité utile pour vous aider à connecter vos *shaders* ; elle s'avère d'autant plus utile lors de l'implémentation d'étapes supplémentaires comme le *geometry shader*.

## VIII-C - Tampon de variables uniformes

Nous avons utilisé OpenGL depuis quelque temps et appris pas mal d'astuces, mais aussi quelques trucs pénibles. Par exemple, lorsque nous utilisons plus d'un *shader*, nous devons définir des variables uniformes, et ce, même si ce sont exactement les mêmes pour chaque *shader*. Pourquoi donc s'embêter à les redéfinir ?

OpenGL nous offre un outil nommé les tampons de variables uniformes (*uniform buffer objects*). Cela nous permet de déclarer un ensemble de variables uniformes globales, qui sera disponible au travers de plusieurs *shaders*. Lors de l'utilisation d'un tampon de variables uniformes, nous avons donc un ensemble de variables uniformes défini une seule fois. Nous pouvons toujours définir des variables uniformes manuellement, qui sont accessibles uniquement par un *shader*. La création et la configuration d'un tampon de variables uniformes demande toutefois un peu de travail.

Comme un tampon de variables uniformes est un tampon, nous le créons comme tout autre tampon avec la fonction `glGenBuffers()`, lié à la cible `GL_UNIFORM_BUFFER` et nous stockons toutes les données des variables uniformes dans le tampon, il faut suivre certaines règles pour faire cela. Pour le moment, nous allons simplement prendre un *vertex shader* et stocker notre matrice de projection et de vue dans un tampon de variables uniformes :

```

#version 330 core
layout (location = 0) in vec3 aPos;

layout (std140) uniform Matrices
{

```

```

mat4 projection;
mat4 view;
};

uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}

```

Dans la majorité de nos exemples, nous définissons la matrice de projection et de vue à chaque itération, et ce, pour chaque *shader* que nous utilisons. C'est un exemple parfait pour utiliser un tampon de variables uniformes, car nous n'avons alors plus qu'à stocker ces matrices une seule fois.

Ici, nous avons déclaré un bloc de variables uniformes appelé *Matrices*, qui stocke deux matrices 4x4. Les variables dans un bloc de variables uniformes sont accessibles sans devoir utiliser le nom du bloc comme préfixe. Ensuite, nous stockons ces matrices dans un tampon quelque part dans le code OpenGL et chaque *shader* déclaré avec ce bloc de variables uniformes pourra accéder aux matrices.

Vous vous demandez sûrement ce qu'est ce layout (std140). Cela indique que le bloc de variables uniformes défini utilise une disposition mémoire spécifique.

### VIII-C-1 - Disposition du bloc de variables uniformes

Le contenu du bloc de variables uniformes est stocké dans un tampon, qui n'est rien d'autre qu'un morceau de mémoire. Comme ce morceau ne contient aucune information sur ce qu'il contient, nous devons indiquer à OpenGL quelle partie de cette mémoire correspond à quelle variable uniforme du *shader*.

Imaginez le bloc de variables uniformes suivant dans un *shader* :

```

layout (std140) uniform ExampleBlock
{
    float value;
    vec3 vector;
    mat4 matrix;
    float values[3];
    bool boolean;
    int integer;
};

```

Ce que nous voulons connaître est la taille (en octets) et le décalage (à partir du début du bloc) de chacune de ces variables afin que nous puissions les placer correctement dans le tampon, et ce, en respectant l'ordre. La taille de chacun de ces éléments est définie dans OpenGL et correspond au type de données C++ ; les vecteurs et les matrices étant des (grands) tableaux de nombres flottants. Par contre, OpenGL n'indique pas l'espacement entre les variables. C'est au matériel de positionner les variables comme il le peut. Certaines machines peuvent placer un `vec3` à côté d'un `float`, par exemple, mais toutes ne peuvent pas gérer une telle disposition et devront placer le `vec3` dans un emplacement de quatre nombres flottants, puis placer le `float`. Une bonne fonctionnalité, mais pénible pour nous.

Par défaut, le GLSL utilise une disposition de la mémoire appelée disposition partagée – partagée, car les décalages sont définis par le matériel et sont donc partagés entre plusieurs *shaders*. Avec une disposition partagée, le GLSL peut repositionner les variables uniformes à des fins d'optimisation tant que l'ordre n'est pas impacté. Comme nous ne savons pas à quel endroit les variables uniformes seront, nous ne savons pas comment remplir notre tampon. Nous pouvons récupérer cette information avec des fonctions comme `glGetUniformIndices()`, mais cela dépasse le cadre de ce tutoriel.

Même si la disposition partagée permet de gagner un peu d'espace, nous ne souhaitons pas devoir récupérer le décalage de chaque variable uniforme. Une pratique générique est de ne pas utiliser une disposition partagée, mais la disposition std140. La disposition std140 indique explicitement la disposition mémoire de chaque variable.

en définissant leur décalage respectif grâce à un ensemble de règles. Comme cela est explicitement défini, nous pouvons déterminer manuellement le décalage pour chaque variable.

Chaque variable a un alignement de base qui est égal à l'espace que la variable prend (en prenant en compte le remplissage [*padding*]) dans un bloc de variables uniformes. L'alignement de base est calculé en utilisant les règles de la disposition mémoire std140. Ensuite, pour chaque variable, nous calculons son décalage aligné, qui correspond au décalage en octets d'une variable par rapport au commencement du bloc. Le décalage aligné en octets est une variable qui **doit** être égale à un multiple de l'alignement de base.

Les règles de disposition peuvent être trouvées dans la spécification des tampons de variables uniformes d'OpenGL [ici](#), mais voici une liste des règles les plus communes. Chaque type de variables en GLSL, tels que les `int`, `float` et `bool` sont définis sur quatre octets et chaque entité de quatre octets est représentée par **N**.

| Type   | Règle de disposition   |
|--|--|
| Scalaire (c'est-à-dire <code>int</code> ou <code>bool</code> ) | Chaque scalaire a un alignement de base de N.  |
| Vecteur  | Soit $2N$ , soit $4N$ . Cela signifie qu'un <code>vec3</code> a un alignement de base de $4N$ .                        |
| Tableau de scalaires ou de vecteurs                            | Chaque élément a un alignement de base d'un <code>vec4</code> .  |
| Matrice  | Stockée comme un grand tableau de vecteurs colonnes, où chacun de ces vecteurs a l'alignement d'un <code>vec4</code> . |
| Structure  | Équivalente à la taille calculée par ces éléments suivant les règles précédentes.                                      |

Comme avec la plus grande partie de la spécification OpenGL, c'est plus simple de comprendre avec un exemple. Nous prenons le bloc de variables uniformes appelé `ExampleBlock`, que nous introduirons plus tard et nous calculons le décalage aligné de chacun de ses membres avec la disposition std140 :

```
layout (std140) uniform ExampleBlock
{
    float value;           // alignement de base // décalage aligné
                           // 4               // 0
    vec3 vector;          // 16              // 16  (doit être un multiple de 16, donc 4 -> 16)
    mat4 matrix;          // 16              // 32  (colonne 0)
                           // 16              // 48  (colonne 1)
                           // 16              // 64  (colonne 2)
                           // 16              // 80  (colonne 3)
    float values[3];      // 16              // 96  (valeur[0])
                           // 16              // 112 (valeur[1])
                           // 16              // 128 (valeur[2])
    bool boolean;         // 4               // 144
    int integer;          // 4               // 148
};
```

Comme exercice, essayez de calculer les valeurs de décalage vous-même et comparez-les avec ce tableau. Avec les décalages calculés, basés sur les règles de la disposition std140, nous pouvons remplir le tampon avec les données des variables à chaque décalage avec la fonction `glBufferSubData()`. Même si ce n'est pas plus efficace, la disposition std140 nous garantit que la disposition mémoire reste la même pour chaque programme ayant déclaré ce bloc de variables uniformes.

En ajoutant le `layout (std140)` avant la définition du bloc de variables uniformes, nous indiquons à OpenGL d'utiliser la disposition std140. Il y a deux autres dispositions qui nécessitent de récupérer le décalage avant de remplir le tampon. Nous avons déjà vu la disposition **partagée**, la seconde étant la disposition **empaquetée**. Lors de l'utilisation de la disposition empaquetée, il n'y a pas de garantie que la disposition reste la même entre les `shader` (non partagée), car elle permet au compilateur de retirer les variables uniformes du bloc, ce qui peut différer entre les `shaders`.

## VIII-C-2 - Utiliser un tampon de variables uniformes

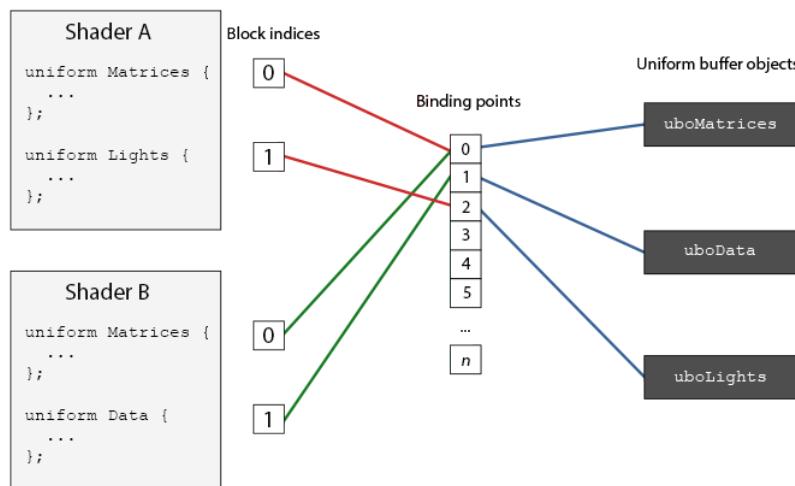
Nous avons vu comment définir les blocs de variables uniformes dans les *shaders* et comment spécifier la disposition mémoire, mais nous n'avons pas encore vu comment les utiliser.

Premièrement, nous devons créer un tampon de variables uniformes avec la fonction `glGenBuffers()`. Une fois que nous avons un tampon, nous le lions à la cible `GL_UNIFORM_BUFFER` et lui allouons assez de mémoire grâce à `glBufferData()`.

```
unsigned int uboExampleBlock;
 glGenBuffers(1, &uboExampleBlock);
 glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
 glBufferData(GL_UNIFORM_BUFFER, 152, NULL, GL_STATIC_DRAW); // allouer 150 octets de mémoire
 glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Maintenant, chaque fois que nous mettons à jour ou insérons des données dans le tampon, nous utiliserons `uboExampleBlock` et la fonction `glBufferSubData()` pour mettre à jour sa mémoire. Nous ne mettons à jour le tampon de variables uniformes qu'une seule fois, et chaque *shader* l'utilisant aura accès aux données à jour. Maintenant, comment OpenGL sait-il quel tampon de variables uniformes correspond à quel bloc ?

Dans le contexte OpenGL, il y a un nombre de points de liaison auxquels nous pouvons lier le tampon. Une fois le tampon de variables uniformes créé, nous le lions à un de ces points de liaison et nous lions aussi le bloc de variables uniformes à ce même point, les liant ainsi. Le diagramme suivant illustre cela :



Comme vous pouvez le voir, nous pouvons lier plusieurs tampons de variables uniformes à plusieurs points de liaison. Comme un *shader A* et un *shader B* ont un bloc de variables uniformes lié à un même point de liaison 0, leur bloc partage les mêmes données de variables uniformes, provenant de `uboMatrices`. La seule contrainte est que les deux *shaders* doivent définir le même bloc de variables uniformes `Matrices`.

Pour définir un bloc de variables uniformes à un point de liaison spécifique, nous appelons `glUniformBlockBinding()`, qui prend un *program shader* comme premier argument, un index de bloc de variables uniformes et un point de liaison auquel le lier. L'index du bloc de variables uniformes est un emplacement correspondant au bloc de variables uniformes du *shader*. Il peut être obtenu en appelant la fonction `glGetUniformBlockIndex()`, qui accepte un *program shader* et le nom du bloc de variables uniformes. Nous pouvons définir le bloc de variables uniformes `Lights` du diagramme au point de liaison 2 comme suit :

```
unsigned int lights_index = glGetUniformLocation(shaderA.ID, "Lights");
glUniformBlockBinding(shaderA.ID, lights_index, 2);
```

Remarquez que nous devons répéter cela pour **chaque shader**.

À partir d'OpenGL 4.2, il est possible de stocker les points de liaison d'un bloc de variables uniformes directement dans le shader en ajoutant un indicateur de disposition, permettant d'éviter d'appeler `glGetUniformBlockIndex()` et `glUniformBlockBinding()`. Le code suivant définit le point de liaison pour le bloc Lights :

```
layout(std140, binding = 2) uniform Lights { ... };
```

Ensuite, nous devons lier le tampon de variables uniformes au même point de liaison, ce qui peut être réalisé avec `glBindBufferBase()` ou `glBindBufferRange()`.

```
glBindBufferBase(GL_UNIFORM_BUFFER, 2, uboExampleBlock);
// ou
glBindBufferRange(GL_UNIFORM_BUFFER, 2, uboExampleBlock, 0, 152);
```

La fonction `glBindBufferBase()` nécessite une cible, un index de point de liaison et un tampon de variables uniformes. Cette fonction lie le tampon `uboExampleBlock` au point de liaison 2. À partir de maintenant, les deux côtés du point de liaison sont liés. Vous pouvez aussi utiliser la fonction `glBindBufferRange()` qui attend, en plus, un paramètre pour le décalage et un autre pour la taille. De cette façon, vous pouvez lier une sous-partie du tampon de variables uniformes au point de liaison. En utilisant `glBindBufferRange()`, vous pouvez disposer de plusieurs blocs de variables uniformes lié à un seul tampon de celles-ci.

Maintenant que tout est configuré, nous pouvons commencer à ajouter des données au tampon de variables uniformes. Nous pourrions ajouter toutes les données comme un seul tableau d'octets ou mettre à jour des parties comme vous le souhaitez avec la fonction `glBufferSubData()`. Pour mettre à jour la variable booléenne uniquement, nous pourrions mettre à jour le tampon de variables uniformes comme suit :

```
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
int b = true; // les booléens du GLSL sont sur quatre octets, donc nous les stockons comme un entier
glBufferSubData(GL_UNIFORM_BUFFER, 144, 4, &b);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

La même procédure s'applique pour toutes les autres variables uniformes du bloc, mais avec différents arguments pour cibler les autres sous-parties.

### VIII-C-3 - Un exemple simple

Mettons en application l'utilisation d'un tampon de variables uniformes dans un vrai exemple. Si nous regardons en arrière dans tous les exemples de code, nous avons utilisé continuellement trois matrices : la projection, la vue et les matrices de modèles. Seule la matrice des modèles change régulièrement. Si nous avons plusieurs *shaders* qui utilisent ce même ensemble de matrices, nous pouvons certainement utiliser un tampon de variables uniformes.

Nous allons stocker les matrices de projection et de vue dans un bloc de variables uniformes Matrices. Nous n'allons pas stocker la matrice de modèle, car elle change régulièrement entre chaque *shader* et il n'y a donc aucun bénéfice de la stocker dans le tampon.

```
#version 330 core
layout(location = 0) in vec3 aPos;

layout(std140) uniform Matrices
{
    mat4 projection;
    mat4 view;
};
uniform mat4 model;

void main()
```

```
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Rien de spécial ici, sauf que nous utilisons un bloc de variables uniformes avec la disposition std140. Dans cette application, nous allons afficher quatre cubes où chaque cube utilise un *shader program* différent. Chacun de ces quatre *shader programs* utilise le même *vertex shader*, mais un *fragment shader* qui affiche une couleur différente pour chaque *shader*.

Premièrement, nous associons le bloc de variables uniformes du *vertex shader* au point de liaison 0. Notez que nous devons le faire pour chaque *shader*.

```
unsigned int uniformBlockIndexRed    = glGetUniformBlockIndex(shaderRed.ID, "Matrices");
unsigned int uniformBlockIndexGreen  = glGetUniformBlockIndex(shaderGreen.ID, "Matrices");
unsigned int uniformBlockIndexBlue   = glGetUniformBlockIndex(shaderBlue.ID, "Matrices");
unsigned int uniformBlockIndexYellow = glGetUniformBlockIndex(shaderYellow.ID, "Matrices");

glUniformBlockBinding(shaderRed.ID, uniformBlockIndexRed, 0);
glUniformBlockBinding(shaderGreen.ID, uniformBlockIndexGreen, 0);
glUniformBlockBinding(shaderBlue.ID, uniformBlockIndexBlue, 0);
glUniformBlockBinding(shaderYellow.ID, uniformBlockIndexYellow, 0);
```

Ensuite, nous créons le tampon de variables uniformes et le lions aussi au point de liaison 0 :

```
unsigned int uboMatrices
glGenBuffers(1, &uboMatrices);

glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferData(GL_UNIFORM_BUFFER, 2 * sizeof(glm::mat4), NULL, GL_STATIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);

glBindBufferRange(GL_UNIFORM_BUFFER, 0, uboMatrices, 0, 2 * sizeof(glm::mat4));
```

Premièrement, nous devons allouer assez de mémoire pour notre tampon, soit deux fois la taille d'un `glm::mat4`. La taille des types des matrices de GLSL correspond exactement aux `mat4` du GLSL. Ensuite, nous lions la partie spécifique du tampon, soit, dans notre cas, l'intégralité du tampon, au point de liaison 0.

Maintenant, il ne nous reste plus qu'à remplir le tampon. Si nous gardons la valeur du champ de vision constant dans la matrice de projection (donc, plus de zoom), nous n'avons qu'à la définir dans notre application. Ce qui veut dire que nous n'avons qu'à l'insérer dans le tampon. Comme nous avons déjà alloué assez de mémoire pour le tampon, nous pouvons utiliser `glBufferSubData()` pour stocker la matrice de projection avant d'entrer dans la boucle de jeu :

```
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)width/
(float)height, 0.1f, 100.0f);
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4), glm::value_ptr(projection));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Ici, nous stockons la matrice de projection dans la première partie du tampon de variables uniformes. Avant nous dessinons les objets à chaque itération de rendu, puis nous mettons à jour la seconde partie du tampon avec la matrice de vue :

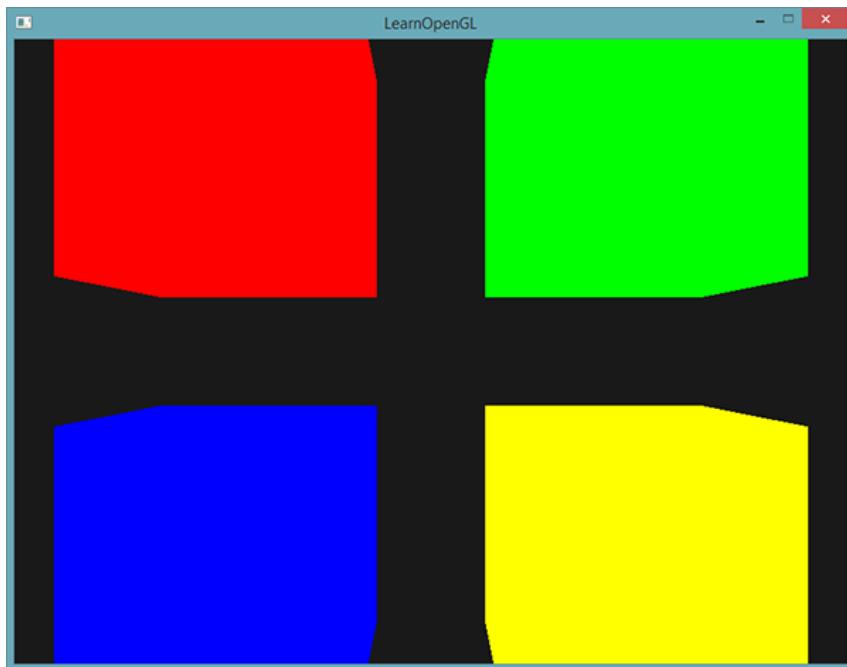
```
glm::mat4 view = camera.GetViewMatrix();
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, sizeof(glm::mat4), sizeof(glm::mat4), glm::value_ptr(view));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Et c'est tout pour le tampon de variables uniformes. Chaque *vertex shader* qui contient le bloc de variables uniformes Matrices contiendra maintenant les données stockées par le tampon `uboMatrices`. Donc, si nous devons maintenant afficher quatre cubes avec quatre *shaders* différent, leur matrices de projection et de vue seront les mêmes :

```

glBindVertexArray(cubeVAO);
shaderRed.use();
glm::mat4 model;
model = glm::translate(model, glm::vec3(-0.75f, 0.75f, 0.0f)); // se déplacer en haut à gauche
shaderRed.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
// ... dessiner le cube vert
// ... dessiner le cube bleu
// ... dessiner le cube jaune
    
```

La seule variable uniforme qu'il nous reste à définir est la variable `model`. En utilisant un tampon de variables uniformes, cela nous évite quelques appels à celles-ci par `shader`. Le résultat est le suivant :



Chacun de ces cubes est placé dans chaque côté de la fenêtre en altérant la matrice du modèle et, grâce au *fragment shader*, chacun affiche une couleur différente. C'est un scénario plutôt simple où nous pouvons utiliser les tampons de variables uniformes, mais une grande application graphique peut avoir des centaines de *shader* actifs et, dans ces cas, les tampons de variables uniformes brillent.

Vous pouvez trouver le code source de l'exemple [Source ici](#).

Les tampons de variables uniformes ont plusieurs avantages face aux simples variables uniformes. Premièrement, la définition de nombreuses variables en une fois est plus rapide que de définir chaque variable une à une. Ensuite, si vous souhaitez changer la même variable uniforme dans plusieurs *shaders*, il est bien plus facile de le faire en une fois grâce au tampon de variables uniformes. Le dernier avantage, qui n'est pas immédiat, est que vous pouvez utiliser plus de variables uniformes dans les *shaders* grâce aux tampons. OpenGL a une limite sur la quantité de données uniformes. Celle-ci peut être obtenue avec l'option `GL_MAX_VERTEX_UNIFORM_COMPONENTS`. En utilisant les tampons de variables uniformes, la limite est plus haute. Donc, chaque fois que vous atteignez le nombre maximal de variables uniformes (lors d'une animation squelettique, par exemple), vous pouvez toujours utiliser un tampon de variables uniformes.

## VIII-D - Remerciements

Ce tutoriel est une traduction réalisée par **Alexandre Laurent** dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

## IX - Geometry shader

Il existe une étape optionnelle entre le *vertex shader* et le *fragment shader* appelée le *geometry shader*. Un *geometry shader* prend en entrée un ensemble de sommets formant une primitive simple, c'est-à-dire un point ou un triangle, qu'il peut ensuite transformer à sa guise avant de l'envoyer à l'étape du pipeline qui suit. Les *geometry shaders* sont intéressants, car ils permettent de transformer l'ensemble des sommets en une primitive complètement différente et, pourquoi pas, d'ajouter d'autres sommets à ceux donnés.

Nous allons directement commencer par un exemple d'un *geometry shader* :

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Au début de tout *geometry shader*, nous devons déclarer le type de primitive en entrée envoyée par le *vertex shader*. Nous le faisons en déclarant une disposition devant le mot clé `in`. Cet indicateur de disposition peut prendre une des valeurs suivantes :

- `points` : lors de l'affichage des primitives `GL_POINTS` (1) ;
- `lines` : lors de l'affichage avec `GL_LINES` ou `GL_LINE_STRIP` (2) ;
- `lines_adjacency` : `GL_LINES_ADJACENCY` ou `GL_LINE_STRIP_ADJACENCY` (4) ;
- `triangles` : `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` ou `GL_TRIANGLE_FAN` (3) ;
- `triangles_adjacency` : `GL_TRIANGLES_ADJACENCY` ou `GL_TRIANGLE_STRIP_ADJACENCY` (6).

En bref, cela reflète les primitives que nous pouvons spécifier lors de l'appel à `glDrawArrays()`. Si nous choisissons de dessiner des `GL_TRIANGLES`, nous devrions choisir le qualificateur `triangles`. Le nombre entre parenthèses représente le nombre minimum de sommets que chaque primitive contient.

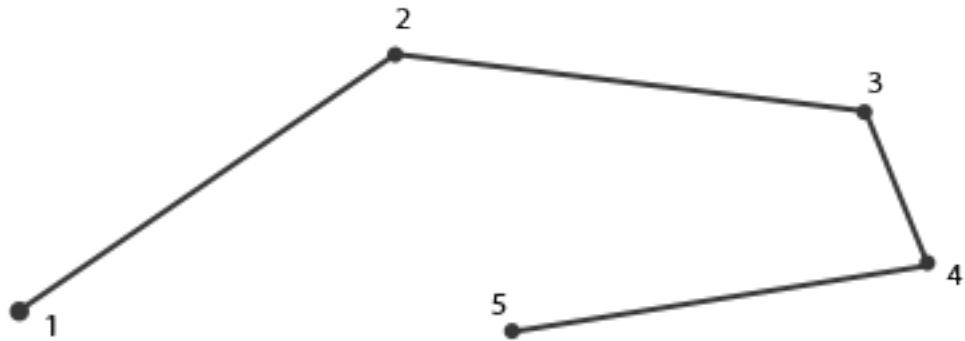
Ensuite, nous devons aussi indiquer le type de primitive que le *geometry shader* génère et nous le faisons en ajoutant un qualificateur de disposition au mot-clé `out`. Comme pour les entrées, le qualificateur de sortie peut prendre une de ces valeurs :

- `points` ;
- `line_strip` ;
- `triangle_strip`.

Avec juste ces trois qualificateurs de sorties, nous pouvons créer toutes les formes que nous voulons à partir des primitives d'entrée. Pour générer un simple triangle, nous pouvons utiliser `triangle_strip` comme sortie et générer trois sommets.

Le *geometry shader* attend aussi que nous définissons le nombre de sommets maximal en sortie (si vous dépassez ce nombre, OpenGL ne dessinera pas les sommets supplémentaires), ce que nous pouvons faire dans le qualificateur de disposition du mot-clé `out`. Dans ce cas, nous allons utiliser une sortie `line_strip` avec un maximum de deux sommets.

Au cas où vous posez la question : un `line_strip` est un ensemble de lignes connectées ensemble permettant de former une ligne continue constituée au minimum par deux points. Chaque point supplémentaire permet de créer une nouvelle ligne entre le nouveau point et le point précédent, comme vous pouvez le voir dans l'image ci-dessous, avec cinq sommets :



Avec le *shader* actuel, nous allons afficher une simple ligne avec un nombre maximal de deux sommets.

Pour générer des résultats adéquats, nous devons trouver un moyen de récupérer la sortie de l'étape précédente du pipeline. Le GLSL nous propose la variable `gl_in` qui, en interne, ressemble (sûrement) à ceci :

```
in gl_Vertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];
```

C'est un bloc d'interface (comme vu dans le [tutoriel précédent](#)) qui contient quelques variables intéressantes, notamment `gl_Position`, qui contient un vecteur similaire à celui en sortie du *vertex shader*.

Notez que c'est un tableau, car la plupart des primitives contiennent plus d'un sommet et que le *geometry shader* reçoit tous les sommets en une entrée.

En utilisant les données des sommets provenant du *vertex shader*, nous pouvons générer de nouvelles données grâce à deux fonctions du *geometry shader* : `EmitVertex` et `EndPrimitive`. Le *geometry shader* attend que vous généreriez au moins une primitive que vous avez spécifiée en sortie. Dans notre cas, nous devons générer au moins une primitive de type `line_strip`.

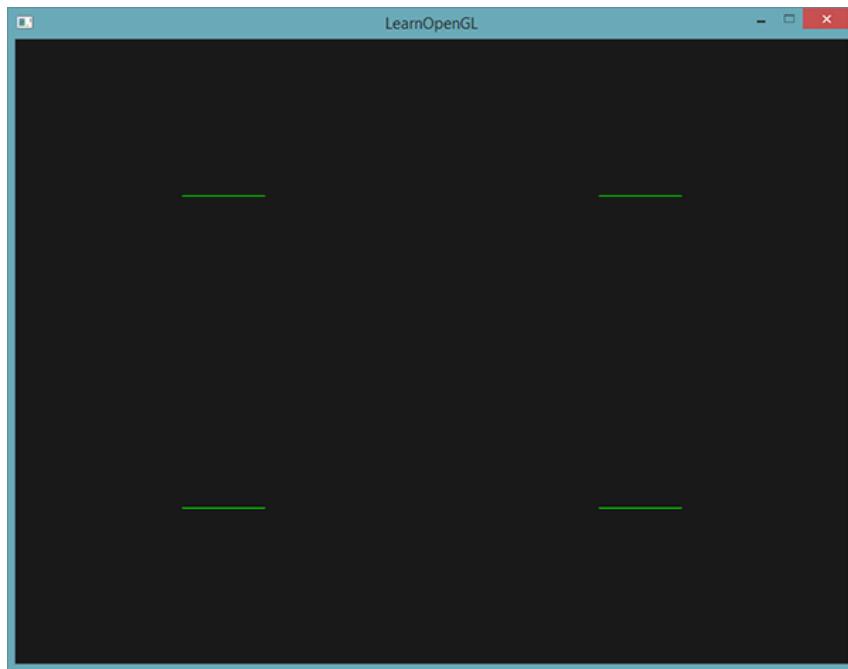
```
void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Chaque fois que nous appelons la fonction `EmitVectex()`, le sommet actuellement défini dans la variable `gl_Position` est ajouté à la primitive. Chaque fois que la fonction `EndPrimitive()` est appelée, tous les sommets de cette primitive sont combinés pour former la primitive à afficher. En appelant `EndPrimitive()` plusieurs fois après les appels à `EmitVertex()`, plusieurs primitives peuvent être générées. Dans ce cas particulier, nous émettons deux sommets ayant un petit décalage par rapport à la position du sommet original. Ensuite, l'appel à la fonction `EndPrimitive()` permet de combiner ces deux sommets en une suite de lignes de deux sommets.

Maintenant que vous savez (en quelque sorte) comment le *geometry shader* fonctionne, vous avez certainement deviné ce qu'il fait. Le *geometry shader* prend des points en entrée et crée une ligne horizontale avec le point en entrée comme centre. Cela donne le résultat visuel suivant :



Ce n'est pas incroyable pour le moment, mais il ne faut pas oublier que cela a été généré avec l'appel de rendu suivant :

```
glDrawArrays(GL_POINTS, 0, 4);
```

Même si c'est un exemple simple, cela montre comment utiliser les *geometry shaders* et comment générer (dynamiquement) de nouvelles formes. Plus tard dans ce tutoriel, nous allons voir comment créer des effets intéressants avec les *geometry shaders*. Avant, nous allons toutefois créer un *geometry shader* simple.

## IX-A - Utiliser les geometry shaders

Pour montrer l'utilisation du *geometry shader*, nous allons afficher une scène simple où nous dessinons quatre points sur le plan Z dans les coordonnées normalisées du périphérique. Les coordonnées de ces points sont :

```
float points[] = {
    -0.5f, 0.5f, // haut gauche
    0.5f, 0.5f, // haut droit
    0.5f, -0.5f, // bas droit
    -0.5f, -0.5f // bas gauche
};
```

Le *vertex shader* n'a besoin que de dessiner les points sur le plan Z, nous avons donc besoin d'un *vertex shader* basique :

```
#version 330 core
layout (location = 0) in vec2 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

Nous donnons une couleur verte pour chaque point dans le *fragment shader* :

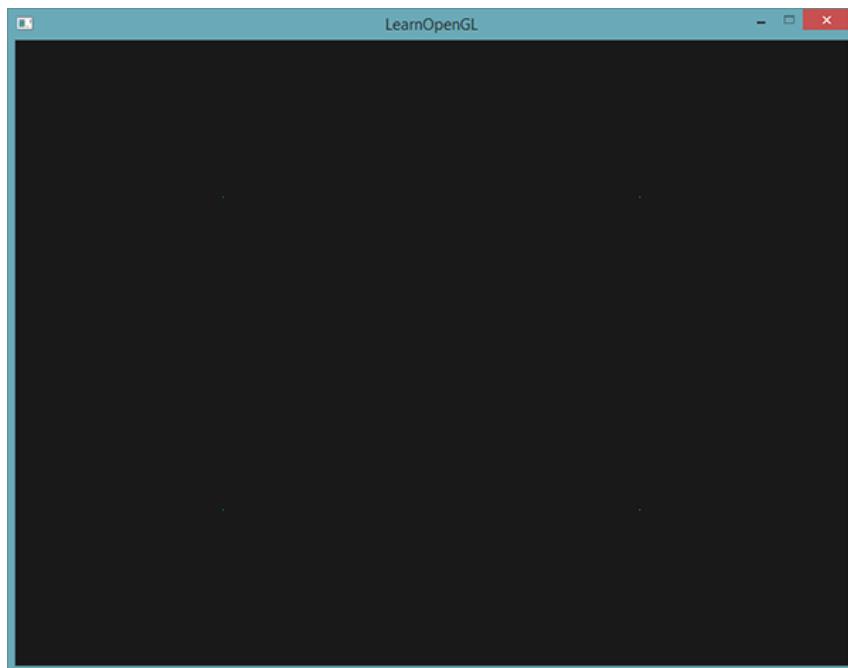
```
#version 330 core
out vec4 FragColor;
```

```
void main()
{
    FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Vous devez générer le VAO et le VBO des points, puis les dessiner avec ce code :

```
shader.use();
glBindVertexArray(VAO);
glDrawArrays(GL_POINTS, 0, 4);
```

Le résultat donne une scène noire avec quatre points verts (difficiles à voir) :



Mais nous avons déjà appris tout cela ? Oui, et maintenant nous allons pimenter un peu cette scène en lui ajoutant un *geometry shader*.

Pour les besoins de l'apprentissage, nous allons créer un *geometry shader* de transfert (*pass-through*) qui prend la primitive de point en entrée et qui la passe, non modifiée, au prochain *shader* du pipeline :

```
#version 330 core
layout (points) in;
layout (points, max_vertices = 1) out;

void main() {
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();
    EndPrimitive();
}
```

Jusqu'à présent, le *geometry shader* doit être facile à comprendre. Il émet simplement la position du sommet reçu en entrée et génère un point en sortie.

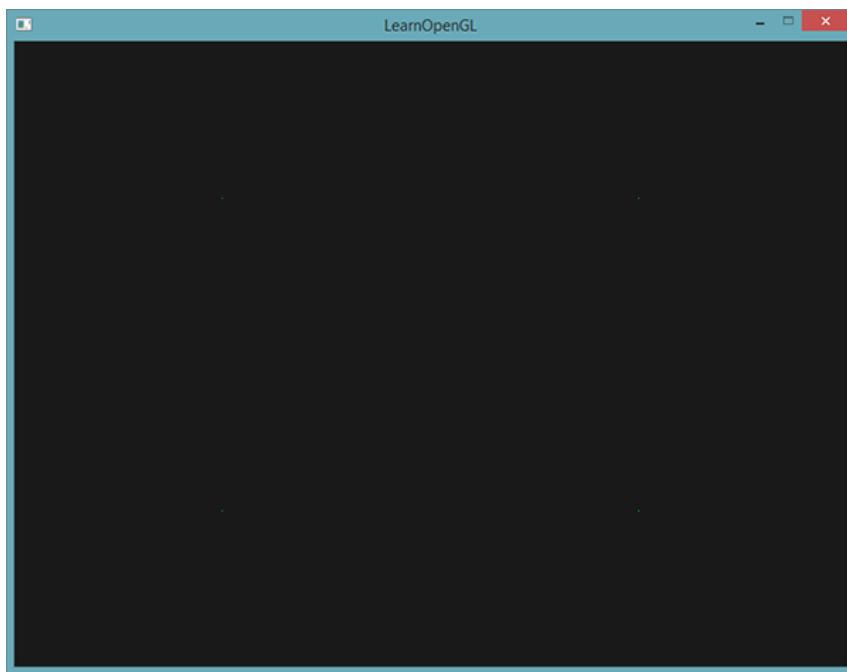
Le *geometry shader* doit être compilé et lié dans le programme, comme pour le *vertex shader* et le *fragment shader*, mais, cette fois, avec le type est `GL_GEOMETRY_SHADER` :

```
geometryShader = glCreateShader(GL_GEOMETRY_SHADER);
glShaderSource(geometryShader, 1, &gShaderCode, NULL);
glCompileShader(geometryShader);
```

```
...  
glAttachShader(program, geometryShader);  
glLinkProgram(program);
```

Le code de compilation du *shader* est basiquement le même que celui pour le vertex et le *fragment shader*. Soyez certain de vérifier pour les erreurs de compilation ou d'édition de liens !

Si vous compilez et exécutez le programme, vous devriez obtenir le résultat suivant :

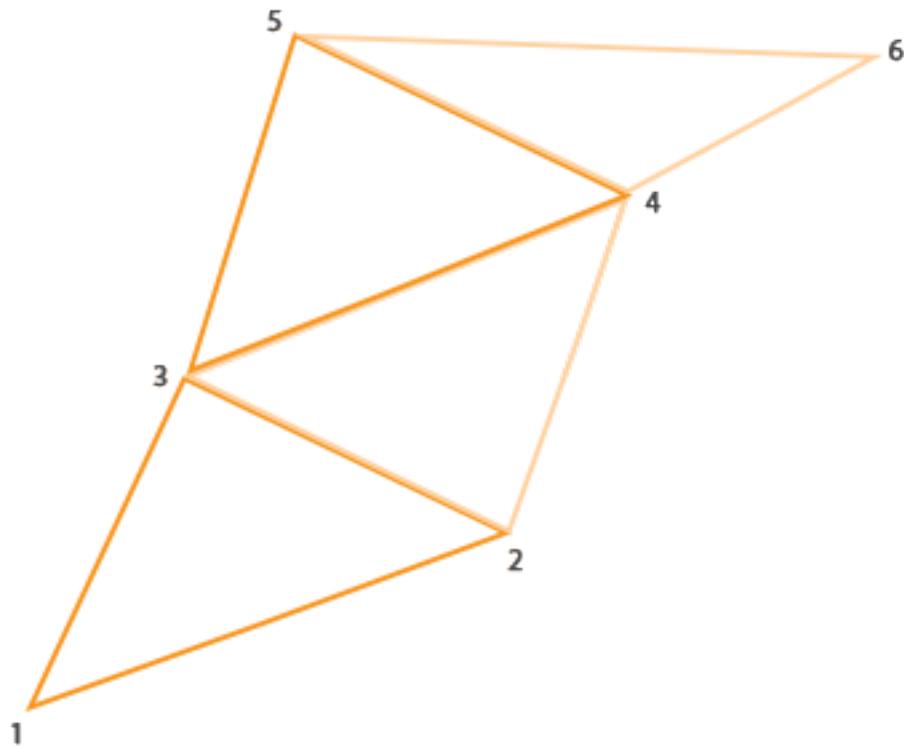


C'est exactement le même résultat que sans le *geometry shader* ! C'est un peu nul, je l'admet, mais, en réalité, si nous avons toujours nos points, cela signifie que le *geometry shader* fonctionne. Il est maintenant l'heure de faire des trucs amusants !

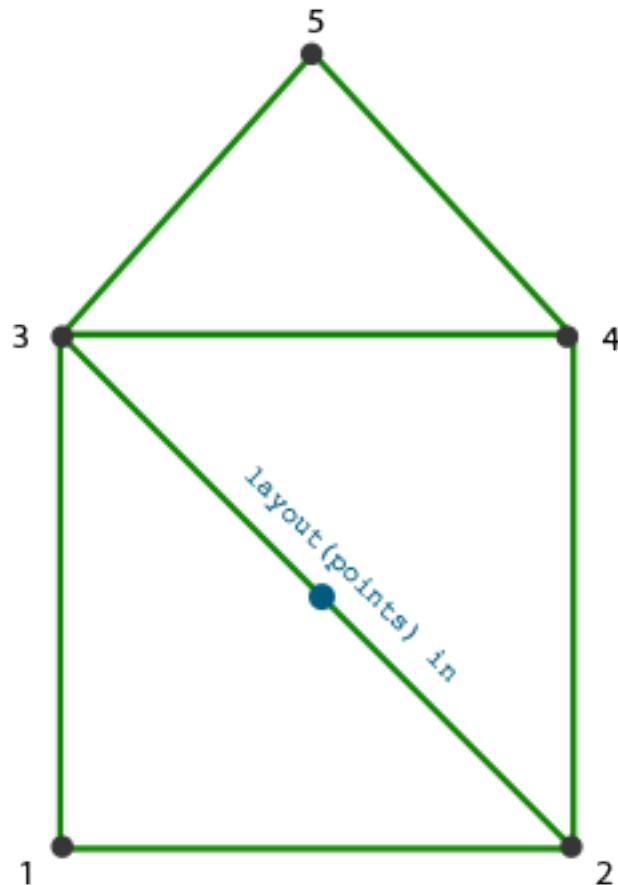
## IX-B - Construisons des maisons

Dessiner des points et des lignes n'est pas très intéressant, alors nous allons être un peu plus créatifs grâce au *geometry shader*. Nous allons dessiner une maison à l'emplacement de chaque point. Nous pouvons réussir cela en indiquant que le *geometry shader* doit produire une bande de triangles (`triangle_strip`) et dessiner trois triangles : deux pour le carré et un pour le toit.

Une bande de triangles en OpenGL est une façon efficace de dessiner des triangles en utilisant moins de sommets. Une fois que le premier triangle est dessiné, chaque sommet qui suit permet de générer un autre triangle à côté du premier : chaque trio de sommets adjacents forme un nouveau triangle. Si nous avons un total de six sommets qui forme une bande de triangles, nous obtiendrons les triangles suivants : (1,2,3), (2,3,4), (3,4,5) et (4,5,6), soit quatre triangles. Une bande de triangles nécessite au moins trois sommets et génère N-2 triangles ; avec six sommets, nous créons  $6-2 = 4$  triangles. L'image suivante schématisé cela :



En utilisant une bande de triangles comme sortie du *geometry shader*, nous pouvons facilement créer une maison avec trois triangles adjacents. L'image suivante montre quel ordre utiliser pour les sommets permettant d'obtenir les triangles dont nous avons besoin. Le point bleu est le point en entrée du *geometry shader* :



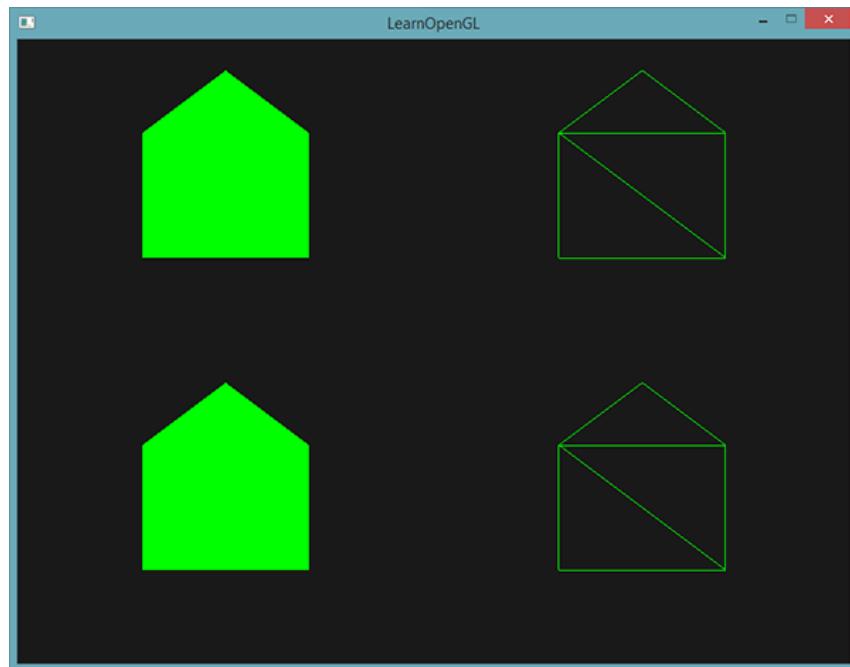
Cela se traduit par le *geometry shader* suivant :

```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 5) out;

void build_house(vec4 position)
{
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);      // 1:bottom-left
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);      // 2:bottom-right
    EmitVertex();
    gl_Position = position + vec4(-0.2, 0.2, 0.0, 0.0);      // 3 : haut gauche
    EmitVertex();
    gl_Position = position + vec4( 0.2, 0.2, 0.0, 0.0);      // 4 : haut droit
    EmitVertex();
    gl_Position = position + vec4( 0.0, 0.4, 0.0, 0.0);      // 5 :haut
    EmitVertex();
    EndPrimitive();
}

void main() {
    build_house(gl_in[0].gl_Position);
}
```

Le *geometry shader* génère cinq sommets, en décalant les coordonnées du point d'entrée, pour former une bande de triangles. La primitive résultante est ensuite traitée par le *fragment shader* qui s'exécute sur l'intégralité de la bande de triangles, donnant ainsi une maison verte pour chaque point dessiné :



Vous pouvez voir que chaque maison comporte trois triangles, tous dessinés en partant d'un point. Les maisons vertes sont un peu ennuyantes : améliorons donc le rendu et donnons une couleur unique à chaque maison. Pour ce faire, nous allons ajouter un attribut de couleur supplémentaire par sommet dans le *vertex shader*, le transférer au *geometry shader*, qui va l'envoyer au *fragment shader*.

Les données des sommets à jour sont comme suit :

```
float points[] = {
    -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, // haut gauche
    0.5f, 0.5f, 0.0f, 1.0f, 0.0f, // haut droit
    0.5f, -0.5f, 0.0f, 0.0f, 1.0f, // bas droit
    -0.5f, -0.5f, 1.0f, 1.0f, 0.0f // bas gauche
};
```

Ensuite, nous mettons à jour le *vertex shader* pour transférer la couleur au *geometry shader* grâce au bloc d'interface :

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out VS_OUT {
    vec3 color;
} vs_out;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    vs_out.color = aColor;
}
```

Nous devons déclarer le même bloc d'interface (avec un nom d'interface différent) dans le *geometry shader* :

```
in VS_OUT {
    vec3 color;
} gs_in[];
```

Comme le *geometry shader* agit sur un ensemble de sommets en entrée, les données en entrée sont dans un tableau, même si nous ne traitons qu'un seul sommet.

Nous n'avons pas nécessairement besoin d'utiliser un bloc d'interface pour transférer les données au geometry shader. Nous aurions pu écrire :

```
in vec3 vColor[];
```



Le vertex shader aurait pu transférer la couleur avec `out vec3 vColor`. Toutefois, les blocs d'interface sont plus simple à utiliser dans les shaders comme le geometry shader. Dans la pratique, les entrées du geometry shader peuvent être conséquentes et les regrouper dans un grand bloc d'interface est plus logique.

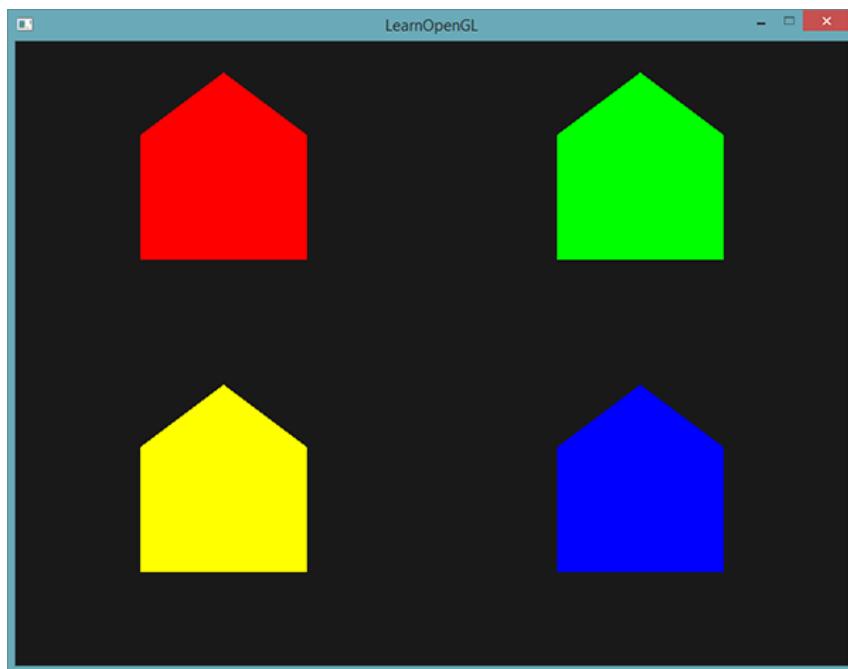
Ensuite, nous devons déclarer une couleur en sortie pour la passer au *fragment shader* :

```
out vec3 fColor;
```

Comme le *fragment shader* attend une seule couleur (interpolée), cela n'est pas logique de transférer plusieurs couleurs. Le vecteur `fColor` n'est donc pas un tableau, mais un simple vecteur. Lors de l'émission d'un sommet, chaque sommet stocke au moins une `fColor` pour l'exécution du *fragment shader*. Pour les maisons, nous allons donner comme valeur à `fColor` la première couleur qui a été émise par le *vertex shader* :

```
fColor = gs_in[0].color; // gs_in[0] comme il n'y a qu'un sommet en entrée
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0); // 1 : bas gauche
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0); // 2 : bas droit
EmitVertex();
gl_Position = position + vec4(-0.2, 0.2, 0.0, 0.0); // 3 : haut gauche
EmitVertex();
gl_Position = position + vec4( 0.2, 0.2, 0.0, 0.0); // 4 : haut droit
EmitVertex();
gl_Position = position + vec4( 0.0, 0.4, 0.0, 0.0); // 5 : haut
EmitVertex();
EndPrimitive();
```

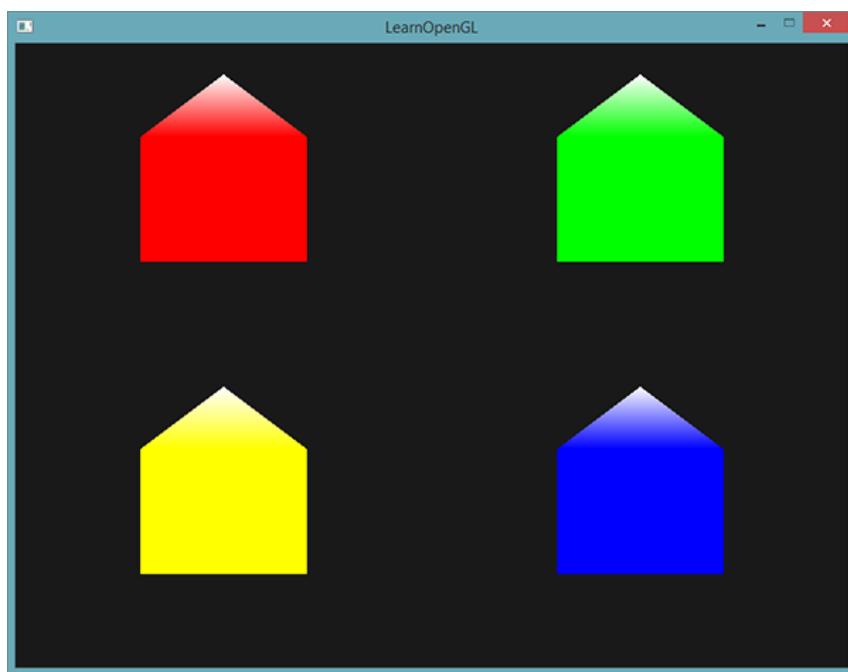
Tous les sommets émis contiennent la dernière valeur de `fColor`, égale à la couleur du sommet définie dans leurs attributs. Toutes les maisons ont maintenant une couleur différente :



Pour nous amuser, nous pouvons imaginer qu'il y a de la neige sur le toit des maisons en donnant une couleur au dernier sommet :

```
fColor = gs_in[0].color;
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0); // 1 : bas gauche
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0); // 2 : bas droit
EmitVertex();
gl_Position = position + vec4(-0.2, 0.2, 0.0, 0.0); // 3 : haut gauche
EmitVertex();
gl_Position = position + vec4( 0.2, 0.2, 0.0, 0.0); // 4 : haut droit
EmitVertex();
gl_Position = position + vec4( 0.0, 0.4, 0.0, 0.0); // 5 : haut
fColor = vec3(1.0, 1.0, 1.0);
EmitVertex();
EndPrimitive();
```

Le résultat donne quelque chose comme suit :



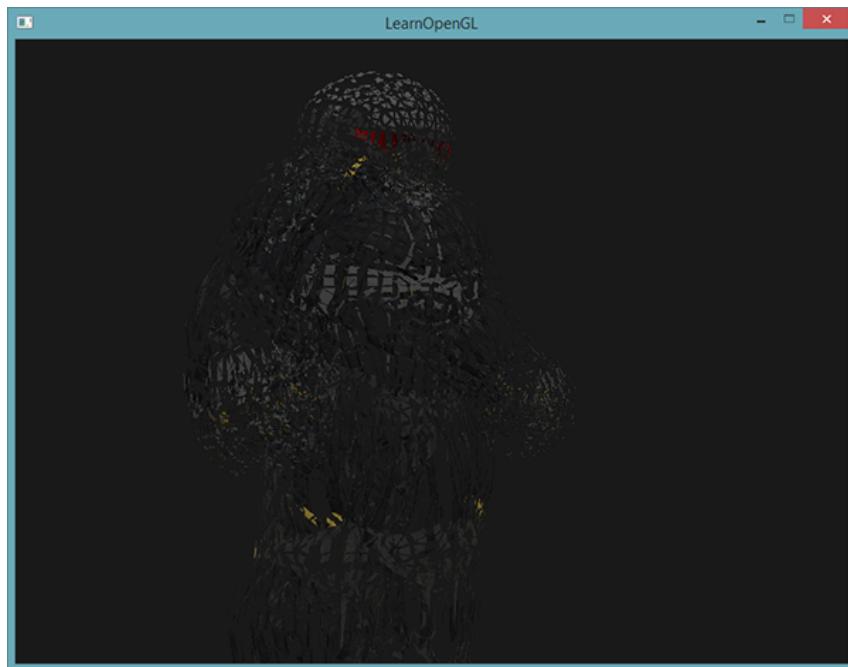
Vous pouvez trouver le code source [Source ici](#).

Vous pouvez voir que les *geometry shaders* permettent des choses créatives avec la plus simple des primitives. Comme la forme est générée dynamiquement sur le matériel ultra-rapide qu'est le GPU, c'est bien plus efficace que de définir des formes par vous-même dans les tampons de sommets. Les *geometry shaders* sont donc un outil d'optimisation pour les formes simples et répétitives comme les cubes d'un monde en voxel ou les brins d'herbe dans un champ.

## IX-C - Exploder les objets

Même si afficher des maisons, c'est cool, nous n'allons pas l'utiliser très souvent. C'est pourquoi nous allons monter d'un cran et exploser les objets ! Nous n'allons peut-être pas les utiliser beaucoup, mais cela montre la puissance des *geometry shaders*.

Lorsque nous parlons d'explorer un objet, nous n'allons pas exploser nos précieux sommets empaquetés, mais nous allons déplacer chaque triangle suivant la direction de leur normale sur une petite période de temps. Cet effet semble faire que l'objet explose suivant la direction de la normale. Cela donne le rendu suivant :



La bonne chose avec un tel effet provenant du *geometry shader* est que cela fonctionne sur tous les objets, quelle que soit leur complexité.

Comme nous allons déplacer tous les sommets dans la direction de la normale de leur triangle, nous devons calculer la normale. Il suffit de calculer un vecteur perpendiculaire à la surface du triangle, à laquelle nous pouvons accéder grâce aux trois sommets. Rappelez-vous ce que nous avons vus dans le [tutoriel sur les transformations](#) : vous pouvez obtenir le vecteur perpendiculaire à partir de deux vecteurs, a et b, qui sont parallèles à la surface du triangle. Cela se réalise grâce au produit vectoriel de ces deux vecteurs. Le *geometry shader* suivant fait exactement cela en obtenant la normale à partir des trois sommets en entrée :

```
vec3 GetNormal()
{
    vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position);
    vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position);
    return normalize(cross(a, b));
}
```

Ici, nous obtenons deux vecteurs a et b parallèles à la surface du triangle grâce à une soustraction. Soustraire deux vecteurs entre eux permet d'obtenir un vecteur qui est la différence. Comme les trois points forme un triangle, la soustraction de n'importe quels vecteurs entre eux donne un vecteur parallèle au plan. Notez que, si nous avions interverti a et b dans la fonction `cross()`, nous aurions obtenu une normale pointant dans la direction opposée. L'ordre est important ici !

Maintenant que nous savons calculer la normale, nous pouvons écrire une fonction `explode()` qui prend cette normale ainsi que la position d'un sommet. La fonction retourne un nouveau vecteur, résultant dans le déplacement du vecteur à la suivant la direction de la normale :

```
vec4 explode(vec4 position, vec3 normal)
{
    float magnitude = 2.0;
    vec3 direction = normal * ((sin(time) + 1.0) / 2.0) * magnitude;
    return position + vec4(direction, 0.0);
}
```

La fonction en elle-même n'est pas compliquée. La fonction `sin()` reçoit une variable `time` comme argument, représentant le temps et retourne une valeur entre -1,0 et 1,0. Comme nous ne voulons pas que l'objet implose, nous

modifions les valeurs retournées pour qu'elles soient comprises dans l'intervalle [0, 1]. La valeur ainsi obtenue est multipliée par la normale, ce qui donne ainsi une direction, qui est ensuite ajoutée à la position.

Le *geometry shader* produisant l'effet d'explosion est celui-ci :

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

in VS_OUT {
    vec2 texCoords;
} gs_in[];

out vec2 TexCoords;

uniform float time;

vec4 explode(vec4 position, vec3 normal) { ... }

vec3 GetNormal() { ... }

void main() {
    vec3 normal = GetNormal();

    gl_Position = explode(gl_in[0].gl_Position, normal);
    TexCoords = gs_in[0].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[1].gl_Position, normal);
    TexCoords = gs_in[1].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[2].gl_Position, normal);
    TexCoords = gs_in[2].texCoords;
    EmitVertex();
    EndPrimitive();
}
```

Notez que nous produisons aussi des coordonnées de texture appropriées avant d'émettre un sommet.

Aussi, n'oubliez pas de définir la variable `time` dans le code OpenGL :

```
shader.setFloat("time", glfwGetTime());
```

Le résultat donne un modèle 3D qui semble exploser continuellement et qui redevient normal. Bien que ce ne soit pas très pratique, cela montre une utilisation un peu plus avancée du *geometry shader*. Vous pouvez comparer votre code avec [Source celui-ci](#).

## IX-D - Visualiser les normales

Dans cette section, nous allons voir comment utiliser le *geometry shader* pour une chose utile : la visualisation des normales d'un objet. Lorsque vous programmez les *shaders* pour l'éclairage, vous allez probablement obtenir des résultats étranges. Une source classique pour les problèmes d'éclairage est un mauvais chargement des données des sommets, ce qui provoque une mauvaise définition des normales. Nous voulons donc une méthode pour vérifier que les normales sont correctes. Une bonne façon est de les afficher grâce au *geometry shader*.

L'idée est la suivante : nous dessinons d'abord la scène, sans *geometry shader*, puis nous la dessinons une deuxième fois, mais uniquement en affichant les normales générées par le *geometry shader*. Ce dernier prend en entrée un triangle et génère trois lignes à partir de ceux-ci, dans la direction de la normale : une normale pour chaque sommet. En pseudo-code, cela ressemblera à :

```
shader.use();
DrawScene();
normalDisplayShader.use();
```

```
DrawScene();
```

Cette fois, nous créons un *geometry shader* qui utilise les normales fournies par le modèle au lieu de les générer nous-mêmes. Pour gérer la mise à l'échelle et les rotations (à cause de la matrice de modèle et de vue), nous allons transformer les normales en premier avec la matrice de normale avant de passer en espace de coordonnées de l'écran (le *geometry shader* reçoit le vecteur de position dans l'espace de découpage, nous devons donc aussi transformer les normales afin de les avoir dans cet espace de coordonnées). Cela peut se faire dans le *vertex shader*:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out VS_OUT {
    vec3 normal;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    mat3 normalMatrix = mat3(transpose(inverse(view * model)));
    vs_out.normal = normalize(vec3(projection * vec4(normalMatrix * aNormal, 0.0)));
}
```

La normale en espace de découpage est ensuite passé à l'étape suivante du pipeline à travers le bloc d'interface. Le *geometry shader* prend chaque sommet (avec leur position et leur normale) et dessine une normale pour chaque position :

```
#version 330 core
layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
    vec3 normal;
} gs_in[];

const float MAGNITUDE = 0.4;

void GenerateLine(int index)
{
    gl_Position = gl_in[index].gl_Position;
    EmitVertex();
    gl_Position = gl_in[index].gl_Position + vec4(gs_in[index].normal, 0.0) * MAGNITUDE;
    EmitVertex();
    EndPrimitive();
}

void main()
{
    GenerateLine(0); // première normale
    GenerateLine(1); // deuxième normale
    GenerateLine(2); // troisième normale
}
```

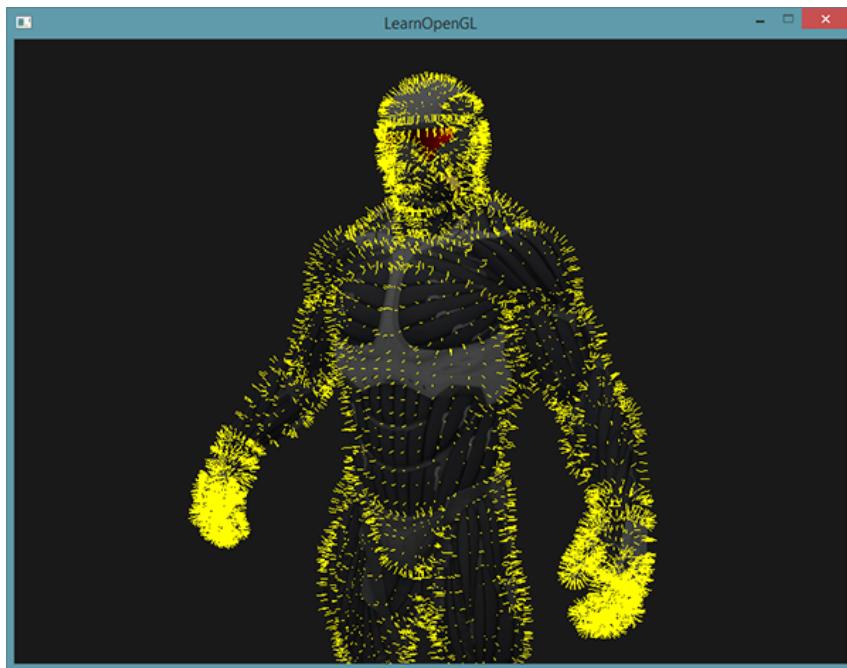
Le contenu du *geometry shader* doit être explicite. Notez que nous multiplions la normale avec le vecteur MAGNITUDE pour limiter la taille de la normale affichée (sinon, elle serait trop grande).

Comme la visualisation des normales est surtout utilisée pour le débogage, nous pouvons les afficher avec une seule couleur (ou de manière très colorée) avec l'aide du *fragment shader*:

```
#version 330 core
out vec4 FragColor;
```

```
void main()
{
    FragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```

L'affichage du modèle en deux passes avec le *shader* spécial de *visualisation des normales* devrait donner quelque chose comme ça :



Mis à part que la *nanosuit* ressemble maintenant à un homme chevelu avec des moufles, cela offre une méthode utile pour déterminer si les normales du modèle sont correctes. Vous pouvez imaginer qu'un tel *geometry shader* est très pratique pour ajouter de la fourrure aux objets.

Vous pouvez trouver le code source OpenGL [Source ici](#).

## IX-E - Remerciements

Ce tutoriel est une traduction réalisée par **Alexandre Laurent** dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

## X - Instanciation

Imaginons que vous avez une scène représentant de nombreux modèles, mais que la plupart utilisent les mêmes données de sommets. Par exemple, une scène avec des brins d'herbe : chaque brin est un petit modèle de quelques triangles. Vous voudriez probablement en dessiner beaucoup, votre scène contiendrait alors des milliers ou des dizaines de milliers de brins d'herbe à afficher à chaque image. Comme chaque brin contient quelques triangles, un brin s'affiche quasiment instantanément, mais l'ensemble de ces milliers d'appels de rendu réduiraient les performances de manière critique.

Si nous souhaitons afficher un grand nombre d'objets, nous aurions un code comme suit :

```
for(unsigned int i = 0; i < nombre_de_modele_a_dessiner; i++)
{
    DoSomePreparations(); // lier les VAO, les textures, définir les variables uniformes...
    glDrawArrays(GL_TRIANGLES, 0, nombre_de_sommets);
```

{}

Lors de l'affichage de beaucoup **d'instances** de votre objet, vous allez sûrement atteindre une baisse de performances à cause du grand nombre d'appels de rendu. En comparaison de l'affichage des sommets, indiquer au GPU de dessiner des données de sommets avec une fonction comme `glDrawArrays()` ou `glDrawElements()` consomme beaucoup de temps de calcul, car OpenGL doit faire un travail préparatoire avant de pouvoir dessiner les sommets (comme dire au GPU dans quel tampon lire, où trouver les attributs et toutes ces choses utilisant le bus CPU vers GPU, d'une lenteur inimaginable). Même si le rendu des sommets est ultra-rapide, le fait de donner des commandes de rendu au GPU ne l'est pas.

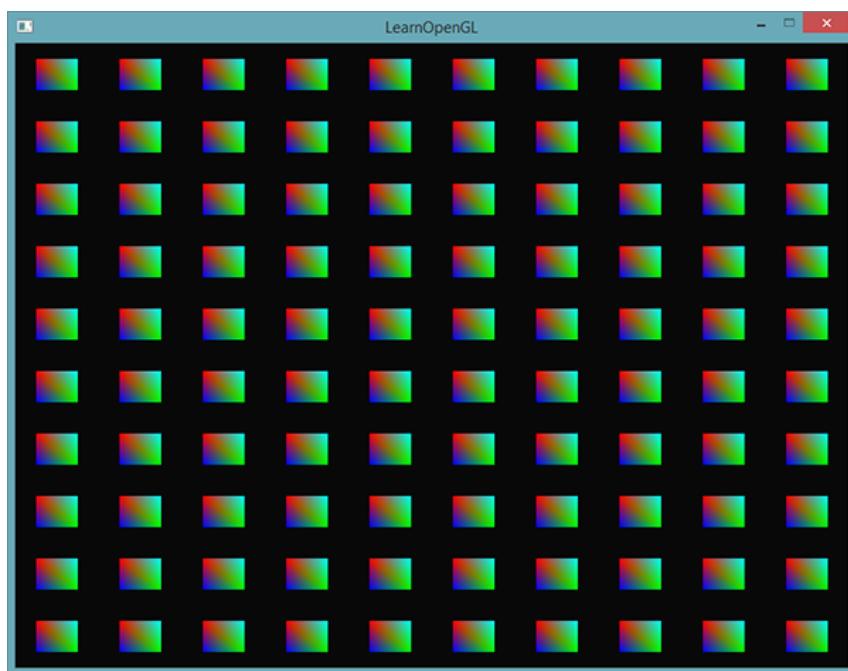
Il serait pratique si nous pouvions envoyer des données au GPU une seule fois et dire à OpenGL de dessiner les objets plusieurs fois avec un seul appel de rendu. C'est le principe de **l'instanciation**.

L'instanciation est une technique où nous dessinons plusieurs objets avec un seul appel de rendu, économisant des communications CPU vers GPU pour chaque rendu de cet objet. Celles-ci n'ont besoin d'être effectuées qu'une fois. Pour effectuer un rendu avec l'instanciation, nous devons juste changer nos appels de rendu pour passer de `glDrawArrays()` et `glDrawElements()` à `glDrawArraysInstanced()` et `glDrawElementsInstanced()`. Ces versions avec instanciation prennent un paramètre supplémentaire, le nombre d'instances (*instance count*) à afficher. Nous souhaitons donc envoyer toutes les données nécessaires au GPU une seule fois et dire au GPU comment il doit dessiner toutes ces instances en une fois. Le GPU va ensuite afficher ces instances sans devoir communiquer avec le CPU.

En elle-même, cette fonction est inutile. Le rendu d'un même objet un millier de fois est inutile si chaque objet est affiché exactement de la même façon et, donc à la même position ; nous voulons voir plus d'un objet ! Pour cela, le GLSL fournit une autre variable dans le *vertex shader* : `gl_InstanceID`.

Lors d'un rendu avec instanciation, la variable `gl_InstanceID`, démarrant à 0, est incrémentée à chaque instance. Si nous sommes en train d'afficher la 43e instance, la variable `gl_InstanceID` aurait pour valeur 42 dans le *vertex shader*. En ayant une valeur unique par instance, nous pouvons l'utiliser comme index pour un grand tableau de positions contenant la position dans le monde de chaque instance.

Pour comprendre le rendu instancié, nous allons prendre un exemple simple où nous affichons une centaine de carrés 2D en coordonnées normalisées avec un seul appel. Pour réussir cela, nous allons ajouter un petit décalage à chaque carré en indexant un tableau de variable uniforme de 100 vecteurs. Le résultat forme une grille remplissant l'intégralité de la fenêtre :



Chaque carré est composé de six sommets formant deux triangles. Chaque sommet contient une position 2D en coordonnée normalisée et une couleur. Ci-dessous le tableau des sommets utilisés pour cet exemple. Les triangles sont plutôt petits afin d'en faire rentrer beaucoup dans l'écran :

```
float quadVertices[] = {
    // positions          // couleurs
    -0.05f,  0.05f,  1.0f,  0.0f,  0.0f,
    0.05f, -0.05f,  0.0f,  1.0f,  0.0f,
    -0.05f, -0.05f,  0.0f,  0.0f,  1.0f,
    -0.05f,  0.05f,  1.0f,  0.0f,  0.0f,
    0.05f, -0.05f,  0.0f,  1.0f,  0.0f,
    0.05f,  0.05f,  0.0f,  1.0f,  1.0f
};
```

La couleur des carrés est obtenue grâce au *fragment shader* qui reçoit la couleur transférée par le *vertex shader* :

```
#version 330 core
out vec4 FragColor;

in vec3 fColor;

void main()
{
    FragColor = vec4(fColor, 1.0);
}
```

Rien de nouveau, mais le *vertex shader* contient quelque chose d'intéressant :

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out vec3 fColor;

uniform vec2 offsets[100];

void main()
{
    vec2 offset = offsets[gl_InstanceID];
    gl_Position = vec4(aPos + offset, 0.0, 1.0);
    fColor = aColor;
}
```

Ici, nous avons un tableau de variables uniformes nommé `offset` qui contient 100 vecteurs pour les décalages. Dans le *vertex shader*, nous récupérons le vecteur de décalage pour chaque instance grâce à la variable `gl_InstanceID`. Si nous dessinons 100 carrés, nous obtiendrons 100 carrés placés à différents endroits grâce à ce simple *vertex shader*.

Nous devons définir les décalages, que nous calculons dans une boucle avant la boucle d'affichage :

```
glm::vec2 translations[100];
int index = 0;
float offset = 0.1f;
for(int y = -10; y < 10; y += 2)
{
    for(int x = -10; x < 10; x += 2)
    {
        glm::vec2 translation;
        translation.x = (float)x / 10.0f + offset;
        translation.y = (float)y / 10.0f + offset;
        translations[index++] = translation;
    }
}
```

Ainsi, nous créons 100 vecteurs permettant de positionner les objets sur une grille de taille 10x10. En plus de la génération du tableau, nous devons transférer les données au tableau de variables uniformes du *vertex shader* :

```
shader.use();
for(unsigned int i = 0; i < 100; i++)
{
    stringstream ss;
    string index;
    ss << i;
    index = ss.str();
    shader.setVec2(("offsets[" + index + "]").c_str(), translations[i]);
}
```

Grâce à ce code, nous transformons le compteur *i* de la boucle en une chaîne de caractères afin de l'utiliser comme nom de variable uniforme. Pour chaque élément du tableau de variables uniformes *offsets*, nous lui assignons son vecteur correspondant.

Maintenant que nous avons préparé la scène, nous pouvons afficher les carrés. Pour les dessiner, nous appelons *glDrawArraysInstanced()* ou *glDrawElementsInstanced()*. Comme nous n'utilisons pas de tampon d'éléments, nous allons utiliser la fonction *glDrawArraysInstanced()* :

```
glBindVertexArray(quadVAO);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);
```

Les paramètres de la fonction *glDrawArraysInstanced()* sont exactement les mêmes que *glDrawArrays()* sauf pour le dernier paramètre, indiquant le nombre d'instances à dessiner. Comme nous voulons afficher 100 carrés sur une grille de 10x10, nous passons 100 en argument. En exécutant ce code, vous devriez obtenir une image familière de 100 carrés colorés.

## X-A - Tableaux instanciés

Bien que l'implémentation précédente fonctionne pour ce cas précis, nous souhaitons instancier plus de 100 instances (ce qui est classique). Nous allons certainement atteindre une **limite** sur le nombre de variables uniformes que nous pouvons passer au *shader*. Une autre alternative est d'appeler un tableau instancié (*instanced array*) défini comme un attribut de sommets (et nous permettant de passer plus de données) qui n'est mis à jour que lorsque le *vertex shader* affiche une nouvelle instance.

Avec les attributs de sommets, chaque exécution du *vertex shader* provoquera la récupération du prochain attribut de sommet correspondant au sommet actuel. Lors de la définition d'un attribut de sommets tel qu'un tableau instancié, le *vertex shader* met à jour uniquement le contenu de l'attribut de sommet par instance et non plus par sommet. Cela permet d'utiliser un attribut de sommet standard pour les données des sommets et un tableau instancié pour les données des instances.

Pour vous donner un exemple d'un tableau instancié, nous allons reprendre l'exemple précédent et passer les décalages à travers un tableau instancié. Nous devons mettre à jour le *vertex shader* pour ajouter un nouvel attribut :

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aOffset;

out vec3 fColor;

void main()
{
    gl_Position = vec4(aPos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

Nous n'utilisons plus la variable `gl_InstanceID` et nous pouvons directement utiliser l'attribut `offset` sans avoir à l'indexer à partir d'un quelconque tableau.

Comme le tableau instancié est un attribut de sommet, tout comme les variables `position` et `color`, nous devons stocker son contenu dans un tampon de sommet et le configurer avec un pointeur d'attribut de sommets. Nous commençons par stocker le tableau `translations` (de la section précédente) dans un nouveau tampon :

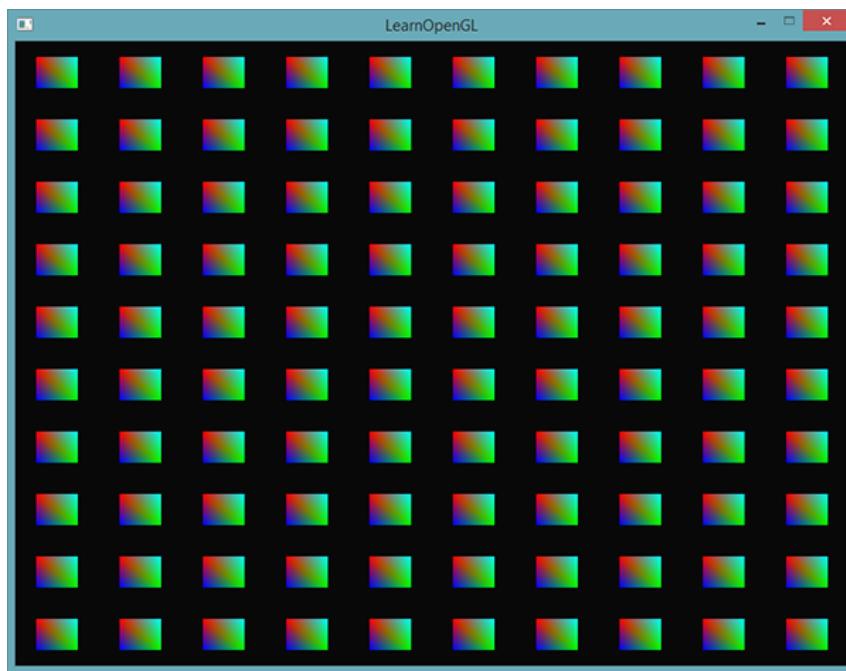
```
unsigned int instanceVBO;
 glGenBuffers(1, &instanceVBO);
 glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec2) * 100, &translations[0], GL_STATIC_DRAW);
 glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Ensuite, nous devons définir le pointeur d'attribut de sommet et activer cet attribut :

```
 glEnableVertexAttribArray(2);
 glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
 glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
 glBindBuffer(GL_ARRAY_BUFFER, 0);
 glVertexAttribDivisor(2, 1);
```

Ce code est intéressant grâce à l'appel de la fonction `glVertexAttribDivisor()`. Celle-ci indique à OpenGL quand mettre à jour le contenu de l'attribut de sommet avec le prochain élément. Le premier paramètre est l'attribut de sommets et le second est le diviseur d'attribut (*attribute divisor*). Par défaut, le diviseur est 0, qui indique à OpenGL de mettre à jour l'attribut de sommet à chaque itération dans le *vertex shader*. En définissant l'attribut à 1, nous indiquons à OpenGL que nous voulons mettre à jour le contenu de l'attribut de sommets lorsque nous démarrons une nouvelle instance. En le définissant à 2, le contenu est mis à jour toutes les deux instances et ainsi de suite. Ici, nous le mettons à 1 pour indiquer à OpenGL que l'attribut de sommets à l'emplacement 2 est un tableau instancié.

Si nous affichons les carrés maintenant, nous obtiendrions le même résultat :

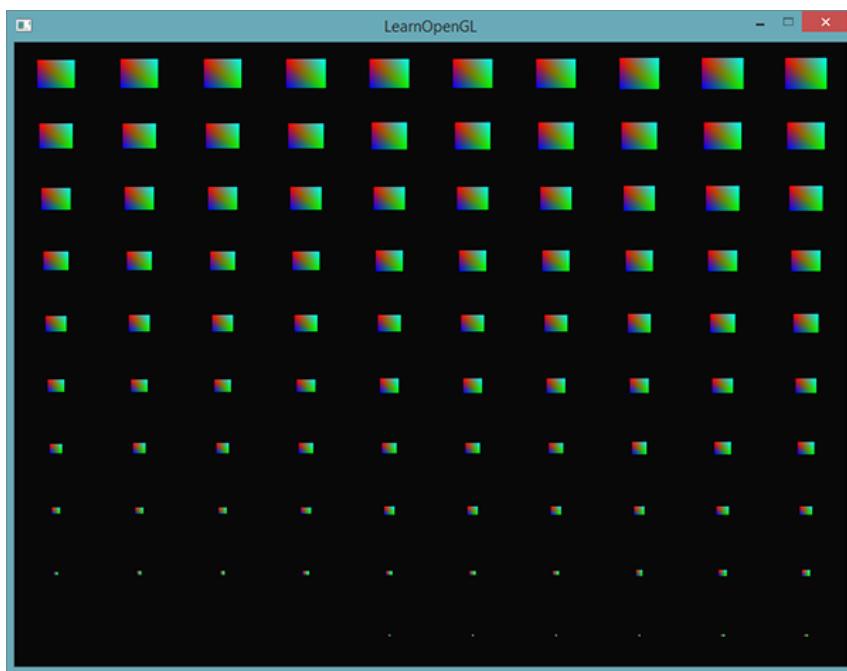


C'est exactement le même résultat que précédemment, mais, cette fois, nous avons utilisé les tableaux instanciés, ce qui nous permet de passer plus de données (autant que nous avons de mémoire) au *vertex shader* pour le rendu instancié.

Pour nous amuser, nous pouvons aussi réduire lentement la taille de chaque carré en partant du haut et de la droite avec `gl_InstanceID`. Pourquoi pas ?

```
void main()
{
    vec2 pos = aPos * (gl_InstanceID / 100.0);
    gl_Position = vec4(pos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

Le résultat est que les premières instances du carré sont dessinées très petit et plus nous progressons dans le processus de rendu, plus la variable `gl_InstanceID` s'approche de 100 et donc plus les carrés retrouvent leur taille originale. C'est parfaitement possible d'utiliser les tableaux instanciés en même temps que la variable `gl_InstanceID`.



Si vous ne voyez toujours pas comment le rendu instancié fonctionne ou que vous souhaitez voir comment le tout s'assemble, vous pouvez trouver le code source de l'application [ici](#).

Même si c'est amusant, ces exemples ne sont pas de bons exemples pour l'instanciation. Oui, cela donne un bon aperçu du fonctionnement de l'instanciation, mais celle-ci est vraiment utile lors de l'affichage d'un très grand nombre d'objets similaires, ce que nous n'avons jamais fait jusqu'à présent. Pour cette raison, nous allons nous aventurer dans l'espace et voir la vraie puissance du rendu instancié.

## X-B - Affichage d'un champ d'astéroïdes

Imaginez une scène où nous avons une très grande planète entourée par un énorme champ d'astéroïdes. Un tel anneau d'astéroïdes doit contenir des milliers et des centaines de milliers de cailloux : il devient donc très rapidement impossible à afficher pour une carte graphique décente. Le scénario s'avère être le terrain propice pour le rendu instancié, car les astéroïdes peuvent provenir d'un seul modèle. Chaque astéroïde contient seulement quelques petites variations à travers la matrice de transformations, une matrice unique à chaque astéroïde.

Pour montrer l'impact du rendu instancié, nous allons d'abord afficher la scène avec des astéroïdes autour de la planète, mais sans rendu instancié. La scène ne contiendra qu'un modèle de planète qui peut être téléchargé [ici](#) et un grand ensemble d'astéroïdes positionnés autour de la planète. Le modèle pour l'astéroïde peut être téléchargé [ici](#).

Grâce au code vu dans le tutoriel sur le [chargement des modèles](#), nous chargeons les modèles pour cette scène.

Pour réussir cet effet, nous allons générer la matrice de transformation pour chaque astéroïde. La matrice de transformation est créée en déplaçant l'astéroïde dans l'anneau – puis en ajoutant un petit déplacement pour obtenir un effet plus naturel. Ensuite, nous appliquons un petit redimensionnement et une petite rotation. Le résultat est une matrice de transformation pour placer chaque astéroïde quelque part autour de la planète, tout en ajoutant un petit plus pour donner un aspect naturel et unique par rapport aux autres astéroïdes. Le résultat est un anneau plein d'astéroïdes où chaque astéroïde est différent des autres.

```

unsigned int amount = 1000;
glm::mat4 *modelMatrices;
modelMatrices = new glm::mat4[amount];
rand(glfwGetTime()); // initialiser la graine de la génération aléatoire
float radius = 50.0;
float offset = 2.5f;
for(unsigned int i = 0; i < amount; i++)
{
    glm::mat4 model;
    // 1. translation : déplacer autour d'un cercle de rayon 'radius' dans l'espace [-offset,
    offset]
    float angle = (floati / (float)amount * 360.0f;
    float displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float x = sin(angle) * radius + displacement;
    displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float y = displacement * 0.4f; // garder la taille du champ d'astéroïdes comparé à la largeur
    de x et z
    displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float z = cos(angle) * radius + displacement;
    model = glm::translate(model, glm::vec3(x, y, z));

    // 2. mise à l'échelle : facteur 0.05 et 0.25f
    float scale = (rand() % 20) / 100.0f + 0.05;
    model = glm::scale(model, glm::vec3(scale));

    // 3. rotation : ajoute une rotation aléatoire autour d'un axe de rotation pris
    (semi) aléatoirement
    float rotAngle = (rand() % 360);
    model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

    // 4. ajout à la liste des matrices
    modelMatrices[i] = model;
}

```

Ce morceau de code peut sembler un peu intimidant, mais nous ne faisons que transformer la position x et z de l'astéroïde autour d'un cercle suivant un rayon défini par la variable radius. De plus, l'astéroïde est légèrement déplacé autour du cercle d'un décalage entre -offset et offset. L'impact sur le déplacement en y est légèrement réduit pour donner un aspect plus plat à l'anneau d'astéroïde. Ensuite, nous appliquons un redimensionnement et une rotation puis nous stockons la matrice obtenue dans modelMatrices de la taille amount. Nous générerons ici 1 000 matrices, une par astéroïde.

Après avoir chargé les modèles de la planète et de l'astéroïde et compilé l'ensemble des *shaders*, le code du rendu ressemble à :

```

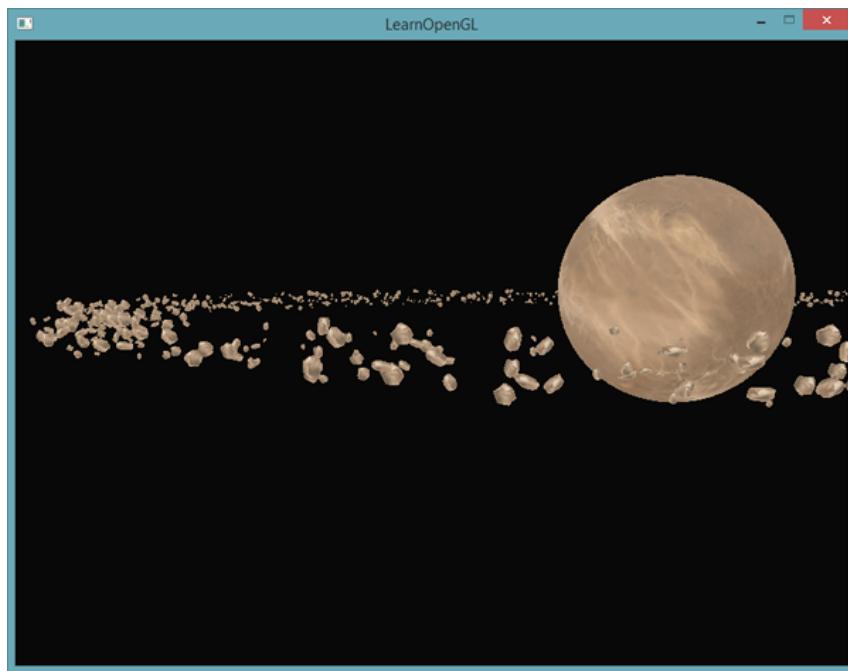
// dessiner la planète
shader.use();
glm::mat4 model;
model = glm::translate(model, glm::vec3(0.0f, -3.0f, 0.0f));
model = glm::scale(model, glm::vec3(4.0f, 4.0f, 4.0f));
shader.setMat4("model", model);
planet.Draw(shader);

// dessiner les astéroïdes
for(unsigned int i = 0; i < amount; i++)
{
    shader.setMat4("model", modelMatrices[i]);
    rock.Draw(shader);
}

```

Premièrement, nous dessinons la planète que nous déplaçons et redimensionnons pour que cela corresponde mieux à la scène, puis nous dessinons les 1000 astéroïdes avec les matrices que nous avons calculées. Toutefois, avant de dessiner chaque astéroïde, nous définissons la matrice de transformation dans le *shader*.

Le résultat donne une scène spatiale où nous pouvons voir un anneau d'astéroïdes presque naturel autour de la planète :



La scène nécessite au total 1001 appels de rendu par image, dont 1000 pour les astéroïdes. Vous pouvez trouver le code source de cette scène [Source ici](#).

Aussitôt que nous augmentons le nombre, nous pouvons remarquer que la scène n'est plus fluide et que le nombres d'images par seconde diminue fortement. Dès que nous passons à 2000 astéroïdes, la scène devient tellement lente qu'il est impossible de s'y déplacer.

Essayons maintenant d'afficher la même scène, mais en utilisant le rendu instancié. Nous allons d'abord adapter le *vertex shader* :

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in mat4 instanceMatrix;

out vec2 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    gl_Position = projection * view * instanceMatrix * vec4(aPos, 1.0);
    TexCoords = aTexCoords;
}
```

Nous n'allons plus utiliser la variable uniforme : nous la remplaçons par une variable de type `mat4` comme attribut de sommet. Ainsi, on pourra stocker un tableau instancié pour les matrices de transformation. Toutefois, nous utilisons, comme attribut de sommets, un type de données plus grand qu'un `vec4` et qui s'utilise un peu différemment. Le nombre maximum de données permise par un attribut de sommet correspond à `vec4`. Comme un `mat4` n'est que

quatre `vec4`, nous avons réservé quatre attributs de sommets pour cette matrice. Sachant que le premier `vec4` est à l'emplacement 3, les colonnes de la matrice seront aux emplacements 3, 4, 5 et 6.

Ensuite, nous devons définir chaque pointeur d'attributs pour ces quatre attributs de sommets et les configurer comme tableau instancié :

```
// tampons de sommets
unsigned int buffer;
glGenBuffers(1, &buffer);
 glBindBuffer(GL_ARRAY_BUFFER, buffer);
 glBufferData(GL_ARRAY_BUFFER, amount * sizeof(glm::mat4), &modelMatrices[0], GL_STATIC_DRAW);

for(unsigned int i = 0; i < rock.meshes.size(); i++)
{
    unsigned int VAO = rock.meshes[i].VAO;
    glBindVertexArray(VAO);
    // attributs de sommet
    GLsizei vec4Size = sizeof(glm::vec4);
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)0);
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(vec4Size));
    glEnableVertexAttribArray(5);
    glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(2 * vec4Size));
    glEnableVertexAttribArray(6);
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(3 * vec4Size));

    glVertexAttribDivisor(3, 1);
    glVertexAttribDivisor(4, 1);
    glVertexAttribDivisor(5, 1);
    glVertexAttribDivisor(6, 1);

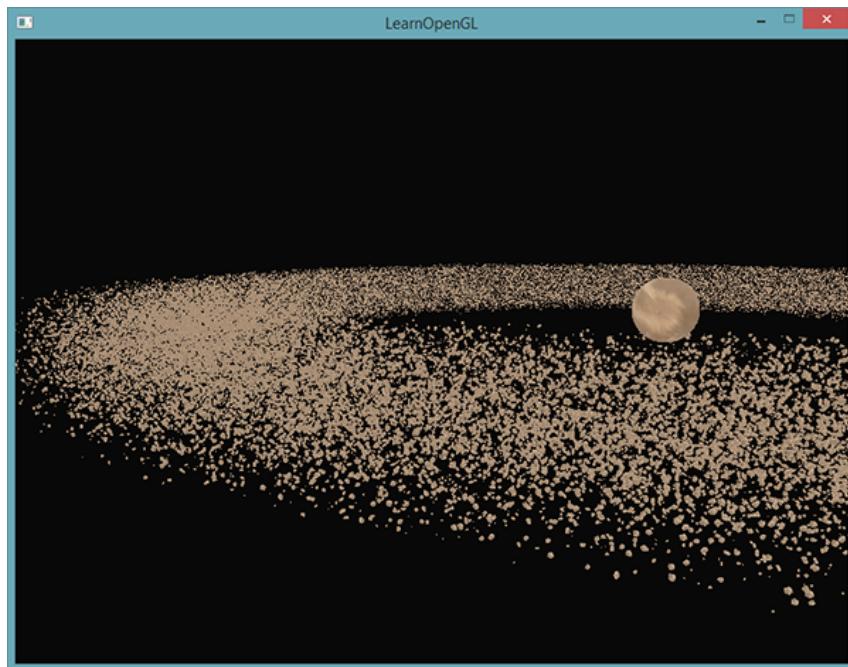
    glBindVertexArray(0);
}
```

Notez que vous pouvez tricher en déclarant le VAO de la classe `Mesh` comme variable publique au lieu d'une variable privée, ce qui permet d'y accéder avec un tableau de sommets. Ce n'est pas la solution la plus propre, mais juste une modification pour correspondre à ce tutoriel. Mis à part cette petite astuce, le code doit être évident. Nous avons défini comment OpenGL doit interpréter le tampon pour chaque attribut de sommets de la matrice et pour chaque attribut de sommets, un tableau instancié.

Ensute, nous prenons le VAO du modèle une nouvelle fois et nous les dessinons avec la fonction `glDrawElementsInstanced()` :

```
// dessiner les astéroïdes
instanceShader.use();
for(unsigned int i = 0; i < rock.meshes.size(); i++)
{
    glBindVertexArray(rock.meshes[i].VAO);
    glDrawElementsInstanced(
        GL_TRIANGLES, rock.meshes[i].indices.size(), GL_UNSIGNED_INT, 0, amount
    );
}
```

Maintenant, nous dessinons le même nombre d'astéroïdes que dans l'exemple précédent, mais cette fois, à l'aide du rendu instancié. Le résultat doit être similaire, mais dès que vous commencez à augmenter le nombre grâce à la variable `amount` vous voyez l'effet de l'instanciation. Sans rendu instancié, nous étions capable d'afficher 1000 à 1500 astéroïdes sans problème. Avec le rendu instancié nous pouvons définir cette valeur à 100 000. Le modèle d'astéroïde possède 576 sommets : la scène contient 57 millions de sommets, affichés à chaque image, sans perte de performance !



Cette image a été affichée avec 100 000 astéroïdes dans un rayon de 150,0 et un décalage de 25,0. Vous pouvez trouver le code source de cette démo de rendu instancié [Source ici.](#)

**i** *Sur certaines machines, un nombre de 100 000 astéroïdes peut être trop haut, essayez donc des valeurs plus faibles pour obtenir un rendu acceptable.*

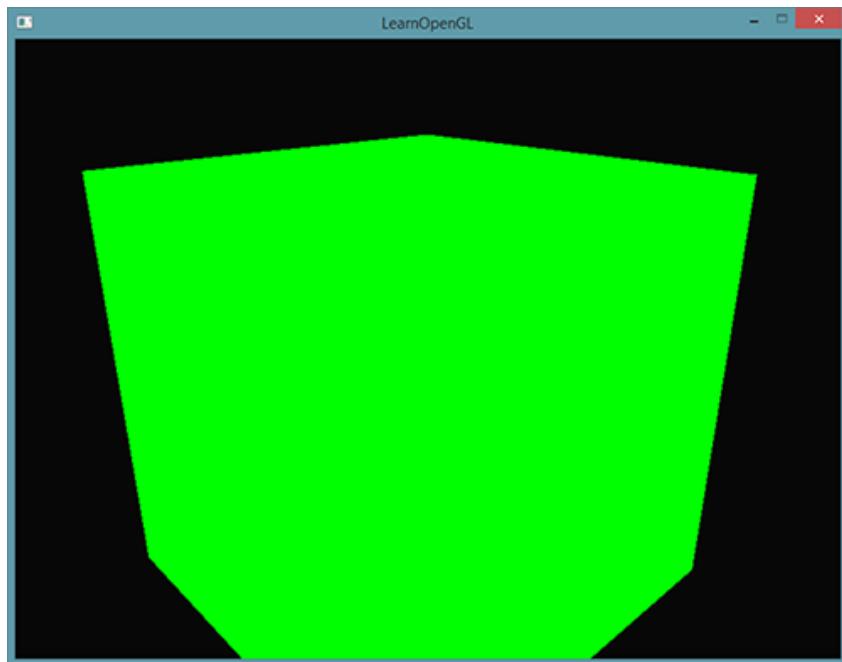
Comme vous pouvez le voir, avec le bon environnement, linstanciation peut permettre une énorme différence dans les capacités de rendu de votre carte graphique. Pour cela, linstanciation est couramment utilisée pour lherbe, la flore, les particules et les scènes comme celle-ci – soit, pour nimporte quelle scène ayant un grand nombre de formes se répétant.

## X-C - Remerciements

Ce tutoriel est une traduction réalisée par **Alexandre Laurent** dont loriginal a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

## XI - Anticrénelage

À un certain moment de notre aventure dans le monde du rendu, vous avez dû rencontrer des bords irréguliers sur vos modèles. La raison de ces **bords escarpés** est liée au fonctionnement du *rasterizer* transformant les données des sommets en fragments dans la scène. Voici un exemple dune bordure escarpée qui peut être vue rien quavec un cube :



Même s'il n'est pas visible immédiatement, si vous regardez en détail les bordures du cube, vous pouvez voir un motif irrégulier. Si nous zoomons, nous pourrions voir ceci :



Ce n'est pas vraiment ce que nous voulons dans notre version finale. Cet effet où l'on voit les pixels sur les bordures est appelé effet d'escalier (*aliasing*). Il y a quelques techniques justement nommées techniques de diminution de l'effet d'escalier ou d'anticrénelage (*antialiasing*) permettant de réduire cet effet et de produire des bordures plus douces.

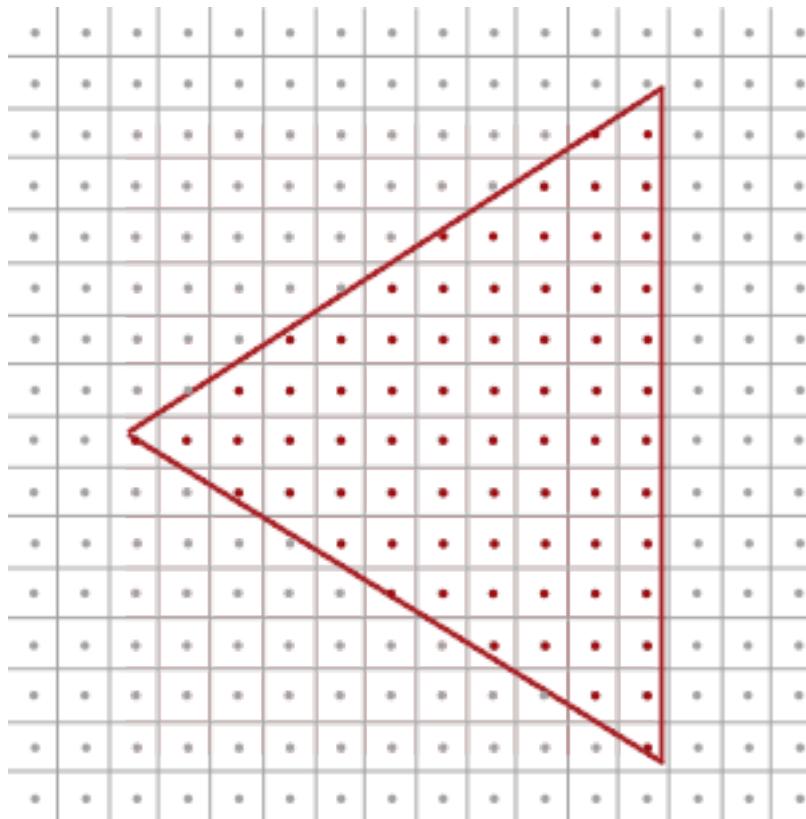
Premièrement, nous avons une technique nommée **super sample anti-aliasing** (SSAA) qui utilise temporairement une résolution supérieure pour afficher la scène (super-échantillonnage), puis, lors de l'affichage dans le tampon d'image, la résolution est réduite pour revenir à la normale. Cette augmentation de résolution permet d'éviter l'effet d'escalier sur les bordures. Même si cela offre une solution au problème, la contrepartie en performance est énorme, car la carte graphique doit afficher beaucoup plus de fragments que d'habitude. Cette technique a toutefois eu un petit moment de succès.

Cette technique a donné naissance à des techniques modernes comme l'anticrénelage multi-échantillonné (**multisample anti-aliasing**, MSAA) qui emprunte certains concepts du SSAA, tout en offrant une approche plus efficace. Dans ce tutoriel, nous allons grandement parler de cette technique qui est incluse à OpenGL.

## XI-A - Multiéchantillonnage

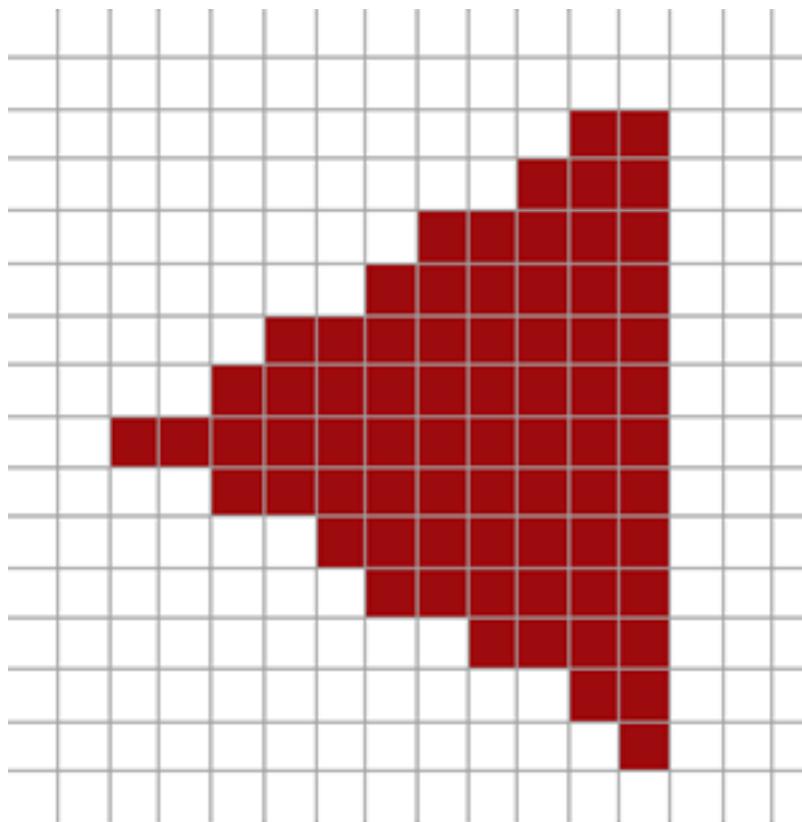
Pour comprendre ce qu'est le multiéchantillonnage et comment il fonctionne dans la réduction de l'effet d'escalier, nous devons d'abord nous plonger dans le fonctionnement du *rasterizer* d'OpenGL.

Le rasterizer est la combinaison d'algorithme et de processus qui sont effectués entre le traitement de vos sommets et le *fragment shader*. Le *rasterizer* prend tous les sommets appartenant à une primitive et les transforme en un ensemble de fragments. Les coordonnées des sommets peuvent théoriquement être n'importe lesquelles, mais ce n'est pas le cas pour les fragments, car ils sont liés à la résolution de la fenêtre. Il n'y aura donc jamais de correspondance parfaite entre les coordonnées des sommets et les fragments. Le *rasterizer* doit donc déterminer à quel fragment/coordonnées correspond un sommet.



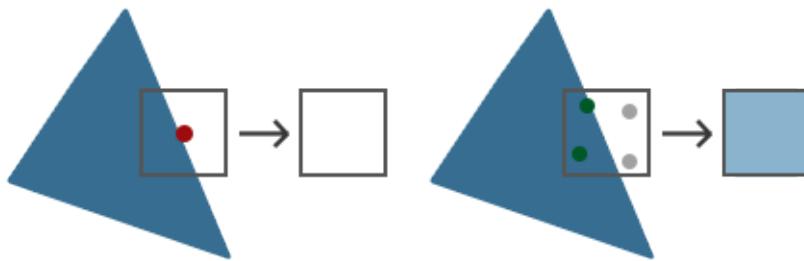
Ici, nous voyons une grille de pixels à l'écran, où le centre de chaque pixel contient un **point d'échantillonnage** qui est utilisé pour déterminer si le pixel est dans le triangle ou non. Les points en rouges sont ceux dans le triangle et un fragment sera généré pour ce pixel. Même si certaines parties du triangle couvre tel ou tel pixel, si le point d'échantillonnage n'est pas dans le triangle, il n'y aura pas de *fragment shader* exécuté pour ce pixel.

Vous pouvez certainement commencer à comprendre l'origine de l'effet d'escalier. Vous obtenez un triangle comme ceci sur votre écran :



Comme le nombre de pixels à l'écran est limité, certains seront affichés en bordure du triangle, alors que d'autres non. Le résultat est que nous obtenons des primitives avec des bordures non douces, provoquant l'effet d'escalier vu précédemment.

Avec le multi-échantillonnage, au lieu d'avoir un seul point d'échantillonnage pour déterminer si le pixel est dans le triangle, nous utilisons plusieurs points (devinez d'où vient le nom). Au lieu d'un seul point au centre de chaque pixel, nous allons placer quatre **suréchantillons** suivant un certain motif et les utiliser pour déterminer si le pixel est dans le triangle. Cela signifie que la taille du tampon de couleurs est augmentée du nombre de suréchantillons que nous utilisons par pixel.



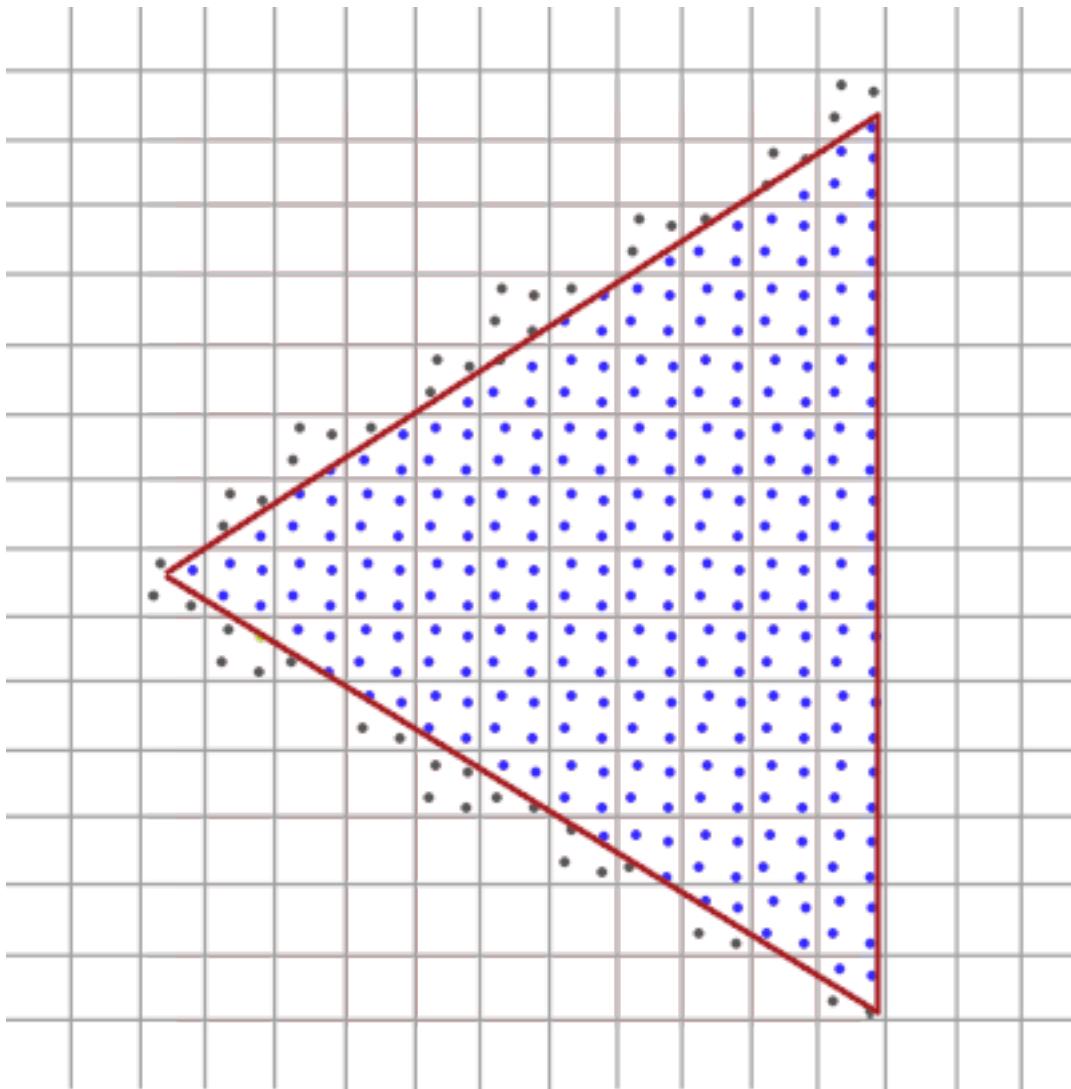
Du côté gauche de l'image, vous pouvez voir ce que nous aurions eu pour déterminer si le pixel est dans le triangle. Ce pixel n'exécutera pas de *fragment shader* (et restera donc blanc), car son point d'échantillonnage n'est pas couvert par le triangle. Du côté droit de l'image, chaque pixel est suréchantillonné, c'est-à-dire, contient quatre points d'échantillonnage. Ici, nous pouvons voir que seuls deux des points d'échantillonnage sont dans le triangle.

**i** *Le nombre de points d'échantillonnage peut être celui que vous souhaitez afin d'obtenir une meilleure précision.*

C'est ici que le suréchantillonnage devient intéressant. Nous avons déterminé que deux suréchantillons sont dans le triangle. Il reste à déterminer la couleur du pixel. Notre hypothèse initiale est que nous exécutons le *fragment shader* pour chaque suréchantillon couvert, puis nous calculons la moyenne des couleurs des suréchantillons du pixel. Dans ce cas, nous exécutons le *fragment shader* deux fois et interpolons les données des sommets pour chaque suréchantillon et stockons le résultat. Heureusement, ce n'est pas ainsi que cela fonctionne, car cela voudrait dire que nous exécutons beaucoup plus de *fragment shader* que sans le suréchantillonnage, ce qui réduirait les performances.

En réalité, le MSAA fait que le *fragment shader* n'est exécuté qu'une seule fois par pixel (pour chaque primitive), peu importe le nombre de suréchantillons couverts par le triangle. Le *fragment shader* est exécuté avec les données des sommets interpolés au centre du pixel et la couleur obtenue est stockée dans chaque suréchantillon couvert. Une fois que les suréchantillons du tampon de couleurs sont remplis pour toutes les primitives que nous avons dessinées, une moyenne des couleurs est effectuée pour chaque pixel. Comme deux des quatre échantillons sont couverts par le triangle, la couleur du pixel est une moyenne entre la couleur du triangle et la couleur stockée dans les deux points d'échantillonnage (dans ce cas, une couleur claire), ce qui donne une couleur bleutée claire.

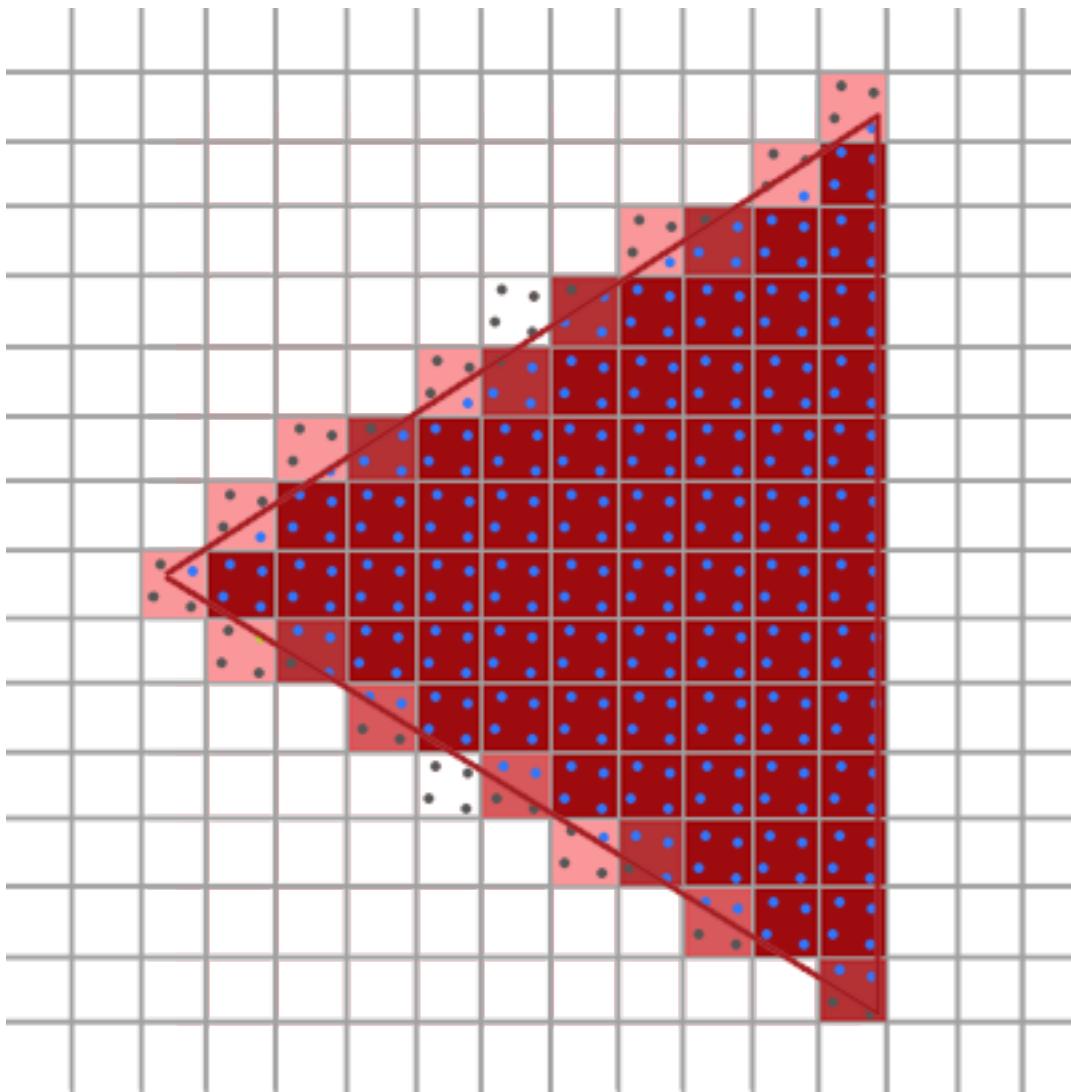
Le tampon de couleurs résultant où toutes les bordures des primitives sont plus douces. Voyons voir comment le résultat du suréchantillonage sur le triangle précédent :



Chaque pixel contient quatre suréchantillons (les échantillons non utiles sont cachés) : les suréchantillons qui se trouvent dans le triangle sont bleus, ceux qui sont hors du triangle sont gris. À l'intérieur du triangle, tous les pixels exécutent le *fragment shader* une seule fois et le résultat sera stocké dans les quatre suréchantillons. Sur les bordures du triangle, le résultat du *fragment shader* ne sera stocké que dans certains suréchantillons. Suivant le nombre de

suréchantillons à l'intérieur du triangle, la couleur du pixel sera déterminée grâce à la couleur du triangle et à la couleur des échantillons stockés.

Grossièrement, plus il y a de points suréchantillonés dans le triangle, plus la couleur du pixel sera proche de celle du triangle. Si nous remplissons les pixels avec la couleur correspondante, nous obtiendrons l'image suivante :



Pour chaque pixel, moins il y a de suréchantillons dans le triangle, moins le pixel aura la couleur du triangle. Les bordures du triangle sont entourées par des pixels de couleurs plus clairs, rendant ainsi les bords plus doux.

Non seulement les couleurs sont affectées par le suréchantillonnage, mais aussi par la profondeur et le test de pochoir. Pour le test de profondeur, celle-ci est interpolée pour chaque suréchantillon avant d'exécuter le test. Pour le test de pochoir, chaque valeur est stockée pour chaque suréchantillon et non pour chaque pixel. Cela veut donc dire que la taille de ces tampons est augmentée pour contenir les suréchantillons.

Nous avons donc vu un aperçu du fonctionnement du suréchantillonnage comme technique de suppression de l'effet d'escalier. La logique du *rasterizer* est plus compliquée que ce nous avons vu, mais vous devriez être capable de comprendre la logique.

## XI-B - MSAA dans OpenGL

Si vous souhaitez utiliser le MSAA en OpenGL, vous devez utiliser un tampon de couleurs qui peut stocker plus d'une couleur par pixel (car le suréchantillonnage nécessite de stocker une couleur par point d'échantillonnage). Nous avons donc besoin d'un nouveau type de tampon qui peut stocker un certain nombre de suréchantillons : le tampon de suréchantillonnage (*multisample buffer*).

La plupart des systèmes de fenêtrage sont capables de fournir un tampon de suréchantillonnage à la place du tampon de couleurs par défaut. GLFW nous donne aussi cette possibilité et nous devons juste l'indiquer à GLFW, ainsi que le nombre d'échantillons à la place du tampon de couleurs, grâce à la fonction `glfwWindowHint()` avant de créer une fenêtre :

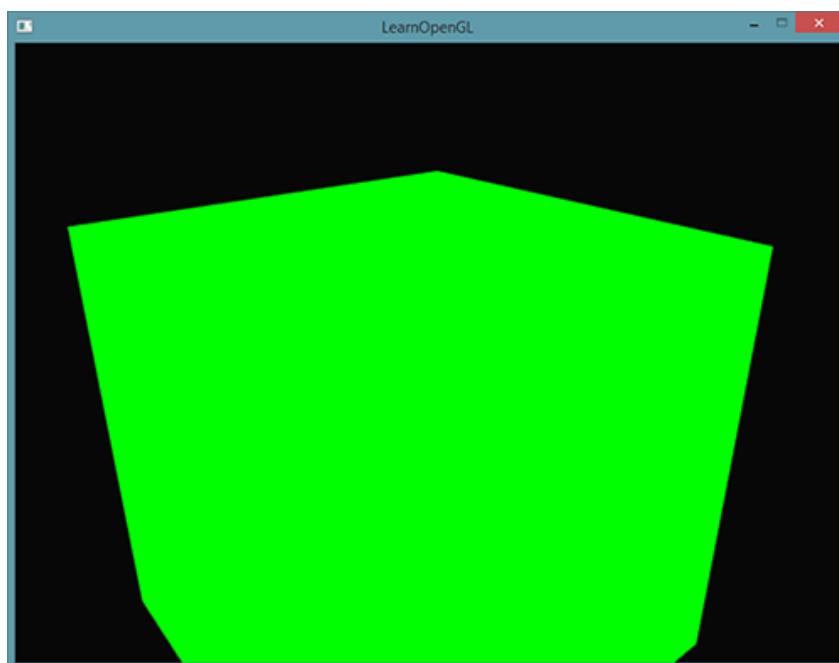
```
glfwWindowHint(GLFW_SAMPLES, 4);
```

Lorsque nous appelons la fonction `glfwCreateWindow()`, la fenêtre de rendu est créée, mais cette fois avec un tampon de couleurs contenant quatre suréchantillons par coordonnée à l'écran. GLFW crée aussi un tampon de profondeur et de pochoir avec quatre suréchantillons par pixel. Cela veut dire que la taille des tampons est augmentée d'un facteur quatre.

Maintenant que nous avons demandé des tampons suréchantillonnés à GLFW, nous devons activer le suréchantillonnage en appelant la fonction `glEnable()` avec `GL_MULTISAMPLE`. Avec la plupart des pilotes, le suréchantillonnage est activé par défaut, c'est donc un peu redondant, mais c'est toujours mieux de l'activer manuellement. C'est tout pour l'activation du suréchantillonnage.

```
glEnable(GL_MULTISAMPLE);
```

Une fois que le tampon d'image par défaut est attaché à des tampons suréchantillonnés, il ne nous reste plus qu'à activer le suréchantillonnage en appelant `glEnable()` et c'est bon. Comme les algorithmes de suréchantillonnage sont implémentés dans le *rasterizer* de votre pilote OpenGL, il n'y a rien que nous puissions faire. Si nous affichons maintenant le cube vert, nous verrions des bordures plus douces :



Évidemment, ce conteneur a des bords plus doux et il en sera de même avec les autres objets de votre scène. Vous pouvez trouver le code de cet exemple [Source ici](#).

## XI-C - MSAA hors écran

Comme c'est GLFW qui crée les tampons suréchantillonnés, l'activation du MSAA est plutôt facile. Si nous souhaitons utiliser notre propre tampon d'image, pour un quelconque rendu hors écran, nous devons générer le tampon suréchantillonné nous-même.

Il y a deux façons de créer des tampons suréchantillonnés à attacher au tampon d'image : les textures et les tampons de rendu, comme ce que nous avons vu dans le [tutoriel sur les tampons d'image](#).

### XI-C-1 - Attache d'une texture suréchantillonnée

Pour créer une texture qui supporte le stockage de plusieurs points d'échantillonnage, nous utilisons la fonction `glTexImage2DMultisample()` avec la cible `GL_TEXTURE_2D_MULTISAMPLE` :

```
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, tex);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, samples, GL_RGB, width, height, GL_TRUE);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);
```

Le deuxième argument définit maintenant le nombre d'échantillons que nous souhaitons pour la texture. Si le dernier argument est `GL_TRUE`, l'image utilisera les mêmes échantillons et le même nombre de suréchantillons pour chaque texel.

Pour attacher une texture suréchantillonnée à un tampon d'image, nous utilisons `glFramebufferTexture2D()` mais avec le type de texture `GL_TEXTURE_2D_MULTISAMPLE` :

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D_MULTISAMPLE, tex, 0);
```

Le tampon d'image est maintenant lié à un tampon de couleurs suréchantillonné correspondant à une texture.

### XI-C-2 - Tampon de rendu suréchantillonné

Comme pour les textures, la création des tampons de rendu suréchantillonnés n'est pas très difficile. C'est même plutôt simple, sachant que nous devons juste appeler la fonction `glRenderbufferStorageMultisample()` au lieu de `glRenderbufferStorage()` lorsque nous spécifions le tampon de rendu (actuellement lié) :

```
glRenderbufferStorageMultisample(GL_RENDERBUFFER, 4, GL_DEPTH24_STENCIL8, width, height);
```

La seule chose qui a changé ici est le paramètre supplémentaire après la cible du tampon de rendu, qui indique le nombre d'échantillons (ici, 4).

### XI-C-3 - Rendu dans un tampon d'image suréchantillonné

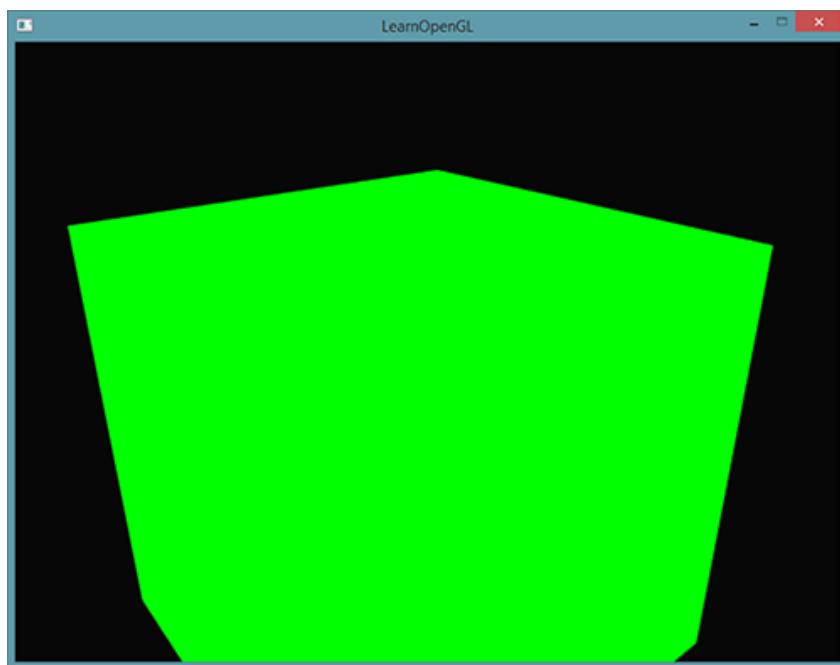
Le rendu dans un tampon d'image suréchantillonné s'effectue automatiquement. Chaque fois que nous dessinons quelque chose alors que ce tampon d'image est lié, le *rasterizer* tiendra compte des opérations de suréchantillonnage. Nous avons donc un tampon de couleurs suréchantillonné et/ou un tampon de profondeur et de pochoir. Toutefois, le tampon suréchantillonné est un peu spécial, car nous ne pouvons pas l'utiliser dans toutes les opérations, par exemple pour de l'échantillonnage dans un *shader*.

Une image suréchantillonnée contient plus d'informations qu'une image classique et nous devons donc en réduire ou changer sa résolution. Le changement de résolution d'une image suréchantillonnée se fait généralement avec la fonction `glBlitFramebuffer()`. Cette dernière permet de copier une région d'un tampon d'image vers une autre tout en résolvant le suréchantillonnage.

La fonction `glBlitFramebuffer()` transfère une région source indiquée par quatre coordonnées écran vers une région cible aussi définie par quatre coordonnées écran. Rappelez-vous du [tutoriel sur les tampons d'image](#), si nous lions `GL_FRAMEBUFFER` comme cible de tampon, cela lie aussi les cibles du tampon d'image en lecteur et en écriture. Nous pouvons aussi lier ces cibles individuellement, en liant les tampons d'image aux cibles `GL_READ_FRAMEBUFFER` et `GL_DRAW_FRAMEBUFFER` respectivement. La fonction `glBlitFramebuffer()` lit ces deux cibles pour déterminer qui est la source et qui est la destination. Nous pouvons ensuite transférer le tampon d'image suréchantillonné vers l'écran en copiant l'image vers le tampon d'image par défaut :

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, multisampledFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

Si ensuite nous affichons l'application, nous obtiendrons la même sortie que sans le tampon d'image : un cube vert rendu avec du MSAA et ayant des bords moins escarpés.



Vous pouvez trouver le code source [Source ici](#).

Que faire si nous voulons utiliser la texture provenant du tampon d'image suréchantillonné afin de faire des choses comme du post-traitement ? Nous ne pouvons pas utiliser directement la texture suréchantillonnée dans le *fragment shader*. Ce que nous pouvons faire, par contre, c'est copier le tampon suréchantillonné dans un autre FBO avec une texture non suréchantillonnée attachée. Ensuite, nous pourrions utiliser le tampon de couleurs attaché pour le post-traitement. Cela veut dire que nous devrions générer un nouveau FBO qui agit uniquement comme un tampon d'image intermédiaire pour changer la résolution du tampon suréchantillonné. Ce processus ressemble à cela en pseudo-code :

```
unsigned int msFBO = CreateFBOWithMultiSampledAttachments();
// créer ensuite un autre FBO avec une texture comme attache de couleurs
...
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, screenTexture, 0);
...
while (!glfwWindowShouldClose(window))
{
    ...

    glBindFramebuffer(msFBO);
    ClearFrameBuffer();
    DrawScene();
}
```

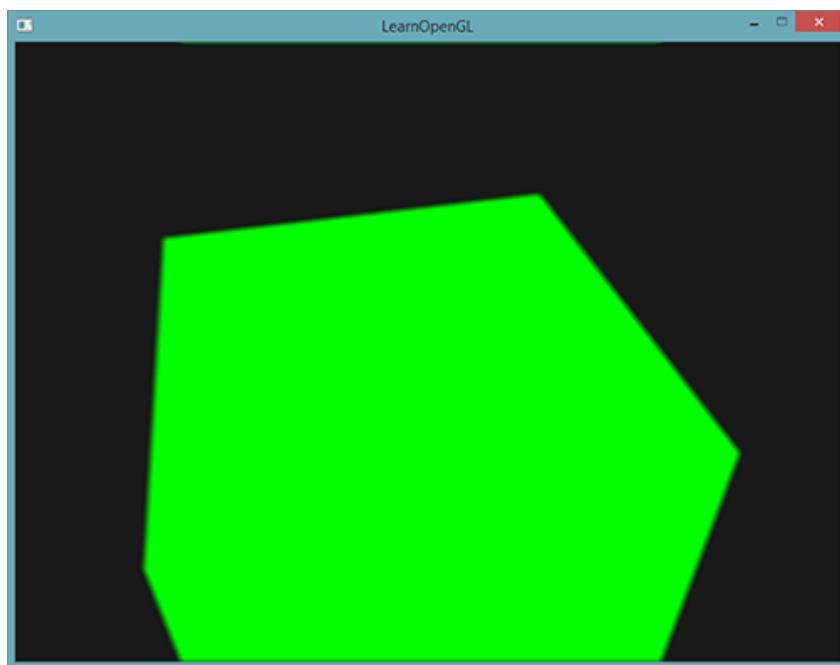
```

// maintenant, change la résolution du tampon suréchantillonné et place le résultat dans le
FBO intermédiaire
glBindFramebuffer(GL_READ_FRAMEBUFFER, msFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, intermediateFBO);
glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);
// maintenant, la scène est stockée dans une image 2D : on peut l'utiliser dans le post-
traitement
glBindFramebuffer(GL_FRAMEBUFFER, 0);
ClearFrameBuffer();
glBindTexture(GL_TEXTURE_2D, screenTexture);
DrawPostProcessingQuad();

...
}

```

Nous pouvons maintenant implémenter les effets de post-traitement du [tutoriel des tampons d'image](#) pour créer des effets intéressants avec une texture de la scène avec (pratiquement) plus de bordure escarpées. Avec le filtre de noyau de flou, cela donne :



**i** Comme la texture est une texture normale avec juste un point échantillonné, certains filtres de post-traitement comme celui de la détection de bordure produisent à nouveau des bordures escarpées. Pour contrer cela, vous pouvez flouter la texture après ou créer votre propre algorithme pour supprimer les effets d'escalier.

Vous pouvez voir que, lorsque vous voulez combiner le suréchantillonnage avec un rendu hors-écran, vous devez prendre en compte plus de choses. Toutes ces opérations sont nécessaires, car cela augmente grandement la qualité de la scène. Notez que l'activation du suréchantillonnage réduit la performance de votre application suivant le nombre d'échantillons utilisés. Au moment de l'écriture de ce tutoriel, l'utilisation de quatre échantillons pour le MSAA est courant.

## XI-D - Algorithme personnalisé de suppression de l'effet d'escalier

Il est aussi possible de passer la texture suréchantillonnée au *shader* sans faire de traitement. Le GLSL nous donne l'option d'échantillonner une texture par suréchantillon afin de créer nos propres algorithmes de suppression de l'effet d'escalier, ce qui est courant dans les grandes applications graphiques.

Pour récupérer la couleur par suréchantillon, vous devez définir l'échantillonneur comme un `sampler2DMS` à la place du classique `sampler2D` :

```
uniform sampler2DMS screenTextureMS;
```

Avec la fonction `texelFetch()`, il est possible de récupérer la couleur par échantillon :

```
vec4 colorSample = texelFetch(screenTextureMS, TexCoords, 3); // 4e suréchantillon
```

Nous n'entrerons pas dans les détails de la création d'une technique de suppression d'effet d'escalier, mais donnons simplement des pointeurs sur la manière d'implémenter une fonctionnalité comme celle-ci.

## XI-E - Remerciements

Ce tutoriel est une traduction réalisée par **Alexandre Laurent** dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).