[Les Frame Buffer Objects](http://fr.openclassrooms.com/informatique/cours/developpez-vos-applications-3d-avec-opengl-3-3/les-frame-buffer-objects)

## Définitions

## Les différents types de buffer

Avant de nous intéresser au sujet principal, nous allons faire un petit retour en arrière sur une fonction que nous utilisons depuis pas mal de temps déjà : la fonction **glClear**():

// Nettoyage de l'écran

glClear(GL\_COLOR\_BUFFER | GL\_DEPTH\_BUFFER);

Celle-ci permet de nettoyer les buffers (zones mémoires ou tampons si vous préférez) qu'ils lui sont donnés en paramètres. Par exemple, nous lui demandons ici de nettoyer le Color ainsi que le **Depth Buffer**, le premier correspondant simplement aux pixels affichés sur notre fenêtre et le second à la position des objets 3D les uns par rapport aux autres pour pouvoir les cacher si besoin.

Il existe encore un autre buffer, ce qui est un peu normal vu que c'est celui qui est le moins utilisé. Il s'agit du **Stencil Buffer** ou en français **tampon du pochoir**.

Le **Stencil Buffer** permet de faire exactement la même chose. Par exemple, au lieu d'afficher une texture carrée représentant un buisson, vous pouvez lui appliquer un pochoir (donc utiliser le **Stencil Buffer**) pour n'afficher que les pixels qui vous intéressent de façon à ne pas vous retrouver avec un buisson carré mais avec un buisson "réaliste".

Ce qu'il faut retenir c'est qu'il existe **3** types de buffers :

* **Color** : qui contient la couleur de chaque pixel de votre fenêtre
* **Depth** : qui permet à OpenGL de gérer la profondeur et de cacher les objets
* **Stencil** : qui permet de filtrer un rendu avec des pochoirs

### Les Frame Buffers

Pour donner une définition simple d'un **FBO**, on pourrait dire qu'il s'agit d'un ***écran caché*** qui utilise les **3** types de buffers que nous avons vus à l'instant :



L’utilisation du terme **d'écran caché** c'est par les **FBO** permettent de faire ce que l'on appelle un rendu **off-screen** (en dehors de l'écran). C'est-à-dire qu'au lieu d'afficher votre scène dans votre fenêtre, vous le ferrez dans un endroit à part dans la carte graphique comme s'il y en avait une deuxième à l'intérieure. Le rendu sera exactement le même sauf que vous ne le verrez pas directement.

À quoi ça sert d'afficher quelque chose si on ne le voit pas ?

Les **FBO** étant associés aux shaders, ils permettent de faire plein d'effets réalistes tels que :

* **Le flou** : pour la vitesse ou l'éblouissement
* **Les reflets** : pour les miroirs et l'eau
* **Les ombres dynamiques** : qui sont modifiées à chaque fois que leur "modèle" bouge
* **La capture vidéo** : pour filmer un endroit tout en étant dans un autre (caméra de surveillance, télévision, etc.)
* ...

**Petit détail:** votre écran lui-même est considéré comme un **Frame Buffer**. Il reprend donc toutes les caractéristiques dont nous venons de parler, sauf pour les effets réalistes parce qu'il est impossible de modifier les pixels une fois qu'ils sont affichés sur l'écran.

L'ensemble de ces canaux sont appelés un **Frame Buffer** (littéralement un tampon d'image). Par défaut, le canal « couleur » est copié vers l'écran.

Dans les jeux vidéos, certaines techniques demandent à lire les pixels affichés à l'écran afin des les réutiliser comme source d'information (telle que la technique de calcul des ombres, appelées « shadow mapping ») ou d'appliquer des effets supplémentaires sur l'image (post process). Pour ce faire, il était nécessaire de demander à OpenGL de lire le tampon mémoire de sortie et de renvoyer les informations au programme OpenGL. Cette technique est coûteuse, car elle repose sur la vitesse de transfert GPU/CPU. Une autre technique mise en place à l'époque a été d'utiliser des pixels buffers (pbuffers) afin d'avoir un espace de stockage pour le rendu. Le problème de ce dernier est que les pbuffers étaient gérés par le système de fenêtrage, forçant l'utilisateur à créer différents contextes pour les différents formats de texture voulus. De plus, le changement entre la fenêtre principale et ce second buffer pouvait être lent.

Afin de répondre à une demande de plus en plus importante, le consortium autour d'OpenGL a décidé de faire une extension simple et optimisée permettant de faire du rendu **hors écran**.

L'extension **GL\_ARB\_framebuffer\_object**, apporte de nouveaux éléments au sein d'OpenGL. Le premier, nommé « **image attachable** au **render buffer** » correspond à un objet pouvant être attaché à un framebuffer afin qu'il soit utilisé comme source et destination du framebuffer. L'information de remplissage est soit la couleur, la profondeur ou le stencil. Actuellement, il existe deux objets de ce type :

* les textures ;
* les renderbuffers.

Les buffers de profondeur et les stencil buffer possédaient des formats de pixels non représentables dans de simple textures. Afin de conserver ces buffers à la fin du rendu, les renderbuffers ont été intégrés. Il est possible de les utiliser lorsque nous n'avons pas besoin de texture dans la suite de notre rendu.

Ainsi, pour faire un rendu vers une texture, il suffit d'attacher cette texture à l'un des canaux et d'effectuer le rendu. Ensuite, comme toute autre texture, elle est utilisable dans le fragment shader.

Le second élément est le **FrameBuffer** **Object**. C'est un objet OpenGL permettant de gérer facilement les collections d'attachements. Grâce à cet objet il est facile d'avoir plusieurs ensembles de textures de destination ou encore de changer simplement les textures attachées.

## Lexique

**Buffer logique** : un des canaux du framebuffer (couleurs, profondeur, stencil).

**Frame buffer** : un ensemble de buffers logiques et d'états définissant l'emplacement où le rendu OpenGL sera effectué.

**Frame buffer object** : objet représentant un frame buffer.

**RenderBuffer** : nouvel espace de stockage contenant un simple tableau 2D de pixels pouvant être utilisé comme cible de rendu.

**Renderbuffer object** : objet représentant un renderbuffer.

**Attacher** : le fait de lier un objet avec un autre. Dans le cas du frame buffer, le fait d'attacher un objet au framebuffer permettra à celui-ci de l'utiliser comme endroit où faire le rendu.

**Passe**(s) : nous parlons de passe, le fait d'effectuer une opération de dessin sur la carte graphique.

**Rendu direct** : le rendu direct est un rendu en une seule et unique passe. C'est le rendu que nous utilisons très couramment afin d'effectuer un affichage simple des objets.

**Rendu différé** : le rendu différé est un rendu constitué de plusieurs passes. Généralement, dans les premières passes, la scène est rendue dans des textures qui seront utilisées dans les passes suivantes.

## Est-ce vraiment plus rapide ?

Les framebuffer objects ont l'avantage de permettre de conserver et d'utiliser les informations calculées par le GPU dans la mémoire du GPU. En effet, avec les anciennes méthodes, il fallait faire des allers-retours entre le programme OpenGL et le GPU. Maintenant, les données restent sur le GPU, les programmes ne subissent plus des latences à cause du bus de données.

Pour certaines techniques de rendu, telle que le calcul des lumières, il est plus efficace de conserver toutes les informations de la scène (position dans le monde, couleurs, normales) dans des textures et des les réutiliser pour toutes les sources de lumière. De plus, si vous souhaitez effectuer des effets sur l'image finale, il vous suffit de dessiner la scène dans une texture et de lui appliquer l'effet à l'aide d'un shader.

## Fonctionnement

Un FBO est un objet OpenGL qui correspond au buffer à remplir en sortie du pipeline graphique. Celui-ci possède plusieurs canaux de sortie dans lesquels le fragment shader écrira. Parmi ceux-ci, nous comptons :

* les canaux de couleurs (lorsque nous dessinons à l'écran, nous n'en utilisons qu'un seul, mais les FBO permettent d'en avoir plusieurs) ;
* le canal de profondeur ;
* le canal stencil.

**Remarque** : Le nombre de canaux de couleurs dépend de la carte graphique. Le minimum défini par la spécification est quatre.

Pour chaque canal, vous pouvez soit attacher une texture, soit un render buffer.

Voici un petit schéma récapitulatif du fonctionnement :



Lorsque le FBO est actif, le shader écrira dans les canaux. Pour les canaux attachés à des texture ou des render buffer, les informations iront directement dans ces objets.

## La bibliothèque OpenGL et les FBO

### 1. Evolution des versions

Au début, les FBO ont été ajoutés à OpenGL 2.x sous la forme d'une extension appelée « **GL\_ARB\_framebuffer\_object** ». Depuis OpenGL 3, les frames buffers sont directement intégrés comme fonctionnalités du cœur d'OpenGL. Entre les deux versions, seul les noms des fonctions et énumérations changent en perdant le suffixe ARB depuis la normalisation de la fonctionnalité. Cet article est écrit en OpenGL 3.

### 2. Énumérations

Une série d'énumérations a été créée pour permettre la gestion des FBO. Voici un tableau des valeurs les plus utilisées. Consulter la documentation pour plus d'informations :

|  |  |  |
| --- | --- | --- |
| Valeurs | Description | Fonctions |
| GL\_FRAMEBUFFERGL\_READ\_FRAMEBUFFERGL\_DRAW\_FRAMEBUFFER | Description du buffer à utiliser. GL\_FRAMEBUFFER est une union de GL\_READ\_FRAMEBUFFER et GL\_DRAW\_FRAMEBUFFER. | GlFramebufferTextureglBindFramebufferglFramebufferRenderbufferglGetFramebufferAttachmentParameterivglCheckFramebufferStatus |
| GL\_RENDERBUFFER | Désigne un render buffer. | BindRenderbufferRenderbufferStorageGetRenderbufferParameterivGetFramebufferAttachmentParameteriv |
| RENDERBUFFER\_WIDTHRENDERBUFFER\_HEIGHTRENDERBUFFER\_INTERNAL\_FORMATRENDERBUFFER\_RED\_SIZERENDERBUFFER\_GREEN\_SIZE RENDERBUFFER\_BLUE\_SIZERENDERBUFFER\_ALPHA\_SIZERENDERBUFFER\_DEPTH\_SIZERENDERBUFFER\_STENCIL\_SIZE RENDERBUFFER\_SAMPLES | Indicateurs pour récupérer des informations sur les render buffers à passer au paramètre pname. | GetRenderbufferParameteriv |
| FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPEFRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LEVELFRAMEBUFFER\_ATTACHMENT\_TEXTURE\_CUBE\_MAP\_FACEFRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LAYERFRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODINGFRAMEBUFFER\_ATTACHMENT\_COMPONENT\_TYPEFRAMEBUFFER\_ATTACHMENT\_RED\_SIZEFRAMEBUFFER\_ATTACHMENT\_GREEN\_SIZEFRAMEBUFFER\_ATTACHMENT\_BLUE\_SIZE FRAMEBUFFER\_ATTACHMENT\_ALPHA\_SIZEFRAMEBUFFER\_ATTACHMENT\_DEPTH\_SIZEFRAMEBUFFER\_ATTACHMENT\_STENCIL\_SIZE | Indicateurs pour récupérer des informations sur le frame buffer, à passer au paramètre pname de GetFramebufferAttachmentParameteriv. | GetFramebufferAttachmentParameteriv |
| COLOR\_ATTACHMENT0 COLOR\_ATTACHMENT1 COLOR\_ATTACHMENT2 COLOR\_ATTACHMENT3 COLOR\_ATTACHMENT4 COLOR\_ATTACHMENT5 COLOR\_ATTACHMENT6 COLOR\_ATTACHMENT7 COLOR\_ATTACHMENT8 COLOR\_ATTACHMENT9 COLOR\_ATTACHMENT10 COLOR\_ATTACHMENT11COLOR\_ATTACHMENT12COLOR\_ATTACHMENT13COLOR\_ATTACHMENT14COLOR\_ATTACHMENT15DEPTH\_ATTACHMENTSTENCIL\_ATTACHMENTDEPTH\_STENCIL\_ATTACHMENT | Valeurs désignant les différents canaux des FBO. | FramebufferTextureFramebufferRenderbufferGetFramebufferAttachmentParameteriv |
| FRAMEBUFFER\_COMPLETE FRAMEBUFFER\_INCOMPLETE\_ATTACHMENT FRAMEBUFFER\_INCOMPLETE\_MISSING\_ATTACHMENT FRAMEBUFFER\_INCOMPLETE\_DRAW\_BUFFER FRAMEBUFFER\_INCOMPLETE\_READ\_BUFFERFRAMEBUFFER\_UNSUPPORTEDFRAMEBUFFER\_INCOMPLETE\_MULTISAMPLEFRAMEBUFFER\_UNDEFINED | Valeurs identifiants les erreurs liées aux FBO. | CheckFramebufferStatus |
| INVALID\_FRAMEBUFFER\_OPERATION | Nouvelle valeur identifiant une erreur liée aux FBO. | glGetError() |

Nous allons aller un peu plus loin dans la structure des **FBO** car ils ont une façon bien particulière de fonctionner, en particulier au niveau de ses 3 buffers qui ne se gèrent pas tous de la même manière. On distinguera d'un coté le **Color Buffer** et de l'autre le **Depth** et le **Stencil Buffer**.

### Le Color Buffer

Comme nous l'avons vu un peu plus haut, un **FBO** peut être considéré comme un ***écran caché*** au sein de la carte graphique, nous pouvons donc sans aucun problème faire n'importe quel rendu à l'intérieur. L'intérêt dans tout ça vient du fait que l'on peut modifier les pixels une fois qu'ils sont affichés, ce qui est impossible avec le véritable écran.

Ces fameux pixels sont contenus dans le **Color Buffer**, celui que nous nettoyons à chaque tour de boucle avec la constante **GL\_COLOR\_BUFFER**. Et si je peux vous apprendre quelque chose d'étonnant avec lui, **c'est qu'il s'agit en fait d'une simple *texture* !**

Oui, vous avez bien lu, le Color Buffer est une texture :



C'est un peu logique puisque le but d'OpenGL est d'afficher des pixels sur un écran, et une texture est justement faite pour en contenir plein. L'avantage avec elle, c'est que nous pourrons l'utiliser dans un shader pour la modifier très facilement. Rajouter un effet sera donc un jeu d'enfant.

Le **Color Buffer** de l'écran est lui-aussi une texture ?

Oui on peut dire ça comme ça. C'es comme si on avait une grosse texture, contenant l'affichage de notre scène, plaquée sur la fenêtre.

Petite précision importante, les **FBO** peuvent contenir jusqu'à **16 Color Buffers**. C'est-à-dire que vous pouvez afficher ou modifier votre rendu **16** fois dans le **même FBO** :



Ceci est particulièrement utile si l'on souhaite stocker des données autres que celles relatives à l'affichage. Les ombres en sont un exemple car elles utilisent plusieurs **Color Buffers** pour stocker leurs données.

Mais bon, pour le moment nous n'en utiliserons qu'un seul ne vous en faites pas, c'est déjà bien assez.

### Le Depth et le Stencil Buffer

Les deux autres buffers (**Depth** et **Stentil**) se gèrent différemment du premier. Leurs données sont un peu plus complexes et ne peuvent pas être représentées par des textures. À la place, nous utiliserons ce que l'on appel des **Render Buffers**, ce sont des espaces mémoires conçues pour accueillir ces types de données :



### En résumé

Pour résumer un peu tout ce flot d'informations : un **Frame Buffer** est composé de 3 sous-buffers (**Color, Depth et Stencil**) :

* Le premier est représenté pas une **texture** et contient l'affichage d'une scène. Il peut y en avoir jusqu'à **16**.
* Les deux autres sont représentés par des **Render Buffer**.

Ce sont les points à connaitre par coeur en ce qui concerne les **FBO**.

### Fonctions

* **void** [**glGenFramebuffers**](http://www.opengl.org/sdk/docs/man3/xhtml/glGenFramebuffers.xml) (Glsizei n, Gluint\* ids) :

Créer un ou des FBO.

**n** : nombre de FBO a créer.

**ids** : tableau où les identifiants des FBO créés se trouveront.

* **void** [**glBindFramebuffer**](http://www.opengl.org/sdk/docs/man3/xhtml/glBindFramebuffer.xml) **(Glenum target, Gluint framebuffer) :**

Lie le frame buffer au contexte cible 'target' (les appels suivants aux fonctions OpenGL seront effectués sur ce FBO).

**target** : la cible sur laquelle le FBO sera attaché. Cela peut être **GL\_READ\_FRAMEBUFFER** afin de désigner le framebuffer de lecture, ou **GL\_DRAW\_FRAMEBUFFER** pour celui d'écriture. Si vous voulez attacher un FBO aux deux buffers à la fois, vous pouvez utiliser **GL\_FRAMEBUFFER**.

**framebuffer** : l'identifiant du framebuffer que vous souhaitez attacher.

* **void** [**glGenRenderbuffers**](http://www.opengl.org/sdk/docs/man3/xhtml/glGenRenderbuffers.xml) **(Glsizei n, Gluint\* renderbuffers) :**

Génère un ou des render buffers.

**n** : nombre de render buffers à créer.

**renderbuffers** : tableau où les identifiants des render buffers créés se trouveront.

* **void** [**glBindRenderbuffer**](http://www.opengl.org/sdk/docs/man3/xhtml/glBindRenderbuffer.xml) **(Glenum target, Gluint renderbuffer) :**

lie le render buffer au contexte courant

**target** :doit être GL\_RENDERBUFFER

**renderbuffer** : l'identifiant du render buffer à lier

* **void** [**glRenderbufferStorage**](http://www.opengl.org/sdk/docs/man3/xhtml/glRenderbufferStorage.xml) **(Glenum target, Glenum internalformat, Glsizei width, Glsizei height) :**

génère un render buffer à partir du format et des dimensions.

**target** : doit être GL\_RENDERBUFFER

**internalformat** : le format du render buffer

**width** : la largeur du render buffer

**height** : la hauteur du render buffer

* **void** [**glFramebufferTexture**](http://www.opengl.org/sdk/docs/man3/xhtml/glFramebufferTexture.xml) **(Glenum target, Glenum attachment, Gluint texture, Glint level) :**

indique la texture à attacher au FBO.

**target :** le FBO cible (en lecture et/ou en écriture)

**attachment** : la canal auquel nous attachons la texture (GL\_COLOR\_ATTACHMENT, GL\_DEPTH\_ATTACHMENT, GL\_STENCIL\_ATTACHMENT ou GL\_DEPTH\_STENCIL\_ATTACHMENT)

**texture** : l'identifiant de texture à attacher au FBO

**level** : indique le niveau de mipmap de la texture attachée

* **GLsync** [**glFramebufferRenderbuffer**](http://www.opengl.org/sdk/docs/man3/xhtml/glFramebufferRenderbuffer.xml) **(Glenum target, Glenum attachment, Glenum renderbuffertarget, Gluint renderbuffer) :**

indique le render buffer à attacher au FBO.

**target** : le FBO cible (en lecture et/ou en écriture)

**attachment** : le canal auquel nous attachons le render buffer

**renderbuffertarget** : GL\_RENDERBUFFER

**renderbuffer** : l'identifiant sur le render buffer à attacher

* **Glenum (Glenum target) :**

vérifie si le FBO est correctement construit.

**target** : GL\_DRAW\_FRAMEBUFFER ou GL\_READ\_FRAMEBUFFER ou GL\_FRAMEBUFFER

Renvoie un code d'erreur.

* **Void** [**glDrawBuffers**](http://www.opengl.org/sdk/docs/man3/xhtml/glDrawBuffers.xml) **(Glsizei n, const Glenum\* bufs) :**

spécifie la liste des canaux de couleurs qui seront utilisés.

**n** : le nombre de canaux dans la liste

**bufs** : pointeur sur les canaux dans lesquels le fragment shader écrira

* **void** [**glBlitFramebuffer**](http://www.opengl.org/sdk/docs/man3/xhtml/glBlitFramebuffer.xml) **(Glint srcX0, Glint srcY0, Glint, srcX1, Glint srcY1, Glint dstX0, Glint dstY0, Glint dstX1, Glint dstY1, Glbitfield mask, Glenum filter) :**

copie le FBO lié à GL\_READ\_FRAMEBUFFER dans le frame buffer lié à GL\_DRAW\_FRAMEBUFFER. Cette fonction n'est accessible qu'à partir d'OpenGL 3. Pour l'utiliser avec une version plus ancienne d'OpenGL, il vous faudra charger l'extension [GL\_EXT\_framebuffer\_blit](http://www.opengl.org/registry/specs/EXT/framebuffer_blit.txt) .

**SrcX0 srcY0** : coin supérieur gauche à partir duquel l'image sera copiée

**srcX1 srcY1** : coin inférieur droit jusqu'où l'image sera copiée

**dstX0 dstY0** : coin supérieur gauche à partir duquel l'image sera collée

**dstX1 dstY1** : coin inférieur droit jusqu'où l'image sera collée

**mask** : le canal à copier (GL\_COLOR\_BUFFER\_BIT, GL\_DEPTH\_BUFFER\_BIT et/ou GL\_STENCIL\_BUFFER\_BIT)

**filter** : interpolation appliquée durant la copie (GL\_NEAREST ou GL\_LINEAR)

## Mise en place

Maintenant que nous avons pris connaissance des fonctions pour les FBO, nous pouvons commencer à les utiliser.

Le schéma suivant récapitule les différentes actions afin d'utiliser les FBO.



Il est aussi possible de changer de texture et de render buffer dans la boucle d'affichage. Pour cela, il suffit d'attacher la nouvelle ressource avec [**glFramebufferTexture**](http://www.opengl.org/sdk/docs/man3/xhtml/glFramebufferTexture.xml) ou [**glFramebufferRenderbuffer**](http://www.opengl.org/sdk/docs/man3/xhtml/glFramebufferRenderbuffer.xml) et de valider celle-ci avec [**glCheckFramebufferStatus**](http://www.opengl.org/sdk/docs/man3/xhtml/glCheckFramebufferStatus.xml).

## Les FBO dans les shaders

Pour finaliser l'intégration des FBO, il est aussi nécessaire de faire quelques modifications dans le fragment shader, lui permettant ainsi de sauvegarder des informations dans les multiples textures attachées aux FBO.

### En OpenGL 2

Dans le GLSL < 150, lorsque nous devons envoyer des informations au FBO, nous utilisons **gl\_FragData[]**. C'est un tableau dont la taille est « **gl\_MaxDrawBuffers**  », soit le nombre maximum d'attachements que nous pouvons assigner. Son utilisation est simple, le premier élément du tableau correspond au premier attachement de couleur '**GL\_COLOR\_ATTACHMENT0**', le deuxième correspond à '**GL\_COLOR\_ATTACHMENT1**' et ainsi de suite.

**Remarque :**

Il n'est pas possible, dans le même fragment shader, d'utiliser gl\_FragColor et gl\_FragData. En effet, gl\_FragData[0] est identique à gl\_FragColor lors de l'utilisation d'un unique FBO. Cela veut aussi dire que, si vous avez un FBO avec GL\_COLOR\_ATTACHMENT0, l'utilisation de gl\_FragColor fonctionnera.

### En OpenGL 3

À partir du GLSL 330, glFragData n'existe plus. Au final, cela est un avantage, car les variables de sortie du fragment shader peuvent être nommées comme nous le souhaitons. Pour spécifier une variable de sortie, nous utilisons le mot clé 'out'. Maintenant, votre fragment shader aura plusieurs variables out et pour spécifier laquelle correspond à quel attachement de couleur, vous devez utiliser « layout(location=0) » comme suit :

layout(location=0) out vec4 positionOut;

Ainsi, la variable positionOut permettra d'écrire dans la cible de rendu attachée à la couleur 0 dans le FBO.

**Remarque :**

Si vous ne mettez aucun indication de « location » par défaut, la variable écrira dans l'attachement de couleur 0 (ou sur l'écran s'il n'y a pas de FBO).

## Exemple d'utilisation

Voici un petit [exemple simple](http://jeux.developpez.com/telecharger/detail/id/2773/Exemple-de-FBO) de l'utilisation des FBO avec OpenGL. Le programme est simplifié à son minimum. Pour les shaders, j'ai en partie utilisé le code de [ce tutoriel](http://alexandre-laurent.developpez.com/tutoriels/OpenGL/OpenGL-GLSL/?page=page_2). Pour le VBO, je me suis inspiré [du tutoriel de raptor70](http://raptor.developpez.com/tutorial/opengl/vbo/).



Le programme effectue un premier rendu (ou première passe) d'un cube coloré dans une texture. Dans la passe suivante, le programme dessine un cube et utilise la texture créée lors de la première passe. De plus, le résultat de la première passe est affiché sur l'écran à l'aide de la fonction glBlitFramebuffer().

Afin de compiler le projet, la lecture du README.txt est recommandée.

Afin de faciliter le débogage des FBO, il est conseillé de modifier la couleur de nettoyage (avec glClearColor()), ce qui permettra de reconnaître quel FBO est affiché.

Il est nécessaire d'utiliser un renderbuffer afin que le cube de la première passe utilise le buffer de profondeur. Sans celui-ci vous aurez les artefacts visuels suivants :



# 3. Techniques utilisant les Frame Buffer Objects

Grâce aux FBO, certaines techniques sont devenues possibles et d'autres ont été facilitées. Voici une liste non exhaustive de techniques utilisant les FBO.

## 3-A. Flou (Blur)

Un flou simple est très aisé à mettre en place. Il suffit de faire un rendu de la scène dans une texture. Ensuite, pour chaque pixel nous calculons la moyenne des couleurs alentour. Pour ce faire, nous dessinons la texture générée directement sur l'écran, à l'aide d'un rectangle de la taille de la fenêtre. Ainsi, le pixel shader va être exécuté sur chaque pixel. Voici un exemple du calcul de la couleur moyenne effectué dans le pixel shader :

float blurValue = 0.0015;

vec4 color = texture2D(fboTex, vec2(io\_texCoord.s + blurValue, io\_texCoord.t + blurValue));

color += texture2D(fboTex, vec2(io\_texCoord.s - blurValue, io\_texCoord.t + blurValue));

color += texture2D(fboTex, vec2(io\_texCoord.s + blurValue, io\_texCoord.t - blurValue));

color += texture2D(fboTex, vec2(io\_texCoord.s - blurValue, io\_texCoord.t - blurValue));

gl\_FragColor = color / 4.0;

Vous pouvez trouver [ici](http://jeux.developpez.com/telecharger/detail/id/2774/Exemple-de-Blur-utilisant-un-FBO) le programme d'exemple affichant un cube légèrement flou. On remarquera sur l'image ci-dessous que les bords du cube sont plus lisse.



Cet effet de flou peut être utilisé pour remplacer la technique d'anti-aliasing et cela pour un faible coût.

## 3-B. Shadow Mapping

Le shadow mapping est une technique permettant de calculer les ombres pour les objets d'une scène.

La technique est décomposée en deux passes : la calcul des ombres puis le rendu proprement dit. Pour le calcul des ombres, il faut placer la caméra à l'emplacement de la lumière, l'orienter avec la direction de la lumière, faire un rendu de la scène et ne garder que la profondeur (distance entre la lampe et le mesh, donc). Celle-ci est conservée dans l'espace homogène. Grâce aux FBO ce résultat est stocké dans une texture, appelée shadow map (littéralement : carte des ombres). Nous devons créer la shadow map pour chaque source, afin de connaître les ombres pour correspondante à chaque lumière.

Dans la seconde passe, la scène est rendue normalement, sauf que dans son pixel shader, on calculera si la distance entre le point actuel et la lampe est supérieur à la valeur retournée par la shadow map pour la même position. Si la valeur de la texture est inférieure, cela indique qu'il y a un objet entre la lampe et ce pixel, donc, qu'il doit être ombré.

## 3-C. Point Light

Avec le pipeline fixe nous étions limité à huit sources de lumière. Avec les shaders, le nombre de sources n'est plus limité mais le calcul des lumières pour tous les pixels et toutes les lampes est lent. Heureusement, il est possible de faire beaucoup mieux.

Cette technique se décompose en deux passes.

Dans la première, il faut afficher la scène, comme nous le ferions habituellement. Par contre, nous faisons le rendu dans un FBO qui contiendra au minimum trois textures attachées. Lors de l'exécution du shader, la position du vertex, la normale et la couleur de l'objet seront sauvegardées dans les textures (une textures pour chaque données).

La seconde passe utilise les trois textures précédemment créées. Par contre, nous allons éviter de calculer l'influence de toutes les lampes pour tous les pixels. Pour ce faire, nous allons limiter l'exécution du pixel shader aux seuls endroits où la lampe influe sur les pixels. Cela est possible en dessinant des rectangles de la taille du rayon de la lumière (puissance de la lampe). Ainsi, le pixel shader, qui contiendra l'algorithme pour l'éclairage ne sera exécuté que pour les pixels réellement éclairés. De plus, pour le cas où deux lampes affectent les mêmes pixels, il suffit de régler le mode de fondu (blend mode) afin d'avoir une composition.

# 4. Conclusion

Cet article présente les Frame Buffer Objects et explique comment s'en servir. Après une présentation de la documentation, nous avons mis en place notre premier programme utilisant les FBO. Celui-ci est simple, mais il pose les bases d'une fonctionnalité maintenant présente dans tous les moteurs 3D. Pour terminer, nous avons rapidement exploré quelques techniques utilisant les FBO afin de montrer leur utilité.