

# Apprendre OpenGL moderne

## Septième partie : mise en pratique

Par [Joey de Vries](#) - [Jean-Michel Fray](#) (traducteur)

Date de publication : 5 juillet 2018

TOUT PUBLIC

Developpez.com a la joie d'accueillir une traduction en français du célèbre cours de grande qualité **Learn OpenGL**. Au cours de celui-ci, 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 septième partie du tutoriel, c'est-à-dire : la mise en pratique de tout ce qui a été vu précédemment. Plus particulièrement, vous apprendrez à déboguer vos applications graphiques et à y afficher du texte.

Vous pouvez retrouver les autres parties ci-dessous :

- **introduction** ;
- **éclairage** ;
- **chargement de modèle** ;
- **OpenGL avancé** ;
- éclairage avancé ;
- PBR ;
- mise en pratique.

**Commentez**

I - Débogage.....	3
I-A - glGetError().....	3
I-B - Extension debug output.....	5
I-B-1 - Sortie de débogage avec GLFW.....	5
I-B-2 - Filtrer la sortie de débogage.....	7
I-B-3 - Retrouver l'origine de l'erreur.....	7
I-B-4 - Affichages personnalisés des erreurs.....	8
I-C - Déboguer les sorties des shaders.....	8
I-D - Compilateur GLSL de référence.....	9
I-E - Affichage d'un tampon de rendu.....	10
I-F - Logiciel externe de débogage.....	11
I-F-1 - RenderDoc.....	12
I-F-2 - CodeXL.....	12
I-F-3 - NVIDIA Nsight.....	13
I-G - Ressources supplémentaires.....	13
I-H - Remerciements.....	13
II - Affichage de texte.....	13
II-A - Affichage classique de texte : images de fontes.....	14
II-B - Affichage moderne de texte : FreeType.....	15
II-B-1 - Les shaders.....	18
II-B-2 - Afficher une ligne de texte.....	19
II-C - Pour aller plus loin.....	21
II-D - Remerciements.....	22

## I - Débogage

La programmation graphique est très amusante, mais peut aussi être source de frustration quand quelque chose ne donne pas le résultat escompté, voire pas de résultat du tout ! Afficher tout ce que l'on veut implique de manipuler des pixels et il peut s'avérer difficile de trouver la cause d'une erreur lorsque tout ne se passe pas comme prévu. Déboguer ce genre d'erreurs visuelles est différent de ce que vous avez l'habitude de faire en cherchant les erreurs CPU. On ne dispose pas de console pour afficher du texte, pas de points d'arrêt pour examiner le code GLSL et pas d'outil pour examiner l'exécution sur le GPU.

Dans ce tutoriel, nous verrons plusieurs techniques et astuces pour déboguer un programme OpenGL. Ce n'est pas si difficile et acquérir ces techniques sera payant à long terme.

### I-A - glGetError()

Dès que vous utilisez OpenGL incorrectement (comme configurer un tampon sans l'avoir lié), ce sera noté et cela positionnera un ou plusieurs indicateurs d'erreurs en arrière-plan. Vous pourrez récupérer ces informations en utilisant la fonction `glGetError()` qui teste ces indicateurs et retourne un code d'erreur si OpenGL a été malmené.

```
GLenum glGetError();
```

Lors de l'appel, cette fonction retourne soit un code d'erreur ou pas d'erreur du tout. Ces codes d'erreur sont listés ci-dessous :

Flag	Code	Description
<code>GL_NO_ERROR</code>	0	Pas d'erreur depuis le dernier appel à <code>glGetError()</code> .
<code>GL_INVALID_ENUM</code>	1280	Positionné si une énumération passée en paramètre est invalide.
<code>GL_INVALID_VALUE</code>	1281	Positionné si une valeur passée en paramètre est invalide.
<code>GL_INVALID_OPERATION</code>	1282	Positionné si l'état d'une commande est illégal compte tenu des paramètres passés.
<code>GL_STACK_OVERFLOW</code>	1283	Positionné si placer un objet sur la pile provoque un dépassement.
<code>GL_STACK_UNDERFLOW</code>	1284	Positionné si on essaie de retirer un objet d'une pile alors qu'elle est vide.
<code>GL_OUT_OF_MEMORY</code>	1285	Positionné si une allocation mémoire est impossible.
<code>GL_INVALID_FRAMEBUFFER</code>	1286	Positionné en cas de lecture ou d'écriture dans un tampon de rendu incomplet.

Dans la documentation des fonctions OpenGL, vous pourrez toujours trouver les codes d'erreur qu'une fonction génère au moment où l'erreur se produit. Par exemple, si vous regardez la documentation de la fonction `glBindTexture()`, vous trouverez tous les codes d'erreur dans la section [Errors](#).

Dès qu'un indicateur d'erreur est positionné, plus aucune erreur ne sera enregistrée. De plus, l'appel de `glGetError()` efface tous les indicateurs d'erreurs (ou seulement un dans un système distribué, voir la note ci-dessous). Cela implique que si vous appelez `glGetError()` à la fin d'un rendu et que vous trouvez une erreur, vous ne pouvez pas en conclure que c'était la seule erreur et de plus, la source de l'erreur peut se trouver n'importe où dans la boucle de rendu.



*Noter que si OpenGL s'exécute de façon distribuée, comme cela est souvent le cas sur les systèmes X11, d'autres codes d'erreur utilisateur peuvent encore être générés tant qu'ils ont différents indicateurs. L'appel à `glGetError()` n'efface alors que l'un des indicateurs et non pas tous. De ce fait, il est recommandé d'appeler `glGetError()` à l'intérieur d'une boucle.*

```
glBindTexture(GL_TEXTURE_2D, tex);
std::cout << glGetError() << std::endl; // retourne 0 (pas d'erreur)

glTexImage2D(GL_TEXTURE_3D, 0, GL_RGB, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
std::cout << glGetError() << std::endl; // retourne 1280 (énumération invalide)
Textures(-5, textures);
std::cout << glGetError() << std::endl; // retourne 1281 (valeur incorrecte)
std::cout << glGetError() << std::endl; // retourne 0 (pas d'erreur)
```

L'intérêt majeur de `glGetError()` est de permettre la localisation précise des erreurs et aussi de valider l'utilisation correcte d'OpenGL. Supposons que vous obteniez un écran noir sans savoir d'où vient le problème : le tampon d'affichage est-il bien établi ? Aurais-je oublié de lier une texture ? En appelant `glGetError()` vous pouvez trouver rapidement le premier endroit où une erreur apparaît et donc ce qui pose un problème.

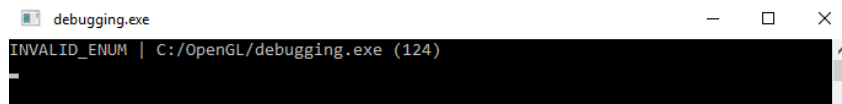
Par défaut, `glGetError()` n'affiche que les codes d'erreur, ce qui n'est pas facile à comprendre, à moins d'avoir mémorisé ces codes. Il est souvent plus pratique d'utiliser une petite fonction utilitaire pour afficher un message d'erreur à l'endroit où la vérification est effectuée :

```
GGLenum glCheckError_(const char *file, int line)
{
    GGLenum errorCode;
    while ((errorCode = glGetError()) != GL_NO_ERROR)
    {
        std::string error;
        switch (errorCode)
        {
            case GL_INVALID_ENUM: error = "INVALID_ENUM"; break;
            case GL_INVALID_VALUE: error = "INVALID_VALUE"; break;
            case GL_INVALID_OPERATION: error = "INVALID_OPERATION"; break;
            case GL_STACK_OVERFLOW: error = "STACK_OVERFLOW"; break;
            case GL_STACK_UNDERFLOW: error = "STACK_UNDERFLOW"; break;
            case GL_OUT_OF_MEMORY: error = "OUT_OF_MEMORY"; break;
            case GL_INVALID_FRAMEBUFFER_OPERATION:
                error = "INVALID_FRAMEBUFFER_OPERATION"; break;
        }
        std::cout << error << " | " << file << " (" << line << ")" << std::endl;
    }
    return errorCode;
}
#define glCheckError() glCheckError_( __FILE__, __LINE__ )
```

Si vous n'êtes pas familier des directives du préprocesseur, `__FILE__` et `__LINE__` sont remplacés lors de la compilation par le nom du fichier et la ligne en cours. Si l'on utilise un certain nombre de ces appels `glCheckError()`, mieux vaut savoir précisément lequel affiche l'erreur.

```
glBindBuffer(GL_VERTEX_ARRAY, vbo);
glCheckError();
```

Cela produira l'affichage suivant :



Il nous paraît **important** de signaler que GLEW renferme un bogue ancien : lors de l'appel `glewInit()`, la fonction `glGetError()` positionne l'indicateur d'erreur `GL_INVALID_ENUM` et retourne donc un code d'erreur qui peut vous tromper. Pour éviter cela, on peut simplement appeler `glGetError()` pour effacer l'indicateur d'erreur :

```
glewInit();
glGetError();
```

`glGetError()` ne donne pas beaucoup d'informations sur la cause de l'erreur, mais cela vous aidera à localiser des fautes de frappe et à trouver où votre code est erroné : c'est un outil de débogage assez simple, mais efficace.

## I-B - Extension debug output

Un outil moins courant, mais plus pratique que `glGetError()` est l'extension d'OpenGL nommée **debug output** (sortie de débogage) qui fait partie d'OpenGL depuis la version 4.3. Avec cette extension, OpenGL renverra lui-même une erreur et un message d'avertissement à l'utilisateur, bien plus détaillé que `glCheckError()`. Non seulement cela donne plus d'informations, mais vous pouvez localiser les erreurs plus précisément en utilisant le support d'un débogueur.



*La sortie de débogage fait partie du noyau d'OpenGL depuis la version 4.3, ce qui rend cette fonctionnalité disponible sur toute machine disposant de la version 4.3 ou ultérieure. Si ce n'est pas le cas, cette fonctionnalité peut être obtenue avec l'extension `ARB_debug_output` ou bien `AMD_debug_output`. Cela semble ne pas fonctionner sur OS X (je ne l'ai pas testé moi-même, dites-moi si je me trompe).*

Pour utiliser la sortie de débogage, il faut obtenir un contexte spécifique auprès d'OpenGL lors du processus d'initialisation. Ce processus dépend de votre système de fenêtrage ; ici, nous utilisons GLFW, mais vous pouvez trouver l'information nécessaire pour d'autres systèmes dans les ressources additionnelles à la fin de ce chapitre.

### I-B-1 - Sortie de débogage avec GLFW

Obtenir un contexte pour la sortie de débogage est très facile avec GLFW, la seule chose à faire est de le demander, avant de faire appel à `glfwCreateWindow()` :

```
glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);
```

Après avoir initialisé GLFW, on devrait disposer d'un contexte de débogage si l'on utilise OpenGL 4.3 ou plus, ou sinon, il faut espérer que le système peut fournir un contexte de débogage. Enfin, on peut obtenir un contexte de débogage avec les extensions d'OpenGL.



*Utiliser OpenGL en mode debug peut ralentir significativement l'exécution, il faudra penser à le supprimer si vous travaillez l'optimisation ou la version finale de l'application.*

Pour vérifier que le contexte de débogage a été initialisé avec succès, on peut interroger OpenGL :

```
GLint flags; glGetIntegerv(GL_CONTEXT_FLAGS, &flags);
if (flags & GL_CONTEXT_FLAG_DEBUG_BIT)
{
    // initialisation de la sortie de débogage
}
```

La sortie de débogage fonctionne de cette façon : on définit une fonction callback d'enregistrement d'erreur (comme les fonctions callback pour les entrées utilisateur) et nous traitons les données d'erreur dans la fonction callback comme bon nous semble. Dans notre cas, nous afficherons un message d'erreur sur la console. Ci-dessous un prototype de fonction callback qu'OpenGL attend pour la sortie de débogage :

```
void APIENTRY glDebugOutput(GLenum source, GLenum type, GLuint id, GLenum severity, GLsizei length, const GLchar *message, void *userParam);
```

Noter que dans certaines implémentations d'OpenGL, le dernier paramètre doit être de type `const void*` au lieu de `void*`.

Compte tenu du nombre important de données à traiter, on pourra créer un outil pratique d'affichage des erreurs comme ceci :

```
void APIENTRY glDebugOutput(GLenum source,
                           GLenum type,
                           GLuint id,
                           GLenum severity,
                           GLsizei length,
                           const GLchar *message,
                           void *userParam)
{
    // on ignore les codes ou avertissements non significatifs
    if(id == 131169 || id == 131185 || id == 131218 || id == 131204)
        return;
    std::cout << "-----" << std::endl;
    std::cout << "Debug message (" << id << "): " << message << std::endl;
    switch (source)
    {
        case GL_DEBUG_SOURCE_API:
            std::cout << "Source: API"; break;
        case GL_DEBUG_SOURCE_WINDOW_SYSTEM:
            std::cout << "Source: Window System"; break;
        case GL_DEBUG_SOURCE_SHADER_COMPILER:
            std::cout << "Source: Shader Compiler"; break;
        case GL_DEBUG_SOURCE_THIRD_PARTY:
            std::cout << "Source: Third Party"; break;
        case GL_DEBUG_SOURCE_APPLICATION:
            std::cout << "Source: Application"; break;
        case GL_DEBUG_SOURCE_OTHER:
            std::cout << "Source: Other"; break;
    }
    std::cout << std::endl;
    switch (type)
    {
        case GL_DEBUG_TYPE_ERROR:
            std::cout << "Type: Error"; break;
        case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR:
            std::cout << "Type: Deprecated Behaviour"; break;
        case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR:
            std::cout << "Type: Undefined Behaviour"; break;
        case GL_DEBUG_TYPE_PORTABILITY:
            std::cout << "Type: Portability"; break;
        case GL_DEBUG_TYPE_PERFORMANCE:
            std::cout << "Type: Performance"; break;
        case GL_DEBUG_TYPE_MARKER:
            std::cout << "Type: Marker"; break;
        case GL_DEBUG_TYPE_PUSH_GROUP:
            std::cout << "Type: Push Group"; break;
        case GL_DEBUG_TYPE_POP_GROUP:
            std::cout << "Type: Pop Group"; break;
        case GL_DEBUG_TYPE_OTHER:
            std::cout << "Type: Other"; break;
    }
    std::cout << std::endl;
    switch (severity)
    {
        case GL_DEBUG_SEVERITY_HIGH:
```

```
std::cout << "Severity: high"; break;
case GL_DEBUG_SEVERITY_MEDIUM:
std::cout << "Severity: medium"; break;
case GL_DEBUG_SEVERITY_LOW:
std::cout << "Severity: low"; break;
case GL_DEBUG_SEVERITY_NOTIFICATION:
std::cout << "Severity: notification"; break;
}
std::cout << std::endl;
std::cout << std::endl;
}
```

Dès lors que le pilote détectera une erreur OpenGL, la fonction callback sera appelée, et nous afficherons un message d'erreur intéressant. Noter que nous ignorons certains codes d'erreur qui ne sont pas d'un intérêt notable (comme 131185 avec les pilotes NVIDIA qui dit qu'un tampon a bien été créé).

Maintenant que nous avons notre fonction callback, il est temps d'initialiser la sortie de débogage :

```
if (flags & GL_CONTEXT_FLAG_DEBUG_BIT)
{
    glEnable(GL_DEBUG_OUTPUT);
    glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
    glDebugMessageCallback(glDebugOutput, nullptr);
    glDebugMessageControl(GL_DONT_CARE,
                        GL_DONT_CARE,
                        GL_DONT_CARE,
                        0, nullptr, GL_TRUE);
}
```

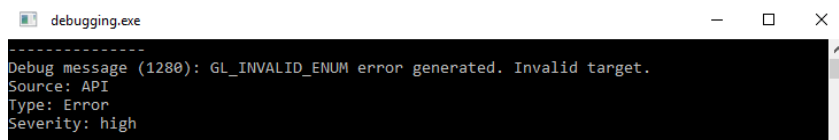
Nous demandons à OpenGL d'activer la sortie de débogage. L'appel `glEnable(GL_DEBUG_OUTPUT)` demande à OpenGL d'appeler directement la fonction callback lorsqu'une erreur se produit.

## I-B-2 - Filtrer la sortie de débogage

Avec `glDebugMessageControl()` vous pouvez filtrer le type d'erreur dont vous voulez être averti. Dans notre cas, nous avons décidé de n'appliquer aucun filtre sur la source, le type ou la gravité de l'erreur. Si nous voulions n'afficher que les messages provenant de l'API OpenGL, qui sont des erreurs, et qui sont graves, on aurait configuré les choses comme ceci :

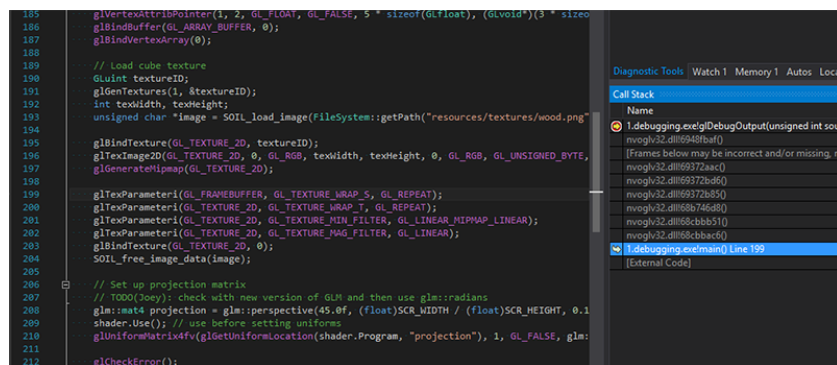
```
glDebugMessageControl(GL_DEBUG_SOURCE_API,
                    GL_DEBUG_TYPE_ERROR,
                    GL_DEBUG_SEVERITY_HIGH,
                    0, nullptr, GL_TRUE);
```

En fonction de votre configuration et en supposant que vous avez un contexte qui supporte la sortie de débogage, chaque commande OpenGL incorrecte vous affichera un message utile et détaillé :



## I-B-3 - Retrouver l'origine de l'erreur

Un autre avantage de la sortie de débogage est de pouvoir facilement connaître la ligne exacte où s'est produite l'erreur. En plaçant un point d'arrêt dans `DebugOutput()` sur un type d'erreur particulier (ou bien au début de la fonction si cela vous va bien), le débogueur interceptera l'erreur et vous pourrez remonter la pile des appels jusqu'à la fonction ayant produit l'erreur :



Il faut intervenir manuellement, mais si vous savez à peu près ce que vous cherchez, c'est extrêmement utile de déterminer rapidement quel appel a produit l'erreur.

## I-B-4 - Affichages personnalisés des erreurs

En plus de la lecture simple des messages, vous pouvez aussi placer les messages de votre choix dans le système de la sortie de débogage :

```
glDebugMessageInsert(GL_DEBUG_SOURCE_APPLICATION, GL_DEBUG_TYPE_ERROR, 0,
    GL_DEBUG_SEVERITY_MEDIUM, -1, "votre message d'erreur ici");
```

Cela est très utile si vous vous intégrez dans une autre application ou un autre code OpenGL qui utilise aussi un contexte supportant la sortie de débogage. D'autres développeurs peuvent ainsi rapidement signaler un bogue qui se produit dans votre propre code.

Pour résumer, la sortie de débogage (si vous pouvez l'utiliser) est extrêmement utile pour trouver rapidement les erreurs, cet outil mérite d'être installé, il vous fera gagner beaucoup de temps de développement. Vous trouverez une copie du code [ici](#), configuré avec `glGetError()` et la sortie de débogage, voyez si vous pouvez retrouver toutes les erreurs.

## I-C - Déboguer les sorties des shaders

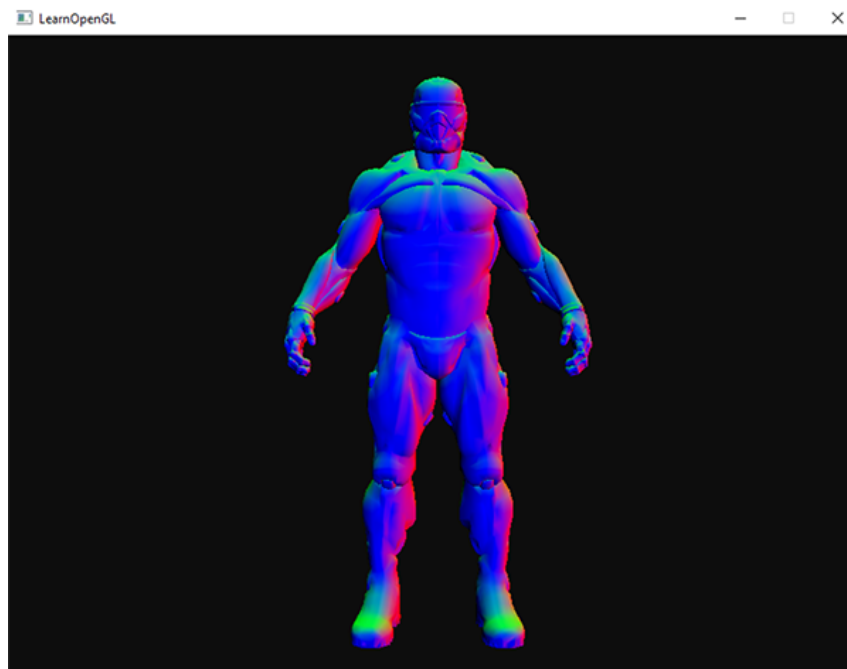
Quant à GLSL, nous ne disposons malheureusement pas d'une fonction comme `glGetError()` ni de la possibilité de placer des points d'arrêt dans le code d'un shader. Si vous obtenez un écran tout noir ou un effet visuel complètement faux, il est assez difficile de savoir ce qui ne fonctionne pas dans un shader. Les erreurs de compilation sont bien signalées, mais trouver les erreurs de logique est une autre histoire.

Une astuce souvent utilisée pour déboguer un shader consiste à évaluer toutes les variables d'un programme shader en les envoyant directement en sortie du fragment shader. En substituant les variables directement aux couleurs produites par le fragment shader, on peut souvent obtenir des informations intéressantes en inspectant les résultats visuellement. Par exemple, supposons que l'on veuille vérifier les normales d'un modèle, on peut les passer (transformées ou non) du vertex shader au fragment shader et on les sortira comme suit :

```
#version 330 core
out vec4 FragColor;
in vec3 Normal;
[...]
void main()
{
    [...]
    FragColor.rgb = Normal;
    FragColor.a = 1.0f;
}
```



En sortant une variable (qui n'est pas une couleur) comme une couleur, on peut visualiser si la variable semble correcte. Si par exemple le résultat est complètement noir, il est clair que les normales ne sont pas correctement passées aux shaders ; et si elles s'affichent, il est facile de vérifier si elles semblent correctes ou non :



On voit que les normales semblent correctes, les normales pointant vers la droite sont rouges, celles pointant vers l'avant sont bleues, et celles vers le haut sont vertes.

Cette approche peut être étendue à tout type de variable à tester. Si vous suspectez un problème dans un shader, essayez d'afficher certains résultats sous forme de couleur.

## I-D - Compilateur GLSL de référence

Chaque pilote graphique a ses propres manies et bizarreries. Par exemple, les pilotes NVIDIA sont assez relax et négligents quant aux spécifications, tandis que les pilotes ATI/AMD sont plus rigoureux et tendent à renforcer la spécification d'OpenGL (ce qui me semble la bonne approche). Le problème, c'est qu'un shader peut fonctionner sur une machine et pas sur une autre, du fait de pilotes différents.

Avec quelques années d'expérience, vous pourrez peut-être apprendre les différences mineures entre les constructeurs de GPU, mais si vous voulez être sûr que votre code fonctionne sur toutes les machines, vous pouvez le tester en utilisant le **compilateur GLSL de référence**. Vous pouvez télécharger le binaire « [GLSL lang validator](#) » [ici](#) ou le code source complet [ici](#).

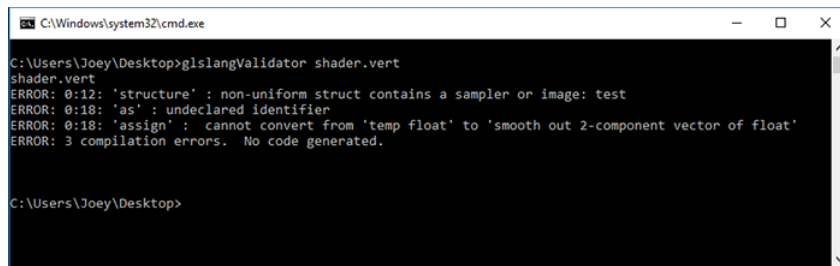
Avec « [GLSL lang validator](#) », vous pouvez facilement vérifier le code de vos shaders en passant le code de votre shader comme premier argument. Sachez que « [GLSL lang validator](#) » détermine le type de shader au moyen d'une liste d'extensions fixes :

- `.vert` : vertex shader.
- `.frag` : fragment shader.
- `.geom` : geometry shader.
- `.tesc` : tessellation control shader.
- `.tese` : tessellation evaluation shader.
- `.comp` : compute shader.

Exécuter le compilateur de référence de GLSL est très simple :

```
glslangvalidator shaderFile.vert
```

S'il n'y a pas d'erreur, il ne retourne rien. Si un vertex shader comporte une erreur, on obtiendra ce genre d'affichage :



```

C:\Windows\system32\cmd.exe
C:\Users\Joey\Desktop>glslangValidator shader.vert
shader.vert
ERROR: 0:12: 'structure' : non-uniform struct contains a sampler or image: test
ERROR: 0:18: 'as' : undeclared identifier
ERROR: 0:18: 'assign' : cannot convert from 'temp float' to 'smooth out 2-component vector of float'
ERROR: 3 compilation errors. No code generated.

C:\Users\Joey\Desktop>
  
```

Il ne vous donnera pas les différences subtiles entre les versions des compilateurs GLSL d'AMD, NVIDIA ou Intel, ni ne vous aidera à complètement déboguer vos shaders, mais au moins vous aidera à les comparer aux spécifications officielles de GLSL.

## I-E - Affichage d'un tampon de rendu

Une autre astuce utile pour déboguer consiste à afficher le contenu d'un tampon de rendu (voir chapitre 26) dans une zone prédéfinie de votre application OpenGL. Vous allez très souvent utiliser ces tampons de rendu nommés « framebuffer », et comme l'essentiel de leur rôle se déroule en arrière-plan, il est quelquefois difficile de réaliser ce qu'il s'y passe. Afficher le contenu d'un tampon de rendu dans votre application est un moyen pratique de voir rapidement si tout se passe bien.



*Notons que l'affichage du contenu d'un tampon de rendu comme expliqué ici ne fonctionnera que pour le rendu dans une texture, et non pas pour les objets de tampon de rendu.*

En utilisant un simple shader qui affiche une texture, on peut facilement écrire une petite fonction utilitaire pour afficher n'importe quelle texture en haut à droite de l'écran :

```

// vertex shader
#version 330 core
layout (location = 0) in vec2 position;
layout (location = 1) in vec2 texCoords;
out vec2 TexCoords;
void main()
{
    gl_Position = vec4(position, 0.0f, 1.0f);
    TexCoords = texCoords;
}

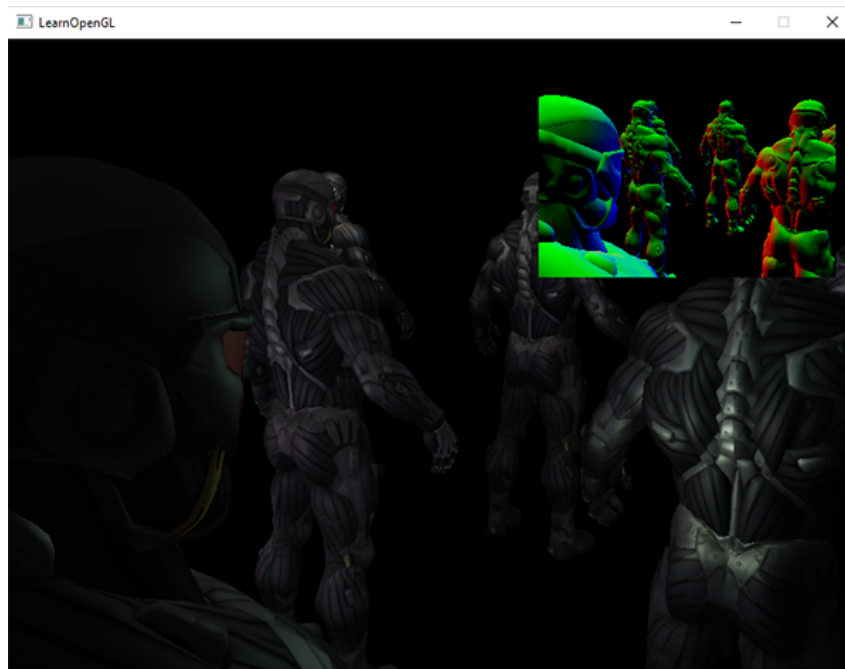
// fragment shader
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D fboAttachment;
void main()
{
    FragColor = texture(fboAttachment, TexCoords);
}

void DisplayFramebufferTexture(GLuint textureID)
{
    if(!notInitialized)
    {
        // initialise le shader et le VAO avec les coordonnées de sommets en espace de
        // périphérique, en haut à droite de l'écran
        [...]
    }
}
  
```

```
glActiveTexture(GL_TEXTURE0);
glUseProgram(shaderDisplayFBOOutput);
glBindTexture(GL_TEXTURE_2D, textureID);
glBindVertexArray(vaoDebugTexturedRect);
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
glUseProgram(0);
}

int main()
{
    [...]
    while (!glfwWindowShouldClose(window))
    {
        [...]
        DisplayFramebufferTexture(fboAttachment0);
        glfwSwapBuffers(window);
    }
}
```

Ce code vous donnera une petite fenêtre sympa au coin de votre écran pour déboguer un tampon de rendu. Utile, par exemple, pour déterminer si les normales d'un rendu différé semblent correctes :



On peut bien sûr étendre ce genre de fonction pour supporter le rendu de plus d'une texture. C'est un moyen rapide, mais peu orthodoxe d'obtenir un retour de ce qui se passe dans vos tampons de rendu.

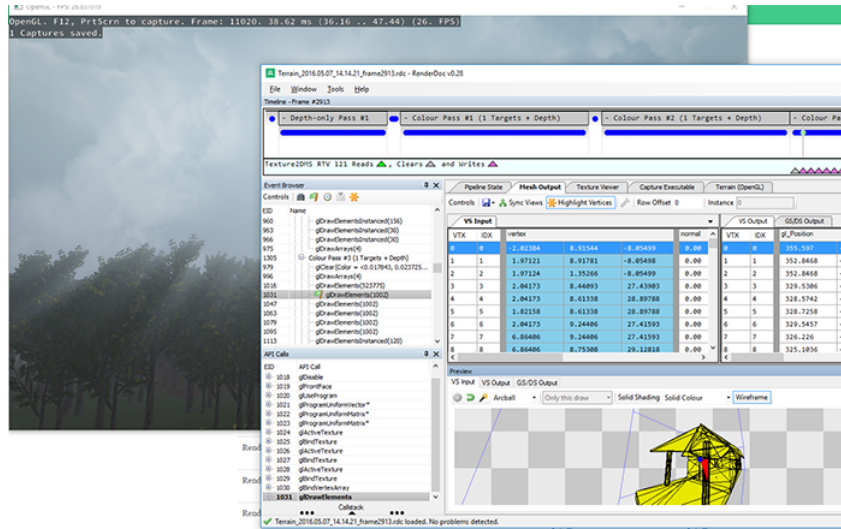
## I-F - Logiciel externe de débogage

Quand tout ceci ne suffit pas, on peut encore utiliser un outil tiers pour aider au débogage. Ces applications externes s'insèrent souvent elles-mêmes dans les pilotes d'OpenGL et sont capables d'intercepter toutes sortes d'appels OpenGL pour vous donner un vaste tableau de données intéressantes concernant votre application. Ces outils peuvent vous aider dans de nombreux domaines comme : suivre l'utilisation des fonctions OpenGL, trouver des goulots d'étranglement, inspecter les tampons mémoire, afficher des textures et les objets attachés au tampon de rendu. Si vous travaillez sur des codes de grosses productions, ces outils procurent une aide inestimable pour votre processus de développement.

J'ai listé ci-dessous, certains parmi les plus connus de ces logiciels de débogage ; essayez-les et voyez ce qui vous convient le mieux.

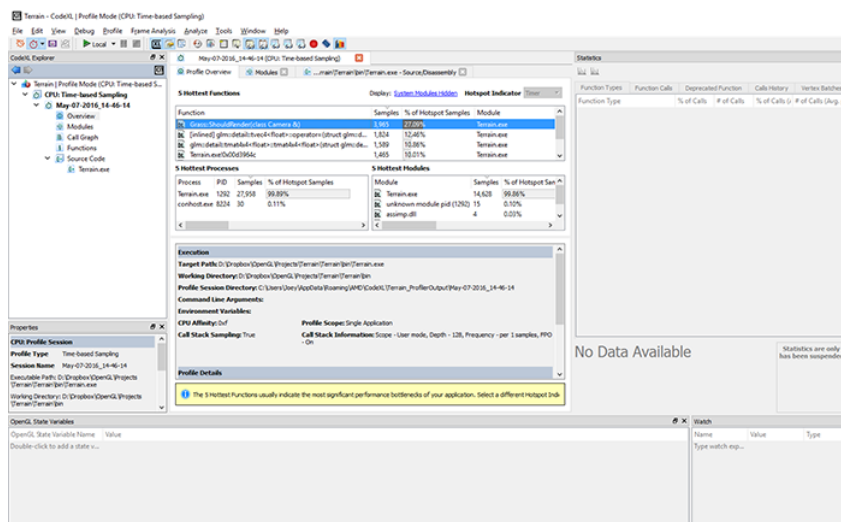
## I-F-1 - RenderDoc

RenderDoc est un puissant outil autonome (entièrement **open source**). Pour lancer une capture, vous spécifiez l'exécutable que vous voulez capturer et un répertoire de travail. Votre application fonctionne alors comme d'habitude, et à chaque fois que vous souhaitez inspecter un rendu particulier, vous laissez RenderDoc capturer un ou plusieurs rendus dans l'état courant de l'exécutable. Dans les rendus capturés, vous pouvez voir l'état du pipeline, toutes les commandes OpenGL, les tampons et textures en cours d'utilisation.



## I-F-2 - CodeXL

**CodeXL** est un outil de débogage du GPU délivré comme autonome ou bien comme plugin de Visual Studio. CodeXL donne pas mal d'informations et permet de suivre efficacement les applications graphiques. CodeXL fonctionne avec les cartes NVIDIA et Intel, mais sans support pour déboguer OpenGL.

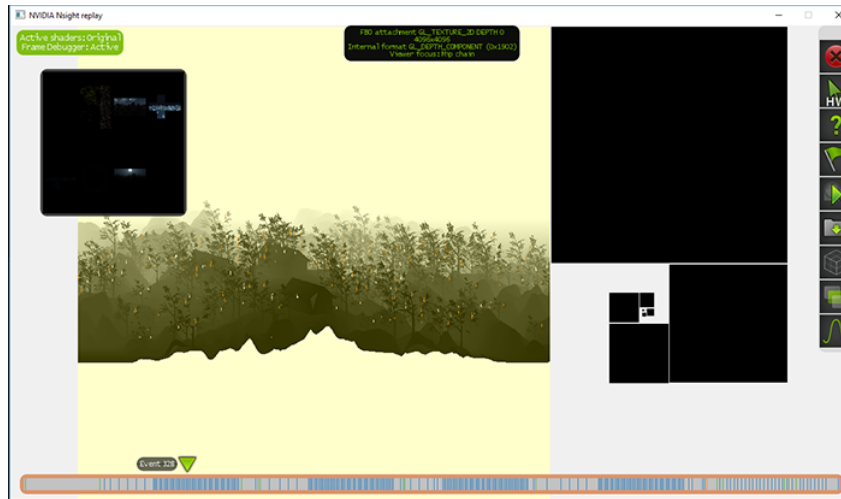


Je n'ai pas beaucoup d'expérience sur CodeXL, car je trouve RenderDoc plus facile à utiliser, mais je le mentionne, car il semble être un outil solide et développé par l'un des plus grands fabricants de GPU.

## I-F-3 - NVIDIA Nsight

**Nsight** de NVIDIA est un outil de débogage GPU bien connu. Ce n'est pas un logiciel autonome, mais un plugin pour Visual Studio ou encore Eclipse. Le plugin Nsight est un outil incroyablement utile pour les développeurs graphiques, il présente de nombreuses statistiques sur l'utilisation du GPU et de l'état du GPU passe de rendu par passe de rendu.

Dès que vous lancez votre application depuis Visual Studio (ou Eclipse) en utilisant le débogage ou le profilage, Nsight s'exécute dans l'application elle-même. Le grand intérêt de Nsight est d'afficher une interface utilisateur au premier plan, à partir de l'application, que l'on peut utiliser pour obtenir toutes sortes d'informations sur l'application, soit pendant l'exécution, soit pendant l'analyse des passes de rendu.



Nsight est un outil très puissant qui, selon moi, est le plus performant, mais son inconvénient majeur est de ne fonctionner qu'avec les cartes NVIDIA. Si vous travaillez avec NVIDIA et (Visual Studio), cela vaut vraiment la peine de se doter de Nsight.

Je suis sûr que d'autres logiciels méritent d'être mentionnés (comme **VOGL** de Valve et **APItrace**), mais cette courte liste vous donne déjà beaucoup d'outils à essayer. Je ne suis pas un expert avec ces outils, dites-moi dans les commentaires si je me trompe quelque part et je corrigerai où ce sera nécessaire.

## I-G - Ressources supplémentaires

- **Pourquoi votre code donne-t-il une fenêtre noire ?** : liste des causes principales sur le fait que l'application n'affiche rien, par Reto Koradi.
- **Debug Output** : un article approfondi sur la sortie de débogage avec des informations détaillées sur la mise en place d'un contexte de débogage des systèmes multifenêtrés, par Vallentin Source.

## I-H - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](https://learnopengl.com).

## II - Affichage de texte

Lors de certaines étapes de votre aventure graphique, vous souhaitez afficher du texte avec OpenGL. Contrairement à ce que vous pourriez penser, afficher une simple chaîne de caractères à l'écran est assez difficile avec une bibliothèque de bas niveau comme OpenGL. S'il vous suffit d'afficher les 128 premiers caractères, ce n'est pas trop compliqué. Les choses se corsent dès que chaque caractère peut avoir ses propres largeur, hauteur et

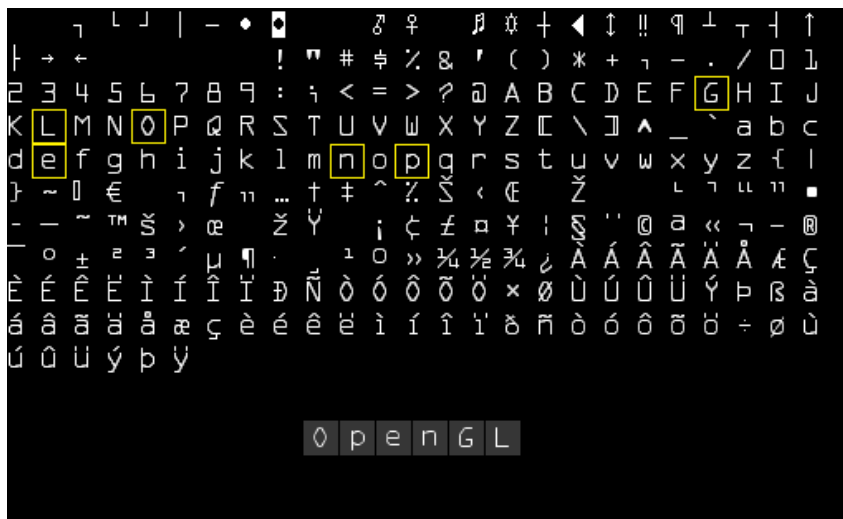
marge. Selon votre langue, vous pourrez avoir besoin de plus de 128 caractères, ou encore vouloir afficher des expressions mathématiques ou des partitions de musique, voire écrire de haut en bas. Si vous prenez tout cela en compte, cela ne vous surprendra pas qu'une API de bas niveau comme OpenGL ne propose pas ces fonctionnalités.

Puisqu'il n'existe pas dans OpenGL d'outils pour le texte, il nous appartient de définir un système pour afficher du texte à l'écran. Nous n'avons pas de primitive graphique pour les caractères, il va falloir être créatif. Voici quelques idées : tracer des lettres avec [GL\\_LINES](#), créer des mailles 3D pour les lettres ou afficher des textures de caractères dans un environnement 3D.

Le plus souvent, les développeurs choisissent d'utiliser des textures pour afficher les caractères dans des rectangles 2D. Cela n'est pas trop difficile, mais obtenir les bons caractères sur une texture peut s'avérer délicat. Dans ce chapitre, nous verrons plusieurs méthodes et nous implémenterons une technique plus évoluée, mais aussi plus souple pour afficher du texte, avec la bibliothèque FreeType.

## II-A - Affichage classique de texte : images de fontes

Auparavant, afficher du texte consistait à choisir une fonte (ou en créer une) pour votre application puis en extraire les caractères et les coller dans une seule grande texture. Une telle texture, que l'on peut appeler une image de fonte (bitmap font) contient tous les symboles de caractères dans des zones prédéfinies de la texture. Ces symboles sont appelés des **glyphes**. Chaque glyphe est positionné à un endroit précis de la texture. Quand vous souhaitez afficher un caractère, vous sélectionnez le glyphe correspondant en affichant cette partie de l'image de fonte dans un rectangle 2D.



Vous voyez ici comment afficher le texte « OpenGL » avec une telle fonte, en sélectionnant chacun des glyphes de la texture (au moyen de leurs coordonnées relatives à la texture), pour être affichés dans des petits rectangles. En activant le **blending** et avec un fond transparent, on obtient juste une chaîne affichée à l'écran. Cette fonte graphique a été générée avec le **générateur de fonte** de Codehead.

Cette approche a ses avantages et inconvénients. En premier lieu, c'est assez facile à mettre en œuvre et comme les images sont déjà préréglées, c'est assez performant. Cependant, ce n'est pas très souple. Si vous voulez changer de fonte, vous devez recompiler une fonte complète et de plus, ce système est limité à une seule résolution ; zoomer donnera très vite une image pixelisée. Enfin, on est limité à un petit ensemble de caractères et donc les caractères étendus ou Unicode ne sont pas envisageables.

Cette approche était (et reste) très courante, car elle est rapide et fonctionne sur toutes les machines, mais aujourd'hui des techniques plus souples existent. L'une d'entre elles consiste à charger des fontes TrueType grâce à la bibliothèque FreeType.



## II-B - Affichage moderne de texte : FreeType

FreeType est une bibliothèque de développement logiciel capable de charger des fontes, de les afficher dans une image bitmap et d'offrir un support pour les opérations relatives à ces fontes. C'est une bibliothèque largement diffusée, utilisée dans Mac OS X, Java, les consoles PlayStation, Linux et Android pour ne citer que ceux-là. Ce qui rend FreeType très intéressante est sa capacité à charger les fontes TrueType.

Une fonte TrueType est un ensemble de glyphes de caractères, non pas définis par des pixels ni par toute sorte de solutions statiques, mais par des équations mathématiques (des combinaisons de splines). Comme des images vectorielles, les images d'une fonte peuvent être générées de la taille que l'on veut, permettant d'obtenir des glyphes de taille quelconque sans perte de qualité.

FreeType peut être téléchargée à partir de ce [site web](#). Vous pouvez compiler la bibliothèque vous-même à partir de leur code source ou utiliser leurs bibliothèques précompilées si votre système cible est dans la liste. Assurez-vous de lier *freetype.lib* et que votre compilateur pourra trouver les fichiers d'en-tête.

Incluez les en-têtes prévus :

```
#include <ft2build.h>
#include FT_FREETYPE_H
```

*De la façon dont FreeType a été construite (du moins à l'heure où j'écris), vous ne pouvez pas placer les fichiers en-tête dans un nouveau répertoire, il faut les placer dans le répertoire racine de vos fichiers d'en-tête.*

*#include <FreeType/ft2build.h> causera sans doute de sérieux conflits.*

FreeType charge ces fontes TrueType et pour chaque glyphe génère une image bitmap et calcule plusieurs tailles. On peut extraire ces images pour générer des textures et positionner chaque glyphe de caractère en bonne place selon les tailles choisies.

Pour charger une fonte, il suffit d'initialiser la bibliothèque FreeType et charger la fonte en tant que **face**, comme FreeType les nomme. Ici nous chargeons la fonte TrueType *arial.ttf* qui a été copiée depuis le répertoire *Windows/Fonts*.

```
FT_Library ft;
if (FT_Init_FreeType(&ft))
    std::cout << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;
FT_Face face;
if (FT_New_Face(ft, "fonts/arial.ttf", 0, &face))
    std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;
```

Chacune de ces fonctions FreeType retourne un entier non nul si une erreur survient.

Après avoir chargé la face, on doit définir la taille que nous voulons extraire de cette face :

```
FT_Set_Pixel_Sizes(face, 0, 48);
```

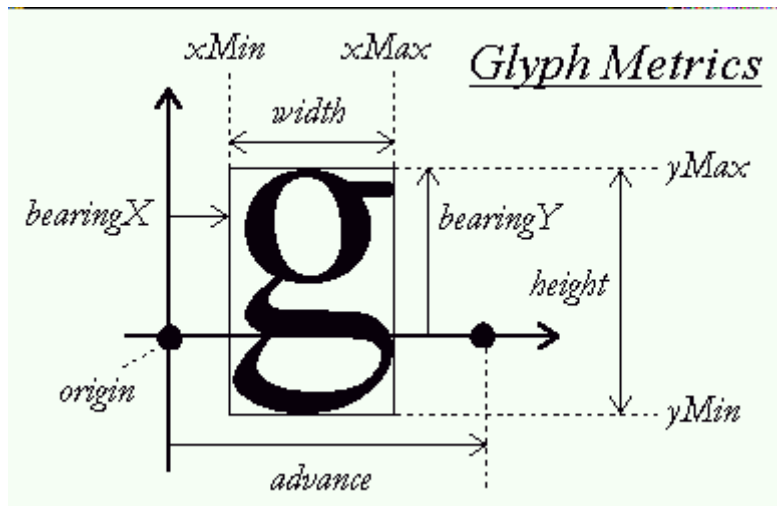
La fonction détermine les paramètres de hauteur et largeur de la fonte. Si *width* est laissé à 0, la face calcule dynamiquement la largeur en fonction de la hauteur.

Une face FreeType contient un ensemble de glyphes. On peut choisir un de ces glyphes comme étant le glyphe courant en appelant *FT\_Load\_Char()*. Ici, on choisit le glyphe du caractère 'X' :

```
if (FT_Load_Char(face, 'X', FT_LOAD_RENDER))
    std::cout << "ERROR::FREETYPE: Failed to load Glyph" << std::endl;
```

En utilisant `FT_LOAD_RENDER` comme un des paramètres de chargement, nous demandons à FreeType de créer une image bitmap 8 bits en niveaux de gris, que nous pourrions accéder par `face->glyph->bitmap`.

Les glyphes que nous chargeons avec FreeType n'ont cependant pas la même taille (ce qui était le cas avec les images de fontes). L'image générée par FreeType est juste assez large pour contenir la partie visible du caractère. Par exemple, l'image du caractère '.' est bien plus petite que celle du caractère 'X'. Pour cette raison, FreeType charge aussi plusieurs tailles en spécifiant la largeur de chaque caractère et comment les positionner correctement. Ci-dessous une image FreeType qui montre toutes les dimensions calculées pour chaque glyphe.



Chaque glyphe est défini par rapport à une ligne horizontale (figurée par la flèche horizontale), certains sont placés exactement sur cette ligne de base (comme 'X'), et d'autres sont légèrement en dessous (comme 'g' ou 'p'). Ces dimensions définissent exactement les décalages pour placer chaque glyphe sur la ligne de base, la largeur de chaque glyphe, et combien de pixels sont nécessaires avant le glyphe suivant. Ci-dessous une courte liste des propriétés dont on aura besoin :

- **width** : la largeur (en pixels) de l'image, accessible par `face->glyph->bitmap.width`.
- **height** : la hauteur (en pixels) de l'image, accessible par `face->glyph->bitmap.rows`.
- **bearingX** : le repère gauche, c'est-à-dire la position horizontale (en pixels) de l'image relative à l'origine, accessible par `face->glyph->bitmap_left`.
- **bearingY** : l'œil, c'est-à-dire la position verticale (en pixels) de l'image relative à l'origine, accessible par `face->glyph->bitmap_top`.
- **Advance** : l'empattement, c'est-à-dire la distance horizontale (en 1/64e de pixels) de l'origine du caractère au caractère suivant, accessible par `face->glyph->advance.x`.

On pourrait charger un glyphe, retrouver ses dimensions et générer une texture à chaque fois que l'on voudrait afficher un caractère à l'écran, mais ce serait peu efficace de faire cela à chaque image. Il vaut mieux mémoriser les données quelque part et les reprendre à chaque affichage du caractère. Nous définissons une structure que nous mémorisons dans une map.

```
struct Character {
    GLuint    TextureID;    // Identificateur de la texture du glyphe
    glm::ivec2 Size;        // Taille du glyphe
    glm::ivec2 Bearing;     // Approche gauche et œil du glyphe
    GLuint    Advance;      // Empattement du glyphe
};
std::map<GLchar, Character> Characters;
```



Dans ce tutoriel, nous simplifions en nous limitant aux 128 premiers caractères de la table ASCII. Pour chaque caractère, nous générons une texture et mémorisons les données dans une structure que nous ajoutons à la table de hachage Characters. De cette façon, toutes les données nécessaires pour afficher chaque caractère seront disponibles pour un usage ultérieur.

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); // Désactiver la restriction d'alignement d'octets
for (GLubyte c = 0; c < 128; c++)
{
    // Chargement du glyphe du caractère
    if (FT_Load_Char(face, c, FT_LOAD_RENDER))
    {
        std::cout << "ERROR::FREETYPE: Failed to load Glyph" << std::endl;
        continue;
    }
    // Générer la texture
    GLuint texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(GL_TEXTURE_2D,
                0,
                GL_RED,
                face->glyph->bitmap.width,
                face->glyph->bitmap.rows,
                0,
                GL_RED,
                GL_UNSIGNED_BYTE,
                face->glyph->bitmap.buffer);

    // Fixer les options de la texture
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // Mémoriser le caractère pour un usage ultérieur
    Character character = {
        texture,
        glm::ivec2(face->glyph->bitmap.width, face->glyph->bitmap.rows),
        glm::ivec2(face->glyph->bitmap_left, face->glyph->bitmap_top),
        face->glyph->advance.x
    };
    Characters.insert(std::pair<GLchar, Character>(c, character));
}
```

On boucle sur les 128 caractères de la table ASCII et on retrouve leur glyphe. Pour chaque caractère, on génère une texture, on positionne les options et les dimensions. À noter que l'on utilise `GL_RED` pour l'argument `internalFormat` de la texture, ainsi que pour l'argument `format`. L'image générée à partir du glyphe est une image 8 bits en niveaux de gris où chaque couleur est représentée sur un octet. Pour cette raison, nous mémoriserons chaque octet de l'image comme la valeur d'une couleur de texture. Nous réalisons cela en créant une texture dans laquelle chaque octet correspond à la composante rouge (premier octet du vecteur couleur). Puisque nous n'utilisons qu'un seul octet pour représenter les couleurs de la texture, il faut prendre garde à une restriction d'OpenGL :

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

OpenGL requiert que les textures aient une taille multiple de 4 octets (pour des raisons d'alignement). En principe ce n'est pas un problème puisque la plupart des textures ont une taille multiple de 4 et/ou utilisent 4 octets par pixel, mais puisque nous n'utilisons qu'un octet par pixel, elles peuvent avoir n'importe quelle taille. En paramétrant l'absence d'alignement à 1, nous assurons qu'il n'y aura pas de problème d'alignement (ce qui provoquerait des erreurs en mémoire).

Assurez-vous de supprimer les ressources FreeType quand vous avez fini de traiter les glyphes :

```
FT_Done_Face(face);
FT_Done_FreeType(ft);
```

## II-B-1 - Les shaders

Pour effectivement afficher les glyphes, nous utiliserons le vertex shader suivant :

```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 pos, vec2 tex>
out vec2 TexCoords;
uniform mat4 projection;
void main()
{
    gl_Position = projection * vec4(vertex.xy, 0.0, 1.0);
    TexCoords = vertex.zw;
}
```

Nous combinons la position et les coordonnées de texture dans un `vec4`. Le vertex shader multiplie les coordonnées par une matrice de projection et transmet les coordonnées de texture au fragment shader :

```
#version 330 core
in vec2 TexCoords;
out vec4 color;
uniform sampler2D text;
uniform vec3 textColor;
void main()
{
    vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);
    color = vec4(textColor, 1.0) * sampled;
}
```

Le fragment shader utilise deux variables uniformes : l'une pour l'image du glyphe en niveaux de gris, l'autre est une couleur pour ajuster la couleur finale du texte. Nous échantillonons d'abord la couleur de la texture. Cette couleur est codée sur la seule composante rouge, nous utilisons la composante `r` de la texture comme valeur de transparence. En jouant sur la valeur alpha de la couleur, la couleur résultante sera transparente pour la couleur du fond des glyphes et opaque pour la couleur des pixels du caractère. Nous multiplions aussi les couleurs RGB par la variable uniforme `textColor` pour la couleur du texte.

Il faut activer le **blending** pour que cela fonctionne :

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

Pour la matrice de projection, nous choisissons une projection orthogonale. Pour afficher du texte, nul besoin de perspective (en général) et l'utilisation d'une projection orthogonale nous permet de spécifier les coordonnées des sommets directement en coordonnées d'affichage si nous la définissons ainsi :

```
glm::mat4 projection = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f)
```

Nous fixons le bas de la projection à `0.0f` et le haut égal à la hauteur de la fenêtre d'affichage. Par conséquent, nous spécifions les coordonnées avec des valeurs de `y` s'étendant du bas de la fenêtre (`0.0f`) jusqu'en haut (`600.0f`). Et donc le point (`0.0`, `0.0`) correspond au coin en bas à gauche.

Il reste à créer un VBO et un VAO pour le rendu des rectangles. Pour l'instant, nous réservons assez de mémoire en initialisant le VBO de façon à pouvoir mettre à jour la mémoire du VBO lors du rendu des caractères.

```
GLuint VAO, VBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 6 * 4, NULL, GL_DYNAMIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat), 0);
```

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Les rectangles 2D nécessitent 6 sommets de 4 réels chacun, on réserve ainsi 6\*4 réels en mémoire. Étant donné que nous mettrons à jour très souvent le contenu du VBO, on alloue la mémoire avec `GL_DYNAMIC_DRAW`.

## II-B-2 - Afficher une ligne de texte

Pour afficher un caractère, on extrait la structure correspondante de la table de hachage Characters et on calcule les dimensions du rectangle en utilisant les dimensions du caractère. On peut ensuite générer dynamiquement un ensemble de six sommets que nous utiliserons pour rafraîchir le contenu de la mémoire gérée par le VBO avec `glBufferSubData()`.

On crée une fonction `RenderText()` pour afficher une chaîne de caractères :

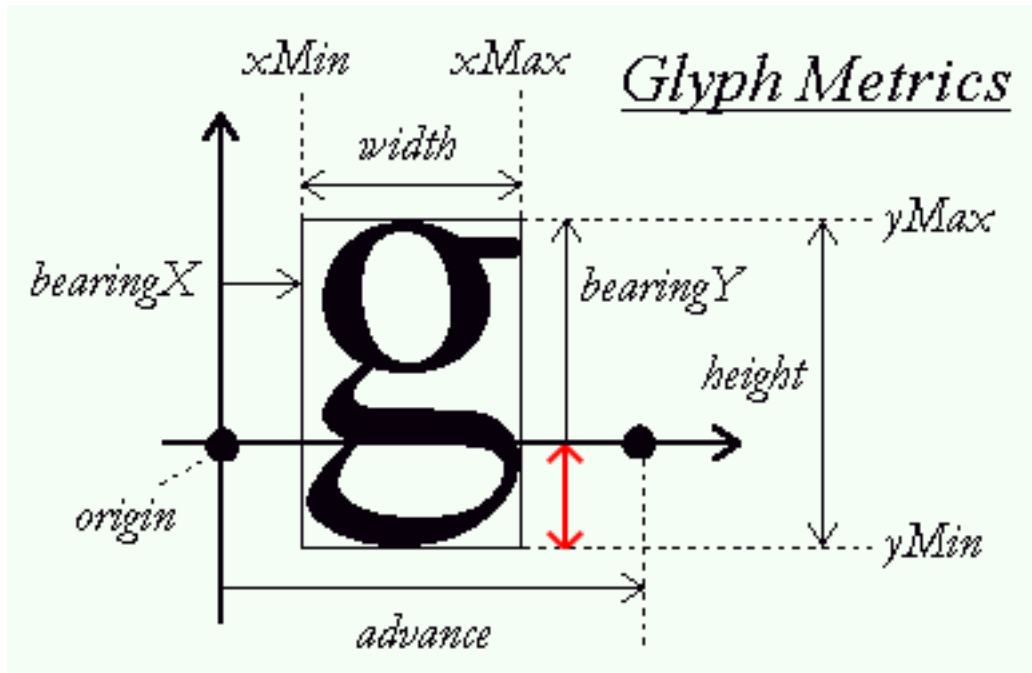
```
void RenderText(Shader &s, std::string text, GLfloat x, GLfloat y, GLfloat scale, glm::vec3
color)
{
    // Activation du rendu
    s.Use();
    glUniform3f(glGetUniformLocation(s.Program, "textColor"), color.x, color.y, color.z);
    glActiveTexture(GL_TEXTURE0);
    glBindVertexArray(VAO);
    // Boucle sur tous les caractères
    std::string::const_iterator c;
    for (c = text.begin(); c != text.end(); c++)
    {
        Character ch = Characters[*c];
        GLfloat xpos = x + ch.Bearing.x * scale;
        GLfloat ypos = y - (ch.Size.y - ch.Bearing.y) * scale;
        GLfloat w = ch.Size.x * scale;
        GLfloat h = ch.Size.y * scale;
        // Mise à jour du VBO
        GLfloat vertices[6][4] = {
            { xpos,      ypos + h,   0.0, 0.0 },
            { xpos,      ypos,       0.0, 1.0 },
            { xpos + w,  ypos,       1.0, 1.0 },
            { xpos,      ypos + h,   0.0, 0.0 },
            { xpos + w,  ypos,       1.0, 1.0 },
            { xpos + w,  ypos + h,   1.0, 0.0 }
        };
        // Rendu du glyphe sur le rectangle
        glBindTexture(GL_TEXTURE_2D, ch.textureID);
        // Mise à jour de la mémoire du VBO
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        // Rendu du rectangle
        glDrawArrays(GL_TRIANGLES, 0, 6);
        // Avancer le curseur au glyphe suivant (noter que l'avance est calculée en 1/64e pixels)
        x += (ch.Advance >> 6) * scale; // Décalage à droite de 6 pour obtenir la valeur en
pixels
    }
    glBindVertexArray(0);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Le contenu de la fonction devrait parler de lui-même : nous calculons la position de l'origine du rectangle (xposyposw et h) ainsi que sa taille (w et h) puis on génère un ensemble de six sommets pour former le rectangle ; noter que nous effectuons une mise à l'échelle avec scale. On met à jour le contenu du VBO et on affiche le rectangle.

Le code suivant mérite cependant un peu d'attention :

```
GLfloat ypos = y - (ch.Size.y - ch.Bearing.y);
```

Certains caractères (comme 'p' ou 'g') sont affichés légèrement en dessous de la ligne de base, et le rectangle doit aussi être positionné légèrement décalé en y. La valeur de ce décalage de ypos peut être déterminée avec les dimensions du glyphe :



Cette distance, c'est-à-dire le décalage que l'on doit appliquer vers le bas, est figurée par la flèche rouge. Comme on le voit, cette distance se calcule comme la différence entre la hauteur du glyphe et la dimension `bearingY`. Cette valeur est nulle pour les caractères qui sont sur la ligne de base alors qu'elle est positive pour les caractères comme 'g' ou 'j'.

Si vous avez tout exécuté correctement, vous devriez pouvoir afficher des chaînes de caractères avec les lignes suivantes :

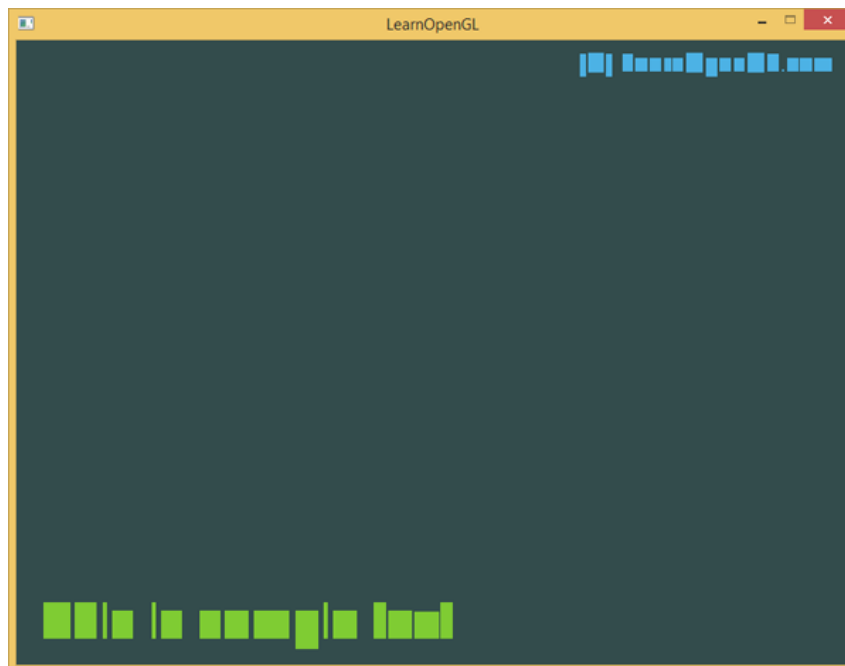
```
RenderText(shader, "This is sample text", 25.0f, 25.0f, 1.0f, glm::vec3(0.5, 0.8f, 0.2f));
RenderText(shader, "(C) LearnOpenGL.com", 540.0f, 570.0f, 0.5f, glm::vec3(0.3, 0.7f, 0.9f));
```

Vous devriez obtenir l'image suivante :



Vous trouverez le code de cet exemple [ici](#).

Pour vous donner une idée du calcul des sommets des rectangles, on peut désactiver le blending pour mieux voir leurs dimensions et leur position :



Ici, on voit bien que la plupart des rectangles reposent sur la ligne de base tandis que certains sont décalés vers le bas.

## II-C - Pour aller plus loin

Ce chapitre montre une technique d’affichage de texte avec les fontes TrueType en utilisant la bibliothèque FreeType. Cette approche est souple, permet la mise à l’échelle, et fonctionne avec beaucoup de codes de caractères. Cependant, elle peut s’avérer excessive pour votre application, car elle requiert de générer et rendre une texture pour chaque glyphe.

Les images de fontes sont plus simples et rapides, car elles ne requièrent qu'une seule texture pour tous les caractères. La meilleure approche est de combiner les deux façons en générant dynamiquement une texture d'image de fonte représentant tous les glyphes de caractères chargés avec FreeType. Cela décharge le moteur de rendu de tous ces changements de texture et selon comment les glyphes sont rangés peut améliorer nettement les performances.

Une autre question avec les fontes FreeType vient de ce que les textures sont mémorisées avec une fonte de taille fixe et une mise à l'échelle importante conduit à des bords crénelés. De plus, les rotations appliquées aux glyphes les rendront flous. Cela peut être atténué : au lieu de mémoriser la couleur réelle des pixels, on mémorise la distance au contour le plus proche du glyphe pour chaque pixel. Cette technique est appelée « **signed distance fields** », et Valve a publié un **article** voilà quelques années sur sa mise en œuvre qui fonctionne très bien pour des applications de rendus 3D.

## II-D - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.