Déboguer avec OpenGL 4

Lorsque l'on débute l'apprentissage d'OpenGL et des shaders (et même ensuite), on est vite confronté au problème du débogage, soit parce que le programme s'arrête brusquement, soit parce que le résultat obtenu ne correspond pas à ce que l'on attend. Traditionnellement, on utilise la fonction glGetError, mais elle est encore trop souvent « oubliée » par les développeurs et elle donne finalement assez peu d'informations. Heureusement, cette problématique a été prise en compte dans les dernières spécifications d'OpenGL avec l'ajout de nouvelles extensions pour le débogage. Ce billet de blog aborde les fonctionnalités de débogage introduites dans OpenGL 4.1 avec l'extension ARB_debug_output et complétées dans OpenGL 4.3 avec l'extension KHR_debug.

Les outils externes de débogage ne sont pas abordés.

Les nouvelles extensions

L'extension ARB_debug_output (GL 4.1)

Dans la version OpenGL 4.1, a été ajouté l'extension ARB_debug_output, qui est une promotion de l'extension GL_AMD_debug_output proposée par AMD. Cette extension ajoute la possibilité de créer des contextes avec un mode de débogage. En l'activant, un système d'événements est activé et permet d'obtenir des informations variées, par exemple sur les problèmes rencontrés lors de la compilation des shaders GLSL, des appels de fonction non valides ou des problèmes potentiels de performance. Les messages générés par ce système peuvent être stockés dans une pile (« message log ») ou être récupérés par une fonction callback définie par l'utilisateur.

L’extension KHR_debug (GL 4.3)

Dans le but d’uniformiser les différentes versions d’OpenGL (en particulier avec OpenGL ES), l’évolution d’OpenGL tend à reprendre des fonctions provenant d’OpenGL ES. C’est le cas de l’extension KHR_debug dans OpenGL 4.3 qui est une promotion de l’extension ARB_debug_output d’OpenGL 4.1 et des extensions EXT_debug_marker et EXT_debug_label d’OpenGL ES. Cette extension ajoute ainsi des fonctions pour annoter les événements ou des groupes de commandes OpenGL.

Création d’un contexte de débogage (GL 4.1)

La première chose à faire pour utiliser un contexte de débogage est de vérifier que l’extension ARB_debug_output est bien prise en charge. Un exemple de code pour vérifier une extension est donné dans le tutoriel Les extensions OpenGL.

Pour créer un contexte de débogage, il faut simplement créer un contexte en appelant la fonction CreateContextAttribs avec le paramètre CONTEXT_DEBUG_BIT. En fonction du système d’exploitation, on aura donc un code équivalent à celui-ci :

```
int attribs[] =
{
#ifdefWIN32
    WGL_CONTEXT_MAJOR_VERSION_ARB, 4,
    WGL_CONTEXT_MINOR_VERSION_ARB, 1,
    WGL_CONTEXT_FLAGS_ARB, WGL_CONTEXT_DEBUG_BIT_ARB,
    WGL_CONTEXT_PROFILE_MASK,
WGL_CONTEXT_CORE_PROFILE_BIT_ARB,
#endif
#ifdef__linux__
    GLX_CONTEXT_MAJOR_VERSION_ARB, 4,
    GLX_CONTEXT_MINOR_VERSION_ARB, 1,
    GLX_CONTEXT_FLAGS_ARB, GLX_CONTEXT_DEBUG_BIT_ARB,
    GLX_CONTEXT_PROFILE_MASK,
GLX_CONTEXT_CORE_PROFILE_BIT_ARB,
```

```
#endif
    0
};

// intialisation
HDC hdc;        // Handle to a device context
HGLRC hglrc;    // Handle to an OpenGL rendering context
hglrc = wglCreateContext (hdc);
wglMakeCurrent (hdc, hglrc);

// contexte attribs
int pixelFormat, numFormats;
glChoosePixelFormatARB(hdc, attribList, NULL, 1, &
pixelFormat,
    &numFormats);
HGLRC wglCreateContextAttribsARB(HDC hDC, HGLRC
hshareContext,
    const int *attribList);

// après utilisation
wglMakeCurrent (NULL, NULL) ;
wglDeleteContext (hglrc);
```

Les fonctionnalités de débogage sont activées en utilisant la constante GL_DEBUG_OUTPUT. Celle-ci peut être activée ou désactivée avec glEnable et glDisable. Par défaut, cette constante est activée dans un contexte de débogage et désactivée dans le cas contraire.

```
// active les messages de débogage
glEnable(GL_DEBUG_OUTPUT);

// désactive les messages de débogage
glDisable(GL_DEBUG_OUTPUT);
```

Plusieurs niveaux de débogage sont possibles en fonction de la création ou non d'un contexte de débogage et de si l'on active GL_DEBUG_OUTPUT :

- si on ne crée pas un contexte de débogage et que l'on active GL_DEBUG_OUTPUT, les messages envoyés et le contenu sont laissés à l'appréciation des implémentations d'OpenGL, mais

l'appel des fonctions de débogage ne produit pas d'erreur ;
- si on crée un contexte de débogage et que l'on n'active pas GL_DEBUG_OUTPUT, aucun message de débogage n'est lancé ;
- si on crée un contexte de débogage et que l'on active GL_DEBUG_OUTPUT, toutes les fonctionnalités de débogage sont activées.

Créer des contextes de débogage à partir d'une bibliothèque (GL 4.1)

Il existe plusieurs bibliothèques graphiques qui permettent de travailler sur des contextes OpenGL (SFML, SDL, Qt, etc.). Voyons comment créer un contexte de débogage dans ces cas d'utilisation.

Voir : http://vallentinsource.com/opengl/debug-output

Avec FreeGlut

FreeGlut fournit directement une constante pour créer un contexte en mode débogage : GLUT_DEBUG.

```
// FreeGlut
glutInitContextFlags(GLUT_DEBUG);
```

Avec Qt

La bibliothèque Qt fournit des outils pour créer et manipuler directement des fenêtres contenant un contexte OpenGL avec le module QtOpenGL. Pour créer un contexte spécifique d'une version OpenGL, il suffit simplement d'utiliser la fonction setVersion de la classe QGLFormat :

```
#include QtOpenGL
...
```

```
QGLFormat format;
format.setVersion(4, 3);
format.setProfile( QGLFormat::CoreProfile ); // nécessite Qt
>= 4.8
format.setSampleBuffers( true );
GLWidget* w = new QGLWidget(format);
```

Pour plus de détails, voir l'article How to use OpenGL Core Profile with Qt sur le wiki de Qt.

Avec SFML 2.0

La bibliothèque SFML permet de demander lors de la création d'une fenêtre la version du contexte OpenGL à utiliser avec la classe sf::ContextSettings. S'il n'est pas possible de créer un contexte avec les paramètres demandés, SFML va créer un contexte le plus proche possible. Il est donc important de vérifier la version de contexte réellement créée.

```
#include SFML/OpenGL.hpp
...
sf::ContextSettings settings;
settings.depthBits = 24;
settings.stencilBits = 8;
settings.antialiasingLevel = 4;
settings.majorVersion = 4;
settings.minorVersion = 3;

sf::Window window(sf::VideoMode(800, 600), "OpenGL",
    sf::Style::Default, settings);

sf::ContextSettings settings = window.getSettings();
std::cout << "depth bits:" << settings.depthBits << std::
endl;
std::cout << "stencil bits:" << settings.stencilBits << std
::endl;
std::cout << "antialiasing level:" << settings.
antialiasingLevel
    << std::endl;
std::cout << "version:" << settings.majorVersion << "."
```

```
    << settings.minorVersion << std::endl;
```

Pour plus de détails, voir l'article Using OpenGL in a SFML window sur le site de SFML.

Avec SDL 1.3

La bibliothèque SDL fournit les constantes SDL_GL_CONTEXT_MAJOR_VERSION pour définir la version du contexte que l'on souhaite utiliser.

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 4);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
mainwindow = SDL_CreateWindow(PROGRAM_NAME,
SDL_WINDOWPOS_CENTERED,
    SDL_WINDOWPOS_CENTERED, 512, 512,
    SDL_WINDOW_OPENGL | SDL_WINDOW_SHOWN);
maincontext = SDL_GL_CreateContext(mainwindow);
```

Pour plus de détails, voir l'article Creating a Cross Platform OpenGL 3.2 Context in SDL sur le site de SFML.

Les modes de synchronisation (GL 4.1)

Par défaut, les appels de fonction OpenGL sont asynchrones, ce qui veut dire que les fonctions peuvent simplement envoyer les commandes aux contextes OpenGL puis passent à l'instruction suivante dans le code. Par exemple, le code suivant peut poser des problèmes puisque rien ne garantit que l'on mesure réellement le temps d'exécution de la commande :

```
boost::timer t;
glDrawElement(...);
double elapsed_time = t.elapsed();
```

Lorsque l'on débogue une application utilisant OpenGL, il peut alors y avoir un décalage entre l'appel d'une fonction et une erreur générée.

Pour faciliter le débogage, il est possible de changer le mode de synchronisation pour forcer OpenGL à attendre la fin de la commande avant de passer à l'instruction suivante dans le code. Cette fonctionnalité garantit la synchronisation dans un contexte, mais pas la synchronisation entre plusieurs contextes, qui reste sous la responsabilité de l'application.

```
// active le mode synchrone
glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
```

Il est possible de tester le mode avec glIsEnabled et de le désactiver avec glDisable. Activer cette fonctionnalité peut être très coûteux en termes de performance et doit être utilisée uniquement pour le débogage.

La pile des événements (GL 4.1)

La pile des événements, appelée « message log » contient les messages lancés dans un contexte de débogage. Cette pile est de taille limitée, ce qui signifie que si elle est pleine, les événements les plus anciens seront perdus. Il faut donc la vider régulièrement en lisant les messages. Lorsque l'on utilise plusieurs contextes, chaque contexte possède sa propre pile des événements.

La fonction glGetIntegerv permet d'obtenir des informations sur les capacités et le contenu de la pile.

```
GLint maxMessages, totalMessages, len, maxLen, lens[10];
GLenum source, type, id, severity, severities[10];

glGetIntegerv(GL_MAX_DEBUG_LOGGED_MESSAGES_ARB,&maxMessages);
printf("Nombre de messages maximum que peut contenir la pile
: %d\n",
    maxMessages);

glGetIntegerv(GL_DEBUG_LOGGED_MESSAGES_ARB, &totalMessages);
printf("Nombre de messages contenus actuellement dans la
pile : %d\n",
    totalMessages);

glGetIntegerv(GL_MAX_DEBUG_MESSAGE_LENGTH_ARB, &maxLen);
```

```
printf("Taille maximale de message : %d\n", maxLen);

glGetIntegerv(GL_DEBUG_NEXT_LOGGED_MESSAGE_LENGTH_ARB, &len);
printf("Taille du prochain message : %d\n", len);
```

Les messages contenus dans la pile (GL 4.1)

Les messages dans la pile contiennent plusieurs informations comme l'origine du message, son type et son importance.

Origine du message :

- GL_DEBUG_SOURCE_API : le message est lancé lors d'un appel d'une fonction OpenGL ;
- GL_DEBUG_SOURCE_WINDOW_SYSTEM : le message est lancé par le gestionnaire de fenêtre ;
- GL_DEBUG_SOURCE_SHADER_COMPILER : le message est lancé par le compilateur de shader ;
- GL_DEBUG_SOURCE_THIRD_PARTY : le message est lancé par une bibliothèque externe ;
- GL_DEBUG_SOURCE_APPLICATION : le message est lancé par l'application (voir la partie « Envoyer un message personnalisé ») ;
- GL_DEBUG_SOURCE_OTHER : le message ne provient pas d'une des sources précédentes.

Type de message (l'utilisation des constantes spécifiques d'OpenGL 4.3 sera décrite dans la partie « L'extension KHR_debug » de ce billet) :

- GL_DEBUG_TYPE_ERROR : message d'erreur récupérable avec glEvent ;
- GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR : utilisation fonction dépréciée ;
- GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR : appel qui provoque un comportement indéfini ;

- GL_DEBUG_TYPE_PORTABILITY : problèmes de portabilité ;
- GL_DEBUG_TYPE_PERFORMANCE : notifications de performance ;
- GL_DEBUG_TYPE_OTHER : ne correspond pas aux types précédents ;
- GL_DEBUG_TYPE_MARKER : annotation (GL 4.3) ;
- GL_DEBUG_TYPE_PUSH_GROUP : entrée dans un groupe (GL 4.3) ;
- GL_DEBUG_TYPE_POP_GROUP : sortie d'un groupe (GL 4.3).

Importance du message :

- GL_DEBUG_SEVERITY_NOTIFICATION : les messages qui ne concernent pas les performances et les erreurs ;
- GL_DEBUG_SEVERITY_LOW : les alertes de performance concernant les changements redondants de contextes, les comportements indéfinis triviaux ;
- GL_DEBUG_SEVERITY_MEDIUM : les alertes de performance sévères, les alertes de compilation ou de linkage GLSL, utilisation d'un comportement déprécié ;
- GL_DEBUG_SEVERITY_HIGH : les erreurs OpenGL, les comportements indéfinis dangereux, les erreurs de compilation ou de link GLSL.

Les messages sont identifiés de façon unique grâce à la combinaison de leur type, leur source et un numéro d'identification.

Le dernier élément composant un message est une chaîne de caractères terminée par le caractère NULL (chaîne de caractères, style C) et donnant des informations plus détaillées sur le message.

Lire les messages dans la pile (GL 4.1)

La fonction glGetDebugMessageLog permet de lire un ou plusieurs messages contenus dans la pile.

```
GLuint glGetDebugMessageLog(
    GLuint count,           // nombre de messages que l'on
souhaite
                            // récupérer
    GLsizei bufSize,        // taille du tampon de
caractères
    GLenum * sources,       // récupère les sources
    Glenum * types,         // récupère les types
    GLuint * ids,           // récupère les identifiants
    GLenum * severities,    // récupère l'importance des
messages
    GLsizei * lengths,      // tableau contenant la taille
des chaînes
                            // pour chaque message
    GLchar * messageLog);   // récupère les chaînes de
caractères des
                            // messages
```

Pour récupérer un message :

```
char* message = (char*) malloc(sizeof(char) * len);
glGetDebugMessageLog(1, len, &source, &type, &id, &severity,
NULL,
    message);
```

Pour récupérer les dix premiers messages :

```
char* messages = (char*) malloc(sizeof(char) * maxLen * 10);
glGetDebugMessageLog(10, maxLen*10, NULL, NULL, NULL,
severities, lens,
    messages);
// retourne une chaîne séparée par NULL
// lens retourne la taille de chaque message
```

Pour effacer tous les messages dans la pile :

```
glGetIntegerv(GL_DEBUG_LOGGED_MESSAGES_ARB, &totalMessages);
glGetDebugMessageLog(totalMessages, 0, NULL, NULL, NULL,
NULL, NULL,
    NULL);
```

Une erreur GL_INVALID_VALUE est lancée si le paramètre bufSize est négatif.

Envoyer un message personnalisé (GL 4.1)

Il est possible d'envoyer un message personnalisé avec la fonction glDebugMessageInsert. Cela permet d'écrire des bibliothèques graphiques ou de transmettre des erreurs entre différentes parties d'une application.

```
void glDebugMessageInsert(GLenum source, GLenum type, GLuint
id,
    GLenum severity, GLsizei length, const char * message);
```

Les paramètres sont équivalents aux autres fonctions. Si length est négatif, alors message est une chaîne de caractères terminée par NULL.

Par exemple :

```
glDebugMessageInsert(GL_DEBUG_SOURCE_APPLICATION_ARB,
    GL_DEBUG_TYPE_ERROR_ARB, 1234, GL_DEBUG_SEVERITY_LOW_ARB,
-1,
    "Un exemple de message personnalise");
```

La fonction glDebugMessageInsert peut générer les erreurs suivantes :

- INVALID_ENUM : type n'est pas une valeur valide ou source ne vaut pas DEBUG_SOURCE_APPLICATION ou DEBUG_-
- SOURCE_THIRD_PARTY ;
- INVALID_VALUE : severity n'est pas une valeur valide ;
- INVALID_VALUE : le nombre de caractères de message est supérieur à MAX_DEBUG_MESSAGE_LENGTH.

Les filtres de messages (GL 4.1)

Dans de nombreux cas, il est souhaitable de pouvoir limiter les messages, pour éviter de perdre des messages s'ils sont trop nombreux et éviter de devoir gérer un trop grand nombre de messages. Il est alors possible d'ajouter un ou plusieurs filtres, pour supprimer certains types de messages ou au contraire n'en garder que certains. Les filtres appliqués ne seront actifs que sur les nouveaux messages et non sur ceux qui sont déjà sur la pile.

La fonction glDebugMessageControl permet d'ajouter un filtre :

```
void glDebugMessageControl(GLenum source, GLenum type,
    GLenum severity, GLsizei count, const GLuint * ids,
    GLboolean enabled);
```

Les paramètres source, type et severity prennent l'une des valeurs décrites au-dessus (GL_DEBUG_SOURCE_API, GL_DEBUG_SOURCE_WINDOW_SYSTEM, GL_DEBUG_SOURCE_SHADER_COMPILER, GL_DEBUG_SOURCE_THIRD_PARTY, GL_DEBUG_SOURCE_APPLICATION ou GL_DEBUG_SOURCE_OTHER pour source, GL_DEBUG_TYPE_ERROR, GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR, GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR, GL_DEBUG_TYPE_PORTABILITY, GL_DEBUG_TYPE_PERFORMANCE ou GL_DEBUG_TYPE_OTHER pour type et GL_DEBUG_SEVERITY_LOW, GL_DEBUG_SEVERITY_MEDIUM ou GL_DEBUG_SEVERITY_HIGH pour severity) ou GL_DONT_CARE pour ne pas prendre en compte le paramètre.

Le paramètre ids permet de donner un tableau d'identifiants de taille count. Si le paramètre count est supérieur à 0, alors le filtre s'applique aux messages dont les identifiants correspondent aux valeurs données dans le tableau ids. Dans ce cas, les paramètres source et type ne doivent pas être GL_DONT_CARE et severity doit être GL_DONT_CARE.

Le paramètre enabled permet de préciser si le filtre supprime les messages correspondant aux critères (GL_FALSE) ou les garde (GL_TRUE).

Par exemple, pour garder tous les messages :

```
glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE,
GL_DONT_CARE, 0, NULL, GL_TRUE);
```

Par exemple, pour ne garder que les messages importants :

```
glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE,
GL_DEBUG_SEVERITY_LOW, 0, NULL, GL_TRUE);
```

Pour désactiver les messages correspondant à l'appel de fonctions dépréciées :

```
glDebugMessageControlARB(GL_DONT_CARE,
    GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR_ARB, GL_DONT_CARE, 0,
NULL,
    GL_FALSE);
```

Pour ne garder que les messages correspondant à une liste d'identifiants donnés :

```
// on supprime tous les messages
glDebugMessageControlARB(GL_DONT_CARE, GL_DONT_CARE,
GL_DONT_CARE,
    0, 0, FALSE);

// on ne garde ensuite que les messages possédant les
identifiants 1280 ou 1282
GLuintid[2] = { 1280, 1282 };
glDebugMessageControlARB(GL_DEBUG_SOURCE_API_ARB,
    GL_DEBUG_TYPE_ERROR_ARB, GL_DONT_CARE, 2, id, GL_TRUE);
```

La fonction GLDebugMessageControl peut générer les erreurs suivantes :

- INVALID_ENUM : utilisation d'une valeur incorrecte pour source, type ou severity ;
- INVALID_VALUE : count est négatif ;
- INVALID_OPERATION : count est positif, mais source ou type valent DONT_CARE ou severity ne vaut pas DONT_CARE.

Pour éviter d'avoir à écrire deux codes, pour les versions debug et release, il est possible de créer une macro définissant ou non une

fonction appelant les filtres :

```
#ifdef OPENGL_DEBUG
    #define GLDebugMessageControl(source, type, sev, num, id,
enabled) \\
        glDebugMessageControlARB(source, type, sev, num, id,
enabled)
#else
    #define GLDebugMessageControl(source, type, sev, num, id,
enabled)
#endif
```

Les fonctions callback (GL 4.1)

Une autre possibilité pour récupérer les messages est de créer une fonction callback qui sera appelée dès qu'un message est lancé. Les messages ne sont alors plus mis dans la pile et ne peuvent être lus avec glGetDebugMessageLog. Par contre, les filtres ajoutés avec GLDebugMessageControl continuent de fonctionner.

Une fonction callback doit avoir la signature suivante :

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

Les paramètres source, type, id, severity et message correspondent aux paramètres des messages comme décrit au-dessus. Le paramètre length est la taille de la chaîne de caractères message. La mémoire allouée pour le paramètre message est gérée directement par OpenGL. Sa portée correspond à la fonction callback, donc il est nécessaire de copier le message si on souhaite l'utiliser en dehors de la fonction. Le paramètre userParam est un pointeur vers des données pouvant être utilisées dans le corps de la fonction et passées lorsque l'on attache la fonction callback à un contexte.

Elle ne doit pas appeler de fonction OpenGL ou du système de fenêtrage. Cette fonction est appelée par un thread différent du thread appelant la

fonction GL, il faut donc faire particulièrement attention au respect du thread-safe.

Lorsque l'on travaille avec plusieurs contextes OpenGL, il faut fournir une fonction callback pour chaque contexte pour lesquels on souhaite capturer les messages. Dans une application multithread, il est possible d'avoir une fonction callback sur un contexte appelé par plusieurs threads différents, mais l'application est responsable du respect du thread-safe.

Par exemple, la fonction suivante permet d'afficher dans la console les informations des messages reçus.

```
void DebugLog(GLenumsource, GLenumtype, GLuintid,
GLenumseverity,
    GLsizeilength, constGLchar*message, GLvoid*userParam)
{
    printf("Message reçu :\n");
    switch (source) {
        case GL_DEBUG_SOURCE_API : printf("    Source :
GL_DEBUG_SOURCE_API\n"); break;
        case GL_DEBUG_SOURCE_WINDOW_SYSTEM : printf("
Source : GL_DEBUG_SOURCE_WINDOW_SYSTEM\n"); break;
        case GL_DEBUG_SOURCE_SHADER_COMPILER : printf("
Source : GL_DEBUG_SOURCE_SHADER_COMPILER\n"); break;
        case GL_DEBUG_SOURCE_THIRD_PARTY : printf("
Source : GL_DEBUG_SOURCE_THIRD_PARTY\n"); break;
        case GL_DEBUG_SOURCE_APPLICATION : printf("
Source : GL_DEBUG_SOURCE_APPLICATION\n"); break;
        case GL_DEBUG_SOURCE_OTHER : printf("    Source :
GL_DEBUG_SOURCE_OTHER\n"); break;
        default: printf("    Source inconnue\n");
    }
    switch (type) {
        case GL_DEBUG_TYPE_ERROR : printf("    Type :
GL_DEBUG_TYPE_ERROR\n"); break;
        case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR : printf("
Type : GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR\n"); break;
        case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR : printf("
Type : GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR\n"); break;
        case GL_DEBUG_TYPE_PORTABILITY : printf("    Type :
```

```
GL_DEBUG_TYPE_PORTABILITY\n"); break;
        case GL_DEBUG_TYPE_PERFORMANCE : printf("    Type :
GL_DEBUG_TYPE_PERFORMANCE\n"); break;
        case GL_DEBUG_TYPE_OTHER : printf("    Type :
GL_DEBUG_TYPE_OTHER\n"); break;
        default: printf("    Type inconnu\n");
    }
    switch (severity) {
        case GL_DEBUG_SEVERITY_LOW : printf("    Type :
GL_DEBUG_SEVERITY_LOW\n"); break;
        case GL_DEBUG_SEVERITY_MEDIUM : printf("    Type :
GL_DEBUG_SEVERITY_MEDIUM\n"); break;
        case GL_DEBUG_SEVERITY_HIGH : printf("    Type :
GL_DEBUG_SEVERITY_HIGH\n"); break;
        default: printf("    Type inconnu\n");
    }
    printf("    Message : %s\n\n", message);
}
```

La fonction glDebugMessageCallback permet d'attacher une fonction callback à un contexte. Elle prend en paramètre la fonction callback et un pointeur vers des données utilisateurs. Sa signature est la suivante :

```
void glDebugMessageCallback(DEBUGPROC callback, void*
userParam);
```

Le code d'exemple suivant présente comment attacher une fonction callback et modifier les données utilisateurs entre deux messages.

```
int myData;
glDebugMessageCallbackARB(DebugLog, &myData);

myData = 2
glEnable(GL_UNIFORM_BUFFER); // erreur

myData=3;
glPolygonMode(GL_FRONT, GL_LINE); // erreur
```

Pour supprimer une fonction callback, il suffit d'appeler glDebugMessageCallback avec des valeurs nulles ;

```
glDebugMessageCallback(0, 0);
```

La fonction glGetPointerv permet de récupérer la fonction callback en utilisant le paramètre GL_DEBUG_CALLBACK_FUNCTION et les données userParam avec GL_DEBUG_CALLBACK_USER_PARAM :

```
void (*callback) (GLenum, GLenum, GLuint, GLenum, GLsizei,
const GLchar*, void*);
glGetPointerv(GL_DEBUG_CALLBACK_FUNCTION, (GLvoid **) &
callback);

void* userParam;
glGetPointerv(GL_DEBUG_CALLBACK_USER_PARAM, (GLvoid **) &
userParam);
```

Les annotations de débogage (GL 4.3)

Un des problèmes que l'on peut rencontrer lorsque l'on débogue une application est de savoir quels objets provoquent une erreur. Il est nécessaire alors de garder une table contenant les identifiants des objets. Dans OpenGL 4.3 a été ajouté les annotations de débogage (debug label), qui permettent d'attacher une chaîne de caractères à n'importe quel objet OpenGL grâce à la fonction glObjectLabel :

```
void glObjectLabel(enum identifier, uint name, sizei length,
const char *label);
```

Le paramètre name correspond au numéro retourné par les fonctions create. Le paramètre identifier indique le type d'objet correspondant à name. Il peut prendre les valeurs suivantes :

- BUFFER
- FRAMEBUFFER
- PROGRAM_PIPELINE
- PROGRAM
- QUERY
- RENDERBUFFER

- SAMPLER
- SHADER
- TEXTURE
- TRANSFORM_FEEDBACK
- VERTEX_ARRAY

La fonction glObjectLabel peut générer les erreurs suivantes :

- INVALID_ENUM : si identifier n'est pas une valeur valide ;
- INVALID_VALUE : si name ne correspond pas à un objet valide de type identifier ;
- INVALID_VALUE : si la taille de la chaîne de caractères label est plus grande que MAX_LABEL_LENGTH.

La fonction glGetObjectLabel permet de récupérer l'annotation associée avec un objet identifié par identifier et name :

```
void glGetObjectLabel( enum identifier, uint name, sizei
bufSize, sizei *length, char *label );
```

Cette fonction peut retourner les erreurs suivantes :

- INVALID_ENUM : si identifier ne correspond pas à un type d'objet valide ou si name ne correspond pas à un objet valide de type donné ;
- INVALID_VALUE : si bufSize est négatif.

En plus de la fonction glObjectLabel, la fonction glObjectPtrLabel permet d'annoter un objet de synchronisation (un objet de synchronisation permet de tester si des commandes OpenGL sont en cours ou terminées ; voir le chapitre 4 – Event model des spécifications d'OpenGL 4.3 pour plus de détails).

```
void glObjectPtrLabel(void *ptr, sizei length, const char *
label);
```

La fonction glObjectPtrLabel peut générer les erreurs suivantes :

- INVALID_VALUE : si ptr ne correspond pas à un objet de synchronisation valide ;
- INVALID_VALUE : si la taille de la chaîne de caractères label est plus grande que MAX_LABEL_LENGTH.

La fonction glGetObjectPtrLabel permet de récupérer le label d'un objet de synchronisation :

```
void glGetObjectPtrLabel( void *ptr, sizei bufSize, size *
length, char *label );
```

Les groupes de débogage (GL 4.3)

Les groupes permettent d'annoter un ensemble de fonctions OpenGL. Un message GL_DEBUG_TYPE_PUSH_GROUP est envoyé lorsque l'on entre dans le groupe puis GL_DEBUG_TYPE_POP_GROUP lorsque l'on sort. Il est également possible d'utiliser plusieurs groupes, qui seront alors hiérarchisés dans une pile de groupes. Les filtres appliqués sont actifs sur le groupe courant et les groupes qui seront inclus dans ce groupe. Au démarrage, la pile contient un groupe par défaut. Ce système permet ainsi de retrouver plus facilement l'origine des messages.

Les fonctions glPushDebugGroup et glPopDebugGroup permettent de débuter et finir un groupe.

```
void glPushDebugGroup(enum source, uint id, sizei length,
const char *message);
void glPopDebugGroup( void );
```

Le message généré contient la chaîne de caractères message, l'identifiant id, le type sera DEBUG_TYPE_PUSH_GROUP ou DEBUG_TYPE_POP_GROUP, la source devra être GL_DEBUG_SOURCE_APPLICATION ou GL_DEBUG_SOURCE_THIRD_PARTY et severity sera GL_DEBUG_SEVERITY_NOTIFICATION.

La fonction glPushDebugGroup peut générer les erreurs suivantes :

- INVALID_ENUM : si la source ne correspond pas à DEBUG_SOURCE_APPLICATION ou DEBUG_SOURCE_THIRD_PARTY ;
- INVALID_VALUE : si la taille de message est supérieure à MAX_DEBUG_MESSAGE_LENGTH ;
- STACK_OVERFLOW : si le nombre de groupes dépasse MAX_DEBUG_GROUP_STACK_DEPTH – 1.

La fonction glPopDebugGroup peut générer une erreur STACK_UNDERFLOW si la pile de groupes ne contient que le groupe par défaut.

Notes sur les implémentations

Les situations pouvant générer des messages et le contenu de ces messages sont laissés à l'appréciation des implémentations. On observe donc des différences de comportement entre les différentes implémentations. Pour le moment, la taille de la pile semble être de 128 messages et la taille des messages au maximum de 1024 octets chez AMD et NVIDIA.

Exemples de situations et de message générés

Lorsque l'on attache un tampon mémoire sur un objet ne correspondant pas à un tampon, NVIDIA ne donne aucun message alors que AMD retourne un message signalant une fonction dépréciée :

```
glBindBuffer in a Core context performing invalid operation
with parameter « name » set to '0×5′ which was removed
from Core OpenGL (GL_INVALID_OPERATION)
```

Lorsque l'on appelle une fonction dépréciée ou avec une combinaison dépréciée de paramètres, par exemple glPolygonMode(GL_FRONT, GL_LINE), NVIDIA et AMD retournent une erreur GL_INVALID_ENUM :

```
// NVIDIA
GL_INVALID_ENUM error generated. Polygonmodes for « face »
are
disabled in the current profile.
// AMD
Using glPolygonMode in a Core context with parameter
« face » and enum ’0×404′ which was removed from
Core OpenGL (GL_INVALID_ENUM)
```

Les messages peuvent également servir lorsqu’il n’y a pas d’erreur, par exemple pour faire le suivi des appels de fonction. Par exemple, lorsque l’on attache un tampon mémoire et que les données sont correctement envoyées, NVIDIA lance le message suivant en GL_DEBUG_SEVERITY_LOW alors que AMD ne lance rien.

```
Buffer detailed info: Buffer object 3 (bound to
GL_ELEMENT_ARRAY_BUFFER_ARB, usage hint
is GL_ENUM_88e4) will use VIDEO memory as
the source for buffer object operations.
```

Lorsqu’on appelle la fonction glDrawElement pour lancer un rendu alors que la mémoire graphique est pleine, NVIDIA va générer le message suivant :

```
Unknown internal debug message . The NVIDIA
OpenGL driver has encountered an out of memory
error. This application might behave inconsistently and fail.
```

Pour terminer, une situation posant des problèmes de performance. Lorsque l’on utilise un nombre important de Vertex Array Object de petites tailles, NVIDIA lancera un message signalant une perte de performance alors que AMD ne donne aucun message.

Sources et remerciements

Certains exemples (en particulier les messages d’erreurs générés dans certaines situations) proviennent du livre OpenGL Insight de Patrick Cozzi et Christophe Riccio, dont certains chapitres sont accessibles gratuitement sur le site du livre. Merci également à Christophe Riccio

pour ses revues des spécifications d'OpenGL qu'il publie sur g-truc. Vous pouvez également consulter les spécifications d'OpenGL 4.3, en particulier le chapitre consacré au débogage : « Chapter 20 – Debug Output ».

Merci à f-leb pour sa relecture orthographique.

OpenGL