Programmation OpenGL partie 01

Bref historique
et
Introduction

M. Malek Bengougam
malek.bengougam@gmail.com

Partie 1 - Bref historique et Introduction

Qu'est-ce qu'OpenGL ?

- Librairie orientée rendu graphique créée en 1992 par Silicon Graphics (**SGI**) sur la base de leur librairie interne **IrisGL**.

- Depuis plusieurs années de nombreux acteurs ont contribué à l'extension d'OpenGL standardisé par *l'Architecture Revision Board*. C'est maintenant le consortium **Khronos Group** qui est en charge de son développement et de la standardisation.

- Développée en C mais il existe des versions compatible avec la plupart des langages.

Cf. http://fr.wikipedia.org/wiki/OpenGL , ou, pour plus de détail http://en.wikipedia.org/wiki/OpenGL

Pourquoi OpenGL ?

- Depuis l'arrivée de l'accélération matérielle de la 3D, OpenGL est devenue l'API de référence.
- API standardisée par l'**ARB** (*Architecture Revision Board*) puis par le Khronos Group (**KHR**).
- Tout constructeur qui se respecte propose une version de pilotes OpenGL (même minimale).
- Beaucoup d'API propriétaires comme le vénérable Glide/3dfx ou celles de Sony et Nintendo s'en inspirent.

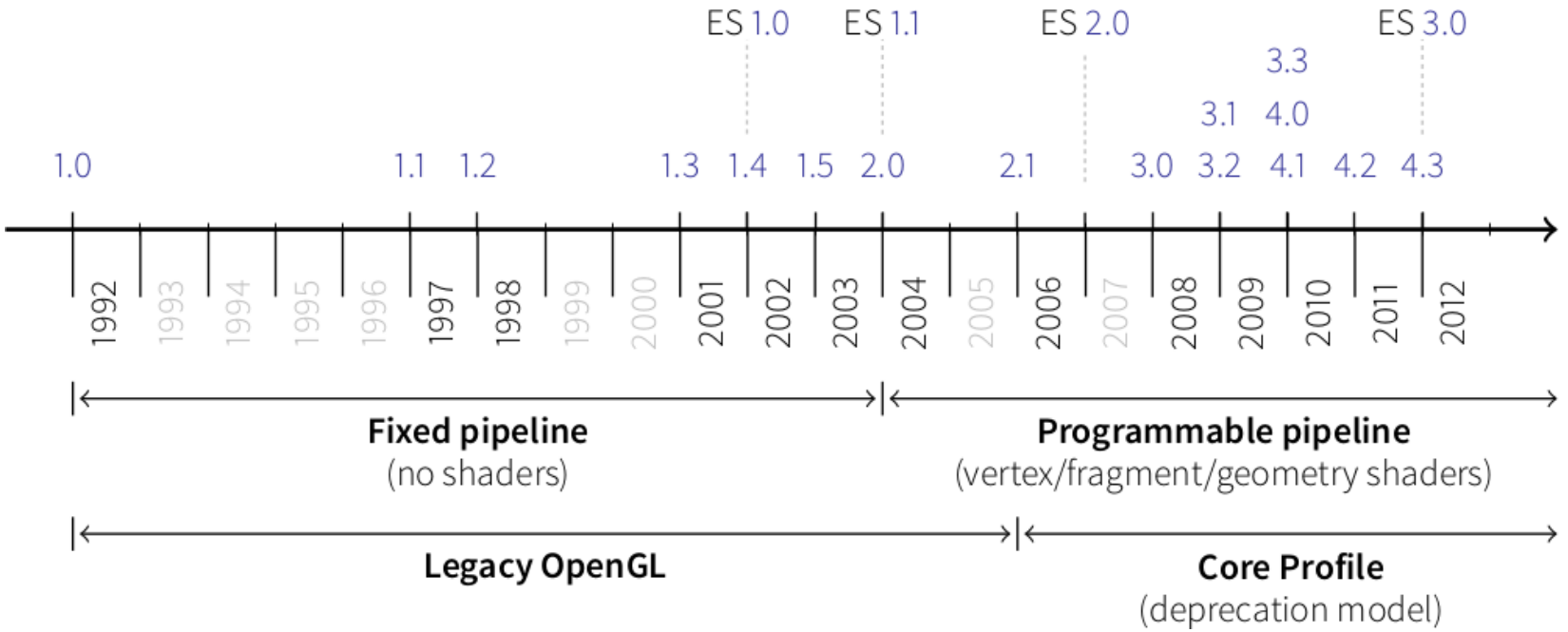
OpenGL et ses déclinaisons

- **OpenGL 1.0** - version historique
- **OpenGL 1.1** - support des extensions
- **OpenGL 1.2** - normalisation de l'API

A partir de cette version les fonctionnalités sont séparées en trois branches:
- fonctionnalités obligatoirement présentes (***core***),
- extensions approuvées (préfixes **KHR**, **ARB**),
- extensions officielles (**EXT**) et spécifiques (NV pour Nvidia, AMD etc…).

- Jusqu'à la version **1.5** OpenGL a tenté de suivre le rythme impulsé par son concurrent direct (et propriété de **Microsoft**): **Direct3D**.

Frise sur l'évolution d'OpenGL



OpenGL: évolution et problématiques

- Paradoxalement, le principal frein à l'évolution d'OpenGL est son long ancrage (25 ans+).
- OpenGL se devait de rester retro-compatible avec la version 1.0, quand bien elle se révèle lente et inadaptée aux architectures modernes.
- OpenGL ES 1.0 fut la première version allégée adaptée aux supports embarqués, l'occasion d'un premier nettoyage.
- L'ajout de nouvelles fonctionnalités programmable au niveau du hardware (shaders) est un changement important de paradigme…nécessite de repenser l'API.

OpenGL 2.0 un premier tournant

- L'arrivée des shaders programmables a tout bouleversé, et **DirectX** en a profité pour devenir l'API préférée des développeurs de jeu du fait de la collaboration étroite entre Microsoft et les constructeurs tels **Nvidia**, **ATI** (racheté par **AMD**) et **Intel**.

- En 2004, à la sortie d'OpenGL 2.0, DirectX9 s'est déjà largement imposé avec un support complet 3ème génération de shaders (Shader Model 3).

- Mais le véritable tournant pour OpenGL c'est l'accélération matérielle (et donc ainsi la 3D) qui fait son entrée dans le monde de la téléphonie mobile et plus globalement de l'embarqué (Tv, Box...)

OpenGL 3, le renouveau

- **OpenGL 3.0**, décision de l'abandon progressif des fonctionnalités historiques, et focalisation sur le hardware moderne. Ajout d'une rétro-comptabilité optionnelle (profil ***compatibility*** aussi appelé *'legacy',* héritage).
- le profil ***core*** contient de nombreux changements au niveau de l'API. Le profil ***compatibility*** permet d'utiliser l'ancienne API avec les bénéfices de 3.x…
- …cependant le profil ***compatibility*** n'est pas rendu obligatoire par les spécifications et les pilotes sont souvent limites voire problématiques.

- **OpenGL 3.1**, les fonctions marquées pour la déprécation ont été supprimée.

- **OpenGL 3.2**, proche de D3D10. Sous OSX seule version avancée jusqu'à Maverick !

- **OpenGL 3.3**, support total D3D10.x. De nombreux ajouts et changements de l'API pour le support du hardware moderne. GLSL 330 !

- **OpenGL 4.6**, dernière version à ce jour (aout 2017), orienté vers les cartes graphiques les plus récentes, avec un changement sensible de l'API. OS X limité à la version 4.1 ?!
  - La compatibilité entre OpenGL 4.3+ est OpenGL ES 3.2 est totale.

Un autre tournant: le mobile

- **OpenGL ES 1.0** reprend les specs d'OpenGL 1.3. Sur PlayStation 3, Sony et Nvidia ont implémenté une version custom (PSGL) basée sur OpenGL ES !

- **OpenGL ES 1.1** est l'API de référence des premiers smartphone et tablette sous iOS (iPad et iPhone jusqu'au 3g) et Android 1.6… ainsi que de la 3DS !

- **OpenGL ES 2.0** ajoute le support des shaders et devient la version référence de l'embarqué.

- **OpenGL ES 3.0** et **ES 3.1** basés respectivement sur OpenGL 3.3 et OpenGL 4.3.

- **OpenGL ES 3.2** inclus les fonctionnalités avancés d'OpenGL 4.x

WebGL – OpenGL pour le Web

- **WebGL**, déclinaison navigateur web. Basé sur les *specs* d'OpenGL ES 2.0.
  - Comme nous le verrons sans doute, le fait que le langage de développement soit le Javascript, tout comme les problématiques de sécurité inhérentes au Web, font que certaines fonctionnalités sont différentes d'OpenGL ES.
- **WebGL 2** est en cours de déploiement sur la base des fonctionnalités d'OpenGL ES 3.2. Actuellement seuls Google Chrome et Mozilla Firefox supportent WebGL 2.

OpenGL durant la formation

- Etudes des fonctionnalités communes d'OpenGL et OpenGL ES 2.0 / WebGL et des spécificités d'OpenGL ES 2.0.

- Pas d'étude des fonctions obsolètes et/ou dépréciées d'OpenGL 1.x et 2.x ni des fonctionnalités non supportées en OpenGL ES.

  OpenGL 1.x envoyait au matériel les données liées aux vertices une à une.

  *Ce mode de rendu est appelé **mode immédiat**. Ce mode a été supprimé dans toutes les versions d'OpenGL ES. .*

- Focus sur les **shaders** (*vertex shaders* et *fragment shaders* seulement). Ceci exclus dont l'étude du pipeline dit « fixe » (dont OpenGL ES 1.x).

- Passage progressif vers OpenGL (ES) 3.+ et apprentissage des shaders avancés (geometry shaders, compute shaders, tesselation shaders…).

Préparer le terrain

- S'assurer que les pilotes sont à jour. Attention, tous les OS ne fournissent pas des pilotes OpenGL par défaut!

  .Windows: opengl32.dll fourni le strict minimum (OpenGL 1.1 !). Il faut aller sur le site du constructeur (Nvidia, AMD, Intel…) ou du vendeur (Asus, Gigabyte…).

  .Linux: idem, mais il existe aussi des pilotes libres souvent très corrects (Mesa, Gallium…).

  .Mac OS X et iOS : chaque MàJ de l'OS contient une nouvelle version des pilotes.

  .Androïd et Blackberry : même chose.

- Créer manuellement un contexte de rendu spécifique à l'OS

  .Windows: **WGL** (opengl32.dll)

  .Linux/X11: **GLX**

  .Mac OS X: **CGL**, **AGL** (Carbon, C/C++ 32-bit), **NSOpenGL** (Cocoa, obj-C)

  .Mobiles et embarqué: **EGL** (iOS: **EAGL,** avec **'A'** pour … Apple)

  .web: **Canvas3D** (webGL)

- Ces API permettent aussi d'accéder aux extensions, de définir la destination du rendu et bien d'autres choses dépendantes de la plateforme.

3.1 Utiliser une libraire d'aide (*helper*)

- Plusieurs librairies existent (liste non exhaustive):

- **Fenêtrage**: GLUT, qui, en interne, utilise WGL, GLX, CGL en fonction de la plateforme (OpenGL 1.x et 2.x). Une version OpenSource et compatible OpenGL 3+: FreeGLUT.
- Alternativement, avec une API complètement différente, et un meilleur support des architectures modernes GLFW.

- **Extensions**: GLEW ou GLEE … ou encore manuellement à l'aide de WGL, GLX, CGL, NSGL ou EGL, EAGL (*embedded*).

- **Clé en main**: Qt, SDL ou SFML ,mais cela laisse peu de liberté quand à l'implémentation de la boucle principale.

Vous aurez besoin

- Sous Windows: de Visual C++, la version « express » ou « community » peut suffire.
- Sous Linux: vim + gcc/clang ou votre IDE préféré.
- Sous OS X: d'au minimum Leopard ou Snow Leopard mais de préférence Maverick. Et surtout Xcode >= 5.
- *Possibilité d'utiliser un autre IDE si vous le souhaitez mais il ne tient qu'à vous de bien le configurer…*

Pilotes OpenGL

- Sous Windows et Mac OS X tout est fonctionnel par défaut. Mais sous Windows il faut tout de même avoir les pilotes propriétaires pour l'accélération matérielle.

- Sous Linux, vérifier la présence d'OpenGL:

```
glxinfo | grep OpenGL
```

- Récupérer éventuellement les DRI/DRM de Mesa/Gallium ou chez AMD/Nvidia/Intel si existant.

Fichiers d'entêtes

- **Sous Windows et Linux:**
  <GL/gl.h>
- Et récupérer ces fichiers sur ***https://www.khronos.org/registry/OpenGL/index_gl.php***
  <GL/glext.h> // definition des extensions OpenGL
  <GL/**wgl**ext.h> // extensions specifique au render context Windows !
  <GL/**glx**ext.h> // extensions specifique au render context Unix / Linux !
  <GL/glcorearb.h> // non obligatoire

- **Sous Mac OS X** (tout est fourni par défaut)**:**
  <OpenGL/gl.h>
  <OpenGL/OpenGL.h> // optionnel, CGL seulement (Carbon, 32 bit seulement)

- **Sous iOS:**
  <OpenGLES/ES2/gl.h>
  <OpenGLES/ES2/glext.h>

- **Sous Android:**
  <GLES2/gl2.h>
  <GLES2/gl2ext.h>

Librairies

- Sous Windows la seule librairie requise au lien est ‘opengl32.lib’
- Sous Linux, ‘libGL.so’
- Sous Mac OS X et iOS il suffit d’ajouter ‘OpenGL.framework ou OpenGLES.framework’
- Sous Android la librairie se nomme GLESv2.lib ou libGLESv2.lib
- Sans oublier, pour OpenGL ES, EGL et EAGL.

Pour Windows

- https://www.khronos.org/registry/OpenGL/index_gl.php
- Récupérez « gl/glext.h »

  et « gl/wglext.h » ou « gl/glxext.h »

http://www.transmissionzero.co.uk/software/freeglut-devel/

Prendre la version **Freeglut 3.0.0 MSVC**

- **GLEW** glew.sourceforge.net prendre les binaires win32/win64

FreeGLUT (1)

- http://freeglut.sourceforge.net/docs/api.php
- Sources disponibles ici: http://freeglut.sourceforge.net/

- Distributions binaire:

- Sous Windows: télécharger les headers + libs
- http://www.transmissionzero.co.uk/software/freeglut-devel/

- Sous Linux: récupérer freeglut3-dev via un package manager (ce qui installera aussi les dépendances, en l'occurrence les libs et includes GL).
- Si cela ne compile pas, récupérer en plus mesa-common-dev.

FreeGLUT (2) - Compilation et linkage

- N’oubliez pas de spécifier les chemins vers les includes et libs dans votre IDE (ou en paramètre de GCC si ils ne se trouvent pas dans les répertoires standards).

- Dans le code «GL/glut.h» ou « GL/FreeGlut.h ».

- Sous Windows: glut32.lib dans les options du projet, ou dans le code via un

  #pragma comment(lib, « FreeGlut32.lib »)

- Sous Linux: –Lglut pour GCC / Clang en ligne de commande

GLFW et GLEW

- GLFW est une API de fenêtrage plus récente et mieux supportée que FreeGLUT:
- http://www.glfw.org/download.html

- GLEW est nécessaire sous Windows et Linux afin de récupérer les extensions disponibles sans efforts.
- http://glew.sourceforge.net/index.html
- http://glew.sourceforge.net/install.html

Partie 2

OpenGL et le système d’exploitation

Le chaînon manquant: le pilote

- Il ne suffit pas de définir une norme, il faut aussi pouvoir communiquer avec le hardware.

- C'est le rôle des pilotes (*drivers*, **SW**) de faire le lien entre le système d'exploitation (**OS**) et le matériel (**HW**).

- Les performances d'un rendu 3D tiennent tout autant du matériel que de la qualité des pilotes!

Le pilote et l'OS

- OpenGL n'est au final qu'un ensemble de fonctions permettant de faire du rendu graphique (GL = Graphics Library).

- Le pilote fait le lien entre l'API, l'OS et le HW.

- OpenGL a particulièrement besoin que l'OS lui fournisse deux éléments:

  1. Un contexte de rendu (***render context***).

  2. Une zone mémoire (***surface***, ***canvas***) issue du « ***frame buffer*** » pour l'affichage (***display***).

Le contexte de rendu

- Il fait le lien entre OpenGL et l'OS :

- Quelle(s) version(s) des pilotes sont disponibles ?
- plusieurs contextes sont-ils utilisables en parallèle ?

- chargement des extensions depuis une librairie dynamique spécifique à l'OS, *shared object* (.so) ou *dynamic linkable library* (.dll)

- Des librairies haut niveau facilitent la création de contexte sur plusieurs plateformes (GLUT, Qt, SDL, GLFW, EGL etc…)

Le canevas (*canvas*)

- OpenGL doit savoir où dessiner. On parle de surface de rendu (***render surface***).

- Cela peut-être dans l'ensemble de l'affichage ou bien dans une portion de celui-ci (fenêtre, *canvas* html5…) … voire même sur une autre machine.

- L'OS n'utilise pas forcément la mémoire vidéo pour gérer son affichage. Il peut très bien utiliser la mémoire centrale ou périphérique.

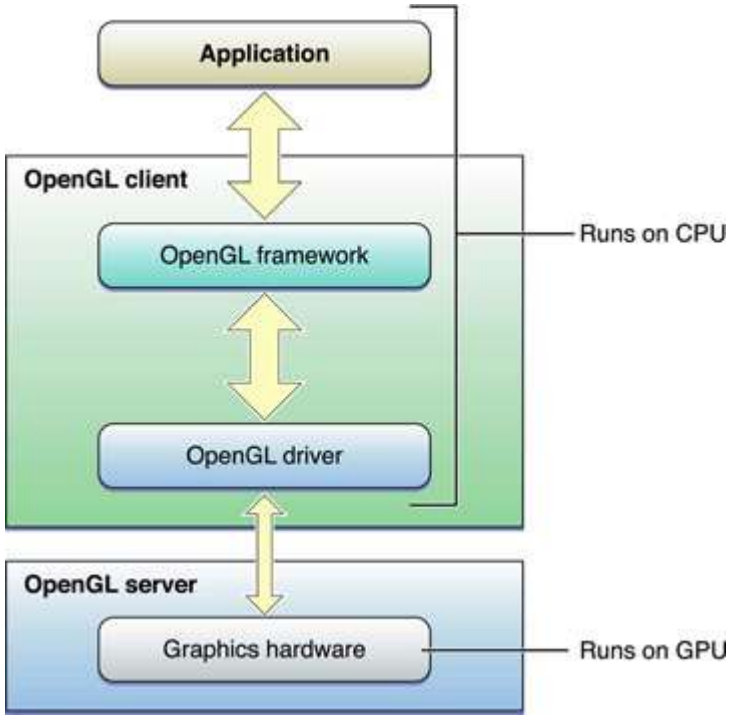
OpenGL: une machine à état et un modèle client/serveur

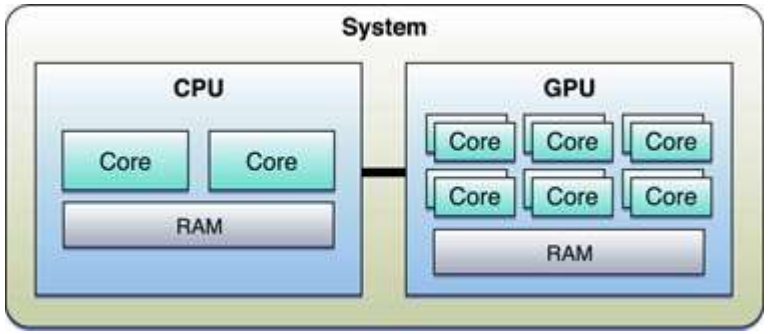
- OpenGL est configurable par le biais d'états gérant les différents aspects du rendu (***render states***).

- OpenGL établi une communication entre une partie dite cliente créant des commandes (ici le **CPU+Ram**) et une autre partie, le serveur, exécutant ces commandes (souvent **GPU+Ram Vidéo**), et ce, pas nécessairement sur la même machine.

  - Note: le serveur peut aussi être le CPU dans le cas d'une émulation logicielle.

Exemple (sous Mac OS X)





cf. [OpenGL on the Mac Platform](), sur le site developer.apple.com

OpenGL 1.x et OpenGL ES 1.x

- Avant l'arrivée des shaders programmables, le rendu était codé en « dur ». Certaines étapes (*stages*) étaient néanmoins configurables.
- On parle ainsi de «pipeline à fonctionnalité fixe» (***fixed function pipeline***).
- Le programmeur a un contrôle limité et se borne à activer ou désactiver des états de rendu (***render states***), sans customisation possible.
- OpenGL est bâti sur ce principe même qui est celui d'une machine à état. Les versions OpenGL 1.0 à 1.5 ne supportent que le pipeline fixe…
- …ou presque. Par le biais des extensions, et si les pilotes et le matériel le supportent, on peut accéder à de nouvelles fonctionnalités.

Un exemple minimal et classique avec GLUT…

```
#include <GL/glut.h>

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
  glEnd();
  glutSwapBuffers();
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutCreateWindow("Hello World");
  glutDisplayFunc(display);
  glutMainLoop();
}
```

Listing 1.1. Hello World.

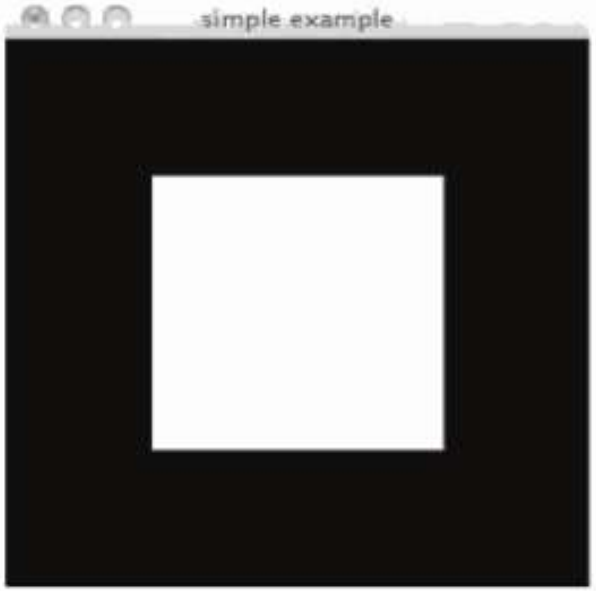


Figure 1.1. Hello World output.

Tiré de OpenGL Insights, Chap 1, « Teaching Computer Graphics Starting With Shader-Based OpenGL »

…qui pose de nombreux problèmes.

- D'une part, **GLUT** est une librairie de fonctions utilitaire pour OpenGL (GL Utility Toolkit) mais qui utilise fortement le mode immédiat.

- Le code utilise le mode immédiat : tout ce qui se trouve entre **glBegin**() et **glEnd**() inclus a été supprimé des versions ES (y compris WebGL)!

- **Ce code, très simple et pratique en apparence, ainsi que GLUT sont incompatibles avec toutes versions d'OpenGL ES !**

- De plus OpenGL ES ne support plus les polygones, les quadrilatères et les surfaces. Seules les variantes de points, lignes et triangles sont « rasterisables » comme nous le verrons.

Fonctions obsolètes (1)

- **glCallList**() et toutes les fonctionnalités liées aux display lists,
- **glVertexPointer**(), **glTexCoordsPointer**(), **glNormalPointer**(), etc…
- **glMaterial**{}()

- **glBegin**()
- **glEnd**()

- Ainsi que tout ce qui peut aller entre glBegin() et glEnd() comme:

- **glVertex**{1,2,3,4}{if…}[v]()
- **glColor**{1,2,3,4}{if…}[v]()
- **glNormal**{1,2,3,4}{if…}[v]()
- **glTexCoord**{1,2,3,4}{if…}[v]()
- **glMultiTexCoord**{1,2,3,4}{if…}[v]()

Fonctions obsolètes (2)

- Toutes les fonctions de manipulation sur les piles de matrices prédéfinies, ainsi que leurs variantes:

- **glRotate(), glTranslate(), glScale(),**
- **glMatrixMode(),**
- **glLoadIdentity(), glLoadMatrix()**
- **glPushMatrix(), glPopMatrix()**
- **glFrustum(), gluPerspective(), gluLookAt()**

- … et pas mal d'autres fonctions encore…

- Bien que toutes ces fonctions n'existent plus en OpenGL ES 2/3 elles n'ont pas pour autant disparu en OpenGL standard (sauf si vous forcez le profil « forward ») … elles sont certes dépréciées (et non recommandées) mais toujours utilisables.

Partie 3

OpenGL et la carte graphique

Le pilote et le matériel

- Sur PC, Direct3D est une certification HW !

Shader Model	Classe HW	OS min.	OpenGL	Cartes
1.0 à 1.4	DX8	WinXP / OS X Tiger	1.2-1.5 avec ext.	GeForce3/4 Radeon série R200 (8000/9000) Intel GMA
2.0	DX9	WinXP / OS X Tiger	2.0	GeForce FX Radeon R300 (9500,x700,x800,Xpress) Intel GMA (avec émulation SW)
3.0	DX9 / DX9c	WinXP / OS X Leopard	2.1	GeForce série 6,7 Radeon R500 (x1300->x1950) Intel GMA x3000/x3150
4.0/4.1	DX10 / DX10.1	Vista / OS X Lion	3.x, ou 2.x + ext	GeForce série 8/9 et 100/200/300 Radeon HD2000/HD3000/HD4000 Intel GMA x3500/x4500 (4.0) Intel chipset Ironlake, Sandy Bridge
5.0	DX11	Windows 7 / OS X Maverick à El Capitan	4.x, ou 3.x + ext	GeForce série 400/500/600/700 Radeon HD5000/HD6000/HD7000 Intel Ivy Bridge
5.0	DX11.1/DX11.2	Windows 8 / 10	4.x	Nvidia Kepler support partiel… Radeon HD77xx->7900/HD8530+ Intel Haswell

OpenGL ES et GPU compatibles

Shader Model	Classe HW	Android	iOS	GPUs min
2.0/3.0	OpenGL ES 2.0 WebGL 1.0	Android 2.2	3GS, iOS 3.x	Power VR SGX / Apple A4+ ARM Mali 400 Qualcomm Adreno 200 Vivante GC400
4.0/4.1	OpenGL ES 3.0	Android 4.3	iOS 7.x	Power VR serie 6 / Apple A7+ ARM Mali-T600 Qualcomm Adreno 300/400 Vivante GC800/1000->8000
4.1 + Compute	OpenGL ES 3.1 WebGL 2.0	Android 5.0	iOS 9	PowerVR serie 6 et 7 / Apple A9+ ARM Mali-T6xx->
Qualcomm Adreno 400/500 Vivante GC2000->8000+ Nvidia Tegra K1 et X1
5.0	OpenGL ES 3.2	Android 6.0		Qualcomm Adreno 420->
ARM Mali-t760 ->

Quelques statistiques (oct. 2018)

- OpenGL ES 1.1: 0,0% des machines sous Android
  http://developer.android.com/about/dashboards/index.html
- Et…0% des jeux iOS développés avec *Unity3d* pour le Web
  http://stats.unity3d.com/mobile/gpu-ios.html

- Concernant le PC, moins de 1% des utilisateurs de la plateforme marchande *Steam* disposent d'une carte DX8 (OpenGL 1.5+) ou sans support des shaders.
  http://store.steampowered.com/hwsurvey (tableau milieu-haut)

- Il est donc aisé d'affirmer que plus de 98% des cartes graphiques de ces utilisateurs supportent au moins le Shader Model 3.0, et donc, OpenGL 2.1.
  - Quand aux plateformes mobiles, l'API de référence est maintenant OpenGL ES 3.+ avec 78,9% de part de marché sur Android.

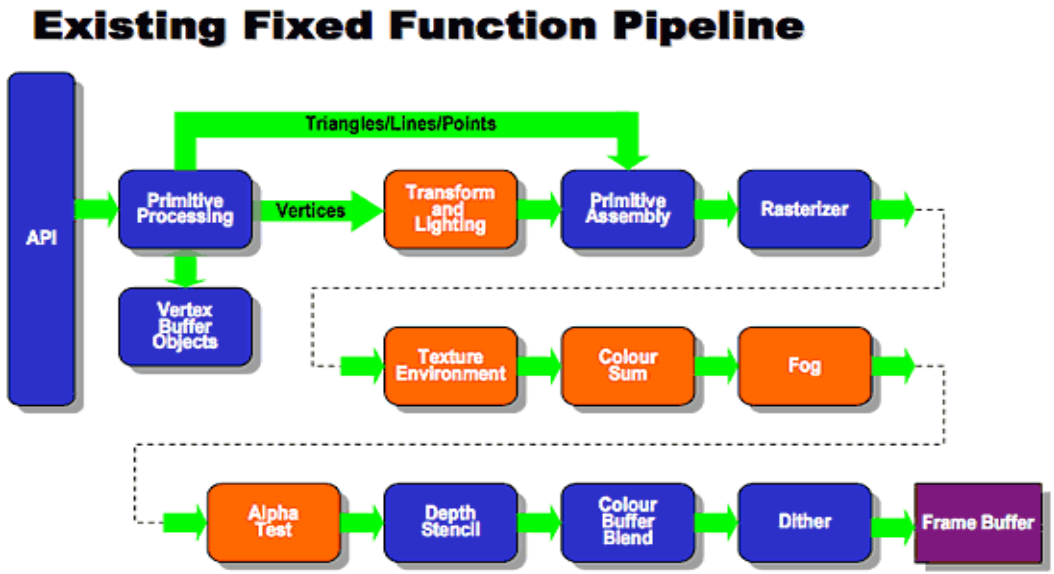
Support WebGL

- https://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists

Navigateur	Windows	Mac	Linux	Mobiles
Firefox	Windows XP Nvidia >= 257.21 ATI/AMD >= 10.6 Intel >= 09/2010	MacOS X 10.6	Firefox 4 à 5 : Nvidia only Firefox6+: idem Win XP Mesa >= 7.10.3	
Chrome	Windows XP Nvidia >= 257.21 ATI/AMD >= 10.6 Intel >= 14.42.7.5294		Idem WinXP	ARB_robustness ou EXT_robustness requis
Safari				
Edge				

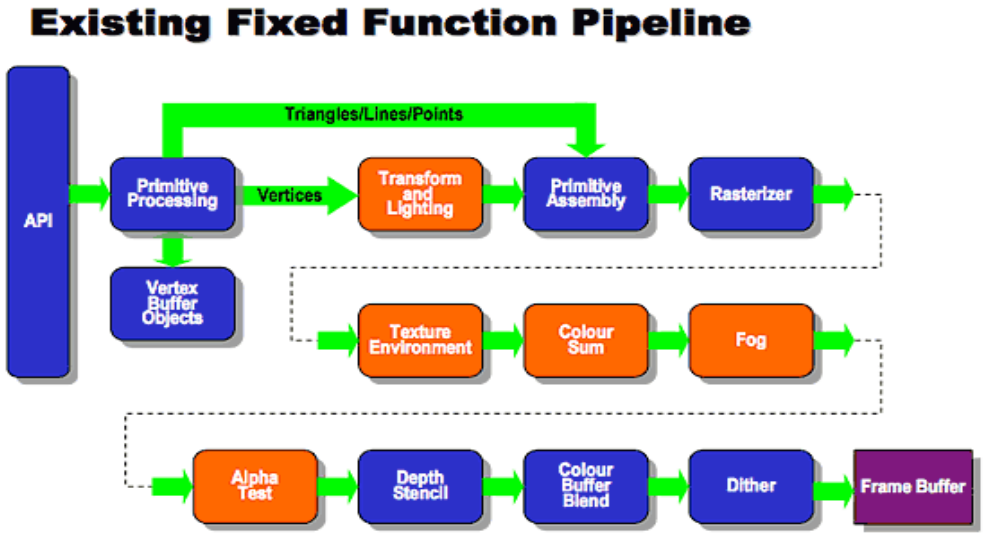
Pipeline fixe (OpenGL ES 1.1)

Multiples étapes (stages)



- Les pilotes OpenGL (ici nommés ***API*** dans le schéma) se chargent essentiellement du traitement des vertices (***Vertex Processing***) et de la gestion du stockage mémoire des **vertices** (*Vertex Array* ou *Vertex Buffer*)

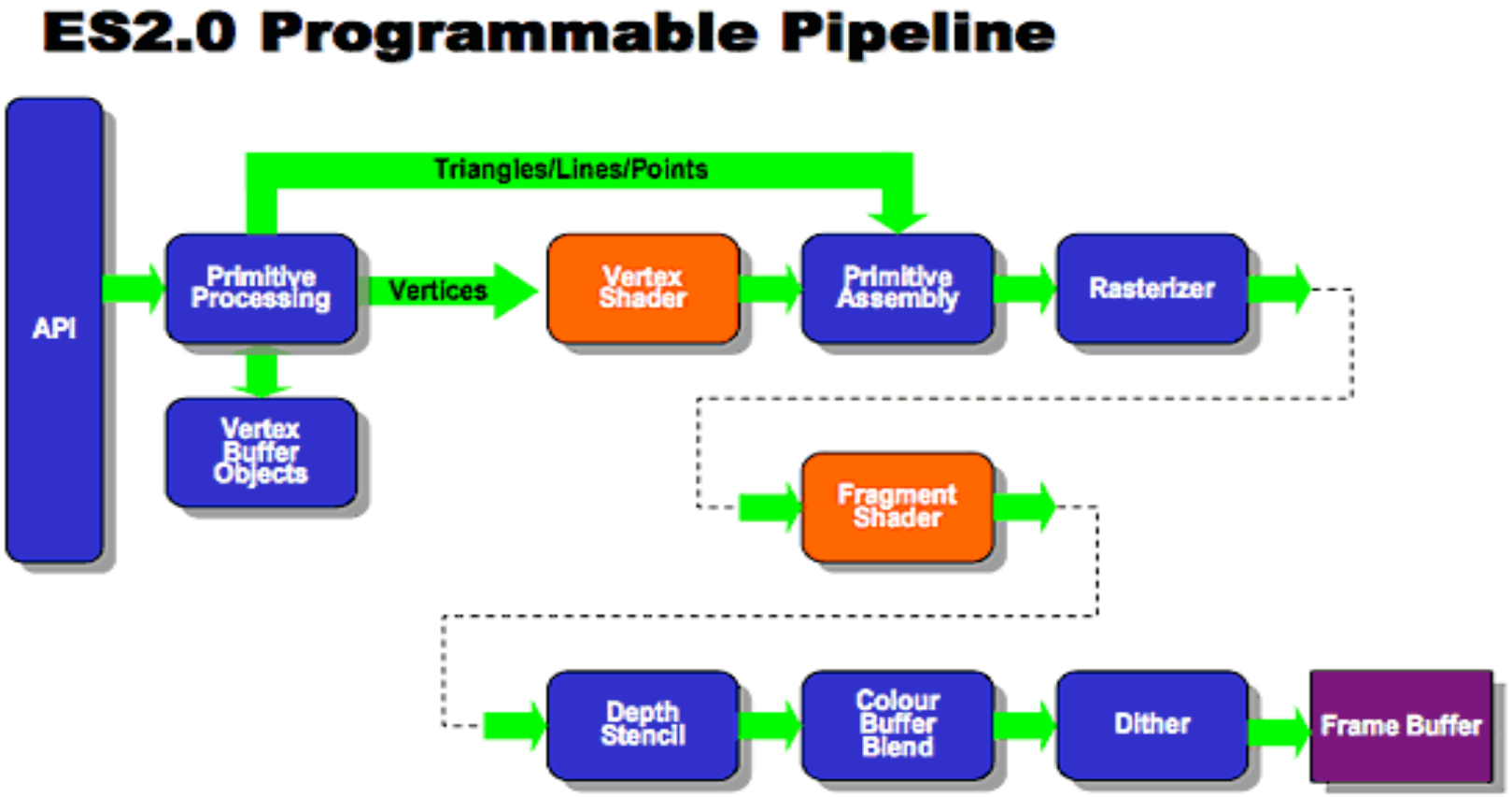
Multiples étapes (stages)



- Toutes les étapes suivantes relèvent du processeur graphique (**GPU**, *Graphics Processing Unit*). Les pilotes se chargent de transmettre au GPU la configuration des étapes fixes telle que voulue par le programmeur.

- Les étapes en orange sont dépendante du HW. Deux applications sur une même machine peuvent avoir un rendu similaire du fait que tout est précâblé / prédéfini.

Pipeline programmable (OpenGL ES 2.0)



Note: Le pipeline d'OpenGL 2.0/2.1 est très fortement similaire à celui d'OpenGL ES 2.0 si l'on exclus le mode immédiat et d'autres aspects désuets de l'API qui ont persisté pour des raisons de retro compatibilité.

On remarque que tout ce qui avait trait au traitement des vertices et des fragments générés par le *rasterizer* a été remplacé par deux étapes programmables: **Vertex Shading** et **Fragment Shading**

Deux étapes programmables

- ***Vertex processing*** : remplace le module fixe de transformation et illumination (***TnL****, Texture and Lighting*). Cette étape est totalement customisable, et permet modifier les attributs (*position, couleur, normale, coordonnées de textures*…) des sommets.

- ***Fragment processing*** : remplace toutes les étapes de calcul de la couleur d'un fragment. On peut dès lors influer sur l'aspect final d'un pixel à l'écran. Les fragments sont générés par l'étape de ***rasterisation*** qui consiste à interpoler les attributs des sommets et appeler le fragment shader durant l'interpolation.

Mais avant cela…

Partie 4. La syntaxe

préfixes

- Toutes les fonctions OpenGL sont préfixées par **gl**. Exemple: **glEnable**();

  *petite exception: certains langages comme **Javascript** et d'autres peuvent utiliser n'importe quel préfixe du fait que l'on passe via un Framework.*

- Les états et paramètres standard d'OpenGL sont des constantes, toujours en majuscule, et préfixées par **GL_**. Exemple: **GL_DEPTH_TEST**.

Exemple: glEnable(GL_ALPHA_TEST)

- **glEnable**() sert à activer un état de rendu; à contrario
- **glDisable**() désactive l'état (*render state*)

- **GL_ALPHA_TEST** active le test de comparaison de la composante alpha d'une couleur RGBA. Toute valeur au delà d'un seuil défini par le programmeur est ignorée, cependant…

- ATTENTION…**l'Alpha testing n'est plus géré** directement par l'API et le HW en OpenGL 3.x et OpenGL ES 2+!
  - Il faut le coder explicitement à l'aide d'un Fragment Shader comme on le verra dans une partie ultérieure.

- GL_ALPHA_TEST n'est donc pas défini pour ces versions d'OpenGL mais reste disponible (voire émulé) en OpenGL 2.0/2.1

suffixes

- Certaines fonctions ont plusieurs variantes, avec des paramètres (éventuellement multiples) de types différents
- Exemple:

  . glVertexAttrib**1i**()
  . glVertexAttrib**2f**()
  . glVertexAttrib**4fv**()

- Plusieurs combinaisons existent mais l'ordre reste le suivant

1. En **vert olive**, le nombre de valeurs ou composantes
2. En **rouge grenadine**, le type de donnée
3. En **bleu foncé**, **v**, indique un passage par adresse (pointeur)

Même exemple tiré des specs ES 2.0

void VertexAttrib**{**1234**}**f**[**v**]**(**uint** index, T values)

- Entre **{}** : au moins une de ces valeurs est obligatoire
- Entre **[]** : indique une alternative non obligatoire
- Le type est toujours obligatoire mais peut être multiple.

- On comprend ici que, pour cette fonction, OpenGL ES 2.0 n'accepte que le type réel à virgule flottante (**f** pour float), et que la fonction se décline de 1 à 4 paramètres de ce type.

- On remarque aussi l'absence des préfixes dans les specs (ici, document listant les différences entre OpenGL 2.0 et ES 2.0)

http://www.khronos.org/registry/gles/specs/2.0/es_cm_spec_2.0.24.pdf , page 5

Tableau synthétique des types (non exhaustif)

Suffixe	Typedef GL	Type C++	Taille mémoire	Paramètre OpenGL
b	byte	char	1 octet	**GL_BYTE**
s	short	short	2 octets	**GL_SHORT**
i	Int	int	4 octets	**GL_INT**
f	float	float	4 octets	**GL_FLOAT**
d	double	double	8 octets	**GL_DOUBLE**
ub	ubyte	unsigned char	1 octet	**GL_UNSIGNED_BYTE**
us	Glushort	unsigned short	2 octets	**GL_UNSIGNED_SHORT**
ui	GLuint	unsigned int	4 octets	**GL_UNSIGNED_INT**
GL ES 1.1 seulement, désuet	GLfixed	inexistant (spécifique GL)	4 octets	**GL_FIXED**

Autres types (moins usités)

- **GLboolean**, deux valeurs **GL_TRUE**, **GL_FALSE**
- **GLvoid**, type opaque sans taille réelle
- **GLsizei**, indique une taille ou capacité
- **GLenum**, enumération typée
- **GLbitfield**, défini un masque binaire
- **GLclampf**, une valeur normalisée entre **0.0** et **1.0**

Autres exemples (issus des specifications ES 2.0)

void Uniform{1234}{if}(**int** location, T value)

Variantes possibles :

. **gl**Uniform4f(**GLint** location, **GLfloat value0**, **GLfloat value1**, **GLfloat value2**, **Glfloat value3**);
. **gl**Uniform2i(**GLint** location, **GLint value0**, **GLint value1**);

void Uniform{1234}{if}**v**(**int** location, **sizei** count, T **valeurs**)

Variantes possibles :

. **gl**Uniform3f**v**(**GLint** location, **GLsizei** count, **GLfloat * valeurs**);
. **gl**Uniform4i**v**(**GLint** location, **GLsizei** count, **GLint * valeurs**);

Partie 5. Le Frame Buffer

Render Buffers

- Un Render Buffer est une zone mémoire dite tampon (buffer) dédiée au stockage du résultat d'un rendu graphique.
- Le Frame Buffer peut contenir plusieurs tampons de rendu.
- Il y'a toujours au moins 1 *buffer* dédié au stockage de la couleur (*color buffers*).
- Le Frame Buffer contient aussi les tampons de profondeur (*depth buffer*) et de « pochoir » (*stencil buffer*).
- C'est le rôle du contexte (et donc de l'OS et des pilotes) de fournir à OpenGL l'accès au Frame Buffer.

Les Color Buffers (1)

- La taille usuelle des composantes couleurs est de 8 bits. On utilise 4 composantes pour représenter une couleur: **Rouge**, Vert (**Green**), **Bleu** et Opacité (**Alpha**).

  Pour 1 pixel (*picture element*) on utilise 24 bits (3*8 bits) **RGB** ou 32 bits **RGBA**

- Une autre représentation ompacte utilisée sur certaines plateformes embarquées:

  Pour 1 pixel on utilise 16 bits répartis ainsi : **RGB** (**5_6_5**) ou **RGBA** (**5_5_5_1** ou **4_4_4_4**)

- La fonction **glGetIntegerv**(**GLenum** name, **GLint** *params) permet de récupérer des informations sur le pilote, comme par exemple la taille (en bits) des composantes couleurs:

```
int R, G, B, A;
glGetIntegerv(GL_RED_BITS, &R);
glGetIntegerv(GL_GREEN_BITS, &G);
glGetIntegerv(GL_BLUE_BITS, &B);
glGetIntegerv(GL_ALPHA_BITS, &A);
```

Premières fonctions (1)

- Définir la couleur avec laquelle on efface le Color Buffer

  void glClearColor(**GLclampf** R, **GLclampf** G, **GLclampf** B, **GLclampf** A)

- Les valeur **R**,**G**,**B**,A (**rouge**, **vert**, **bleu**, alpha) sont normalisées et limitées entre **0.0** et **1.0**.

- La composante alpha correspond, pour simplifier, à l'opacité.
  0.0 = totalement transparent, 1.0 = opaque.

- La couleur par défaut est noir opaque
  RGBA = (0.0, 0.0, 0.0, 1.0)

- Alternativement les composantes couleurs sont souvent codées sur **8 bits** (1 octet non signé) chacune, domaine [**0-255**]. Ce qui nous donne **24 bits** (16,777,216 couleurs) pour une couleur RGB et **32 bits** (4 octets) en ajoutant l'alpha, RGBA.

Premières fonctions (2)

- Pour effacer concrètement la zone de dessin on utilise

  void glClear(**GLbitfield** mask)

- Les bit-masks les plus utilisés sont:

  GL_COLOR_BUFFER_BIT, efface le Color Buffer
  GL_DEPTH_BUFFER_BIT, efface le Depth Buffer
  GL_STENCIL_BUFFER_BIT, efface le Stencil Buffer

- En OpenGL il est possible de n'activer le rendu (y compris le clear) que pour certaines composantes couleurs à l'aide de la fonction:

  void glColorMask(**GLboolean** red, **GLboolean** green, **GLboolean** blue, **GLboolean** alpha)

Les Color Buffers (2)

- En pratique, il y'en a toujours au moins deux. Un premier toujours visible à l'écran (***front buffer***) et un autre caché, en arrière plan (***back buffer***).

- C'est la base de la technique du ***Double Buffer***, qui permet de réduire l'effet de scintillement lorsque l'on affiche les images successives d'une animation rapide.

- OpenGL est aussi capable de faire du rendu **stéréoscopique**. Pour ce faire il utilise un tampon pour la partie gauche (*left*), et un autre à droite (*right*).
  - En OpenGL ES / WebGL ces fonctionnalités ne sont pas disponibles mais il est toujours possible de simuler ce mode de rendu l'aide des *buffers* hors-écran (*offscreen*).

Le « *Double Buffering* »

- on effectue le rendu d'une trame (***frame***) dans le *back buffer*, tandis que le *front buffer* affiche le résultat du précédent rendu.

- Une fois le rendu terminé, on demande au contexte d'échanger (*swap*) les deux *buffers*.

- Idéalement le *swapping* a lieu durant le temps mort vertical (***vertical blank***) pour éviter que le deux trames (N et N-1) ne se chevauchent à l'écran. Phénomène appelé « *tearing* ».

- On parle aussi de « synchronisation verticale » du fait que le taux d'affichage (***frame rate***) est synchronisé sur celui du moniteur.

- Note: OpenGL n'effectue pas de lui-même le *swapping* mais via le contexte de rendu (*render context)*.

Le Depth Buffer (1)

- Algorithme découvert par Ed Catmull en 1974 (LucasFilm Ltd en 1979, président de Pixar Animation Studios et Walt Disney Studios)

```
for each pixel (i,j)do
    Z-buffer [i,j] ←FAR
    Framebuffer[i,j] ←<background color>
end for
for each polygon A do
    for each pixel in A do
        Compute depth z and shade s of A at (i,j)
        if z < Z-buffer [i,j] then
            Z-buffer [i,j] ←z
            Framebuffer[i,j] ←s
        end if
    end for
end for
```

Le Depth Buffer (2)

- L'infographie contemporaine est fondée autour du tampon de profondeur. Sa simplicité se prête bien à une implémentation HW.

- Il est au cœur des GPU et essentiel pour résoudre les problèmes de profondeur et suppressions des surfaces cachées en 3D.

- L'algorithme du Z-Buffer nécessite une profondeur importante en pour éviter les problèmes de rendu lorsque les objets sont éloignés.

- Les tailles usuelles du *depth buffer* sont de **16**, **24**, voire **32** bits en entier, ainsi que **16** ou **32** bits en réel (*float, nécessite une extension en GL-ES*).

Afin de tenter de fixer le problème de « *z-fighting* » à l'aide d'un décalage OpenGL fourni la fonction **glPolygonOffset**(**GLfloat** factor, **GLfloat** units), et l'état **GL_POLYGON_OFFSET_FILL** .

OpenGL et le Depth Buffer

- Activer/désactiver l'écriture dans le depth buffer (actif par défaut)
  glDepthMask(**GLboolean** flag)

- Définir la correspondance NDC -> Ecran de la profondeur
  glDepthRangef(**GLfloat** near, **GLfloat** far) où *near* et *far* compris entre [0.0, 1.0]

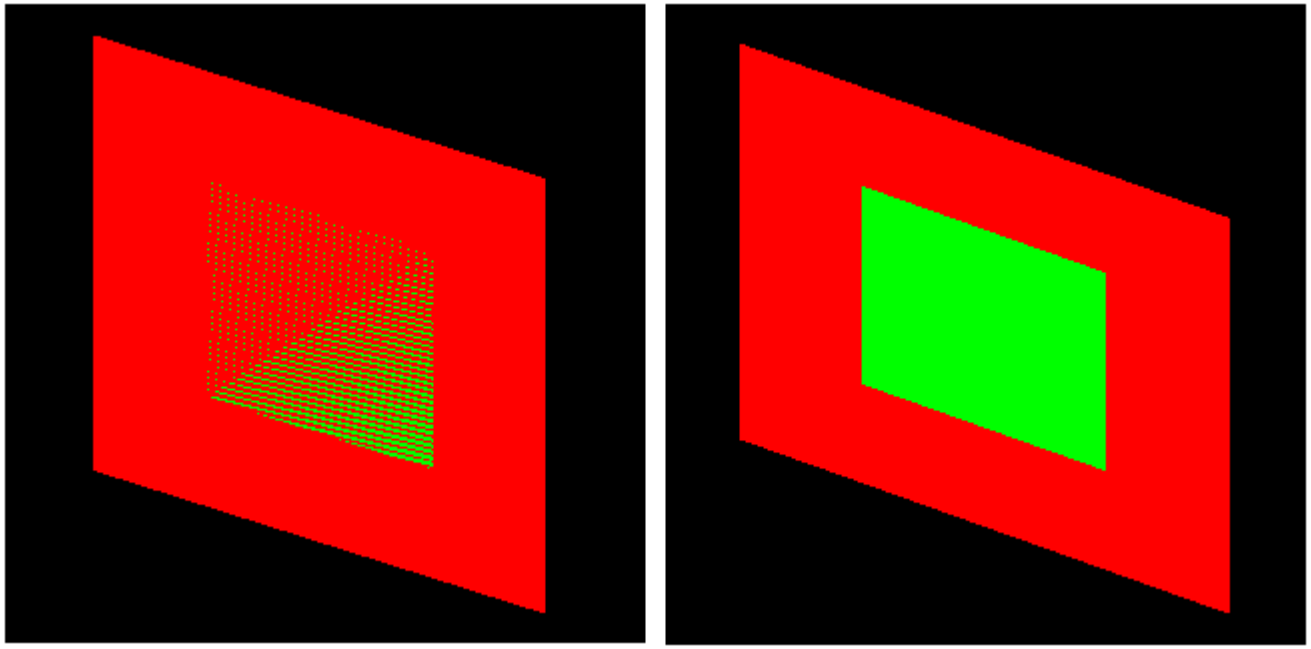
- Définir la valeur d'éffacement, par défaut 1.0 (valeur la plus grande)
  glClearDepth(**GLfloat** depth)

- Effacer le *depth buffer*
  glClear(**GL_DEPTH_BUFFER_BIT**);

- Activer/désactiver la comparaison (inactif par défaut)
  glEnable(**GL_DEPTH_TEST**) / **glDisable**(**GL_DEPTH_TEST**)

- Définir la fonction de comparaison
  glDepthFunc(**GLenum** func) avec **GL_LESS** (défaut), **GL_LEQUAL** (préférable), **GL_EQUAL**, **GL_NOTEQUAL**, **GL_GEQUAL**, **GL_GREATER**, **GL_NEVER**, **GL_ALWAYS**

Le « Z-fighting »



- Ceci arrive lorsque plusieurs fragments issus de surfaces différentes ont le même Z.
- Ces fragments entrent en compétition (*fight*, d'où le terme) pour écrire la couleur du pixel. Cela provoque aussi une forme de scintillement (*flickering* ou *flimmering*).

- Ce problème peut se fixer en appliquant un décalage (*offset*) à l'une des surfaces.

Le Stencil Buffer

- Le fonctionnement du Stencil Buffer est quelque peu complexe du fait de sa logique de calcul booléen.

- Son rôle initial est de faciliter les effets type pochoirs (de forme libre), mais il permet de faire bien plus.

- Un *stencil buffer* sur 1 bit suffit pour des effets de découpage de forme (*cookie-cut*). La majeure partie des GPU supporte un stencil buffer sur 8 bits, indépendants ou packés avec ceux du depth buffer (depth 24 bits + stencil 8 bits, le tout sur 32 bits).

- Le stencil a été détourné de façon originale au profit de plusieurs techniques avancées telles la simulation de CSG en CAO ou le rendu des ombres volumétriques (*shadow volumes*) !

OpenGL et le Stencil Buffer

- Pour activer/désactiver le rendu dans un plan
  glStencilMask(**GLuint** mask) : 0 coupe le rendu, 255 (0xff) active tous les plans.

- Pour définir la valeur d'éffacement (0 par défaut)
  glClearStencil(**GLint** s)

- Effacer le *stencil buffer*
  glClear(**GL_STENCIL_BUFFER_BIT**);

- Activer/désactiver la comparaison (inactif par défaut)
  glEnable(**GL_STENCIL_TEST**) / **glDisable**(**GL_STENCIL_TEST**)

- **glStencilFunc**() et **glStencilOp**() seront vues en TP car complexes.

Informations du pilote (1)

- La fonction **glGetIntegerv**(**GLenum name**, **GLint *params**) permet également de récupérer des informations sur tous ces buffers.

- Pour le nombre de bits du *depth buffer* avec **GL_DEPTH_BITS** et du *stencil buffer* avec **GL_STENCIL_BITS**.

- On peut tester la présence d'un *back buffer* à l'aide de **glGetBooleanv**() et du paramètre **GL_DOUBLEBUFFER**.

- ***Note**: attention à ne pas appeler ces fonctions dans la boucle principale du fait de leur lenteur !*

Informations du pilote (2)

- Connaitre la version d'OpenGL supportée par le contexte:

  const **char*** **glGetString**(**GL_VERSION**)

- Connaitre l'auteur du pilote avec **GL_VENDOR**

- Obtenir le nom du périphérique: **GL_RENDERER**

- Autre information très utile, obtenir la version du GLSL supportée par le pilote:
  GL_SHADING_LANGUAGE_VERSION.

Informations du pilote (3)

- Lister toutes les extensions supportées avec **GL_EXTENSIONS**. On peut aussi récupérer le nombre total d'extensions avec

  int numExt = **glGetIntegerv (GL_NUM_EXTENSIONS)**;

- On peut alors les lister l'une après l'autre en bouclant entre 0 et numExt-1 et appeler **glGetStringi**(**GL_EXTENSIONS, index**) à chaque itération.

note: glGetStringi est elle-même une fonction extension !

glGetStringi() est une extension !

```
#include "GL/glext.h "
#include "GL/wglext.h" // Si Windows
#include "GL/glxext.h" // Si Linux

PFNGLGETSTRINGIPROC glGetStringi; // Pointeur global
…
glGetStringi = (PFNGLGETSTRINGIPROC)wglGetProcAddress("
glGetStringi"); // Si Windows
glGetStringi = (PFNGLGETSTRINGIPROC)glXGetProcAddress("
glGetStringi"); // Si Linux
```

Autres fonctions utiles

- Pour débuguer correctement il faut prendre l'habitude d'utiliser la fonction **glGetError**(), mais seulement en version debug ou à l'initialisation, car la fonction ralenti le pilote.
- La fonction renvoi **GL_NO_ERROR** (type Glenum) si tout est ok
  - (cf. http://www.opengl.org/sdk/docs/man/xhtml/glGetError.xml).

- OpenGL stocke les commandes dans un *buffer* interne afin de réduire les temps de transfert. On peut forcer le vidage du tampon de commande par l'appel de la fonction **glFlush**() mais son utilisation est très spécifique et plus que rare.

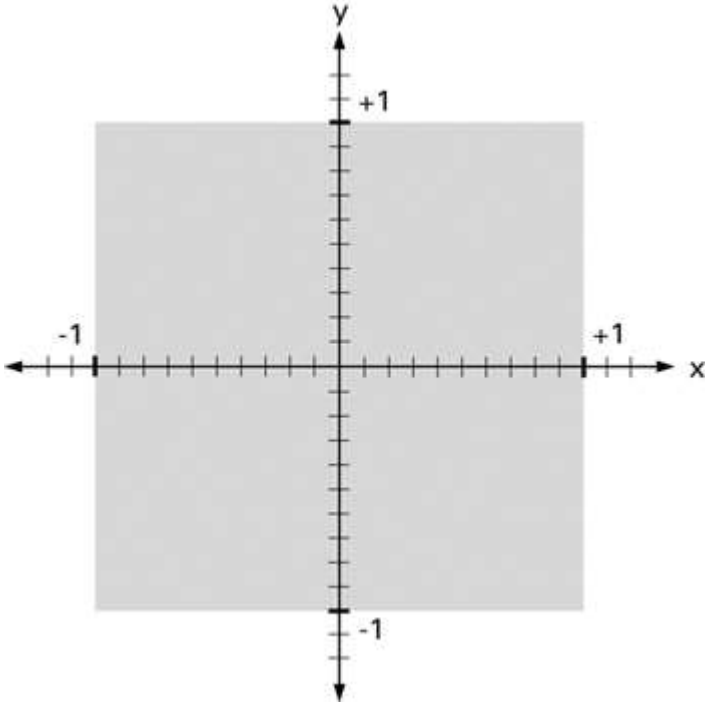
- Il existe une autre fonction similaire à **glFlush**() nommée **glFinish**() qui agit de la même façon mais bloque totalement jusqu'à ce que le rendu est totalement effectué. A éviter (presque) totalement de nos jours.

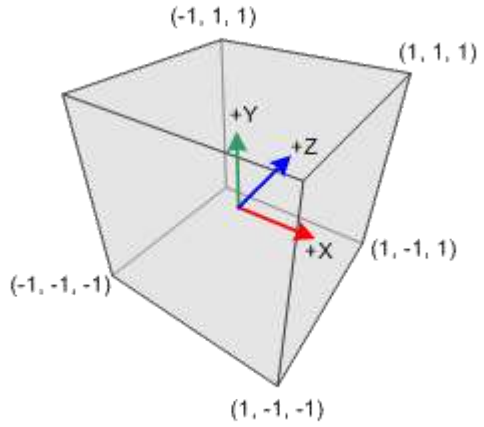
Partie 6. Le système de coordonnées

Repères par défaut

Repère par défaut

- Le repère est orthonormée et centré sur l'écran





Le *rasterizer* effectue le rendu dans un repère 3D borné entre **-1.0** et **1.0** sur les trois axes. On parle alors de *Normalized Device Coordinates* – ***NDC***.

Repères, unités et pixels

- L'origine du repère orthonormé est au centre. L'axe positif des abscisses pointe vers la droite, celui des ordonnées vers le haut.
  - L'axe positif de la profondeur pointe vers nous c'est-à-dire en dehors de l'écran car, par défaut, le repère est un repère direct (main-droite).

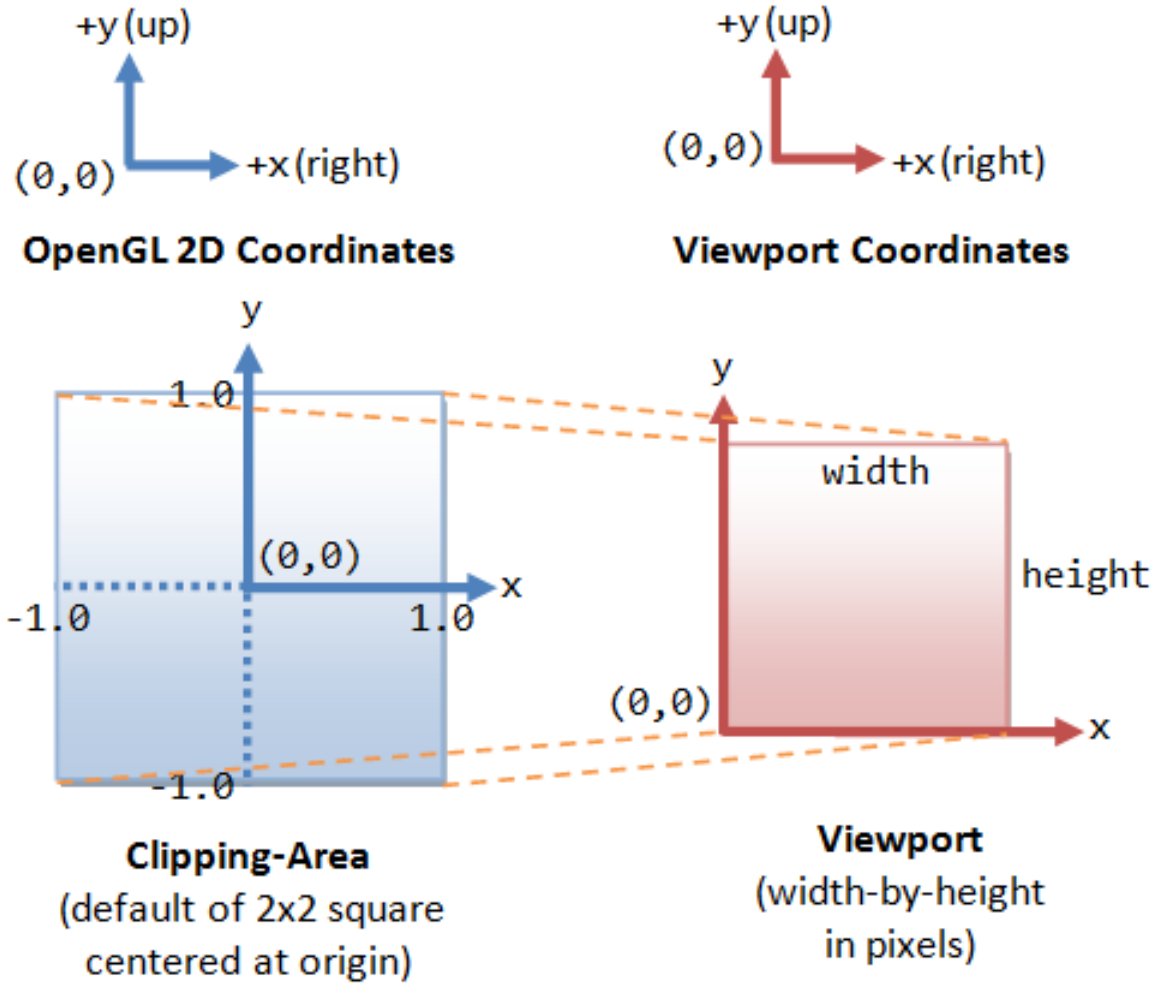
- Il est aussi possible d'effectuer le rendu dans une portion du frame buffer appelée « ***viewport*** ».

void glViewport(**GLint** x, **GLint** y, **GLsizei** w, **GLsizei** h)

- (x, y) = position du *viewport* dans le frame buffer
- (w, h) = respectivement longueur (*width*) et hauteur (*height*) du *viewport*

Attention! Ces valeurs sont exprimées en coordonnée écran (en pixels) et non en NDC!

NDC et Viewport



Exemple: Ecran Gris Clair

```
void affichage(void)
{
   /* generalement, meme taille que la fenetre */
   glViewport(0, 0, 640, 480);
   /* on souhaite effacer le fond en gris clair opaque */
   glClearColor(0.5, 0.5, 0.5, 1.0);
   /* efface le color buffer */
   glClear(GL_COLOR_BUFFER_BIT);
}
```

Autre fonction de fenêtrage

- OpenGL propose une fonction similaire en apparence à glViewport() mais dont le but est quelque peu différent.

- On appelle cette technique le « découpage » (*scissoring*). Elle permet de définir un rectangle en dehors duquel il n'y a pas de rendu.

void **glScissor**(**GLint** x, **GLint** y, **GLsizei** w, **GLsizei** h)

Le *scissoring* est activé/désactivé par glEnable() / glDisable() et l'état **GL_SCISSOR_TEST**

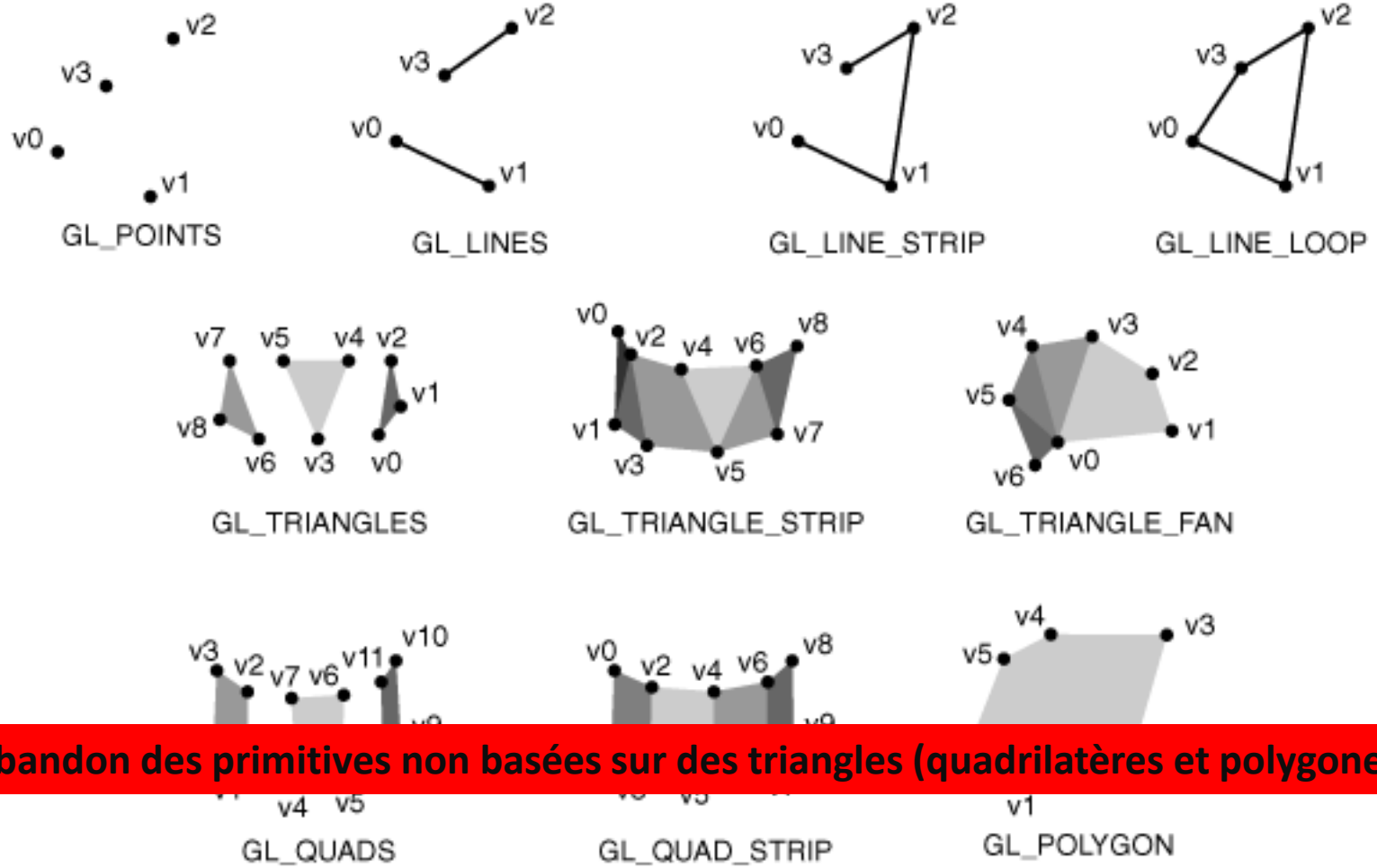
Partie 7. Dessin de primitives

Définition

et

affichage

Les primitives OpenGL 1.x



Abandon des primitives non basées sur des triangles (quadrilatères et polygones)

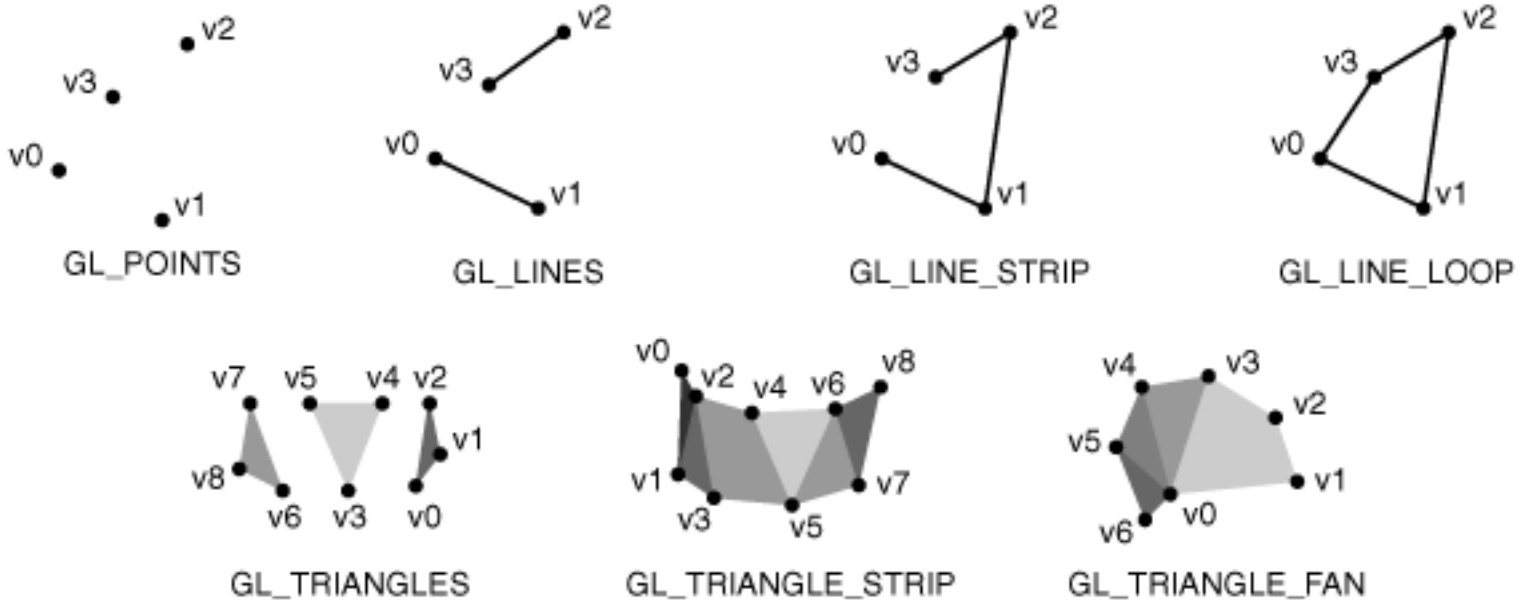
Les types de primitives

- OpenGL ES supporte les primitive suivantes:

- **GL_POINTS,**
- **GL_LINES**, **GL_LINE_STRIP**, **GL_LINE_LOOP**,
- **GL_TRIANGLES, GL_TRIANGLE_STRIP**,
- **GL_TRIANGLE_FAN**

- Mais plus du tout celles-ci
- ~~GL_QUADS, GL_QUAD_STRIP,~~
- ~~GL_POLYGON~~

Les primitives OpenGL ES



remarquez que les primitives sont définies en suivant les vertices dans le sens inverse des aiguilles d'une montre (counter clockwise)

Ordre des sommets (1)

- Lorsque l'on représente un objet en 3D, la plupart du temps une partie de cet objet - les faces arrières (*back faces*) - est masquée par une autre - les faces avants (*front faces*).

- Pour éviter d'avoir à calculer et dessiner ces faces cachées, OpenGL offre la possibilité d'activer la suppression de faces (*face culling*). Par défaut l'état **GL_CULL_FACE** est inactif.

- Il faut suivre une règle simple: toujours ordonner les faces dans le même sens, que ce soit horaire (*clockwise*), ou, de préférence, anti-horaire (*counter- clockwise*). En anglais on parle de « *winding* » pour définir cet ordonnancement.

Ordre des sommets (2)

- OpenGL permet de spécifier quelles faces sont considérées comme visible (*front face*) avec la fonction **glFrontFace**(). Deux modes possibles

  GL_CW (*clockwise*)

  GL_CCW (*counter clockwise*, par défaut).

- De plus il est possible de modifier le ***culling*** en indiquant le type de faces qui seront éliminées avec **glCullFace**(). Trois modes possibles parmi :

  GL_FRONT

  GL_BACK (par défaut)

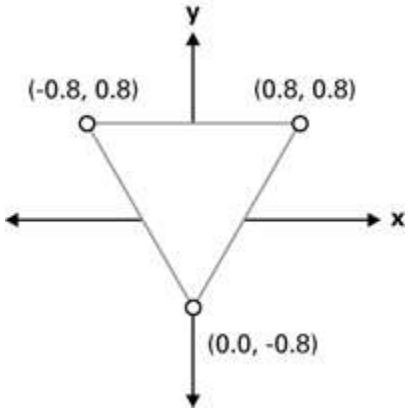
  et **GL_FRONT_AND_BACK** (retire tout!).

- **Par défaut, OpenGL élimine donc les faces arrières. Et, toujours par défaut, une face est considérée « arrière » lorsque les sommets sont définis dans le sens des aiguilles d'une montre.**

Un triangle à l'ancienne

- Le mode immédiat est simple mais **a été retiré d'OpenGL ES**.

```
glBegin(GL_TRIANGLES);
   glVertex2f(-0.8f, 0.8);
   glVertex2f(0.0f, -0.8f);
   glVertex2f(0.8f, 0.8f);
glEnd();
```



- Afficher le même triangle demande maintenant un peu plus d'efforts de la part du programmeur.

La bonne façon de dessiner

void glDrawArrays(**GLenum** mode, **GLint** first, **GLsizei** count)

- **mode** = un type de primitive parmi celles disponibles.
- **first** = l'indice dans le tableau du premier element à dessiner.
- **count** = le nombre total de sommets des primitives à dessiner.

Exemple 2.3.a, dessin de 6 points:

```
glDrawArrays(GL_POINTS, 0, 6);
```

Exemple 2.3.b, dessin de 2 triangles (2*3 sommets = 6 sommets) :

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

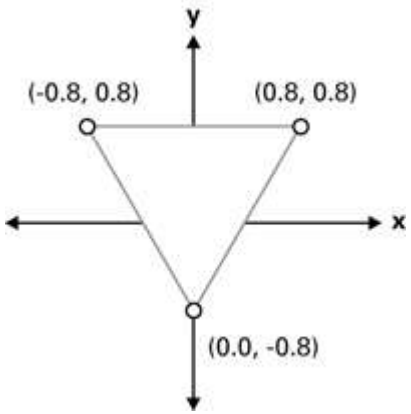
Exemple 2.3.c, dessin du deuxieme des deux triangles seulement :

```
glDrawArrays(GL_TRIANGLES, 3, 3);
```

Décrire les primitives

- Chacun des sommets peut être décrit par un ou plusieurs attributs.
- On peut définir ces attributs de façon séparée (***packed***) c'est-à-dire un tableau par attribut, ou bien entrelacée dans une structure (***interleaved***).
- OpenGL accepte toute adresse mémoire, qu'elle soit statique (dans l'exécutable) ou dynamique (allouée sur le tas, «*heap*» ou «*free-store* »).
- Un triangle peut se décrire ainsi en en C:

```
/* 3 sommets, 2 coordonnées => 6 float-s */
static const float g_Triangle[] = {
        -0.8f, 0.8f,
        0.0f, -0.8f,
        0.8f, 0.8f
};
```



Reste à transférer ces vertices

Les « Arrays »

- Les données sont stockées de façon contigüe en mémoire centrale, côté client (*client-side*), dans un tableau (*array*).
- Il suffit de passer les adresses des tableaux à OpenGL pour dessiner.

- Le **pipeline fixe** a pré-câblé la gestion des différents attributs de sommets (*position, couleur, normale…*)
- …mais cela manque de souplesse et le pipeline programmable ne fonctionne plus ainsi.

- Gros bémol: cela reste proche du mode immédiat. Le driver doit tout ré-envoyer à chaque fois que l'on doit dessiner (*draw calls*).

Exemple oldschool: Vertex Arrays

Rappel: bien que dépréciés les Vertex Arrays, fonctionnent en OpenGL ES 1.1 mais pas en OpenGL ES 2.0 et OpenGL ES 3.x. Ils sont présentés à titre illustratif.

- En premier lieu il faut activer les registres des attributs utilisés, ici, la position

```
glEnableClientState(GL_VERTEX_ARRAY);
```

- Ensuite on précise la source des attributs

```
glVertexPointer(2, GL_FLOAT, 0, g_Triangle);
```

- On peut alors dessiner le triangle

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

- Puis éventuellement désactiver les registres

```
glDisableClientState(GL_VERTEX_ARRAY);
```

Moderne: Vertex Attribute Arrays

- Syntaxe quasiment similaire à *glVertexPointer()*

void glVertexAttribPointer(**GLuint** index, **GLint** size, **GLenum** type,
GLboolean normalized, **GLsizei** stride, **const void** ***ptr)

- **index** = numéro du registre à utiliser (0 pour le moment)
- **size** = le nombre de composant par attribut: 1,2,3 ou 4
- **type** = cf. « tableau synthétique des types »
- **normalized** = lorsque des entiers doivent être réduit à [0.0;1.0]
- **stride** = indique le décalage en octet entre deux attributs (utile si les attributs sont entrelacés).
- **ptr** = adresse du tableau des attributs en mémoire centrale (client). Si ptr vaut **0** ou **NULL**, la fonction utilisera alors une zone mémoire définie par OpenGL (cf. Vertex Buffer Objects)

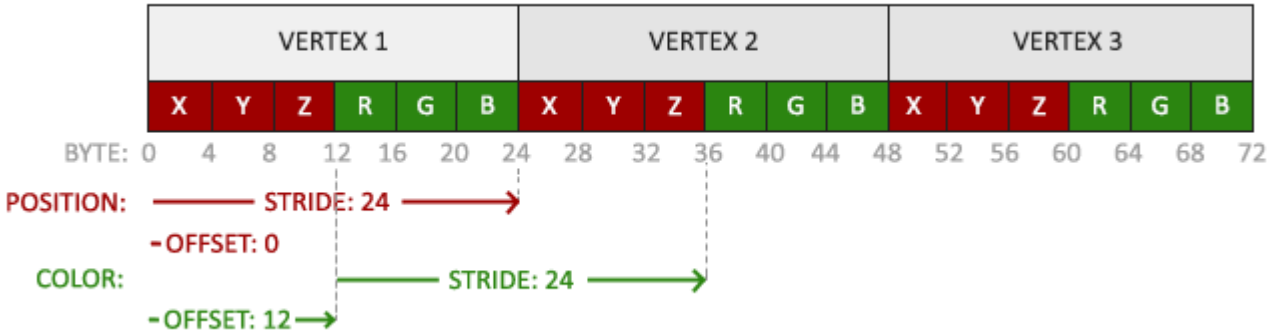
Entrelacement des données

- Les deux derniers paramètres de la fonction **glVertexAttribPointer**() sont particulièrement important car ils nous permettent de spécifier à OpenGL comment sont ordonnancées les données en mémoire.

- Les attributs peuvent être stockés dans des tableaux séparés (un tableau pour les positions, un tableau pour les couleurs etc…).

- Dans ce cas il suffit de spécifier un écart (stride) de 0 et fournir l'adresse de chaque tableau d'attributs aux différents appels de glVertexAttribPointer [1 par attribut].

Entrelacement des données



- Cependant, si les attributs sont entrelacés il est nécessaire de procéder comme suit:
  - L'avant dernier paramètre recevra la taille totale (en octets) d'un vertex. Dans le schéma on a 3 floats pour position et 3 floats pour couleur RGB ce qui fait donc 6 * sizeof(float) comme valeur de *stride*.
  - Le dernier paramètre contient l'adresse absolue du premier attribut en mémoire. Dans l'exemple on aurait &vertex[0] (ou simplement vertex) pour les positions et &vertex[3] (ou (vertex+3)) pour les couleurs.
  - L'adresse relative (offset) des positions par rapport au début du tableau est de 0 octets tandis qu'elle est de 3*sizeof(float) = 12 octets pour les couleurs.

Exemple: Vertex Attribute Arrays

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
                      2*sizeof(float), g_Triangle);
glEnableVertexAttribArray(0);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableVertexAttribArray(0);
```

- **glEnableVertexAttribArray()** indique à **glDrawArrays()** que les données devront être relues à chaque sommet car elles vont changer. Autrement il s'agirait de valeurs communes que l'on spécifie avec glVertexAttrib().

- Nous verrons plus tard comment rendre le passage des attributs plus générique encore.

Et vous obtenez…rien ?

- A ce stade nous n'avons fait qu'envoyer les données !

- Certes, nous avons exécuté une commande de rendu et spécifié l'agencement des données.
- Nous avons également indiqué les valeurs des attributs de sommets ainsi que le type de primitive à dessiner.
- Mais pas
  - comment traiter chaque sommet (vertex processing)
  - Quelle couleur générer pour chaque pixel (fragment processing)

Dans la suite du cours nous verrons les shaders en détail

- La syntaxe du GLSL.
- Les Vertex Shaders.
- Les Fragment Shaders.

Partie 8. Avant cela…

En WebGL on ne peut dessiner que des primitives qui sont déjà stockées en VRAM sur le GPU. C’est très fortement recommandé en OpenGL-ES mais non obligatoire. Voyons comment cela fonctionne.

Objects ?

- Les « Objets », au sens OpenGL, sont en quelque sorte des structures servant à gérer l'état d'un ensemble de fonctionnalités.

- Il en existe plusieurs variantes et celles destinées au stockage et au transfert de données utilisent le même mécanisme, et partagent la même API.

- On s'intéressera en premier lieu à une variante de « *buffers* », les Vertex Buffer Objects ou **VBO**s.

VBO, ou comment nourrir la bête

- En l'occurrence, les Vertex Buffer Objects servent au stockage et au transfert des données géométriques (vertices). Le Vertex Buffer étant un tableau de vertex.

- Leur intérêt principal réside dans le fait que les données peuvent être stockées par le pilote là où les performances seront le plus optimales (le plus proche possible du GPU).

- Il existe deux types de données (ou target) utilisable avec un VBO, les vertices et les indices.

  GL_ARRAY_BUFFER, dédié aux vertices
  GL_ELEMENT_ARRAY_BUFFER, dédié aux indices (ou éléments)

Gérer un objet type « buffer »

- 1. Un objet de type « buffer » se créé et se détruit à l'aide de:

  void **glGenBuffers**(**GLsizei** nb, **GLuint** *buffers)
  void **glDeleteBuffers**(**GLsizei** nb, **GLuint** *buffers)

- Où 'nb' est le nombre de « buffer objects » à créer, et 'buffers' un pointeur vers les variables où stocker le(s) identifiant(s).

- 2. Pour être utilisé, un buffer object doit être défini actif, ou 'bindé'.

  void **glBindBuffer**(**GLenum** target, **GLuint** id)

- Un seul buffer peut-être 'bindé' à une target. L'id spécial zéro (0) retire le dernier buffer défini sur la target (*unbind*).

Relation avec l'API de dessin

- Dans la présentation sur les bases de l'API nous avons vu (en 3.8) que la fonction **glVertexAttribPointer**() permet de préciser où sont stockées les données à transférer au GPU.

- Lorsque un VBO est actif (***bind***é), OpenGL interprète différemment le dernier paramètre indiquant l'emplacement (l'adresse) des données.
- En effet, OpenGL considère qu'il s'agit alors d'un décalage en octets (offset) indiquant le début des données d'un attribut particulier.

Petite particularité: comme le type de donnée du paramètre est (void*) il est souvent préférable de réinterprété le type pour que le compilateur ne soit pas trop énervé. Par exemple avec la macro suivante:

#define **BUFFER_OFFSET**(offset) ((void*)(offset))

Alternativement on peut utiliser la macro **offsetof**() définie dans **cstddef.h** si elle supportée par le compilateur.

Allocation et transfert (1)

- Il faut en premier lieu, après avoir bindé un Buffer Object à une target, allouer de la mémoire:

  void **glBufferData**(**GLenum** target, **GLsizei** size, **const void** *data, **GLenum** usage)

- Lorsque 'data' à une valeur non nulle OpenGL effectue une copie des données pointées par 'data' vers la zone créée par la fonction.

- Le paramètre le plus intéressant est 'usage':

  GL_STATIC_DRAW, indique que le contenu ne sera pas modifié. Garantie un stockage proche du GPU

  GL_STREAM_DRAW, indique que le contenu ne sera modifié que peu de fois (une fois par image par exemple)

  GL_DYNAMIC_DRAW, le contenu peut varier très souvent

  Note: ceci n'est qu'à titre indicatif, le pilote est libre de ne pas tenir compte de votre choix si il ne gère pas l'usage ou si il détecte une utilisation différente.

Allocation et transfert (2)

- Si **glBufferData**() a été appelé avec une valeur **nulle** pour ‘data’, OpenGL n’aura fait qu’allouer de la mémoire.

- Pour **transférer** (mettre à jour) des données il faut réappeler **glBufferData**() en indiquant l’adresse des données au paramètre ‘data’.

- Ceci aura pour conséquence l’écrasement de toute les données précédemment transférées.

- Dans le cas où l’usage ou la taille seraient différents OpenGL procédera à une **réallocation**.

Allocation et transfert (3)

- Si l'on souhaite modifier une partie, et non l'ensemble, des données déjà dans le VBO:

  void **glBufferSubData**(**GLenum** target, **GLint** offset, **GLsizei** size, **const void** *data)

- La principale différence réside dans l'utilisation d'une valeur de décalage (offset) en octets indiquant le début de la zone mémoire à modifier. Cette fonction ne modifie pas l'usage.

  Attention, **glBufferSubData**() n'alloue pas de mémoire, utilisez **glBufferData**() au préalable !

Récapitulatif

- Pour utiliser un VBO il faut

- 1. le créer avec **glGenBuffers**()
- 2. le définir comme actif avec **glBindBuffer**()
- 3. Allouer de la mémoire **glBufferData**()
- 4. Transférer des données à l'aide de **glBufferData**() ou **glBufferSubData**() puis indiquer à l'adresse relative (offset) des attributs dans le VBO avec **glVertexAttribPointer**()

- 5. Re-bind du « Buffer Object » si nécessaire
- 6. Indiquer les attributs qui vont servir avec **glEnableVertexAttribArray**()
- 7. Dessiner avec **glDrawArrays**() ou **glDrawElements**()
- 8. Désactiver ce dont on ne se sert plus, **glBindBuffer**(target, **0**) et/ou **glDisableVertexAttribArray**()

- 9. Ne pas oublier à la fin de le détruire avec **glDeleteBuffers**()

Alors, ces « elements » ?

- Supposons que l'on souhaite dessiner deux triangles formant un carré, on aura donc besoin de 4 sommets.

- Numérotons les sommets, en partant de zéro comme base.

- Avec **glDrawArrays**() il faudra afficher par exemple les triangles <0-1-2> et <3-2-1>. On notera que les sommets '1' et '2' doivent être envoyé deux fois.

- Supposons que les vertices soient en 3D, il nous faut transférer 2 fois 3 sommets de 12 octets (3*sizeof(**GLfloat**)) ce qui fait **72 octets**.

Indices

- Si l'on envoyait en sus les indices [0-1-2] et [0-2-3] on n'aurait qu'à transférer nos 4 sommets en une seule fois. Ces 6 valeurs d'index pourrait être stockées sur 1, 2 ou 4 octets chacun.

- Ceci nous donnerait pour les sommets un total de 4*3*sizeof(**GLfloat**) soit **48 octets**.
- Et pour les indices en 16-bit on aurait 6*sizeof(**GLushort**) soit **12 octets**.

- Total: **60 octets**, un gain d'environ **17%** en mémoire !

Note: Le gain aurait été encore plus probant si les vertices contenaient plus d'informations. Par exemple avec un vertex de 32 octets, on aurait 2*3*32 = 192 octets sans indices, et 4*32 + 12 = 140 octets, soit **27%** !

Oui mais j’ai 8 go de GDDR5…

- Que vous développiez pour un Raspberry Pi ou une Playstation4, ce qui compte surtout est le fait d’éviter de faire faire au GPU des calculs complexes et redondants.

- Précisément, ici, avec glDrawArrays() utilisé pour dessiner un maillage 3D, on doit nécessairement transformer le même vertex à plusieurs reprises.

- C’est là tout l’intérêt de ces ‘indices de sommet’ qu’OpenGL appelle des « elements ».

Dessin de primitive avec indices

void **glDrawElements**(**GLenum** mode, **GLsizei** count, **GLenum** type, **void** ***indices)

- 'mode' et 'count' sont ici similaires à glDrawArrays()
- 'type' désigne ici le type de donnée, à savoir
  GL_UNSIGNED_BYTE (8 bits)
  GL_UNSIGNED_SHORT (16 bits)
  GL_UNSIGNED_INT (32 bits), sauf OpenGL ES 2.0 (sauf si extension)

- 'indices' est tout simplement un pointeur sur un tableau d'éléments de la taille déclarée par 'type'.
- Si 'indices' vaut **NULL**, OpenGL utilisera le VBO bindé en tant que **GL_ELEMENT_ARRAY_BUFFER**

Récapitulatif

- Dessin de primitives simples (non indexées):
  - **glBindBuffer**() de(s) VBO(s) de type **GL_ARRAY_BUFFER**
  - 1 ou plusieurs **glVertexAttribPointer**(…, **offset**)
  - **glDrawArrays**()
- Dessin de primitives indexées:
  - **glBindBuffer**() de(s) VBO(s) de type **GL_ARRAY_BUFFER**
  - 1 ou plusieurs **glVertexAttribPointer**(…, **offset**)
  - 1 VBO de type **GL_ELEMENT_ARRAY_BUFFER**
  - **glDrawElements**(…, **NULL**)

Harder, Better, Stronger parfois Faster

- La plupart des implémentations permettent aussi de récupérer l'adresse mémoire d'un « buffer » (mapping) et de le manipuler directement.

  void * **glMapBuffer**(**GLenum** target, **GLenum** access)
  GLboolean **glUnmapBuffer**(**GLenum** target)

- 'access' peut prendre les valeurs suivantes:

  GL_READ_ONLY, lecture seule (la lecture depuis la mémoire peut être très lente)
  GL_WRITE_ONLY, écriture seule (le plus rapide, forte chance d'accès direct)
  GL_READ_WRITE, lecture et écriture

- Le principal intérêt du *mapping* d'un VBO dans l'espace d'adressage du client est d'éviter d'avoir à gérer une copie locale (client) des données et de tout écrire directement, potentiellement, côté AGP ou GPU (server).
  - Le principal risque niveau performance est de modifier une zone mémoire alors qu'elle est utilisée par le GPU pour dessiner!

Stockage des données (1)

De manière séparée:

- On stocke chaque classe d'attribut dans un tableau à part c'est-à-dire 1 VBO par attribut

  - Position: *x0, y0, z0, x1, y1, z1, x2, y2, z2…*
  - Tex. Coords: *s0, t0, s1, t1, s2, t2…*
  - Couleur: *rgba0, rgba1, rgba2 …*

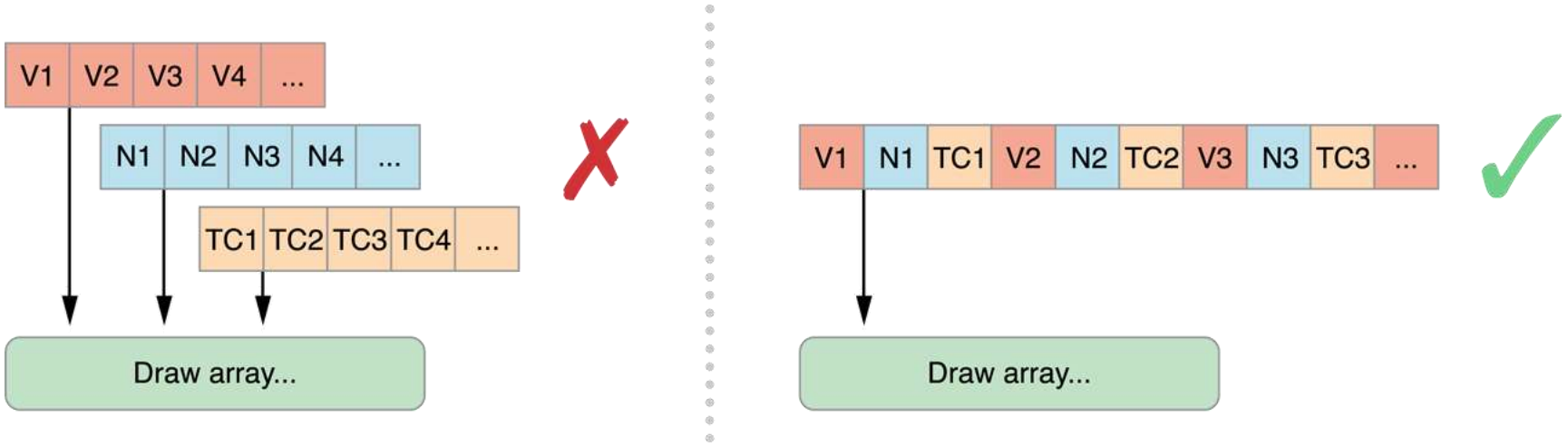
De manière compacte:

- On stocke l'ensemble des attributs d'un vertex dans le même tableau, donc 1 VBO pour l'ensemble des données:

  x0, y0, z0, s0, t0, rgba0, x1, y1, z1, s1, t1, rgba1…

Stockage des données (2)

- OpenGL ES (iOS) : Best Practices for Working with Vertex Data
- Cela reste vrai sur tout les GPU



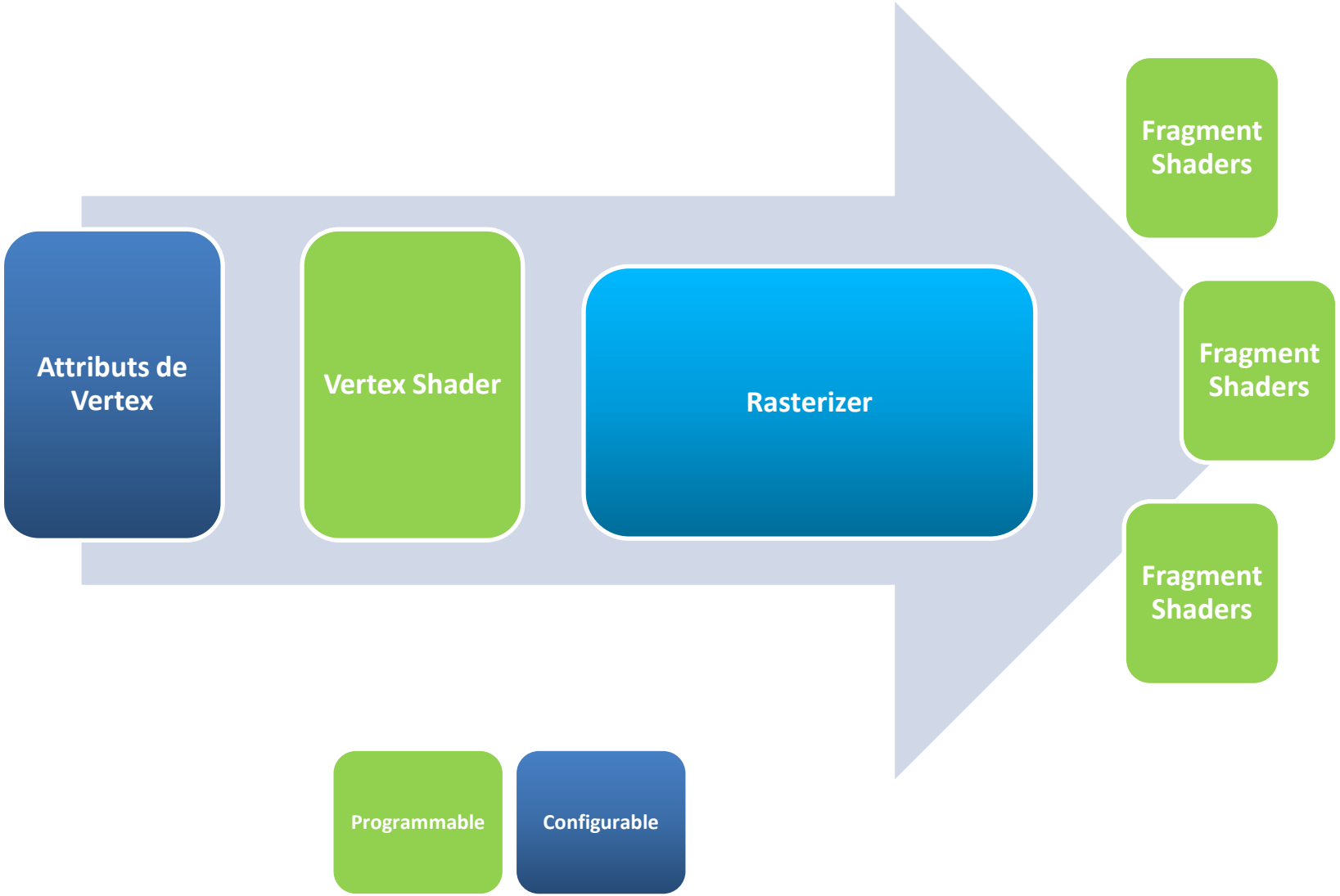
- Cependant, si un des attributs est modifiés (par exemple les vertices) il peut être parfois plus judicieux de stocker ces attributs (ici, les vertices) séparément des autres attributs (normales, coordonnées de textures, ….)

A retenir

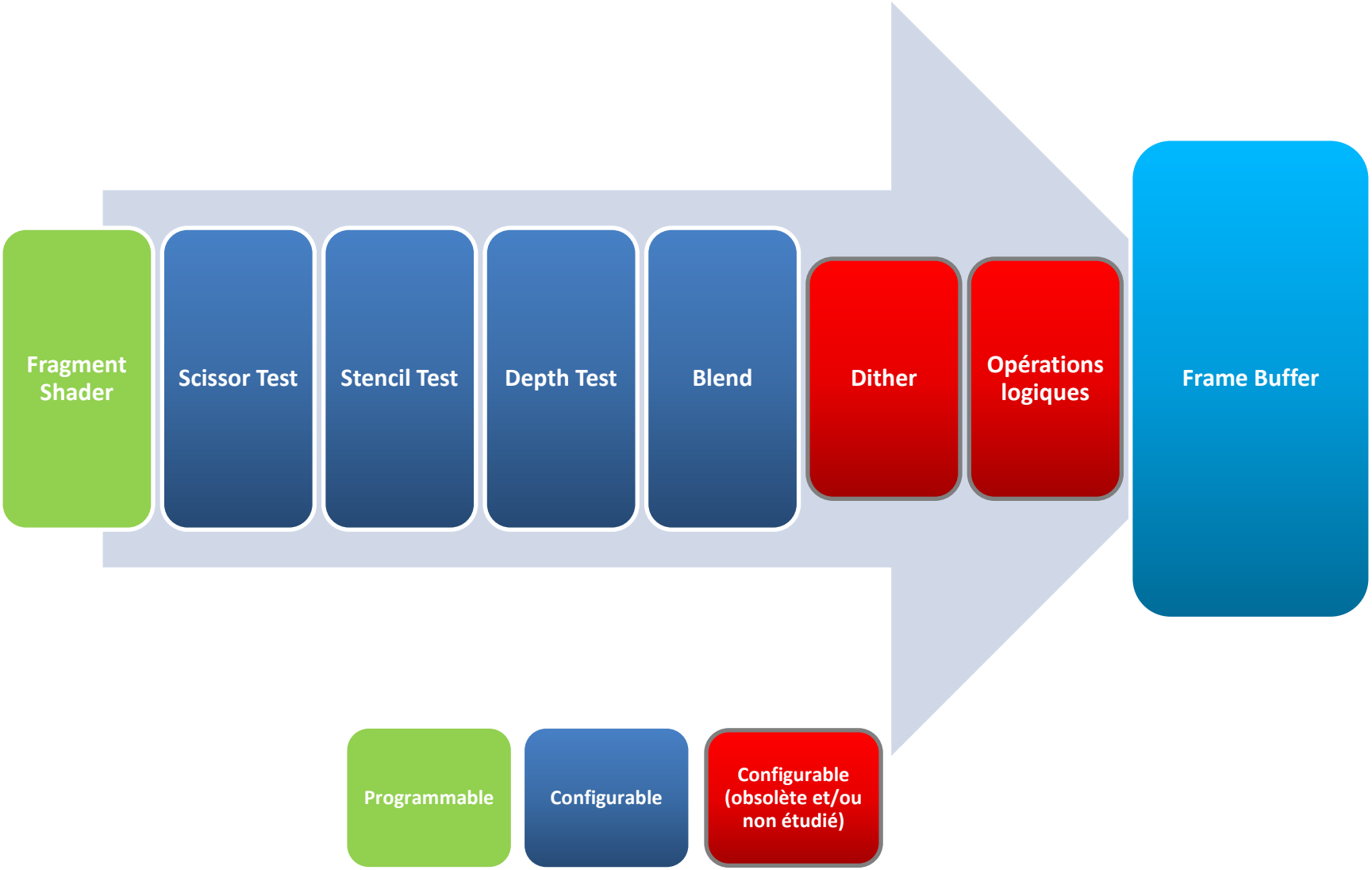
A quelques rares exceptions près, les GPUs préfèrent:

- Le stockage des données dans des VBOs.
- Des données compactes dans un seul VBO, pour peu que chaque vertex ne soit pas trop large ou que les attributs soient updatés à une fréquence différente.
- Des primitives définies à l'aide d'indices dans un VBO que l'on appelle **Index Buffer** ou **Element Array Buffer**.

Résumé: trajet des vertices (sommets)



Résumé: trajet des fragments



Références

- https://www.khronos.org/opengles/
- https://developer.android.com/about/dashboards/index.html
- https://developer.apple.com/library/prerelease/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008793

- https://www.khronos.org/webgl/
- http://www.webglacademy.com/
- http://learningwebgl.com/blog/?page_id=1217/
- http://webglfundamentals.org/

- http://malideveloper.arm.com/
- https://imgtec.com/blog/tag/opengl-es/