GMIN 317 – Moteur de Jeux

GPU – Techniques avancées

Université Montpellier 2

Rémi Ronfard

remi.ronfard@inria.fr

https://team.inria.fr/imagine/remi-ronfard/

Aujourd'hui, nous allons nous intéresser à un un matériel spécifique pour vos moteur de jeux: le GPU. Ce matériel ne sert pas seulement à rasteriser votre scène il sert également à …

Mais avant tout, intéressons nous aux textures !

- Une texture est un rectangle de données, composé de:
  - Couleur, luminance, alpha, normales
  - Chaque élément d'une texture est appelé : texel
- Dans OpenGL, la longueur et la largeur d'une texture devait être une puissance de 2 ..
  - Mais ça c'était avant ..

- Comment utiliser une texture:
  - Charger la texture
  - Indiquer comment *mapper* la texture sur l'objet
  - Activer la texture sur l'objet:
    - Ex: glEnable(GL_TEXTURE_2D)
  - Dessiner la scène

- Trois types de textures :
  - glTexImage1D, **glTexImage2D**, glTexImage3D

- Pour une texture 2D:

```
void glTexImage2D(
    GLenum target,          //target texture
    GLint level,            //level-of-detail number
    GLint internalFormat,   //number of color components in the texture
    GLsizei width,          //width of the texture image
    GLsizei height,         //height of the texture image, or the number of layers in a texture array
    GLint border,           //This value must be 0.
    GLenum format,          //format of the pixel data.
    GLenum type,            //data type of the pixel data.
    const GLvoid * data);   //pointer to the image data in memory.
```

- Attention, OpenGL ne fournit pas de chargeur d'images.

- Pour appliquer la texture à un objet:

void glTexEnv{if}[v](

GLenum *target*, //texture environment.

GLenum *pname*, //symbolic name of a single-valued texture environment parameter.

GLfloat *param* //Specifies a single symbolic constant,

)

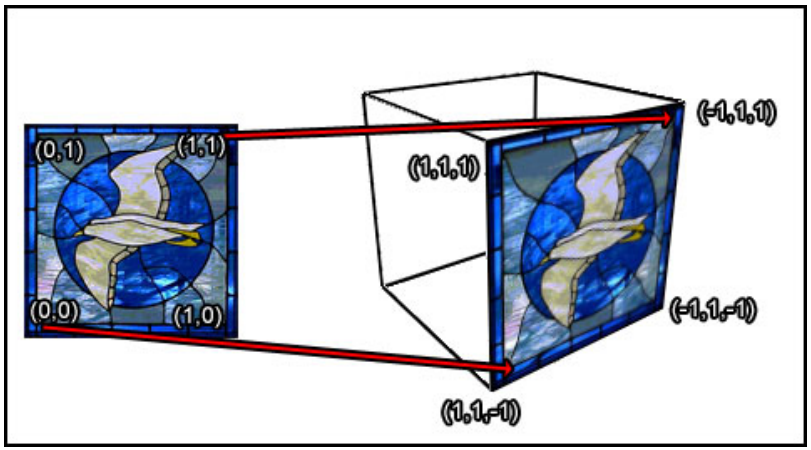
Est utilisé pour spécifier comment les texels seront combiné à l'objet

- Une fois la texture chargé, il est nécessaire de l'appliquer à notre objet pour cela:

void glTexCoor{1234}{sidf}[v](TYPEcoords)

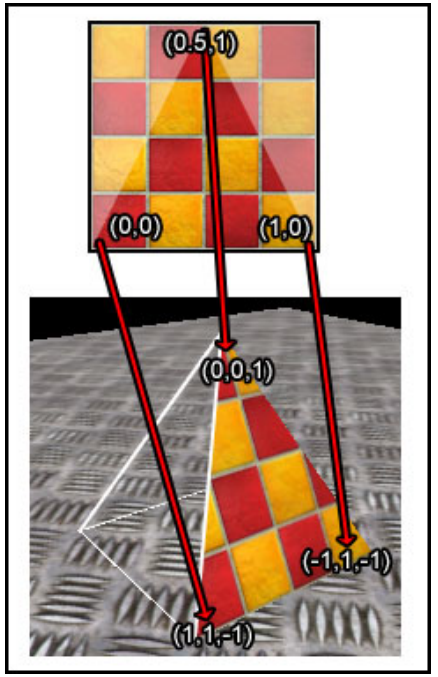
A appliquer directement sur les sommets, par exemple :

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f);  // Lower-left corner of texture
  glVertex3f(.....)
  glTexCoord2f(0.0f, 1.0f);  // Upper-left corner of texture
  glVertex3f(.....)
  glTexCoord2f(1.0f, 0.0f);  // Lower-right corner of
texture
  glVertex3f(.....)
  glTexCoord2f(1.0f, 1.0f);  // Upper-right corner of
texture
  glVertex3f(.....)
glEnd();
```



- Pour un triangle

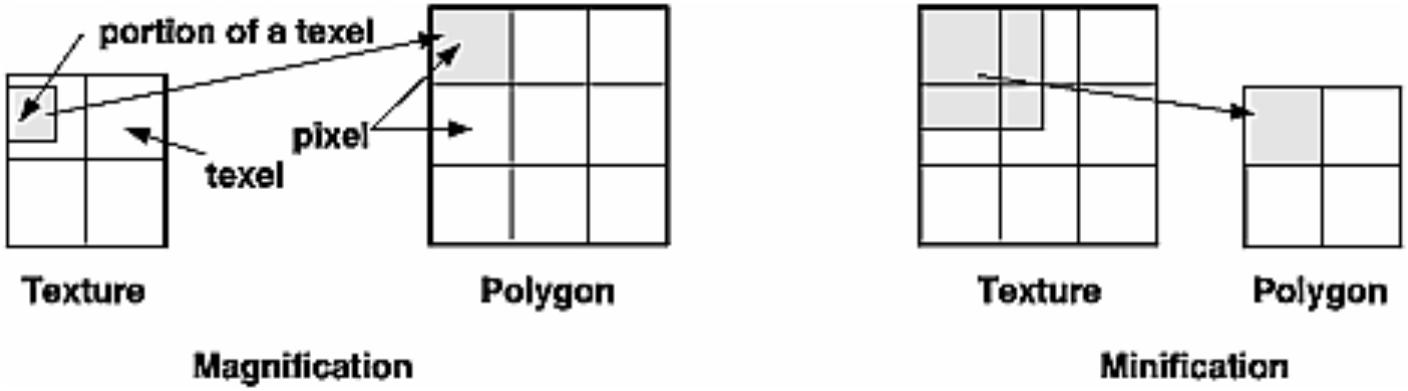
```
glBindTexture(GL_TEXTURE_2D, texture2);
glBegin(GL_TRIANGLES);
   glTexCoord2d(0,0);     glVertex3d(1,1,-1);
   glTexCoord2d(1,0);     glVertex3d(-1,1,-1);
   glTexCoord2d(0.5,1);   glVertex3d(0,0,1);
glEnd();
```



- Comme présenté dans cet exemple, on peut utiliser seulement une partie de la texture.

- Pour bien gérer vos textures, il faut:
  - Les considérer comme des objets
  - Leur donner un nom
  - Assigner une *image* sur des données *textures*
  - Paramétrer chaque texture
  - Et lors du rendu simplement recoller la texture sur l'objet alloué
- Différentes fonction pour cela:
  - glGenTextures(..) < retourne le nom de textures
  - glIsTexture(..) < la texture est elle assigné ?
  - glBindTexture(..)
  - glDeleteTextures(..) < supprimer des textures

- Le problème de cette méthode:
  - Un texel correspond rarement à un pixel affiché à l'écran…
- Deux problèmes apparaissent:
  - Magnification
  - Minification



  - Pour filtrer ce problème:

    glTexParameter{if}(..)

- Multi-résolution des textures (mipmap)
  - Il s'agit d'une série de textures pré-filtrés dont la résolution diminue

Utilisation:

- Appeler glTexImage2D()
  Avec les différents niveaux, w,h et images
- OpenGl choisit automatiquement les niveaux de texture en fonction de la taille des pixels

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0,
        GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage32);
  glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 16, 16, 0,
        GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage16);
  glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 8, 8, 0,
        GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage8);
  glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 4, 4, 0,
        GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage4);
…
```

- Les textures sont définies entre 0.0 et 1.0
- Mais on peut modifier ces valeurs par:
  - glTexParameter{if}(..)
- Parfois, il est nécessaire de répéter la texture:

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f);  // Lower-left corner of texture
  glVertex3f(.....)
  glTexCoord2f(0.0f, 10.0f);  // Upper-left corner of texture
  glVertex3f(.....)
  glTexCoord2f(10.0f, 0.0f);  // Lower-right corner of texture
  glVertex3f(.....)
  glTexCoord2f(10.0f, 10.0f);  // Upper-right corner of texture
  glVertex3f(.....)
glEnd();
```

- Il est également possible de combiner des couleurs avec des textures.

- Pour plus d'informations sur les textures, vous référer à la doc OpenGL.

Par exemple :

- Génération automatique des coordonnées:
  - glTexGen{ifd}[v](..)
- …

- Un contexte OpenGL gère plusieurs types de zones mémoire pour le stockage et la génération de données image.

- Les types de buffers image sont:
  - Color Buffer
  - Depth Buffer
  - Stencil Buffer
  - Accumulation Buffer

- Combinaison : framebuffers

En général, le rendu s'effectue en double voire triple buffering:

- Double-buffering: une image est affichée pendant que l'autre est générée, puis échange des deux images au rafraîchissement d'affichage suivant (swapping)
- Triple-buffering: une image est affichée, et deux sont disponibles alternativement pour le rendu.

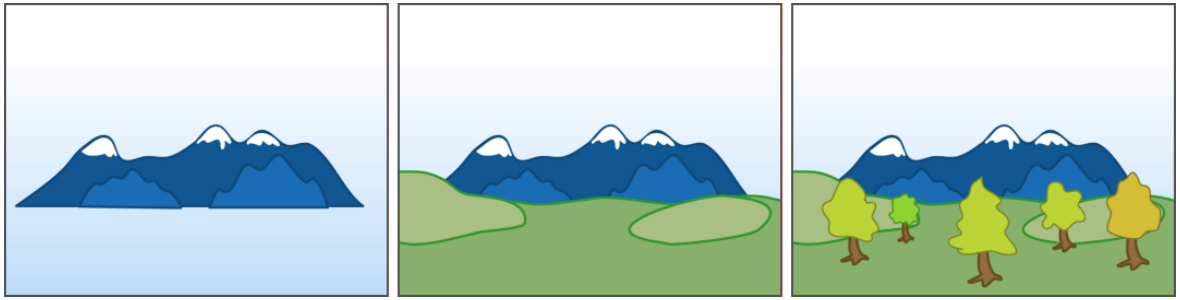
Pourquoi le double ou le triple buffering ?

- Lorsque la vue adresse directement le frame buffer, le changement d'image revient à un changement de pointeur, sans copie de l'image
- En triple-buffering, on augmente le parallélisme en évitant l'attente de la synchronisation verticale de l'écran : le rendu s'effectue sans blocage, mais un rendu est toujours disponible pour un affichage dès que possible. En contrepartie, nécessite plus de mémoire pour le stockage des multiples buffers.

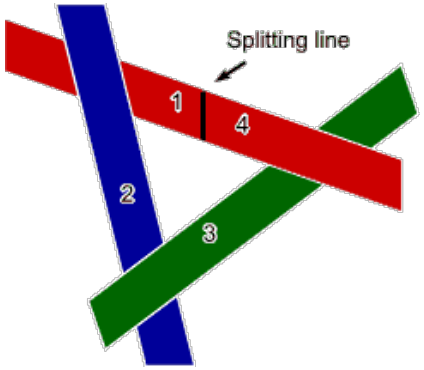
- Color Buffers
  - Il s'agit de celui qu'on utilise pour dessiner
  - Il contient à la fois:
    - L'index de couleur
    - La couleur au format RGB
    - Les métas données éventuelle (alpha, ..)
  - En mode stéréoscopie, ce buffer contient les informations des « deux yeux »
  - Dans le cadre d'un système double buffer, ce buffer contient les informations de *back* et *front*.

- Depth Buffer
  - Ce tampon stocke les profondeurs des pixels de l'image.
  - La profondeur est mesuré en fonction de la distance avec la camera.
  - Souvent appelé *zbuffer.*

- Sans Z-buffer, le rendu nécessite le tri des primitives de l'arrière vers l'avant-plan pour un résultat approximativement correct (algorithme du peintre)

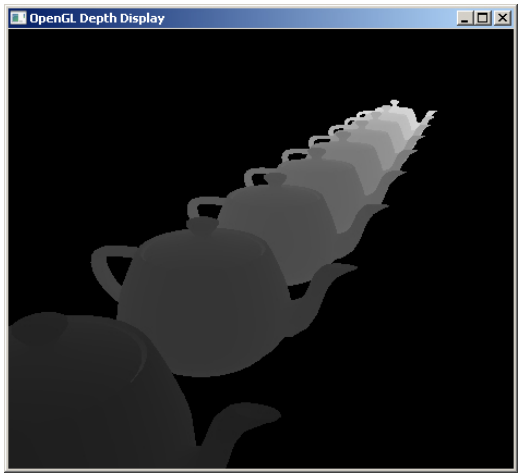


- Algorithme du peintre : on affiche en premier lieu le fond, puis les objets à mi-distance par-dessus, et enfin les objets du premier plan.



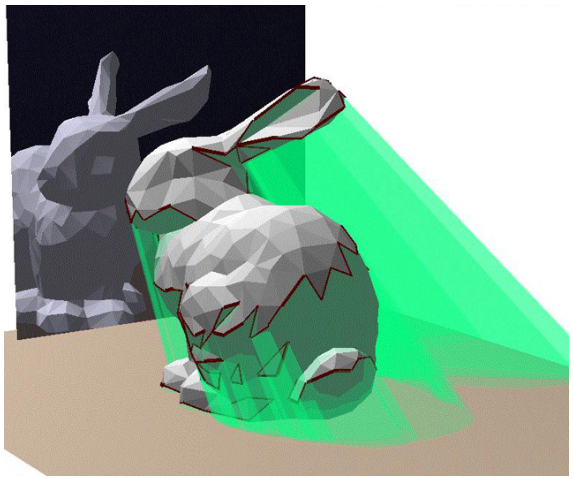
- Cas où l'algorithme du peintre est approximatif: lorsque les primitives se croisent sur l'axe de la profondeur. Il faudrait ici découper les primitives pour un résultat correct.

- Un Z-buffer, en revanche, stocke pour chaque pixel sa profondeur relativement à la caméra.



- Au début de chaque rendu, le Z-buffer est initialisé à une valeur constante (en général le maximum)
- Pour chaque pixel de chaque primitive tracée, sa profondeur est comparée à la valeur déjà stockée pour ce pixel dans le Z-buffer
- Si le nouveau pixel est plus proche, il est écrit dans le colorbuffer et le depth buffer est mis à jour. Sinon, le pixel est rejeté.
- Note : La fonction de comparaison des pixels du depth buffer est paramétrable.

- Stencil (pochoir) buffer
  - Zone image qui va contenir un masque de l'image générée, à l'instar d'un pochoir, et va permettre des opérations de masquage pendant le rendu.
  - Usuellement 8 bits, stockés avec le Z-buffer (format D24S8).
  - Mêmes dimensions que le color buffer.
  - Usage typique : ombres, miroirs, portals.



Calcul des ombres portées en utilisant le stencil buffer (shadow volumes). La silhouette de l'objet est calculée relativement à la position de la lumière, et les faces du cône d'ombre sont générées dans le stencil buffer. Le résultat de l'opération donne un masque correspondant aux pixels dans l'ombre.

- Accumulation Buffer
  - Ce tampon stocke les informations de couleur (RGBA) comme le fait le tampon de couleur.
  - Souvent utilisé pour accumuler les informations (composition d'images)
    - Utilisé dans le cadre de l'anti-alising
  - Buffer image additionnel, permettant l'accumulation pixel à pixel d'images composites.
  - Dimensions identiques au frame buffer.
  - En général utilisé pour les effets du type motion blur, profondeur de champ, anti aliasing.
  - Rarement (voire pas du tout) disponible sur les cartes graphiques grand public.
  - Noté comme obsolète depuis OpenGL 3.0.

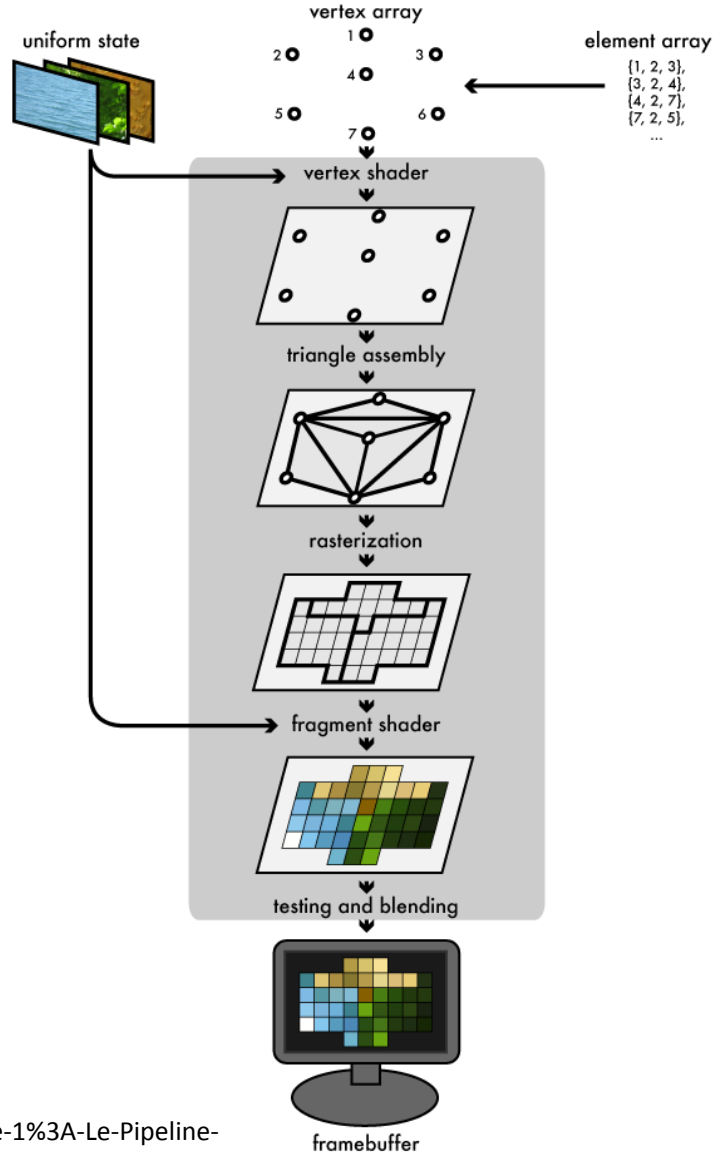
- Vider les buffers:
  - Il s'agit d'une des opérations les plus couteuse.
  - Cette opération peut prendre plus de temps qu'afficher les données.
  - Nécessaire de vider les tampon un par un …
  - .. Mais certains mécanismes sont mis en place pour palier ce problème et écrire sur toutes les mémoires en même temps.

- *void **glClearColor**(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);*
- *void **glClearIndex**(GLfloat index);*
- *void **glClearDepth**(GLclampd depth);*
- *void **glClearStencil**(GLint s);*
- *void **glClearAccum**(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);*

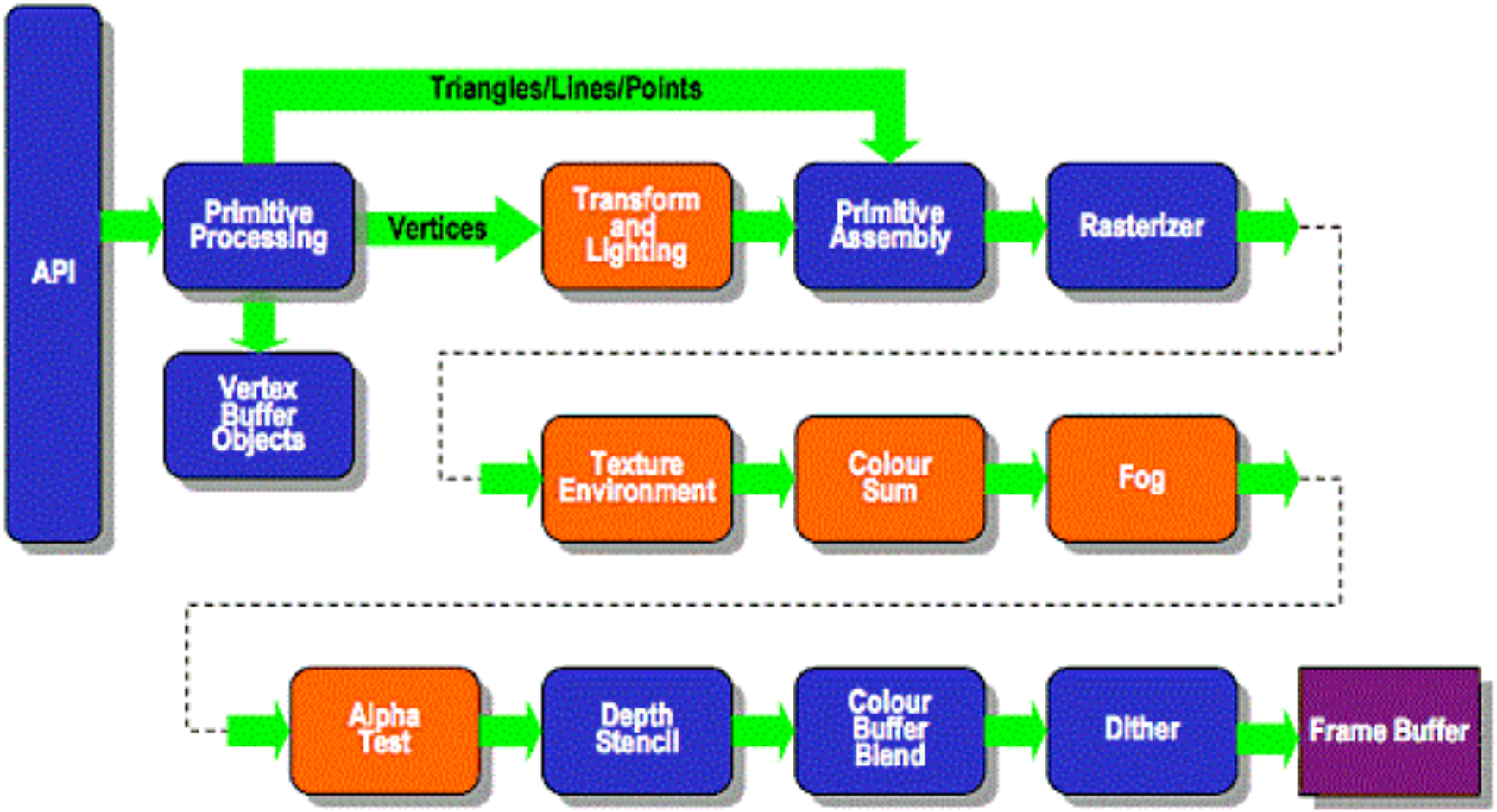
- *void **glClear**(GLbitfield mask); (spécifier le buffer)*

• Pipeline openGL:

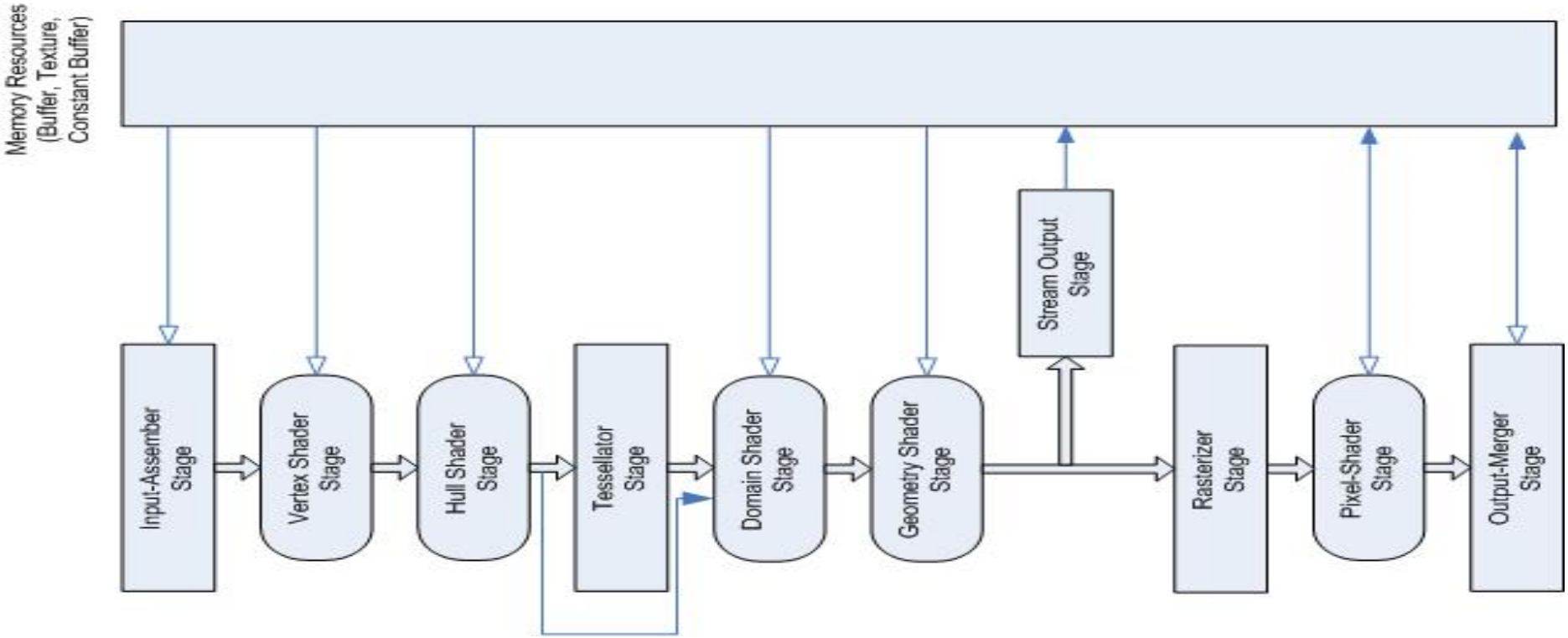
- Le programme principal remplit des buffers de la mémoire géré par OpenGL avec des vertex arrays (des tableaux de vertex).
- Ces vertex sont projetés dans l'espace écran ("screen space" en anglais) par les vertex shaders, assemblé en triangle et enfin "rasterizé" (pixelisé) en fragments de la taille d'un pixel (en gros, un fragment = un pixel).
- Finalement, ces fragments (les pixels) se voit assigné des couleurs (par des fragments shaders) et sont dessinés sur le framebuffer.



http://www.fevrierdorian.com/blog/post/2010/10/04/Une-introduction-à-l-OpenGL-Moderne-Chapitre-1%3A-Le-Pipeline-Graphique#le_pipeline_graphique


API
Primitive Processing
Triangles/Lines/Points
Vertices
Transform and Lighting
Primitive Assembly
Rasterizer
Vertex Buffer Objects
Texture Environment
Colour Sum
Fog
Alpha Test
Depth Stencil
Colour Buffer Blend
Dither
Frame Buffer

Pipeline programmable:



Historiquement, la chaîne de traitement des primitives n'offrait pas de customisation autre que la configuration des états de rendu. L'avènement des pipelines programmables au début des années 2000 a permis d'augmenter de manière considérable la flexibilité du pipeline de rendu, au prix de quelques efforts de programmation supplémentaires:

- D'abord *vertex* et *pixel* (ou fragment) *shaders*, pour le traitement des primitives au niveau du vertex et le traitement individuel des pixels générés
- Puis plus récemment *geometry shaders*, pour la génération à la volée de primitives par la carte
- Enfin, *hull shader* et *domain shader*, pour la tesselation de polygones directement par la carte

Nous ne nous intéresserons dans cette présentation qu'à l'étude des vertex et pixel shaders.

Qu'est ce qu'un shader ?

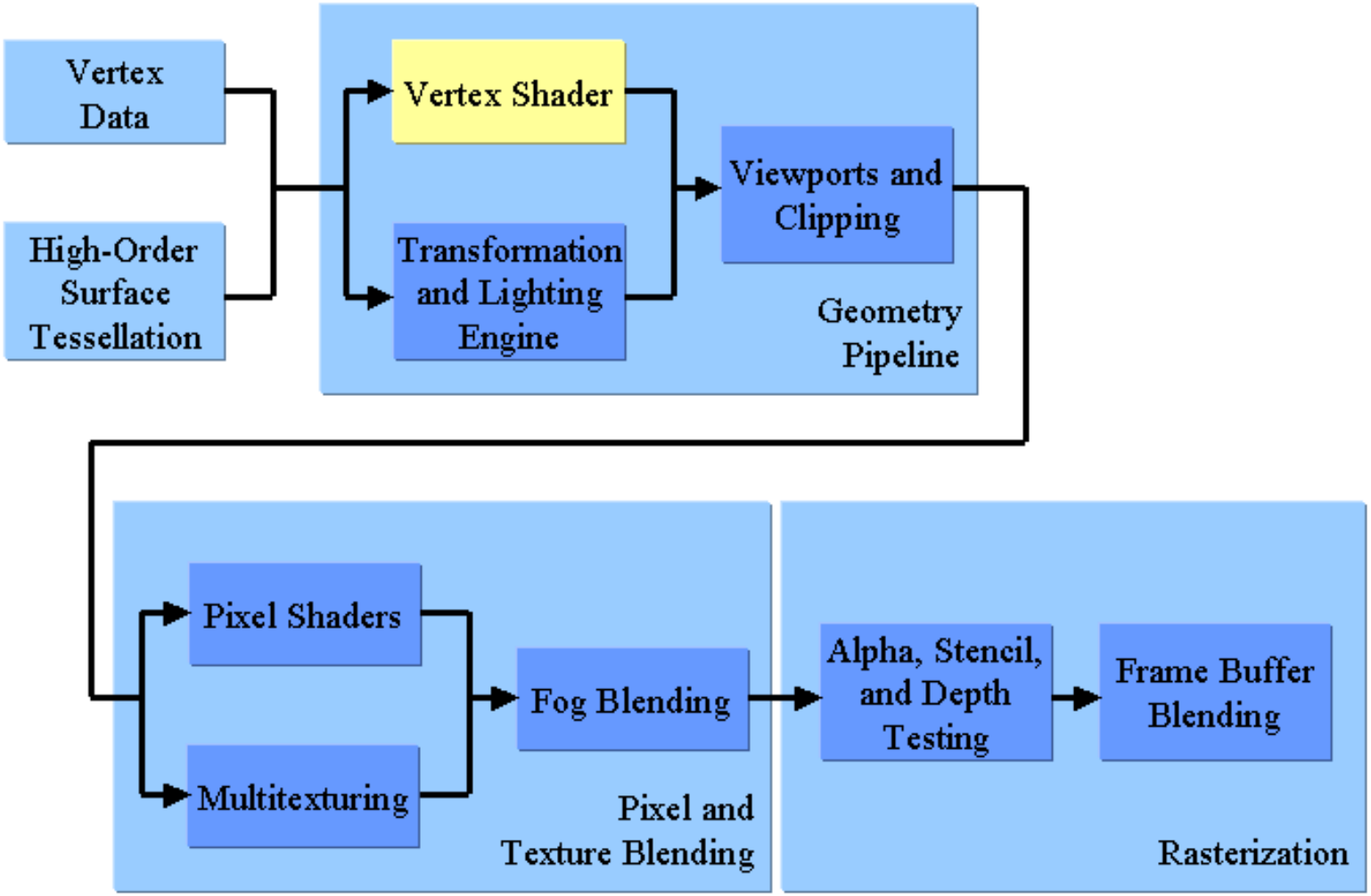
- Un shader est un programme, compilé par le driver (soit au run-time, soit offline), et exécuté par la carte graphique (GPU).

- Bien qu'on puisse écrire des shaders en assembleur, les A.P.I. de rendu supportent des langages haut-niveau, proches du C, pour l'écriture des shaders (Cg, HLSL, GLSL).

- La programmation d'un shader est du type SIMD (Single Instruction, Multiple Data): la quasi-totalité des instructions est applicable sur des données vectorielles, de dimensions 1 à 4, en entier ou en flottants.

- Chaque shader est spécialisé dans un traitement bien précis. Selon le type de shader, les données accessibles diffèrent.

Vertex shader



Vertex shader

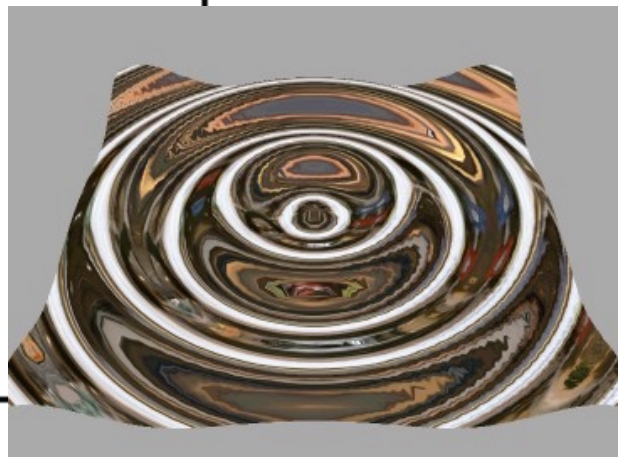
- Le vertex shader calcule l'éclairage et la position des sommets (vertices) du pipeline fixe (pré-OpenGL 3.0). Un seul vertex shader peut être actif à un instant donné dans un contexte.
- Le shader est exécuté pour chaque vertex d'une primitive. Il n'est pas capable de créer ou supprimer des vertices.
- En entrée, le vertex shader a accès aux matrices de transformation et à toutes les propriétés d'un vertex : position, normale, coordonnées de texture, couleur, … Il a également accès à des variables déclarées par l'utilisateur, et initialisées depuis le CPU.
- En sortie, le vertex shader fournit le vertex transformé, et ses propriétés éventuellement modifiées.
- Exemples d'usage : transformations/projections, éclairage, skinning, animations

glCreateShader(), glCompileShader(), glAttachShader(), glCreateProgram(), glLinkProgram(), glUseProgram(), glUniform()

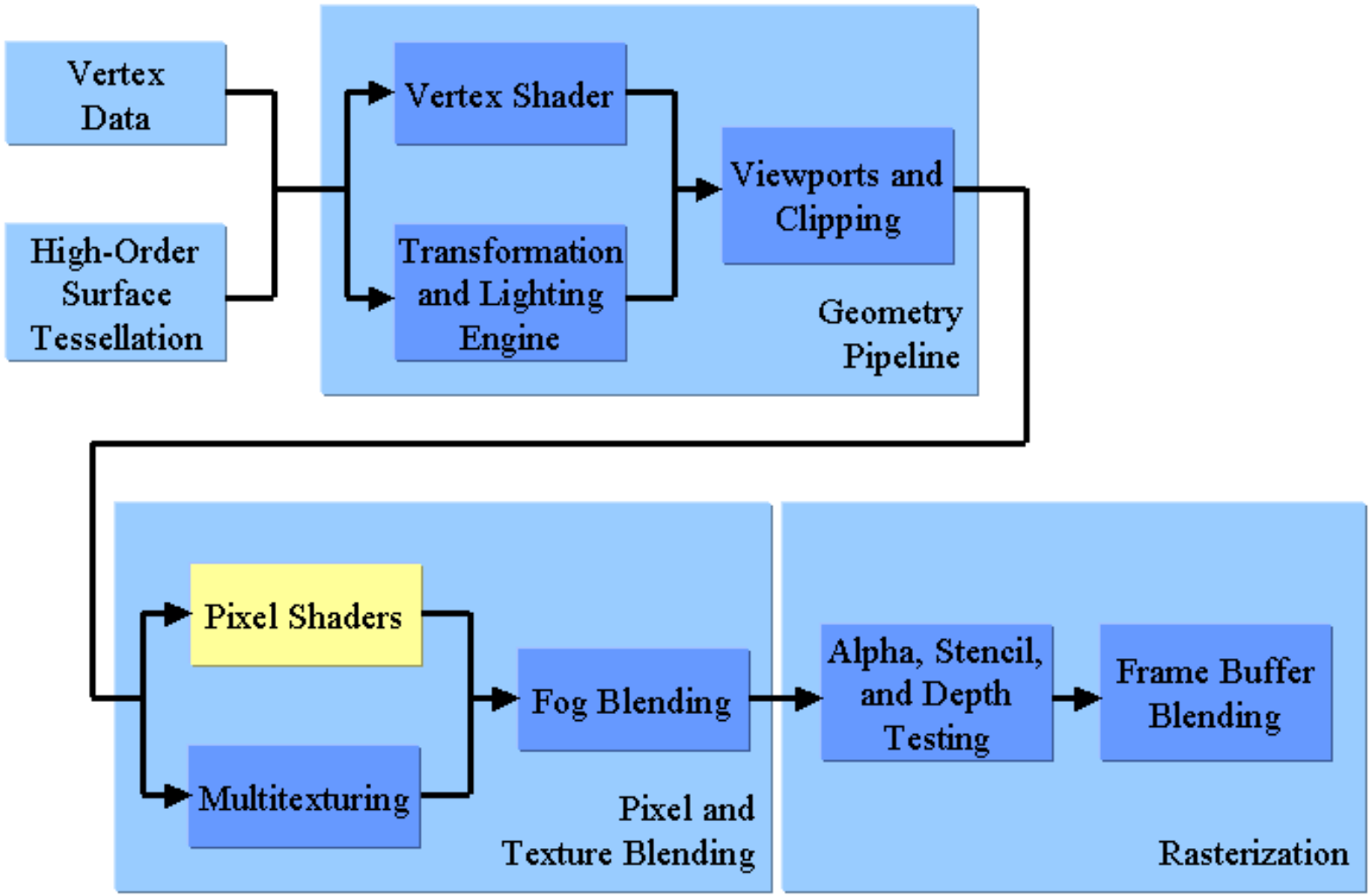
Vertex shader

Déformation sinusoïdale appliquée sur un plan à l'aide d'un vertex shader

```
VS_OUTPUT RenderSceneShallowWaveVS(…)
{
  for(int k=0; k<numOfWavesToSum; k++)
  {
    phaseConstant = ((speed*2*(float)MYPI)/wavelength);

    adjustedWavelength = AdjustWavelengthForShallowWave(
                              wavelength,oceanFloorDepth);

    adjustedK = AdjustKForShallowWave(kexp,oceanFloorDepth);

    dotresult = dot(direction,posVect)*(2.0f*(float)MYPI)/
                   adjustedWavelength;

    finalz = (dotresult+(phaseConstant*time));
    finalz = (sin(finalz)+1.0f)/2.0f;
    finalz = amplitude*pow(finalz,adjustedK);
    …
  }
    …
}
```



Pixel Shader

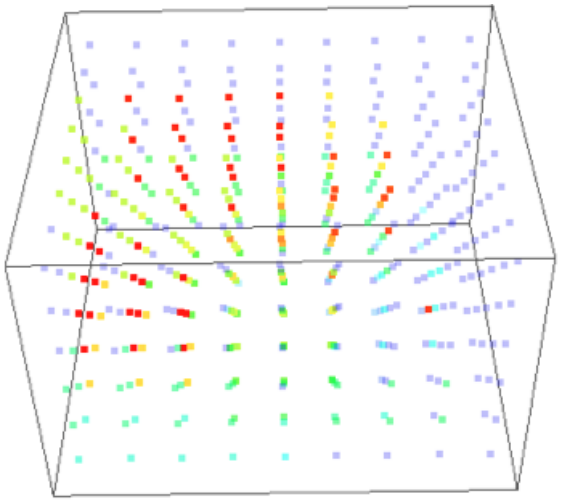
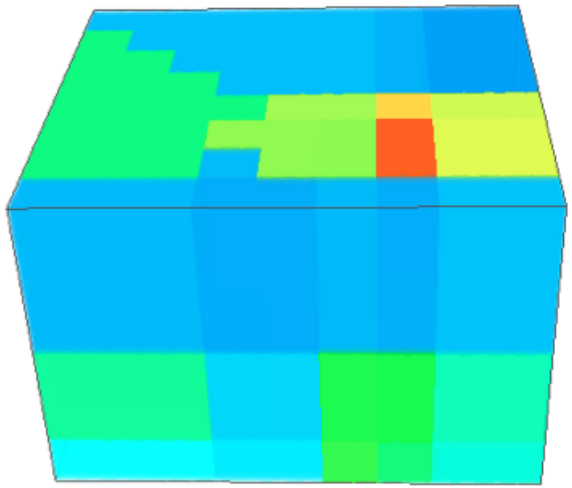
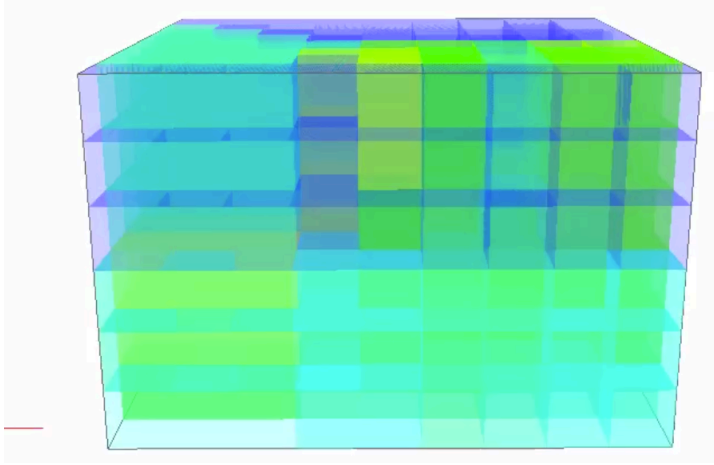
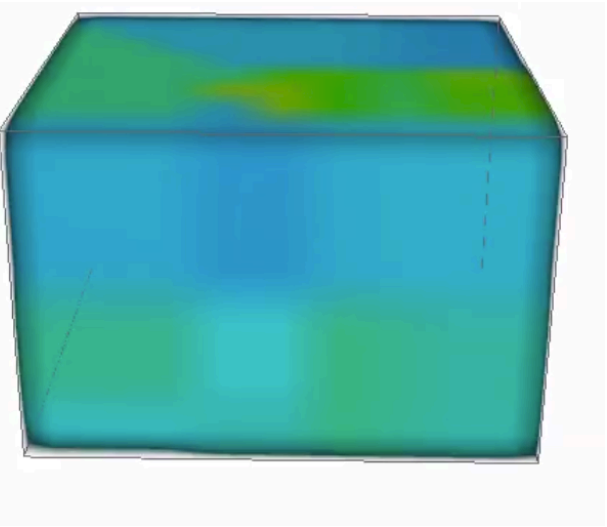
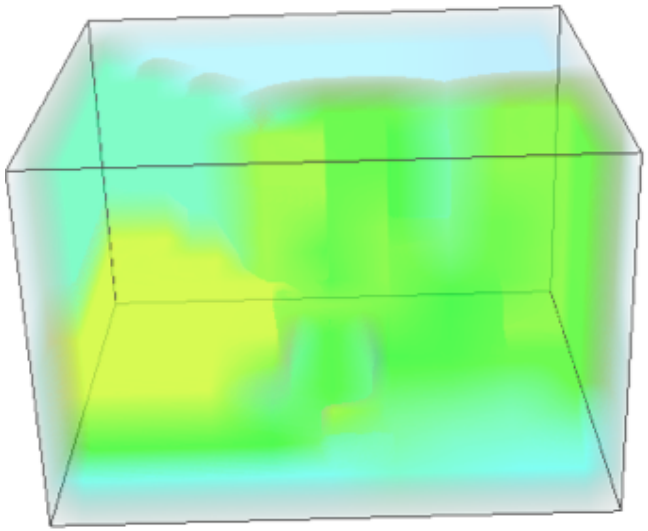
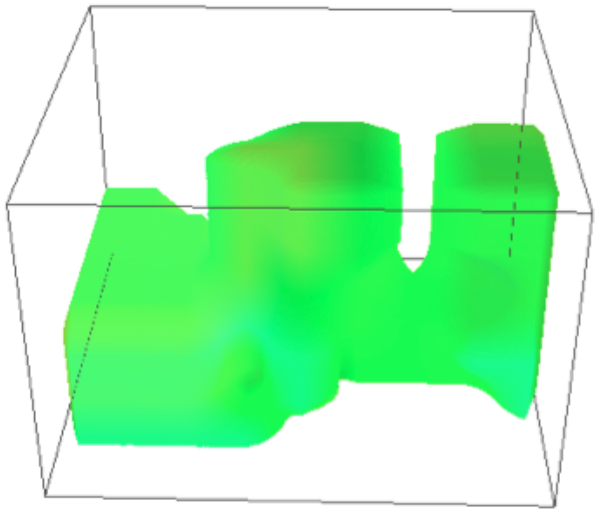


Pixel shader

- Le pixel shader calcule les paramètres (couleur, transparence, profondeur) de chaque pixel de l'image. Un seul pixel shader au plus peut être actif à un instant donné dans un contexte.
- Le shader est exécuté pour chaque pixel de chaque primitive (sous conditions: early Z-test).
- En entrée, le pixel shader reçoit les paramètres associées au pixel (issus du vertex shader et interpolés): couleur, coordonnées de texture, transparence, profondeur. Il a également accès à des variables déclarées par l'utilisateur, et initialisées depuis le CPU.
- En sortie, le pixel shader fournit une couleur/transparence/profondeur pour le pixel traité, et peut éventuellement le supprimer.
- Tous les effets graphiques avancés utilisent un pixel shader (réflexions, bump mapping, deferred rendering, etc..).

glCreateShader(), glCompileShader(), glAttachShader(), glCreateProgram(), glLinkProgram(), glUseProgram(), glUniform()

Des exemples

Programmation GPU

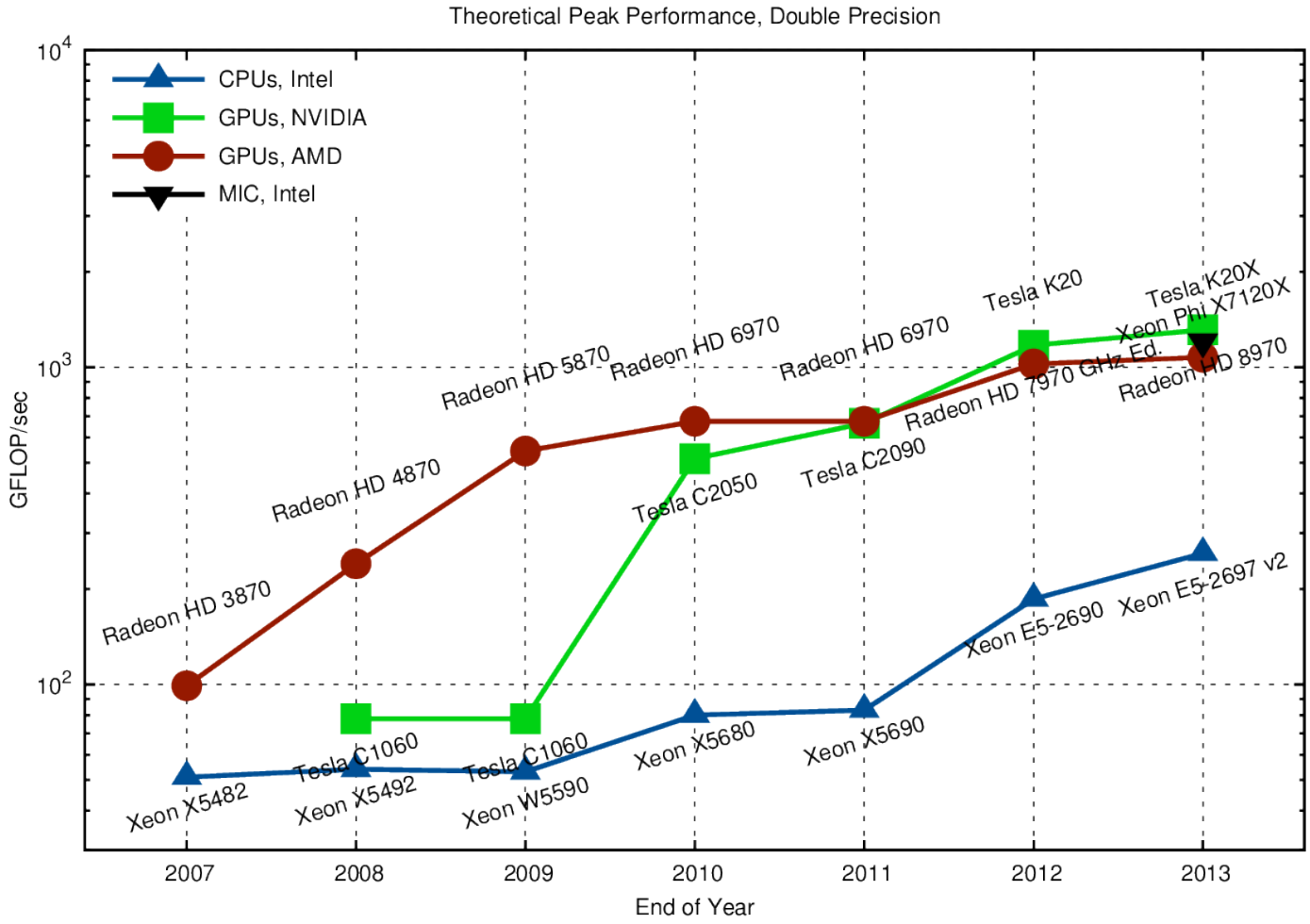
- Vers 2001, développement du GPGPU
  - General-purpose computing on GPUs
- Utiliser l'architectures de GPU (100 de cœurs simples)
- Très efficace pour certaines applications

- Deux points importants:
  - Les shaders
  - Les opérations flottantes

- Arrivé de nouveaux langauge:
  - CUDA
  - OpenCL
  - …

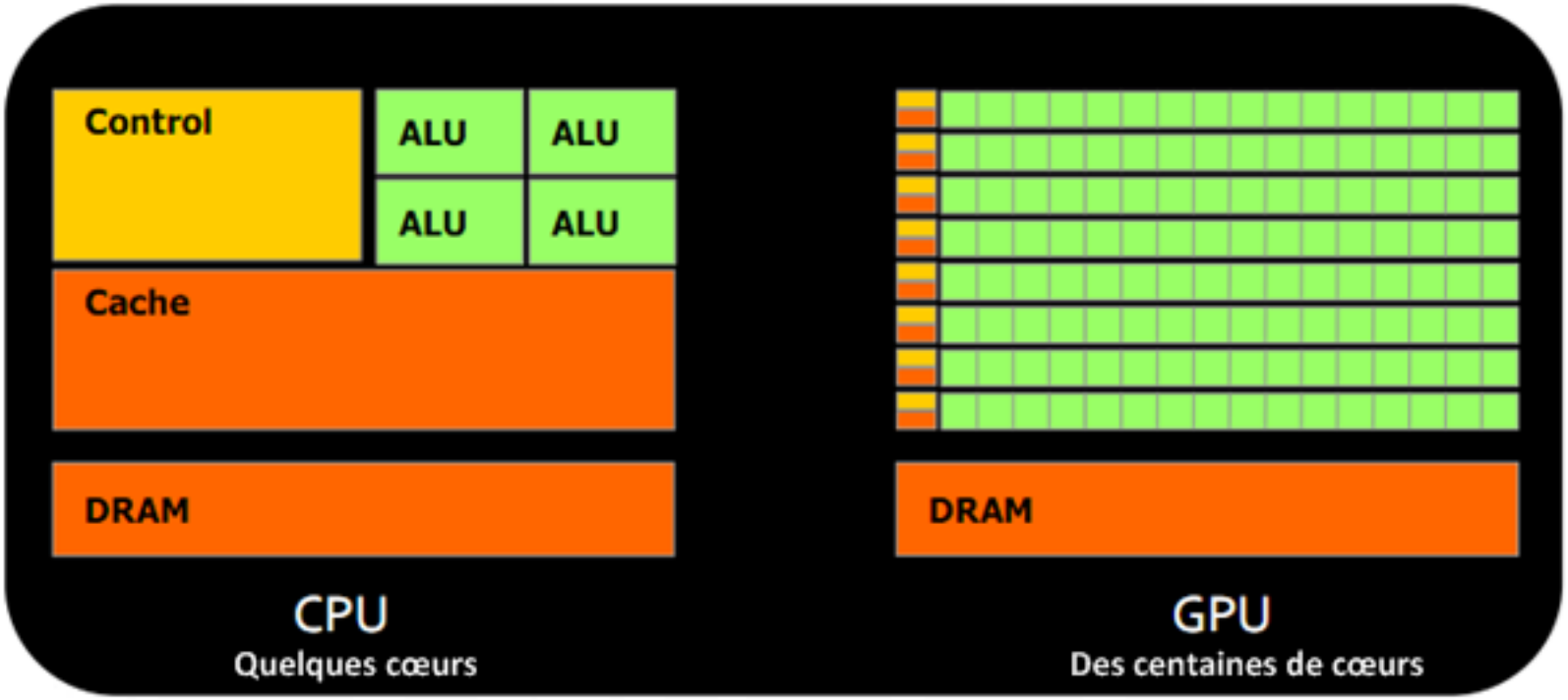
- OpenCL:
  - Permet de paralléliser des applications sur de nombreuses architectures:
    - GPU
    - CPU
    - Many-core
  - Support de nombreux constructeurs
  - Un modèle extensible

- Pourquoi réaliser un calcul sur GPU ?

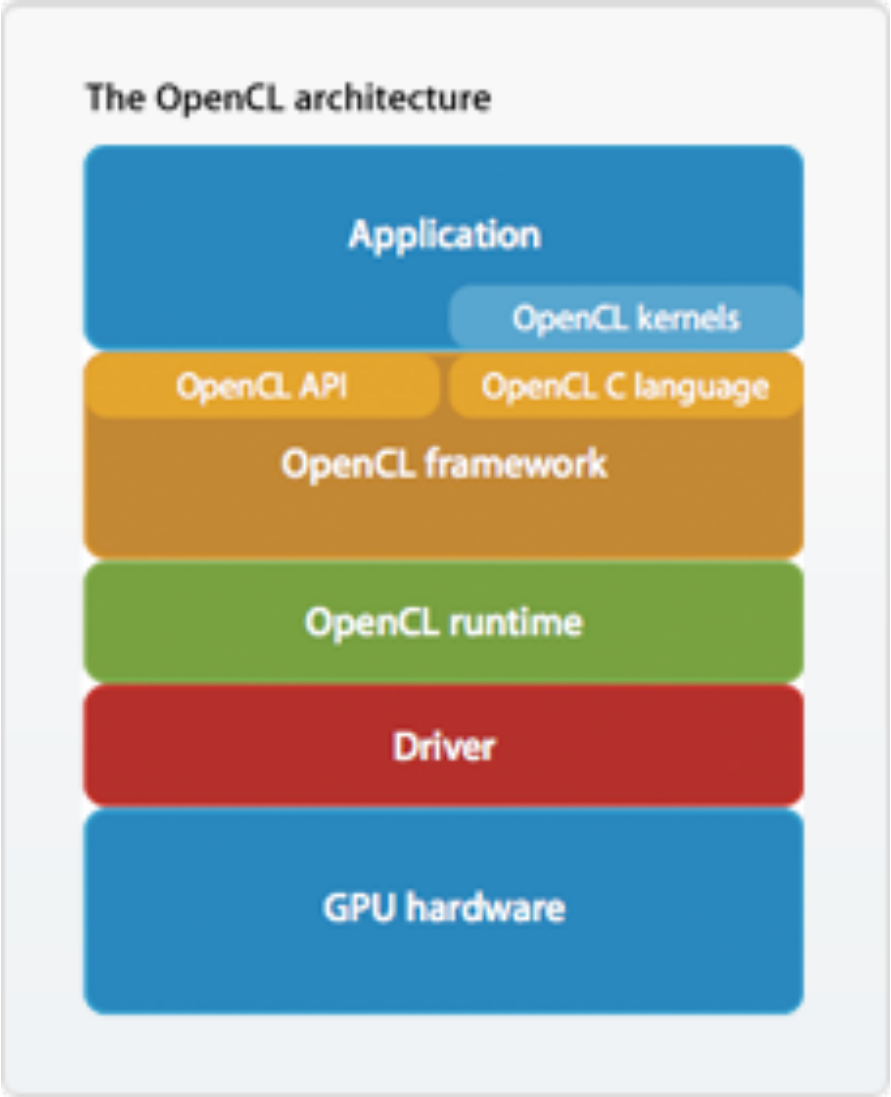


- GPU vs CPU

Plus de transistors sur un GPU dédié au calcul



- Au sommet *application*, le code source du programme appelant (C, Java, Python, etc.) exécuté par le CPU
- En dessous, le *framework* comprenant l'API (fonctions utilisables par le code source appelant) et le langage OpenCL permettant de développer des programmes exécutables par le GPU.
- Plus bas, on trouve le *runtime*, c'est-à-dire l'implémentation permettant d'exécuter le code OpenCL.
- Ensuite le *driver*, le pilote permettant de communiquer avec le GPU
- En dernier se trouve le périphérique GPU (la carte graphique).



http://www-igm.univ-mlv.fr/~dr/XPOSE2011/openclgpucomputing/comment_programmer.php

Principe d'OpenCL:

1. Obtenir le contexte
2. Récupérer l'id de l'accélérateur
3. Créer le contexte pour le matériel
4. Créer le programme à partir du code source
5. Build du programme
6. Création des kernels
7. Création de la queue de commande pour le matériel
8. Allocation de la mémoire/ déplacement des données
9. Association des arguments aux kernels
10. Déploiement des kernels pour l'exécution
11. Déplacement des résultats vers la mémoire hôte
12. Relâchement du contexte, programme, kernel et mémoire.

1. Obtenir le contexte
   - Méthode:
     - clGetPlatformIDs(1, &platform, NULL);
   - Utilisé deux fois pour récupérer le nombre de dispositifs

2. Récupérer l'id de l'accélérateur
   - Méthode:
     - clGetDeviceIds(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

3. Créer le contexte pour le matériel
   - Contexte: conteneur abstrait attaché au matériel
   - Contient : kernels, objets en mémoire, liste des commandes
     - Méthode:

       context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);

4. Créer le programme à partir du code source
   - L'hôte lit le fichier source (**.cl)*
   - Il crée un *cl_program* attaché au contexte
     - Méthode:
     - program = clCreateProgramWithSource (context, 1, (const char**) &program_buffer, &program_size, &err);

5. Build du programme
   - Compilation du programme lors de l'exécution
     - Compile même en cas d'erreur
     - Vérification des erreurs lors de l'exécution
   - Méthode:

     clBuildProgram(program, 0,..) ;

6. Création des kernels
   - Noyaux de calculs
   - Création des *cl_kernel* à partir des programmes précédents
   - Méthode:

     kernel = clCreateKernel(program, "kernel_name", &err) ;

7. Création de la queue de commande pour le matériel
   - Chaque queue est attaché à un matériel
     - Méthode:
       - queue = clCreateCommandQueue(context, device, 0, &err);

8. Allocation de la mémoire/ déplacement des données
   - Méthode:
     - memObject = clCreateBuffer (context, NULL, SIZE_N, NULL, &err)
     - clEnqueueWriteBuffer(command_queue, memObject, ..., TOTAL_SIZE, hostPointer, ...)
   - La mémoire peut être stocké en buffer ou en images.
     - Mémoire contigu
     - Possibilité de lecture / écriture

9. Association des arguments aux kernels
   - Méthode:

     cl_int clSetKernelArg (kernel, arg_index, arg_size, *arg_value)

10. Déploiement des kernels pour l'exécution
   - Méthode:

     clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL);

     global_size = TOTAL_NUM_THREADS;

     local_size = WORKGROUP_SIZE;

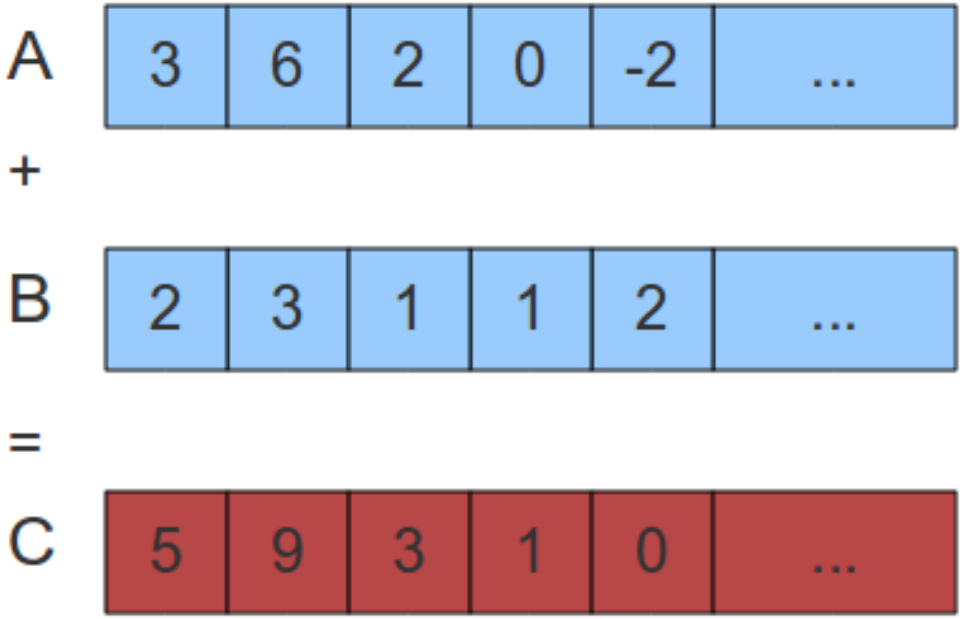
11. Déplacement des résultats vers la mémoire hôte
    - Méthode:

      clEnqueueReadBuffer(command_queue, memObject, blocking_read, offset, TOTAL_SIZE, hostPointer, 0, NULL, NULL)

12. Relâchement du contexte, programme, kernel et mémoire.
    - Méthode:
      - clReleaseMemObject(memObject)
      - clReleaseKernel(kernel)
      - clReleaseProgram(program)
      - clReleaseContext(context)

Exemple simple : somme de deux tableaux

```
__kernel void vector_add(__global const int *A,
                         __global const int *B,
                         __global int *C)
{
    // Get the index of the current element to be processed
    int i = get_global_id(0);

    // Perform operation
    C[i] = A[i] + B[i];
}
```



Autre exemple simple (opencl)

```
__kernel void square(
    __global float* input, __global float* output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

- Nombreuses applications possibles:
  - Simulations physiques
  - Systèmes de particules
  - Simulation des tissus
  - Modélisation et rendu par voxels
  - Etc.

- Pour aller plus loin:
  - Documentation OpenGL

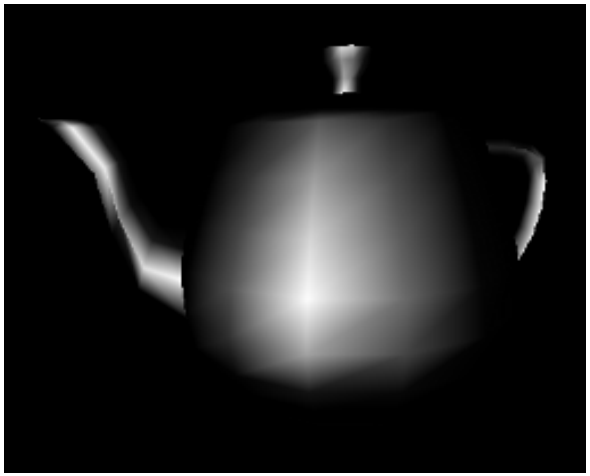
    http://www.opengl.org/documentation/

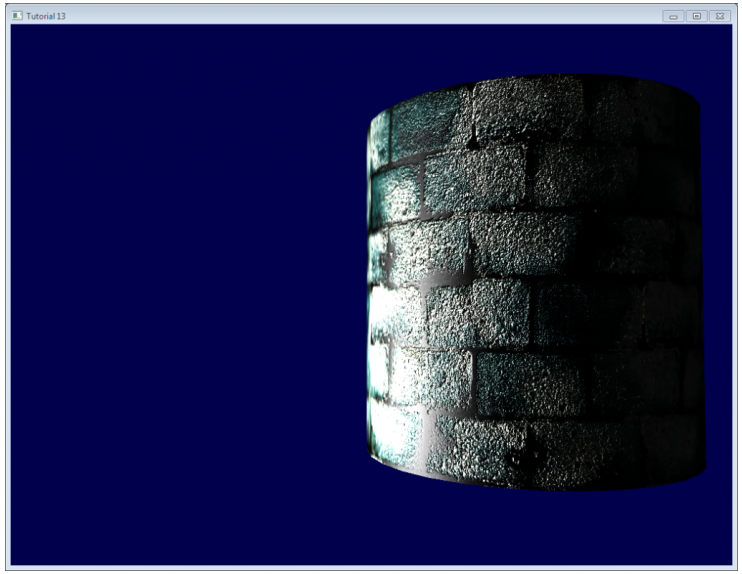
  - Documentation OpenCL

    http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/

- Et ….
- Maintenant …

- Vous pouvez réaliser votre dernier commit pour le TP précédent.





TP