M1 - Aquisition, Analyse et Traitement d'Images

OpenGL et compute shaders

J.C. Iehl

April 8, 2014

Programmation GPU

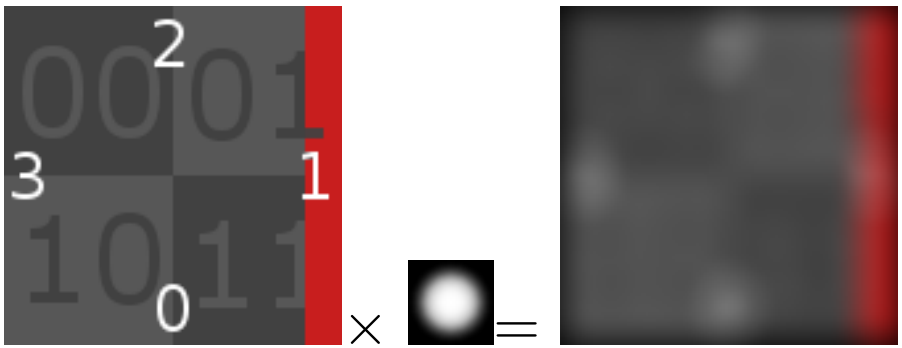
pourquoi ?

- processeurs de calcul parallèles,
- rapides sur certaines taches,
- disponibles (inclus dans tous les processeurs recents) !

Programmation GPU

rapides ?

- exemple : filtrer une image
- cpu : 200ms
- gpu : 0.8ms
- assez rapide ?
- (remarque : c'est un des meilleurs cas possible pour le gpu)



Programmation GPU

comment ça marche ?

- plusieurs API dédiées existent :
- CUDA,
- OpenCL,
- plusieurs API 3D existent :
- Direct3D 11 + DirectCompute,
- OpenGL 4.3 + compute shader,
- extensions C++ : C++AMP, OpenACC,

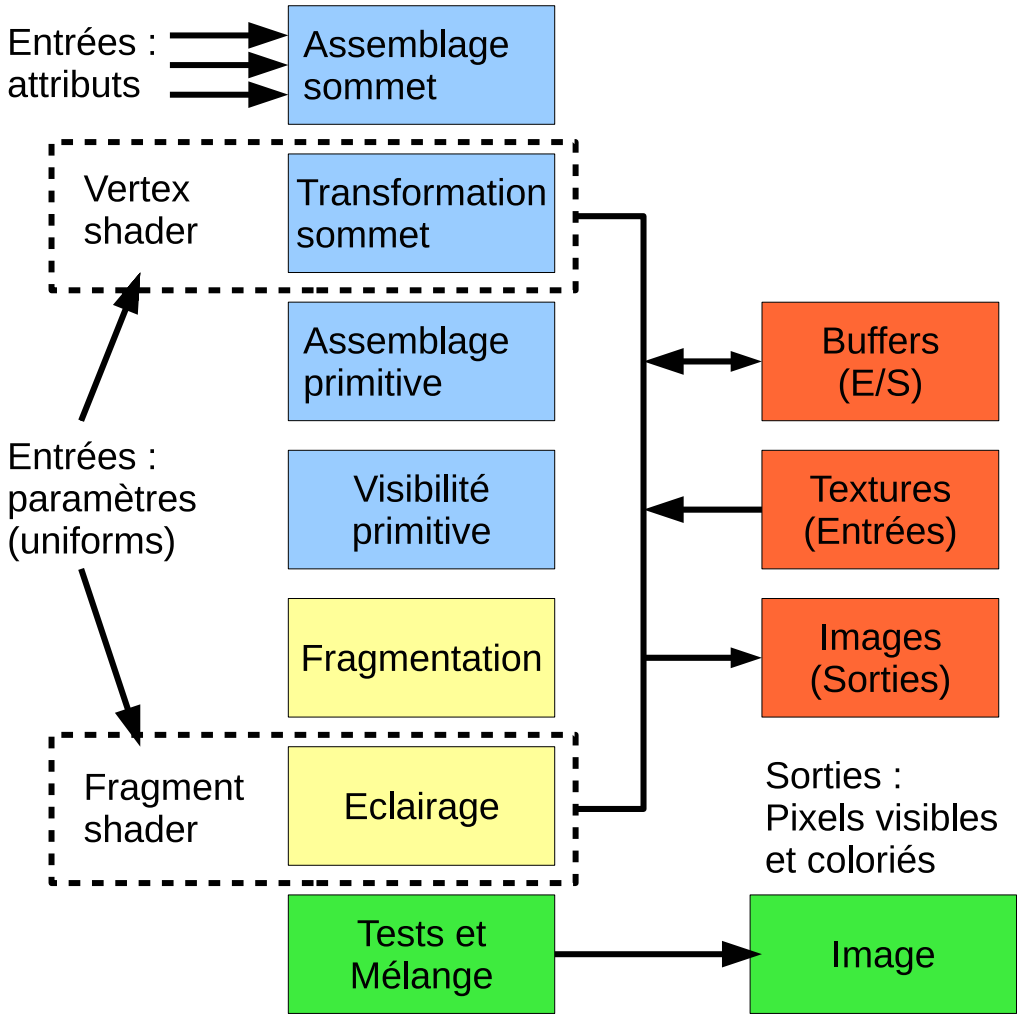
dans ce cours : openGL 4.3 + compute shader.

OpenGL : pipeline graphique

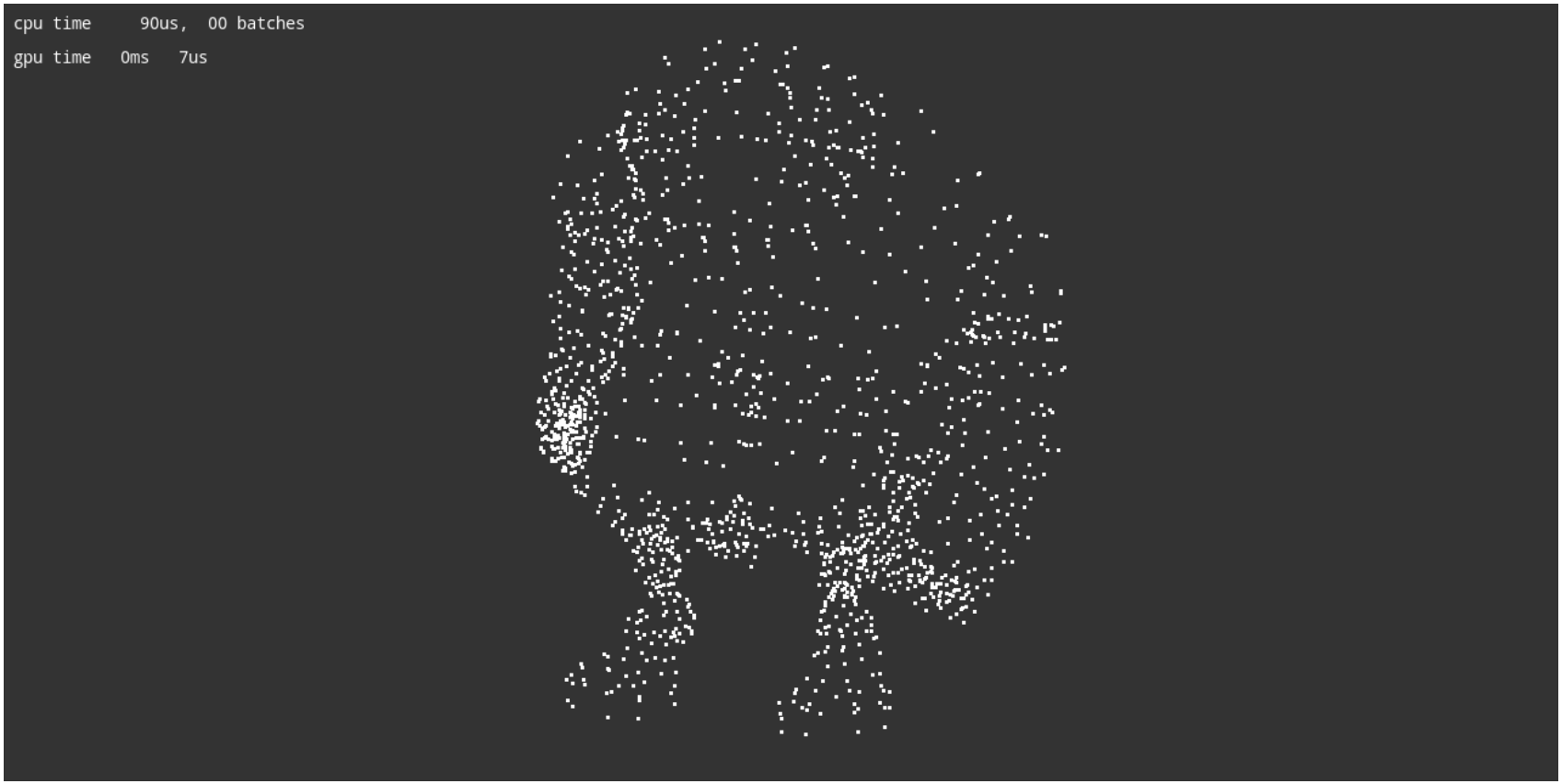
qu'est ce que c'est ?

- une API 3D :
- permet d'utiliser une carte graphique pour afficher des objets maillés (surface découpée en triangles),
- plusieurs étapes :
- certaines sont "cablées" / fixes,
- d'autres sont programmables : shaders.
- + utiliser une carte graphique pour réaliser des calculs : compute shaders

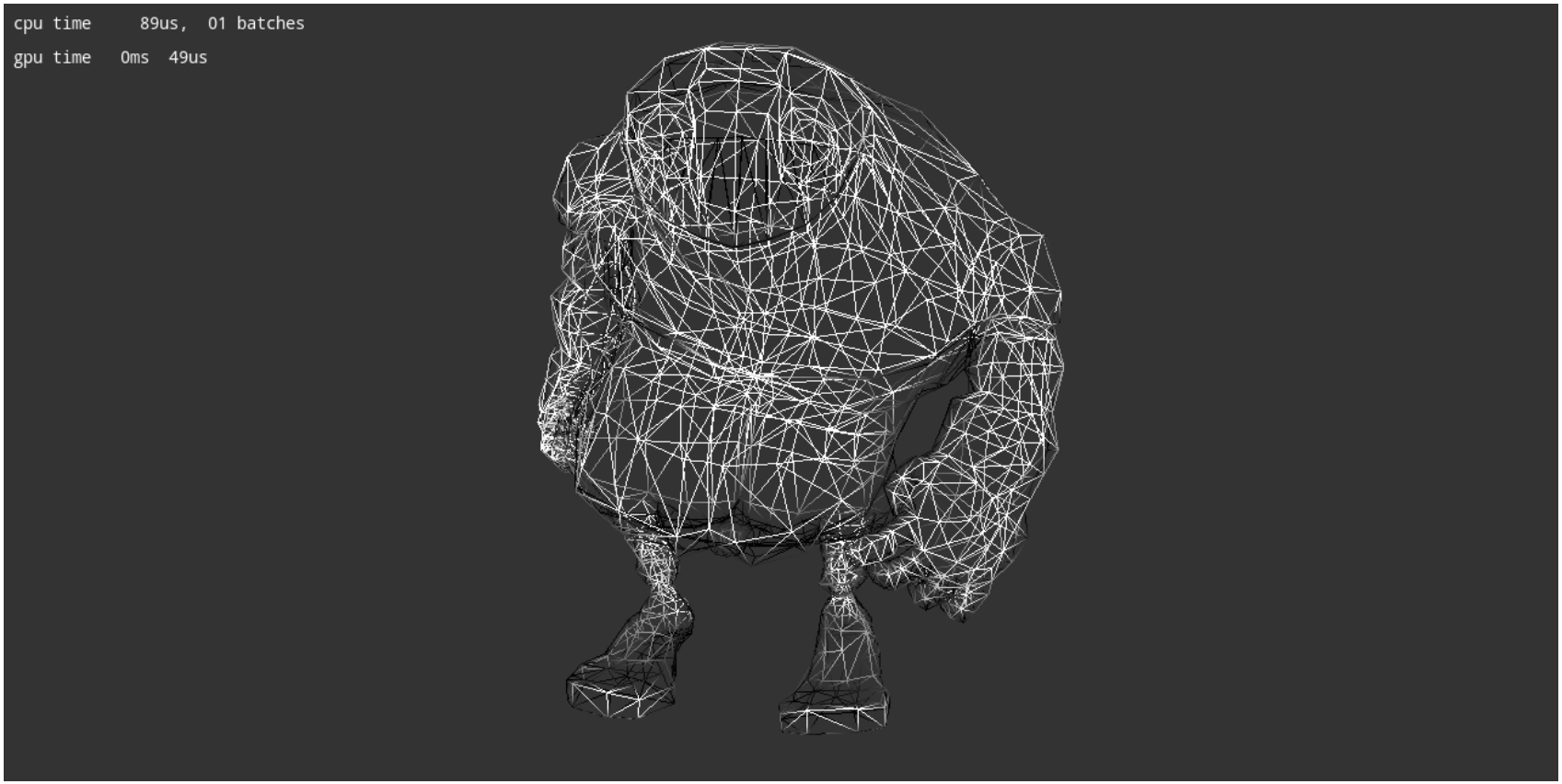
OpenGL : pipeline graphique



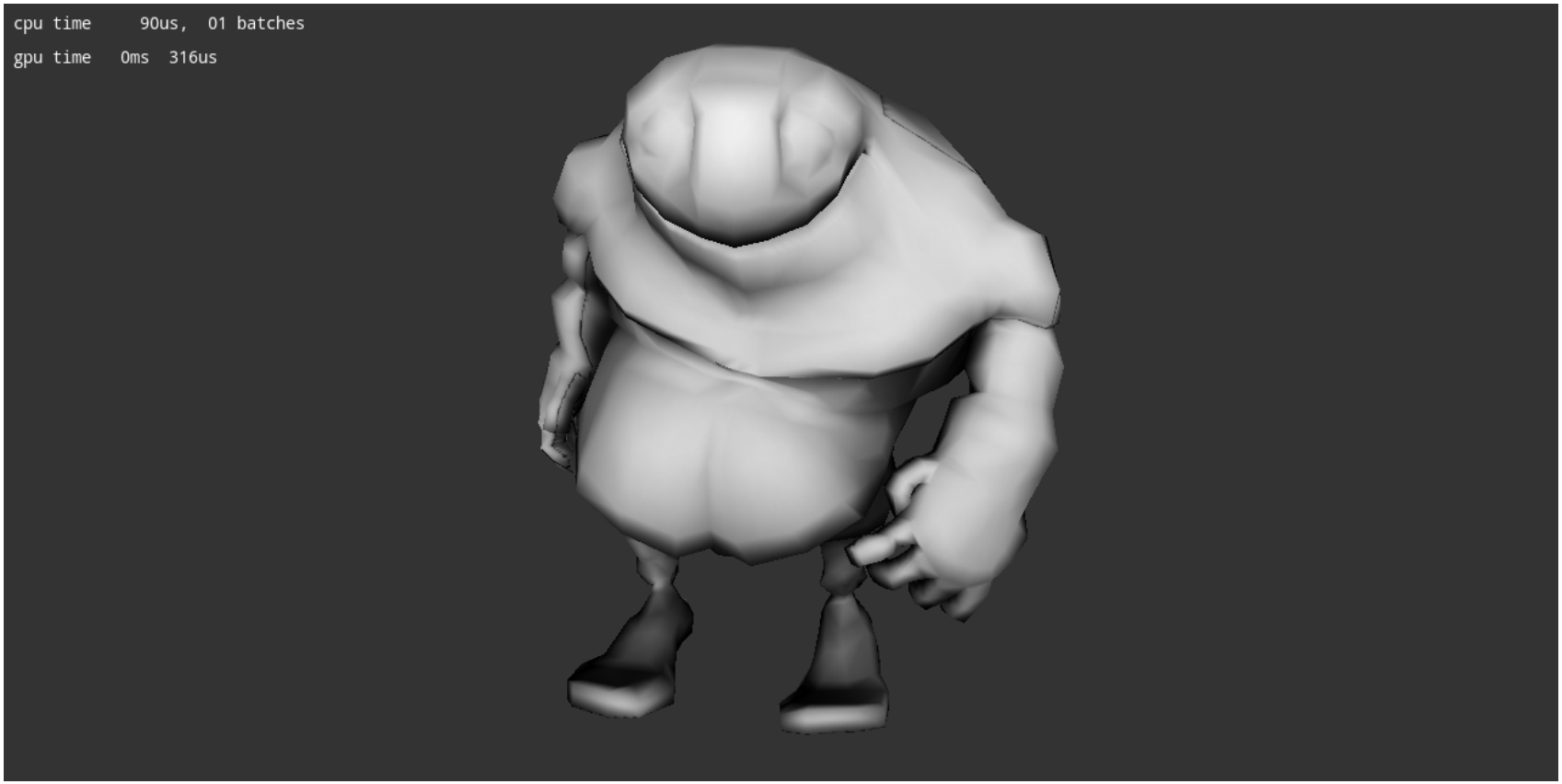
OpenGL : pipeline graphique



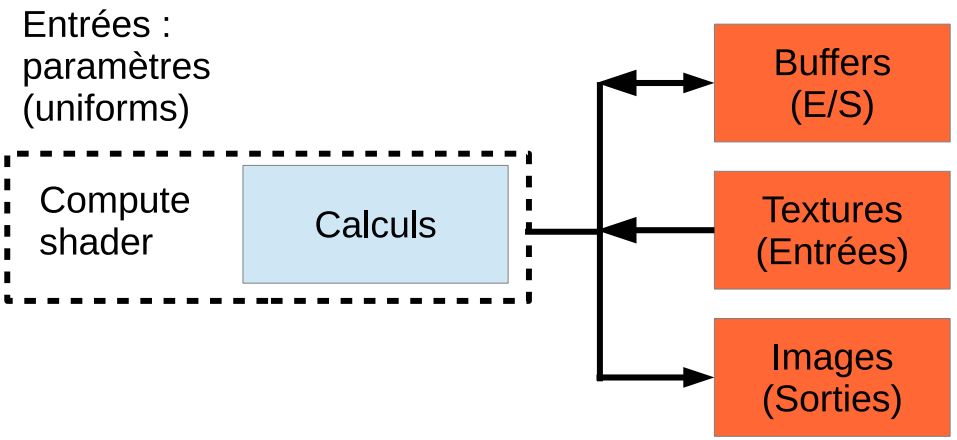
OpenGL : pipeline graphique



OpenGL : pipeline graphique



OpenGL : pipeline calcul



OpenGL : coté application / coté materiel

programmation openGL :

- coté application : création / configuration des objets openGL (shaders, buffers, textures, images, etc.)
- coté matériel / carte graphique : shaders (langage dédié GLSL).

objectif :

- paramétrer le pipeline (les parties cablées / fixes),
- paramètres / entrées des shaders,
- sorties des shaders,
- exécution du pipeline.

OpenGL : coté application

API OpenGL :

- OpenGL est une (très) grosse API :
- complexe à manipuler,
- dans ce cours :
- utilisation d'une librairie gKit,
- utilisation d'une application configurable dédiée : computeKit.

OpenGL : computeKit

Langage GLSL

sans (trop de) surprises :

- basé sur le C,
- types de base : int, float, vecteurs (vec2, ivec2, vec3, ivec3), matrices (mat4, etc)
- + constructeurs et opérateurs C++,
- + opérateurs sur les vecteurs, les produits matrices vecteurs,
- + librairie standard math : trigonometrie, etc.
- + lecture / écriture dans des textures / images (avec gestion automatique des bordures...)
- primitives de synchronisation (opérations atomiques, spinlocks...)

Langage GLSL

référence GLSL :
http://www.opengl.org/sdk/docs/manglsl/

référence openGL :
http://www.opengl.org/sdk/docs/man4/

doc complète d'openGL :
http://www.opengl.org

Anatomie d'un shader GLSL

```
#version 430                        // compute shader, core profile
#ifdef COMPUTE_SHADER               // impose par computeKit
// entrees
uniform sampler2D image0;           // textures a filtrer,
uniform sampler2D image1;           // noms imposes par computeKit

// parametres "supplementaires"
uniform vec4 alpha= vec4(0.5, 0.5, 1, 0);

// sortie : declare une image standard en ecriture
layout( binding= 0, rgba8 ) writeonly
uniform image2D resultat;           // 1 sortie imposee par computeKit

// configuration du nombre de threads par groupe
layout( local_size_x= 16, local_size_y= 16 ) in;
void main( )
{
    // coordonnees du pixel associe au thread
    ivec2 pixel= ivec2(gl_GlobalInvocationID.xy);

    // lire un pixel dans la texture
    vec4 p= texelFetch(image0, pixel, 0);

    // ecrire le pixel dans l'image resultat
    imageStore(resultat, pixel, p);
}
#endif
```

Types de base

bool, uint, int, float :

- identiques au C/C++

bvec2/3/4, uvec2/3/4, ivec2/3/4, vec2/3/4 :

- `vec4 a= vec4(1, 2, 3, 4);`
- `vec3 b= a.xyz;`
- `vec3 c= a.rgb;`
- `vec3 d.rgb= a.bgr;` !!
- `vec3 d= a.rrr;` !!
- `vec3 d= vec3(a.r);` !!

Opérations de base

vecteurs :

- `vec3 a; xxx b; vec3 r= a + b;`
- `vec3 a; xxx b; vec3 r= a * b;`
- `vec3 a; xxx b; vec3 r= a / b;`
- `vec3 a, b; float r= distance(a, b);`
- `vec3 a; float r= length(a);`
- `vec3 a, b; vec3 r= dot(a, b);`
- `vec3 a, b; vec3 r= cross(a, b);`
- `vec3 a; vec3 r= normalize(a);`

lire un pixel dans une texture

déclarer une texture en entrée :

- `uniform sampler2D image0;`
- sampler2D: texture classique,
- sampler1D, sampler3D, sampler2DArray, etc...

lire un pixel /texel :

- `vec4 couleur= texelFetch(image, coordonnees, 0);`
- `ivec2 coordonnees` $\in$ [0..largeur[$\times$[0..hauteur],
- 0 : une texture openGL contient plusieurs surfaces indexées...

lire un pixel dans une texture

remarque :

- computeKit numérote les textures automatiquement :
- `image0, image1`, etc.

lire un pixel dans une texture

les pixels en dehors de la texture :

- sont noirs,
- si `coordonnees` $\notin$ [0..largeur[$\times$[0..hauteur], `texelFetch()` renvoie `vec4(0);`

un pixel / texel est composé :

- de 4 canaux : rgb et transparence (a),
- en général : on ne s'interresse qu'aux canaux rgb,
- `vec3 couleur= texelFetch(image, coordonnees, 0).rgb;`

utiliser des paramètres "supplémentaires"

déclaration :

- `uniform vec3 couleur;`
- les compute shaders n'utilisent que des paramètres `uniform`,

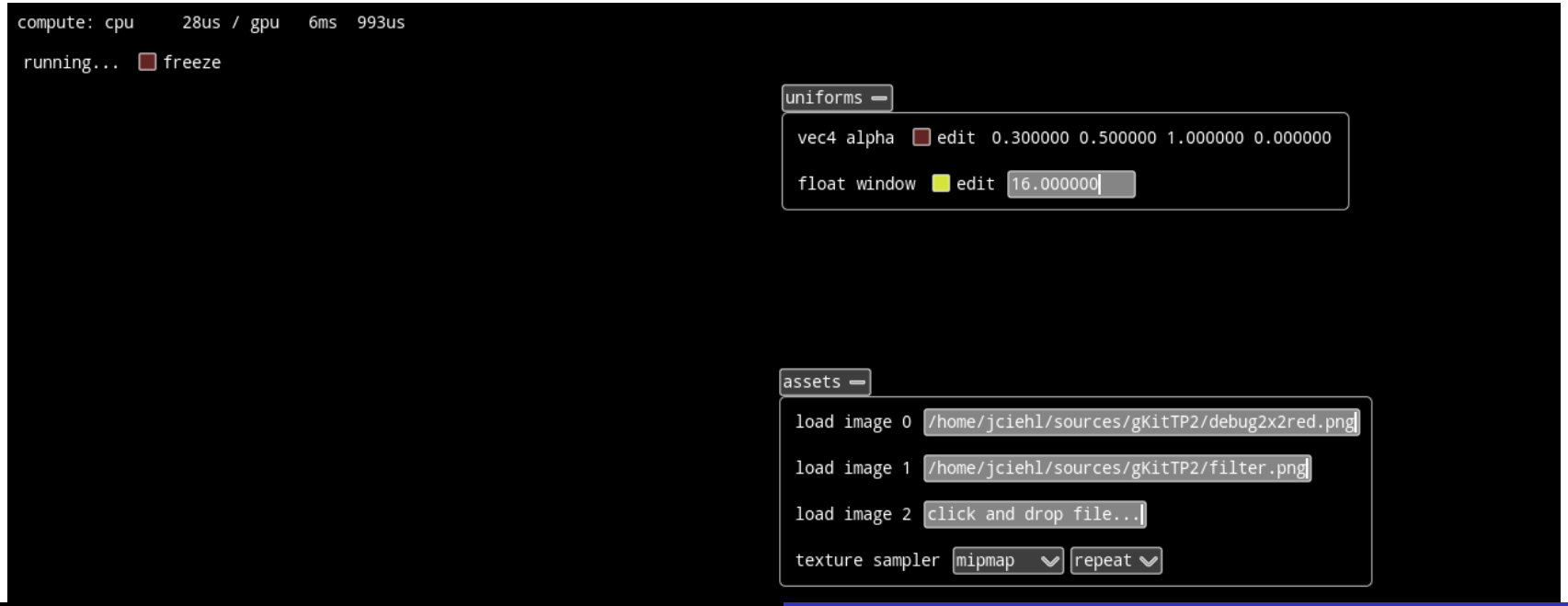
valeur par défaut :

- affectation standard :
- `uniform vec3 couleur= vec3(1, 2, 3);`

utiliser des paramètres "supplémentaires"

ou utiliser le panneau "uniforms" :

- dans l'interface de computeKit :
- affiche par défaut la valeur actuelle des paramètres,
- mais possibilité de les éditer / modifier.



écrire un pixel dans l'image résultat

déclarer une image en sortie :

- `layout( binding= 0, rgba8 ) writeonly uniform image2D resultat;`
- le nom de l'image est libre,
- mais `layout( binding= 0, rgba8 ) writeonly` est "imposé" par openGL et computeKit.

ecrire :

- `imageStore(image, coordonnees, couleur);`

Parallélisme et GPU

un GPU est composé de plusieurs groupes d'unités de calcul :

- qui travaillent en parallèle sur un ensemble de taches,
- les taches sont "affectées" à des threads,
- déterminer quelle tache doit réaliser chaque thread...

un GPU est construit pour :

- le parallélisme de données et d'instructions,
- difficile de faire du parallélisme de tache
  (une tache se décompose dynamiquement en sous-taches),
- solutions techniques spécialisées assez lourdes et gains limités par rapport à une solution CPU.

parallélisme de données

exemples :

- traiter tous les sommets d'un maillage,
- traiter tous les triangles visibles d'un maillage,
- traiter tous les "pixels" des triangles visibles d'un maillage,
- ou ... traiter tous les pixels d'une texture,
  par exemple, pour filtrer une image.

en gros :

- un tableau de données en entrée,
- traiter chaque élément du tableau,
- écrire le résultat pour chaque élement.

espace d'iteration

en gros :

- *traiter chaque élément du tableau*,
- une tache / un thread par élement,
- *écrire le résultat pour chaque élement*.
- déterminer où écrire le résultat de chaque élement.

espace d'iteration

solution séquentielle / cpu :

```
const vector<entree>& donnees;
vector<sortie>& resultats;

void traiter( )
{
    for(int i= 0; i < donnees.size(); i++)
        resultats[i]= traiter(donnees[i]);
}
```

solution parallèle / gpu :

```
creer threads(0 .. donnees.size() -1)

const vector<entree>& donnees;
vector<sortie>& resultats;

void traiter( thread_index index )
{
    resultat[index]= traiter(donnees[index]);
}
```

espace d'iteration

tout n'est pas toujours aussi simple / direct :

- certains algorithmes créent plusieurs `sortie` par `entree`,
- certains algorithmes créent très peu de `sortie` pour l'ensemble d'`entree`,
  `entree m= min(donnees)`...
- un thread peut aussi réaliser plusieurs taches...

cas simple pour l'instant.

OpenGL + compute shaders :

espace d'iteration / création de threads :

- espace 3D,
- les threads sont groupés,
- `glDispatchCompute( groupes_x|y|z );`
- le shader déclare le nombre de threads par groupe : cf. `layout( local_size_x= 16, local_size_y= 16 ) in;`

au total : nombre de groupes $\times$ nombre de threads par groupe.

exemple :

combien de taches ?

- `glDispatchCompute( 16, 16, 1 );`
- `layout( local_size_x= 16, local_size_y= 16 ) in;`

comment les repérer ?

- déterminer quelle tache correspond à quel thread...

espace d'iteration OpenGL :

comment identifier le thread ?

- `gl_GlobalInvocationID` : index "lineaire" du thread,
- `gl_LocalInvocationID` : index du thread dans son groupe,
- `gl_WorkGroupID` : index du groupe,
- `gl_WorkGroupSize` : nombre de threads par groupe,
- `gl_GlobalInvocationID= gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID`

comment choisir la tache réalisée par un thread ?

- c'est le programmeur qui décide...
- très souple,
- mais...

En Résumé

compute shaders :

- GLSL, langage sans (trop de) surprises,
- attention à la décomposition / association tache / thread,
- utilisation cohérente de `glDispatchCompute()` et de `layout(local_size= )`,
- espace d'iteration :
  "boucle externe" remplacée par ensemble de threads.

les autres API :

mêmes concepts, mais emballage différent.

shader : copier une image

```
#version 430                        // compute shader, core profile
#ifdef COMPUTE_SHADER               // impose par computeKit

uniform sampler2D image0;           // texture en entree

layout( binding= 0, rgba8 ) writeonly
uniform image2D resultat;           // image en sortie

// configuration du nombre de threads par groupe
layout( local_size_x= 16, local_size_y= 16 ) in;
void main( )
{
    // tache / coordonnees du pixel associe au thread
    ivec2 pixel= ivec2(gl_GlobalInvocationID.xy);

    // lire un pixel dans la texture
    vec3 couleur= texelFetch(image0, pixel, 0).rgb;

    // eventuellement modifier couleur
    {...}

    // ecrire le pixel dans l'image resultat
    imageStore(resultat, pixel, vec4(couleur, 1));
}
#endif
```