

M2-Images

Rendu Temps Réel - OpenGL 4 et compute shaders

J.C. lehl

December 18, 2013

résumé des épisodes précédents...

- ▶ création des objets opengl,
- ▶ organisation des données,
- ▶ configuration du pipeline,
- ▶ draw,
- ▶ ...

opengl 4.3 : nouveau type de shader : compute shader

Compute shaders

c'est quoi :

- ▶ un shader qui ne s'exécute pas dans le pipeline graphique,
- ▶ pas de vertex buffer, pas de framebuffer, ...
- ▶ mais des uniforms, des buffers, des textures, des images
- ▶ en entrée...
- ▶ et en sortie !

Compute shaders

à quoi ça sert :

- ▶ à faire autre chose que de l'affichage ?
- ▶ de la simulation ?
- ▶ du culling/élimination des parties cachées ?
- ▶ à aller plus vite ?

cf programmation générique *parallèle*
des processeurs "graphiques"


Création et exécution

création :

- ▶ `glCreateShader(GL_COMPUTE_SHADER);`
- ▶ seul shader présent dans un programme,
- ▶ le reste est classique :
`glCreateProgram(), glAttachShader(), glLinkProgram()`

draw ?

- ▶ non, le compute shader ne dessine rien, il s'exécute pour traiter des données :
- ▶ `glDispatchCompute(...);`

mais on peut produire une image / texture... 

ehh, et les résultats ?

les resultats :

- ▶ sont stockés dans des objets opengl classiques :
- ▶ des buffers ou des textures selon le cas.

et les entrées sont des uniform, des buffers et des textures.

Parallélisme de données

parallélisme de données :

- ▶ exécuter une tâche / un thread par donnée à traiter,
- ▶ au lieu d'écrire une boucle pour traiter chaque donnée.

quoi ?

- ▶ même chose que les autres types de shaders :
- ▶ transformer tous les sommets, calculer la couleur de tous les fragments, etc.
- ▶ mais les différents threads peuvent communiquer !

exemple

```
#version 430    // core profile, compute shader

uniform mat4 mvpMatrix;

layout( binding= 0 )    // buffer 0: contient un tableau de position
readonly buffer positionData // nom du buffer pour l'application,
// cf glGetProgramResourceIndex et glBindBufferBase()
{
    vec3 position[];
};

layout( binding= 1 )    // buffer 1: tableau de positions transformees
writeonly buffer transformedData
{
    vec4 transformed[];
};

layout( local_size_x= 32 ) in;
void main( )
{
    int i= gl_GlobalInvocationID;
    transformed[i]= mvpMatrix * vec4(position[i], 1.0);
}
```


Espace d'itération

en pratique :

- ▶ les threads sont exécutés par groupes,
- ▶ l'espace d'itération est découpé en groupes, et chaque groupe est composé de plusieurs threads :
- ▶ `glDispatchCompute(num_groups); // opengl`
- ▶ `layout(local_size= 32) in; // glsl`

pourquoi ?

- ▶ ressources partagées par les threads d'un groupe :
- ▶ mémoire...
- ▶ processeur : unités de calculs, instructions, cache, etc.

Espace d'itération et indexation des threads

limites :

- ▶ nombre max de groupes :
`glGet(GL_MAX_COMPUTE_WORK_GROUP_COUNT),`
- ▶ nombre max de threads par groupe :
`glGet(GL_MAX_COMPUTE_WORK_GROUP_SIZE),`
- ▶ mémoire partagée accessible aux threads d'un groupe :
`glGet(GL_MAX_COMPUTE_SHARED_MEMORY_SIZE)`

Espace d'itération

exemple :

- ▶ transformer les sommets de bigguy : 1754 positions,
- ▶ groupes de 32 threads (arbitraire),
- ▶ donc $1754 / 32 (+1)$ groupes nécessaires.

Espace d'itération et indexation des threads

indexation des threads :

- ▶ `gl_GlobalInvocationID` : indice du thread dans l'ensemble,
$$== \text{gl_WorkGroupID} * \text{gl_WorkGroupSize} + \text{gl_LocalInvocationID},$$
 - ▶ `gl_WorkGroupID` : indice du groupe,
 - ▶ `gl_LocalInvocationID` : indice du thread dans le groupe,
 - ▶ `gl_WorkGroupSize` : nombre de threads dans le groupe.
-
- ▶ et l'espace d'indexation est 3d...
 - ▶ plus simple pour travailler sur une image 2d ou une grille 3d...

Exécution parallèle des threads

tous les threads fonctionnent en même temps :

- ▶ dans un ordre quelconque choisit par l'ordonnanceur des processeurs graphiques,
- ▶ partager des résultats intermediaires ?
- ▶ comment s'assurer que les résultats sont disponibles ?
- ▶ synchronisation !
- ▶ `barrier()` et `groupMemoryBarrier()`
- ▶ déclarer une variable partagée par les thread du groupe :
- ▶ `shared type variable;`

Exemple

```
#version 430    // core profile, compute shader

layout( binding= 0 )    // buffer 0: contient un tableau de position
readonly buffer positionData { vec3 positions[]; };

layout( binding= 1 )    // buffer 1: tableau de positions transformees
writeonly buffer transformedData { vec3 transformed[]; };

shared vec3 edges[3];

layout( local_size_x= 3 ) in;
void main( )
{
    int i= gl_GlobalInvocationID.x;
    int e= gl_LocalInvocationID.x;

    edges[e]= positions[i];
    barrier(); // edges[0..3] contient les sommets d'un triangle

    edges[e]= edges[(e+1) % 3] - edges[e];
    barrier(); // edges[0..3] contient l'arete (e, e+1)

    transformed[i]= edges[e];
}
```

Synchronisation

opérations atomiques :

- ▶ chaque thread ajoute 1 à une variable...
- ▶ quel est le "bon" résultat ?

problème ?

- ▶ ajouter 1 à une variable :
- ▶ == 3 opérations,
- ▶ lire la valeur,
- ▶ ajouter 1,
- ▶ écrire la nouvelle valeur dans la variable...

Synchronisation

solution :

- ▶ opérations atomiques,
- ▶ `atomicAdd(variable, 1)`, etc.

autres opérations :

- ▶ `atomicCompSwap()`,
- ▶ `atomicExchange()`,
- ▶ permettent de créer un type particulier de mutex, un *spinlock*.

rappel du cours de système : les spinlocks et l'attente active, c'est le *mal*.

Exemple

modifier plusieurs valeurs :

- ▶ nécessite plusieurs opérations,
- ▶ et il n'existe pas d'opérations "atomiques" pour ce cas...

```
if( fragment_z < zbuffer )  
{  
    cbuffer= fragment_color  
    zbuffer= fragment_z  
}
```

Exemple

```
int lock;  
  
while(lock != 0) {} // attendre  
lock= 1;           // bloquer : DOIT etre atomique avec le test  
  
    // modifier les valeurs  
    if( fragment_z < zbuffer )  
    {  
        cbuffer= fragment_color  
        zbuffer= fragment_z  
    }  
  
lock= 0;           // relacher
```

Exemple

```
int lock;  
  
while(atomicCompSwap(lock, 0, 1) != 0) {} // attendre  
  
    // modifier les valeurs  
    if( fragment_z < zbuffer )  
    {  
        cbuffer= fragment_color  
        zbuffer= fragment_z  
    }  
  
atomicExchange(lock, 0);           // relacher
```

Shader Storage Buffers

buffers en entrée et en sortie :

- ▶ déclarés par le shader :
- ▶ `layout(binding = x) readonly buffer nom { };`
- ▶ `layout(binding = x) writeonly buffer nom { };`

`layout(binding = x)` indique l'indice du buffer :

`glBindBufferBase(GL_SHADER_STORAGE_BUFFER, x, buffer);`

Shader Storage Buffers

quel contenu ?

- ▶ a priori un tableau de valeurs,
- ▶ éventuellement un tableau de structures,

mais :

- ▶ le cpu et les gpu ont des organisations mémoire différentes :
- ▶ un `vec3` est représenté par 3 floats sur cpu,
- ▶ aussi représenté par 3 floats sur gpu, mais aligné sur des multiples de 4...
- ▶ "convertir" les données pour les rendre accessibles au gpu.

Shader Storage Buffers

alignement des données :

- ▶ cf **OpenGL specification**, section 7.6.2.2 "Standard Uniform Block Layout"
- 1. If the member is a scalar consuming N basic machine units, the base alignment is N ,
- 2. If the member is a two- or four-component vector with components consuming N basic machine units, the base alignment is $2N$ or $4N$, respectively.
- 3. If the member is a three-component vector with components consuming N basic machine units, the base alignment is $4N$.

Shader Storage Buffers

alignement des données :

- ▶ If the member is an array of scalars or vectors, the base alignment and array stride are set to match the base alignment of a single array element, according to rules (1), (2), and (3), and rounded up to the base alignment of a vec4. array may have padding at the end; the base offset of the member following the array is rounded up to the next multiple of the base alignment.

Shader Storage Buffers

alignement des données :

- ▶ If the member is a structure, the base alignment of the structure is N , where N is the largest base alignment value of any of its members, and rounded up to the base alignment of a `vec4`. The individual members of this sub-structure are then assigned offsets by applying this set of rules recursively, where the base offset of the first member of the sub-structure is equal to the aligned offset of the structure. The structure may have padding at the end; the base offset of the member following the sub-structure is rounded up to the next multiple of the base alignment of the structure.

Shader Storage Buffer

alignement des données :

- ▶ 2 types d'alignement : std140 ou std430
- ▶ std430 est réalisable en C/C++ sur cpu,
- ▶ avec `__attribute__((aligned (x)))`;
- ▶ ou `gk::glsl::scalar`, `gk::glsl::vec2`, `vec3`, `vec4`, `mat2`, `mat3`, `mat4`, etc.
- ▶ ou ne pas utiliser de matrices / vecteurs à 3 composantes...

Exemple

```
#version 430    // core profile, compute shader

struct triangle
{
    vec3 p;
    vec3 edge1;
    vec3 edge2;
};

layout( std430, binding= 0 ) readonly buffer triangleData
{
    triangle triangles[];
};
```

Exemple

```
#include "GL/GLSLUniforms.h"

struct Triangle
{
    gk::glsl::vec3 p;
    gk::glsl::vec3 edge1, edge2;

    Triangle( const gk::Vec3& a, const gk::Vec3& b, const gk::Vec3& c )
        :
        p(a),
        edge1(gk::Point(b) - gk::Point(a)),
        edge2(gk::Point(c) - gk::Point(a)) {}
};

std::vector<Triangle> triangles;
for(int i= 0; i < mesh->triangleCount(); i++)
{
    const gk::Triangle& t= mesh->triangle(i);
    triangles.push_back( Triangle(t.a, t.b, t.c) );
}

gk::GLBuffer *input=
    gk::createBuffer(GL_SHADER_STORAGE_BUFFER, triangles);
```

Shader Storage Buffer

vérifications ?

- ▶ `glGetProgramResource()`,
- ▶ et les propriétés `GL_OFFSET`, `GL_TYPE`,
`GL_TOP_LEVEL_ARRAY_STRIDE`.

cf. l'affichage de `gKit` après la compilation d'un shader :

```
program 'triangle.glsl'...
  compute shader...
  uniform 'mvpInvMatrix' location 2, index 2, size 1, type 'mat4'
  buffer 'triangleData' index 0
    'triangles[0].color' offset 48 type 'vec3', ... top level stride 64
    'triangles[0].edge1' offset 16 type 'vec3', ... top level stride 64
    'triangles[0].edge2' offset 32 type 'vec3', ... top level stride 64
    'triangles[0].p' offset 0 type 'vec3', ... top level stride 64
done.
```

Texture Images

Textures en entrée et en sortie :

- ▶ textures en entrée : comme dans les autres shaders :
`uniform sampler2D texture;`
- ▶ textures en sortie : `uniform image2D image;`

pour l'application :

- ▶ `glActiveTexture()` + `glBindTexture()` pour les entrées,
- ▶ `glBindImageTexture(index, texture, ...)`
pour les sorties.

Texture Images

mais :

- ▶ accès à un seul niveau de mipmap (cf `glBindImage()`),
- ▶ pas de filtrage,
- ▶ quelques contraintes de format :
- ▶ déclarer le type des canaux : `int`, `float`, etc.
- ▶ `layout(rgba8)`, `layout(rgba32f)`,
- ▶ et utiliser le bon type d'image :
`image2D`, `iimage2D`, `uimage2D`, etc.

Exemple

```
#version 430 // fragment shader

layout(binding= 0, r32ui) uniform uimage2D overdraw;    // uint32

in vec3 vertex_normal;
out vec4 fragment_color;

void main( )
{
    imageAtomicAdd(overdraw, ivec2(gl_FragCoord.xy), 1u);

    fragment_color= vec4(abs(vertex_normal), 1.0);
}

// application
gk::GLtexture *image=
    gk::createTexture2D(gk::UNIT0, w, h, gk::TextureR32UI);
glBindTexture(GL_TEXTURE_2D, 0);

// utilisation
glBindImageTexture(0, image->name, 0, GL_FALSE, 0,
    GL_READ_WRITE, GL_R32UI);
```

Exemple

```
#version 430 // compute shader
layout( binding= 0, rgba8 ) writeonly uniform image2D framebuffer;

layout( local_size_x= 16, local_size_y= 16 ) in;
void main( )
{
    // rayon associe au pixel (local.x, local.y) de la tuile
    vec4 oh= mvpvInvMatrix * vec4(gl_GlobalInvocationID.xy, -1, 1);
    vec4 eh= mvpvInvMatrix * vec4(gl_GlobalInvocationID.xy, 1, 1);
    struct ray r;
    r.o= oh.xyz / oh.w;
    r.d= eh.xyz / eh.w - oh.xyz / oh.w;

    // intersections avec les triangles
    float t;
    float h= 1.0;
    vec4 color;
    for(int i= 0; i < triangles.length(); i++)
        if(intersect(triangles[i], r, h, t))
        {
            h= t;
            color= vec4(triangles[i].color, 1.0);
        }

    imageStore(framebuffer, ivec2(gl_GlobalInvocationID.xy), color);
}
```


Exemple

```
// application
gk::GLTexture *output=
    gk::createTexture2D(gk::UNIT0, w, h, gk::TextureRGBA);
glBindTexture(GL_TEXTURE_2D, 0);

// utilisation
glBindImageTexture(0, output->name, 0, GL_FALSE, 0,
    GL_WRITE_ONLY, GL_RGBA8);
```