

M2-Images

Rendu Temps Réel - OpenGL 3 et textures

J.C. Iehl

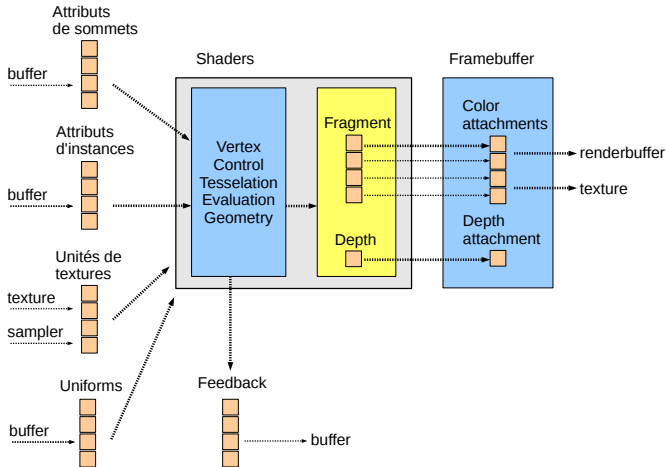
January 20, 2011

Résumé des épisodes précédents

résumé :

- ▶ création de buffers,
- ▶ création de maillages indexés ou non,
- ▶ affichage de maillages,
- ▶ affichage de plusieurs maillages,
- ▶ vertex et fragment shaders,
- ▶ ??

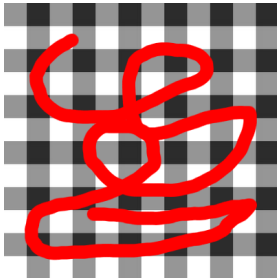
Résumé de l'api opengl 3



Textures

idée :

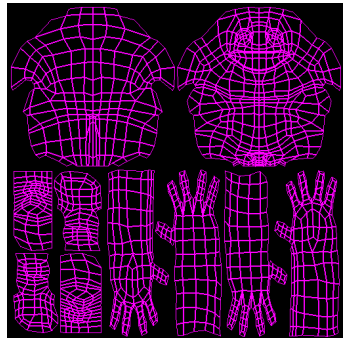
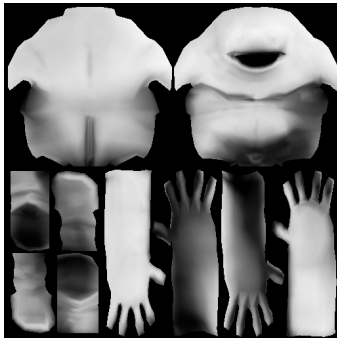
- ▶ envelopper une image autour d'un objet,
- ▶ modifier la couleur des points de la surface de l'objet.



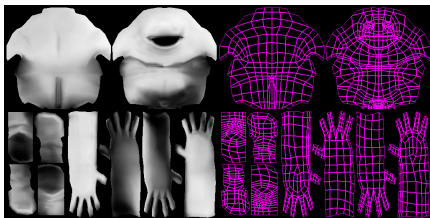
Textures : paramétrisation de la surface

décrire l'association image / primitive :

- ▶ donner la correspondance entre les sommets de l'objet,
- ▶ et les pixels de la texture (des *texels*).



Textures : paramétrisation de la surface



Textures : paramétrisation de la surface

comment ?

- ▶ en théorie, on peut le calculer automatiquement,
- ▶ en pratique : c'est un graphiste qui le fait et qui peint la texture.

utilisation avec OpenGL :

- ▶ nouvel attribut associé aux sommets :
les coordonnées de texture,
- ▶ + décrire le contenu de la texture.

Textures : utilisation OpenGL

coordonnées de texture :

- ▶ un attribut de sommet supplémentaire,
- ▶ à déclarer dans le vertex shader :
`attribute vec2 texcoord;`

contenu de la texture :

- ▶ créer un identifiant, `glGenTextures()`,
- ▶ décrire le type de texture (1d, 2d, 3d), le type des texels (rgb, rgba, etc.), `glTexImageXXD()`,
- ▶ + transférer les données.

Textures : utilisation OpenGL

```
// pixels contigus en memoire
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

GLuint name;
glGenTextures(1, &name);
glBindTexture(GL_TEXTURE_2D, name);

glTexImage2D(GL_TEXTURE_2D,
//      format interne
             0, GL_RGB, width, height, 0,
//      format des donnees
             data_format, data_type, data);
```

Textures : utilisation OpenGL

”bouclage” de la paramétrisation :

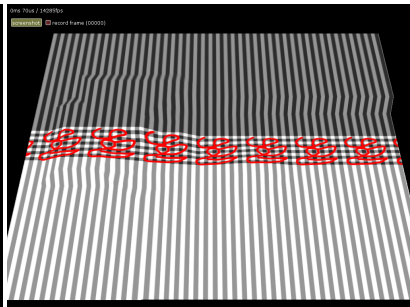
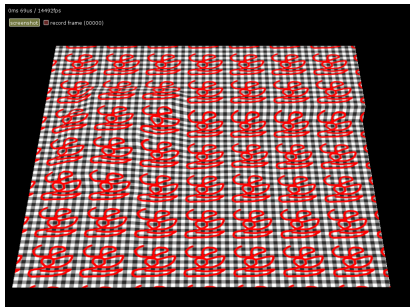
- ▶ normale, sans répétition : `GL_CLAMP`,
- ▶ répétition : `GL_REPEAT`,
- ▶ (répétition inversée : `GL_MIRRORED_REPEAT`).

paramètres sur chaque axe de la texture :

- ▶ (s, t, r, q) pour OpenGL,
- ▶ `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`,
`GL_TEXTURE_WRAP_R`,
- ▶ (s, t, p, q) pour les shaders ...

Textures : utilisation OpenGL

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);  
// ou GL_REPEAT, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER
```



Textures : quelques précisions

plusieurs types de textures :

- ▶ 1d, 2d, 3d,
- ▶ cube (6 faces),
- ▶ tableau (1DArray, 2DArray).

utiliser :

- ▶ `GL_TEXTURE_1D`, `GL_TEXTURE_1D_ARRAY`,
- ▶ `GL_TEXTURE_2D`, `GL_TEXTURE_2D_ARRAY`,
- ▶ `GL_TEXTURE_3D`,
- ▶ `GL_TEXTURE_CUBE`,
- ▶ (`GL_TEXTURE_RECTANGLE`).

Textures : quelques précisions

plusieurs types de texels :

- ▶ 1, 2, 3 ou 4 *canaux*,
- ▶ GL_RED, GL_RG, GL_RGB, GL_RGBA

plusieurs précisions :

- ▶ entier 8bits,
- ▶ réel 16bits, réel 32bits,
- ▶ profondeur 16, 24, ou 32 bits.
- ▶ GL_RGB8, GL_RGB16F, GL_RGB32F.

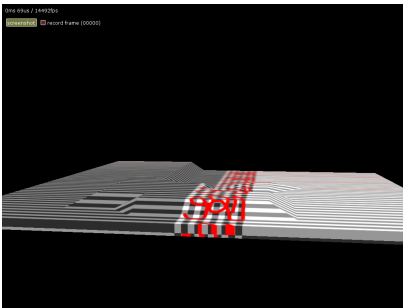
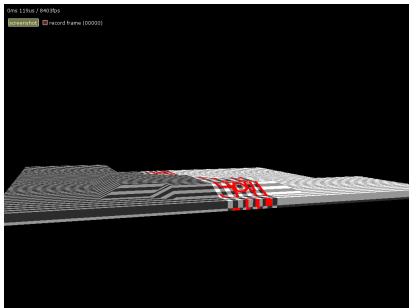
Samplers

filtrer :

- ▶ selon la distance et l'orientation de l'objet,
- ▶ un pixel de l'image correspond à :
- ▶ moins de 1 texel,
- ▶ ou plusieurs texels.

choisir le texel le plus proche `GL_NEAREST`, ou filtrer `GL_LINEAR`.

Filtrage



Samplers : pré-filtrage et mipmaps

pré-filtrage :

- ▶ lorsqu'un pixel correspond à grand nombre de texels,
- ▶ filtrer devient très couteux ...
- ▶ pré-filtrer la texture.

mipmaps :

- ▶ plusieurs "versions" de la texture,
- ▶ à des résolutions *filtrées* différentes,

choisir le texel *pré-filtré* le plus proche :
GL_NEAREST_MIPMAP_NEAREST, ou filtrer
GL_LINEAR_MIPMAP_LINEAR.

Textures pré-filtrées

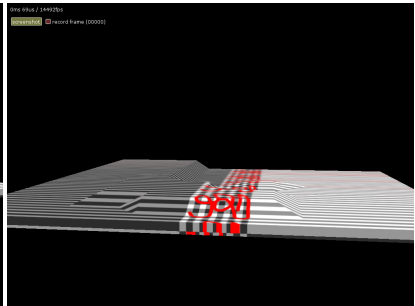
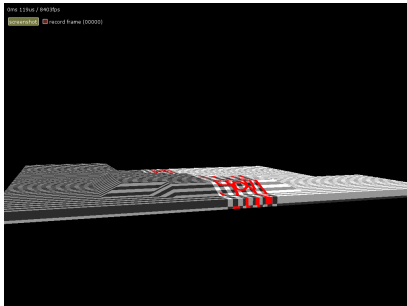
textures pré-filtrées :

- ▶ les *niveaux*, les versions pré-filtrées de la texture, sont fournis par l'application :
`glTexImageXXXD(level),`
- ▶ les niveaux sont calculés par OpenGL :
`glGenerateMipMap().`

- ▶ erreur courante : OpenGL suppose, par défaut, que les textures sont pré-filtrées.

Samplers : pré-filtrage

```
// choix du mode de filtrage  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
// creation des versions pre-filtrees de la texture  
glGenerateMipMap(GL_TEXTURE_2D);
```



Samplers : création

samplers :

- ▶ regroupent tous les paramètres de filtrage d'une texture :
- ▶ `glGenSamplers()`,
- ▶ `glBindSampler()`,
- ▶ `glSamplerParameter()`, mêmes paramètres que `glTexParameter()`

Plusieurs textures

utilisation de plusieurs textures :

- ▶ créer et paramétrer les textures (+ samplers),
- ▶ activer une unité de texture : `glActiveTexture()`,
- ▶ activer la texture sur l'unité : `glBindTexture()`,
- ▶ (activer le sampler sur l'unité : `glBindSampler()`)
- ▶ paramétrer le shader pour accéder à l'unité de texture : `glUniform1i()`.

attention :

- ▶ l'unité de texture est identifiée par son indice,
- ▶ ou une constante `GL_TEXTURE0` + indice.

Textures et Samplers : utilisation par un shader

utilisation par un shader :

- ▶ déclarer un sampler (en fonction du type de la texture),
- ▶ lire un texel dans la texture.

```
// opengl 3.3 core profile, fragment shader :  
#version 330  
  
uniform sampler2D color;  
in vec2 texcoord;  
  
void main( void )  
{  
    gl_FragColor= texture(color, texcoord);  
}  
  
// dans l'application :  
glUniform1i( glGetUniformLocation("color"), 0 );  
// utiliser la texture active sur l'unité 0
```

Textures : résumé

résumé :

- ▶ paramétrisation de la surface d'un objet,
- ▶ création, configuration d'une texture,
- ▶ (création, configuration d'un sampler),
- ▶ activation d'une texture (+ sampler) sur une unité de texture,
- ▶ déclaration et utilisation dans un shader,
- ▶ configurer le shader avec l'unité de texture,
- ▶ utiliser simultanément plusieurs textures :
configurer plusieurs unités de textures.

Framebuffer

à quoi ça sert ?

- ▶ à réaliser un traitement trop complexe pour le pipeline OpenGL,
- ▶ découper le traitement en plusieurs étapes (traitables par le pipeline),
- ▶ + stocker résultats intermédiaires.

Framebuffer

idée :

- ▶ même principe que les unités de textures,
- ▶ mais appliqué aux sorties, au lieu des entrées des shaders.

plusieurs textures de sortie :

- ▶ regroupées dans un *framebuffer*,
- ▶ créer un framebuffer,
- ▶ créer des textures,
- ▶ associer les textures aux *draw buffers* du *framebuffer*.

Framebuffer : création

création :

- ▶ `glGenFramebuffers()`,
- ▶ `glBindFramebuffer()`.

plusieurs sorties :

- ▶ identifiées par les constantes `GL_COLOR_ATTACHMENT0` + indice, pour les textures couleurs,
- ▶ ou `GL_DEPTH_ATTACHMENT`, pour une texture de profondeur (type `GL_DEPTH_COMPONENT`).

Framebuffer : utilisation

activer une texture sur un draw buffer :

- ▶ `glFramebufferTextureXXX()`,
- ▶ `(glFramebufferRenderbuffer())`.

activer un ou plusieurs draw buffers :

- ▶ `glDrawBuffers()`.

Framebuffer : exemple

```
GLint texture; // texture 2d couleur
{ ... }
GLint depth; // texture 2d profondeur
{ ... }

GLint framebuffer;
glGenFramebuffers(1, &framebuffer);
// activer le framebuffer
glBindFramebuffer(GL_FRAMEBUFFER);
// associer la texture couleur au draw buffer 0 du framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
    GL_TEXTURE_2D, texture, 0);
// associer la texture profondeur au depth buffer du framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_TEXTURE_2D, depth, 0);

// activer les draw buffers
GLenum drawbuffers[] = { GL_COLOR_ATTACHMENT0, GL_DEPTH_ATTACHMENT };
glDrawBuffers(2, drawbuffers);

// activer le shader
// dessiner
```

Framebuffer : utilisation par un shader

utilisation par un shader :

- ▶ déclarer un varying par sortie du fragment shader :
`out vec4 color;`
`out vec3 normale;`
- ▶ associer les varying aux draw buffers avant de linker le shader program : `glBindFragDataLocation()`
exercice bonus : traduire la phrase en français !
- ▶ ou :
- ▶ déclarer l'association au draw buffer dans le fragment shader :
`layout(location= 0) out vec4 color;`
`layout(location= 1) out vec3 normale;`