Chapitre 8: Shaders

INF5071 — Infographie

Alexandre Blondin Massé

Université du Québec à Montréal

Hiver 2019

Plan

1. Introduction
2. Le langage GLSL
3. Shaders 2D
4. Shaders 3D

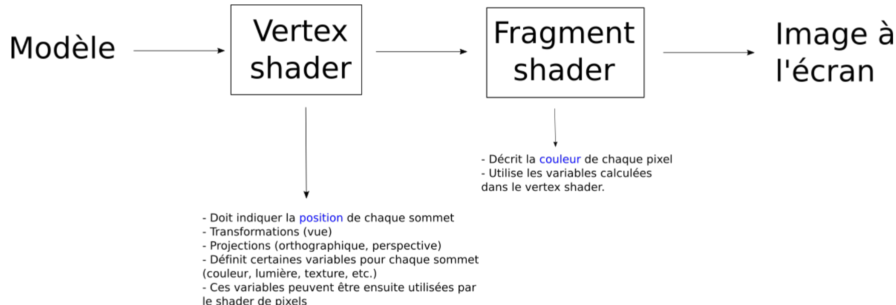
Introduction

Qu'est-ce que c'est?

- Les **shaders** sont des **programmes** qui indiquent à votre carte graphique comment **dessiner** chacun des **pixels**
- On distingue **deux types** de **shaders** :
- Les shaders de **sommets** (*vertex shader*)
- Les shaders de **pixels** (*fragment shader* ou *pixel shader*)
- En réalité, il y en a d'autres (*tesselation*, *geometry*), mais nous n'allons pas les aborder dans le cours

Ordre de traitement

- On traite d'abord les **sommets** (*vertex shader*);
- Puis ensuite, on traite les autres **pixels** (*fragment shader*).



Les shaders de sommets

- **Minimalement**, un shader de sommet (**vertex shader**) a les caractéristiques suivantes:

$\rightarrow$ **Entrée** : la *position* du sommet courant

$\rightarrow$ **Sortie** : la *position* finale du sommet

- On peut également **ajouter** des informations à chaque sommet qui seront utilisées par le **shader de pixels**:

$\rightarrow$ couleur

$\rightarrow$ coordonnée de texture

$\rightarrow$ vecteurs normaux

$\rightarrow$ etc.

Les shaders de pixels

- **Minimalement**, un shader de pixels (**fragment shader**) a les caractéristiques suivantes :

$\rightarrow$ **Entrée** : la *position* du pixel courant

$\rightarrow$ **Sortie** : la *couleur* du pixel courant

- On peut utiliser certaines informations **propres à chaque sommet** qui auront été ajoutées par le **shader de sommet**

Les variables de shaders

- Lorsqu'on conçoit des shaders, il faut considérer les **trois éléments** suivants :

→ Quelle information doit être contenue dans chaque **sommet** ?

→ Quelle information est **constante**, peu importe le sommet ?

→ Quelle information doit être **ajoutées** pour chaque sommet ?

- Les **shaders** communiquent **entre eux** et **avec l'application** en utilisant trois types de variables :

→ *uniform*

→ *attribute*

→ *varying*

Le type `uniform`

Les variables de type `uniform` sont celles qui sont **constantes** pour

- le **shader de sommets** et
- le **shader de pixels**

Quelques exemples

- La **position** d'une ou plusieurs sources lumineuses
- Une **couleur** uniforme pour un objet
- L'**intensité lumineuse** ambiante
- Une **position** importante (celle du joueur, du centre de la scène), etc.

Le type `attribute`

- Les variables de type `attribute` sont des variables qui concernent les **sommets** seulement
- Plusieurs sont fournis **par défaut**:

$\rightarrow$ La **position** du sommet

$\rightarrow$ Sa **couleur**

$\rightarrow$ Son **vecteur normal**

$\rightarrow$ Des **coordonnées de texture**, etc.

- Il est également possible d'ajouter des attributs **personnalisés**, mais ce n'est pas toujours supporté

Le type `varying`

- C'est à l'aide de ces variables que les **shaders de sommets** et de **pixels** interagissent
- On les déclare dans les **deux shaders**
- Dans le **shader de sommets**, on écrit une certaine valeur
- Dans le **shader de pixels**, cette valeur est **interpolée** à partir des sommets qui constituent le **fragment**
- Quelques exemples:

→ La **couleur** du pixel courant

→ La **coordonnée de texture** du pixel courant

→ L'**intensité lumineuse**, etc.

Le langage GLSL

Qu'est-ce que c'est?

- Le langage **GLSL** est un des langages utilisés pour écrire des **shaders**
- **GLSL** = *OpenGL Shading Language*
- Il s'agit d'un langage similaire à **C**
- Il fournit certains **types** par défaut ainsi que de nombreuses **fonctions**

Shader de vertex

```
uniform mat4 gl_ModelViewMatrix;  // Matrice vue-modèle
uniform mat4 gl_ProjectionMatrix; // Matrice de projection
// Variables prédéfinies :
// gl_Vertex (entrée)   : position du sommet courant
// gl_Position (sortie) : position finale du sommet
void main() {
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

- Chaque sommet est **transformé** selon les matrices modèle-vue et projective
- Le nom des **matrices** varie selon le contexte (OpenGL, three.js, etc.)
- On peut remplacer la ligne

```
gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
```

par

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

Shader de pixels

```
// Variables prédéfinies :
// gl_FragCoord (entrée) : position du fragment courant
// gl_FragColor (sortie) : couleur du fragment
void main() {
    // On n'utilise pas la position
    gl_FragColor = vec4(0.4, 0.4, 0.8, 1.0);
}
```

- On précise la couleur dans le système **RGBA**.

Types de données

- **Scalaires:** `bool`, `int`, `float`, `double`
- **Vecteurs:** `bvec2`, `ivec2`, `vec2`, `dvec2`, `bvec3`, `ivec3`, `vec3`, `dvec3`, `bvec4`, `ivec4`, `vec4`, `dvec4`
- **Matrices:** `mat2`, `mat3`, `mat4`, `mat2x3`, `mat2x4`, `mat3x4`

On peut également définir des **tableaux**, des **structures**, etc. comme en C, mais il y a certaines **limites**

Conversions et constructions

- Il est possible de passer d'un type à l'autre de façon **flexible**
- Les **composantes** des vecteurs sont accessibles via les attributs `xyzw`, `rgba` (*couleurs*) ou `stpq` (*textures*)
- On peut même construire des **vecteurs** ! Par exemple, si `v` est de type `vec4`, alors `v.xyx` est de type `vec3` avec composantes (x, y, x)
- On peut passer d'une **dimension** vectorielle à une autre facilement :

```
// Crée le vecteur (1, 1, 1)
vec3 v1 = vec3(1.0);
// Crée le vecteur (1, 2, 3)
vec4 v2 = vec3(vec2(1.0, 2.0), 3.0);
// Si v1, v2, v3, v4 sont de type vec4
mat4 m1 = mat4(v1, v2, v3, v4);
```

Fonctions standards

Le langage GLSL offre de nombreuses fonctions **mathématiques** par défaut

- Les fonctions `min` et `max`
- Les fonctions **trigonométriques**
- Des fonctions logiques `any` et `all`
- Une fonction d'interpolation `mix`
- Les fonctions `ceil` et `floor`
- Les produits **scalaire** et **vectoriel**
- Les fonctions **modulo** (`mod`) et **partie fractionnaire** (`frac`)
- La **normalisation** de vecteurs (`normalize`)
- Des fonctions de **bruit** (`noise`), etc.

Shaders 2D

Shaders 2D

- Familiarisons-nous d'abord avec les **shaders** de pixels en 2D
- Dans ce cas, on s'intéresse principalement aux *fragment shaders*
- Un outil est intéressant pour effectuer différents tests se trouve à https://www.shadertoy.com

Grille

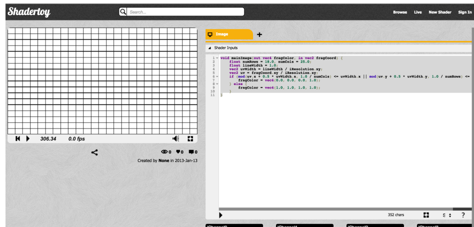
- Comment peut-on dessiner une **grille** de ℓ ligne et de c colonnes
- L'arrière-plan sera **blanc** et les lignes de la grille seront **noires**
- Nous allons utiliser les coordonnées d'écran **normalisées** (c'est-à-dire entre 0 et 1)
- On doit se fixer une certaine **épaisseur** E de ligne et E' l'épaisseur après **normalisation**
- Alors un pixel (x, y) dans le système **normalisé** sera dessiné en

$\rightarrow$ **noir** si $x \bmod C \leq E'.x$ **ou** $y \bmod L \leq E'.y$, où $C = 1.0/c$ et $L = 1.0/\ell$ et

$\rightarrow$ **blanc** sinon

Code

```
float NUM_COLS = 25.0;
float NUM_ROWS = 18.0;
float LINE_WIDTH = 1.0;

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 uvWidth = LINE_WIDTH / iResolution.xy;
    vec2 uv = fragCoord.xy / iResolution.xy;
    if (mod(uv.x + 0.5 * uvWidth.x, 1.0 / NUM_COLS) <= uvWidth.x ||
        mod(uv.y + 0.5 * uvWidth.y, 1.0 / NUM_ROWS) <= uvWidth.y) {
        fragColor = vec4(0.0, 0.0, 0.0, 1.0);
    } else {
        fragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
}
```



Calcul vectoriel

- De nombreux bouts de code peuvent être **compactés** si on utilise le **calcul vectoriel**
- Par exemple, les lignes

```
float NUM_COLS = 25.0;
float NUM_ROWS = 18.0;

if (mod(uv.x + 0.5 * uvWidth.x, 1.0 / numCols) <= uvWidth.x ||
    mod(uv.y + 0.5 * uvWidth.y, 1.0 / numRows) <= uvWidth.y)
```

peuvent être remplacées par

```
vec2 DIMENSIONS = vec2(25.0, 18.0);

if (any(lessThan(mod(uv + 0.5 * uvWidth, 1.0 / dimensions), uvWidth)))
```

Dégradé radial

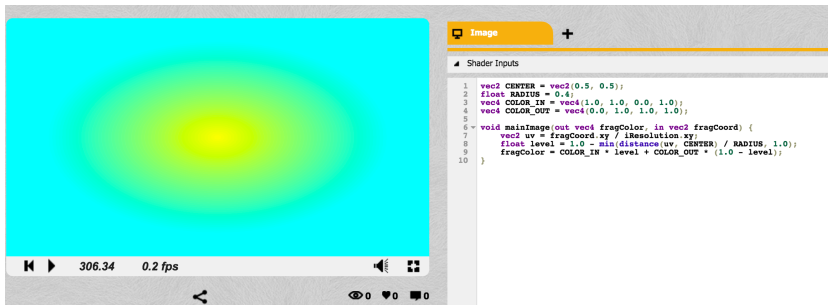
- Étudions la façon de réaliser un **dégradé radial**
- Les **paramètres** à considérer sont les suivants:

→ Le **centre** du cercle (type `vec2`)
→ Le **rayon** du cercle (type `float`)
→ La **couleur intérieure** (type `vec4`)
→ La **couleur extérieure** (type `vec4`)

- Il suffit ensuite de calculer la **distance** entre le **pixel courant** et le **centre** du cercle (fonction `distance` de GLSL)

Code

```
vec2 CENTER = vec2(0.5, 0.5);
float RADIUS = 0.4;
vec4 COLOR_IN = vec4(1.0, 1.0, 0.0, 1.0);
vec4 COLOR_OUT = vec4(0.0, 1.0, 1.0, 1.0);

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 uv = fragCoord.xy / iResolution.xy;
    float level = 1.0 - min(distance(uv, CENTER) / RADIUS, 1.0);
    fragColor = COLOR_IN * level + COLOR_OUT * (1.0 - level);
}
```



La fonction `mix`

- L'interpolation **linéaire** est tellement fréquente qu'il existe une **fonction de GLSL** qui la calcule
- On peut donc remplacer la ligne

```
fragColor = COLOR_IN * level + COLOR_OUT * (1.0 - level);
```

par

```
fragColor = mix(COLOR_OUT, COLOR_IN, level);
```

Exercice

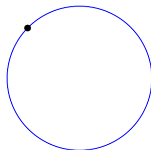
- Écrivez un petit programme GLSL qui anime le déplacement d'un point noir sur un cercle bleu
- Vous devez laisser la possibilité de paramétrer les quantités suivantes:

→ Le **centre** et le **rayon** du cercle

→ La **vitesse angulaire** (en pixel par seconde)

→ La **taille** du point

- La variable `iTime` contient le temps présent en **secondes**.



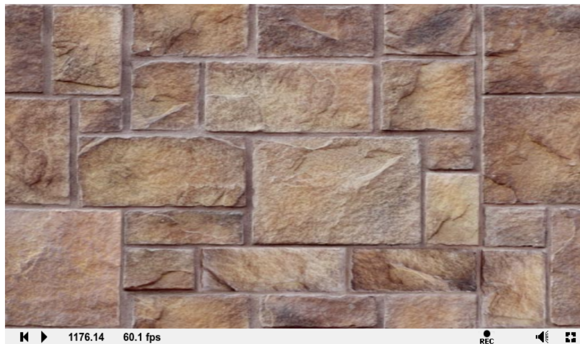
Solution

```
const vec4 BLUE = vec4(0., 0., 1., 1.);
const vec4 WHITE = vec4(1., 1., 1., 1.);
const float R = 150.;
const float W = 2.;
const float P_R = 7.;

// Indicates if p lies on the ring centered in c with radii r1 and r2
float onRing(vec2 p, vec2 c, float r1, float r2) {
    float d = distance(p, c);
    return smoothstep(r1 - 0.5, r1 + 0.5, d)
         * (1. - smoothstep(r2 - 0.5, r2 + 0.5, d));
}

// Indicates if p lies on the disk centered in c of radius r
float onPoint(vec2 p, vec2 c, float r) {
    float d = distance(p, c);
    return 1. - smoothstep(r - 0.5, r + 0.5, d);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 center = iResolution.xy * 0.5;
    float ring = onRing(fragCoord, center, R - 0.5 * W, R + 0.5 * W);
    float a = 2. * iTime;
    vec2 p_center = R * vec2(cos(a), sin(a)) + center;
    float point = onPoint(fragCoord, p_center, P_R);
    fragColor = min(mix(WHITE, BLUE, ring), 1. - point);
}
```

Textures

À l'aide de la fonction `texture`:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 uv = fragCoord / iResolution.xy;
    fragColor = texture(iChannel0, uv);
}
```



Exercice

1. Modifiez le *shader* précédent pour qu'il affiche l'image en **gris**

2. Comment pouvez-vous modifier l'image pour ne conserver qu'un **rectangle**?
3. Un **disque**?
4. Faites **tourner** l'image autour de son centre

Solution (1/3)

1: Il suffit de calculer la moyenne des champs rouge, vert et bleu (le minimum ou le maximum aussi suffirait):

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 uv = fragCoord / iResolution.xy;
    vec4 tex = texture(iChannel0, uv);
    float level = (tex.r + tex.g + tex.b) / 3.;
    fragColor = vec4(level, level, level, 1.);
}
```

2: Il suffit d'ajouter une fonction `inRectangle`:

```
const vec2 P1 = vec2(0.2, 0.2);
const vec2 P2 = vec2(0.8, 0.8);
const vec4 WHITE = vec4(1., 1., 1., 1.);

float inRectangle(vec2 p, vec2 p1, vec2 p2) {
    return step(p1.x, p.x) * step(p.x, p2.x) * step(p1.y, p.y) * step(p.y, p2.y);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 uv = fragCoord / iResolution.xy;
    vec4 tex = texture(iChannel0, uv);
    float rectangle = inRectangle(uv, P1, P2);
    fragColor = mix(WHITE, tex, rectangle);
}
```

Solution (2/3)

3: Même idée que pour le rectangle, mais avec un disque. Cette fois, on utilise les coordonnées absolues (`fragCoord`) et non les coordonnées relatives (`uv`):

```
const vec4 WHITE = vec4(1., 1., 1., 1.);


float inDisk(vec2 p, vec2 c, float r) {
    float d = distance(p, c);
    return smoothstep(d - 0.5, d + 0.5, r);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 uv = fragCoord / iResolution.xy;
    vec4 tex = texture(iChannel0, uv);
    vec2 center = iResolution.xy * 0.5;
    float disk = inDisk(fragCoord, center, 150.);
    fragColor = mix(WHITE, tex, disk);
}
```

Solution (3/3)

4: Il suffit d'utiliser une matrice de rotation 2×2 et d'appliquer la rotation autour du centre de l'image:

```
const vec4 WHITE = vec4(1., 1., 1., 1.);

mat2 rotationMatrix(float theta) {
    return mat2(cos(theta), -sin(theta), sin(theta), cos(theta));
}

vec2 rotate(vec2 p, float theta, vec2 c) {
    return rotationMatrix(theta) * (p - c) + c;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    float theta = iTime * 0.5;
    vec2 xy = rotate(fragCoord, theta, iResolution.xy * 0.5);
    vec2 uv = xy / iResolution.xy;
    vec4 tex = texture(iChannel0, uv);
    vec2 center = iResolution.xy * 0.5;
    fragColor = tex;
}
```

Shaders 3D

Shaders 3D

- Possible avec **ShaderToy**
- En générant les matériaux de façon **procédurale**
- Wolfenstein 3D: https://www.shadertoy.com/view/4sfGWX
- Transformations: https://www.shadertoy.com/view/MslSDN

Données, CPU et GPU

- En 3D, on préfère avoir plus de mécanismes de **communication**
- On utilise des **bibliothèques**
- **Exemple :** three.js, qui permet de créer des applications comme Infinitown
- Ou du code C/C++, en utilisant différentes bibliothèques, comme GLEW, GLFW, GLM et SDL

Dans le cours

Nous allons nous concentrer sur les **bibliothèques** suivantes :

- GLEW, qui rend OpenGL plus portable
- GLFW, qui gère les fenêtres, les surfaces, les événements
- GLM, qui fournit des outils mathématiques

En nous inspirant de la suite de **tutoriels** suivante :

- Site: http://www.opengl-tutorial.org/
- Dépôt Git: https://github.com/opengl-tutorials/ogl

Génération des exemples:

```
$ cd <dossier-du-depot> && mkdir build && cd build
$ cmake .. && make
$ ./launch-tutorial01_first_window.sh
```

Structure de base

```
// Include standard headers
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>

int main(void) {
    // Initialise GLFW
    [...]
    // Open a window and create its OpenGL context
    [...]
    // Initialize GLEW
    [...]
    // Dark blue background
    glClearColor(0.0f, 0.0f, 0.4f, 0.0f);
    // Main loop
    do {
        // Clear the screen to avoid flickering
        glClear(GL_COLOR_BUFFER_BIT);
        // Draw nothing
        glfwSwapBuffers(window);
        glfwPollEvents();
    } // Check if the ESC key was pressed or the window was closed
    while(glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&
          glfwWindowShouldClose(window) == 0);
    // Close OpenGL window and terminate GLFW
    glfwTerminate();
    return 0;
}
```

Triangle rouge (1/5) : préparation des données

```
GLuint VertexArrayID;
glGenVertexArrays(1, &VertexArrayID);
glBindVertexArray(VertexArrayID);

// Create and compile our GLSL program from the shaders
GLuint programID = LoadShaders(
    "SimpleVertexShader.vertexshader",
    "SimpleFragmentShader.fragmentshader"
);

static const GLfloat g_vertex_buffer_data[] = {
    -1.0f, -1.0f, 0.0f,
     1.0f, -1.0f, 0.0f,
     0.0f,  1.0f, 0.0f,
};

GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
             g_vertex_buffer_data, GL_STATIC_DRAW);
```

Triangle rouge (2/5) : chargement des shaders

```
GLuint LoadShaders(const char *vertex_file_path,
                   const char *fragment_file_path) {
    // Create the shaders
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
    [...]
    // Read the vertex shader code from the file
    [...]
    // Read the fragment shader code from the file
    [...]
    // Compile and check vertex shader
    [...]
    // Compile and check fragment shader
    [...]
    // Link the program
    GLuint ProgramID = glCreateProgram();
    glAttachShader(ProgramID, VertexShaderID);
    glAttachShader(ProgramID, FragmentShaderID);
    glLinkProgram(ProgramID);
    [...]
    // Check the program
    [...]
    // Clean
    [...]
    return ProgramID;
}
```

Triangle rouge (3/5) : shaders

Fichier `SimpleVertexShader.vertexshader`

```
layout(location = 0) in vec3 vertexPosition_modelspace;

void main() {
    gl_Position.xyz = vertexPosition_modelspace;
    gl_Position.w = 1.0;
}
```

Fichier `SimpleFragmentShader.fragmentshader`

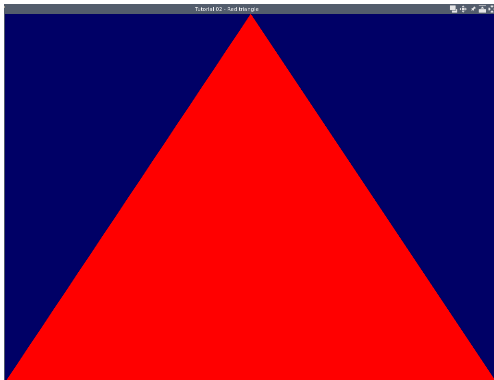
```
out vec3 color;

void main() {
    color = vec3(1, 0, 0);
}
```

Triangle rouge (4/5) : tracé du triangle

```
// Main loop
do {
    [...]
    // Use our shader
    glUseProgram(programID);
    // 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glVertexAttribPointer(
        0,         // attribute 0 must match the layout in the shader
        3,         // size
        GL_FLOAT, // type
        GL_FALSE, // normalized?
        0,         // stride
        (void*)0  // array buffer offset
    );
    // Draw the triangle!
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glDisableVertexAttribArray(0);
    [...]
} while(glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&
        glfwWindowShouldClose(window) == 0);
```

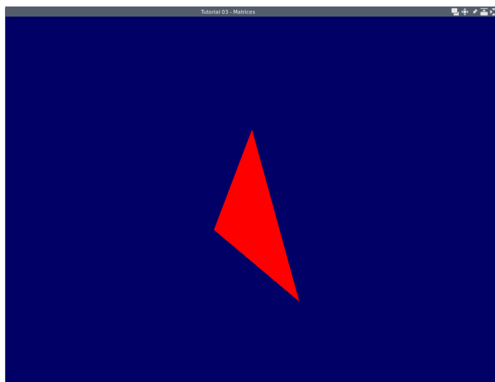
Triangle rouge (5/5) : nettoyage

```
// Cleanup VBO
glDeleteBuffers(1, &vertexbuffer);
glDeleteVertexArrays(1, &VertexArrayID);
glDeleteProgram(programID);
```



Transformations (1/4)

- On souhaite maintenant regarder le triangle avec une **caméra**
- En utilisant la projection **perspective**



Transformations (2/4) : préparation de la matrice

```
// Get a handle for our "MVP" uniform
GLuint MatrixID = glGetUniformLocation(programID, "MVP");
// Projection matrix : 45° Field of View, 4:3 ratio,
//                     display range : 0.1 unit <-> 100 units
glm::mat4 Projection = glm::perspective(glm::radians(45.0f),
                                        4.0f / 3.0f, 0.1f, 100.0f);
// Camera matrix
glm::mat4 View = glm::lookAt(
    glm::vec3(4, 3, 3), // Camera is at (4,3,3)
    glm::vec3(0, 0, 0), // and looks at the origin
    glm::vec3(0, 1, 0)  // Head is up
);
// Model matrix : an identity matrix (model will be at the origin)
glm::mat4 Model      = glm::mat4(1.0f);
// Our ModelViewProjection
glm::mat4 MVP        = Projection * View * Model;
```

Transformations (3/4) : boucle principale

```
do {
    [...]

    // Send our transformation to the currently bound shader,
    // in the "MVP" uniform
    glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);

    [...]

    // Draw the triangle !
    glDrawArrays(GL_TRIANGLES, 0, 3); // 3 indices starting at 0

    [...]
} // Check if the ESC key was pressed or the window was closed
while(glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
      glfwWindowShouldClose(window) == 0);
```

Transformations (4/4) : shaders

Shader de sommet:

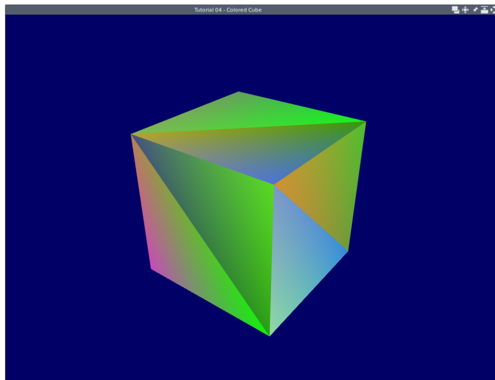
```
layout(location = 0) in vec3 vertexPosition_modelspace;

uniform mat4 MVP;

void main() {
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
}
```

Shader de fragment:

```
out vec3 color;

void main() {
    color = vec3(1, 0, 0);
}
```

Cube colorié (1/4)

- Nous allons maintenant construire un **cube**
- En ajoutant des couleurs aléatoires dans **chaque sommet**
- La couleur des points du triangle va être obtenue par **interpolation**



Cube colorié (2/4) : préparation des données

```
// Our vertices. Tree consecutive floats give a 3D vertex.
// Three consecutive vertices give a triangle.
// There are 6*2=12 triangles, and 12*3 vertices
static const GLfloat g_vertex_buffer_data[] = {
    -1.0f,-1.0f,-1.0f, -1.0f,-1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
     1.0f, 1.0f,-1.0f, -1.0f,-1.0f,-1.0f, -1.0f, 1.0f,-1.0f,
     [...]
     1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f,  1.0f,-1.0f, 1.0f
};

// One color for each vertex. They were generated randomly.
static const GLfloat g_color_buffer_data[] = {
    0.583f,  0.771f,  0.014f, 0.609f,  0.115f,  0.436f,
    0.327f,  0.483f,  0.844f, 0.822f,  0.569f,  0.201f,
    [...]
    0.820f,  0.883f,  0.371f, 0.982f,  0.099f,  0.879f
};

GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
    g_vertex_buffer_data, GL_STATIC_DRAW);

GLuint colorbuffer;
glGenBuffers(1, &colorbuffer);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_color_buffer_data),
    g_color_buffer_data, GL_STATIC_DRAW);
```

Cube colorié (3/4) : boucle principale

```
// 1rst attribute buffer : vertices
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

// 2nd attribute buffer : colors
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

// Draw the triangles !
glDrawArrays(GL_TRIANGLES, 0, 12*3);

glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
```

Cube colorié (4/4) : shaders

Shader de sommet:

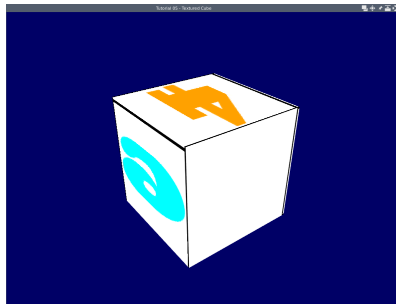
```
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexColor;
out vec3 fragmentColor;
uniform mat4 MVP;

void main(){
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
    fragmentColor = vertexColor;
}
```

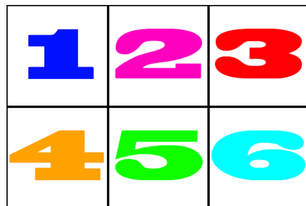
Shader de fragment:

```
in vec3 fragmentColor;
out vec3 color;

void main(){
    color = fragmentColor;
}
```

Cube texturé (1/4)

- Même cube, en y ajoutant une **texture**





Cube texturé (2/4) : préparation des données

```
// Our vertices. Tree consecutive floats give a 3D vertex.
// Three consecutive vertices give a triangle.
// There are 6*2=12 triangles, and 12*3 vertices
static const GLfloat g_vertex_buffer_data[] = {
    -1.0f,-1.0f,-1.0f, -1.0f,-1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
     1.0f, 1.0f,-1.0f, -1.0f,-1.0f,-1.0f, -1.0f, 1.0f,-1.0f,
     [...]
     1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f,  1.0f,-1.0f, 1.0f
};

// Two UV coordinatesfor each vertex, created with Blender
static const GLfloat g_uv_buffer_data[] = {
    0.000059f, 1.0f-0.000004f, 0.000103f, 1.0f-0.336048f,
    0.335973f, 1.0f-0.335903f, 1.000023f, 1.0f-0.000013f,
    [...]
    1.000004f, 1.0f-0.671847f, 0.667979f, 1.0f-0.335851f
};

GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
    g_vertex_buffer_data, GL_STATIC_DRAW);

GLuint uvbuffer;
glGenBuffers(1, &uvbuffer);
glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_uv_buffer_data),
    g_uv_buffer_data, GL_STATIC_DRAW);
```

Cube texturé (3/4) : boucle principale

```
// Bind our texture in Texture Unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
// Set our "myTextureSampler" sampler to use Texture Unit 0
glUniform1i(TextureID, 0);

// 1rst attribute buffer : vertices
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

// 2nd attribute buffer : UVs
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);

// Draw the triangles !
glDrawArrays(GL_TRIANGLES, 0, 12*3);
```

Cube texturé (4/4) : shaders

Shader de sommet

```
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
out vec2 UV;
uniform mat4 MVP;

void main(){
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
    UV = vertexUV;
}
```

Shader de fragment

```
in vec2 UV;
out vec3 color;
uniform sampler2D myTextureSampler;

void main() {
    color = texture(myTextureSampler, UV).rgb;
}
```

Pour en savoir plus...

- **Tutorial 06**: clavier et souris
- **Tutorial 07**: chargement d'un modèle
- **Tutorial 08**: ombrage de base
- **Tutorial 09**: utilisation d'un VBO
- **Tutorial 10**: transparence
- **Tutorial 11**: police de caractères
- **Tutorial 12**: extensions
- **Tutorial 13**: cartes de normales
- **Tutorial 14**: rendu sur texture
- **Tutorial 15**: cartes de lumière
- **Tutorial 16**: cartes d'ombres
- **Tutorial 17**: rotations
- **Tutorial 18**: système de particules