Introduction à OpenGL

Les bases et WebGL

Daniel Meneveaux

Université de Poitiers

Master Gphy, 2015

Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Sommaire

1 Objectifs du cours
- Quelques principes
- Champs d'application

2 Rappels

3 Contenu du cours

4 Pipeline graphique

5 WebGL - démarrage

6 WebGL - geometry buffers

7 Entracte IHM

8 WebGL - shaders

9 Gestion des textures

10 WebGL - buffers principaux

11 Gestion des ombres

12 Traitement d'images

13 Autres outils à voir

Objectifs du cours

- Rappels (modèles, transformations)
- Comprendre les opérations d'une carte graphique
- Quelques exemples de mise en œuvre avec OpenGL

- Tests avec WebGL
  - Affichage en 2D, 3D, fil de fer et polygonal
  - Affichage en mode polygonal, transformations
  - Interactions avec l'utilisateur
  - Développement de shaders (chroma, textures, etc.)

- Mise en pratique, à partir d'une bibliothèque existante
  - Avec javascript (three.js)
  - Module WebGL existant
  - Fonctionnalités à compléter

Principes et Champs d'application

- Objectif principal : imagerie interactive
- Carte graphique (GPU)
  - Carte vidéo, calculs dédiés à l'affichage
  - Calculs parallèles, nombreux processeurs *graphiques*
  - Organisation spécifique des zones de mémoire
  - De plus en plus détournée pour d'autres applications
- OpenGL : bibliothèque pour utiliser le GPU
  - Spécial Gphy : imagerie médicale
  - Jeux vidéos 3D interactifs
  - Pré-visualisation de films d'animation
  - Visualisation de données (visualisation scientifique)
  - Possibilités de faire du traitement d'images rapide
- Alternative sous windows : DirectX (bien plus fermé)
- Et plus récemment : WebGL
  - OpenGL actif sur les navigateurs web, avec javascript
    $\Rightarrow$ *Activez WebGL sur votre navigateur ! ! !*

Sommaire

1. Objectifs du cours
2. Rappels
   - Représentation des données
   - Matrices de transformation
   - Autres opérations utiles
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Modèles géométriques

- Utilisation de maillages uniquement
- Objets représentés par leurs bords (sommets, arêtes, faces)
- Principe de l'affichage :
  - Projection de la géométrie sur l'écran (matrice de projection)
  - Tracé de droites (Algorithme de Bresenham) [Fil de fer]
  - Remplissage des polygones (par lignes de balayages)
  - Interpolation bilinéaire (couleurs, coordonnées de textures, etc.)

- N'oubliez pas, pour le réalisme :
  - La projection et le remplissage ne suffisent pas
  - Prise en compte de la lumière et des matériaux
  - Ajout de textures (chroma, bump maps, displacement, etc.)
  - Ombres dures et ombres douces
  - Inter-réflexions lumineuses

Maillages

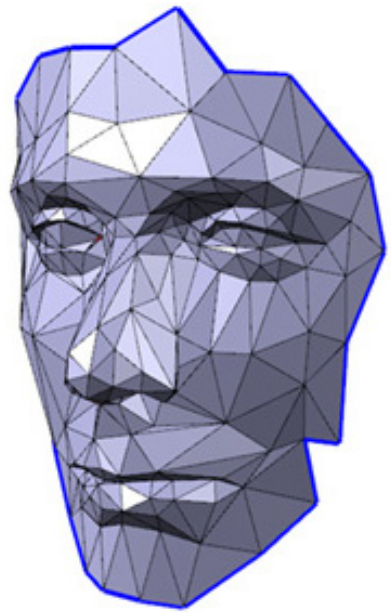
Liste de sommets "organisée"

- Liste de sommets
- Liste d'indices

$$
\left.\begin{array}{l} v_1 : x_1, y_1, z_1 \\ v_2 : x_2, y_2, z_2 \\ v_3 : x_3, y_3, z_3 \end{array}\right\} \text{Triangle 1}
$$

$$
\left.\begin{array}{l} v_4 : x_4, y_4, z_4 \\ v_5 : x_5, y_5, z_5 \\ v_6 : x_6, y_6, z_6 \end{array}\right\} \text{Triangle 2}
$$

$$
\left.\begin{array}{l} \ldots \end{array}\right\} \text{etc.}
$$

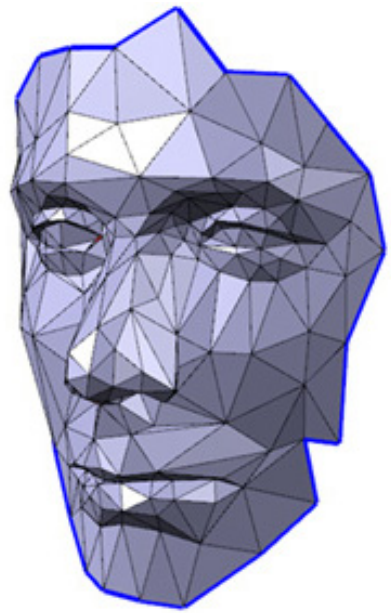


Maillages

Liste de sommets "indexée"

- Liste de sommets
- Liste d'indices

$v_1 : x_1, y_1, z_1$	$t_1 : 1, 2, 3$
$v_2 : x_2, y_2, z_2$	$t_2 : 2, 1, 4$
$v_3 : x_3, y_3, z_3$	$t_3 : 1, 5, 4$
$v_4 : x_4, y_4, z_4$	$\ldots$
$v_5 : x_5, y_5, z_5$	$\ldots$
$\ldots$	$\ldots$
$v_i : x_i, y_i, z_i$	$t_j : i_1, i_2, i_3$



Transformations

- Translations, Rotations, Homothéties, Projections
- Représentation en coordonnées homogènes (matrices 4×4)

$$\text{Translation} : \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{Homothétie} : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/\lambda \end{bmatrix}$$

$$\text{Rotation} : \begin{bmatrix} R_{1,1} & R_{1,2} & R_{1,3} & 0 \\ R_{2,1} & R_{2,2} & R_{2,3} & 0 \\ R_{3,1} & R_{3,2} & R_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{Projection} : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix}$$

Rappels importants

- Pour OpenGL : profondeur selon l'axe Z
- Ordre des opérations (combiner les matrices)
  - Rotation, puis translation, puis projection :
  - M_r, M_t, M_p
  - $M_{total} = M_p \times M_t \times M_r$
- Pour rappel, $M \times V$

$$\begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{bmatrix} \times \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix} = \begin{bmatrix} V_x C_{1,1} + V_y C_{1,2} + V_z C_{1,3} + 1 C_{1,4} \\ V_x C_{2,1} + V_y C_{2,2} + V_z C_{2,3} + 1 C_{2,4} \\ V_x C_{3,1} + V_y C_{3,2} + V_z C_{3,3} + 1 C_{3,4} \\ V_x C_{4,1} + V_y C_{4,2} + V_z C_{4,3} + 1 C_{4,4} \end{bmatrix}$$

Autres opérations importantes

Implémentées par OpenGL

- L'affichage par fil de fer (Bresenham)
- Remplissage par lignes de balayage
- Interpolations bilinéaires
- Plaquage de textures
- Calculs d'éclairement (sources lumineuses)
- Variétés de matériaux (modèles de BRDF)

- Interaction avec l'utilisateur (glut ou ici html/javascript)
  - Souris, clavier
  - Événement $\Rightarrow$ *changement de caméra = matrices !*

Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
   - Les transparents du cours
   - Les fichiers de test associés
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Ce cours

- Ce fichier pdf
- Nécessite des explications
- Nécessite une prise de note
- Probablement inutilisable seul
- Surtout complétez :

  ⇒ *Prenez les exemples associés*

  ⇒ *Modifiez les exemples, adaptez-les*

Tests html et javascript

- Les exemples sont donnés dans l'ordre
  - ⇒ *Difficulté croissante*
  - ⇒ *Nombre de lignes de codes augmente !*
- Ouvrez les fichiers
  - ⇒ *Essayez de suivre le déroulement des opérations*
  - ⇒ *Gardez toujours à l'esprit le fonctionnement "machine à état"*
- Il existe de nombreuses aides sur internet
  - ⇒ *WebGL relativement nouveau*
  - ⇒ *OpenGL jamais très loin, mais faites la part des choses*
  - ⇒ *Comme d'habitude, internet n'est pas toujours notre ami !*

Sommaire

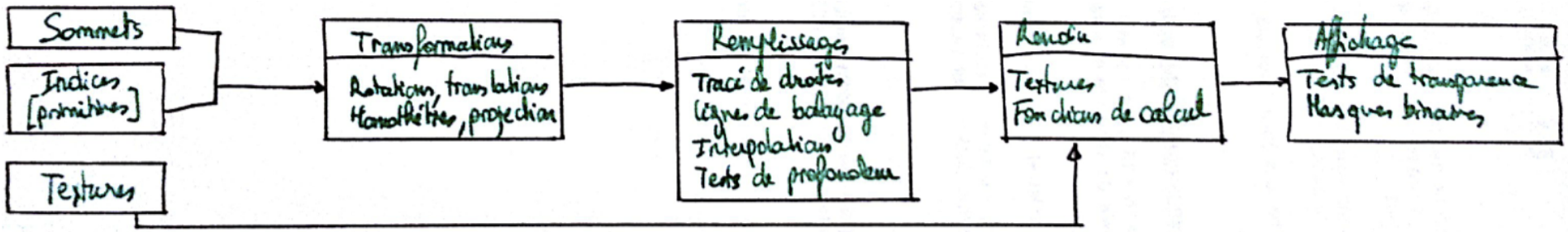
1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
   - Principe général
   - Ordre des opérations
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Bref historique

- Avant OpenGL : GL sur les stations graphiques SILICON
- Apparition d'OpenGL : 1992, disponible sur différents OS
- Aujourd'hui, nombreux langages supportés :
  - C, C++, Java, Fortran, etc.
- Modeleurs 3D importants :
  - Maya, 3DSmax, Blender (gratuit), Catia, Autocad, Archicad, etc.
- Nombreux outils de rendu :
  - Ogre, Renderman (LR), PBRT (LR), Mitsuba (LR), etc.
- Bibliothèques très utiles [deviennent obsolètes] :
  - GLUT (interface graphique simple)
  - GLU (objets paramétriques, placement de caméra)
  - GLX (liens avec X-windows sous linux)
  - Liens existants avec interfaces graphiques (Gtk, Qt, etc.)

Série d'opérations d'affichage

- Données d'entrée
  - Sommets, Indices, Textures (+ d'autres plus tard)
- Première étape : projection des sommets
- Seconde étape : remplissage des polygones et profondeur
- Lors du remplissage : interpolations, textures, éclairement
- À la fin : calculs de transparence, stencil-buffer



OpenGL : une machine à état

- Pour une image :
  - transmettre les nouvelles données à la carte (si besoin seulement)
  - définir les paramètres d'affichage (matériaux, textures, transparence)
  - déclencher les opérations pour un groupe de polygones
  - définir les paramètres d'affichage (matériaux, textures, transparence)
  - déclencher le pipeline pour un groupe de polygones
  - etc.
- Paramètres par groupe de polygones (textures, matériaux)
- Activation/désactivation de fonctionnalités
- Bien définir l'état du GPU avant d'afficher un groupe :
  - les paramètres du matériau
  - activer la texture
  - d'autres choses (shaders, par exemple)

Sommaire

1 Objectifs du cours
2 Rappels
3 Contenu du cours
4 Pipeline graphique
5 WebGL - démarrage
- OpenGL et navigateur Web
- Dessin en 2D
- Affichage 3D

6 WebGL - geometry buffers
7 Entracte IHM
8 WebGL - shaders
9 Gestion des textures
10 WebGL - buffers principaux
11 Gestion des ombres
12 Traitement d'images
13 Autres outils à voir

Principe et variables

- Pour dessiner : récupérer le contexte openGL
  ⇒ *depuis un objet canvas*
- l'objet est référencé par la variable *gl*
- les fonctionnalités *openGL* sont associées à *gl*

- N'oubliez pas :
  ⇒ *OpenGL est une machine à état*
  ⇒ *Les fonctionnalités sont activées ou non*
  ⇒ *L'état de la machine détermine la manière dont les objets sont affichés*

Fenêtre OpenGL, et javascript

- Utilisation de l'objet *canvas* dans la page web

```
<body onload="webGLStart();">
  <canvas id="WebGL-test" style="border:none;" width="500" height="500"></canvas>
</body>
```

- Avec l'initialisation javascript *webGLStart()*

```
function webGLStart() {
  var canvas = document.getElementById("WebGL-test");
  initGL(canvas);
  initShaders();
  initBuffers();

  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.enable(gl.DEPTH_TEST);

  drawScene();
}
```

- Possibilité d'utiliser une bibliothèque externe : *Three.js*

  ⇒ *Très peu dans ce cours*

Initialisations

- Pour la zone de dessin OpenGL (viewport) :

```
function initGL(canvas)
{
  try {
    gl = canvas.getContext("experimental-webgl");
    gl.viewportWidth = canvas.width;
    gl.viewportHeight = canvas.height;
    gl.viewport(0, 0, canvas.width, canvas.height);
  } catch (e) {}
  if (!gl) {
    alert("Could not initialise WebGL");
  }
}
```

- Pour l'initialisation des fonctions de rendu (shaders) :

  ⇒ *initShaders() en javascript, mais plus tard*

- Pour l'initialisation des objets (shaders) :

  ⇒ *initBuffers() en javascript, mais plus tard*

Calcul/Affichage d'une image

Pour chaque image, on utilise la fonction drawScene()

```
function drawScene() {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
      squareVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

  mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix, [0.0, 0.0, -7.0]);
  setMatrixUniforms();

  gl.drawArrays(gl.LINE_LOOP, 0, squareVertexPositionBuffer.numItems);
}
```

⇒ *gl.clear()* : réinitialisation de l'image

⇒ *gl.bind()* : choix des shaders, texture, buffer de géométrie

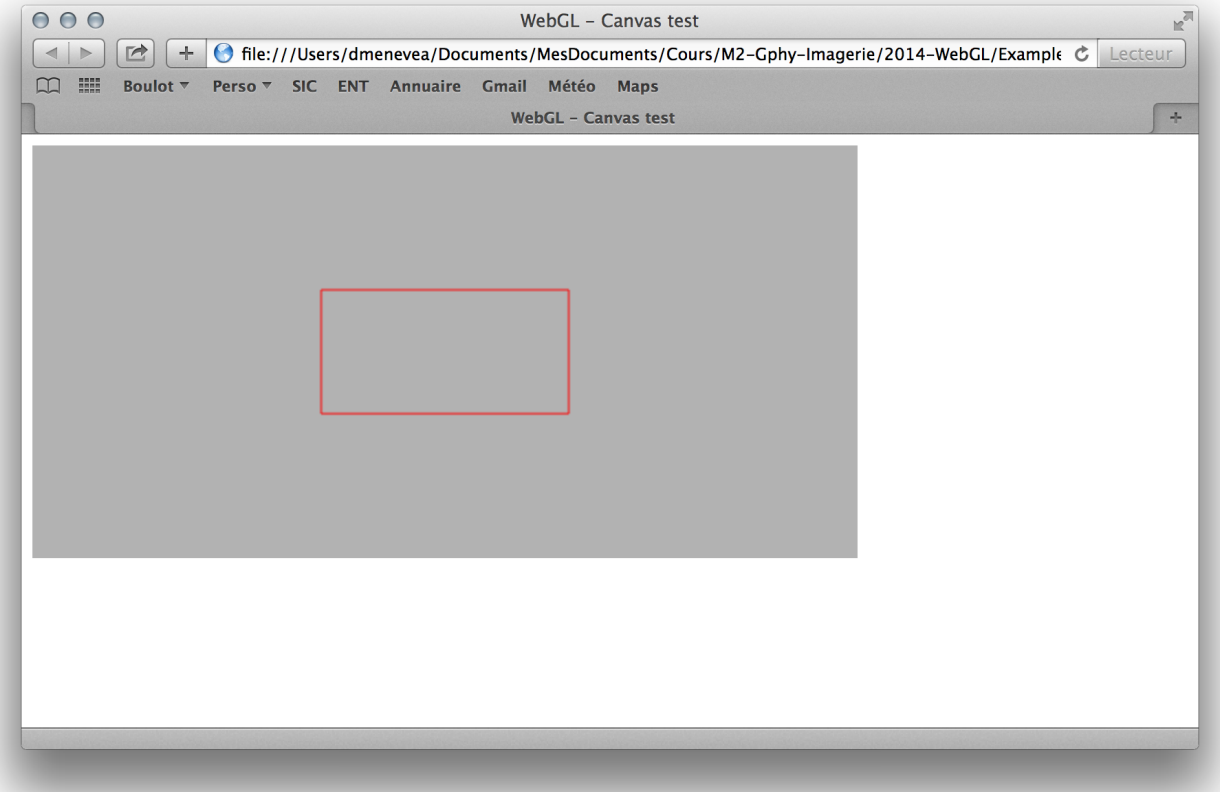
⇒ *gl.uniformMatrix()* : définition des matrices de transformations

⇒ *gl.drawArrays()* : déclenchement de l'affichage (avec le mode)

Les détails suivent progressivement dans les prochains transparents

Affichage en fil de fer

- Les fichiers utilisés pour ce cours :
  - glMatrix.js : une bibliothèque pour les matrices
  - glCourseBasis.js : notre fichier externe javascript
  - main.html : le fichier html
- Et la page obtenue



tuto1 glCourseBasis.js (1/1)

```
1
2  // ====================================================
3  function getShader(gl, id)
4  {...}
5
6  // ====================================================
7  function initShaders()
8  {...}
9
10 // ====================================================
11 function initBuffers()
12 {...}
13
14 // ====================================================
15 function drawScene()
16 {
17   gl.clear(gl.COLOR_BUFFER_BIT);
18
19   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
20   gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
21       vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
22
23   gl.drawArrays(gl.LINE_LOOP, 0, vertexBuffer.numItems);
24 }
25
26
```

initBuffers() initShaders()

tuto1

main.html (1/3)

```
1  <html>
2
3  <head>
4    <title>WebGL - Canvas test</title>
5    <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
6
7  <!-------------------------------------------------------------------->
8    <script type="text/javascript" src="glMatrix.js"></script>
9    <script type="text/javascript" src="glCourseBasis.js"></script>
10 <!-------------------------------------------------------------------->
11   <script id="shader-vs" type="x-shader/x-vertex">
12     attribute vec2 vertexPosition;
13     void main(void) {
14       gl_Position = vec4(vertexPosition, 0.0, 1.0);
15     }
16   </script>
17 <!-------------------------------------------------------------------->
18   <script id="shader-fs" type="x-shader/x-fragment">
19     precision mediump float;
20     void main(void) {
21       gl_FragColor = vec4(1.0,0.0,0.0,1.0);
22     }
23   </script>
24 <!-------------------------------------------------------------------->
25
26 <script type="text/javascript">
27
28 // ========================================
29 var gl;
30
```

tuto1 main.html (2/3)

```
31 // ============================================
32 function initGL(canvas)
33 {
34   try {
35     gl = canvas.getContext("experimental-webgl");
36     gl.viewportWidth = canvas.width;
37     gl.viewportHeight = canvas.height;
38     gl.viewport(0, 0, canvas.width, canvas.height);
39   } catch (e) {}
40   if (!gl) {
41     alert("Could not initialise WebGL");
42   }
43 }
44
45 // ============================================
46 function webGLStart() {
47   var canvas = document.getElementById("WebGL-test");
48   initGL(canvas);
49   initShaders();
50   initBuffers();
51
52   gl.clearColor(0.7, 0.7, 0.7, 1.0);
53   //gl.enable(gl.DEPTH_TEST);
54
55   drawScene();
56 }
57 </script>
58 <!-------------------------------------------------------------------->
59 </head>
60
```

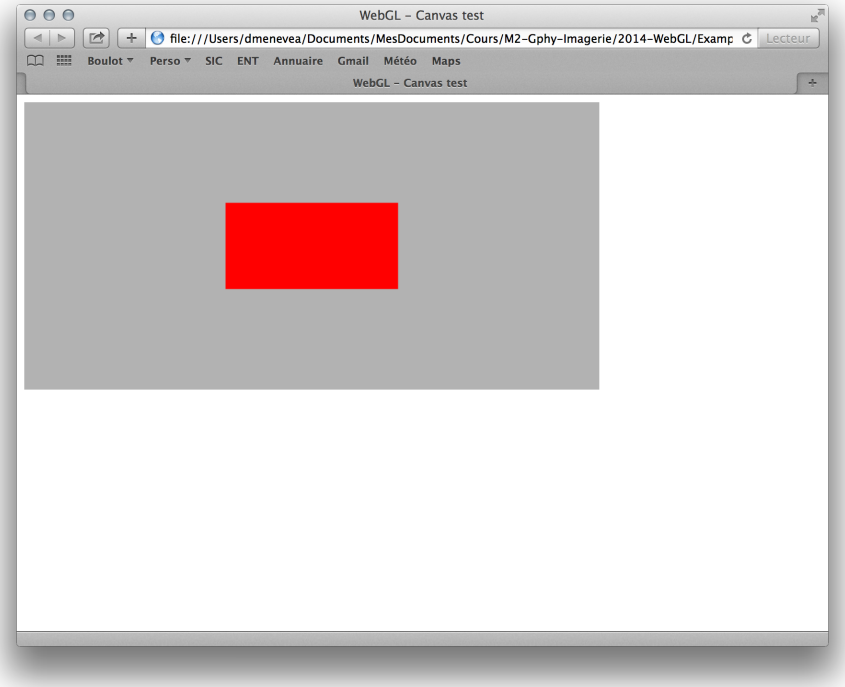
tuto1 main.html (3/3)

```
61 <!-------------------------------------------------------------------------->
62 <!-------------------------------------------------------------------------->
63 <!-------------------------------------------------------------------------->
64 <!-------------------------------------------------------------------------->
65 <!-------------------------------------------------------------------------->
66
67 <body onload="webGLStart();">
68   <canvas id="WebGL-test" style="border:none;" width="800" height="500"></canvas>
69 </body>
70
71 <!-------------------------------------------------------------------------->
72 <!-------------------------------------------------------------------------->
73 <!-------------------------------------------------------------------------->
74 <!-------------------------------------------------------------------------->
75 <!-------------------------------------------------------------------------->
76
77 </html>
78
79
```

Affichage plein (remplissage)

- Une seule modification dans le code ⇒ *la fonction drawScene()*

```
gl.drawArrays(gl.LINE_LOOP, 0, vertexBuffer.numItems);
gl.drawArrays(gl.TRIANGLE_FAN, 0, vertexBuffer.numItems);
```



⇒ *Attention, l'ordre des sommets a une importance*

En 3D : initialisations et gestion de la projection

- Deux matrices prévues par OpenGL :
  ⇒ *ModelView (utilisée plutôt pour déplacer les objets)*
  ⇒ *Projection (utilisée plutôt pour la projection)*
- En pratique, elles sont initialisées par le programmeur
  ⇒ *Il peut donc choisir l'utilisation qu'il veut en faire*
- Exemple :

```
var mvMatrix = mat4.create();
var pMatrix = mat4.create();
mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
mat4.identity(mvMatrix);
mat4.translate(mvMatrix, [0.0, 0.0, -7.0]);
gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
```

⇒ *Les fonctions utilisées ici pour les matrices sont dans glMatrix.js*

En 3D : tout est dit, il suffit d'afficher

- Bien faire attention à ce qui doit être réinitialisé par image
- Si le point de vue et les objets ne changent pas
  ⇒ *inutile de reconstruire les matrices*
- Si seul le point de vue change
  ⇒ *inutile de reconstruire la matrice de projection*
- Et voici la fonction *drawScene()*

```
99  function drawScene() {
100   gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
101
102   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
103   gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
104       vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
105
106   mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
107   mat4.identity(mvMatrix);
108   mat4.translate(mvMatrix, [0.0, 0.0, -7.0]);
109   setMatrixUniforms();
110
111   gl.drawArrays(gl.LINE_LOOP, 0, vertexBuffer.numItems);
112 }
```

Exemple d'utilisation des matrices

- Dessin d'un rectangle (*vertexBuffer*) d'une façon
- Dessin du même rectangle d'une autre façon

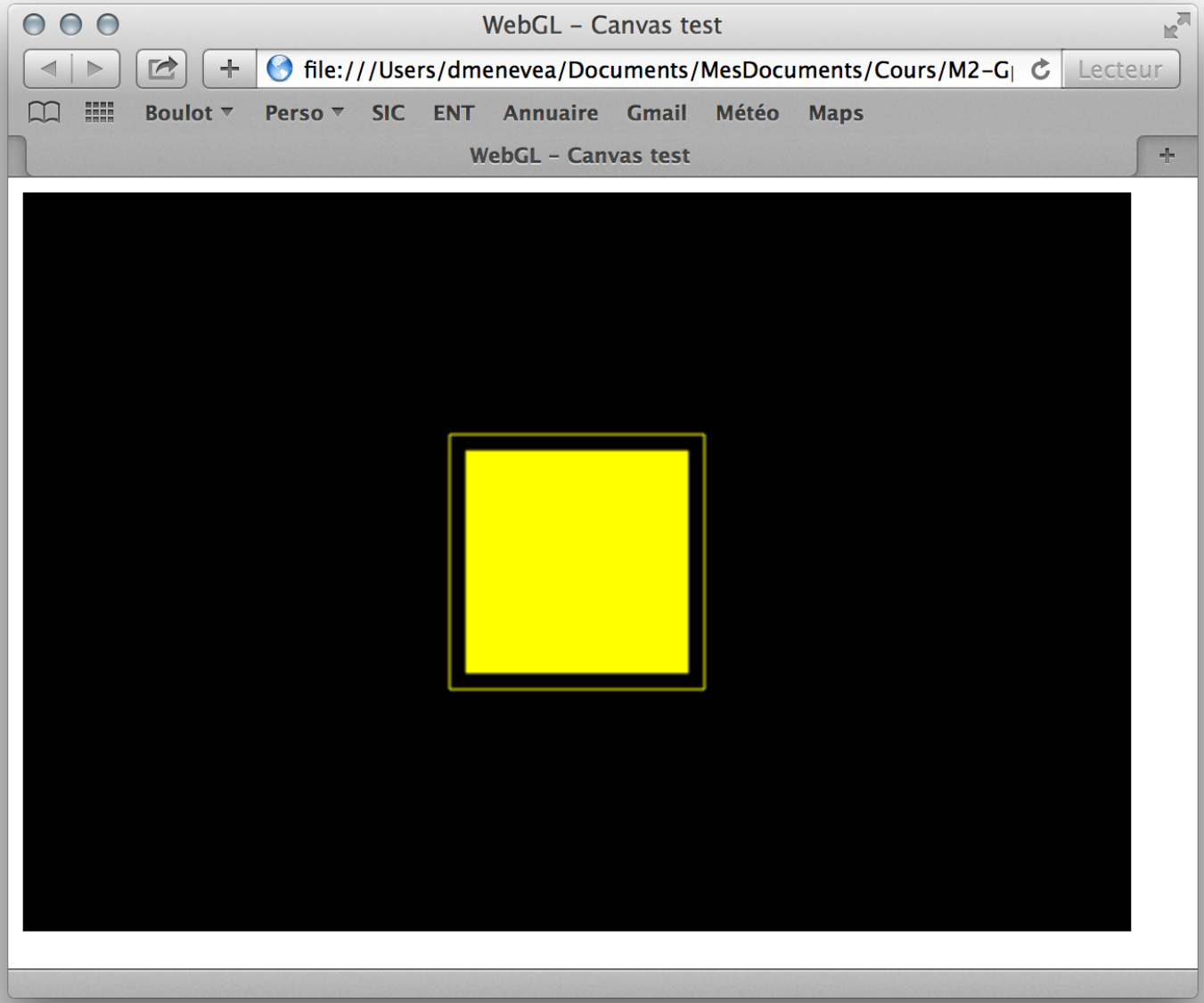
```
function drawScene() {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
      vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

  mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix, [0.0, 0.0, -7.0]);
  setMatrixUniforms();
  gl.drawArrays(gl.LINE_LOOP, 0, vertexBuffer.numItems);

  mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix, [0.0, 0.0, -8.0]);
  setMatrixUniforms();
  gl.drawArrays(gl.TRIANGLE_FAN, 0, vertexBuffer.numItems);
}
```

Résultat de l'affichage



Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
   - Liste unique
   - Listes indexées
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Liste de sommets unique

rappels

- Les sommets sont donnés directement en une seule liste
- Un sommet commun est donné autant de fois que nécessaire

  ⇒ *Redondance de données*

  ⇒ *Aucune notion d'adjacence/incidence*
- Stockage d'un objet :
  - Tableau organisé de valeurs
  - Trois valeurs par sommet (x,y,z)
  - Toujours le même nombre de sommets par primitive

    ⇒ *soit toujours des triangles*

    ⇒ *soit toujours des quadriltères*
- Le tableau de valeurs est ensuite envoyé sur la carte

  ⇒ *appel à la fonction gl.bufferData(...)*
- Le nombre de sommets par primitive est également donné

  ⇒ *appel à la fonction gl.drawArrays(...)*

Construction d'une liste pour le rectangle ▸ tuto1

```
function initBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  vertices = [
              -0.3, -0.3,
              -0.3,  0.3,
               0.3,  0.3,
               0.3, -0.3
              ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 2;
  vertexBuffer.numItems = 4;
}
```

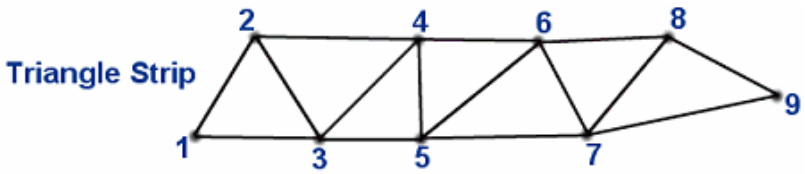
2D

```
function initBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  vertices = [
              -1.0,  -1.0,  0.0,
              -1.0,   1.0,  0.0,
               1.0,   1.0,  0.0,
               1.0,  -1.0,  0.0
              ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numItems = 4;
}
```

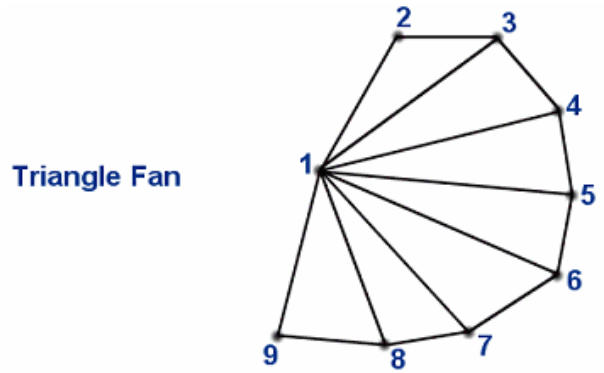
3D

Construction d'une liste pour un cube 3D

- Le cube compte : 6 faces, 12 arêtes, 8 sommets
- La structure dépend du mode d'affichage ⇒ *gl.drawArrays()*
  - *gl.LINES* : dessine des segments de deux points
  - *gl.LINE_STRIP* : dessine une seule ligne brisée
  - *gl.LINE_LOOP* : dessine une seule ligne brisée et la ferme
  - *gl.TRIANGLES* : dessine des triangles pleins
  - *gl.TRIANGLE_STRIP* : suite de triangles (partage de la seconde arête)



  - *gl.TRIANGLE_FAN* : suite de triangles (partage du premier sommet)



Choix de la structure pour différents modes

Déjà compliquée pour un cube...

- Pour un affichage en fil de fer :
  ⇒ *gl.LINE_LOOP* nécessite de trouver un chemin sur l'objet
  ⇒ *gl.LINE_STRIP* idem, avec une dernière arête
  ⇒ *gl.LINES* nécessite de répéter tous les sommets deux fois
  ⇒ *gl.TRIANGLES** affiche seulement des triangles pleins
- Pour un affichage en mode plein
  - Autres modes d'affichage
  - Qui ne correspondent pas pour un cube
  - Ni pour la plupart des autres maillages...

  ⇒ *Voir les résultats sur un navigateur (tuto3-CubeWireI)*

- La meilleure solution : utiliser des structures indexées !

Utilisation de listes indexées

▸ rappels

Souvent la seule bonne méthode

- Les sommets sont donnés une seule fois
- Selon l'affichage, seule l'indexation nécessite de changer
- Permet de simplifier la "saisie" et le stockage
- Ajoute une liste d'indices et intègre une forme de topologie
  $\Rightarrow$ *Sommets communs utilisés par chaque maille incidente*

- Méthode
  - La liste de sommets existe toujours
  - Un second tableau indique l'organisation des primitives

Structure indexée

Tableau de sommets ⇒ *façon identique (ARRAY_BUFFER)*

```
vertices = [ -1.0, -1.0,  -1.0,     1.0, -1.0,  -1.0,
              1.0,  1.0,  -1.0,    -1.0,  1.0,  -1.0,
             -1.0, -1.0,   1.0,     1.0, -1.0,   1.0,
              1.0,  1.0,   1.0,    -1.0,  1.0,   1.0 ];
vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
vertexBuffer.itemSize = 3;
vertexBuffer.numItems = 8;
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                       vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

Tableau des indices ⇒ *nouveau buffer (ELEMENT_ARRAY_BUFFER)*

```
var indices = [ 0, 1, 1, 2, 2, 3, 3, 0,
                4, 5, 5, 6, 6, 7, 7, 4,
                0, 4, 1, 5, 2, 6, 3, 7 ];
indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);
indexBuffer.itemSize = 1;
indexBuffer.numItems = 24;
```

Affichage de la structure

- Sélection du buffer d'indices
- Définition des paramètres d'affichage (machine à état)
- Déclenchement de l'affichage

```
function drawScene()
{
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);

  mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix, [0.0, 0.0, -7.0]);
  mat4.multiply(mvMatrix, objMatrix);
  setMatrixUniforms();

  gl.drawElements(gl.LINES, indexBuffer.numItems, gl.UNSIGNED_SHORT, 0);
}
```

Sommaire

1 Objectifs du cours
2 Rappels
3 Contenu du cours
4 Pipeline graphique
5 WebGL - démarrage
6 WebGL - geometry buffers
7 Entracte IHM
- Côté HTML
- Côté javascript

8 WebGL - shaders
9 Gestion des textures
10 WebGL - buffers principaux
11 Gestion des ombres
12 Traitement d'images
13 Autres outils à voir

Événements à déclarer pour le canvas

- Déclaration identique pour le canvas

```
<body onload="webGLStart();">
  <canvas id="WebGL-test" style="border:none;" width="600" height="400"></canvas>
</body>
```

- Légère modification pour *webGLStart()*

```
function webGLStart() {
  var canvas = document.getElementById("WebGL-test");
  initGL(canvas);
  initShaders();
  initBuffers();

  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.enable(gl.DEPTH_TEST);

  canvas.onmousedown = handleMouseDown;
  document.onmouseup = handleMouseUp;
  document.onmousemove = handleMouseMove;

  tick();
}
```

Evenements à gérer pour OpenGL

- La fonction *tick()*

```
function tick() {
  requestAnimFrame(tick);
  drawScene();
}
```

- La gestion de la souris ⇒ *handleMouse*() utilisés par webGLStart()*
- La gestion des événements d'affichage ⇒ *requestAnimFrame()*

- Le code est défini dans les exemples du cours
  ⇒ *tuto4-CubeIndex, par exemple*
- A partir de maintenant
  ⇒ *nous pouvons nous concentrer sur l'affichage !*

Sommaire

1 Objectifs du cours
2 Rappels
3 Contenu du cours
4 Pipeline graphique
5 WebGL - démarrage
6 WebGL - geometry buffers
7 Entracte IHM
8 WebGL - shaders
- Principe et fonctionnement
- Vertex shader
- Fragment shader
- Passage de paramètres

9 Gestion des textures
10 WebGL - buffers principaux
11 Gestion des ombres
12 Traitement d'images
13 Autres outils à voir

Premiers shaders

- Shader : fonction permettant de décider de l'aspect des objets
- Deux fonctions principales :
  - *Vertex shader* : valeurs attribuée aux sommets
  - *Fragment shader* : valeur attribuée aux pixels
- Fonctions programmables sur la carte graphique
- Avec un langage spécifique (proche du C)
- Compilées, liées, et stockées directement sur la carte graphique
- Possibilité de transmettre des paramètres
- Avec des variables pré-existantes dans le langage

⇒ *Le langage est assez puissant, mais il reste des limites à son utilisation*

Premier vertex shader

- Déjà défini dans le fichier *main.html* fourni

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
  }
</script>
```

⇒ *A la manière d'un script*

⇒ *Comporte une fonction main()*

⇒ *Plusieurs types de variables/constantes globales*

⇒ *Objectif : définir les attributs de l'affichage pour le sommet courant*

Premier fragment shader

- Egalement défini dans le fichier *main.html* fourni

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  void main(void) {
    gl_FragColor = vec4(1.0,1.0,0.0,1.0);
  }
</script>
```

⇒ *Egalement à la manière d'un script*

⇒ *Egalement avec une fonction main()*

⇒ *Plusieurs types de variables/constantes globales*

⇒ *Objectif : définir les attributs de l'affichage pour chaque pixel*

Passage de paramètres

- Trois types de paramètres :
  - Les attributs (associés aux sommets d'une forme)
  - Les valeurs uniformes (associés aux deux shaders)
  - Les valeurs variables (variables par interpolation lors du remplissage)
- Pour récupérer une valeur dans un shader
  - Réservation d'un nom de variable à la création du *ShaderProgram* :
    $\Rightarrow$ *gl.getUniformLocation(shaderProgram, "uPMatrix");*
  - Avec des tableaux, il faut en plus déclarer la présence du tableau :
    $\Rightarrow$ *gl.enableVertexAttribArray([...])*
  - Pour une image, passage des valeurs :
    $\Rightarrow$ *gl.uniformMatrix4fv([...], pMatrix);*

$\Rightarrow$ *Il s'agit du code de la fonction initShaders (transparent suivant)*

Déclaration des paramètres

tuto1

```
function initShaders() {
  var fragShader = getShader(gl, "shader-fs");
  var vertexShader = getShader(gl, "shader-vs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Could not initialise shaders");
  }

  gl.useProgram(shaderProgram);

  shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "position");
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
  shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");
}
```

Lecture des deux shaders de main.html, inutile de voir les détails, le code est donné...

Passage des valeurs

```
// ====================================================
function initVertexBuffers()
{
  vertices = [ -1.0, -1.0,  -1.0,     1.0, -1.0,  -1.0,
                1.0,  1.0,  -1.0,    -1.0,  1.0,  -1.0,
               -1.0, -1.0,   1.0,     1.0, -1.0,   1.0,
                1.0,  1.0,   1.0,    -1.0,  1.0,   1.0 ];
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numItems = 8;
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                         vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
  var indices = [ 0, 1, 1, 2, 2, 3, 3, 0,
                  4, 5, 5, 6, 6, 7, 7, 4,
                  0, 4, 1, 5, 2, 6, 3, 7 ];
  indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);
  indexBuffer.itemSize = 1;
  indexBuffer.numItems = 24;
}

// ====================================================
function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
}
```

Exemple avec le choix d'une couleur

- A l'initialisation du shader ⇒ *réservation d'une variable*

```
shaderProg.color = gl.getUniformLocation(shaderProg, "color");
```

- Pour passer une valeur ⇒ *envoi vers la carte*

```
gl.uniform3fv(shaderProg.color, [1.0,0.0,0.0]);
```

- Pour le vertexShader :

  ⇒ *rien à faire dans notre cas (fil de fer uniforme)*

- Pour le fragmentShader :

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  uniform vec3 color;
  void main(void) {
    gl_FragColor = vec4(color,1.0);
  }
</script>
```

⇒ *Tout le code dans le répertoire tuto5-CubeColor.*

Couleur par sommet / par face

- Parfois : couleur par face (ou groupe de faces)
- Parfois : couleur par sommet (et interpolation)
- Solution :
  - Passage d'un tableau de couleurs
    $\Rightarrow$ *à la manière des sommets, mais sans indexation*
  - utilisation d'une couleur par sommet
  - utilisation d'une couleur par face : répétition des couleurs par sommet
- Pour cela :
  - création d'un nouveau buffer (`createBuffer()`)
  - activation du buffer (`bindBuffer()`)
  - typage des données (`bufferData()`)
  - association des données (`vertexAttribPointer()`)
  - et utilisation dans le shader...

Couleur par sommet / par face

⇒ *initShaders()*

```
shaderProgram.position = gl.getAttribLocation(shaderProgram, "position");
gl.enableVertexAttribArray(shaderProgram.position);

shaderProgram.color = gl.getUniformLocation(shaderProgram, "color");
gl.enableVertexAttribArray(shaderProgram.color);

shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");
```

⇒ *initBuffers()*

```
var colors = [ 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
               0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0
               ];
colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
colorBuffer.itemSize = 3;
colorBuffer.numItems = 8;
gl.vertexAttribPointer(shaderProgram.color,
                       colorBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

Couleur par sommet / par face

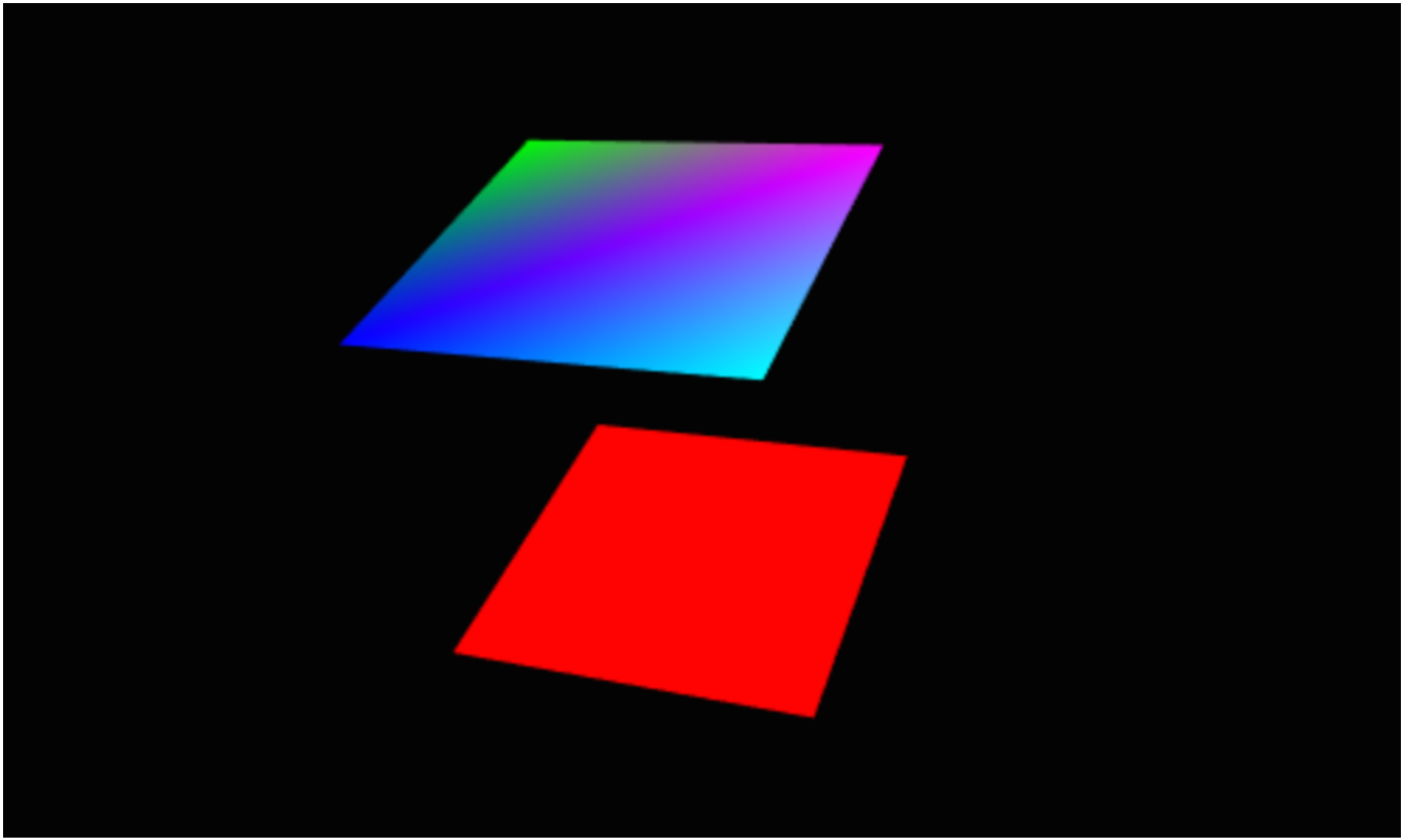
⇒ *vertex-shader*

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 position;
  attribute vec3 color;
  varying vec4 vColor;
  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(position, 1.0);
    vColor = vec4(color,1.0);
  }
</script>
```

⇒ *fragment-shader*

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec4 vColor;
  void main(void) {
    gl_FragColor = vColor;
  }
</script>
```

Résultat



⇒ *Complétez tuto5-CubeFilled pour faire apparaître les 6 faces*

Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
   - Principes généraux
   - Déclarations et initialisations
   - Utilisation dans les shaders
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Méthodologie

- Principe encore identique :
  - lecture d'une image dans un fichier (stockage en mémoire RAM)
  - déclaration d'un buffer
  - transmission sur la carte graphique
  - utilisation par les shaders
- Questions à traiter :
  - Definition des coordonnées des sommets 3D sur la texture
  - Lecture d'un fichier image
  - Commandes OpenGL pour le stockage
  - Commandes OpenGL pour l'activation
  - Utilisation par le shader
- Les textures restent en mémoire
- Nécessite d'activer une texture pour son utilisation

Déclarations pour les shaders

- Réservation des variables pour l'utilisation de la texture

  ⇒ *les coordonnées de texture pour le sommet courant*

  ⇒ *un sampler permettant de récupérer les données dans la texture*

```
shaderProgram.position = gl.getAttribLocation(shaderProgram, "position");
gl.enableVertexAttribArray(shaderProgram.position);

shaderProgram.texcoord = gl.getAttribLocation(shaderProgram, "texcoord");
gl.enableVertexAttribArray(shaderProgram.texcoord);
shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");

shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");
```

Initialisation (appelée par la fonction webGLStart())

- Paramètres de stockage des données de l'image
- Lecture d'une image dans un fichier (ici au format gif)

```
function initTexture()
{
  var texImage = new Image();

  texture = gl.createTexture();
  texture.image = texImage;

  texImage.onload = function () {
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, texture.image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.uniform1i(shaderProgram.samplerUniform, 0);
    gl.activeTexture(gl.TEXTURE0);
  }

  texImage.src = "crate.gif";

}
```

Initialisation dans la fonction initBuffers()

- Même méthode que pour les sommets et les couleurs avant
- Déclaration des coordonnées de texture par sommet

```
var texcoords = [0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0,
                 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0 ];
var texBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, texBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(texcoords), gl.STATIC_DRAW);
texBuffer.itemSize = 2;
texBuffer.numItems = 8;
gl.vertexAttribPointer(shaderProgram.texcoord, texBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

- Tout est prêt pour les deux shaders

  $\Rightarrow$ *Les images sont prêtes sur la carte graphique*

  $\Rightarrow$ *Les coordonnées des sommets sur les images des textures sont définies*

Texture et vertex shader

- Récupération des coordonnées de texture pour le sommet courant
- Utilisation du lissage par interpolation pour les coordonnées

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 position;
  attribute vec2 texcoord;
  varying vec2 vTexCoord;
  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(position, 1.0);
    vTexCoord = texcoord;
  }
</script>
```

⇒ *La valeur de vTexCoord est destinée au fragment shader*

⇒ *Il n'y a rien d'autre à faire : interpolation (comme pour les couleurs)*

Texture et fragment shader

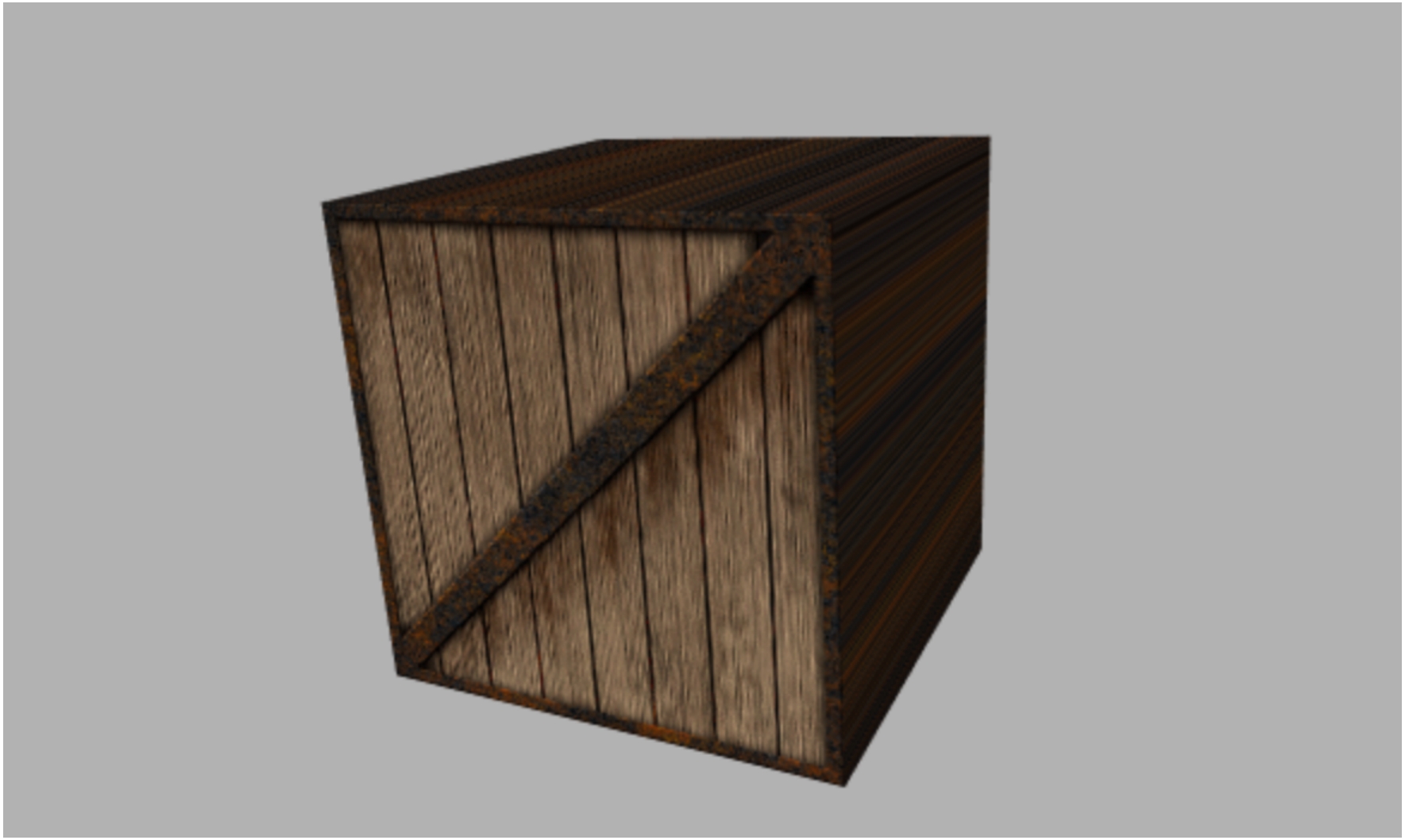
- Utilisation des coordonnées de textures pour le pixel courant
- Récupération effective de la valeur du pixel (sampler)

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec2 vTexCoord;
  uniform sampler2D uSampler;
  void main(void) {
    gl_FragColor = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));
  }
</script>
```

⇒ *La valeur finalement affichée est donnée par la texture*

⇒ *Les shaders restent très simples à écrire*

Résultat d'affichage



Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
    - Vertex buffer
    - Depth buffer
    - Fragment buffer
    - G-buffer
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Vertex Buffer Object (VBO)

- Il s'agit des buffers que nous avons déjà vus
- Pour le stockage des sommets
- Pour la représentation des indices

⇒ *Obligatoires pour WebGL, seule manière de faire*

⇒ *Manière différente versions précédentes (fonctions glVertex en C)*

⇒ *Maintenant obsolètes, mais souvent sur internet...*

Chargement d'objets OBJ

- Bibliothèque *Three.js*
- Regarder le chargement et la compatibilité

Depth/Z-buffer

- Fontionnalités

Fragment Buffer Object (FBO)

- Fonctionnalités

G-buffer et rendu en plusieurs passes

- Principe
- Fontionnalités

Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
    - Ombres dures
    - Ombres douces
12. Traitement d'images
13. Autres outils à voir

TITRE

TITRE

Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
    - Utilisation du Frame-Buffer
13. Autres outils à voir

Réaliser des opérations sur les images

Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir
    - Architecture des cartes
    - WebCL
    - Documentations

Processeurs disposés en parallèle

- Architecture dépendante des générations de cartes
- Organisation des processeurs en blocs
- Mémoire locale, par bloc et globale
- Calculateur très puissant
- Nécessite une bonne connaissance pour être optimal

Faire du calcul parallèle avec la carte graphique

Pour trouver des informations

- OpenGL : communauté très active (jeux, 3D, Web, etc.)
  http://www.linuxgraphic.org
  http://bittar.free.fr/OpenGL
  http://www.paulsprojects.net/
  http://nehe.gamedev.net/
  http://www.codesampler.com/oglsrc.htm
- Pour WebGL, plusieurs tutoriels complets

Sommaire

1. Objectifs du cours
2. Rappels
3. Contenu du cours
4. Pipeline graphique
5. WebGL - démarrage
6. WebGL - geometry buffers
7. Entracte IHM
8. WebGL - shaders
9. Gestion des textures
10. WebGL - buffers principaux
11. Gestion des ombres
12. Traitement d'images
13. Autres outils à voir

Géométrie

- Complétez les exemples avec le cube
  - ⇒ *Pour avoir un cube complet avec une couleur par face*
  - ⇒ *Pour avoir la texture correctement sur les 6 faces*
- Construisez et affichez une sphère 3D
  - ⇒ *Regardez comment réaliser un placage de texture*
  - ⇒ *Définissez également une normale par sommet pour le lissage*

Affichage

- Réalisez un affichage du cube en fil de fer

  $\Rightarrow$ *avec seulement les faces visibles*

- Essayez de généraliser avec n'importe que autre objet

Shaders

- Affichez le tampon de profondeur, avec des fausses couleurs
- Créez un shader permettant de représenter des objets brillants

  ⇒ *Utilisez le modèle de Phong modifié vu en cours les années précédentes*

  ⇒ *Eclairés par une source ponctuelle*

  ⇒ *Que faudrait-il faire avec un environnement lumineux ?*
- Réalisez une IHM pour modifier les paramètres de la BRDF

  ⇒ *Utilisez jQuery pour faciliter les choses*