WebGL par la pratique – erratum

Stéphane Gobron, Mario Gutierrez, PPUR // EPFL, 2015~2020

Liste des corrections relevées dans le livre.

Introduction

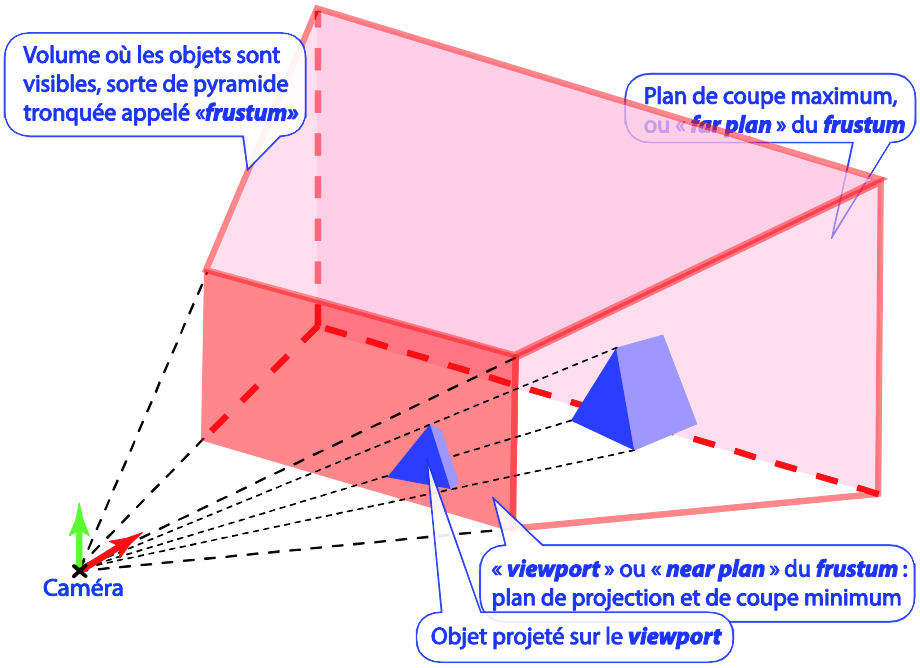
p.7, note de page, http://caniuse.com/#feat=webgl et non « canisue ».

Chapitre 1 ./.

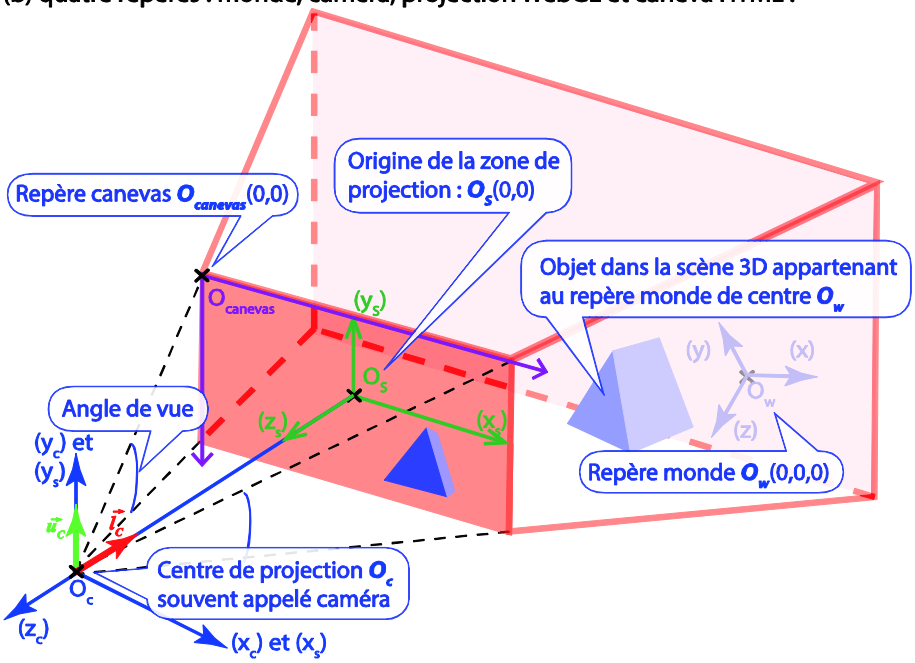
Chapitre 2

p.16, amélioration de la figure 2.2 :

(a) objet du repère monde au plan de projection :



(b) quatre repères : monde, caméra, projection WebGL et caneva HTML :



p.20, en bas de la page, la dernière ligne de la matrice de transformation n'est pas nulle :

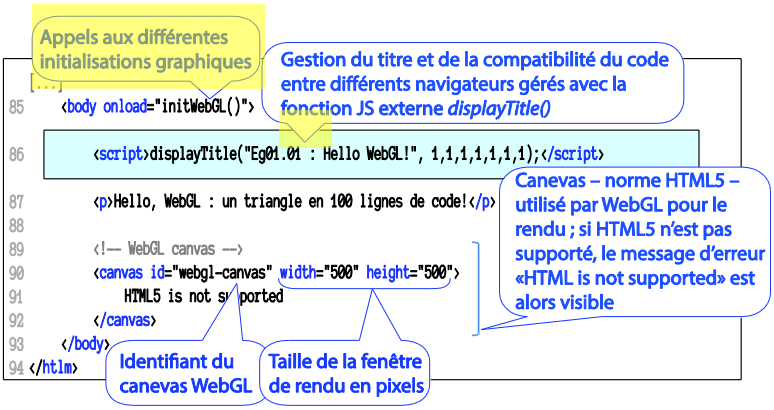
$$P'\begin{pmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{pmatrix} = \begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix} \cdot \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} = \begin{pmatrix} M_{0,0}P_x + M_{0,1}P_y + M_{0,2}P_z + M_{0,3} \\ M_{1,0}P_x + M_{1,1}P_y + M_{1,2}P_z + M_{1,3} \\ M_{2,0}P_x + M_{2,1}P_y + M_{2,2}P_z + M_{2,3} \\ 1 \end{pmatrix}$$

p.21, légende de la Figure 2.7 : « […] sur l'axe *y* avec D(~~0~~, d_y, ~~0~~) ➔ D(**1**, d_y, **1**) »

Chapitre 3

p.37, les deux bulles « voir code 3.4 » et « voir code 3.5 » sont interverties.

p.39, la première bulle était mal placée et l'appel aux différentes initialisations graphiques (concept très important) n'était pas souligné.



p.42, « Dans la première partie du code 3.7, on déclare et initialise trois tableaux » ➔ «Dans la première partie du code 3.7, on initialise trois tableaux **déclarés dans le code 3.5** ».

p.42, « Dans la deuxième partie du code 3.8… » ➔ « Dans la deuxième partie du **code 3.7**… »

Chapitre 4

p.55, Fig 4.2, la bulle « Programme Vertex Shader » ➔ « Programme **Fragment** Shader ».

p.57, amélioration d'une phrase : "[…] projette […]" ➔ **transforme les positions 3D repère monde en position 3D repère caméra** : ils se situent alors sur le *viewport* du *frustum* (voir Figure 2.2).

p.62, Fig 4.7, la sortie du fragment shader « gl_Fragment » ➔ « gl_**FragColor** ».

p.64, pour les normales (gl_normal) un *vec3* est suffisant.

p.65, Concernant les champs (x,y) de `gl_FragCoord` : "[…] des valeurs bornées de -1 à +1 :" ➔ "de **(0,0)** à **(canevasX, canevasY)** :"

p.65, en complément aux explications : "Notons que `gl_FragCoord` représente donc un variable qui contient le coordonnée du point relatifs au canevas. Son type est `vec4` : (`x, y, z, w`) donc quatre `float` avec `x` et `y` décrit ci-dessus ; mais que représentent les champs `z` et `w` ?

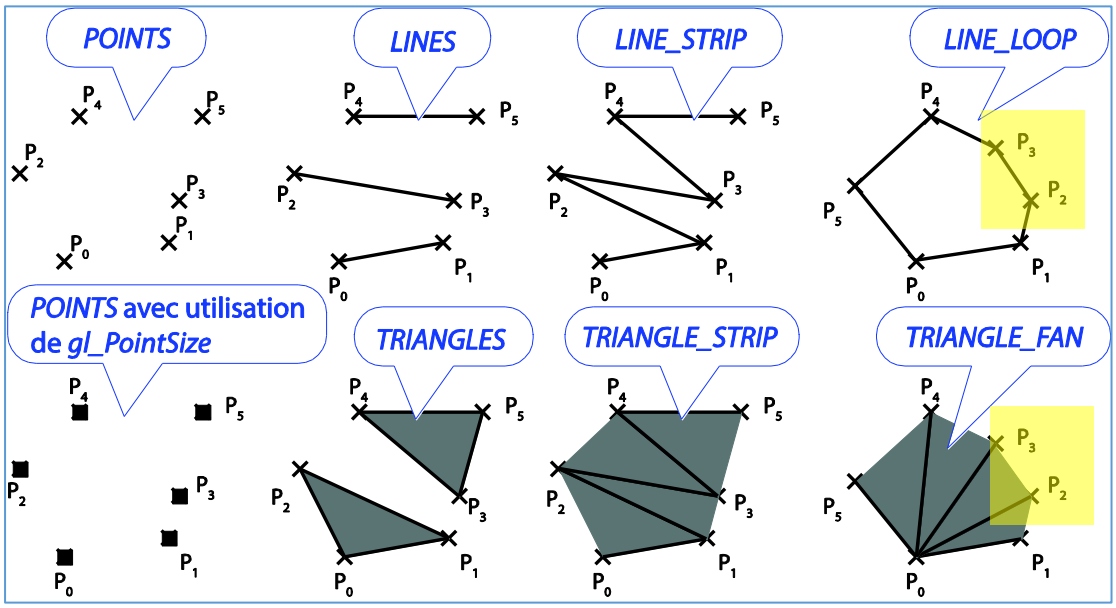
- son champ « `z` » a une valeur entre `0` et `1` avec '`0`' si le fragment est sur le « *near plan* » (*i.e.* plan de projection) et '`1`' s'il se trouve sur le « *far plan* ». Cette valeur n'est toutefois pas linéaire et pour utiliser la profondeur « `vDistToCam` » dans le **Fragment Shader** il faut ajouter au **Vertex Shader** les deux lignes :

```
varying float vDistToCam;
[…]
vDistToCam = gl_Position.z;
```

- son champ « `w` » est égale à (`1 / clip.w`), avec `clip.w` comme composant W de la position de l'espace de projection, on pourrait donc faire le lien entre *Vertex* et *Fragment* :

```
gl_Position.xyz = gl_FragCoord.xyz / gl_FragCoord.w
```

p.73, les points P2 et P3 étaient inversés dans les deux formes de droites (*i.e.* LINE_LOOP et TRIANGLE_FAN) :



p.76, 1ier point à retenir : "[…] en coordonnée 2D de l'écran → en coordonnée **3D dans le monde caméra correspondant pour (x,y) au monde** 2D de l'écran"

p.76, 2ème point à retenir : "[…] par deux matrices : « **projection matrix » (pMatrix) – généralement avec de la *perspective*** – et « model view matrix » **(mvMatrix), voir code 3.8**."

Chapitre 5

p.82, gestion des déplacements latéralement en appuyant sur "A" et "S" ➔ "A" et "D"

p.84, la bibliothèque externe *gl matrix* est passé de la version 1.x à la version 2.x ; dès lors la fonction *rotateModelViewMatrixUsingQuaternion()* du code 5.4 a changé de forme :

```
1  function rotateModelViewMatrixUsingQuaternion() {
2      let rx = degToRad(rotY);
3      let ry = degToRad(rotX)
4
5      rotYQuat = quat.create();
6      quat.setAxisAngle(rotYQuat,[0,1,0],ry);
7
8      rotXQuat = quat.create();
9      quat.setAxisAngle(rotXQuat,[1,0,0],rx);
10
11     myQuaternion = quat.create();
12     quat.multiply(myQuaternion,rotYQuat,rotXQuat);
13
14     rotationMatrix = mat4.create();
15     mat4.fromQuat(rotationMatrix,finalQuat);
16     mat4.mul(mvMatrix, rotationMatrix,mvMatrix);
17
18     rotX = 0.;
19     rotY = 0.;
20 }
```

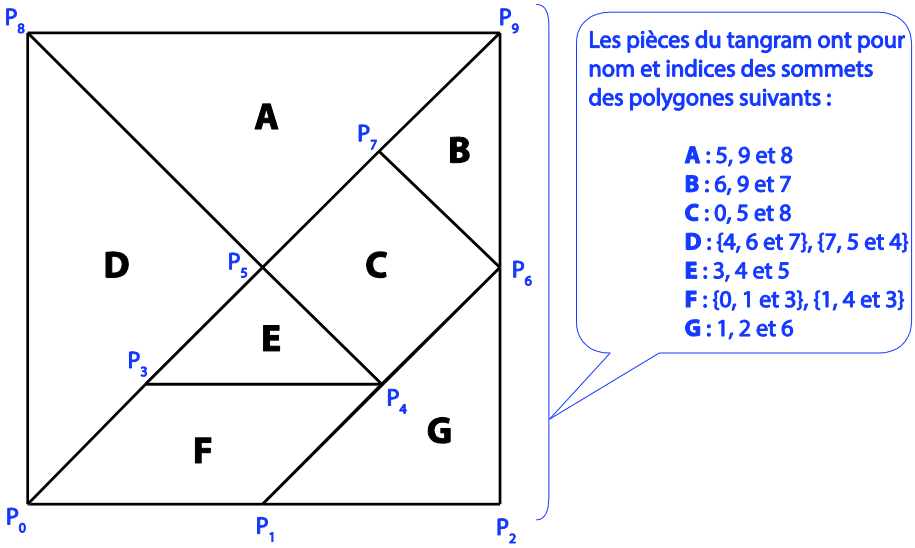
Utilisation des quaternions pour tourner la scène avec la souris

Créé le quaternion pour la rotation sur l'axe Y : [0,1,0] puis celui pour la rotation sur l'axe X : [1,0,0]

Multiplie les deux quaternions pour en faire un troisième puis modifie la matrice de vue avec le resultat

Ne pas oublier de remettre à zéro le déplacement, sans cela les rotations vont s'accumuler !

p.92, erreur dans l'ordre des indices de C et D (*i.e.* inversés) de la Figure 5.5.



Chapitre 6

p.110, 1ère ligne de la légende de la Fig.6.4 « quadratique » ➔ « **cubique** »

p.112, section *Courbe et points de contrôle* : « Le rendu à […] ce troisième paramètre est **non** nul si c'est un point de contrôle, […] son champ ~~*z*~~ ➔ ***x*** où est stockée la valeur de *t*. […] »

p.115, légende de la Fig.6.7, « D_0 » ➔ « Δ_0 »

p.116, dans l'exercice 6.3 « […] dans la section 6.~~*3*~~ ➔ 6.**2, p. 109** »

p.117, dans l'exercice 6.4, « […] de l'exercice 6.~~*4*~~ ➔ 6.**3** ».

Chapitre 7

p.125, paragraphe de bas de page, « Le lien […] est proposé code 7.~~*1*~~ ➔ 7.**4** […] »

p.129, Figure 7.4 : « A la ~~*troisième*~~ ➔**quatrième** …» ; Figure 7.5 : « A la ~~*quatrième*~~ ➔ **sixième** …»

p.131, Code 7.8, confusion entre X, Y et Z :

L'icosaèdre étant parfaitement symétrique, on peut déterminer l'ensemble des coordonnées avec seulement deux attributs *X* et *Z*

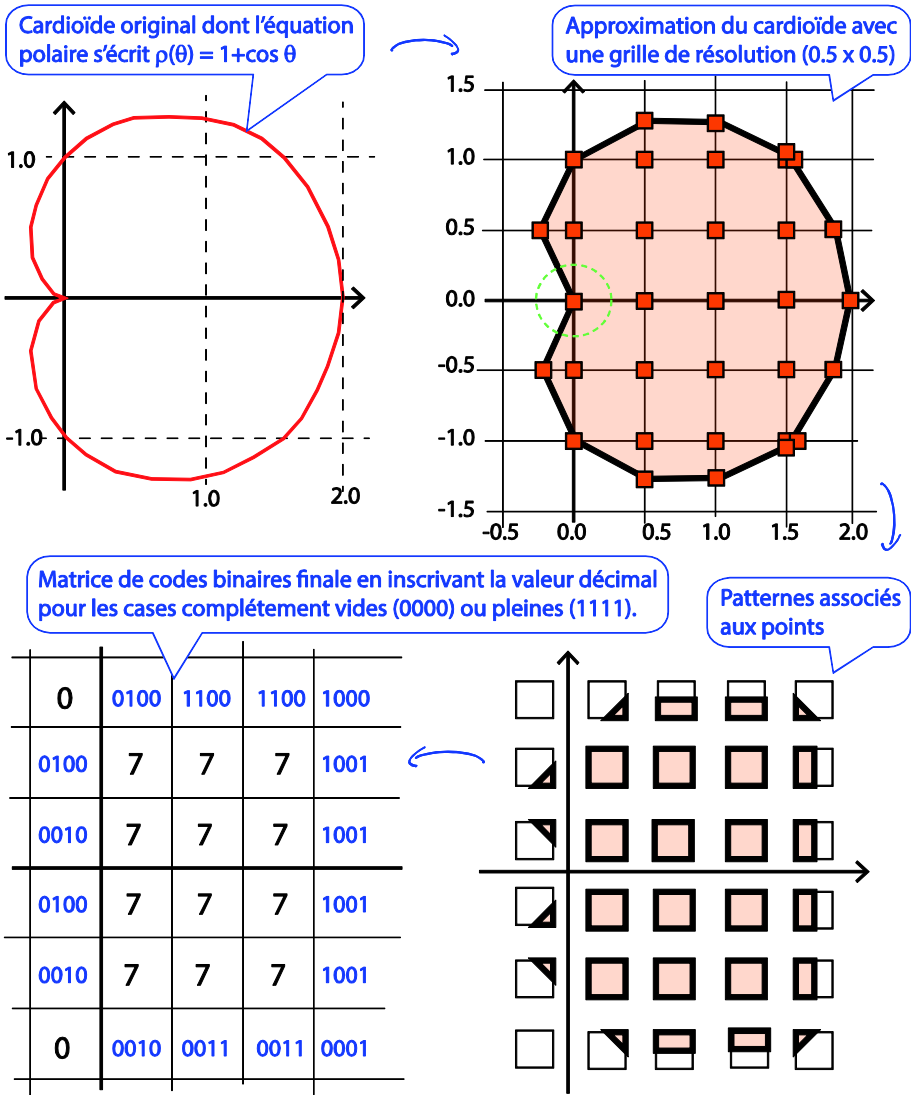
```
1  var X = 0.525731112119133696;
2  var Z = 0.850650808352039932;
3
4  // icosahedron  vertices
5  var vertex = [];
6  vertex.push(-X,0.0,Z);vertex.push(X,0.0,Z);  vertex.push(-X,0.0,-Z); vertex.push(X,0.0,-Z);
7  vertex.push(0.0,Z,X); vertex.push(0.0,Z,-X); vertex.push(0.0,-Z,X);  vertex.push(0.0,-Z,-X);
8  vertex.push(Z,X,0.0); vertex.push(-Z,X,0.0); vertex.push(Z,-X,0.0);  vertex.push(-Z,-X,0.0);
9
10 // icosahedron faces (indices)
11 var triangleIndice = [];
12 triangleIndice.push(1,4,0);  triangleIndice.push(4,9,0);  triangleIndice.push(4,5,9);
13 triangleIndice.push(8,5,4);  triangleIndice.push(1,8,4);  triangleIndice.push(1,10,8);
14 triangleIndice.push(10,3,8); triangleIndice.push(8,3,5);  triangleIndice.push(3,2,5);
15 triangleIndice.push(3,7,2); triangleIndice.push(3,10,7); triangleIndice.push(10,6,7);
16 triangleIndice.push(6,11,7); triangleIndice.push(6,0,11); triangleIndice.push(6,1,0);
17 triangleIndice.push(10,1,6);  triangleIndice.push(11,0,9); triangleIndice.push(2,11,9);
18 triangleIndice.push(5,2,9); triangleIndice.push(11,2,7);
```

Pour former un icosaèdre bien équilibré et ainsi respecter les proportions géométriques, il est impératif que X et Z respectent l'équation du nombre d'or « ϕ » : $(Z+X)/Z=Z/X\approx1{,}618$. On déduira $X\approx0.525$ et $Z\approx0.850$

Construction des 12 sommets de l'icosaèdre

Construction des indices pour les 20 futurs triangles définis par cinq rangées de quatre triangles : les 3x4x5 indices de l'icosaèdre

p.136, amélioration de la Fig. 7.10 (*i.e.* comment passer d'une courbe à une grille de code) :



p.137, remplacer ~~*montages*~~ par **montagnes**

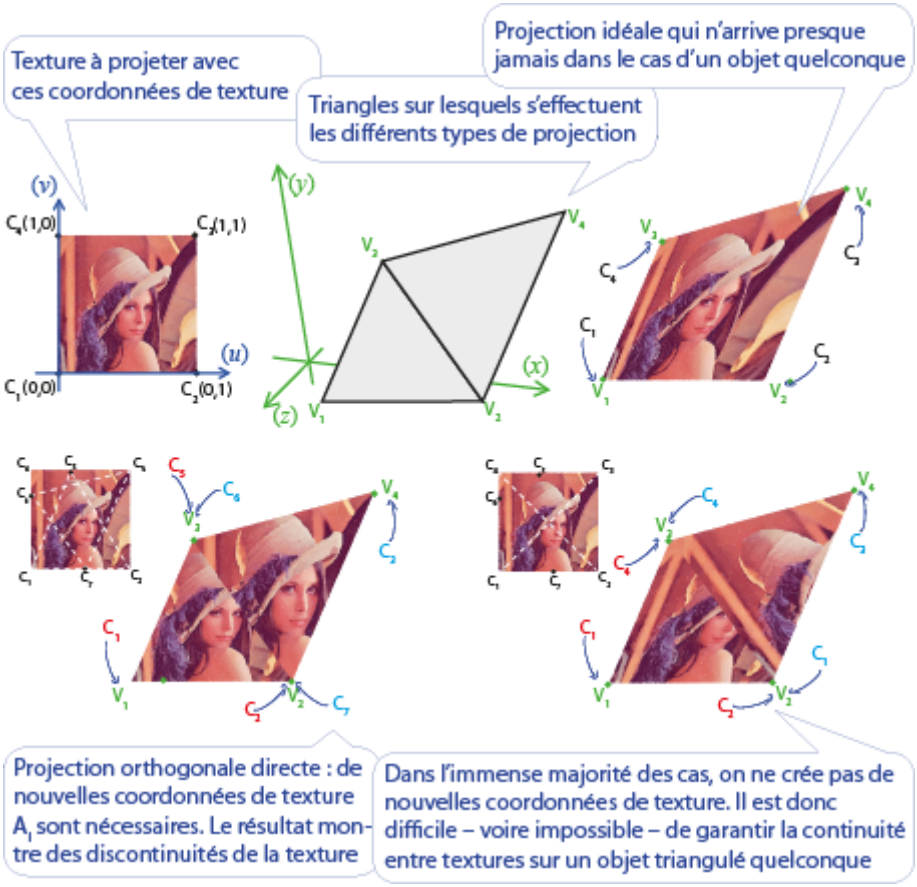
p.138, section "Résultats et discussion sur cette approche", « ~~*ligne du haut*~~ » → « **deux 1ier cas** ».

p.140, Fig. 7.15, bulle en bas à gauche : « i_a, i_d et i_s les intensités […] » →« I_a, I_d et I_s les intensités […] ».

p.140, dernière phrase, figures : 7.13 et 7.14 → 7.14 et 7.15.

Chapitre 8

p.161, amélioration de la Fig 8.7 – avec une correction sur l'association aux sommets –permettant de mieux distinguer coordonnées de texture et de sommets 3D et (2) :



p.162, deux dernières lignes, problème de police de caractère : « ~~p~~ » ➔ « π »

p.163, il manque un complément intéressant concernant le cas général pour passer de coordonnées cartésiennes (x,y,z) en coordonnée de texture (u,v) :

$$u = 0.5 + \frac{arctan2(z,x)}{2\pi}$$
$$v = 0.5 + \frac{\arcsin(y)}{\pi}$$

p.176, la figure « Code 8-8 » devrait être en jaune (*i.e.* Fragment Shader) et non en rose.

p.189, l'exercice 8.7 avait été retiré des corrections, son **énoncé** doit donc être **retiré**. Tous les numéros d'exercices sont décalés : Ex. 8.~~8~~ → Ex. 8.**7**, Ex. 8.~~9~~ → Ex. 8.**8**,… Ex. 8.~~13~~ → Ex. 8.**12**

Chapitre 9

p. 205, 2ème paragraphe : « […] Np1 = norm(AC' ~~^~~**x** AB+N); […] »

p. 210, référence au tutorial de Unity® : [8] → [28]

Chapitre 10 ./.

Solutions

p. 248, corrigé ex. 4.5: la Sol-Figure ~~4.3~~ → **4.4**