IUT - Département Informatique

LP Dawin - WebGL - 2017-2018

Semaine 1 - Travaux Pratiques

WebGL TP1

1 Premiers pas

Le but de cet exercice est de vous familiariser avec WebGL.

1.1 Création de la page HTML

Pour utiliser WebGL, vous avez besoin d'une page HTML à ouvrir dans un navigateur. Le squelette de la page comprend simplement une balise `canvas` (avec un `id` pour le retrouver plus tard) en plus des balises `html` et `body`, ainsi qu'un script javascript externe pour le code. Copier-coller le code suivant dans un fichier `tp1.html`

```
<html>
   <body onload='main()'>
       <canvas id='dawin-webgl' width=800 height=800>
         Utilisez un navigateur compatible avec WebGL
       </canvas>
       <script src='tp1.js'></script>
   </body>
</html>
```

1.2 Récupération du contexte WebGL

La première étape pour utiliser WebGL est de récupérer le contexte sur lequel on va pouvoir agir. Pour cela, il faut tout d'abord récupérer le `canvas`, puis le contexte WebGL attaché au `canvas` et vérifier qu'il a été reçu correctement :

```
var canvas = document.getElementById('dawin-webgl');
var gl = canvas.getContext('webgl');
if (!gl) {
      console.log('ERREUR : echec chargement du contexte');
      return;
}
```

Question Ouvrez la page dans un navigateur. Que constatez-vous ?

1.3 Effacer la zone de dessin

Le programme WebGL le plus simple qui soit se contente d'effacer la zone de dessin. Cela se fait en deux étapes :

1. Définir une couleur pour l'effacement avec `gl.clearColor`. Cette fonction prend 4 arguments : les valeurs des canaux rouge, vert, bleu et alpha.
2. L'effacement lui-même avec la fonction gl.clear. Son unique argument est le buffer à effacer. Ici, nous voulons effacer le buffer de couleur, l'argument est donc `gl.COLOR_BUFFER_BIT`.

Questions

1. Effacez la zone de dessin avec du noir. Quelles sont les valeurs des différents canaux ?
2. Changez de couleur d'effacement. Testez-en plusieurs. Trouvez notamment le blanc, le jaune, le magenta et le cyan.
3. Changez la valeur du canal alpha. Quel effet cela a-t-il ? Expérimentez avec différentes couleurs de fond pour le `body` (attribut `bgcolor`). Vous pouvez aussi mettre du texte ou un div sous le canvas.

2 Dessin de points

NB : Ce travail se fait à partir des fichiers `tp1.js` et `tp1.html`.

2.1 Rappels sur les shaders

Avant de pouvoir dessiner quoi que ce soit avec WebGL, vous devez charger les shaders. Les shaders sont de petits programmes qui s'exécutent sur la carte graphique, codés en GLSL. Ils sont de deux types en WebGL :

— Le **vertex shader** est exécuté pour pour chaque vertex que vous définissez.
— Le **fragment shader** (ou pixel shader) est exécuté pour chaque fragment (pour simplifier, un pixel).

Le code de chacun de ces shaders doit être compilé, puis linké, comme un programme C ou C++ classique. Cependant, cette étape a lieu à l'exécution du javascript (dans la fonction `main()`) et non en amont (avant l'exécution). L'ensemble d'un **vertex** et d'un **fragment shader** compilés et linkés forme un programme.

Nous définirons les shaders dans des fichiers séparés. Vous devez donc avoir quatre fichiers : un `.html`, un `.js` et les deux shaders (extension `.glsl`)

A faire Créer deux fichiers shaders avec le code minimal suivant :

```
void main() {}
```

2.1.1 Initialisation des shaders

Complétez la fonction `initShaders()` avec le code des réponses aux questions suivantes (toutes les méthodes appartiennent au contexte `gl`) :

1. Dans le fichier `tp1.js` fourni, lire attentivement et comprendre la fonction `loadText(url)`. Que fait cette fonction ? Quel paramètre faut-il passer lors de son appel ?
2. Créer un **vertex shader** et un **fragment shader** à l'aide de `createShader(type)`. L'argument type est `gl.VERTEX_SHADER` ou `gl.FRAGMENT_SHADER`.
3. Lier les fichiers à leur shader respectif à l'aide de la fonction `shaderSource(shader, source)`.
4. Compiler chaque shader à l'aide `compileShader(shader)`.
5. Créer un nouveau programme avec `createProgram()`
6. Attacher les deux shaders au programme avec `attachShader(program, shader)`.
7. Linker le programme (faire l'édition de liens) avec `linkProgram(program)`
8. Demandez à WebGL d'utiliser ce programme avec `useProgram(program)`.

2.1.2 Gestion des erreurs de compilation

Afin de débugger le code des shaders, voici comment vous assurer que la compilation fonctionne et, au besoin, récupérer les erreurs associées.

1. Pour les erreurs de compilation, après l'étape 4 de l'exercice précédent, vérifiez le statut de la compilation avec `gl.getShaderParameter(shader, gl.COMPILE_STATUS)`. En cas d'échec, il est possible de récupérer les erreurs avec `gl.getShaderInfoLog(shader)`.
2. De même, pour les erreurs de linking (après l'étape 7), utilisez `gl.getProgramParameter(program, gl.LINK_STATUS)` et `gl.getProgramInfoLog (program)`.

2.2 Point 2D simple

Maintenant que vous avez des shaders actifs, vous allez pouvoir (enfin !) dessiner un point. La fonction `drawArrays(mode, first, count)` déclenche le dessin. Son paramètre `mode` spécifie le type de dessin à effectuer, pour le moment vous utiliserez `POINTS` pour dessiner des points. Les paramètres `first` et `count` contrôlent ce qui est dessiné. Pour l'instant nous ne dessinons qu'un seul point, donc first=0 et count=1.

Complétez la fonction `draw()` avec le code des réponses aux questions suivantes :

1. Utilisez `drawArrays` pour demander le dessin d'un point. Le point apparaît-il à l'appel de cette fonction sur la page HTML ? Pourquoi ?

2. Dans le vertex shader, assignez aux variables `gl_Position` (de type vec4) et `gl_PointSize` (de type float) la position et la taille voulues respectivement, et dans le fragment shader, assignez à `gl_FragColor` (de type vec4) la couleur désirée. Pour `gl_Position`, on ne s'occupe pas du z et le quatrième paramètre w (facteur d'échelle) reste à 1.0. Le point apparaît-il ?
3. Essayez différentes valeurs pour les trois variables précédentes. Déduisez-en leur rôle.
4. Determinez les valeurs limites de `gl_Position` pour lesquelles le point est visible à l'écran.

2.3 Points 2D multiples

Nous allons maintenant dessiner plusieurs points. Pour cela, nous utiliserons un attribut pour modifier la position dans le **vertex shader**, afin de changer l'état de la variable `gl_Position`. Voici les étapes à suivre :

1. Ajoutez un `attribute vec2 position` dans le **vertex shader**. Sa déclaration doit se trouver avant le `main`. Pensez à modifier votre shader pour utiliser cette valeur dans `gl_Position`.
2. Récupérez l'adresse de l'attribut dans le programme avec `getAttribLocation(program, attribName)`
3. Passez une valeur à l'attribut avant de dessiner, à l'aide de `vertexAttrib2f(attrib, x, y)`

Questions

1. **Grille de points** Écrivez un programme qui dessine une grille de 9*9 points. Indice : utilisez (au moins une) une boucle for.
2. **Variations de couleur** Utilisez les coordonnées de chaque fragment pour modifier sa couleur, grâce à la variable `gl_FragCoord`. Cette variable globale, définie par le language, permet de récupérer les coordonnées du pixel traité à un instant donné. Dans quel shader faut-il faire les modifications ? Faites en sorte que les points les plus à droite soient plus rouges, et les points les plus à gauche soient plus bleus.
3. **Variations de taille** Utilisez les coordonnées de chaque fragment pour modifier sa taille. Dans quel shader faut-il faire les modifications ? Faites en sorte que les points les plus à gauche soient plus petits, et les points les plus à droite soient plus grands.
4. **Évènements avec la souris** Dessiner plusieurs points placés avec un clic de souris. Cela implique la gestion des évènements avec la fonction :

```
canvas.onclick = function(e){
   //extraire les coordonnees du clic
   //sauvegarder les positions du clic pour faire l'affichage ensuite
}
```

5. **Pour aller plus loin : suivre le point à la souris** Implémenter une fonction qui permette d'afficher un point qui se déplace sous le curseur de la souris dans le canvas.