Travaux Dirigés d'Infographie n°4

Cours de Synthèse d'Images I

—IMAC première année—

Textures

Ce TD nous permettra d'aborder le concept de texture qui est fondamental en infographie.

De manière intuitive, l'opération de texturage consiste à plaquer une image sur un objet. OpenGL nous permet de réaliser facilement cette opération. En 2D cela nous permettra d'afficher des images à l'écran en les plaquant sur des quads. Nous pourrons alors facilement leur appliquer des transformations en transformant le quad support. Les textures sont très utilisés dans les jeux vidéos et les films d'animation pour donner plus de réalisme aux objets.

Il existe un grand nombre de type de texture : textures 2D (images), textures 3D (volumiques), cube maps, textures procédurales, etc. Puisque nous travaillons en 2D, nous ne nous interesserons pour l'instant qu'aux textures 2D.

▸ Exercice 1. Chargement d'images avec la SDL et son extension SDL_image

La SDL permet de charger facilement des images au format BMP avec la fonction `SDL_Surface* SDL_LoadBMP(const char* filename)`. Celle ci prend en paramètre le chemin vers le fichier et renvoit un pointeur vers une structure de type `SDL_Surface` contenant l'image chargée en mémoire. La fonction renvoit `NULL` si l'ouverture a échouée.

En plus de cette fonction, une extension appelée `SDL_image` est installée sur les machines de l'université. Cette ci vous permet de charger beaucoup plus de formats d'images :

- TGA
- BMP
- PNM
- XPM
- XCF
- PCX
- GIF
- JPG
- TIFF
- LBM
- PNG

Deux conditions doivent être réunies pour l'utiliser :

- Ajouter l'option `-lSDL_image` dans la variable `LIB` du Makefile (ou sur la ligne de compilation si vous n'utilisez pas le Makefile).
- Ajouter `#include <SDL/SDL_image.h>` au début des fichiers sources utilisant l'extension.

Vous aurez alors accès à la fonction `SDL_Surface* IMG_Load(const char* filename)` qui fonctionne de la même manière que `SDL_LoadBMP`.

Lorsque vous n'avez plus besoin d'une image chargée, il faut demander à la SDL de libérer la mémoire allouée. Pour cela il suffit de passer à la fonction `SDL_FreeSurface(SDL_Surface* surface)` le pointeur vers la `SDL_Surface` contenant l'image.

☞ Ecrivez un programme qui initialise la SDL, charge une image, la libère et quitte la SDL. Pour tester si l'image à bien été chargée vous vérifierez si le pointeur renvoyé par la fonction de chargement est bien différent de `NULL`. Dans le cas contraire vous afficherez un message d'erreur.

▸ Exercice 2. Création d'un objet texture dans OpenGL

La deuxième étape consiste à demander à OpenGL un espace mémoire sur la carte graphique pour y placer notre texture. Cela se fait grace à la fonction `void glGenTextures(GLsizei n, GLuint* textures)`. Son premier paramètre correspond au nombre de textures que l'on désire. Le deuxième est un pointeur vers une zone mémoire pouvant contenir `n` entiers non signés (dans le cas ou $n = 1$ il suffit de passer l'adresse d'une variable de type `GLuint`, sinon il faut passer un tableau suffisement grand). Un appel à cette fonction entraine donc la crétation de `n` textures identifiés par des entiers non signés. Ces entiers sont placés par OpenGL à l'adresse `textures`. Ce sont les identifiants des textures, ils nous permettront de travailler sur nos textures.

Une texture OpenGL contient bien plus qu'une zone mémoire. Elle contient également différents attributs associés à la texture. Certains attributs possèdent des valeurs par défaut, d'autres doivent être fixés par le programmeur. Cela se fait grace à la fonction `glTexParameter`. Mais avant de pouvoir utiliser cette fonction, il faut binder la texture que l'on veut modifier.

En effet, en temps que machine à état, OpenGL considère l'identifiant de la texture sur laquelle on veut travailler comme un état possible. L'opération de binding consiste à attacher l'identifiant de la texture à un point de bind. Il existe plusieurs points de bind : `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, ... Il faut choisir le bon en fonction du type de texture que l'on désire utiliser. Vous aurez donc deviné que nous allons utiliser `GL_TEXTURE_2D`. Pour binder la texture il suffit alors d'utiliser la fonction `glBindTexture(GLenum target, GLuint texture)`. Le premier paramètre est le point de bind (`GL_TEXTURE_2D` donc) et le second l'identifiant de la texture que l'on désire attacher à ce point.

Une fois que notre texture est attachée, il faut modifier ses paramètres. Un seul paramètre doit absolument être fixé par le programmeur : le filtre de minification. De manière très simplifié, la minification est un phénomène qui se produit lorsque la primitive sur laquelle on plaque la texture devient plus petite que la texture (lors d'un dé-zoom par exemple). Dans ce cas OpenGL doit appliquer un filtre et il demande au programmeur de spécifier lequel. Nous allons utiliser un filtre linéaire, ce que l'on spécifie grace à l'appel suivant : `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);`. De manière générale la fonction à le prototype suivant : `glTexParameteri(GLenum target, GLenum pname, GLint param)`. Le premier paramètre est le point de bind sur lequel est attachée la texture à modifier. Le second est une constante OpenGL identifiant le paramètre à modifier. Enfin le dernier paramètre est la valeur à affecter. Vous pouvez consulter la documentation de OpenGL pour connaitre les différents paramètres existants (`http://www.opengl.org/sdk/docs/man/`).

Dernière étape pour que notre texture soit prête : l'envoie des données à OpenGL. L'image qui doit consituer notre texture est actuellement stockée dans la structure `SDL_Surface`. Il va falloir dire à OpenGL d'aller la chercher pour la copier dans l'espace mémoire associé à notre texture. Voici le prototype de la fonction à utiliser :

```
void glTexImage2D(
 GLenum target,
```

```
  GLint level,
  GLint internalFormat,
  GLsizei width,
  GLsizei height,
  GLint border,
  GLenum format,
  GLenum type,
  const GLvoid* data);
```

(je sais, une fonction avec autant de paramètres ça devrait être interdit...)

Voici une description des paramètres :

`target` est le point de bind sur lequel est attachée la texture (`GL_TEXTURE_2D` encore et toujours).

Vous pouvez mettre `level` à 0, nous ne l'utiliserons pas.

`internalFormat` correspond au format que doit utiliser OpenGL pour stocker les pixels de la texture en interne. Généralement on lui passe `GL_RGB` ou `GL_RGBA` si on veut pouvoir jouer avec la transparence. Utilisez `GL_RGB`.

`width` et `height` correspondent aux dimensions de notre image. Si le pointeur renvoyé par la fonction de chargement d'image est `surface`, alors les dimensions sont stockées dans `surface->w` et `surface->h`.

`border` peut être mis à 0 (il correspond à la taille du bord que l'on veut ajouter à notre image, ça tombe bien on n'en veut pas !).

`format` est le paramètre le plus compliqué à obtenir puisqu'il correspond au format des pixels non pas coté OpenGL (qui lui est spécifié par `internalFormat` et que l'on choisit) mais coté SDL, c'est à dire dans l'image chargée. Il va falloir passer une valeur différente selon que notre image soit codé sur 1 octet par pixel (image en niveau de gris), sur 3 octets par pixel (rouge-vert-bleu ou bleu-vert-rouge selon le processeur de votre ordinateur) ou encore sur 4 octets par pixels (RGBA ou ABGR, encore en fonction du processeur). Les cas les plus courants sont les deux derniers. Le dernier apparait pour les formats gérant la transparence (png). Voici le code permettant de déterminer le format à utiliser :

```
GLenum format;
switch(image->format->BytesPerPixel) {
  case 1:
      format = GL_RED;
      break;
  case 3:
      /* Ne gère pas les machines big-endian (à confirmer...) */
      format = GL_RGB;
      break;
  case 4:
      /* Ne gère pas les machines big-endian (à confirmer...) */
      format = GL_RGBA;
      break;
  default:
      /* On ne traite pas les autres cas */
      fprintf(stderr, "Format des pixels de l'image %s non pris en charge\n", IMAGE);
      return EXIT_FAILURE;
}
```

Le paramètre `type` correspond au type utilisé pour représenter une composante de couleur dans notre image. La SDL utilise des `unsigned char`. Il faut donc passer la constante `GL_UNSIGNED_BYTE` à la fonction (un `char` correspond à un octet, soit byte en anglais).

Enfin `data` est un pointeur vers les pixels de notre image (`surface->pixels` dans notre cas).

Pour récapituler, voici l'appel que l'on doit effectuer :

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image->w, image->h, 0, format,
  GL_UNSIGNED_BYTE, image->pixels);
```

A noter qu'une fois que vous avez envoyé les données de votre image à OpenGL, vous pouvez la détruire avec `SDL_FreeSurface` (sauf si vous en avez encore besoin).

Ensuite il faut dé-binder notre texture. Pour cela il suffit d'appeler à nouveau `glBindTexture` avec le même point de binding en premier paramètre et 0 comme identifiant de texture.

Pour finir, à la fin de l'execution de notre programme (avant le `SDL_Quit`) il faut libérer la mémoire que OpenGL à alloué pour la texture. Pour cela il suffit d'appeler la fonction `glDeleteTextures` avec les même arguments que `glGenTextures`.

☞ Modifiez votre programme pour créer et initialiser une texture OpenGL après avoir chargé votre image. Il faudra attendre l'exercice suivant pour tester si ça à fonctionné !

▸ Exercice 3. Plaquage de texture

☞ On y est presque ! Dans un premier temps ajoutez la boucle d'affichage à votre programme (après chargement et initialisation de la texture). Dans celle ci vous dessinerez un quad au milieu de l'écran.

Notre but à présent est de colorer notre quad non pas avec des couleurs que l'on spécifie avec `glColor` mais avec notre texture. Il faut donc dire à OpenGL d'activer les texturage 2D. Pour cela il faut placer `glEnable(GL_TEXTURE_2D)` avant le dessiner de notre quad et `glDisable(GL_TEXTURE_2D)` après. Ensuite il faut lui indiquer quelle texture il doit utiliser pour notre quad. Cela se fait en bindant à nouveau notre texture sur `GL_TEXTURE_2D`. On place donc un `glBindTexture(GL_TEXTURE_2D, texture)` juste après le `glEnable(GL_TEXTURE_2D)` et un `glBindTexture(GL_TEXTURE_2D, 0)` avant le `glDisable(GL_TEXTURE_2D)`. Cela donne le shéma suivant :

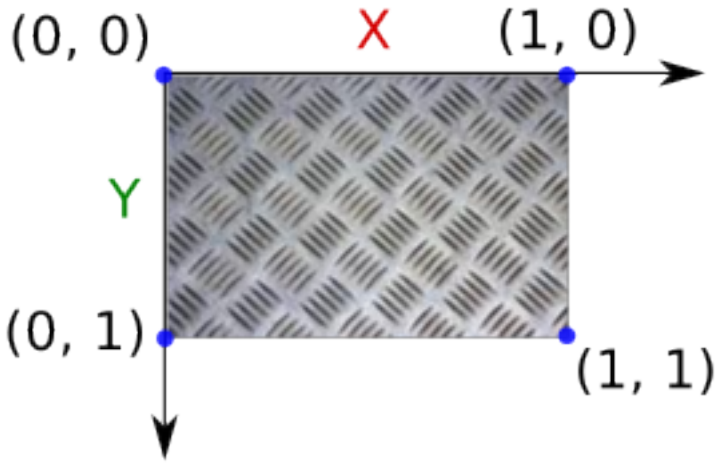
```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texture);

/* Dessin du quad */

glBindTexture(GL_TEXTURE_2D, 0);
glDisable(GL_TEXTURE_2D);
```

Remarquez à nouveau comment apparait la structure de machine à états d'OpenGL : on active des états (le texturage 2D et notre texture), on dessine, puis on désactive ces même états. Lorsqu'en plus de cela on ajoute des transformations, cela revient encore à modifier des états, dessiner, puis les restaurer (avec les `glPop`).

La dernière chose à faire pour voir votre image s'afficher à l'écran est la spécification des coordonnées de texture. Lorsqu'on dessinait en couleur, on spécifiait une couleur pour chaque sommet en faisant

un appel à `glColor` avant chaque appel à `glVertex`. L'idée est de faire la même chose ici mais en remplaçant `glColor` par `glTexCoord2f(x, y)`. Mais avant tout, qu'est ce que les coordonnées de texture ?

Lorsque vous donnez votre image à OpenGL, il lui associe un repère à deux dimensions de la manière suivante :



Les couleurs dans notre texture sont donc accessibles via des coordonnées (x, y) comprisent entre 0 et 1. Ce qui est génial c'est qu'on perd toute dépendance aux dimensions de l'image : c'est OpenGL qui se charge de faire la correspondance.

Il va falloir pour chacun des sommets de notre quad dire à OpenGL quelles sont ses coordonnées de texture dans l'image. Pour cela on utilise `glTexCoord2f`. Par exemple, imaginons que le coin bas gauche de mon quad soit placé en $(-0.5, -0.5)$ dans ma scène. Sur l'image ci dessus on voit que le coin bas gauche de la texture à pour coordonnées de texture $(0, 1)$. Il faut donc faire les appels suivants dans le code de dessin :

```
glTexCoord2f(0, 1);
glVertex2f(-0.5, -0.5);
```

A partir des coordonnées de texture des 4 points OpenGL se chargera de determiner les couleurs de chaque pixel couvrant notre quad.

☞ Modifiez le code de dessin du quad pour ajouter les coordonnées de texture. Testez votre programme pour constater que l'image s'affiche bien.

▸ **Exercice 4. Transformation !**

☞ Ajoutez la possibilité de tourner le quad avec la souris, ou encore de modifier sa taille afin de constater que OpenGL gère parfaitement les transformations appliquées sur la texture.