UNIVERSITÉ DU LITTORAL CÔTE D'OPALE

TP lancer de rayons 2
Image et caméra

Licence Informatique 3ème année

Année 2017-2018
durée : 3 heures

1 Introduction

Dans ce second TP, vous allez être amenés à enrichir votre application, de telle sorte qu'elle puisse gérer des images et les sauver. Dans un second temps, les classes nécessaires au lancement du calcul des images seront mises en place.

2 Représentation d'une image

2.1 La classe Image

On souhaite développer une classe permettant de représenter une image. Une image est un tableau 2D de pixels, chaque pixel étant représenté par une couleur. On vous propose la classe `Image` suivante (voir les explications qui suivent) :

```
class Image{
private:
  Couleur *pixel;// tableau contenant les pixels de l'image
  int largeur, hauteur;// largeur et hauteur (en pixels) de l'image
public:
  Image();
  Image(int largeur, int hauteur);
  ~Image();

  int getLargeur();
  int getHauteur();

  void setPixel(int x, int y, Couleur c);
  Couleur getPixel(int x, int y);

};// Image
```

Explications

- les attributs `largeur` et `hauteur` représentent respectivement la largeur et la hauteur de l'image ;
- l'attribut `pixel` représente l'adresse du premier octet de la zone mémoire qui doit contenir l'image. Les pixels de l'image sont organisés comme suit dans cette zone (voir figure 1) :
  - dans une ligne, les pixels sont rangés les uns après les autres en mémoire, le pixel d'indice 0 étant celui le plus à gauche et celui d'indice $largeur - 1$ le plus à droite ;
  - les lignes sont rangées les unes derrière les autres, la ligne 0 étant la première ligne stockée dans la zone et la ligne $hauteur - 1$, la dernière ligne stockée dans cette même zone.

Application : Développer le code de chacune des méthodes présentes dans la classe `Image` proposée ci-dessus. Vous utiliserez obligatoirement les attributs qui y sont inscrits. Vous prendrez garde aux risques d'erreur dans les méthodes suivantes :

- `setPixel()` : le changement de couleur d'un pixel ne peut être réalisé que si ses coordonnées sont valides ;

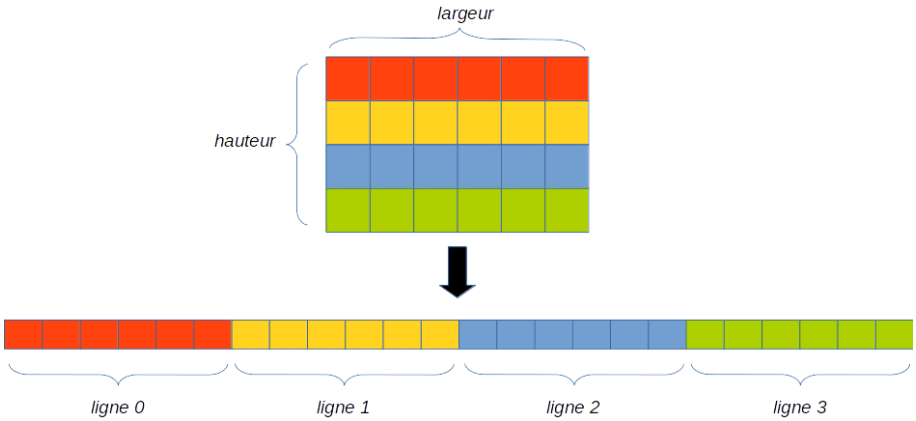

FIGURE 1 – Principe de stockage d'une image dans une zone mémoire 1D.

— `getPixel()` : si les coordonnées du pixel à examiner sont invalides, la méthodes retournera la couleur « noir ».

2.2 Sauvegarde d'une image

Lorsque l'on souhaite sauvegarder une image sur disque, il est nécessaire de sauver toutes les informations qui la composent, à savoir : ses dimensions (largeur et hauteur) et tous les pixels de l'image, rangés dans un ordre donné. Ceci permettra ensuite de savoir comment relire ces données afin de pouvoir recharger correctement l'image.

De nombreux formats d'images existent (png, jpeg, gif, etc.), chacun d'entre-eux ayant une manière différente de sauvegarder les données. Afin de les différencier au moment de la relecture de l'image, chaque fichier commence par un entête, spécifique au format choisi. Cet entête correspond aux tous premiers octets (caractères) sauvés dans le fichier et porte le nom de *nombre magique* ou encore de *signature*.

Dans ce TP, nous utiliserons un format très simple nommé `PPM`. Il permet, entre autres, de sauvegarder une image couleur dans un format texte, qui peut être facilement examiné et relu. Ce format est le suivant :

```
P3
largeur  hauteur
255
r00  v00  b00
r10  v10  b10
...
r01  v01  b01
r11  v11  b11
...
```

avec :

— `P3` : la signature d'un fichier de type `PPM` ;
— `largeur hauteur` : les valeurs (en format texte) des dimensions de l'image ;
— `255` : la valeur maximale pour chaque primaire (cette valeur peut être changée pour d'autres types de formats `PPM` qui ne seront pas abordés ici) ;
— r_{ij} v_{ij} b_{ij} : les trois couleurs d'un pixel, sachant que les pixels sont sauvés dans l'ordre croissant des lignes et, pour une ligne donnée, dans l'ordre croissant des colonnes.

Vous pouvez éditer si vous le souhaitez les fichiers `.ppm` fournis avec cet énoncé, afin de mieux comprendre l'organisation de ce format. Vous noterez que l'on peut avoir plus que les 3 valeurs de primaires par ligne ou au contraire une seule valeur de primaire par ligne, selon l'application qui a généré le fichier. De même vous noterez la présence, sur la ligne suivant la signature `P3`, d'une ligne commençant par le caractère `#` qui introduit un commentaire.

Application Complétez votre classe `Image` de telle sorte qu'elle dispose d'une méthode de prototype :

```
bool sauver(string filename);
```

permettant de sauver votre image au format `PPM`. Il pourra être utile de compléter la classe `Couleur` avec les méthodes suivantes : `int getRougei()`, `int getVerti()` et `int getBleui()`, qui permettent de renvoyer la valeur du canal de couleur correspondant sous la forme d'un entier compris entre 0 et 255.

On précise que lorsque la valeur d'un canal de couleur est supérieure à 1 lors de l'appel d'une de ces fonctions, sa valeur est supposée être 1.

Vous testerez ensuite cette méthode pour :

1. générer une image totalement rouge ;
2. générer une image totalement bleue, possédant une diagonale principale blanche.

Notez que vous pouvez vérifier que l'image est correcte, soit avec la prévisualisation de votre explorateur de fichiers, soit en tentant de l'ouvrir avec l'application de visualisation des images par défaut.

3 Représentation d'une caméra

Lors du calcul d'une image, il sera nécessaire de spécifier un certain nombre de paramètres, tels que la position de l'observateur ou encore la direction dans laquelle il regarde. Ces informations, et quelques autres qui évoluereont au cours des TP, seront gérées par la classe `Camera` qui doit à présent être développée.

Vous noterez les informations suivantes, illustrées par la figure 2 :

— le repère utilisé est le repère OpenGL, avec l'axe Oz orienté vers l'utilisateur ;
— l'observateur par défaut est positionné au point de coordonnées `(0.0, 0.0, 2.0)` ;
— l'écran par défaut se trouve dans le plan `z=0`, est de taille 2×2 et est centré par rapport à l'origine du repère.

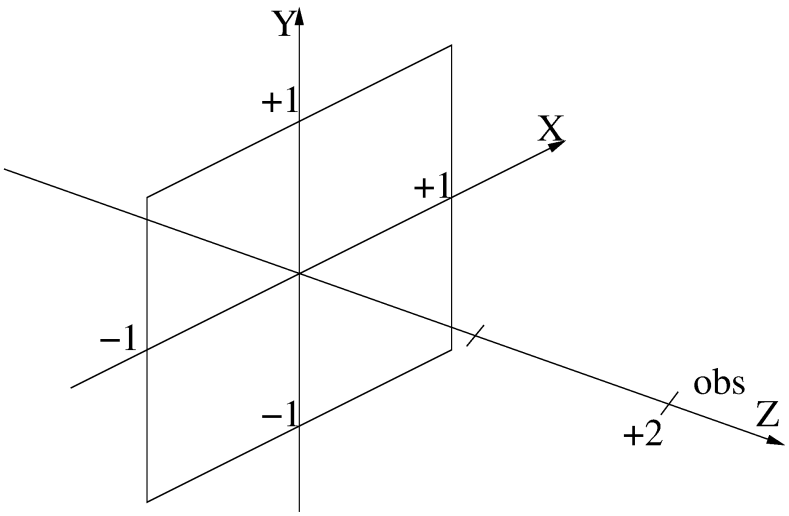

FIGURE 2 – Représentation de la géométrie par défaut d'une caméra.

3.1 La classe `Camera`

Dans cette première version très simplifiée, une caméra sera représentée par les attributs suivants :

— la position de l'observateur ;
— l'endroit où il regarde, qui sera considéré comme devant se trouver au centre de l'image à générer ;
— la distance du plan image par rapport à la position de l'observateur.

On considérera pour le moment que la caméra est toujours positionnée horizontalement par rapport au repère (l'axe horizontal de l'image sera l'axe Ox du repère) et que l'image à générer est carrée, de dimension 2×2.

Développer les premiers éléments de cette classe et vérifier leur fonctionnement.

3.2 Générer une image

Une image sera générée à partir des paramètres d'une caméra. Une méthode de calcul d'une image doit donc être ajoutée à cette classe et aura le prototype suivant [1] :

```
void genererImage(const Scene& sc, Image& im)
```

avec :

1. Le nom des classes des paramètres peut varier en fonction des noms que vous avez choisis dans votre application.

— `sc` la scène dont il faut effectuer le rendu ;
— `im` l'image à générer.

Ajouter cette méthode à votre classe, sachant que cette première version, elle se contentera de générer une image bleue avec une diagonale principale blanche. Tester cette fonctionnalité dans votre programme principal.

4 Représentation d'un rayon

Ecrire la classe `Rayon` qui permettra de représenter un rayon. On rappelle (voir cours) qu'un rayon dispose d'une origine et d'une direction. Vous pourrez éventuellement laisser ses attributs en mode public, afin de faciliter leur utilisation.

5 Genération des rayons primaires

Dans cette partie, vous allez être amenés à rajouter la boucle principale de l'algorithme du lancer de rayons dans la méthode `genererImage`. Cette boucle lancera les rayons primaire à partir de la position de l'observateur, au travers de chacun des pixels de l'image.

Calcul des coordonnées du centre d'un pixel

Lorsqu'un rayon primaire doit être lancé, il est nécessaire de connaître le point de l'écran par lequel il passe. À ce stade de l'application, on suppose qu'un seul rayon primaire est lancé par pixel et que ce rayon passe par le centre de chaque pixel (voir figure 3).

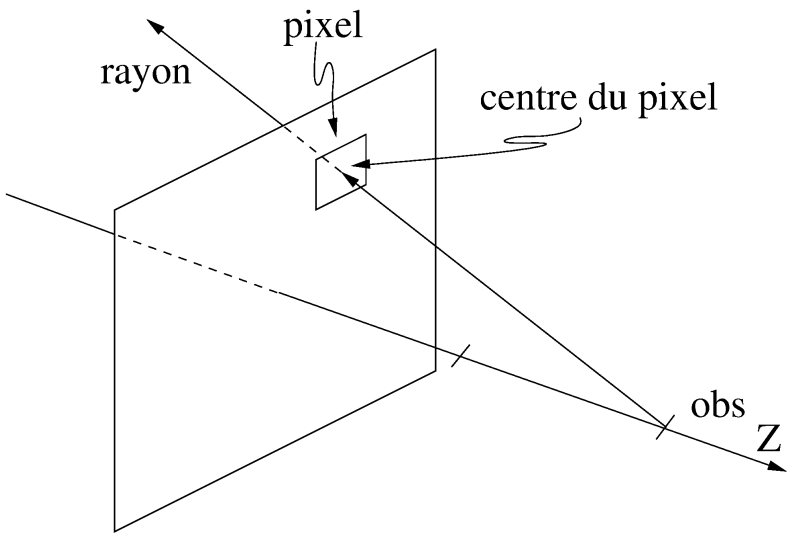

FIGURE 3 – Représentation de la géométrie par défaut d'une caméra.

Question 1

Donnez les équations permettant de calculer la largeur et la hauteur d'un pixel.

— `largeurPixel =`

— `hauteurPixel =`

Question 2

À partir de ces valeurs, donnez les équations permettant de calculer les coordonnées `milieuX` et `milieuY` d'un pixel de coordonnées image `(x,y)`. On précise que le pixel de coodonnées $(0, 0)$ se trouve en haut à gauche de l'image.

— `milieuX =`

— `milieuY =`

Calcul des rayons primaires

En utilisant les réponses aux questions précédentes, modifiez le code de votre fonction `genererImage()` de telle sorte qu'un nouveau rayon primaire soit généré pour chaque pixel de l'image. Vous afficherez le contenu du rayon créé pour les pixels `(0,0)`, `(12,7)` et `(101,200)` et comparerez les résultats obtenus à ceux qui figurent ci-dessous (Pour obtenir les mêmes résultats, vous vous assurerez que la résolution du l'image à calculer soit 256×256).

```
pixel (0,0)     : rayon = (0,0,2) -> (-0.996094,0.996094,-2)
pixel (12,7)    : rayon = (0,0,2) -> (-0.902344,0.941406,-2)
pixel (101,200) : rayon = (0,0,2) -> (-0.207031,-0.566406,-2)
```

6 Normalisation des vecteurs

Dans la plupart des calculs qui seront effectués dans la suite de cette application, les vecteurs utilisés doivent être normalisés (c'est à dire que leur norme - ou longueur - doit valoir 1). Modifier la classe `Vecteur` de manière à y ajouter une méthode permettant de normaliser le vecteur appelant. On rappelle :

— $v_{normalise} = \frac{v}{||v||}$

— $||v|| = \sqrt{v_x^2 + v_y^2 + v_z^2}$

Vous testerez cette nouvelle méthode sur les rayons primaires générés dans la question précédente : les 3 rayons vérifiés ci-dessus doivent fournir les résultats ci-après, après normalisation de leur direction.

```
pixel (0,0)     : rayon = (0,0,2) -> (-0.407183,0.407183,-0.81756)
pixel (12,7)    : rayon = (0,0,2) -> (-0.377935,0.394295,-0.837673)
pixel (101,200) : rayon = (0,0,2) -> (-0.0991082,-0.271145,-0.957423)
```