



R my Malgouyres,
Algorithmes pour la synth se d'images
et l'animation 3D,
DUNOD, 2002

TP n  10 : Algorithme du z -buffer

Objectifs :

Le but de ce TP est d'impl menter l'algorithme du z -buffer pour  liminer les parties cach es lors de l'affichage d'une sc ne 3D.

On commencera par se familiariser avec les classes `Scene3D` et `Objet3D`.

Il n'est pas n cessaire d'avoir r alis  le TP sur la mod lisation et facettisation, si l'on se limite   l'affichage de cylindres de r volution (le constructeur de cylindres de r volution est d j  impl ment ). Cependant, si l'on a impl ment  le constructeur, par exemple, de sph res dans le pr c dent TP, on pourra aussi afficher des sph res. De plus, la r alisation d'un constructeur d'objet 3D permet une familiarisation pr alable avec la classe `Objet3D` mise en jeu dans ce TP.

Pour visualiser par votre algorithme un objet dont vous n'avez pas impl ment  le constructeur, vous pouvez cr er cet objet avec l'ex cutable *French3D.exe* fourni, exporter cet objet au format *polytext*, puis importer le fichier *polytext* dans votre propre programme.

1 Les classes du fichier *Scene3D.h*

1.1 Listes cha n es d'objets 3D

Une sc ne 3D comprend notamment une liste d'objets 3D. Dans l'impl mentation propos e, un objet 3D est repr sent  par une instance de la classe `Objet3D`, qui a  t  vue lors de pr c dents TP.

La liste des objets sera en fait une *liste cha n e* de pointeurs sur des `Objet3D`. Pour cela, il nous faut une classe cellule de liste, qui sera la classe `CelluleObj`, comprenant un pointeur sur `Objet3D` et un pointeur `suiv` sur la cellule suivante. La liste cha n e elle m me sera impl ment e dans la classe `Listobj` ci-dessous. La classe `Listobj` est d j  impl ment e (voir le fichier `Scene3D.cpp`), il n'y a donc pas lieu de la r crire.

```
class Celluleobj{
    friend class Listeobj;
    friend class Scene3D;
    Objet3D *pobjet;    // pointeur sur un Objet3D
    Celluleobj * suiv;  // pointeur sur la cellule suivante

public:
    Celluleobj(void);
```

```

    Celluleobj(Objet3D * pobj);
};

```

Voici la classe "liste chaînée d'objets 3D" :

```

class Listeobj{
    friend class Scene3D;

    Celluleobj *L; // pointeur sur la tête de liste
public:
    // construction et destruction :
    Listeobj(){L = NULL;}
    ~Listeobj();
    Listeobj(Listeobj &);
    Listeobj & operator=(Listeobj &);
    void destruction(void);

    // insertion en tete de liste :
    void inseretete(Objet3D * pobj); // pobj doit etre alloue precedament
};

```

1.2 La classe Scene3D

La classe Scene3D contient tout ce qui est nécessaire pour représenter une scène 3D :

```

class Scene3D{
    // données nécessaires au stockage de la scène :

    Listeobj objets; // Liste des objets 3D de la scène

    ETC...

    // Donnees necessaires a l'algorithme du z-buffer :
    int dim2Dx, dim2Dy; // dimension de l'image calculée
    float ** minz; // z-buffer contenant le min de la profondeur z en chaque pixel
    // matrices pour stockage des couleurs :
    unsigned char **bufferR, **bufferG, **bufferB;
    Arete * tabarettes; // contient les arêtes d'un polygone 2D
                        // pour remplissage (nécessaire a l'algorithme du z-buffer)
    int nbmaxarettes; // dimension maximale du tableau tabarettes
    double *inclipx, *inclipy, *outclipx, *outclipy; // pour fenêtrage

public:
    ETC...

    // z-buffer dans le cas d'une projection en perspective
    void zbufferperspect(imcouleur& bitmap_dib); // affichage de la scène

    // Affichage d'un facette

```

```
void parcourspolyperspect(Facette& face, const Objet3D &objet, bool phong);
};
```

Le but du TP est d'implémenter la fonction `zbufferperspect`.

2 La fonction `zbufferperspect`

Dans la fonction `zbufferperspect`, on va tout d'abord remplir les matrices `bufferR`, `bufferG` et `bufferB` qui représentent l'image à calculer.

Comme on l'a vu en cours, après avoir initialisé les pixels à la couleur du fond et les éléments de la matrice `minz` à $+\infty$, il faut parcourir l'ensemble de toutes les facettes de tous les objets de la scène (parcours de liste chaînée).

Notons que toutes les matrices `bufferR`, `bufferG`, `bufferB` et `minz` sont déjà allouées (voir constructeur de `Scene3D`).

Il faut parcourir la liste des objets, et, pour chaque objet, parcourir le tableau des facettes de l'objet (voir classe `Objet3D`). Pour chaque facette, on appellera la fonction `parcourspolyperspect`, que l'on implémentera.

On transférera ensuite les couleurs de pixels dans un *Bitmap DIB* (classe `imcouleur`) passé en paramètre, en utilisant la fonction :

```
imcouleur : :setpixel(int x, int y, unsigned char R, unsigned char G, unsigned char B);
```

3 La fonction `parcourspolyperspect`

La structure de la fonction `parcourspolyperspect` est la suivante :

1. Initialisation des tableaux `inclipx` et `inclipy` (qui sont déjà alloués) avec les coordonnées des projetés des sommets de la facette. Le tableau `inclipx` contiendra les coordonnées x des projetés des sommets, et le tableau `inclipy` les coordonnées y . L'angle d'ouverture de la caméra est donné par `tabcam[nocamselect]->accesanglex()`
2. Fenêtrage avec en sortie les tableaux `outclipx` et `outclipy` (qui sont déjà alloués). Si l'on a réalisé le TP précédent sur le fenêtrage, on ajoutera les fichiers `fenetrage.h` et `fenetrage.cpp` au projet, et on utilisera ses propres fonctions pour le fenêtrage. Sinon, on pourra utiliser la fonction

```
void Scene3D::clip(int nbsommets, // nombre de sommets en entrée
                  double *tabx, double *taby, // inclipx et inclipy
                  int* outcount, // nombre de sommets nbaretes en sortie
                  double *taboutx, double * tabouty, // outclipx et outclipy
                  double xmin, // = -dim2Dx/2 dimensions
                  double ymin, // = -dim2Dy/2 de la
                  double xmax, // = dim2Dx/2 fenetre
                  double ymax) // = dim2Dy/2 graphique
```

qui se trouve dans le fichier `zbuffer.cpp`, qui est déjà implémentée, et qui réalise le fenêtrage par un autre algorithme.

3. Initialisation, à partir de `outclipx` et `outclipy`, du tableau `tabaretes` d'Arete (qui est déjà alloué), en vue d'appliquer l'algorithme de remplissage de polygone. On effectuera pour cela des arrondis sur les coordonnées des sommets.
4. Calcul des coefficients A, B, C, D de l'équation $Ax + By + Cz + D = 0$ de la facette. On pourra mémoriser les données calculées dans les données membres `normale.x`, `normale.y`, `normale.z` et `D` de la classe `Facette`.
5. Remplissage de polygone **non convexe** en copiant-collant le code de la fonction `Polygone2D : :remplissage` qui se trouve dans le fichier `Polygone2d.cpp`. On remplacera l'appel de la fonction `SetPixelV` par le traitement approprié du pixel.
6. Ajout du test pour calculer le minimum `minz[x][y]` des profondeurs en chaque pixel (x, y) , avec mise à jour des matrices `bufferR`, `bufferG`, `bufferB`. Pour le moment, on mettra comme couleur du pixel les données


```
objet.material.couleur.R,
objet.material.couleur.G,
objet.material.couleur.B
```

 (seule la silhouette des objets sera visible en attendant d'implémenter les modèles d'illumination au TP suivant).

Pour tester, on exécutera le programme, on créera une nouvelle scène (menu *File/New scene*), et on insérera un objet dont le constructeur a été implémenté (par exemple *Insert/Revolution cylinder*)

4 Antialiasage par suréchantillonnage

L'antialiasage par suréchantillonnage consiste à calculer un image dont les dimensions sont le double (double z -buffer) ou le triple (triple z -buffer) de la fenêtre graphique, pour obtenir (moyennant un temps d'affichage plus long) une image de meilleure qualité. Dans le programme des *TP*, les dimensions `dim2Dx` et `dim2Dy` correspondent à ces dimensions doublées ou triplées, selon l'option choisie par l'utilisateur.

Pour calculer la couleur effective d'un pixel du bitmap, on effectue des moyennes de 4 pixels des buffers dans le cas du double z -buffer, et de 9 pixels des buffers dans le cas du triple z -buffer.

Une variable `etatzbuffer` dans la classe `scene3D` vaut 1 pour le simple, 2 pour le double et 3 pour le triple z -buffer. Par ailleurs, les données `dimfenetrex` et `dimfenetrey` donnent les dimensions effectives de la fenêtre graphique (et donc du bitmap DIB passé en paramètre à la fonction `zbufferperspect`). On modifiera l'initialisation du bitmap DIB pour prendre en compte des valeurs de `etatzbuffer` différentes de 1.

Pour tester, on utilisera le menu *Display.Options antialiasage*.