Visualiseur 3D

Algo1

7 octobre 2013

1 Introduction

On se propose pour ce premier projet de réaliser un visualiseur de fichiers STL. Ce type de fichiers est utilisé notamment en fablab [1] pour stocker des objets à réaliser sur imprimantes 3D ou fraiseuses numériques.

Il existe bien sûr de nombreux visualiseurs open source sur internet mais nous nous proposons d'en créer un en Ada afin de mettre en œuvre les compétences acquises en TP.

Ce projet est également l'occasion de se familiariser avec un format de fichier utilisé en pratique et de vous prouver que l'on peut réaliser un projet 'conséquent' avec finalement peu de connaissances en programmation.

2 Fichiers STL

On peut depuis quelques années trouver de nombreux fichiers STL en ligne. Le site web Thingiverse propose ainsi un nombre important de fichiers au téléchargement.

Il existe en fait 2 formats STL différents. Un format 'ASCII', lisible facilement qui consomme beaucoup de place et un format "binaire" plus compact. Dans le cadre du projet, nous nous intéresserons uniquement au format ASCII.

On peut trouver une description de ce format sur wikipedia [2].

Comme le fichier est en ASCII, il est facile à lire ligne par ligne par exemple à l'aide de la fonction Get_Line.

Le fichier commence par un en-tête, indiquant le nom du modèle stocké.

Il continue ensuite par un ensemble de facettes sous le format suivant :

```
facet normal 1.147369e-02 5.757608e-01 8.175377e-01
  outer loop
    vertex 2.732172e+01 -1.023870e+01 1.118774e-03
    vertex 2.718306e+01 -1.023594e+01 1.118774e-03
```

1. http://fr.wikipedia.org/wiki/Fab_lab
2. http://fr.wikipedia.org/wiki/Fichier_de_st%C3%A9r%C3%A9olithographie

```
      vertex 2.719880e+01 -1.025195e+01 1.217291e-02
    endloop
  endfacet
```

Chaque facette est en fait un triangle apparaissant à la surface de notre objet. Un triangle est défini par 3 points, chaque point étant lui-même défini par 3 coordonnées flottantes. Chaque ligne commençant par *vertex* définit donc un point. L'ordre des points permet aussi de déterminer l'orientation de la facette (où se trouve le 'dessus' et le 'dessous'), cependant cette information n'est pas utilisée dans ce projet. Chaque facette contient également un vecteur normal dont nous n'aurons également pas besoin.

Attention, le nombre et la position des espaces sont variables d'un fichier à l'autre.

Enfin le fichier se termine par :

```
endsolid le_nom_du_modele
```

3 3D

3.1 Principe

L'algorithme pour effectuer le rendu d'une scène est très simple : on itère sur toutes les facettes. Pour chaque facette, on dispose d'un ensemble de 3 points 3D que l'on convertit en points 2D à l'aide d'une projection. La figure 1 en illustre le principe. On dispose d'une caméra (représentée ici par un œil) regardant dans une certaine direction. Un écran (ici une plaque de verre par exemple) est posé dans un plan orthogonal à la direction regardée. Les objets 3D forment une image 2D sur l'écran en étant projetés sur l'écran en direction de la caméra.

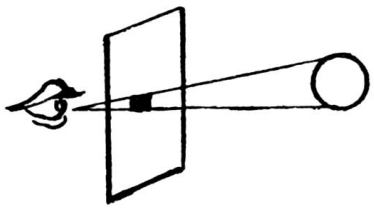

FIGURE 1 – De 3 à 2 dimensions

On obtient donc pour chaque facette un triangle dans le plan de l'écran. Si tous les points du triangle sont à l'intérieur de l'écran alors il suffit de dessiner les 3 segments formant les côtés du triangle grâce à trois appels à une procédure de tracé de segments.

On obtient ainsi un rendu en "fil de fer" permettant de visualiser l'ensemble de l'objet traité.

Le principe est donc très simple mais les opérations de projection sont un peu plus compliquées.

3.2 Projection

On dispose d'un repère formé de 3 axes : l'axe des X, l'axe des Y et l'axe des Z, qui correspondent aux 3 coordonnées de nos points. Nous avons en fait 2 de ces axes qui servent également pour la 2D : l'axe X indique le numéro de colonne d'un pixel et l'axe Y le numéro de ligne (comme dans le TP sur le feu). L'origine du repère est située en haut à gauche de l'écran. L'axe des X est orienté vers la droite, celui des Y vers le bas et celui des Z vers l'intérieur de l'écran.

Nous supposons dans un premier temps que la caméra est placée sur un point C de coordonnées $(c_x, c_y, c_z) = (0, 0, -R)$ $(R > 0)$ et regarde vers le bas (vers l'écran).

Nous diposons également d'un vecteur $E = (e_x, e_y, e_z)$ contenant une translation (e_x, e_y) à appliquer après projection ainsi qu'une valeur e_z de focale. Pour le projet, nous prendrons les valeurs : $(e_x, e_y, e_z) = (-400, -300, 400)$ qui permettent de visualiser le point $(0, 0, 0)$ au milieu de l'écran.

Soit $A = (a_x, a_y, a_z)$ un point que l'on cherche à projeter et $D = (a_x - c_x, a_y - c_y, a_z - c_z)$ les coordonnées de A dans le repère translaté sur la position de la caméra.

A est alors projeté sur l'écran en $B = (b_x, b_y)$ par les formules suivantes :

$$b_x = \frac{e_z}{d_z} d_x - e_x$$

$$b_y = \frac{e_z}{d_z} d_y - e_y$$

Notons au passage que les points tels que $d_z < 0$ ne doivent pas être affichés car ils se situent derrière la caméra.

3.3 Rotation de la caméra

En pratique, le visualiseur serait un peu inutile s'il se contentait d'afficher une image fixe.

On se donne donc la possibilité de déplacer la caméra. Les coordonnées de la caméra sont définies pour ce projet par 4 flottants : (R, ρ, θ, ϕ). En fait, on place la caméra sur une sphère de rayon R, centrée sur l'origine. Pour convertir les coordonnées (R, ρ, θ, ϕ) en coordonnées (c_x, c_y, c_z) on place la caméra en $(0, 0, -R)$ puis on effectue une rotation d'angle ρ autour de l'axe des X, suivie d'une rotation de θ autour de l'axe des Y, suivie d'une rotation de ϕ autour de l'axe des Z.

Quelle que soit sa position, la caméra regarde toujours le centre de la sphère. Placer une caméra de cette manière permet de tourner très facilement

autour de l'objet affiché en changeant simplement les valeurs des différents angles. On peut également éloigner ou rapprocher la caméra en changeant R.

La figure 2 illustre un exemple de déplacement de caméra. Pour simplifier le dessin, nous avons pris ici une valeur de 0 pour l'angle θ. L'écran est situé sur le plan (X, Y) horizontal.

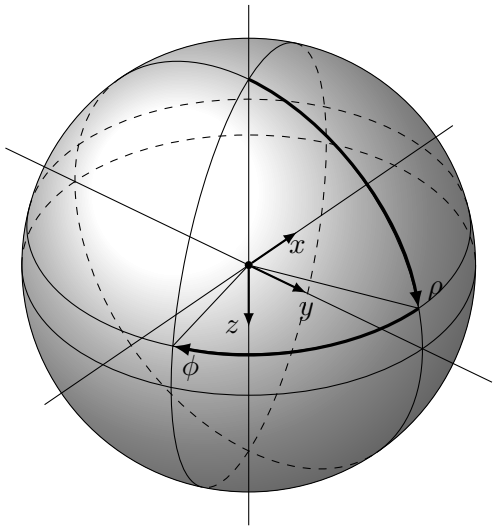

FIGURE 2 – Déplacement de la camera

À noter que nous vous demandons de réaliser les rotations à l'aide d'un produit matrice-vecteur.

3.4 Rotation des points

Il reste enfin à calculer les coordonnées des projections des points alors que la caméra peut désormais regarder dans n'importe quelle direction.

Pour ce faire, il suffit de se ramener au cas de la section 3.2 où la caméra regarde vers le bas.

Pour projeter un point $A = (a_x, a_y, a_z)$ il faut alors :

1. déplacer le repère de l'origine vers la caméra en calculant $P = (a_x - c_x, a_y - c_y, a_z - c_z)$.
2. effectuer les rotations inverses de celles de la caméra en calculant le point D de la manière suivante :
   — on part de P ;
   — on effectue une rotation de $-\phi$ autour de l'axe des Z ;
   — on effectue une rotation de $-\theta$ autour de l'axe des Y ;
   — on effectue une rotation de $-\rho$ autour de l'axe des X ;

3. une fois dans cette situation il suffit alors de terminer comme en section 3.2 en calculant :

$$b_x = \frac{e_z}{d_z} d_x - e_x$$

$$b_y = \frac{e_z}{d_z} d_y - e_y$$

Vous pouvez retrouver toutes ces informations en anglais sur wikipedia [3]. Attention néanmoins, les notations utilisées diffèrent au niveau des angles ($\theta_x = -\rho$, ...).

4 Fichiers fournis

Comme il s'agit de votre premier projet, nous vous fournissons un certain nombre de fichiers pré-remplis.

L'objectif est d'une part de vous guider un peu dans la structuration de votre projet et d'autre part de vous fournir une partie des procédures qui soient directement utilisables afin de réduire un peu la charge de travail.

Vous pouvez donc trouver dans l'archive fournie les fichiers suivants :

```
algebre.adb algebre.ads
dessin.adb dessin.ads
scene.adb scene.ads
frame.adb frame.ads
ligne.adb ligne.ads
stl.adb stl.ads
visualiseur.adb
```

Le fichier *visualiseur.adb* contient la procédure principale du projet. Il est copié sur le fichier utilisé lors du TP d'animation d'un feu car il utilise lui aussi la bibliothèque SDL pour l'affichage graphique.

Le visualiseur se compile donc par la commance *gprbuild visualiseur* (attention à ce que votre $PATH commence bien par */opt/gnat/bin*).

Il s'utilise ensuite de la manière suivante : *./visualiseur monfichier.stl* où *monfichier.stl* est le chemin vers le fichier à visualiser.

Pour nettoyer le dossier de tous les fichiers générés par la compilation, vous pouvez utiliser la commande *gprclean* (cela efface aussi votre exécutable *visualiseur*).

Regardons maintenant le paquetage *scene*. L'objectif de ce paquetage est de stocker les données de la scène rendue tout en permettant un accès facile. Le fichier *scene.adb* définit toutes les variables nécessaires pour stocker la

3. `http://en.wikipedia.org/wiki/3D_projection#Perspective_projection`

position de la caméra (R, Rho, Theta, Phi) ainsi qu'une variable M stockant l'objet à visualiser.

Ceci nous amène à parler des types proposés pour stocker nos données. Tout d'abord, le fichier *algebre.ads* définit le type Vecteur suivant :

```
type Vecteur is array(Positive range<>) of Float;
```

Ce type nous servira à stocker des points, qu'ils soient 2D ou 3D. La range d'un vecteur est donc variable, mais nous indicerons toujours le premier élément par '1' dans un souci de simplicité.

Le fichier *algebre.ads* définit également un type Matrice qui servira pour le calcul des différentes rotations.

Le fichier *algebre.adb* est à remplir et devra implémenter les calculs des différentes matrices de rotation ainsi que les calculs des projections.

Le fichier *stl.ads* se sert ensuite de vecteurs pour définir une facette comme un ensemble de 3 vecteurs et un maillage comme un pointeur vers un tableau de facettes.

```
type Facette is record
        P1, P2, P3 : Vecteur(1..3);
end record;

type Tableau_Facette is array(positive range<>) of Facette;

type Maillage is access Tableau_Facette;
```

On notera que le type Maillage est un pointeur sur un tableau de taille variable. La création d'un nouveau maillage se fait donc à travers une allocation mémoire. Pour allouer un maillage de 10 facettes, on utilisera par exemple :

```
M : Maillage := new Tableau_Facette(1..10);
```

Le fichier *stl.adb* contient déjà la partie concernant l'allocation du maillage.

Les accès aux éléments du tableau pointé par M (et à ses attributs) se font ensuite avec la même syntaxe que pour un tableau (M'First, M(I),...).

Le fichier *stl.adb* est à compléter pour contenir le code de chargement d'un fichier STL.

Enfin, la procédure principale pour l'affichage (Calcul_Image) se trouve dans *frame.adb*. À vous encore une fois de remplir cette procédure. Pour vous aider, vous disposez de la procédure Tracer_Segment du paquetage Ligne.

5 Travail attendu

5.1 Code

Vous devez fournir un visualiseur opérationnel, fonctionnant sur tout fichier STL respectant les spécifications.

Il est important de garder un code le plus clair possible et la clarté de ce que vous écrivez sera le critère le plus important pour la notation. N'hésitez donc pas à structurer votre projet correctement : pas de procédures trop longues ; des commentaires sur les parties complexes ; des noms de variables faisant sens ; un code indenté (attention : soit tabulations soit espaces) ; des paquetages permettant de bien organiser le projet.

Attention, n'attendez pas d'avoir terminé votre projet pour commencer à déboguer. Il vaut mieux vérifier au fur et à mesure ce que l'on fait afin de construire sur du solide. N'hésitez donc pas à créer des programmes de test.

Vous rendrez une archive contenant le code source et le rapport via *TEIDE*. Attention à ne pas inclure les fichiers issus de la compilation dans l'archive, les correcteurs recompileront votre code source pour obtenir l'exécutable. Attention aussi aux fichiers cachés (ceux dont le nom commence par un '.')

5.2 Rapport

Nous vous demandons 2 pages (max) de rapport au format *pdf* que vous incluerez dans l'archive que vous rendrez électroniquement sur *TEIDE*.

Le rapport a pour objectif de permettre aux correcteurs une plongée plus facile dans votre code. Vous expliquerez donc les choix réalisés qui diffèrent du squelette fourni.

5.3 Rendu et évaluation

Le projet est à rendre au plus tard le vendredi 25 octobre 2013 à 23h59. Tout retard est sujet à une pénalité.

L'accent du projet est mis sur la clarté. Il est possible de réaliser plus de fonctionnalités que prévues pour le projet (lecture de fichiers STL binaires, rendu "plein", meilleurs contrôles, animations,...). Néanmoins ces fonctionnalités ne seront pas un critère déterminant de votre note. C'est par contre l'occasion de vous faire plaisir ou de profiter du retour des correcteurs sur votre programme.

5.4 FAQ

Est-il possible de réaliser le projet seul ?

Oui, mais nous souhaiterions limiter le nombre de projets à corriger et donc nous n'accepterons que très peu de demandes.

Est-il possible de réaliser le projet à 3 ?

Non.

Est-il possible de réaliser le projet avec quelqu'un d'un autre groupe ?

Éventuellement. Préférez plutôt quelqu'un de votre groupe.

Est-il possible de modifier les fichiers .ads et .adb fournis ?

Oui. Pensez néanmoins à expliquer pourquoi dans votre rapport. La seule contrainte que nous posons est l'utilisation de produits de matrices pour réaliser les rotations.