

Création d'objets

Algo1

18 novembre 2013

1 Introduction

On se propose pour le second projet d'algo de continuer à travailler autour des fichiers STL mais cette-fois en s'intéressant à leur génération.

On cherche désormais à créer des objets de style "tour à bois". Pour ce faire, on part d'un profil que l'on fait tourner autour d'un axe. La figure 1 montre ainsi qu'il est possible d'obtenir des objets intéressants en quelques clics de souris.

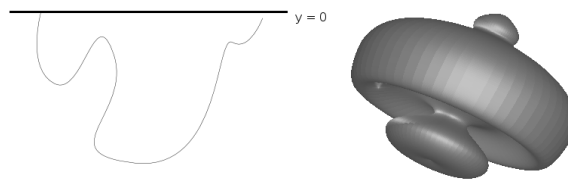


FIGURE 1 – Profilé; Objet obtenu en tournant autour de ($y = 0$)

Afin de faciliter le travail du concepteur, on utilisera le logiciel *inkscape* de dessin vectoriel pour le dessin du profilé.

Le profilé sera donc tracé sous *inkscape* à l'aide de l'outil "courbes de bézier". Cet outil permet aux graphistes de tracer de belles courbes à l'aide de très peu de points.

Le fichier obtenu sera ensuite sauvegardé au format *SVG* que vous connaissez déjà partiellement.

Notre programme devra donc simplement convertir ce fichier *SVG* en fichier *STL*.

2 Fonctionnement

Le filtre fonctionne en quatre étapes :

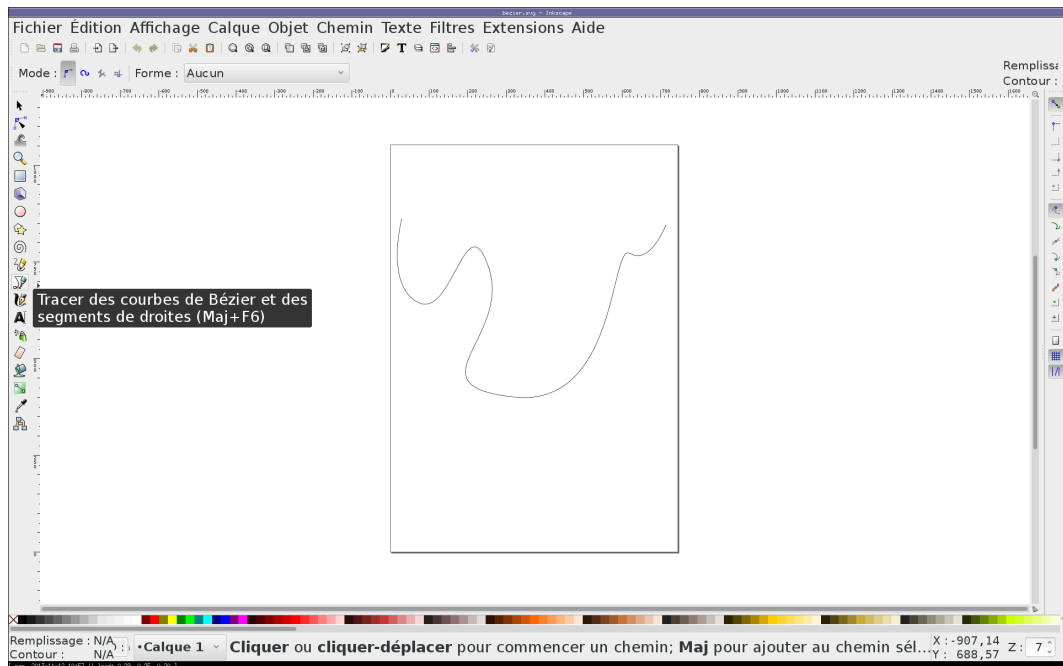


FIGURE 2 – Inkscape ; Courbes de bézier

- lecture du fichier SVG : extraction de la courbe de bezier ;
- conversion de la courbe en une liste de points (décrivant une ligne polygonale) ;
- conversion de la liste de points en une liste de facettes ;
- sauvegarde du fichier STL.

Le fichier *main.adb* que nous vous fournissons complet permet de réaliser l'ensemble de ces étapes.

2.1 SVG

La lecture d'un fichier SVG est une opération complexe qui nécessite un travail approfondi.

Dans le cadre de ce projet, nous n'allons par réellement lire tout type de fichiers svg mais en réalité seule une ligne du fichier *généré par inkscape* nous intéresse.

Ceci signifie que notre programme ne fonctionnera sans doute qu'avec inkscape (sans doute pas d'autre logiciels) et potentiellement uniquement avec sa version actuelle (0.48).

Prenons par exemple le fichier *bezier.svg* de la figure 2. Ce fichier de 60 lignes contient une très longue en-tête suivie d'un calque contenant un *chemin* dont voici le code SVG :

```
<path
  style="fill:none;stroke:#000000;stroke-width:1px;stroke-linecap:butt;stroke-linejoin:miter;stroke-opacity:1"
  d="m 30,190.93361 c 0,0 -45.714286,172.85714 42.857143,217.14286 C 161.42857,452.36218 197.14286,140....
  id="path2983"
  inkscape:connector-curvature="0" />
```

En réalité à l'intérieur du chemin, seule la ligne commençant par $d=$ nous intéresse.

Lire le fichier SVG consistera donc simplement à l'ouvrir, puis à lire les lignes une par une jusqu'à tomber sur une ligne commençant par $d=$. Cette ligne sera ensuite analysée en détails.

On supposera pour le projet qu'un fichier SVG "valide" ne contient qu'un seul chemin.

Un chemin SVG est composé d'une suite de lettres et de coordonnées de points en 2D. En fait, le chemin se lit de gauche à droite. Lorsque l'on rencontre une lettre, on en déduit comment interpréter toutes les coordonnées suivantes, jusqu'à la prochaine lettre. Par exemple, la lettre "m" (minuscule) indique que le couple de coordonnées suivantes (30,190.93361) forme le point de départ du chemin (en coordonnées relatives à la position courante). Ensuite la lettre "c" indique de lire les coordonnées 6 par 6. Chaque groupe de 6 coordonnées décrit 3 points (toujours en coordonnées relatives à la position courante) C_1, C_2, P . Ceci signifie que l'on a une courbe de bézier "cubique" du point courant au point P avec comme points de contrôles C_1 et C_2 (voir section 2.3).

Il existe une documentation importante des chemins SVG sur le web qui devrait amplement compléter cette rapide introduction.

On vous demande dans ce projet de supporter les lettres MLHVCQ (en minuscules et majuscules).

Le fichier *parser_svg.adb* est à compléter.

La **procedure** `Chargement_Bezier(Nom_Fichier : String; L : out Liste)` prend en entrée un nom de fichier et génère directement une liste de points approximant le chemin contenu dans le fichier.

2.2 Listes

Le projet va nécessiter des structures de données plus avancées que pour le projet précédent.

On se propose d'utiliser des listes simplement chaînées pour le stockage des différents objets manipulés (points et facettes). Les listes sont intéressantes ici car elles permettent de stocker un nombre quelconque d'objets; sont bien adaptées à des accès séquentiels et enfin se fusionnent facilement.

Nous travaillerons en Ada avec des listes génériques définies dans le fichier *liste_generique.ads*.

```
type Cellule ;
type Pointeur is Access Cellule ;
type Cellule is record
```

```

        Contenu : Element;
        Suivant : Pointeur;
    end record;
type Liste is record
    Debut, Fin : Pointeur := null;
    Taille : Natural := 0;
end record;

```

Une liste est définie de manière très classique comme 2 pointeurs vers les premières et dernières cellules; ainsi qu'un champ Taille contenant le nombre d'éléments à l'intérieur de la liste. Chaque cellule contient un élément ainsi qu'un pointeur vers la cellule suivante.

Les procédures à écrire sont toutes très simples, mais deux d'entre elles nécessitent peut-être une description plus détaillée.

Les procédures Parcourir et Parcourir_Par_Couples permettent de réaliser un traitement sur l'ensemble des éléments ou couples d'éléments d'une liste.

De telles procédures permettent de bien découper le travail sur les listes d'un côté et leur utilisation d'un autre côté.

La procédure Parcourir est déclarée de la manière suivante :

```

— appelle Traiter sur chaque element de L, dans l'ordre
generic
    with procedure Traiter(E : in out Element);
procedure Parcourir(L : Liste);

```

Elle a pour effet d'appeler la procédure générique Traiter sur chaque élément de la liste. L'utilisation d'une telle procédure nécessite donc une instantiation d'une procédure de la manière suivante :

```

package Liste_Points is new Liste_Generique(Point2D);
procedure A_Appliquer(P : in out Point2D) is
begin
    ...
end;
procedure Test is new Liste_Points.Parcourir (A_Appliquer);

```

Nous vous demandons pour ce projet de ne pas modifier le fichier *liste_generique.ads*.

2.3 Bézier

Nous utilisons potentiellement différents types de courbes de Bézier : cubiques et quadratiques.

Vous pouvez trouver une explication détaillée des courbes sur Wikipedia¹. Approximer une courbe de Bézier par une suite de segments est très

1. http://fr.wikipedia.org/wiki/Courbe_de_B%C3%A9zier

simple : il suffit de choisir le nombre n de segments désirés puis de faire varier t de 0 à 1 de $1/n$ en $1/n$ dans les formules proposées.

Le fichier *math.adb* implémentera les différentes procédures requises.

2.4 3D

2.4.1 Pré-Traitement

Avant de créer l'objet 3D, on effectue un léger pré-traitement sur les points des segments de manière à simplifier la vie du graphiste dessinant les courbes.

On calcule x_{\min} et y_{\min} l'abscisse et l'ordonnée minimales. On translate ensuite tous les points de $(-x_{\min}, -y_{\min})$. On rajoute, si nécessaire, un point au début (ou à la fin) de la liste afin de se raccorder à l'axe ($y = 0$). L'abscisse de ce nouveau point est la même que celle du premier (ou du dernier) point.

De cette manière on évite les "trous" au sommet ou en bas de l'objet réalisé.

2.4.2 Création

Le passage d'une liste de points à une liste de facettes est relativement simple. On travaille ici sur les couples de points voisins à l'aide de la procédure *Parcourir_Par_Couples* (voir section 2.2). Pour chaque point, on dispose de 2 coordonnées x et y . On choisit un pas m puis on fait tourner m fois chaque point autour de l'axe $y = 0$ (de $360/m$ degrés). Cette opération nécessite un peu de trigonométrie. On obtient ainsi deux "disques" en segments. Il suffit ensuite de relier les points (voisins) des 2 disques entre eux pour créer les différentes facettes.

Attention !

Dans un fichier STL les facettes ont une orientation. Ceci signifie qu'une face est orientée vers l'intérieur de l'objet et l'autre vers l'extérieur. L'orientation est déterminée par l'ordre des sommets. La facette (A, B, C) est donc la même que la facette (B, C, A) mais différente de (C, B, A) .

À vous de générer une orientation correcte pour vos facettes. On peut facilement vérifier que tout est ok dans un visualiseur de fichiers STL (remplissant les surfaces).

La génération de l'objet 3D est à réaliser en complétant la procédure *Creation du paquetage STL*.

2.5 STL

Le format STL est le même que pour le premier projet. On se permettra néanmoins de ne pas spécifier le vecteur normal à une facette ; la plupart des visualiseurs se passant de cette information.

Une facette sera donc déclarée de la manière suivante :

```

facet
  outer loop
    vertex 11.740260089957 -0.177550051035522 -0.102508569759991
    vertex 11.740260089957 -0.165862350009804 -0.120506051077033
    vertex 11.675108660976 -0.397138211716509 -0.229287853440013
  endloop
endfacet

```

3 Travail attendu

3.1 Code

Le code doit être fonctionnel, clair et concis. Les contraintes posées sont de travailler avec des listes et de ne pas modifier l'interface de programmation du paquetage de listes.

Nous vous conseillons *fortement* d'écrire des programmes annexes permettant de tester indépendamment les différentes procédures et fonctions. Ces programmes vont permettront ainsi de déboguer votre programme principal au fur et à mesure. N'hésitez pas à inclure vos tests dans les fichiers rendus ainsi que dans votre rapport.

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 ``'')

3.2 Rapport

Nous vous demandons 2 pages (max) de rapport au format *pdf* que vous inclurez 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.

3.3 Rendu et évaluation

Le projet est à rendre au plus tard le vendredi 13 décembre 2013 à 23h59. Tout retard est sujet à une pénalité.