Création d'objets

Algo1

14 décembre 2015

1 Introduction

On se propose pour le second projet d'algo de continuer à travailler autour des fichiers *SVG* mais en les utilisant cette fois ci comme entrées.

Nous générerons en sortie des objets en 3D au format *STL*. Ce format de fichier est un format simple, très utilisé, notamment comme format d'entrée pour les imprimantes 3D.

On cherche ici 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.

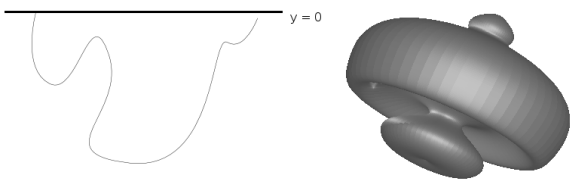

FIGURE 1 – Profilé ; objet obtenu en tournant autour de $(y = 0)$

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*.

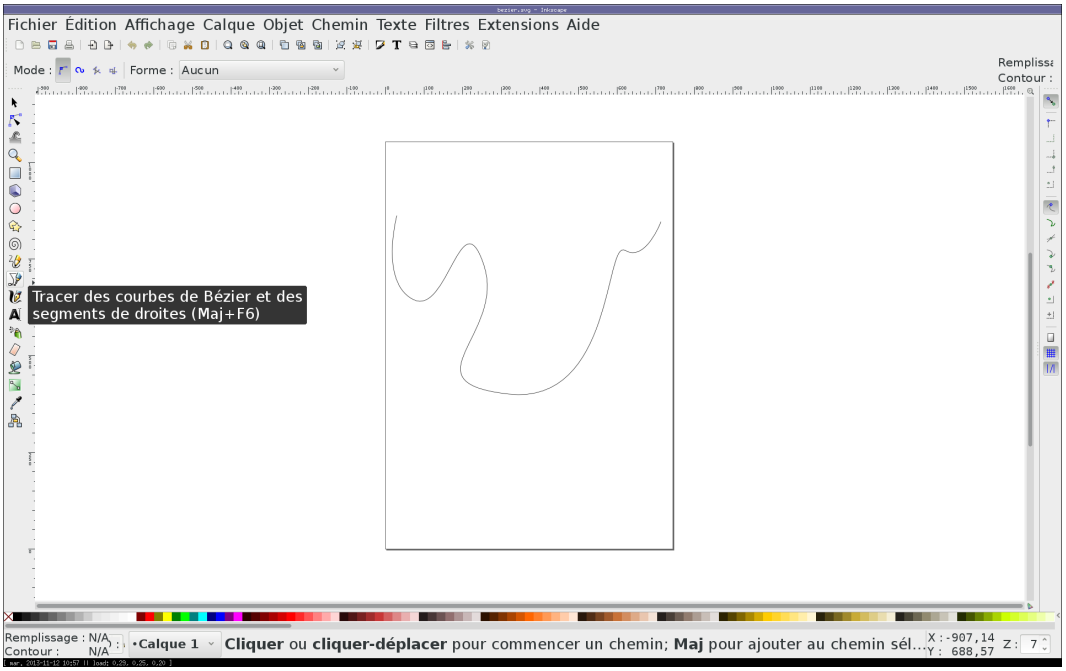

FIGURE 2 – Inkscape ; courbes de Bézier

2 Fonctionnement

Le filtre fonctionne en quatre étapes :

— lecture du fichier SVG : extraction de la courbe de Bézier ;

— 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 réllement 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 (0.91).

Prenons par exemple le fichier *bezier.svg* de la figure 2. Ce fichier de 60 lignes contient un très long 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 détail.

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. Le chemin se lit de gauche à droite. Lorsqu'on recontre 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 complémenter 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 chainées 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 ; elles 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 première et dernière cellules, ainsi qu'un champ Taille contenant le nombre d'éléments 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 une description plus détaillée.

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

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'appeller la procédure Traiter sur chaque élément de la liste. Son utilisation nécessite donc une instanciation 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 Wikipedia [1]. Approximer une courbe de Bézier par une suite de segments est très simple : il suffit de choisir le nombre n de segments désirés puis de faire varier t de 0 à 1 avec un pas de $1/n$ 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 et/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.

On évite ainsi les "trous" aux extrémités 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.

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

2.5 STL

Il existe différents formats de fichiers *STL*. Nous utiliserons ici le format le plus simple [2], fonctionnant à l'aide de fichiers texte.

Un fichier *STL* contient un objet 3D. Cet objet est composé d'un ensemble de facettes, chaque facette étant composée de trois points dans l'espace.

Le fichier se découpe en une en-tête, un corps de fichier contenant toutes les facettes et enfin un pied de fichier.

1. `http://fr.wikipedia.org/wiki/Courbe_de_B%C3%A9zier`
2. `https://fr.wikipedia.org/wiki/Fichier_de_st%C3%A9r%C3%A9olithographie`

L'en-tête est une seule ligne *solid nom_du_modele* et le pied de fichier est également très simple avec une ligne *endsolid nom_du_modele*.

Pour les facettes, on se permettra ici une simplification en spécifiant uniquement les coordonnées des trois points la composant. (on ne spécifie pas le vecteur normal, de nombreux 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
```

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 correct dans un visualiseur de fichiers STL (remplissant les surfaces). On peut par exemple utiliser le logiciel *stlviewer*[3].

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 indépendemment les différentes fonctionnalités. 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 incluerez dans l'archive rendue sur *TEIDE*.

3. `https://github.com/vishpat/stlviewer`

Le rapport a pour objectif de permettre aux correcteurs une plongée plus facile dans votre code. Vous expliquerez donc les choix effectués, ainsi que les éventuelles modifications du squelette fourni.

3.3 Rendu et évaluation

Le projet est à rendre au plus tard le vendredi 20 janvier 2016 à 23h59. Tout retard fera l'objet d'une pénalité.