Dossier n°5

Dossier final

mardi 26 mai 2009

Projet ***Mono****fin*

4ième année Informatique

INSA Rennes

YOANN CHAUDET
PAUL GARCIA
QUENTIN GAUTIER
NICOLAS LE SQUER
NICOLAS MUSSET
XAVIER VILLOING

Encadreurs

PATRICE LEGUESDRON
LAURENT MONIER
FULGENCE RAZAFIMAHERY

Sommaire

Table des figures

I. Introduction

Ce rapport est le cinquième qui est rédigé dans le cadre de notre projet : Monofin. Après les dossiers de pré-étude, spécifications, planification et conception, ce rapport a pour objectif de dresser le bilan du projet.

Le projet a connu quelques changements vis à vis des dossiers de spécification et de conception. Ces changements sont inhérents à la vie de tout projet et ont été minimisé dans la mesure du possible. Les changements sont intervenus de part ces différents points :

- Les réunions avec les utilisateurs pour discuter de leurs attentes
- Les remarques qui nous ont été faites à la première soutenance orale du projet
- Les itérations successives dans le cycle de développement du projet

Dans un premier temps le rapport présentera les modifications de certaines parties de notre application en terme de spécifications comme de conception. Pour terminer, avant de conclure, le rapport rendra compte brièvement de la planification et de la gestion de projet en corrigeant ce qui avait été planifié tout au début.

II. Interface graphique

L'interface graphique a beaucoup changé depuis la spécification. En effet, suite à une remarque pertinente faite durant notre première soutenance orale de projet, nous avons décidé de laisser tomber le coté « assistant » qui devait guider l'utilisateur à travers toute l'application a disparu. L'assistant était trop étouffant pour les utilisateurs et une conception plus classique a été choisie.

Il a été également choisi d'offrir la possibilité aux utilisateurs de pouvoir travailler sur plusieurs monopalmes à la fois. On offre ainsi à l'utilisateur une interface multi-fenêtrée (MDI : *Multiple Document Interface*). Ce qui lui dispense d'ouvrir plusieurs fois notre application. Ce changement a eu quelques influences sur la répartition des fonctionnalités dans l'interface. Par exemple, il fallait décider si la barre d'outils devait être commune à toutes les monopalmes ou si chaque monopalme devait gérer sa propre barre d'outils. Pour des raisons de simplicités d'intégration, c'est la deuxième solution qui a été préférée bien qu'elle induise une charge mémoire un peu plus importante.

Une autre choix a été fait, il n'a qu'une importance mineure sur le fonctionnement de l'application mais il existe et il est important de le citer : l'interface est disponible en plusieurs langues. La souplesse de la librairie Qt dans le domaine de la localisation permet d'accomplir cet aspect de manière assez aisée. Au chargement, l'application détecte la configuration du système, si une la langue est disponible dans les traductions, alors on l'utilise, sinon l'anglais est choisi par défaut. La solution choisie pour gérer les différentes langues est évolutive : il est possible d'ajouter une nouvelle langue sans avoir à recompiler le programme. Cette action doit néanmoins être faite par une personne possédant les sources du programme et sachant se servir de l'outils de traduction de Qt (*Linguist*). Le choix d'avoir une interface traduisible nous a permis de respecter une règle que nous nous sommes imposé lors de la conception : coder et commenter en anglais uniquement. De cette manière nous n'avons pas de mélange français, anglais dans les sources du programme et cela est plus claire.

L'interface a suivi les évolutions apportées aux différents modules de l'application et a été modifiée en conséquence. Entre autre, il a fallu ajouter les boutons nécessaires pour accéder aux nouvelles fonctionnalités, ce qui a induit l'écriture de nouvelles fonctions dont la plupart ne sont que des interfaces entre l'application graphique et les différents modules. Au final l'interface graphique a changé sur la forme depuis la rédaction des spécifications, le fond reste relativement identique. Pour plus de détails sur la nouvelle interface, il faudra consulter le manuel utilisateur qui sera rédigé ensuite.

III. Zone de dessin

La « Zone de dessin » représente la partie graphique dans laquelle l'utilisateur peut dessiner le contour de sa palme à l'aide de lignes droites et courbes. Les strates ne sont pas abordées dans cette annexe. Il s'agit d'une partie développée d'abord en autonomie par rapports aux autres, testée, puis intégrée au reste de l'application au fur et à mesure, tout en réalisant des tests unitaires et globaux à chaque intégration.

A. Les fonctionnalités

La plupart des fonctionnalités présentes dans le rapport de spécification ont été gardées, mais certaines ont dû être légèrement modifiées, voire supprimées pour des raisons pratiques ou par manque de temps. Par contre, certaines fonctionnalités non présentes dans le rapport ont été ajoutées parce qu'elles ont été jugées utiles ou importantes, et oubliées dans le rapport.

Voici tout d'abord une liste des fonctionnalités présentes dans le rapport de spécifications, partie « Modélisation de la palme », uniquement pour ce qui concerne la partie dessin du contour, avec leur situation actuelle. On rappelle que la partie dessin devait être constituée d'une zone de dessin possédant une grille en fond pour aider visuellement l'utilisateur, ainsi qu'une éventuelle image d'arrière-plan que celui-ci pourrait choisir, et que cette zone de dessin devait pouvoir afficher des lignes reliées par des points d'intersection. Ces lignes pouvaient être des droites ou des courbes de Bézier manipulables par un point de contrôle.

1. Modification des fonctionnalités spécifiées

1. Barre d'outils
   - 1.1. Prévisualiser la monopalme en 3D

     Conservée, voir « Spécifications et attentes » à la page 23.
   - 1.2. Afficher ou maquer la grille de sélection

     Conservée.
   - 1.3. Annuler la dernière action

     Conservée.
   - 1.4. Répéter une action annulée

     Conservée.
   - 1.5. Déplacer la vue dans la fenêtre de dessin

     Il s'agissait d'avoir une « main » pouvant « accrocher » la vue pour la déplacer. Cette fonctionnalité a été supprimée par manque de temps, l'utilisateur peut toujours déplacer la vue en utilisant les barres de défilement.
   - 1.6. Modifier le zoom sur la fenêtre de dessin

     Conservée.
   - 1.7. Choisir une image d'arrière-plan

     Conservée.

1.7.1. à 1.7.6 Choisir une image, la redimensionner, la déplacer, la tourner, la supprimer, l'intégrer à l'arrière-plan de la partie dessin

Tout est conservé, mais pas dans une fenêtre différente, c'est-à-dire que toutes ces opérations peuvent se faire directement dans la même fenêtre que celle où l'utilisateur dessine.

3. Fenêtre de dessin

3.0. Objets manipulables de la fenêtre de dessin

À l'origine, quatre types d'objets devaient être manipulables par l'utilisateur : les points d'intersection, les lignes (segments ou courbes de Bézier), les points de contrôle des courbes de Bézier, et les tangentes. Il est apparu qu'il n'était pas nécessaire de pouvoir manipuler les tangentes, cela aurait de plus, compliqué grandement l'implémentation.

3.1. Dessiner une ligne brisée

Conservée telle que décrite dans le rapport de spécifications. L'utilisateur place un premier point sur l'axe de symétrie, place ensuite ses autres points, formant des lignes droites, et place son dernier point sur l'axe de symétrie.

3.2. Sélectionner un point délimiteur (d'intersection)

Conservée, l'utilisateur peut même en sélectionner plusieurs.

3.3. Sélectionner un point délimiteur (d'intersection)

Conservée, l'utilisateur peut en déplacer un ou plusieurs, dans les limites de la zone de dessin.

3.4. Insérer un point délimiteur (d'intersection)

Conservée, l'utilisateur peut choisir de conserver ou non des courbes de Bézier s'il insère un point sur une telle courbe.

3.5. Sélectionner une courbe

Annulée, jugée inutile, puisque les points de contrôle apparaissent en permanence. Toutefois, lorsque l'utilisateur insère un point d'intersection ou de contrôle, les segments et courbes de Bézier apparaissent en surbrillance lorsque la souris passe dessus.

3.6. Déplacer une tangente

Comme précisé dans les objets manipulables (point 3.0), les tangentes ne sont pas manipulables directement, elles le sont toutefois via point de contrôle.

3.7. Déplacer un point de contrôle

Conservée, l'utilisateur peut déplacer un point de contrôle dans la limite de la zone de dessin.

3.8. Insérer un point de contrôle

Conservée, l'utilisateur clique sur une ligne droite, ce qui place automatiquement un point de contrôle en son centre, point qui peut être bougé par la suite.

3.9. Supprimer un point de contrôle

Conservée, la courbe redevient une ligne droite.

3.10. Aligner deux tangentes

Conservée, quand l'utilisateur sélectionne un point d'intersection, il peut le faire s'aligner avec les points de contrôle des deux lignes autour, ou bien avec les points d'intersection de ces lignes si elles ne possèdent pas de point de contrôle. Il faut noter que cette fonctionnalité marche si plusieurs points sont sélectionnés, mais qu'il est déconseillé de l'appliquer dans ce cas (l'algorithme est travaillé, seulement, si trop de points sont sélectionnés, le résultat n'est souvent pas digne d'intérêt).

2. Nouvelles fonctionnalités ajoutées

- Effacer l'intégralité du dessin pour recommencer : fonction compatible avec le *undo/redo*, on peut donc l'annuler ;
- Simplifier la vue : permet d'avoir une vue d'ensemble de l'intégralité de la palme (symétrie incluse) ;
- Agrandir ou réduire la taille de la scène : si la zone de dessin est trop petite ou trop grande, on a possibilité de la modifier ;
- Agrandir ou réduire la taille des carreaux de la grille ;
- « Aimanter » les points d'intersections à la grille, c'est-à-dire que si cette fonction est activée, les points que l'on va créer ou déplacer vont automatiquement se placer aux intersections des carreaux de la grille. Cela permet notamment de créer des lignes parfaitement horizontales ou verticales ;
- Sélectionner plusieurs points avec la touche « CTRL » ou avec un rectangle de sélection.

Toutes ces fonctionnalités ont été travaillées pour être le plus confortable possible pour l'utilisateur, sachant dans quel contexte l'application est utilisée. Toutefois, toutes les fonctions d'un éditeur de dessin ne sont pas présentes, mais il s'agissait ici de concevoir une application restreinte et surtout spécifique à un problème, ce qui fait que les fonctionnalités minimales étant présentes, l'application est tout à fait fonctionnelle pour son utilisation.

B. Structure de données

La conception de la partie dessin de monopalme en particulier n'a pas été décrite en détail dans le rapport de conception, puisque celle-ci dérive intégralement de celle de la bibliothèque Qt. En effet, un certain nombre de classes de Qt s'adapte parfaitement aux fonctionnalités désirées de cette partie. Il s'agit des classes `QGraphicsScene`, `QGraphicsItem` et `QGraphicsView`. C'est un modèle simple, qui comprend une scène contenant des *items* graphiques pouvant être affichés, déplacés, etc., et qui est affichée par une vue graphique (ce que voit l'utilisateur au final). Il a donc suffit de dériver de ces trois classes, puis de rajouter les fonctionnalités désirées grâce notamment aux évènements souris qu'elles peuvent recevoir. Les détails du modèle sont présentés ci-dessous.

1. Les QGraphicsItem

Les *items* graphiques sont tout ce que l'utilisateur peut ajouter, bouger ou supprimer. Il s'agit ici, des points d'intersection, des lignes droites et des courbes de Bézier, ainsi que des points de contrôle de ces courbes de Bézier.

Un *item* graphique peut recevoir des évènements souris (il est averti lorsque la souris passe dessus, lorsque l'utilisateur clique dessus, etc.), ainsi il est facile d'implémenter la fonctionnalité de déplacement des points, par exemple. De plus, chaque *item* possède sa propre fonction de dessin, ainsi, en différenciant les points d'intersection, de contrôle et les lignes, il est très facile de gérer l'affichage de chaque élément, notamment en ce qui concerne les courbes de Bézier, puisque Qt possède des fonctions de calculs et d'affichage de telles courbes, adaptées à ce type d'*item*.

Voici la liste des items implémentés :

- `BoundingPoint` : les points d'intersection ;
- `ControlPoint` : les points de contrôle des courbes de Bézier ;
- `BrLine` : Les lignes droites ou courbes de Bézier ;
- `ExtremityPoint` (dérive de `BoundingPoint`) : les points d'intersections situés sur l'axe de symétrie ;
- `Tangent` : les tangentes des courbes de Bézier ;
- `SymmetryAxis` : l'axe de symétrie ;
- `GhostPoint` : aide visuelle pour l'utilisateur, avant qu'il ne place un point d'intersection ;
- `GhostLine` : aide visuelle pour l'utilisateur, ligne reliant le dernier point d'intersection créé au `GhostPoint` ;
- `SelectionRectangle` : rectangle de sélection pour sélectionner plusieurs points à la fois.

Quelques fonctionnalités ont pu être implémentées directement dans les *items* graphiques, les autres l'ont été dans la scène graphique.

2. La QGraphicsScene (PaintingScene)

La scène graphique, appelée `PaintingScene` et dérivant de la classe Qt : `QGraphicsScene`, permet de manipuler tous les *items*, et fait également le lien entre la barre d'outils et la vue graphique. Cette scène peut recevoir des évènements souris (clics…) et possède un certains nombre de fonctionnalités utiles pour par exemple sélectionner un ensemble d'*items*.

La classe `QGraphicsScene` contient une référence sur tous les *items* qu'elle contient, mais pour plus de confort, trois listes (`QList`) ont été rajoutées, gérant chacune un type d'*item* important, soit : une liste pour les points d'intersections, une pour les points de contrôle et une pour les lignes. Cet ajout permet de faciliter certaines opérations, comme l'ajout et la suppression de points, ou la détection de la souris sur une ligne (puisque malheureusement, cette opération n'est pas très adaptée dans les items contenant des lignes).

La scène contient un certain nombre de *slots* Qt, c'est-à-dire de fonctions pouvant recevoir les signaux émis par les boutons de l'interface générale (notamment de la barre d'outils), donc c'est elle qui fait le lien avec le reste de l'application. Possédant toutes les références nécessaires sur les coordonnées des points d'intersections, ainsi que des points de contrôle, c'est cette classe qui fait le lien avec la structure interne de la monopalme, c'est-à-dire que chaque modification d'un *item* effectuée par l'utilisateur, est détectée et rapportée à la structure. À l'inverse, lorsque l'utilisateur souhaite annuler une action ou charger un fichier contenant une structure existante, la scène doit s'occuper de récupérer les coordonnées des points depuis la structure, puis de créer les *items* correspondants.

Si la scène récupère elle-même les données de la structure, c'est parce que gérer une classe supplémentaire faisant office d'intermédiaire aurait été très lourd à gérer, dans la mesure où il s'agit de faire transiter régulièrement plusieurs listes de coordonnées (à chaque *undo/redo*). De plus, la scène se sert de la classe `ProjectFile` (voir «

Structure de données / Couche métier » page 18), qui est déjà une façade pour accéder à la structure.

3. La QGraphicsView (PaintingView)

Cette classe sert principalement à afficher la scène et tous les *items* qu'elle contient, selon le modèle de la bibliothèque Qt. C'est également dans cette classe que l'axe Y est inversé, afin d'obtenir directement des coordonnées utilisables par le script COMSOL, et que le zoom est géré.

IV. Intégration et manipulation d'image

L'intégration d'image est utilisée dans deux parties : la zone de dessin (page 6) et l'algorithme d'extraction de contours (page 13). Pour la zone de dessin, l'utilisateur souhaitant travailler avec une image de fond a la possibilité d'en charger une et de l'intégrer comme empreinte dans la scène. L'image intégrée peut être déplacée, tournée, agrandie et rétrécie librement dans la scène. Pour la partie algorithme, l'utilisateur souhaitant extraire une forme depuis une image doit en charger une et la positionner correctement selon un axe de symétrie. Pour cela, il peut déplacer, tourner, agrandir et rétrécir librement l'image dans la zone de travail.

Les deux utilisations de la manipulation d'image étant similaires pour la partie zone de dessin et la partie algorithme, une seule entité gère toutes les fonctionnalités.

A. Fonctionnalité

Les fonctionnalités n'ont pas changé en ce qui concerne l'intégration d'image et sa manipulation.

1. Intégration de l'image

L'image est chargée puis intégrée dans la scène depuis un bouton cliquable qui ouvre une boite de dialogue. L'utilisateur sélectionne l'image qu'il souhaite intégrer.

Pour la manipulation d'image en général, plus aucun bouton n'est nécessaire. Pour une manipulation précise de l'image, l'utilisateur a la possibilité de rentrer des valeurs numériques dans des champs de saisie de l'interface graphique (uniquement pour la partie extraction de contours).

2. Déplacement de l'image

L'image peut être déplacée par un simple glisser/déplacer en cliquant dessus avec le bouton gauche de la souris. Dans la partie extraction de contours, il y a possibilité de rentrer la position X et Y directement dans des champs de saisie de l'interface. Ces champs affichent la position de l'image en permanence.

3. Rotation de l'image

En ce qui concerne la rotation de l'image, un satellite positionné au dessus apparaît en même temps qu'elle lors de son intégration et permet de la tourner. Ce satellite gravite selon un cercle centré sur le centre de l'image. Le satellite est immobile sans action de l'utilisateur. En le pressant avec le bouton gauche de la souris, puis en déplaçant la souris, il se positionne à l'intersection de l'axe, décrit par le centre de l'image et le curseur de la souris, et du cercle de gravitation. Ce déplacement sur le cercle de gravitation correspondant à un angle de déplacement, l'image pivote selon ce même angle. Dans la partie extraction de contours, il y a possibilité de rentrer la valeur de l'angle de rotation directement dans un champ de saisie de l'interface. Ce champ affiche la valeur de l'angle en permanence.

4. Agrandissement / Rétrécissement

En ce qui concerne la partie extraction de contours, la molette de la souris permet de jouer directement sur la taille de l'image. Pour la partie zone de dessin, les boutons « *up* » et « *down* »

modifient la taille de l'image. Dans la partie extraction de contours, il y a possibilité de rentrer la valeur de l'échelle de l'image directement dans un champ de saisie de l'interface. Ce champ affiche la valeur de l'échelle de l'image en permanence.

À noter que par défaut, une image choisie comme arrière-plan dans la zone de dessin ne peut pas être manipulée directement. Il faut pour cela sélectionner le bouton « modification de l'image d'arrière plan ».

B. Structure de données

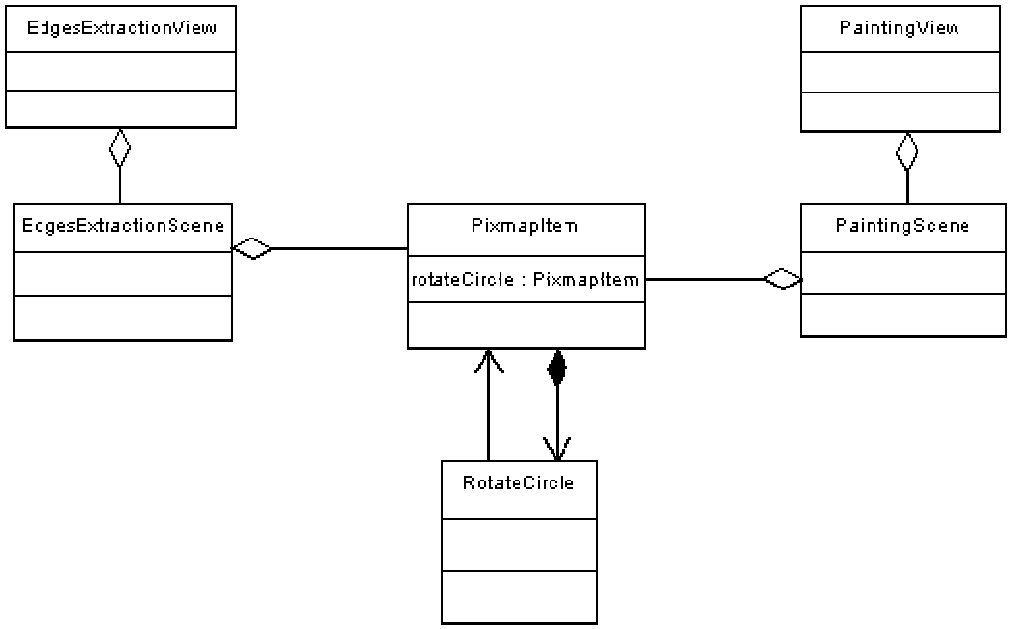


Figure 1 Diagramme UML des classes responsable de la manipulation d'image

Comme écrit dans la partie zone de dessin, une scène graphique (`GraphicsScene`) manipule des *items*. L'image et le satellite associé sont donc aussi deux *items* manipulés dans une scène. Une classe `PixmapItem` contient l'image que l'on souhaite intégrer et manipuler. Un objet `PixmapItem` peut être déplacé, agrandi ou rétréci. Le satellite quant à lui est désigné par la classe `RotateCircle` et est obligatoirement associé à un objet `PixmapItem`. Il permet de tourner l'image contenu dans le `PixmapItem`. Ces structures sont aussi bien utilisées pour la zone de dessin que pour l'extraction de contours. Ainsi, pour l'extraction de contours, le `PixmapItem` est situé dans une `EdgesExtractionScene` et vu par `EdgesExtractionView`. Pour la zone de dessin, le `PixmapItem` est situé dans une `PaintingScene` et vu par `PaintingView`.

V. Extraction de contours

Comme précisé dans les spécifications, l'utilisateur a la possibilité de détecter la forme d'une image et de l'extraire sous forme de vecteurs directement utilisables dans la zone de dessin. Aucune fonctionnalité n'a été modifiée, excepté la manipulation d'image décrite à la page 11. Divers algorithmes ont été utilisés (et testés) pour la détection et l'extraction de contours. Ci-après est donnée une description des algorithmes testés et le choix final de ceux utilisés dans l'application. Ensuite sont données les structures utilisées pour implémenter l'algorithme en question.

A. Algorithmes

1. Détection de contours

Lors de la conception, le choix de l'algorithme des contours actifs (S*nake*) a été choisi. Cet algorithme a l'avantage de détecter une unique forme dans une image et de posséder une complexité acceptable. Son principe est d'utiliser un ensemble de points définis sur un cercle englobant l'image et de rapprocher ces points du centre de l'image. Lorsqu'un point détecte un changement de luminosité dans les pixels qu'ils parcourent, il se bloque. L'algorithme se termine lorsque tous les points sont bloqués ou atteignent le centre de l'image. Le nombre de points défini doit être un minimum en rapport avec la taille de l'image. En effet, il est inutile de définir mille points si l'image a une résolution de 100*100. De manière général, plus le nombre de points est important, plus il a de chance de bien détecter le contour. Pour un bon fonctionnement de l'algorithme, il est nécessaire d'avoir une forme à détecter se détachant bien du fond de l'image. Après plusieurs tests sur différentes images, cet algorithme s'est révélé tout à fait satisfaisant pour détecter des formes de monopalmes et il est maintenant utilisé dans l'application.

2. Extraction de contours

Le but est d'extraire le contour détecté par le *snake* et de le vectoriser sous la forme utilisée dans la zone de dessin. L'intérêt est de minimiser le nombre de points d'intersection de la forme vectorisée extraite et de définir des points de contrôle pour créer des courbes de Bézier se rapprochant le plus possible de la forme détectée.

a. Algorithme inventé

Il semblait intéressant pour cette partie d'utiliser des algorithmes déjà existants. Cependant après des recherches infructueuses, une première approche fut de créer un algorithme spécifique à notre problème. Le principe de cet algorithme est, dans un premier temps, de détecter les points de « cassure » dans la forme détectée. Pour cela, on calcule le coefficient de la tangente de tous les points de l'ensemble des points entourant la forme. On parcourt ensuite l'ensemble des points dans l'ordre, et lorsqu'on remarque une importante variation dans les coefficients des tangentes, on garde le point en tant que point d'intersection d'une courbe de Bézier. Pour le calcul des points de contrôle, il suffit de prendre l'intersection des tangentes de deux points consécutifs et ce, pour chaque points deux-à-deux. Après avoir implémenté l'algorithme et l'avoir testé, celui-ci s'est révélé assez inefficace. Le calcul des tangentes des points est très hasardeux et par conséquent les points détectés ne sont pas forcément pertinents (encore moins les points de contrôle).

b. Potrace

Finalement, en continuant les recherches sur des algorithmes existants, un algorithme a été retenu et testé : Potrace. Cet algorithme fut créé par Peter Selinger et est disponible librement sur http://potrace.sourceforge.net/. Cet algorithme permet de vectoriser une image (bitmap) selon une méthode basée sur des polygones. Pour information, la méthode consistant à vectoriser un bitmap s'appelle *tracing*, d'où le nom de l'algorithme Po(lygon)Trace.

L'algorithme fonctionne à la base sur des images en noir et blanc (pas en niveau de gris). Il est décomposé en quatre étapes :

- la détection des contours ;
- la création de polygones ;
- l'approximation des courbes paramétrées (vecteurs) ;
- l'optimisation des courbes paramétrées.

Pour plus d'informations sur le fonctionnement de l'algorithme :

http://potrace.sourceforge.net/potrace.pdf.

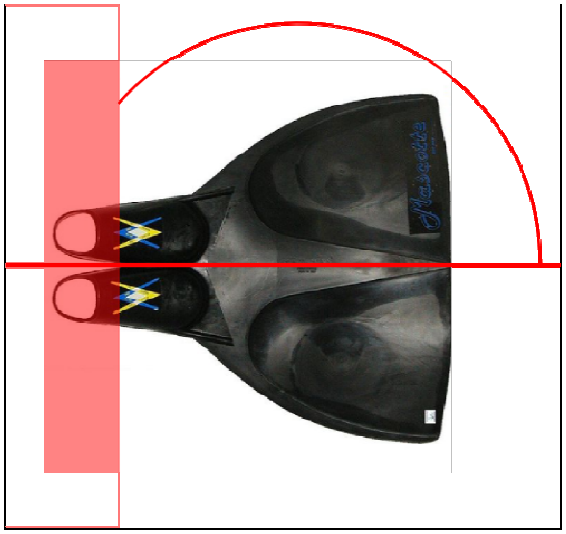
Grossièrement, l'algorithme calcule les contours de l'image en détectant les frontières entre le noir et blanc. Ces frontières sont continues et forment ce qu'on appelle des chemins. Les polygones sont calculés à partir de ces chemins en approximant un tracé. Ensuite, le calcul des vecteurs se fait à partir du polygone : chaque sommet du polygone devient un point de contrôle dans une courbe de Bézier. Les extrémités des courbes sont obtenues en prenant le milieu de chaque côté du polygone. Enfin, les courbes sont optimisées pour diminuer le nombre d'extrémités.

Dans notre cas, la détection de contours est déjà faite par le *snake*, le chemin est déjà obtenu. Il reste donc à implémenter l'étape de l'approximation du tracé par les polygones et le calcul des courbes paramétrées. La dernière étape n'est pas utile car les courbes de Bézier obtenues lors de cette étape ne correspondent pas à celles utilisées dans la zone de dessin.

Un schéma explicatif est donné ci-après et explique le fonctionnement de toutes les étapes de la détection de contours et extraction de contours pour notre projet.

Après avoir testé l'algorithme, celui-ci s'est révélé efficace et est donc utilisé dans l'application.

3. Schémas explicatifs

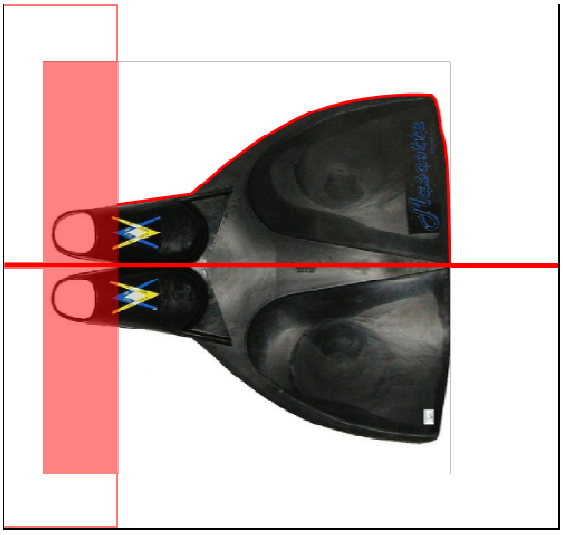


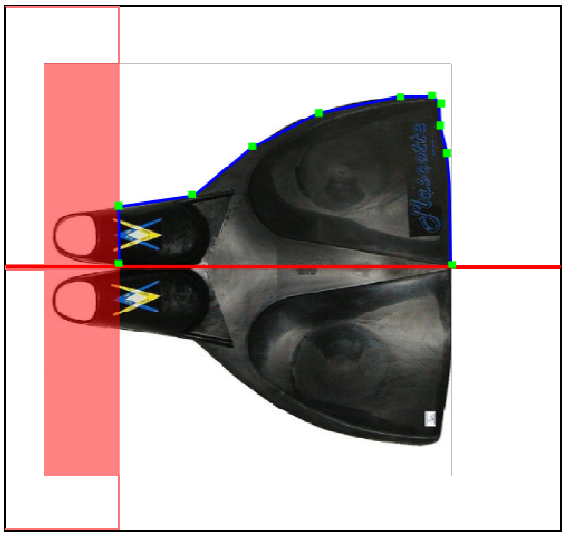
Etape 1

Chargement de l'image et positionnement par rapport à l'axe de symétrie. La zone rouge à gauche représente un « talon », tout ce qui est dans cette zone ne sera pas détecté. De plus, les algorithmes ne fonctionneront que sur la partie supérieure à l'axe de symétrie. L'arc de cercle rouge représente une partie de l'ensemble des points du *snake*.

Etape 2

L'utilisateur a lancé les algorithmes. L'algorithme des contours actifs (*Snake*) est le premier à passer. Sur l'image on aperçoit l'ensemble des points du *snake* qui englobe la forme détectée après passage de l'algorithme.





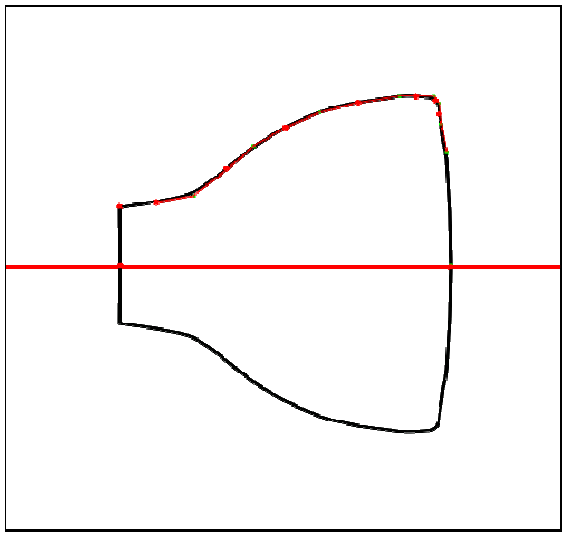
Etape 3

Le Potrace démarre ensuite. Comme expliqué auparavant, on applique directement l'étape d'approximation du polygone. Le polygone apparait en bleu sur l'image (on ne voit que la moitié du polygone car l'algorithme ne fonctionne que sur une moitié d'image). Chaque point vert est un sommet du polygone. Les segments du polygone approxime le contour détecté par le *snake,* c.-à-d. que chaque point du *snake* est dans une zone alentour d'un segment.

Etape 4

L'étape d'approximation des courbes paramétrées de Portrace est appliquée. Chaque sommet du polygone est devenu un point de contrôle de la courbe de Bézier. Les points d'intersection sont calculés en prenant le milieu de chaque segment du polygone.

Tous les algorithmes sont passés, on peut détecter par symétrie la partie inférieure de la forme.



B. Structure de donnée

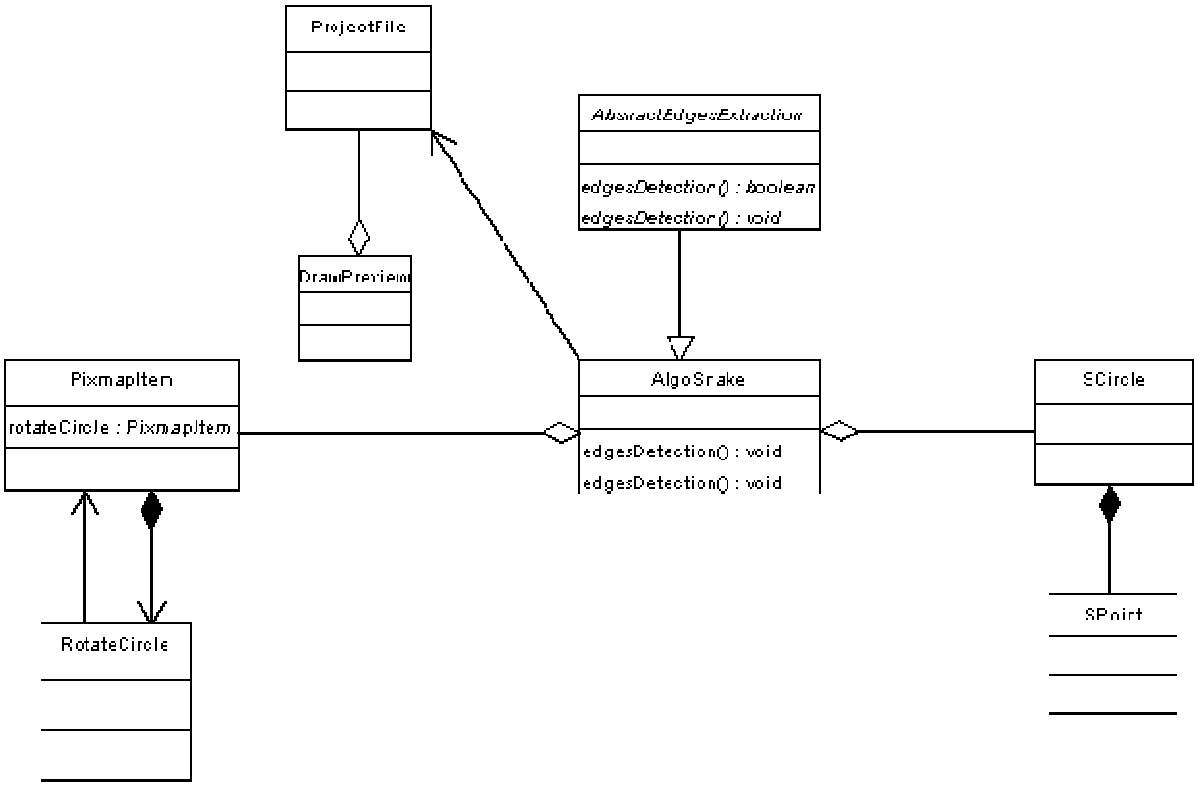


Figure 2 Schéma UML, structure de données utilisées pour les algorithmes

Dans cette structure, tout est centré sur la classe `AlgoSnake` héritant de la classe abstraite `AbstractEdgesDetection`. On utilise donc un *pattern stratégie* en supposant que d'autres algorithmes d'extraction de contours peuvent être implémentés. Cette classe abstraite possède deux méthodes devant obligatoirement être implémentées dans toute classe la dérivant. La méthode `edgesDetection()` permet de détecter le contour d'une forme. Dans la classe `AlgoSnake`, c'est l'algorithme des contours actifs (*Snake*) qui est utilisé. La méthode `edgesExtraction()` permet d'extraire la forme sous forme vectorielle à partir de la forme détectée. Dans la classe `AlgoSnake`, c'est le Potrace qui est implémenté (du moins partiellement).

La classe `AlgoSnake` a besoin de deux objets pour fonctionner : un `PixmapItem` (décrit dans la partie « Intégration et manipulation d'image » page 11 Intégration et manipulation) contenant une image positionnée et tournée dans une scène, et un `SCircle` qui est en fait le *snake* détectant la forme. Lors de l'appel de la méthode `edgesDetection()`, chaque point `SPoint` composant le `SCircle` va se positionner sur un pixel de l'image contenu dans `PixmapItem`. Lorsque la méthode se termine, les `SPoint` sont positionnés sur la forme que l'on souhaite détecter.

Enfin, pendant l'appel de la méthode `edgesExtraction`, après que l'algorithme Potrace soit appliqué et que les courbes de Bézier soit déterminées, toute la structure de la forme extraite est enregistrée dans un `ProjectFile` (voir «

Structure de données / Couche métier » page 18). Ce `ProjectFile` peut donc être ensuite utilisé dans la zone de dessin.

La classe `DrawPreview` permet d'afficher un aperçu de la forme extraite. Ainsi l'utilisateur peut choisir de garder ou non la forme extraite.

VI. Structure de données / Couche métier

A. Modifications apportées à la Structure de Donnée

Rappel de la structure telle que décrite dans le dossier de conception :

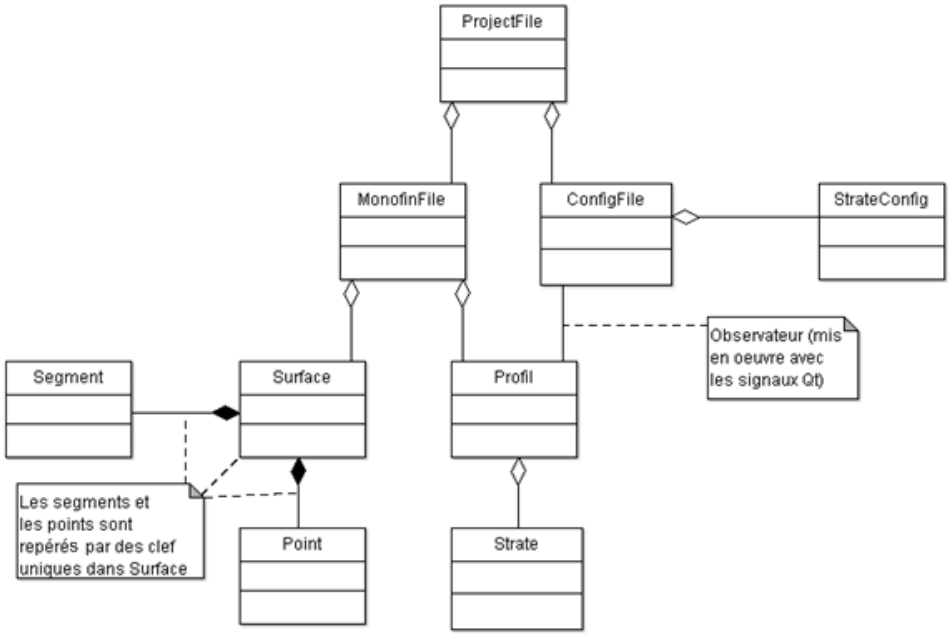


Figure 3 Schéma UML, structure de données principale

Schéma de la structure actuelle :

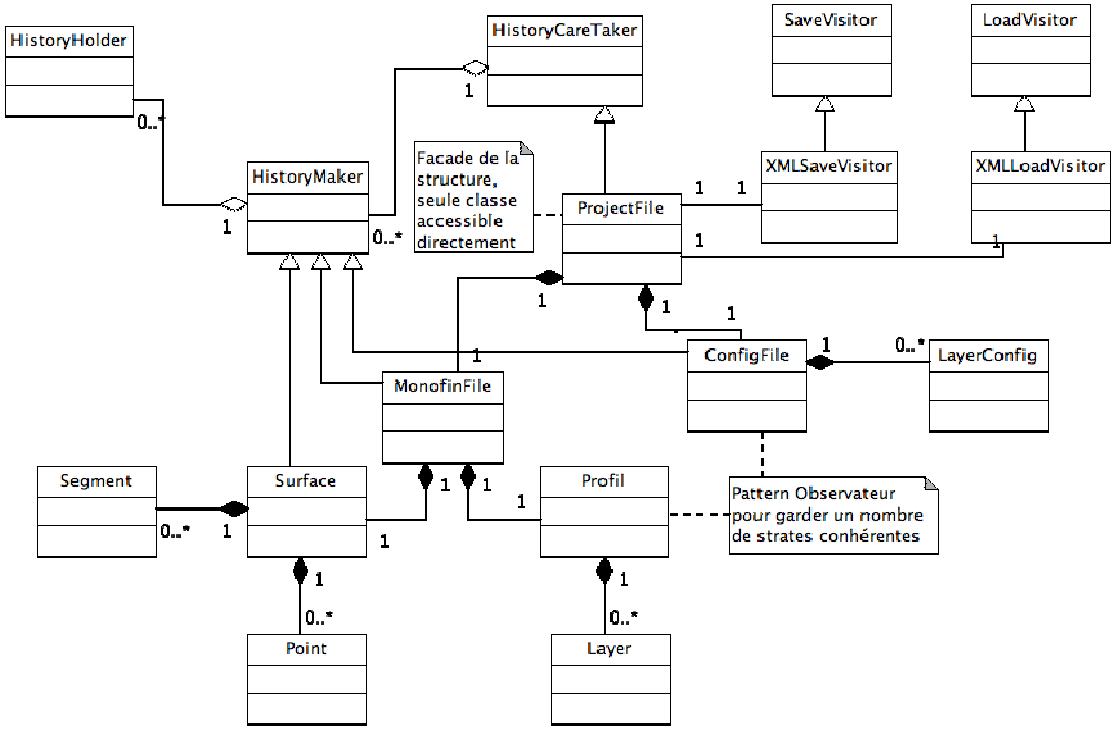


Figure 4 Schéma UML final de la structure de donnée

B. Structure générale

L'arborescence présentée dans le rapport de conception a été conservée car elle représentait clairement les différentes structures de données qui étaient nécessaires. On retrouve donc à la racine `ProjectFile`. Il s'agit d'une entité regroupant pour un projet donné les informations concernant la géométrie de la palme (`MonofinFile`) d'une part, et ses propriétés physiques d'autre part (`ConfigFile`). C'est à elle que s'adressent toutes les requêtes provenant des étapes de configuration et de l'interface de dessin. Son principal travail est d'aiguiller ces requêtes vers les éléments concernés.

`MonofinFile` gère la géométrie de la palme. Il s'agit à la foi de la forme (`Surface`) et du profil (`Profil`). Là encore, il s'agit de rediriger les fonctions d'appel et de modification vers les structures inférieures.

`Surface` contient tout les éléments représentant le dessin de la palme. Il s'agit de points (disposant de coordonnées et listant les segments dans lesquels ils sont utilisés). Ces points sont décomposés en `IntersectionPoint` ou en `ControlPoints` selon qu'ils délimitent des segments, ou qu'ils paramètrent la courbe de Bézier remplaçant le segment. Les segments sont donc décrits par les points les délimitant et ceux qui définissent la courbe de Bézier éventuelle. Il est à noter que lorsque le « segment » n'est pas une courbe Bézier, il possède tout de même un `ControlPoint`. Pour éviter les doublons, les points sont représentés par leur indice dans une table comportant tous les points. Les fonctions de mises à jour interviennent donc essentiellement sur ces tables.

`Profil` est utilisé pour représenter les différentes strates (`Layer`) qui composent la palme. En réalité, `Profil` doit également faire suivre les informations concernant le nombre et a position des strates à `ConfigFile`. En effet, le nombre et la taille des strates sont définis dans le `Profil`, mais leurs propriétés physiques ont été séparées, et sont paramétrables dans un second temps, via les options de configuration. `Profil` à donc la possibilité de notifier à son homologue `ConfigFile` les créations/modifications de strates. `Layer` propose simplement une structure pour les strates : épaisseur et longueur. Les propriétés physiques sont détaillées dans `ConfigFile`.

`ConfigFile` est donc asservie aux informations de `Profil`, ce qui permet de paramétrer correctement les strates. Les méthodes de `ConfigFile` ajoutent donc des strates lorsqu'elles en sont notifiées, et les modifient selon les valeurs fournies par l'utilisateur. Il s'agit simplement du module d'Young, du coefficient de Poisson et de la masse volumique.

Nous avons enfin rajouté un certain nombre de classes pour la gestion du chargement-déchargement de la structure et pour une gestion efficace des *undo/redo* au niveau même de la structure.

1. Chargement-déchargement de la structure

La sauvegarde de la structure doit fournir un fichier XML qui correspond à la définition proposée dans le rapport de conception. La technique employée ici est celle du visiteur. Il existe un visiteur pour la sauvegarde et un pour le chargement. Tout deux s'intéressent aux « feuilles » de l'arborescence, à savoir `Surface`, `Profil` et `ConfigFile`.

L'instruction de sauvegarde est envoyée comme toujours à la racine, `ProjectFile`. Celle-ci se charge d'envoyer le visiteur dans les structures inférieures où il sera accepté, jusqu'à atteindre une des trois structures « feuilles ». Selon le type de structure qui accepte le visiteur, une fonction de

sauvegarde est appelée par la fonction d'acceptation. Cette dernière fournit en paramètre un accès à la structure afin d'être explorée par le visiteur. Celui-ci construit sous forme de texte la partie du fichier XML correspondante en parcourant les données de la structure. Par la suite, les trois parties sont fusionnées pour fournir le fichier de sauvegarde.

Le chargement d'un fichier de sauvegarde obéit au même processus. Pour plus de flexibilité, le visiteur utilise un *parser* XML qui permet de reconnaître les diverses balises et crée ainsi les éléments de structure correspondant. Selon leur type, ces informations sont ensuite chargées dans chaque portion de la structure, ce qui la construira et permettra de la lire depuis l'application comme si elle venait d'être conçue.

2. Le trio HistoryHolder/HistoryMaker/HistoryTakeCarer

Cet ensemble de classe sert à la gestion des historiques d'actions pour offrir des fonctionnalités d'*undo/redo*. Nous avons choisi de faire gérer les historiques au niveau de la structure pour faciliter l'abstraction pour les modules utilisateur. Cependant cette abstraction à un coût, en effet une action qui peut être atomique au niveau des modules peut se décomposer en une séquence de modification au niveau de la structure. Il fallait donc que l'historique puisse défaire une séquence d'action (et la mémoriser) comme s'il s'agissait d'une seule et même action. Un autre problème de cette gestion était le fait que chaque action doit faire mémoriser des informations qui lui sont particulières pour pouvoir se défaire. Il était soit possible d'utiliser une pile sauvegardant à chaque état toutes la structure, soit de concevoir pour chaque méthode son inverse et de seulement enregistrer l'enchaînement des actions. C'est de cette dernière solution que s'inspire la gestion des *undo/redo* proposée dans `HistoryHolder`.

Nous avons donc élaboré un ensemble de classes pour gérer à notre convenance les *undo/redo*. L'idée est basée, grossièrement, sur l'histoire des hommes. Il y a, en bas de l'échelle ceux qui font l'histoire, par exemple les ouvriers qui bâtissent une cathédrale. Il y a les réceptacles de ces évènements, les récits des architectes et des habitants de l'époque par exemple. Enfin il y a ceux qui rassemblent ces fragments d'histoires et forme un tout cohérent et le transmettent, pour continuer dans la comparaison : des historiens. Dans notre structure, les bâtisseurs sont `Surface`, `Profil` et `ConfigFile`, ce sont véritablement eux qui agissent. Le récit de leurs épopées est stocké dans la classe `HistoryHolder`, une classe spécifique qui sauvegarde tout type de données et les séquences. Enfin il y a l'historien de notre structure, la racine de l'arbre, `ProjectFile`, qui va donc stocker faire remonter les listes d'`HistoryHolder` en fonction des besoins des utilisateurs (en termes de gestion d'historique et d'atomicité d'action). Voyons maintenant le fonctionnement de ces classes.

a. Fonctionnement général :

Dans un premier temps, lorsque l'utilisateur réalise une action, comme ajouter un point d'intersection dans un segment, les différentes requêtes suscitées par cette action sont transmises à la structure entre deux fanions `startHistory` et `stopHistory`. Le gestionnaire d'historique (`HistoryCareTaker`) envoie les requêtes aux éléments de la structure via la racine (`ProjectFile`). Si le type des actions correspond au type des « feuilles » (`MonofinSurface` pour `Surface`, `MonofinLayer` pour `Profil`, `MonofinLayerConfig` pour `ConfigFile`), elles réaliseront les requêtes, et empileront, pour chacune, tous les paramètres ainsi que le nom de la méthode nécessaire pour revenir en arrière, dans un `HistoryHolder`. Les `HistoryHolder`s ainsi obtenus via les différentes

méthodes sont alors stockées et séquencées dans une liste, et remonté à la racine, qui regroupe toutes les actions à défaire dans le cas d'un Undo, lors d'un `stopHistory`.

Lorsqu'un undo est sollicité, le gestionnaire d'historique transmettra la dernière liste d'action stockée à la structure correspondant au type de l'action, qui utilisera sa propre méthode undo. Cette méthode, selon le contenu du `HistoryHolder` fourni devra dépiler les informations et les traduire pour appeler les méthodes défaisant les actions. Ces méthodes créeront à leur tour un nouveau `HistoryHolder`, qui sera stocké, séquencé dans une liste et remonté à la racine, qui permettra ainsi de faire simplement un *redo*, en réappelant la méthode *undo*.

b. *HistoryHolder*

Il s'agit d'un conteneur universel. En fait il s'agit d'une pile ou les `HistoryMaker` stockent ce qu'ils souhaitent. Ce conteneur possède un type qui indique quel `HistoryMaker` l'a utilisé afin de pouvoir l'aiguiller lors d'un Undo vers le bon `HistoryMaker`.

c. *HistoryMaker*

Ce sont les agents qui modifient les données, la classe `HistoryMaker` fournit un ensemble de fonctions aux classes qui en hériteront, pour séquencer facilement les `HistoryHolder` de sorte que les classes implémentant `HistoryMaker` n'ont qu'à redéfinir les fonctions pour effectuer les *undo*.

d. *HistoryCareTaker*

Il s'agit de la classe qui récupère, répertorie, et redistribue les informations quand nécessaire pour faire les *undo*. Pour ce faire il se base sur le principe de la chaîne de responsabilité et transmet à tous les `HistoryMaker` qui sont ses fils direct les informations à défaire, ceux-ci se servent du type contenu dans `HistoryHolder` pour savoir si les informations les concernent ou non. Pour les classes qui implémentent `HistoryCareTaker` il n'y a quasiment aucun code à rajouter, toutes les fonctions étant déjà codées. Pour gérer le *redo* `HistoryCareTaker` fait mémoriser par les `HistoryMaker` toutes les actions qu'ils réalisent lors d'un *undo* et les stockent et fait finalement un *undo* de ces actions, ce qui permet de n'avoir à coder que la fonction *undo* dans les `HistoryMaker`.

C. Nouvelles fonctionnalités de la classe Surface

Lors du développement de la structure de donnée, en nous concertant avec les développeurs des modules utilisateurs (notamment du module de dessin), nous avons pu apporter des fonctionnalités, qui facilitaient leurs travaux mais qui n'avaient pas été envisagées lors de la conception.

Ainsi les fonctions suivantes ont été rajoutées :

- `QList<int> getAllSegmentKeys()` : cette fonction renvoie l'ensemble des clefs d'identification des segments sous forme de liste non ordonnée ;
- `QList<int> getAllIntersectionPointKeys()` : cette fonction renvoie l'ensemble des clefs d'identification des points d'intersections sous forme d'une liste non ordonnée ;
- `QList<int> getAllControlPointKeys()` : cette fonction renvoie l'ensemble des clefs des points de contrôle sous forme d'une liste non ordonnée ;

- `QList<int> getExtremityPoint()` : cette fonction permet d'avoir les deux points extrémités du demi-segment (étant donné que `Surface` travaille sur une demi-surface) sous forme d'une liste non ordonnée ;
- `QList<int> getSegmentKeysLinkedToPoint(int pointKey)` : cette fonction renvoie une liste de clef(s) de segment(s) à laquelle est relié un point intersection dont la clef est passé en paramètre ;
- `void clearSurface()` : cette fonction efface toute la surface, enregistre aussi dans l'historique les modifications apportés, de sorte que si on a effacé la surface on puisse revenir en arrière.

VII. Interfaçage avec COMSOL

Notre projet se compose principalement d'une interface graphique, celle-ci est destinée à faciliter l'utilisation du logiciel de simulation physique COMSOL Multiphysic en permettant la modélisation de monopalmes. Notre application est un intermédiaire entre l'utilisateur et COMSOL. Quelques modifications ont été faites concernant l'interfaçage de notre application avec COMSOL. Les parties qui suivent présentent un état des lieux détaillé de tout ce qui concerne l'interfaçage avec COMSOL.

A. Spécifications et attentes

Au moment de la spécification il était question que notre application génère des scripts à partir des données collectées depuis notre interface graphique. Ces scripts devaient ensuite être exécutés par COMSOL en mode *batch.* Pour terminer, notre interface devaient être capable de récupérer les résultats générés par les scripts pour les exposer aux utilisateurs sous différentes formes (images, vidéos, données brutes).

La non connaissance de COMSOL avant le projet ne nous a pas permis de fournir des spécifications très précises au début. Durant la phase de développement le travail sur COMSOL a été accentué et a permis de fixer les attentes concernant l'interfaçage COMSOL avec les utilisateurs.

Notre application est capable à travers un module appelé `Scripting` de générer des scripts COMSOL à partir de nos structures de données. Une fois généré, un script est capable d'être exécuté en arrière plan d'une manière transparente pour l'utilisateur. Enfin on peut récupérer les résultats sous forme de fichier(s) généré(s). Cela rentre en accord avec les spécifications.

Plus précisément notre application génère deux scripts :

- `ViewerScript`, ce script est utilisé pour répondre à la fonctionnalité « Prévisualiser la monopalme en 3D » du dossier de spécification ;
- `DefaultScript`, c'est le script principal qui est généré pour lancer une simulation physique sur une monopalme.

Remarque, `ViewerScript` et `DefaultScript` sont des classes représentant les deux scripts en interne.

B. Communication avec COMSOL

La communication avec COMSOL a légèrement changé. Normalement les scripts COMSOL devaient pouvoir être exécutés en arrière plan à travers un programme appelé COMSOL batch. Il s'agit d'une interface en ligne de commande capable d'exécuter le script dont le fichier est passé en paramètre.

Les premiers développements de l'interface avec COMSOL ont utilisé COMSOL batch. Il fallait donc générer les scripts utilisés dans des fichiers temporaires qui étaient ensuite passés en paramètre à COMSOL batch. Malheureusement le programme COMSOL batch retourne quoi qu'il arrive un code d'erreur constant et ne permet pas de savoir si l'exécution d'un script a réussi ou échoué.

Une solution alternative a dû être trouvée. Il s'avère que COMSOL expose des API autre que le programme COMSOL batch pour avoir accès à son moteur d'exécution (de script). Ces API sont exclusivement accessibles depuis JAVA (éventuellement C/C++ avec la JNI). Pour diverses raisons techniques, l'utilisation des API depuis C++ directement avec notre interface graphique a été rejetée :

- Dépendance trop forte avec le dossier d'installation de COMSOL sur le poste de l'utilisateur ;
- Complexité de l'interfaçage C++/Java dans le cadre de COMSOL.

Un petit programme JAVA a été écrit, celui-ci lit sur son entrée standard un script COMSOL (on se passe donc de fichiers temporaires), l'exécute à la volée et retourne un code d'erreur correcte en fin d'exécution (0 = pas d'erreur, 1 = une erreur est survenue) en plus d'afficher des informations détaillées sur la sortie standard en cas d'erreur.

Le schéma suivant présente le nouveau processus de communication avec COMSOL.

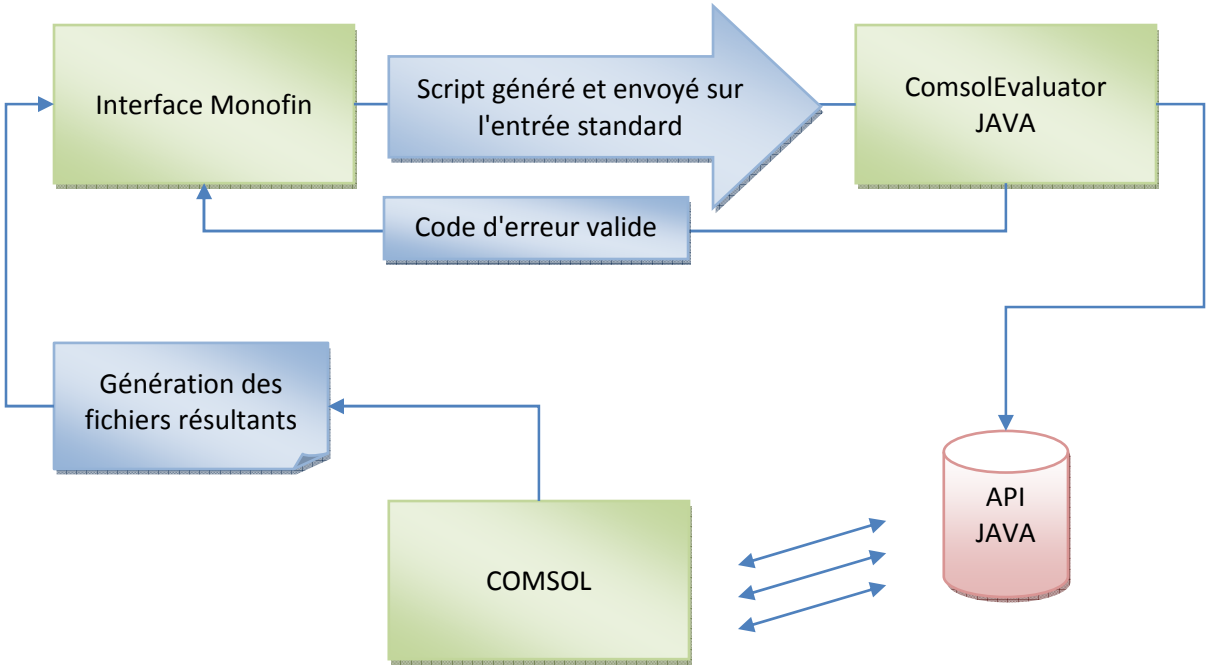


Figure 5 Schéma de communication avec COMSOL

C. Simulations avec le script principal

Le script principal est celui qui est généré pour effectuer une simulation. Ce script contient la description de la géométrie de la palme ainsi que les paramètres physiques de ses strates. Le type de simulation qui doit être lancée a été choisi avec les utilisateurs après la phase de spécification. Il s'agit d'une analyse modale en utilisant la méthode des *Eigenfrequency*, c'est à dire que l'on calcul les fréquences propres de la monopalme (les six premières dans notre cas). Ce type de simulation n'a pas besoin qu'on lui précise l'équation du mouvement de la palme ou le temps de simulation à calculer, contrairement à ce qui avait été spécifié.

L'analyse dynamique qui elle nécessite l'équation du mouvement, un pas de temps, etc. sera traitée plus tard dans une future évolution du logiciel. Pour offrir une solution la plus modulable possible et simple à faire évoluer, les scripts générés par notre application (`ViewerScript` et `DefaultScript`) ont la même structure :

- Déclaration d'une structure « monofin » qui va contenir toutes les informations générées par notre interface graphique pour un script donné ;
- Remplissage de la structure avec les informations décrivant la géométrie de monopalme : contour en courbes de Bézier, strates et propriétés physiques associées ;
- Remplissage de la structure avec des paramètres en fonction du script généré ;
- Appel d'une fonction externe représentant le point d'entrée d'un script.

Dans un souci de flexibilité, notre application ne génère que des scripts qui contiennent des données que notre interface graphique est capable de collecter (la géométrie de la monopalme principalement). Ensuite, le script appel une fonction externe qui va faire le travail à partir de la structure appelée « monofin » déclarée dans le script.

Les fonctions externes sont contenues dans des fichiers de scripts qui sont en clair et à la disposition des utilisateurs. Ces scripts externes peuvent même être modifiés pour changer le comportement qu'ils ont sans que notre interface graphique ait besoin d'être modifiée. Voici la liste des scripts COMSOL qui ont été écrits pour effectuer le travail demandé :

- main_default.m : le point d'entrée du script par défaut, il modélise une monopalme et lance une simulation modale dessus ;
- main_viewer.m : le point d'entrée du script de visualisation de la monopalme, il se contente de modéliser une monopalme pour en récupérer une image en 3D de sa géométrie ;
- build_geometry.m : une fonction qui génère une géométrie COMSOL à partir des informations générées par notre interface graphique ;
- build_fem.m : une fonction qui génère une structure FEM (Finite Element Method) qui est la structure que COMSOL utilisée pour stocker une simulation, ici la fonction créer une structure contenant une simulation modale par fréquences propres.

D. Impacts sur les classes C++

Tout le travail de modélisation d'une géométrie de monopalme compatible avec COMSOL à partir des informations de notre interface graphique a été fait en langage de script COMSOL. Cela a considérablement simplifié la structure des scripts générés par notre application et a donc simplifié le schéma UML des classes initialement prévues.

Le dossier de conception n'avait modélisé que la partie "Génération des scripts COMSOL" et pas les classes responsables de leur exécution. Cela a été corrigé pendant la phase de développement, les fonctionnalités de la librairie Qt ont été utilisées au maximum pour bénéficier de l'interopérabilité sur la création de processus par exemple (et ainsi garantir le fonctionnement de notre application sous Linux, Mac et Windows).

Le schéma UML simplifié suivant présente brièvement l'architecture de l'espace de nom `Scripting` du projet qui est responsable de l'interfaçage avec COMSOL.

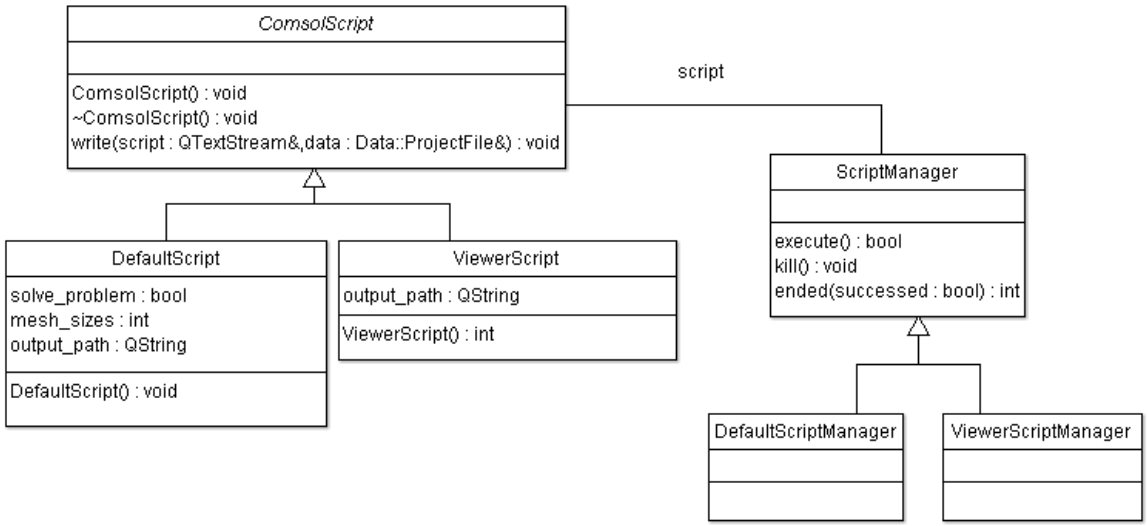


Figure 6 Schéma UML simplifié de l'espace de nom Scripting

La classe abstraite `ComsolScript` représente un script, elle contient le nécessaire à l'écriture d'un script dans un objet de la librairie Qt appelé `QtextStream`. Les classes `DefaultScript` et `ViewerScript` héritent toutes deux de `ComsolScript` et représente les deux scripts que notre application est capable de générer.

La classe abstraite `ScriptManager` est responsable de l'exécution des scripts. Sa méthode `execute` permet l'exécution en arrière plan d'un script, `kill` permet d'annuler l'exécution d'un script et enfin la méthode `ended` est un signal Qt qui est automatiquement "déclenché" à la fin d'exécution du script et indique dans un paramètre booléen si l'exécution s'est bien déroulée ou non. Deux implémentations de cette classe existent pour gérer les scripts `DefaultScript` et `ViewerScript`.

Comme dans le reste du projet nous avons utilisé au maximum les fonctionnalités de la librairie Qt qui garantie l'interopérabilité de notre application et qui de plus, est très fiable.

VIII. Conclusion

Tout projet quel qu'il soit subit des altérations avec le temps. Il est humainement impossible de tout prévoir et de tout chiffrer a priori. Notre projet ne faisant pas exception à la règle, des éléments prévus dans nos rapports de spécification et de conception ont été modifiés, supprimés, et ajoutés.

La majorité des fonctionnalités ont pu être gardées. Cependant, certaines fonctionnalités ont dû être éliminées, en accord avec les utilisateurs, car trop difficile à mettre en œuvre dû au temps limité dont nous disposions et dû au fait qu'elles n'étaient pas primordiales.

À contrario, de nouvelles possibilités très pratiques ont pu être proposés aux utilisateurs, fonctionnalités qui nous apparues au fur et à mesure du développement, par raffinement successifs. Néanmoins nous avons dû à un moment donné arrêter d'ajouter des fonctionnalités car la date butoir approchait et qu'il fallait un produit fini.

Voici une liste non exhaustive des fonctionnalités qui pourront être rajoutées plus tard :

- Ecrire et intégrer un script pour lancer une simulation dynamique
    - Traiter les résultats de ce script à l'aide de MATLAB
- Proposer une interface rudimentaire permettant d'effectuer des post-traitements de base sur les résultats de simulation, par exemple :
    - Récupération de vidéos (animation du mouvement)
    - Données numériques brutes sous une forme permettant le traitement
    - Images diverses
- Proposer un système permettant de comparer les résultats issue des simulations avec les résultats issue du robot employé pour faire actuellement des tests en bassin