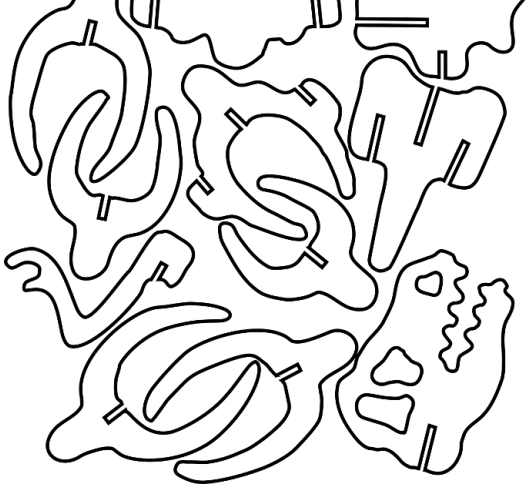


◆ ◆ ◆

Département informatique 2ème année
ENSEIRB-MATMECA

PFA

Packing d'images vectorielles

Par : **Dorian Cuquemelle**
Élodie Feng
Hugo Le Dily
Maxime Mondot
Youssef Naim
Virgile Robles
Adrien Rodrigues

Client : D. Renault
Encadrant : J. Allali

Table des matières

Introduction

Dans le cadre de l'enseignement de $2^{ème}$ année informatique de l'ENSEIRB-MATMECA, nous avons été amenés à travailler en équipe de sept étudiants sur un projet de longue durée. L'objectif était d'une part de développer nos compétences techniques, mais également et principalement de nous confronter à des problématiques s'apparentant à celles rencontrées en entreprise, telles que le contact client et la méthodologie de travail de groupe.

L'utilisation de machines de découpe de matériaux telle que la découpeuse laser est de plus en plus répandue parmi les *fab labs*, leur permettant un prototypage simple, rapide et automatisé. Inévitablement, un besoin d'économiser le matériau naît, et avec lui la volonté d'optimiser la disposition des pièces à découper sur la surface donnée. Comme il n'est pas évident de réaliser cette tâche manuellement, nous avons été chargés de trouver une solution pour l'automatiser de la meilleure manière possible.

Ce rapport présente dans un premier temps le contexte de ce projet et la méthodologie de travail que nous avons adoptée, puis dans un second temps la réponse que nous avons apportée à ce problème.

1 Contexte

1.1 Le client

Ce projet est initié par Eirlab, le fab lab de l'ENSEIRB-MATMECA/Bordeaux INP. Il s'agit d'un atelier de fabrication numérique mettant à disposition de ses membres des outils de prototypage rapide. Les locaux de l'atelier se trouvent dans l'école ENSEIRB-MATMECA, à Talence.

L'atelier a ouvert au printemps 2016. Il est plus particulièrement régi par l'association Eirlab Community. Cette dernière a pour objet la gestion, l'animation et la promotion de l'espace Eirlab au travers d'évènements, de formations gratuites ou payantes. L'association propose également un service de vente de matériaux pour les réalisations.

Peuvent adhérer au club personnes physiques, étudiantes de Bordeaux INP ou non, et personnes morales.

Eirlab dispose de 400m² d'espace *innovation* d'une part avec des stations de travail, et espaces de discussion, et espace de *prototypage* d'autre part. Ce dernier met à disposition de ses membres six imprimantes 3D, une machine de découpe et gravure laser, une perceuse colonne, et autres outils électroniques et mécaniques.

Le client est représenté par M. David RENAULT, comme initiateur du projet et principal interlocuteur pour les prises de décisions et validation du travail accompli.

1.2 Description du besoin

Le projet se place dans le cadre de l'utilisation de la machine de découpe/gravure laser pour usiner des pièces en deux dimensions.

Description de l'existant

La machine présente à l'atelier est le modèle 400 Flexx de Trotec. Elle permet de graver de nombreux matériaux, et de découper ou marquer certains d'entre eux.

L'envoi de travaux à la machine se fait depuis le poste de travail à côté de celle-ci, depuis le logiciel de conception en passant par le logiciel de contrôle d'impression propriétaire *JobControl*. L'usage à Eirlab est d'utiliser le logiciel *Inkscape* pour éditer les fichiers vectoriels SVG à découper/graver.

FIGURE 1 – Machine laser 400 Flexx Trotec

Besoins du client

Afin d'optimiser le temps de découpe et d'économiser les matériaux, le client souhaite faire intervenir un algorithme

de *packing* avant lancement de la découpe. Il s'agit de calculer une disposition des pièces à découper occupant le moins d'espace possible. Cette étape est pour le moment réalisée manuellement, ce qui est fastidieux pour l'utilisateur, et n'assure pas l'optimalité de l'agencement choisi. Voici un exemple en figure 2 fichier source et de résultat attendu :

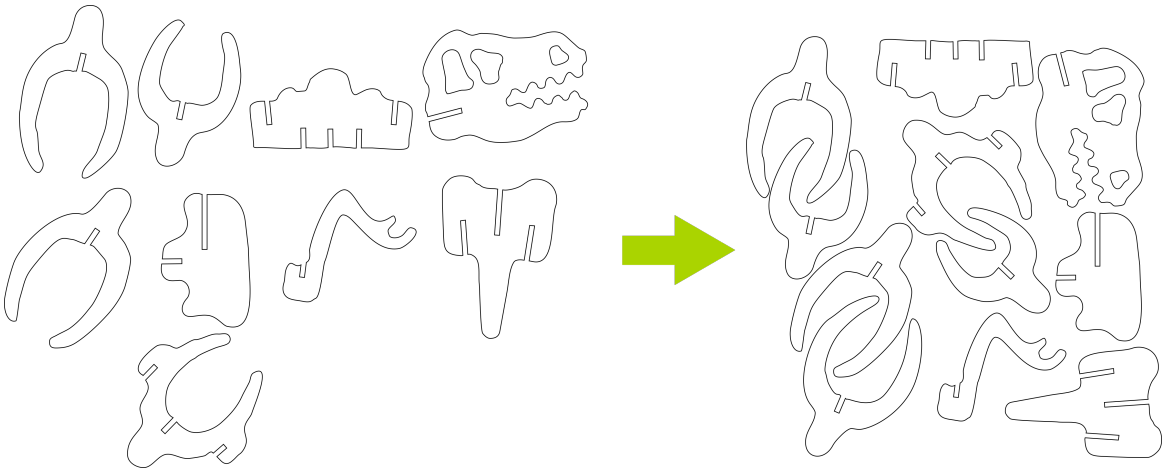


FIGURE 2 – Exemple de packing attendu

La figure 3 suivante représente les différentes étapes traversées lors de l'utilisation de la découpeuse laser. Notre projet doit donc s'inscrire au niveau de l'étape d'édition du fichier vectoriel dans le logiciel Inkscape.

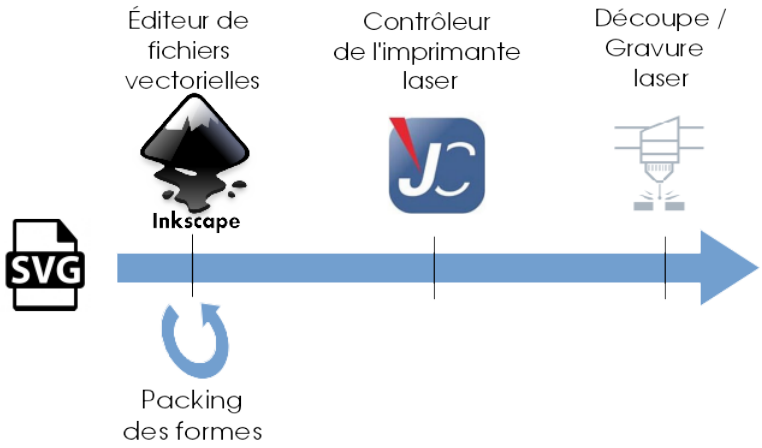


FIGURE 3 – Schéma des étapes du cas d'utilisation général souhaité pour la découpe laser

1.3 Le sujet

Le client a souhaité que soit fourni un plug-in pour le logiciel Inkscape, depuis lequel il puisse déclencher les fonctionnalités de *packing* développées dans notre solution, en plus de l'application depuis le terminal.

Il nous a été demandé de mettre en place un ensemble d'algorithmes réalisant de différentes manières une solution au problème de *packing*. Ces derniers doivent pouvoir être lancés en

parallèle, reprendre la solution de l'un comme entrée de l'autre etc., en somme, garantir une certaine modularité dans leur utilisation.

Le problème de Bin-Packing

La problématique d'optimiser l'espace et le nombre de planches utilisées constitue une instance d'un problème de bin-packing.

Le bin-packing est un problème relevant de l'optimisation combinatoire, ou il s'agit de ranger des objets de tailles $c_1, c_2, ..., c_n$ dans le minimum de boites de taille C(correspondant à des plaques dans notre cas).

Ce problème est présent dans plusieurs secteurs d'activité tels que le rangement de fichier dans un système informatique ou le découpage du verre.

Le problème de Bin-Packing est NP-difficile, donc sauf pour les problèmes de tailles très réduites, les méthodes de résolutions sont principalement heuristiques et métaheuristiques.

Les méthodes heuristiques les plus utilisées de nos jours sont :

— *first-fit decreasing (FFD)* : qui consiste à trier les objets dans un ordre décroissant de taille, puis pour chaque objet, on le range dans la première boite qui peut le contenir.

— *best-fit decreasing (BFD)* : le même tri qu'auparavant, sauf qu'on range l'objet dans la boîte la plus remplie qui puisse le contenir.

Plusieurs articles et thèses existent pour ce problème se basant sur différentes techniques d'optimisation telle que le branch and bound :

Two-Dimensional Finite Bin-Packing Algorithms J. O. Berkey and P. Y. Wang The Journal of the Operational Research Society Vol. 38, No. 5 (May, 1987), pp. 423-429)

A Genetic Algorithm for a 2D Industrial Packing Problem E. Hopper, B. Turton University of Wales, Cardiff, Computers and Industrial Engineering, vol. 37/1-2, pp. 375-378, 1999.

2 Méthodologie et outils de programmation

2.1 Méthode Kanban

La réalisation du projet a été faite en suivant la méthode Kanban, une méthode agile et itérative. La méthode consiste notamment à représenter les tâches composant le projet sur un tableau en plusieurs colonnes, afin que tous les participants puissent suivre l'avancement de ce dernier.

Comme plusieurs autres méthodes agiles, elle repose sur le concept d'*User Story* (U.S.). Cette expression désigne un besoin exprimé du point de vue du client, que l'on peut mettre sous la forme *En tant que [...], je veux [...] afin de [...]*, que le développeur est chargé de réaliser.

Le fait de représenter l'ensemble des tâches à réaliser par le développeur sous forme d'U.S. impose de toujours voir le projet du point de vue du client, ce qui permet de rester pertinent sur ce qui est développé. Une U.S. commence à l'extrême gauche du tableau, et se déplace vers la droite au fur et à mesure qu'elle passe les différentes étapes de réalisation, pour finir à l'extrême droite lorsqu'elle est validée (indépendemment de ses opinions politiques).

La méthode Kanban, à l'inverse d'une méthode comme SCRUM, ne fonctionne pas par cycles mais en flux tendu : au lieu d'avoir un certain nombre d'U.S. à réaliser toutes les x semaines, on se fixe un nombre minimal et maximal d'U.S. qui doivent être en cours de réalisation à tout moment. Ces nombres sont choisis en fonction de la capacité de travail de l'équipe de développement.

De plus, à toute U.S. doit être attribuée une difficulté de réalisation ainsi qu'une valeur client (utilité), qui sont des valeurs sur une échelle de Fibonacci (puisqu'il est contre-productif d'être extrêmement précis dessus). On utilise alors ces deux valeurs pour déterminer quelles U.S. réaliser parmi celles qui sont spécifiées, en prenant par exemple celles qui ont le plus grand ratio *utilité/difficulté*.

Nous avons choisi de mettre dans notre tableau les colonnes suivantes (entre autres) :

— **Product backlog** : c'est ici qu'apparaissent toutes les U.S. du projet à leur création, ainsi que les idées d'ordre plus général, issues de divers brainstormings.

— **U.S. "INVEST"** : recense toutes les U.S. mises au format "INVEST" : pour chacune, des niveaux de difficulté technique et de valeur marchandes ont été déterminés. C'est donc la liste des U.S. normalisées à faire, susceptibles de passer à la réalisation.

— **U.S. en cours** : liste les U.S. en cours de réalisation. Le nombre d'U.S. dans cette colonne est limité afin de ne pas surcharger les collaborateurs du projet. A ce niveau, chaque U.S. doit être attribuée à une partie de l'équipe.

— **Révision** : contient les U.S. dont l'implémentation est finie mais à faire réviser par d'autres membres de l'équipe pour la logique, la lisibilité, les commentaires et plus généralement la qualité du code écrit.

— **Démo** : stocke les U.S. validées par l'équipe mais pas encore par le client. C'est la dernière étape avant qu'une U.S. soit définitivement réalisée et archivée.

Nous avons exploité le site *kanboard.ipb.fr* pour travailler sur ce tableau, dont un peu voir un aperçu en figure 4.

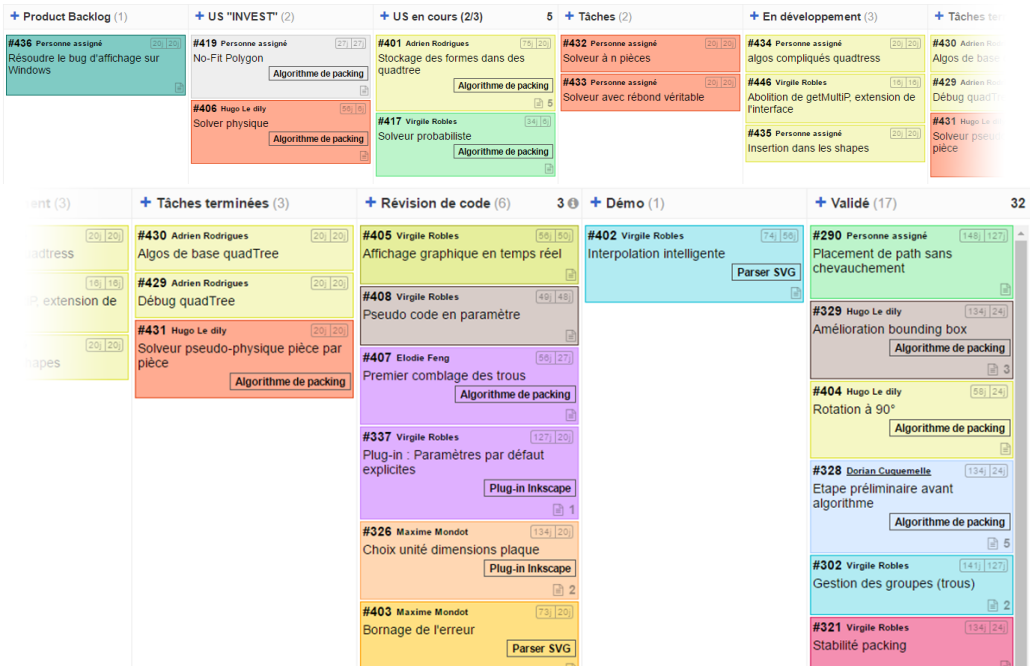


FIGURE 4 – Aperçu de notre Kanboard

2.2 La méthode dans les faits

L'organisation du travail en équipe s'est déroulée comme suit :

— A chaque réunion avec le client et/ou le responsable pédagogique, nous présentions notre travail hebdomadaire, avec démonstration d'une exécution fonctionnelle lorsque cela s'avérait pertinent.

— A l'issue de la réunion, nous organisions les nouvelles tâches à réaliser sous forme de nouveaux éléments dans le product backlog et d'User Stories dans notre Kanboard.

— Ensuite, les membres de l'équipe s'accordaient à travailler en petits groupes sur plusieurs nouvelles U.S., ou continuaient à travailler sur celles qui n'avaient pas été complétées lors de la semaine.

Nous avons apprécié la méthode Kanban car l'agilité nous a permis de confronter régulièrement notre avancement avec les besoins du client et ainsi de mettre à jour les attentes prioritaires, de prendre en compte ses remarques vis-à-vis de notre vision du sujet pour éviter les mauvaises interprétations, etc.

Nous avons cependant eu quelques légers problèmes de travail en équipe.

Conserver une bonne méthodologie, particulièrement pour ce qui est d'une mise-à-jour régulière du Kanban, a parfois été négligé par certains aspects : Les membres de l'équipe communiquant sur l'avancement du projet en dehors des U.S. du Kanboard, ce dernier pouvait ne pas être mis à jour systématiquement. Cet écart à la méthode s'est également manifesté par l'abandon de l'attribution des business value et des difficultés aux User Stories, la perte de la division des U.S. en tâches et la création d'U.S. de manière mal organisée.

De plus, notre projet était réalisé en `C++`, langage appris en même temps que le début du projet. La différence d'expérience en programmation des différents membres de l'équipe se ressentait sur la contribution sur le travail au départ, car certaines personnes étaient déjà habituées à ce langage et d'autres non.

Plus précisément, l'environnement de développement du projet ne favorisait pas celui-ci. En effet, dans le contexte de notre formation, nous avions de nombreux autres projets et examens, nous empêchant de nous consacrer à ce travail aussi régulièrement que l'exige la méthode. Cela met en valeur un problème intrinsèque du PFA : effectuer un travail d'entreprise dans un contexte d'école d'ingénieurs. Il est impossible de se réunir tous les matins pour faire le point sur l'avancement de chacun, et difficile de travailler en pair-programming du fait de la différence des emplois du temps de chacun, entre autres problèmes.

Dans une autre mesure, il s'agissait pour tous de notre première expérience dans un projet à long terme, et avec une équipe aussi grande. La communication est un aspect essentiel au bon fonctionnement d'un projet. Mais par manque de savoir-faire, celle-ci a parfois était imparfaite, si bien que cela a pu nous ralentir voire empêcher la compréhension mutuelle.

Ces problèmes de gestion d'équipe rencontrés nous ont permis de mieux comprendre l'importance de certains points. Si nous avions à refaire ce projet, nous chercherions une méthodologie plus adaptée à l'école, ou encore à adapter une certaine méthode à cet environnement, de sorte à garantir assez de rigueur pour un meilleur travail d'équipe. Le rôle du Scrum Master, membre de l'équipe chargé d'assurer l'application de la méthode par tous les membres, a été envisagé mais jamais mis en place. Il serait donc intéressant de revoir la structuration de l'équipe (un Scrum Master fixe ou à tour de rôle). La rétrospective hebdomadaire, réunion portant sur la méthode et non sur le projet en lui-même, qui vise à déterminer ce qui fonctionne et ne fonctionne pas et à trouver des solutions sous forme de plan d'action, pourrait également être mise en place pour une amélioration continue de notre façon de travailler.

2.3 Outils

Les outils utilisés pour la réalisation du produit sont :

— Langage d'extension Inkscape
— Python : langage de programmation utilisé pour l'interface entre Inkscape et le solveur.
— C++ : langage de programmation pour le solveur
— CMake : outil permettant de gérer la compilation et l'installation des sources.
— Boost : ensemble de bibliothèques C++ utilisé ici pour parser les options passées au solveur
— svgpp : bibliothèque aidant au parsing de fichiers SVG
— rapidXML : bibliothèque de parsing de fichiers XML

— Doxygen : permet de générer la documentation à partir des commentaires sur les sources du produit
— mingw-w64 : ensemble d'outils permettant la cross compilation du produit pour Windows depuis Linux.

Il a été choisi de développer le produit sur Linux à l'aide de la collection de compilateurs GCC. Les plates-formes supportées sont Windows et Linux, les deux devant être compilées depuis Linux.

2.4 Le format SVG

Les données représentant les formes à packer sont décrites sous le format SVG qui est une spécialisation du format XML. Ce format représente une arborescence de données utilisant des balises.

Un noeud est délimité par une balise ouvrante et une fermante, ou bien une balise seule. La balise peut contenir un ensemble d'attributs dépendant de son type. Si le noeud est sous la forme de balises ouvrantes/fermantes, il peut y avoir d'autres noeuds entre ces balises (ce sont alors des noeuds fils).
Un ensemble particulier de noeuds et d'attributs est utilisé lors de notre programme (un exemple de syntaxe SVG est visible en figure 5.

— **Paths** : Les *paths*, et primitives de formes (`rectangle`, `circle`, ...). Ce sont les noeuds qui sont utilisés pour représenter un contour. Ils possèdent un attribut `d` qui contient l'ensemble des informations permettant de décrire ce contours, sous différentes formes : segments, courbes de Bézier, etc.
— **Matrices de transformations** Ce sont des attributs qui sont appliqués au *path* du noeud courant ainsi que tous ses fils. Elles permettent de déplacer un objet sans modifier explicitement les coordonnées de ses points. Elles doivent donc être prises en compte lors de la lecture du fichier. Nous les utilisons également pour appliquer les déplacement effectués par le programme dans le fichier de sortie.
— **Groupes** : Ils forment des ensembles d'objets qui peuvent se comporter comme une entité unique, et sont particulièrement utilisés pour la gestion des formes à trous ou lors de la génération du fichier de sortie (pour délimiter les différentes plaques).
— **Identifiants** : Les noeuds possèdent généralement un attribut `ID`. Ils permettent d'identifier rapidement un objet, chaque objet ayant une valeur unique pour cet attribut. Celui-ci peut être modifié lors de la génération du fichier de sortie en cas de duplication des objets.
— **Width/Height/Viewbox** : Ce sont des attributs propre au fichier. Ils sont utilisés pour déterminer les dimensions d'une plaque par défaut, ils peuvent être exprimés dans différentes unités, qu'il nous faut donc convertir au préalable dans une unité universelle utilisable par le programme.

La figure 5 contient un échantillon de fichier SVG, montrant l'imbrication de différentes balises, et les attributs que l'on peut y trouver.

```
<g
   id="g3363">
  <g
     id="g3465">
    <ellipse
       ry="162.12949"
       rx="110.6117"
       cy="422.5321"
       cx="552.04834"
       id="path3338"
       style="fill:#ffd42a" />
    <ellipse
       ry="31.819807"
       rx="74.751289"
       cy="484.15137"
       cx="542.45197"
       id="path3340"
       style="fill:#ff0000" />
    <ellipse
       ry="42.931484"
       rx="65.154839"
       cy="386.1666"
       cx="545.98743"
       id="path3342"
       style="fill:#ff0000" />
    <rect
       style="fill:#ffff00"
       id="rect3449"
       width="172.73608"
       height="214.15234"
       x="465.6803"
       y="620.01691" />
```

FIGURE 5 – Un morceau de fichier SVG

3 Architecture

3.1 Fonctionnement général

Le projet, comme évoqué précédemment, s'articule autour de deux parties : le plugin `Inkscape` et les algorithmes.

Le plugin `Inkscape`, décrit plus en détail par la suite, envoie à un script Python un certain nombre d'arguments bruts. Ce dernier est chargé d'interpréter les arguments d'Inkscape et de les convertir (notamment au niveau des unités) en un format accepté par notre programme, qu'il appelle avec ces nouveaux arguments.

Le solveur reçoit un fichier `SVG` et les arguments puis renvoie un fichier `SVG` résultat sur la sortie standard, qui est alors affiché par `Inkscape`.

La partie la plus importante du projet est ce programme, dont nous allons par la suite détailler l'architecture. Un diagramme de classe de ce programme est visible en figure 6.

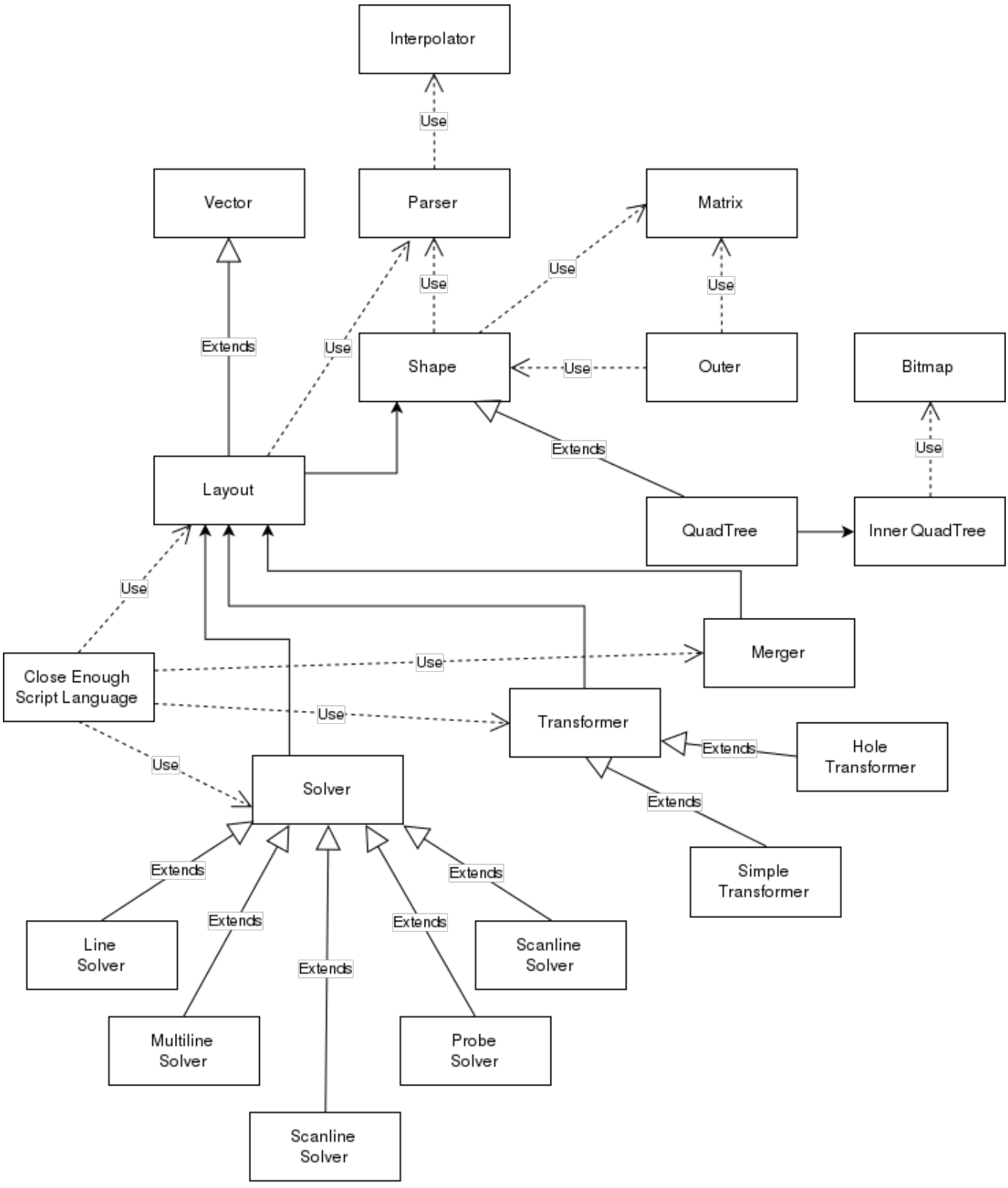


FIGURE 6 – Diagramme de classe du programme principal

3.2 Représentation interne (`Shape`)

Nous introduisons le concept de `Shape` (forme), que nous utiliserons par la suite. Il s'agit d'une classe représentant une forme interpolée. Elle peut représenter un ou plusieurs polygones non indépendants, potentiellement chacun composés de trous. En effet, cette classe contient un

`MultiPolygon`, type de la librairie Boost, lequel contient plusieurs `Polygons`. Enfin ce dernier contient un `Ring` (polygone fermé) décrivant son contour, et potentiellement plusieurs autres décrivant ses trous. Nous verrons dans la partie sur le parseur `SVG` comment nous transformons le document en un ensemble de telles `Shapes`. Cette hiérarchie est illustrée en figure 7.

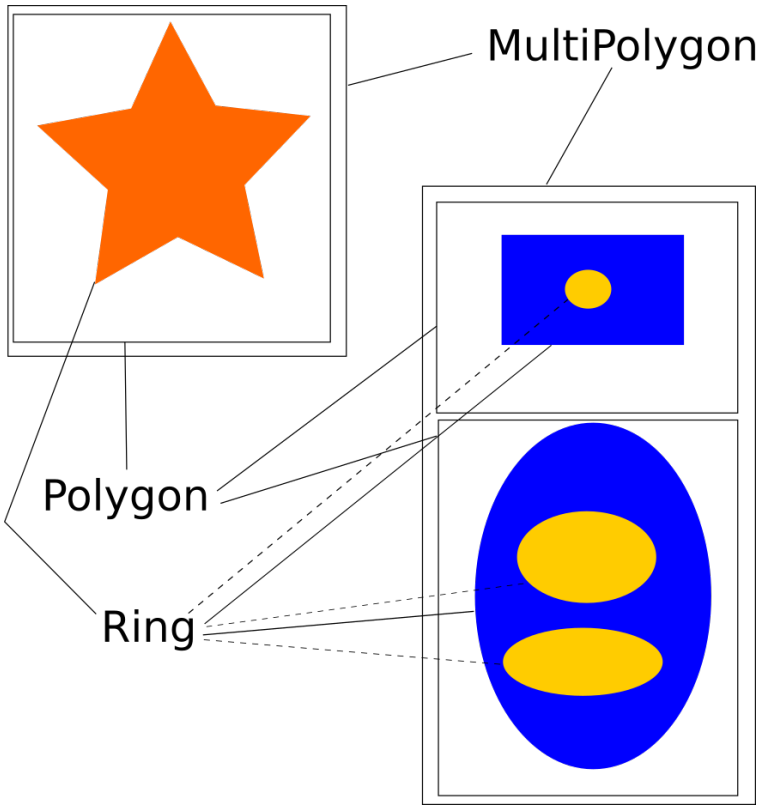


FIGURE 7 – Hiérarchie `MultiPolygon`/`Polygon`/`Ring` (pointillés : `Ring` interne)

Par la suite, il arrivera que nous fassions l'union de deux `Shapes` : il s'agit de générer un unique `MultiPolygon` contenant tous les `Polygons` de la première et de la deuxième `Shape`. On parlera alors de *bloc* de `Shapes` pour décrire une union (en effet faire l'union de deux `Shapes` revient à les considérer comme dépendantes l'une de l'autre : elles seront déplacées "ensemble").

Toutes ces formes sont stockées dans une classe `Layout`, qui décrit donc une disposition des pièces à un instant donné.

3.3 Fonctionnalités mineures

3.3.1 Préparation des pièces

Une fois les pièces générées, on commence par appliquer un *buffer* sur ces dernières, c'est-à-dire augmenter leur épaisseur dans leur représentation interne. Ceci est utile non seulement pour compenser une erreur potentielle lors de l'interpolation des points, mais aussi si l'utilisateur demande à ce qu'une distance minimale entre les pièces soit respectée lors du processus de packing, correspondant à l'épaisseur du trait de découpe de la machine notamment.

Le client a potentiellement intérêt à vouloir une distance minimale entre les plaques. En effet le laser de la machine découpe avec une épaisseur non nulle, ce qui fait que deux pièces collées parfaitement peuvent se retrouver tronquées si le laser passe sur leur frontière commune.

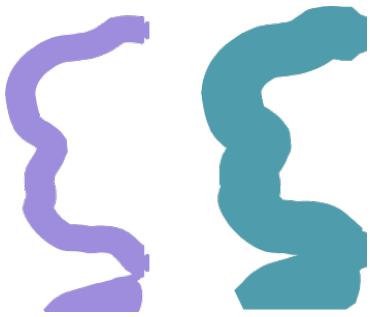


FIGURE 8 – Application d'un *buffer* de 20 *px* à une pièce

On obtient bien le résultat voulu en augmentant l'épaisseur en interne, puisque les pièces seront positionnées comme si elles étaient plus épaisses, sans augmenter l'épaisseur réelle à la sortie. Cette étape est illustrée en figure 8.

3.3.2 Choix concernant la sortie

En effet nous avons choisi lors de la génération du fichier de sortie de ne pas créer de nouvelles formes à partir de notre représentation interne (potentiellement erronée), mais plutôt d'appliquer des transformations sur les formes originales. De cette manière on conserve l'ensemble des attributs donnés pour l'utilisateur.

Cela provoque quelques inconvénients : nous le verrons par la suite dans la classe `Outer`, mais celà nécessite (à cause de l'utilisation à l'entrée de `SVG++`) de parcourir une nouvelle fois le fichier original à la sortie.

3.3.3 Interruption du programme

Le programme contient un gestionnaire de signal, qui intercepte le signal `SIGINT`. Lors de la reception de ce dernier, des variables statiques à certaines classes sont changées, et les algorithmes doivent s'arrêter le plus vite possible et produire une sortie. Ceci permet d'interrompre un calcul trop long pour obtenir une solution rapidement.

Si le signal est une nouvelle fois reçu avant la fin du calcul, le programme s'arrête alors brutalement (via une exception), ce qui est le comportement attendu de `SIGINT` à l'origine.

Ce signal ne peut être envoyé que grâce à la commande `Ctrl-C` dans un terminal ou grâce à une commande de type `kill`. Dans le cas d'Inkscape, le bouton *Cancel* qui s'affiche pendant

le calcul se génère aucun signal récupérable par notre programme. Il faut donc l'envoyer via les méthodes classiques.

3.3.4 Affichage

Pour des besoins de développement, nous avons réalisé un petit affichage basé sur la `SFML` qui permet de visualiser un `Layout` (ensemble de pièces). L'affichage se manipule entièrement avec des méthodes statiques, qui permettent de charger un `Layout` ou de mettre à jour des pièces. Il offre également la possibilité d'afficher du texte dans la fenêtre, pour par exemple suivre la progression du programme.

En effet, pour ne pas avoir d'impact sur les performances des algorithmes, nous avons fait en sorte que cet affichage consomme le moins possible de ressources. Puisque suivre les mouvements des pièces en temps réel demande beaucoup de ressources, il est exécuté dans un *thread*, et l'écran n'est rafraîchi que lorsque que l'utilisateur demande manuellement de mettre à jour la position d'une pièce.

Cet affichage est très pratique, notamment lorsqu'il s'agit de tester les solveurs non déterministes (un exemple est visible en figure 9). Il est d'autant plus pratique qu'il permet de visualiser la représentation **interne** des algorithmes, c'est à dire avec le *buffer* appliqué et après interpolation.

Toute la partie affichage doit être activée à la compilation (avec le flag `DISPLAY`), et activée à l'exécution (avec le flag `-D`) pour être effective. Sans ces activations, les appels vers des fonctions de `Display` sont remplacés par des instructions sans effet.

FIGURE 9 – Exemple d'affichage interne

3.3.5 Logs

Nous avons mis en place un système de *logging* minimaliste. Tout message doit avoir un niveau de sévérité parmi `{trace, debug, info, warning, error}`, et est affiché sur la sortie d'erreur (la sortie standard étant réservée au format SVG) seulement si son niveau est supérieur ou égal au niveau général de *logging* défini à la compilation.

Le système s'utilise comme un flux : pour écrire un message on fait par exemple :

```
LOG(info) « "Ceci est un message d'information" « endl;
```

La partie `LOG(...)` est alors remplacée par `cerr` ou bien par un flux vide (équivalent à `/dev/null`) selon le niveau.

3.4 Fonctionnement interne des modules

3.4.1 Plug-In

Afin de gérer l'interface entre Inkscape et notre programme, nous avons implémenté un plug-in python. Celui-ci s'occupe de récupérer l'ensemble des paramètres transmis par Inkscape. Il effectue également un ensemble d'opérations permettant de convertir les unités de mesure en pixel, qui sont utilisables par notre programme. Il permet notamment d'extraire les dimensions de la plaque depuis le fichier, puis de les convertir en pixels.
Pour réaliser ce travail nous avons utilisé la bibliothèque `inkex.py` fournie par Inkscape.

3.4.2 Parser et SVG++

Nous utilisons la bibliothèque `SVG++` pour lire le fichier d'entrée du programme . Pour utiliser celle-ci, il faut définir un ensemble de comportements à adopter lors de la lecture des balises qui nous intéressent. Les méthodes implémentées au cours du projet sont visibles sur la figure 10.

```
class Context {
public:
    ///SVG++ Methods
    void transform_matrix(const boost::array<double, 6>& matrix);
    void set(svgpp::tag::attribute::id, const boost::iterator_range<const char*> pId);
    void set(svgpp::tag::attribute::width, double width);
    void set(svgpp::tag::attribute::height, double height);
    void set(svgpp::tag::attribute::viewBox, double, double, double, double);
    void path_move_to(double x, double y, svgpp::tag::coordinate::absolute);
    void path_line_to(double x, double y, svgpp::tag::coordinate::absolute);
    void path_cubic_bezier_to(double x1, double y1, double x2, double y2, double x,
                              double y, svgpp::tag::coordinate::absolute);
    void path_close_subpath();
    void path_exit();

    void on_enter_element(svgpp::tag::element::any);
    void on_enter_element(svgpp::tag::element::g);
    void on_exit_element();
};
```

FIGURE 10 – Méthodes à implémenter dans la classe Context de SVG++

Ces comportements peuvent être généralisés. Par exemple, il est possible d'utiliser le comportement des courbes de Bézier sur tout les `path` rencontrés. La bibliothèque effectue alors la conversion nécessaire pour appliquer le comportement choisi.

Ceci permet notamment de ne pas gérer au cas par cas toutes les balises existantes, en ignorant celle n'étant pas utiles pour le bon fonctionnement du programme, et en utilisant un comportement générique pour une certaine variété de balises (`path`).

Cependant, l'implémentation de la bibliothèque est riche en métaprogrammation template. La métaprogrammation est un outil permettant en particulier de générer du code à la compilation. Par exemple, il est possible d'écrire une fonction avec un type générique qui sera ensuite déterminé lors de la compilation.
Il est également possible de pré-calculer des données lors de la compilation qui seront généralement stockées dans des variables statiques.
Ceci permet d'améliorer l'efficacité de certains programmes et d'écrire du code plus générique. Les temps de compilation sont néanmoins plus longs, les métaprogrammes sont assez peu lisibles, et il est impossible de redéfinir une méthode template dans une classe fille en C++.

Il n'est alors pas possible d'utiliser le comportement habituel de la Programmation Orienté Objet (POO) (redéfinition de méthodes dans une classe fille). Il est donc nécessaire de préciser quelles méthodes sont implémentées dans un ensemble utilisé par cette bibliothèque.
De plus celle-ci ne nous permet pas de stocker le contenu du fichier. Nous devons donc réeffectuer une lecture au moment de la génération du fichier de sortie, et les temps de compilation sont fortement affectés.

La grande majorité des formes étant décrites avec des courbes de Bézier, il était nécessaire d'avoir une méthode d'interpolation permettant d'obtenir des points à partir de ces courbes. Grâce à SVG++ toute courbe est ramenée à des courbes de Bézier en dimension 3. Par la suite, nous avons utilisé une méthode plus intelligente, mais la méthode d'interpolation la plus simple est la suivante (avec $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$ et $\mathbf{P}_3$ les points définissant la courbe de Bézier) :

$$\mathbf{P}(t) = \mathbf{P}_0(1-t)^3 + 3\mathbf{P}_1 t(1-t)^2 + 3\mathbf{P}_2 t^2(1-t) + \mathbf{P}_3 t^3, t \in [0,1]$$

On fait varier t avec un pas fixé pour obtenir $\frac{1}{t}$ points sur une courbe, que l'on insère dans les `Shapes`.

On obtient pour chaque `path` un polygone fermé (`Ring`). Pour répartir ces polygones dans des `Shapes`, nous utilisons un algorithme qui trie les `Rings` par aire décroissante et teste la superposition des uns sur les autres. On détermine alors pour chaque `Ring` s'il s'agit d'un contour ou d'un trou.

Enfin il faut décider si un ring fait partie d'un bloc : l'utilisateur a la possibilité de définir des ensembles de pièces qui seront déplacées ensembles en les mettant dans un **groupe** SVG. Il faut alors utiliser la présence des balises de groupe pour trier les polygones.

3.4.3 Algorithmes de packing

Transformers, Merger

La classe `Transformer` est une classe abstraite dont les filles doivent s'occuper de déplacer plusieurs ensembles de `Shapes` qui seront ensuite considérés comme des 'blocs'. Après déplacement des `Shapes`, le transformer retourne un vecteur contenant les vecteurs représentant les `Shapes`, liées par 'bloc".

Cette variable de retour est ensuite récupérée par la classe `Merger` qui s'occupe de modifier la structure interne des données pour gérer les 'blocs' créés par le Transformer dans les algorithmes suivants.

Lors de sa création, la classe `Merger` sauvegarde les informations nécessaires à la génération du fichier SVG. En effet lors d'une phase de fusion, on ne considère que les points, eux seuls étant utiles aux algorithmes de packing.

Afin de conserver l'intégrité de la représentation au cours de l'exécution, le `Merger` stocke des ensembles d'identifiants indiquant à quelle forme appartenait un polygone (ou quadtree) à l'origine.

Un exemple de fonctionnement du merger est donné en figure 11.

Les deux classes `Transformer` et `Merger` ont été séparées car l'exécution du `Merger` est indépendante de celle du Transformer choisi. Cela permet également de simplement modifier la configuration de départ d'un algorithme en ne faisant aucun appel à `Merger`.

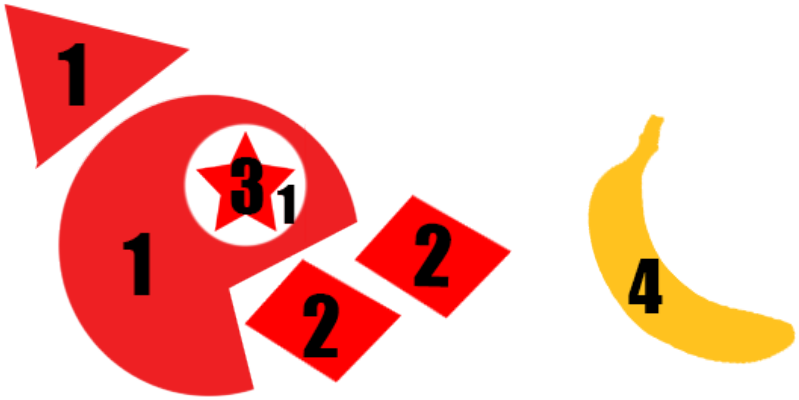
Solvers

La classe `Solver` est également abstraite, son rôle est d'appliquer un algorithme de résolution sur la totalité des formes.

Elle contient les méthodes virtuelles `preSolve` et `solveBin`, qui permettent respectivement d'effectuer le traitement préliminaire nécessaire au bon fonctionnement du `Solver` et d'appliquer l'algorithme de résolution sur une seule plaque (en utilisant un sous-ensemble des `Shapes`). La méthode `solveBin` travaille sur une liste d'indices lui indiquant les formes qu'il reste à packer. Lorsqu'une forme est effectivement packée, son indice est retiré de la liste.

`soveBin` est appelée tant que la liste d'indices restants est non vide. Travaillant à chaque fois sur une nouvelle plaque. L'algorithme appliqué dépend du `Solver` choisi (Scanline, Proba, ...).

(a) Représentation des `Shapes` {<1,1,1>,<2,2>,<3>,<4>}



(b) Fusion des `Shapes` 1, 2 et 3

FIGURE 11 – Fusion des groupes de `Shapes` : {<1,1,1,2,3,3>,<4>}

Les tranformations faites par un `Solver` sont appliquées en place sur le `Layout` passé en paramètre à `Solver`, lequel est copié uniquement en cas d'exécution parallèle de plusieurs `Solvers` comme c'est le cas dans le `TaskSolver`.

3.4.4 Outer

La dernière étape de l'exécution a pour objectif d'écrire un fichier SVG contenant toutes les formes sélectionnées par l'utilisateur après leur traitement par la classe `Solver`. Ceci est réalisé par la classe `Outer` qui, à partir des formes précédemment déplacées par le `Solver` et de leurs identifiants récupérés par la classe `Parser`, va recopier les éléments associés à ces identifiants sur la sortie standard.

On re-parcourt donc l'arborescence du fichier SVG, le texte correspondant n'ayant pas pu être stocké au préalable par SVG++. Nous utilisons donc `rapidXML`, de manière à sélectionner les éléments en fonction de leur identifiant, pour ensuite ajouter un champ contenant la matrice de transformation associée aux déplacements effectués lors du packing (ou de modifier si le champ existe déjà).

La représentation obtenue au format SVG est alors stockée sous forme de chaîne de carac-

tères dans un attribut de l'instance associée.

Dans le cas d'une duplication en bas de page, tous les éléments de l'arborescence parsés lors du parcours sont d'abord recopiés sans être modifiés. Les formes sont ensuite regroupées par plaques en utilisant leurs coordonnées. Ces groupes sont ensuite écrits dans le format SVG sur la sortie standard qui sera récupérée par le plugin ou par l'utilisateur pour être écrite dans un fichier.

3.5 Langage de script

Tous les algorithmes précédemment évoqués sont dans des classes. Cela signifie que pour exécuter certains algorithmes dans un certain ordre, il fallait à l'origine instancier chacune des classes et appeler les bonnes méthodes sur ces derniers, imposant de recompiler le projet pour tout changement dans cet ordre ou dans les paramètres passés aux algorithmes.

Le temps de compilation du projet étant tout sauf instantané, il nous est apparu qu'il était nécessaire d'avoir une solution permettant de définir cet ordre à l'exécution plutôt qu'à la compilation. Nous avons alors eu l'idée d'introduire un langage de script extrêmement minimaliste permettant simplement de définir un ordre d'exécution ainsi que de passer des paramètres aux algorithmes.

Plutôt que de réaliser un parser ad-hoc pour ce nouveau langage, nous avons préféré le définir sous la forme d'une grammaire, et nous reposer sur le module `Qi` de la librairie `Boost Spirit` qui permet de construire des parsers basés sur une grammaire définie entièrement en C++. Cela permet d'éviter une étape supplémentaire du type `yacc` grâce à une redéfinition habile de multiples opérateurs.

Nous voulions comme fonctionnalités de base l'appel d'un `Solver` ou d'un `Transformer` avec des paramètres potentiels, ainsi que la possibilité de répéter des blocs d'instructions. Nous sommes alors arrivé à la grammaire suivante :

$$
\begin{aligned}
\langle \textit{string} \rangle &::= [\hat{}\,,()=]^{+} \\
\langle \textit{value} \rangle &::= \text{int} \\
&\quad | \quad \text{double} \\
&\quad | \quad \langle \textit{string} \rangle \\
\langle \textit{parameter} \rangle &::= \langle \textit{string} \rangle \text{ '=' } \langle \textit{value} \rangle \\
\langle \textit{parameter\_list} \rangle &::= \langle \textit{parameter} \rangle \\
&\quad | \quad \langle \textit{parameter} \rangle \text{ ',' } \langle \textit{parameter\_list} \rangle \\
&\quad | \quad \epsilon
\end{aligned}
$$

⟨*transformer*⟩ ::= `SimpleTransformer`
| `HoleTransformer`
| ...

⟨*solver*⟩ ::= `LineSolver`
| `MultilineSolver`
| ...

⟨*function*⟩ ::= ⟨*transformer*⟩
| ⟨*solver*⟩

⟨*instruction*⟩ ::= ⟨*function*⟩ '(' ⟨*parameter_list*⟩ ')' ';'

⟨*block*⟩ ::= ⟨*instruction*⟩
| `BEGIN` ⟨*instruction*⟩$^+$ `END`

⟨*big_block*⟩ ::= ⟨*block*⟩
| `DO` int `TIMES` ⟨*block*⟩

⟨*program*⟩ = ⟨*big_block*⟩*

On constate qu'une instruction est alors toujours de la forme

`Fonction(param1=valeur1, param2=valeur2, ...);`.

et qu'on peut faire des blocs de telles instructions avec les mots-clés `BEGIN` et `END`, puis repéter de tels blocks avec la syntaxe `DO <X> TIMES <bloc>`.

```
DO 2 TIMES
     BEGIN
        HoleTransformer();
        SimpleTransformer(rotate_step=30, criteria=box);
     END
ScanlineSolver();
ProbaSolver(amplitude_proba=40., initial_placement=0);
```

FIGURE 12 – Un exemple de script

On peut ainsi voir un exemple de script en figure 12. Dans ce dernier, on commence par exécuter `HoleTransformer` puis `SimpleTransformer` (auquel on passe un entier et une chaîne de caractères), le tout deux fois. On appelle ensuite `ScanlineSolver` et `ProbaSolver`, auquel on passe un flottant et un entier.

À la manière de `yacc`, on associe à chaque règle un type et donc une valeur qui sera affectée à une instance de cette règle lors du parsing. On définit les actions grâce à une syntaxe spécifique à `Qi`. Nous avons choisi de stocker des informations basiques comme les noms, l'ordre, les paramètres des fonctions au fur et à mesure du parsing, et d'exécuter tout ce qui a été stocké à la fin de chaque `<big_block>`.

Lors de l'exécution, il fallait associer à un nom de fonction dans le script une classe à instancier. Cela était au départ réalisé par une simple structure du type `if ... else if...`, mais quand le nombre d'algorithme a augmenté, nous avons voulu adopter une méthode plus générique.

Nous avons alors réalisé un mécanisme de registre, utilisant de manière intense la metaprogrammation et les templates, permettant simplement d'instancier sur le tas une classe à partir de son nom et de lui transmettre les paramètres de l'utilisateur de façon entièrement automatique. Un registre est en fait un type disposant de méthodes statiques et se crée de la manière suivante :

```
using NewReg = Registry<Base, boost::mpl::set<Derived1, Derived2, ...>::type>
```

Les classes `Derived` devant hériter de `Base`. Toutes les classes doivent prendre en paramètre un ensemble de formes et un tableau de paramètres. Après la définition de ce nouveau registre, il faut initialiser sa table interne en appelant `NewRegistry::Init()` une unique fois. Il suffit ensuite pour instancier une classe depuis son nom d'appeler :

```
NewReg::instanciate("Derived2", shapes, params)
```

Cette fonction retourne un `Base*` contenant une instance de `Derived2`. De plus, les règles de la grammaire peuvent être écrites de manière automatique grâce à ce mécanisme (en exploitant le registre, qui contient en interne une `map` entre chaînes de caractère et pointeurs sur fonction).

Ainsi lorsqu'on veut ajouter un `Solver` ou un `Transformer` à la grammaire, il suffit seulement d'ajouter son type à la définition du registre, et tout le reste est généré automatiquement. Nous obtenons donc un langage de script très facile à étendre.

Pour exécuter nos algorithmes, il ne reste alors qu'à appeler le parser généré depuis le `main`, et tous les algorithmes sont exécutés au cours du parsing.

4 Algorithmes

4.1 Transformers

La classe Transformer a pour but de regrouper plusieurs sous-ensembles de formes en 'blocs'. Ce sont des algorithmes appliqués sur les formes avant l'étape de packing, ils servent à augmenter l'efficacité globale du programme, en cherchant des formes à assembler ensemble selon différents critères : les formes d'un 'bloc' sont choisies de manière à ce que l'aire occupée par le bloc soit la plus faible possible. Certains transformers utilisent plusieurs critères de qualité : aire d'intersection entre enveloppes convexes, taille de la bounding box résultante etc.

L'intérêt de la classe Transformer est de pouvoir proposer un packing plus optimal qu'un algorithme solveur, mais seulement de manière locale (temps d'exécution qualitativement excessif).

En fin de tâche, la classe Transformer envoie un ensemble d'informations à la classe Merger pour que celle-ci puisse modifier la représentation interne des formes afin que les algorithmes suivants considèrent effectivement les formes déplacées comme un bloc insécable.

4.1.1 Simple Transformer

Le `Simple Transformer` est un algorithme de packing *brute force* qui, pour deux formes données, teste un ensemble de dispositions entre elles pour optimiser un critère de l'union résultante.

Pour cela, on effectue une série de rotations de la première forme sur elle-même, d'un angle α. Puis on effectue une autre série de rotations d'un angle β sur la seconde forme de la même façon. Pour chaque couple de rotations, la seconde forme est ensuite déplacée de façon à être alignée avec la première forme, selon l'axe x. Un léger *offset* selon l'axe y est ajouté pour tester toutes les configurations possibles (à la précision de l'*offset* et des angles près).

On minimise la distance entre les deux formes en translatant la deuxième selon l'axe des x, jusqu'à ce que les deux pièces se touchent, le tout par dichotomie. Tout le procédé est illustré en figure 13.

On garde finalement le triplet $(\alpha, \beta, offset)$ minimisant un certain critère, passé en paramètre. Nous avons par exemple la minimisation de la bounding box résultante (utile lorsqu'on va utiliser un algorithme sur rectangles ensuite) ou bien la maximisation de l'aire d'intersection entre les enveloppes convexes.

4.1.2 Hole Transformer

Le but de cet algorithme est de remplir les trous de formes trouées avec d'autres pièces de l'ensemble de Shapes. Cette étape peut s'avérer utile car dans les solveurs employés par la suite, on utilise au mieux les enveloppes extérieures des formes, sinon leur bounding box, empêchant ainsi toute pièce de jamais combler les trous d'une autre.

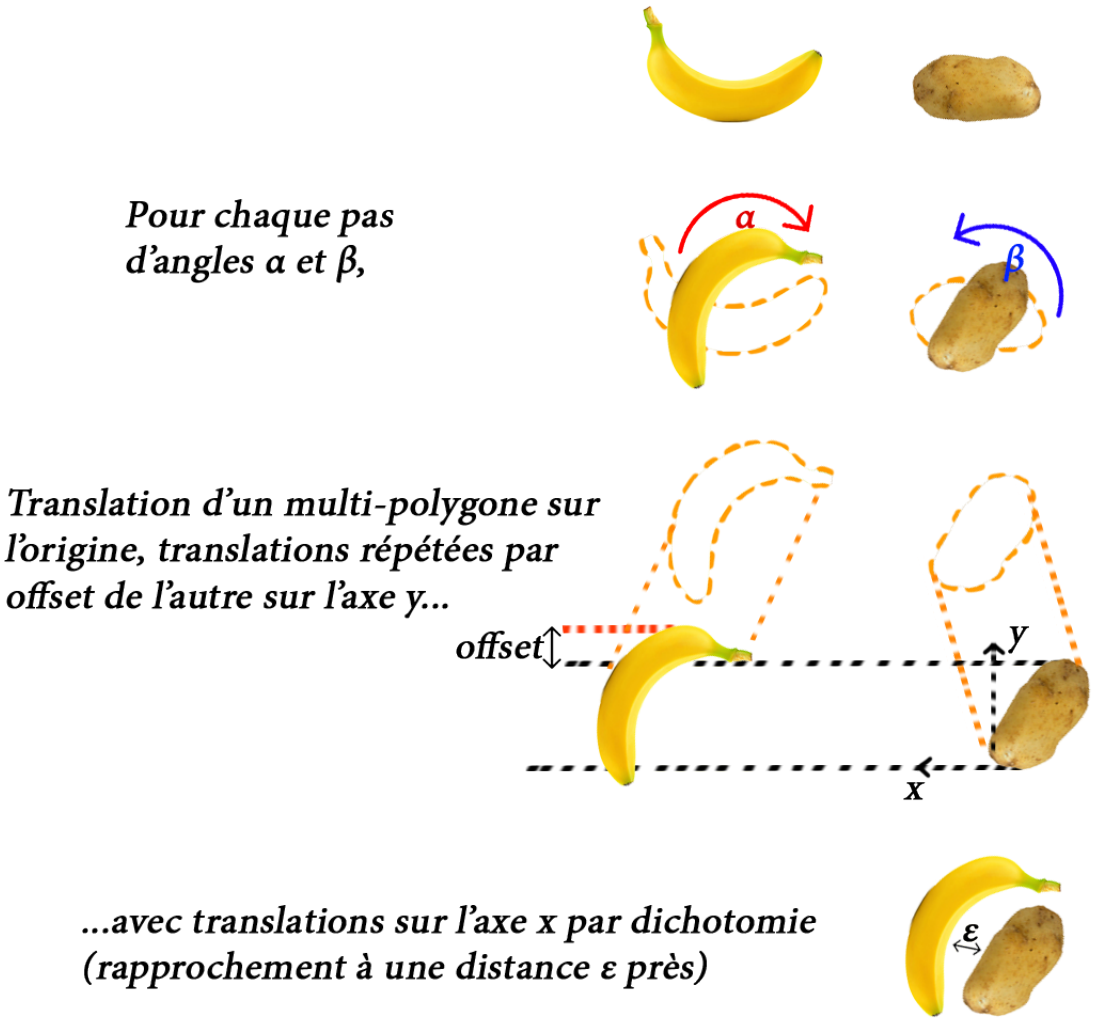


FIGURE 13 – Illustration du procédé de `SimpleTransformer`

Le pseudo-algorithme 1 ci-dessous décrit le fonctionnement de ce transformer.

Cet algorithme est assez simple, il ne place au mieux qu'une seule forme par trou - en son centre, et n'effectue pas de rotation pour une meilleure recherche de compatibilité.

4.2 Solvers

4.2.1 Algorithmes par *bounding box*

Trois algorithmes ont été implémentés en utilisant des boîtes englobantes rectangulaires englobant les différentes formes : le problème est alors ramené à un packing de rectangles, beaucoup plus simple (mais qui reste *NP-complet*).

Les deux premiers solveurs, `LineSolver` et `MultilineSolver` sont des plus basiques, mais ils permettent au moins de disposer les pièces sur les plaques sans intersection. L'algorithme

Algorithme 1 - Hole Transformer

Entrée : Ensemble de formes
Sortie : Vecteur de n-uplets des indices de formes à fusionner, avec leur nouvelle position
Début
  Établir une liste des trous triés par ordre décroissant d'aire ;
  Trier les formes par ordre croissant d'aire ;
  Pour chaque trou t :
    Pour chaque forme f :
    Si aire(f) < aire(t) **et** f n'a pas déjà été choisie :
      Déplacer f au centre de t ; //centre f = centre t
      Si pas de collision entre f et t :
        Ajouter (f,t) au vecteur de sortie ;
        Marquer f comme déjà choisie ;
  Retourner le vecteur de tuples ;
Fin

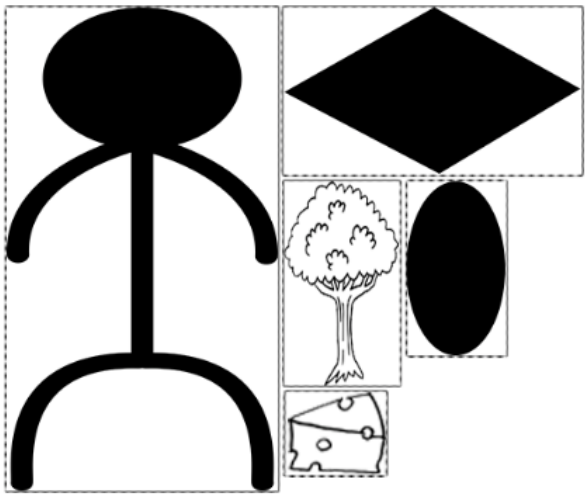


FIGURE 14 – Illustration du procédé des bounding boxes

`ScanLineSolver` est plus complexe et fournit une bien meilleure compacité.

LineSolver, MultilineSolver

L'algorithme `LineSolver` se contente d'aligner les formes de façon désordonnée sur une ligne, sans retour lorsque l'on dépasse la taille de la plaque. Pour cela il colle les rectangles entre eux au plus proche. Lorsque la largeur de la plaque est dépassée, il passe à une plaque suivante.

L'algorithme `MultilineSolver` en est une version améliorée. Les différentes boîtes sont triées par hauteur et sont placées par hauteurs décroissantes. Le packing va alors se faire ligne par ligne, en insérant les pièces depuis le bord gauche jusqu'à atteindre le bord droit de la plaque. Les lignes sont ensuite juxtaposées les unes en dessous des autres. Un exemple de sortie peut être constaté en figure 15.

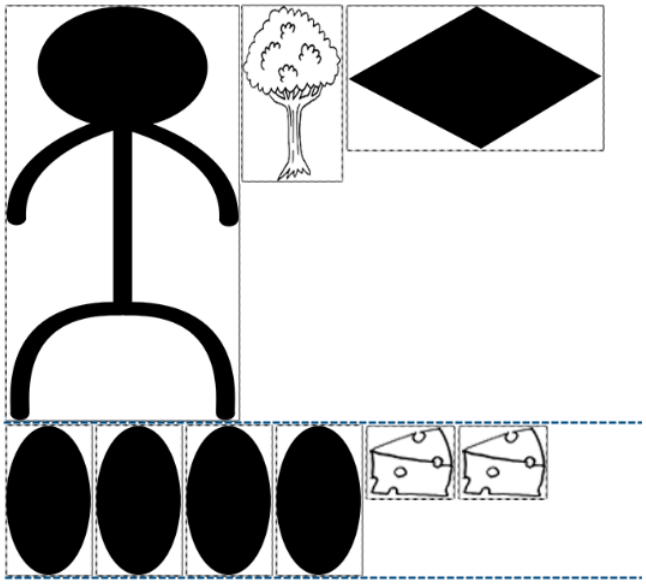


FIGURE 15 – Exemple d'une sortie du `MultilineSolver`

ScanlineSolver

Le solveur `ScanLine` utilise l'algorithme connu du *first-fit*. Il commence par affiner la boîte englobante au mieux : des rotations sont effectuées sur les pièces afin d'obtenir la boîte avec un rapport (aire vide / aire boîte) le plus faible. Ces boîtes sont ensuite triées par ordre décroissant de hauteur, comme l'illustrent la figure 16.

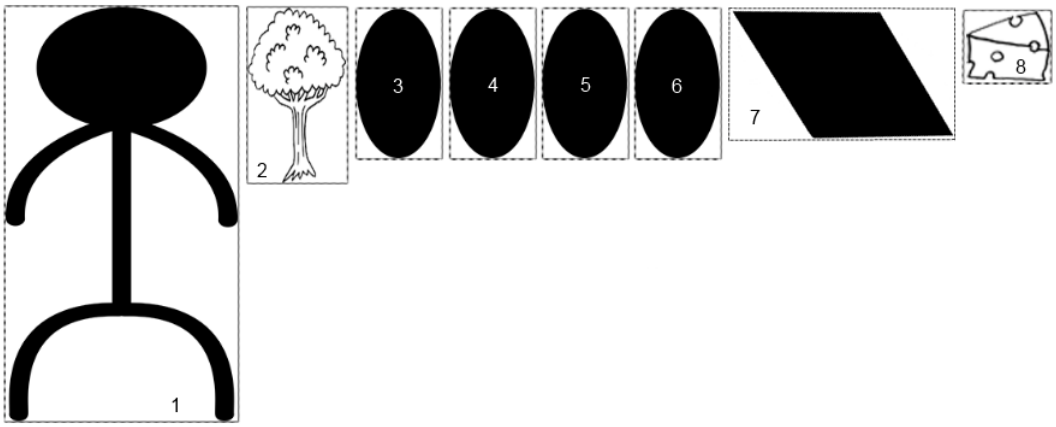


FIGURE 16 – Jeu de pièces, affinées à la meilleure bounding box, puis triées

L'essentiel du packing va consister à manipuler une grille de cellules rectangulaires dans l'état "plein" ou "vide", et de déterminer un ensemble de cellules dans lequel une boîte peut rentrer. Le parcours de cette grille s'effectue colonne par colonne de haut en bas, de gauche à droite. Dès qu'une place est trouvée pour la boîte, c'est-à-dire un ensemble de cellules toutes vides, la grille est mise à jour pour correspondre exactement aux nouvelles cellules pleines et vides avec les boîtes placées (figure 18).

Il se peut que la succession des insertions et la maximisation de la compacité laisse une cellule vide entre les pièces, les plus grandes ne pouvant pas s'y placer (figure 19). Toutefois les pièces sont triées par ordre décroissant de hauteur, aussi les pièces les plus petites combleront

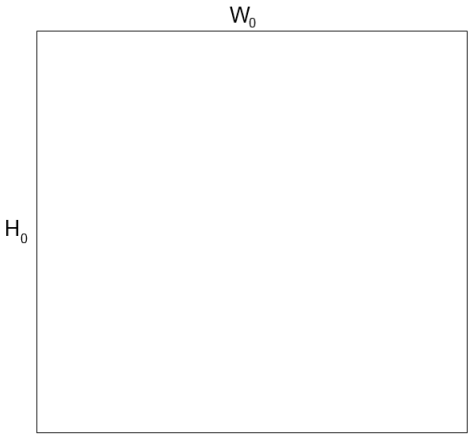
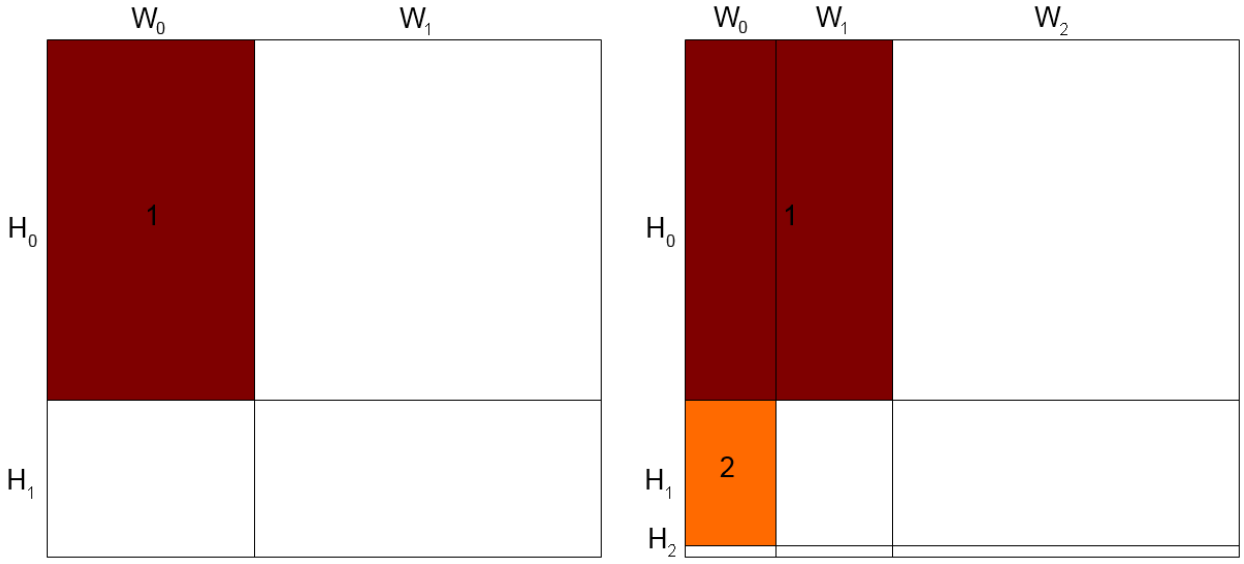


FIGURE 17 – La plaque initiale est une grille à une seule cellule vide

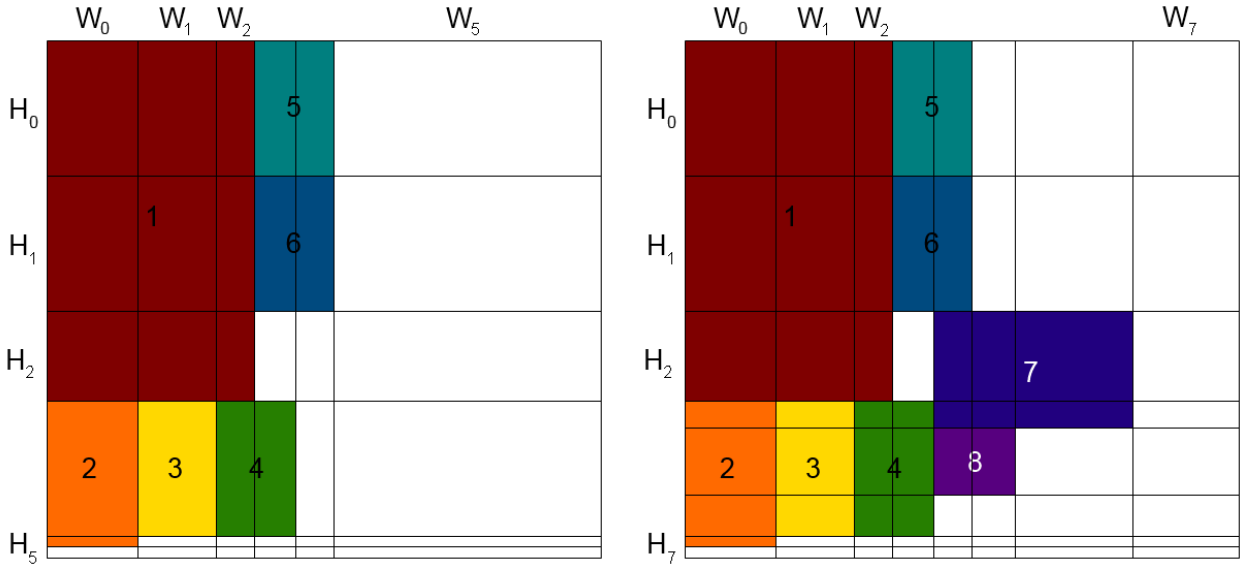


(a) Ajout de la pièce 1 en (W_0, H_0) (b) Ajout de la pièce 2 en (W_0, H_1)

FIGURE 18 – A chaque ajout de pièce, la grille est redécoupée

ces cellules vides pour améliorer davantage la compacité de la plaque finale.

Enfin, une dernière optimisation est effectuée une fois que toutes les pièces ont tenté une insertion : ces pièces sont tournées de 90°, puis un nouvel essai d'insertion. Cela conserve les propriétés rectangulaires dont le solver a besoin, et une pièce plus large que longue peut donc rentrer dans un ensemble de cellules jusque-là trop petit.



(a) Grille après insertions des pièces 1 à 6 (b) Les insertions peuvent créer des trous

FIGURE 19 – Une pièce s'insère le plus à gauche, puis le plus en haut possible

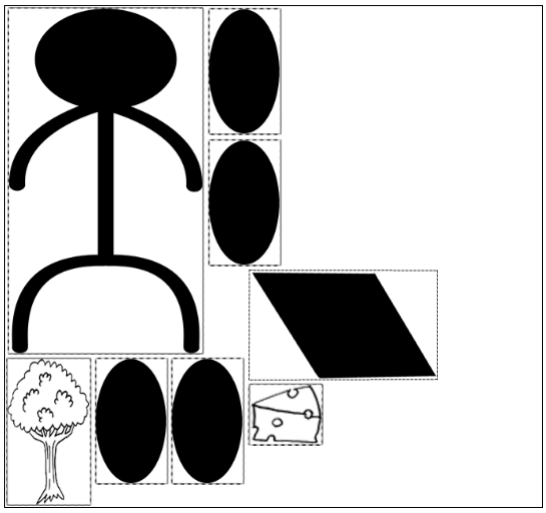


FIGURE 20 – Sortie de l'algorithme `ScanlineSolver`

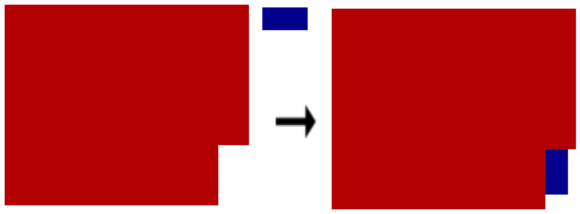


FIGURE 21 – Une pièce trop large peut être insérée après rotation

Performances

Sur des jeux de pièces générés aléatoirement, Le solveur `MultilineSolver` tend vers un ratio de compression (l'aire totale des pièces packées avec trous générés, sur la somme des aires des pièces sans les trous) supérieur à 400% pour un jeu de rectangles, et de 300% pour un jeu de

triangles.

Le solveur `ScanlineSolver` offre de meilleures performances en moyenne, avec des ratios de compression entre 200% et 130% pour les même types de jeux de pièces.

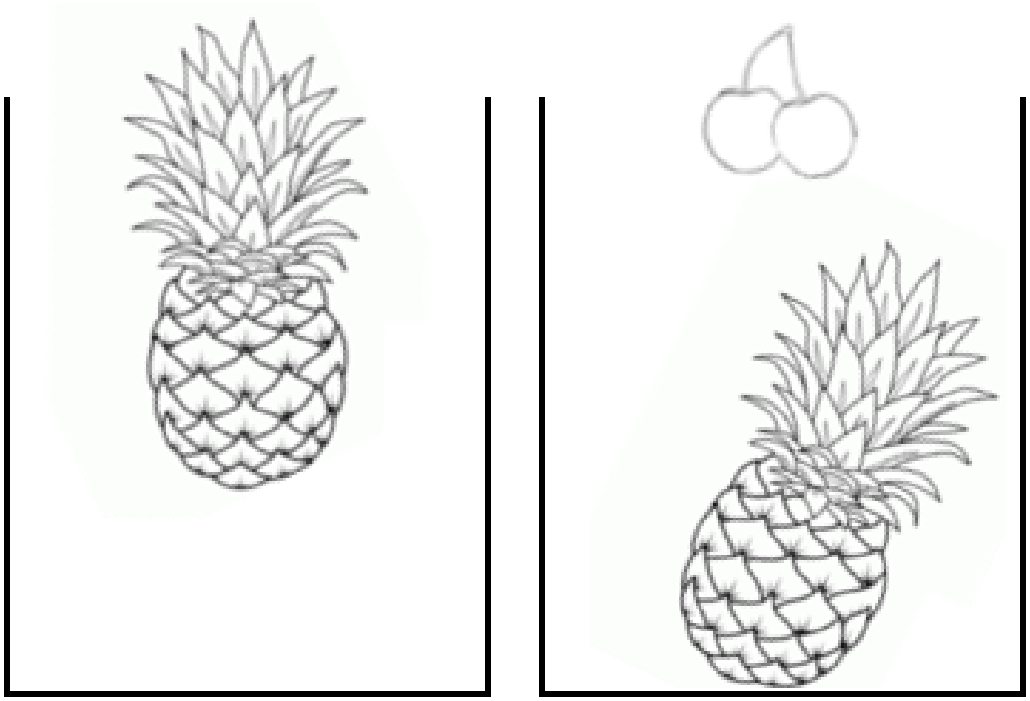
4.2.2 Algorithmes non déterministes

Les autres algorithmes implémentés insèrent les pièces de manière partiellement aléatoire sur la plaque.

FreezeSolver

Le solver `FreezeSolver` assimile la plaque à un panier que l'on remplit, et fait tomber les pièces une à une dans la plaque grâce à de petites translations qui simulent la gravité pendant un certain nombre d'itérations.

La pièce tombe alors vers le bas de la plaque et en cas de collision avec les bords ou le fond, la pièce est repoussée vers l'intérieur. Après un certain temps de calcul, le solver vérifie que la position de la pièce est valide (c'est-à-dire entièrement dans la plaque, et sans collision générée par les dernières translations-rotations), la pièce se fige et la suivante est lâchée à partir du haut (figure 22).



(a) La pièce 1 est lachée du haut de la plaque

(b) La pièce 1 se fige, et la pièce 2 est lâchée

FIGURE 22 – Le `FreezeSolver` s'apparente à un panier

Si une pièce entre entre collision avec une autre lors du calcul, alors elle est expulsée vers le

haut, translatée aléatoirement à gauche ou à droite, et pivotée aléatoirement dans un sens ou dans l'autre avant de recommencer à tomber, ce qui simule un entassement des pièces dans la plaque (figure 23).

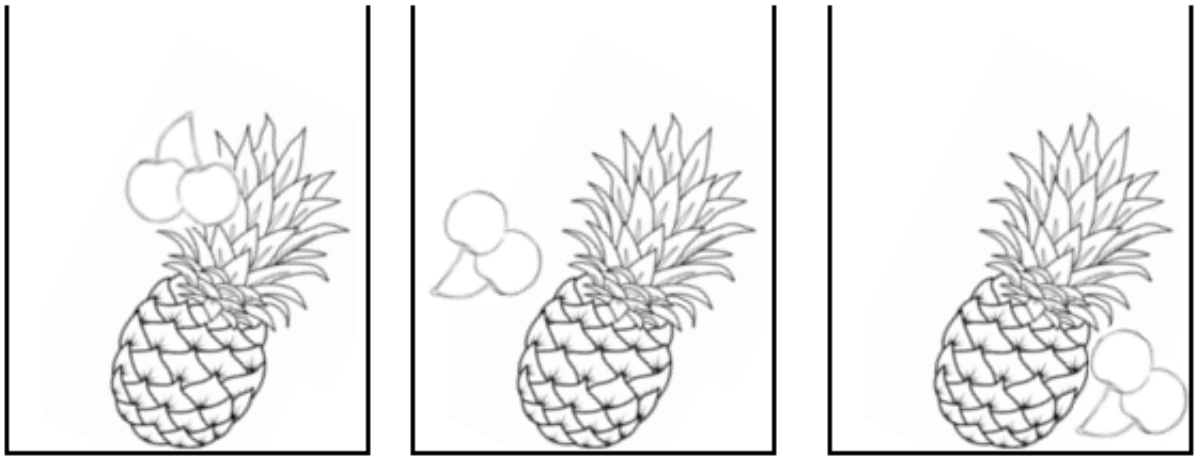


FIGURE 23 – Lors d'une collision, la pièce est repoussée à gauche ou à droite

De plus, si le solveur n'arrive pas à stabiliser la pièce, c'est-à-dire si elle est toujours en collision avec une autre après de multiples essais de corrections, celle-ci est envoyée vers une seconde plaque pour être packée. Cela se produit naturellement lorsque la plaque est déjà saturée et "déborde".

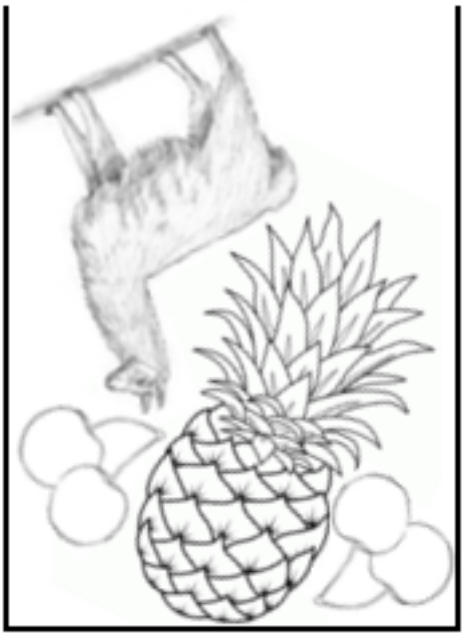


FIGURE 24 – Sortie de l'algorithme `FreezeSolver`

ProbaSolver Notre deuxième algorithme "non déterministe" est ce que nous avons appelé le `ProbaSolver`. Il s'agit cette fois-ci d'un solveur entièrement basé sur l'aléatoire. Le principe de

base est le suivant : en partant d'une configuration sans intersection des pièces, on effectue pour chaque pièce une translation et une rotation aléatoire. L'amplitude de ces mouvements dépend d'une distribution exponentielle, de la forme $\lambda e^{-\lambda t}$, avec λ passé en paramètre du solveur. Cette distribution permet d'obtenir la plupart du temps des mouvements de faible amplitude (d'autant plus que λ est grand), sans pour autant exclure des mouvements importants permettant d'explorer plus de configurations.

Pour chaque pièce, on vérifie une fois ce mouvement effectué qu'elle ne touche aucune autre pièce et ne sort pas de sa plaque, autrement le mouvement est inversé. Seule exception : si la pièce sort de la plaque par le haut, on lui offre une opportunité de gravir les échelons (monter d'une plaque).

On effectue ces pas aléatoires pendant un nombre fixé d'itérations, après quoi on s'arrête et on évalue la qualité de la solution. On sauvegarde cette solution et sa qualité (sous la forme d'un tableau de matrices de transformations), et on recommence le procédé entièrement. À la fin de celui-ci, on regarde si la solution obtenue est meilleure que la solution sauvegardée, auquel cas remplace la sauvegarde (sinon, on restaure l'ancienne solution). On répète encore ce procédé d'amélioration un certain nombre de fois, jusqu'à arriver à une solution normalement satisfaisante, avec de la chance.

On remarque que cet algorithme, bien qu'hautement paramétrable, n'a tendance à être efficace que pour un point de départ favorable. Dans ce cas cet algorithme a tendance à réduire les interstices entre les pièces, ce qui fait qu'il est complémentaire aux algorithmes déterministes précédemment détaillés.

5 Optimisation

5.1 QuadTree

Nous avons implémenté une autre manière de considérer les formes lors du packing en utilisant une structure de donnée particulière : les `QuadTrees`.

5.1.1 Principe général

Prenons la bounding box d'un `MultiPolygon`. On le représente par une matrice de 0 et de 1 (bitmap). L'algorithme de création du QuadTree consiste en subdivisions successives de la bounding box en zones. On délimite la bounding box de départ en quatre parties de tailles égales, puis on délimite ces zones en quatre parties récursivement, jusqu'à arrivée à un stade terminal.

A chaque itération, on regarde si une partie du `MultiPolygon` est contenue dans la zone en cours. Si non, on considère la zone comme "blanche" (0) et la récursivité se termine sur cette

zone. Si le `MultiPolygon` recouvre intégralement la zone en cours, on considère cette dernière comme "noire" (1) et la récursivité se termine également. Le stade terminal peut également être une zone de taille atomique (pixel), une zone de précision voulue.

Si le `MultiPolygon` est en partie dans la zone en cours, il s'agit d'une zone dite "grise", c'est-à-dire que la précision de la zone n'est pas suffisante pour décrire suffisamment le `MultiPolygon` sous forme de QuadTree. Dans ce cas-ci, la subdivision récursive a lieu. Quelques itérations sont représentées en figure 25.

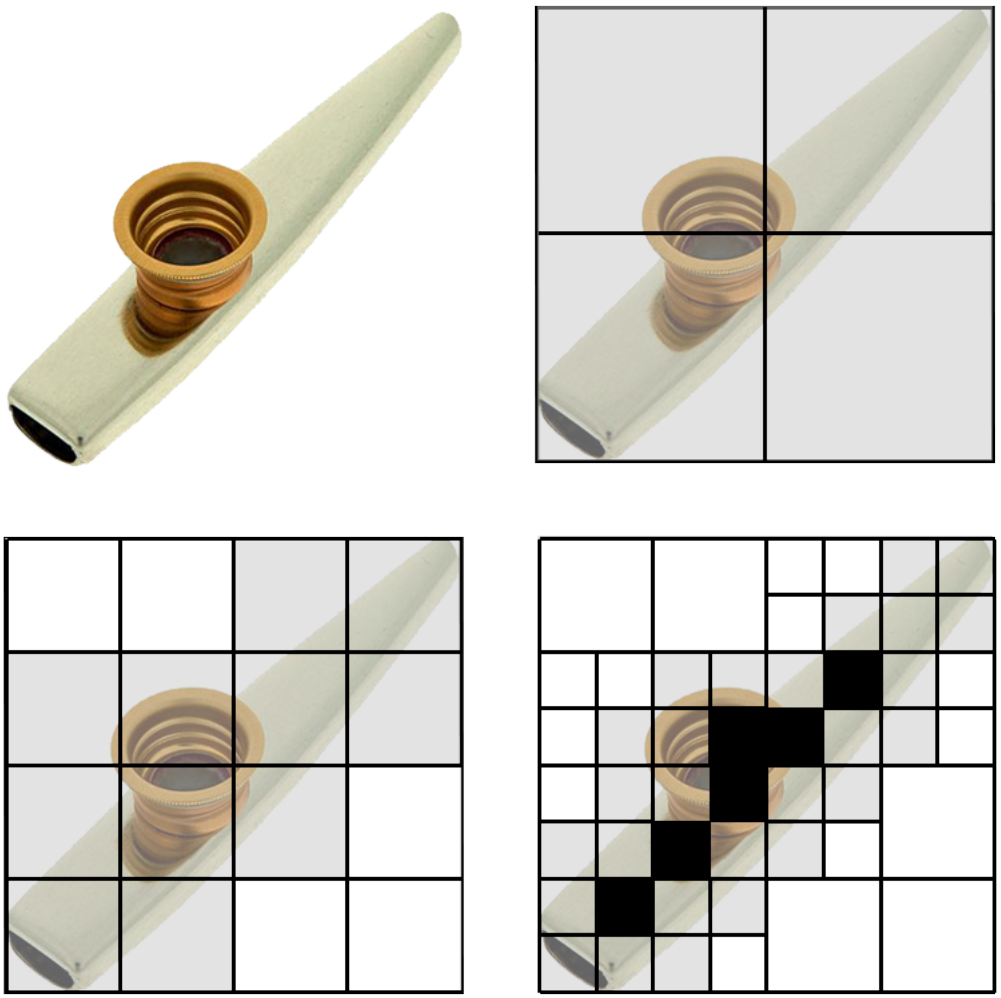


FIGURE 25 – Principe du QuadTree, itérations 0 à 3.

5.1.2 Représentation

Un QuadTree représenté par une classe de même nom contenant plusieurs InnerQuadTree avec plusieurs informations sur ces derniers pour les utiliser par la suite. Plus clairement, on a choisi de précalculer pour un même `MultiPolygon` plusieurs QuadTrees pour un ensemble de rotations possibles (30 degrés par défaut, donc 12 QuadTrees précalculés). La classe Quadtree les garde en mémoire, ainsi que les offsets nécessaires (pour gérer la différence de point d'origine

entre un `MultiPolygon` (au centre) et un QuadTree(en haut à gauche)), les centres de gravité de chaque rotation (centroids) et d'autres informations (précision de l'approximation etc).

Les InnerQuadTrees s'occupent de la récursivité en elle-même. La classe contient la couleur de la zone, sa profondeur par rapport à la racine, sa bounding box, sa taille et les 4 InnerQuadTree éventuels (sous-espaces) à la profondeur suivante (voir figure 26).

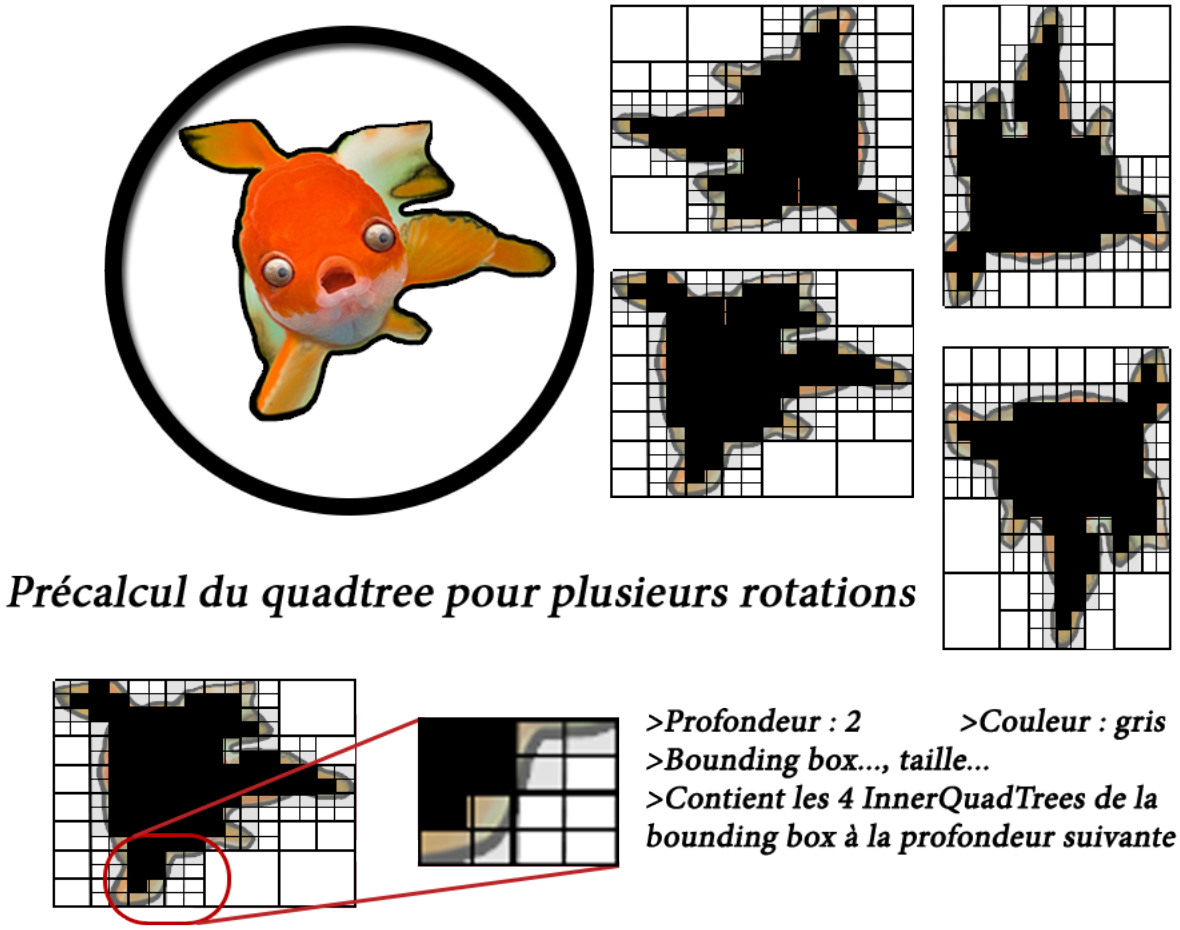


FIGURE 26 – Les InnerQuadTrees et leurs informations

Pour tester la couleur de son espace et créer les sous-espaces suivants, un InnerQuadTree possède des méthodes appelant celles des bitmaps, représentés également par une classe. Un `bitmap` est une classe représentant un tableau de booléens qui est le même pour toutes les classes (utilisation d'un pointeur), avec les informations nécessaires pour situer la sous-zone concernée dans l'espace (offsets, taille...). La classe `bitmap` possède les méthodes primaires d'intersection, de translation, de rotation, de création etc.

5.1.3 Intérêt

Cette représentation spatiale particulière des QuadTrees permet d'avoir une meilleure complexité sur les algorithmes qu'on peut leur appliquer. En effet les algorithmes `area`, `envelope`,

`centroid`, `translate` et `rotate` s'effectuent en temps constant sur les QuadTrees alors qu'ils s'effectuent en temps au moins linéaire par rapport au nombre de points sur les `MultiPolygon`.

On peut voir sur la figure 27 une comparaison de vitesse entre la représentation sous forme de QuadTree et celle sous forme de `MultiPolygon`. On peut remarquer des améliorations considérables pour la plupart des opérations sur les `Shapes` (on améliore d'un facteur 40 à 2000), excepté pour l'union qui n'est absolument pas optimisée pour les `QuadTrees` (elle nécessite la génération de nouveaux `QuadTrees`). Leur utilisation est donc fortement bénéfique, tant qu'on n'applique pas de `Merge`.

Algorithme	Nombre d'exécutions	Temps QuadTree (μs)	Temps `MultiPolygon` (μs)	Ratio d'amélioration
Intersection (position par default)	121000 1	43043 0.36	2003721 16	46
Intersection (origine)	121000 1	108411 0.9	7544721 16	69
Rotation	1100000 1	35328 0.03	9181464 8	259
Translation	1100000 1	35328 0.03	9181464 8	259
Centre de gravité	1100000 1	5560 0.005	345058 0.313	1769
Bounding box	1100000 1	6057 0.05	299819 0.727	1616
Calcul d'aire	1100000 1	4067 0.003	142004 0.129	2368
Union	10 1	1489924 150000	9 0.9	10^{-6}

FIGURE 27 – Comparaison de vitesse entre QuadTrees et `MultiPolygon`

5.1.4 Intégration dans le code

Notre architecture était au départ assez peu flexible, dans la mesure où `Shape` était une classe concrète (et l'est toujours) sans interface particulière, avec un grand nombre d'accesseurs et de modifieurs sur ses attributs. Lors de l'intégration des `QuadTrees` il a donc fallu réécrire le code des algorithmes en utilisant une interface nouvellement créée sur `Shape`, pour pouvoir faire hériter `QuadTree` de cette dernière et que les deux soient interchangeables dans le code.

Il a également fallu faire attention à ne plus directement utiliser dans le code des instances de `Shape`, mais plutôt des références pour permettre le polymorphisme. Cela passait notamment par l'interdiction de la copie des `Shapes` dans ces algorithmes (lors de la copie, l'utilisateur ne

sait pas s'il doit instancier une `Shape` ou un `QuadTree`.

Nous avons donc créé pour l'occasion la classe `Layout`, qui contient deux tableaux : l'un de `Shapes` et l'autre de `QuadTrees` (initialement vides). À l'instanciation de cette classe, on choisit lequel des deux sera utilisé et on doit lui passer un tableau de `Shapes` généré par le parseur. `Layout` va alors le garder tel quel ou bien générer des `QuadTrees` à partir de celui-ci.

Ensuite, pour que notre classe soit indexable de manière transparente, nous avons l'avons fait hériter du type `std::vector<Shape&>` (en réalité, le tableau contient des `reference_wrapper<Shape>`, car un tableau ne peut contenir des références). Nous remplissons après la génération du `Layout` ce tableau de références vers le tableau de `Shapes` ou celui de `QuadTrees` de la classe, selon l'utilisation choisie.

Ainsi lorsque l'utilisateur indexe le `Layout`, il reçoit une des références de ce tableau, laquelle pointe vers la bonne instance. De plus cela permet de rendre notre classe entièrement compatible avec les algorithmes de la librairie standard (`random_shuffle`, `sort`, ...) puisqu'elle peut être considérée comme un `vector`. De telles opérations sont alors optimisées puisque lorsqu'on trie un `Layout` par exemple, on trie en réalité le tableau de référence, et pas les tableaux d'instances (ce qui est moins coûteux en terme de copie).

Une méthode peut-être plus astucieuse aurait pu être mise en place si ce problème avait été prévu en amont, mais cette classe permet finalement la bonne interchangeabilité des deux représentations, sans toutefois rendre très facile l'ajout d'une nouvelle représentation.

5.2 Parallélisme

Une autre possibilité pour améliorer l'efficacité générale du programme est d'exploiter les architectures multi-coeurs qui sont disponibles sur les machines récentes. Nous avons utilisé cette notion de parallélisme de deux façons.

Parallélisation d'un algorithme

Au cours du projet, nous avons développé plusieurs solutions. Certaines d'entre elles effectuant un grand nombre d'opérations indépendantes (notamment le `Simple Transformer`), il est possible de répartir le calcul de ces opérations sur plusieurs processeurs afin d'accélérer l'exécution du programme.

Parallélisation de plusieurs algorithmes

Certains algorithmes ayant un comportement non déterministe, plusieurs exécutions d'un même algorithme vont conduire à des résultats différents. Nous avons donc implémenté un `Solver` permettant d'effectuer plusieurs calculs de solutions non déterministes et de conserver seulement le meilleur résultat.

5.3 Interpolation améliorée

Dans le but d'améliorer la vitesse d'exécution des algorithmes nous avons mis en place une méthode d'interpolation visant à réduire au maximum le nombre de points, tout en minimisant l'erreur induite.

Ceci permet d'effectuer plus rapidement les calculs internes aux algorithmes fonctionnant directement sur les modèles basés sur les points (`Shape`) en plus de permettre un calcul plus rapide des `QuadTrees`.

Nous utilisons pour cela différentes propriétés des courbes de Bézier. Notamment le fait qu'une courbe de Bézier définie sur un intervalle restreint reste une courbe de Bézier, il est donc possible de découper récursivement les courbes et ce sans perte de précision.

Il a ensuite été nécessaire de déterminer une condition d'arrêt pour cet algorithme. Il nous a semblé naturel d'approximer la courbe par une droite lorsque celle-ci est suffisamment "plate". Nous avons donc choisi d'approximer l'erreur générée par cette approximation par le maximum des distances des points de la courbe à la droite.

Ceci permet en effet d'avoir une approximation relativement précise de l'écart minimal qu'il est possible de se permettre en représentation interne afin d'éviter les intersections en sortie. Nous n'avons cependant pas été en mesure de calculer un majorant exact de l'erreur due à cette approximation.

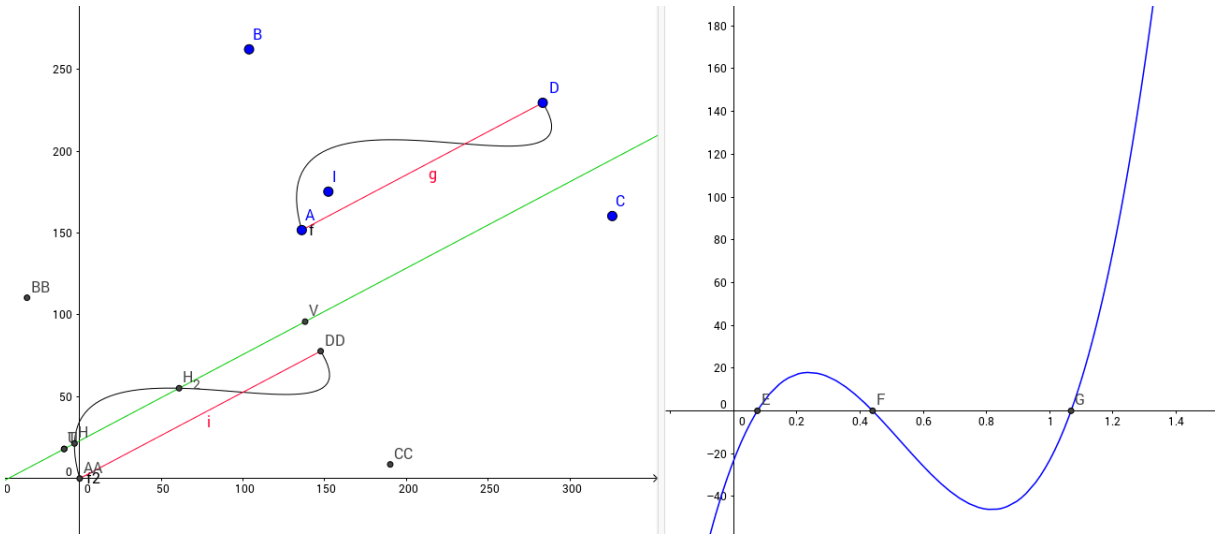


FIGURE 28 – Intersection entre une courbe de Bézier et la droite verte

Pour effectuer le calcul de cette distance maximale, nous calculons quatre points définissant les deux droites parallèles à celle définie par les deux points de contrôle de la courbe, qui se trouvent à une distance d de celle-ci.

Nous utilisons ensuite la représentation paramétrique de ces droites, et de la courbe pour calculer s'il y a ou non intersection avec la courbe en résolvant une équation polynomiale du troisième degré sur l'intervalle $[0, 1]$. Puis nous déterminons par dichotomie la distance d maximale pour laquelle il y a intersection avec la courbe.

On peut voir sur la figure 28 un exemple de droite s'intersectant avec une courbe de Bézier, et le polynôme associé.

Afin de compenser l'erreur ainsi produite, nous appliquons ensuite un *buffer* autour de la forme définie par ces points, qui permet d'assurer que les bords de la nouvelle forme se trouvent à une distance égale à au moins deux fois l'erreur autorisée par l'approximation.

La marge d'erreur est réglable à la compilation, et une plus grande marge implique un *buffer* plus important. On peut constater en figure 29 un exemple d'approximation très exagéré, avec le *buffer* ajouté.

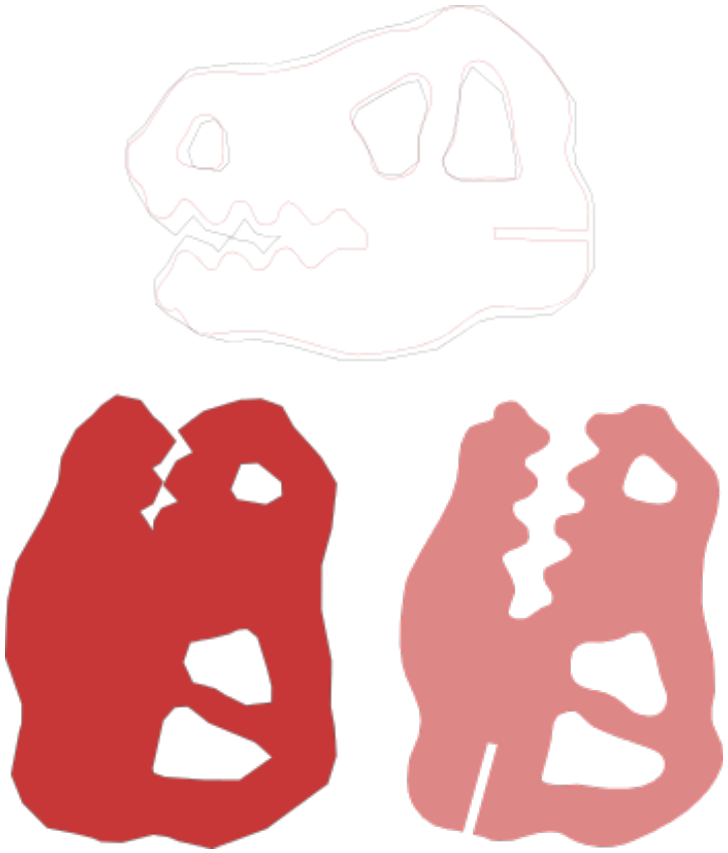


FIGURE 29 – Approximation (exagérée) d'une forme

Conclusion

Le client a maintenant à sa disposition un ensemble d'outils pour le packing d'images vectorielles, utilisable sous Inkscape. Il peut facilement changer l'ordonnancement des solveurs grâce à un langage simple d'utilisation, changer la manière dont les objets vont être représentés lors des calculs, et ajouter de nouveaux algorithmes grâce à l'approche en programmation orientée objet.

On peut imaginer de nombreux algorithmes pouvant s'ajouter à ceux implémentés. Par exemple, à la place d'utiliser des bounding boxes rectangulaires (Scanline), on pourrait affiner le packing avec d'autres formes comme des octogones, ou encore faire un solveur "physique", c'est-à-dire gérant les pièces comme des objets possédant un poids, dans une simulation de pesanteur, en les "remuant" d'une manière efficace pour parvenir à une solution assez stable. Encore une fois, c'est un problème NP-Complet, toute piste est intéressante dans une certaine mesure.

Dans le cadre de l'utilisation de la découpeuse laser, on peut également envisager une partie d'analyse en amont s'occupant de définir les zones de packing relativement à la plaque réelle, pouvant contenir des trous. Plus précisément, il s'agirait de construire par traitement d'image la plaque dans laquelle il faut packer les formes, selon présence de découpes déjà effectuées dedans ou non (utilisation éventuelle de la webcam présente au fablab). Cela permettrait de récupérer au maximum les chutes de matériaux, et de mettre à profit nos solveurs sur plusieurs utilisations consécutives.

Une extension du projet encore plus poussée serait de chercher un niveau de plus d'automatisation du processus. Afin de faciliter la gestion des matériaux par le client, on pourrait envisager la création d'une base de données comportant les plaques vierges et celles déjà découpées, dans le but de faire une recherche de meilleure plaque pour le packing d'un ensemble de pièces données, tout en cherchant à réutiliser les chutes en priorité.

Ce projet complexe nous a principalement permis de développer notre capacité à réaliser une architecture logicielle modulaire. Nous nous sommes également formés à nous servir de fonctionnalités avancées du gestionnaire de versions Git, à étendre notre maîtrise du C++, ou encore à intégrer des outils extérieurs (Boost, SVG++, ...). Enfin nous avons acquis une meilleure connaissance du parallélisme et des Quad-trees comme outils d'optimisation.

Outre ces apprentissages techniques, cette première expérience travail de groupe en situation professionnelle a été riche d'enseignements organisationnels. Nous avons pu constater à quel point il est difficile de centraliser l'information en permanence, et de répartir équitablement le travail entre les membres. Ainsi, nous avons compris que la communication au sein de l'équipe doit être une priorité pour que tout le monde puisse avancer au même rythme, favorisant ainsi l'épanouissement de chacun et l'avancement du projet.

Enfin, nous tenons à remercier chaleureusement M. Allali, notre responsable pédagogique, et M. Renault, notre client, pour leurs conseils tout au long de ce projet, leur patience et pédagogie, nous permettant de mieux comprendre les différents aspects du développement de projet informatique en équipe.