Origram

Rapport de projet

20 janvier 2014

Projet intégré octobre 2013 - janvier 2014

Encadrant : Petru Valicov

Grégoire Beaudoire
Baptiste Jonglez
Fabrice Mouhartem
Pierre Pradic

Armaël Guéneau
Jérémy Ledent
Antoine Pouille
Damien Rouhling

Table des matières

Introduction

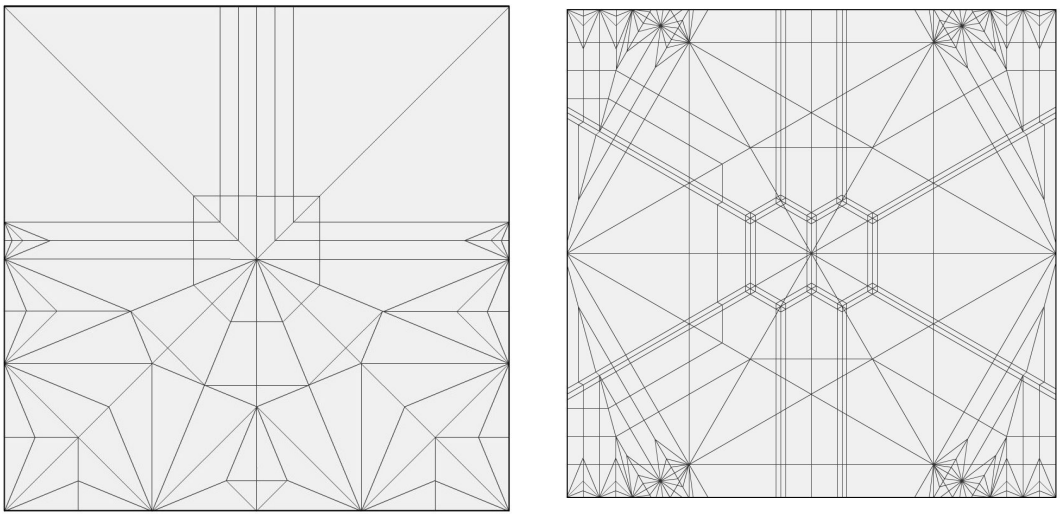
L'art de l'origami est très ancien, et pourtant ce domaine a peu fait usage des outils modernes offerts par les avancées de l'informatique. Il consiste à prendre une feuille de papier (en général carrée) et à la plier sans la déchirer ni la coller afin de construire une forme, plane ou en volume.

Il a toujours été difficile de « transmettre » l'information en origami. En effet, même s'il y a bien sûr la méthode directe (d'enseignant à élève), transmettre une méthode de pliage à grande échelle reste un exercice délicat. Comment communiquer le cheminement de plis nécessaire pour arriver au résultat final ?

L'une des méthodes les plus faciles à lire et à comprendre pour un plieur est la découpe en *diagramme* : le pliage est découpé en étapes, à l'image d'une notice de montage de meuble, et chaque étape est agrémentée d'un dessin en trois dimensions. Cela permet au plieur de visualiser le cheminement à suivre. Il existe d'autres méthodes, mais beaucoup moins faciles à interpréter, et donc moins efficaces pour partager les connaissances d'origami.

A l'heure actuelle, quand un créateur veut partager son travail — sa création — et l'enseigner ainsi à tous les potentiels plieurs dans le monde, il ne dispose pas de moyens automatisés pour l'aider. Ainsi, il peut, soit :

— ne pas partager la construction, ce qui est clairement le plus rapide mais aussi le plus regrettable d'un point de vue de la connaissance globale ;
— partager en utilisant des moyens de communication peu lisibles — par exemple via un *canevas de plis* (*crease pattern*, en anglais), c'est-à-dire en dépliant complètement l'origami et exposant les plis, sans indiquer aucunement la méthode ou l'ordre suivi pour les faire :



(a) Un papillon (b) Une tortue

FIGURE 1 – Deux canevas de plis

— réaliser un diagramme *manuellement* : avec du papier et un crayon, ou bien un logiciel de dessin vectoriel comme *Inkscape*. Cela prend un temps non négligeable : à titre d'exemple, pour un modèle relativement simple d'une trentaine d'étapes, une semaine de travail est nécessaire.

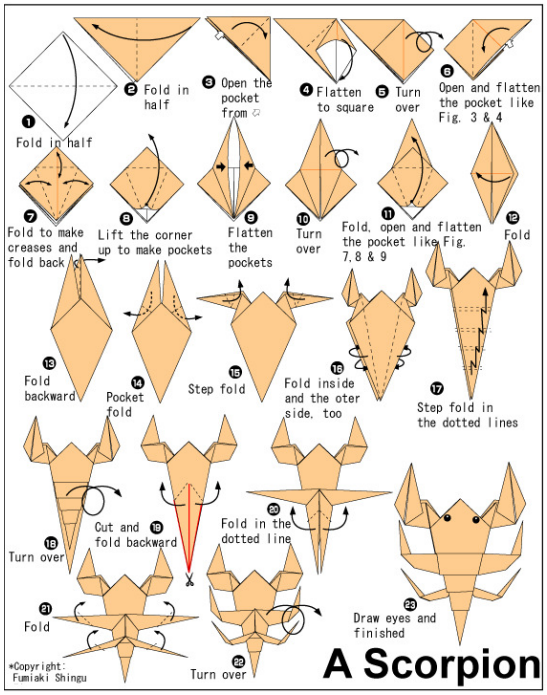


FIGURE 2 – Un diagramme de scorpion

Il y a ainsi beaucoup d'origamis dont la construction est inconnue sauf pour le créateur du pliage. Un nombre plus important encore est de portée limitée car ils nécessitent, pour arriver à reproduire le pli, des connaissances techniques très poussées. Cela met un frein à l'expansion de cet art, d'une part à cause du manque de données informatisées, et d'autre part en raison du manque de pliages couverts par les diagrammes.

Notre objectif est de répondre à ce manque et de parvenir à fournir aux créateurs et aux plieurs un outil pour leur permettre de communiquer et d'échanger des connaissances plus facilement. En termes plus précis, nous souhaitons créer un logiciel — Origram — permettant de simplifier le processus de construction de diagrammes, en fournissant une interface intuitive et facile à utiliser, et en automatisant la création du diagramme. Rendre la construction de diagrammes plus simple et plus rapide permettra de promouvoir ce mode de communication, ce qui bénéficierait à la fois aux créateurs et aux origamistes débutants.

Nous revenons tout d'abord sur les logiciels existants et les références bibliographiques (section 1). Nous énonçons ensuite les spécifications de notre programme (section 2), les objectifs tirés de notre étude de l'état de l'art. Nous précisons alors l'architecture de notre programme (section 3), ainsi que notre organisation (section 4.1), puis faisons un point sur ce qui a été réalisé (section 5).

1 État de l'art

Deux catégories peuvent être distinguées dans l'état de l'art actuel de l'informatique de l'origami : les logiciels aidant à la conception de diagrammes, et les logiciels orientés vers la création de l'origami. Nous reprenons ici les caractéristiques de ceux que nous avons trouvés afin d'argumenter de l'intérêt de notre logiciel et de dégager ce qui pourrait être réutilisé dans le futur.

Il existe de plus des résultats théoriques. Nous les présentons brièvement à l'exception d'un article qui a joué un rôle significatif dans l'élaboration de notre projet.

1.1 Dans la conception de diagramme

Il y a actuellement différentes manières d'aborder ce problème : *Doodle* [9] est un langage de description générant un diagramme ; *Origami simulator and diagrammer* [12] suit une approche identique à la nôtre mais peu ergonomique et dont résulte un diagramme peu lisible. Enfin, on a avec Foldinator [20] une tentative d'application web — avec l'idée de toucher un large public — inactive aujourd'hui.

Doodle *Doodle* est un langage destiné à décrire un diagramme, et permet de produire des diagrammes esthétiques et lisibles. Il se fonde sur des règles appliquées séquentiellement pour décrire l'avancement du diagramme.

Cependant, ces règles sont majoritairement des primitives de dessin, permettant d'indiquer sur quelle ligne marquer un pli (comme les règles sont incrémentales, ces lignes restent dessinées, ce qui permet au concepteur du diagramme de ne pas avoir à les redessiner à chaque étape). Toutefois, dès que l'on décide de faire un pli, on est obligé d'indiquer manuellement comment retracer le nouvel état de l'origami, ce qui ne rend pas forcément la tâche très aisée et accessible. Pour avoir un ordre d'idée, un diagramme de 9 étapes se fait en 200 instructions [1]. La tâche reste donc fastidieuse.

Notre projet reprend l'idée d'un langage de description, dans la partie *représentation intermédiaire* (voir 5.4). Cependant, notre langage a pour but de générer non-seulement un diagramme, mais aussi la structure 3D correspondant au pliage. Il s'agit donc d'un langage décrivant une séquence de plis, et non un dessin. L'avantage par rapport à *Doodle* est que l'on peut grâce à cela ajouter une interface graphique générant le code, simplifiant ainsi l'utilisation du programme pour les origamistes n'ayant pas de connaissances en informatique.

Origami simulator and diagrammer Ce programme a été conçu dans le cadre d'un mémoire [13] pour savoir si un diagramme pouvait être généré informatiquement ou non. Il a donc été initialement créé pour fournir des résultats, plutôt que dans le but de s'adresser aux concepteurs d'origami.

On retrouve des fonctionnalités similaires à notre projet :

- le pliage se fait via une interface graphique ;
- fenêtre 3D dans laquelle il est possible de faire tourner le pliage ;
- production d'un diagramme en PostScript.

Cependant, de nombreux défauts rendent le programme inutilisable en pratique. L'interface est mal pensée et peu intuitive, en plus d'être graphiquement très pauvre. Les diagrammes obtenus sont souvent incomplets, et ne sont pas suffisamment clairs pour être utilisables par un origamiste. Enfin, le programme utilise des bibliothèques spécifiques à une plate-forme (`win32` et `DirectX`), les sources ne sont pas disponibles, et il n'est plus maintenu depuis 2005.

Nous disposions du manuscrit du mémoire, mais il nous a peu servi.

Foldinator Il s'agit d'un projet prometteur en flash développé en 2009 : un simulateur-diagrammeur, mais dont la version actuelle ne fonctionne plus. Il est possible qu'une mise à jour du plugin flash ait nui à la rétro-compatibilité, le programme ne réagissant pas aux entrées.

En revanche, d'après les démonstrations données sur le site du projet, il semblerait que les diagrammes obtenus, ainsi que l'interface utilisateur soient clairs et esthétiques. Le programme étant disponible sous la forme d'une application web, il serait également facile d'accès.

Nous pouvons néanmoins signaler que dans le cadre d'un diagramme compliqué ou non traditionnel (les pliages faisant intervenir par exemple des grilles, ne serait-ce que dans une moindre mesure), le programme montre ses limites, d'une part par les risques inhérents aux programmes web (saut de connexion, fermeture du navigateur pendant un pliage qui nécessite un minimum de temps), ou par le manque de

1. `http://doodle.sourceforge.net/resources.html` : pajarita.

précision au niveau du pliage (pour revenir aux cas des grilles parallèles, c'est le manque de repères qui fait défaut). Il est d'ailleurs difficile de trouver des traces d'utilisation de cet outil, et nous ne pouvons pas savoir comment il se comporterait face à des modèles complexes. Par ailleurs, nous ne disposons pas des sources du projet, ni de quelconques spécifications.

Comme *Doodle*, *Foldinator* fonctionne à l'aide d'un langage de description de diagramme, *origami XML* [21]. Celui-ci est d'un peu plus haut niveau que le langage *Doodle*, mais il s'agit toujours d'un langage de dessin 2D que nous ne pouvons pas réutiliser.

1.2 Dans l'aide à la création

Il existe actuellement plusieurs outils destinés à aider le créateur d'origami, principalement développés par Robert Lang : *ReferenceFinder* [15], *TreeMaker* [15], *Oripa* [19], *BPMaker* [10] ainsi que *Eos Project* [1].

ReferenceFinder *ReferenceFinder* est un programme destiné à générer une séquence de plis construisant un point de repère.

Il permet de définir un point de repère par ses coordonnées sur la feuille. Ces points sont utiles en origami dit « traditionnel », où les points peuvent difficilement être déplacés sur le canevas de plis puisque le déplacement d'un point entraine le déplacement des autres points du modèle.

La méthode utilisée par *ReferenceFinder* se base sur des coordonnées flottantes, ce qui donne en fait une approximation du point demandé. L'avantage par rapport à une méthode exacte est qu'on obtient ainsi une séquence de plis relativement courte, qui permet d'éviter les plis de construction superflus.

ReferenceFinder donnant de très bons résultats, nous pourrions nous en servir dans le futur afin de fournir un outil permettant de spécifier un point de repère via ses coordonnées, et de générer automatiquement toutes les étapes de diagramme correspondant à sa construction.

TreeMaker Il s'agit d'un programme développé par Robert Lang dont le principe est de partir d'un arbre, c'est-à-dire une représentation d'une base expliquant la taille des volets indépendants de la feuille ainsi que leur disposition, pour construire le canevas de plis de la base associée.

Le fonctionnement de ce programme utilise la méthode du remplissage de cercles, qui est une méthode géométrique permettant de déterminer un remplissage maximal de la feuille par des polygones, on utilise ensuite des « molécules »[18, 8] pour les remplir.

Il aurait été intéressant de récupérer l'algorithme de construction de la base pour fournir un outil permettant de partir d'une base spécifiée par un arbre, en plus des bases classiques. Cependant, il s'agit d'une fonctionnalité très avancée que nous ne comptons pas implémenter dans un futur proche

Oripa *Oripa* est un éditeur de canevas de plis calculant pour les origamis les plus simples un pliage en deux dimension — s'il existe — .

Il est utilisé par la communauté lorsqu'il s'agit de construire des canevas de plis à distribuer puisqu'il propose une interface facile d'accès et relativement complète, permettant de vérifier qu'il n'y a pas d'erreurs en simulant le pliage final.

Nous ne réutiliserons pas *Oripa* puisque notre programme a pour but la création de diagrammes et non de canevas de plis. Le fonctionnement de son plieur 2D nous a tout de même inspirés, en première approche, pour la gestion des couches de notre origami simulé.

Box pleating maker Il s'agit d'un projet neuf (septembre 2013) mais avançant assez vite permettant de travailler sur des modèles en box-pleating, c'est à dire les modèles fondant leur conception sur une grille à partir de laquelle des séquences simples existent afin de faire sortir des parties indépendantes.

Son but est de construire un canevas de plis de box-pleating en se fondant sur la technique du remplissage par carrés (équivalent en box-pleating du remplissage de cercle, mais plus simple à mettre en œuvre).

Nous pouvons remarquer qu'il s'agit d'un projet web, en effet un modèle en *box-pleating* comme nous l'avons signalé en introduction est facile à concevoir, on peut donc se permettre de le faire au travers d'une connexion internet pour générer un canevas de plis à travailler ensuite.

Il aurait été intéressant d'ajouter à *Origram* un mode pour le pliage en box-pleating, afin d'avoir un logiciel vraiment complet. Mais il s'agirait là d'un futur assez lointain.

Eos Project *Eos* est l'outil le plus proche d'Origram en terme d'objectifs et de fonctionnalités. Implémenté en Mathematica, il permet de construire un origami grâce à un langage de manipulation de plis. Il fournit également une visualisation 3D de la construction, étape par étape. Cependant, Eos et Origram diffèrent sur les points suivants :

— Eos est un projet de recherche, et l'ergonomie n'est donc pas la priorité (par exemple, pas de manipulation interactive du modèle 3D pour réaliser des plis)
— Eos est destiné à l'aide à la construction de preuves géométriques grâce à l'origami. En particulier, il ne génère pas de diagrammes.

Le code source d'Eos n'est pas publiquement disponible. Il existe également une version web, décrite par plusieurs publications [1], mais qui ne semble plus fonctionner.

Notons que cet outil a été créé par Ida et Takahashi, les auteurs de l'article *Origami fold as algebraic graph rewriting* [11], dont nous nous sommes beaucoup servis pour Origram. Cette publication décrit les aspects théoriques et les algorithmes que les auteurs ont construit pour formaliser le travail réalisé avec Eos.

1.3 Dans la théorie

Nous pouvons distinguer deux types de résultats en informatique de l'origami : les résultats sur la constructibilité, et les résultats de conception.

Les résultats sur la constructibilité ont un intérêt principalement théorique, et n'ont pas vraiment d'application dans le cadre de notre projet. On peut citer par exemple la quintisection d'un angle, proposée par Robert Lang [14]. Si ce résultat est mathématiquement intéressant, on ne s'en servira pas dans notre logiciel puisque c'est une construction peu utile en origami.

Il existe également des résultats permettant de savoir si un canevas de pli est pliable ou non, ainsi que des résultats de complexité algorithmique (en particulier, déterminer si un pli est localement pliable se vérifie en temps linéaire par le théorème de Kawasaki [8], en revanche déterminer si tout un canevas de pli peut produire un origami plat est un problème NP-difficile [8]). Nous n'utiliseront pas ces résultats non plus, car ceux-ci s'appliquent aux canevas de plis qui ne sont pas le but d'*Origram*.

Les résultats de conception en revanche nous intéressent grandement plus, notamment l'axiomatisation de l'origami proposée par Huzita et Justin [16]. Il s'agit d'un ensemble d'axiomes permettant de décrire tous les plis qu'il est possible de faire en origami plan. C'est sur cet ensemble d'axiomes que se base notre langage de représentation intermédiaire (voir 5.4).

Toutefois, ces axiomes peuvent se résumer à effectuer un pli selon une ligne bien choisie. La difficulté réside donc dans la modélisation de ce type de pli. Nous avons beaucoup réfléchi à ce sujet et souvent de façon infructueuse : à chaque fois que nous arrivions à dégager un algorithme, nous trouvions un cas particulier auquel nous n'avions pas pensé qui rendait la tâche encore plus fastidieuse. Les travaux de

T. Ida et de H. Takahashi [11], que nous avons découverts assez tard au cours du projet, décrivent une méthode pour la simulation de ces plis, qui est particulièrement adaptée à notre approche :

— Leur article se restreint, comme nous, aux origamis plans.
— L'idée générale décrite dans l'article utilise les mêmes idées que celles que nous avions dégagées au cours de nos réflexions : représenter l'origami par une liste de faces ainsi que des relations exprimant l'adjacence et la superposition entre les faces.

Cet article nous a donc fourni un algorithme précis pour effectuer les pliages. Cela nous a permis d'éviter la phase de conception de l'algorithme par recherches successives de cas pathologiques, qui nous aurait pris beaucoup trop de temps.

Dans l'article de Ida et Takahashi, un origami est décrit comme un ensemble de faces définies par les arêtes de leur frontière. Ces faces sont réliées par deux relations : l'adjacence et la superposition. Deux faces sont dites adjacentes si elles partagent une arête. Deux faces se superposent si dans le modèle plié l'une est au dessus de l'autre. Il y a bien sûr plusieurs manières de définir cet « au dessus ». Nous précisons ce point dans la section 5.3.

La modélisation du pli se fait alors par des mises à jours de l'ensemble des faces et des relations qui les lient. En premier lieu, il y a une phase de division de faces : on coupe les faces concernées au niveau de leur intersection avec la ligne de pli, ce qui définit de nouvelles faces. Une première mise à jour des relations peut donc avoir lieu. Ensuite, on « effectue » le pli par le biais d'une modification de la relation de superposition.

Cette dernière modification, assez compliquée, nécessite de déterminer quelles sont les faces qui « se déplacent », afin de savoir si elles seront « au dessus » ou « en dessous » des autres faces. Nous détaillons en section 5.3 l'implémentation de cet algorithme.

Conclusion

Nous avons donc pu observer qu'il existe en origami de nombreuses tentatives de fournir des outils informatisés pour aider cet art. Cela montre que la communauté est intéressée par cette voie pour permettre à l'art de se diffuser. Cependant, parmi ces programmes, très peu sont réellement utilisés par la communauté origamiste. En effet, à part *Reference Finder*, on trouve peu de traces d'utilisation de ces logiciels, et la grande majorité des diagrammes sont encore dessinés avec *Inkscape*.

La cause est qu'aucun logiciel à l'heure actuelle ne propose à la fois une interface utilisateur pratique, ainsi qu'un rendu lisible et soigné. Ce sont ces conditions que vise à remplir le projet *Origram*.

Certains résultats de théorie ([11]) nous ont aidés à trouver des structures adaptées à la modélisation des plis, ainsi qu'à implémenter des algorithmes de plis qui fonctionnent correctement.

2 Spécifications

2.1 Besoins fonctionnels

Cette partie spéifie les besoins fonctionnels d'*Origram*. Ils correspondent au but premier du logiciel. Ces besoins ont été établis en interrogeant les origamistes sur leurs attentes pour un tel logiciel, lors d'une convention d'origami à Lyon et de sondages sur des forums d'origami.

La fonction principale d'*Origram* est de permettre de diffuser un origami en concevant un diagramme :

— *Conception de diagramme* : concevoir un diagramme d'un modèle d'origami, décrivant étape par étape le processus permettant sa réalisation. Ce diagramme doit être clair et lisible, et il doit respecter les conventions de l'origami [17].

De plus, ce qui différencie *Origram* de logiciels de diagrammage tels que *Doodle*, c'est la présence d'une interface graphique et d'outils d'aide au diagramme permettant de simplifier son utilisation.

— *Interface graphique* : disposer d'une interface graphique permettant un affichage 3D du pliage courant, ainsi que sa manipulation à la souris.
— *Aide à la création* : disposer d'outils d'aide à la création, c'est à dire de permettre de simuler un pliage d'origami sans forcément vouloir en extraire un diagramme. Par exemple, la construction de points de référence avec *ReferenceFinder*, la division de longueurs / angles en puissances de 2, ...
— *Outils de plis classiques* : disposer de macros implémentant toutes les séquences de plis classiques qui sont souvent utilisées en origami. Cela comprend le fait de pouvoir débuter d'une base prédéfinie (voir B.3), et l'implémentation de tous les plis usuels (voir B.1).
— *Feuille de départ* : pouvoir choisir une feuille de départ de forme non carrée. En particulier, tout n-gone régulier doit pouvoir être choisi comme feuille de départ.
— *Langage de description plis* : fournir aux utilisateurs ayant des connaissances avancées en informatique un langage permettant de décrire une séquence de plis, si ceux-ci préfèrent écrire du code plutôt que d'utiliser l'interface graphique. Ce langage doit être totalement transparent pour les autres utilisateurs. (voir 5.4 : langage de représentation intermédiaire)
— *Fonctionnement pas-à-pas* : permettre à l'utilisateur de créer son diagramme étape par étape. En particulier, cela implique que le langage de représentation intermédiaire doit fonctionner de manière incrémentale.
— *Rendu personnalisable* : permettre à l'utilisateur d'éditer manuellement le diagramme généré automatiquement par le logiciel, s'il le souhaite. En particulier, il doit pouvoir spécifier l'angle de la caméra et le découpage en étapes du diagramme (voir 5.1 : le fonctionnement par « snapshot »).
— *Détection d'erreurs* : certifier la correction de l'origami simulé, en détectant lorsque l'utiliateur essaie d'effectuer un pli incorrect, et en l'avertissant de l'erreur.

2.2 Besoins non fonctionnels

Nous allons maintenant détailler les besoins non-fonctionnels de notre projet, c'est à dire les contraintes implicites qu'il doit satisfaire. Ceux-ci se divisent en deux catégories : les besoins relatifs à l'implémentation du logiciel, et ceux liés à son utilisation.

Implémentation Il s'agit de tous les besoins qui concernent la programmation et la compilation du logiciel.
— *Conventions de codage* : afin de pouvoir travailler en groupe sur ce projet, il est nécessaire de disposer d'un code propre et lisible par tous ceux qui participeront à la programmation. Pour cela, il est nécessaire de définir des conventions de codage claires et précises (cf. 4.3).
— *Modularité* : le code doit être découpé en plusieurs parties indépendantes, de façon à ce qu'un changement dans l'une d'elles n'affecte pas le fonctionnement des autres. En particulier, l'interface graphique doit être complètement séparée du cœur de l'algorithme, et ne servir qu'à appeler des fonctions internes, sans effectuer d'actions intelligentes.
— *Stabilité des technologies utilisées* : le bon fonctionnement du logiciel ne doit pas être trop sensible aux changements de versions des technologies utilisées. La liste des logiciels et versions nécessaires pour faire tourner le logiciel doit être clairement définie.
— *Compatibilité sur différents systèmes d'exploitation* : le logiciel doit pouvoir fonctionner sous tous les principaux systèmes d'exploitation existants (Windows, Linux, Mac OS X).
— *Non-duplication d'informations* : le code doit être factorisé de manière à éviter la duplication de code. De même, les algorithmes utilisés ne doivent pas conserver de copies des structures de données internes lorsque ce n'est pas nécessaire.
— *Tests* : toutes les fonctionnalités du logiciel doivent être testées séparément afin de s'assurer de leur bon fonctionnement.

Utilisation Nous allons maintenant détailler les besoins concernant l'ergonomie, la facilité et la praticité de l'utilisation du logiciel.
— *Performances* : le logiciel doit mettre en place des algorithmes efficaces, afin d'assurer une fluidité d'utilisation et la minimisation des temps de calcul.

— *Robustesse* : le logiciel doit pouvoir diagrammer des origamis de taille conséquente (une centaine d'étapes) sans erreur.
— *Stockage de données* : on doit pouvoir à chaque étape sauvegarder la séquence de plis/étapes de diagramme produite et la recharger. Cela permettra entre autres de définir des bases personnalisées.
— *Fiabilité* : des sauvegardes de l'origami en cours de création doivent être faites de manière automatique et régulière, afin de ne pas perdre toutes les données en cas de plantage du logiciel.
— *Interface intuitive* : l'interface graphique doit être simple et ergonomique. Un utilisateur peu formé à l'informatique doit pouvoir s'en servir. L'utilisation des outils de plis doit être intuitive et proche de la démarche habituelle des origamistes.
— *Interface pratique* : le rendu visuel de l'interface doit être agréable. Les différents boutons doivent être disposés de manière logique, pas trop dense, et disposer d'icônes évocatrices. Le logiciel doit disposer de raccourcis clavier logiques et configurables, ainsi que permettre la définition de macros personalisées et de les rendre accessible via un menu.
— *Représentation intermédiaire* : le langage de description de plis (représentation intermédiaire) doit permettre de générer à la fois le modèle 3D et la séquence de diagrammes 2D correspondante. Cela le différencie des langages déjà existants tels que *Doodle* et *Origami XML* qui ne permettent de décrire que des dessins 2D, et non une séquence abstraite de plis. Cela permet notamment d'utiliser le code de ce langage comme unique moyen de stockage des données.
— *Documentation* : le logiciel doit disposer d'une documentation claire et complète.

3 Architecture

Nous présentons ici l'architecture logicielle globale du projet, ainsi que les outils et bibliothèques que nous utilisons. Nous rappelons aussi les conventions de codage que nous nous sommes fixés.

3.1 Architecture globale

L'architecture globale est visible sur le diagramme de classes de la figure 3.

Nous avons une structure représentant un modèle 3D d'origami (Bare3DModel et classes dérivées). Cette structure est utilisée à la fois pour donner un retour à l'utilisateur (visualisation 3D du pliage en cours) et pour générer les images du diagramme. Nous avons souhaité rendre cette structure indépendante de la manière de l'afficher, d'où l'utilisation d'une classe abstraite AstractDrawable3DModel qu'on peut spécialiser selon la technologie utilisée.

Le plus gros des manipulations se trouve dans la classe OrigamiBackend :

— tables internes représentant l'état du modèle (relations d'adjacence et superposition entre faces, etc.) ;
— algorithme de pli, agissant sur ces structures internes ;
— mise à jour du modèle 3D à partir des actions de l'utilisateur ;
— génération du diagramme à partir du modèle 3D et de méta-données (grammage du papier, etc).

L'interaction entre la GUI (*Graphical User Interface*) et le *backend* se fait par le biais de la classe OrigamiInteraction, qui contient une instance de la classe OrigamiBackend. Celle-ci offre une interface permettant de réaliser des actions (plis), tout en gardant trace de celles-ci dans un script Lua (la « représentation intermédiaire »). Celui-ci décrit exactement la succession de plis nécessaires pour aboutir à l'étape courante, et sert par ailleurs de format de sauvegarde.

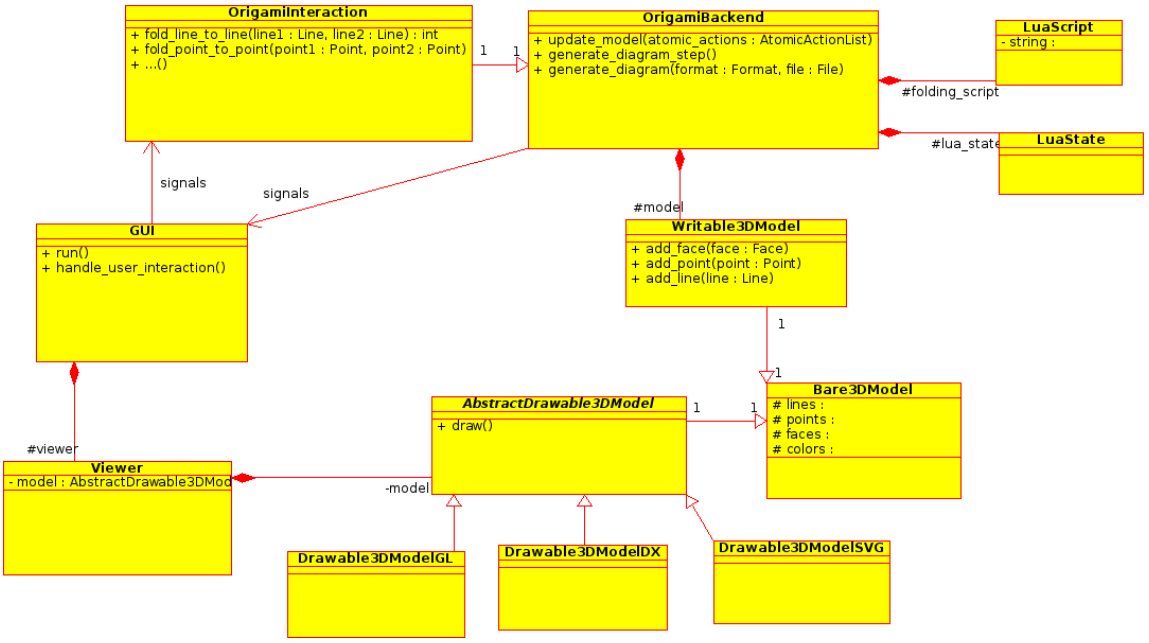


FIGURE 3 – Diagramme de classes du projet

3.1.1 Modularité

Un des buts secondaires était de produire du code le plus modulaire possible.

Ainsi, même si l'interface graphique actuelle est écrite en Qt et OpenGL, on peut imaginer des interfaces utilisant d'autres technologies (GTK+, DirectX), voire un mode en ligne de commande sans interface graphique — permettant uniquement de produire un diagramme à partir d'un fichier de pliage.

L'organisation actuelle du code distingue clairement les algorithmes et l'interface, et il est ainsi possible de changer l'interface sans modifier le coeur des algorithmes de manipulation d'Origami.

3.2 Choix techniques

Origram est développé en `C++11`, avec une interface graphique en `Qt4` et une visualisation 3D en `OpenGL`. L'export de diagramme est réalisé avec `Magick++`, tandis que les pliages sont sauvegardés sous forme de scripts `Lua`.

Nous travaillons tous avec des distributions GNU/Linux, mais le système de compilation [2] et les bibliothèques utilisées sont multi-plateformes : Linux, Windows, OS X, voire même d'autres BSD.

3.2.1 `C++11`

Nous avons décidé d'utiliser la dernière version de `C++`, à savoir `C++11`. Cela fournit des fonctionnalités pratiques, notamment :

Itération simplifiées Il est possible d'itérer facilement sur des structures compliquées, grâce à l'inférence de type. Par exemple, au lieu d'écrire

```
for (vector<Vertex*>::iterator v = t.begin(); v != t.end(); ++v) {
```

on écrira

2. Il s'agit de `CMake` : http://www.cmake.org/

```
for (auto v : t) {
```

Tuples La STL inclut un template pour manipuler des tuples, ce qui peut se révéler pratique.

Fonctions anonymes Il est possible d'utiliser des *fonctions anonymes*, ou fonctions lambda.

Compilateurs supportés Ce standard `C++` a maintenant plus de deux ans, sans compter les *drafts* plus anciens. Les compilateurs les plus courants (Clang et GCC) fournissent un support complet et stable de `C++11`.

Ceci étant dit, nous souhaitons garder la compatibilité avec des compilateurs un peu plus anciens. Nous nous sommes fixés comme objectif *GCC 4.6* et supérieur. GCC 4.6 implémente une bonne partie du standard `C++11` [3], et est une version assez répandue (Ubuntu 12.04).

Du côté de Clang, nous imposons au moins la version 3.1, qui introduit les fonctions lambda et les listes d'initialisation [4].

3.2.2 Qt4

Nous utilisons Qt4 pour l'interface graphique. Nous avons envisagé Qt5, qui diffère assez peu de Qt4 (contrairement à la transition de Qt3 à Qt4 il y a quelques années), et qui est globalement plus moderne.

Cependant, nous utilisons une bibliothèque pour l'affichage 3D, libQGLViewer, qui est uniquement empaquetée pour Qt4 dans les distributions Linux majeures (Debian, Fedora, Archlinux). Cela impose donc Qt4, au moins pour l'instant.

3.2.3 OpenGL et libQGLViewer

Le rendu 3D du pliage en cours est réalisé grâce à OpenGL, dont la sortie est directement intégrée à l'interface graphique Qt4.

Nous utilisons également une bibliothèque, libQGLViewer, offrant une interface de manipulation intuitive : il est possible de tourner le modèle, zoomer, effectuer une translation, afficher les axes, sélectionner des points...

3.2.4 Conteneurs STL

Puisque nous utilisons Qt, nous avons le choix entre les conteneurs de la STL (`std::vector`, `std::list`, etc) ou ceux de Qt (`QVector`, `QLinkedList`, etc).

Nous avons choisi la STL, principalement par habitude. Les quelques fonctionnalités spécifiques aux conteneurs Qt, notamment le Copy-On-Write, ne nous sont pas utiles. De plus, des différences subtiles à l'usage (`QList` n'est pas l'équivalent de `std::list`, usage différent des itérateurs) pouvant porter à confusion, nous préférons rester sur un terrain connu avec la STL.

3.2.5 Imagemagick

Pour créer le diagramme, nous utilisons `Imagemagick`, grâce aux *bindings* `C++` : `Magick++`. Cela permet de faire des compositions (i.e. arranger plusieurs images sur une même page) puis d'exporter le résultat dans différents formats.

Les images elle-mêmes sont obtenues en prenant les données brutes (pixels) dans le buffer OpenGL, ce qui donne des images de type bitmap. Ce n'est pas très satisfaisant : manipuler un format vectoriel

3. http://gcc.gnu.org/projects/cxx0x.html C++0x/C++11 Support in GCC
4. http://clang.llvm.org/cxx_status.html C++98, C++11, and C++14 Support in Clang

permet beaucoup plus facilement de rajouter des éléments (flèches, lignes, etc). De plus, cela permettrait d'exporter les diagrammes dans Inkscape pour les retoucher.

3.2.6 Lua pour la représentation intermédiaire

Le langage de description intermédiaire utilisé est Lua, auquel nous avons rajouté les primitives de plis (axiomes de Justin & Huzita [16]).

Cela permet un format de sauvegarde robuste (*i.e.* ne dépendant pas des algorithmes utilisés par Origram).

L'avantage de Lua est que nous n'avons pas besoin d'écrire un *parser* pour un langage de notre confection. Par ailleurs, Lua s'interface bien avec des programmes en C/C++, en particulier. Cependant, l'environnement d'exécution Lua n'est pas vraiment adapté à une exécution incrémentale, notamment si l'utilisateur veut annuler un pli.

Pour une future version d'Origram, il est envisagé d'écrire un *parser* pour le sous-ensemble de Lua nous intéressant pour la représentation intermédiaire et les sauvegardes ; tout en permettant à l'utilisateur d'écrire un pliage en utilisant toute la puissance de Lua, s'il le veut.

4 Organisation

4.1 Répartition des tâches

Nous avons divisé le projet en plusieurs *workpackages*, eux-mêmes subdivisés en *tâches*. La répartition temporelle des différentes tâches est présentée sur le diagramme de Gantt en annexe. Le détail et la répartition humaine des tâches sont les suivants (la personne en gras est responsable du bon déroulement de la tâche) :

WP 1 : Communication Ceci comprend toutes les opérations de communication, du sondage des attentes des gens à la gestion d'un site internet [5]. Cela a commencé par la convention d'origami LUO7, avec des plieurs se disant en général très intéressés par notre projet, et certains se disant prêts à tester notre logiciel dès que celui-ci serait utilisable. Nous avons aussi, par l'intermédiaire de Pierre Pradic, pu organiser une réunion avec Pierre Hyvernat, qui nous a donné son avis et ses conseils au sujet de notre projet.

Initial communication : Consiste à créer le site web, contacter la communauté origamiste pour connaître les fonctionnalités attendues

Production : site web, document résumant les attentes des utilisateurs.

Fabrice Mouhartem, Damien Rouhling

Real-world tests : Organiser des séances de test du logiciel auprès de la communauté origamiste. Cibler des publics variés : origamiste débutant ou créateur, âge, ... Concevoir des tests pertinents pour avoir des retours d'expériences sur différentes parties du projet.

Production : document décrivant les retours d'expérience des testeurs.

Jérémy Ledent, Fabrice Mouhartem

WP 2 : Interface graphique Ceci comprend toute la partie ergonomie du projet : ce à quoi l'interface finale doit ressembler, ainsi que la GUI et toutes ses fonctionnalités. Le comportement standard de fonctionnement était de type besoin/solution. Tous les autres *packages* fournissant au fur et à mesure des demandes d'accessiblité à des fonctionnalités au sein de l'interface graphique, qui devront ensuite être regroupées et organisées dans une interface logique et ergonomique.

5. http://perso.ens-lyon.fr/fabrice.mouhartem/origram

GUI ergonomy : Concevoir l'interface utilisateur : ce qu'elle doit permettre de faire et ce à quoi elle doit ressembler en pratique.

Production : document décrivant les fonctionnalités attendues et la façon dont la GUI répond à cette attente. Maquette d'interface.

Grégoire Beaudoire, Jérémy Ledent, Antoine Pouille

GUI implementation : Mettre en place les outils permettant de construire la GUI. Construire effectivement la GUI d'après le travail de l'équipe d'ergonomie. Concevoir un moyen de relayer les actions de l'utilisateur aux différents modules du programme.

Production : implémentation de la GUI. Moyen d'interfacer la GUI avec les autres parties du projet qui en ont besoin (API, bibliothèque, boucle événementielle, ...).

Antoine Pouille, Pierre Pradic

User feedback on GUI : Modifier et adapter l'interface graphique en suivant les retours d'expérience.

Grégoire Beaudoire, Antoine Pouille

WP 3 : Représentation intermédiaire Ceci comprend la définition de primitives permettant de représenter les étapes d'un pliage, et la production d'un format adapté à la représentation interne d'un pliage (avec éventuelle réutilisation d'un format existant). Ce format de représentation intermédiaire est indispensable aux *workpackages* « 3D » et « Production de diagramme ».

Intermediate representation : Concevoir un langage de description ou une structure de données permettant de représenter une séquence de pliage. Les actions de l'utilisateur doivent pouvoir être représentées. D'autre part, cette représentation doit permettre de générer un modèle 3D pour l'afficher à l'utilisateur, et doit permettre de générer un diagramme (réalisé par la tâche « diagram generation »).

Production : un langage de description ou une structure de données permettant de représenter une séquence de pliage.

Damien Rouhling, Jérémy Ledent, Fabrice Mouhartem, Armaël Guéneau, Pierre Pradic

User interaction : Prendre en compte les actions de l'utilisateur de la GUI, notamment sur la vue 3D (sélection de point, ligne, pointe, pliage). Traduire les actions de l'utilisateur en utilisant la représentation intermédiaire, notamment par l'implémentation de macros dans le langage intermédiaire. Fait beaucoup appel au code écrit par le groupe « physique du papier ». Initialement, c'est ce *package* qui devait écrire ce code, mais la difficulté de ces opérations a provoqué un changement dans notre organisation.

Production : code permettant d'appeler les différentes primitives de la représentation intermédiaire.

Pierre Pradic, Damien Rouhling, Antoine Pouille

User feedback on intermediate representation : Modifier et adapter la représentation intermédiaire pour répondre aux nouveaux besoins.

Armaël Guéneau, Jérémy Ledent, Pierre Pradic

WP 4 : Modélisation du papier/3D Ce *package* fournit le contenu de l'affichage OpenGL de la fenêtre Qt, où l'affichage du modèle et le choix des opérations s'effectue. Il développe une traduction de l'entrée utilisateur pour le langage intermédiaire, et ensuite la conversion en un modèle 3D à afficher. Cela demande alors un travail de modélisation de physique du papier, gérant le marquage des plis et les tensions parfois nécessaires à la construction du modèle et qui apparaissent dès que plusieurs couches de papier se superposent.

3D OpenGL : Mettre en place les primitives d'affichage et de manipulation 3D : lumière, rotation, textures, modèle 3D. Les besoins de la tâche « user interaction » doivent être anticipés (l'utilisateur devra pouvoir sélectionner des points, des lignes, ...).

Production : format permettant de décrire un modèle 3D quelconque, code générique permettant à l'utilisateur d'afficher et de manipuler un modèle 3D (faire tourner, zoomer).

Armaël Guéneau, Baptiste Jonglez, Antoine Pouille

Paper physics : Appliquer les opérations de pliage sur la représentation intermédiaire. Utiliser la représentation intermédiaire pour générer un modèle 3D du pliage. C'est là que la gestion des couches de papier intervient. C'est la partie la plus importante et la plus difficile du projet. Initialement, ce *package* ne gérait que la génération du modèle 3D. Mais la modification de la représentation intermédiaire pose déjà des questions en rapport avec la physique du papier et nous avons donc réorganisé nos tâches. Il est à noter que l'effectif de ce *package* a été augmenté jusqu'à faire intervenir tous les membres du projet.

Production : code permettant de modifier la représentation intermédiaire et d'en déduire une vue 3D du pliage.

Baptiste Jonglez, tous les autres

WP 5 : Production de diagramme Ceci comprend la génération d'un diagramme dans différents formats, à partir de la représentation intermédiaire, des rendus 3D, et d'éventuels choix de l'utilisateur, en respectant les conventions des origamistes. Cela implique de plus le développement d'heuristiques afin d'obtenir un résultat lisible et esthétique. Ce *workpackage* dépend fortement des autres : en particulier du *workpackage* « 3D ».

Production : code permettant de générer un diagramme dans différents formats (PDF, PS, image, HTML), en respectant les conventions usuelles des diagrammes.

Fabrice Mouhartem, Jérémy Ledent, Damien Rouhling

WP 6 : Intégration Assembler les différents composants logiciels pour obtenir un produit complet. Cela peut inclure l'harmonisation des systèmes de compilation, l'écriture de code d'interface entre composants, voire la définition de nouvelles sous-tâches au sein de composants existants. Ce *workpackage* dirigera le cycle intégration/test qu'il faudra répéter deux ou trois fois au cours du projet.

Production : un prototype complet du projet

Baptiste Jonglez, Damien Rouhling

Nous avons de plus défini quatre tâches annexes ne faisant pas partie des *workpackages* :

Bibliographie : Réunir tous les documents trouvés par les différents membres du projet et les rendre aisément accessibles.

Production : résumés des différents articles, permettant de cibler rapidement ce que l'on doit relire.

Grégoire Beaudoire, Fabrice Mouhartem

Coding conventions : Définir les standards à suivre, aussi bien pour le code (indentation, style) que pour les dépôts git ou le système de compilation du projet.

Production : document décrivant les bonnes pratiques à suivre.

Baptiste Jonglez, Armaël Guéneau

Mid-term report : Coordonner l'écriture du rapport de mi-parcours. Collecter les différentes parties écrites par chaque module.

Production : le rapport de mi-parcours

Damien Rouhling

Advanced features : Implémentation de fonctionnalités additionnelles, éventuellement proposées au cours des tests. Intégration de logiciels existants comme ReferenceFinder. Cette tâche n'a pas eu lieu par manque de temps.

Antoine Pouille, Baptiste Jonglez

4.2 Remises à niveau

Dans le groupe, chacun a des compétences différentes. Par exemple, Antoine a déjà travaillé sur des gros projets en `C++` avec GUI, tandis que Baptiste ou Armaël ont une expérience raisonnable de `git`.

Pour remettre tout le monde à niveau, nous avons donc organisé des « séances de rattrapage » d'environ une heure, où un membre du groupe explique aux autres un concept utile dans le cadre du projet.

Nous avons en tout organisé deux séances de ce type : une axée sur les fonctionnalités objets de `C++`, et l'autre pour l'utilisation de `git` dans le cadre d'un projet de taille conséquente.

4.3 Conventions de codage

Pour le bon fonctionnement du projet, et pour faciliter l'intégration de nouveaux programmeurs, nous avons fixé des conventions. Celles-ci portent aussi bien sur le code écrit que sur l'usage du dépôt `git`. Nous espérons par ce biais garantir une certaine cohérence du code et l'accessibilité de notre projet.

Les conventions complètes sont disponibles dans le dépôt git (`conventions/conventions.org`) ou sur http://ze.polyno.me/ens/origram/conventions.html.

À la fin du projet, le bilan est le suivant :
- L'objectif principal a été atteint : le code est propre, homogène, et relativement lisible
- Certaines des contraintes se sont naturellement relâchées au cours du projet (introduction d'attributs publics, 80 colonnes, syntaxe des commentaires). Cela a, au final, assez peu nui à la propreté du code ; en revanche, cela nous a permis d'avancer beaucoup plus efficacement.
- De façon plus générale, nous avons pu remarquer que s'imposer des contraintes sur la forme est généralement une bonne idée, mais que trop de contraintes introduit une grande perte d'efficacité et un frein important à la contribution. Trouver le bon équilibre est difficile.

4.3.1 Écriture de code

Anglais L'anglais est la langue de notre projet : les fichiers, variables, fonctions sont nommés en anglais. Les commentaires sont aussi rédigés en anglais.

Largeur du texte Le texte et le code doivent être limités à *80 colonnes*.

Tabulations Nous ne nous servons pas des tabulations. Elles sont remplacées par des indentations de deux espaces. Ce choix est motivé par la longueur variable des tabulations selon les éditeurs de texte. Sans tabulation, il est aussi plus aisé de déterminer la largeur du texte.

Classes
- Le nom des attributs doit commencer par _ ;
- Les attributs sont soit privés, soit protégés : des accesseurs sont alors requis ;
- Les classes sont définies dans des fichiers `.h` et les méthodes dans des fichiers `.cpp`, excepté si l'implémentation de la méthode est très courte.

Sources
- Chaque fichier source doit inclure le texte de la licence
- Chaque fichier `.h` doit être protégé : le contenu (hors licence) du fichier `toto.h` doit être encadré du texte suivant :

```
#ifndef TOTO_H
#define TOTO_H
...
#endif // TOTO_H
```

Standard C++ Nous utilisons le style Linux [6] pour C, avec quelques adaptations à C++.

Voici les éléments principaux de ce style :
- utiliser des espaces autour des opérateurs binaires et ternaires ;
- largeur de texte à 80 colonnes ;
- conditions très longues : casser la ligne juste avant l'opérateur booléen ;
- les ouvertures d'accolades se font sur une nouvelle ligne pour les déclarations de fonctions, `struct` et classes et en fin de ligne pour les boucles et pour la conditionnelle `if`.

6. https://www.kernel.org/doc/Documentation/CodingStyle

Voici nos modifications :
— utiliser des espaces plutôt que des tabulations ;
— les indentations sont larges de deux caractères ;
— les accolades sont obligatoires, même pour les conditionnelles à une ligne ;
— dans un `switch`, chaque `case` est indenté ;
— lors de la déclaration de pointeurs, l'étoile est accollée au type.
Pour vérifier que toutes ces conventions sont respectées, nous utilisons `astyle` [7].

Nous proposons notamment des fichiers de configurations pour les éditeurs `Emacs` [8] et `Vim` [9] afin d'aider au respect de nos conventions de codage.

Commentaires Les commentaires sont placés avant le code qu'ils décrivent. Ce sont des commentaires multiligne (/*...*/ et pas //). Tout invariant à maintenir doit être documenté.

4.3.2 Usage du dépôt Git

Anglais Tout comme le code du projet, le dépôt Git doit être exclusivement en anglais : les noms de branches, les messages de commit...

Compilation À chaque commit commit, le code doit toujours compiler. Pour cela, il est nécessaire de tester le programme avant de faire un commit.

Outre l'ennui que l'on évite ainsi aux autres membres du projet, cela nous permet d'utiliser `git-bisect`, un outil permettant de trouver de façon semi-automatique le commit introduisant un bug donné.

Branches Chaque tâche doit être réalisée sur une branche séparée, partant de la branche `master`. Cela évite de devoir fusionner inutilement des branches non directement corrélées. Les branches sont fusionnées régulièrement dans la branche `master`, si et seulement si elles sont propres : le code compile et les commentaires de commit sont clairs.

Réécriture d'historique La réécriture d'historique est autorisée tant que les commits n'ont pas été propagés sur le serveur. Elle est tolérée pour les commits propagés sur une branche autre que `master`, tant que tout les membres concernés sont avertis. Elle est strictement interdite sur la branche `master`.

4.4 Licence du code

Nous avons adopté un modèle le plus ouvert possible (code et historique de développement accessibles à tous), notamment parce que ce n'est pas le cas des projets similaires à Origram et que cela nous a gêné. Dans ce cadre, placer le code sous une licence libre est naturel.

Reste la question de la licence proprement dite. Nous avons choisi la licence MIT [10] (plus précisément, la version « Expat [11] »), car nous voulons une licence permissive. Si des parties de notre travail peuvent être utiles à d'autres, nous en sommes très heureux, et voulons leur simplifier la tâche (même dans le cas où le résultat ne serait pas lui-même libre).

Tous les fichiers sources d'Origram contiennent le texte suivant :

7. http://astyle.sourceforge.net/
8. http://ze.polyno.me/ens/origram/conventions.html#sec-1-7-1
9. http://ze.polyno.me/ens/origram/conventions.html#sec-1-7-2
10. http://opensource.org/licenses/MIT
11. https://en.wikipedia.org/wiki/MIT_License#Various_versions

4.5 Approche de l'implémentation

Progression en spirale ? Pendant la première moitié du projet (jusqu'au « midterm report »), nous pensions adopter une approche en « spirale » : c'est-à-dire, commencer par implémenter la fonction de pli la plus simple, pour ajouter peu à peu des plis plus avancés. Ainsi, nous aurions pu nous assurer à chaque étape que nous avions une version du logiciel stable, qui compilait et qui fonctionnait.

En réalité, nous avions largement sous-estimé la difficulté de plier un origami. Lorsque nous avons commencé à vouloir implémenter la toute première fonction de pli – plier selon une droite –, nous nous sommes rendus comptes que même cette fonction, supposée être la plus simple, posait beaucoup de problèmes. Chaque idée d'algorithme nous faisait trouver de nouveaux contre-exemples auxquels nous n'avions pas pensés, qui complexifiaient encore plus la tâche. C'est à ce moment-là que nous avons trouvé l'article de Ida et Takahashi (cf. 5.3), qui nous a fourni un algorithme pour implémenter ce premier pli. L'idée générale de cet article était assez proche de ce que nous avions déjà, mais il nous a fourni la bonne manière de propager les contraintes d'adjacence et de superposition des faces via les formules des ensembles C et Π_M, qui était le point principal sur lequel nous bloquions.

L'algorithme de pli décrit dans cet article étant déjà très complexe, il nous a fallu plusieurs semaines pour l'implémenter, avant de pouvoir avoir le moindre résultat à l'écran. Cela a été bien plus difficile que l'approche en « spirale »prévue initialement, mais c'était inévitable puisque ce pli de base ne pouvait pas vraiment être divisé en sous-étapes. Une fois que le premier pli a été implémenté, le débuggage et l'implémentation des plis suivants ont été bien plus rapides, car nous pouvions constater l'avancement du logiciel petit-à-petit, ce qui était bien plus motivant.

L'étape suivante sera d'implémenter d'autres plis plans plus compliqués que les 7 axiomes de base (cf. B.1 : pli inversé, pli aplati, ...). Nous avons déjà réfléchi à une manière possible de les implémenter via un « mode manuel »autorisant à faire une séquence de plis simultanément en autorisant l'algorithme à déchirer le papier. Mais ces plis n'étant pas traités par l'algorithme de Ida et Takahashi, ni par aucun autre article à notre connaissance, nous nous attendons à ce que leur implémentation pose elle-aussi de nombreux problèmes inattendus...

Tests et débogage Une fois que la première fonction de pli a été implémentée, il a fallu la tester afin de corriger les bugs que nous trouvions. Dans un premier temps, nous nous sommes contentés d'effectuer des plis jusqu'à en trouver un qui ne fonctionnait pas correctement, résoudre le bug, et réitérer. Cela nous a permis d'arriver à un stade où toutes les séquences courtes, vérifiables à la main, semblaient se comporter correctement.

Nous avons ensuite implémenté les axiomes suivants, que nous avons testés un à un, de manière similaire. Il a alors été possible de commencer à essayer de réaliser de vrais origamis.

Dans la version actuelle d'*Origram* sont implémentés 6 des 7 axiomes de Huzita et Justin. Théoriquement, notre logiciel devrait donc être capable de simuler tous les origamis dits « Pureland ». Ce sont les origamis plans qui sont faisables en n'effectuant que des plis *montagne* ou *vallée*, c'est-à-dire qu'on ne s'autorise pas les plis inversés, aplatis, etc.

Nous avions défini à l'avance des tests permettant de vérifier l'avancement et le bon fonctionnement de notre logiciel. Ceux-ci sont disponibles en annexe D.1. Le premier modèle, celui du panda, a pu être réalisé correctement avec *Origram*. Le résultat obtenu est disponible en annexe E. Il existe encore quelques problèmes sur des modèles plus conséquents, cette phase de test doit donc encore être prolongée avant de commencer à ajouter d'autres plis.

Côté performances, si on plie une feuille en deux successivement pendant une dizaine d'étapes, l'algorithme de pli atteint ses limites et le calcul prend plusieurs secondes. Cela est inévitable : comme le nombre de faces double à chaque étape, la complexité est exponentielle. Cependant, il s'agit là du pire cas, qui n'arrivera pas en pratique : un origami fait rarement plus de 16 couches, à cause des contraintes physiques du papier. Tant qu'on fait des pliages raisonables sans chercher à atteindre les limites du logiciel, le temps de calcul reste négligeable.

Des tests auprès de vrais origamistes n'ont pas encore été effectués. Il sera nécessaire d'en faire afin d'avoir des retours sur l'ergonomie du logiciel, pour pouvoir améliorer l'interface et ajouter des fonctionalités qui leur semblent nécessaires. Mais dans la mesure où *Origram* ne permet pour l'instant que de faire des diagrammes très basiques, il faudra encore attendre que le logiciel se développe pour envisager de tels tests.

5 Fonctionnalités

5.1 Interface utilisateur (GUI)

Depuis le début du projet, l'interface utilisateur a été conçue par le workpackage *GUI* lors de réunions et d'échanges de mails, avant d'avoir été raffinée incrémentalement jusqu'à l'état actuel.

La conception a utilisé QtCreator. L'équipe d'implémentation de la GUI a produit plusieurs prototypes, qui ont été présentés à l'ensemble des membres du projet afin de disposer d'avis. Cela a permis de les modifier en ajoutant des fonctions devenues nécessaires, modifiant des comportements afin d'améliorer l'ergonomie et à aboutir quelque chose de satisfaisant en termes d'utilisation.

Les boutons grisés dans l'interface correspondent aux fonctionnalités non implémentées, parfois non prévues.

Structure de la fenêtre Le design actuel comporte :

- une barre de menus contenant les fonctions utilisées ponctuellement, et indiquant les racourcis clavier disponibles. Elle est divisée logiquement en plusieurs parties, permettant d'agir sur : le projet courant ; l'édition du modèle ; la vue 3D ; la génération de diagramme.
- une barre d'outils en haut, comportant à gauche les outils principaux relatifs à la création/sauvegarde de fichiers, et des boutons annuler et rétablir.
  À droite, un bouton permet de prendre une vue de l'origami et de l'insérer dans le diagramme courant, en y ajoutant un commentaire. Un autre bouton permet de directement exécuter du code Lua [4] au lieu d'utiliser la vue 3D pour plier.

— une fenêtre de visualisation où est affichée une vue en 3D de l'état courant de l'origami, que l'utilisateur peut faire tourner à la souris par un « cliquer-glisser ».
Les points et arêtes peuvent être sélectionnés de façon intelligente en fonction de l'outil sélectionné. La dernière sélection peut aussi être annulée, et la sélection d'un objet peut être faite plusieurs fois (non consécutives) si nécessaire.

— un panneau sur la gauche contenant les différentes opérations (plis) pouvant être effectuées sur le modèle courant de l'origami. En bas de ce panneau est affichée une description de l'outil sélectionné et une explication pas-à-pas de son utilisation.
Dans la section « Simple folds » sont localisés les outils permettant de faire des plis de base correspondant aux axiomes de l'origami (B.4). Ce sont en fait les fonctions de pli fournies par les fonctions du backend. La section « Advanced folds » comporte des plis plus complexes qui sont couramment utilisés. Ils correspondent en fait à plusieurs plis simples simultanés. Ils n'ont pas encore été implémentés.
Entre les outils et leur description sont affichées des options pour ceux-ci. On peut choisir l'orientation d'un pli (vallée/montagne), choisir si on veut marquer les plis, et définir une valeur flottante si cela est pertinent (outil Place Point).

— à droite, une liste des étapes du diagramme qui ont été construites pour l'instant. Il est possible de cliquer sur une étape antérieure pour y revenir. Cela n'a pas encore été implémenté, car des boutons et menus permettent d'effectuer ces actions, et cette implémentation n'était ainsi pas cruciale. Au dessus, un slider permet de choisir l'espace entre les faces utilisé pour l'affichage dans la vue 3D.

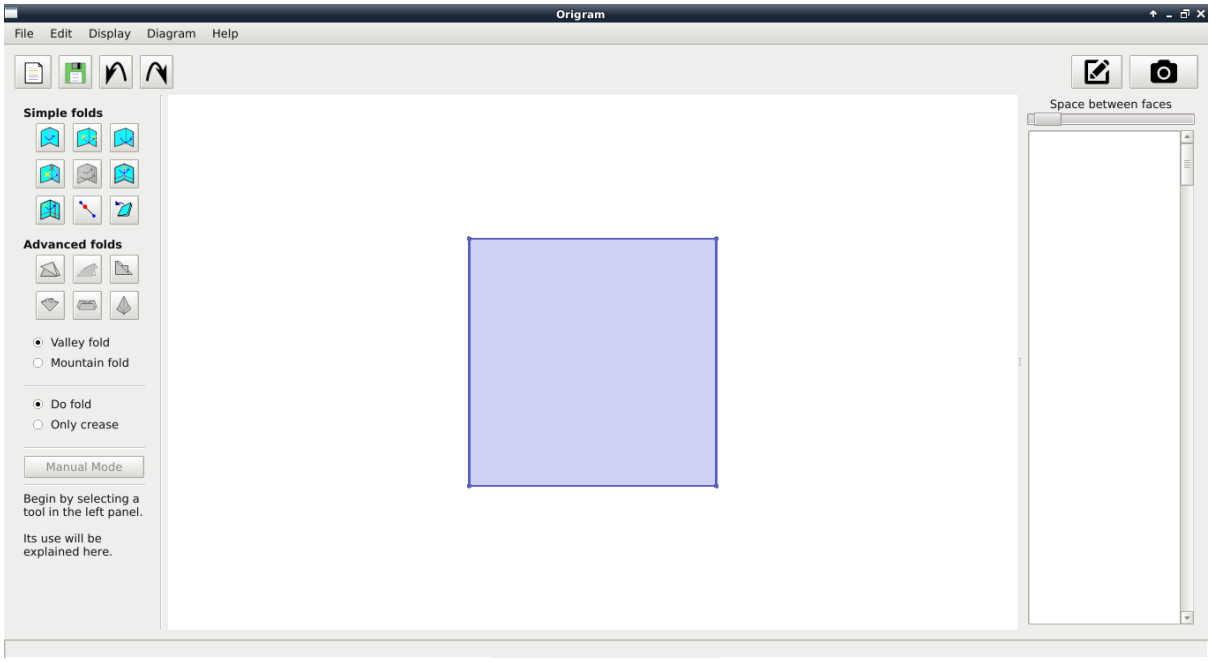


FIGURE 4 – GUI actuelle, à l'ouverture du programme

Ergonomie Dans un premier temps, nous pensions séparer l'utilisation d'Origram en deux phases : d'abord, spécifier la séquence de plis complète de l'origami, puis, après génération automatique de toutes les étapes de diagramme, éditer le diagramme étape par étape si le rendu n'est pas satisfaisant. Cela posait plusieurs problèmes, notamment le fait de devoir inférer la bonne position de caméra pour chacune des étapes du diagramme.

Finalement, pour la construction du diagramme, nous avons choisi une approche par « snapshot » : l'utilisateur construit une étape de pliage, puis oriente la caméra correctement et prend un snapshot (à l'aide du bouton en forme d'appareil photo, en haut à droite), pour générer une étape de diagramme. Celle-ci s'affiche à l'écran, et il peut alors la valider ou bien annuler et changer d'orientation. Ainsi,

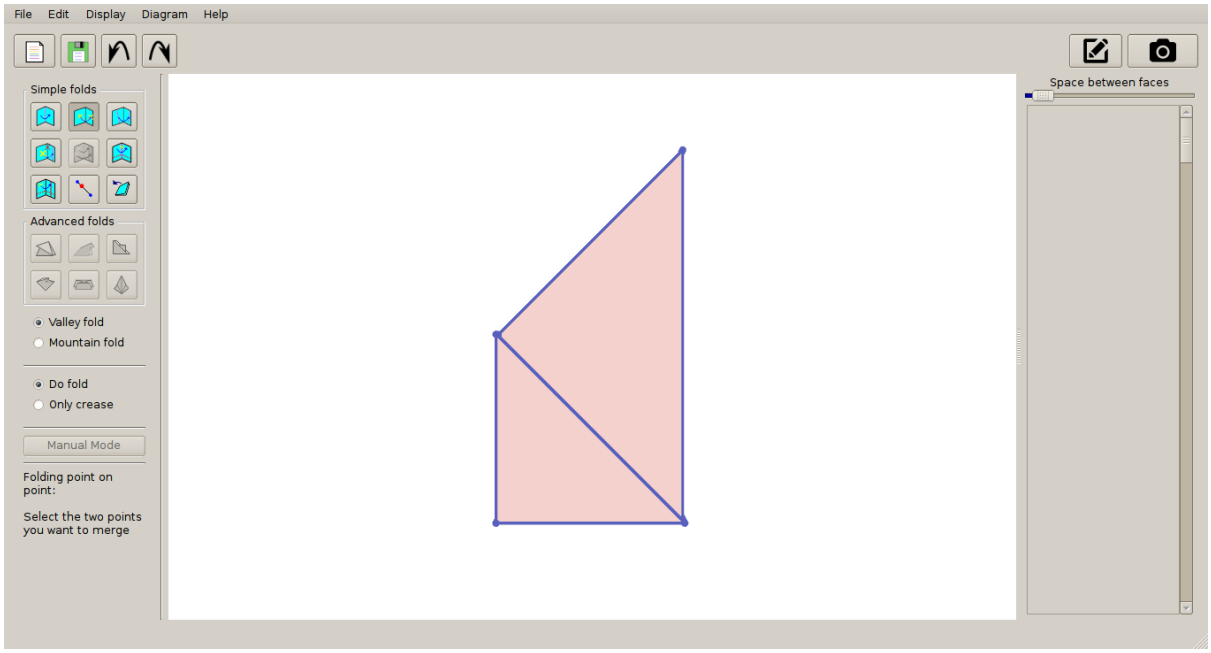


FIGURE 5 – GUI après avoir effectué quelques plis

c'est l'utilisateur qui choisit l'orientation. Il s'agit d'une approche étape par étape, les phases de pli et de diagramme sont mélangées. Les enquêtes du workpackage *communication* auprès de la communauté d'origamistes ont confirmé que les utilisateurs préféraient ce type d'approche.

Lors de la prise de l'image, un recadrage est fait et les sommets du modèle ne sont plus affichés, n'ayant pas de sens lors du pliage d'un papier. Accepter une image l'ajoute au diagramme courant. Des options dans le menu permettent de retirer la dernière image prise et de changer les informations supplémentaires ajoutées lors de la génération (telles que le titre du modèle, le nom de l'auteur...).

Concernant la vue 3D, des modes de vue différents sont disponibles, où les faces sont écartées par un paramètre sélectionnable dans le panneau de droite. Un des modes affiche les connections des faces. Une vue plane est possible en mettant le slider à 0. Il est aussi possible de centrer la vue, si un pli décentre le modèle.

Quant à l'utilisation des fonctions de pli, l'utilisateur doit fournir un à un les arguments de la fonction de pli qu'il veut utiliser. Pour cela, il est guidé par les instructions écrites en dessous des boutons de la barre d'outils. Par exemple, s'il clique sur l'outil « plier un point p_1 sur un point p_2 », une description de l'outil s'affiche, demandant de sélectionner deux points p_1 et p_2, conformément à ce qu'il veut faire.

Le programme sait ce qui doit être sélectionné (ce qui dépend de l'outil) et permet la désélection en cas d'erreur. Par ailleurs, une ligne peut être sélectionnée par deux points quand on veut spécifier l'orientation, ce qui est souvent nécessaire.

5.2 Gestion du modèle 3D

Afin de permettre à l'utilisateur de spécifier graphiquement les opérations de pliage, et de visualiser en temps réel l'avancement de celui-ci, notre programme comporte une vue 3D.

Le workpackage *3D* est responsable de son développement, avec comme objectif la définition d'une structure simple de *modèle 3D*, représentant l'origami en cours de construction et destinée à l'affichage.

La structure est clairement définie. Il s'agit d'une collection de faces, arêtes et points, ainsi que de couleurs (le papier pouvant comporter deux faces de couleurs différentes).

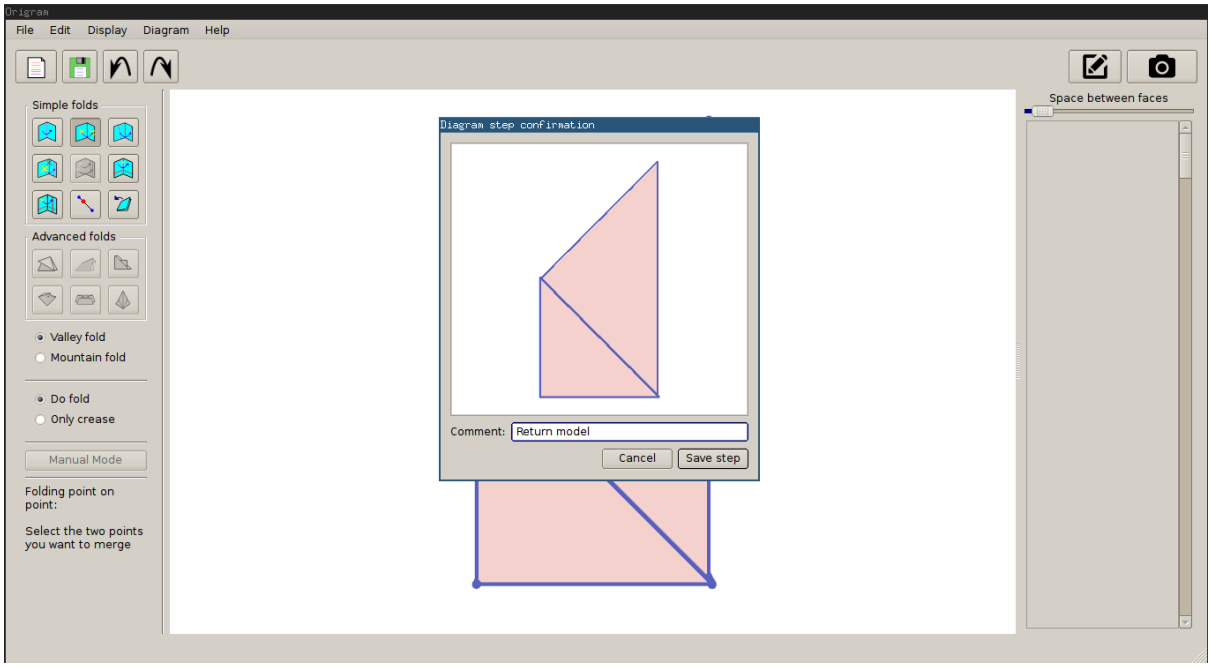


FIGURE 6 – Fenêtre de confirmation d'une étape de diagramme

Spécifier des arêtes et points en plus des faces peut sembler redondant ; en réalité ceux-ci doivent être dessinés séparément puisqu'ils désignent les plis du modèle et les points pouvant être utilisés pour créer de nouveaux plis (ils sont ainsi susceptibles d'être sélectionnés à la souris).

Cette structure est implémentée comme une classe abstraite, comportant une méthode `draw` virtuelle pure.

Une classe concrète hérite de cette classe abstraite qui décrit un modèle ; elle implémente `draw` en utilisant *OpenGL* pour afficher le modèle dans l'interface graphique. Le code s'occupant du rendu est clairement séparé du code décrivant le modèle, ce qui permettrait d'implémenter facilement des optimisations ou d'autres méthodes de rendu (rendu vectoriel par exemple).

Grâce à l'utilisation de la bibliothèque *libQGLViewer*, le modèle est visualisable dans l'interface ; il est possible de déplacer la vue à la souris (la caméra tourne autour du modèle), et l'on peut afficher si nécessaire les axes du repère.

5.3 Algorithme de pli

Afin de construire notre structure d'origami et d'implémenter les algorithmes permettant d'effectuer des pliages sur celle-ci, nous nous sommes appuyés sur un article de Tetsuo Ida et Hidekazu Takahashi, *Origami fold as algebraic graph rewriting* [11].

Idées générales de l'article Formellement, on définit la structure d'origami par un triplet $(\Pi, \sim, \succ)$, où :

- Π est un ensemble fini de faces.
- $\sim$ est une relation binaire symmétriques sur Π, appelée la *relation d'adjacence*. Celle-ci représente le fait que deux face peuvent être liées entre elles : si f_1 et f_2 sont des faces, on a $f_1 \sim f_2$ si f_1 et f_2 partagent une arête.
  Par exemple, sur la figure 7, à gauche, la face f_1 partage une arête XY avec f_2. Cette situation est représentée comme sur le dessin de droite, avec l'information supplémentaire que $f_1 \sim f_2$.
- $\succ$ est une relation binaire sur Π, appelée la *relation de superposition*. Elle représente l'agencement vertical des faces : $f_1 \succ f_2$ si f_1 est située au-dessus de f_2, et les deux faces sont en contact.

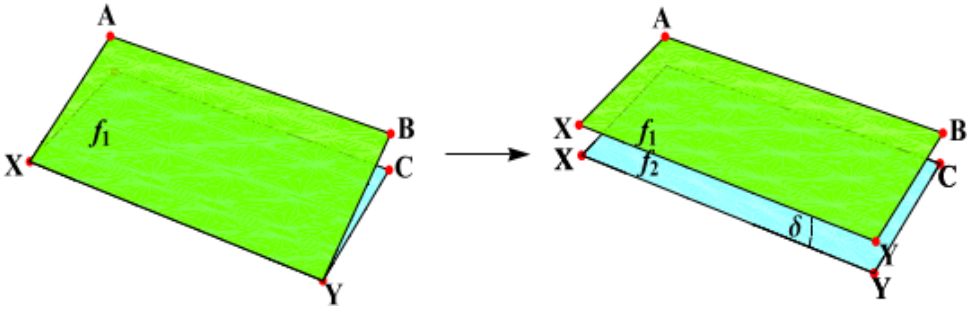


FIGURE 7 – $f_1 \sim f_2$ car elles partagent l'arête XY. Figure tirée de [11].

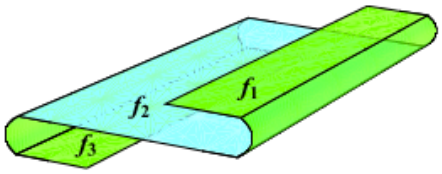


FIGURE 8 – Relation de superposition. Figure tirée de [11].

Par exemple, sur la figure 8, on a $f_1 \succ f_2$ et $f_2 \succ f_3$ mais pas $f_1 \succ f_3$ (l'espace entre les faces sur la figure 8 n'est là que par soucis de clarté – on le considère comme infiniment petit en réalité).

On doit maintenant réaliser une opération de pli sur cette structure. L'utilisateur fournit une ligne de pli orientée (ou *rayon*) r, une liste de faces F à plier selon cette ligne, et l'orientation du pli (vallée/montagne). On décide arbitrairement que ce sont les faces situées à droite de r qui bougent, le sens de la rotation dépend de l'orientation. Nous devons alors calculer le nouvel origami $(\Pi', \sim', \succ')$ obtenu. On divise cette action en deux étapes :

(1) Tracer la ligne de pli : sans plier l'origami, on divise chacune des faces à plier en deux nouvelles faces. Cette étape met donc à jour l'ensemble de faces Π' et la relation d'adjacence $\sim'$. La relation de superposition est également modifiée en $\succ''$, mais il s'agit pour l'instant d'un simple renommage des faces qui apparaîssent dans $\succ$.

(2) Effectuer le pli : on plie réellement l'origami, en calculant la nouvelle relation de superposition $\succ'$.

Pour la première étape, la difficulté réside dans le calcul de l'ensemble de faces à diviser. En effet, toutes les faces qui sont traversées par la ligne de pli ne vont pas forcément être pliées. On définit l'ensemble des *candidats* pour la division de faces par :

$$C = \bigcup_{f \in F} \{g \mid g \sim^*_{\neg \text{left}(r)} f\}$$

où $\sim^*_{\neg \text{left}(r)}$ désigne la clôture réflexive transitive de $\sim$ restreinte aux faces qui ne sont pas à gauche de r. Il faut ensuite vérifier que la ligne de pli traverse bien ces candidats.

Pour la seconde étape, on a recours à deux relations annexes :

— $f \asymp g$: signifie que, si on projette f dans le plan de g, l'intersection des deux faces est non-vide. C'est une relation symétrique

— $f \succcurlyeq g$: signifie que f est au-dessus de g et que $f \asymp g$. L'intérêt de cette relation est que sa réduction transitive est égale à $\succ$ (la preuve est dans [11]).

La relation $\asymp$ se calcule grâce aux coordonnées des sommets des faces. Pour calculer $\succcurlyeq'$, il faut d'abord calculer l'ensemble des faces qui vont bouger :

$$\Pi_M = \bigcup_{f \in D} \{g \mid g \ (\sim'_{\neg \text{on}(r)} \cup \succ'')^* \ f\}$$

où D est l'ensemble des faces qui ont effectivement été divisées à l'étape précédente, et qui sont à droite de r.

On peut enfin calculer $\succcurlyeq$ par induction sur la construction de l'origami : si $\succcurlyeq$ est bien définie avant la rotation des faces de Pi_M,

— Si $f \in \Pi_M$ et $g \in \Pi_M$: on renverse la relation entre f et g.
— Si $f \notin \Pi_M$ et $g \notin \Pi_M$: on conserve la relation entre f et g.
— Si $f \in \Pi_M$ et $g \notin \Pi_M$: si $f \asymp g$, alors $f \succcurlyeq g$ (resp. $g \succcurlyeq f$) si on fait un pli vallée (resp. montagne).

Nous avons donc besoin de structures de données adaptées pour pouvoir manipuler ces relations et effectuer les opérations nécessaires.

Différences d'implémentation par rapport à l'article Nous nous sommes assez vite rendus compte que l'algorithme décrit par Ida et Takahashi n'était pas directement implémentable dans le cadre d'*Origram*. En effet, il s'agit d'un article de recherche, qui a pour but d'étudier les aspects algorithmiques de l'origami, et non de produire un logiciel ergonomique et utilisable par tout le monde.

Ainsi, dans leur article, l'utilisateur doit préciser non seulement une ligne de pli, mais aussi un ensemble F de faces à plier. Nous ne voulions pas que l'utilisateur doive préciser manuellement cet ensemble, ce qui aurait été trop lourd côté ergonomie. De plus, pour comprendre comment choisir convenablement cet ensemble F, il faut connaître le fonctionnement de l'algorithme de pli – ce qui n'est pas envisageable pour un utilisateur lambda.

Nous avons donc dû trouver des heuristiques pour inférer un ensemble F convenable à partir de l'unique donnée de la ligne de pli. Le calcul de l'ensemble C a aussi été changé en conséquence. Au lieu d'appliquer directement la formule de l'article, on prend comme ensemble C un point fixe de la fonction $(\phi \circ \psi)$, où :

— $\psi : P \mapsto \bigcup_{f \in F \cup P} \{g \mid g \sim^*_{\neg \mathrm{left}(r)} f\}$
— $\phi : P \mapsto \bigcup_{(f,g) \in P^2} \{h \mid g \succcurlyeq^* h \wedge h \succcurlyeq^* f\}$

En pratique, on part de l'ensemble F, on applique la formule de l'article, puis s'il y a des faces situées entre deux faces de C qui ne sont pas dans C, on les y rajoute. Puis on réitère jusqu'à ce que l'ensemble C soit stable.

Dans l'article, Ida et Takahashi représentent un origami (*i.e.*, un ensemble de faces + deux relations) par un hypergraphe avec des labels sur les arêtes. Cela est sans doute fait dans un souci "esthétique", pour avoir une seule structure de données représentant tout l'origami. En pratique, ça n'a que peu d'intérêt : nous avons préféré avoir un graphe séparé pour chaque relation, et pas d'hyper-arêtes.

De plus, dans l'article, les mises à jour se font par des règles de réécriture, en considérant un codage des hypergraphes sous forme de mots. Nous en avons extrait des algorithmes s'appliquant directement sur nos structures de données.

Structures de données La liste des faces, ainsi que les deux relations, sont stockées dans des graphes. Les nœuds de ces graphes sont les faces ; il y a un graphe pour chaque relation.

Lors de la division des faces, les graphes sont dans un état intermédiaire et d'autres structures sont utilisées pour effectuer les mises à jour. Les faces sont divisées selon un rayon orienté (ce qui définit la gauche et la droite), et on place les couples (ancienne face, face fille) dans des ensembles qui correspondent respectivement aux faces filles à droite et aux faces filles à gauche du rayon.

Algorithmes Comme on l'a vu dans dans le paragraphe « idée générale », le calcul du pliage de l'origami nécessite d'effectuer des opérations sur les relations que l'on manipule. En particulier, un algorithme de clôture réflexive-transitive est nécessaire pour calculer les ensembles C et Π_M, ainsi qu'un algorithme de réduction transitive pour calculer $\succ'$ à partir de $\succcurlyeq'$. Ceux-ci étant assez classiques, nous ne les détaillerons pas ici.

L'article d'Ida et Takahashi décrit un algorithme permettant de plier l'origami selon une ligne. Une fois que cette fonction est implémentée, il est immédiat d'implémenter les 7 axiomes de Huzita et Justin (cf.

B.4) : chaque axiome revient juste à calculer une ligne de pli via une opération géométrique (calcul de médiatrice, de bissectrice, ...), puis plier selon cette ligne.

Input : Un rayon de pli orienté r et un origami $(\Pi, \sim, \succ)$
Output : Un origami plié $(\Pi', \sim', \succ')$
Algorithme :
• Calculer l'ensemble F requis par l'article d'Ida et Takahashi

(1) Première étape : « marquer le pli ».
- Calculer l'ensemble C des candidats à la division
- Diviser les faces de C : on obtient Π' le nouvel ensemble de faces. Lors de cette opération, il faut aussi *classifier* les faces de l'ensemble C, suivant qu'elles aient bel et bien été divisées ou non, et suivant qu'elles soient situées à gauche ou à droite du rayon. Cela servira à calculer l'ensemble Π_M lors de l'étape suivante.
- Calculer $\sim'$, la nouvelle relation d'adjacence. Cela se fait en deux passes, d'abord en calculant la relation d'adjacence sur les faces nouvellement créées, puis en mettant à jour la relation sur les anciennes faces pour prendre en compte les nouvelles faces.
- Calculer $\succ''$, une relation de superposition intermédiaire qui prend en compte les nouvelles faces. On utilise la formule d'induction de l'article.
- Supprimer les anciennes faces qui ont été divisées

(2) Deuxième étape : « effectuer le pli ».
- Calculer l'ensemble Π_M des faces qui vont bouger
- Vérifier la cohérence de l'ensemble Π_M. Pour cela, il faut vérifier qu'aucune face de l'ensemble Π_M n'est traversée par la ligne de pli. Si ce n'est pas le cas, alors le pli demandé est impossible : renvoyer une erreur.
- Appliquer une rotation aux points des faces de Π_M. Il s'agit de simples calculs sur les coordonnées des points.
- Calculer $\succ'$, la nouvelle relation de superposition. Pour cela, on calcule d'abord la relation $\succcurlyeq'$ à partir de $\succ''$ via la formule d'induction, puis on calcule sa réduction transitive.

5.4 Représentation intermédiaire

Pour assurer la connexion entre l'interface utilisateur et le modèle 3D, nous avons eu recours à une représentation intermédiaire. Celle-ci a pour but de stocker toute l'information nécessaire à la construction de l'origami, afin de générer d'une part le modèle 3D, et d'autre part le diagramme final. Pour assurer cette fonction, nous avons décidé d'utiliser un petit langage de script permettant de décrire une séquence de plis. C'est ce script, et uniquement lui, qui sera stocké pour sauvegarder un origami construit par le logiciel, par exemple.

Les manipulations de l'utilisateur génèrent, via des méthodes, un script décrivant une séquence de plis. Le script est donc généré automatiquement, et il est totalement transparent pour l'utilisateur. Lorsque ce script est exécuté, il manipule des structures de données internes, qui permettent de construire le modèle 3D et le diagramme.

Syntaxe La syntaxe complète du langage de représentation intermédiaire est spécifiée dans l'annexe C. Ce document est destiné aux utilisateurs avancés souhaitant écrire directement du code Lua afin de générer un diagramme, plutôt que d'utiliser l'interface graphique.

Les différentes fonctions et structures accessibles par l'utilisateur sont les suivantes :

— Fonctions de pli : elles sont basées sur les sept axiomes de l'origami [16]. Elles prennent en arguments des points et lignes (une ligne étant simplement la donnée de deux points), la parité du pli (Montagne ou Vallée), et un booléen indiquant s'il faut tourner les faces ou juste marquer le pli. Il était prévu qu'un argument optionnel permette de dessiner automatiquement ou non les flèches lors de la génération de diagramme ; ce n'est pas implémenté pour le moment.
— Fonction permettant de déplier un pli. Elle prend une ligne (une arête d'une face) en argument.

— Fonction permettant de placer un point arbitrairement. Ceci n'est pas conforme aux axiomes de l'origami, mais est pratique pour éviter de placer trop de lignes de construction. Elle prend en argument une ligne (deux points) et la proportion de la ligne où plier le point.
— Fonctions de diagrammage (non implémentées dans cette version) : toutes les fonctions relatives à l'aspect final du diagramme (flèches, orientation de la caméra). A l'origine, nous voulions que le langage ne contienne que des fonctions relatives aux plis et non au diagramme, mais il s'est finalement avéré nécessaire d'ajouter des fonctions de diagrammage afin de pouvoir afficher correctement certains plis spéciaux (ex. plis enfoncés B.2).
— Structure `foldRes` : renvoyée par les fonctions de pli, elle indique le résultat d'une division de faces. En notant `map<a → b>` une structure associative du type `a` vers `b`, `foldRes` = `map<face → [face, face, sommet, sommet]>` (`[...]` désigne un tableau).
Après un pli, la structure `foldRes` renvoyée associe, pour chaque (ancienne) face ayant été divisée, le tableau contenant la sous-face gauche obtenue, la sous-face droite, et les deux points définissant la ligne de division (dans l'ordre du rayon).

La description complète de la syntaxe du langage peut être trouvée en Annexe C.

Abstraction vis à vis de l'algorithme de pli En interne, pour désigner les sommets et faces l'algorithme de pli du modèle utilise des noms, générés automatiquement. La correspondance *nom* ↔ `objet géométrique` est dépendante de l'implémentation de l'algorithme. En effet, si l'algorithme est modifié, même de manière non significative - en inversant l'ordre de création de deux sommets par exemple - alors les noms seront différents.

Ainsi, on ne peut utiliser ces noms internes dans le script pour sauvegarder un pli : le moindre changement dans l'algorithme pourrait casser toutes les sauvegardes existantes.

Au lieu d'utiliser directement ces noms internes, on les lie à des variables Lua. Initialement, les variables `v0`, `v1`, `v2` et `v3` (dans le cas d'une face carrée) sont définies, désignant les quatre sommets ; `f0` désigne la face initiale.

Après un pli, de nouvelles faces et sommets sont créés : on peut les lier à de nouvelles variables Lua en utilisant la structure `foldRes` retournée. C'est ce que fait automatiquement la classe OrigamiInteraction lorsque l'on clique sur les boutons de la GUI.

5.5 Génération de diagrammes

L'un des objectifs de notre programme est de faciliter l'écriture de diagrammes pour origamistes. Nous avons opté pour des choix simples, car la vraie difficulté de ce projet résidait surtout dans l'algorithmique liée à la modélisation de l'origami.

Pour générer un diagramme, nous procédons en récupérant la vue 3D de l'origami, affichée dans l'interface. Chaque fois que l'utilisateur veut générer une étape de diagramme, il le spécifie dans l'interface et une image est récupérée. Nous obtenons à la fin une collection d'images qui sont agencées dans un document pdf.

Nous n'avons pas eu le temps d'implémenter la génération des flèches dans le diagramme. Cependant, nous l'avions prévue. Chaque action de base devait créer une flèche, mais il aurait été possible de désactiver cette fonctionnalité. En effet, certains plis complexes ne sont que des « macros », et seules quelques flèches correspondantes sont communément dessinées pour ces plis. Il aurait donc fallu choisir quelles flèches dessiner.

Nous utilisons donc OpenGL pour la récupération des modèles (par lecture du buffer). Il ne nous restait qu'à l'interfacer avec une librairie pour la création de documents au format pdf. Plusieurs librairies ont été considérées :
— LibHaru [3], qui nous a paru trop simpliste : elle ne peut pas lire de document pdf.
— GL2PS [2], qui exporte une structure GL en pdf mais la concaténation des documents obtenus paraissait compliquée.
— PoDoFo [6], qui n'a pas les défauts des librairies précédentes.

Cependant, compiler une image en pdf puis la décompiler pour recompiler un document plus gros n'est pas très efficace. Nous avons enfin trouvé l'interface C++ de ImageMagick [5]. Nous pouvons transformer nos projections OpenGL au format d'image ImageMagick, puis les assembler pour finalement compiler un document pdf.

Un exemple de diagramme obtenu se trouve en annexe E.

Conclusion

Notre projet visait à faciliter le partage et la transmission des connaissances en origami par le biais des diagrammes. Nous avons décidé de dépasser les nécessités purement liées au dessin pour fournir un programme permettant aussi la simulation des origamis. Ce programme se devait d'être intuitif, car il doit être accessible à tous, mais surtout moins fastidieux que les logiciels de dessins. Nous nous sommes rapidement limités aux origamis plans, car la simulation des origamis en général est un défi réellement difficile qui nécessite une bonne connaissance de la physique du papier.

Nous avons construit une interface graphique qui permet de visualiser l'origami en trois dimensions, ainsi que les relations entre chaque couche de papier. Nous avons modélisé l'origami par un ensemble de faces, parfois adjacentes, qui se superposent parfois. Ce sont ces faces, et ces relations, qui sont affichées dans l'interface graphique.

Pour la construction d'un modèle, nous nous sommes ramenés aux axiomes de Justin et Huzita [16] car tout modèle plan peut se construire comme une succession de ces axiomes. Nous avons alors écrit un langage intermédiaire qui, indépendamment de l'implémentation de ces axiomes, permet via un script codé en Lua de construire effectivement les modèles. Notre implémentation des axiomes a été construite à partir de l'article d'Ida et Takahashi [11] : cette article donne des algorithmes permettant de réaliser un pli selon une ligne. Il ne nous restait plus qu'à calculer la ligne de pli pour chaque axiome.

Nous avons enfin permis la génération de diagrammes en récupérant le modèle affiché dans l'interface graphique. L'utilisateur spécifie étape par étape ce qu'il veut afficher.

Nous avons rencontré quelques difficultés lors du déroulement de notre projet. Outre les problèmes de communication, classiques lorsqu'il y a un nombre conséquent de participants, la simulation effective des plis nous as longtemps bloqué. Nous n'avons trouvé que tardivement l'article *Origami fold as algebraic graph rewriting* [11] qui nous a permis d'avancer. Nous avons aussi passé un temps non négligeable à la définition de notre langage intermédiaire en raison de l'importance de ce point.

Nous ne sommes pas parvenus au bout de nos objectifs, mais nous avons fourni des structures adaptées à leur intégration dans le futur. Le diagrammes ne respectent pas encore les conventions de l'origami, mais nous disposons de primitives dans notre langage intermédiaire qui sont spécifiques aux questions relatives aux diagrammes. En particulier, l'ajout des flèches sur les diagrammes sera nécessaire.

Les plis les plus avancés, mais toutefois classiques en origami, ne sont pas encore implémentés. Ils peuvent se construire comme des macros à partir des axiomes de l'origami. Cependant, ils nécessitent de pouvoir effectuer plusieurs plis en parallèle. Nous avons pensé à un mode « manuel », qui permettrait de simuler ce parallèlisme en autorisant le déchirement du papier tant que le résultat final est correct (le déchirement permet de décorréler les faces et donc d'effectuer « en parallèle » des plis sur ces dernières).

Enfin, des progrès restent à faire quant à l'ergonomie de l'interface graphique. Les questions d'orientation des lignes sont peu intuitives, si bien que le comportement du programme peut surprendre l'utilisateur, même averti. Une solution pourrait être d'indiquer le comportement du modèle, par exemple par un fléchage, pendant la sélection.

Références

[1] Eos project. http://www.i-eos.org/.

[2] Gl2ps. http://www.geuz.org/gl2ps/.

[3] Libharu. http://libharu.org.

[4] Lua. http://lua.org.

[5] Magick++. http://www.imagemagick.org/Magick++/.

[6] Podofo. http://podofo.sourceforge.net.

[7] Origami Resource Center. Pureland origami diagrams. http://www.origami-resource-center.com/pure-pureland.html.

[8] Erik D Demaine and Martin L Demaine. Recent results in computational origami. In *Proceedings of the 3rd International Meeting of Origami Science, Math, and Education*, pages 3–16. Citeseer, 2001.

[9] Jerôme Goût and Vincent Osele. Doodle. http://doodle.sourceforge.net/about.html.

[10] Mickaël Hassine. Box pleating maker. http://h3md.free.fr/travaux/BPMaker/.

[11] Tetsuo Ida and Hidekazu Takahashi. Origami fold as algebraic graph rewriting. pages 393–413, 2010.

[12] Tung Ken Lam. Origami simulator and diagrammer. http://www.angelfire.com/or3/tklorigami0/.

[13] Tung Ken Lam. An investigation of the usability of software for producing origami instructions. Master's thesis, Open University, 2005.

[14] Robert J. Lang. Angle quintisection. http://www.langorigami.com/science/math/quintisection/quintisection.php.

[15] Robert J. Lang. Computational origami. http://www.langorigami.com/science/computational/computational.php.

[16] Robert J. Lang. Huzita-justin axioms. http://www.langorigami.com/science/math/hja/hja.php.

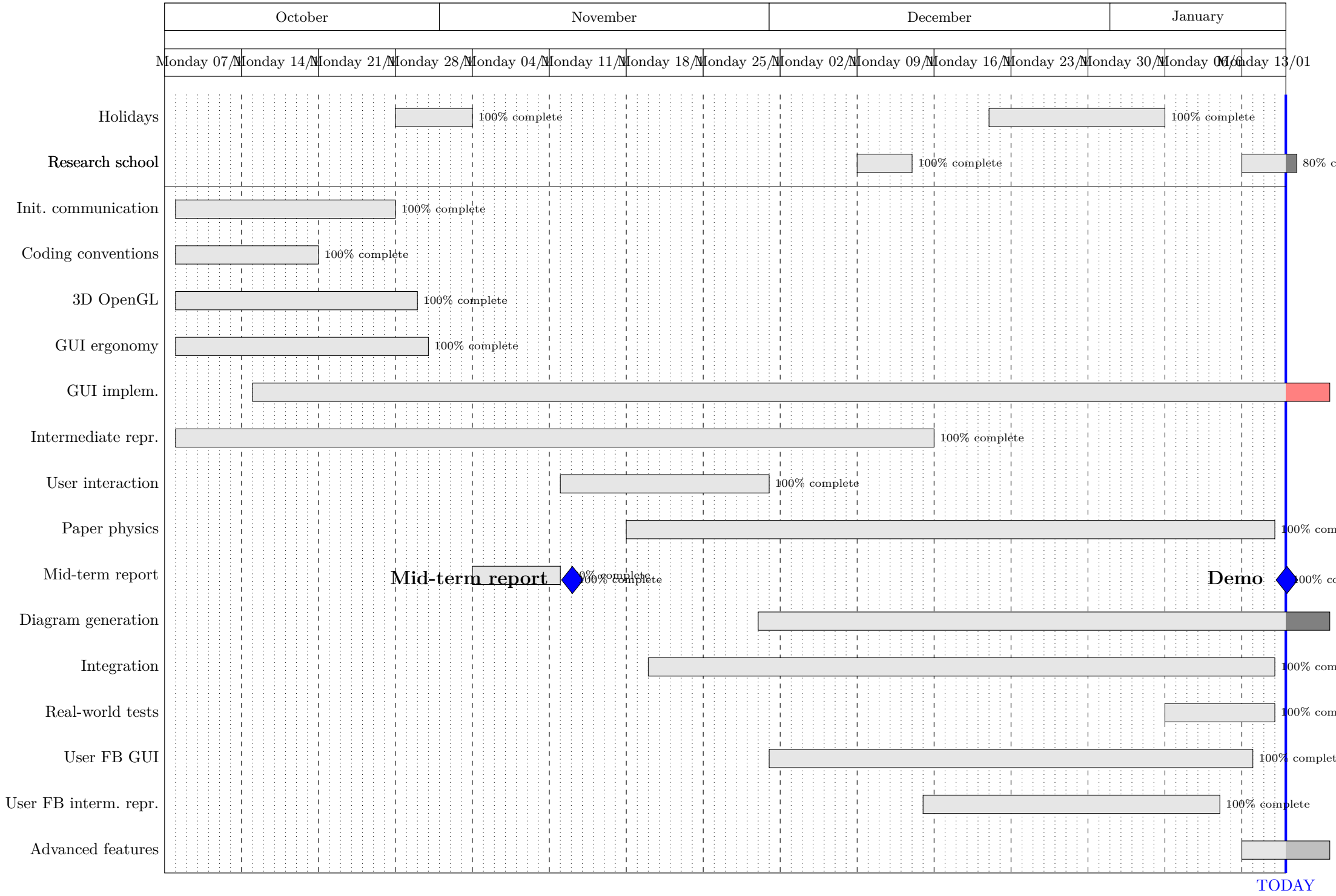
[17] Robert J. Lang. Origami diagramming conventions. http://www.langorigami.com/diagramming/diagramming.php.

[18] Robert J. Lang. *Origami Design Secrets : Mathematical Methods for an Ancient Art, Second Edition*. Taylor & Francis, 2011.

[19] Jun Mitani. Oripa. http://mitani.cs.tsukuba.ac.jp/oripa/.

[20] John Szinger. Foldinator. http://zingman.com/origami/foldinator.php.

[21] John Szinger. Origami xml. http://zingman.com/origami/origami_xml.php.


October
November
December
January
Monday 07/10
Monday 14/10
Monday 21/10
Monday 28/10
Monday 04/11
Monday 11/11
Monday 18/11
Monday 25/11
Monday 02/12
Monday 09/12
Monday 16/12
Monday 23/12
Monday 30/12
Monday 06/01
Monday 13/01
Holidays
100% complete
100% complete
Research school
100% complete
80% c
Init. communication
100% complete
Coding conventions
100% complete
3D OpenGL
100% complete
GUI ergonomy
100% complete
GUI implem.
Intermediate repr.
100% complete
User interaction
100% complete
Paper physics
100% com
Mid-term report
Mid-term report
0% complete
100% complete
Demo
00% co
Diagram generation
Integration
100% com
Real-world tests
100% com
User FB GUI
100% complet
User FB interm. repr.
100% complete
Advanced features
TODAY

A Glossaire

Canevas de plis (*crease pattern* en anglais) : document montrant les plis obtenus sur la feuille de papier lorsqu'on réalise puis déplie complètement l'origami.

Diagramme : document décrivant les étapes nécessaires à l'obtention d'un origami. Cela comprend des figures de l'origami en cours de réalisation, des flèches décrivant les actions à réaliser entre chaque étape, et des instructions textuelles.

Pli montagne (*mountain fold* en anglais) : pli selon une ligne, consistant à rabattre une des faces derrière l'autre. Le pli ainsi formé donne l'impression de « sortir » de la feuille.

Pli vallée (*valley fold* en anglais) : pli selon une ligne, consistant à ramener une des faces devant l'autre. Le pli ainsi formé donne l'impression de « rentrer » dans la feuille.

Origami plan (*flat-foldable* en anglais) : origami qui est pliable en deux dimensions. Tous les types de plis sont autorisés.

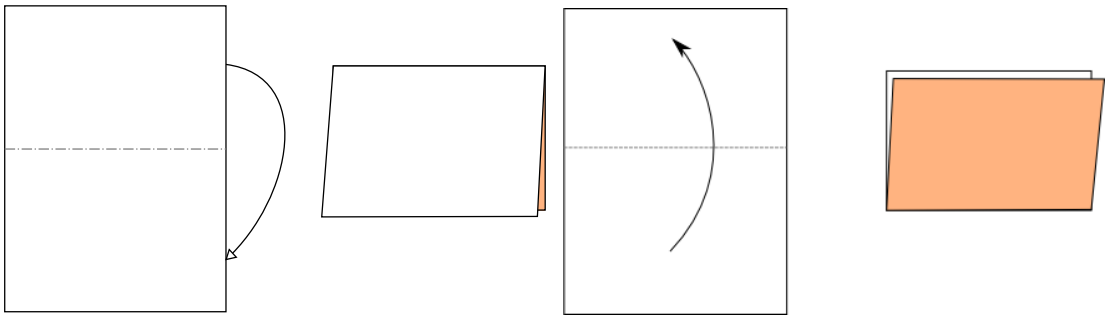
Origami « Pureland » : origami qui est pliable en deux dimensions, en n'utilisant que des plis *montagne* ou *vallée*. En particulier, on ne peut pas faire de pli complexe nécessitant d'ouvrir le papier, ou d'effectuer plusieurs plis simultanément.

B Liste des plis

Pour la réalisation des plis les plus courants, nous avions eu besoin de recenser les plis apparaissant dans l'origami. Nous avons ainsi distingué les plis simples — qui doivent être accessible rapidement dans l'interface — des plis complexes — qui peuvent être accessible de manière moins directe.

B.1 Les plis simples

Les plis montagnes et vallée : Il s'agit des pliages de base, représentés dans la représentation intermédiaire par des droites, et dans le diagramme par des pointillés pour les plis vallées et des points tirets pour les plis montagnes. Nous appellerons par la suite ces deux séquences des plis basiques.

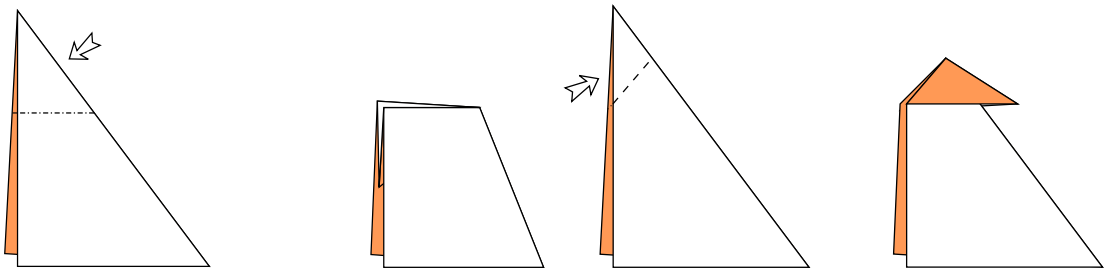


(a) Pli montagne (b) Pli vallée

FIGURE 9 – La brique de base : les plis vallées et montagnes

Les plis inversés : Dans ce pli, on inverse la *polarité* d'un pli en rajoutant deux plis basiques selon le sens de l'inversion : intérieur ou extérieur.

C'est un pli très représenté en origami, notamment pour donner une forme à la base ou encore pour les *changements de couleurs*, par exemple pour colorer le museau d'un animal.



(a) Pli inversé intérieur (b) Pli inversé extérieur

FIGURE 10 – Pli inversé intérieur et extérieur

Le pli aplati : En pratique, ce pli se compose d'un pli inversé auquel s'ajoute un pli vallée afin d'ouvrir la partie inversée.

Le pli aplati est souvent un préliminaire au pli pétale que nous présenterons ensuite.

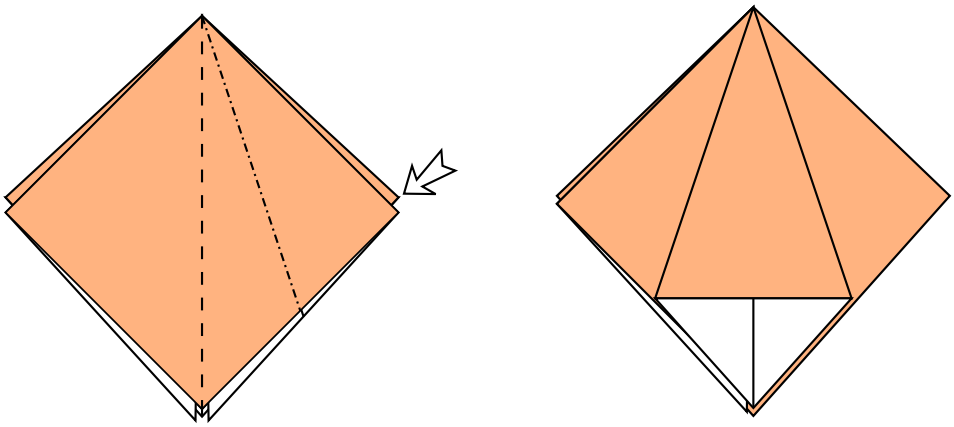


FIGURE 11 – Pli aplati

Le pli pétale : Il s'agit de deux plis inversés sur des bords « ouverts » d'un pli aplati auxquels se rajoute un pli vallée pour ouvrir le pli.

Son nom est tiré du fait que ce pli a initialement été utilisé pour former les pétales de la fleur de lys.

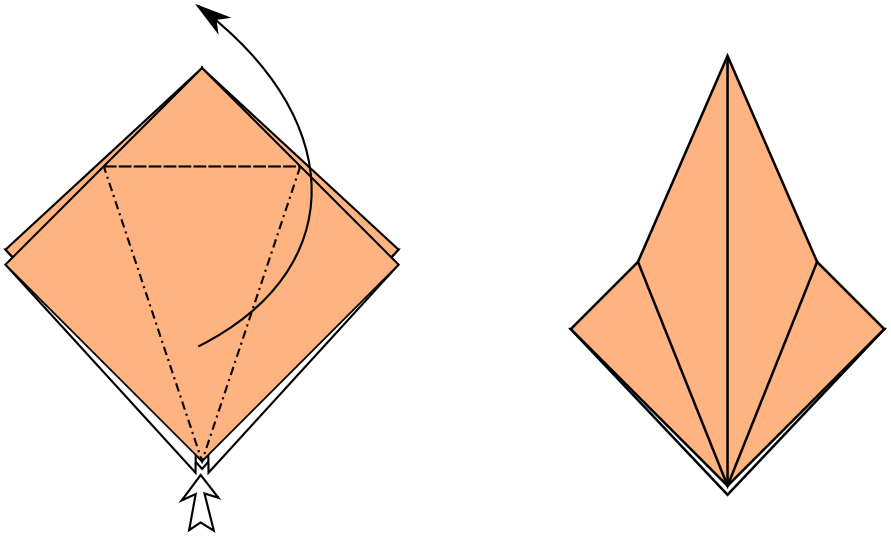


FIGURE 12 – Pli pétale

Le pli oreille de lapin : Un pli considéré comme *compliqué* car difficile à expliquer aux débutants, il s'avère en revanche très utilisé en origami (par exemple dans le cadre de la base du poisson, cf. chapitre B.3).

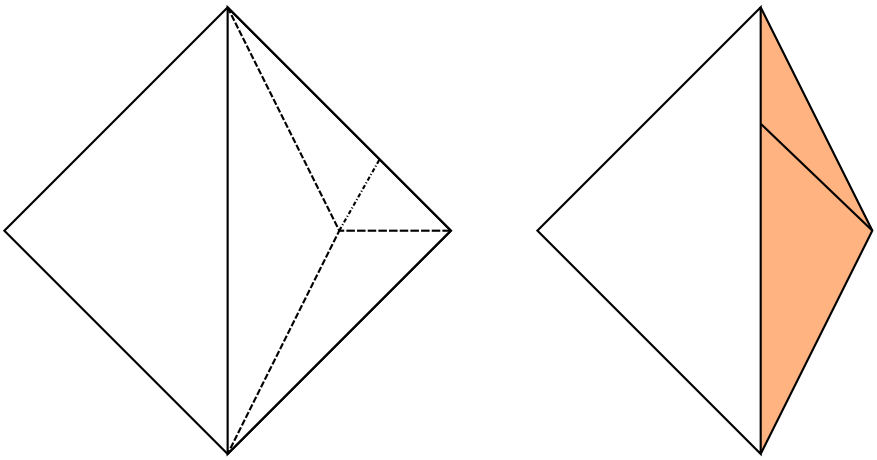


FIGURE 13 – Pli oreille de lapin

B.2 Les plis complexes

Le pli pivot : Ce pli, se compose d'un pli inversé ne se terminant à un emplacement arbitraire, ce qui force la création d'un pli vallée pour que le modèle reste plat.

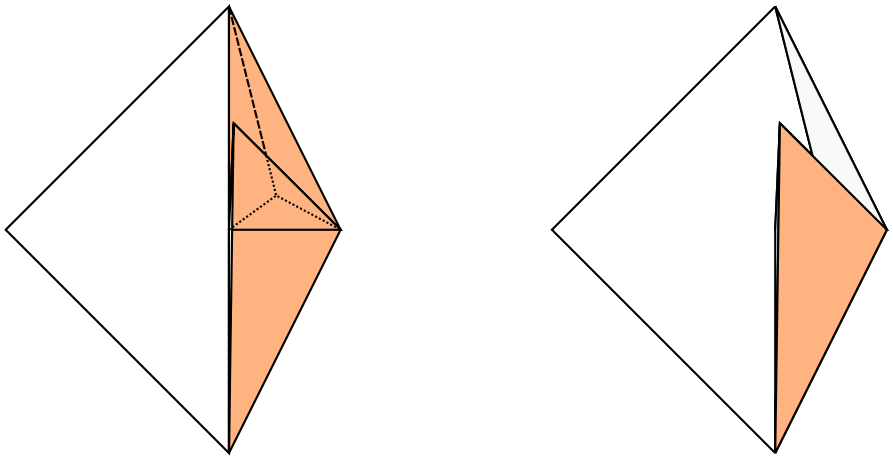
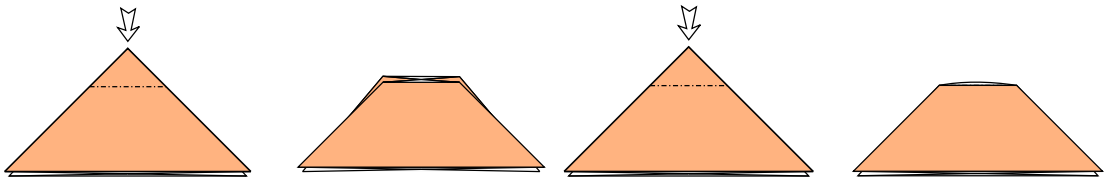


FIGURE 14 – Pli pivot

Les plis enfoncés : Il s'agit de prendre une partie du modèle et d'inverser tous les plis sur cette partie, provoquant ainsi un enfoncement.

Les enfoncements peuvent être de type ouvert (le modèle est entièrement ouvert ensuite les plis sont inversés) ou fermés (le pli est forcé en gardant le modèle fermé).



(a) Pli enfoncé ouvert (b) Pli enfoncé fermé

FIGURE 15 – Les deux types de plis enfoncés

Le pli zigzag : Il s'agit d'un pli vallée suivi d'un pli montagne.

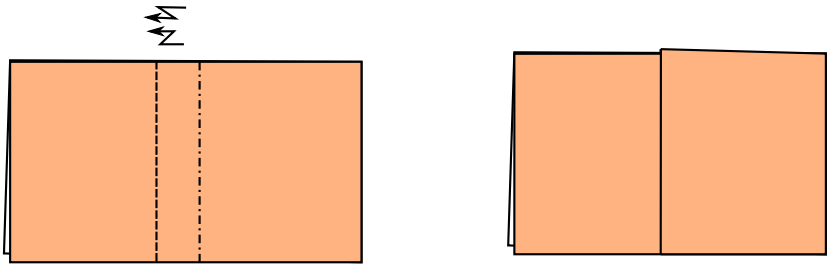


FIGURE 16 – Pli zigzag

Les plis courbes : Il s'agit des plis qui ne sont pas complètement appuyés, utilisés pour achever le pliage en le modelant.

Ce peuvent aussi être des plis terminant au milieu du papier, forçant alors le papier à se courber.

B.3 Les bases

La conception en origami peut se diviser en deux phases. La première consiste à choisir une base qui va déterminer le nombre de pointes du modèle : la grue traditionnelle par exemple est composée de **4** pointes : deux pour les ailes, une pour la tête et une pour la queue. La seconde phase va ensuite consister à affiner la base pour obtenir la forme voulue, pour la grue traditionnelle on va ainsi rajouter un pli pour former la tête, et deux plis pour ouvrir les ailes.

Ainsi on a vu naître des bases standards souvent réutilisées au début de la conception afin d'obtenir le nombre de pointes adéquat avant de commencer à plier. Les diagrammes commencent ainsi la plupart du temps par la description de ces bases souvent utilisées, parfois sans en préciser les séquences de plis (il est courant de voir un diagramme commencer par l'instruction « prendre une base de l'oiseau »). C'est pourquoi il est indispensable que le logiciel *Origram* permette de choisir une base de départ parmi les plus utilisées.

La base de la bombe à eau : Une base constituée des deux diagonales du carré pliées en montagnes et des deux médianes pliées en vallée. L'axe du modèle repose sur les côtés, ce qui donne une forme triangulaire à la base. Elle est constituée de 4 pointes.

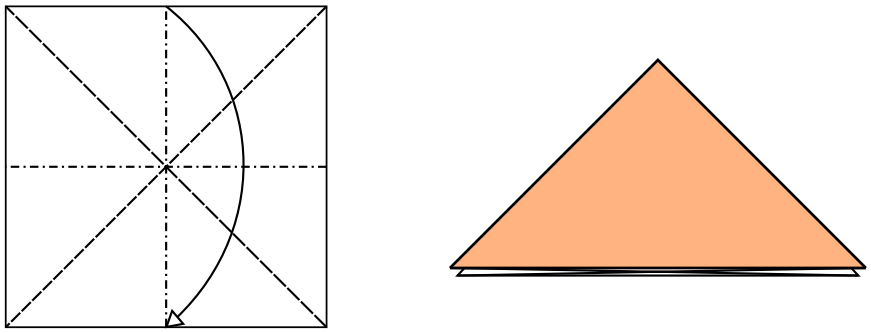


FIGURE 17 – Base de la bombe à eau

La base préliminaire : Il s'agit des mêmes plis que la base de la bombe à eau, elle dispose donc de 4 pointes. La différence réside dans l'axe du modèle qui se situe sur les diagonales, ce qui donne une forme carrée à la base.

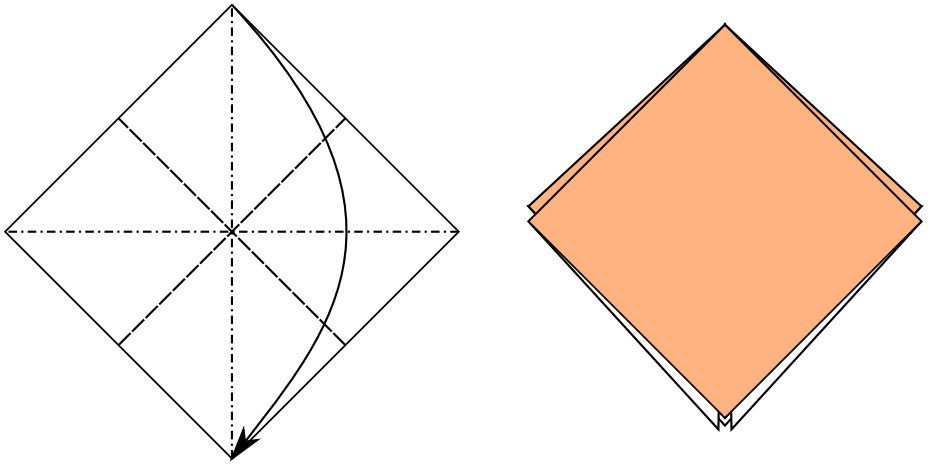


FIGURE 18 – Base préliminaire

La base de l'oiseau : On effectue deux plis pétales sur deux faces opposées d'une base préliminaire. On garde toujours la composition en 4 pointes.

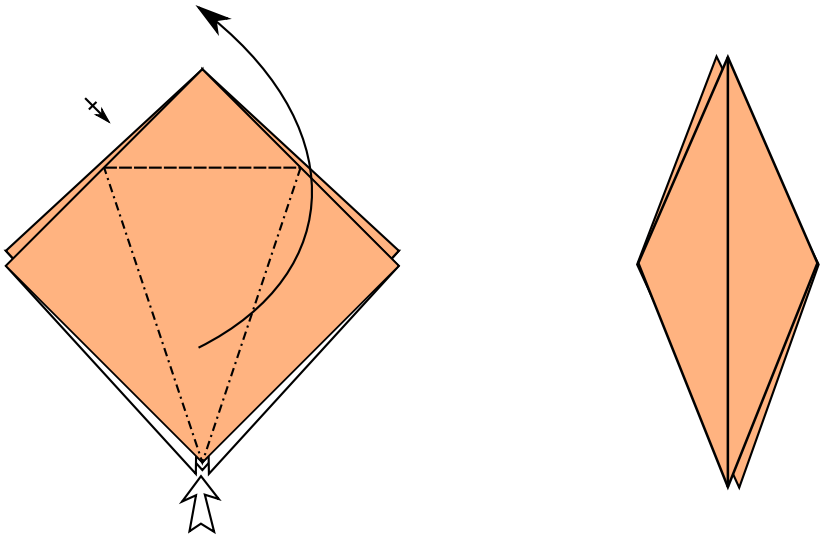
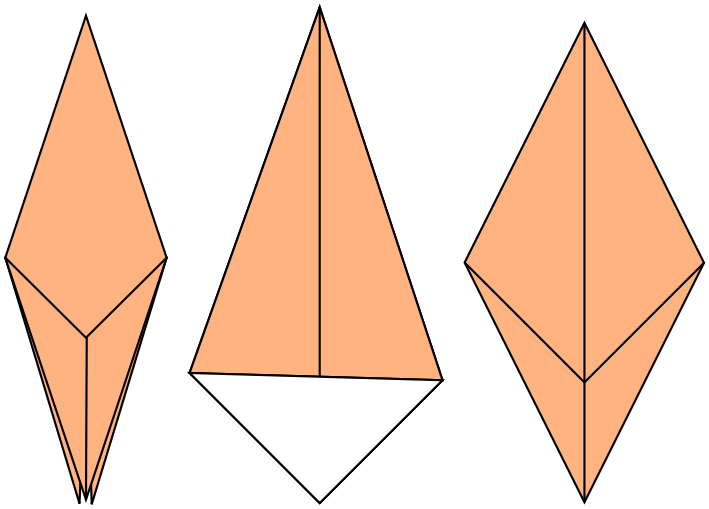


FIGURE 19 – Base de l'oiseau

La base de la grenouille : À partir de la base préliminaire on effectue un pli aplati avant d'effectuer les plis pétales. Ce qui monte le nombre de pointes à 8 (4 grandes et 4 petites).

La base du cerf-volant : On plie la diagonale et on replie deux bords adjacents (en une extrémité de la diagonale) le long de cette diagonale. Cette base dispose d'une pointe.

La base du poisson : Au lieu de replier les bords adjacents le long de la diagonale on effectue deux plis oreilles de lapins des deux côtés de la diagonale. Il en résulte une base composée de 2 grandes pointes et 2 petites pointes.



(a) Base de la grenouille (b) Base du cerf volant (c) Base du poisson

FIGURE 20 – Base de la grenouille, du cerf volant et du poisson

La base du moulin : Il s'agit d'une base disposant de 4 pointes (disposées de manière différente de celle de la base de l'oiseau).

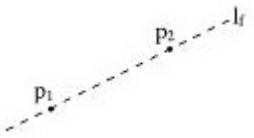
Remarque sur les bases traditionnelles : Les bases traditionnelles sont celles qui disposent du meilleur rapport longueur des pointes/largeur du papier pour leur nombre de pointes respectifs. C'est ce qui explique pourquoi elles sont toujours utilisées [18].

B.4 Différentes façons de faire un pli

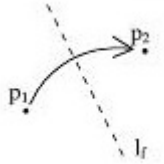
Les différents plis de l'origami classique peuvent tous se décomposer en étapes atomiques consistant à plier selon une ligne. Les travaux mathématiques de Humiaki Huzita et de Koshiro Hatori [16] ont permis de classifier toutes les manières de spécifier cette ligne de pli, sous la forme de 7 axiomes de base. Lorsqu'ils font un pliage, les origamistes ne pensent pas forcément à quel axiome ils utilisent à chaque étape. Mais pour simuler informatiquement un pliage, c'est sur ces axiomes que nous nous sommes appuyés. Ainsi, notre langage de représentation intermédiaire (cf. annexe C) comporte par exemple 7 fonctions de pli correspondant aux 7 axiomes.

Pour l'implémentation de ces axiomes, nous avons procédé ainsi : nous calculons la ligne de pli, puis nous plions selon cette ligne. La manière de plier selon une ligne a été déduite de l'article d'Ida et Takahashi [11] (cf. 5.4). Comme, dans l'article, les lignes de pli sont orientées (la partie qui se situe à droite de la ligne est celle qui subit la rotation), nous calculons aussi des lignes orientées.

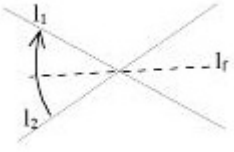
Axiome 1 : Etant donnés deux points p_1 et p_2, on peut faire un pli reliant ces deux points. La ligne de pli est directement définie par les deux points.



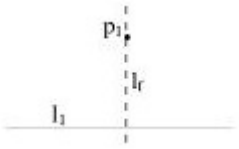
Axiome 2 : Etant donnés deux points p_1 et p_2, on peut faire un pli ramenant p_1 sur p_2. La ligne de pli est la médiatrice du segment $[p_1p_2]$, orientée de telle sorte que p_1 se trouve à sa droite.



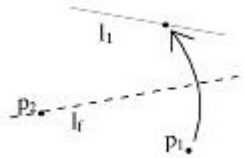
Axiome 3 : Etant donnés deux lignes l_1 et l_2, on peut faire un pli ramenant l_1 sur l_2. La ligne de pli est la bissectrice d'un des angles entre les lignes : si u_1 et u_2 sont les vecteurs directeurs *normalisés* respectifs de l_1 et l_2, la direction de la ligne de pli est $u_1 + u_2$. Si les lignes sont parallèles, la ligne de pli a l'orientation de l_1 et est équidistante des deux lignes.



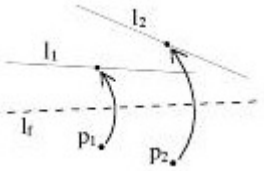
Axiome 4 : Etant donnés un point p_1 et une ligne l_1, on peut faire un pli perpendiculaire à l_1 passant par p_1. On calcule la normale à l_1 passant par p_1. Son orientation est le vecteur normal à la direction de l_1, dans le sens indirect.



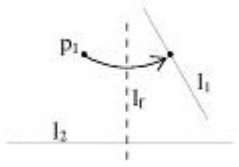
Axiome 5 : Etant donnés deux points p_1, p_2 et une ligne l_1, on peut faire un pli passant par p_2 et qui ramène p_1 sur l_1. On calcule *un* point d'intersection du cercle de centre p_2 et de rayon p_1p_2 avec l_1 (s'il existe) et on retourne la ligne de pli envoyant p_1 sur ce point. Ceci est correct, puisque si la ligne de pli passe par p_2, alors p_2 est sur la médiatrice du segment défini par p_1 et son point d'arrivée sur l_1.



Axiome 6 : Etant donnés deux points p_1, p_2 et deux lignes l_1, l_2, on peut faire un pli ramenant p_1 sur l_1 et p_2 sur l_2. Cet axiome n'est pas encore implanté. Cependant, nous savons comment calculer la ligne de pli. En effet, l'ensemble des tangentes à la parabole de foyer p_1 et de directrice l_1 est l'ensemble des lignes de plis envoyant p_1 sur l_1. Il « suffit » alors de trouver une tangente commune (si elle existe) aux paraboles de foyers respectifs p_1 et p_2 et de directrices respectives l_1 et l_2.



Axiome 7 : Etant donnés un point p_1 et deux lignes l_1, l_2, on peut faire un pli perpendiculaire à l_2 ramenant p_1 sur l_1. Pour calculer la ligne de pli, on calcule l'intersection de la parallèle à l_2 passant par p_1 avec l_1 (si elle existe) et on retourne la ligne de pli envoyant p_1 sur ce point. Cette ligne étant la médiatrice d'un segment parallèle à l_2, elle est bien orthogonale à l_2.



C Syntaxe de la représentation intermédiaire

Nous présentons ici les fonctions utilisées pour écrire le script décrivant l'évolution de l'origami.

Dans le code lua, pour des raisons d'implémentation, le nom des fonctions est en pratique préfixé par `backend:`. On ne l'indique pas ici pour plus de clarité.

Les fonctions marquées par **(*)** sont prévues, mais pas encore implémentées.

C.1 Fonctions de pli

Il y a sept fonctions de pli, correspondant aux sept axiomes de l'origami (voir B.4). Chaque pli est paramétré par sa parité, « montagne » ou « vallée », et par un booléen `creaseOnly` qui, lorsqu'il est à `true`, indique que l'on doit uniquement marquer le pli et non déplacer les faces. Ces fonctions retournent toutes la structure `foldRes`.

On écrit *nom_du_pli* (*arguments*) → *sortie* pour définir une nouvelle fonction, suivie de sa description.

- $foldLine(v_1, v_2, parity, creaseOnly)$: plie le long de la ligne $v_1 - v_2$.
- $foldPointToPoint(v_1, v_2, parity, creaseOnly)$: plie v_1 sur v_2.
- $foldLineToLine(v_1, v_2, v_3, v_4, parity, creaseOnly)$: plie la ligne $v_1 - v_2$ sur la ligne $v_3 - v_4$.
- $foldPerpLineThgPoint(v_1, v_2, v_3, parity, creaseOnly)$: plie selon la ligne orthogonale à $v_1 - v_2$ passant par v_3.
- $foldPointToLineThgPoint(v_1, v_2, v_3, v_4, parity, creaseOnly)$: plie v_1 sur $v_2 - v_3$, la ligne de pli obtenue passant par v_4.
- **(*)** $foldPointPointToLineLine(v_1, v_2, v_3, v_4, v_5, v_6, parity, creaseOnly)$: plie v_1 sur la ligne $v_3 - v_4$, v_2 sur $v_5 - v_6$.
- $foldPerpLinePtoL(v_1, v_2, v_3, v_4, v_5, parity, creaseOnly)$: plie perpendiculairement à la ligne $v_1 - v_2$, en envoyant v_3 sur $v_4 - v_5$.

Par ailleurs, chaque fonction de pli devraient avoir un argument booléen optionnel, indiquant si les flèches de pliage correspondantes doivent être générées automatiquement ; par défaut, cet argument valant `true`. Il est utile de le mettre à `false` lors de la définition de macros composant des plis avancés et représentées par une (série de) flèche(s) différente(s). Ceci n'est pas encore implémenté.

Il est également possible de changer un pli existant. Il existe donc des fonctions de « mise à jour » :

- **(*)** $reverse(v_1, v_2)$: inverse le pli défini par $v_1 - v_2$, ne retourne rien.
- $unfold(v_1, v_2)$: déplie selon $v_1 - v_2$, ne retourne rien.

C.2 Définition d'un point arbitraire

- $assertVertex(v_1, v_2, coef)$: Définit un nouveau point, donné par une position arbitraire le long d'une ligne existante ($v_1 - v_2$). Cette position est donnée comme le barycentre des extrémités de la ligne, barycentre de coefficients $coef$ et $1 - coef$. Cette fonction viole les axiomes de l'origami, mais est utile pour décrire des origamis non totalement traditionnels.

C.3 Fonctions de diagrammage (*)

La représentation intermédiaire est utilisée pour décrire à la fois le modèle 3D de l'origami, et le diagramme final. Nous avons besoin de plusieurs fonctions spécifiques à l'affichage.

Celles-ci ne sont pas implémentées pour le moment.

- *step*(*annotation*) → *unit* : fin d'une étape de diagramme. Cette fonction crée une nouvelle étape dans le diagramme, accompagnée de l'annotation passée en argument. Cela correspond au bouton "snapshot" dans l'interface utilisateur.
- *show*() → *unit* : affiche l'étape courante dans le modèle 3D de l'interface graphique. Ceci peut être fait plusieurs fois pour une même étape du diagramme. Cette fonction est utilisée pour spécifier que plusieurs plis doivent être faits simultanément : certaines macros (par exemple, *squash fold*) sont composées de plusieurs plis de base qui ne peuvent être réalisés de façon séquentielle sans intersection de plans ; or nous voulons n'afficher que des étapes décrivant un origami cohérent dans l'interface graphique. Toutes les fonctions de pli appelées entre deux appels à *show*() sont appliquées simultanément.
- *move_cam*(θ, ϕ, ψ) → *unit* : déplace la caméra relativement à la position courante. Les arguments sont les angles d'Euler.
- *reset_view*() → *unit* : réinitialise la caméra à sa position initale.

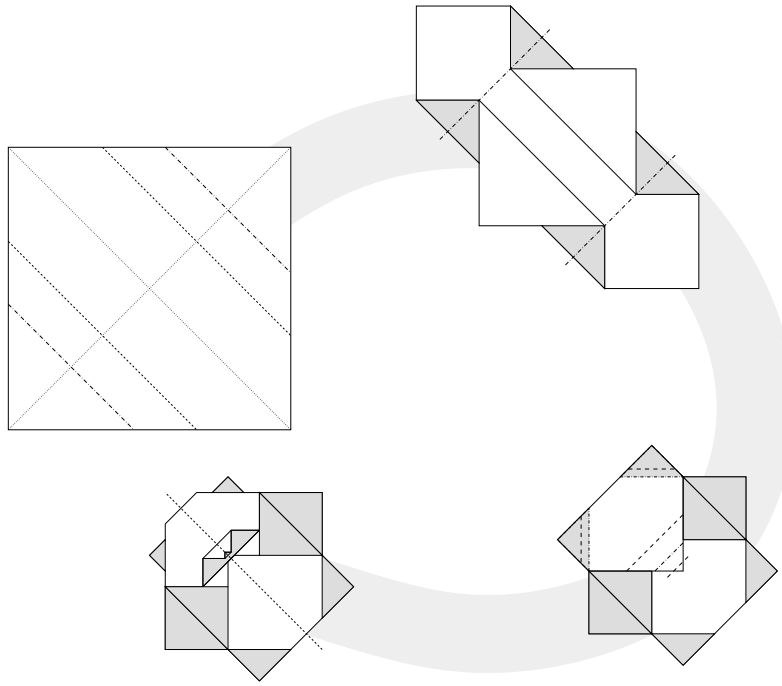
— $arrow\_fold(point_1, point_2, type) \rightarrow unit$: dessine une flèche du point $point_1$ au $point_2$. L'argument *type* spécifie le type de flèche à dessiner.
— $arrow\_push(line) \rightarrow unit$: dessine la flèche correspondant à l'action d'appuyer selon la ligne *line*.
— $arrow\_pull(line) \rightarrow unit$: dessine la flèche correspondant à l'action de tirer selon la ligne *line*.

D Tests

Cette annexe contient des tests que nous nous sommes fixés pour vérifier le bon fonctionnement et l'avancée de notre logiciel.

D.1 Origamis purs

La version actuelle d'*Origram* permet de plier et diagrammer la plupart des origamis dits *Pureland*. Il s'agit des origamis faisables à partir d'une feuille carrée en ne faisant que des plis simples, c'est-à-dire qu'on ne s'autorise qu'à plier le modèle en deux selon une droite. Dans la liste des plis de l'annexe B.1, on a donc uniquement le droit aux *pli vallée* et *pli montagne*. Voici deux diagrammes d'origamis purs faisables avec *Origram*. D'autres diagrammes purs sont disponibles sur [7].



Pureland panda (Sy Chen ??)

FIGURE 21 – Diagramme de panda. La version Origram est disponible en annexe E

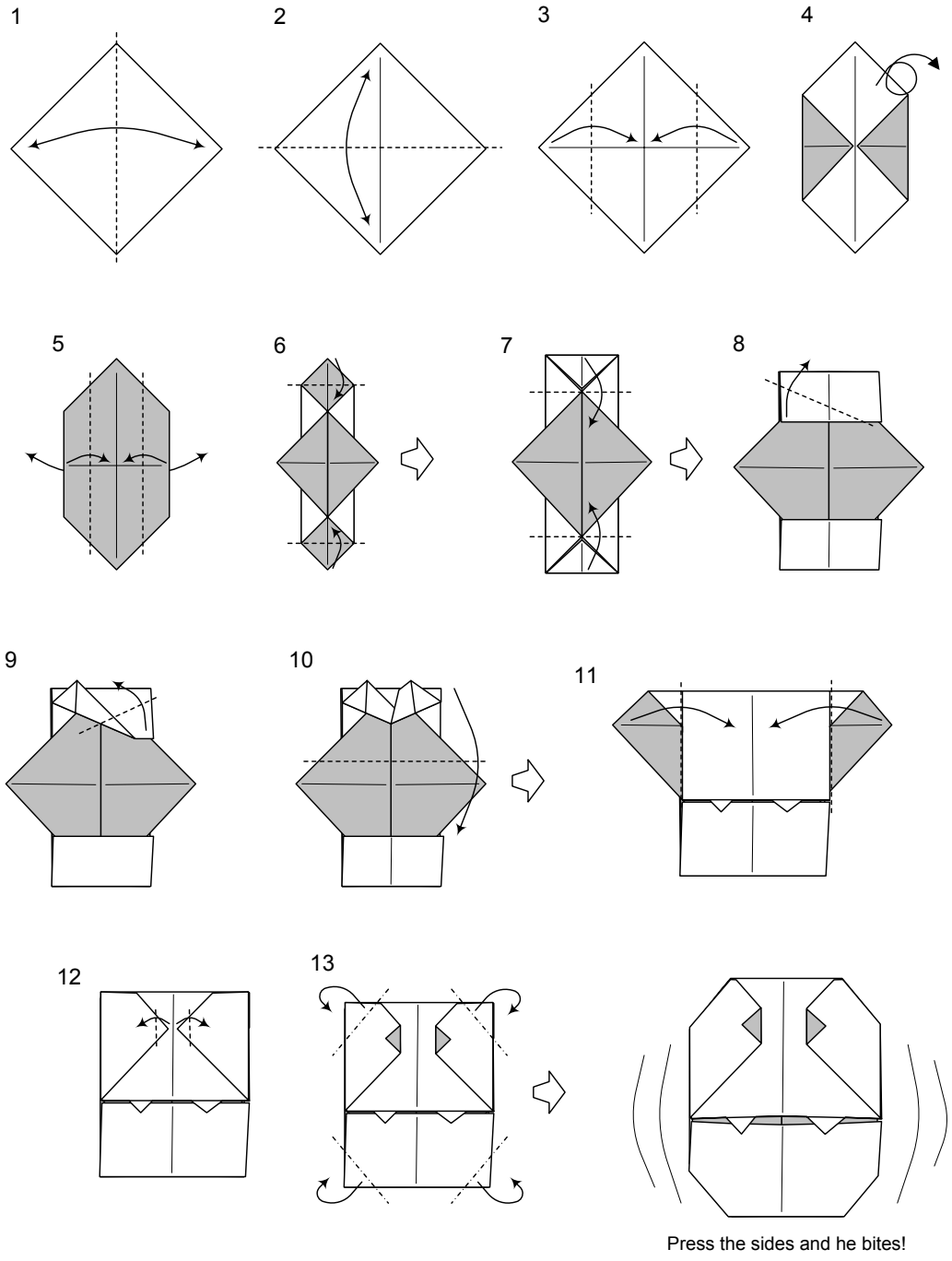


FIGURE 22 – Diagramme de dracula

D.2 Pliages plans

La prochaine étape pour Origram serait d'arriver à simuler tous les plis plans (cf. B.1 : pli inversé, pli aplati, ...). Les deux diagrammes suivants sont parmi les origamis les plus simples à réaliser et à diagrammer. À part pour l'étape finale consistant à ouvrir le modèle, ils ne nécessitent que des pliages plans. Notre prochain objectif pour le logiciel *Origram* est d'arriver à produire ces diagrammes.

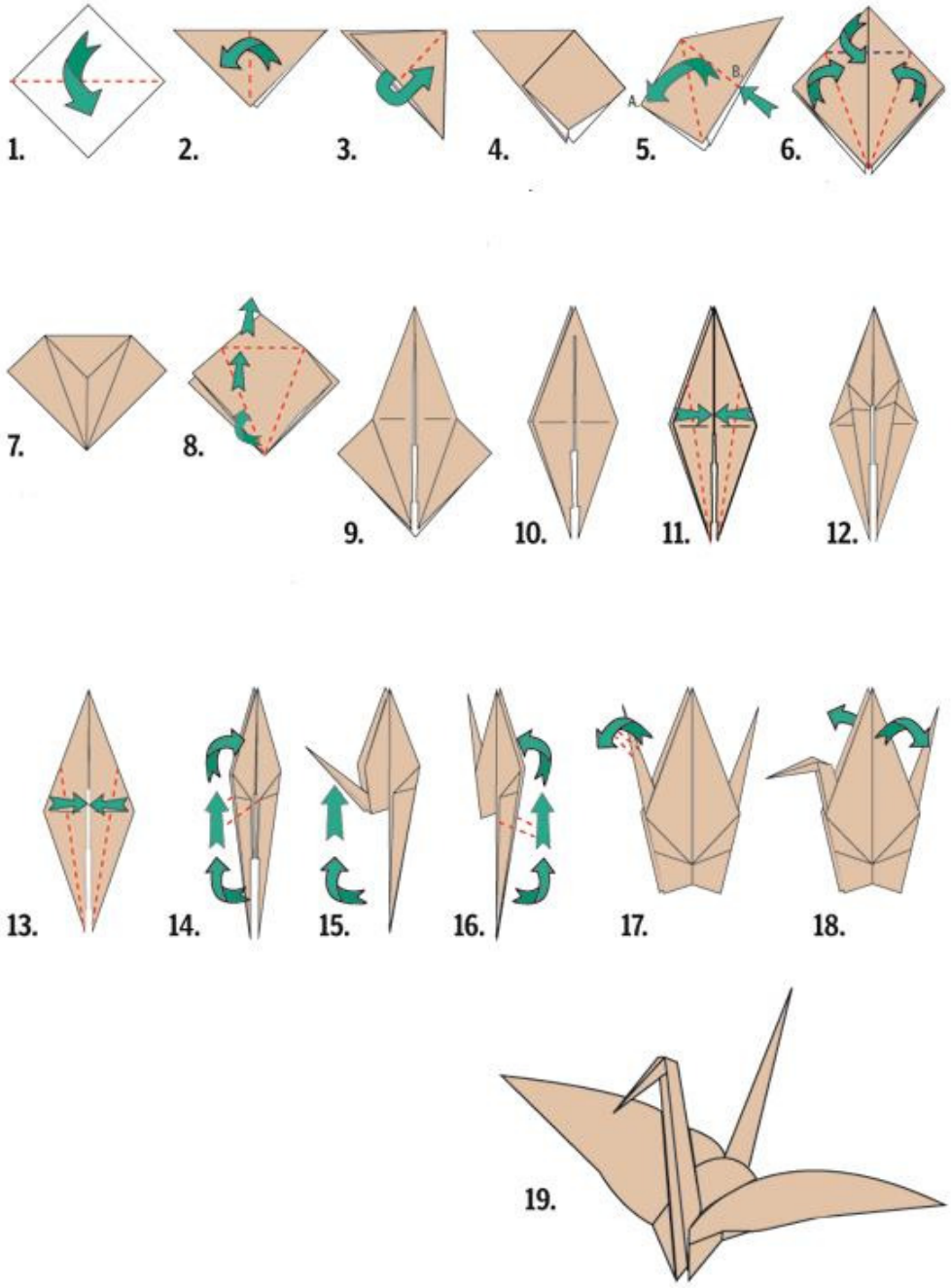


FIGURE 23 – Diagramme de la grue

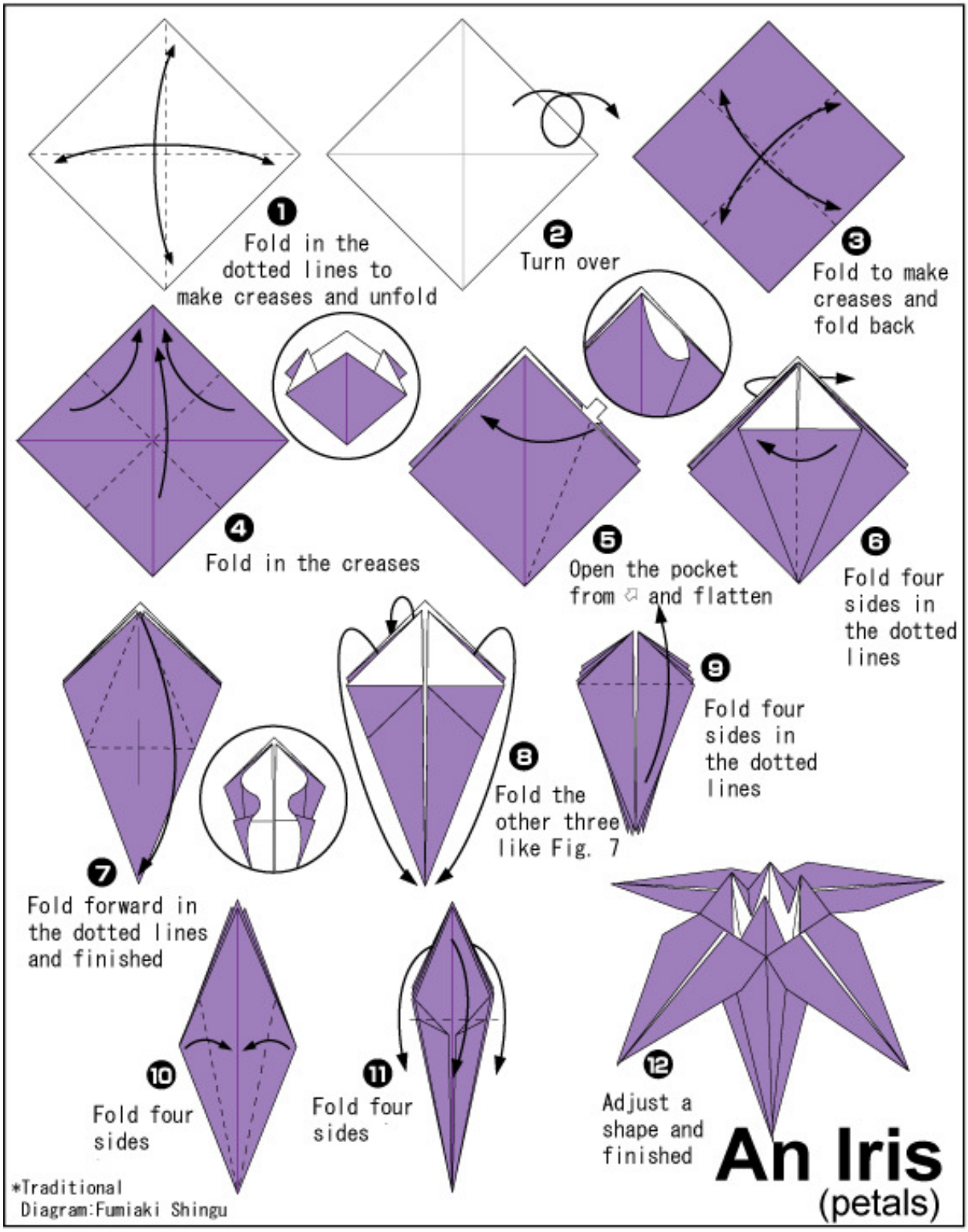


FIGURE 24 – Diagramme de la fleur de lys

E Exemple de diagramme

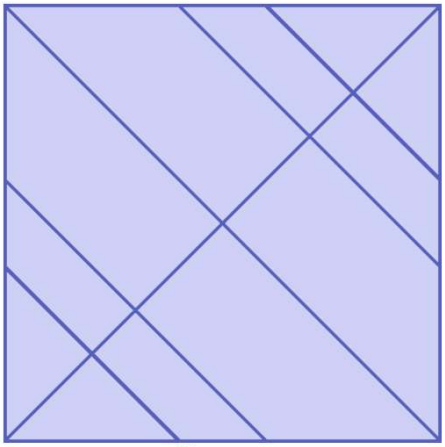
Voici un diagramme obtenu avec notre programme. Certaines étapes ont été commentées, d'autres non.

Model name: Panda

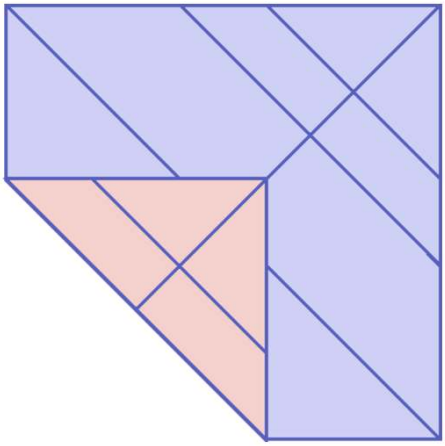
Author: Nunurse

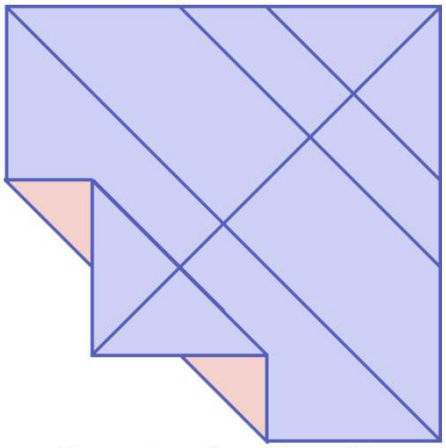
Difficulty: Much difficult

Comment: Love panda

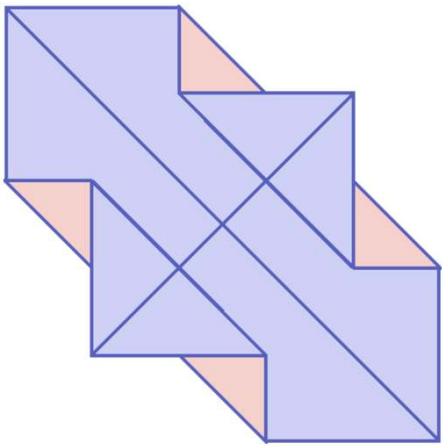


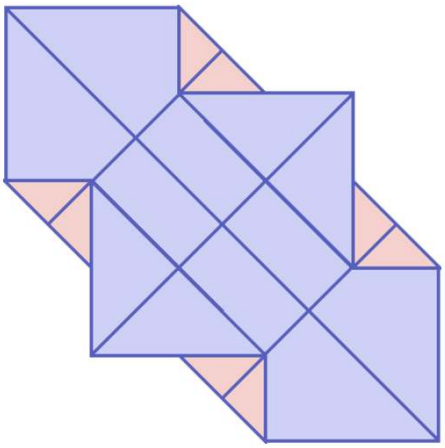
Crease

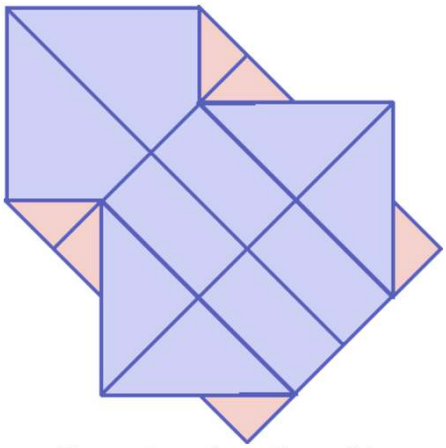




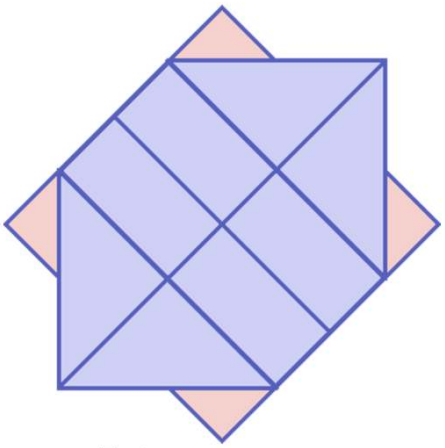
Repeat on the other side



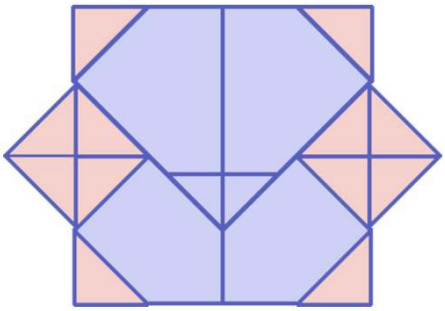


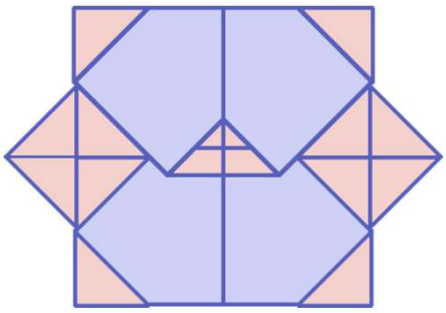


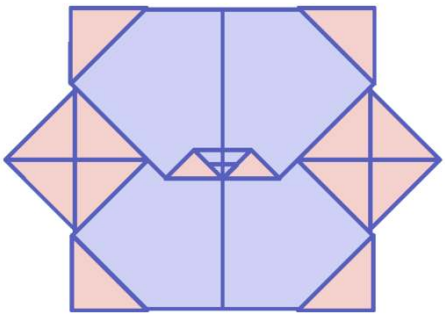
Repeat on the other side

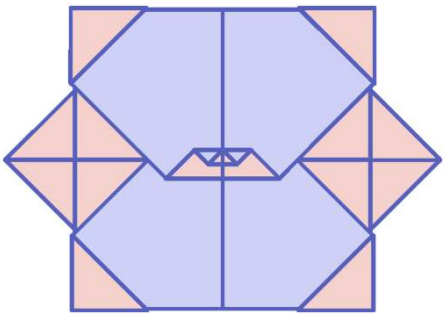


Return your paper









The end

F Statistiques

Cette section décrit quelques statistiques collectées au cours du projet, principalement sur le dépôt git.

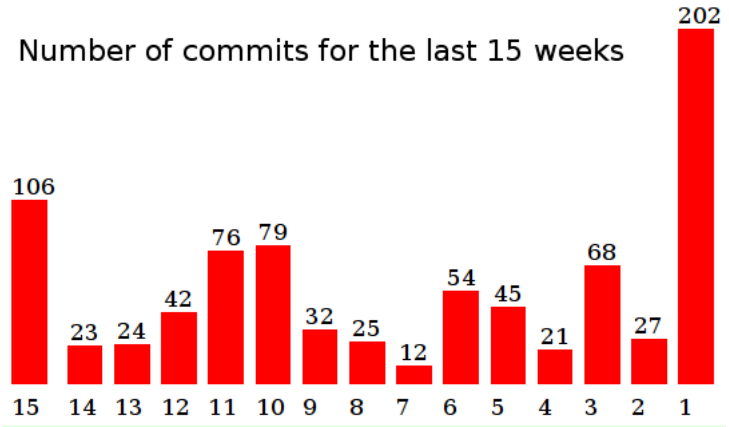


FIGURE 25 – Nombre de commits par semaine (la semaine 0 étant la semaine de la présentation de projet)

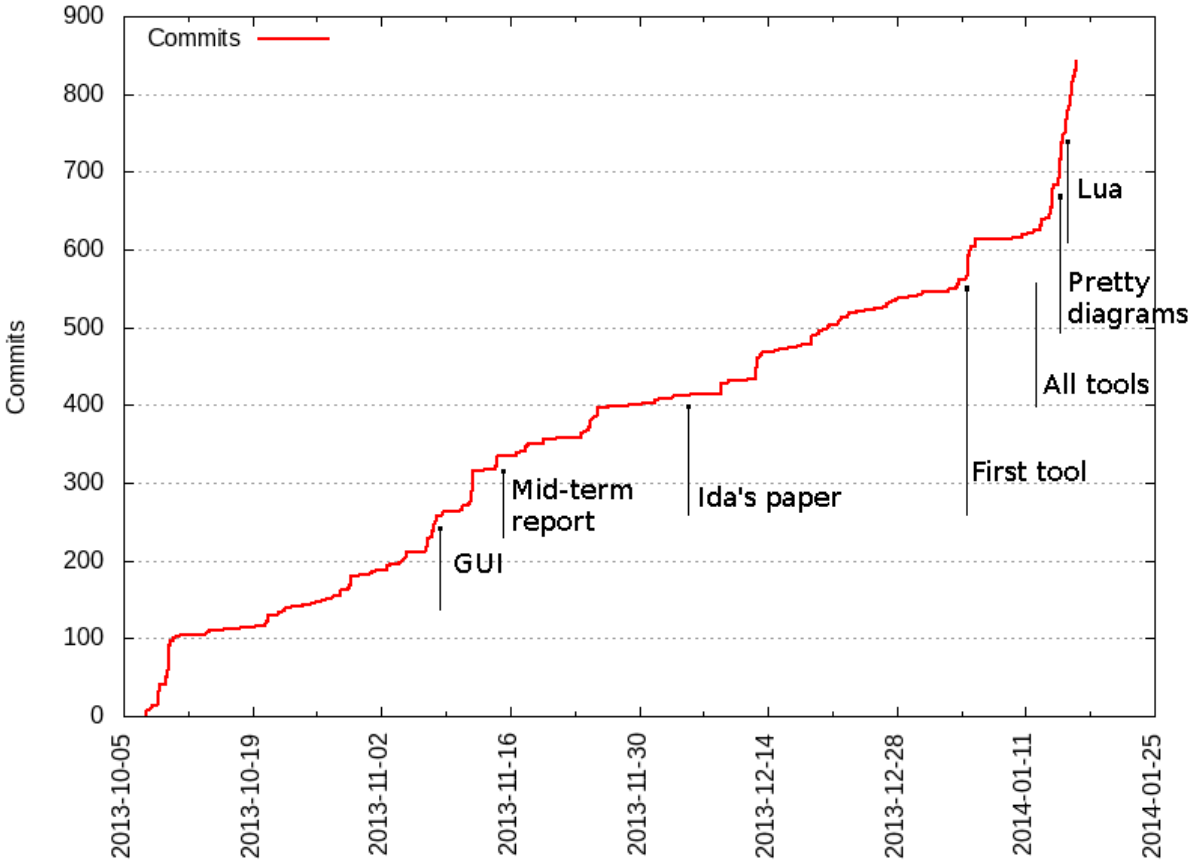


FIGURE 26 – Nombre de commits total en fonction du temps, avec les événements importants

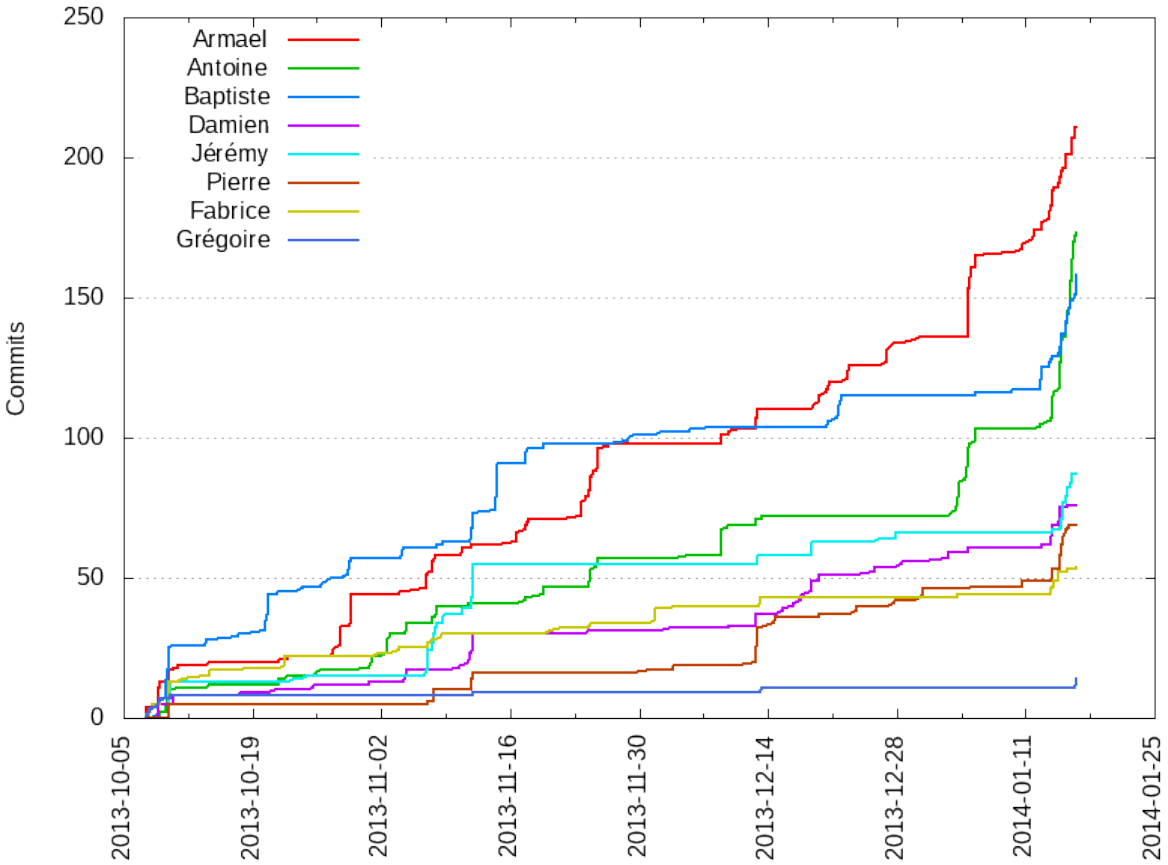


FIGURE 27 – Nombre de commits par membre du projet