École doctorale : Sciences et Ingénierie des Systèmes, Mathématiques, Informatique

Développement d'un langage de programmation dédié à la modélisation géométrique à base topologique, application à la reconstruction de modèles géologiques 3D

THÈSE

présentée et soutenue publiquement le 17 janvier 2019

pour l'obtention du grade de

Docteur de l'Université de Poitiers

(mention informatique)

(Faculté des Sciences Fondamentales et Appliquées)

(Diplôme National - Arrêté du 25 mai 2016)

par

Valentin Gauthier

Composition du jury

Rapporteurs :	Pascal Schreck	Pr. ICube, *Université de Strasbourg*
	Guillaume Caumon	Pr. ENSG-GeoRessources, *Université de Lorraine*
Examinateur :	Jean-Luc Mari	Pr. LIS, *Université d'Aix-Marseille*
Directeur de thèse :	Pascal Lienhardt	Pr. XLim, *Université de Poitiers*
Encadrants :	Agnès Arnould	M.d.C. XLim, *Université de Poitiers*
	Hakim Belhaouari	M.d.C XLim, *Université de Poitiers*





Sommaire

Introduction **vii**

Partie I Modélisation Géométrique à base topologique **1**

1 État de l'art **3**

1.1 Modèles à base topologique . . . 3

1.1.1 Introduction . . . 3

1.1.2 Cartes généralisées . . . 5

1.2 Langages et outils de modélisation géométrique . . . 10

1.2.1 Modeleurs géométriques . . . 10

1.2.2 MOKA . . . 11

1.2.3 CGOGN . . . 12

1.2.4 Computational Geometry Algorithms Library (CGAL) . . . 13

1.2.5 Outils de modélisation possédant un langage spécifique . . . 14

1.2.6 L-Systèmes . . . 16

1.3 Jerboa . . . 19

1.3.1 Fondements théoriques . . . 19

1.3.2 Transformation de la structure topologique . . . 21

1.3.3 Transformation des plongements . . . 32

1.3.4 Condition de préservation de la cohérence . . . 34

1.3.5 Architecture et fonctionnement . . . 39

1.3.6 Bilan . . . 44

2 Extension du langage Jerboa **47**

2.1 Cycle d'exécution des règles . . . 47

2.2 Langage de calcul . . . 49

2.3 Langage de *script* . . . 57

3 Mise en œuvre **65**

3.1 Éditeur et vérification . . . 65

3.1.1 Un nouvel éditeur de modeleur . . . 66

3.1.2 Vérifications automatiques . . . 70

3.2 Génération de code . . . 71

3.2.1 Construction de l'arbre syntaxique . . . 72

3.2.2 Analyse sémantique . . . 73

3.2.3 Traduction dans le langage cible . . . 76

3.3 Composition de règles . . . 79

3.4 Spécialisation du moteur d'application . . . 86

3.5 Gestion mémoire . . . 88

4 Résultats **89**

4.1 Étude de performance Java/C++ . . . 89

4.1.1 Subdivision de Loop . . . 90

4.1.2 Éponge de Menger . . . 92

4.1.3 Dual 3D . . . 93

4.2 Étude de performance des compositions de règle . . . 94

4.3 Étude de performance de la spécialisation de moteur . . . 97

5 Perspectives d'évolution **99**

5.1 Évolution du langage . . . 99

5.1.1 Opérateur de choix avec historique . . . 99

5.1.2 Lambda expressions . . . 102

5.1.3 Filtrage des *Scripts* . . . 103

5.2 Évolution de l'éditeur et des vérifications . . . 105

5.2.1 Composition de règles . . . 105

5.2.2 Comutativité et associativité des opérateurs et fonctions . . . 112

5.3 Évolution de la génération du code . . . 113

5.3.1 Compilation et optimisation . . . 113

5.3.2 Génération dans d'autres langages . . . 114

5.4 Évolution de la bibliothèque d'exécution . . . 114

5.4.1 Invariants de modeleur . . . 114

5.4.2 Parallélisation et gain de performance . . . 114

5.4.3 Optimisation des collectes . . . 115

5.4.4 Optimisation des moteurs . . . 116

Partie II Modélisation du sous-sol (réservoirs) 119

6 État de l'art 121
6.1 Besoin . . . 122
6.2 Approches . . . 127
6.2.1 Maillage de l'espace 3D . . . 127
6.2.2 Espace de dépôt . . . 130

7 Méthode proposée 133
7.1 Modélisation structurale 2D . . . 133
7.2 Objets et définition des plongements . . . 136
7.3 Étapes de l'algorithme de maillage 3D . . . 138
7.3.1 Maillage des horizons . . . 138
7.3.2 Maillage des failles . . . 145
7.3.3 Création des piliers 3D . . . 147
7.3.4 Extraction des blocs avec une représentation par les bords . . . 151
7.3.5 Découpe stratigraphique des unités . . . 153

8 Réalisation 157
8.1 ResQML et données réelles . . . 158
8.2 Résultats . . . 160
8.2.1 Re-maillage des surfaces 2D . . . 160
8.2.2 Maillage des failles . . . 161
8.2.3 Construction du maillage 3D . . . 162
8.3 Perspectives . . . 163

Conclusion générale 167

Bibliographie 169

Annexes 175

A Grammaire BNF du langage utilisé pour les scripts de règle et les expressions de plongement 175

Remerciements

Je tiens en premier lieu à remercier mon directeur de thèse Pascal Lienhardt pour sa rigueur, ses conseils et ses relectures. Je tiens également à remercier mes encadrants Agnès Arnould et Hakim Belhaouari pour leur investissement et leurs conseils, notamment pour les réflexions que nous avons eu sur nos travaux et pour leurs relectures de ce document. Ils m'ont toujours soutenu et ont pris énormément de temps, notamment pendant leurs vacances et leurs weekends, pour m'aider lorsque j'en avais besoin.

Je remercie également Jean-François Rainaud qui nous a fait confiance pour cette collaboration et pour son aide sur les problématiques géologiques. Je remercie également Michel Perrin qui nous a apporté les connaissances géologiques nécessaires à nos travaux et ses conseils pour l'écriture de ce document.

Je remercie Françoise, Sylvie, Sophie et Caroline pour leur efficacité et leurs compétences pour résoudre les problèmes administratifs.

Je tiens a remercier les membre de l'équipe Informatique Graphique de l'institut XLim avec qui j'ai pu avoir des discussions intéressantes sur d'autres problématiques. Particulièrement ceux avec qui j'ai eu la chance de travailler pendant mes heures d'enseignement et qui ont fait leur maximum pour me laisser le plus de temps possible pour travailler en priorité sur ma thèse.

J'ai eu la chance de partager mon bureau avec des personnes avec qui j'ai pu avoir des moments de réflexion, mais également faire des parties de ping-pong, d'échec, de challenge de programmation ou encore de jeux vidéos. Je remercie donc Mathias Brousset et Mahmoud Omidvar, sans qui mes années de thèse n'auraient pas été aussi agréables. Je remercie également mes nouveaux voisins de bureau, Wassim Rharbaoui et Simon Courtin, pour m'avoir supporté pendant ma période de rédaction. Je remercie également Maxime Maria pour sa bonne humeur et son investissement trop peu souvent apprécié à sa juste valeur eu sein de l'association des doctorants. Je remercie également tous les autres doctorants qui ont participé à cette association et avec qui j'ai eu le plaisir de travailler.

Il est parfois des personnes qu'on ne connait pas personnellement mais qui comptent malgré tout pour ce qu'elle nous apportent. Je remercie donc Richard Monvoisin, que j'ai rencontré lors d'une formation doctorale, pour ses conférences passionnantes. Sans lui je n'aurais jamais découvert la zététique et toutes les personnes qui produisent du contenus scientifique ou de vulgarisation à ce sujet. Je remercie les membres de l'ADC : le Captain, LTP, Manox et Kwakos, pour leur bonne humeur, leur univers, mais surtout pour leur ponctualité. Ils ont contribué à me faire garder le morale pendant ces quatre années, même s'ils ont réussi à me faire perdre foi en l'humanité.

Je remercie ma famille et mes amis pour leur soutien moral et pour leurs encouragements. Je remercie particulièrement Élodie qui a su me motiver et me soutenir dans les moments difficiles et pour sa patience quand je travaillais tard le soir et les weekends.

Introduction

Cette thèse s'inscrit dans le domaine de la modélisation géométrique. Ce domaine comprend l'ensemble des outils et des méthodes qui permettent de représenter et de manipuler virtuellement des objets réels ou virtuels. C'est un domaine très vaste qui a de nombreuses applications.

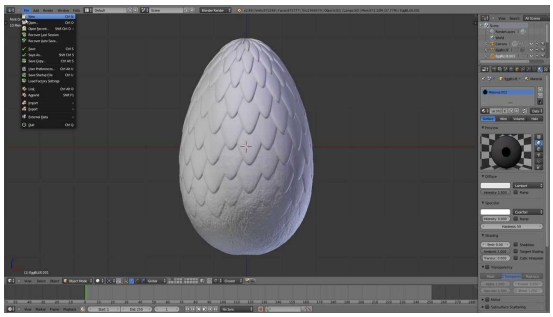
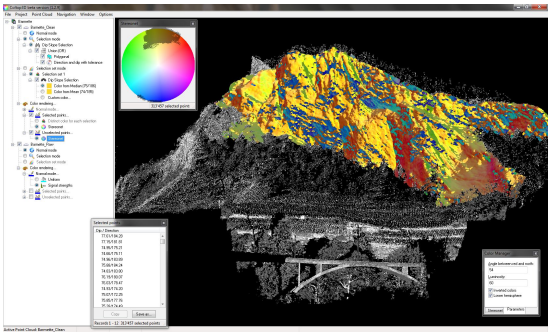
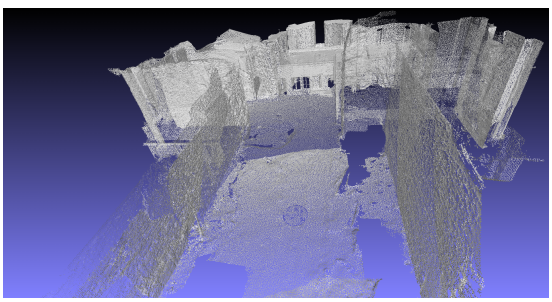


FIGURE 1 – Blender, logiciel de modélisation 3D

Le domaine du cinéma par exemple a beaucoup évolué ces dernières années grâce à l'outil informatique, notamment pour réaliser des effets spéciaux ou pour produire des films d'animation. Pour produire les objets géométriques utilisés dans les films, il existe des logiciels appelés modeleurs qui proposent des outils de création et de manipulation d'objets virtuels. Ces outils s'inscrivent dans le domaine de la Conception Assistée par Ordinateur (ou CAO). On peut par exemple citer Blender [Ble] qui permet de créer des objets très complexes à partir d'objets de bases (triangles, hexaèdres, *etc.*). Il inclut des opérations de transformation géométriques qui permettent de déformer les objets pour les modéliser selon les besoins. Une capture d'écran du logiciel est montrée en figure 1.



(a) Vue d'un nuage de points de sous-sol dans le logiciel d'analyse Coltop3D



(b) Nuage de points issu de mesures lasers

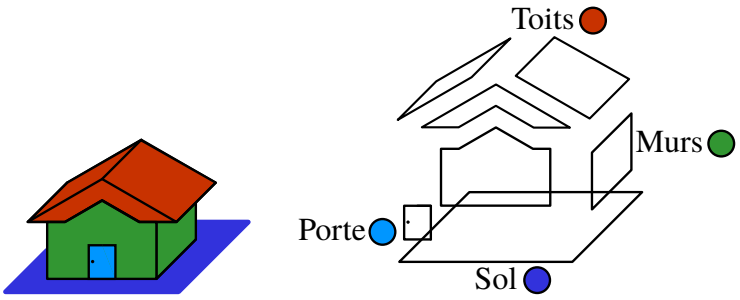
FIGURE 2 – Prises de mesures sous forme de nuages de points

Dans certains domaines comme la géologie ou l'architecture par exemple, les besoins sont assez différents du cinéma car on souhaite représenter précisément des objets concrets et non des objets créés de toute pièce. Dans ce cas, les objets que l'on souhaite représenter sont mesurés à l'aide d'appareils dédiés. Ces données de mesures sont ensuite transmises à un logiciel qui permet de lire et de travailler sur ces données, comme le logiciel Coltop3D [Ter] dont une prise d'écran est montrée en figure 2(a). Les données peuvent être obtenues par différentes techniques comme la profilométrie [TM83], ou la télémétrie-laser par exemple. Un exemple de données issue de prise de mesure laser [1] est montré en figure 2(b) et représente un bâtiment par un nuage de points.



FIGURE 3 – Modèle 3D d'un visage scanné par un téléphone portable par prise de photos successives sous différents angles

De nouvelles techniques se démocratisent également avec l'omniprésence des smartphones, ou des drones, et les photos qu'ils permettent de prendre. C'est le cas par exemple du logiciel Générateur 3D [Son] de Sony, ou l'application téléphone Snapsculpt [Sna] qui permettent de scanner en 3D un objet ou une personne en tournant autour, et en prenant des photos sous différents angles. Ces logiciels permettent ensuite de réaliser une impression 3D de l'objet scanné, comme le visage de la figure 3 imprimé par Sony. À plus grande échelle, le logiciel ContextCapture [Acu] par exemple, permet de reconstruire des modèles 3D à partir de photos de satellites ou de drones.



(a) Objets (b) Objet décomposé

FIGURE 4 – Modèle d'une maison

Dans les modèles 3D, il est souvent nécessaire d'identifier et d'attribuer des propriétés particulières à différents éléments d'un objet ou d'une scène. En architecture par exemple, on identifie les différents objets d'une scène par leur type. La figure 4(a) montre une maison qui est composée de différents éléments (murs, toit, *etc.*). Sur la figure, le sol est en bleu, la porte en bleu ciel, les murs en vert et le toit en rouge. On peut ajouter à chaque élément une couleur et une étiquette

1. `https://fredericduvivier.wordpress.com/2012/03/05/modelisation-3d-en-archeologie-techniques-nouvelles/`

d'identification comme illustré sur la figure 4(b). Ces informations sont ajoutées dans l'objet et permettent de définir une sémantique propre à l'objet. On peut ainsi définir des contraintes de cohérence en interdisant par exemple de créer une fenêtre sur une porte ou sur un sol. En géologie on identifie différents éléments comme les couches géologiques, pour leur attribuer des informations comme le type de roche.

Dans le cadre de cette thèse, nous nous sommes concentrés sur des problématiques de modélisation d'un sous-sol géologique afin de produire une représentation virtuelle du sous-sol. Notre objectif est de produire un modèle qui permet de réaliser des simulations d'écoulement de fluide et qui conserve toutes les informations sémantiques du modèle. Les simulations de fluide dans un modèle géométrique sont très complexes à réaliser et la qualité de leurs résultats dépend énormément de la qualité du modèle et des informations qu'il porte. Ainsi nous avons dû mettre en œuvre une méthode de maillage du sous-sol en nous appuyant sur différentes contraintes. Nous nous sommes appuyé sur des contraintes géologiques qui déterminent si un modèle est cohérent en fonction du réalisme des déformations du sous-sol. Nous avons également dû suivre des contraintes de bonne formation du maillage géométrique afin de produire un modèle exploitable pour des simulations de fluide.

Pour être en mesure de produire un modèle géométrique qui suit ces différentes contraintes, nous nous sommes appuyés sur un modèle mathématique qui permet de représenter à la fois la forme des objets mais également les informations sémantiques du modèle géologique. Nous nous appuyé sur la plate-forme Jerboa [BALGB14] qui permet de réaliser des opérations géométriques dédiées. Les opérations sont définies graphiquement par des règles de transformations de graphe. L'avantage de la plate-forme Jerboa est qu'elle inclut des vérifications automatiques de préservation de la cohérence des modèles géométriques, qui peuvent être enrichies par des contraintes métier. Elle inclut également un module de visualisation 3D qui permet de visualiser les objets et de les manipuler avec les opérations dédiées que nous avons définies dans l'outil. Jusqu'alors, Jerboa permettait de réaliser des opérations définies par une seule règle. Nous avons donc contribué à augmenter les possibilités de manipulation de la plate-forme en développant un langage dédié qui permet de réaliser des opérations plus complexes. Avec ce langage nous avons également pu ajouter de nouvelles vérifications automatiques de préservation de la cohérence géométrique des objets. En complément, nous avons modifié le processus d'application des opérations dans Jerboa afin d'augmenter les possibilités de contrôle du processus d'application par l'utilisateur.

Première partie

Modélisation Géométrique à base topologique

Chapitre 1

État de l'art

1.1 Modèles à base topologique

1.1.1 Introduction

Dans le cadre de la modélisation géométrique, la topologie d'un objet est la description des propriétés qui le caractérisent en faisant abstraction de sa forme. Elle permet de décrire la structure d'un objet mais ne donne aucune information géométrique ou physique sur ce dernier. Ces informations dissociées de la topologie sont appelées plongements. Ces notions seront abordées plus tard dans cette partie.

Les modèles topologiques permettent de représenter des subdivisions de l'espace en sous-ensembles appelés cellules topologiques. Chaque cellule a une dimension topologique $i \in \{0..,n\}$ (avec n la dimension du modèle topologique), on parle alors de $i-cellule$. Les sommets (respectivement les arêtes, les faces, *etc.*) sont des cellules topologiques de dimension 0 (respectivement 1, 2, *etc.*). Deux cellules sont incidentes si elles sont de dimension différentes et que l'une appartient au bord de l'autre. Par exemple, sur la figure 1.1, l'arête c et la face F_1 sont incidentes, de même que le sommet A et la face $F1$. Plus précisément, un modèle topologique représente les cellules d'un objet ainsi que leurs relations d'incidence. Deux $i-cellules$ sont dites adjacentes si elles sont de même dimension et incidentes à une même cellule de dimension $i-1$. Par exemple, sur la figure 1.1, les faces $F1$ et $F2$ sont toutes les deux incidentes à l'arête c, elles sont donc adjacentes.

Il existe différents modèles topologiques plus ou moins complexes permettant de représenter des classes variées de subdivisions. On peut classer ces modèles selon différents critères, par exemple, ceux qui permettent de représenter uniquement des cellules régulières et ceux qui permettent de représenter des cellules quelconques. On peut également distinguer les modèles sans multi-incidence de ceux avec multi-incidence. Par exemple, les complexes simpliciaux abstraits

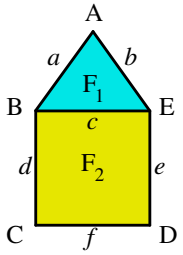
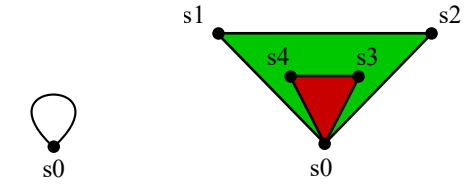


FIGURE 1.1 – Objet



(a) Arête (b) Deux faces, dont une (verte) est multi-incidente à $s0$

FIGURE 1.2 – Cas de multi-incidence

[Spa66] représentent les objets par des simplexes. Un simplexe de dimension i est un ensemble de $i+1$ sommets. Les relations de voisinage entre deux simplexes sont représentées comme l'intersection de ceux-ci. Prenons l'exemple d'un triangle comprenant 3 sommets : $s1, s2, s3$. Le modèle contient ainsi 3 simplexes de dimension 0 : $\{s1\}$, $\{s2\}$, $\{s3\}$; 3 simplexes de dimension 1 : $\{s1, s2\}$, $\{s2, s3\}$, $\{s3, s1\}$; et un simplexe de dimension 2 : $\{s1, s2, s3\}$. L'intersection entre deux simplexes de dimension 1 est un sommet partagé par ces dernières : $\{s1, s2\} \cap \{s2, s3\} = \{s2\}$. Si l'intersection est vide, c'est qu'il n'existe pas de relation de voisinage entre ces deux simplexes.

Les complexes simpliciaux abstraits permettent de représenter uniquement des objets dont les cellules sont des triangles mais pas de représenter de multi-incidence, comme illustré en figure 1.2(a). En effet, le simplexe $\{s0, s0\}$ est équivalent à $\{s0\}$. L'objet de la figure 1.2(b) ne peut également pas être représenté par ce modèle : la face verte n'est pas un triangle et elle est multi-incidente au sommet $s0$. Les ensemble simpliciaux [May67, Lan95] quant à eux sont une généralisation des complexes simpliciaux abstrait qui permettent de prendre en compte la multi-incidence.

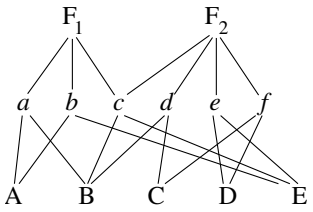


FIGURE 1.3 – Graphe d'incidence de l'objet de la figure 1.1

Les graphes d'incidence permettent de représenter des cellules quelconques mais ne permettent pas de représenter de multi-incidence. Ils permettent de représenter les objets sous forme de graphes dont les nœuds correspondent aux cellules et les arcs correspondent aux relations d'incidence. Par exemple, sur la figure 1.3, on observe dans le graphe que les faces F_1 et F_2 sont liées à leurs arêtes respectives a, b, c et c, d, e, f, elles-mêmes liées à leurs sommets. On peut extraire de cette structure des informations implicites de voisinage : dans le graphe, les faces F_1 et F_2 sont toutes les deux liées à l'arête c, donc adjacentes.

Les modèles de type carte combinatoire [Vin83] quant à eux permettent de représenter des objets subdivisés en cellules quelconques et autorisent la multi-incidence. Ce type comprend par exemple la famille des "edges" : *half-edge* ou demi-arêtes [Wei85], *winged-edge* ou arêtes ailées [Bau75], *radial edge* [Wei88]... Le modèle des demi-arêtes, par exemple, repose sur les relations d'adjacence entre les arêtes pour représenter les objets. On peut associer à chaque demi-arête sa suivante ou précédente (au sein d'une même face), ou sa demi-arête opposée (sur la face voisine). Pour nos travaux, nous utilisons les cartes généralisées qui font partie des cartes combinatoires.

1.1.2 Cartes généralisées

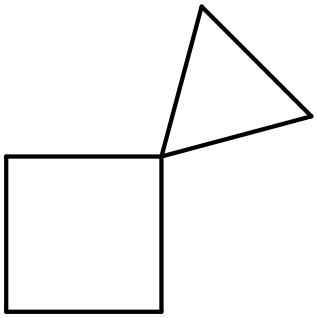
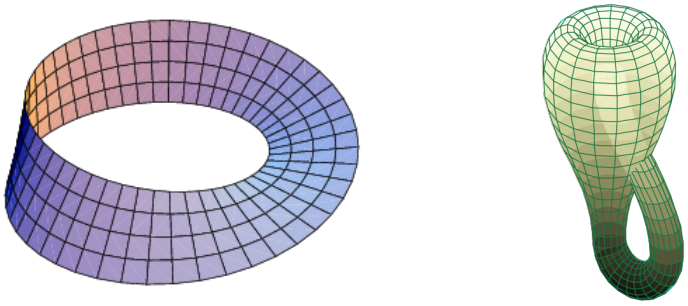


FIGURE 1.4 – Non variété

Les cartes généralisées (ou G-cartes), définies par [Lie94], permettent de représenter des quasi-variétés. Une variété désigne un objet dont le voisinage est en tout point équivalent à une boule. Un objet est une quasi-variété si chaque cellule de dimension i est adjacente à une autre cellule de même dimension le long d'une cellule de dimension $i-1$. Intuitivement, cela implique par exemple que deux faces voisines le sont le long d'une arête, ce qui n'est pas le cas de l'objet de la figure 1.4.



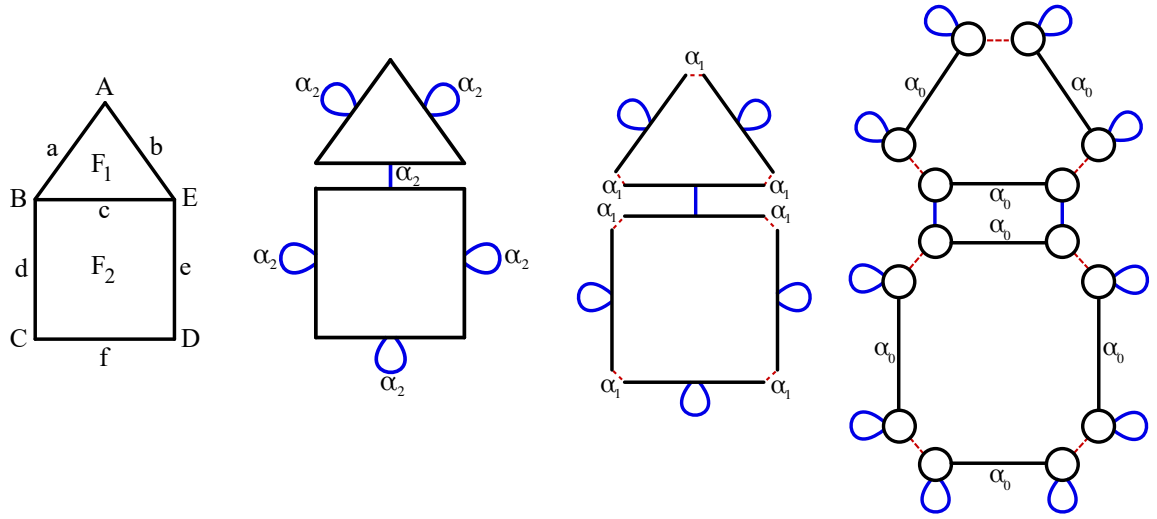
(a) Ruban de Möbius

(b) Bouteille de Klein

FIGURE 1.5 – Objets non orientables

Les cartes généralisées permettent également de représenter des objets ouverts ou fermés, orientables ou non. Contrairement aux objets orientables, les objets non orientables ne séparent pas l'espace en plusieurs parties distinctes : un recto/verso pour une surface ouverte ou un intérieur/extérieur pour une surface fermée. C'est par exemple le cas du ruban de Möebius (cf. figure 1.5(a)), dont le bord du ruban est formé par une unique courbe. Un autre exemple connu est celui de la bouteille de Klein (cf. figure 1.5(b)), dont le goulot traverse la bouteille si elle est plongée en 3D, ce qui fait que l'intérieur de celle-ci est également l'extérieur.

Une G-carte peut intuitivement être présentée de la manière suivante. Étant donnée un objet, par exemple celui de la figure 1.6(a), on décompose successivement cet objet en cellules de dimensions décroissantes. La décomposition commence par les cellules de dimension 2, ici un carré et un triangle figure 1.6(b) ; pour se souvenir que les faces étaient adjacentes, on les relie par une liaison α_2. Pour être homogène, les arêtes du bord sont liées avec elles-mêmes par α_2. Ces liaisons sont représentées par des boucles sur la figure. La décomposition se poursuit en dimension inférieure (cf. figure 1.6(c)). On décompose chaque cellule de dimension 2 en cellules



(a) Objet géométrique

(b) "Individualisation" des faces (par éclatement des arêtes)

(c) "Individualisation" des arêtes (par éclatement des sommets)

(d) "Individualisation" des sommets

FIGURE 1.6 – Décomposition d'un objet géométrique 2D en une 2-G-carte

de dimension 1 ; pour se souvenir des relations d'adjacence, on relie ces arêtes par une liaison α_1. Finalement, on décompose les cellules de dimension 1 en cellules de dimension 0 que l'on relie par α_0. Les "sommets" obtenus à la fin de la décomposition sont appelés des brins. Les "liaisons" sont reportées sur ces brins. On obtient ainsi la représentation de l'objet initial par la 2-G-carte présentée en figure 1.6(d). Notons que les liaisons sont non orientées. Cela correspond au fait qu'elles représentent des relations d'adjacence. Les liaisons vérifient d'autres propriétés. Par exemple, sur la figure 1.6(b), α_2 relie deux arêtes ; chacune de ces arêtes va être éclatée en deux brins liés par α_0 : cela induit le fait que les liaisons α_0 et α_2 forment des cycles. Plus formellement, nous définissons les cartes généralisés par des graphes topologiques (cf. Définition 1.1.1 et Définition 1.1.2).

Définition 1.1.1 (Graphe topologique) *Soit une dimension topologique $n \geq 0$. Un graphe topologique de dimension n est un graphe étiqueté $G = (V, E, s, t, \alpha)$ tel que :*

- *V est un ensemble de brins ;*
- *E est un ensemble d'arcs ;*
- *$s : E \to V$ et $t : E \to V$ sont deux fonctions respectivement source et cible ;*
- *$\alpha : E \to [0, n]$ est une fonction d'étiquetage des arcs.*

Définition 1.1.2 (G-carte) *Une G-carte de dimension n est un graphe topologique G de dimension n qui vérifie les contraintes de cohérence suivantes :*

- ***non-orientation** : G est non orienté, formellement pour tout $e \in E$ il existe un arc inverse $e' \in E$ tel que $s(e') = t(e)$, $t(e') = s(e)$ et $\alpha(e') = \alpha(e)$;*
- ***arcs adjacents** : pour tout $i \in [0, n]$ et pour tout $v \in V$ il existe exactement un arc $e \in E$ tel que $\alpha(e) = i$ et $s(e) = v$;*
- ***cycle** : pour tout $i, j \in [0, n]^2$ tel que $i+2 \leq j$, pour tout $v \in V$ il existe un cycle $\alpha_i\alpha_j\alpha_i\alpha_j$ de source v.*

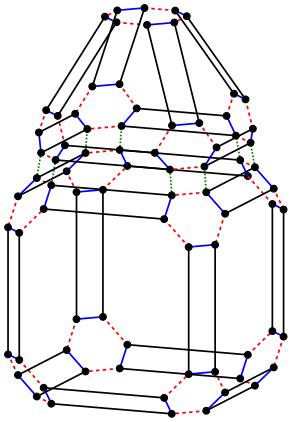
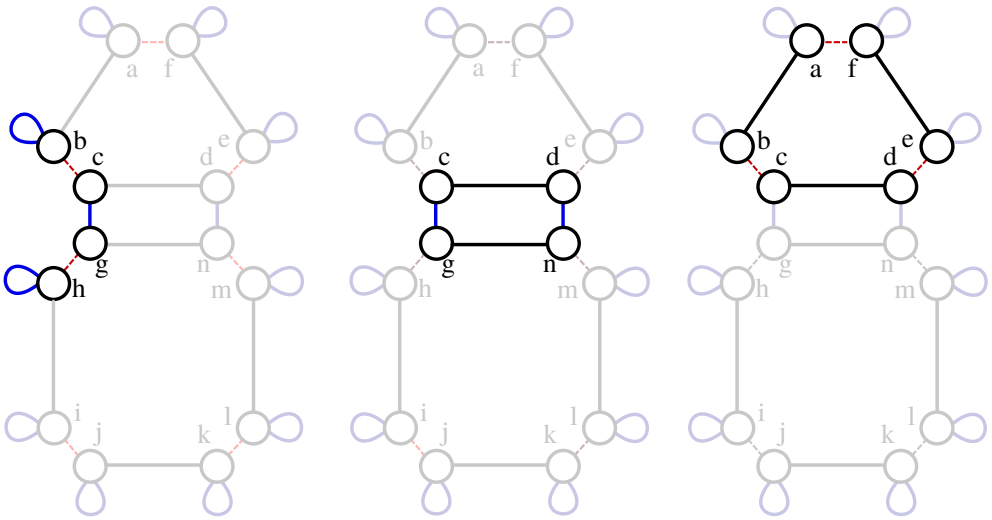


FIGURE 1.7 – Représentation 3D d'un objet par une 3-G-carte

Une représentation d'un objet 3D par une 3-G-carte est illustrée en figure 1.7.

Dans les G-cartes, les cellules topologiques des objets sont représentées implicitement. Elles sont définies par des sous-graphes appelés orbites.

Définition 1.1.3 (Orbite) *Pour o un sous-ensemble de $\{\alpha_0, \dots \alpha_n\}$ et v un nœud d'un graphe G de dimension n, l'orbite $\langle o \rangle(v)$ est le sous-graphe de G contenant v et les nœuds de G atteignables par des arcs étiquetés sur o, ainsi que ces arcs. On dit que $\langle o \rangle(v)$ est de type $\langle o \rangle$, c'est une $\langle o \rangle$-orbite.*



(a) Sommet : $\langle \alpha_1, \alpha_2 \rangle(c)$ (b) Arête : $\langle \alpha_0, \alpha_2 \rangle(c)$ (c) Face : $\langle \alpha_0, \alpha_1 \rangle(c)$

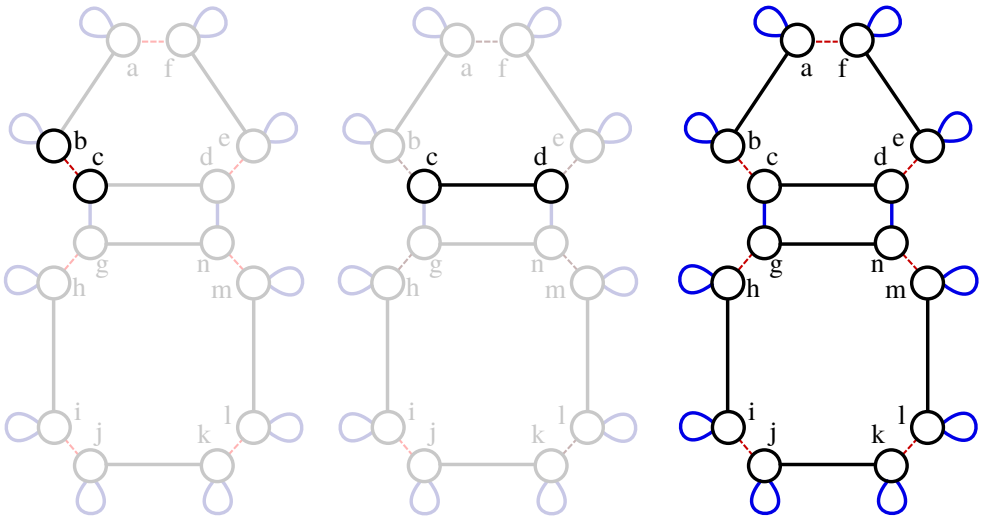
FIGURE 1.8 – Cellules topologiques

Par exemple sur la figure 1.8(a), la 0-cellule incidente au brin c est le sous-graphe contenant tous les brins accessibles depuis c en parcourant les arcs α_1 (en rouge) et α_2 (en bleu). Ce sous-graphe contient les brins b, c, g, et h, ainsi que les liaisons α_1 et α_2 qui les relient (boucles comprises). Cette orbite est notée $\langle \alpha_1, \alpha_2 \rangle(c)$. La liste de liaisons entre chevrons définie le type d'orbite et le brin entre parenthèse est un brin incident à cette orbite. Plus généralement, une i-cellule est définie par la Définition 1.1.4.

Définition 1.1.4 (i-cellule) *Soit G une carte généralisé de dimension n et b un brin de celle-ci. La i-cellule, avec $i \in \{0, \ldots, n\}$ incidente à b est :*

- *si $i = 0$: $\langle \alpha_1, \ldots, \alpha_n \rangle$;*
- *sinon si $i = n$: $\langle \alpha_0, \ldots, \alpha_{n-1} \rangle$;*
- *sinon : $\langle \alpha_0, \ldots, \alpha_{i-1}, \alpha_{i+1} \ldots, \alpha_n \rangle$.*

Ainsi, la 1-cellule incidente au brin c, est le sous-graphe contenant tous les brins accessibles depuis c en parcourant les arcs α_0 et α_2. Elle est illustrée en figure 1.8(b) et contient les brins c, d, n et g, ainsi que les liaisons α_0 et α_2 qui les relient. De même, la 2-cellule incidente au brin c est illustrée en figure 1.8(c), contient les brins c, d, e, f, a et b, ainsi que les liaisons α_0 et α_1 qui les relient.



(a) Portion de sommet : $\langle \alpha_1 \rangle(c)$ (b) Demi arête : $\langle \alpha_0 \rangle(c)$ (c) Composante connexe : $\langle \alpha_0, \alpha_1, \alpha_2 \rangle(c)$

FIGURE 1.9 – Orbites quelconques

La notion d'orbite permet de généraliser la notion usuelle des cellules topologiques. Ainsi, toutes les cellules topologiques sont des orbites. De plus, il existe des orbites qui ne sont pas des cellules topologiques. Par exemple, le coin de la face incidente au sommet B, incident au brin c est représenté par l'orbite $\langle \alpha_1 \rangle(c)$, illustrée en figure 1.9(a). De même, la demi-arête incidente à c se définit par l'orbite $\langle \alpha_0 \rangle(c)$ que l'on peut observer sur la figure 1.9(b). Enfin, la composante connexe incidente à c, illustrée en figure 1.9(c), est représentée par l'orbite $\langle \alpha_0, \alpha_1, \alpha_2 \rangle(c)$.

Pour modéliser complètement des objets, la structure topologique doit être complétée par des informations géométriques et physiques appelées plongements. Le nombre et les types d'informations ajoutées au modèle topologique dépendent de l'application métier voulue.

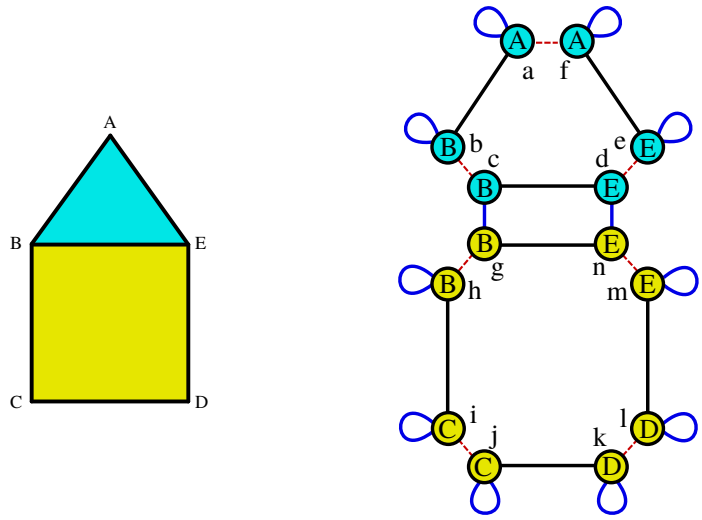
Prenons le cas de la géologie, où il est nécessaire de définir des propriétés géo-physiques sur les objets. Certaines propriétés comme la porosité des roches, leur composition, *etc.* sont nécessaires pour définir des opérations adaptées. Pour ce faire, on peut associer une étiquette à chaque objet du modèle, pour leur donner des propriétés géo-physiques, et ainsi compléter la topologie avec des informations sémantiques liées à l'application.

Dans une carte généralisée, un plongement décrit l'association d'un type d'information à un type d'orbite. Ici nous appelons type d'information un type abstrait connu par son nom et

ses opérations. Par exemple, nous avons défini un plongement `point` qui représente l'association d'une position géométrique de type `Point2D` aux orbites sommet, et un plongement `couleur` de type `CouleurRGB` qui représente l'association d'une couleur aux orbites face. Sur notre exemple, nous avons attribué la couleur jaune au carré, et le bleu au triangle, comme illustré sur la figure 1.10(a). Nous avons également attribué une position géométrique (les points A, B, C, D, et E sont représentés par des coordonnées cartésiennes) à chaque sommet. Formellement, un plongement se défini comme dans la Définition 1.1.5.

Définition 1.1.5 (Plongement) *Soit $n \geq 0$ une dimension, $\langle o \rangle$ un type d'orbite et τ un type d'information.*

Le plongement π et de profil $\langle o \rangle \to \tau$ est une fonction des orbites de type $\langle o \rangle$ vers τ noté $\pi : \langle o \rangle \to \tau$.



(a) Objet géométrique (b) G-carte plongée

FIGURE 1.10 – Objet plongé

Ainsi, nos plongements sont définis comme suit : `point` : $\langle \alpha_1, \alpha_2 \rangle \to$ `Point2D` et `couleur` : $\langle \alpha_0, \alpha_1 \rangle \to$ `CouleurRGB`. Ici `point` associe les instances d'orbites de type face à des plongement de type `Point2D` (A, B, C, D, E). De même pour `couleur` associe les brins des faces de l'objet aux valeurs de plongement de type `CouleurRGB` : jaune et bleu. La figure 1.10 montre l'association de la couleur et de la position géométrique sur chaque brin de la G-carte.

Formellement, la notion de carte généralisée est précisée à la Définition 1.1.6

Définition 1.1.6 (G-carte plongée) *Soit une dimension topologique $n \geq 0$ et $\Pi = (\pi : \langle o \rangle \to \tau)$ une famille de plongements de dimension n. Une carte généralisée plongée de dimension n est une carte généralisée G de dimension n comme défini en Définition 1.1.2 étiquetée $G = (V, E, s, t, \alpha, \Pi)$ qui satisfait la condition suivante :*

- *pour tout plongement $(\pi : \langle o \rangle \to \tau) \in \Pi$ et pour tous brins $(v, v') \in V^2$ tels que $v' \in \langle o \rangle(v), \pi(v) = \pi(v')$.*

1.2 Langages et outils de modélisation géométrique

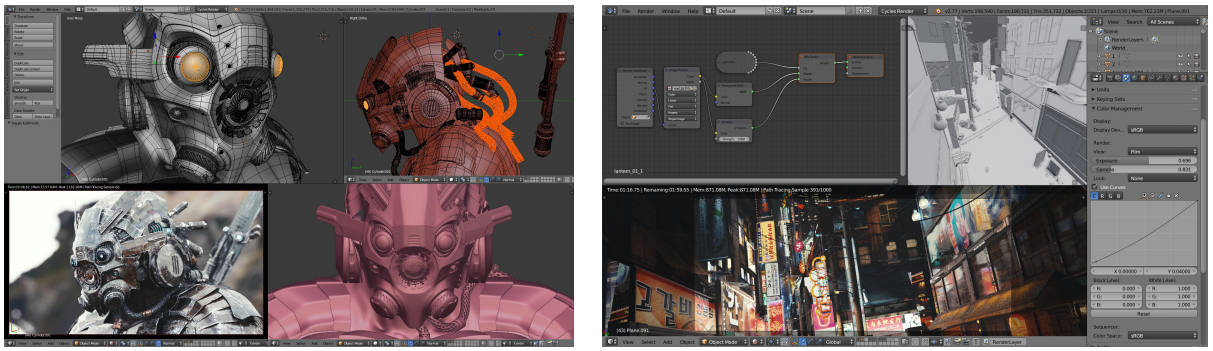
Il existe de très nombreux outils de modélisation géométrique. Parmi les plus connus on nommera Blender[Ble], CATIA[Sys], AUTODESK[Auta, Autb, Autc], *etc.* Nombre d'outils sont développés et utilisés par des entreprises privées.

Les structures internes de représentation des objets sont très diverses. Cependant, la plupart des outils ne fournissent pas leur code source, et il n'est pas toujours possible de connaître les modèles utilisés pour représenter les objets. Afin d'être en mesure de comparer correctement notre outil à d'autres, nous nous sommes concentrés sur l'étude de modeleurs utilisant la topologie et fournissant leur code source.

De plus, nos travaux ont porté sur le développement d'un langage dédié à la création de modeleur et de leurs opérations de manipulation des objets, ainsi que sur l'ajout de vérifications automatiques de préservation de la cohérence des objets. Nous nous sommes donc intéressés aux possibilités offertes par d'autres modeleurs pour développer de nouvelles opérations. Certains le permettent avec un langage de programmation généraliste (*e.g.* C++, Python, *etc.*). D'autres, ont développé leur propre langage dédié qui permet de définir de nouvelles opérations dans les limites des possibilités du langage. Nous présentons ici quelques exemples d'outils pour illustrer ces différents cas.

1.2.1 Modeleurs géométriques

Il existe énormément d'outils qui permettent de réaliser de façon très efficace des opérations spécifiques aux besoins des entreprises qui les manipulent.

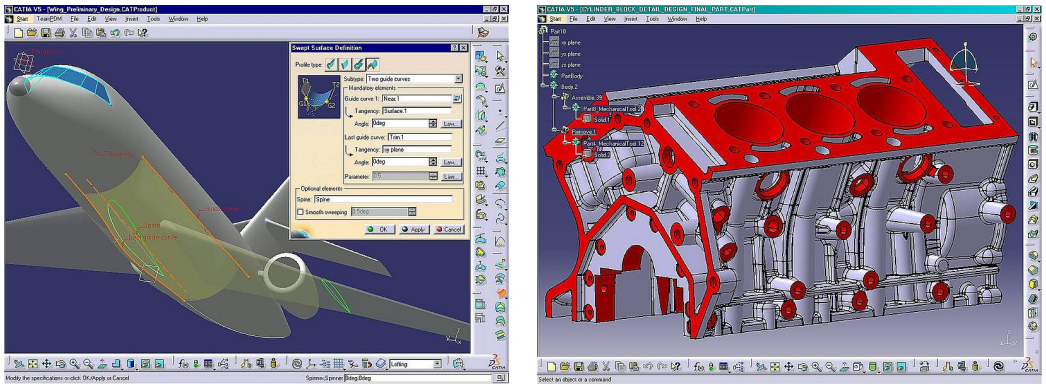


(a) (b)

FIGURE 1.11 – Blender

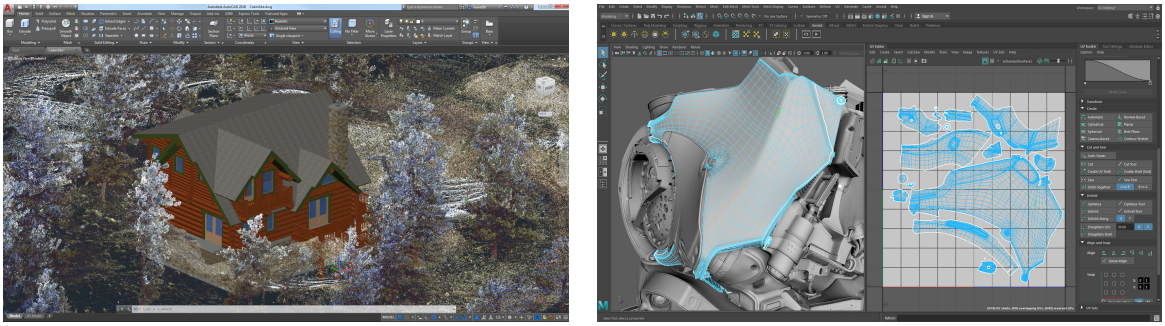
Blender [Ble] est un outil de modélisation gratuit qui permet de manipuler des objets 3D, comme illustré sur la figure 1.11(a). Il permet également de les animer et de réaliser des images ou des vidéos de rendu 3D (cf figure 1.11(b)). L'utilisation de *scripts* Python, permet d'étendre les fonctionnalités de l'outil. Dans un *script* il est possible d'accéder à toutes les fonctionnalités de l'outil et aux différents objets du modèle, ainsi qu'aux éléments de rendu (caméra, lumière ...).

Catia[Sys] est un outil de conception assistée par ordinateur (CAO). Il permet notamment de concevoir des pièces mécaniques et de tester leur fonctionnement par simulation. On peut voir sur la figure 1.12(a), un modèle d'avion et sur la figure 1.12(b), une pièce de moteur. Catia inclut un système de macros qui permet de définir des opérations dans les langages VBA ou VB.net par exemple.



(a) (b)

FIGURE 1.12 – Catia



(a) AutoCAD (b) Maya

FIGURE 1.13 – Autodesk

Autodesk propose plusieurs logiciels comme AutoCAD (cf. figure 1.13(a)) qui permet de faire de la conception 2D ou 3D et de réaliser des plans, des circuits électriques, *etc.* Maya (cf. figure 1.13(b)) est un de leurs logiciels qui permet de faire des rendus d'images de synthèse et est parfois utilisé dans le milieu du cinéma. Autodesk développe également un autre logiciel de modélisation appelé *Fusion 360*. L'intérêt de ce produit est qu'il existe une API dédiée qui permet de développer des modules qui peuvent s'intégrer au logiciel afin de créer de nouvelles opérations par exemple. Les modules ont accès aux informations de l'interface graphique et à la structure des objets du modèle. Ils peuvent être écrits sous forme de *scripts* Python ou en C++.

1.2.2 MOKA

Moka [VD] est une bibliothèque de modélisation basée sur les 3-G-cartes, développées en C++. Elle inclut également un module d'interface graphique (illustrée en figure 1.14) qui permet de manipuler les objets et leur appliquer différentes opérations. Moka fournit la structure d'une G-carte générique et une structure de 3-G-carte qui implémente cette interface. Cette dernière inclut comme plongement la position géométrique des sommets.

Le développement de nouvelles opérations se fait dans le langage de la bibliothèque, c'est à dire en C++. Pour ce faire, il est possible d'utiliser les fonctions de manipulation pré-définies pour les 3-G-cartes. Parmi ces fonctions, certaines mettent en place des vérifications de préser-

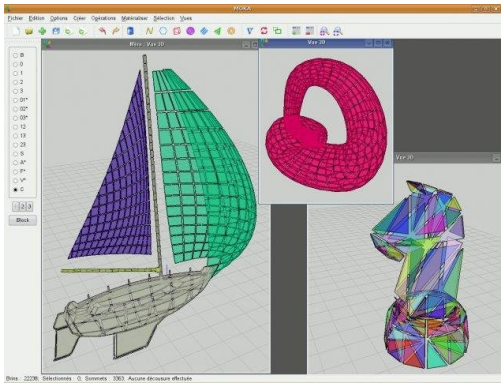


FIGURE 1.14 – Moka

	Nombre d'arête	`Sew()`	`LinkAlpha()`	Ratio
Moyenne	6	0,04	0,0035	11,4
	100	0,501	0,052	9,6
	999	20,124	0,2625	76,7

TABLE 1.1 – Comparaison des temps d'exécution de fonction de couture avec (`Sew()`) et sans (`LinkAlpha()`) vérification

vation de la cohérence des cartes généralisées. Par exemple, il existe des fonctions de "couture" (*sew*) qui permettent de lier deux brins, en liant automatiquement tous les autres brins nécessaires pour conserver les cycles des G-carte. Ces fonctions sont évidemment un peu plus lentes à l'exécution que d'autres qui ne font aucune vérification. Après avoir fait des tests avec une opération de triangulation, nous avons noté un ratio de différence jusqu'à environ 70 comme l'illustre la Table 1.1.

Moka présente également des classes de contrôleurs qui permettent notamment d'appliquer les opérations et de faire des affichages 3D. Plusieurs extensions ont été développées, notamment pour faire de l'architecture [HDMB07] ou du rendu (en développement).

1.2.3 CGOGN

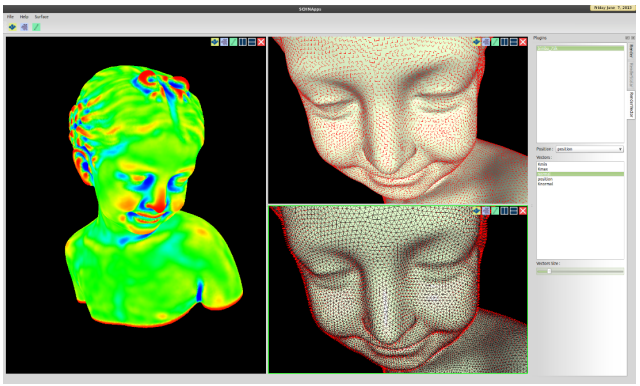


FIGURE 1.15 – SCHNApps : visualiseur graphique de CGOGN

CGOGN [KUJ+13] est une librairie de modélisation topologique. Cette librairie permet d'uti-

liser les cartes combinatoires, les G-cartes ou d'autres structures à base de brins. Chaque modèle a ses propres implantations des différentes opérations qui leur sont spécifiques. Il existe par exemple des implantations pour des cartes orientées et des G-cartes en dimensions deux et trois.

Elle propose également une bibliothèque basée sur un système de plug-in python appelée SCHNApps. Cette extension permet de manipuler les objets avec des *scripts*, et propose un visualiseur graphique. Une illustration du visualiseur est exposée en figure 1.15. Le logiciel GeoTopoModeleur [Gé], par exemple, développé par Géosiris (l'entreprise avec laquelle nous avons collaboré pendant cette thèse), utilise ce système de plug-in pour développer des opérations de re-construction du sous-sol géologique.

Pour développer de nouvelles opérations, CGOGN propose des opérations de base de manipulation des modèles topologiques mais également des opérations classiques comme la découpe d'arêtes, de faces et de volumes (selon la dimension topologique du modèle utilisé), la création de faces en fonction d'un nombre de sommets, *etc.* Ces fonctions sont codées directement dans le langage de programmation du noyau (en C++). CGOGN fournit des implantations de ses modèles topologiques jusqu'en dimension trois. En revanche, si l'on souhaite travailler dans des dimensions supérieures, il est nécessaire de réimplanter le modèle et toutes les opérations nécessaires dans la dimension souhaitée. La bibliothèque ne semble pas forcer l'utilisateur à manipuler des objets satisfaisant les contraintes de cohérence des modèles topologiques (comme celles des G-cartes que nous avons détaillées dans les sections précédentes).

Notons qu'une version 2 de CGOGN est actuellement en développement. Cette version apporte des améliorations techniques (*e.g.* parallélisation, *etc.*) afin d'augmenter les performances de la librairie notamment en terme de temps de calcul. Cette nouvelle version utilise notamment la puissance du compilateur pour améliorer les temps d'appel à des fonctions internes fréquemment utilisées. Les parcours d'orbite ont également été particulièrement optimisés et parallélisés.

1.2.4 Computational Geometry Algorithms Library (CGAL)

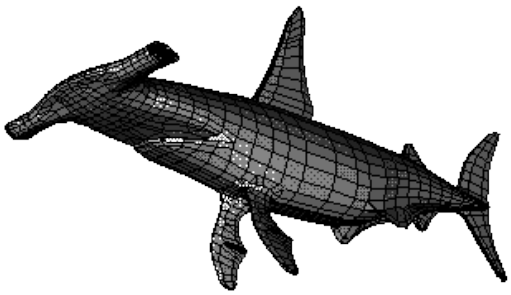


FIGURE 1.16 – Visuel d'un modèle utilisant le module Polyhedron de CGAL

CGAL [FT13] est une bibliothèque proposant différentes opérations de manipulation des objets. Elle comporte des modules de calcul exact basés sur des outils arithmétiques efficaces qui permettent d'obtenir des résultats robustes. Elle possède plusieurs modules permettant de manipuler différents types de structures topologiques. Certain de ces modules proposent également une interface graphique pour visualiser les éléments. Elle possède notamment des modules pour les arêtes ailées (cf. figure 1.16) et les Linear Cell Complex (LCC).

Par exemple, la classe Polyhedron [Ket18] est basée sur les arêtes ailées et permet de représenter des surfaces 2D plongées 3D. Cette classe est une spécialisation de la structure d'arête ailées interne à la bibliothèque. CGAL comprend également un certain nombre d'outils permettant

de mettre en place diverses vérifications sur les objets (e.g. vérifier si un polygone est convexe ou non). Ces vérifications peuvent être mises en place sous forme de pré ou post-conditions, d'assertions, *etc.* Elle fournit également plusieurs opérations de base comme la création de tetraèdres, ou des opération de subdivision. Ces fonctionnalités peuvent être exploitées pour définir de nouvelles opérations dans le langage de la librairie (C++).

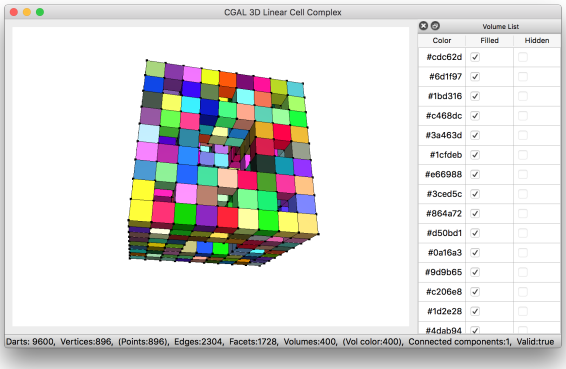


FIGURE 1.17 – Visualiseur 3D des LCC de CGAL

La classe Linear Cell Complex (LCC) [Dam18] repose sur les cartes combinatoires pour représenter des objets nD. Dans ce module, les plongements sont associés aux cellules topologiques et il est possible de définir leur comportement lorsque deux cellules sont fusionnées, ou lorsqu'une cellule est coupée afin de pouvoir conserver une cohérence de ces plongements. Une impression d'écran de l'interface de visualisation 3D des LCC est exposée en figure 1.17.

1.2.5 Outils de modélisation possédant un langage spécifique

GMSH[GR09] est un outil de génération de maillage à partir d'entités de base comme des points, des courbes orientées (segments, cercles, *etc.*), ou encore des surfaces orientées (plans, surfaces triangulées ou non, *etc.*). OpenSCAD[Wil14] est un logiciel de CAO basé sur la "géométries de construction de solide" (CSG). Autrement dit, elle permet de créer des objets en utilisant notamment des opérations booléennes (union, différence ...).

Pour créer de nouveaux objets, ces deux outils proposent leur langage dédié. Chaque langage possède ses propres opérateurs et fonctionnalités que l'utilisateur peut exploiter pour créer ses opérations. GMSH et OpenSCAD ont chacun leur analyseur syntaxique qui permet de "lire" et d'exécuter les commandes définies dans le langage, il n'est donc pas nécessaire de recompiler ces outils pour inclure les nouvelles opérations.

GSMH est très efficace pour générer des maillages de triangles ou de tétraèdres. Il est possible de spécifier des grilles de scalaires, des champs de vecteurs ou de tenseurs afin de réaliser des calculs de génération de maillages. Cependant, il n'est pas autorisé d'attribuer des informations quelconques aux objets. La bibliothèque propose un visualiseur 3D qui permet de manipuler des objets et d'appliquer des opérations pré-définies. Une image du visualiseur est montré en figure 1.18, il n'est cependant pas obligatoire pour utiliser la bibliothèque.

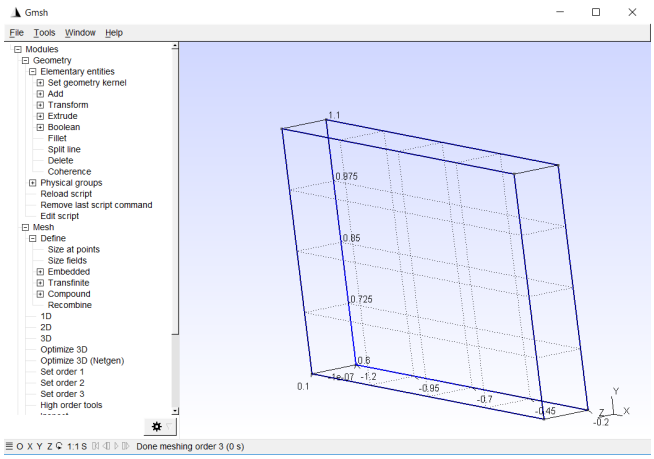
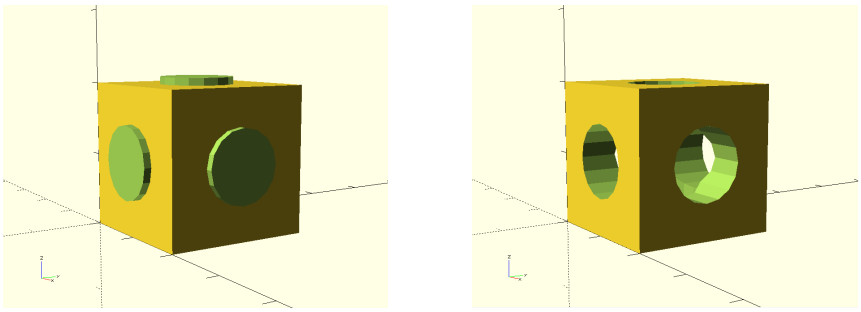


FIGURE 1.18 – Visualiseur 3D de GMSH

```
1  lc = 1e-2; // déclaration de variable
2  DefineConstant[ proportion = {100, Min 0, Max 300, Step 1,
3                                Name "Parameters/Proportion"} ];
4  Point(1) = {0, 0, 0, lc};   // création d'un sommet géométrique
5  Point(2) = {.1*proportion, 0,  0, lc} ;
6  Point(3) = {.1*proportion, .3, 0, lc} ;
7  Point(4) = {0,  .3, 0, lc} ;
8  Line(1) = {1,2} ;   // création d'une ligne entre les points 1 et 2
9  Line(2) = {3,2} ;
10 Line(3) = {3,4} ;
11 Line(4) = {4,1} ;
```

Listing 1.1 – *Script* GMSH de création d'un rectangle



(a) Objets initiaux (b) Objet résultat

FIGURE 1.19 – Vues 3D de l'objet créé par Listing 1.2

Le langage d'OpenSCAD assez similaire. Les objets peuvent être créés à l'aide d'opérations sur des forme de base : pavé, cylindre, sphère, *etc.* Il est également possible de définir des objets avec des polyèdres qui sont définis par un modèle indexé. Un exemple d'application dans le langage est montré en Listing 1.2 et son résultat est illustré en figure 1.19. Ce *script* crée un cube et 3 cylindres. Le volume des 3 cylindres est retiré du cube pour réaliser l'objet final.

```
difference(){
    difference(){
        cube([20,20,20]);
        translate([10,10,0])
        cylinder(r=5,h=20+1);
    }
    translate([10,20,10]){
        rotate([90,0,0]){
            cylinder(r=5,h=20+1);
        }
    }
    translate([0,10,10]){
        rotate([90,0,90]){
            cylinder(r=5,h=20+1);
        }
    }
}
```

Listing 1.2 – *Script* OpenSCAD

Pour ces outils, les manipulations des objets sont réalisées dans le cadre des langages et de leurs fonctionnalités de base. Les structures des objets ne sont pas manipulées directement, empêchant ainsi de créer des objets incohérents vis à vis de la structure topologique interne quand ils en ont une.

1.2.6 L-Systèmes

Les L-Systèmes ont été créés par un biologiste et botaniste : Aristid Lindenmayer [PL91]. Ce sont des grammaires formelles qui comprennent un ensemble de règles de production et un axiome de départ.

Définition 1.2.1 (Grammaire de Lindenmayer) *Une grammaire Gr(V, ω, P) se définit par :*

- *une suite finie V de symboles décrivant l'alphabet ;*
- *un axiome ω ;*
- *une suite finie P de production de règles.*

```
Axiom : A
A -> AB
B -> A
```

Listing 1.3 – Grammaire de l'algue de Lindenmayer

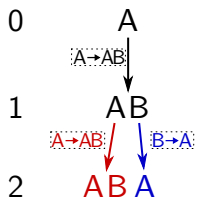


FIGURE 1.20 – Arbre de production

Par exemple on peut imaginer la grammaire présentée en Listing 1.3 telle que $V = (A, B)$, $\omega = A$, et son arbre de production est représenté en figure 1.20 pour deux itérations.

L'axiome est ici le mot A. Sur cet axiome, on applique toutes les règles de la production possibles. La première règle $A \rightarrow AB$ transforme le mot A en AB. La deuxième règle ne s'applique

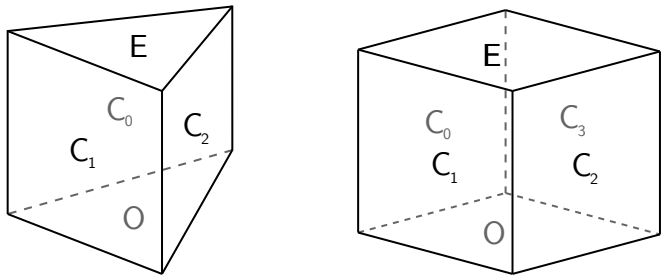
pas à cette 1ere étape, faute de présence du symbole B. Si on itère une deuxième fois la production, la première règle s'applique sur le symbole A et produit AB. La deuxième règle s'applique sur B et produit A. Le résultat de la deuxième itération de la production est donc ABA.

Les règles de la grammaire sont appliquées jusqu'à satisfaire une condition d'arrêt, qui peut être simplement le nombre d'itérations par exemple.

Les L-systèmes permettent de définir rapidement des opérations en utilisant des règles formelles. En revanche, bien qu'elles soient pratiques pour définir des opérations complexes, la visualisation des résultats nécessite de créer des outils dédiés et spécifiques à la grammaire définie. Il faut définir l'interprétation géométrique de chaque symbole de l'alphabet, par exemple A pourrait représenter un volume de tige de plante, et B une feuille.

G-cartes L-Système

Une utilisation conjointe des L-Systèmes et des G-cartes a été réalisée notamment par O. Terraz. Ces travaux visent à modéliser la croissance de feuilles [PTMG08], de troncs d'arbres [TGM+09], de fleurs [PSTG13], ou encore de fruits [Boh15].



(a) Volume d'arité 3 (b) Volume d'arité 4

FIGURE 1.21 – Exemples de volumes utilisés par les G-cartes L-Systèmes

Ici, les lettres de l'alphabet représentent un type de prisme. Ils présentent tous une face d'origine notée O et une face d'extrémité notée E. Ces deux faces sont découpées d'un même nombre d'arêtes et sont reliées entre elles par autant de faces qu'elles ont d'arêtes. Ces faces de coté sont notées C_i, avec $i \in \{0, \ldots, n-1\}$, où n est le nombre de coté du prisme (ou arité). La figure 1.21(a) montre un prisme d'arité trois et la figure 1.21(b) un d'arité quatre. Dans les G-cartes L-Systèmes, l'axiome de la grammaire décrit le volume présent avant l'application des règles. Les règles décrivent ensuite l'ajout ou la suppression de prismes, ou le collage de faces. Tous les prismes créés par les règles sont accrochés par leur face d'origine à des volumes existants avant l'application de la règle. Pour créer un nouveau prisme, les règles décrivent le type de prisme à créer, et le type de prisme d'accroche ainsi que le type de face sur laquelle les nouveaux prismes seront accrochés. On peut par exemple décrire l'ajout de prismes B sur toutes les faces C des prismes. Cette opération est décrite par la règle suivante : $A \rightarrow A[B]_C$.

Le Listing 1.4 montre un exemple de grammaire utilisant les G-cartes L-Systèmes, et le résultat de son application est illustré en figure 1.22. Sur cet exemple, les faces d'origine sont en rouge, les faces d'extrémité sont en violet et les faces de coté (C_i) sont en vert. Cette grammaire permet de réaliser la *Triforce* du jeu *Zelda*. Elle est composée de trois parties. La première est la définition des différents prismes et leurs propriétés. La définition d'un type de prisme se fait comme suit : $\#define\ P(O, ct, atx, aty, atz, H, L, W)$. Ici le prisme est nommé P, et est d'arité O. Le paramètre ct est un facteur de translation qui permet de déplacer P par rapport à la face sur laquelle il va être accroché lors de sa création. Le déplacement s'effectue en suivant

la direction de la normale à la face d'accroche. Les paramètres atx, aty, et atz décrivent une rotation de P par rapport à la normale de la face d'accroche. Finalement, H décrit la hauteur du prisme, L la longueur et W la largeur. Dans notre exemple, 3 prismes sont définis. A décrit un prisme d'arité 3 et B et C des prisme d'arité 4.

A est défini comme l'axiome de la grammaire. La première règle décrit la création de prismes B sur toutes les faces C de A. La deuxième règle décrit le collage de toutes les faces $C2$ et $C4$ adjacentes qui appartiennent à des prismes B. La troisième règle décrit la création de prismes C sur les faces E des prismes B. Finalement les deux dernières règles décrivent l'application d'une couleur transparente sur tous les prismes A et B.

```
#define A(3,1,0,0,0,10,10,10)
#define B(4,1,0,0,0,10,10,10)
#define C(4,2.3,0,0,0,1,1,10)
#axiome : A                 @step 0@
A -> A[B]_C                 @step 1@
B -> B^{C2} | B^{C4}        @step 2@
B -> B[C]_E                 @step 3@
A -> #A#                    @step 4@
B -> #B#                    @step 5@
```

Listing 1.4 – G-carte L-Système de la Triforce

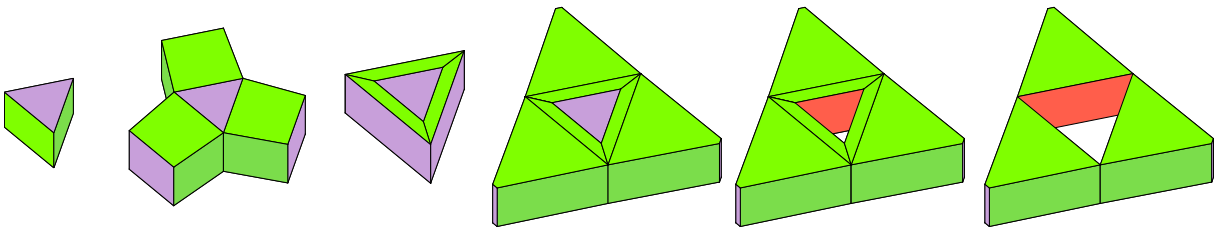
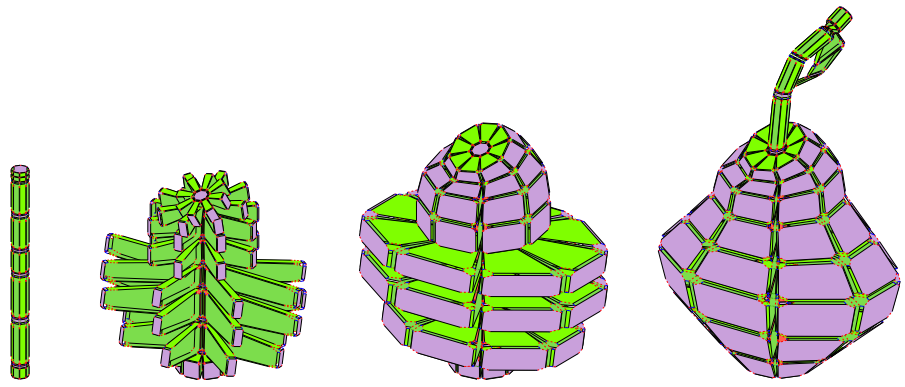


FIGURE 1.22 – Résultat de l'application de la grammaire de la Triforce étape par étape

Il est bien entendu possible de réaliser des modèles plus complexes comme la poire présentée en figure 1.23.

Les L-Systèmes peuvent également être utilisés pour des domaines d'application très différents de la modélisation. Il est par exemple possible de générer des mélodies musicales (cf. [WS05] et [MaK14]). Ils sont également utilisés pour générer des dessins de fractales (triangle de Sierpinski, arbres binaires, *etc.*).

En conclusion, les L-Systèmes permettent de réaliser des modèles complexes dans des domaines d'applications très variés. Cependant, dans le cadre de la modélisation géométrique, il est nécessaire de définir à quoi correspond chaque lettre de la grammaire, comme nous l'avons vu pour les G-cartes L-Systèmes où chaque lettre décrit un prisme avec des propriétés particulières. Selon le type d'application, les objets n'ont pas les mêmes propriétés. En géologie par exemple, on peut souhaiter attribuer aux objets des informations de type de roche, de porosité, *etc.* Ainsi, il serait difficile de créer un unique outil qui puisse être utilisé dans deux domaines d'application complètement différents.



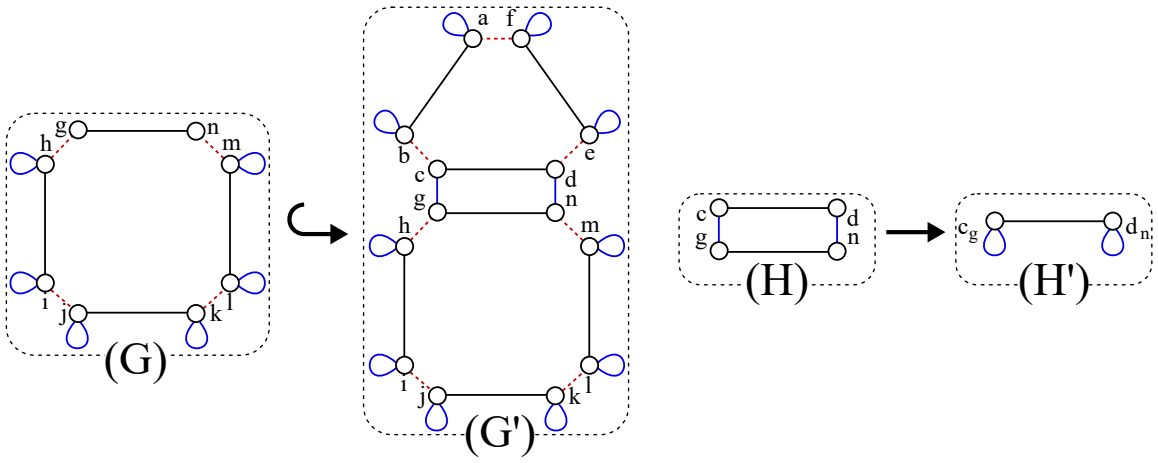
(a) (b) (c) (d)

FIGURE 1.23 – Étapes de génération d'une poire avec les G-cartes L-Système

1.3 Jerboa

Jerboa est une plateforme de modélisation géométrique à base topologique basée sur les cartes généralisées. Elle permet de concevoir et programmer des opérations graphiquement, en exploitant les règles de transformations de graphes

1.3.1 Fondements théoriques



(a) Morphisme d'inclusion (b) Morphisme non injectif

FIGURE 1.24 – Morphisme de graphe

Il est possible de définir des relations entre différents graphes avec des morphismes. Intuitivement, un morphisme $m : G \to G'$, ou simplement $G \to G'$, où G et G' sont deux graphes, associe respectivement les nœuds et les arcs de G aux nœuds et arcs de G'. Plus formellement, un morphisme m d'un graphe $G = (V, E, s, t, \alpha, \Pi)$ vers un graphe $G' = (V', E', s', t', \alpha', \Pi')$, est défini par deux fonctions $m_V : V \to V'$ et $m_E : E \to E'$ qui préservent les sources, cible et étiquettes topologiques et de plongements.

Par exemple, le morphisme de la figure 1.24(a) associe les nœuds de G à ceux du même nom

dans G' et les arcs de G aux mêmes arcs dans G'. Sur cet exemple, tous les nœuds/arcs de G sont des nœuds/arcs de G'. Ainsi on peut dire que $m : G \to G'$ est un morphisme d'inclusion et se note $m : G \hookrightarrow G'$.

Un morphisme m est dit injectif si les fonctions m_V et m_E sont injectives. Dans le cas contraire, un morphisme peut associer plusieurs éléments à la même image. Par exemple, celui de la figure 1.24(b) associe respectivement les nœuds c, g et d, n de H à c_g et d_n dans H'. Il associe également les deux arcs α_0 de H à celui de H', et les α_2 de H à leur boucles correspondantes dans H'.

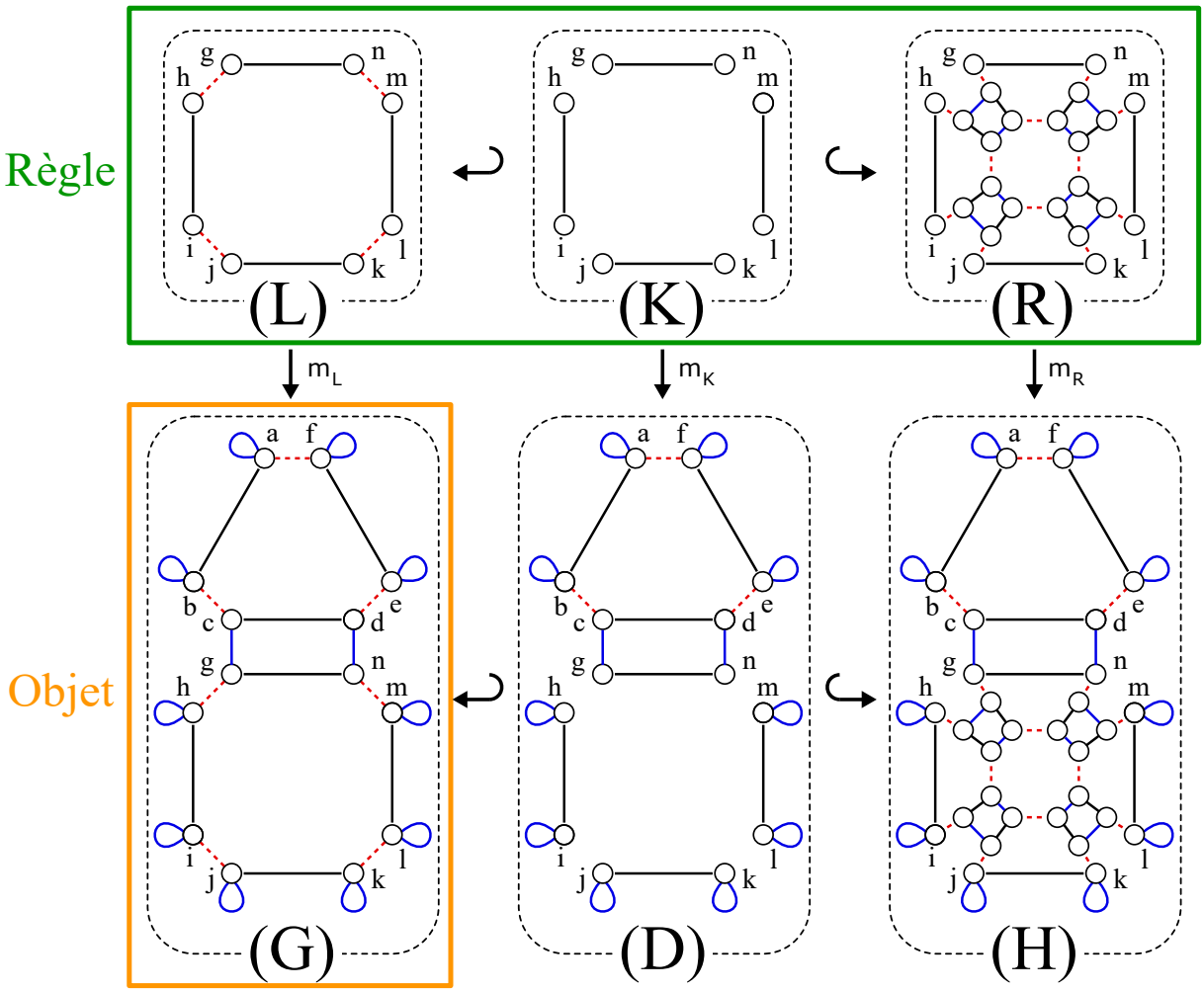


FIGURE 1.25 – Application d'une règle de transformation de graphe

Dans Jerboa, les objets sont représentés par des cartes généralisées. Les opérations de manipulation des objets sont définies par des transformations de graphe (cf. [EEPT06][Bel12]). Les transformations de graphes sont définies par un triplet $L \hookleftarrow K \hookrightarrow R$, où L est le motif filtré, le sous-graphe que l'on souhaite transformer, R le motif transformé et K est l'interface des motifs L et R. K peut être défini par défaut comme l'intersection de L et R. Sur l'image figure 1.25, on peut observer une règle de transformation de graphe et son application sur un objet représenté par le graphe G. Ici la règle définit la triangulation du quadrangle.

L'application de la règle sur le graphe de l'objet concret se fait par l'association des nœuds de L avec les nœuds du graphe de l'objet. Cette association est un morphisme injectif dit de

filtrage. Sur la figure ce morphisme est noté m_L.

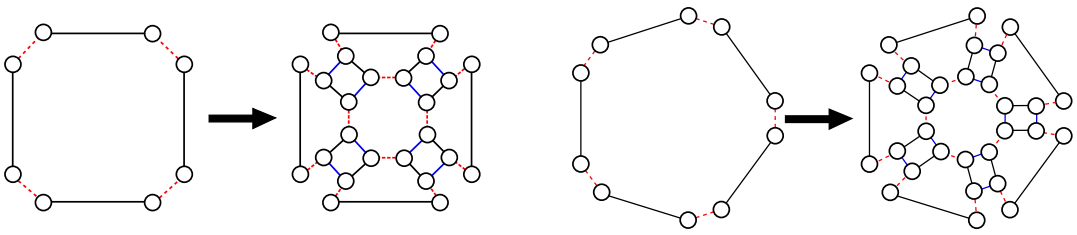
Notons que dans le cas où l'association entre les nœuds de la règle et ceux du graphe concret n'est pas possible, (il n'existe pas de morphisme de filtrage m_L) le filtrage échoue et la règle ne peut pas s'appliquer.

Sur l'exemple, l'application de la règle filtre le quadrangle de l'objet, et les liaisons α_1 filtrées sont supprimées, comme entre g et h par exemple. Un nouveau motif est créé et accroché au motif filtré, par α_1. Ce nouveau motif décrit les nouvelles faces triangulaires. La présente règle permet de trianguler exclusivement le quadrangle de cet objet. Pour trianguler une face avec un nombre différent de sommets, il faudrait écrire une autre règle. Il est cependant possible d'éviter d'écrire une règle pour chaque type de face que l'on souhaite trianguler, en utilisant des variables. Pour cela les règles Jerboa présentent deux types de variables : les variables orbites qui permettent de filtrer toute orbite d'un type donné, et les variables de nœud qui permettent d'accéder aux plongements des brins filtrés par les nœuds de la règle.

L'application des transformations de graphes aux cartes généralisées a été définie dans la thèse de Mathieu Poudret [Pou09]. Le graphe K étant l'intersection de L et R, M. Poudret a proposé une approche où les opérations ne sont définies que par les deux graphes L et R. La syntaxe concrète est donc : $L \rightarrow R$. Dans ce cas, $\rightarrow$ est une simple notation et ne décrit donc plus un morphisme.

1.3.2 Transformation de la structure topologique

Nous venons de voir le fonctionnement des transformations de graphe, nous allons à présent décrire le processus de construction d'une règle Jerboa.



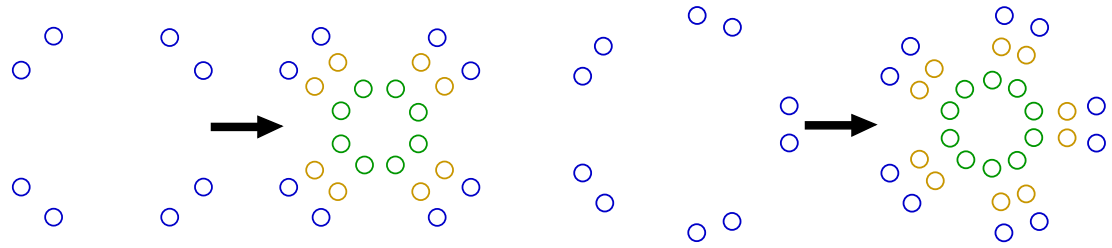
(a) Triangulation d'un quadrangle (b) Triangulation d'un pentagone

FIGURE 1.26 – Règles de transformations de graphe pour l'opération de triangulation barycentrique

Supposons que nous souhaitons écrire une règle de triangulation barycentrique. Nous devons commencer par comprendre les modifications que cette opération réalise sur des objets. Les figures 1.26(a) et 1.26(b) illustrent, l'avant et l'après triangulation, respectivement d'un quadrangle et d'un pentagone. Sur ces figures, on peut observer des régularités dans les modifications.

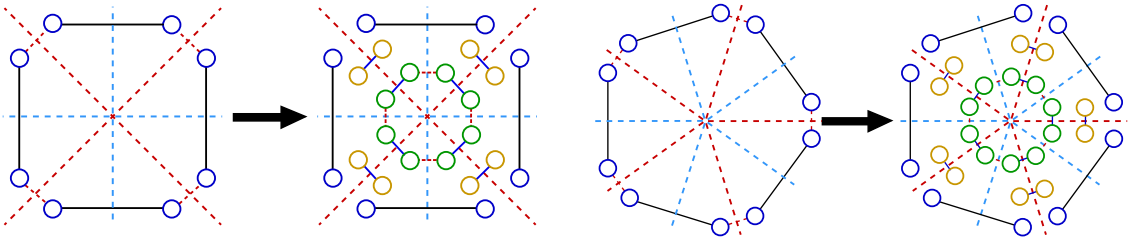
En effet, comme on peut l'observer sur la figure 1.27 sur laquelle ne sont présentés que les brins, il y en a trois fois plus dans les objets triangulés que dans les objets initiaux. Pour chaque brin de l'objet, deux copies sont créées. Sur la figure, nous avons identifié les brins d'origine par la couleur bleu, la première copie par le jaune, et la deuxième par le vert.

On peut également observer des régularités sur les modifications des liaisons. Les brins bleus liés par α_1 ne sont pas liés sur l'objet triangulé, et leur copies jaunes le sont entre elles par α_2, de même que les vertes. Nous avons mis en évidence ces similarités par des axes de symétrie rouge sur la figure 1.28. Les brins bleus liés par α_0 sont également liés sur l'objet triangulé, leurs copies

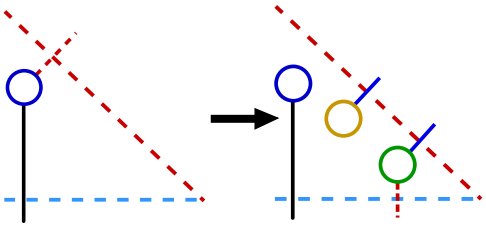


(a) Quadrangle (b) Pentagone

FIGURE 1.27 – Étape de copie



(a) Quadrangle (b) Pentagone



(c) Zoom sur un cadran

FIGURE 1.28 – Étape de recherche des symétries

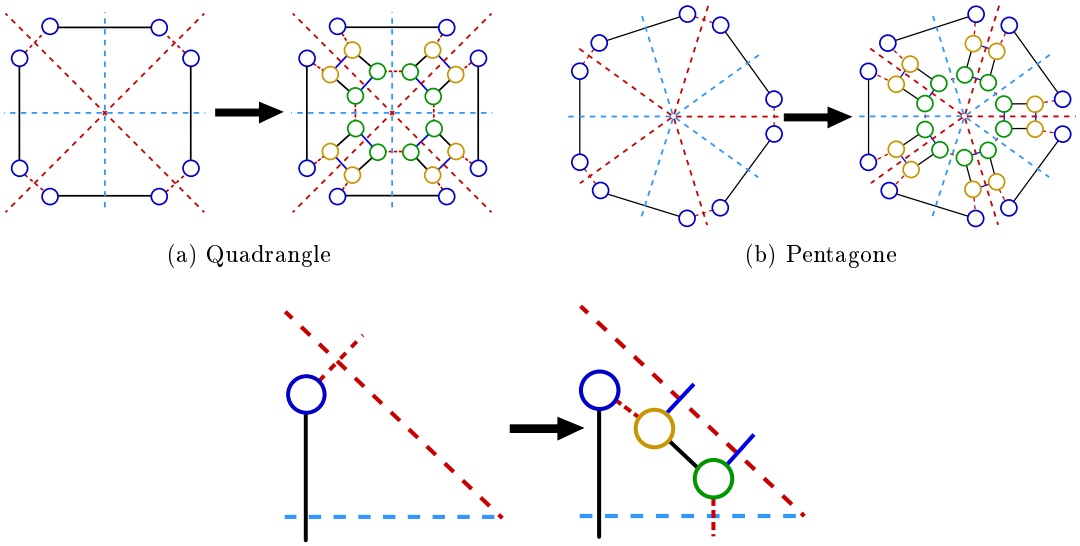
jaunes ne le sont pas, mais les vertes le sont entre elles par α_1. Nous avons mis en évidence ces similarités par des axes de symétrie cyan sur la figure. Ces axes de symétrie séparent les brins dans plusieurs cadrans, contenant chacun un brin bleu, un jaune et un vert. Les liaisons entres les brins des différents cadrans sont celles que nous venons de décrire avec les symétries.

Pour finir, on observe en figure 1.29 des motifs similaires au sein de chaque cadran. Les brins bleus sont liés à leur copie jaunes par α_1, et les jaunes aux verts par α_0.

Nous avons donc mis en évidence que les transformations sont identiques pour tous les brins bleus, de même que pour les brins jaunes entre eux, et les verts entre eux, et ce, quel que soit le type de face triangulé. Nous allons à présent écrire une règle Jerboa qui décrit ces transformations.

Dans les règles Jerboa, les transformations identiques sont rassemblées grâce à la notion de schéma de règle, introduite par Mathieu Poudret dans sa thèse [Pou09]. Les schémas de règle reposent sur la notion de variable topologique, qui permettent de rassembler les brins d'une même orbite, afin de leur appliquer des transformations similaires. Les variables topologiques sont les nœuds des graphes des règles, étiquetés par l'orbite qu'elles représentent.

Pour rassembler tous les brins du quadrangle entre eux ou du pentagone entre eux, nous définissons une variable topologique qui permet de représenter ces deux objets, c'est à dire une



(a) Quadrangle

(b) Pentagone

(c) Zoom sur un cadran

FIGURE 1.29 – Illustration des iaisons des brins dans les cadrans

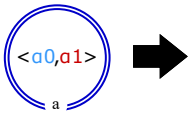
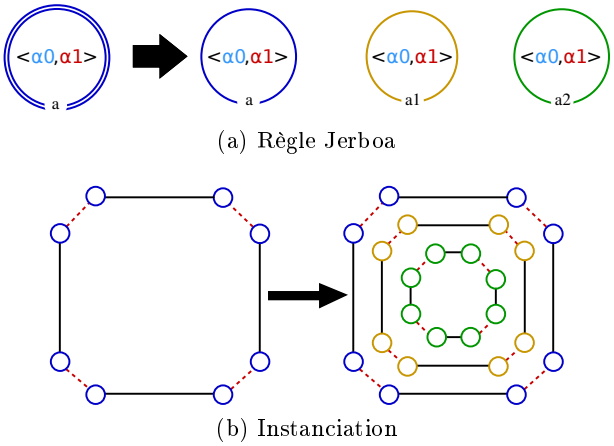


FIGURE 1.30 – Graphe gauche de la règle de triangulation

variable typé par l'orbite d'une face : $\langle \alpha_0, \alpha_1 \rangle$. Cette variable permet de représenter le type d'orbite sur lequel peut s'appliquer la règle. Nous verrons plus tard que les nœuds d'accroche, représentés par un double cercle, permettent d'instancier cette variable (comme pour le nœud a sur la figure 1.30).



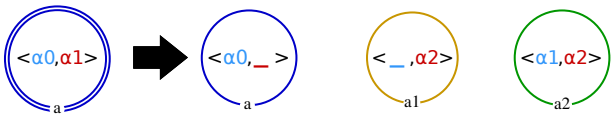
(a) Règle Jerboa

(b) Instanciation

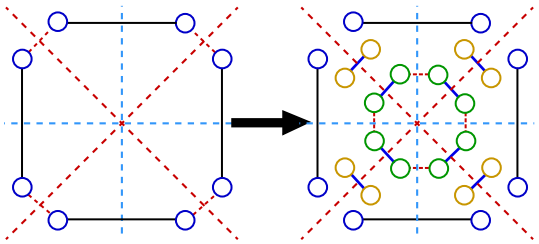
FIGURE 1.31 – Étape de copie du motif filtré

Nous avons observé que les brins bleus étaient également présents dans l'objet triangulé. Ainsi, la variable a doit également être présente dans le graphe droit de la règle. De plus, chaque brin

bleu présente une copie jaune et une verte. Nous ajoutons donc deux autres variables nommées $a1$ et $a2$ que nous étiquetons également par l'orbite α_0, α_1 pour copier tous les brins bleus, comme illustré sur la figure 1.31(a). Sur la figure, nous avons coloré les liaisons des orbites de la même couleur que les axes de symétrie qu'elles coupent sur l'objet avant triangulation : cyan pour α_0 et rouge pour α_1. La figure 1.31(b) montre le résultat sur un quadrangle de l'instanciation de cette règle. Nous avons défini les différentes copies des brins bleus, nous devons à présent décrire les transformations des liaisons topologiques pour chaque variable.



(a) Règle Jerboa



(b) Instanciation

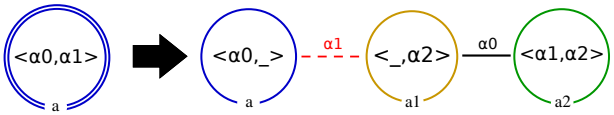
FIGURE 1.32 – Étape de renommage des liaisons implicites

En effet, les brins bleus sont toujours liés par α_0 mais ne sont plus liés par α_1 sur l'objet triangulé. Ainsi, il faut supprimer cette liaison de l'étiquette topologique de la variable a afin de décrire la suppression de cette liaison par l'application de la règle. La suppression d'une liaison se note par le caractère '_'. Dans le graphe droit de la règle, a est donc étiqueté par $\langle \alpha_0, _ \rangle$. Les liaisons des brins jaunes et des brins verts doivent également être correctement définies. Pour deux cadrans séparés par un axe cyan, leurs brins jaunes ne sont pas liés. Pour deux cadrans séparés par un axe rouge, leurs brins jaunes sont liés par α_2. Pour effectuer cette transformation, nous supprimons la liaison α_0, qui représente le voisinage des brins jaunes à travers la symétrie cyan, de l'orbite du nœud jaune. Nous renommons la liaison α_1, qui représente le voisinage des brins jaunes à travers la symétrie rouge, en α_2. La variable jaune est donc étiquetée par $\langle _, \alpha_2 \rangle$. Nous procédons de façons similaire avec la variable verte que l'on étiquette par : $\langle \alpha_1, \alpha_2 \rangle$. Ainsi nous obtenons la règle présentée en figure 1.32(a). Son application sur un quadrangle est illustrée en figure 1.32(b).

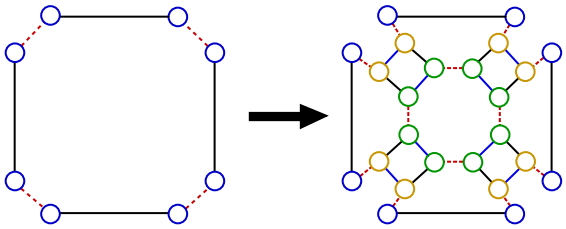
Nous venons de définir les liaisons entre les différents cadrans. Nous appelons ces liaisons des liaisons implicites. À l'intérieur d'un cadran, les liaisons sont toujours définies de façon similaire. Nous appelons ces liaisons des liaisons explicites. Dans chaque cadran de l'objet triangulé, le brin bleu est lié au jaune par α_1 et le jaune au vert par α_0. Nous reportons simplement ces liaisons entre les différentes variables afin de compléter la règle.

On obtient ainsi la règle Jerboa complète, présentée en figure 1.33(a), dont le résultat appliqué au quadrangle est illustré en figure 1.33(b).

Ici nous avons choisi d'instancier la règle par un quadrangle mais il est également possible de l'instancier sur le pentagone comme illustré en figure 1.34. Sur cette instance, les transformations que nous avons décrites pour le quadrangle ont été appliquées de façon identiques. Plus généralement, cette règle peut être instanciée par n'importe quelle face, quelle que soit son arité.



(a) Règle Jerboa



(b) Instanciation

FIGURE 1.33 – Opération de triangulation

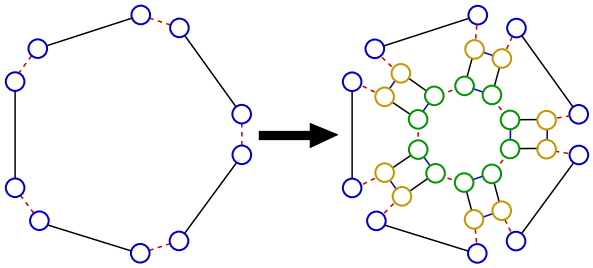
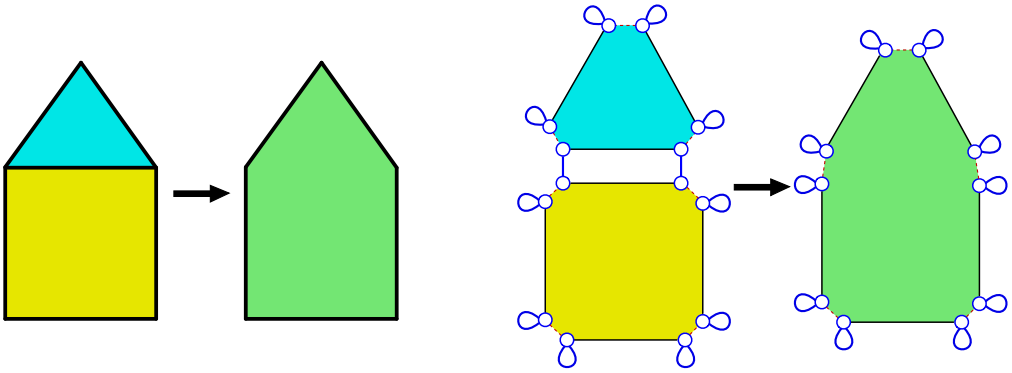


FIGURE 1.34 – Instanciation de la règle de la figure 1.33(a) en un pentagone



(a) Objet (b) G-carte

FIGURE 1.35 – Fusion de deux faces voisines

Nous venons de voir un exemple où tous les brins de l'objet d'origine peuvent être rassemblés dans une seule variable, mais ce n'est pas toujours le cas. Le graphe gauche des règles peut comporter plusieurs nœuds selon les configurations à filtrer ou les transformations à effectuer. Prenons l'exemple de la fusion de deux faces, par la suppression de leur arête commune. Un exemple de l'application de cette opération sur les deux faces de la "maison" est illustré en figure 1.35(a). La figure 1.35(b) montre les brins à supprimer et les liaisons à modifier.

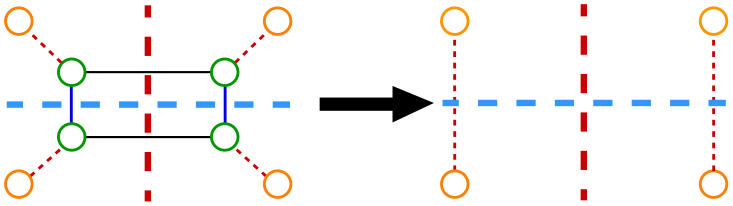


FIGURE 1.36 – Partie de l'objet modifiée par la fusion de deux faces et résultat de l'opération

En procédant de façon similaire à notre exemple précédent, nous cherchons à rassembler les transformations similaires. La figure 1.36 montre la partie de l'objet sur laquelle porte les modifications, et le résultat de l'opération. Nous identifions deux groupes de brins. Ceux qui font partie de l'arête à retirer (en vert) et ceux qui appartiennent aux arêtes adjacentes (en jaune). Nous identifions également deux types de transformations entre les brins jaunes. Les couples de brins jaunes dont le voisins verts sont liés par α_0, ne sont pas liés sur l'objet final. Les couples de brins jaunes dont le voisins verts sont liés par α_2, sont liés par α_1 dans l'objet final. Nous avons mis en évidence ces deux types de transformations par des axes de symétrie rouge et cyan. Ces axes séparent les brins dans quatre cadrans. Chacun d'eux contient un brin jaune et un vert dans l'objet initial et un unique brin jaune dans l'objet final.

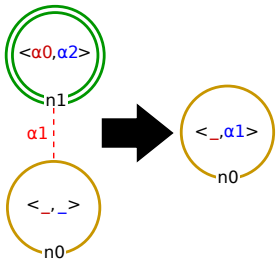


FIGURE 1.37 – Règle Jerboa de fusion de deux faces

La figure 1.37 présente la règle Jerboa qui représente l'opération. Son graphe gauche contient une variable verte et une jaune, liées par α_1. Elle contient seulement une variable jaune dans son graphe droit. À gauche, la variable $n1$ est étiquetée par $\langle \alpha_0, \alpha_2 \rangle$ pour décrire les relations entre les brins verts, entre les différents cadrans, séparés par des axes rouges et cyan. Sur l'objet initial, les brins jaunes ne sont pas liés entre eux, quel que soit l'axe de symétrie qui les sépare. On marque l'absence d'une liaison par un '_'. La variable jaune à gauche est donc étiqueté par $\langle _, _ \rangle$. Sur l'objet final, les couples de brins jaunes appartenant à deux cadrans voisins par un axe rouge ne sont pas liés. En revanche, ceux voisins par un axe cyan sont liés entre eux par α_1. La variable $n0$ du graphe droit est donc étiquetée par $\langle _, \alpha_1 \rangle$.

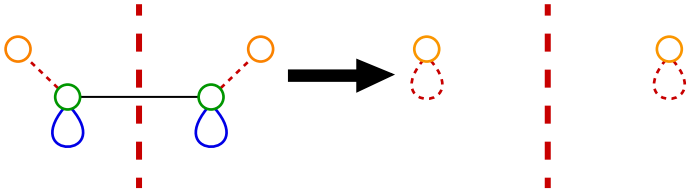


FIGURE 1.38 – Instanciation de la règle de la figure 1.37 en une arête de bord

Comme nous l'avons vu précédemment, les règles Jerboa peuvent s'instancier sur n'importe

quelle orbite que la règle peut filtrer. Dans notre exemple, il est possible d'instancier la règle de fusion de face en une arête de bord, comme illustré en figure 1.38. Cette instanciation revient à replier l'axe de symétrie bleu sur lui-même. Ainsi les brins de part et d'autre de cet axe sont eux mêmes. Pour cette raison, les brins jaunes sont liés à eux-même par α_1 sur l'objet final. Il est parfois souhaitable de ne pas autoriser toutes les intanciations possibles d'une règle. On peut préférer que cette opération ne puisse s'instancier qu'en une arête interne, et non sur une arête de bord.

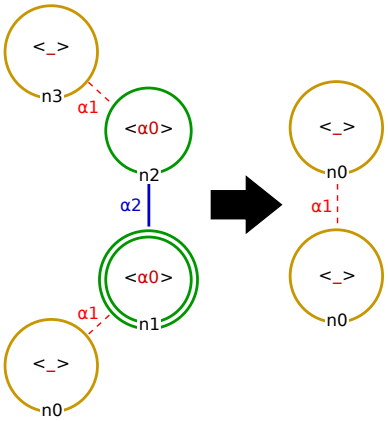


FIGURE 1.39 – Règle Jerboa de suppression d'arête interne uniquement

Pour ce faire, il faut forcer la présence de voisins en α_2 pour les brins verts. Il ne faut donc pas regrouper les brins verts voisins par α_2 dans la règle. L'axe cyan n'est pas pris en compte, on obtient ainsi deux cadrans séparés par un axe rouge, contenant chacun deux brins verts et deux brins jaunes. Avec cette modification, on obtient la règle présentée en figure 1.39. Aucune instance de cette règle ne peux être une arête de bord.

L'utilisation des variables orbites, typées par des liaisons qui coupent des axes de symétrie permet donc de généraliser une opération de transformation de graphe. Cependant, il n'est pas obligatoire de prendre en compte tous les regroupements possibles s'il n'est pas souhaitable de généraliser l'opération à tous les cas.

Pour appliquer une règle Jerboa, l'utilisateur doit fournir l'association entre les nœuds du graphe gauche et les brins de l'objet. Nous appelons ces nœuds, des nœuds d'accroche, et ces brins, des brins accrochés. Reprenons l'exemple de la règle de triangulation que nous avons construite dans cette section, et appliquons la sur le quadrangle de la maison comme illustré sur la figure 1.40. Ici, nous avons choisi d'associer le nœud a de la règle au brin n de l'objet. Cette association est représentée par une flèche discontinue noire.

Grâce à la contrainte des *arcs adjacents* des G-cartes (chaque brin a une et une seule liaison par dimension), chaque brin appartient à une et une seule orbite d'un type donné. Ainsi, si la variable a, étiquetée par l'orbite $\langle \alpha_0, \alpha_1 \rangle$ est associée au brin n, sa seule instanciation possible est l'orbite $\langle \alpha_0, \alpha_1 \rangle$ incidente à n, c'est à dire un quadrangle.

L'instanciation d'une règle qui comporte plusieurs nœuds à gauche se fait de façon similaire. Il faut associer les nœuds du graphe gauche avec des brins de l'objet. Cependant, pour que la règle puisse s'instancier, il n'est pas nécessaire de fournir l'association de tous les nœuds. En effet, nous avons construit la règle en rassemblant les transformations similaires dans des cadrans, qui comportent tous le même nombre de brin. On considère qu'il y a dans chaque cadran, un brin de

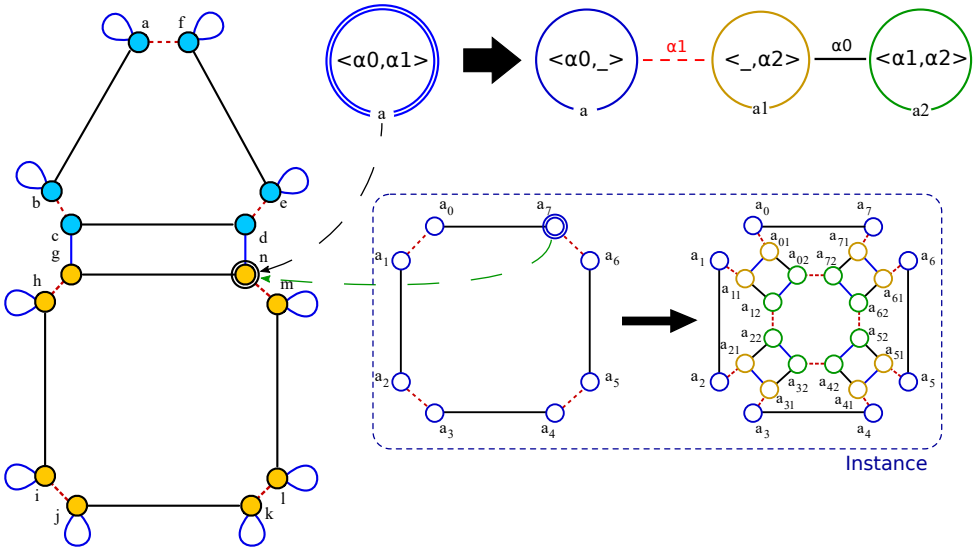


FIGURE 1.40 – Instanciation des variables topologiques

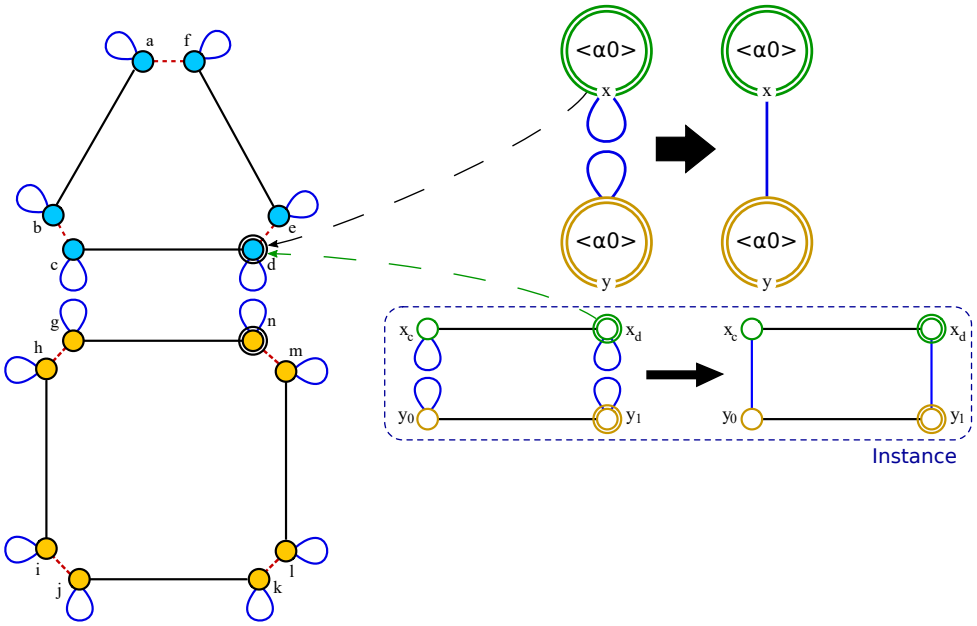


FIGURE 1.41 – Instanciation des variables topologiques sur la règle de couture en α_2

référence et que tous les autres sont des copies, dont les liaisons entre les cadrans sont différentes du brin de référence. Ainsi, les règles n'ont qu'une seule variable, les autres nœuds du graphe gauche ne sont que des copies de celle-ci, à renommage et suppression près des liaisons implicites. Le choix du nœud qui représente la variable dans le graphe gauche de la règle importe peu car ils présentent tous le même nombre de brins. Cependant, le nœud de la variable doit être étiqueté par une orbite pleine, c'est-à-dire sans absence de liaison entre les cadrans, afin d'être en mesure d'accéder à tous les brins qu'il représente dans tous les cadrans. En associant la variable de la règle avec un brin de l'objet, la règle peut être instanciée.

Prenons l'exemple d'une règle de couture en α_2 comme illustré en figure 1.41, pour l'appliquer sur l'objet à gauche de la figure et coudre le quadrangle au triangle. Ici nous avons choisi d'associer

le nœud x au brin d. Avec cette association, il est possible de construire l'instanciation de la règle correspondante. x est instancié par l'orbite $\langle\alpha_0\rangle(d)$ c'est à dire une arête. y est instancié comme une copie de l'instance de x. La liaison implicite de y est la même que x, son instance ne subit donc pas de modification topologique.

L'étape suivant l'instanciation de la règle est celle de la création du morphisme de filtrage. Ce morphisme permet d'associer les brins de l'instance de la règle aux brins de l'objet, afin de spécifier la partie de l'objet sur laquelle s'applique la règle.

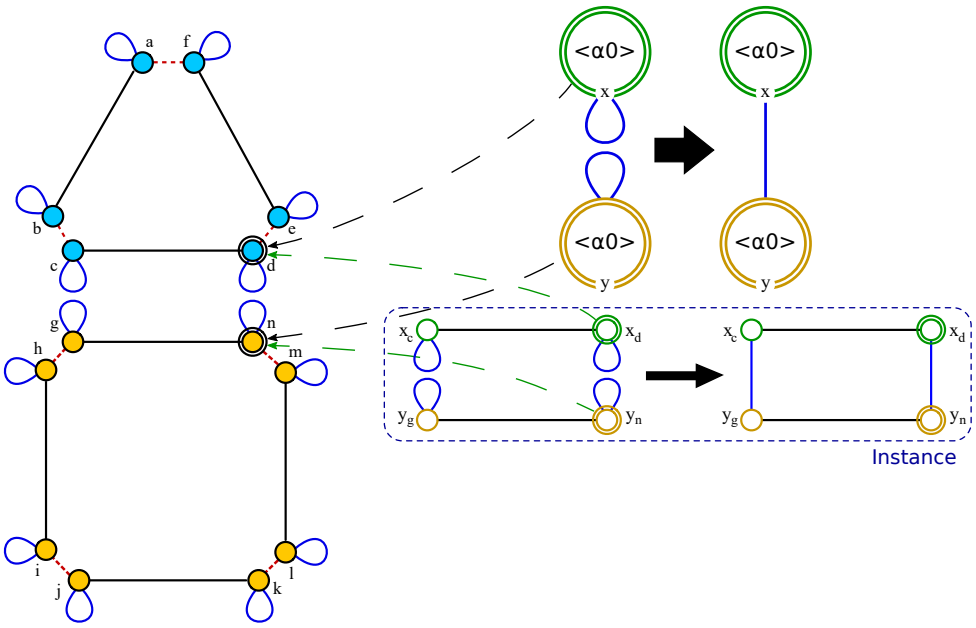


FIGURE 1.42 – Instanciation des variables topologiques sur la règle de couture en α_2

De même que l'instanciation est calculée à partir du brin accroché de l'objet, le morphisme de filtrage est créé automatiquement entre les instances des variables orbites et les brins de l'objet. Dans notre exemple de couture en α_2, les nœuds x et y de la règle appartiennent à deux composantes connexes différentes. Nous avons donné l'association entre le nœud x et le brin d, mais nous n'avons pas fourni d'association à y car elle n'est pas nécessaire pour instancier la règle. Sans cette association, il n'est pas possible de créer le morphisme de filtrage. En effet, il est possible d'associer les instances de x (brin verts : x_c et x_d) respectivement aux brins c et d de l'objet grâce à l'association donnée entre x et d. L'association entre y_g et y_n avec des brins de l'objet n'est pas déterminable. Ici, ces brins peuvent être associés à n'importe quelle arête de bord de l'objet. Pour pouvoir construire le morphisme de filtrage, il faut lever cette ambiguïté en associant y à un deuxième brin accroché sur l'objet. Nous avons choisi d'associer y au brin n, comme illustré en figure 1.42.

Avec les deux brins accrochés, il est possible de créer le morphisme m de filtrage de la règle comme illustré en figure 1.43. Ce morphisme associe respectivement les brins c, d, g, et n de l'objet aux brins x_c, x_d, y_g et y_n de l'instance. Plus généralement, il est nécessaire de déterminer un nœud d'accroche pour chaque composante connexe du graphe gauche des règles. Nous notons ces nœuds par des doubles cercles dans les règles.

Le morphisme de filtrage n'existe cependant pas pour toutes les associations nœud/brin possibles entre la règle et l'objet. Il existe seulement si il n'y a pas de recouvrement entre les

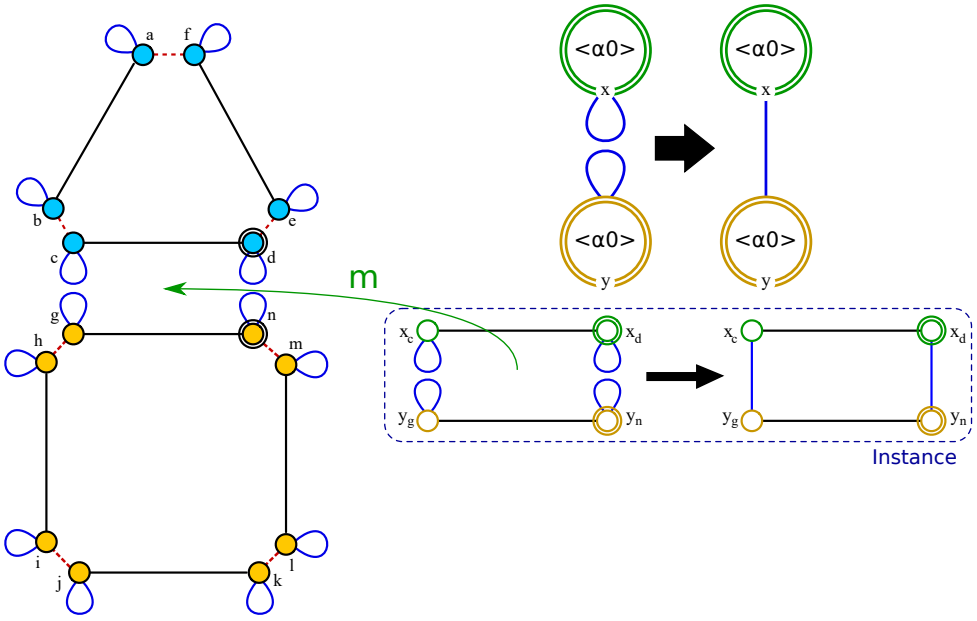


FIGURE 1.43 – Instanciation des variables topologiques sur la règle de couture en α_2

différentes instances de nœud et si elles sont isomorphes. On appelle recouvrement l'association entre un brin de l'objet avec plusieurs brins de l'instance de la règle. Sur la figure 1.42, il y a recouvrement si par exemple, on associe y au brin n, et x au brin n ou au brin g. Dans ce cas, n et g seraient représentés à la fois par x et par y.

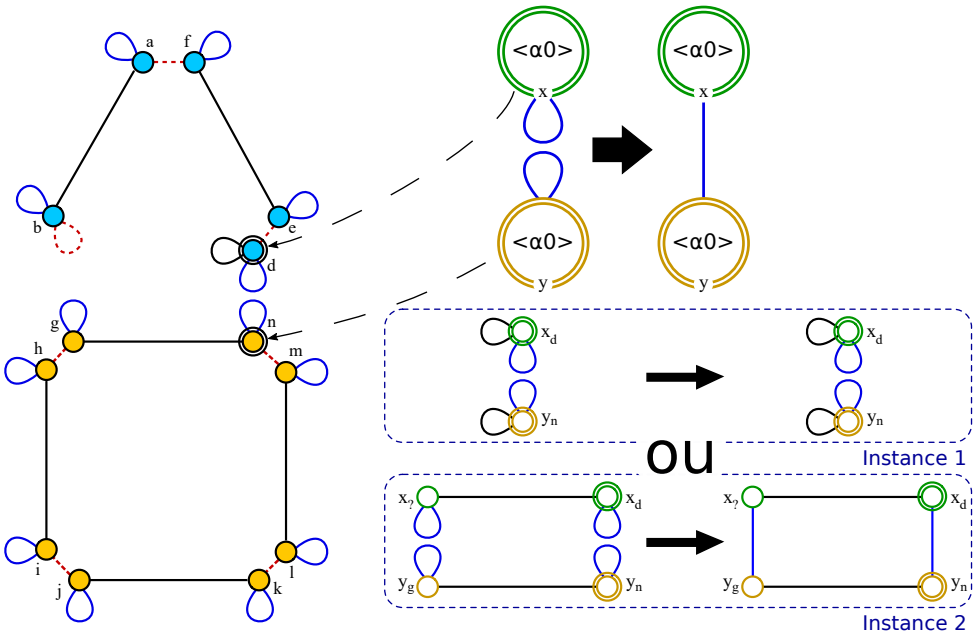


FIGURE 1.44 – Orbites instances non isomorphes

Pour illustrer la nécessité de l'isomorphie, prenons l'exemple de la figure 1.44, où nous avons supprimé un brin du triangle. Si la variable de la règle est x, alors la règle est instanciée par l'instanciation 1 de la figure. Dans le cas où y est la variable, la règle est instanciée par l'ins-

tanciation 2. Sur cet exemple, le morphisme de filtrage n'existe pas car les orbites instances de x et y ne sont pas isomorphes avec les orbites des brins qui leurs sont associées quelle que soit l'instance choisie.

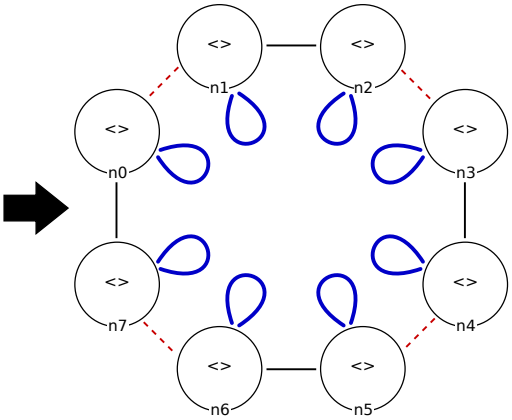
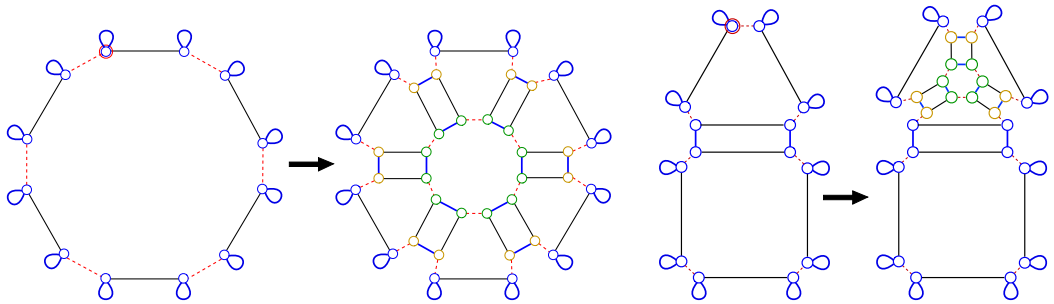


FIGURE 1.45 – Règle Jerboa de création d'un quadrangle

Nous venons de voir l'utilisation des variables topologiques pour modifier des objets existants, filtrés par le graphe gauche des règles. Ces dernières peuvent également avoir un graphe gauche vide, comme sur la figure 1.45. Dans ce cas, elles créent un nouvel objet sans s'appuyer sur un objet existant. Le graphe gauche étant vide, les nœuds du graphe droit ne sont pas des copies du motif filtré, et ne dépendent pas d'un objet particulier. Ce sont des constantes. Le type constante des variables topologiques est $\langle\rangle$, qui est l'orbite des brins.



(a) Triangulation appliquée à un hexagone

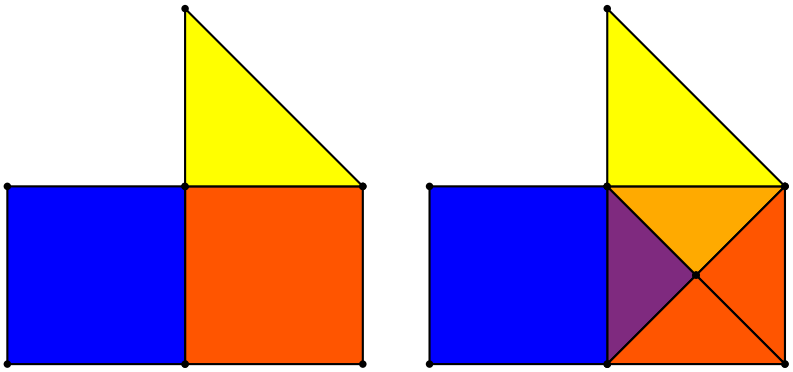
(b) Triangulation du triangle de l'exemple de la maison

FIGURE 1.46 – Exemples d'application de la règle de triangulation

Grâce aux variables topologiques, les règles peuvent s'appliquer sur n'importe orbite qui satisfait leur motif gauche. Ainsi, il est possible d'appliquer la règle de triangulation barycentrique que nous avons présentée ici sur un hexagone comme en figure 1.46(a), ou sur le triangle de la maison, comme en figure 1.46(b).

Nous avons décrit dans cette section la création de règles permettant la modification de la topologie des objets. Dans la règle de triangulation, un nouveau sommet est créé, mais la règle que nous venons de définir ne lui attribue pas de position géométrique particulière (au barycentre de la face par exemple). Pour créer des opérations complètes, il est nécessaire de définir des transformations de plongement.

1.3.3 Transformation des plongements



(a) Objet (b) Résultat

FIGURE 1.47 – Application de l'opération de triangulation sur un carré d'un objet quelconque

La figure 1.47 montre un objet et le résultat de l'application de l'opération de triangulation barycentrique sur son carré rouge. Dans la section précédente nous avons écrit une règle Jerboa qui permet de réaliser les transformations topologiques de cette opération. Nous souhaitons à présent compléter cette règle en ajoutant la description des transformations des plongements. Ici nous avons deux plongements : la position géométrique des sommets et la couleur des faces. Nous avons choisi de positionner le nouveau sommet du résultat, au barycentre de la face. Les nouvelles faces partagent une arête avec la face d'origine. Nous avons choisi d'attribuer à chaque nouvelle face, la couleur moyenne entre la face initiale et la face voisine par α_2 de cette arête. Par exemple sur la figure 1.47, on observe que la face issue de l'arête commune au triangle jaune et au carré rouge est de la couleur moyenne orange. De même, la face issue de l'arête commune entre le carré bleu et le carré rouge est de la couleur violette. Les faces issues d'arêtes du bord sont quand à elles de la même couleur que la face initiale. En effet, avec la présence de boucles sur les bords dans les G-cartes, les arêtes de bords sont voisines avec elles-mêmes. La couleur des nouvelles faces de bord est la moyenne entre le rouge et le rouge.

La figure 1.48 montre la carte généralisée du quadrangle triangulé avec les informations de couleur et de position géométrique. Nous avons conservé le code couleur des brins de la section

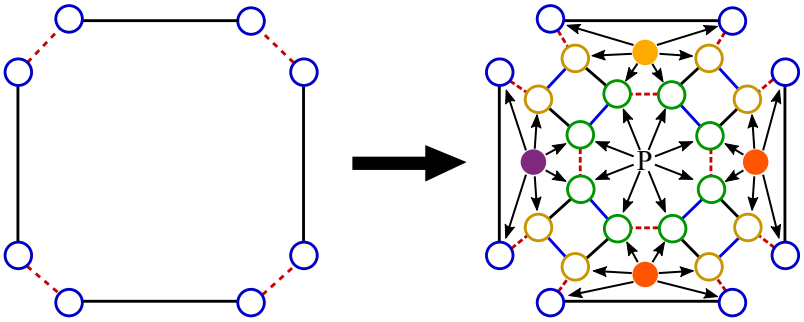


FIGURE 1.48 – Attribution des transformations de plongements aux brins du quadrangle de l'objet de la figure 1.47

précédente. On observe que seuls les brins verts portent une transformation de plongement, notée P, pour la position géométrique. En revanche, tous les brins portent une transformation de la couleur.

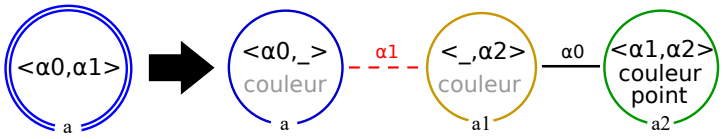


FIGURE 1.49 – Règle de triangulation portant les variables de plongement couleur et point

L'ajout de la description des transformations de plongements dans les règles se fait en étiquetant les nœuds des règles par des expressions. Pour décrire les expressions de plongement de la triangulation, nous étiquetons le nœud vert par une expression du plongement `point` et tous les nœuds par une expression du plongement `couleur`. Nous obtenons ainsi la règle de la figure 1.49. Par souci de lisibilité, seul le nom du plongement apparaît graphiquement sur le nœud de la règle, et non l'expression complète. Ces expressions sont écrites dans un langage formel défini dans [Bel12], mais qui n'est pas implanté dans l'outil. Ce langage formel est fonctionnel et contient des opérateurs dédiées à la manipulation du modèle topologique. Dans les faits, les expressions sont écrites en JAVA, le langage dans lequel est implanté Jerboa.

Commençons par écrire l'expression du plongement de couleur qui est la moyenne entre la couleur de la face filtrée par le nœud a et ses faces voisines par α_2. Dans le langage des expressions, les noms des nœuds du graphe gauche de la règle sont des variables pré-définies. Elles permettent d'accéder aux brins d'origine de l'objet depuis le motif droit de la règle. L'accès à un plongement de l'objet filtré se fait en utilisant le nom du plongement et une notation pointée. Ainsi par exemple, l'accès à la couleur de la face d'origine, représentée par le nœud a, s'écrit `a.couleur`. L'accès aux faces voisines par α_2 à la face d'origine, se fait avec la notation pointée suivante : $a.\alpha 2$. L'accès au plongement couleur de ces faces s'écrit donc $a.\alpha 2.couleur$. Pour mélanger les deux couleurs, nous avons besoin d'une méthode permettant de manipuler des instances du plongement `couleur`. Cette manipulation n'est pas spécifique à Jerboa, c'est une méthode externe. Dans le cadre d'une programmation orientée objet, cette méthode est une fonction membre de la classe du plongement `couleur`. Nous appelons cette méthode mix. On obtient donc l'expression suivante : $mix(a.couleur, a.\alpha 2.couleur)$. Cette méthode prend en paramètre deux instances du plongement `couleur` et retourne le mélange des deux. Les trois nœuds de la règle appartiennent à la même orbite face. Afin de satisfaire la condition de préservation de cohérence des plongements des G-cartes que nous avons vue plus avant, les trois nœuds de la règle portent cette même expression, car ils s'instancient en brins de la même orbite face qui porte le plongement couleur.

Lors de l'instanciation de la règle, l'expression est réécrite sur tous les brins de l'instance de la règle représentés par le nœud qui la porte. Pour chaque brin, l'expression réécrite correspond à l'expression portée par le nœud qui le représente et dans laquelle les noms des variables topologiques ont été substitués par le nom du brin qui lui correspond. La figure 1.50, montre la réécriture de l'expression. Le brin $a0$ par exemple, porte l'expression de couleur réécrite où la variable a a été substituée par $a0$. L'expression substituée est donc $mix(a0.couleur, a0.\alpha 2.couleur)$. Pour un autre brin, la substitution est différente. Pour le brin $a71$ par exemple, la variable a est substituée par $a7$.

Les expressions ré-écrites sont évaluées pour donner une valeur de plongement aux brins qui les portent. Ici, l'évaluation de l'expression est faite avec le morphisme $m1$ qui est le morphisme de filtrage sur la règle après l'étape d'instanciation topologique. L'expression de $a0$ est évaluée par :

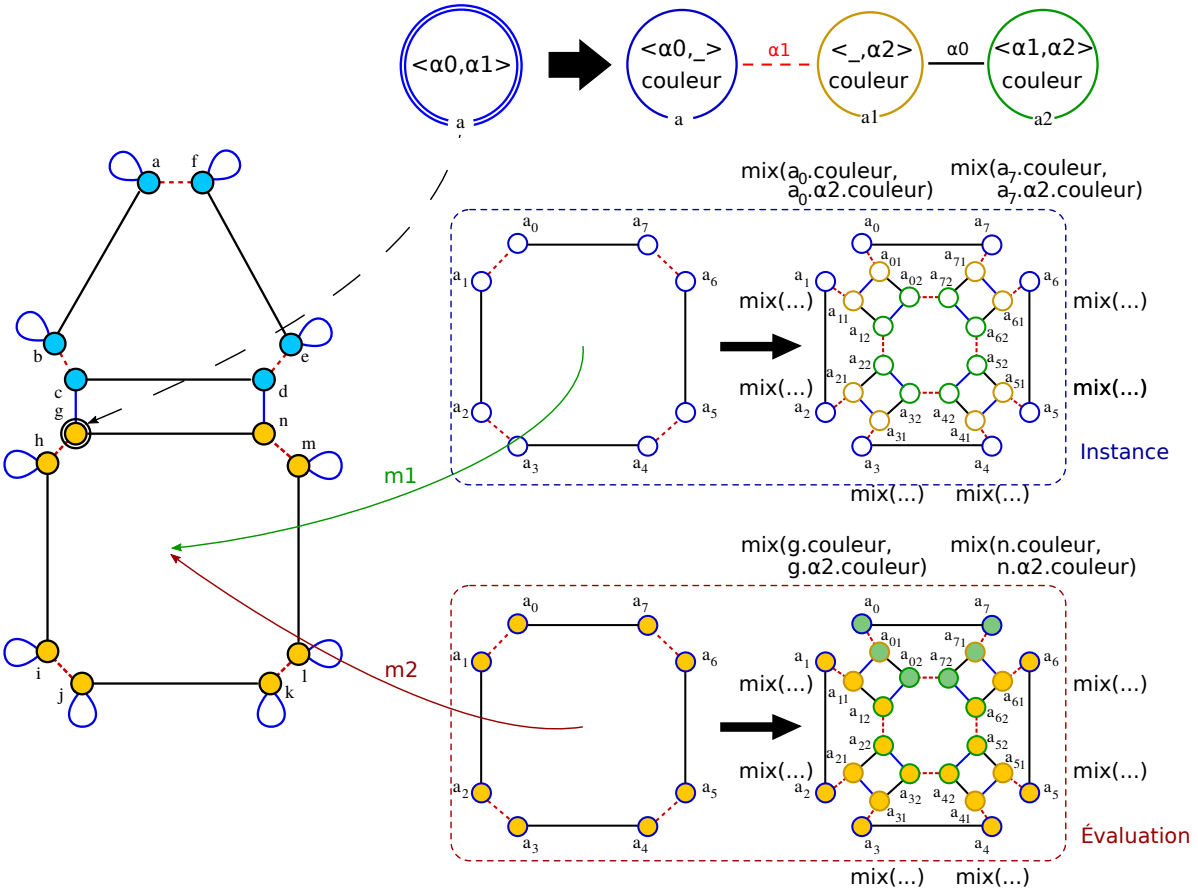


FIGURE 1.50

$mix(g.couleur, g.\alpha2.couleur)$ car $m1$ associe $a0$ au brin g de l'objet. Ainsi, lors de l'évaluation ce sont les couleurs du brin g de l'objet, et de son voisin par α_2 (brin c) qui sont utilisées, comme illustré dans le cadre "Évaluation" de la figure 1.50. Les copies instances $a01$ et $a02$ de $a0$ portent la même expression réécrite que $a0$ et une évaluation identique. En effet, la variable a est également substituée par $a0$, dans l'expression de $a01$ et $a02$.

Passons à présent à l'écriture de l'expression du plongement `point` portée par le nœud $a2$. Nous avons choisi de placer ce sommet au barycentre de la face triangulée. Le calcul du barycentre nécessite de connaître la position géométrique de chaque sommet de la face d'origine. Pour ce faire, il faut collecter les instances du plongement point de tous les sommets de la face d'origine représentée par le nœud a. Dans le langage Jerboa, la collecte de plongement s'écrit comme suit : $\langle 0,1\rangle_{point}(a)$, avec $\langle 0,1\rangle$, le type orbite sur laquelle sont collectés les plongements, a un brin (ou une variable topologique) appartenant à l'orbite de collecte, et *point* le nom du plongement à collecter. Pour calculer le barycentre de cette collecte, nous définissons une méthode externe nommée *barycentre* qui prend en entrée une collection de points et retourne leur barycentre. On obtient l'expression : $barycentre(\langle 0,1\rangle_{point}(a))$.

1.3.4 Condition de préservation de la cohérence

Dans la section précédente, nous avons écrit une règle Jerboa qui permet de décrire des transformations topologiques et des transformations de plongement. Cependant, l'application

d'une règle doit préserver les contraintes des cartes généralisées. Ainsi l'application d'une règle qui préserve ces contraintes, sur un objet dont la structure topologique est cohérente (i.e. une G-carte), produit un objet cohérent.

Pour garantir que l'application d'une règle préserve la cohérence des objets, il faut par exemple, qu'à l'issue des transformations, tous les brins aient une liaison pour chaque dimension (contrainte des arcs adjacents des cartes généralisées). Dans notre exemple d'utilisation d'une 2-G-carte, chaque brin doit porter une liaison α_0, α_1 et α_2. Pour vérifier que l'application d'une règle préserve cette contrainte, on peut distinguer plusieurs cas.

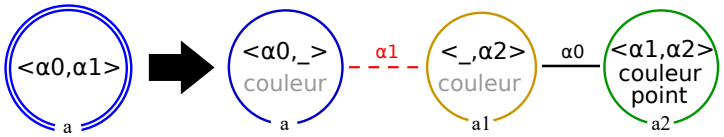


FIGURE 1.51 – Règle de triangulation barycentrique

Les nouveaux brins créés par une règle doivent porter une liaison par dimension. Ainsi, il en est de même pour leur représentant dans la règle, comme pour les nœuds $a1$ et $a2$ de la règle de triangulation montrée en figure 1.51. Tous les nœuds présents uniquement dans le graphe droit des règles doivent porter une liaison (implicite ou explicite) par dimension. Ici, le nœud jaune $a1$ porte deux liaisons explicites α_0 et α_1 et une liaison implicite α_2. De même, le nœud vert $a2$ porte un α_0 explicite et deux liaisons implicites α_1 et α_2.

Dans le cas des nœuds préservés par la règle, comme le nœud bleu a sur la figure 1.51, les liaisons non filtrées ne doivent pas être modifiées par la règles. Par exemple, les liaisons α_2 de a ne sont pas filtrées par le motif gauche. Il ne faut donc pas définir de transformation α_2 pour a dans la règle. Si une transformation d'une liaison non filtrée est définie, à l'issue de l'application de la règle, les brins représentés par le nœud concerné porteront plusieurs liaisons de même dimension, ce qui ne vérifie par la contrainte de cohérence. En revanche, toutes les liaisons filtrées (ici α_0 et α_1), doivent être présentes à droite. C'est le cas ici car la liaison α_0 du nœud bleu n'a pas été modifiée, et la liaison implicite α_1 qui a été supprimée, est remplacée par une liaison explicite avec le nœud jaune $a1$.

Pour les brins supprimés par l'application de la règle, il est nécessaire que le nœud qui les représente dans la règle, filtre toutes leurs liaisons. En effet, si un brin est supprimé alors que certaines de ses liaisons n'ont pas été filtrées, ses voisins par ces liaisons les conservent et présentent des arcs dits *pendants*, dont la cible n'existe plus. Pour éviter de créer des arcs pendants, tout nœud présent uniquement dans le graphe gauche des règles doit porter toutes les liaisons possibles, implicitement ou explicitement. Cette condition précise porte le nom de condition d'arcs pendants (ou *dangling condition*).

Nous ne nous attardons pas ici sur l'explication des autres conditions de préservation de cohérence topologique par les règles car ce n'est pas le propos de ce document, mais pour plus d'informations, elles sont expliquées dans la thèse de Mathieu Poudret[Pou09].

Pour garantir la cohérence des objets, l'application d'une règle doit également préserver la cohérence des plongements. Dans une G-carte plongée, tous les brins appartenant à la même orbite de plongement portent le même plongement. Pour que cette contrainte soit préservée par l'application d'une règle, toutes les instances des nœuds appartenant à la même orbite de plongement doivent porter des évaluations identiques de l'expression de plongement. Ainsi, tous les nœuds du graphe droit des règles appartenant à la même orbite de plongement doivent porter

la même expression de plongement.

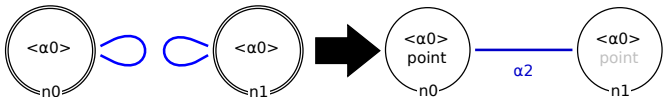
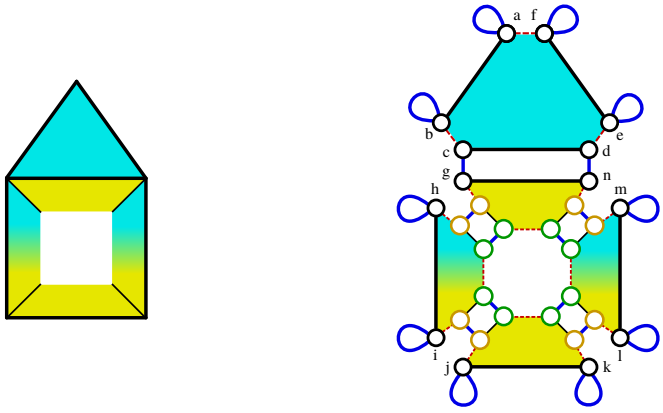


FIGURE 1.52 – Règle de couture d'arête

De plus, les transformations topologiques peuvent créer des incohérences de plongement. Par exemple, sur la règle de couture d'arête (cf. figure 1.52), deux arêtes sont liées par α_2. Les sommets de chaque arête avant application de la règle possèdent leur propre position géométrique. L'application de la règle lie ces sommets deux à deux par α_2. Or, le plongement `point` est portée par le type d'orbite $\langle \alpha_1, \alpha_2 \rangle$. Ainsi les brins reliés par l'application de la règle doivent porter le même plongement. Sans transformation de plongement, chaque sommet conserve sa propre position et l'objet est incohérent. Il faut donc définir la nouvelle position géométrique des nouveaux sommets formés par la liaison en α_2. Ici nous avons choisi de leur attribuer le milieu des deux sommets d'origine : $milieu(n0.point, n1.point)$.

Les deux faces partiellement filtrées par $n0$ et $n1$ restent, quand à elles, inchangées topologiquement. L'application de la règle ne produit donc pas d'incohérence pour le plongement `couleur`. Il n'est donc pas nécessaire de définir une expression de ce plongement pour $n0$ et $n1$ dans la règle de la figure 1.52. Précisons cependant qu'il est possible de leur attribuer une expression du plongement `couleur`, si nous souhaitons modifier la couleur des faces lors de la couture, mais ce n'est pas obligatoire.

Plus généralement, si une liaison explicite relie deux nœuds filtrés par la règle, dans une même orbite de plongement, ces nœuds doivent porter une expression de ce plongement.



(a) Objet (b) G-carte plongée

FIGURE 1.53 – Résultat de la triangulation sur le carré de la maison avec des termes de plongements évalués différemment par des brins appartenant à une même orbite de plongement

Ces vérifications ne sont cependant pas suffisantes pour garantir que tous les brins d'une même orbite de plongement portent la même valeur de plongement. En effet, il est possible qu'une même expression soit évaluée de deux façons différentes pour deux brins instances appartenant à la même orbite de plongement.

Gardons l'exemple de la triangulation barycentrique définie en figure 1.51 et modifions les termes des expressions des plongements *point* et *couleur* comme suit :

Point : $milieu(a.point, barycentre(\langle 0,1\rangle_{point}(a)))$

Couleur : $mix(a.couleur, a.\alpha_1.\alpha_2.couleur)$

Le résultat de l'application de la règle modifiée sur le carré de la maison est illustré en figure 1.53(a). On observe que les brins appartenant au nouveau sommet ne sont pas tous à la même position géométrique bien qu'ils appartiennent tous à la même orbite sommet. De même, deux des quatre nouvelles faces présentent non pas une mais deux couleurs. Ce résultat est dû à des évaluations différentes des termes de plongement des nœuds pour chacune de leurs instances appartenant à une même orbite de plongement.

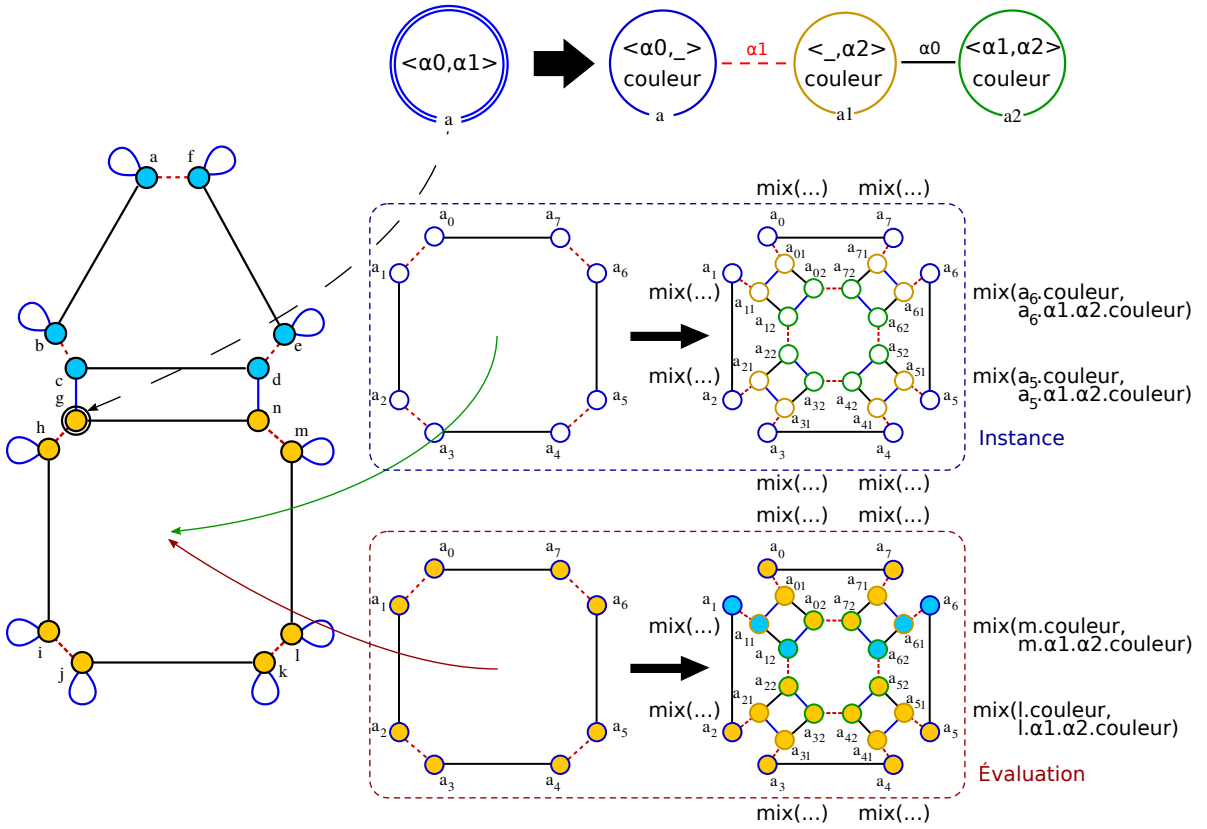


FIGURE 1.54 – Instanciation des expressions du plongement `couleur`

L'évaluation de l'expression du plongement `couleur` est illustrée en figure 1.54. On observe par exemple que l'évaluation est différente pour les brins $a5$ et $a6$ qui appartiennent à la même face. Les couleurs évaluées sur $a5$ et $a6$ sont respectivement jaune et bleu. Pour écrire une règle Jerboa correcte, nous devons donc vérifier que les expressions de plongements portées par ses nœuds ont toujours la même évaluation. Pour vérifier si un terme est évalué de la même manière sur toute l'orbite de plongement, il faut s'assurer que toutes les valeurs de plongements utilisées dans le terme sont équivalentes pour tous les brins qui le portent. L'équivalence entre deux valeurs de plongement a été défini dans [BABLG17]. L'équivalence de deux expressions de plongement est appelée équivalence à orbite près.

L'équivalence à orbite $\langle o\rangle$ près de deux termes t et t', notée $t \equiv_{\langle o\rangle} t'$ est définie syntaxiquement sur la forme des expressions. Par exemple $a \equiv_{\langle\alpha_0,\alpha_1\rangle} a.\alpha_0$, car pour tout brin a, $a.\alpha_0$ est évalué par un brin appartenant la même orbite $\langle\alpha_0,\alpha 1\rangle$ que a. Pour appliquer cette vérification d'équivalence

à orbite près des termes portés par $a5$ et $a6$, nous devons vérifier si deux termes portés par des représentants de a voisin par α_0 sont équivalent.

Prenons l'exemple du plongement `couleur` dont l'orbite support est $\langle \alpha_0, \alpha_1 \rangle$. Nous devons vérifier que deux brins voisins par α_0 ou par α_1 portent des expressions équivalentes. Commençons par vérifier si deux brins représentés par le nœud `a` (cf. figure 1.54) voisins par α_0 (comme `a5` et `a6` par exemple) portent des expressions équivalentes à l'orbite du plongement `couleur` près :

$$mix(a.couleur, a.\alpha_1.\alpha_2.couleur) \equiv_{\langle \alpha_0,\alpha_1 \rangle} mix(a.\alpha_0.couleur, a.\alpha_0.\alpha_1.\alpha_2.couleur)$$

La fonction mix est une fonction utilisateur, on ne peut faire aucune supposition sur son fonctionnement. Il faut donc vérifier si chacun de ses arguments vérifient deux à deux la condition précédente. On obtient le système suivant.

$$\begin{cases} a.couleur \equiv_{\langle \alpha_0,\alpha_1 \rangle} a.\alpha_0.couleur \\ a.\alpha_1.\alpha_2.couleur \equiv_{\langle \alpha_0,\alpha_1 \rangle} a.\alpha_0.\alpha_1.\alpha_2.couleur \end{cases}$$

On peut réduire les expressions obtenues en enlevant les accès aux plongements pour les expressions de type $x.plongement$, afin de vérifier si les brins qui les portent vérifient l'équivalence à orbite près.

$$\begin{cases} a \equiv_{\langle \alpha_0,\alpha_1 \rangle} a.\alpha_0 & (1) \\ a.\alpha_1.\alpha_2 \equiv_{\langle \alpha_0,\alpha_1 \rangle} a.\alpha_0.\alpha_1.\alpha_2 & (2) \end{cases}$$

L'équivalence (1) est vérifiée car α_0 est une étiquette de $\langle \alpha_0, \alpha_1 \rangle$.

Pour l'équivalence (2), poursuivons le raisonnement. α_2 n'appartient pas à l'orbite $\langle \alpha_0, \alpha_2 \rangle$, mais $\alpha_0\alpha_2\alpha_0\alpha_2$ forme un cycle. Donc l'équivalence (2) sera vérifiée si l'équivalence suivante l'est aussi.

$$a.\alpha_1 \equiv_{\langle \alpha_0 \rangle} a.\alpha_0.\alpha_1$$

De même, α_1 n'appartient pas à l'orbite $\langle \alpha_0 \rangle$, et cette dernière ne contient pas de liaison formant un cycle avec α_1. Donc l'équivalence (2) sera vérifiée si l'équivalence suivante l'est aussi.

$$a \equiv_{\langle \rangle} a.\alpha_0$$

α_0 n'appartient pas à $\langle \rangle$, donc l'équivalence n'est pas vérifiée. On peut à présent conclure que le système précédent n'est pas vérifié.

Ainsi, pour deux instances de a voisins par α_0, comme $a5$ et $a6$ sur la figure 1.54, l'expression du plongement couleur n'a pas la même évaluation. Les brins $a5$ et $a6$ ont ainsi deux couleurs différentes lors de l'évaluation, alors qu'ils appartiennent à la même orbite de plongement couleur.

L'autre liaison appartenant à l'orbite support du plongement `couleur` est α_1. On pourrait également vérifier si pour deux voisins par α_1 l'expression vérifie l'équivalence à orbite près. Le nœud a n'a pas de liaison implicite α_1, aucune vérification n'est nécessaire pour cette liaison. De même, le nœud $a1$ n'a ni de liaison implicite α_0, ni α_1, donc aucune vérification n'est nécessaire. Par contre, le nœud $a2$ a une liaison implicite qui nécessite une vérification. Or, les accès aux plongements se font toujours sur le motif filtré, l'objet avant transformation. En raison du renommage, la liaison implicite α_1 et $a2$ correspond à une liaison α_0 dans l'orbite filtrée par a. Pour vérifier si deux instances de $a2$ liées par α_1 ont la même valeur de couleur, il convient de vérifier si l'expression de couleur est équivalente pour deux instances filtrées par a liées par α_0, c'est à dire exactement l'équivalence déjà vérifiée pour le nœud a.

On peut réaliser la même vérification avec l'expression du plongement point portée par le nœud vert $a2$: $u(a) = milieu(a.point, barycentre(\langle 0,1\rangle_{point}(a)))$. L'orbite du plongement point est $\langle \alpha_1, \alpha_2 \rangle$. Le nœud $a2$ porte toutes les liaisons de cette orbite. En prenant en compte le renommage, il faut donc vérifier l'équivalence pour les liaisons α_0 et α_1. On obtient donc le système suivant.

$$\begin{cases} u(a) & \equiv_{\langle \alpha_1,\alpha_2 \rangle} u(a.\alpha_0) \\ u(a) & \equiv_{\langle \alpha_1,\alpha_2 \rangle} u(a.\alpha_1) \end{cases}$$

Ce système est vérifié si les arguments des fonctions utilisateur (*milieu* et *barycentre*) sont équivalents deux à deux, conformément au système suivant.

$$\begin{cases} a.point & \equiv_{\langle \alpha_1,\alpha_2 \rangle} a.\alpha_0.point \\ \langle 0,1\rangle_{point}(a) & \equiv_{\langle \alpha_1,\alpha_2 \rangle} \langle 0,1\rangle_{point}(a.\alpha_0) \\ a.point & \equiv_{\langle \alpha_1,\alpha_2 \rangle} a.\alpha_1.point \\ \langle 0,1\rangle_{point}(a) & \equiv_{\langle \alpha_1,\alpha_2 \rangle} \langle 0,1\rangle_{point}(a.\alpha_1) \end{cases}$$

Le traitement des collectes de plongement est un peu différent. Deux collectes sont équivalentes à orbite près si les brins à partir desquels elles sont faites, sont équivalents à l'orbite de collecte près. Par exemple, deux collectes sur l'orbite face $\langle \alpha_0, \alpha_1 \rangle$ sont équivalentes si leurs brins supports respectifs appartiennent à la même face. Le système précédent est donc vérifié si le suivant l'est aussi.

$$\begin{cases} a \equiv_{\langle \alpha_1,\alpha_2 \rangle} a.\alpha_0 & (1) \\ a \equiv_{\langle \alpha_0,\alpha_1 \rangle} a.\alpha_0 & (2) \\ a \equiv_{\langle \alpha_1,\alpha_2 \rangle} a.\alpha_1 & (3) \\ a \equiv_{\langle \alpha_0,\alpha_1 \rangle} a.\alpha_1 & (4) \end{cases}$$

L'équivalence (1) n'est pas vérifiée car α_0 n'est pas une étiquette de $\langle \alpha_1, \alpha_2 \rangle$. Ainsi même si les trois équivalences suivantes sont vérifiées, le système ne l'est pas. Ceci explique les problèmes de cohérence soulevés plus avant sur la figure 1.53.

1.3.5 Architecture et fonctionnement

Jerboa est une bibliothèque de modélisation à base topologique écrite en JAVA. Elle permet de manipuler des objets et leurs plongements quelle que soit la dimension topologique de travail. Elle permet également de définir autant de plongements que nécessaire, et ce quel que soient leurs types et leurs orbites.

La bibliothèque comprend un éditeur de modeleur qui permet de définir non seulement le modèle des objets manipulés mais aussi les opérations de manipulation, à l'aide des règles de transformation précédentes. Elle comprend également un noyau d'application des règles qui permet de manipuler les objets en utilisant les opérations définies via l'éditeur. L'utilisateur n'est pas un développeur, il n'a pas besoin de connaître Jerboa et son langage de règles, mais doit maîtriser son domaine d'application. Dans le cadre applicatif de cette thèse par exemple, les utilisateurs du modeleur dédié à la géologie que nous avons implanté sont des géologues.

La figure 1.55 montre l'organisation de la bibliothèque Jerboa. Sur cette figure, les éléments automatisés par la librairie sont marqués d'une roue crantée, et les éléments apportés par l'utilisateur sont marqués par un personnage. Le vert représente le développeur, la personne qui va créer le modeleur dédié. Le personnage rayé représente l'utilisateur final, du modeleur généré.

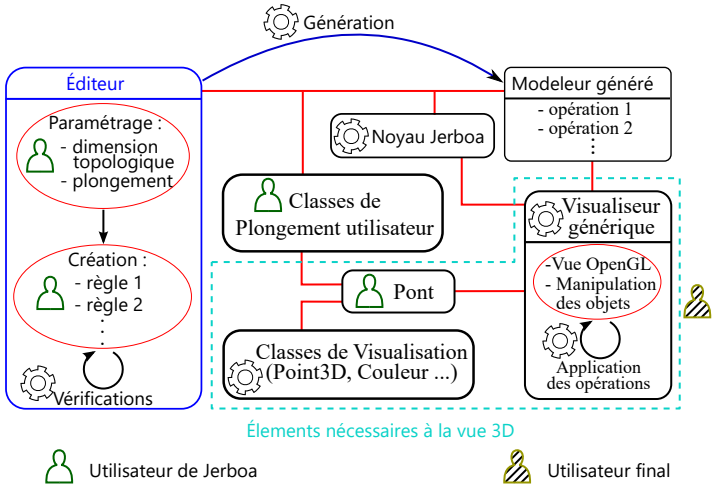


FIGURE 1.55 – Organisation de Jerboa

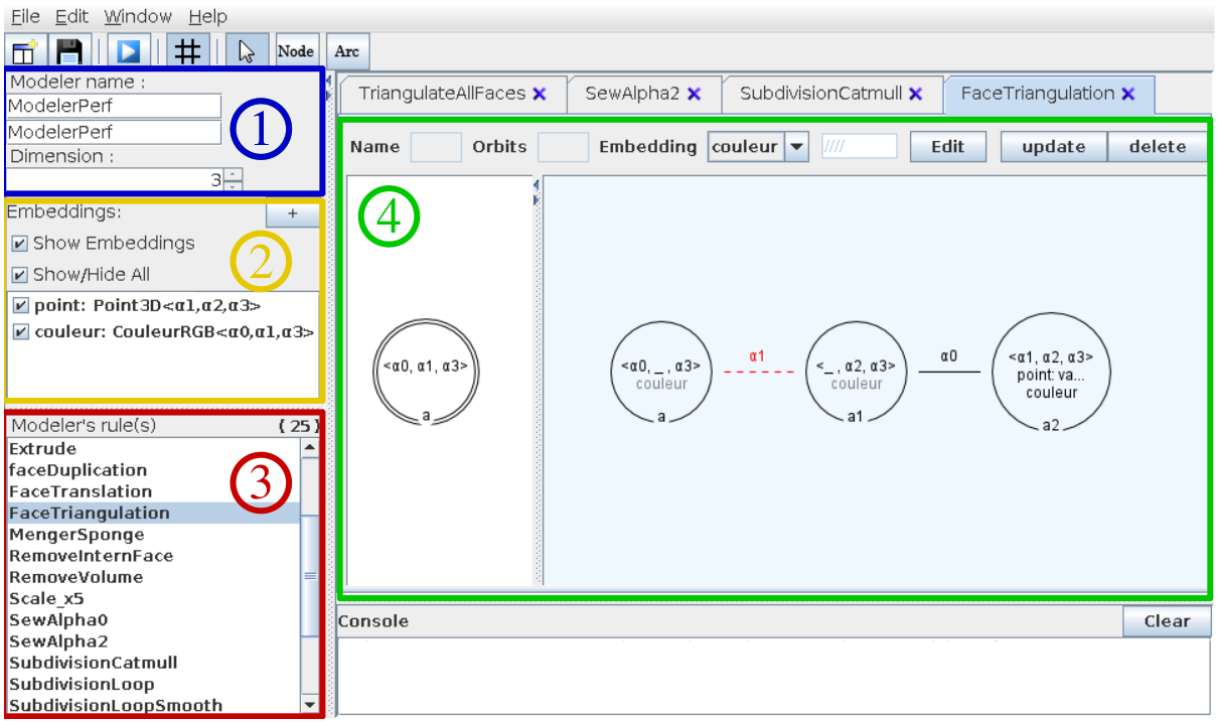


FIGURE 1.56 – Éditeur de modeleur

Avec Jerboa, le programmeur n'intervient que sur la définition des opérations. Le noyau d'application de Jerboa fonctionne pour n'importe quelle règle définie via l'éditeur, quel que soit le nombre ou la nature des plongements définis dans le modeleur. L'utilisateur doit simplement implanter ses classes de plongement et définir ses opérations graphiquement via l'éditeur.

La figure 1.56 montre l'interface de l'éditeur de modeleur. L'éditeur permet de définir des modeleurs. Chaque modeleur porte un nom et est défini dans une dimension topologique particulière comme illustré dans le cadre 1 de la figure 1.56. Ici le modeleur est appelé "ModelerPerf" et permet de travailler en dimension 3.

Les modeleurs peuvent présenter autant de plongements que nécessaire. Chaque plongement

possède un nom, un type et une orbite support comme illustré dans le cadre 2 de la figure. Ici, le modeleur présente deux plongements : *point* : $\langle 1, 2, 3 \rangle \rightarrow Point3D$ et *couleur* : $\langle 0, 1, 3 \rangle \rightarrow CouleurRGB$. Notons que le type des plongements doit être une classe d'objet programmée en Java, le langage du noyau d'application des règles.

L'éditeur permet également de définir les opérations des modeleurs. Elles sont listées dans la zone du cadre 3 (rouge) de la figure. Ici le modeleur comporte par exemple des règles appelées "Extrude", "faceDuplication", *etc.*

Les opérations sont définies par des règles Jerboa, chacune décrite par deux graphes. La partie droite de l'éditeur (cadre 4 vert) comporte des outils permettant de les décrire graphiquement. Sur la figure, la règle décrite est la règle de triangulation barycentrique que nous avons définie dans la section précédente, mais en dimension 3. Pour chaque graphe il est possible de créer, supprimer et nommer des nœuds, ainsi que de définir leurs relations topologiques en les liant entre eux par des arcs. Chacun d'eux peut être étiqueté par un type d'orbite. Les nœuds du graphe droit peuvent également porter des expressions de plongement. Comme les types de plongement, ces expressions doivent être écrites en Java.

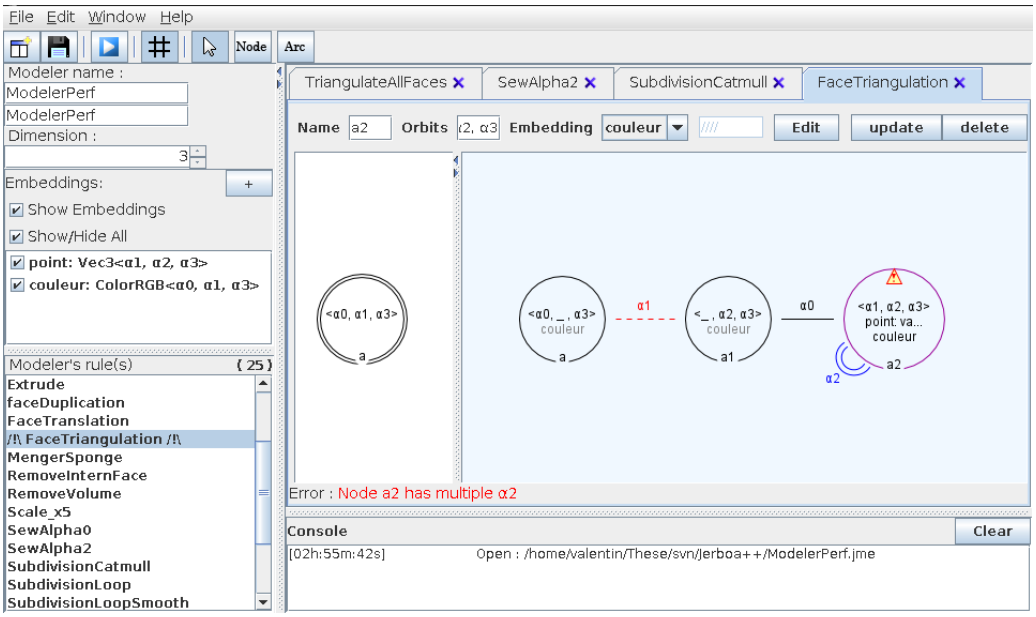


FIGURE 1.57 – Erreur topologique détectée à la volée par l'éditeur

Pendant la conception des règles, l'éditeur vérifie automatiquement à la volée la préservation de la cohérence par les règles. Un exemple d'erreur est illustrée en figure 1.57. Ici l'éditeur a détecté que le nœud $a2$ présente un arc implicite de dimension 2, mais également un arc explicite de la même dimension. L'application de cette règle crée des brins présentant plusieurs liaisons α_2, ce qui viole la contrainte d'arc incident des G-cartes.

L'éditeur vérifie également que les expressions de plongements sont correctement définies dans les nœuds. Comme nous l'avons vu dans la section précédente, tous les nœuds appartenant à une même orbite de plongement doivent porter la même expression de plongement. Afin d'éviter à l'utilisateur de recopier manuellement ces expressions, l'éditeur propage automatiquement une expression définie sur un nœud à tous les nœuds appartenant à la même orbite de plongement. Les expressions propagées sont affichées en gris, et les expressions définies par l'utilisateur en noir. C'est notamment le cas de l'expression du plongement *couleur* qui est définie sur le nœud

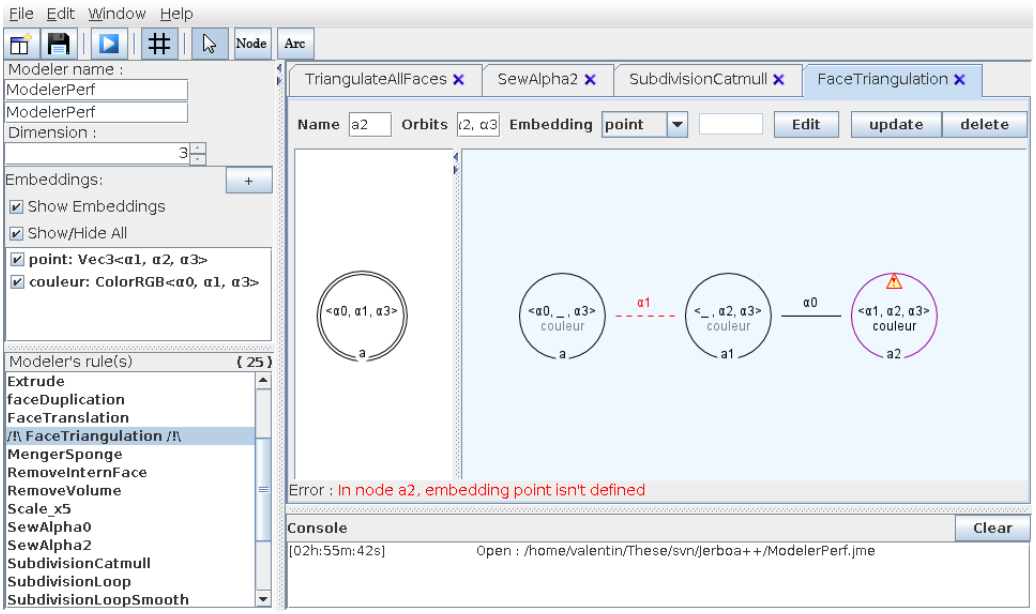


FIGURE 1.58 – Erreur de plongement détectée à la volée par l'éditeur

$a2$ et propagée sur les nœuds a et $a1$ de la figure 1.58.

L'éditeur vérifie également que tous les plongements nécessaires sont bien définis comme nous l'avons vu à la section 1.3.4. Lors de la création d'un nouveau sommet par exemple, il est nécessaire de lui attribuer une expression de plongement. Sur la figure 1.58, l'éditeur a détecté que l'expression du plongement *point* n'a pas été défini pour le nouveau sommet créé par $a2$. Dans le cas où un nœud ajouté (présent uniquement à droite) appartient à la même orbite de plongement qu'un nœud préservé (présent des deux côtés), il n'est pas obligatoire de lui attribuer une expression de plongement. Pour la règle de triangulation par exemple, Jerboa autorise de ne pas définir d'expression couleur sur les nœuds. En effet, les nouvelles faces partagent toutes le nœud a, qui à gauche filtre la face d'origine. Dans ce cas, si aucune couleur n'est spécifiée, lors de l'application de la règle, Jerboa propage automatiquement la couleur de la face filtrée à toutes les nouvelles faces, et donc aux brins représentés par $a1$ et $a2$. En revanche, les expressions étant définies en Java, l'éditeur ne peux pas vérifier l'équivalence à orbite près. Cependant, le moteur d'application assure quand même que les objets créés vérifient la contrainte de cohérence des plongements. En effet, si théoriquement les expressions sont évaluées sur chaque brin instance de la règle, en pratique le moteur évalue une seule expression, et propage l'instance de plongement sur toute l'orbite de plongement. Ceci permet d'éviter des calculs inutiles (car tous les brins d'une même orbite de plongement sont censés porter le même plongement), et assure également la cohérence des objets.

Pour être utilisées par le moteur d'application de Jerboa, les règles définies dans l'éditeur doivent être transformées dans le langage cible. Précisément, l'éditeur génère automatiquement du code JAVA à partir des paramètres du modeleur, des plongements et des règles éditées graphiquement. Le code engendré doit ensuite être compilé avec le moteur d'application, pour être utilisé. Afin de pouvoir lancer l'application des règles, charger des objets, *etc.*, l'utilisateur doit simplement programmer sa fonction de programme principal. Le moteur d'application fournit toutes les fonctionnalités nécessaires pour utiliser le modeleur produit.

Jerboa fournit une interface graphique par défaut, qu'il n'est pas obligatoire d'utiliser. Elle permet d'utiliser n'importe quel modeleur défini par l'éditeur, et de visualiser les objets dans un

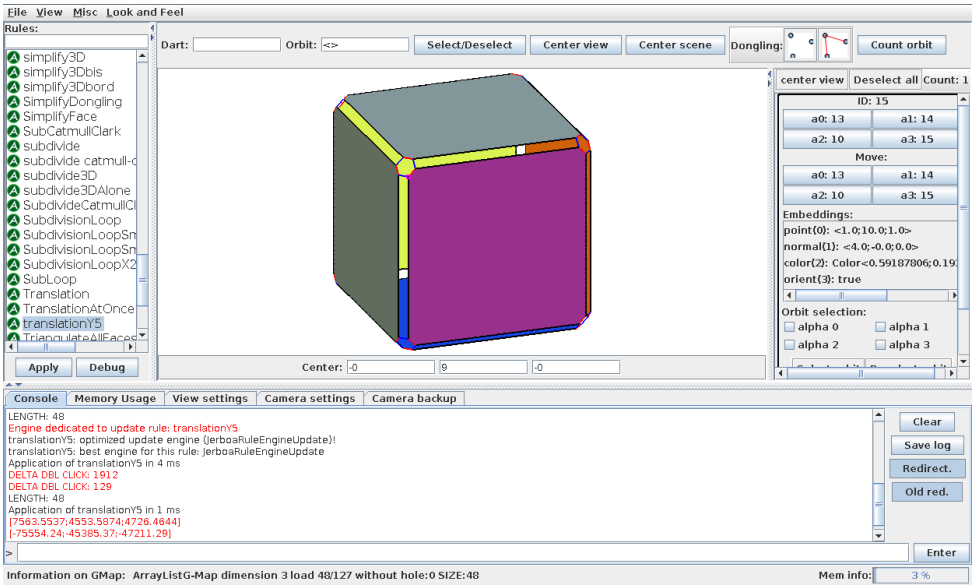
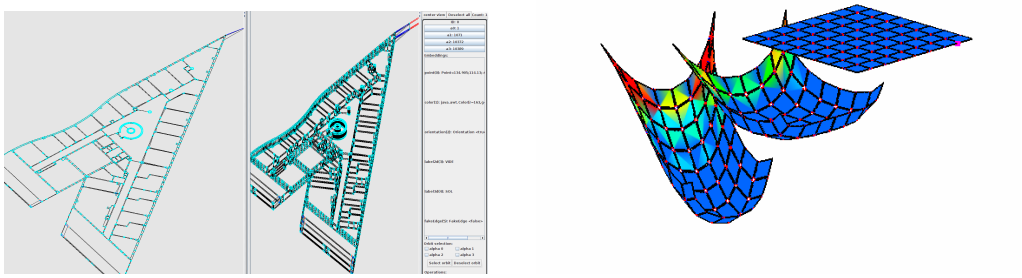


FIGURE 1.59 – Interface graphique 3D par défaut de Jerboa

espace plongé en 3D, comme illustré sur la figure 1.59.

Pour visualiser les objets, des classes dédiées sont incluses dans Jerboa (cf. encadré pointillés turquoise de la figure 1.55). Elles sont utilisées dans le visualiseur pour réaliser un rendu 3D, et colorer les objets. Il n'est cependant pas obligatoire d'utiliser ces classes pour plonger les objets d'un modeleur. En revanche il est nécessaire de définir un *pont* qui fait l'association entre les classes de plongement du modeleur et celles du visualiseur. Le visualiseur présente une interface appelée `Bridge`. L'utilisateur doit donc créer sa propre classe qui implémente cette interface. Elle comprends diverses fonctions comme `Point3D coord(JerboaDart)`. Cette fonction prend en paramètre un brin de la G-carte et retourne une instance de la classe *Point3D* utilisée par le visualiseur pour positionner les éléments dans l'espace. Ainsi, si la position géométrique est définie par une classe implantée par l'utilisateur lui-même, il doit définir cette fonction en transformant une instance de sa classe en celle du visualiseur. Avec le pont, le visualiseur peut être utilisé pour manipuler les objets, et leur appliquer les opérations définies dans l'éditeur. Un panneau contenant toutes les opérations du modeleur est présent dans la partie gauche du visualiseur, comme illustré en figure 1.59. Le panneau central montre une vue OpenGL des objets présents dans la G-carte, et le panneau de droite montre les informations relatives aux brins sélectionnés dans la scène. Pour appliquer une opération, il est nécessaire de sélectionner par un clic les brins sur lesquels elle s'applique. Les brins sélectionnés définissent les brins accrochés des règles, comme nous l'avons vu dans la section 1.3.2.

Notons que le moteur d'application et le visualiseur sont deux bibliothèques distinctes. Le moteur peut ainsi être utilisé sans l'interface par défaut. Il est ainsi possible de définir facilement une autre interface graphique répondant à des besoins spécifiques, sans être gêné par l'interface par défaut.



(a) JerboaArchi : projet d'architecture (b) JAPHY : projet de simulation physique

FIGURE 1.60 – Projets réalisés avec Jerboa

1.3.6 Bilan

Les domaines d'applications de Jerboa sont très variés. Nous avons pu utiliser la bibliothèque dans différents projets applicatifs. Jerboa a par exemple été utilisé dans le domaine de l'architecture comme on peut le voir en figure 1.60(a). Un projet appelé 'Jermination', dont on voit des images dans la Section 1.2.6 dédiée au L-Systèmes (cf. figure 1.22 et figure 1.23), a été réalisé en combinant les G-cartes L-systèmes avec Jerboa. Dans ce projet, la grammaire L-système est interprétée par l'application et transcrite en des applications successives de règles définies via l'éditeur au préalable.

Jerboa a également été utilisée pour faire de la simulation physique, en modélisant des déformations d'objets comme illustré en figure 1.60(b). Cette image illustre le travail de Fatma BenSallah [BSBAM16, BSBAM17a, BSBAM17b]. Ici l'utilisation de Jerboa a permis de réaliser une étude complète de la simulation physique à base topologique pour différents modèles physiques (masse-ressort, *etc.*) et différents modèles topologiques (2D ou 3D et triangles/tétraèdres, quadrangulaires/pavés ou mixtes). Cette étude a portée plus spécifiquement sur la localisation (les orbites) de chaque donnée physique. Jerboa a facilité cette étude, car la bibliothèque permet d'associer les plongements à une grande variété d'orbites et de prototyper très rapidement les opérations qui s'appliquent sur des motifs variés (grâce au filtrage et aux variables topologiques) et manipulent un grand nombre de plongements (chacun géré par une expression de plongement différente).

Les projets ont été réalisés aussi bien par les contributeurs de cette librairie que par des collègues et étudiants extérieurs au projet. Les résultats donnés par les projets étudiants ont été assez encourageants. Ils ont réussi à utiliser la librairie facilement après une phase de prise en main de quelques semaines. Comme tout nouveau langage, Jerboa nécessite un temps d'apprentissage, mais qui est largement compensé par le temps gagné lors de la définition des opérations et leurs vérifications automatiques de l'éditeur.

La définition des opérations a été formalisée par les thèses de [Pou09] et [Bel12]. Les expressions de plongements sont définies formellement par un langage abstrait dédié. Cependant, ce langage n'existait pas concrètement et les expressions de plongement étaient définies en Java dans l'éditeur. Il était ainsi compliqué de réaliser la vérification automatique de la préservation de la cohérence des plongements. Comme nous l'avons vu dans la section précédente, l'éditeur ne teste pas l'équivalence à orbite près. Un des travaux de cette thèse a été de définir un langage concret permettant d'écrire ces expressions de plongement, d'effectuer des vérifications de préservation

de la cohérence, et de générer le code correspondant.

De même les différents projets réalisés ont mis en avant l'impossibilité d'écrire certaines opérations complexes avec une seule règle de transformation de graphe. Il était jusqu'alors nécessaire de programmer des opérations manuellement (dans le langage sous-jacent), afin de composer différentes règles. Un exemple type d'opération non réalisable avec les règles de base est la découture d'un volume 3D. Si aucune face du volume n'est au bord de l'objet, alors elles peuvent toutes être décousues simultanément par une seule règle. Mais au contraire, si certaines de ces faces sont bord de l'objet, il convient de considérer séparément chaque face et ne découdre que celles qui en ont besoin. Il était donc nécessaire de définir un langage de *script* permettant de composer les opérations définies par les règles. Nous décrirons dans la suite du manuscrit ce langage qui est une extension du langage d'expressions de plongement.

Chapitre 2

Extension du langage Jerboa

La programmation d'opérations géométriques peut être une tâche complexe. L'utilisation d'un modèle topologique permet de définir la cohérence topologique des objets, et l'utilisation de règles de transformation de graphe facilite la définition des opérations grâce d'une part à un langage graphique et d'autre part à des vérifications automatiques de préservation de la cohérence. Dans ce chapitre, nous présentons un langage dédié au calcul des plongements et à la réalisation d'opérations géométriques complexes qui utilisent plusieurs règles Jerboa. Nous présentons également les différentes vérifications, notamment de préservation de la cohérence des plongements, que nous avons mises en œuvre grâce à ce langage.

2.1 Cycle d'exécution des règles

Comme nous l'avons vu dans le chapitre précédent, les règles comportent deux graphes qui définissent le motif à transformer et la transformation proprement dite. Elles contiennent également d'autres éléments, comme les pré-conditions, déjà identifiés dans la thèse de Thomas Bellet [Bel12]. Mais nous en avons définis de nouveaux dont l'utilité a été révélée par plusieurs projets applicatifs. Nous allons à présent détailler les différents éléments qui composent une règle.

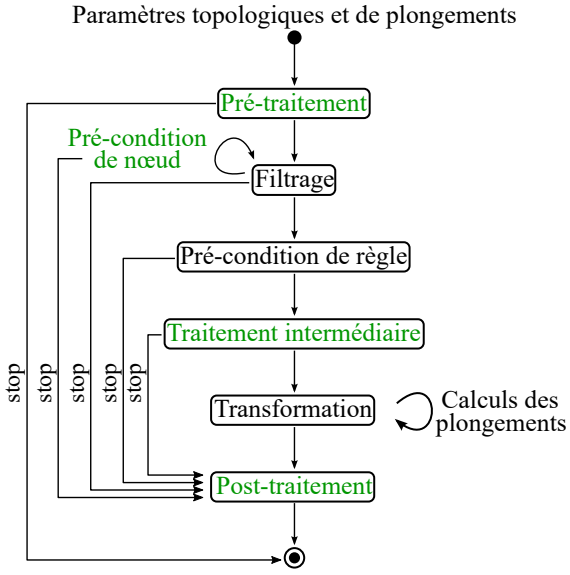


FIGURE 2.1 – Nouveau cycle d'exécution de l'application d'une règle dans Jerboa

L'ordonnancement des différents traitements est illustré en figure 2.1. Sur la figure, les nouveaux éléments ont été mis en vert. Ce nouveau flot d'exécution est plus transparent pour l'utilisateur et lui offre la possibilité d'intervenir aux moments clés de l'application d'une règle. A l'exception du post-traitement, l'utilisateur peut interrompre le traitement de l'opération. Une interruption n'entraîne aucune modification de la G-carte.

Les paramètres topologiques : Comme nous l'avons vu précédemment, les nœuds d'accroches sont explicités dans le graphe gauche de la règle. Ils définissent les paramètres topologiques formels de la règle. Au moment de l'application, un paramètre effectif (un brin accroché) est associé à chacun d'eux. Ces derniers sont ordonnés. Cet ordre explicite défini l'ordre dans lequel l'utilisateur doit sélectionner ses brins afin d'appliquer la règle.

Les paramètres de plongement : Des paramètres nécessaires à certains calculs de plongement de la règle peuvent être définis. Ces paramètres sont ordonnés, typés et nommés explicitement. Ils sont accessibles à tout moment de l'application de la règle. On peut leur affecter une valeur par défaut qui leur est allouée à l'instanciation de la règle. Si leur valeur est modifiée lors de l'application de la règle (valeur passée en paramètre de l'application de la règle), cette nouvelle valeur sera conservée pour l'application suivante de la règle.

Le pré-traitement : Avant l'étape de filtrage du motif gauche, il est parfois nécessaire de réaliser un pré-traitement. Ce peut par exemple être un calcul devant être vérifié ou effectué directement à partir des brins accrochés. Cette étape permet d'annuler si besoin l'application de la règle avant l'étape de filtrage. Par exemple, pour une règle qui doit s'appliquer sur un objet qui porte une étiquette particulière, il est possible à cette étape de tester l'étiquette sur les brins accrochés avant même d'effectuer l'étape de filtrage. Dans le cas de la géologie, on souhaite par exemple appliquer certaines opérations uniquement sur des surfaces qui représentent des failles. Dans un pré-traitement, on peut tester lors de l'application de la règle si la surface sur laquelle elle s'applique est effectivement une faille. Dans le cas contraire, il est possible d'interrompre l'application de la règle.

Le filtrage : Le motif gauche de la règle est calculé à l'étape de filtrage, à partir des brins accrochés et des orbites des nœuds de la règle leur correspondant. Les transformations effectuées lors de l'application d'une règle sont appliquées sur le motif filtré. Cette étape a été détaillée plus précisément dans les sections précédentes (cf. section 1.3, page 19).

La pré-condition de nœud : Ces pré-conditions sont des optimisations particulières des pré-conditions classiques. Considérons le cas de l'application d'une opération sur une face rouge. En temps normal, le filtrage parcourt toute la face avec succès avant de passer à l'étape de pré-condition classique qui vérifie la couleur. Si la couleur n'est pas la bonne, le moteur a filtré tout le motif alors qu'il était inutile de le faire complètement pour déterminer que la règle ne doit pas s'appliquer. Les pré-conditions de nœud répondent à ce besoin en vérifiant à la volée, pendant le filtrage, des propriétés sur les brins parcourus. Dans notre exemple, dès qu'un brin n'ayant pas la bonne couleur est rencontré lors du parcours, le filtrage de la règle échoue, évitant ainsi une perte de temps et de ressources inutiles. L'aspect dynamique de ces vérifications imposent de faire attention à la condition à vérifier pour éviter une diminution des performances du filtrage.

La pré-condition de règle : Introduite dans la thèse de Thomas Bellet [Bel12], c'est une expression booléenne qui dépend des plongements et de la topologie de l'objet sur lequel la règle s'applique. Par exemple, si un schéma de subdivision s'applique uniquement sur des faces quadrangulaires, la pré-condition permet de vérifier si la face sur laquelle la règle s'applique est bien un quadrangle.

Le traitement intermédiaire : Ce traitement permet de réaliser certains (pré-)calculs avant l'application de la règle mais après le filtrage du motif gauche. Les calculs réalisés ici ont donc la possibilité d'utiliser le motif filtré pour réaliser différents traitements. Ils ont généralement pour objet d'optimiser le calcul des plongements. À cette étape, la pré-condition a été évaluée et validée.

La transformation : Cette étape correspond à la transformation effective du graphe. Comme nous l'avons décrit dans la section précédente, elle effectue les transformations topologiques et la mise à jour des plongements en accord avec le graphe droit de la règle.

Le post-traitement : De même qu'il est possible de réaliser un traitement spécifique avant l'application d'une règle il est possible de réaliser d'autres traitements après son application, comme le nettoyage des paramètres de plongement par exemple. Ces traitements sont réalisés quoi qu'il arrive si le pré-traitement a réussi, même si le filtrage a échoué, afin d'être en mesure de libérer de la mémoire allouée, si besoin.

Pour réaliser les traitements utilisateurs au sein des différentes étapes, nous avons défini un langage offrant des éléments syntaxiques pour manipuler aisément la topologie et les plongements. Ce langage se base sur le langage abstrait défini par Thomas Bellet [Bel12].

2.2 Langage de calcul

Cette section présente le langage de calcul des expressions qui a été mis en place pour réaliser les calculs de plongement et pour définir les différents processus exécutés lors de l'application d'une règle. Dans la version initiale de Jerboa, la définition des expressions de plongement permettant de mettre à jour les informations géométriques et physiques des objets étaient directement écrites en Java. Si l'utilisation de règles de transformation de graphe permet de garantir la préservation de la cohérence topologique des objets, la définition des expressions de plongement dans le langage sous-jacent ne le permet pas pour les plongements.

Pour être en mesure de vérifier cette cohérence via l'éditeur de modeleur, nous avons défini un langage concret qui englobe le langage abstrait défini par Thomas Bellet. La plate-forme Jerboa ayant été programmé en Java, nous avons choisi d'implanter un langage concret impératif avec une syntaxe proche du Java et du C++, contrairement au langage abstrait qui est fonctionnel. Dans le langage abstrait il existe des opérateurs spécifiques, comme la collecte de brins, l'accès aux plongements ou au voisinage topologique. Nous avons donc ajouté ces opérateurs et instructions spécifiques dans notre langage. Les différents opérateurs spécifiques que nous avons définis sont présentés en Table 2.1. Pour plus de détails, la grammaire BNF (forme de Backus-Naur) complète du langage, est donnée en annexe (cf. annexe A, page 175). Voyons un peu plus en détail l'utilisation des opérateurs spécifiques à Jerboa.

Opérateur/instruction	Sémantique
`b@i`	Opérateur d'accès au voisin topologique α_i du brin `b`
`collection#i`	Accès au i^e élément d'une collection/liste
`b.point`	Opérateur d'accès au plongement plongement `point` du brin `b`
`<0,1,3>`	Type d'orbite
`<0,1,3>(n)`	Collecte de tous les brins de l'orbite `<0,1,3>` de `n`
`<0,1,2,3>_<0,1,3>(n)`	Collecte un brin par sous-orbite `<0,1,3>` de l'orbite `<0,1,2,3>(n)`
`<0,1,3>_point(n)`	Collecte les plongements `point` de l'orbite `<0,1,3>(n)`
`@ebd<point>`	Type (classe) du plongement `point`

TABLE 2.1 – Opérateurs et instructions spécifiques à Jerboa définis dans le langage de calcul d'expressions de plongement

`b@i :` L'opérateur `@` permet d'accéder au voisin topologique par une liaison α_i d'un brin `b`. Ainsi, si deux brins `b1` et `b2` sont liés par une liaison α_2, `b1@2` représente le brin `b2`. La dimension de voisinage i doit évidement être comprise entre 0 et la dimension topologique du modeleur.

`b.point :` L'accès à la valeur d'un plongement d'un brin `b` se fait comme l'accès à l'attribut d'une classe, par une notation pointée. Ici l'expression donne la valeur du plongement `point` du brin `b`.

`@ebd<point> :` Cet opérateur permet d'abstraire le type du plongement. Il peut être utilisé comme le type du plongement, pour déclarer une variable ou en créer une instance : `@ebd<point> p = new @ebd<point>(1,0,0)`. Il est également possible d'accéder aux informations de ce plongement comme son orbite support : `@ebd<point>.orbit`; ou son nom : `@ebd<point>.name`.

Cet opérateur permet également de changer le type des plongements d'un modeleur sans avoir à modifier les expressions de plongement de ses règles, à condition que les méthodes de manipulation de ces types aient les mêmes signatures dans le langage cible. Par exemple, il est possible de changer le type de notre plongement `point` qui est la classe `Point3D`, par une classe `Point2D` pour passer d'un plongement d'un espace 3D à un 2D. Dans ce cas, cette dernière doit comporter une méthode `barycentre()` qui nous permet de calculer la position du centre d'une liste de points. Dans le cas contraire, l'expression de ce plongement dans la règle de triangulation barycentrique que nous avons défini dans la section 1.3.3 deviendrait incohérente.

`<0,1,3> :` Les types d'orbites sont définies comme dans le langage abstrait. Les dimensions sont spécifiées entre chevrons et doivent être comprises entre 0 et la dimension topologique du modeleur. Notons que par convention, nous écrivons les dimensions dans les orbites dans l'ordre croissant, mais le langage accepte les dimensions dans un ordre quelconque.

`<0,1,3>(b) :` Une collecte permet d'obtenir une collection de brins appartenant à une orbite donnée. Cet exemple collecte tous les brins de l'orbite $\langle 0, 1, 3\rangle$ incidente au brin `b`. Notons que dans Jerboa, les collectes retournent les brins dans un ordre dépendant de l'implantation. Il est donc déconseillé de supposer un ordre particulier lors des traitements.

`<0,1,2,3>_<0,1,3>(b) :` Il s'agit de la collecte d'un brin par sous-orbite dans une orbite donnée. Dans cet exemple, la collecte parcourt toute l'orbite principale $\langle 0, 1, 2, 3\rangle$ du brin `b` et retourne

un unique brin par sous-orbite $\langle 0, 1, 3 \rangle$. En particulier, cet exemple collecte un brin par face, sur toutes les faces d'une composante connexe.

`<0,1,3>_point(b)` : La collecte de plongements permet d'obtenir toutes les instances d'un plongement sur une orbite donnée. Ici la collecte donne une collection contenant toutes les positions géométriques de la face du brin `b`.

Ce langage comprend également les instructions classiques des langages impératifs comme les alternatives, les boucles (`while`, `for`, `foreach`), les déclarations de variables, les opérateurs de calculs, *etc.* Les alternatives sont écrites de façon similaire au Java et C++ :

```
if([CONDITION]){
    [INSTRUCTION];
}else{
    [INSTRUCTION];
}
```

Les boucles sont également écrites de façon classiques :

```
while([CONDITION]){
    [INSTRUCTION];
}
for([TYPE] [VARIABLE] : [COLLECTION]){ // foreach
    [INSTRUCTION];
}
for([TYPE] [VARIABLE]; [CONDITION]; [INSTRUCTION]){
    // for à la JAVA et C++
    [INSTRUCTION];
}
for i in [DEBUT]..[FIN] { // for à la ADA
    [INSTRUCTION];
}
for i in [DEBUT]..[FIN] step [ENTIER] {
    // step permet de définir le pas d'incrémentation
    [INSTRUCTION];
}
```

Le langage concret étant impératif, le résultat du calcul des expressions de plongement et autres éléments est donné comme un retour de fonction avec le mot clef `return`.

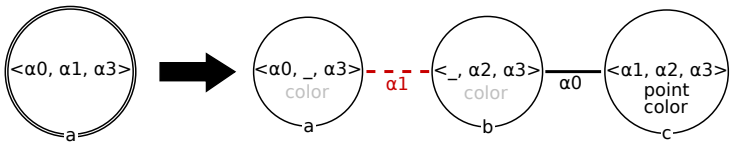


FIGURE 2.2 – Règle Jerboa de triangulation barycentrique 3D

Voyons à présent quelques exemples d'utilisation de ces opérateurs et instructions dédiées. Considérons la règle de triangulation barycentrique pour une 3-G-carte comme illustré en figure 2.2. L'application de cette règle triangule une face, et colore nouveaux triangles par un mélange de couleur.

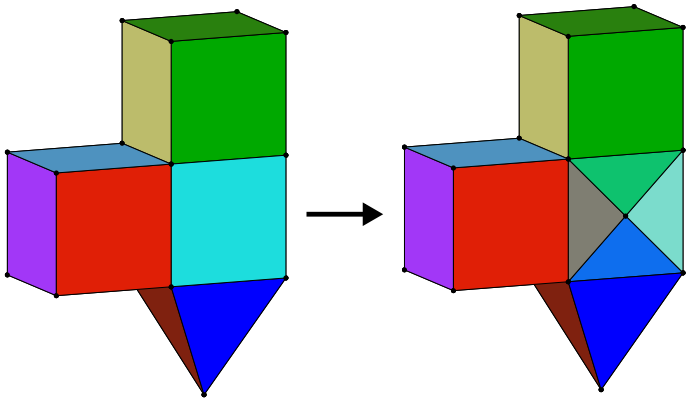


FIGURE 2.3 – Résultat de l'application de la règle de triangulation barycentrique sur une face d'un objet quelconque

La figure 2.3 illustre l'application de la règle sur le carré cyan. Sur cette règle, deux expressions de plongement sont définies : la position géométrique du barycentre de la face, et la couleur des nouvelles faces (ces deux expressions sont portées par le nœud c). Voici la traduction dans le langage concret de l'expression que nous avons défini dans la section 1.3.3, pour calcul du plongement *point* :

```
return @ebd<point>::barycentre(<0,1>_point(a));
```

Cette expression calcule le barycentre de la face à l'aide d'une fonction statique définie dans la classe du plongement `point` nommée `barycentre`. Cette fonction prend en paramètre une collection de plongements `point` et calcule le barycentre des positions.

Faisons de même avec l'expression du plongement couleur, qui calcule la moyenne entre la couleur de la face que l'on triangule, et de la face voisine incidente à l'arête dont est issue la nouvelle face :

```
return @ebd<couleur>::mix(a.couleur,a@2.couleur);
```

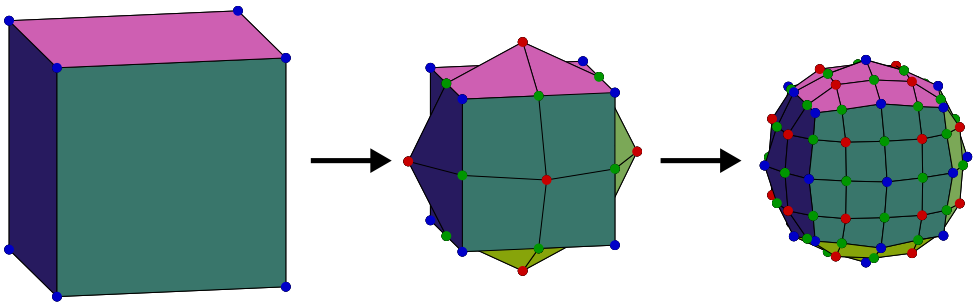


FIGURE 2.4 – Deux applications successives du schéma de subdivision de Catmull-Clark sur un cube

A présent, prenons l'exemple du schéma de subdivision de Catmull-Clark défini dans [CC78]. Ce schéma s'applique à des faces quadrangulaires et permet de réaliser une subdivision avec lissage. Une illustration de deux applications successives de cette opération sur un cube est montrée en figure 2.4. La subdivision réalise une découpe de chaque arête de départ (dont les

extrémités sont les points bleus), et relie ces nouveaux sommets (points verts), à un sommet créé au barycentre des faces (points rouges) dont ils sont issus.

La règle associée à cette opération est montrée en figure 2.5. Elle porte le calcul des positions géométriques sur les nœuds représentant les sommets bleus $n0$, verts $n1$ et $n2$, et rouges $n3$.

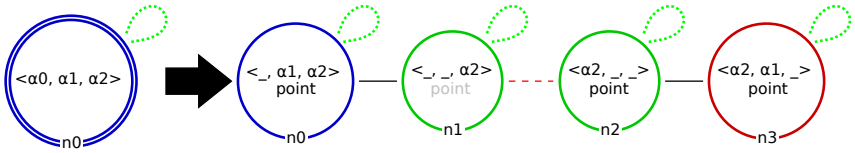


FIGURE 2.5 – Règle Jerboa du schéma de subdivision de Catmull-Clark

La position des sommets représentés par le nœud rouge se calcule comme le barycentre de la face de laquelle il est issu. La position des sommets représentés par les nœuds verts se calcule en fonction des deux faces incidentes à l'arête qu'ils découpent et des deux extrémités de celles-ci. En utilisant la fonction `barycentre` (celle que nous avons utilisé pour la règle de triangulation dans la section 1.3.3, page 32), on obtient l'équation suivante :

$$\begin{aligned} n2.point &= \frac{\frac{n0.point + n0@0.point}{2} + FP_0 + FP_1}{3}, \\ \text{avec } FP_0 &= barycentre(\langle 0,1\rangle_point(n0)), \\ \text{et } FP_1 &= barycentre(\langle 0,1\rangle_point(n0@2)), \end{aligned}$$

Ainsi, nous illustrons l'intérêt de la structure topologique dans le calcul des valeurs de plongements et les différentes fonctions de parcours, possibles dans Jerboa.

La modification des positions des sommets existant avant l'application de la règle, représentés par $n0$, permet de lisser le maillage. Leur position se calcule en fonction des faces et arêtes qui leurs sont incidentes, selon la formule suivante (où k est le nombre d'arêtes incidentes) :

$$\begin{aligned} n0.point &= \frac{F + 2R + (k-3)n0.point}{k}, \\ \text{avec } R &= \sum_{n \in \langle 1,2\rangle_\langle 2\rangle(n0)} \frac{n.point + n@0.point}{2k} \\ F &= \sum_{n \in \langle 1,2\rangle(n0)} \frac{barycentre(\langle 0,1\rangle_point(n))}{\langle 1,2\rangle(n).size()} \end{aligned}$$

La traduction de ces expressions mathématiques dans le langage de Jerboa est illustrée dans les listings : 2.1, 2.2, 2.3. Ces traductions utilisent les opérateurs classiques de calcul pour réaliser les calculs des plongements [2] ('+', '-', '/', ...).

```
return @ebd<point>::barycentre(<0,1>_point(n0));
```

Listing 2.1 – Expression du nœud rouge $n3$

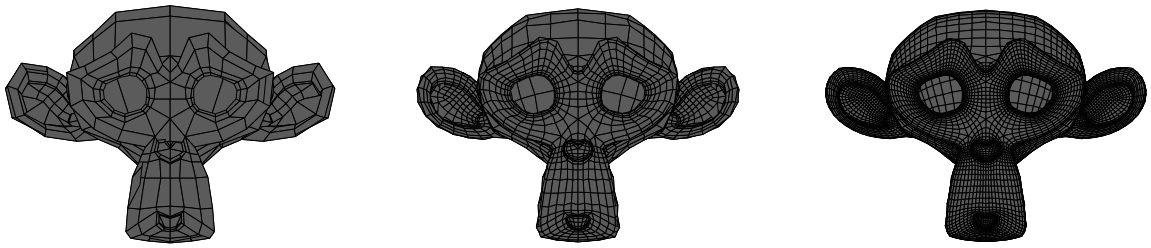
2. Cela est possible en C++ grâce à la surcharge des opérateurs mais ne l'est pas en Java

```
@ebd<point> FP0 = @ebd<point>::barycentre(<0,1>_point(n0));
@ebd<point> FP1 = @ebd<point>::barycentre(<0,1>_point(n0@2));
return (((n0.point+n0@0.point)*0.5
    + FP0 + FP1) / 3.0);
```

Listing 2.2 – Expression du nœud vert $n2$

```
JerboaList<@ebd<point>> listeBarycentresFacesIncidentes;
for(JerboaDart faceIncidente : <1,2>(n0)){
    listeBarycentresFacesIncidentes.add(
        @ebd<point>::barycentre(<0,1>_point(faceIncidente))
      );
}
// Faces points
@ebd<point> F = @ebd<point>::barycentre(listeBarycentresFacesIncidentes)
    ;
JerboaDarts listEdge = <1,2>_<2>(n0);
int k = listEdge.size();
// Edges points
@ebd<point> R(0,0,0);
for(JerboaDart edge : listEdge){
    R = R + (edge@0.point+edge.point)/2.0;
}
R = R/k;
return (F+2*R+(k-3)*n0.point)/k;
```

Listing 2.3 – Expression du nœud bleu $n0$



(a) Modèle de départ (b) 1ère application (c) 2e application

FIGURE 2.6 – Application du schéma de subdivision de Catmull-Clark sur "Suzanne"

Le résultat de deux applications successives de cette règle sur un modèle 3D représentant une tête de singe, appelée "Suzanne", issue du logiciel de modélisation *Blender*, est montré en figure 2.6.

En l'état, ces expressions ne donnent de "bons" résultats que pour des surfaces fermées (sans bord). En effet, sur les bords, le lissage n'est pas souhaitable car les arêtes du bords se rapprocheraient de l'intérieur du maillage, au fur et à mesure des applications successives de l'opération, comme illustré en figure 2.7. Afin de conserver une consistance au niveau des bords, il faut modifier le calcul de plongement des nœuds bleus et verts qui sont concernés par cette déformation. Pour chacun d'eux, il faut tester si l'un des brins de leur orbite sommet n'a pas de

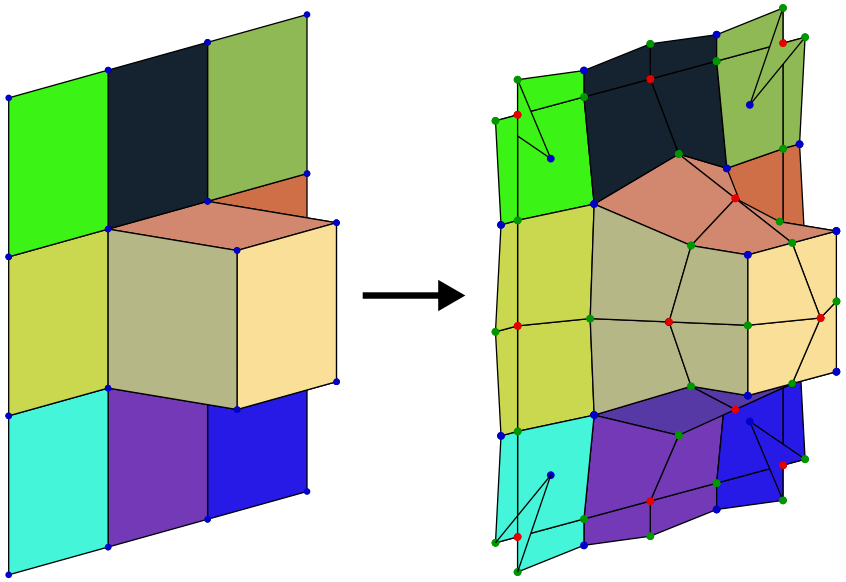


FIGURE 2.7 – Applications de l'opération de subdivision de Catmull-Clark sur une surface ouverte

voisin par α_2, ce qui indique que le sommet en question est sur le bord du maillage. Dans ce cas, la position calculée par l'expression portée par le nœud vert est le milieu de l'arête dans laquelle le sommet est inséré. On ajoute donc le code suivant au début du Listing 2.2 :

```
for(JerboaDart d:<1,2>(n0)){ // pour tous les brins du sommet de n0
    if(d@2==d) // teste si d est sur un bord
        return (n0.point+n0@0.point)*0.5f;
}
```

Pour les sommets bleus filtrés par `n0` on procède à un ajout similaire (au début du Listing 2.3) sauf que la position retournée est celle de départ pour ne pas procéder au lissage :

```
for(JerboaDart d:<1,2>(n0)){ // pour tous les brins du sommet de n0
    if(d@2==d) // teste si d est sur un bord
        return n0.point;
}
```

Avec ces ajouts, l'application de l'opération produit des résultats corrects sur des surfaces ouvertes ou fermées comme illustré en figure 2.8.

Le langage de calcul permet également d'écrire les pré-conditions et les autres traitements du cycle d'exécution des règles. Pour ce faire, il existe quelques instructions et opérateurs dédiés pour la manipulation du motif filtré pour les processus qui y ont accès. Les pré-conditions permettent de tester, avant de réaliser les transformations, si l'état de l'objet sur lequel la règle s'applique le permet. Par exemple le schéma de subdivision de Catmull-Clark est censé s'appliquer uniquement à des maillages quadrangulaires. Or nous avons montré que la règle peut s'appliquer sans modification quelle que soit la topologie des faces. L'utilisation de la pré-condition de règle s'assure que la règle ne s'applique que pour des faces quadrangulaires. Pour tester le nombre d'arêtes des faces sur lesquelles s'applique la règle, la pré-condition s'appuie sur le motif filtré. Ce motif est sous la forme d'une matrice de brins et accessible avec l'opérateur `@leftPattern`. Il regroupe les brins qui sont représentés par un même nœud. Ainsi, `@leftPattern#n0` permet d'obtenir tous les représentants du nœud `n0`. De même, `@leftPattern#n0#i` permet d'obtenir le i^e brin qui instancie le nœud `n0`.

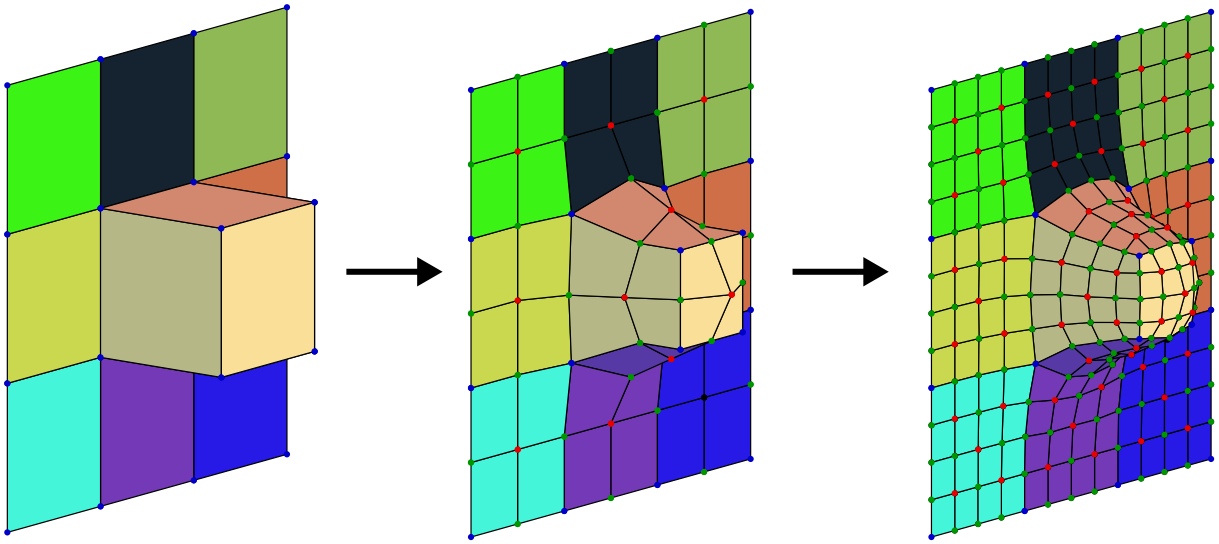


FIGURE 2.8 – Application de l'opération de subdivision de Catmull-Clark sans lissage sur les bords, sur une surface ouverte

```
// pour tous les représentants de n0
for i in 1..@leftPattern#n0.size(){
    JerboaDarts areteFace = <0,1>_<0>(@leftPattern#n0#i);
    // on utilise le ieme représentant de n0 dans le motif filtré
    // pour réaliser la collecte de toutes les arêtes de sa face
    if(areteFace.size()!=4)
        return false;
}
return true;
```

Cette pré-condition vérifie si toutes les faces filtrées contiennent bien quatre arêtes. Si une face en contient un nombre différent, la pré-condition échoue et la règle ne s'applique pas. Notons que l'algorithme présenté ici n'est pas optimal. Les représentants de `n0` sont tous les brins des faces filtrées. Ainsi, le parcours de recherche des arêtes des faces et le test de leur nombre est réalisé sur chaque brin de chaque face. Il est possible d'éviter cette redondance en utilisant un système de marques mais ce n'est pas l'objectif de cet exemple.

On pourrait également écrire une pré-condition similaire pour définir que la règle de triangulation barycentrique ne peut pas s'appliquer sur des faces triangulaires.

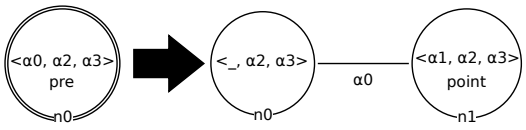


FIGURE 2.9 – Règle Jerboa de découpe d'arête.

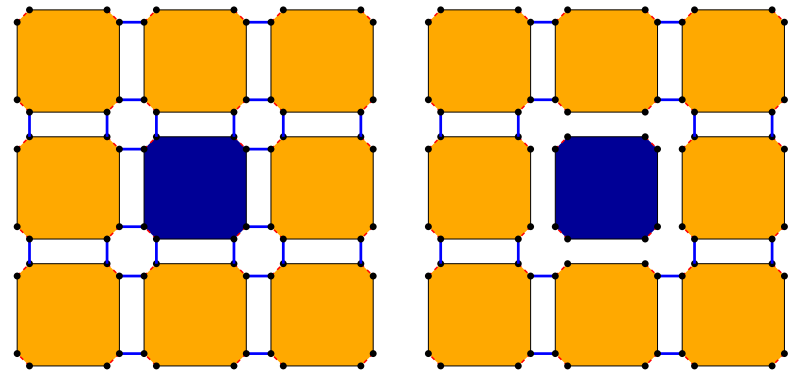
Prenons à présent l'exemple de la découpe d'une arête pour illustrer l'utilisation des pré-conditions de nœud. Cette opération découpe une arête en son milieu. Nous souhaitons que la règle Jerboa associée à cette opération, illustrée en figure 2.9, ne soit pas applicable si l'arête filtrée est de longueur nulle, c'est-à-dire si les deux sommets de l'arête sont à la même position

géométrique. L'expression du plongement `point` portée par `n1` est définie comme suit : `return @ebd<point>::barycentre(<0>_point(n0));`.

Cette expression décrit le calcul de la position du sommet inséré dans l'arête comme le milieu de celle-ci. Pour vérifier que l'arête n'est pas de longueur nulle, nous avons défini une pré-condition de nœud sur $n0$. Sur la figure, cette pré-condition est une étiquette nommée `pre` portée par $n0$. Pour tester la longueur de l'arête, nous testons si les valeurs des plongements `point` de chaque représentant de `n0` et de leurs voisins en α_0 sont identiques ou non : `return n0.point != n0@0.point;`. Notons que nous avons utilisé ici un opérateur d'inégalité strict mais bien souvent, il est nécessaire de tester à un ε-près l'égalité ou l'inégalité des plongements géométriques pour éviter les erreurs de précision. Dans ce cas on peut écrire la pré-condition de la manière suivante : `return n0.point.differentEpsilonPres(n0@0.point,EPSILON);`. La fonction `differentEpsilonPres` est interne à la classe de plongement `point`, et prend en paramètre un autre `point`, avec lequel le `point` courant est comparé, et une valeur d'epsilon.

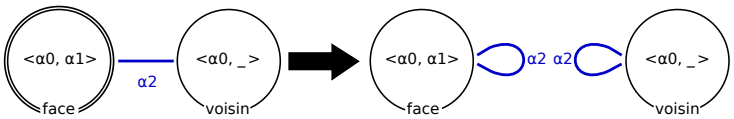
2.3 Langage de *script*

Les règles de transformation de graphe permettent de réaliser des opérations facilement mais uniquement sur une orbite donnée. Le graphe gauche d'une règle permet de filtrer une ou plusieurs orbites, selon le nombre de nœuds, mais toutes les orbites filtrées doivent être isomorphes à renommage et suppression près des liaisons. Ainsi, certaines opérations ne sont pas réalisables avec une seule règle de transformation de graphe car elle s'appliquent à plusieurs orbites. Pour réaliser ces opérations, il est nécessaire d'appliquer successivement plusieurs règles.



(a) Face présentant des voisins sur toutes ses arêtes

(b) Face décousue

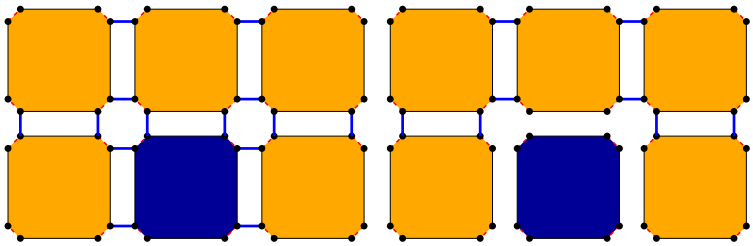


(c) Règle de dé-couture de face

FIGURE 2.10 – Cas de dé-couture d'une face (ici bleue)

Prenons l'exemple de la dé-couture en α_2 d'une unique face, comme le carré bleu de la figure 2.10(a). Le résultat de cette dé-couture est donné en figure 2.10(b) La règle Jerboa associée à cette opération est présentée en figure 2.10(c). Cette règle fonctionne parfaitement sur cet exemple, et fonctionnerait également sur un exemple similaire quelle que soit la topologie de la

face à découdre. En revanche, elle ne fonctionne que dans un cas précis : la face sur laquelle elle est appliquée doit avoir toutes ses arêtes liées à d'autres faces (c'est le cas ici avec les faces oranges).



(a) Face ne présentant pas de voisins sur toutes ses arêtes (b) Face décousue

FIGURE 2.11 – Deuxième cas de dé-couture d'une face (ici bleue)

Si la face a au moins une de ses arêtes qui ne présente pas de voisin par α_2, la règle ne peut pas s'appliquer car les motifs filtrés par les deux nœuds ne sont plus isomorphes. C'est le cas sur l'exemple montré en figure 2.11(a). Le résultat de dé-couture de la face bleu comme illustré en figure 2.11(b) ne peut pas être obtenu avec la règle présentée en figure 2.10(c). En effet, le motif gauche impose que lors du filtrage, tous les brins filtrés par `face` présente un voisin par α_2. Or ce n'est pas le cas pour deux des brins du carré bleu qui sont au bord. Ainsi, le processus de filtrage échoue et la règle ne s'applique pas.

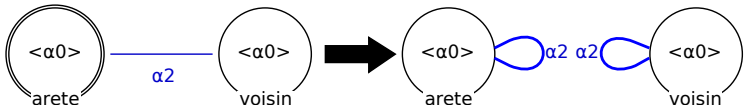


FIGURE 2.12 – Règle `DecoutureAreteA2` Jerboa de dé-couture d'arête

Il n'est donc pas possible de découdre toutes les arêtes de la face en même temps. Pour découdre la face, il faut découdre seulement les arêtes qui présente une arête voisine. Pour ce faire, commençons par écrire une règle qui découd une seule arête. Cette règle est présentée en figure 2.12. Nous pouvons ensuite appliquer cette règle sur chaque arête de la face qui n'est pas sur un bord.

Afin de pouvoir réaliser ce type d'opérations, nous avons mis en place un langage de *script*, dont l'instruction de base est l'application d'une règle. Ce langage offre toutes les possibilités du langage de calcul, avec quelques instructions supplémentaires.

Les *scripts* sont des opérations qui présentent les mêmes propriétés que les règles. Ils peuvent avoir des paramètres d'entrée topologiques et de plongement. Leur flux d'activité est présenté en figure 2.13, et est assez similaire à celui des règles. La différence notable avec les règles est le processus de transformation qui est différent. Dans les règles, le graphe droit permet de définir les transformations des objets. Dans les *scripts*, ces transformations sont réalisées par un enchaînement de plusieurs opérations (règle ou *script*) qui étend le langage de calcul.

Les paramètres topologiques des *scripts* sont définis un peu différemment des règles classiques. Les *scripts* présentent un graphe gauche qui permet de nommer les différents paramètres topologiques. Dans notre exemple, l'application de notre *script* de dé-couture nécessite un brin d'accroche sur la face à découdre. Le graphe gauche du *script* présente donc un nœud unique comme illustré en figure 2.14. Notons que ce nœud présente une étiquette topologique vide. En

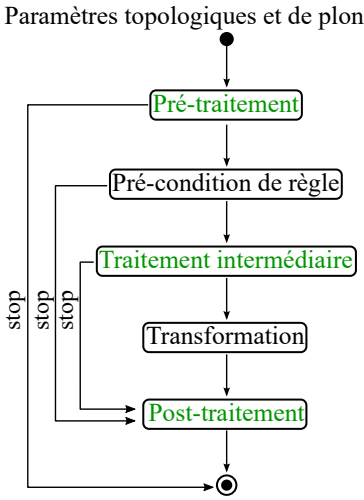


FIGURE 2.13 – Flux d'activité de l'application d'un *script* de règle dans Jerboa



FIGURE 2.14 – Graphe gauche du *script* de dé-couture de face

effet, les *scripts* permettent d'enchaîner l'application d'autres règles, auxquelles ils fournissent les brins accrochés. Les *scripts* délèguent le processus de filtrage aux règles qu'ils appellent, et ne réalisent pas de filtrage eux-même. L'autre particularité des variables topologiques des *scripts* est leur *multiplicité*. La multiplicité d'une variable topologique permet de déterminer, au moment de la définition du *script*, le nombre de brins accrochés possible qu'elle peut représenter. La multiplicité {42} par exemple indique que la variable doit accrocher exactement 42 brins, {2..5} qu'elle peut accrocher entre deux et cinq brins, {2..∗} pour au moins deux brins, et {∗} pour un nombre quelconque de brins. Ici nous souhaitons que le *script* s'applique à un seul brin d'accroche, celui de la face à découdre, pour appliquer la dé-couture en α_2 sur toutes les arêtes de cette face. Ainsi nous donnons au nœud `faceADecoudre` une multiplicité de {1} pour indiquer que lors de l'application du *script*, il représente exactement un brin.

Détaillons à présent les différentes possibilités du langage en commençant par l'instruction de base : l'application d'une opération. Une règle est identifiée par son nom et son application est écrite de façon similaire à un appel de fonction. Les paramètres d'appel d'une règle sont ordonnés : les paramètres topologiques (obligatoires) puis les paramètres de plongements (facultatifs). En conséquence, le passage des paramètres doit respecter cette convention.

Dans le cas de l'application de la règle de dé-couture d'une arête en α_2, l'instruction s'écrit comme suit : `@rule<DecoutureAreteA2>(arete)`, avec `arete` un brin de la G-carte appartenant à l'arête à découdre de sa voisine. Comme nous venons de le voir, découdre une face demande de découdre successivement toutes ses arêtes. Ainsi, nous avons écrit le *script* de dé-couture d'une face comme suit :

```
for(JerboaDart arete : <0,1>_<0>(faceADecoudre)){
    try{
        @rule<DecoutureAreteA2>(arete);
    }catch(JerboaException e){
        // Ici l'application de la règle a échouée
    }
}
```

Rappelons que ce *script* comporte un seul paramètre topologique dont le nom est `faceADecoudre` (défini dans le membre gauche du *script*). La boucle permet de réaliser la dé-couture sur la collection `<0,1>_<0>(faceADecoudre)`, qui donne un brin par arête, de la face incidente au brin `faceADecoudre`. On note également la présence d'un `try/catch` qui permet de savoir à chaque itération si la règle a échouée, et dans ce cas d'empêcher l'exception levée d'interrompre le *script* en cours. Sur notre exemple lorsque la règle est appelée sur une arête sans voisin en a2, il y a une erreur dans la pré-condition provoquant une exception. Il est donc nécessaire de la neutraliser pour pouvoir continuer le traitement sur les autres arêtes.

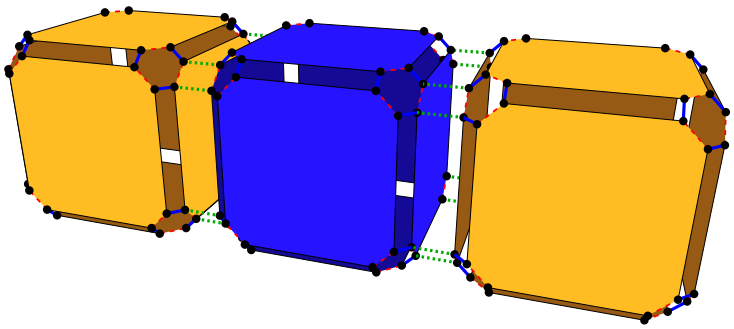


FIGURE 2.15 – 3 cubes liés

Pour donner un exemple en 3D, le problème est identique pour la dé-couture d'un volume (orbite $\langle 0,1,2\rangle$)) par α_3 si toutes ses faces n'ont pas de voisins, comme illustré en figure 2.15.

La commande @rule renvoie le résultat de l'application de la règle mentionnée. Dans le cas, où l'opération est une règle et non un *script*, le résultat de l'application retourne le motif droit de la règle appliquée. Dans le cas où l'opération est un *script*, le résultat est en accord avec le motif droit de la règle mais rempli manuellement par l'utilisateur dans une variable résultat. Le type de ces résultats est : `JerboaRuleResult`. Ces résultats encapsulent les graphes droits des règles et permettent aisément d'accéder aux représentants que l'on souhaite. Dans l'exemple de la règle de dé-couture en α_2 (cf. figure 2.12), il est possible d'accéder à tous les représentants des nœuds `arete` et `voisin`, comme dans l'exemple suivant :

```
JerboaRuleResult result = @rule<DecoutureAreteA2>(faceADecoudre);
JerboaList<JerboaDart> listeArete = result#arete;
JerboaList<JerboaDart> listeVoisin = result#voisin;
```

Ce *script* décrit l'application de la règle `DecoutureAreteA2` sur le brin `faceADecoudre`, et le stockage du résultat de l'application dans la variable `result`. Il décrit ensuite la création de deux listes de brins dans lesquelles sont stockés respectivement les représentants des nœuds `arete` et `voisin` du motif droit, résultat de l'application de la règle `DecoutureAreteA2`.

Cet exemple permet également d'illustrer l'utilisation des listes dans Jerboa. Le type des listes est `JerboaList`. Le type des éléments stockés dans la liste est spécifié entre chevrons. Ici le type des éléments stockés est `JerboaDart` (le type des brins). L'accès aux différents éléments de la liste se fait avec l'opérateur `[ ]`. Comme pour tout langage utilisant les tableaux et les listes, `listeArete[i]` permet d'accéder au i^e élément de la liste `listeArete`.

Les *scripts* sont des opérations au même titre que les règles de transformation de graphe. En ce sens, il est logique qu'un *script* puisse faire appel à l'application d'un autre *script*. Le filtrage n'est pas réalisé par les *scripts* mais par les règles qu'ils appellent, et les paramètres topologiques des *scripts* sont les nœuds de leur membre gauche. Chaque nœud peut représenter un nombre de brins définit par sa multiplicité. Nous avons donc défini dans le langage l'appel à l'application d'un *script* de façon similaire à l'application d'une règle. Cependant, pour être en mesure de fournir plusieurs brins pour chaque nœud du *script*, les paramètres topologiques passés lors de l'appel sont sous forme de listes de brins.

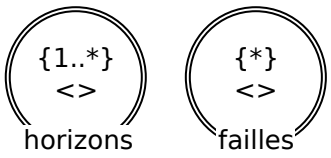


FIGURE 2.16 – Membre gauche d'un *script* réalisant la découpe d'horizons par des failles

Pour illustrer l'appel à l'application d'un *script* par un autre *script*, prenons un exemple géologique. Nous appellerons *horizons* les surfaces limites qui séparent deux strates géologiques dont la composition ou la date de dépôt sont différentes. Imaginons que nous disposons d'un *script* appelé `calculeIntersectionHorizonFaille`, qui coupe tous les horizons du sous-sol par les surfaces des failles du modèle. Le membre gauche de ce *script* est montré en figure 2.16. Le membre gauche est composé de deux nœuds, l'un représentant les horizons et l'autre représentant les failles. Leurs multiplicités sont différentes, il faut fournir au *script* au moins un brin accroché appartenant à un horizon, mais un nombre indéterminé de brins accroché appartenant à une faille. Pour appliquer ce *script*, il est possible que l'utilisateur sélectionne manuellement les éléments à fournir au *script* pour l'appliquer, mais en pratique, il est souhaitable que la recherche des horizons et des failles soit un processus automatisé. Le *script* suivant décrit ce processus de recherche et d'application automatique du *script* `calculeIntersectionHorizonFaille` :

```
JerboaList<JerboaDart> brinsHorizon;
JerboaList<JerboaDart> brinsFaille;
JerboaMark marqueDejaVu; // création d'une marque
for(JerboaDart brin: @gmap){
// pour tous les brins de la G-carte
    if(brin$marqueDejaVu){
    // si la surface de "brin" n'a pas déjà été ajoutée à une liste
        if(brin.typeSurface == HORIZON){
            brinsHorizon.add(brin);
            @mark(<0,1,2,3>(brin), marqueDejaVu);
            // la composante connexe de "brin" est marquée déjà ajoutée
        }else if(brin.typeSurface == FAILLE){
            brinsFaille.add(brin);
            @mark(<0,1,2,3>(brin), marqueDejaVu);
            // la composante connexe de "brin" est marquée déjà ajoutée
        }
    }
```

```
}
delete marqueDejaVu; // dé-marquage des surfaces
@rule<calculeIntersectionHorizonFaille>(brinsHorizon, brinsFaille);
// application de l'autre script
```

Ce *script* décrit le parcours de tous les brins de la carte généralisée, et leur tri dans deux listes distinctes `brinsHorizon` et `brinsFaille`, dans lesquelles est ajouté un unique brin par composante connexe grâce au processus de marquage. On suppose ici que le modèle contient un plongement `typeSurface` qui définit les informations sémantiques des surfaces et que toutes les surfaces d'une même composante connexe portent la même sémantique. Nous reviendrons sur cette particularité dans la deuxième partie de ce document. Ceci nous permet de tester si les surfaces sont de type `HORIZON` ou de type `FAILLE`. Une fois les composantes identifiées, le *script* `calculeIntersectionHorizonFaille` est appliqué avec les composantes trouvées en paramètre.

le langage de *script* inclut également quelques opérateurs spécifiques comme par exemple **@modeler** et **@gmap**. L'opérateur **@modeler** permet d'accéder au modeleur, ses attributs et ses fonctions. On peut par exemple accéder à la dimension du modeleur : **@modeler**`.getDimension()`. L'opérateur **@gmap** donne accès à la G-carte. Il permet ainsi d'accéder à des fonctions ou des attributs de celle-ci. Cet opérateur est "itérable" et donc utilisable dans une boucle de type "foreach". Les éléments sur lesquels se fait l'itération sont les brins de la G-carte.

Gestion des marques : dans toutes les applications manipulant des graphes, il est nécessaire de pouvoir en marquer les nœuds. Ces marques permettent de réaliser des parcours dans le graphe, et de se rappeler des brins déjà parcourus. Les marques sont utilisées dans le moteur de Jerboa dans tous les parcours du graphe, tant pour les différentes collectes que pour d'autres utilisations internes au noyau. Notre langage fournit un ensemble d'opérateur pour que l'utilisateur puisse exploiter ce mécanisme. Dans les implantations actuelles, nous pouvons exploiter simultanément 64 marqueurs différents.

Notons qu'un certain nombre de marques est nécessaire au moteur d'application des règles pour fonctionner correctement (moins d'une dizaine). Ceci laisse à l'utilisateur un nombre tout de même raisonnable de marques disponibles pour réaliser les calculs dont il a besoin.

Dans le langage de calcul, la déclaration d'une marque se fait comme une variable classique : `JerboaMark marque;`. Lorsqu'une variable de type marque est déclarée, elle est automatiquement allouée dans le code généré qui assigne une marque libre dans celle-ci. À la fin de l'exécution du *script* ou de l'expression, la marque doit être détruite afin de ne pas la perdre. Grâce au langage, nous pouvons détecter automatiquement à la génération du code, et de façon transparente, un appel à la destruction de la marque aux endroits nécessaires comme la fin de l'expression, juste avant un `return`, ou avant le lancement d'une exception avec `throw`. La destruction des marques est donc réalisée automatiquement.

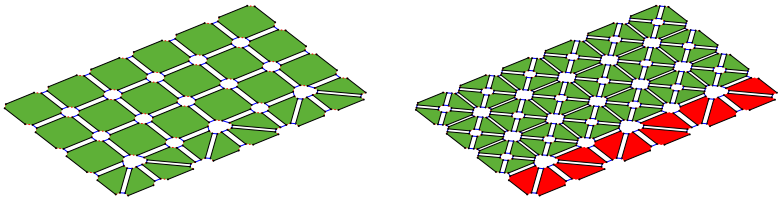
L'opérateur **@mark** permet de marquer un brin ou une collection de brin. Pour marquer un brin `n` avec la marque `marque` on procède comme suit : **@mark**`(n,marque)`. Il est également possible de remplacer le brin par un ensemble de brins résultant d'un parcours d'orbite par exemple. Le marquage de l'orbite volume s'écrit comme suit : **@mark**`(<0,1,2>(n),marque)`. Le démarquage d'un brin `n` s'écrit de manière symétrique : **@unmark**`(n,marque)`.

Le test de la présence d'une marque sur un brin se fait avec les opérateurs suivants : `@isMark(brin,marque)`, `@isNotMark(brin,marque)`. Ces constructions syntaxiques possèdent des notations plus compactes pour améliorer la lisibilité du code avec l'opérateur `$`, elles s'écrivent respectivement : `brin$marque` et `!brin$marque`. Il existe un opérateur dédié au test du "non marquage"

car le moteur d'application présente deux fonctions distinctes pour tester si un brin est marqué ou non, dans un souci d'optimisation. Comme ces fonctions sont amenées à être appelées très souvent (lors des collectes ou autres parcours), un simple test du non marquage par la négation du marquage ferait perdre un test booléen à chaque appel. Cette optimisation est contrôlée lors de la génération du code de manière transparente à l'utilisateur.

Notons que ce système de marques existe également dans le langage de calcul et est donc utilisable dans les expressions de plongements et les différents traitements.

Opérateur de choix : Le filtrage et les pré-conditions permettent la définition par cas, à la manière des langages fonctionnels. Ainsi chaque cas est défini par une règle ou un *script* donné.



(a) Objet original (b) Résultat

FIGURE 2.17 – Application d'un *script* qui colore en rouge les triangles et triangule les autres faces

Prenons l'exemple d'un maillage composé de quadrangles et de triangles, comme l'illustre la figure 2.17. Supposons que nous souhaitons trianguler ce maillage, sauf pour les faces qui sont déjà des triangles. Nous allons changer la couleur des triangles pour du rouge et trianguler les faces qui ne le sont pas, avec l'opération de triangulation barycentrique présentée plus tôt. Dans notre partie applicative géologique, nous avons souvent des maillages globalement quadrangulaires, mais triangulés au niveau des coupures des horizons par les failles. Cette opération pourrait être utilisée pour obtenir un maillage complètement triangulé, avec des triangles d'une couleur particulières dans les zones faillées. Dans cet exemple simple, on pourrait, avec un *script*, parcourir toutes les faces de l'objet et tester pour chacune si c'est un triangle ou non, puis appliquer la règle qu'il convient. Nous avons vu plus tôt que ce test était déjà réalisé dans la pré-condition de la règle de triangulation barycentrique. Avec l'opérateur de choix on peut utiliser la puissance du langage de Jerboa afin de se servir directement de cette pré-condition. Voici une portion de *script* qui décrit cette opération, avec `surface` un brin de référence sur la surface :

```
for(JerboaDart iemeFace : <0,1,2>_<0,1>(surface)){
// boucle foreach sur toutes les faces d'une surface.
    @rule<TriangulationBarycentrique>(iemeFace)
    | @rule<ChangeCouleurFace>(iemeFace, CouleurRGB(255,0,0));
}
```

Cette portion de *script* décrit le parcours de toutes les faces de la surface incidente au brin `surface`, et illustre l'utilisation de l'opérateur de choix pour réaliser l'opération. Si l'application de la règle de triangulation échoue (ici si la face est déjà un triangle), la règle de changement de couleur de face s'applique. Pour colorer en rouge les faces déjà triangulaires, nous donnons comme paramètre de plongement à la règle de changement de couleur, l'instance de la classe `CouleurRGB` correspondante.

Le résultat de l'application d'un opérateur de choix peut être stocké. La syntaxe est proche du stockage d'une simple application de règle :

```
JerboaRuleResult resultat =
        @rule<TriangulationBarycentrique>(iemeFace)
      | @rule<ChangeCouleurFace>(iemeFace, CouleurRGB(255,0,0));
```

À l'issue de l'application de cette instruction, la variable `resultat` contient le résultat de l'opération de triangulation barycentrique si la face incidente au brin `iemeFace` n'est pas un triangle, et le résultat de la règle `ChangeCouleurFace` sinon. Notons que nous avons ici pris l'exemple du choix de deux règles mais cet opérateur peut s'utiliser avec un nombre quelconque de règles et *scripts*.

Opérateur d'injection de code sous-jacent : Le langage a été conçu pour être en mesure de réaliser différents calculs et d'appliquer des opérations. Pour générer du code dédié au moteur d'application que l'utilisateur a choisi, les instructions définies dans le langage sont traduites en C++ ou en Java. Le langage est conçu pour être générique et intuitif pour les habitués de ces langages. Cependant, certaines spécificités des langages peuvent être difficiles à réaliser avec le langage Jerboa. Afin de ne pas brider l'utilisation du langage et les possibilités de calcul dans la réalisation des opérations, nous avons mis en place un opérateur **@lang** permettant d'écrire du code directement dans le langage sous-jacent. L'implantation de ce type d'opérateur est courante dans la création d'un langage dédié à une application spécifique.

Ce code est exporté tel quel par l'éditeur lors de la génération. Afin de conserver la portabilité d'un langage cible à l'autre du modeleur, l'opérateur prend en paramètre le nom du langage cible ce qui permet de différencier le code produit en fonction du langage cible . Son utilisation se fait comme suit :

```
TypeListeUtilisateur<Type1> var;
for(int i=0;i<var.nombeElement();i++){
  @lang(C++){
      std::cerr << var.template getValeur<Type1>(0) << std::endl;
  }
  @lang(Java){
      System.err.println(var.<Type1>getValeur(0));
  }
}
```

Conclusion

Pour conclure ce chapitre, nous avons réussi à rendre les étapes d'application d'une règle plus transparente à l'utilisateur. Ce dernier peut, à présent, mieux contrôler l'exécution de sa règle. Ces étapes permettent par exemple de réaliser des optimisations avec des pré-calculs. Les processus précédant la transformation effective peuvent interrompre le processus d'application si un calcul ne s'est pas fait correctement ou si l'état de l'objet ne permet pas de poursuivre l'application de la règle.

Nous avons également mis en place un langage qui permet grâce à différents opérateurs dédiés de définir les expressions de plongements, les différents processus et des opérations plus complexes appelés *scripts*. Grâce à ce langage nous pouvons à présent contrôler les expressions de plongement qui étaient auparavant écrites dans le langage du moteur d'application, et ainsi réaliser des vérifications automatiques.

Chapitre 3

Mise en œuvre

Au cours de cette thèse, nous avons fait évolué l'outil Jerboa en étendant son langage et les vérifications automatiques de préservation de la cohérence. Par ailleurs, une analyse fine des besoins dans le domaine de la modélisation et l'étude des outils connexes, nous ont permis d'ajouter de nouvelles fonctionnalités à notre plateforme. Nous pouvons citer par exemple la modification du cycle d'exécution des règles, et le langage de *script*. Ces nouveautés demandaient de profondes modifications de la structure de l'éditeur et sa mise à niveau a été assez complexe. Nous avons implanté une nouvelle interface capable de gérer tous les nouveaux aspects. Le nouvel éditeur inclut les fonctionnalités déjà présentes dans l'ancien comme la définition des paramètres du modeleur, des plongements et des règles. Ces fonctionnalités ont été adaptées pour inclure les modifications du processus d'application des règles que nous avons détaillé dans le chapitre précédent. Il inclut également de nouvelles fonctionnalités comme la définition de *scripts*, et de nouvelles vérifications automatiques de préservation de la cohérence.

Ce chapitre décrit les différentes caractéristiques du nouvel éditeur, les vérifications qu'il met en œuvre et l'implantation du langage de calcul et de *script* lors de la génération des opérations.

3.1 Éditeur et vérification

Le premier éditeur de Jerboa, a été développé en 2012 pour compléter le premier prototype Java [BALGB14]. Cet éditeur ne supporte que la définition des règles : c'est-à-dire des opérations ne pouvant travailler que sur une seule orbite et des expressions de plongement assez courtes dans la lignée des langages fonctionnels. Suite à ce développement plusieurs applications ont été développées :

- JerboaArchi : un premier outil se focalisant sur une opération complexe d'extrusion d'un plan en 2D en modèle 3D ;
- Jermination : une mise en œuvre des G-cartes L-Systèmes en Jerboa ;
- Japhy [BSBAM17b] première version : une bibliothèque de simulation physique, proposant des méthodes de calcul physique (masse-ressort, masse-tenseur, éléments finis, ...) sur des maillages 2D, 3D variés (hexaédrique, tétraédrique, triangulaire, quadrangulaire, mixte, ...).

Ces projets ont été réalisés par les contributeurs de Jerboa, mais également par des collègues et des étudiants non initiés à Jerboa ni aux transformations de graphes. Grâce à ces applications, nous avons détecté des manques et des problèmes d'ergonomie pour satisfaire pleinement les utilisateurs. La partie applicative de cette thèse a montré que le principe de Jerboa est tout à fait viable pour réaliser des opérations de modélisation fiables. De plus, les stages successifs

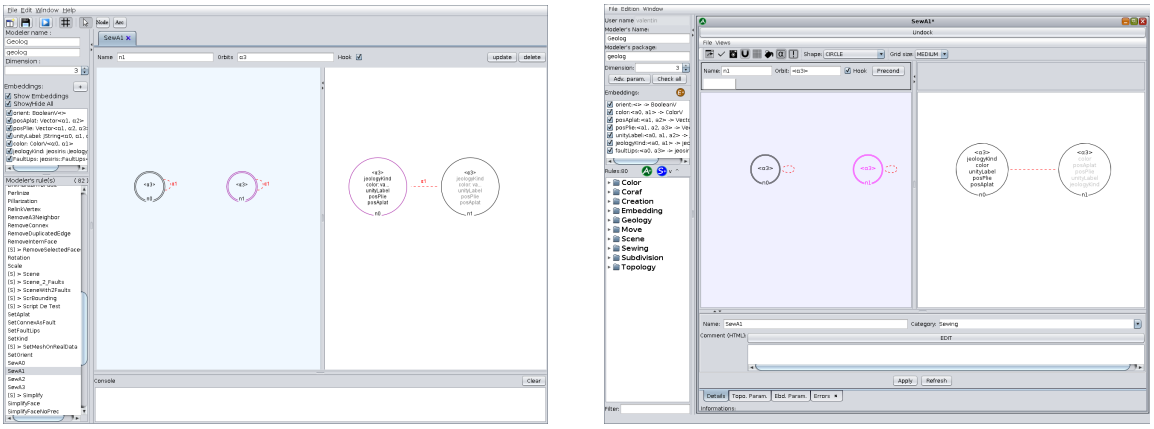
d'étudiants de licence et master, ont montré que la prise en main de l'outil est à la portée des non initiés aux transformations de graphes en quelques semaines.

3.1.1 Un nouvel éditeur de modeleur

Au début de ma thèse, nous avons réalisé une nouvelle interface pour répondre aux problèmes que nous avions identifiés pendant les différents projets précédents. Ces modifications facilitent le développement des opérations pour la partie applicative de cette thèse, la géologie. En effet, les opérations géologiques sont beaucoup plus complexes que celles réalisées lors des précédents projets.

Parmi les gros problèmes relevés nous avions par exemple :

— le manque d'information lors de la détection de l'omission d'une expression de plongement ;
— l'affichage des noeuds des règles sous la forme de cercle, ce qui complique l'affichage si le modeleur contient de nombreux plongements ;
— l'impossibilité de classer les règles ;
— l'impossibilité d'afficher plusieurs règles simultanément ;
— l'espace d'édition des expressions de plongement qui était beaucoup trop petit.



(a) Ancienne version (b) Nouvelle version

FIGURE 3.1 – Comparaison de l'ancienne et de la nouvelle version de l'éditeur de modeleur

Nous avons donc réimplanté l'éditeur afin d'améliorer son ergonomie et d'intégrer les différentes extensions du langage Jerboa. Cependant, nous avons conservé l'organisation globale de l'interface qui avait reçu un retour positif. Ainsi, à première vue, l'interface du nouvel éditeur reste assez proche de l'ancien comme on peut le voir sur la figure 3.1. Le paramétrage du modeleur se fait toujours dans la partie gauche de l'application, et l'édition des opérations est restée dans la partie droite.

Les nouveautés sont présentes dans l'édition des opérations qui ne sont plus uniquement des règles, mais peuvent être des *scripts*. Pour tenir compte du nouveau processus d'application des règles, nous avons ajouté la possibilité d'éditer, à l'aide du langage défini au chapitre précédent, les différents processus et conditions. Chaque panneau d'édition peut être ouvert indépendamment depuis le menus *Views* des opérations, comme illustré en figure 3.2.

Nous avons également ajouté des panneaux afin de définir les paramètres topologiques et de plongement. Le panneau des paramètres topologiques (cf. figure 3.3) permet de définir l'ordre dans lequel ils doivent être utilisés au moment de l'appel de l'application de la règle. Ici la

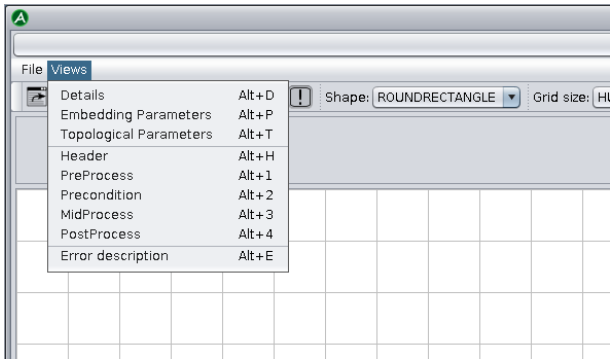


FIGURE 3.2 – Menu d'accès aux zones d'édition pour chaque étape du cycle d'exécution

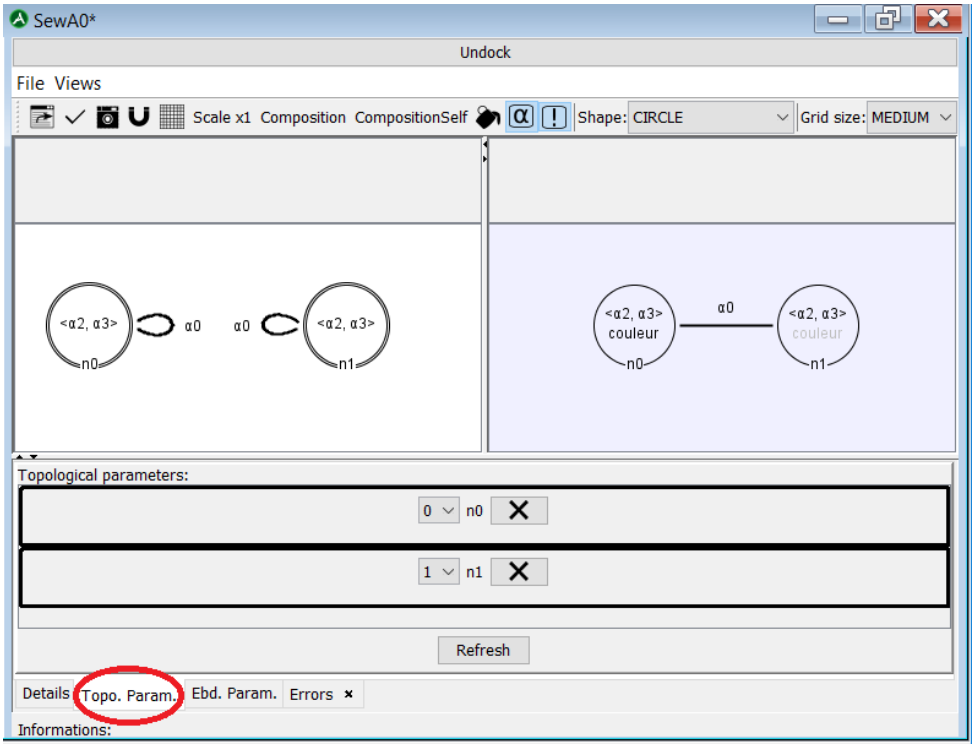


FIGURE 3.3 – Panneau des paramètres topologiques

figure illustre l'édition de l'opération de couture en α_0. Cette règle présente deux paramètres topologiques qui correspondent aux nœuds d'accroches de la règle. Ils sont ordonnés comme suit : `n0` est le premier et `n1` le second.

On peut faire de même avec les paramètres de plongements. Le panneau pour les éditer est illustré en figure 3.4. Il permet de définir chaque paramètre avec son type et sa valeur par défaut qui est donnée à la création de la règle. Lors de l'application d'une règle, il n'est pas obligatoire de fournir les paramètres de plongements. Si un paramètre n'est pas fourni, sa valeur est celle fournie lors de la précédente application de la règle, ou la valeur par défaut dans le cas ou elle n'a jamais été modifiée. La figure représente une règle qui change la couleur des faces d'une composante connexe. Le paramètre de plongement nommé `color` permet de spécifier la couleur à attribuer à la composante. Nous avons choisi ici d'attribuer par défaut une couleur aléatoire.

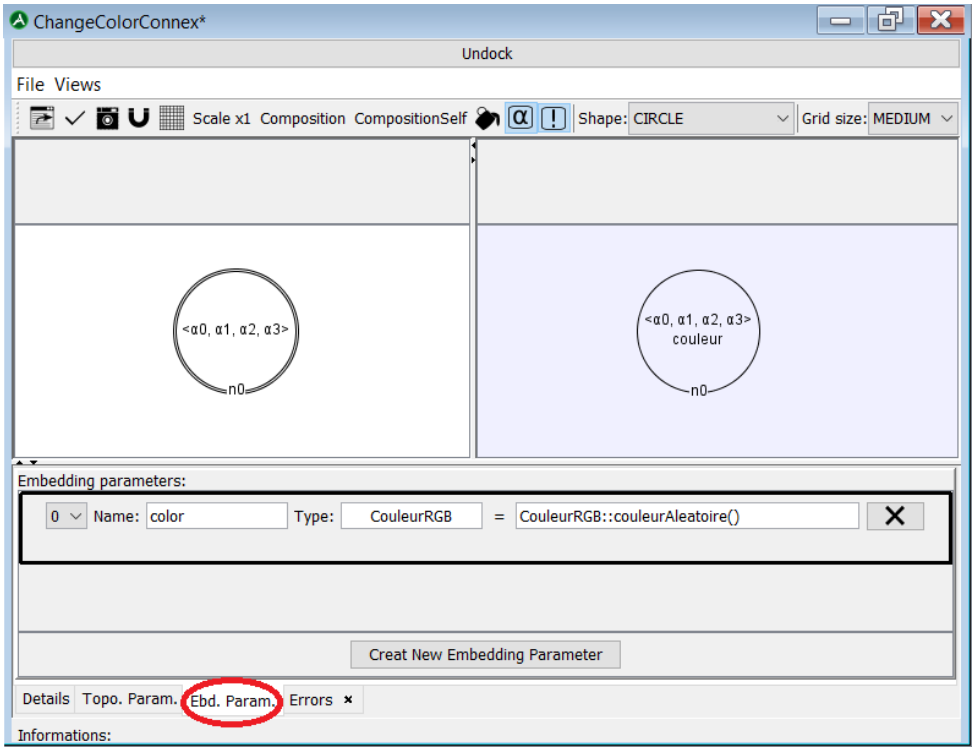


FIGURE 3.4 – Panneau des paramètres de plongements

L'ajout de paramètres de plongements explicites permet d'enrichir le contexte d'environnement et d'effectuer des vérifications. Par exemple, la déclaration des variables utilisées dans les expressions de plongement est désormais vérifiée. Ainsi, si un paramètre de plongement est défini dans la règle comme `color` sur la figure 3.4, cette variable est accessible dans tous les différents processus (pré-processus, précondition, *etc.*) et dans les expressions de plongement.

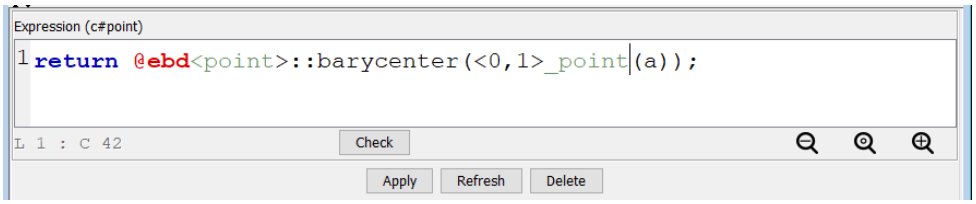


FIGURE 3.5 – Expression avec coloration syntaxique dans le nouvel éditeur

Pour rendre l'écriture dans le langage de calcul et de *script* plus ergonomique, nous avons implanté une coloration syntaxique. La figure 3.5 montre un exemple de cette coloration sur l'expression du plongement `point` du nœud `c` de la règle de triangulation barycentrique que nous avons vu en section 2.2, page 49.

Nous avons également amélioré le système de gestion des erreurs. Chaque règle possède à présent un panneaux contenant la liste de ses erreurs. Les erreurs sont catégorisées pour permettre à l'utilisateur de savoir si les vérifications indiquent une erreur sévère, une alerte, ou une simple information. Les informations sont levées lorsque les vérifications ne sont pas en mesure d'être testées. Par exemple, Jerboa n'est pas en mesure d'effectuer des vérifications sur le type de retour d'une fonction externe.

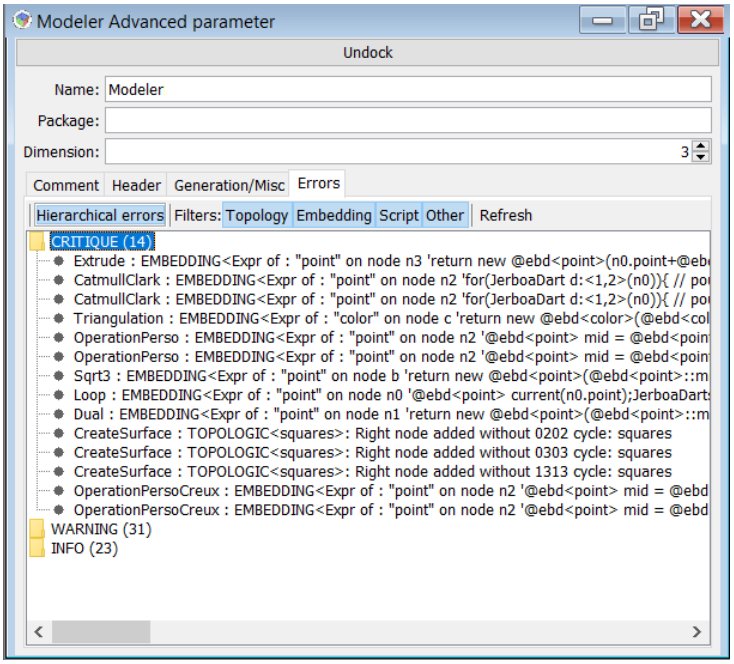


FIGURE 3.6 – Panneau d'erreur du modeleur rassemblant toutes les erreurs de toutes les opérations

Il est également possible d'accéder à toutes les erreurs du modeleur à partir de son panneau de paramétrage comme illustré en figure 3.6.

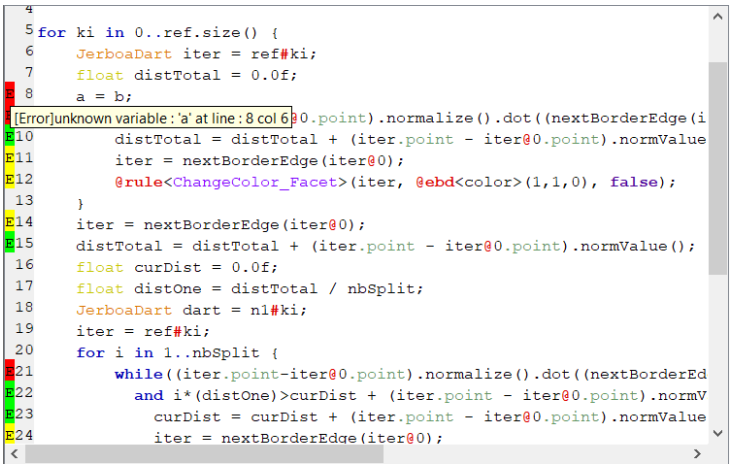


FIGURE 3.7 – Affichage des erreurs dans le panneau d'édition du langage de calcul

Avec le langage de calcul nous avons mis en place des vérifications qui indiquent la présence potentielle d'erreurs dans les expressions de plongements, les différents processus et dans les *scripts*. Nous avons donc ajouté une fonctionnalité à l'éditeur de texte des expressions pour afficher sur la ligne qui contient une erreur une indication colorée indiquant la gravité de l'erreur (rouge, jaune ou vert), comme illustré sur la figure 3.7. En passant la souris sur cet indicateur, un pop-up s'affiche pour indiquer à quelle erreur il correspond.

Avec un très grand nombre d'opérations, il est parfois difficile, même avec les noms des règles,



FIGURE 3.8 – Panneau de pré-visualisation d'une règle

de savoir à quoi chacune correspond avant de l'avoir ouverte dans l'éditeur. Nous avons donc mis en place un système de pré-visualisation des graphes des opérations en passant la souris sur le nom de la règle dans le panneau qui liste les opérations. Une pré-visualisation de l'opération de triangulation barycentrique est illustré en figure 3.8.

Afin de nous faciliter la tâche lors de l'écriture d'articles ou pour la présentation de notre travail, nous avons également implanté un export vectoriel des règles. Ainsi l'éditeur permet d'exporter au format SVG les règles avec toutes les informations affichées dans la vue graphique de l'éditeur (plongements, nom des nœuds, *etc.*). Cet outil nous a notamment servi à réaliser toutes les figures de ce manuscrit représentant les règles.

3.1.2 Vérifications automatiques

Les vérifications automatiques faites dans Jerboa, permettent aux utilisateurs de se concentrer sur les opérations et d'alléger les longues phases de recherche de bogues. La première version de l'éditeur vérifiait les aspects topologiques. Par exemple, un modeleur est paramétré par sa dimension topologique. Dans les orbites des nœuds des règles, il n'est pas possible d'utiliser une dimension inférieure à 0 ou supérieur à la dimension du modeleur. Les vérifications topologiques des règles comme la préservation des contraintes de cohérences des cartes généralisées par les règles (cf. section 1.3.4, page 34) étaient également déjà implantées dans l'ancien éditeur. Cependant, les vérifications des plongements qu'il intégrait étaient limités. En effet, jusqu'à présent, l'éditeur ne vérifiait que l'existence ou l'omission d'une expression sans aucune vérification syntaxique ou sémantique.

Dans la nouvelle version, le développement du langage d'expression a permis de réaliser des vérifications de préservation de cohérence sur les plongements ainsi que sur les *scripts*. Ces vérifications, avec celles sur la topologie, sont suffisantes pour garantir la préservation de la cohérence du modèle lors de l'application de nos opérations.

Nous avons également mis en places des vérifications plus classiques comme le typage par exemple. Le langage permet également de détecter le code mort, code qui ne sera jamais exécuté, notamment s'il est placé après une instruction `return` ou un `break` dans une boucle.

Ces vérifications sont cependant limitées car le langage est de haut niveau, et n'a pas accès à toutes les informations qui permettraient de réaliser des vérifications complètes. Nous n'avons évidemment pas vocation à remplacer une compilation complète. Nous déléguons certaines vérifications qui ne sont pas spécifiques à Jerboa au compilateur du langage sous-jacent dans lequel

sont générées les expressions actuellement (C++ ou Java). Par exemple, l'appel à des fonctions utilisateur (depuis une classe de plongement ou non), ne permet pas de connaître son type de retour car elles ont été définies à l'extérieur de Jerboa. Dans ce cas, l'éditeur informe seulement que le type de retour est inconnu et que la correction de l'expression n'est pas assurée pour cette instruction précise. Ceci n'a aucun impact sur la génération du code effectuée par l'éditeur, qui laisse la main au compilateur du langage cible, qui informera l'utilisateur d'une éventuelle erreur.

Cependant, il nous est possible grâce aux informations connues par l'éditeur de réaliser des vérifications spécifiques. Nous vérifions par exemple l'existence des plongements. En effet, l'expression `@ebd<positionGeometrique>(1,0,0)` n'est cohérente que si le modeleur contient un plongement nommé `positionGeometrique`. De même, nous vérifions au moment de l'appel de l'application d'une règle avec l'opérateur `@rule<Triangulation>()` si la règle `Triangulation` existe effectivement dans le modeleur. La cohérence des paramètres de règle est également vérifiée lors de l'application d'une règle. Ces vérifications sont similaires à celles d'un compilateur classique pour des appels de fonctions. Elles nécessitent de connaître l'ordre et le type des paramètres passés lors de l'appel. Dans la nouvelle version de l'éditeur, ces paramètres sont explicitement définis et ordonnés, ce qui nous permet d'effectuer cette vérification. Par convention, les paramètres topologiques sont donnés en premiers, puis suivent les paramètres de plongement. La présence de tous les paramètres topologiques est obligatoire, et une erreur est levée si l'un d'entre eux manque.

Le langage inclut un système de marquage des brins. Nous avons mis en place des vérifications qui permettent de déterminer si une marque est libérée ou non. Le langage teste pour chaque marque créée, si l'opérateur `delete` a bien été appelé sur leur variable respective avant l'utilisation d'un `return` ou d'un `break` selon le contexte, dans le `catch` si la marque est déclarée dans un `try` ou encore simplement avant la fin du code de l'expression ou du *script*.

Si la vérification des règles atomiques implique des vérifications de la topologie et des expressions de plongement, celle des *scripts* est triviale. Les *scripts* permettent l'enchaînement de l'application de règles ou de *scripts*. Le langage interdit toute modification de la topologie ou des plongements via les *scripts* autrement que par l'application de règles définies dans l'éditeur. Ainsi, l'application d'un *script* n'utilisant que des règles qui préservent la cohérence, préserve par définition la cohérence, et par construction, un second *script* qui utilise le premier, également. La vérification des *scripts* est donc une simple vérification syntaxique des instructions qu'il contient.

Avec son modèle topologique de G-carte, Jerboa permet de travailler en dimension quelconque, pourvu qu'elle soit fixée au départ dans le modeleur. Dans le langage, plusieurs expressions utilisent une dimension topologique qui doit être cohérente par rapport à celle du modeleur. C'est le cas notamment pour l'accès à un voisin topologique avec l'opérateur `@`, pour les collectes de brins ou de plongements et plus généralement lors de l'utilisation des orbites : `<0,1,2>`. Pour une dimension de modeleur n, chaque dimension d utilisée dans les expressions ou les *scripts*, doit satisfaire la condition suivante : $0 \leq d \leq n$. Bien entendu, si la dimension d utilisée est donnée via une variable, aucune supposition ne peut être faite sur la correction de l'expression.

3.2 Génération de code

Le langage Jerboa appartient à la catégorie des langages spécifiques à un domaine d'application (*Domain Specific Language* ou DSL [Fow10]). Le principal intérêt de ces langages est de fournir une syntaxe simple et compacte pour réaliser des opérations complexes de manière transparente sans que l'utilisateur ait à en connaître tous les détails de mise en œuvre. Cette section décrit la traduction du langage Jerboa dans le langage cible et les vérifications réalisées pendant ce traitement.

3.2.1 Construction de l'arbre syntaxique

Nous avons décrit dans la section 2.2, page 49, l'ajout d'un langage de calcul qui permet d'écrire les expressions de plongement des règles Jerboa. Nous avons également modifié le processus d'application des règles pour intégrer différents processus dont le comportement est décrit dans un langage dédié. Nous avons également décrit dans la section 2.3, page 57, un nouveau type d'opération appelé *script*, dont le comportement est décrit par un langage de *script*. Chacun de ces langages comporte ses propres spécificités. Par exemple, le langage de *script* permet d'appliquer des règles avec l'opérateur **@rule**. Cet opérateur n'existe pas syntaxiquement dans les expressions de plongement ni dans le langage des différents processus. Les différents langages sont cependant très proches et ne diffèrent que pour quelques constructions.

En pratique, nous avons donc implanté une seule grammaire regroupant toutes les constructions syntaxiques des langages présentés ci-avant. La syntaxe concrète reste très proche de nos inspirations, à savoir les langages C++ et Java et embarque des opérateurs dédiés à la manipulation des G-cartes. Rappelons que la description complète de cette grammaire est donnée en annexe (cf. annexe A, page 175).

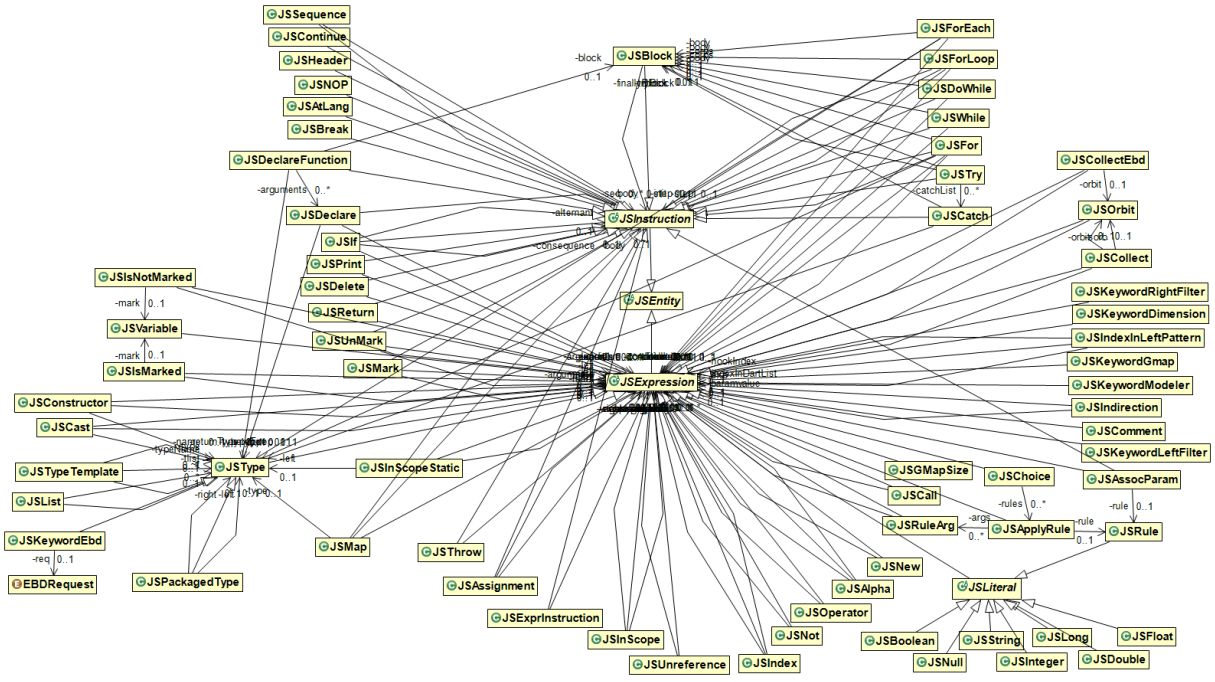


FIGURE 3.9 – Diagramme de classe de l'arbre syntaxique

La figure 3.9 présente le diagramme de classes de tous les termes de notre langage. Le grand nombre de classe est nécessaire pour réaliser les vérifications et l'écriture d'un code proche des habitudes d'un développeur C++ ou Java. L'analyse syntaxique de nos expressions produit donc une instanciation de tout ou partie de cette hiérarchie. Ces différentes classes permettent de définir les éléments du langage et la façon dont ces éléments s'imbriquent.

```
int x;
f(x+3);
```

Listing 3.1 – Exemple d'expression simple

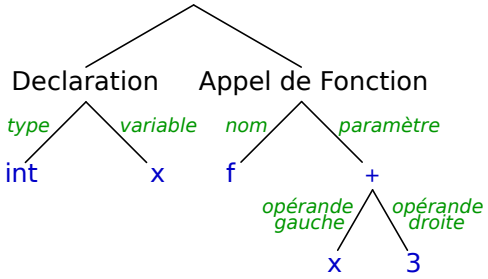


FIGURE 3.10 – Exemple d'arbre de syntaxe

Le listing 3.1 montre un exemple d'expression. La figure 3.10 montre la traduction de cette expression en arbre de syntaxe et illustre l'implantation de quelques classes de la figure 3.9.

Un des problèmes de l'unification des différents langages au sein d'une même syntaxe concrète, est que cela conduit à la possibilité d'écrire des constructions qui mixent deux langages abstraits. Par exemple, d'un point de vue grammatical il est possible d'appeler une règle dans une expression de plongement (ce qui est absolument interdit pour des problèmes de réentrance). Pour résoudre ce genre de problème. Nous avons ajouté une phase d'analyse sémantique dans nos arbres pour vérifier et contrôler nos transformations.

3.2.2 Analyse sémantique

L'analyse sémantique permet d'effectuer les premières vérifications et les traductions nécessaires pour préparer la génération du code adéquat. Dans notre langage, certaines expressions sont ambiguës comme : `b.point`. Cette expression peut avoir un sens différent selon le type de la variable `b`. Dans le cas où `b` est un brin de la G-carte, cette expression représente l'accès à la valeur du plongement `point` portée par le brin `b`. Si `b` est un résultat d'application de règle de type `JerboaRuleResult`, alors `point` est le nom d'un nœud de la règle, et l'expression renvoie l'ensemble des brins représentés par le nœud `point` dans le résultat de l'application de la règle. Dans les autres cas où `b` est d'un type utilisateur `T`, cette expression peut se traduire comme l'accès à un attribut existant de `b` défini dans la classe `T`.

L'arbre syntaxique généré lors de l'analyse d'une expression n'est pas suffisant pour être en mesure d'interpréter une expression. Il est nécessaire de lever les différentes ambiguïtés pour déterminer la bonne sémantique de cette expression. Pour cela, nous procédons à des analyses successives de l'arbre syntaxique initial, dans notre jargon nous les nommons arbres intermédiaires. Ces analyses consistent à définir un contexte de traduction. Le contexte de traduction décrit l'ensemble des variables qui ont été déclarées, leur portée, leur type, *etc.* Prenons l'exemple suivant :

```
T b;
f(b.point);
```

Les différents arbres sémantique possibles de cette expression sont montrés en figure 3.11. Sur cet exemple, on observe que selon le type de la variable `b`, l'interprétation sémantique est différente. Notons que dans le cas où le type de `b` est un type utilisateur dont nous ne connaissons pas les attributs, nous levons un avertissement, afin d'en informer l'utilisateur que nous ne sommes pas en mesure d'effectuer des vérifications dans du code extérieur à Jerboa.

Grâce à l'éditeur de modeleur, nous sommes en mesure lors de la traduction d'une expression, de savoir si l'expression à traduire est une expression de plongement, le code d'un *script*, une précondition, *etc.* Cette information est très importante pour l'analyse sémantique car elle définit un

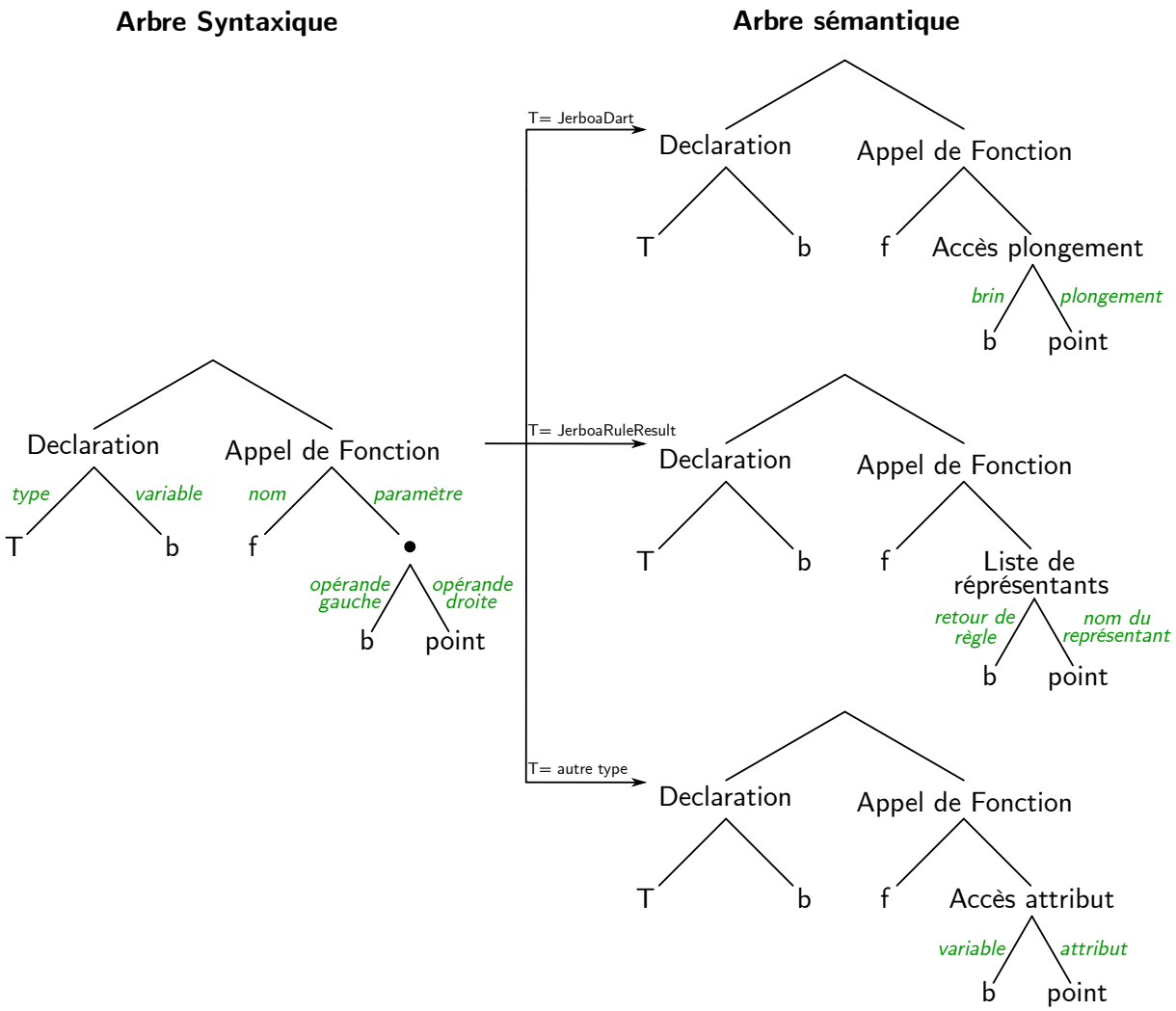


FIGURE 3.11 – Transformation de l'arbre syntaxique (à gauche) en fonction du type de b

contexte de traduction initial des éléments utilisables dans l'expression. Ainsi, certains opérateurs n'existent que dans certains contextes et l'analyse sémantique détecte les utilisations incohérentes. L'opérateur **@rule** par exemple dont nous avons déjà parlé dans la section précédente, ne peut être utilisé que dans le contexte de l'écriture d'un *script*. Nous vérifions ainsi que la syntaxe est cohérente avec le contexte de traduction.

Dans le langage, l'accès à une variable topologique peut représenter plusieurs choses différentes, selon le contexte de traduction. Dans une expression de plongement, nous avons vu que les variables topologiques représente l'accès à un brin filtré par la règle. Dans un *script*, les nœuds des graphes ont une multiplicité, donc une variable topologique représente une liste de brins. Dans un pré-traitement, le filtrage de la règle n'a pas encore été effectué, donc une variable topologique représente le brin accroché de l'objet sur lequel la règle s'applique. Dans les pré-conditions, traitements intermédiaires ou post-traitments, une variable topologique représente la liste des brins filtrés par la variable dans le cas d'une règle, et la liste des brins accrochés dans le cas d'un *script*.

Reprenons la règle de triangulation barycentrique d'une unique face, illustrée en figure 3.12. La pré-condition suivante permet de tester si le motif filtré par `a` contient 6 brins, et est donc un

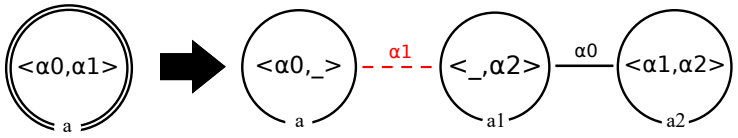
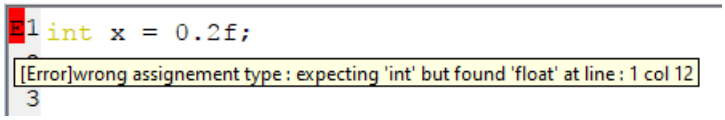


FIGURE 3.12 – Règle de triangulation barycentrique

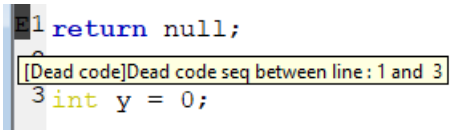
triangle. Si c'est le cas, la valeur de retour est `false` afin d'arrêter l'application de la règle pour ne pas trianguler un triangle.

```
return a.size()!=6;
```

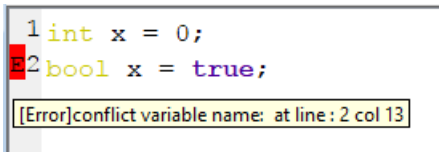
Ici, la variable `a` représente la liste de tous les brins filtrés par le nœud `a`.



(a) Erreur de type



(b) Détection de code mort



(c) Conflict de nom de variable

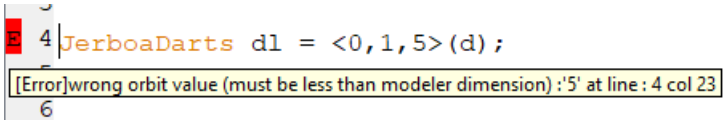
FIGURE 3.13 – Impressions d'écran montrant des erreurs de programmation détectées par l'éditeur

Après la résolution des ambiguïtés de typage, nous procédons à plusieurs transformations de l'arbre intermédiaire afin de réaliser d'autres vérifications, comme la détection de code mort, les conflits de nom, *etc.* Des impressions d'écran de ces erreurs détectées par l'éditeur sont présentées en figure 3.13.

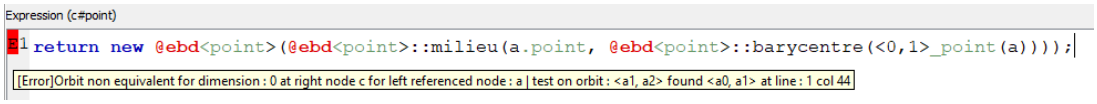
En plus des vérifications classiques, nous avons réalisé des vérifications propres à Jerboa. La première est la vérification de la cohérence des dimensions dans les orbites. Par exemple, dans un modeleur de dimension 3, nous sommes en mesure de détecter des orbites incohérentes telle que : $\langle 3, 4 \rangle$. La détection de cette erreur par l'éditeur est illustré en figure 3.14(a). Bien évidemment, ce problème peut ne pas être décidable si l'utilisateur effectue un appel d'orbite au travers d'expressions de dimensions. Par exemple, la vérification ne peux pas déterminer à priori si une orbite de type $\langle d1, d2 \rangle$, avec $d1$ et $d2$ des variables entières, est correcte ou non. Cette vérification ne peut être faite qu'à l'exécution du programme car la valeur des variables ne sont connues qu'à l'exécution.

Nous avons également mis en place la vérification de l'équivalence à orbite près, afin de garantir la préservation de la cohérence des plongements. Cette vérification consiste à déterminer si l'instanciation des expressions est identique pour tous les brins d'une même orbite de plongement, pour toutes les instanciations possibles de la règle. La détection de l'erreur de l'expression présentée en section 1.3.4, page 34 par l'éditeur est illustré en figure 3.14(b).

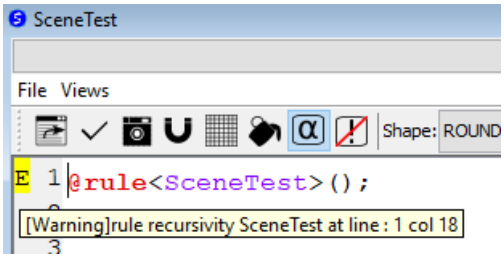
Un *script* est en mesure d'appeler l'application d'autres opérations. Il est donc possible qu'il s'appelle lui-même pour faire des opérations récursives. Cependant, les règles et les *scripts* ont



(a) Détection de l'utilisation d'une dimension supérieur à celle du modeleur



(b) Détection de l'équivalence à orbite près



(c) Détection d'un appel récursif

FIGURE 3.14 – Impressions d'écran montrant les erreurs spécifiques à Jerboa détectées par l'éditeur

des états. Les paramètres de plongements n'étant pas obligatoires, leur valeur est conservée d'un appel à l'autre du *script* s'ils ne sont pas fournis au moment de l'appel. Ainsi, nous déconseillons les appels récursifs, à moins de connaître précisément le comportement des opérations, ou de fournir tous les paramètres de plongements au moment de l'appel. Pour cette raison l'éditeur lève dans ce cas une information à l'utilisateur comme illustré en figure 3.14(c).

Pour conclure, nous avons mis en place différentes vérifications partant de l'arbre syntaxique initial pour obtenir successivement des arbres intermédiaires, capables de lever certaines ambiguïtés ou d'apporter des informations complémentaires. Une fois les ambiguïtés levées et toutes les informations retrouvées, nous sommes en mesure d'effectuer la traduction effective dans le langage cible.

3.2.3 Traduction dans le langage cible

Dans les étapes précédentes nous avons résolu tous les problèmes potentiels empêchant la traduction des expressions dans le langage cible. Nous avons procédé à plusieurs vérifications permettant de limiter les erreurs. Cependant, nous n'avons pas pour objectif de déterminer absolument toutes les erreurs. Les compilateurs actuels permettent de faire des optimisations de bas niveaux beaucoup plus poussées et donc de tirer ainsi le meilleur parti de chaque langage sous-jacent, nous avons donc préféré générer du code dans le langage cible sans erreur dans les parties concernant le langage Jerboa. Et, ensuite nous donnons la main à un vrai compilateur pour vérifier les dernières erreurs qui ne sont plus du ressort de notre langage, comme la vérification de l'existence d'un attribut dans un type utilisateur.

Générer le code dans deux langages n'est pas évident, car il faut tenir compte des spécificités de chacun. Malgré des caractéristiques communes dans le fonctionnement du C++ et du Java, il est a noté une réelle difficulté due à l'existence des pointeurs en C++ et qui ne se retrouve

pas en l'état en Java. En particulier, ce choix de pointeur dans le noyau Jerboa en C++ pour partager les plongements au sein d'une orbite, doit se retrouver harmonisé dans l'utilisation de notre langage de plus haut niveau sans ajouter les contraintes syntaxiques liées aux pointeurs.

De plus, lors de la génération dans le langage cible, une instruction en langage Jerboa est parfois traduite par plusieurs instructions dans le langage cible. Prenons l'exemple d'une opération écrite dans l'ancienne version de Jerboa où les *scripts* n'existaient pas, et nous devions écrire les enchaînement d'application de règle manuellement (avec `n0` et `n1` des brins de la G-carte) :

```
JerboaInputHooksGeneric brinsAccroche = new JerboaInputHooksGeneric();
brinsAccroche.addCol(n0);
brinsAccroche.addCol(n1);
CoutureAlpha2 regleCA2=(CoutureAlpha2)modeler.getRule("CoutureAlpha2");
JerboaRuleResult resultat = regleCA2.applyRule(gmap, brinsAccroche);
```

Ici l'opération crée une liste de brins accrochés dans laquelle est ajoutée les brins `n0` et `n1`. On recherche ensuite la règle de couture par α_2 dans le modeleur pour l'appliquer sur les brins accrochés, en stockant le résultat de l'application dans une variable `resultat`.

Dans le nouveau langage cette opération complète peut être réduite à l'expression suivante :

```
JerboaRuleResult resultat = @rule<CoutureAlpha2>(n0,n1);
```

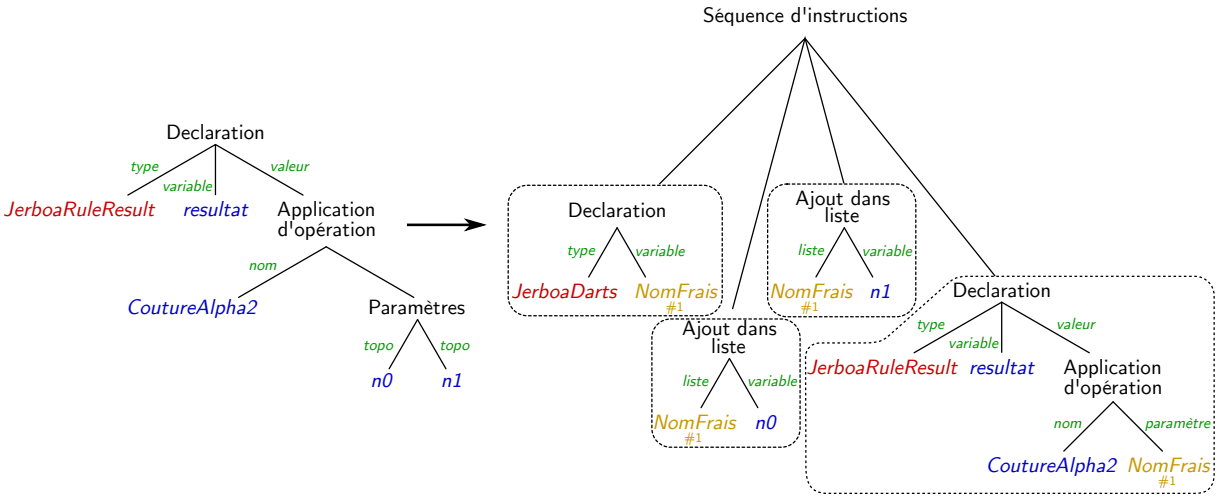


FIGURE 3.15 – Traduction de l'arbre sémantique en arbre de syntaxe abstraite

L'arbre de syntaxe associé à cette expression est illustré sur la partie gauche de la figure 3.15. Afin de réaliser la traduction nous devons créer des arbres intermédiaires comme nous l'avons vu dans la section précédente, afin de produire un arbre similaire à celui de la partie droite de la figure. Cet arbre a été créé à partir des arbres intermédiaires, afin de créer des structures au plus proche du code généré. On observe dans cet arbre que nous avons besoin de créer plusieurs variables temporaires. Nous avons donc mis en place un système de génération de "nom frais" de variables, afin de générer des noms qui n'entrent pas en conflit avec d'autres variables. Dans cet exemple le nom attribué à la variable temporaire qui permet de stocker les brins accrochés est `_v_hook0`. Avec l'arbre complet nous sommes en mesure de générer le code dans les deux langages.

Ainsi l'expression `@rule<CoutureAlpha2>(n0,n1);` est traduite en Java par :

```
JerboaInputHooksGeneric _v_hook0 = new JerboaInputHooksGeneric();
_v_hook0.addCol(n0);
_v_hook0.addCol(n1);
((CoutureAlpha2)modeler.getRule("CoutureAlpha2")).applyRule(gmap,
    _v_hook0);
```

où `modeler` représente le modeleur propriétaire du *script* courant.

Et en C++ par :

```
JerboaInputHooksGeneric _v_hook0;
_v_hook0.addCol(n0);
_v_hook0.addCol(n1);
((CoutureAlpha2*)_owner->rule("CoutureAlpha2"))->applyRule(gmap,
    _v_hook0);
```

où `_owner` représente le modeleur propriétaire du *script* courant.

Prenons un exemple plus complet d'un *script* qui crée un cube de 2 unités de coté, à partir d'une surface carrée :

```
JerboaRuleResult res = @rule<CreationCarre>();
@rule<Extrude>(res.n0#0);
@rule<Homothetie>(res.n0#0,@ebd<point>(2,2,2),false);
delete res;
```

La traduction de cette opération en C++ est la suivante (aux commentaires près) :

```
1  JerboaInputHooksGeneric _v_hook0;
2  // Application de la règleCreationCarre avec stockage
3  // du résultat dans "res"
4  JerboaRuleResult* res = ((CreationCarre*)_owner->rule("CreationCarre"))
       ->applyRule(gmap, _v_hook0, JerboaRuleResultType::FULL);
5  JerboaInputHooksGeneric _v_hook1;
6  _v_hook1.addCol(res->get(0,0));
7  // Application de la règle Extrude
8  ((Extrude*)_owner->rule("Extrude"))->applyRule(gmap, _v_hook1);
9  JerboaInputHooksGeneric _v_hook2;
10 _v_hook2.addCol(res->get(0,0));
11 // Passage des paramètres de plongements pour la règle Homothetie
12 ((Homothetie*)_owner->rule("Homothetie"))->setscaleVector(Point3(2,2,2))
       ;
13 ((Homothetie*)_owner->rule("Homothetie"))->setaskToUser(false);
14 // Application de la règle Homothetie
15 ((Homothetie*)_owner->rule("Homothetie"))->applyRule(gmap, _v_hook2);
16 delete res;
```

La traduction en Java est la suivante (aux commentaires près) :

```
1  JerboaInputHooksGeneric _v_hook0 = new JerboaInputHooksGeneric();
2  // Application de la règle CreationCarre avec stockage
3  // du résultat dans "res"
4  JerboaRuleResult res = ((CreationCarre)modeler.getRule("CreationCarre"))
       .applyRule(gmap, _v_hook0);
5  JerboaInputHooksGeneric _v_hook1 = new JerboaInputHooksGeneric();
6  _v_hook1.addCol(res.get(0).get(0));
```

```
7  // Application de la règle Extrude
8  ((Extrude)modeler.getRule("Extrude")).applyRule(gmap, _v_hook1);
9  JerboaInputHooksGeneric _v_hook2 = new JerboaInputHooksGeneric();
10 _v_hook2.addCol(res.get(0).get(0));
11 // Passage des paramètres de plongements pour la règle Homothetie
12 ((Homothetie)modeler.getRule("Homothetie")).setScaleVector(new Vector
       (2,2,2));
13 ((Homothetie)modeler.getRule("Homothetie")).setAskToUser(false);
14 // Application de la règle Homothetie
15 ((Homothetie)modeler.getRule("Homothetie")).applyRule(gmap, _v_hook2);
```

On peut observer quelques différences entre les deux codes générés, hormis les noms de fonctions spécifiques à chaque moteur. En Java, l'instanciation d'un objet se fait avec l'opérateur *new*, alors qu'en C++, le code produit un objet local. De même, la gestion automatisée de la mémoire en Java rend la suppression des objets implicites, alors qu'en C++ la gestion mémoire doit être faite explicitement. Ainsi, la traduction de l'opérateur *delete* sur un pointeur (ici un résultat de règle) est généré en C++, mais pas en Java.

Notons également que l'accès au paramètre topologique du résultat de règle (ligne 2 du *script* et ligne 6 des listings des codes générés) dont l'expression est `res.n0#0`, est transcrite en traduisant `n0` par son numéro de place dans la liste des nœuds d'accroche du *script* (ici 0). Ce qui optimise et démontre l'intérêt de l'analyse sémantique et les remplacements opérés lors des arbres intermédiaires.

On observe également que dans le code généré, l'ajout de paramètres topologiques se fait avec l'ajout des brins accrochés dans une liste dédiée. Mais l'ajout des paramètres de plongement est fait avec l'appel à des fonctions qui modifient l'attribut adéquat dans la classe de la règle comme pour le paramètre `ScaleVector`, pour lequel on fait appel à l'accesseur `setScaleVector` (cf ligne 12 dans les codes générés).

Nous avons fait de même pour l'accès aux plongements. Prenons l'exemple de l'expression d'accès au plongement `point` sur un brin `brin` : `brin.point`. Dans le moteur, nous avons implanté une fonction qui permet d'accéder à la valeur d'un plongement par son nom : `brin.getEmbedding("point")`. Cette fonction n'est cependant pas très efficace car elle recherche dans la liste des plongements du modeleur le plongement qui porte le nom passé en paramètre pour trouver son numéro d'identification, et accéder à la valeur de ce plongement pour le brin courant. Nous avons donc mis en place une optimisation de la traduction de l'accès à un plongement en traduisant le nom du plongement par son identifiant interne. Cet identifiant est connu dès l'édition car c'est à ce moment qu'est paramétré le modeleur et que ses plongements sont définis. Ainsi l'expression `brin.point` est traduite par : `brin.getEmbeddging(0)`. Si `point` est le premier plongement défini dans l'éditeur. Dans ce cas, `brin.couleur`, le deuxième plongement, est traduit par : `brin.getEmbeddging(1)`.

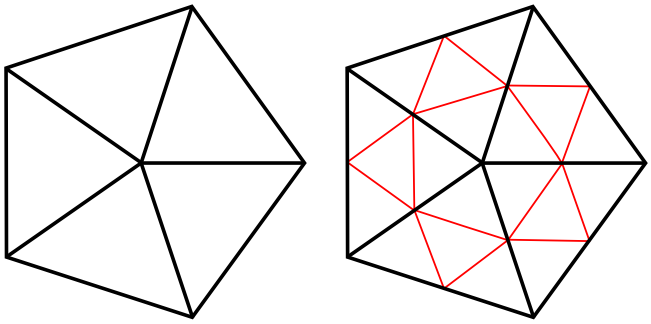
3.3 Composition de règles

D'un point de vue utilisateur, la création d'une opération complexe s'obtient par composition de plusieurs autres opération. Le langage de *script* a été développé en ce sens. L'optimisation de manière transparente pour l'utilisateur de telles compositions seraient très intéressante. Nous avons donc commencé à étudier cette problématique en proposant dans l'éditeur la possibilité de composer directement des règles Jerboa pour accélérer certains traitements redondants.

La composition de deux règles Jerboa est l'application d'une règle R_2 sur le résultat d'une autre règle R_1. Les règles Jerboa transforment les graphes des objets. Les règles étant composées

elles mêmes de graphes, il est possible de transformer le graphe droit d'une règle en lui appliquant une autre règle.

Cette section présente le fonctionnement de la composition de règles existantes au sein de l'éditeur de modeleur. Actuellement, l'outil compose la topologie des règles. La composition des expressions de plongement doit être réalisée manuellement par l'utilisateur. Nous détaillerons plus précisément cet aspect en section 5.2.1, page 105.



(a) Objet (b) Objet subdivisé

FIGURE 3.16 – Application du schéma de Loop sur un pentagone triangulé

Prenons l'exemple du schéma de subdivision de Loop [Loo87]. Ce schéma de subdivision permet de lisser des maillages à condition qu'ils soient triangulés. Ce schéma subdivise des faces triangulaires en insérant un sommet au milieu de chacune de leurs arêtes, et les relie pour former une face triangulaire centrale comme illustré en rouge sur la figure 3.16. Nous verrons cette opération plus en détail dans la section 4.1.1, page 90. Le schéma de Loop devant s'appliquer à des maillages triangulés, on peut imaginer que pour l'appliquer à un maillage quelconque, on commence par trianguler ce maillage avant d'appliquer le schéma de Loop. Nous avons choisi de créer une opération qui compose la triangulation barycentrique des faces d'un volume isolé, avec le schéma de Loop sur le même type d'objet.

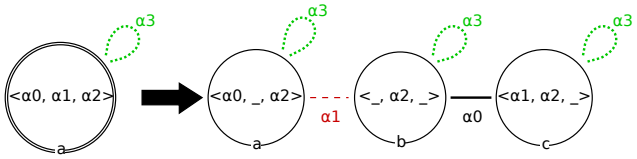


FIGURE 3.17 – Règle Jerboa de la triangulation barycentrique d'un volume isolé

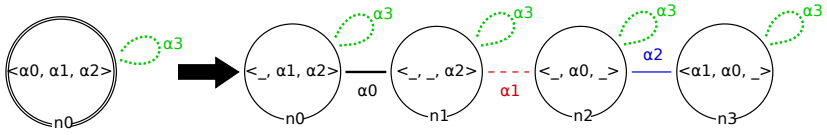


FIGURE 3.18 – Règle Jerboa du schéma de subdivision de Loop d'un volume isolé

Pour composer les deux opérations, on applique sur le graphe droit de la règle de triangulation barycentrique des faces d'un volume isolé illustrée en figure 3.17, la règle Jerboa du schéma de Loop illustrée en figure 3.18. Nous pouvons reprendre le processus d'application des règles décrit dans la section 1.3.2, page 21. Pour appliquer la règle de Loop, on associe son nœud d'accroche

n0 à l'un des nœuds du motif droit de la règle de triangulation. Ici le motif gauche de la règle de Loop filtre l'orbite $\langle 0, 1, 2 \rangle$. Le nœud n0 permet de filtrer entièrement le motif droit de la règle de triangulation, quel que soit le nœud accroché choisi dans le graphe droit de cette dernière (nous choisissons a pour présenter la suite). Commençons le processus d'application de la règle par la création des copies du motif filtré.

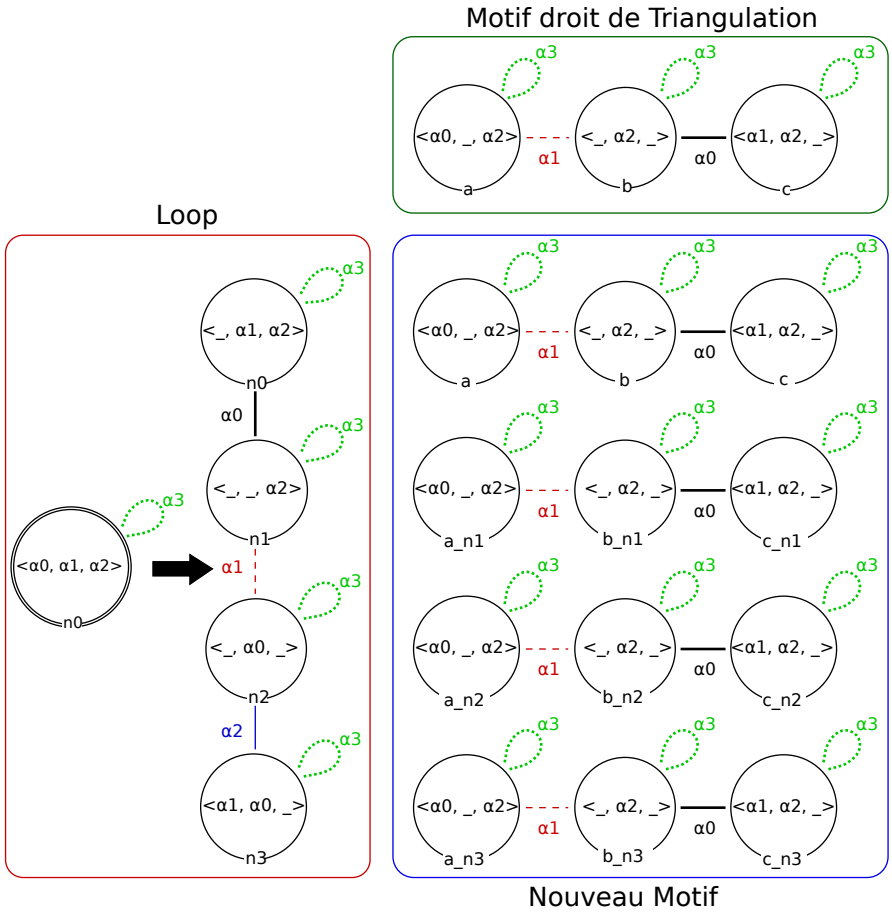


FIGURE 3.19 – Copie du motif filtré

Le motif filtré est préservé par la règle de Loop, il est donc présent dans le nouveau motif. Ce motif filtré est représenté sur la première ligne du nouveau motif de la figure 3.19, face à n0. Pour comprendre les mécanismes de copie et de renommage, nous avons exposé en haut des figures le motif droit de la règle de triangulation, qui est ici le graphe sur lequel la règle Loop s'applique. Cette dernière est exposée sur la gauche.

Le graphe droit de la règle de Loop comporte quatre nœuds et crée donc trois nouvelles copies du motif filtré par n0. Ainsi le nouveau motif comporte 3 copies du motif filtré par n0, correspondant aux copies des nœuds a, b, c, par les nœuds n1, n2 et n3. Le nouveau motif présente donc 12 nœuds. Nous avons nommé les nœuds créés par la règle de Loop en composant le nom de leur nœud de référence (a, b ou c) avec le nœuds du motif droit de la règle de Loop qui leur correspond.

Appliquons à présent les transformations topologiques définies dans la règle de Loop, comme illustré sur la figure 3.20. Considérons les transformations effectuées sur n2, les liaisons α_0 de n2 sont supprimées. Sur la figure, la suppression est marquée par le cercle rouge entre b_n2 et c_n2.

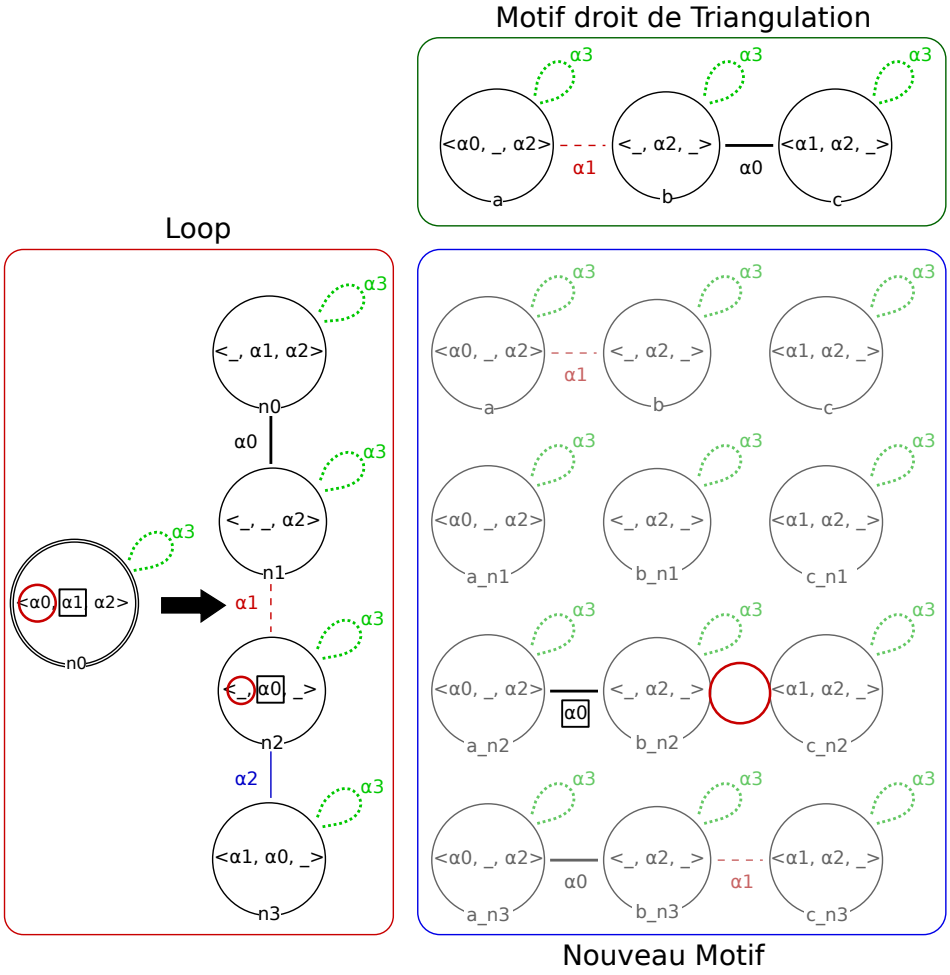


FIGURE 3.20 – Renommage des liaisons explicites

Ainsi, les liaisons α_0 explicites entre les copies que représente `n2`, soit les nœuds `a_n2`, `b_n2` et `c_n2`, sont supprimées. De même, les liaisons α_1 sont renommées en α_0. Cette transformation, marquée d'un carré noir sur la figure, est appliquée aux nœuds représentés par `n2`. Enfin, les liaisons α_2 sont supprimées. Les nœuds `a_n2`, `b_n2` et `c_n2` ne présentent pas de liaison α_2 explicite, il n'y a donc pas de transformation. On applique ce processus pour toutes les autres transformations et obtenons le nouveau motif de la figure 3.20.

Le processus de renommage des liaisons implicites est similaire. Nous appliquons donc les mêmes renommages dans les orbites des nœuds. On obtient ainsi le nouveau motif illustré sur la figure 3.21.

Le dernier type de transformation topologique à appliquer est l'ajout des liaisons explicites comme le montre la figure 3.22. La règle de Loop présente une liaison α_0 entre les nœuds `n0` et `n1`. On crée donc des liaisons α_0 entre les couples {`a, a_n1`}, {`b, b_n1`} et {`c, c_n1`}. On complete le nouveau motif avec les autres liaisons explicites de la règle de loop : un α_1 entre `n1` et `n2`, et un α_2 entre `n2` et `n3`. On obtient ainsi le motif droit final de la règle de composition.

Le motif gauche de la règle composée, est le motif filtré par la première règle de la composition, c'est-à-dire le motif gauche de la règle de triangulation barycentrique. La figure 3.23 représente la règle composée de la la règle de triangulation barycentrique illustrée en figure 3.17 avec la règle du schéma de subdivision de Loop illustrée en figure 3.18.

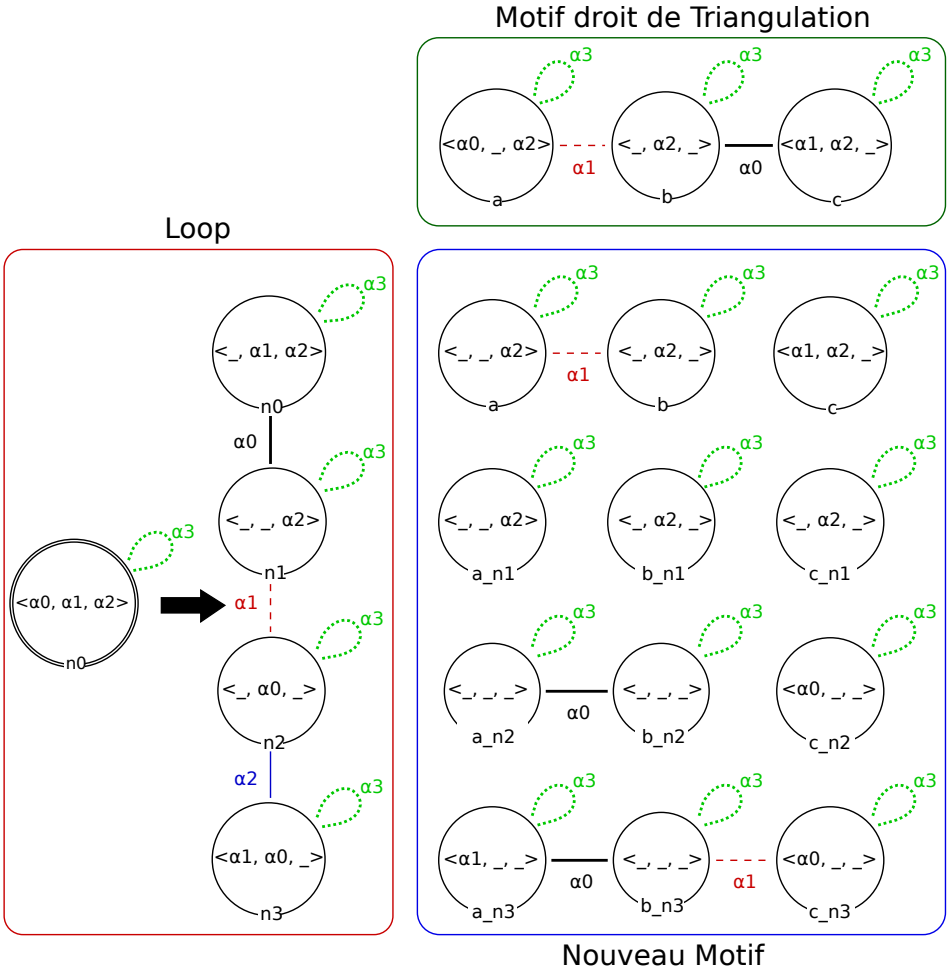


FIGURE 3.21 – Renommage des liaisons implicites

Actuellement, notre outil de composition fonctionne uniquement pour les règles qui comportent un unique nœud à gauche. Les transformations sont appliquées sur tout le motif droit de la 1ere règle.

Pour composer deux règle, la 2e doit filtrer un sous-motif du motif droit de la 1ere. Par exemple, si nous avions pris la règle de triangulation d'une unique face, c'est-à-dire celle qui ne filtre pas les α_2 (cf. figure 3.24), nous n'aurions pas été en mesure de la composer avec la règle de Loop. En effet, cette dernière décrit des modifications des liaisons α_2 filtrées, et filtre une orbite plus large que le motif droit de la règle de triangulation. Ainsi, la composition de cette règle de triangulation avec la règle de Loop (dans cet ordre), produit une règle qui ne préserve pas la cohérence topologique. La règle composée est illustrée en figure 3.25. Les vérifications automatiques de l'éditeur désignent une erreur sur le nœud `a_n1`, marqué d'un point d'exclamation sur la figure. Ici `a_n1`, ne présente pas de liaison α_2. Pour garantir la préservation de la contrainte de cycle des G-carte, le nouveau sommet créé par la règle de Loop, représenté par `a_n1` et `a_n2`, doit également être créé sur l'arête adjacente. Or cette arête n'est pas filtrée par le motif de triangulation qui filtre uniquement la face locale.

Nous avons prévu d'améliorer notre outil pour lui donner la possibilité de gérer la composition de règles comportant plusieurs nœuds à gauche. Pour ce faire, il faut demander à l'utilisateur de définir l'association des différents nœuds d'accroche. Nous mettrons en place des vérifications

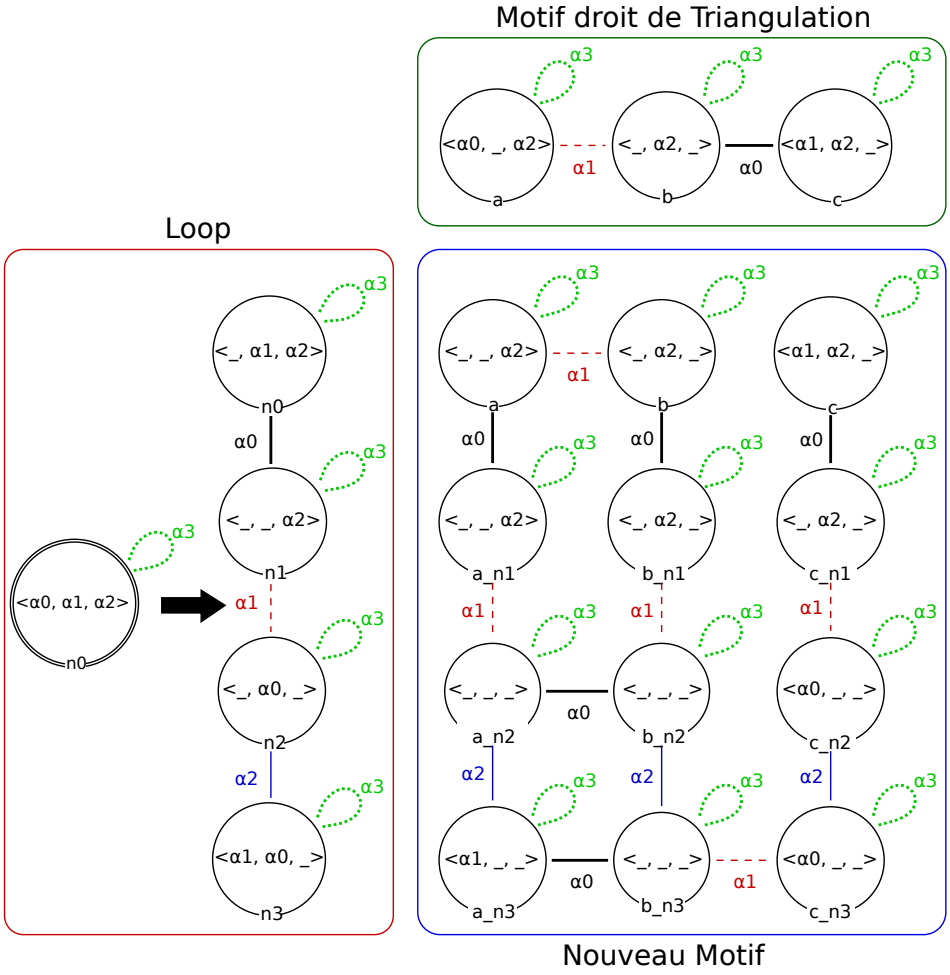


FIGURE 3.22 – Ajout des liaisons explicites

afin de déterminer si chaque nœud d'accroche peut effectivement être accroché au motif droit de la première règle. La composition d'opérations est un processus similaire à l'application d'une règle de transformation de graphe. Nous définirons donc un processus de filtrage afin de pouvoir gérer tous les cas possibles, comme nous le faisons pour l'application d'une règle sur un objet.

Nous avons pu testé la viabilité de cet outil en l'état et il offre déjà des performances prometteuses (nous verrons un test de performance dans la section 4.2, page 94). Il permet également de détecter les expressions de plongements qui doivent être définies. Nous souhaitons également ajouter un processus transparent pour l'utilisateur afin de composer également les expressions de plongements. Nous présentons notamment un exemple illustrant cet aspect en section 5.2.1, page 105.

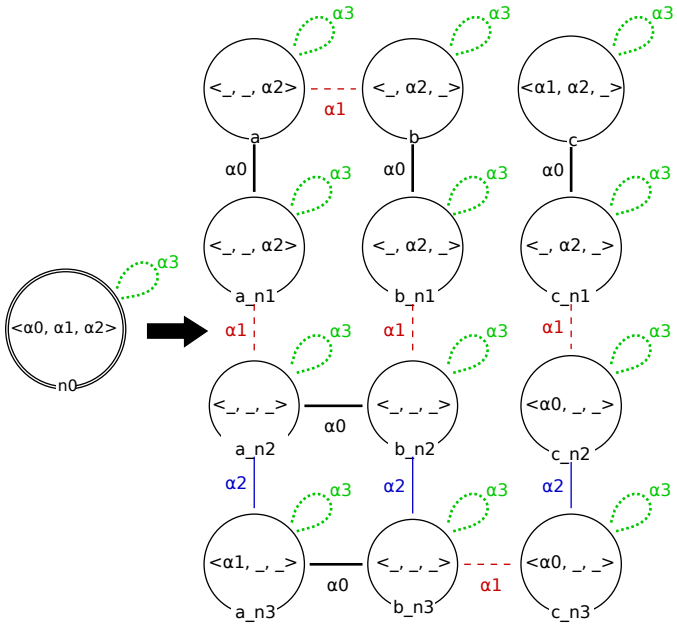


FIGURE 3.23 – Règle de composition de la règle de triangulation barycentrique illustrée en figure 3.17 avec la règle du schéma de subdivision de Loop illustrée en figure 3.18

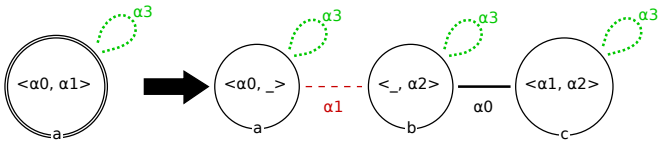


FIGURE 3.24 – Règle Jerboa de la triangulation barycentrique d'une face

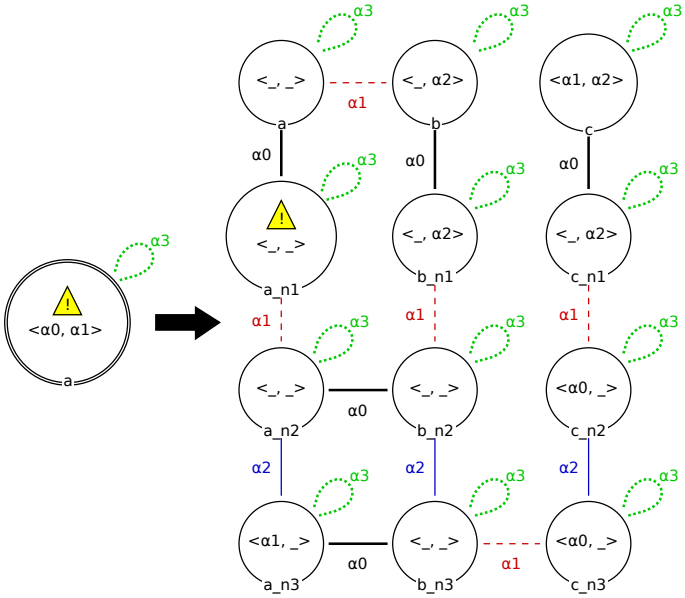


FIGURE 3.25 – Règle Jerboa de composition de la triangulation de la figure 3.24 avec celle du schéma de subdivision de Loop illustrée en figure 3.18

3.4 Spécialisation du moteur d'application

L'application des opérations définies par des règles de transformation de graphes dans Jerboa est réalisée par le moteur d'application. Ce moteur est le résultat fondamental des transformations de graphe. L'implantation de ce moteur permet d'appliquer n'importe quelle règle que l'on modélise dans notre éditeur et qui passe les étapes de vérifications. Pour ce faire, le moteur réalise plusieurs étapes illustrées en figure 3.26 :

1. le filtrage du motif gauche ;
2. la création des nouveaux brins ;
3. la liaison des nouveaux brins ;
4. le calcul et l'affectation des plongements en utilisant si besoin le motif gauche non encore modifié ;
5. la liaison du motif filtré avec les nouveaux brins ;
6. la propagation des plongements calculés sur leur orbite support.

Toutes ces étapes ne sont pas nécessaires pour toutes les opérations. Par exemple, les règles de création, qui n'ont pas de motif gauche à filtrer, n'ont pas besoin des étapes 1 et 5. De même, une opération de mise à jour de plongements qui ne fait aucune modification topologique n'a pas besoin des étapes 2, 3 et 5. D'autres types de règles seraient identifiables comme celles qui réalisent uniquement des modifications topologiques sans modification de plongement (*e.g.* règle de couture), ou encore des règles de suppression (supprimer une composante connexe par exemple). Dans un soucis d'optimisation mémoire et/ou de temps de calcul, il est possible de créer un moteur dédié à chaque type d'opération identifié. Comme dans tout programme, une opération spécialisée est plus facilement optimisable qu'un programme générique, en utilisant des structures de données, ou des méthodes de calcul spécifiques par exemple.

L'attribution à chaque règle d'un moteur optimisé peut être réalisée à deux moments. Soit lors de la génération du code par l'éditeur qui peut détecter statiquement le type de règle et lui attribuer directement un moteur spécifique. Soit au moment de la création du modeleur dans l'application finale. Dans ce cas le moteur est attribué dynamiquement à chaque lancement de l'application. Nous avons choisi d'implanter l'attribution dynamique au moment du lancement de l'application permet de ne pas avoir à re-générer le modeleur via l'éditeur en cas de mise à jour du noyau de Jerboa, mais surtout de tenir compte des spécificités matérielles de la machine sur laquelle l'application est exécutée. Par exemple, un moteur d'application optimisé pour paralléliser son traitement pourrait choisir la meilleure technologie en fonction de son environnement d'exécution. Ainsi, l'utilisation de la parallélisation avec la technologie CUDA [Nvi11] n'est possible qu'avec des cartes de marque Nvidia au lieu d'utiliser un moteur faisant de la parallélisation sur processeurs (qui couvre une plus grande catégorie de machines). Ce type de moteur s'activerait ou non en fonction de la machine sur laquelle l'application s'exécute, sans que l'utilisateur n'ait à s'en soucier et sans modification du noyau de Jerboa.

Cependant, cette solution implique que nous ajoutons un traitement dynamique (et donc potentiellement coûteux) au moment de l'attribution du moteur à l'instantiation de la règle dans le système. Cependant, cette étape joue uniquement sur le lancement de l'application et non sur l'exécution de notre programme et se produit qu'une seul fois au cours de la vie de notre modeleur. Nous le considérons dans les faits comme négligeable face à la généricité et la facilité d'utilisation de cette approche. Le noyau de Jerboa comprend actuellement 2 moteurs d'application. Le premier est le moteur générique qui permet d'appliquer n'importe quelle règle définie dans l'éditeur.

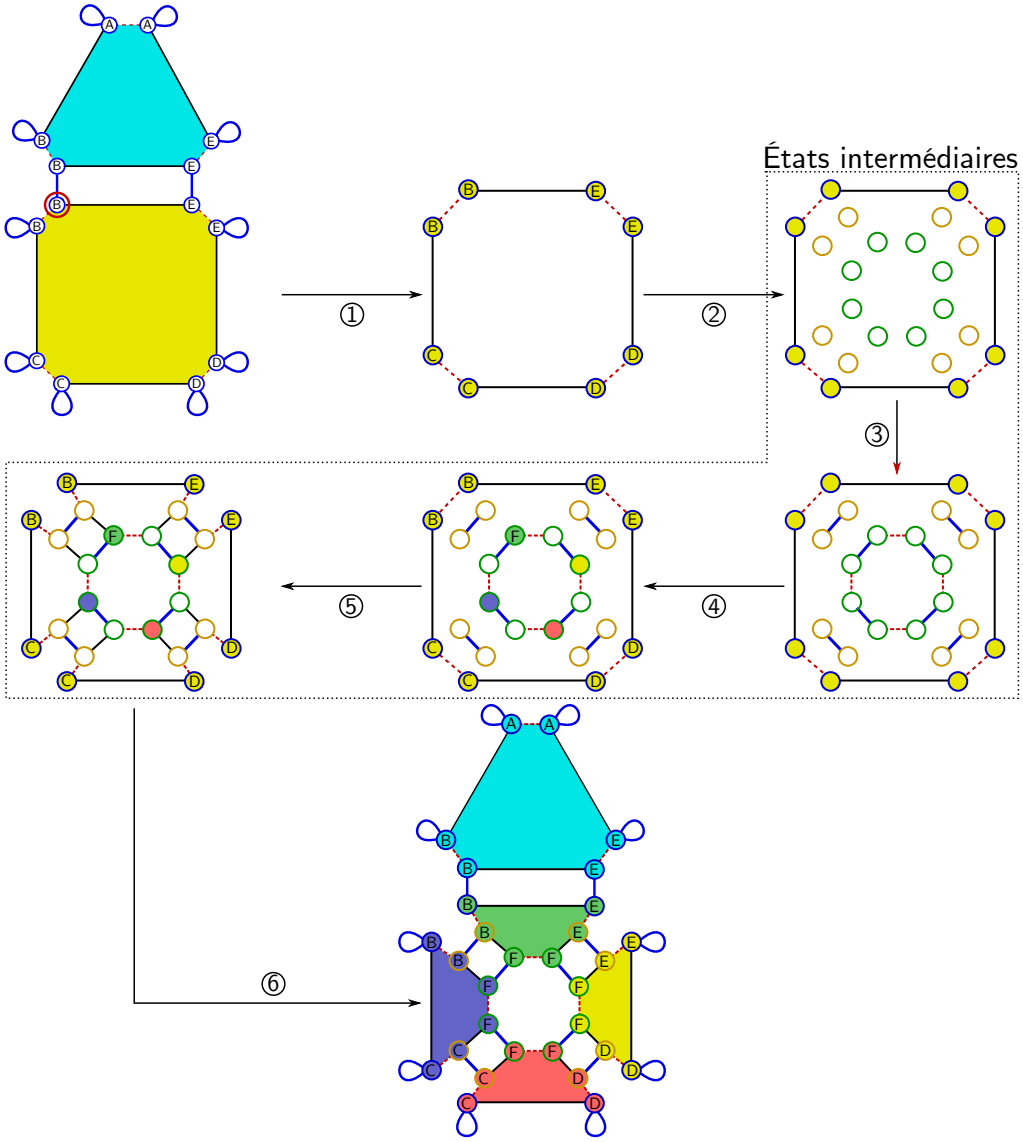


FIGURE 3.26 – Étapes du moteur d'application des règles, détaillé sur l'application de la règle de triangulation au carré de la maison

Le deuxième est un moteur optimisé pour les règles de *mise à jour* des plongements. Les règles de mise à jour sont les règles qui n'effectuent aucune modification topologique, elles ont donc le même motif à gauche et à droite et ont une ou plusieurs expressions de plongement pour mettre à jour les valeurs portées. Dans ce cas, nous sommes capable d'optimiser les traitements pour éviter la construction du filtrage gauche pour le faire à la volée et diminuer le nombre de parcours d'orbite par rapport au moteur classique.

Nous avons fait une étude comparative des deux moteurs dans la section 4.3, page 97. Les premiers résultats étant encourageants, nous souhaitons prochainement étoffer le nombre de moteurs d'applications spécifiques.

3.5 Gestion mémoire

La version Java de Jerboa bénéficie de la puissance de son langage pour la gestion de la mémoire qui est déléguée au glaneur de cellule (ou *garbage collector*) de la machine virtuelle Java. Lorsqu'un objet n'est plus utilisé par l'application, il l'identifie et le détruit pour libérer son espace mémoire automatiquement sans que l'utilisateur ou le programmeur n'aient à s'en soucier. Cette gestion automatique est très pratique car à chaque application d'une opération qui modifie les plongements des objets, de nouvelles valeurs de plongement viennent remplacer les anciennes qui seront automatiquement détruites plus tard par le glaneur de cellule.

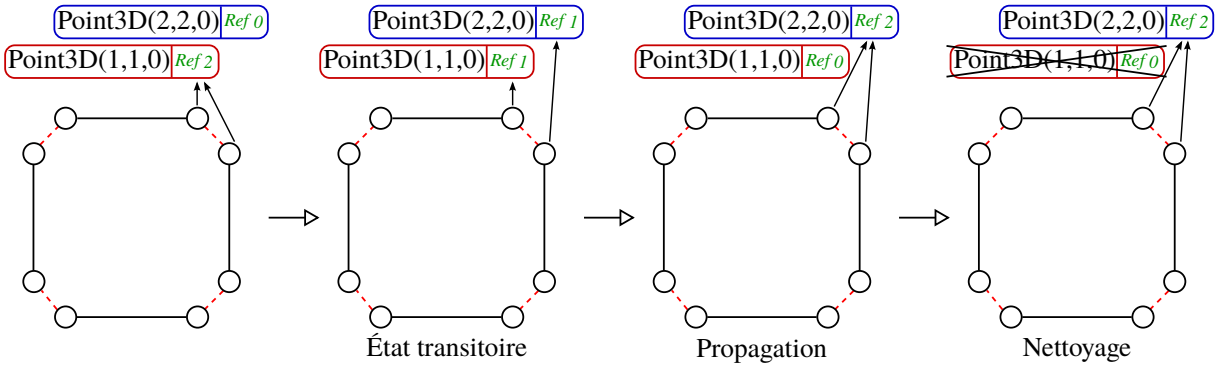


FIGURE 3.27 – Gestion mémoire des plongements

En revanche, la version C++ du noyau de Jerboa doit déterminer à chaque application de règle si des plongements doivent être supprimés ou non. Pour cela, une méthode à été mise en place afin de savoir si un plongement est utilisé ou non. Les plongements présentent tous un compteur individuel permettant de savoir combien de brins les utilisent. La mise à jour de ce compteur est réalisée dans le moteur d'application des règles, comme illustré en figure 3.27. Lorsqu'une règle crée un nouveau plongement (ici le plongement `Point3D` encadré en bleu), elle l'ajoute à la liste de tous les plongements existants. Ce plongement est ensuite assigné au brin auquel la règle l'a associé (état transitoire de la figure), en incrémentant le compteur de référence sur le nouveau plongement (en bleu) et décrémentant celui de l'ancien (en rouge). Ensuite l'application de la règle propage le plongement sur l'orbite de plongement, et met à jour les références des plongements concernés. Une fois tous les nouveaux plongements ajoutés et propagés, le moteur réalise une phase de nettoyage des plongements afin de supprimer tous ceux qui ne sont plus utilisés. Cette méthode permet donc de garder une gestion de la mémoire transparente pour l'utilisateur afin qu'il n'ait à aucun moment la contrainte de la gestion de la mémoire des plongements.

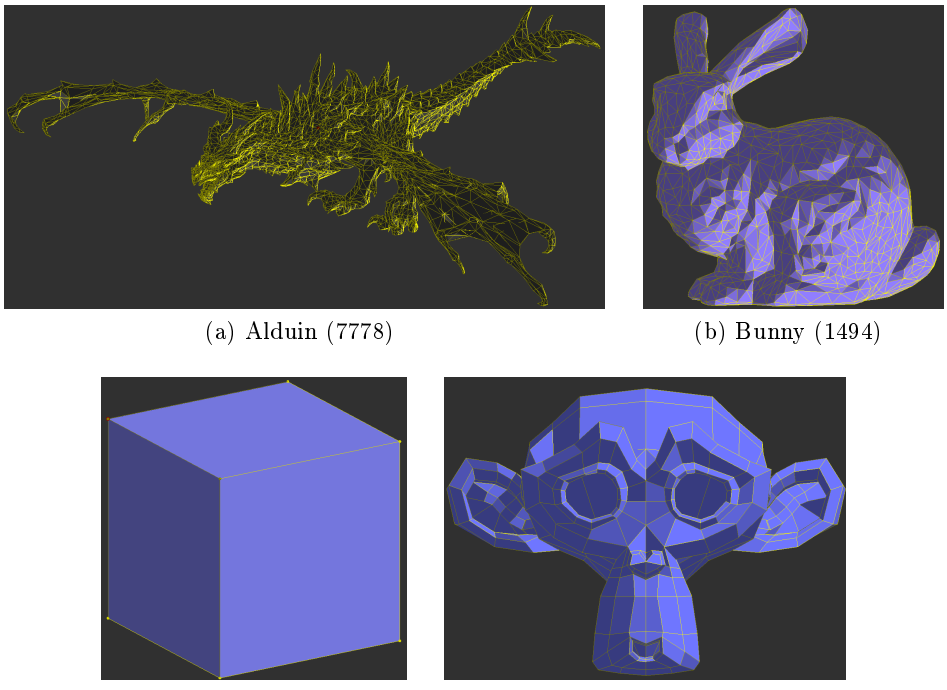
Notre méthode impose cependant que toutes les classes des plongements utilisés avec le moteur C++, soient des classes filles de la classe `JerboaEmbedding` dans laquelle est stockée le compteur d'utilisation. Cela n'empêche cependant pas l'utilisation de bibliothèques pré-existantes pour les types de plongement ou leur manipulation à condition de faire un pont entre les classes externes et une classe qui hérite de `JerboaEmbedding`. Nous pensons prochainement faire évoluer cette gestion mémoire pour utiliser les *smart pointers* qui ont été introduits dans la norme C++11. Cette évolution permettrait de simplifier l'utilisation de bibliothèques externes pour les types de plongement, mais cela nécessite quelques modifications de la traduction des expressions de plongement qui devront retourner exclusivement des pointeurs intelligents. Ce dernier point fera l'objet de nouvelles vérifications automatiques.

Chapitre 4

Résultats

Ce chapitre propose un ensemble d'expérimentations faites sur le nouveau moteur C++, pour tester sa fiabilité et présenter ses performances. Une première étude présente des schémas de subdivision, qui sont très faciles à écrire en Jerboa. Puis, nous comparons des opérations plus classiques de la modélisation. Enfin, nous présentons les résultats des aspects avancés de nos travaux à savoir une étude sur l'application de règles et des règles composées comme décrit dans la section 3.3, page 79, ainsi que l'utilisation de différents moteurs d'application.

4.1 Étude de performance Java/C++



(a) Alduin (7778) (b) Bunny (1494)

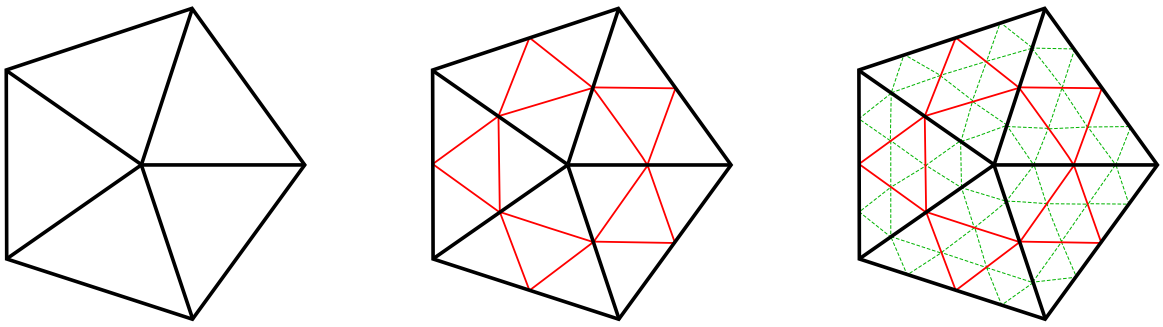
(c) Cube (8) (d) Suzanne (441)

FIGURE 4.1 – Modèles utilisés pour les tests de performance (et taille en nombre de sommets)

Cette section présente une étude de performance sur des opérations variées (cf. [GBA15]). L'expérimentation a été réalisée sur un Intel® Xeon® CPU E5-2630v3 avec 32Go de mémoire vive, OpenJDK 7, et les temps correspondent uniquement à ceux de l'application des opérations sans les temps de chargement ou de préparation des données. Nous avons utilisé la version 4.5.2 de CGAL. Dans les tables, la version C++ de Jerboa est appelée *Jerboa++* et la version Java *JJerboa*. La figure 4.1 présente l'ensemble des modèles utilisés pour nos expérimentations. Les maillages sont variés en terme d'origine (jeux vidéos, littérature, exemple type), en terme de type de mailles (triangulaires et quadrangulaires) et en terme du nombre de sommets, qui varie d'une dizaine à une dizaine de milliers. Nous avons aussi utilisé un modèle nommé *mini cubes*, qui s'obtient à partir d'un cube auquel nous avons appliqué un certain nombre de subdivisions en 8 petits cubes afin de grossir rapidement le nombre de brins dans nos tests.

Notons que les tables présentent certaines cases vides, qui peuvent être dues à des temps de calculs trop importants, ou à un outil qui n'arrive pas à réaliser le calcul après un certain nombre d'itérations.

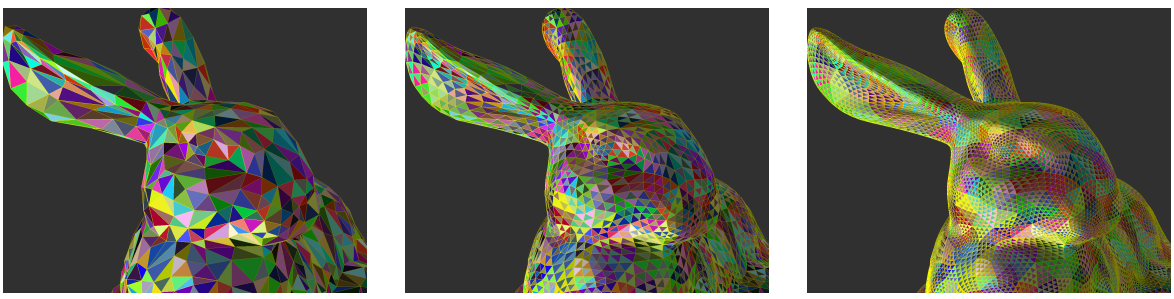
4.1.1 Subdivision de Loop



(a) Schéma de Loop maillage initial (b) 1re application (c) 2eme application

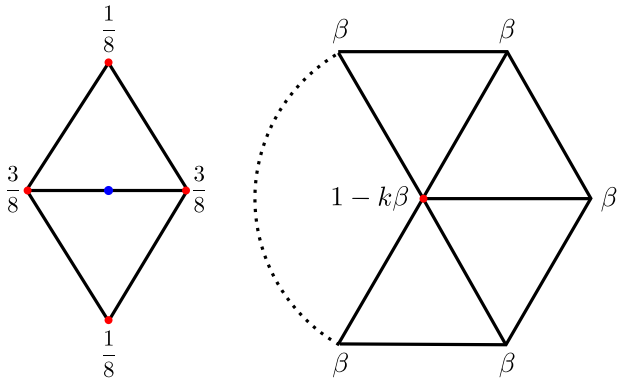
FIGURE 4.2 – Application successive du schéma de Loop

La subdivision de Loop [Loo87] est un schéma de subdivision qui s'applique sur des surfaces triangulées. Le principe est de créer un nouveau sommet au milieu de chaque arête, et de relier ces nouveaux sommets entre eux pour découper la face originelle en 4 triangles, comme le montre la figure 4.2.



(a) Bunny original (b) 1re application (c) 2eme application

FIGURE 4.3 – Application successive du schéma de Loop



(a) Nouveaux points (b) Anciens points

FIGURE 4.4 – Masques de calcul des positions des points

Cette méthode de subdivision est utilisée pour lisser des objets 3D comme le montre la figure 4.3 sur le Bunny. Le lissage repose sur un calcul qui assure asymptotiquement une continuité de la surface. La position des points existants et des nouveaux points n'est pas un simple calcul de milieu, mais un barycentre de son voisinage avec des coefficients particuliers [Loo87]. La figure 4.4 présente les coefficients communément utilisés en fonction du voisinage pour un nouveau point et un point existant.

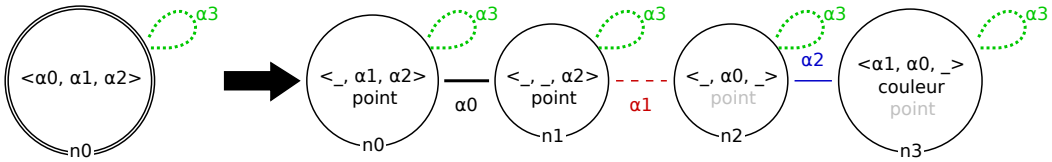


FIGURE 4.5 – Règle pour la subdivision de Loop

La figure 4.5 présente la règle Jerboa de la subdivision de Loop lissée. Le membre gauche permet de filtrer un volume isolé et le droit de créer la subdivision. Le calcul des positions se fait grâce à différents coefficients, le nœud `n0` représente les sommets existants et les nœuds `n1`, `n2` et `n3` représentent les nouveaux sommets. Pour l'expression du plongement de couleur nous avons choisi le code suivant :

```
return @ebd<couleur>::mix(L0@2.couleur,
                          L0@1@2.couleur,
                          L0@0@1@2.couleur);
```

Cette expression permet de donner à la nouvelle face centrale la couleur moyenne des trois faces voisines de la face initiale dont elle est issue. Les autres faces gardent la couleur de la face d'origine. Notons que dans l'éditeur, les vérifications de préservation de cohérence lèvent une alerte. En effet, si la face initiale n'est pas triangulaire, cette expression est erronées. Si on applique le principe d'équivalence à orbite près que nous avons vu en section 1.3.4, page 34, nous trouvons une erreur car les vérifications sont faites pour assurer la cohérence dans tous les cas (ici pour tout type de face, même non triangulaire). Dans les faits cependant, même si une expression est erronée et ne donne pas le même résultat pour tous les brins d'une même orbite de plongement, le moteur d'application ne calcule l'expression qu'une seule fois, sur le premier

brin de l'orbite qu'il parcourt. Le nouveau plongement est ensuite propagé à toute l'orbite de plongement, garantissant ainsi tout de même la cohérence des objets. Mais dans ce dernier cas, la valeur de plongement effectivement calculée est difficilement prévisible puisqu'elle dépend de l'ordre de parcours des brins.

	iteration	Jerboa++	JJerboa	CGAL (Polyhedron)	CGOGN
cube	6	233,36	808,44	11,20	16,74
	8	3914,02	9716,39	181,80	264,17
bunny	2	299,07	1123,50	21,40	17,21
	4	5699,56	26720,16	905,40	257,17
suzanne	2	56,82	394,18	4,40	4,00
	4	1080,76	3101,99	85,00	75,46
alduin	2	1873,49	5613,00	133,60	88,82
	4	31012,00	116783,57	11274,40	1832,64

TABLE 4.1 – Shema de subdivision de Loop (temps en milliseconde)

La table 4.1 présente les différents résultats que nous avons obtenus. Comme nous l'avons dit dans la section section 1.2, page 10, CGAL et CGOGN sont très efficaces, car chaque opération a été optimisée manuellement. On peut également noter un certain gain avec la version C++ par rapport à la version Java.

4.1.2 Éponge de Menger

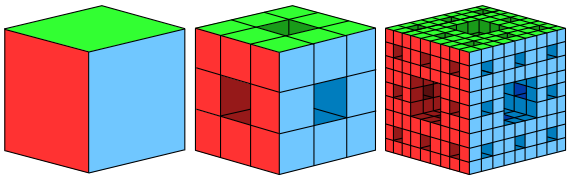


FIGURE 4.6 – Éponge de Menger

L'éponge de Menger est une opération qui subdivise un volume donné en plusieurs petits volumes et crée des tunnels. Comme le montre la figure 4.6, un cube est subdivisé en 27 petits cubes, où on retire les 6 cubes au centre des faces d'origine et le cube au centre du volume. À la fin de l'opération, il reste donc 20 petits cubes. Cette opération peut être répétée alors sur l'ensemble des nouveaux volumes, augmentant rapidement le nombre de cubes.

La règle Jerboa associée à cette opération repose sur les régularités de l'opération pour chaque brin et liaison existante. Ainsi, la règle consiste en la création des nouveaux brins et des liaisons entre eux afin de former les nouveaux petits cubes comme illustrée sur la figure 4.7. C'est une règle assez complexe topologiquement, mais dont les calculs de plongements sont simples : chaque arête est coupée en trois parties égales pour créer les nouveaux cubes.

La table 4.2, montre que la version C++ a permis à Jerboa d'augmenter sa capacité à monter en charge. En effet, pour la cinquième itération, la version Java de Jerboa n'a pas réussi à appliquer l'opération. On peut également observer sur cette opération que pour les premières itérations les Linear Cell Complex de CGAL sont plus rapides que Jerboa. Cependant, la quatrième itération a été plus rapidement calculée avec Jerboa, et la machine n'a pas réussi à calculer la cinquième itération avec les LCC au moment des tests.

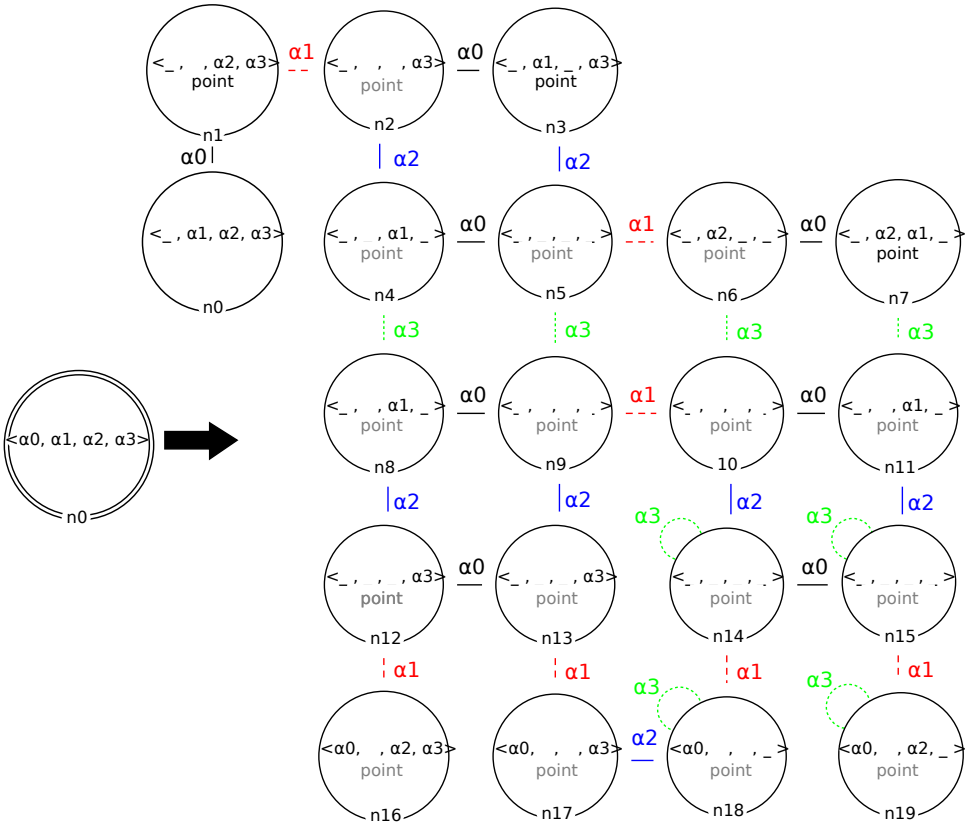


FIGURE 4.7 – Règle de l'éponge de Menger

	iteration	Jerboa++	JJerboa	LCC
cube	1	1,39	20,09	,69
	2	18,16	127,44	12,17
	3	293,96	794,61	235,21
	4	5858,47	8780,03	42368,99
	5	146126,40		

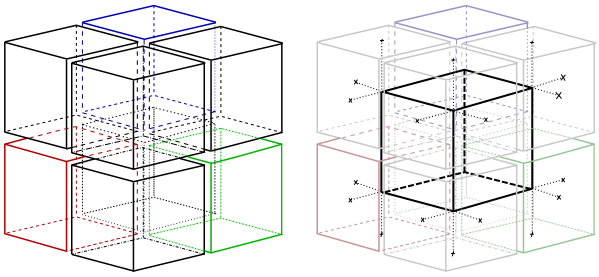
TABLE 4.2 – Éponge de Menger (temps en milliseconde)

4.1.3 Dual 3D

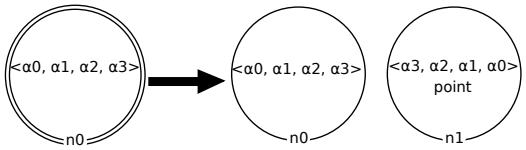
L'opération de dual 3D transforme un maillage de manière à ce que chaque volume (respectivement chaque sommet) du maillage initial devienne un sommet (respectivement un volume) et que chaque face commune à deux anciens volumes, devienne une arête reliant les deux nouveaux sommets créés au centre de chaque volume. La figure 4.8(a) illustre un exemple de maillage sur lequel on applique l'opération de dual 3D.

D'un point de vue transformation de graphe, cette opération crée une copie du maillage. Cette copie est un graphe ayant la même structure mais les liaisons sont renommées et les coordonnées des nouveaux sommets calculées.

La figure 4.8(b) représente la règle du dual 3D, le membre gauche filtre une composante connexe qui est conservée dans le membre droit. Le nœud `n1` du membre droit représente le



(a) Objet et son dual



(b) Règle Jerboa

FIGURE 4.8 – Opération de dual 3D

maillage dual, où chaque liaison α_i est renommée par son opposée α_{n-i}. Cette règle a été réalisée en une quinzaine de minutes.

	iteration	Jerboa++	JJerboa	Moka
miniCubes1	1	,99	17,28	,48
miniCubes2	1	4,50	58,06	2,26
miniCubes3	1	31,62	222,08	17,34
miniCubes4	1	318,75	706,33	168,38
miniCubes5	1	2898,50	4902,48	1368,84

TABLE 4.3 – Dual3D (temps en milliseconde)

Les résultats obtenus pour cette opération sont illustrés dans la table 4.3. Nous observons que la version C++ de Jerboa met deux fois plus de temps que Moka pour réaliser les calculs, mais est presque deux fois plus rapide que la version Java de Jerboa. La rapidité de Moka est certainement due au fait que l'opération a été optimisée manuellement, alors que dans notre outil, le code des opérations a été générées automatiquement.

4.2 Étude de performance des compositions de règle

Il serait possible d'utiliser la composition des règles à des fins d'optimisation. En effet, la composition de deux règles comme nous l'avons vu dans la section 3.3, page 79, pourrait également être utilisée pour réaliser certaines optimisations. En effet, l'application successive de plusieurs opérations implique que chacune doit réaliser son étape de filtrage. En revanche, l'opération composée réalise un seul et unique filtrage et applique directement toutes les transformations en une seule fois. Cette section présente une expérimentation de la composition de notre outils pour créer des opérations avec un temps d'exécution plus rapides. Pour cela, nous allons comparé des opérations classiques avec le résultat de leur composition avec d'autres opérations. Pour rappel, cette étude ne fera qu'une comparaison sur les temps d'application de la topologie, qui pour

l'instant est le résultat de notre composition. Il n'y a donc aucun calcul de plongement pris en compte dans cette étude.

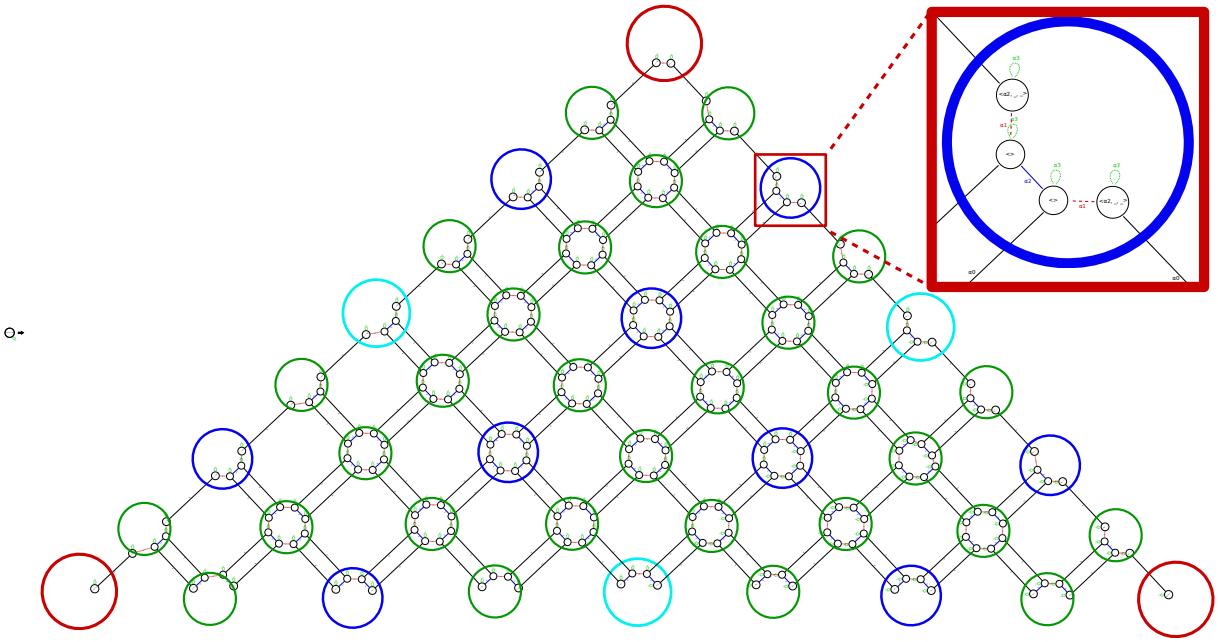


FIGURE 4.9 – Règle `CatmullX3`, issue de la composition successive de 3 schémas de Catmull-Clark

Pour commencer, nous vous proposons de composer une règle de subdivision avec elle-même. Nous choisissons les schémas de Catmull-Clark et de Loop.

La figure 4.9 montre la composition de trois règles Catmull-Clark avec elles-mêmes. On obtient ainsi une règle appelée `CatmullX3` qui réalise trois applications de la subdivision en une seule fois. Nous avons ré-organisé les nœuds dans l'éditeur afin de mettre en avant le motif créé par la subdivision. Nous avons également entouré les différents sommets créés. Les sommets cyans sont créés par la première règle de la composition, les bleus par la deuxième et les verts par la troisième.

Temps (en ms)				
Iteration	Catmull	CatmullX2	CatmullX3	CatmullX4
2	1,25	0,9		
3	**5,14**		**2,94**	
4	20,8	14,51		11,25
6	327,12	224,14	198,68	
8	5206,45	3627,22		2881,99
9	21057,62		12591,99	

TABLE 4.4 – Temps d'application (en milli-secondes) des compositions du schéma de subdivision de Catmull-Clark sur un cube

La table 4.4 montre les résultats obtenus (en milli-secondes) pour l'application de différentes compositions de notre schéma selon le nombre de fois où nous avons réalisé la composition (indiqué en suffixe du nom). Les résultats que nous représentons ici, sont des moyennes de plusieurs applications. La colonne de gauche spécifie le nombre d'itérations du schéma classique.

Ainsi, la première application de `CatmullX3` donne des chiffres pour les itérations 3, 6 et 9. On peut observer que l'application d'une règle composée donne effectivement un gain en temps. Pour trois itérations par exemple, les applications successives de la règle initiale mettent 5.14ms, alors qu'une application de `CatmullX3` met 2.94ms, soit un peu plus d'un tiers de temps en moins. Ce gain est du au fait que les opérations composée n'effectuent qu'un seul filtrage de l'objet, contrairement à la règle non composée qui est appliquée plusieurs fois. De plus, les règles composées appliquent directement les transformations finales de l'objet, sans créer les topologies intermédiaires créées par les itérations de la règle non composée.

Temps (en ms)				
Iteration	Loop	LoopX2	LoopX3	LoopX4
2	1,14	0,85		
3	4,74		2,82	
4	21,15	14,85		11,54
6	324,1	234,46	218,1	
8	6236,87	3761,37		3176,35
9	21699,22		13627,88	

TABLE 4.5 – Temps d'application des compositions du schéma de subdivision de Loop

Nous avons fait de même avec le schéma de Loop dont les résultats sont présentés sur la table 4.5. Les résultats que nous obtenons sont similaires au schéma de Catmull-Clark. On peut même observer que les chiffres sont très similaires, ce qui est assez logique dans le sens où seules les transformations topologiques sont effectuées ici, et que les deux règles créent le même nombre de brins (1 nœud à gauche et 4 à droite pour les deux).

Temps (ms)		
Model	TrLoopC	TrLoopS
cube	0,71	0,94
suzanne	48,24	64,62
bunny	242,04	350,91
alduin	1268,37	1767,07

TABLE 4.6 – Temps d'application sur plusieurs modèles de la triangulation suivie du schéma de loop en version composée (TrLoopC) ou par un *script* (TrLoopS)

Nous avons également voulu tester l'exemple que nous avions pris pour l'explication de la composition, à savoir la triangulation barycentrique composée avec le schéma de Loop. Ici nous voulons tester la différence d'efficacité entre l'application de la règle composée (`TrLoopC`) et d'un *script* (`TrLoopS`) dont le code est (avec `n0` un paramètre topologique du *script*) :

```
@rule<Triangulation_NoEbd>(n0#0);
@rule<Loop_NoEbd>(n0#0);
```

Les résultats de cette étude sont présentés en table 4.6. Cette étude confirme que la composition est plus rapide que l'application successive (ici via un *script*).

Nous pouvons donc conclure que l'application d'une règle composée plutôt que l'application successive de plusieurs règles permet effectivement un gain de performance. Notons cependant que

cette étude a été effectuée avec les seules transformations topologiques. Il est possible qu'avec la prise en compte des calculs de plongements, le gain soit encore plus important. En effet, l'application successive de plusieurs opérations impose de calculer les plongements intermédiaires qui ne sont pas calculés par la règle composée qui donne directement le résultat final. Une étude plus approfondie sur le sujet serait intéressante.

Dans le cas où la composition d'opérations permettrait un gain significatif avec la prise en compte des plongements, il serait possible de réaliser à la volée des optimisations dans les *scripts*. En effet, avec le langage, nous pourrions détecter que, dans le *script* ci-dessus, les deux opérations s'appliquent sur le même motif. Nous pourrions ainsi créer à la volée l'opération composée, puis remplacer l'appel de règles consécutives d'un *script* par l'appel d'une nouvelle règle composée.

4.3 Étude de performance de la spécialisation de moteur

La nouvelle architecture des noyaux de Jerboa permet de définir plusieurs moteurs d'application spécialisés pour différents types de règle, et ce dans un but d'optimisation transparente à l'utilisateur. Pour tester l'intérêt de cette méthode, nous avons implanté un moteur dédié à l'application de règle de mise à jour de plongement, qui ne font pas de transformation topologique. C'est-à-dire que le membre gauche et le droit sont identique à l'exception qu'il y a des expressions de plongement sur le nœud de droit. Sans modification topologique, plusieurs étapes du moteur d'application deviennent inutiles et l'étape de calcul des plongements devient l'élément central. Des optimisations de parcours peuvent être réalisées car les orbites supports des plongements ne sont pas modifiées.

Temps (ms)			
Moteur	TranslationConnexe	ChangeCouleurConnexe	RotationConnexe
Défaut	290	308,6	318,6
Mise à jour	97,4	80,8	115,4

TABLE 4.7 – Temps d'application de plusieurs règles de mise à jour de plongement sur un cube, avec deux moteurs d'application différents

Nous avons testé ce moteur sur trois opérations : `TranslationConnexe` qui déplace par translation une composante connexe dans l'espace 3D, `ChangeCouleurConnexe` qui attribue une couleur aléatoire à toutes les faces d'une composante connexe, et `RotationConnexe` qui applique une rotation à une composante connexe. Les résultats que nous avons obtenu sont illustrés dans la table 4.7. Ces résultats nous montrent que le moteur dédié aux règles de mise à jour est presque trois fois plus efficace que le moteur par défaut. Ces résultats sont encourageants pour l'implantation d'autres moteurs spécifiques. À l'avenir nous pourrions développer différents moteurs pour d'autres types de règles, comme des règles de création ou de modification uniquement topologique, *etc.*

Chapitre 5

Perspectives d'évolution

Au cours de cette thèse nous avons réalisé un certain nombre de développements, tant au niveau de Jerboa (son langage et ses outils) qu'au niveau applicatif comme nous le verrons dans la deuxième partie de ce manuscrit. Nos pistes de réflexion ont évolué au fur et à mesure que nous utilisions nos outils sur des exemples concrets. Certaines fonctionnalités ont été ajoutées récemment, mais nous n'avons malheureusement pas eu le temps d'implanter toutes les pistes que nous souhaitions explorer. Ce chapitre présente des pistes de réflexion portant sur les travaux futurs concernant Jerboa et son langage.

5.1 Évolution du langage

5.1.1 Opérateur de choix avec historique

Nous avons mis en place un langage dédié qui permet de décrire les expressions de plongement, les calculs des différents processus des règles et l'écriture de *scripts* de règles. Ce langage permet d'abstraire le langage sous-jacent des noyaux de Jerboa et de simplifier l'écriture de certaines instructions comme l'application d'une règle ou l'accès à un plongement. Grâce à ce langage de haut niveau, nous avons mis en place des transformations permettant d'interpréter les éléments en fonction d'un contexte particulier comme nous l'avons vu en section 3.2.2, page 73 avec l'accès aux plongements. Cependant il est possible de créer de nouvelles interprétations et de réaliser des optimisations.

Prenons l'exemple de l'opérateur de choix. Il permet de spécifier une liste de règles que l'on souhaite appliquer selon les cas, à un objet. Voici un exemple avec un brin `b` :

```
@rule<regle0>(b)
    | @rule<regle1>(b)
    | @rule<regle2>(b)
    | @rule<regle3>(b);
```

Ici chacune des quatre règles peut potentiellement s'appliquer. La règle `regle1` s'applique si `regle0` a échoué, *etc.* Lors de l'application de la règle `regle0`, le motif de filtrage est calculé en fonction de son graphe gauche, de même que pour les suivantes. Or il est fortement probable que lors de l'utilisation de l'opérateur de choix, les membres gauches, tout ou partie, des règles soient très similaires car cet opérateur a été créé pour lister les règles qui peuvent s'appliquer sur un même objet (même si il est possible d'utiliser cet opérateur différemment). Ainsi, il est possible que les règles de la liste aient des graphes gauches identiques (mais des pré-conditions et transformations différentes). Dans ce cas, si les règles s'appliquent aux mêmes brins accrochés

(ici le brin `b`), le même motif sera potentiellement calculé plusieurs fois. Il est possible d'optimiser cette partie et réaliser un filtrage unique pour les règles qui possèdent un graphe gauche identique. Précisément, la première règle calcule le motif filtré, et les règles suivantes qui ont le même membre gauche, utilisent ce motif filtré sans le re-calculer. L'éditeur permet de détecter statiquement si plusieurs règles ont des motifs filtrés identiques, c'est à dire des graphes gauches identiques accrochés sur les mêmes brins. Cette optimisation est généralisable au cas où certaines règles de la liste filtrent le même motif mais pas toutes. Par exemple, si `regle0` et `regle3` filtrent le même motif d'une part, et que `regle1` et `regle2` filtrent un autre motif identique d'autre part, il est possible de filtrer chacun des deux motifs une seule fois.

L'utilisation d'un système de cache permettrait de mettre en œuvre cette optimisation. Chaque motif filtré commun à plusieurs règles est stocké dans une variable dédiée qui est réutilisée par les autres règles partageant ce même motif. Dans le noyau de Jerboa, le processus de filtrage du membre gauche et l'application des transformations du membre droit sont deux processus bien distincts. Le pseudo-code suivant montre une solution, pour le code généré, qui exploite cette distinction des deux processus pour factoriser les motifs filtrés de l'exemple précédent :

```
variable filtreGauche_regle0_regle3, filtreGauche_regle1_regle2;

filtreGauche_regle0_regle3 = regle0.filtre(b);
try{
  regle0.appliqueRegle(filtreGauche_regle0_regle3);
}catch(echecApplicationRegle){
  filtreGauche_regle1_regle2 = regle1.filtre(b);
  try{
    regle1.appliqueRegle(filtreGauche_regle1_regle2);
  }catch(echecApplicationRegle){
    try{
      regle2.appliqueRegle(filtreGauche_regle1_regle2);
    }catch(echecApplicationRegle){
      try{
        regle3.appliqueRegle(filtreGauche_regle0_regle3);
      }catch(echecApplicationRegle){
        // aucune règle ne s'applique
      }
    }
  }
}
```

Dans cet exemple, la fonction `appliqueRegle` est la fonction d'application des règles qui prend en paramètre le motif filtré de la règle pour y appliquer directement la transformation spécifiée par le membre droit. Notons que les motifs filtrés dépendent non seulement des motifs de filtrage mais aussi des brins accrochés. Reprenons l'exemple précédent avec les mêmes règles mais avec des brins `b1` et `b2` différents :

```
@rule<regle0>(b1)
  | @rule<regle1>(b2)
  | @rule<regle2>(b2)
  | @rule<regle3>(b2);
```

Avec ces modifications, l'optimisation du calcul des filtres gauche n'est réalisée que pour les règles `regle1` et `regle2`, car même si les règles `regle0` et `regle3` ont des graphes gauches identiques, le brin accroché qui leur est fourni à l'application est différent. Les motifs filtrés lors de l'application sont donc différents.

Dans l'exemple de départ, les filtres communs `filtreGauche_regle0_regle3` et `filtreGauche_regle1_regle2` sont calculés une seule fois. Le filtrage peut cependant échouer. Ainsi il convient d'aller plus loin et de produire un résultat qui n'essaie pas d'appliquer les règles dont le calcul du filtrage commun a échoué. Si le calcul du filtrage de la règle `regle0` échoue, les règles qui filtrent le même motif (ici `regle3`) ne peuvent pas s'appliquer non plus. On peut donc parfaire le pseudo code précédent comme suit :

```
variable filtreGauche_regle0_regle3, filtreGauche_regle1_regle2;
try{
  filtreGauche_regle0_regle3 = regle0.filtre(b);
  try{
    regle0.appliqueRegle(filtreGauche_regle0_regle3);
  }catch(echecApplicationRegle){
    try{
      filtreGauche_regle1_regle2 = regle1.filtre(b);
      try{
        regle1.appliqueRegle(filtreGauche_regle1_regle2);
      }catch(echecApplicationRegle){
        try{
          regle2.appliqueRegle(filtreGauche_regle1_regle2);
        }catch(echecApplicationRegle){}
      }
    }catch(echecFiltrage){ // le deuxième filtrage a échoué
      try{
        regle3.appliqueRegle(filtreGauche_regle0_regle3);
      }catch(echecApplicationRegle){}
    }
  }
}catch(echecFiltrage){ // le premier filtrage a échoué
  try{
    filtreGauche_regle1_regle2 = regle1.filtre(b);
    try{
      regle1.appliqueRegle(filtreGauche_regle1_regle2);
    }catch(echecApplicationRegle){ // le deuxième filtrage a échoué
      try{
        regle2.appliqueRegle(filtreGauche_regle1_regle2);
      }catch(echecApplicationRegle){}
    }
  }catch(echecFiltrage){} // tous les filtrages ont échoués
}
```

Avec l'utilisation de l'opérateur de choix, lors de l'échec de l'application des règles il n'y a aucun moyen de connaître la raison de ces échecs. Les raisons des échecs peuvent être utiles dans certains cas, pour réagir en conséquence, ou pour réaliser un système de logs par exemple. En ce sens nous avons imaginé un opérateur dédié permettant de retourner en plus du retour de règle, les exceptions levées lors des échecs d'application des règles. Cet opérateur est noté || et s'utilise comme l'opérateur de choix classique :

```
JerboaChoiceResult resultat =  @rule<regle0>(b)
                            || @rule<regle1>(b)
                            || @rule<regle2>(b)
                            || @rule<regle3>(b);
```

Ici le type de retour `JerboaChoiceResult` est une structure contenant la liste des exceptions générées par les échecs des applications des règles, ainsi que le potentiel retour de règle si l'une d'elle s'est appliquée. Le résultat de l'application de règle est décrit avec une notation pointée : `resultat.ruleResult`. La liste des différentes exceptions est décrite de façon similaire : `resultat.exceptions`. Ces exceptions permettent de connaître les règles qui les ont générées ainsi qu'un message ou un type permettant de les identifier :

```
for(JerboaChoiceException e : resultat.exceptions){
  @print("Exception : ", e.type, " -", e.message,
      "- from ", e.rule);
}
```

Si l'application de la règle `regle0` du listing précédent échoue lors du calcul du filtrage, cet exemple affiche un message tel que : `Exception : LeftMatchingException - error while computing left Pattern - from rule0`.

5.1.2 Lambda expressions

Le langage de Jerboa permet de définir des calculs et des *scripts*. Les possibilités du langage peuvent être étoffées par de nouvelles fonctionnalités comme les lambda-expressions, qui sont inclues dans les langages sous-jacents depuis les normes C++11 et Java 8.

Prenons l'exemple de l'opération `map`. Cette opération applique à tous les éléments d'une collection une même opération. `Map` pourrait permettre de simplifier certaines expressions comme dans l'expression suivante, avec `@map` l'opérateur `map` :

```
// Application de la règle de découture en a3 pour toutes les
// faces du volume incident au brin "volume"
JerboaList<JerboaRuleResult> resultats =
  @map(@rule<DecoutureFaceAlpha3>(), <0,1,2>_<0,1>(volume));

// Calcul de la liste de toutes les faces voisines par
// alpha 3 du volume incident au brin "volume"
JerboaList<JerboaDart> voisinDeFace =
  @map(getVoisinA3(), <0,1,2>_<0,1>(volume));
// Ou encore pour la même opération :
JerboaList<JerboaDart> voisinDeFace =
  @map(@3, <0,1,2>_<0,1>(volume));
```

D'autres opérations similaires de réduction, de compressions ou d'accumulation (également appelées *fold*) pourraient également être intéressantes à ajouter. Avec ce type d'opération, il est possible de réaliser la somme des éléments d'une liste, leur moyenne, *etc.*

Notons que habituellement *fold* sur une liste permet de faire une réduction gauche ou une réduction droite (soit faire l'opération en partant du début de la liste soit en partant de la fin). La figure 5.1 illustre la différence entre les deux sur un exemple d'indication de direction d'un chemin sur une route. Si une personne demande son chemin à un tiers pour attendre une destination, le fait de prendre les indications à rebours ne le mènera pas nécessairement à la bonne destination comme l'illustre la figure. Ici le chemin indiqué est : tourner à droit, tourner deux fois à gauche, aller tout droit, tourner à droite puis aller tout droit. Selon le sens dans lequel sont prises les indications, le point d'arrivée est différent.

Les collectes dans Jerboa donnent des éléments sous forme de collections sans ordre et non de listes. Il n'est donc pas souhaitable de se servir de l'ordre des éléments dans celle-ci. Dans

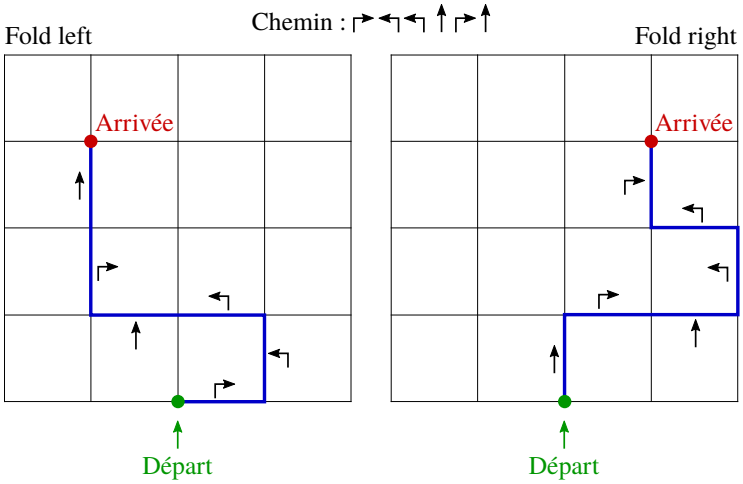


FIGURE 5.1 – Illustration de la différence entre un *fold left* et un *fold right* sur un exemple d'indication de direction sur une route

ce sens seul un type de *fold* est acceptable, et seules les lambda expressions indépendantes de l'ordre devraient être utilisées.

5.1.3 Filtrage des *Scripts*

Les *scripts* de règles permettent de réaliser des opérations sur plusieurs orbites non isomorphes contrairement aux règles classiques. Nous avons vu dans les sections précédentes que les paramètres topologiques des *scripts* étaient fournis à l'application par des listes de brins accrochés, dont chacune est représentée par un nœud dans le graphe gauche des *scripts*. Pour permettre de fournir plusieurs éléments pour un même nœud du graphe gauche, nous avons mis en place un système de multiplicité, qui spécifie le nombre de représentant possible pour chaque nœud. Cependant, ces représentants sont fournis sous forme de simples brins. Contrairement aux règles classiques, il n'y a pas de processus de filtrage lors de l'application des *scripts*. Ainsi, les *scripts* appliquent les règles à partir des brins passés en paramètre. Cette solution n'est pas homogène avec les règles classiques, et ne permet pas de se servir du filtrage pour déterminer si un *script* peut s'appliquer ou non. De plus, il n'est pas possible avec le système actuel d'assurer que deux nœuds auront effectivement le même nombre de représentants à l'application du *script*.

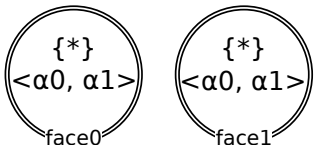


FIGURE 5.2 – Graphe gauche d'un *script* de collage de volume par des faces

Prenons l'exemple d'un *script* qui prend en paramètre deux listes de faces pour les coudre deux à deux par α_3 en appelant la règle adéquate. Le graphe gauche de ce *script* contient deux nœuds comme illustré en figure 5.2, et le code du *script* pourrait être le suivant :

```
for(int i=0;i<face0.size();i++){
    @rule<CoutureA3>(face0#i, face1#i);
}
```

Les *scripts* n'ayant pas de processus de filtrage, les orbites présentes dans les nœuds sont présentes uniquement pour une cohérence sémantique pour l'utilisateur mais ne sont pas prises en compte par le moteur d'application. Ceci implique qu'il n'est pas possible de déterminer avant d'appliquer le *script*, si `face0` et `face1` contiennent le même nombre de représentants. De plus, rien n'assure que les faces à coudre sont isomorphes deux à deux. Ces vérifications ne peuvent être faites en amont que si les *scripts* appliquent le processus de filtrage avant de réaliser les transformations.

Avec la multiplicité définie sur les nœuds, ces vérifications ne sont pas réalisables. Nous avons imaginé une autre solution plus homogène avec les règles qui répond aux besoins que nous venons d'énoncer. Le principe est de définir le motif gauche des *scripts* non pas par un graphe contenant des nœuds ayant chacun leur multiplicité, mais par plusieurs graphes définis de façon identique aux règles classiques. Chaque graphe a une multiplicité, afin de pouvoir filtrer plusieurs motifs identiques.

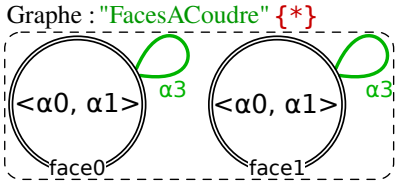


FIGURE 5.3 – Graphe gauche d'un *script* de collage de faces (nouvelle version)

Reprenons l'exemple précédent en appliquant ce nouveau concept. Le graphe gauche du *script* est a présent nommé (ici `FacesACoudre`) et présente une multiplicité $\{*\}$, comme illustré en figure 5.3. Le code de ce *script* est également modifié en conséquence :

```
// itération sur les différents filtrages du graphe "FacesACoudre"
for(int i=0;i<FacesACoudre.size();i++){
    @rule<CoutureA3>(FacesACoudre#i);
}
```

Dans ce *script*, `FacesACoudre` est une variable accessible depuis le *script* et représente un graphe du membre gauche du *script*. Elle contient la liste de tous les motifs filtrés représentés par ce graphe (ici toutes les paires de faces). L'expression `FacesACoudre#i` représente le i^{eme} motif filtré. Il peut être passé directement en paramètre de l'application de la règle de couture car cette dernière porte un graphe gauche identique au graphe `FacesACoudre`. La règle `CoutureA3` ne re-calcule alors pas le motif filtré, mais transforme directement celui que le *script* lui fournit.

Notons que cette notation peut faire l'objet d'une nouvelle vérification automatique par l'éditeur. En effet, la cohérence du motif filtré par le script avec le membre gauche de la règle appelée peut être vérifiée statiquement. Précisément, cette notation est possible si le graphe représenté par la variable (ici `FacesACoudre`) est rigoureusement identique à celui de la règle appelée (ici `CoutureA3`).

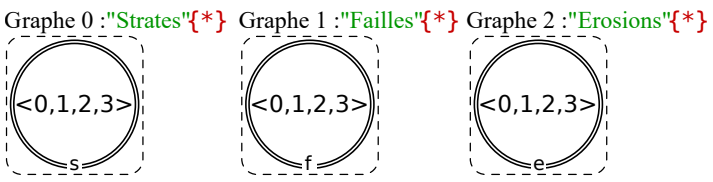


FIGURE 5.4 – Graphes du membre gauche d'un *script* nommé `ScriptGeologique`, procédant à des transformations géologiques

Avec la nouvelle définition du membre gauche des *scripts*, le passage des paramètres topologiques pour l'application des *scripts* doit être adapté. Comme pour les règles classiques, l'application d'un *script* doit pouvoir se faire soit sur une liste de brins pour chaque graphe du membre gauche, soit directement sur des motifs déjà filtrés.

Prenons l'exemple d'un *script* que nous nommerons `ScriptGeologique`, qui réalise une opération géologique, prenant en paramètre un nombre indéterminé de failles, d'érosions, et de couches géologiques. Ces paramètres ne sont pas nécessairement liés, et ne sont pas tous aussi nombreux les uns que les autres. Le *script* présente donc un graphe par entité comme illustré en figure 5.4. Voici un exemple de pseudo-code de ce *script* :

```
for(JerboaLeftPattern str_i: Strates){
  // opérations sur chaque motif filtré de Strates
  for(JerboaDart dei: str_#s){
    // opérations sur chacun des brins représentés par le noeud "s"
    // du motif filtré "str_i"
  }
}
for(JerboaLeftPattern fi: Failles){
    // opérations sur chaque motif filtré de Failles
}
for(int si=0;si<Erosions.size();si++){
  // opérations sur chaque motif filtré de Erosions
}
```

Nous venons de voir une possible amélioration pour le membre gauche des *scripts*. Cette amélioration peut également être appliquée au membre droit. Ainsi, le graphe droit des *scripts* permettra de représenter tout un motif d'une règle appliquée dans le *script* (ou de plusieurs), ou encore de construire un graphe droit manuellement comme c'est le cas dans la version actuelle.

5.2 Évolution de l'éditeur et des vérifications

5.2.1 Composition de règles

Nous avons vu dans les sections précédentes que nous étions en mesure de composer plusieurs règles de transformations de graphe en une seule (cf. section 3.3, page 79) grâce à l'ajout d'un outil dans notre éditeur de modeleur. Cependant l'outil fonctionne uniquement pour les transformations topologiques. Actuellement, les expressions des plongements des règles composées ne sont pas calculées automatiquement. Il est nécessaire de réaliser les calculs manuellement puis d'ajouter les expressions correspondantes dans les règles composées dont la topologie a été générée. Si le calcul de la topologie des règles de composition peut être calculée en appliquant les modifications classiques des transformations de graphe, les calculs des expressions de plongement composées ne sont pas aisés. Nous présentons ici quelques pistes de réflexion.

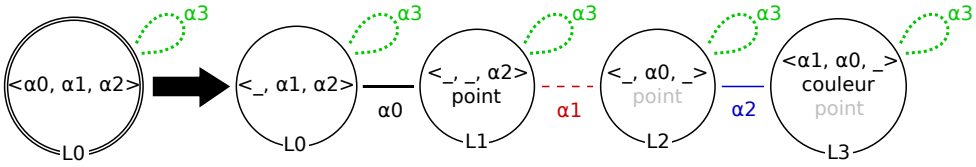


FIGURE 5.5 – Règle du schéma de subdivision de Loop

Reprenons l'exemple de la composition de la triangulation barycentrique avec le schéma de subdivision de Loop. Afin de simplifier les calculs, nous allons utiliser une version sans lissage du schéma de Loop. La règle Jerboa associée à cette opération est illustrée en figure 5.5. Sans lissage, le schéma de Loop ne modifie pas la position des sommets existants et `L0` ne porte donc pas d'expression du plongement `point`. L'expression de `L1` est également simplifiée pour calculer le milieu des arêtes existantes :

```
return @ebd<point>::barycentre(<0,2,3>_point(L0));
```

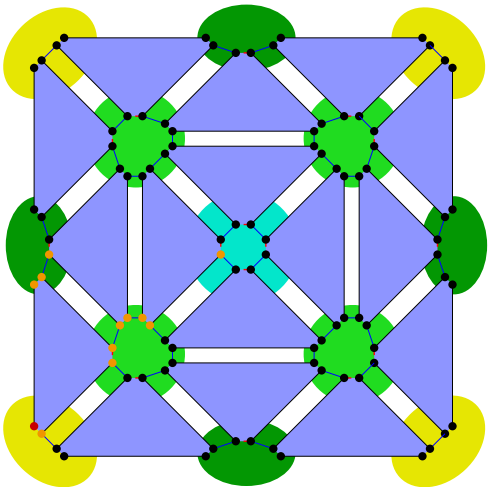


FIGURE 5.6 – Triangulation composée avec le schéma de Loop

La figure 5.6 montre l'objet résultat de l'application de composition de la triangulation barycentrique avec le schéma de Loop sur un carré. Ici les sommets initiaux (ceux du carré) sont sur fond jaune, celui créé par la triangulation sur fond cyan et ceux créés par le schéma de Loop en vert.

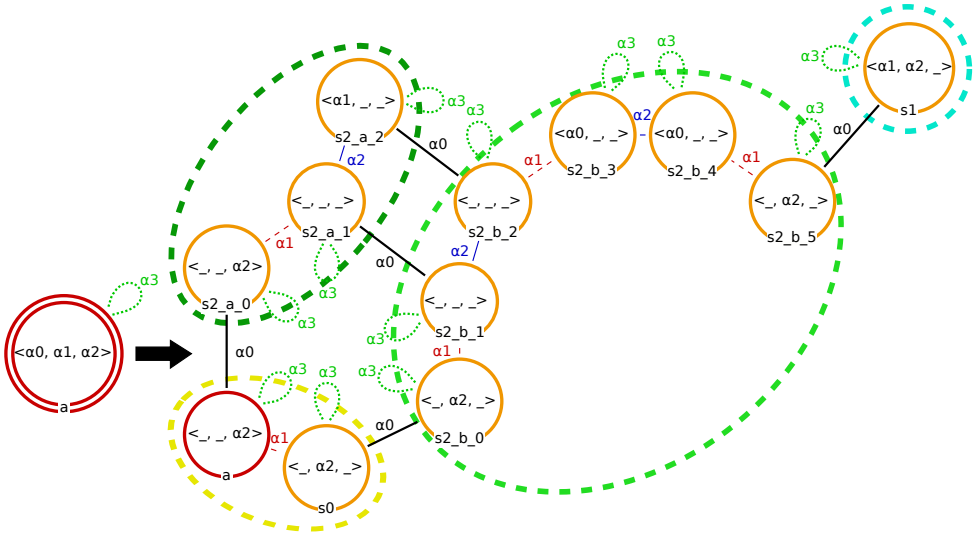


FIGURE 5.7 – Triangulation composée avec Loop

Nous avons vu dans la section 3.3, page 79, la règle de composition de ces deux opérations. Nous l'avons reprise ici avec la figure 5.7, en réorganisant graphiquement les nœuds afin de mettre en avant la forme créée par la règle. Le nœud d'accroche est en rouge, de même que le brin accroché sur l'objet de la figure 5.6 (en bas à gauche).

Nous allons compléter la règle composée avec les expressions de plongements. Nous ne nous intéresserons pour l'instant qu'au plongement `point`. La position des sommets jaunes ne sont pas modifiés. Ainsi les nœuds `a` et `s0` ne doivent pas porter d'expression du plongement `point`. En revanche, le nœud `s1` correspond au nouveau sommet cyan créé par la triangulation et doit porter une expressions de plongement. De même pour les nœuds `s2_a_0, s2_a_1, s2_a_2` et `s2_b_0, s2_b_1, s2_b_2, s2_b_3, s2_b_4, s2_b_5` qui représentent les sommets verts créés par la subdivision de Loop. L'expression du plongement cyan est celle que nous avions présenté pour la règle de triangulation barycentrique classique, à savoir le calcul du barycentre de la face :

```
return @ebd<point>::barycentre(<0,1>_point(a));
```

Rappelons qu'ici `barycentre` est une méthode statique de la classe du plongement `point` et que le type de ce plongement est donné par `@ebd<point>`.

L'application de la règle composée crée deux types de sommets verts différents. Dans la règle les nœuds sont préfixés par `s2_a` (sommet vert foncé) ou `s2_b` (sommet vert clair) selon celui qu'ils représentent. La position des sommets verts foncés est le milieu de l'arête présente sur l'objet initial (ici sur les bords) incidente au nœud `a` comme le montre la figure 5.6 sur les bords. Nous définissions cette expression comme suit :

```
return @ebd<point>::barycentre(<0,2,3>_point(a));
```

Cette expression retourne le barycentre entre les deux points de l'arête incidente à `a` (ces deux points sont obtenus par la collecte des instances du plongement `point` sur l'arête incidente à `a`).

Les sommets verts clairs représentés par les nœuds préfixés par `s2_b`, sont les milieux des arêtes créées par la triangulation, soit le milieu des arêtes incidentes au nœud a et au sommet cyan. Or le sommet cyan n'existe pas encore car la règle composée crée les sommets verts clairs et le cyan en même temps. Il faut donc le re-calculer dans cette expression, par exemple :

```
return new @ebd<point>( ( a.point +
         @ebd<point>::barycentre(<0,1>_point(a)) ) * 0.5);
```

Cette expression retourne une nouvelle instance du plongement `point` dont la position est le milieu entre le barycentre de la face incidente à `a` et la position de `a`.

La question qui se pose ici est : est-ce que ces expressions sont déductibles à partir des seules informations des règles de triangulation et du schéma de Loop.

La figure 5.8 illustre le processus de composition de la règle de triangulation barycentrique avec le schéma de Loop ainsi que la composition des plongements. Pour définir les expressions de plongement il faut commencer par déterminer sur quels nœuds elles doivent être définies.

Commençons par les expressions de la règle de triangulation. Elle est représentée en haut de la figure 5.8 dans le cadre vert. Les expressions de la première règle de la composition sont appliquées pour calculer le premier motif. Ainsi les brins présents à l'issue de l'application de la triangulation sont concernés par ces calculs de plongements. Sur notre figure, ces brins sont représentés par les nœuds de la première ligne du motif de composition (cadre bleu) : `a`, `s0` et `s1`. En effet, cette ligne correspond au motif filtré par le nœud $L0$ de la règle de Loop (cadre rouge). Ainsi le brin $s1$ doit porter une expression de plongement. Elle correspond au calcul du sommet créé par la triangulation et n'a pas de raison d'être modifiée car la règle de Loop ne modifie pas les sommets filtrés (rappelons que nous utilisons ici une version du schéma de Loop

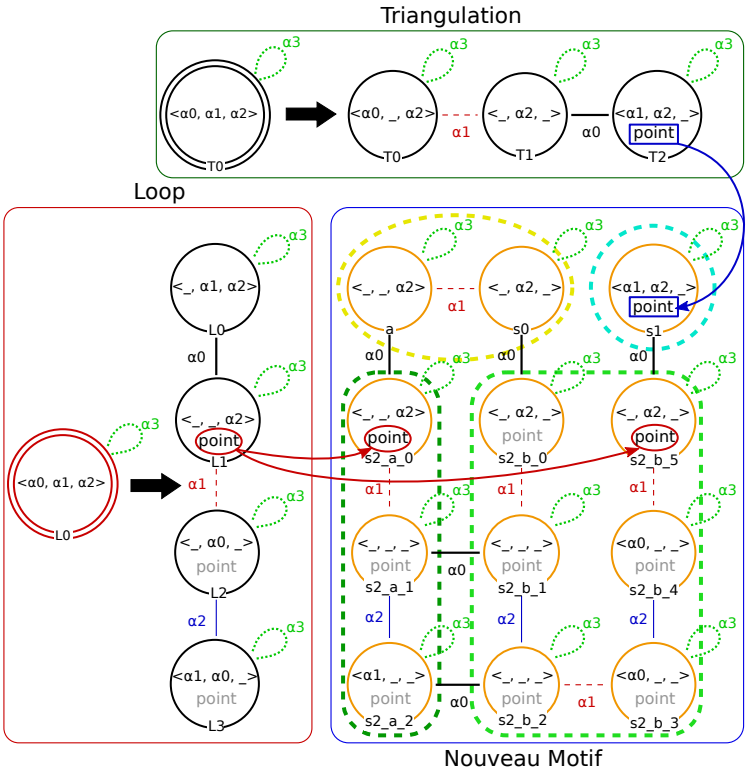


FIGURE 5.8 – Composition de la règle de triangulation barycentrique avec le schéma de Loop, avec propagation des plongements sur le nouveau motif

sans lissage). Le nœud `s1` se voit donc attribuer l'expression de plongement de `T2` du motif du cadre vert (flèche bleue entre les cadres vert et bleu) :

```
return @ebd<point>::barycentre(<0,1>_point(T0));
```

Il suffit de substituer la variable de nœud `T0` par le nom du nouveau nœud d'accroche dans la règle composée, pour obtenir l'expression attendue :

```
return @ebd<point>::barycentre(<0,1>_point(a));
```

Une fois les expressions de la première règle de composition ajoutées (triangulation), nous pouvons faire de même avec celles de la deuxième règle (Loop). Encore une fois il n'y en a qu'une seule, celle de `L1` (cadre rouge) :

```
return @ebd<point>::barycentre(<0,2,3>_point(L0));
```

Notons cependant que les nœuds `L2` et `L3` portent également cette même expression. Comme dans l'éditeur de modeleur, nous avons laissé une seule expression par orbite de plongement (ici les sommets $\langle 1, 2, 3 \rangle$) en noir, et les autres dans la même orbite sont grisées, comme indiqué sur la figure avec les flèches rouges.

Pour les nœuds représentant les sommets verts foncés représentés par `s2_a_0`, l'expression obtenue comme précédemment par simple substitution de `L0` par `a`, donne l'expression attendue :

```
return @ebd<point>::barycentre(<0,2,3>_point(a));
```

Cette expression calcule effectivement le milieu des arêtes de l'objet initial (ici au bord).

Dans le cas du sommet vert clair (représenté par `s2_b_5`), une simple substitution de variable nœud ne peut convenir. En effet, dans ce cas, `L0` ne correspond pas au nœud préservé `a`, mais au nœud créé `s1`. `L0` ne doit donc pas juste être substitué par un nouveau nœud, mais c'est l'expression complète qui doit être réécrite en fonction de la première règle de triangulation.

En particulier, la collecte :

```
<0,2,3>_point(L0)}
```

doit être réécrite pour collecter les points initialement créés par l'arête incidente à `T2`. C'est à dire les points de `T1` et `T2`. Les points de `T1` sont ceux d'origine et sont donc accessibles par l'expression :

```
a.point
```

Ceux de `T2` par la réécriture de son expression `point`, à savoir :

```
@ebd<point>::barycentre(<0,1>_point(a))
```

Le résultat de la collecte initiale peut donc être calculé comme suit :

```
JerboaList listePositions;
listePositons.add(a.point); // T1.point
listePositons.add(@ebd<point>::barycentre(<0,1>_point(a))); // T2.point
```

et l'expression attendue ainsi :

```
JerboaList listePositions;
listePositons.add(a.point);
listePositons.add(@ebd<point>::barycentre(<0,1>_point(a)));
// La variable vient remplacer la collecte dans l'appel de fonction
return @ebd<point>::barycentre(listePositions);
```

Complétons à présent la règle composée avec les expressions du plongement `couleur`, comme illustré en figure 5.9. Rappelons que la règle de triangulation barycentrique porte sur `T2` l'expression de `couleur` suivante :

```
return @ebd<couleur>::mix(T0.couleur, T0@2.couleur);
```

Appliquons les règles de propagation des expressions de plongement que nous avons décrites pour le `point` en commençant par celle de la triangulation (flèche bleue de la figure). La couleur est donc réécrite sur les nœuds `a` et `s0` avant d'être propagée sur les nœuds des mêmes faces. Le schéma de Loop coupe les faces en quatre. Trois d'entre elles sont incidentes aux sommets de la face d'origine, et la quatrième est au centre, incidente aux nouveaux sommets. Les trois faces sont représentées par les nœuds `L0`, `L1` et `L2`, et la quatrième centrale par `L3`. Ici l'application de la règle laisse la couleur de la face d'origine pour les trois faces et définit une nouvelle couleur pour la face centrale.

La règle de Loop ne modifie pas la couleur des trois faces aux coins, donc les expressions propagées par la triangulation (flèches bleues) n'ont pas à être ré-écrites, une simple substitution des variables de nœud suffit. Nous propageons ensuite l'expression de couleur de Loop (flèches rouges), sur le nœud `s2_b_3` qui représente les futures faces centrales. Nous avons choisi que ces faces portent une couleur moyenne de celles des trois faces voisines de la face triangulaire d'origine sur laquelle s'applique le schéma de Loop. L'expression de plongement de couleur de la règle est donc :

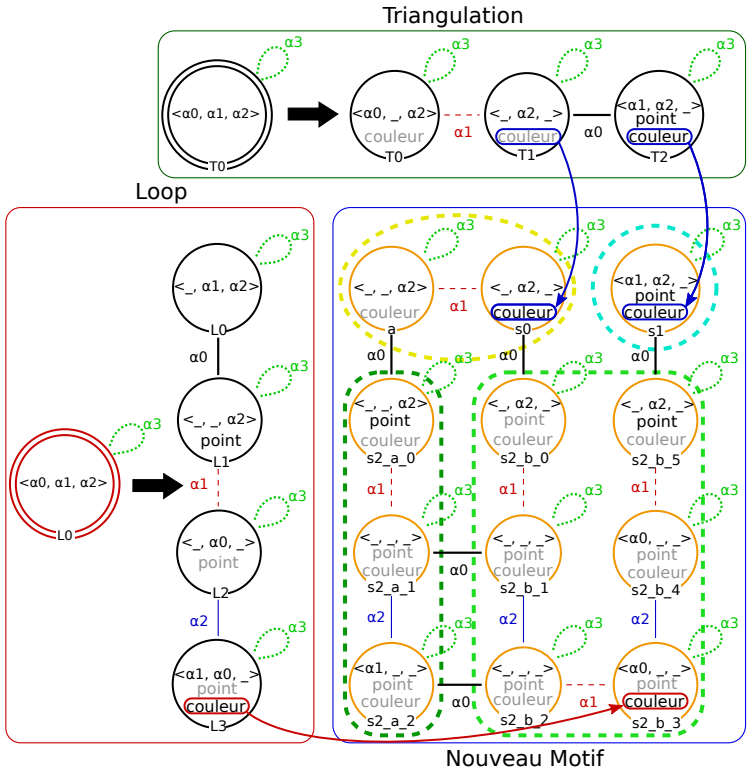


FIGURE 5.9 – Composition de la règle de triangulation barycentrique avec le schéma de Loop, avec l'ajout des expressions de couleur

```
return @ebd<couleur>::mix(L0@2.couleur,
                          L0@1@2.couleur,
                          L0@0@1@2.couleur);
```

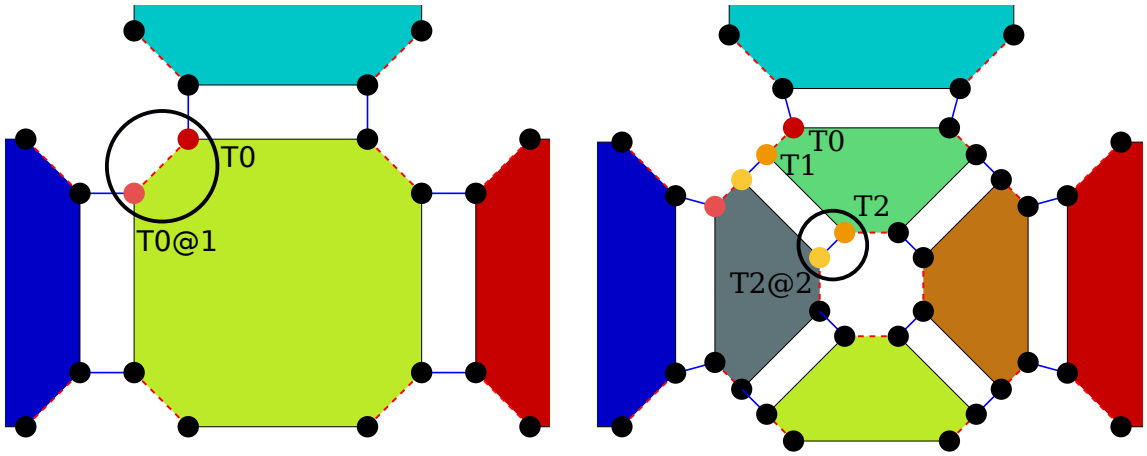
Commençons par la sous-expression `L0@2.couleur`. Nous avons vu pour les expressions du plongement `point` que `L0` correspond pour cette colonne à `T2`. Nous connaissons l'expression de couleur de `T2`, et la liaison α_2 fait partie de l'orbite de ce nœud. Cependant nous ne pouvons pas ré-écrire `L0@2.couleur` par l'expression portée par `T2` directement, même si la liaison est dans l'orbite de `T2`. Si `T2` représente un brin appartenant à la même face que le brin représenté par le nœud `s2_b_3`, ce n'est pas le cas de `T2@2`. Nous devons donc ré-écrire l'expression de couleur de `T2` pour obtenir la couleur de `T2@2`.

Rappelons que l'expression du plongement `couleur` de `T2` est :

```
return @ebd<couleur>::mix(T0.couleur, T0@2.couleur);
```

Si nous utilisons l'expression telle quelle, nous obtenons la couleur de `T2` et non celle de son voisin par α_2. La figure 5.10 illustre la triangulation d'un carré avec des brins étiquetés. Sur la figure 5.10(b), on observe que la couleur de `T2` et celle de son voisin par α_2 ne sont pas les mêmes. Le nœud `T0` ne représente pas le même brin pour le brin local représenté par `T2` et pour son voisin en α_2. Il en va de même pour `T0` et son voisin dans l'objet d'origine (figure 5.10(a)). Et on observe que les brins rouge et rose étaient liés par α_1 sur l'objet original. Ainsi, pour obtenir l'expression de `T2@2`, il faut renommer `T0` par `T0@1`.

Concrètement, pour trouver cette ré-écriture, il faut prendre en compte les liaisons avant renommage, comme nous l'avions fait pour le test d'équivalence à orbite près dans la section 1.3.4,



(a) Objet initial (b) Objet triangulé

FIGURE 5.10 – Illustration des changements de voisinages topologiques de `T0` après application de la règle de triangulation barycentrique.

page 34. Ici `T2` possède une liaison α_2 à la deuxième place dans son orbite. Dans le nœud d'accroche de la règle (`T0`), la deuxième liaison implicite est un α_1. Nous déterminons ainsi que le voisin de `T2` par α_2 est issu, avant transformation, du voisin par α_1 de `T0`.

Une fois les variables de nœud renommées, nous obtenons donc la sous-expression :

```
@ebd<couleur>::mix(a@1.couleur, a@1@2.couleur)
```

La sous-expression `L0@1@2.couleur` peut être réécrite de la même manière, et nous obtenons la sous-expression suivante :

```
@ebd<couleur>::mix(a@0@1.couleur, a@0@1@2.couleur)
```

Pour la sous-expression `L0@0@1@2.couleur`, les choses sont un peu différentes. En effet, comme nous l'avons vu, pour le nœud `s2_b_3`, `L0` doit être instancié par `T2`. Or son voisin par α_0 est le nœud `T1`. Et le voisin par α_1 de ce dernier est `T0`. La réécriture de la sous-expression après renommage des variables de nœud est donc simplement `a@2.couleur`.

En conclusion, nous obtenons pour le nœud `s2_b_3` l'expression du plongement `couleur` suivante :

```
return @ebd<couleur>::mix(
       @ebd<couleur>::mix(a@1.couleur, a@1@2.couleur),
       @ebd<couleur>::mix(a@0@1.couleur, a@0@1@2.couleur),
       a@2.couleur);
```

Nous avons vu un exemple de composition des expressions de plongement, cependant cet exemple était simple et ne comprenait qu'une seule ligne. Des expressions plus longues et complexes qui par exemple utilisent des variables, sont plus complexes à ré-écrire. Il nous faudra donc poursuivre notre réflexion pour prendre en compte l'ensemble des expressions possibles, y compris les alternatives et les boucles.

Notons qu'une telle composition de règle pourra être utilisée comme actuellement dans l'éditeur comme aide à la création d'opérations. Mais surtout, lorsqu'elle sera entièrement automatique, elle pourra être utilisée lors de la génération de code pour optimiser des séquences d'appels de règles.

5.2.2 Comutativité et associativité des opérateurs et fonctions

Grâce à notre langage, nous avons pu mettre en œuvre des vérifications automatiques sur les expressions et notamment implanter les vérifications de la préservation de la cohérence des plongements. Elles utilisent la notion d'équivalence à orbite près que nous avons vu dans la section 1.3.4, page 34. En pratique nous nous sommes aperçu que lors de l'écriture de nos expressions de plongements, une alerte d'équivalence est levée très souvent. Ces alertes sont trop peu souvent pertinentes. En effet, le principe d'équivalence à orbite près spécifie que pour un appel de fonction, on teste l'équivalence de deux expressions en vérifiant deux à deux leur paramètres. Ceci implique par exemple que les expressions :

```
barycentre(L0.point, L0@0.point)
```

et

```
barycentre(L0@0.point, L0.point)
```

ne sont pas équivalentes à orbite près. Le problème ici est que les vérifications ne prennent pas en compte l'éventuelle commutativité ou associativité des fonctions utilisateur. Si au moment de la vérification nous étions en mesure de savoir que la fonction `barycentre` est commutative, nous pourrions conclure que les deux expressions sont effectivement équivalentes à orbite près, en calculant les équivalences à commutativité et associativité près.

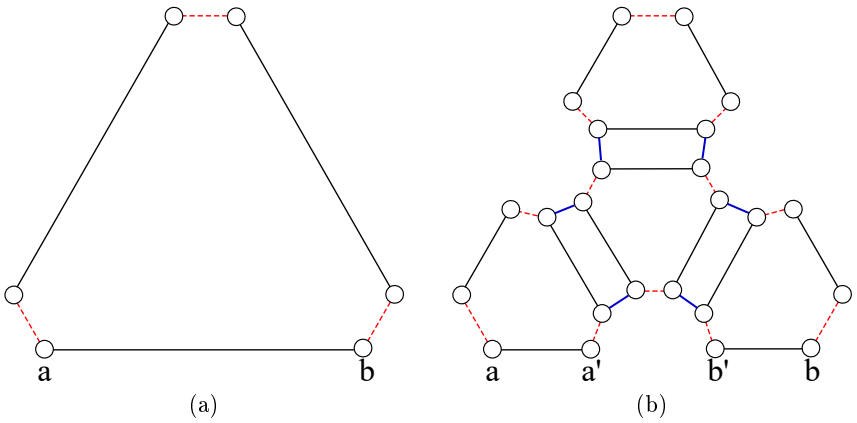


FIGURE 5.11 – Application de la règle du schéma de Loop sur un triangle

Reprenons l'exemple du schéma de Loop que nous avons vu dans la section précédente (cf. figure 5.5). Changeons l'expression du plongement `point` de Loop pour la remplacer par une équivalente (toujours sans lissage) :

```
return (L0.point + L0@0.point) * 0.5f;
```

Cette expression calcule le milieu de l'arête incidente à `L0` mais cette fois sans l'utilisation d'une collecte. Le problème ici est que cette expression est évaluée de deux façons différentes pour deux nœuds appartenant à la même orbite du plongement `point` (i.e. au même sommet).

Par exemple, figure 5.11, l'application du schéma de Loop sur le triangle figure 5.11(a) produit la figure 5.11(b). Or chaque sommet ajouté est calculé par deux expressions différentes. Par exemple le milieu entre les brins `a` et `b` est calculé par l'expression :

```
(a.point + a@0.point)*0.5f
```

pour le brin `a'` et par

```
(b.point + b@0.point)*0.5f
```

pour le brin `b'`. Mais `a.point` n'est pas équivalent à orbite près à `b.point`, mais à `b@0.point` (car `b@0` n'est autre que `a` dans le triangle de départ).

Notons néanmoins que ce problème ne se pose pas lorsque l'on utilise une collecte, comme initialement dans le schéma de Loop :

```
return @ebd<point>::barycentre(<0,2,3>_point(a))
```

car les collectes produisent des collections supposées non ordonnées. Ainsi, sur l'exemple figure 5.11, les deux instantications du milieu pour `a'` et `b'` sont bien équivalentes à orbite près. Charge à la fonction utilisateur `barycentre` de ne pas violer ce principe des collections non ordonnées.

5.3 Évolution de la génération du code

5.3.1 Compilation et optimisation

L'opérateur `map` que nous avons décrit dans la section 5.1.2, page 102 peut être traduit de plusieurs façons. Soit le moteur d'application a une fonction dédiée au traitement de l'opération et la traduction exploite cette fonctionnalité, soit il n'en a pas et l'opération doit être traduite par une boucle, même si ce n'est pas très efficace. Prenons l'expression suivante (avec `face` un brin) :

```
JerboaList<JerboaDart> listeBrinFaceVoisine = @map(@3,<0,1>(face));
```

Sa traduction par une simple boucle `foreach` comme suit ne serait pas efficace :

```
JerboaList<JerboaDart> listeBrinFaceVoisine;
for(JerboaDart d : <0,1>(face)){
  listeBrinFaceVoisine.add(d@3);
}
```

Le problème ici est qu'une liste est créée par la collecte puis parcourue pour accéder au voisin par α_3 de chacun de ses brins. Il y a donc ici deux parcours successifs sur les brins. Si le moteur d'application possède une méthode dédié au `map`, la création de la liste `listeBrinFaceVoisine` est faite pendant le premier parcours des brins de la face incidente au brin `face`, évitant ainsi d'autres parcours inutiles.

Il est possible d'aller plus loin encore dans l'étape de compilation de notre langage. Avec la présence d'un opérateur dédié à l'opération `map`, nous sommes en mesure de détecter que certaines expressions sont optimisables comme :

```
for(JerboaDart d : <0,1>(face)){
  @rule<Triangulation>(d);
}
```

Pendant notre étape de compilation du langage (cf. section 3.2.2, page 73), cette expression peut être traduite par un `map` et ainsi être automatiquement optimisée si le moteur d'application le permet. Ceci rendrait le processus d'optimisation complètement transparent pour l'utilisateur, même s'il n'a pas pensé à optimiser lui-même son expression avec l'opérateur `map`.

5.3.2 Génération dans d'autres langages

Jerboa comprend actuellement deux noyaux dans deux langages de programmation distincts, le Java et le C++. Grâce à l'éditeur de modeleur, nous sommes en mesure d'ajouter la possibilité de générer nos opérations dans d'autres langages. Les processus d'analyse syntaxiques et sémantiques sont communs à tous les langages donc seule l'étape finale de génération du code après analyse doit être spécialisée pour chaque langage. L'implantation de Jerboa dans différents langage permet d'autoriser l'utilisation de bibliothèques externes spécifiques dans notre outil et de pouvoir profiter des avantages de différents langages selon le type d'application que l'on souhaite développer. Nous sommes notamment actuellement en train de développer un noyau en C#. Nous pourrions également développer un noyau de Jerboa qui permet de créer des modeleurs utilisables sur un appareil mobile sous Android par exemple, et réutiliser les modeleurs avec lesquels nous travaillons actuellement en Java ou C++, et les re-générer pour le nouveau moteur sur Android. La force de notre langage est qu'il n'est pas nécessaire de ré-implanter toutes les opérations pour passer de l'utilisation d'un noyau à un autre. La seule différence notable est potentiellement les noms des fonctions externes à Jerboa utilisées dans les expressions, qui peuvent différer entre les langages.

5.4 Évolution de la bibliothèque d'exécution

5.4.1 Invariants de modeleur

Jerboa propose des vérifications automatiques de préservation de la cohérence de la topologie et des plongements. Il serait intéressant d'ajouter des vérifications dynamiques dépendantes du modèle et de ses contraintes métier. Nous avons imaginé ajouter dans l'éditeur la définition de contraintes, qui seraient vérifiées à chaque application d'opération. En géologie par exemple, il n'est pas souhaitable que deux mailles 3D s'interpénètrent géométriquement. Actuellement, si une opération crée des interpénétrations, l'utilisateur n'en est pas informé, sauf si il crée une opération dédiée à la vérification de cette contrainte, et qu'il l'applique régulièrement. Avec les invariants de modeleur, l'utilisateur pourrait définir cette contrainte métier dans l'éditeur. Lors de l'application d'une opération, l'invariant serait testé, afin de vérifier si des mailles s'interpénètrent ou non.

Si l'invariant n'est pas respecté, plusieurs solutions peuvent être envisagées. L'utilisateur peut simplement vouloir en être informé par un message d'erreur. On peut également souhaiter un retour en arrière afin d'annuler l'opération qui a créée l'incohérence. Une autre solution serait de définir des états du modeleur, et faire passer un modeleur en état incohérent lorsqu'un invariant n'est pas respecté. Avec les états de modeleur, nous pourrions spécifier via l'éditeur, que seules certaines règles de correction, choisies par l'utilisateur, peuvent être appliquées lorsque le modeleur est dans un état incohérent, pour rétablir sa cohérence.

L'ajout des invariants de modeleur permettrait d'avoir un processus complet de vérification, à la fois pour la cohérence du modèle, mais également pour assurer la cohérence avec les contraintes métier.

5.4.2 Parallélisation et gain de performance

Actuellement, les traitements effectués par le moteur d'applications des règles dans les noyaux de Jerboa sont réalisés en séquentiel. La parallélisation des moteurs d'applications permettrait d'améliorer grandement leur performances en terme de temps d'exécution. Plusieurs étapes de l'application des règles peuvent être parallélisées, mais cette tâche n'est cependant pas aisée. En

effet, l'application des opérations, règles ou *scripts*, nécessite des parcours lors du filtrage, ou le calcul d'une collecte par exemple. Comme tout parcours de graphe, ils utilisent des marques. Malheureusement, la parallélisation des parcours est complexe, car il faut gérer les accès concurrents de marquage et de test de marque. Or lorsque le nombre d'accès concurrent est important dans un calcul, une version parallélisée d'un code peut parfois être moins efficace que la version séquentielle. Pour éviter un maximum les accès concurrents, une solution possible est de pré-parcourir et stocker dans des tableaux, les éléments sur lesquels se portent des modifications similaires : changement de voisin topologique, calcul d'une expressions de plongement, *etc.* Une étude serait nécessaire pour vérifier si cette solution permet, en pratique, de gagner en performance ou si avec les pré-calculs, le gain n'est pas significatif. Notons que ces résultats dépendent du nombre de brins dans la carte généralisée, du nombre de plongement, *etc.*

Un autre type de parallélisation est envisageable pour l'application parallèle de plusieurs règles. Prenons l'exemple du *script* suivant (avec `volume` un brin) :

```
for(JerboaDart d : <0,1,2,3>_<0,1,3>(volume) ){
  try{
    @rule<TriangulationBarycentrique>(d);
    @rule<ChangeCouleurFace>(d, @ebd<couleur>(255,0,0));
  }catch(JerboaException e){}]
}
```

Cette portion de *script* applique successivement une règle de triangulation barycentrique et un changement de couleur (en rouge), à toutes les faces d'un volume. Nous considérons ici la règle de triangulation qui s'applique sur une seule face (d'orbite $\langle 0,1,3\rangle$), comme la règle `ChangeCouleurFace`, dont l'application échoue si la face est déjà un triangle. Grâce à l'éditeur de modeleur, nous sommes en mesure de détecter que tous les éléments sur lesquels la boucle itère, sont indépendants vis à vis de l'application de la règle de triangulation. En effet, la collection calcule ici un brin par sous-orbite $\langle 0,1,3\rangle$, qui est justement l'orbite de filtrage de la règle de triangulation. Ainsi on est assuré qu'une application de la règle, ne modifie pas plusieurs brins de la collecte mais un seul car les faces voisines de celle qui est triangulée ne sont pas modifiées topologiquement, de même que leurs plongements. Le fait que chaque itération soit indépendante permettrait d'appliquer de façon parallélisée la triangulation et changement de couleur de toutes les faces collectées. Encore une fois, le langage nous permettrait de détecter ce type d'instruction et de générer du code effectuant la parallélisation de façon transparente pour l'utilisateur.

Pour aller plus loin, l'éditeur est également en mesure de détecter statiquement dans un cas similaire à l'exemple précédent, si il est possible de créer une règle qui compose les deux opérations de triangulation et de changement de couleur de la face. Dans ce cas, l'éditeur crée automatiquement la règle composée des deux opérations, puis appelle la règle composée plutôt que les deux règles successivement. Ce type d'optimisation permettrait d'éviter un double filtrage et le calcul des plongements intermédiaires comme nous l'avons vu dans la section 3.3, page 79 et la section 5.2.1, page 105.

Notons la aussi que ce type d'optimisation étant indépendante du langage de Jerboa, un modeleur existant pourra être re-généré pour bénéficier des optimisations des nouvelles versions de la bibliothèque Jerboa.

5.4.3 Optimisation des collectes

Pour réaliser des collectes, dans le moteur nous utilisons un parcours générale qui utilise une pile et un système de marque. Ces calculs peuvent être optimisés. Grâce aux contraintes de cycles des cartes généralisées, il est possible de spécifier des algorithmes de parcours en fonction

de l'orbite à parcourir. Par exemple, pour réaliser la collecte $\langle 0,1\rangle$(`brin`), il est possible d'utiliser un parcours dédié qui n'utilise ni pile ni marque en utilisant un algorithme similaire à celui-ci :

```
JerboaDart tmp = brin;
JerboaList resultat;
int dimension = 0;
resultat.add(tmp);
while(tmp@dimension != tmp && tmp@dimension != brin){
  tmp = tmp@dimension;
  resultat.add(tmp);
  dimension = (dimension+1)%2;
}
if(tmp@dimension!=brin){// si on a une face ouverte
  dimension = 1;
  while(tmp@dimension != tmp && tmp@dimension != brin){
    tmp = tmp@dimension;
    resultat.add(tmp);
    dimension = (dimension+1)%2;
  }
}
```

L'algorithme fait le tour de la face, en alternant l'accès aux voisins par α_0 et par α_1. Il y a deux boucles `while` pour tenir compte de la possibilité de la présence de faces ouvertes. En effet, les cartes généralisées permettent également de représenter des faces ouvertes. Il est donc possible que l'un (plutôt deux) brins de la face, présente une boucle en α_0 ou en α_1. Dans ce cas, il faut parcourir l'autre côté de la face pour atteindre l'autre bout. De même, nous pouvons utiliser les cycles des cartes généralisées pour calculer les orbites $\langle 0,1,3\rangle$ des faces. Dans une carte généralisée, il existe des cycles $\alpha_0\alpha_3$ et $\alpha_1\alpha_3$, il est donc possible d'utiliser un algorithme similaire au précédent, sans parcourir directement la face voisine par α_3 du brin `brin`, en ajoutant à la liste `resultat` chaque brin en même temps que leur voisin par α_3.

Nous pourrions utiliser un algorithme similaire pour d'autres permutations comme les extrémités d'arêtes $\langle 2,3\rangle$ et les arêtes $\langle 0,2,3\rangle$ par exemple. Chaque type d'orbite peut faire l'objet de l'implantation d'un parcours spécifique pour optimiser l'efficacité des parcours.

L'optimisation des parcours d'orbites serait un gain majeur pour la bibliothèque car ce sont des calculs très fréquemment utilisés lors de l'application des opérations.

5.4.4 Optimisation des moteurs

Lors du calcul des expressions de plongements ou des *scripts*, il est courant de réaliser des itérations sur les brins d'une collecte. Nous avons évoqué dans la section 5.1.2, page 102 l'application d'une règle sur toutes les sous-orbites d'une orbite. Par exemple nous avons écrit le *script* suivant qui décrit l'application sur toutes les faces d'un volume, d'une règle de décousure de face en α_3 (avec ici `volume` un brin de la G-carte) :

```
for(JerboaDart fi : <0,1,2>_<0,1>(volume)){
  @rule<DecoutureFaceAlpha3>(fi);
}
```

Ici l'itération sur la collecte n'est pas très efficace, les brins sont parcourus plusieurs fois. Un premier parcours de la G-carte est réalisé pour calculer la collecte. La boucle fait ensuite un deuxième parcours, cette fois sur la collection résultant de la collecte. Une version plus optimisée

serait de réaliser l'application de la règle pendant la collecte, et ainsi éviter l'itération sur le résultat.

Une méthode qui applique une règle directement à tous les brins d'une collecte est plus efficace. Cette méthode pourrait avoir la signature suivante :

```
JerboaRuleResult appliqueRegleSurOrbite(JerboaDart brin,
                JerboaOrbit orbite, string nomRegle)
```

Cette méthode optimisée ne crée pas de liste intermédiaire mais applique la règle au fur et à mesure du parcours de l'orbite. Notons que cette optimisation ici ne fonctionne que si la règle n'a qu'un nœud à gauche. Dans le cas contraire, cette optimisation pourrait être un peu plus complexe selon la façon dont sont choisis les différents brins accrochés.

On peut étendre ce type d'opération à une opération quelconque, autre que l'application d'une règle, comme une fonction par exemple. Ce type d'instruction est usuellement connu sous le nom `map` comme nous l'avons vu dans la section 5.1.2, page 102. Cette instruction applique une fonction sur tous les éléments d'une collection et retourne la liste des retours respectifs de la fonction appliquée à chaque élément de la collection.

La liste est créée par le map directement pendant la collecte, sans passer par la création puis le parcours d'une liste intermédiaire. Cette fonction peut par exemple être implantée dans le moteur Java avec la signature suivante :

```
List<JerboaDart> collecteFunctor(JerboaDart brin, JerboaOrbit orbite,
                  Object obj, String nomFonction)
```

Ici la fonction prend en paramètre un objet et le nom d'une méthode de sa classe pour l'appliquer en utilisant la réflexivité Java. En C++, la signature pourrait ressembler à la suivante avec l'utilisation de template et un pointeur de fonction :

```
template <class C>
std::vector<C> collecteFunctor(JerboaDart brin, JerboaOrbit orbite,
                C(*pointeurFonction)(JerboaDart*))
```

Une autre possibilité est l'utilisation des lambda-termes qui existent à présent dans les deux langages (depuis Java 8, et C++11). Le calcul serait ainsi passé à l'opération `map` sous forme d'une lambda-expression.

Deuxième partie

Modélisation du sous-sol (réservoirs)

Chapitre 6

État de l'art

Nos travaux portent sur la modélisation du sous-sol dans des zones de réservoirs. Les réservoirs sont des zones géologiques desquelles peuvent être extraits du pétrole ou du gaz, et où l'on peut éventuellement envisager par la suite de stocker du CO_2. L'outil informatique est souvent utilisé afin d'optimiser les extractions d'hydrocarbures. Pour ce faire, il est nécessaire d'avoir une représentation informatique du sous-sol. Pour modéliser un sous-sol géologique, des mesures sont prises sur le terrain afin de déterminer l'organisation des couches sédimentaires, la présence de failles, *etc.* Pour obtenir ces informations, on émet des ondes sismiques dans le sol, et on mesure les ondes de retour. Ces dernières permettent à des géologues de déterminer la position des différentes couches géologiques [AMS+04] également appelées *unités*, et des surfaces limites qui les séparent, appelées des *horizons*. Ces mesures sont ensuite transformées en nuages de points plongés dans un espace 3D. Dans les nuages de points, il peut apparaître des zones sans points, ou des zones bruitées indiquant, par exemple, la présence de failles, dont la position précise est difficile à déterminer. La reconstruction 3D du sous-sol s'appuie sur les données de mesures interprétées par des géologues. Mais les informations géométriques ne sont pas les seules à reconstruire pour avoir une représentation complète du sous-sol. En effet, la composition du sous-sol est également importante à déterminer pour connaître les propriétés pétrophysiques du réservoir. Ces propriétés sont obtenues par des prélèvements, notamment sous forme de carottes, qui sont analysées afin de connaître précisément la composition de chaque unité géologique. Ces informations permettent de peupler le maillage géométrique 3D avec des propriétés pétrophysiques dans les zones de carottages, et par extrapolation le long des couches stratigraphiques, qui découpent les unités.

Les modèles 3D complets contenant toutes les informations géométriques et pétrophysiques sont exploités afin de réaliser des simulations d'écoulement de fluides qui permettent d'optimiser l'extraction des ressources.

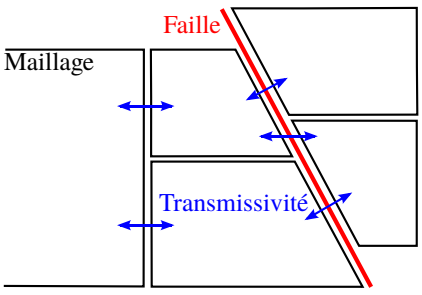
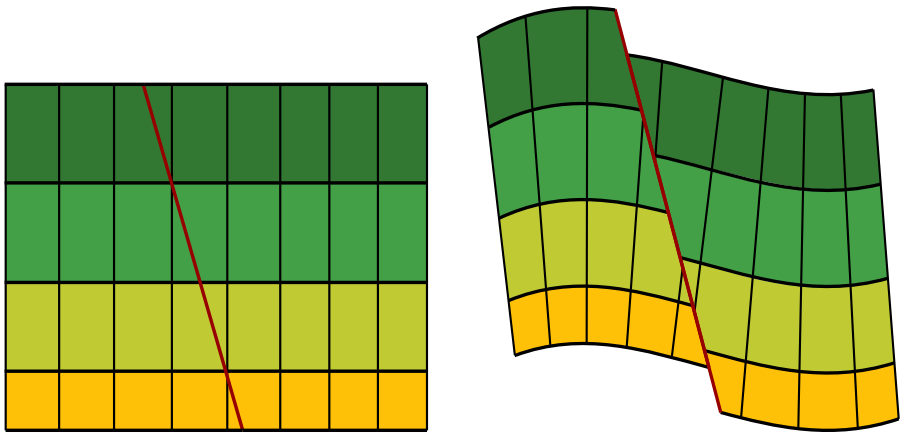


FIGURE 6.1 – Transmissivité entre les mailles

Les modèles 3D sont constitués de mailles 3D (cubes, tétraèdres, *etc.*). L'écoulement des fluides [Lai00] est calculé en fonction des propriétés pétrophysiques des mailles, qui permettent de déterminer la transmissivité le long de leurs différentes faces, comme illustré en figure 6.1 avec une vue en coupe. Le débit de fluide qui transite entre deux mailles voisines est calculé en fonction de la géométrie et des propriétés des mailles. La qualité des simulations des écoulements de fluides dépend directement du maillage 3D utilisé. Ce dernier doit donc respecter au mieux différents critères de qualité. Le premier travail de cette thèse sur cette deuxième partie a été de les identifier et de les hiérarchiser.

6.1 Besoin

Pour réaliser un maillage correct du sous-sol, il faut tenir compte du réalisme géologique. Les maillages doivent suivre les déformations du sous-sol. Par exemple, si un maillage contient des arêtes qui relient les différents horizons entre eux, elles ne doivent pas être créées verticalement mais suivre les déformations. Dans la suite du manuscrit nous appellerons ces arêtes, plus ou moins verticales, des *piliers*.



(a) Maillage d'unités non déformées au moment du dépôt

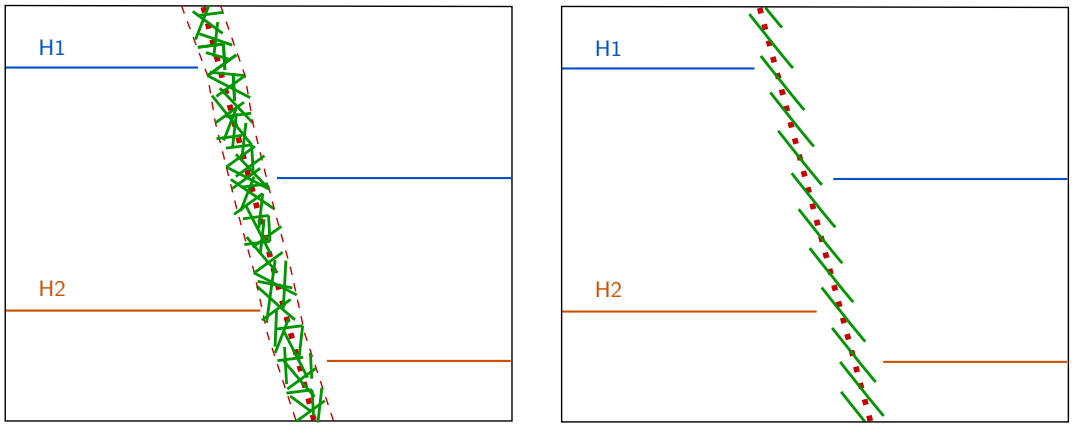
(b) Maillage d'unités pliées et déformées par les failles à l'époque actuelle

FIGURE 6.2 – Maillages d'unités

La figure 6.2 montre le maillage recherché à la figure 6.2(b). Il qui correspond à l'évolution, suite aux divers évènements géologiques. Ce maillage comprend des piliers verticaux dans l'espace de dépôt (figure 6.2(a)).

Il existe plusieurs types d'évènements géologiques qu'il faut prendre en compte pour créer un maillage du sous-sol. Il y a bien évidemment les dépôts sédimentaires, qui forment les unités stratigraphiques, séparées par des horizons, qui sont supposés être des surfaces sans épaisseur. Une unité géologique et son contenu sont définis comme l'espace entre deux horizons distincts. Outre les plissements, les failles et les érosions modifient ces unités.

Déterminer la position des failles n'est pas aisé car dans la réalité, les failles ne sont pas de grandes surfaces sur lesquelles glissent les couches géologiques mais sont des zones complexes occupées par des recristallisations ou des microfractures comme illustré en figure 6.3. En pratique, et pour simplifier, les géologues représentent les failles par des surfaces. La figure montre en vue de coupe, un horizon $H1$ (bleu), un horizon $H2$ (orange), la surface de faille issue de l'interprétation (pointillés rouges) et les zones de faille (vert). Ces zones de faille se traduisent par du bruit

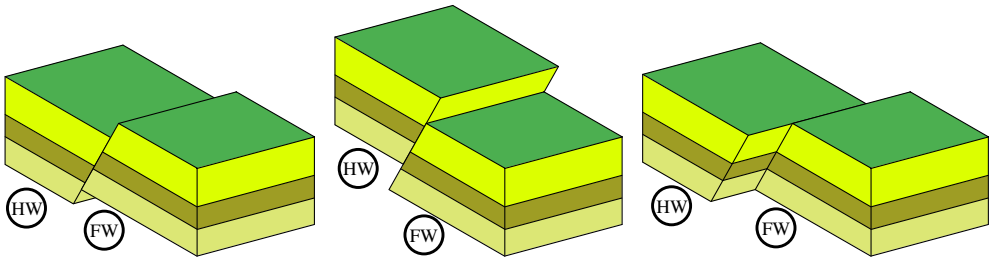


(a) Faille remplie par des recristallisations

(b) Faille constituée par des micro-fractures en échelons

FIGURE 6.3 – Zone faillées

dans les données, ce qui permet de les identifier mais pas de localiser la surface de faille dans une position précise. Les géologues interprètent les données afin de déterminer une surface de faille, qui soit la plus proche possible des multi-fractures réelles. La position des failles étant incertaine, un géologue peut parfois vouloir tester plusieurs hypothèses et produit ainsi plusieurs interprétations d'un même jeu de données.



(a) Faille normale

(b) Faille inverse

(c) Décrochement horizontal (ou *Strike & Slip*)

FIGURE 6.4 – Différentes déformations provoquées par des failles

On peut catégoriser les failles en fonction des types de déformations qu'elles engendrent. Ces différentes déformations sont illustrées en figure 6.4. Les failles coupent les unités géologiques en deux parties qui "glissent" l'une contre l'autre. On appelle *rejet* la longueur du glissement. Dans la suite du manuscrit nous distinguerons ces deux parties par les termes *foot-wall* et *hanging-wall* notés respectivement FW et HW sur la figure. Le *foot-wall* est la partie située sous la surface de faille et le *hanging-wall* est la partie située au dessus. Les failles normales, présentées en figure 6.4(a), indiquent à un glissement qui provoque un écartement du sol de part et d'autre de la faille. Les failles inverses, présentée en figure 6.4(b), indiquent un glissement inverse dû à une compression. Enfin, les failles de décrochement, présentées en figure 6.4(c), indiquent un glissement latéral.

En outre, les failles peuvent provoquer des basculements quand elles sont organisées en systèmes de failles parallèles, à la manière de dominos, comme le montre la figure 6.5.

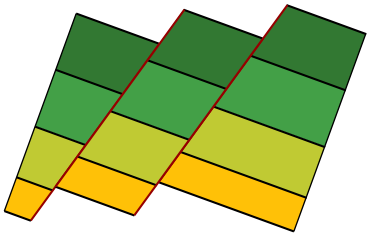
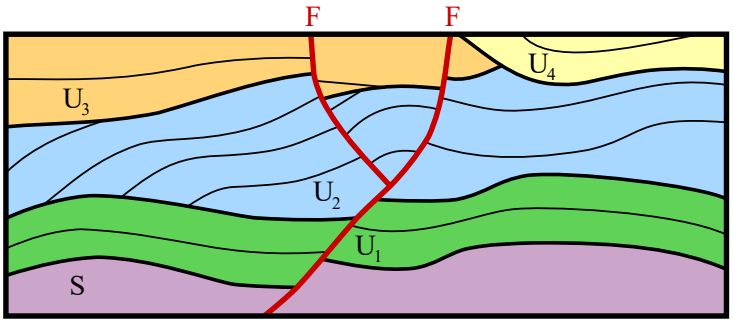


FIGURE 6.5 – Basculement d'unités dans un système de failles

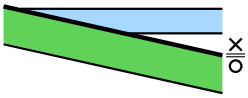
Les érosions aussi modifient les unités stratigraphiques mais par enlèvement de matière. Le dépôt de couches sédimentaires postérieurs aux basculement et aux érosions forment des configurations particulières.



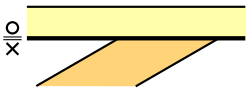
(a) Exemple complet



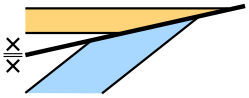
(b) Parallèle (cas de U_1 sur S)



(c) *On lap* (cas de U_2 sur U_1)



(d) Discordance (cas de U_4 sur U_2 et U_3)



(e) Discordance et *on lap* (cas de U_3 sur U_2)

FIGURE 6.6 – Différents types de voisinages entre les unités

Tous ces événements (plissements, basculements, et érosions) créent différentes configurations locales à la surface de contact (horizon ou érosion) de deux unités. Prenons l'exemple complet de la figure 6.6(a). Ce modèle est composé de quatre *Stratigraphic Unit Stack* [PR13] (ou SUS), U_1, U_2, U_3 et U_4 surmontant le soubassement S. Il existe quatre types de voisinages entre deux unités. Ces voisinages peuvent être caractérisés par des étiquettes portés par les faces du haut et du bas des surfaces de contact. Le symbole X signifie que les couches stratigraphiques coupent la surface (noté *cut*) et le symbole O signifie que les couches stratigraphiques sont concordantes à la surface (noté *not cut*). Sur les figures 6.6(b), 6.6(c), 6.6(d), 6.6(e), la surface de contact est représentée par le trait le plus épais et les traits plus fins représentent la stratification, c'est à dire la disposition des unités au sein de chaque *Stratigraphic Unit Stack*. La figure 6.6(b) illustre le cas où deux unités voisines sont conformes. La figure 6.6(c) illustre le cas où l'unité du dessus n'est pas conforme à l'horizon qu'elle partage avec celle du dessous. La figure 6.6(d) illustre le cas où l'unité du dessous n'est pas conforme à l'horizon qu'elle partage avec celle du dessus. La figure 6.6(e) illustre le cas où aucune des deux unités n'est conforme à la surface d'horizon.

Les failles découpent les unités, et avec la terminologie *cut*/*not cut*, les deux côtés des surfaces de faille ont donc des étiquettes *cut*, comme illustré sur la figure 6.6(a) avec les failles notées F.

Il arrive que des dépôts se produisent en même temps que des déformations provoquées par

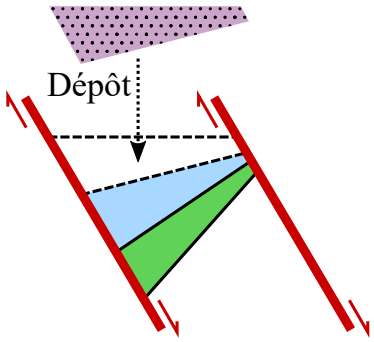


FIGURE 6.7 – Dépôt syntectonique

des failles. On appelle ces dépôts des dépôts syntectoniques. Les dépôts se font en biseaux comme illustré sur la figure 6.7. Dans ce cas, notamment, les piliers du maillage ne peuvent alors pas être rigoureusement perpendiculaires à la stratification sans les représenter par des surfaces courbes.

Pour faciliter les simulations d'écoulement de fluide dans le maillage 3D, ce dernier doit respecter certaines contraintes. L'une de ces contraintes est que les mailles doivent suivre la direction d'écoulement. Dans le sous-sol, le pétrole ou les gaz ne sont pas nécessairement présents sous forme de nappe, mais plutôt emprisonnés dans les pores des roches. Ainsi l'extraction des ressources n'est pas réalisée par simple pompage dans une nappe ou une poche de gaz. Pour faire remonter les hydrocarbures emprisonnées dans les roches, une méthode consiste à injecter de l'eau, qui est plus dense, pour pousser les hydrocarbures dans le réservoir. Dans cet objectif, deux puits sont creusés, l'un pour injecter l'eau et l'autre pour extraire les hydrocarbures. La direction d'écoulement est simplement la direction entre le puits d'injection et le puits d'extraction. Pour une extraction optimale, des études sont réalisées pour placer les puits d'injection et d'extraction de façon à pousser au mieux les fluides à extraire. L'outil informatique permet de tester plusieurs hypothèses et ainsi définir la meilleur position des différents puits.

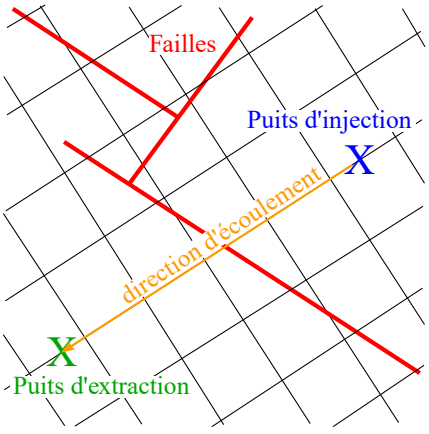


FIGURE 6.8 – Direction des mailles

Pour rendre les simulations d'écoulement plus fiables, les mailles doivent globalement être orientées dans la direction d'écoulement des fluides, comme illustré en figure 6.8 en vue de dessus.

Une autre contrainte à respecter pour améliorer la fiabilité des simulations est la régularité des mailles. Les calculs sont plus faciles à réaliser sur un maillage régulier composé d'un seul type de volume que sur un maillage très hétérogènes. Les maillages qui donnent les simulations les plus fiables sont les maillages hexaédriques. Cette contrainte est cependant contradictoire avec la "verticalité" des piliers (cf. figure 6.2) au abords des failles. En effet, les plans de failles ont couramment une inclinaison de 45° à 60° par rapport à la verticale et viennent ainsi couper les mailles hexaèdriques régulières. Dans cette thèse nous visons des simulateurs d'écoulement de fluides capables de gérer des maillages majoritairement hexaèdriques, mais qui peuvent avoir d'autres types de mailles (tetraèdres, prismes, *etc.*) en particulier le long des failles.

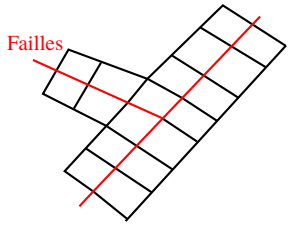


FIGURE 6.9 – Exemple de maillage concordant aux failles

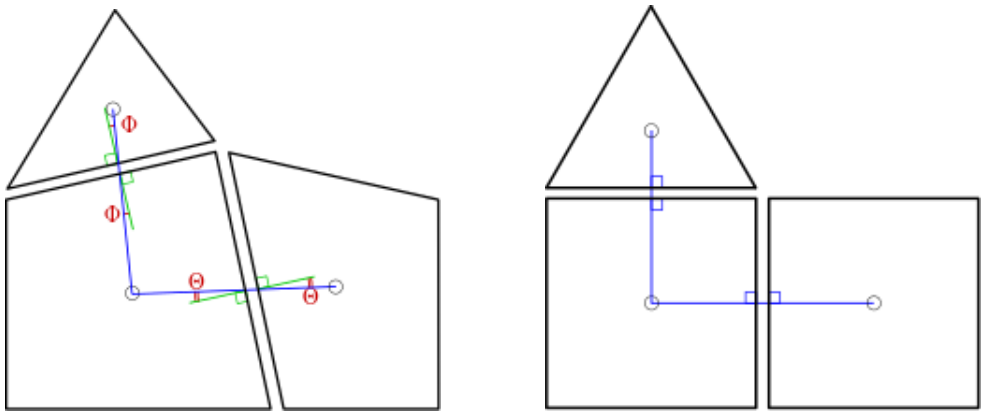
De même, les failles ne sont pas toujours parallèles ou perpendiculaires à la direction d'écoulement, comme le montre la figure 6.8, ce qui peut multiplier les découpes des mailles aux abords des failles. Pour limiter le nombre de ces découpes, il convient d'adapter géométriquement et topologiquement le maillage aux abords des failles comme le montre la figure 6.9.

La plupart du temps, le maillage des horizons, ne pose pas de problème. Comme le montre la figure 6.2, les piliers peuvent traverser plusieurs unités géologiques et ainsi préserver la régularité du maillage. Il en est ainsi chaque fois que les unités successives sont dans le cas de la figure 6.6(b). Mais dans les autres situations des figures 6.6(c), 6.6(d) et 6.6(e), garder la régularité topologique du maillage peut entraîner des déformations géométriques plus ou moins importantes, ou même rompre la régularité topologique. Selon les situations, ces discordances le long des horizons et surfaces d'érosion sont problématiques ou non.

En effet, dans le cas de couches imperméables par exemple, il n'est pas nécessaire de forcer leur concordance avec les couches supérieures ou inférieures [PZRS05, RPB05]. Ces couches ne laissant pas passer les fluides, les calculs de simulations ne tiennent alors pas compte de la transmissivité de leurs horizons incidents. Ainsi, le maillage 3D doit être optimisé pour une zone d'intérêt qui regroupe plusieurs unités toutes concordantes : un *Stratigraphic Unit Stack* [PR13]. Pour les autres *Stratigraphic Unit Stack*, au dessus et en dessous de la zone d'intérêt, selon la perméabilité de leurs surfaces, le maillage 3D sera prolongé, quitte à être géométriquement déformé, ou au contraire, un nouveau maillage sera créé.

Enfin, le maillage doit être le plus régulier possible géométriquement et ses mailles doivent respecter au mieux le principe d'orthogonalité. La figure 6.10 illustre le calcul de l'orthogonalité entre deux mailles. Deux mailles vérifient le principe d'orthogonalité si le segment qui relie leur barycentres (en bleu sur la figure) est perpendiculaire à leur arête commune. Sur la figure 6.10(a), ce critère n'est pas respecté. On observe une déformation d'angle Φ entre le quadrangle de gauche et le triangle, et d'angle Θ entre les deux quadrangles. Un maillage correcte de cette configuration est illustré en figure 6.10(b).

La satisfaction simultanée de toutes ces contraintes est impossible car elles sont contradictoires. L'un des objectifs de cette thèse est de construire un maillage 3D qui offre un bon compromis entre ces différentes contraintes.



(a) Mauvaise qualité (b) Bonne qualité

FIGURE 6.10 – Illustration de la qualité d'un maillage en fonction du critère d'orthogonalité

Comme nous l'avons déjà évoqué, les données de mesures seules ne suffisent pas pour créer un maillage. Elles sont parfois imprécises, notamment au niveau des failles. L'interprétation des géologues est indispensable pour déterminer l'âge des différents évènements géologiques (dépôts, failles, érosions, basculements, *etc.*) dans la région d'intérêt. Pour un même jeu de données, plusieurs interprétations différentes peuvent être définies, qui produisent des maillages 3D différents. Les interprétations s'appuient sur les connaissances géologiques des spécialistes, et les maillages 3D produits doivent respecter notamment les trois règles suivantes :

- principe de superposition : les couches plus récentes se déposent au-dessus des couches antérieures ;
- les sédiments se déposent horizontalement et sont continus latéralement ;
- les variations par rapport à l'état lors du dépôt sont explicables par des événements postérieurs à celui-ci (failles, érosions, basculements, *etc.*).

6.2 Approches

Il existe plusieurs méthodes pour modéliser le sous-sol avec des mailles 3D. Comme nous l'avons vu dans la section précédente, de la qualité du maillage va dépendre la qualité de la simulation d'écoulement de fluide. Afin de la maximiser, il faut donc tenir compte au mieux des différentes contraintes que nous avons présentées. Cette section présente quelques méthodes de maillage existantes qui répondent à certains des critères précédents.

6.2.1 Maillage de l'espace 3D

Une méthode simple de maillage consiste à créer des grilles. L'avantage des grilles est leur régularité, ce qui facilite les calculs des simulations d'écoulement de fluide. Il existe plusieurs types de grille. Certaines sont composées de tetraèdres et d'autres d'hexaèdres. Certaines grilles suivent les déformations et sont bordées par les failles. Ce type de grille peut notamment être réalisé dans le logiciel SKUA [SKU15].

Pour certains types de grille (*Stair-Step Grids* dans SKUA), si les déformations proches des failles sont trop importantes, les mailles ne suivent plus localement les déformations mais sont disposées en marches d'escalier, ce qui permet de conserver la régularité des mailles sans trop de

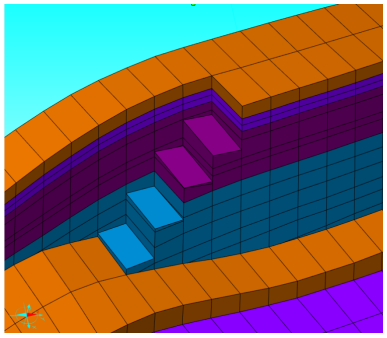
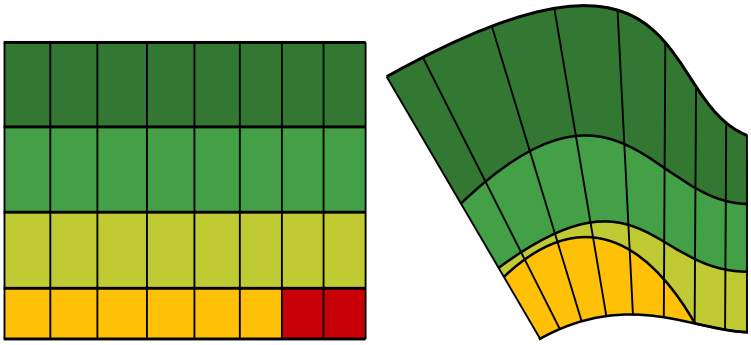


FIGURE 6.11 – Vue d'un maillage *Stair Step* dans SKUA

déformations, comme sur l'image figure 6.11.



(a) Grille complète d'origine (b) Grille plongée sur les données réelles

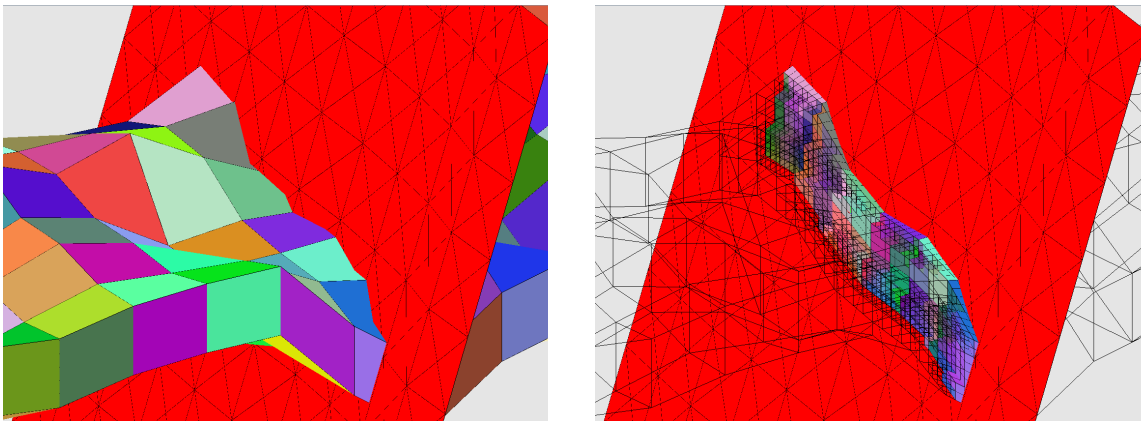
FIGURE 6.12 – Déformations dues a l'utilisation d'une grille régulière

Dans les grilles régulières qui suivent les déformations, lors de déformations trop importantes, certaines mailles peuvent être de volume nul. La figure 6.12 illustre un cas où deux mailles (en rouge) sont présentes dans la grille mais ne devraient pas exister dans la représentation finale, à cause de fortes déformations, dues par exemple à des érosions ou des compressions. Ces mailles sont marquées inactives, mais ne sont pas supprimées, pour conserver la structure régulière de la grille.

Certaines grilles permettent également de réaliser des maillages multi-échelles comme les *Local Grid Refinement* [QRT15] de SKUA. Ces maillages multi-échelles permettent de représenter des zones critiques, comme les failles, avec plus de précision, tandis que les zones plus simples sont représentées à plus grosse échelle.

Nous avions nous-même exploré cette piste au début de cette thèse comme c'est illustré en figure 6.13. Sur cette figure, toutes les mailles initiales coupant la faille (en rouge) ont été subdivisées. Cette opération a été répétée ici trois fois. Nous avons abandonné cette méthode afin de pouvoir créer des maillages qui contiennent des mailles tronquer au proche des failles afin de suivre plus précisément leur position.

Il existe d'autres méthodes sans grille régulière qui utilisent des surfaces limites pour créer des maillages. Ces méthodes consistent à utiliser les surfaces 2D des horizons et des failles comme les bords de boîtes dans lesquels sont générés des mailles pour remplir l'espace 3D. Le maillage

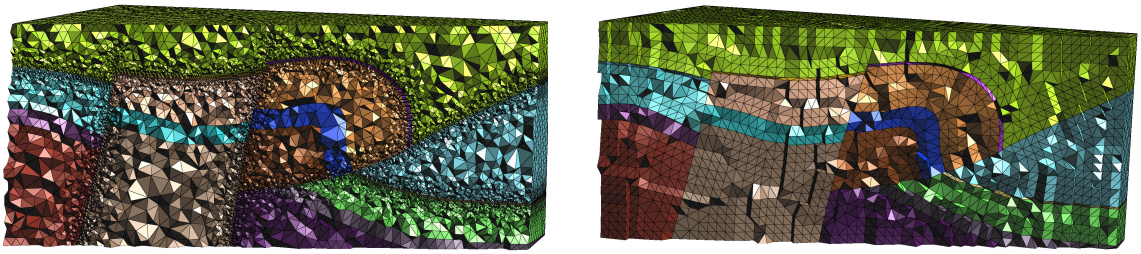


(a) Maillage original

(b) Maillage subdivisé dont les mailles non intersectées par la faille sont transparentes

FIGURE 6.13 – Subdivision des mailles au proche des failles

des horizons et des failles sert de support à la création des mailles. Les mailles sont créées suivant certains paramètres. On peut notamment spécifier la forme la plus courante des mailles à générer : pyramides, tétraèdre, hexaèdre. Le logiciel TetGen [Si15] par exemple permet de générer des maillages composés de tétraèdres, en fonctions de différentes contraintes. Le logiciel utilise notamment des triangulations de Delaunay [Del34] pour réaliser des mailles 3D.



(a) Maillage dont la taille des éléments est contrôlée au proche des failles

(b) Maillage généré avec des champs de direction

FIGURE 6.14 – Maillages tétrahèdriques générés par une méthode proposée dans [Bot16]

Dans sa thèse [Bot16], Arnaud Botella propose une méthode qui permet de créer des maillages non structurés. Avec cette méthode il est possible de définir la taille des mailles en fonction de critères, par exemple la proximité aux failles. Ainsi il est possible de générer des mailles plus fines au niveau de failles, et plus grossières dans les zones non faillées comme illustré en figure 6.14(a). Cependant, ce type de maillage ne satisfait pas les contraintes que nous avons expliquer plus avant. En effet, même si les mailles peuvent être plus petites au niveau des failles, les mailles ne suivent pas de direction particulière et sont assez désordonnées. Nous avons vu précédemment que pour améliorer la simulation de migration de fluide, les mailles doivent globalement suivre la direction d'écoulement. Pour l'améliorer, Botella propose également la possibilité d'utiliser des champs de directions afin de générer des maillages plus concordants avec les failles et plus réguliers comme illustré en figure 6.14(b). Ici les mailles suivent des lignes de directions qui tiennent compte des déformations des failles. Les mailles sont réparties de façon plus régulières.

En jouant sur les paramètres, il serait certainement possible de faire en sorte que les mailles suivent une direction d'écoulement en plus d'être concordant avec les failles. Il faudrait modifier la génération des champs de direction pour tenir compte de la direction d'écoulement. Botella a également participé au développement du logiciel RINGMesh [PBM+15, PBM+17]. RINGMesh inclut des structures de données qui permettent de manipuler des modèles géologiques, et peut être utilisé pour développer des algorithmes de maillages. Il permet également de vérifier la validité de modèles structuraux en détectant certaines dégénérescences dans les maillages par exemple.

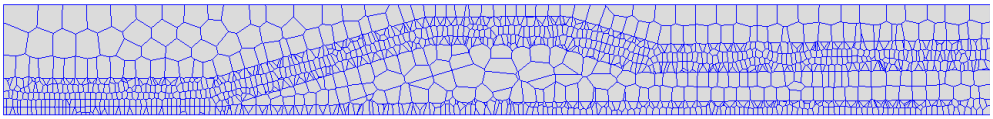


FIGURE 6.15 – Maillage issue de [Mer13] générée avec des diagrammes de Voronoï

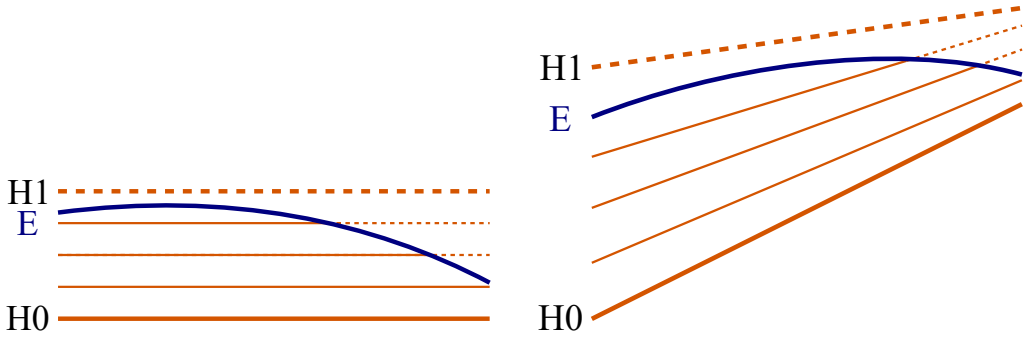
Une autre approche étudiée par Romain Merland [Mer13] consiste à utiliser des diagrammes de Voronoï pour générer des maillages non réguliers, concordants avec les failles. Les diagrammes de Voronoï [DFG99] représentent un ensemble de points définis par une fonction de distance. Cette méthode présente plusieurs fonctions qui permettent de satisfaire différentes contraintes, comme la direction d'écoulement, la taille des mailles, *etc.* Ces fonctions sont utilisées avec certains coefficients pour réaliser un maillage qui permet de prendre en compte les différentes contraintes parfois contradictoires.

On peut voir en figure 6.15 un maillage vu en coupe, créé avec un diagramme de Voronoï. Ici les unités très perméables ont été maillées plus finement que les autres.

6.2.2 Espace de dépôt

Pour prendre en compte des déformations du sous-sol comme les compressions que nous avons vues plus tôt, une technique courante est celle de l'utilisation d'un nouvel espace 3D [Whe58] : l'espace déplié, ou espace de dépôt. Cet espace correspond à la superposition des différents horizons dont la géométrie a été modifiée pour la rendre cohérente avec leur position supposée lors du dépôt sédimentaire des unités. Dans cet espace la coordonnée z ne correspond pas à une hauteur géométrique mais à un temps [HBB+10]. Comme les horizons sont composés de points qui se sont déposés à la même époque, dans l'espace de dépôt, tous les sommets d'un même horizon sont positionnés sur un même plan à un z donné. Cet axe représentant un temps, les horizons plus anciens sont positionnés à un z inférieurs à ceux qui leurs sont postérieurs. Le maillage des horizons dans cet espace est calculé pour que les sommets qui forment les piliers du maillage final, soient alignés sur l'axe z. La méthode de [Mal04], propose une association entre les éléments dans les espaces de dépôt et ceux observés dans l'espace actuel. Un espace de dépôt est calculé par des champs de directions. Dans l'espace de dépôt (ou espace paramétrique pour Mallet), les lignes qui relient les horizons entre eux sont parfaitement verticales, et leur image dans l'espace actuel suivent les déformations du sous-sol. Cette méthode permet de lier les deux espaces en ayant une fonction de transformation qui permet de passer d'un espace à l'autre.

L'utilisation de cet espace présente néanmoins quelques subtilités. En effet, dans l'espace de dépôt, les couches géologiques sont censées être planes et continues, comme au moment de leur dépôt. Cependant, les érosions enlèvent de la matière. Ainsi, la matière érodée ne peut être créée dans l'espace de dépôt à partir des mesures dans l'espace actuel. Certaines méthodes de maillage ne nécessitent pas d'avoir connaissance des volumes occupés par cette matière dans



(a) Espace de dépôt (b) Espace actuel

FIGURE 6.16 – Illustration d'une érosion dans les deux espaces géométriques

l'espace de dépôt, on peut alors laisser ces espaces vides. Mais certains évènements géologiques nécessitent de s'appuyer sur ces données manquantes pour réaliser un maillage 3D de bonne qualité. Les parties érodées sont alors créées par interpolation dans l'espace de dépôt, comme illustré avec les lignes pointillées, sur la figure 6.16. $H1$ est l'horizon qui a été supprimé par l'érosion E. Sa reconstruction est indispensable pour le maillage 3D des couches stratigraphiques proportionnelles entre $H0$ et $H1$, qui subsistent seulement entre $H0$ et l'érosion E à l'époque actuelle.

Chapitre 7

Méthode proposée

Dans le cadre de cette thèse nous avons travaillé en collaboration avec l'entreprise Géosiris. Elle collabore avec des entreprises pétrolières comme TOTAL, et le Bureau de Recherche Géologique et Minière par exemple, afin de mettre en place des solutions de modélisation et d'étude du sous-sol.

Nous avons associé le savoir-faire de Geosiris pour la reconstruction 2D des surfaces géologiques et notre maîtrise en topologie pour mettre en place un processus de modélisation du sous-sol.

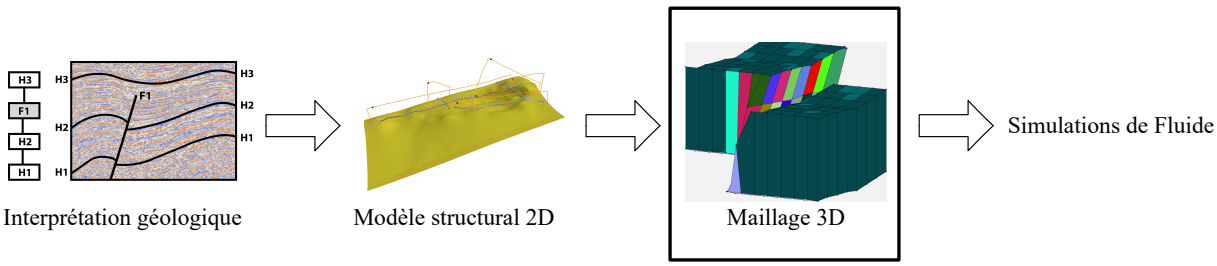


FIGURE 7.1 – Flux d'activité

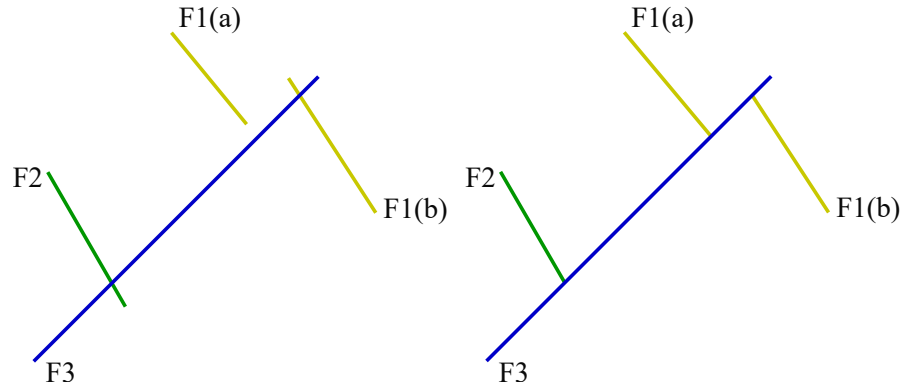
Ce processus est découpé en plusieurs étapes comme illustré en figure 7.1. L'interprétation géologique décrit la phase d'analyse des données collectées par les géologues, comme l'identification et l'âge des évènements géologiques. La création du modèle structural est réalisé par l'entreprise Géosiris avec leur logiciel GeoTopoModeler [Gé]. Cette étape de création des surfaces 2D à partir des données interprétées est décrite dans la section suivante. L'étape de maillage 3D est le cœur de la partie applicative de cette thèse. Elle consiste à construire les mailles 3D à partir des surfaces (horizons, failles, érosions) issues du modèle structural 2D. Cette partie est détaillée dans les sections suivantes.

7.1 Modélisation structurale 2D

La modélisation structurale consiste à créer les surfaces d'horizons, de failles et d'érosions, à partir des données sismiques et des interprétations des géologues. Pour ce faire, la première étape consiste à créer des surfaces (sous la forme de nuages de points ou surfaces paramétriques) à partir des données de mesures, puis à vérifier si la position des failles, des horizons et des érosions est correcte selon les règles de cohérence géologique. On ajoute ensuite sur les surfaces d'horizon les lignes de leurs intersections avec les failles. On distingue chaque côté de la ligne de

faille (*foot-wall* et *hanging-wall*), que l'on appelle des lèvres de faille. Dans un second temps les lèvres de failles sont ouvertes afin de représenter le mouvement de glissement des failles.

Géosiris a développé un logiciel de modélisation structurale appelé GeoTopoModeler. Il utilise les interprétations géologiques et les données (sous forme de nuages de points), stockées dans un format normalisé et spécialisé pour la géologie appelé ResQML, afin de reconstruire le maillage 2D des horizons et des failles. Le GeoTopoModeler utilise un modèle à base topologique de CGoGN qui propose une implantation des cartes combinatoires. L'interface graphique du GeoTopoModeler s'appuie le framework SCHNApps [sch16] également développé dans CGoGN. Le principe de fonctionnement du GeoTopoModeler est illustré en figure 7.3. Les données d'entrée du GeoTopoModeler sont dans le format ResQML V2.0.1, qui contient des données brutes interprétées sous forme de nuages de points (figure 7.3(a)) et des connaissances géologiques formalisées sur les structures du sous-sol.



(a) Trois failles présentant une intersection mutuelle (incorrect)

(b) Réseau de faille correct

FIGURE 7.2 – Réseau de faille avant et après

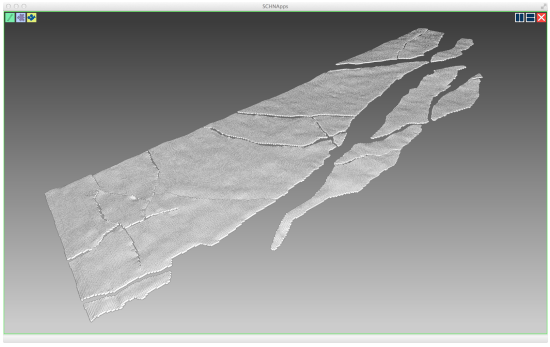
La première étape de la modélisation structurale consiste en la création de surfaces paramétriques à partir des nuages de points des données brutes interprétées [SIN13] comme illustré sur la figure 7.3(b). Mais cette reconstruction réalisée exclusivement à partir des données de mesure mène à des configurations irréalistes en raison des imprécisions trop grandes sur la position des surfaces, en particulier à l'approche des failles qui peuvent bruitent les mesures. La suite de la reconstruction structurale a donc pour but de corriger ces premières surfaces afin d'obtenir une reconstruction réaliste.

L'étape suivante consiste à calculer les intersections horizons/failles (cf. figure 7.3(c)) et failles/failles, puis à les corriger (cf. figure 7.3(d)). En effet, la reconstruction des surfaces doit suivre les contraintes géologiques, issues de l'interprétation des géologues. Par exemple, dans un réseau de failles, il est rare que deux failles se croisent [AFMASWS00]. La faille la plus ancienne s'arrête souvent sur la plus récente. Prenons l'exemple de la figure 7.2(a) qui illustre un réseau de failles dont l'ordre d'apparition historique est l'ordre de numérotation. Sur la figure, $F1$ a été coupée en deux par $F3$. Or $F1(a)$ ne touche pas la ligne de $F3$ et $F1(b)$ la traverse. Cette configuration est incohérente et doit être corrigée. De même, $F2$ coupe également $F3$ mais la ligne est très courte d'un des côtés de $F3$, de plus cette dernière n'a pas subit de cassure comme $F1$ et est en un seul morceau. On peut donc supposer supposer que la ligne de $F2$ doit également être corrigée. La figure 7.2(b) montre le modèle corrigé.

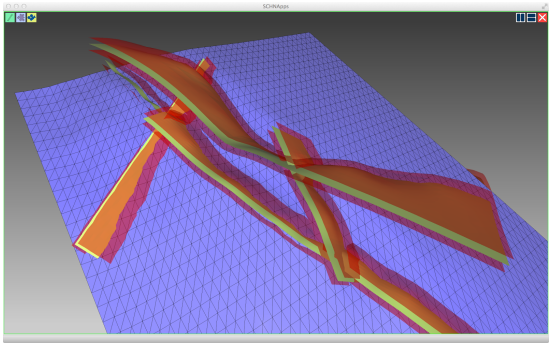
Le GeoTopoModeler propose des outils pour définir les différents contacts entre les éléments

du modèle structural (horizon/horizon et horizon/faille) et calculer les intersections nécessaires, pour enlever les parties des surfaces ne faisant pas partie du modèle, comme dans l'exemple précédent.

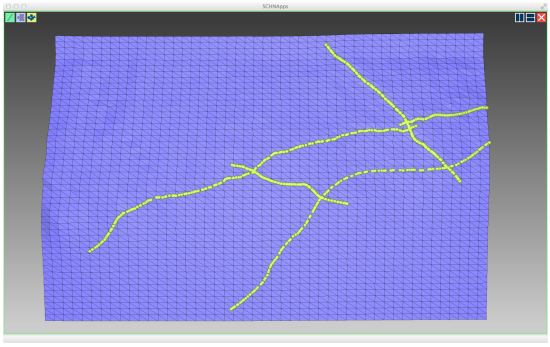
Pour ce faire, les données non consistantes à proximité des failles sont enlevées. Les connaissances géologiques permettent de déterminer si certaines portions des surfaces sont incohérentes, afin de les supprimer, comme illustré sur la figure 7.3(d).



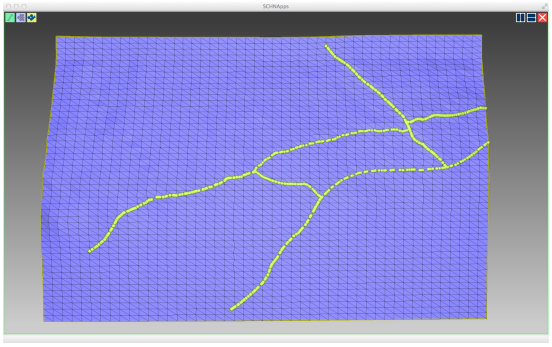
(a) Interprétation brute d'un horizon



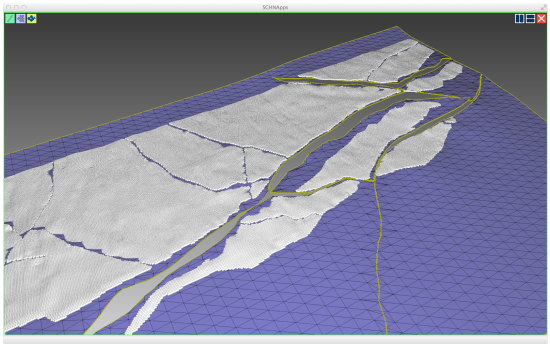
(b) Modélisation des surfaces d'horizons (bleu) et de failles (rouge)



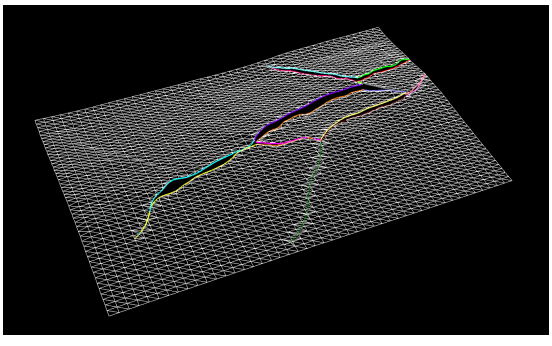
(c) Calcul d'intersection horizon/faille



(d) Prise en compte de l'âge des failles



(e) Ouverture des lèvres de faille



(f) Identification des contacts horizon/faille

FIGURE 7.3 – Modélisation structurale avec GeoTopoModeler

Une fois les failles corrigées, c'est au tour des horizons de l'être. La dernière étape consiste à ouvrir les lèvres de failles en tenant compte de la géométrie locale, comme illustré sur la figure 7.3(e). L'ouverture des lèvres permet de représenter les déformations provoquées par les failles. Les arêtes des lignes de failles sont étiquetées comme étant des *lèvres de failles*, et portent

une étiquette correspondant à la failles dont elles sont issues ainsi qu'au coté de l'intersection qu'elles représentent (*hanging-wall* ou *foot-wall*). Chaque ligne colorée de la figure 7.3(f) représente une lèvre. La convention d'étiquetage permet de connaître :

— la faille à laquelle sont attachées les lèvres ;
— le lien associant une lèvre du *foot-wall* à celle de son *hanging-wall* correspondant ;
— les liens qui unissent les lèvres de failles entre les différents horizons coupés par une même faille.

Le résultat produit par GeoTopoModeler est un modèle surfacique 2D comprenant un ensemble de surfaces d'horizons auxquelles sont ajoutées des informations sous forme d'étiquettes. Ce résultat est le point d'entrée de notre modélisation 3D. Grâce aux informations géométriques et physiques, issues de l'interprétation des géologues, ajoutées au modèle, il est possible de mettre en place une méthode de maillage qui tient compte de ces informations.

7.2 Objets et définition des plongements

Pour intégrer et traiter correctement les données réelles dans notre modèle géométrique à base topologique, nous y avons défini différents plongements. Ces plongements permettent de conserver toutes les informations nécessaires à la manipulation des données. Comme nous l'avons vu précédemment, l'ajout de nouveaux plongements est très facile dans Jerboa. Nous avons donc pu aisément définir un nombre conséquent de plongements pour stocker différentes informations. Voici la liste de ces plongements.

Positions géométriques : Le plus important d'entre eux est évidement la position géométrique. La particularité de ce domaine d'application est qu'il existe deux espaces géométriques. En effet, les géologues ont besoin pour travailler, de reconstruire le sous-sol à l'état dans lequel il était lors du dépôt des couches géologiques. Au moment du dépôt, les sédiments se déposent uniformément sur le sol, tous les points d'un même horizon sont donc déposés à la même époque. La représentation de l'espace dit de *dépôt* (ou à plat), est différente de l'espace *actuel* (ou plié). Dans l'espace de dépôt, les surfaces sont représentées en 2 dimensions (x et y). L'axe vertical du repère (z) ne représente pas la hauteur géométrique de la surface, mais l'époque à laquelle elle s'est déposée. Ainsi, l'axe z est une échelle de temps.

Les points des horizons ont deux positions géométriques, chacune associée à l'un des deux espaces. Grâce à la facilité d'ajout des plongements dans Jerboa, nous avons simplement défini 2 plongements différents représentant les positions géométriques dans chacun des deux espaces. Le plongement `posAplat` représente la position des points dans l'espace de dépôt, et `posPlie` représente la position des points dans l'espace actuel, l'espace issu des mesures du sous-sol. Ce double plongement géométrique d'un même maillage rend immédiat le passage d'un espace à l'autre, ce qui simplifie et accélère grandement la reconstruction du maillage 3D. En effet, comme nous l'avons vu dans le chapitre précédent le maillage 3D à construire dans l'espace actuel est issu d'une grille 3D dans l'espace de dépôt qui a été modifiée par les différents évènements géologiques. Notons que nous ne disposons pas de la positions des failles dans l'espace de dépôt. Les seules informations dont nous disposons sont leurs traces sur les horizons : les lèvres de failles.

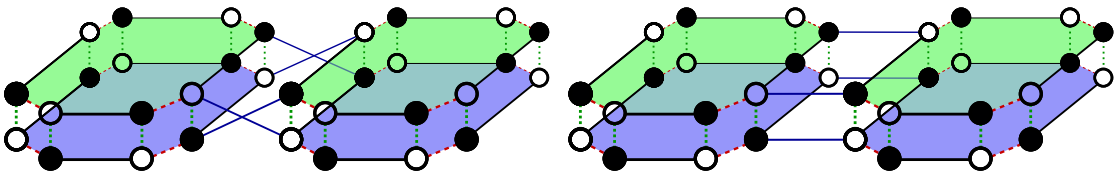
Étiquettes : Afin d'être en mesure d'identifier les différents éléments du modèle, nous avons mis en place différentes étiquettes. Les informations dont nous avons besoin sont les noms des surfaces, leur type (horizons, failles, érosions, *etc.*), et éventuellement le SUS auquel elles appartiennent.

En raison des différentes manipulations que nous réalisons sur les éléments du modèle, nous avons choisi de définir les différents étiquettes sur l'orbite face. Ainsi, même si les différents éléments sont liés entre eux, ou découpés, les étiquettes restent correctes, ce qui n'aurait pas été le cas si nous avions choisi la surface comme orbite support. Pour identifier le *Stratigraphic Unit Stack* auquel appartiennent les surfaces, nous avons défini le plongement `unityLabel` qui est une simple chaîne de caractères. Nous avons défini un plongement qui permet d'identifier les surfaces, nommé `geologyKind`. Il comprend une chaîne de caractère pour nommer la surface et un type énuméré permettant de déterminer le type de surface. Ce type énuméré permet d'identifier des horizons, des failles, des érosions, ou un maillage quelconque pour les éléments créés lors de la phase de maillage 3D.

Lèvres de faille : Un élément important du modèle est la présence de lèvres de faille. Ces lèvres de faille correspondent aux arêtes sur les horizons qui sont issues de la découpe de ces derniers par les failles. L'identification de ces éléments est indispensable pour réaliser le maillage 3D. Les arêtes de lèvre de faille n'ont pas de voisin par α_2. Cependant, cette particularité ne leur est pas propre et ne peut donc pas les caractériser. Les différentes découpes du maillage pour des calculs, ou même les arêtes du bord du maillage, ont également cette particularité. Nous avons donc ajouté un plongement appelé `faultLips` sur l'orbite $\langle \alpha_0, \alpha_3 \rangle$ qui représente une arête de face. Ce plongement contient un booléen pour définir si l'arête est une lèvre de faille, ainsi qu'une étiquette pour identifier la faille dont elle est issue.

Couleur : Pour facilement identifier les éléments lors de visualisation, nous avons ajouté un plongement de couleur. Ce plongement n'a aucune importance dans le calcul du maillage mais facilite la visualisation.

Orientation : Comme nous l'avons vu dans les premiers chapitres, les G-cartes ne sont pas orientées. L'orientation est cependant très importante lorsque l'on réalise des calculs d'intersection dans l'espace, pour relier les éléments entre eux. En effet, sans orientation relier deux faces le long d'une arête commune peut faire apparaître des torsions dans le maillage.



(a) Liaisons avec torsions (b) Liaisons correctes

FIGURE 7.4 – Liaisons α_2 entre deux deux faces

Prenons l'exemple de la figure 7.4, où l'orientation des brins est marquée par un fond transparent ou un fond noir. L'objet est correctement orienté, si chaque brin d'une orientation donnée (fond noir ou fond transparent) est adjacent uniquement avec des brins d'orientation opposée. Sur la figure 7.4(a), on observe une liaison incorrecte entre deux faces, voisines dans l'espace. La figure 7.4(b) montre la liaison correcte pour cette configuration. Sans orientation, il est très difficile de déterminer quelles demi-faces doivent être liées entre elles. Avec la présence de l'orientation, on peut aisément déterminer quelles demi-faces doivent être reliées, en vérifiant que leurs orientations sont inversées.

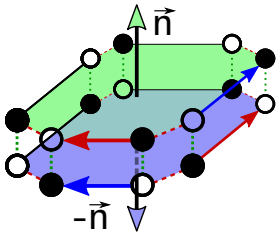


FIGURE 7.5 – Représentation des normales de chacune des demi-faces

L'orientation permet également de calculer la normale de chaque face de volume, comme le montre la figure 7.5. La normale peut être calculée comme le produit vectoriel entre deux arêtes distinctes, incidentes à un même sommet de cette face. L'orientation permet de définir le sens de cette normale. Dans notre exemple, la normale à chaque face de volume est calculée comme le produit vectoriel entre le vecteur directeur partant de la position géométrique d'un brin b transparent, vers un brin noir le long d'une liaison α_0 (flèches bleues), avec le vecteur directeur partant du voisin (noir) par α_1 de b vers son voisin par α_0 (flèches rouges). Ainsi les arêtes sont toujours choisies dans le même sens (ici le transparent en premier, le noir en second), et les normales à la face de volume ont tous la même direction, quel que soit le brin de référence utilisé.

7.3 Étapes de l'algorithme de maillage 3D

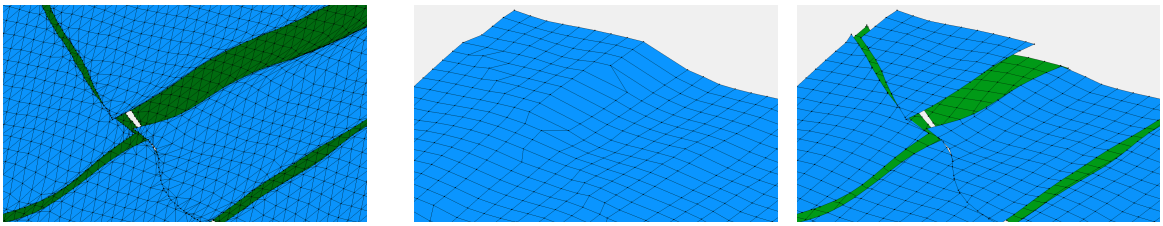
Cette section présente la méthode que nous avons mis en place pour réaliser le maillage 3D du sous-sol. Toutes les étapes de cette méthode ont été réalisées avec Jerboa. Notre objectif est de réaliser un maillage qui répond au mieux aux besoins décrit dans la section 6.1, page 122. Pour réaliser le maillage, nous nous appuyons sur les données structurales reconstruites dans le GeoTopoModeler. Ces données contiennent les surfaces d'horizons plongées dans l'espace actuel mais également dans l'espace de dépôt (leur position dans cet espace est calculée grâce au logiciel APLAT [Bor06]). Elles comprennent également les surfaces d'érosion et de faille (plongées dans l'espace actuel). Nous commençons par re-mailler ces surfaces afin de satisfaire nos contraintes de maillage, comme nous allons le voir dans la section suivante. Une fois les surfaces maillées, nous procédons à la construction des piliers sur chaque unité. Nous terminons ensuite par l'ajout des surfaces intermédiaires (ou *layers*) qui représentent des sous-couches stratigraphiques dans les unités sédimentaires.

7.3.1 Maillage des horizons

Bien que les surfaces structurales issues du GeoTopoModeleur soient des surfaces maillées, ces maillages ne peuvent être utilisés pour construire le maillage 3D. En effet, ces maillages ont été produits pour réaliser les ouvertures de failles, et ne peuvent donc pas respecter les différentes contraintes du maillage 3D (voir la section 6.1, page 122). En particulier, les mailles utilisées par le GeoTopoModeleur sont de tailles inférieures à celles attendues dans le maillage 3D final.

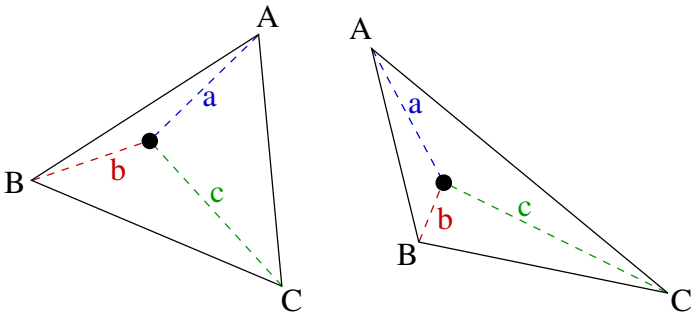
Pour être en mesure de créer les piliers à l'intérieur de chaque unité sédimentaire, notre méthode s'appuie sur un maillage 2D commun qui est projeté sur toutes les surfaces, à commencer par les horizons. Cette méthode permet de regrouper les sommets de toutes les surfaces, qui appartiennent à un même pilier. En effet, un sommet du maillage 2D sera projeté successivement sur toutes les surfaces et définira les points de passage du pilier correspondant. La création d'un

maillage 2D qui satisfait toutes les contraintes décrites dans la section 6.1, page 122 n'est pas l'objectif de cette thèse. Nous avons donc défini une méthode de maillage 3D qui fonctionne avec n'importe quel maillage 2D. Nous avons testé notre méthode sur une grille régulière, mais son principe de fonctionnement est indépendant de la topologie et de la géométrie du maillage 2D.



(a) Données réelles (b) Projection du maillage 2D (c) Insertion des lèvres de faille

FIGURE 7.6 – Projection du maillage 2D sur les horizons du modèle



(a) Espace de dépôt (b) Espace actuel

FIGURE 7.7 – Interpolation dans l'espace actuel, d'un point projeté dans l'espace de dépôt

La première étape de notre algorithme est de mailler les horizons avec le maillage 2D. Ce dernier est projeté sur chacun des horizons dans l'espace de dépôt, parallèlement à l'axe Z. Chaque sommet du maillage 2D est projeté sur l'horizon en lançant un rayon co-linéaire à l'axe Z (vertical). L'intersection de ce rayon avec l'horizon donne sa position projetée dans l'espace de dépôt.

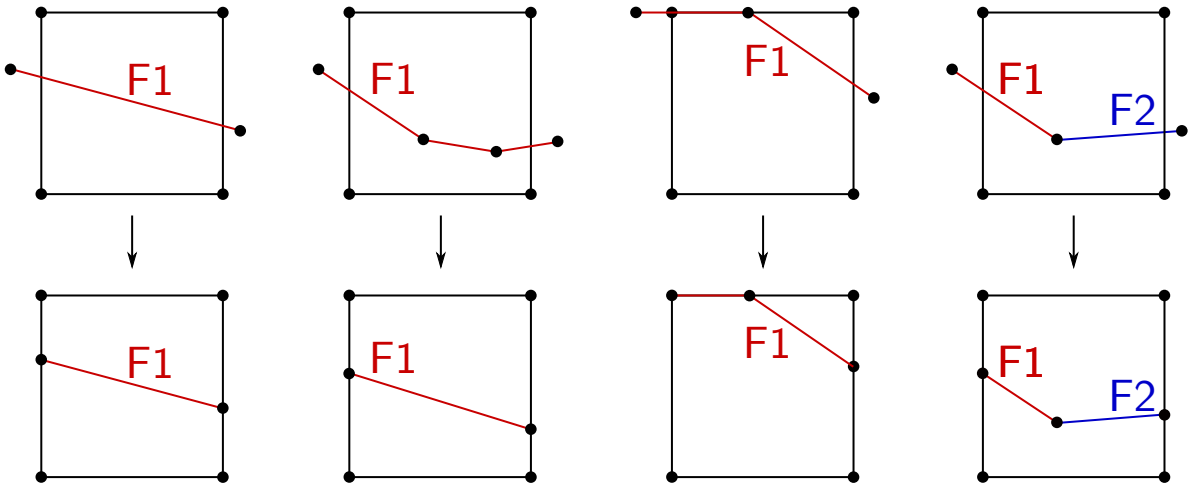
Comme illustré sur la figure 7.7, nous calculons la position dans l'espace actuel du point projeté, par rapport au triangle de la surface d'origine sur lequel il est projeté dans l'espace de dépôt. Sur la figure, A, B et C sont les points du triangle intersecté dans le maillage d'origine de l'horizon (vert ou bleu sur la figure 7.6). Le point noir dans le triangle est le point du maillage 2D, projeté sur l'horizon. Pour obtenir la position de ce point dans l'espace actuel, nous calculons ses coordonnées barycentriques a, b et c dans le triangle A, B , C dans l'espace de dépôt. Ces coefficients sont utilisés pour calculer sa position dans l'espace actuel, par rapport à A, B et C dans cet espace.

La figure 7.6(a) montre l'horizon d'origine issu de la reconstruction structurale. La figure 7.6(b) montre le résultat de la projection du maillage 2D sur l'horizon bleu. Une fois le maillage 2D projeté sur les horizons, nous coupons les maillages résultats en insérant les lèvres de failles, comme illustré sur la figure 7.6(c). Cette découpe du maillage 2D projeté par les lèvres de failles est réalisée comme suit. Tout d'abord, les lèvres de failles sont insérées dans l'espace de dépôt. À ce stade, les deux lèvres d'une même faille sont cousues l'une à l'autre. Elle ont donc la même

position dans l'espace actuel (temporairement), qui est par défaut calculée par interpolation selon la méthode précédente. Enfin, les deux lignes de failles sont décousues et alors leurs deux positions actuelles définitives sont établies.

Pour découper le maillage 2D projeté par les lignes de bord de la surface d'horizon et les lèvres de faille, nous procédons à un co-raffinement 2D. Dans sa thèse, Nicolas Guiard [Gui06] présente une méthode de construction de modèle géologique en utilisant un co-raffinement 3D. Notre cas est un peu plus simple car nous insérons les lèvres de faille dans l'espace de dépôt. Ainsi, les horizons et le maillage 2D sont sur un même plan, et ainsi les intersections sont faites en 2D. Cependant, nous avons quelques contraintes particulières qui diffèrent d'un co-raffinement classique comme nous allons le voir.

Pour couper le maillage 2D par les lèvres de faille nous cherchons toutes les intersections entre les arêtes de ces lignes et le maillage 2D. Pour chaque maille 2D, nous cherchons un point d'entrée et un de sortie d'une lèvre de faille, afin de découper la maille de part en part. Plus précisément nous coupons le maillage 2D par les lignes sans insérer leurs points de discrétisation d'origine présents sur la surface d'horizon de départ. Ces points alourdiraient le maillage final sans apporter de précision pertinente. Au contraire, les extrémités des lèvres de failles doivent être conservées pour préserver la cohérence des réseaux de failles. De même, les extrémités de chaque ligne de bord doivent être conservées pour ne pas tronquer les coins de l'horizon.



(a) Une arête coupe la maille

(b) N arêtes coupent la maille

(c) Lèvre partiellement confondue avec le maillage

(d) Deux lèvres de failles coupent une même maille

FIGURE 7.8 – Détail des cas d'intersection possible entre des lèvres de faille et une maille

La découpe s'effectue en deux étapes. La première étape consiste à insérer les sommets et arêtes des lignes de découpe dans le maillage 2D. La deuxième étape est la découpe proprement dite le long des lignes préalablement insérées. Nous avons identifié plusieurs configurations possibles d'intersections avec une maille 2D :

a) si une unique arête de lèvre de faille ou du bord traverse la maille, cette arête coupe la maille (cf. figure 7.8(a)) ;
b) si plusieurs arêtes coupent successivement une maille, la maille est coupée par une unique arête dont les extrémités sont placées aux intersections de la ligne avec les arêtes de la maille (cf. figure 7.8(b)) ;
c) certaines arêtes à insérer peuvent être confondues avec le maillage 2D (cf. figure 7.8(c)).

Dans ce cas, nous n'insérons pas de nouvelles arêtes dans le maillage, mais nous marquons les arêtes existantes afin de pouvoir les identifier pour les découdre lors de l'étape suivante ;

d) nous avons vu que les points de discrétisation des lèvres de faille ne sont pas insérés. Cependant, lorsque deux arêtes d'identités différentes (deux lèvres de deux failles différentes par exemple) coupent une même maille, le point d'intersection est ajouté au maillage. Comme nous l'avons dit ci-dessus, la préservation de ces points assure la précision du maillage final. Une illustration de ce cas est montré en figure 7.8(d).

Pour réaliser ces insertions, nous avons écrit plusieurs règles Jerboa. La première consiste à découper une arête de maille pour insérer un sommet. Cette règle a déjà été présentée plus tôt dans le manuscrit, mais ici elle a été enrichie pour intégrer tous les plongements géologiques. Elle est illustrée en figure 7.9.

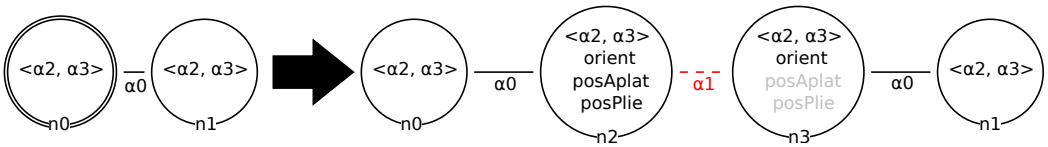


FIGURE 7.9 – Règle Jerboa `InsertionSommet`, d'insertion d'un nouveau sommet sur une arête de maille

Dans cette règle, l'expression d'orientation de `n2` est définie par :

```
return new @ebd<orient>(!n0.orient);
```

et pour `n3` :

```
return new @ebd<orient>(!n1.orient);.
```

Ces expressions permettent de donner une orientation correcte aux nouveaux brins créés par la règle. Les nouveaux brins `n2` et `n3` prennent l'orientation inverse de celle de leurs voisins, respectivement `n0` et `n1`. Les expressions du plongement d'orientation étant toujours très similaires, nous ne les détaillerons plus dans la suite du document. Pour définir la position géométrique, la règle prend en paramètre deux positions géométriques : `positionDepot` et `positionActuelle` pour déterminer la position du nouveau sommet dans les deux espaces géométriques. On a donc comme expression du plongement `posAplat` :

```
return new @ebd<posAplat>(positionDepot);
```

et pour le plongement `posPlie` :

```
return new @ebd<posPlie>(positionActuelle);
```

Ainsi, pour chacun des deux espaces, une nouvelle instance du plongement est créée à la position géométrique donnée en paramètre.

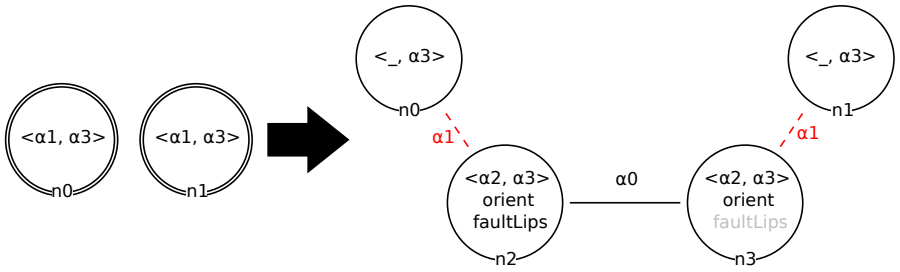
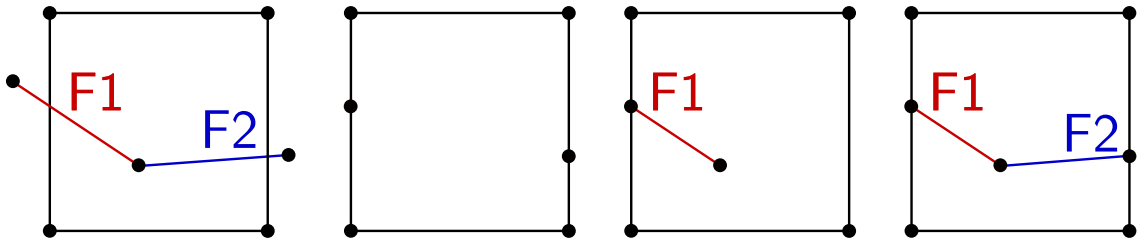


FIGURE 7.10 – Règle Jerboa `InsertionArete`, d'insertion d'une arête entre deux sommets existants d'une même maille

Une fois les sommets insérés, nous insérons les arêtes. La règle Jerboa associée à cette opération dans les cas *a*) et *b*) de la figure 7.8 est illustrée en figure 7.10. Comme une arête est créée par la règle, il faut définir son plongement `faultLips` qui identifie la lèvre de faille, ou la ligne de bord. Comme pour la règle précédente, la règle demande en paramètre une instance de ce plongement (nommée `levreDeFaille`) pour étiqueter correctement cette nouvelle arête. Ainsi l'expression du plongement `faultLips` est la suivante :

```
return new @ebd<faultLips>(levreDeFaille);
```

Dans le cas *c*) de la figure 7.8(c), l'arête du maillage 2D confondue avec la lèvre de faille est simplement étiquetée comme une lèvre de faille.



(a) Lèvre de faille à projeter sur la faille

(b) Insertion des sommets aux intersections

(c) Insertion d'une arête pendante pour l'arête $F1$

(d) Insertion d'une arête pour la faille $F2$

FIGURE 7.11 – Étapes d'insertions d'arêtes pendantes pour couper une face dans le cas *d*)

Dans le cas *d*) de la figure 7.8(d), où plusieurs arêtes doivent être ajoutées comme sur la figure 7.11(a), nous ajoutons les sommets aux intersections, puis (cf .figure 7.11(b)) nous créons une arête pendante à la position du sommet intermédiaire, comme illustré sur la figure 7.11(c). Pour terminer la découpe de la face, nous créons l'arête entre le sommet pendant et le sommet issue de l'intersection de la maille avec $F2$.

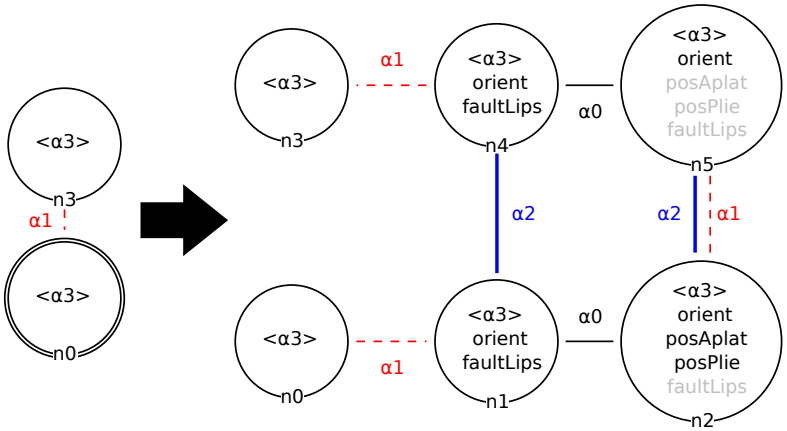
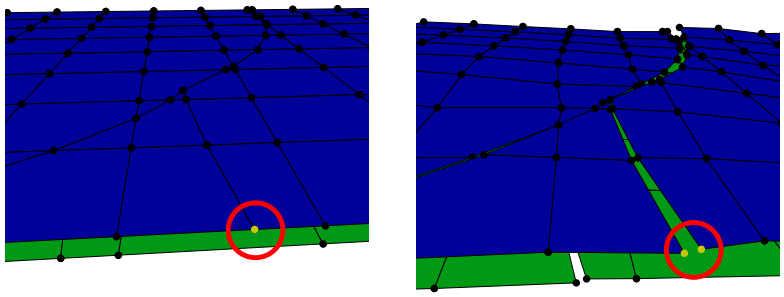


FIGURE 7.12 – Règle Jerboa `CreationAretePendante` d'insertion d'une arête pendante

La règle permettant de créer une arête pendante est illustrée en figure 7.12. Comme pour les règles précédentes, elle prend en paramètre un plongement de lèvre de faille, et une position géométrique dans chacun des deux espaces pour définir la position du sommet créé par `n2` et `n5`. Ainsi, pour réaliser l'insertion du cas *d*), on commence par appliquer la règle `InsertionSommet` pour insérer les deux nouveaux sommets que sont les intersections de la maille respectivement

avec la ligne de faille $F1$ et $F2$. On crée ensuite l'arête pendante à partir du sommet créé par l'intersection avec $F1$ avec la règle `CreationAretePendante`. On termine en appliquant la règle `InsertionArete` sur l'extrémité de l'arête pendante précédemment crée, et le sommet issue de l'intersection de la maille avec $F2$.

Concernant la découpe proprement dite, la principale difficulté est de replacer les lèvres de faille dans l'espace actuel. Car comme nous l'avons déjà vu, toutes les intersections ayant été réalisées dans l'espace de dépôt, les positions des nouveaux points dans l'espace actuel ont été calculées par interpolation. En effet, la position actuelle d'un sommet est nécessairement unique. Il faut donc "ré-ouvrir" les lèvres de failles en les repositionnant correctement dans l'espace actuel. Pour ce faire, il faut parcourir toutes les sommets incidents à une arête étiquetée comme lèvre de faille dans l'étape précédente, et trouver les lèvres de faille desquelles ils sont issus pour les replacer dans l'espace actuel.



(a) Espace de dépôt (b) Espace actuel

FIGURE 7.13 – Double position des points d'intersections du maillage 2D et des lèvres de faille

Pour chaque sommet on trouve deux arêtes de lèvre de faille sur les surfaces de départ, une pour le *hanging-wall* et une pour le *foot-wall*. Elles correspondent aux deux cotés d'une même faille. Après la dé-couture des arêtes des lèvre de failles insérées, chaque nouveau sommet ajouté a été coupé en deux comme on peut le voir avec les sommets orange sur la figure 7.13. Il faut déterminer pour chacune de ces deux parties le coté sur lequel elle se situe (*hanging wall* ou *foot wall*), afin de les placer correctement dans l'espace actuel.

Pour savoir à laquelle des deux arêtes de la surface de départ correspond chaque portion du sommet créé, nous utilisons un plan de découpe. Dans l'espace de dépôt, les surfaces de départ, tout comme les maillages 2D projetés, sont placées sur un plan d'équation $z = c$, avec c une constante. On peut donc séparer les deux cotés des lèvres de faille par un plan vertical, passant par la double arête de lèvre de faille dans l'espace de dépôt. Sur la figure 7.14, la double arête est représentée en jaune, comme le plan de découpe qui sépare les deux côtés (*foot-wall* et *hanging-wall*) du sommet inséré.

L'équation de ce plan peut être trouvée en utilisant un produit vectoriel entre une normale à une face incidente à la lèvre de faille, ici $\overrightarrow{n_{fw}}$ ou $\overrightarrow{n_{hw}}$, et la direction de celles-ci ($\overrightarrow{d_{lf}}$). Une fois le plan calculé, on peut aisément calculer de quel côté du plan se situe chaque face, en s'appuyant sur les directions $\overrightarrow{d_{fw}}$ et $\overrightarrow{d_{hw}}$.

Quand on utilise des données réelles pour réaliser des calculs complexes, on se confronte souvent à des problèmes de précision. Pour les éviter, nous avons utilisé un ε permettant de définir des intervalles de précision. On considère ainsi que deux positions sont équivalentes si

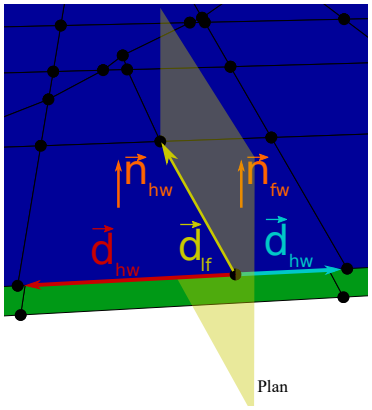


FIGURE 7.14 – Plan de découpe permettant de déterminer de quel côté se situe chaque sommet issue d'une même insertion de lèvre de faille

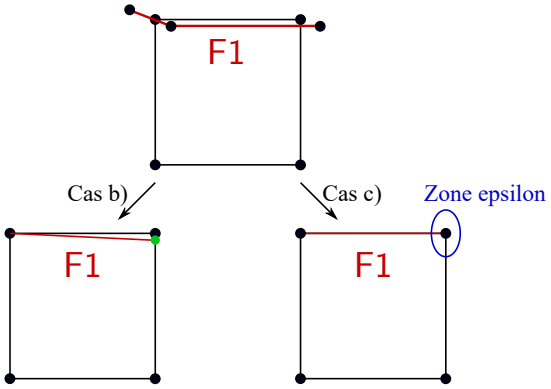
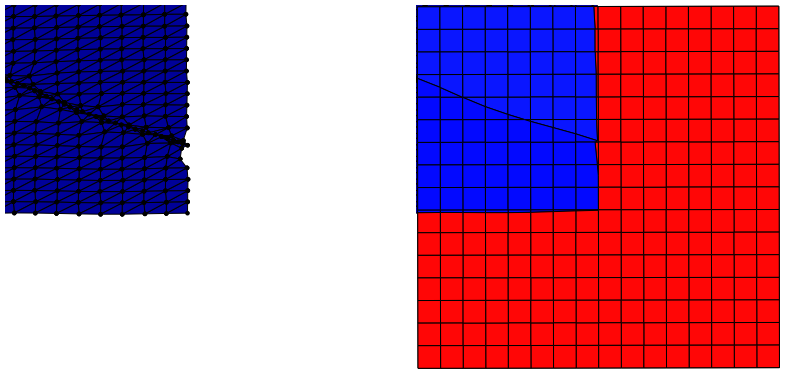


FIGURE 7.15 – Maillage de face à découper par une lèvre de faille (en rouge)

elles sont égales au ϵ près : si leur distance d vérifie la condition suivante : $0 \leq d \leq \epsilon$. Prenons l'exemple de la figure 7.15, qui illustre une lèvre de faille à insérer dans le maillage 2D. Deux solutions sont possibles dans ce cas. Soit l'intersection entre la maille et la lèvre de faille est assez éloignée du sommet B, et on est dans le cas a) décrit précédemment sur la figure 7.8(a), ou dans le cas b) illustré sur la figure 7.8(b). Dans ce cas, on insère le sommet à l'intersection avant d'insérer la lèvre de faille dans le maillage 2D. Soit l'intersection est trop proche de B (dans la zone ϵ), et on est dans le cas c) décrit précédemment sur la figure 7.8(c). Dans ce cas, on ne procède pas à l'insertion du sommet à l'intersection, on considère que la lèvre est confondue avec l'arête AB, et on étiquette cette dernière comme lèvre de faille.

Il est possible que le maillage 2D projeté dépasse du bord des surfaces d'horizons. Ainsi il convient de couper le maillage 2D projeté par les bords du maillage d'horizon d'origine, car les points hors de l'horizon n'ont pas de projection. La figure 7.16 montre la projection d'un maillage 2D sur des une surface d'érosion avec certaines mailles projetées dans le vide colorées en rouge. Pour découper le maillage 2D par les bords des surfaces d'horizon, nous procédons de façons similaire à l'insertion des lignes de faille.



(a) Données réelles

(b) Maillage 2D plaqué avec en rouge les mailles à supprimer

FIGURE 7.16 – Projection du maillage 2D sur des données réelles avec certaines mailles projetées hors de l'horizon

7.3.2 Maillage des failles

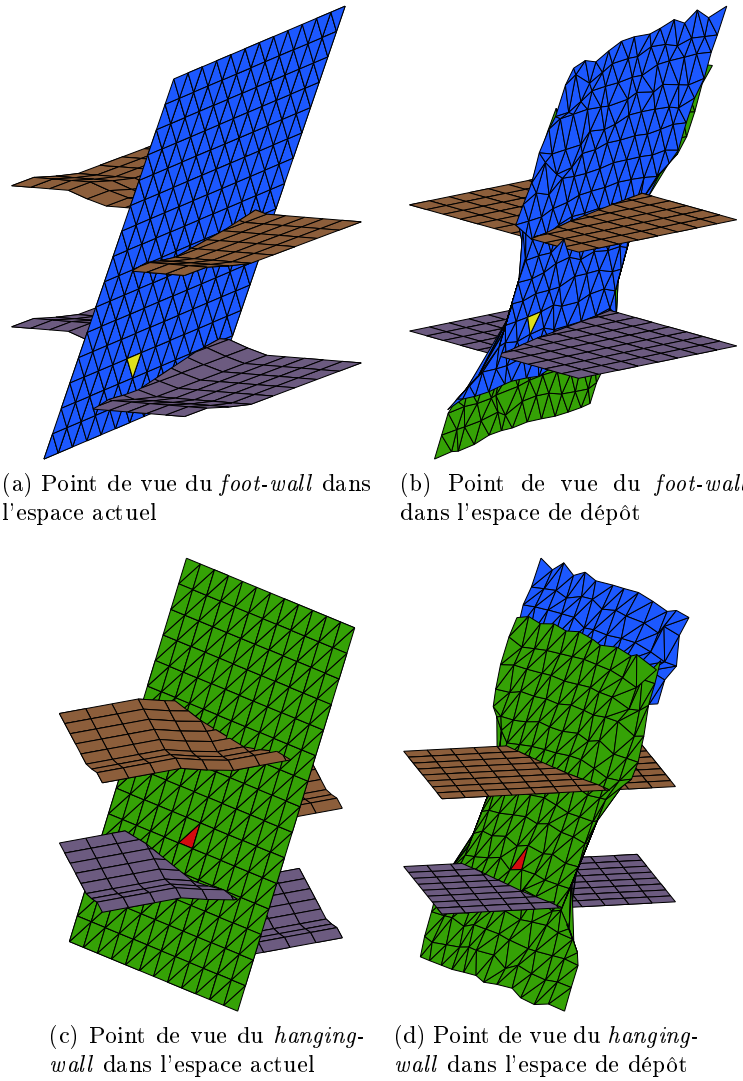
Une fois le maillage 2D projeté sur tous les horizons, nous le projetons également sur les failles. Comme nous l'avons dit plus tôt, il est projeté dans l'espace de dépôt. Or contrairement aux horizons, les failles n'ont pas de position dans l'espace de dépôt. En effet, cet espace représente le moment où les horizons se sont formés. Les failles étant postérieures aux dépôts, leur position dans cet espace n'a pas de sens géologique. Néanmoins, l'emplacement de la futur faille dans l'espace de dépôt peut être estimée. Ainsi, d'un point de vue géométrique, il est possible d'utiliser le même algorithme de projection du maillage 2D sur les failles que sur les horizons.

Pour estimer la position des failles dans l'espace de dépôt, le seul élément sur lequel nous pouvons nous appuyer est la position des lèvres de failles. En effet, ces dernières étant présentes sur les horizons, nous connaissons leur position dans les deux espaces. Une méthode simple pour calculer la position des failles dans l'espace de dépôt est d'utiliser une interpolation entre les lèvres du haut et du bas de chaque unité.

Il faut cependant noter une subtilité dans ce calcul. En effet, les failles présentent *deux* positions dans l'espace de dépôt. La première est celle qui la relie au *hanging-wall* et l'autre pour le *foot-wall*. Ces deux positions sont illustrées en figure 7.17 avec un côté bleu pour le *foot-wall* et un vert pour le *hanging-wall*. Cet exemple est issu de données que nous avons généré artificiellement (nous verrons dans le chapitre suivant comment nous avons procédé). Cette double position est due au fait que dans l'espace de dépôt, les éléments sont replacés tels qu'ils étaient lors de leur création et donc avant toute déformation, dont le glissement des failles. Sur les figures 7.17(a) et 7.17(b), nous avons coloré un triangle en jaune pour avoir un repère du passage d'un espace à l'autre. Nous avons fait de même avec un triangle rouge pour les figures 7.17(c) et 7.17(b).

Pour calculer la position des failles dans l'espace de dépôt, nous calculons chaque côté de la faille indépendamment, en nous servant des lignes de failles qui lui correspondent.

Cette méthode pose cependant plusieurs problèmes. Tout d'abord le calcul par interpolation de la position des failles dans l'espace de dépôt écrase les côtés de failles dans cet espace (cf. figure 7.17(b) et figure 7.17(d)). De plus, avec l'utilisation de données réelles, la position de la faille et celles des lèvres de failles ne sont pas nécessairement alignées, le maillage présente des chevauchements (cf. figure 7.18(a)) ou des trous (cf. figure 7.18(b)). En effet, lors de l'étape de



(a) Point de vue du *foot-wall* dans l'espace actuel

(b) Point de vue du *foot-wall* dans l'espace de dépôt

(c) Point de vue du *hanging-wall* dans l'espace actuel

(d) Point de vue du *hanging-wall* dans l'espace de dépôt

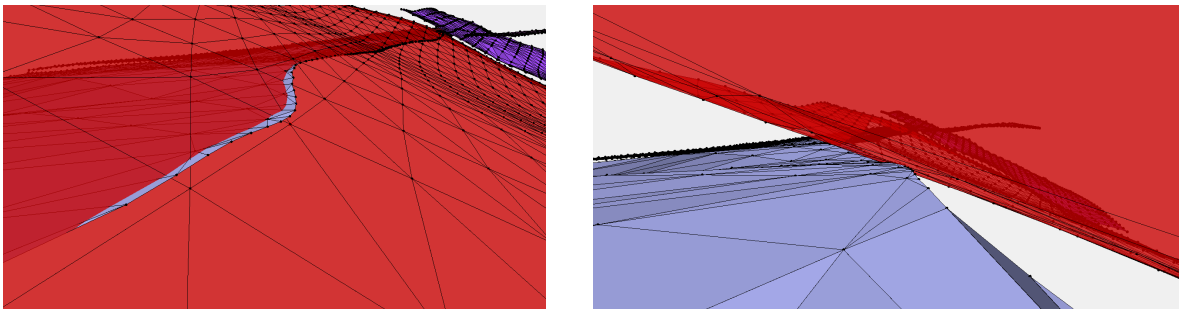
FIGURE 7.17 – Position des failles dans les deux espaces

reconstruction structurale, les arêtes des lèvres ont été calculées avant de prendre en compte le déplacement dû aux failles. Lors de l'ouverture des lèvres, ces dernières ont été déplacées le long des failles. Or comme les failles ne sont pas parfaitement planes, après ouverture, les lèvres ne sont pas situées exactement sur la surface de faille. Ce phénomène est aggravé par la discrétisation des surfaces de faille et d'horizons.

Pour éviter ces problèmes, nous avons décidé de reconstruire les surfaces de faille à partir des lèvres de faille des horizons. Pour chaque faille, nous générons deux surfaces, une pour le *hanging-wall* et une pour le *foot-wall*. Ces surfaces sont créées à partir des paires de lèvres (haut/bas) de chaque côté.

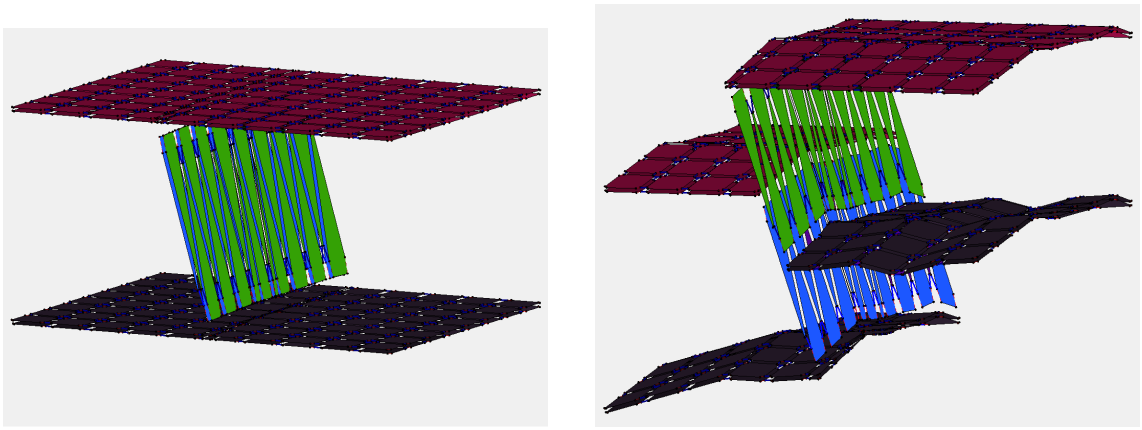
On peut voir sur la figure 7.19 un exemple de génération des deux côté d'une faille, visualisé en vue éclatée (vue de la carte généralisée), dans les deux espaces géométriques. Ici la surface verte correspond au côté *hanging-wall* de la faille et le bleu au *foot-wall*.

Une fois les surfaces de failles reconstruites, nous n'avons plus de problème géométrique, les lèvres de faille passent précisément par les failles. Nous pouvons donc projeter le maillage 2D



(a) Dépassement de l'horizons à travers la faille (b) Trou entre la ligne de faille et la faille

FIGURE 7.18 – Zoom sur les problème de position entre une faille (en rouge) et un horizon (en bleu)



(a) Espace de dépôt (b) Espace actuel

FIGURE 7.19 – Génération des deux côtés de faille

sur ces surfaces sans risque d'erreur géométrique aux abords des lèvres de faille. La projection du maillage 2D suit le même algorithme que pour les horizons.

Notons que nous ne parlons pas ici des érosions. Notre méthode ne les prends pour l'instant pas en compte. Mais il sera possible de les gérer de façon similaire aux failles en estimant leurs positions géométriques dans l'espace de dépôt avant de projeter le maillage 2D dessus.

7.3.3 Création des piliers 3D

La projection d'un même maillage 2D sur toutes les surfaces géologiques (horizon, failles, *etc.*) permet de facilement les lier afin de créer le maillage 3D. L'ensemble des sommets projetés issus d'un même sommet du maillage 2D peuvent être reliés pour former un pilier du maillage 3D.

La construction d'un maillage 3D à partir de deux maillages 2D de même topologie se fait très simplement à l'aide de la règle figure 7.20, comme le montre son application figure 7.21.

Mais de manière générale, les maillages 2D projetés dont nous disposons à l'issue des deux phases précédentes n'ont pas la même topologie. En effet, les lèvres de faille sont à des emplacements différents sur chaque horizon, et les failles ne permettent de projeter qu'une partie

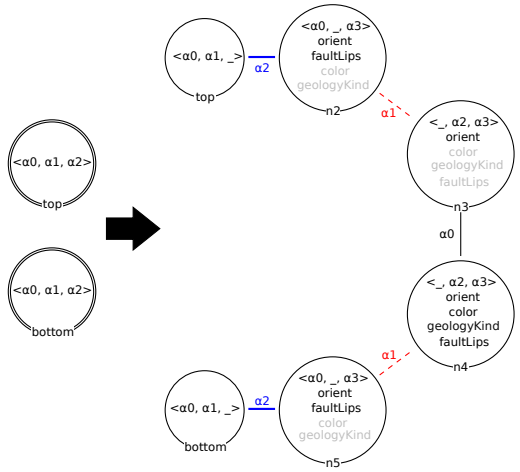
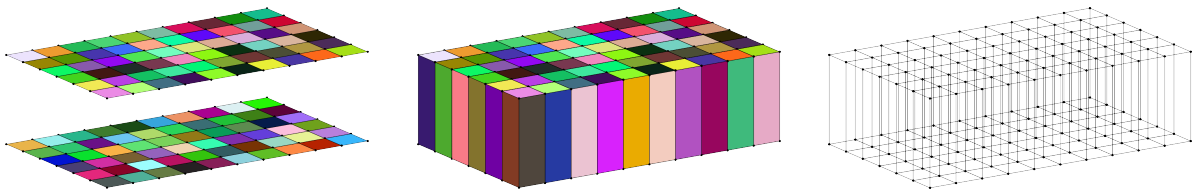


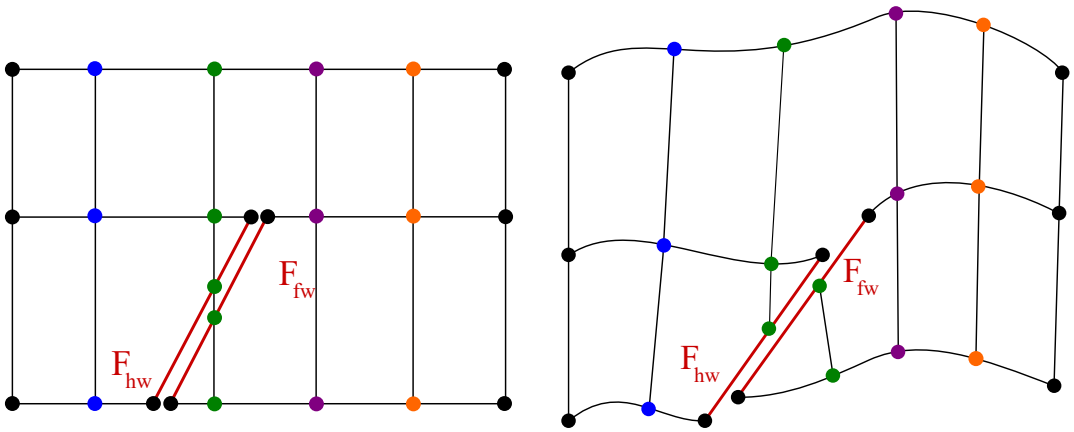
FIGURE 7.20 – Règle Jerboa réalisant un maillage 3D à partir de deux surfaces 2D



(a) Deux maillages 2D (b) Maillage 3D (c) Maillage 3D en vue fil de fer

FIGURE 7.21 – Application de la règle Jerboa de maillage, entre deux maillages 2D ayant une topologie identique

du maillage 2D de départ. Il n'est donc pas possible d'appliquer la règle ci-dessus. Ainsi, pour réaliser un maillage entre deux surfaces quelconques, il faut décomposer l'opération et procéder pilier par pilier.

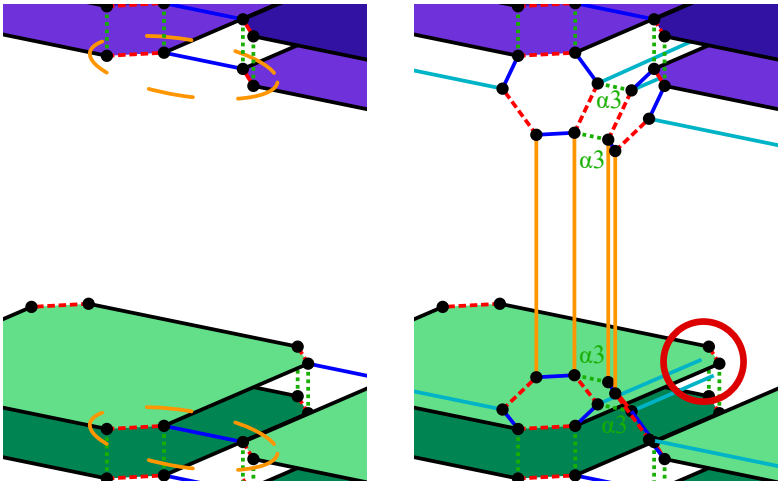


(a) Espace de dépôt (b) Espace actuel

FIGURE 7.22 – Représentation des correspondances entre les sommets issus du maillage 2D

Nous connaissons la correspondance entre les sommets projetés issus d'un même sommet

dans le maillage 2D de départ. Cette correspondance est stockée dans une structure annexe, qui rassemble tous ces sommets dans des listes. Ainsi chaque liste est triée en fonction de la position verticale (l'axe z) des sommets dans l'espace de dépôt. Prenons l'exemple de la figure 7.22. Sur cette figure, les points de même couleurs sont issus d'un même sommet du maillage 2D. Le tri de ces sommets selon leur coordonnée en z dans l'espace de dépôt permet de déterminer l'ordre de liaison. On observe cependant que pour les failles, il y a 2 sommets verts qui sont à la même position géométrique dans l'espace de dépôt (leur position a été éclatée sur la figure). Grâce aux informations géométriques et physiques du modèle nous savons si ils appartiennent au *foot-wall* ou au *hanging-wall*, ce qui permet de les trier. Comme ces sommets appartiennent à une même faille, il ne doivent cependant pas être lié l'un à l'autre. En effet, on observe sur la figure 7.22(b) que les deux sommets verts sur la faille ne sont pas à la même position géométrique dans l'espace actuel. Une fois tous les sommets triés, nous sommes en mesure de savoir comment les lier deux à deux.

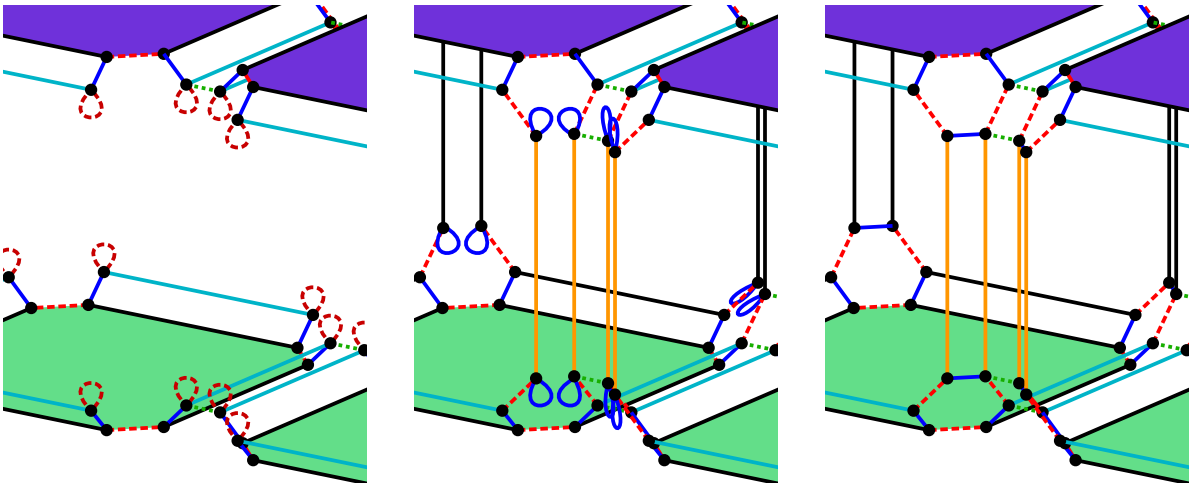


(a) Maillage initial (b) Création du pilier

FIGURE 7.23 – Création d'un pilier entre deux sommets associés

L'étape de liaison des sommets deux à deux ne peut pas être réalisée par une seule règle Jerboa qui relie un sommet à un autre en créant l'arête verticale des volumes intérieurs. Prenons l'exemple de la figure 7.23. Nous avons entouré en trait discontinu oranges les deux "sommets" à reliés sur la figure 7.23(a). L'orbite topologique est $\langle 1, 2 \rangle$. La liaison α_3 n'est pas prise en compte pour ces "sommets" car leurs voisin par α_3 sont les brins du même horizons mais appartiennent à l'unité supérieure (pour le bleu) ou inférieure (pour le vert). Pour relier les deux horizons dans une G-carte, il faut non seulement créer les piliers verticaux, mais créer toutes les faces verticales du maillage 3D, ce qui revient à dupliquer les arêtes des horizons. Mais cette duplication ne peut se faire par moitié car cela rompt les cycles $\alpha_0\alpha_3\alpha_0\alpha_3$ et $\alpha_0\alpha_2\alpha_0\alpha_2$, comme le montre le cercle rouge sur la figure 7.23(b) avec les arêtes de couleur cyan. Ainsi, lors de la création du piliers, pour conserver la cohérence des G-cartes, il est nécessaire de créer d'abord ces arêtes du haut et bas des faces internes des mailles.

Plus précisément, pour créer les différentes faces, nous commençons par dupliquer toutes les arêtes des surfaces qui sont incidentes aux piliers que nous souhaitons créer. Ces duplications serviront de haut/bas des futures faces de coté des mailles, elles sont illustrées en cyan sur la figure 7.24(a).



(a) Arêtes dupliquées (b) Première partie du pilier (c) Création des liaisons α_2 entre les faces incidentes aux piliers

FIGURE 7.24 – Création d'un pilier vertical entre deux sommets associés (haut et bas)

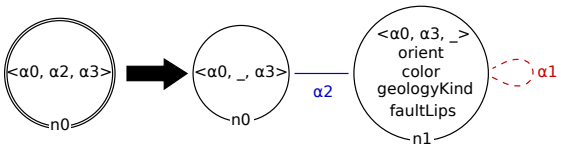


FIGURE 7.25 – Règle de duplication d'une arête

La règle Jerboa réalisant la duplication d'une arête est montrée en figure 7.25. Nous souhaitons uniquement dupliquer les arêtes du maillage 2D et non les lèvres de failles qui ne sont pas incidentes à des piliers. C'est pourquoi la règle filtre une unique arête (orbite $\langle \alpha_0, \alpha_2, \alpha_3 \rangle$) et non toute la surface comme le faisait la règle de la figure 7.20.

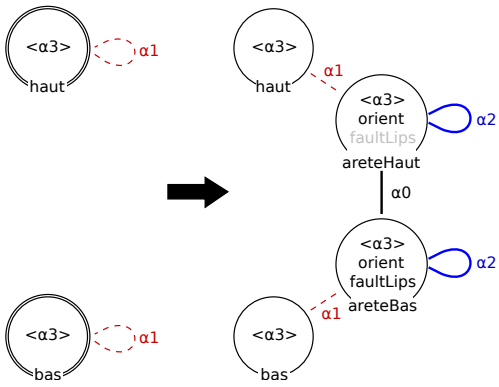


FIGURE 7.26 – Règle de création d'une arête entre celle d'un horizon du haut et d'un du bas

Une fois toutes les arêtes du maillage dupliquées, nous créons les piliers en créant les arêtes de coté des faces internes comme illustré en figure 7.24(b). La règle associée à cette opération est présentée en figure 7.26. Ici l'arête d'une seule face incidente au pilier a été créée. Il faut appliquer cette règle sur les sommets de tous les couples d'arêtes, associées haut/bas, afin de

créer toutes les faces internes.

Une fois les faces internes créées, nous procédons à leur liaison topologique en α_2 autour des piliers, comme illustré en figure 7.24(c). À l'issue de cette étape, nous obtenons le maillage 3D complet des horizons et des failles.

7.3.4 Extraction des blocs avec une représentation par les bords

Nous appelons un bloc un ensemble de mailles délimité par le bord de la scène, par des surfaces d'horizons ou de failles. Ainsi, une scène complète contient plusieurs blocs. Par exemple, l'extraction des blocs est utile pour travailler sur un seul d'entre eux, pour le visualiser ou pour extraire ses informations. Cette extraction s'effectue sous la forme d'une représentation par les bords, avec les bords maillées, mais pas le maillage 3D interne. Notons, qu'il est cependant plus facile d'extraire les blocs une fois les piliers du maillage 3D construits, car c'est eux qui referment les blocs sur les bords. C'est aussi lors de la création des piliers que les portions de surfaces d'horizons et de failles d'un même blocs sont connectées les uns aux autres.

À cette étape, deux blocs séparés par une faille ne sont pas liés, car les failles ont été réalisées à l'aide de deux demi-failles déconnectées pour permettre le glissement. Mais deux blocs séparés par un horizons sont quand à eux connectés, car les surfaces d'horizons sont crées avec leurs deux cotés connectés. La première étape de l'extraction d'un bloc, est donc de découdre par α_3 ses horizons du haut et du bas. Ce qui se fait aisément grâce aux étiquettes de face qui les identifient. Une fois les horizons décousus, chaque bloc est une composante connexe. Il est donc facile d'identifier un bloc pour l'extraire.

L'extraction proprement dite d'un bloc identifié consiste à enlever les faces internes du maillage. Ces faces sont aisément identifiables, soit en utilisant leur étiquette sémantique, soit car ce sont désormais les seules à posséder deux demis-faces voisines par α_3. La suppression des faces internes du maillage se fait en deux étapes. Tout d'abord elles sont isolées, en les décousant par α_2. Cette étape permet de reformer les surfaces de bord. En effet, deux faces de bord sont liées topologiquement par l'intermédiaire des faces internes. Pour reformer correctement les bords, il faut reformer la topologie entre les mailles du bord. Pour réaliser cette étape efficacement, une solution simple consiste à parcourir toutes les arêtes du bord du bloc pour les déconnecter de la face interne et reconstituer l'arête de bord d'origine.

La règle associées à cette opération est montrée en figure 7.27.

Grâce au filtrage, il est possible d'appliquer la règle sur toutes les arêtes du bloc, et seules celles pouvant être filtrées sont modifiées. Sur la règle, c'est la présence des boucles α_3 sur les nœuds `n0`, `n1`, `n6`, et `n7` qui permettent de filtrer seulement les arêtes du bord.

On peut ainsi écrire le *script* suivant pour appliquer la règle à toutes les arêtes du bord :

```
1  for(JerboaDart b : bloc) // pour tous les blocs de la liste
2  for(JerboaDart arete : <0,1,2,3>_<0>(b)){
3      try{
4          @rule<ReformeAreteBord>(arete);
5      }catch(JerboaException e){
6          @print(e.what());
7          // on affiche l'exception sur la sortie standard
8          // dans le cas où la règle ne s'est pas appliquée
9      }
10 }
```

Ici nous avons choisi de réaliser une itération sur toutes les orbites $\langle\alpha_0\rangle$ (ligne 2 du *script*) plutôt que sur toutes les arêtes (orbite $\langle\alpha_0, \alpha_2, \alpha_3\rangle$) afin d'exploiter la puissance du filtrage des règles.

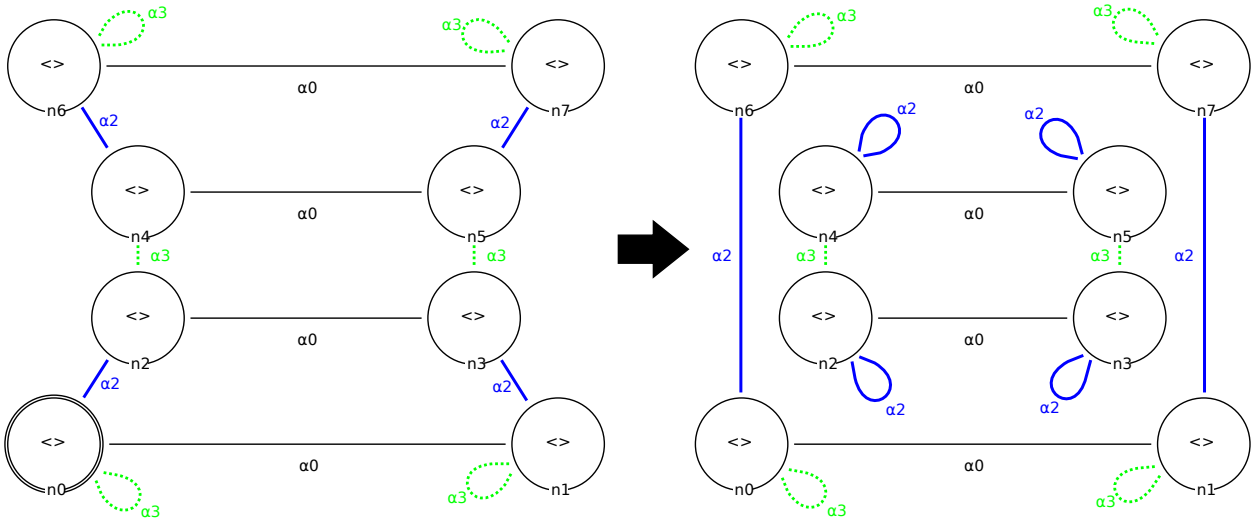


FIGURE 7.27 – Règle Jerboa `ReformeAreteBord` de découture α_2 de la face interne de maille et reconstitution de l'arête externe

En effet, si nous avions itéré sur toutes les arêtes, nous aurions du chercher si cette dernière était incidente à un bord, puis appliquer rechercher un nœud du bord où appliquer la règle. Cela aurait nécessité de réaliser un parcours supplémentaire. Ici, si l'orbite $\langle \alpha_0 \rangle$ que nous parcourons est au bord, la règle s'applique, dans le cas contraire, le filtrage échoue et la règle ne s'applique pas.

Une fois toutes les faces internes déconnectées de celles du bord, des horizons et des failles, et les arêtes de bord reconnectées entre elles, on peut enlever toutes les composantes connexes des faces internes. Avec cette méthode, on obtient une surface connexe contenant toutes les faces de bord du maillage 3D des blocs, sans les faces internes.

Ce résultat n'est cependant pas satisfaisant, dans le cas où l'on souhaite avoir un maillage le plus simple possible. On peut encore le simplifier en rassemblant les faces de bords, adjacentes et coplanaires, en une même face. Cette simplification n'est pas souhaitable sur le maillage des horizons et des failles pour en garder toute la précision, mais très utile sur les surfaces de bord du modèle (étiquetées comme tel). Pour ce faire, nous parcourons toutes les arêtes incidentes à deux faces de bord, et leur appliquons la règle de simplification présentée en figure 7.28.

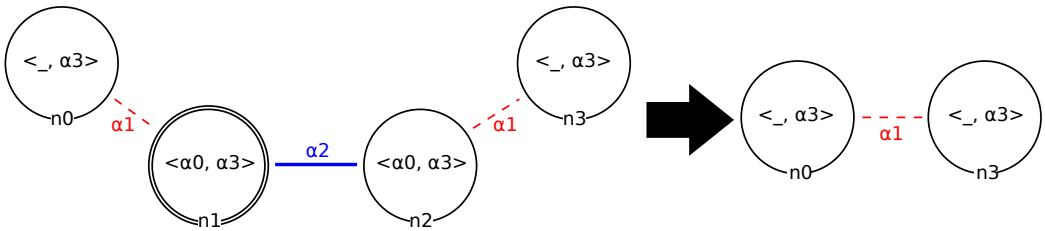


FIGURE 7.28 – Règle Jerboa permettant de fusionner 2 faces incidentes

Cette règle fusionne deux faces le long d'une arête, si l'angle entre les deux normales des faces est supérieur à un ϵ donnée. Pour s'assurer de la satisfaction de cette condition, on définit la pré-condition suivante :

```
@ebd<posPlie> normN1(computeNormal(<0,1>_posPlie(@leftPattern#n1#0)));
@ebd<posPlie> normN3(computeNormal(<0,1>_posPlie(@leftPattern[n3,0])));
// on prend n3 pour avoir la même orientation que n1
// sinon les normales sont inversées
return normN1.dot(normN3) >= cos(M_PI*0.01);
  // Epsilon d'erreur de PI/100.
```

Cette pré-condition calcule la normale des deux faces voisines, et retourne vrai si l'angle entre les deux est inférieur à $\frac{\pi}{100}$. Nous avons choisi cette valeur arbitrairement, et il est aisé de la changer. Cependant, une valeur trop importante risquerait d'autoriser la fusion de face dont les plans supports sont trop différents.

On applique cette règle sur toutes les arêtes de bord. Avec la pré-condition, la règle ne s'applique que sur les arêtes incidentes à des faces formant un angle assez faible.

La figure 7.29 montre l'application des règles que nous venons de présenter sur un bloc. Le bloc est affiché en gros plan afin de distinguer les liaisons topologiques. La figure 7.29(a) montre le bloc isolé avant extraction. La figure 7.29(b) montre la déconnexion des faces internes et externes. Au premier plan, on peut voir les deux faces externes bleues qui sont désormais directement liées par α_2. La figure 7.29(c) montre la suppression proprement dite des faces internes, en particulier les faces grises et vertes au deuxième plan. Finalement, la figure 7.29(d) montre la simplification des faces coplanaires. On voit au premier plan les faces bleues du bord qui on été fusionnées, et de même pour les faces de l'arrière plan. Au contraire, les faces de l'horizon que l'on aperçoit en haut du bloc ne sont pas fusionnées, bien qu'elles soit globalement sur le même plan, car nous souhaitons garder le maillage 2D des horizons et ne simplifier que les bords des blocs.

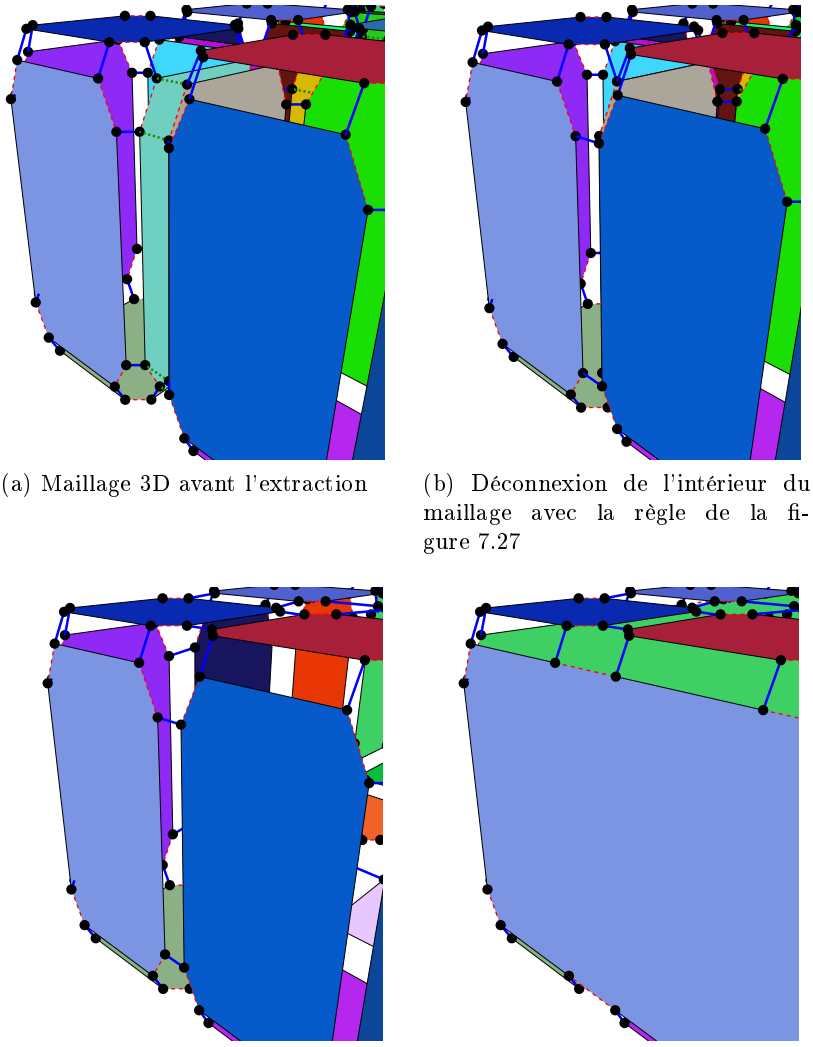
La figure 7.30 montre le résultat de l'extraction de block sur un maillage 3D simple.

7.3.5 Découpe stratigraphique des unités

Le maillage généré par notre méthode à l'issue de la création des piliers (cf. section 7.3.3, page 147) construit une seule maille de haut par unité qui relie directement les horizons du haut et du bas. Cependant, il est souhaitable de subdiviser verticalement ces mailles pour obtenir chacune des strates telles qu'elles sont observées le long des puits de sondage. La figure 7.31 montre la création des strates (ou *layers*) sur deux unités. On peut observer que dans l'espace de dépôt, ils sont bien horizontaux, car chaque strate s'est déposée à une même époque.

La création de ces strates peut être réalisée de plusieurs façons. Soit on procède à une découpe après la création du maillage en découpant toutes les arêtes par un plan placé à un z donné dans l'espace à plat, soit on les crée pendant l'étape de création des piliers verticaux.

Sans la présence de faille ou d'érosion, l'ajout de ces couches stratigraphiques est assez aisé car c'est une simple découpe de tous les piliers d'une même unité avec ajout de faces pour former la surface de l'horizon interne. Mais avec des failles ou des érosions, la découpe est un peu plus complexe car elle doit se faire aussi sur ces éléments qui ne sont pas de simples piliers. Il faut couper le maillage des failles et des érosions qui ne sont pas plans dans l'espace de dépôt et donc trouver les intersections entre le plan de découpe correspondant à l'horizon interne (dans l'espace de dépôt) que l'on souhaite insérer, et les surfaces de faille/érosion. La figure 7.32 montre le *foot-wall* généré sur des données synthétiques entre deux horizons. Les faces en rouge correspondent à une faille et celle en bleu à l'horizon du dessus. L'unité a été coupée en quatre couches stratigraphiques. On observe que les lignes de séparation des couches sont également



(a) Maillage 3D avant l'extraction

(b) Déconnexion de l'intérieur du maillage avec la règle de la figure 7.27

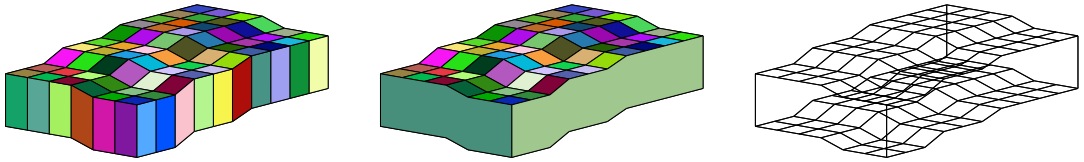
(c) Suppression des faces internes

(d) Simplification avec la règle de la figure 7.28

FIGURE 7.29 – Extraction de block : application sur une arête de bord

présentes sur la surface de faille. Notons que les deux lignes qui traversent la surface de faille en biais sont issues de la projection du maillage 2D sur la faille.

Pour réaliser ces découpes nous avons commencé par couper les faces du maillage 3D par les surfaces de passage des strates. Nous avons insérer les sommets aux intersections puis coupé les faces de façon similaire à notre méthode d'insertion des lèvres de failles sur les horizons présentée en section 7.3.1, page 138. Pour finir nous avons ajouté les faces des horizons internes, de façon similaire à notre méthode de création des piliers présentée en section 7.3.3, page 147.

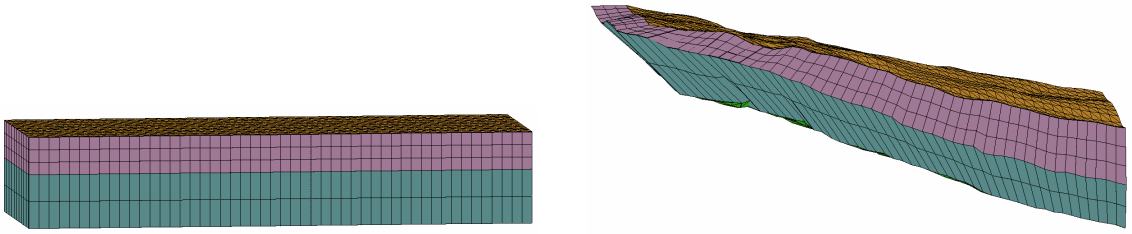


(a) Bloc 3D extrait (faces internes enlevées

(b) Bloc 3D extrait simplifié sur les bords

(c) Bloc 3D extrait simplifié sur les bords en vue fil de fer

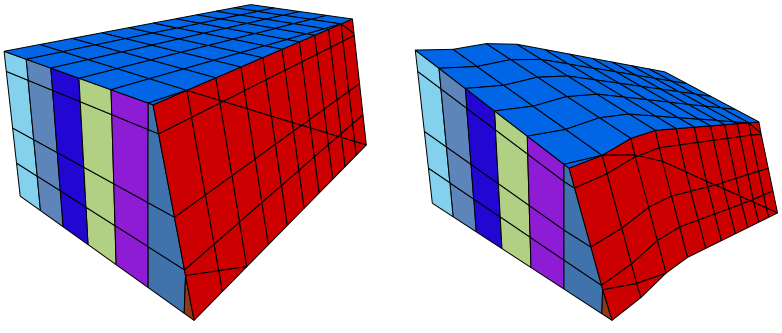
FIGURE 7.30 – Extraction d'un block



(a) Espace de dépôt

(b) Espace actuel

FIGURE 7.31 – Découpe stratigraphique de deux unités successives



(a) Espace de dépôt

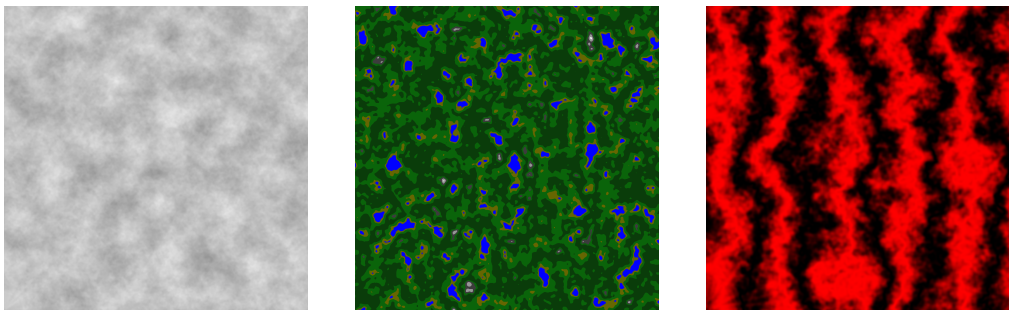
(b) Espace actuel

FIGURE 7.32 – Ajout de couches stratigraphiques (ou *layers*)

Chapitre 8

Réalisation

Comme nous l'avons vu dans le chapitre précédent, les premières expérimentations de nos algorithmes de maillage ont été effectuées sur des données synthétiques. Ces données ont été générées en créant des surfaces composées de quadrangles pour les horizons. Ce sont des grilles régulières dont le nombre de quadrangles en largeur et longueur peut être paramétré. Pour augmenter le réalisme des données, nous avons appliqué un bruit de Perlin [Per85] sur les sommets. Ce bruit permet de générer des données pseudo aléatoires en fonction de différents paramètres. Son avantage est qu'il permet de produire des motifs procéduraux qui ne sont pas de simple bruits aléatoires. Pour deux paramètres proches (comme deux positions géométriques peu éloignées l'une de l'autre par exemple), la valeur du bruit est également assez proche. On peut observer cette particularité sur la figure 8.1, qui illustre plusieurs motifs aléatoires que nous avons généré en fonctions de différents paramètres.



(a) Noir et blanc (b) Carte (c) Avec une sinusoïdale

FIGURE 8.1 – Bruit de Perlin utilisé pour généré une texture

Une fois les horizons générés, nous avons placé des failles planes dans l'espace de dépôt puis coupé les horizons par ces plans. Nous simulons les déformations produites par les failles en déplaçant les différentes parties coupées (*foot-wall* et *hanging-wall*).

L'utilisation de ces données synthétiques nous a permis d'avancer sur notre compréhension du problème et sur la mise au point de nos algorithmes, à un moment où nous n'avions pas accès à des données réelles. De plus, la création de modèles synthétiques permet de créer des cas d'étude simples en se focalisant sur un seul problème identifié, ce qui n'est pas toujours possible avec des données réelles. Une fois notre méthode testée sur ces modèles nous l'avons appliquée sur des données réelles, issues du modèle d'ALWYN [IG91].

8.1 ResQML et données réelles

Les données sismiques sont mesurées directement sur le terrain avec des ondes sismiques émises dans le sol. Les ondes de retour sont analysées pour produire les données brutes du terrain. Ces données sont ensuite interprétées pour définir des nuages de points. Comme nous l'avons vu dans la section précédente, les nuages de points sont également interprétés pour définir la position des différents évènements géologiques. Ces interprétations étant le fruit du travail de géologues, il est possible qu'il en existe plusieurs pour un même jeu de données. Afin de rassembler différentes interprétations d'un même jeu de données brutes et pour être en mesure de communiquer facilement entre différentes entreprises travaillant sur ces données, un format de données a été normalisé. Cette norme s'appelle *ResQML*. C'est une jeune convention qui commence à se stabiliser mais continue néanmoins d'évoluer. Les données stockées étant de taille et en nombre important, ResQML utilise la librairie HDF5 [3], pour gérer facilement l'organisation des données.

ResQML permet de distinguer plusieurs types d'informations. Les *features* permettent de définir les différents éléments abstraits qui composent un jeu de données. Par exemple on peut définir une *feature* faille (respectivement un horizon, une érosion, *etc.*), à laquelle on donne un nom. Pour chaque *feature*, il est possible d'avoir plusieurs *interprétations* des géologues. Un troisième type de données important est le type *representation*. Les représentations sont des données concrètes comme des surfaces triangulées (*TrianglatedSetRepresentaton*), des poly-lignes (*PolylineSetRepresentation*), *etc.* Chaque représentation est bien entendu associée à l'interprétation dont elle est issue, ce qui permet d'importer uniquement les données issues d'une interprétation particulière par exemple. Nous avons donc développé un module d'import ResQML dans Jerboa afin d'être en mesure de charger les différentes données fournies par Géosiris dans ce format.

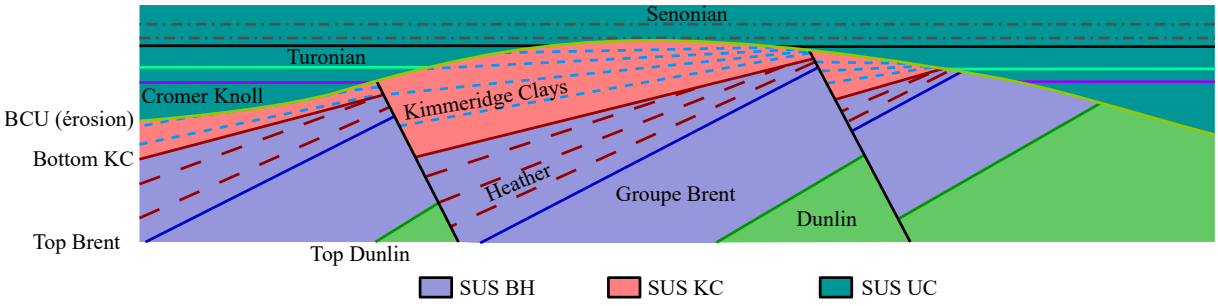
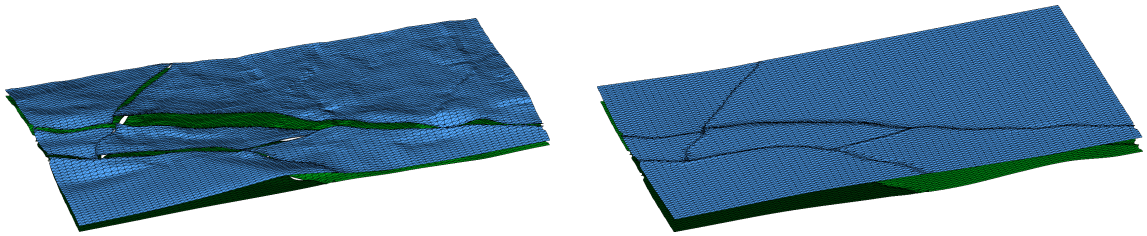


FIGURE 8.2 – Modèle ALWYN

Nous avons concentré notre étude sur le cas du modèle ALWYN [IG91] schématisé sur la figure 8.2. Ce modèle est issue de mesures prises sur un prospect situé en mer du Nord, à l'ouest de la Norvège. Dans ce modèle, le dépôt le plus ancien que nous illustrons ici est appelé *Dunlin*. Suite à ce dépôt, d'autres sédiments se sont déposés et forment le groupe *Brent*. Suite à ces dépôts, des failles se sont formées, ont coupées ces sédiments et ont provoqué des basculements. L'unité *Heather* s'est déposée pendant ces mouvement de faille, et s'est donc formée en biseaux. Le groupe *Brent* et l'unité *Heather* font partie du *Stratigraphic Unit Stack BH*. L'unité *Heather* a ensuite été érodée et l'unité *Kimmeridge Clays* s'est déposée au dessus, toujours pendant les mouvements des failles. Suite à ces dépôts, une autre érosion nommée *BCU* s'est produite, et délimite avec la première le *Stratigraphic Unit Stack Kimmeridge Clay* (SUS KC). Pour finir, d'autres unités se sont déposées sur cette érosion et forment le *Stratigraphic Unit Stack Upper*

3. https ://support.hdfgroup.org/HDF5/

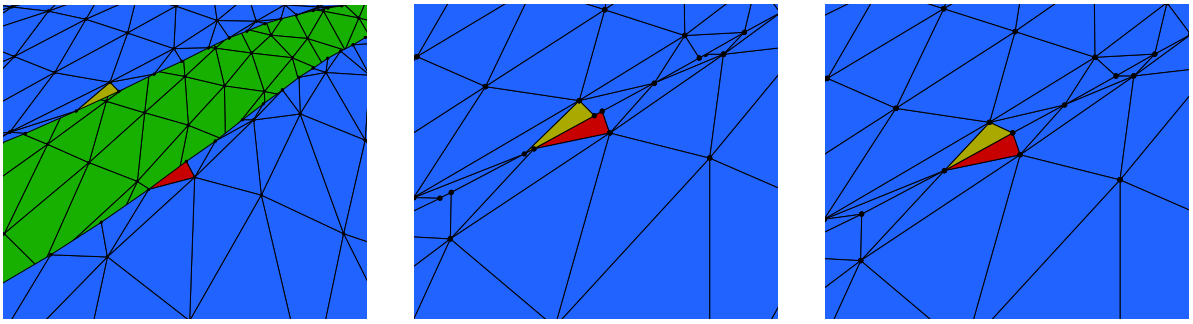
Cretaceous (SUS UC). Les données que Géosiris nous a fournit contiennent les horizons *Top Brent* et *Top Dunlin* et nous avons donc testé notre méthode de maillage sur le groupe *Brent*.



(a) Espace actuel (b) Espace de dépôt

FIGURE 8.3 – Modèle ALWYN avec les surfaces *Top Brent* et *Top Dunlin*

La figure 8.3 montre un exemple de données réelles issues du modèle ALWYN dans les deux espaces. Ces données sont issues de la reconstruction structurale 2D effectuée par Géosiris avec son logiciel GeoTopoModeleur. Ici, l'horizon *Top Brent* est en bleu et *Top Dunlin* est en vert.



(a) Espace actuel (b) Espace de dépôt avant nettoyage des données (c) Espace de dépôt après nettoyage des données

FIGURE 8.4 – Problème de concordance des lèvres de faille dans l'espace de dépôt sur données réelles

Notons que le GeoTopoModeleur est lui-même en cours de développement et la méthode de reconstruction structurale 2D est encore en cours de mise au point. Nous avons donc eu plusieurs versions successives des données d'ALWYN, comprenant plusieurs types d'imperfections. Par exemple, la présence de mauvaises informations sur les lèvres de failles. Ce problème nous empêchait notamment de procéder à la reconstruction de la faille car nous n'étions pas en mesure d'associer deux lèvres correspondant au même côté (*foot-wall* ou *hanging-wall*) de la même faille sur deux horizons (du haut et du bas) incidents à une même unité. Un autre exemple est un mauvais glissement le long des failles dans l'espace de dépôt. En effet, dans l'espace de dépôt, les deux côtés d'une lèvre de faille doivent être exactement face à face, car les failles et l'éventuel glissement qu'elles provoquent est postérieur au dépôt. La figure 8.4, montre ce problème de non concordance des lèvres de failles dans l'espace de dépôt. Sur la figure 8.4(b), on peut voir que les triangles orange et rouge ne sont pas face à face. En attendant que ce problème soit résolu dans le GeoTopoModeleur, nous avons mis en place un *script* Jerboa qui permet de recaler les lèvres de failles. Ce *script* parcours toutes les paires de lèvres de failles (celles qui se font faces), et avance de sommet en sommet pour les recaler deux à deux à leur position médiane. Ceci est bien sûr possible car les deux côtés d'une même lèvre de faille ont exactement le même nombre

de sommets. Le résultat de ce recalage des lèvres est illustré en figure 8.4(c), où les triangles orange et rouge sont cette fois bien en face.

8.2 Résultats

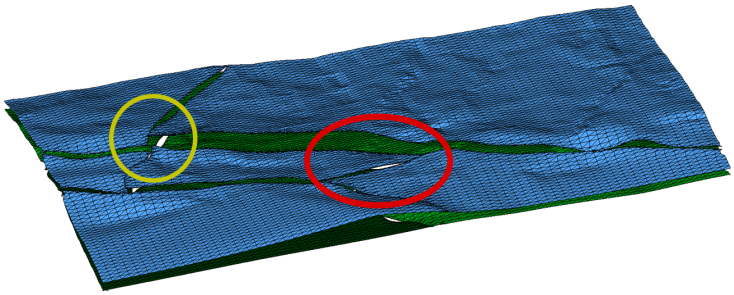
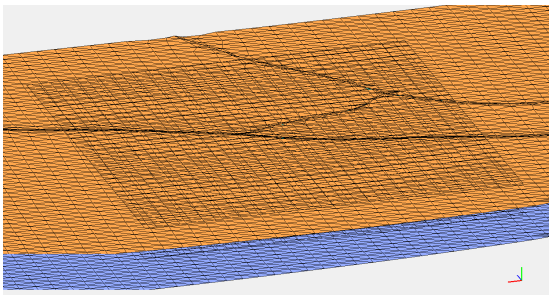


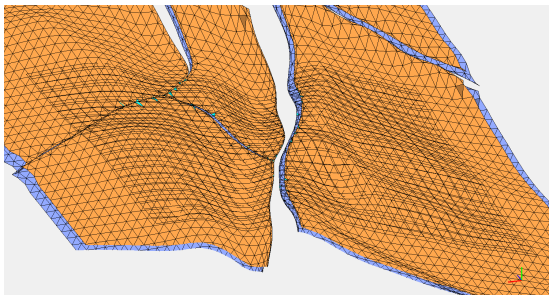
FIGURE 8.5 – Modèle original avec en entouré de couleur les zone dans lesquelles nous allons montrer nos résultats de maillage

Nous avons mis en application l'algorithme décrit dans les sections précédentes sur une portion d'un modèle réel appelé ALWYN contenant les horizons *Top Brent* et *Top Dunlin* (comme précédemment, respectivement bleu et vert sur la figure 8.5). Sur la figure, nous avons entouré les zones sur lesquels nous allons présenter nos résultats de maillage.

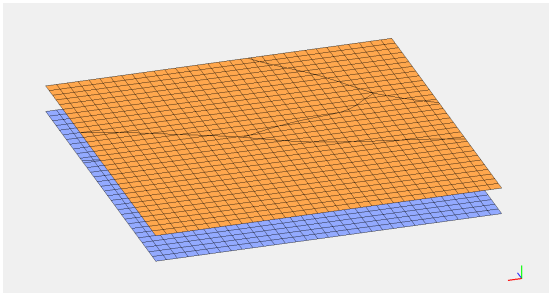
8.2.1 Re-maillage des surfaces 2D



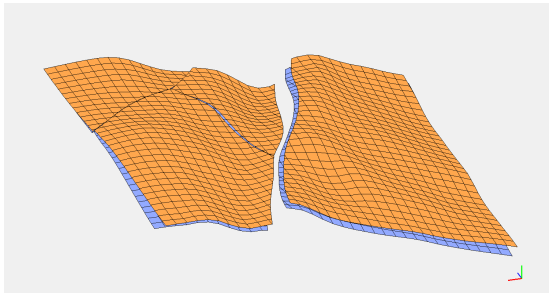
(a) Horizon et maillage 2D dans l'espace de dépôt



(b) Horizon et maillage 2D dans l'espace actuel



(c) Maillage 2D dans l'espace de dépôt

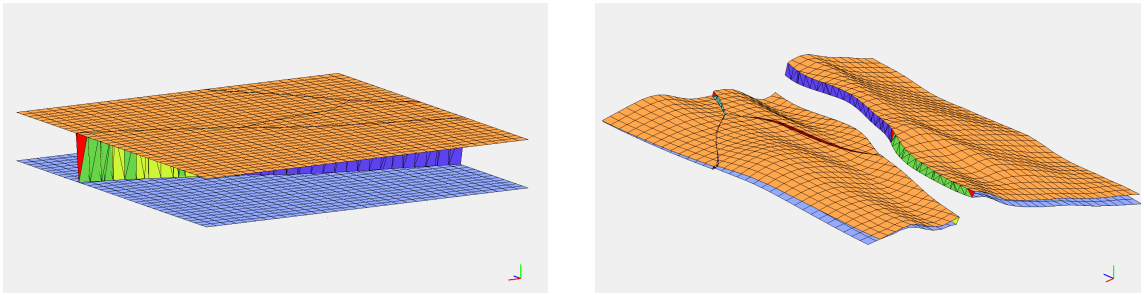


(d) Maillage 2D dans l'espace actuel

FIGURE 8.6 – Placage de la grille sur les horizons

La figure 8.6 montre la première étape de notre algorithme, la projection du maillage 2D sur tous les horizons. La portion de donnée présentée ici est celle du cercle rouge de la figure 8.5. Ici les lèvres de failles on été insérées en assignant à chacune l'étiquette qui lui correspond, afin de conserver les informations relatives à la faille qui a produit la découpe.

8.2.2 Maillage des failles



(a) Espace de dépôt (b) Espace actuel

FIGURE 8.7 – Génération des surfaces de faille

L'étape suivante, illustrée en figure 8.7, est la reconstruction des surfaces de faille, entre les lèvres de faille des horizons. Sur la figure 8.7(b) on distingue deux surfaces de faille différentes dans l'espace actuel, une bleue et une verte. Ces deux surfaces sont en fait les deux parties d'une même faille. Cependant, elle est générée en deux parties car la faille du milieu du modèle vient toucher la première faille et couper ses mailles. C'est le cas des failles en T que nous avons détaillé dans la section 7.1, page 133. Dans l'espace de dépôt, la visualisation est assez difficile car les deux côtés de la faille sont à la même position, mais on distingue bien les surfaces reliant les lèvres de l'horizon du haut à celles du bas.

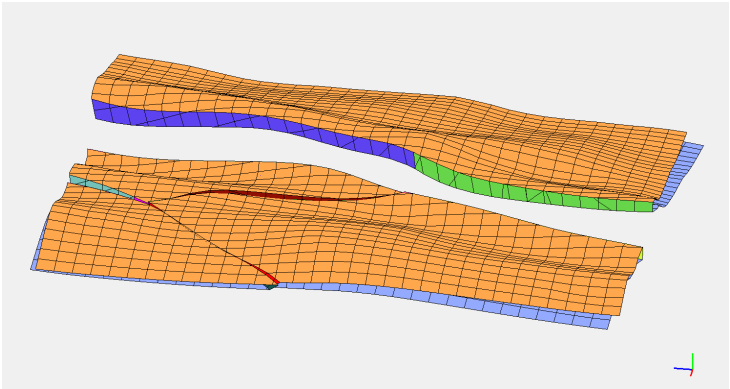
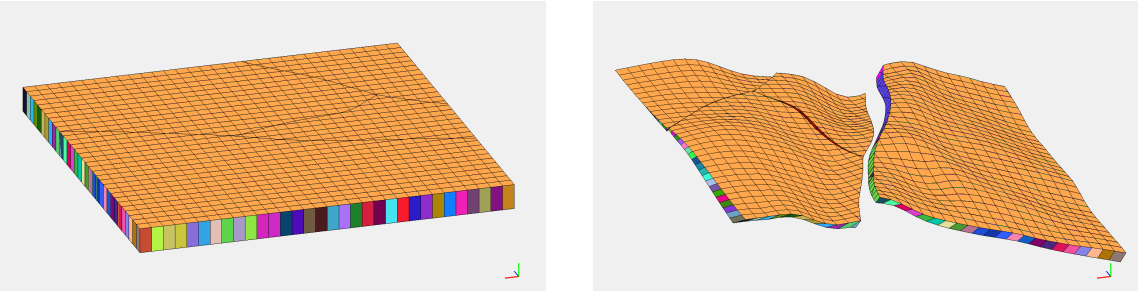


FIGURE 8.8 – Placage de la grille sur les failles

La figure 8.8 montre le résultat du placage du maillage 2D sur les failles. À cette étape les sommets des maillages projetés sont tous stockés dans une structure spécifique qui détermine leurs correspondances haut/bas entre les différentes surfaces, comme nous l'avons vu dans la section 7.3.3, page 147.

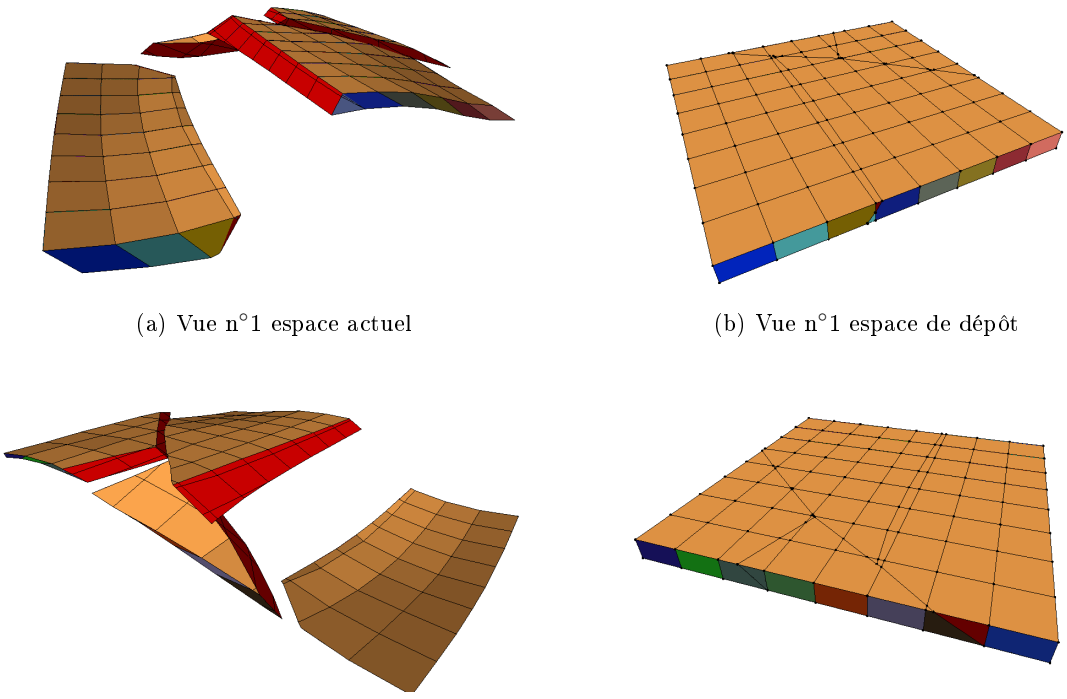
8.2.3 Construction du maillage 3D



(a) Espace de dépôt (b) Espace actuel

FIGURE 8.9 – Maillage final

La figure 8.9 montre le maillage final. Des piliers verticaux ont été tirés dans l'espace de dépôt pour relier les sommets appairés des différentes surfaces. Les couleurs des mailles ont été affectées aléatoirement pour mieux les distinguer.



(a) Vue n°1 espace actuel (b) Vue n°1 espace de dépôt

(c) Vue n°2 espace actuel (d) Vue n°2 espace de dépôt

FIGURE 8.10 – Différentes vue du résultat d'un maillage dans une zone fortement faillée du modèle ALWYN

La figure 8.10 montre plusieurs angles de vue du résultat du maillage final de la zone jaune (cf. figure 8.3). Comme on peut le voir, cette zone comporte deux failles, qui coupent le modèles en quatre blocs. Les surfaces de faille sont ici en rouge.

8.3 Perspectives

Nous avons pu mettre en œuvre une méthode qui tient compte de l'interprétation des données afin de construire des maillages 3D du sous-sol. Nous avons pu tester notre méthode sur les données que nous avons à disposition grâce à Géosiris. Nous espérons à court terme avoir accès à un jeu de données plus important afin de mettre à l'épreuve notre méthode sur un plus gros modèle.

Nous avons développé un module d'importation ResQML qui nous permet de charger les données issues de la reconstruction structurale effectué par Géosiris. Nous allons également développer un module d'export dans ce format afin de produire des résultats exploitables par la communauté.

Nous projetons également d'améliorer certaines étapes de notre algorithme de maillage, notamment la reconstruction des failles. Actuellement, elles sont reconstruites entièrement mais ne tiennent pas compte des surfaces de failles issues de la reconstruction structurale (*cf.* section 7.3.2, page 145). Nous améliorerons cette étape notamment en effectuant une projection des surfaces de failles reconstruites sur la surface issue de la reconstruction structurale dans l'espace actuel. Nous compléterons également notre méthode de découpe stratigraphique pour ajouter la possibilité de gérer les différentes conformités avec les horizons des unités (parallèle à l'horizon du haut, parallèle à celui du bas, ou a une autre surface, *etc.*) telles qu'elles ont été décrites dans la section 6.1, page 122 sur la figure 6.6.

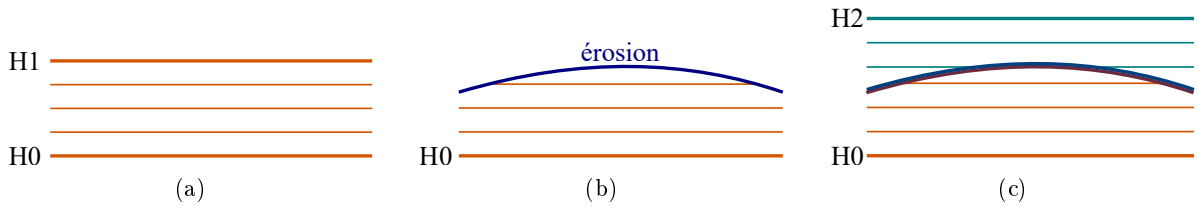


FIGURE 8.11 – Vue de côté dans de l'impact d'une érosion dans l'espace actuel

La principale perspective est de compléter l'algorithme de maillage 3D, et en premier lieu de prendre en compte d'autres évènements géologiques comme les basculements et les érosions. Comme nous l'avons dis plus tôt, nous pensons pouvoir gérer les érosions d'une façon similaire aux surfaces de faille, en leur attribuant deux positions dans l'espace de dépôt. Les érosions enlèvent de la matière des unités géologiques comme le montre la figure 8.11(b). Ces surfaces servent ensuite de support pour le dépôt des horizons suivant comme sur la figure 8.11(c).

La figure 8.12 représente la succession des différents évènements de la figure 8.11(c) au cours du temps. Pour chaque époque, correspond une géométrie différente du sous-sol. Comme nous l'avons vu dans la section 6.2.2, page 130, la position d'un horizon dans l'espace de dépôt (sa coordonnée z) est en fait son époque de dépôt. La figure 8.13 montre un tel espace de dépôt pour l'exemple de la figure 8.11. Bien entendu, une telle représentation est purement technique. Elle permet de construire le maillage 3D, mais ne correspond à aucune vue géométrique du sous-sol quelle que soit l'époque considérée. Pour visualiser le sous-sol à différentes époques, il conviendrait de mémoriser non seulement l'époque de création de chaque horizon, mais aussi sa position géométrique aux différentes époques considérées. En effet, la géométrie des surfaces qui change au gré des évènements géologiques : création d'une faille, plissement d'une unité, *etc.*

Nous avons vu section 7.3.2, page 145 que dans le cas des failles, l'époque de création de chaque faille n'est pas directement utile pour mailler les blocs de part et d'autre de la faille. Ce

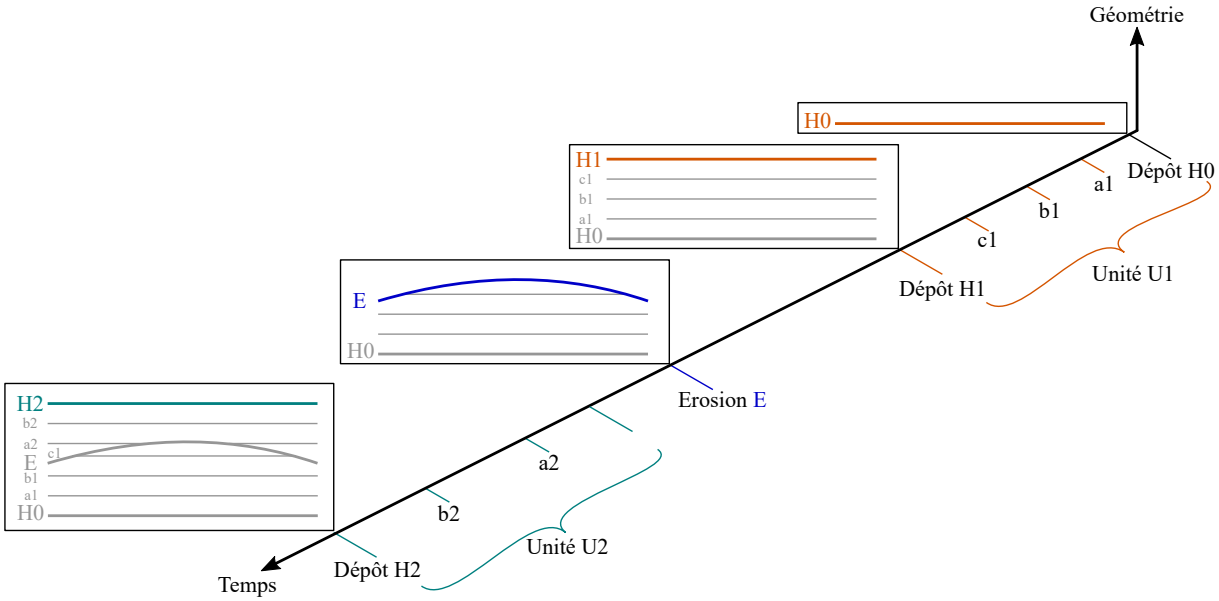


FIGURE 8.12 – Espaces de dépôts dans le cas d'une sédimentation interrompue par une érosion



FIGURE 8.13 – Espace de dépôt représentant les surfaces à leur temps respectif

qui importe, c'est l'époque de dépôt des différentes couches stratigraphiques qui jouxtent la faille de chaque côté. Ainsi, chaque bloc peut être maillé à partir des positions dans les deux espaces, actuel et de dépôt, des surfaces de son bord.

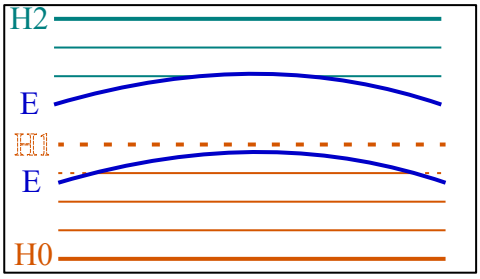
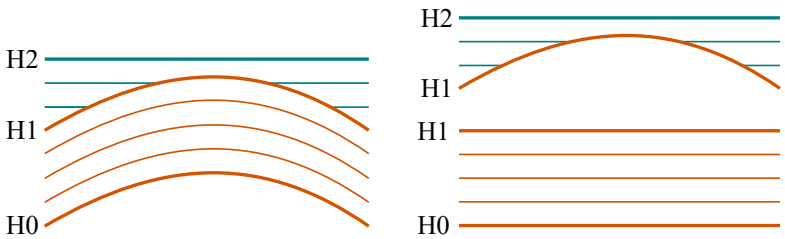


FIGURE 8.14 – Espace de dépôt avec Jerboa

Nous proposons de procéder selon le même principe pour les surfaces d'érosion. L'idée est de ne pas considérer l'époque de création de l'érosion dans l'espace de dépôt, mais les époques de création des couches stratigraphiques respectivement en dessous et au dessus de la surface d'érosion, comme le montre la figure 8.14. En effet, dans notre exemple, l'érosion E supprime le

haut de l'unité $U1$. Ainsi, le haut de l'unité $U1$ restante après l'érosion correspond à des couches stratigraphiques plus ou moins anciennes selon les positions. Cette position dans l'espace de dépôt du dessous de la surface d'érosion va permettre de reconstruire les couches stratigraphiques existantes ou érodées sur l'unité du dessous. De manière symétrique, les couches stratigraphiques au dessus de la surface vont être déposées ou non selon la position géométrique de l'érosion au moment de leur dépôt. Cette position géométrique au moment du dépôt des couches du dessus est ainsi proportionnel à l'époque de dépôt des couches stratigraphiques de l'unité $U2$ du dessus. Comme le montre la figure 8.14, l'espace entre les deux positions de dessous et dessus de l'érosion, correspond aux portions d'horizons qui ont été supprimées par l'horizon ou non créées au dessus de l'horizon.

Comme dans le cas des failles, ces positions dans l'espace de dépôt des deux faces des érosions ne sont pas fournies par l'étape structurale. Elles doivent donc être aussi reconstruites. Comme pour les failles, les positions dans l'espace de dépôt des deux faces des érosions peuvent être interpolées à partir des lignes d'intersection de la surface d'érosion avec les horizons. En effet, comme ces lignes appartiennent aux horizons, nous disposons de leur coordonnées dans les deux espaces.



(a) Espace actuel (b) Espace de dépôt

FIGURE 8.15 – Dépôt d'une unité sur une unité déformée

Ce procédé peut être utilisé pour tous les évènements qui provoquent des discordances entre les unités, comme illustré sur la figure 8.15, avec une unité qui se dépose après la déformation de l'unité du dessous. Dans ce cas, la face du dessous de l'horizon $H1$ a bien une position unique dans l'espace de dépôt, car la couche stratigraphique du dessous s'est déposée horizontalement. Au contraire, les couches qui jouxtent le dessus de l'horizon $H1$ ne se sont pas toutes déposées à la même époque en raison de la déformation. La position de la face supérieure dans l'espace de dépôt n'est donc pas constante.

Notons que les horizons conformes, peuvent être vus comme un cas particulier des horizons non conformes. La surface inférieure, respectivement la surface supérieure, a une position constante dans l'espace de dépôt. La position de ces deux faces peut donc être choisies identiques. Mais un choix de deux positions différentes constantes, produirait exactement le même maillage 3D.

Conclusion générale

Contributions

Avec la bibliothèque Jerboa nous sommes en mesure de définir des opérations de manipulation des objets tout en assurant la préservation de leur cohérence (topologique ou des plongements) après application des opérations. La bibliothèque permet également d'ajouter très rapidement des informations sémantiques appelées plongements afin d'identifier les objets et de leur attribuer des propriété géométriques et physiques. La force de Jerboa est sa facilité et sa rapidité pour développer des modeleurs, leur propriétés et leurs opérations.

Avant que je ne commence ma thèse, la bibliothèque était implantée en Java. Or certains modèles géologiques peuvent contenir un grand nombre d'éléments. Pour permettre la manipulation de gros jeux de données, nous avons réimplanté la bibliothèque en C++. Après étude nous avons été en mesure de manipuler environ 50 millions de brin en Java, alors que la version C++ a pu en manipuler 150 millions mais a été limitée par la mémoire vive de la machine. Nous avons profité de cette phase de développement pour re-travailler l'organisation du noyau afin d'autoriser l'implantation de différents moteurs d'application des règles. Ainsi il est a présent possible de réaliser plusieurs moteurs d'application optimisés pour chaque type de règle. Nous avons nous même implanté un moteur dédié à la mise à jour de plongements. On pourrait également imaginer implanter un moteur qui a des comportement différents du hardware de la machine sur laquelle l'application s'exécute (e.g. utilisation de la technologie CUDA [Nvi11] si la machine dispose d'une carte Nvidia). L'éditeur de modeleur qui permet d'écrire les opérations fonctionne pour les deux versions du noyau de Jerboa dans les deux langages cibles.

Ce dernier a également été réimplanté pour tenir compte des nouveautés de Jerboa et de son langage. Nous avons mis en place un langage qui permet d'écrire les expressions de plongements, qui étaient définies directement dans le langage cible dans la version précédente de la bibliothèque (en Java). Grâce à ce langage, l'éditeur de modeleur est en mesure de réaliser des vérifications automatiques de préservation de la cohérence des plongements par les opérations. Ce langage a été étendu pour permettre l'écriture d'opérations plus complexes que des règles de transformations de graphes qui ne peuvent s'appliquer que sur un type d'orbite à la fois (à renommage et suppression près). Ces opérations sont appelées des *scripts* et permettent d'enchaîner l'application d'autres opérations (règles ou *script*) en utilisant différents concepts des langages impératifs classiques (boucles, alternatives, *etc*). Toujours avec ce langage, nous avons complété le processus d'application des opérations pour permettre à l'utilisateur d'intervenir à chaque étape. Il est à présent possible de réaliser des pré-calculs avant le test de la pré-condition ou juste après, quand le filtrage s'est correctement déroulé mais avant que la règle ne s'applique. Ces différents processus se sont avérés très utiles pour notre application en géologie qui demandait d'utiliser toute la puissance du langage de Jerboa.

Dans le cadre de cette thèse nous avons mis en place, avec la collaboration de Géosiris, un processus de maillage du sous-sol en combinant leurs compétences sur la reconstruction géologique et notre maîtrise des modèles topologiques. Notre méthode permet de conserver la cohérence des

objets tant sur le plan topologique que pour les plongements. Elle consiste à mailler de façon identique les différents horizons et les failles du modèle (aux lèvres de failles près) puis de lier deux à deux les sommets des différentes surfaces, afin de former des mailles 3D entre les éléments.

Grâce à la rapidité de mise en œuvre des opérations dans Jerboa, nous avons pu rapidement tester différentes hypothèses pour notre processus de modélisation. Nous avons montré l'efficacité de notre méthode sur des données synthétiques et sur des données réelles d'ALWYN.

Perspectives

Avec le langage de calcul et de *script* que nous avons mis en place, nous avons développé tout un processus d'interprétation qui permet de générer le code final des opérations. Ce langage étant jeune, certaines parties devraient être améliorées. La traduction du langage pourrait être poussée plus loin afin de mettre en place d'avantage d'optimisations automatiques. Lorsque nous écrivons des opérations par exemple, nous utilisons très souvent les collectes pour appliquer des transformations sur chaque éléments collecté. Dans certains cas, il serait possible de ne pas stocker le résultat de la collecte mais d'appliquer les transformations directement pendant le processus de collecte. Grâce au langage il serait possible de déterminer statiquement dans quels cas ces optimisations seraient réalisables. De même il serait possible détecter certaines parties du code potentiellement parallélisables et de générer le code des opérations en conséquence. Ce serait le cas par exemple de l'application de la règle de triangulation barycentrique sur toutes les faces non triangulaires d'une surface. La règle ne modifie pas la topologie des faces non filtrées par la règle. Ainsi il serait possible d'appliquer cette règle sur plusieurs faces distinctes en même temps. Le test qui détermine si une règle est parallélisable ou non est à la portée de l'éditeur de modeleur qui a accès à toutes les opérations, leurs processus et leurs graphes gauche et droit.

Toujours dans un souci d'optimisation, un travail intéressant serait l'implantation de plusieurs autres moteurs d'application des règles, afin d'appliquer les opérations le plus rapidement possible selon leur type. Nous avons déjà implanté un moteur dédié à la mise à jour des plongements, et avons montré dans la section 4.3, page 97 son efficacité par rapport au moteur général. Ces résultats sont donc encourageants pour l'implantation de nouveaux moteurs.

Nous avons mis en place un processus de composition des règles de transformations de graphe. La composition d'opérations permet d'appliquer en une seule opération une succession de plusieurs autres, et ce afin de ne pas multiplier les étapes de filtrage intermédiaires lorsque l'on sait que deux règles vont s'appliquer successivement sur un même motif. Il est actuellement fonctionnel pour la partie topologique mais pas encore pour les expressions de plongement. Une amélioration intéressante serait de compléter cette méthode avec la composition automatique des expressions de plongements, afin de générer automatiquement des opérations composée sans que l'utilisateur n'ait à re-calculer les plongements de cette opération.

Pour notre partie applicative en géologie, notre méthode devrait être complétée pour prendre en compte les érosions. Actuellement, elle est fonctionnelle uniquement pour un modèle contenant des horizons et des failles. La prise en compte d'érosions permettrait de gérer plus de cas à commencer par le modèle complet d'AWLYN. Nous avons vu dans le chapitre précédent que ce processus pourrait reprendre le principe déjà utilisé pour les failles, d'une géométrie dans l'espace de dépôt différente pour les deux faces de la surface.

Enfin, nous n'avons pu accéder qu'a un jeu réduit de données contenant deux horizons et 5 failles. Notre méthode peut prendre en compte la présence d'un nombre indéterminé d'horizons mais il serait intéressant de pouvoir la mettre à l'épreuve sur un jeu de données plus complet.

Bibliographie

[Acu] A Bentley Systems company Acute3D : Contextcapture. `https://www.acute3d.com/contextcapture`.

[AFMASWS00] David A. Ferrill, Alan Morris, John A. Stamatakos et Darrell W. Sims : Crossing conjugate normal faults. *AAPG bulletin*, 84:1543–1559, 10 2000.

[AMS+04] Mara Abel, Laura S. Mastella, Luís A. Lima Silva, John A. Campbell et Luis Fernando De Ros : How to model visual knowledge : A study of expertise in oil-reservoir evaluation. *In* Fernando Galindo, Makoto Takizawa et Roland Traunmüller, éditeurs : *Database and Expert Systems Applications*, pages 455–464, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Auta] Autodesk : 3Dsmax - 3D modeling, animation and rendering software. `https://www.autodesk.com/3ds-max`.

[Autb] Autodesk : Autocad - 3D CAD design and documentation software. `https://www.autodesk.com/autocad`.

[Autc] Autodesk : Maya - 3D animation visual effects anbd compositing software. `https://www.autodesk.com/maya`.

[BABLG17] Thomas Bellet, Agnès Arnould, Hakim Belhaouari et Pascale Le Gall : Geometric modeling : Consistency preservation using two-layered variable substitutions. *In Graph Transformation : 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*, pages 36–53. Springer International Publishing, 2017.

[BALGB14] Hakim Belhaouari, Agnès Arnould, Pascale Le Gall et Thomas Bellet : JERBOA : A graph transformation library for topology-based geometric modeling. *In 7th International Conference on Graph Transformation (ICGT 2014)*, volume 8571 de *LNCS*, York, UK, juillet 2014. Springer.

[Bau75] Bruce G. Baumgart : A polyhedron representation for computer vision. *In Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, pages 589–596, New York, NY, USA, 1975. ACM.

[Bel12] Thomas Bellet : *Transformations de graphes pour la modélisation géométrique à base topologique*. Thèse de doctorat, Université de Poitiers, 2012. dirigée par Yves Bertrand, Pascale Le Gall et Agnès Arnould, Informatique et applications Poitiers 2012.

[Ble] Blender Online Community : Blender - a 3D modelling and rendering package. `http://www.blender.org`.

[Boh15] Evans Bohl : *Modeling fruits, their internal structure and their defects*. Thèse de doctorat, Université de Limoges, novembre 2015.

[Bor06] Houman Borouchaki : Aplat : déplieur de triangulations surfaciques, 2006.

[Bot16] Arnaud Botella : *Génération de maillages non structurés volumiques de modèles géologiques pour la simulation de phénomènes physiques.* Thèse de doctorat, 2016. dirigée par Guillaume Caumon et Bruno Lévy, Géosciences Université de Lorraine 2016.

[BSBAM16] Fatma Ben Salah, Hakim Belhaouari, Agnès Arnould et Philippe Meseure : Simulation de modèles mécaniques à base topologique à l'aide de règles de transformation de graphes. *In 29e Journées Françaises d'Informatique Graphique*, Actes des 29e Journées Françaises d'Informatique Graphique, Grenoble, France, novembre 2016.

[BSBAM17a] Fatma Ben Salah, Hakim Belhaouari, Agnès Arnould et Philippe Meseure : A general physical-topological framework using rule-based language for physical simulation. *In 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017)*, Proceedings of the Conference on Computer Graphics Theory and Applications, Porto, Portugal, février 2017.

[BSBAM17b] Fatma Ben Salah, Hakim Belhaouari, Agnès Arnould et Philippe Meseure : A Modular Approach Based on Graph Transformation to Simulate Tearing and Fractures on Various Mechanical Models. *Journal of WSCG*, 25(1):39–48, juin 2017.

[CC78] Edwin Catmull et Jim Clark : Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350 – 355, 1978.

[Dam18] Guillaume Damiand : Linear cell complex. *In CGAL User and Reference Manual.* CGAL Editorial Board, 4.12 édition, 2018.

[Del34] Boris Delaunay : Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii Estestvennyka Nauk*, 7:793–800, 1934.

[DFG99] Qiang Du, Vance Faber et Max Gunzburger : Centroidal voronoi tessellations : Applications and algorithms. *SIAM REV*, 41(4):637–676, 1999.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange et Gabriele Taentzer : *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series).* Springer, Secaucus, NJ, USA, 2006.

[Fow10] Martin Fowler : *Domain Specific Languages.* Addison-Wesley Professional, 1st édition, 2010.

[FT13] Efi Fogel et Monique Teillaud : The Computational Geometry Algorithms Library CGAL. *ACM Communications in Computer Algebra*, 47(3):85–87, septembre 2013.

[Gé] Géosiris : GTM - geotopomodeler. `http://geosiris.com/geotopomodeler/`.

[GBA15] Valentin Gauthier, Hakim Belhaouari et Agnès Arnould : Évaluation de la modélisation à base de transformation de graphes avec jerboa. *In Association Française d'Informatique Graphique (AFIG)*, Lyon, décembre 2015.

[GR09] Christophe Geuzaine et Jean-François Remacle : Gmsh : A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79:1309 – 1331, 2009.

[Gui06] Nicolas Emmanuel Guiard : *Construction de modèles géologiques 3D par co-raffinement de surfaces.* Thèse de doctorat, Ecole Normale supérieur des Mines de Paris, 2006.

[HBB+10] Sebastien Horna, Chakib Bennis, Houman Borouchaki, Christophe Delage et Jean-François Rainaud : Isometric unfolding of stratigraphic grid units for accurate property populating : Mathematical concepts. *ECMOR 2010 - 12th European Conference on the Mathematics of Oil Recovery*, 09 2010.

[HDMB07] Sébastien Horna, Guillaume Damiand, Daniel Meneveaux et Yves Bertrand : Building 3D indoor scenes topology from 2D architectural plans. *In Conference on Computer Graphics Theory and Applications. GRAPP'2007.*, Portugal, March 2007.

[IG91] I. Inglis et J. Gerard : The Alwyn north field, blocks 3/9a, 3/4a, UK North Sea. *Geological Society, London, Memoirs*, 14(1):21–32, 1991.

[Ket18] Lutz Kettner : 3D polyhedral surface. *In CGAL User and Reference Manual.* CGAL Editorial Board, 4.12 édition, 2018.

[KUJ+13] Pierre Kraemer, Lionel Untereiner, Thomas Jund, Sylvain Thery et David Cazier : CGoGN : n-dimensional meshes with combinatorial maps. *In* Josep Sarrate et Matthew L. Staten, éditeurs : *Proceedings of the 22nd International Meshing Roundtable, IMR 2013, October 13-16, 2013, Orlando, FL, USA*, pages 485–503. Springer, 2013.

[Lai00] Yong G. Lai : Unstructured grid arbitrarily shaped element method for fluid flow simulation. *AIAA Journal*, 38(12):2246–2252, Dec 2000.

[Lan95] Véronique Lang : *Une etude de l'utilisation des ensembles simpliciaux en modelisation geometrique interactive.* Thèse de doctorat, 1995. dirigée par Pascal Lienhardt, Sciences et techniques communes Strasbourg 1 1995.

[Lie94] Pascal Lienhardt : N-dimensional generalised combinatorial maps and cellular quasimanifolds. *International Journal of Computational Geometry and Applications*, 4(3):275–324, 1994.

[Loo87] Charles Loop : Smooth Subdivision Surfaces Based on Triangles. Department of mathematics, University of Utah, Utah, USA, août 1987.

[MaK14] José Pedro Magalhães et Hendrik Vincent Koops : Functional generation of harmony and melody. *In Proceedings of the 2Nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, FARM '14, pages 11–21, New York, NY, USA, 2014. ACM.

[Mal04] Jean-Laurent Mallet : Space–time mathematical framework for sedimentary geology. *Mathematical Geology*, 36(1):1–32, Jan 2004.

[May67] J. Peter May : *Simplicial objects in algebraic topology.* Van Nostrand Mathematical Studies, No. 11. D. Van Nostrand Co., Inc., Princeton, N.J.-Toronto, Ont.-London, 1967.

[Mer13] Romain Merland : *Génération de grilles de type volumes finis : adaptation à un modèle structural, pétrophysique et dynamique.* Thèse de doctorat, 2013. dirigée par Guillaume Caumon, Pauline Collon et Bruno Lévy, Géosciences Université de Lorraine 2013.

[Nvi11] CUDA Nvidia : Nvidia cuda C programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[PBM+15] Jeanne Pellerin, Arnaud Botella, Antoine Mazuyer, Benjamin P. Chauvin, Bruno Levy et Guillaume Caumon : Ringmesh : A programming library for geological model meshes. *In 35th Gocad Meeting - 2015 RING Meeting*. ASGA, 2015.

[PBM+17] Jeanne Pellerin, Arnaud Botella, Antoine Mazuyer, Benjamin Chauvin, François Bonneau, Guillaume Caumon et Bruno Lévy : RINGMesh : A programming library for developing mesh-based geomodeling applications. *Computers & Geosciences*, 2017.

[Per85] Ken Perlin : An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, juillet 1985.

[PL91] Przemyslaw Prusinkiewicz et Aristide Lindenmayer : *The algorithmic beauty of plants*. The Virtual Laboratory, 1991.

[Pou09] Mathieu Poudret : *Transformations de graphes pour les opérations topologiques en modélisation géométrique : application à l'étude de la dynamique de l'appareil de Golgi*. Thèse de doctorat, Université d'Évry Val D'Essonne, 2009. dirigée par Le Gall, Pascale Informatique.

[PR13] Michel Perrin et Jean-François Rainaud : *Shared Earth Modeling : Knowledge Driven Solutions for Building and Managing Subsurface 3D Geological Models*. Technip, 2013.

[PSTG13] Olga Petrenko, Mateu Sbert, Olivier Terraz et Djamchid Ghazanfarpour : *3Gmap L-Systems Grammar Application to the Modeling of Flowering Plants*, pages 1–21. Springer Berlin Heidelberg, 2013.

[PTMG08] Alexandre Peyrat, Olivier Terraz, Stephane Merillou et Eric Galin : Generating vast varieties of realistic leaves with parametric 2Gmap L-Systems. *The Visual Computer*, 24(7):807–816, 2008.

[PZRS05] Michel Perrin, Beiting Zhu, Jean-François Rainaud et Sébastien Schneider : Knowledge-driven applications for geological modeling. *Journal of Petroleum Science and Engineering*, 47(1):89 – 104, 2005. Intelligent Computing in Petroleum Engineering.

[QRT15] Dongfang Qu, Per Røe et Jan Tveranger : A method for generating volumetric fault zone grids for pillar gridded reservoir models. *Computers and Geosciences*, 81:28 – 37, 2015.

[RPB05] Jean-Francois Rainaud, Michel Perrin et Yves Bertrand : *SPE-94172-MS*, chapitre Innovative Knowledge-Driven Approach For Shared Earth Model Building, page 24. Society of Petroleum Engineers, Madrid, Spain, 2005.

[sch16] SCHNApps. A multi-view interface and a dynamic plugin system for CGoGN. `http://cgogn.unistra.fr`, 2016.

[Si15] Hang Si : Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11 :1–11 :36, février 2015.

[SIN13] SINTEF : MBA multilevel B-spline approximation library. `https://www.sintef.no/projectweb/geometry-toolkits/mba/`, novembre 2013.

[SKU15] SKUA-GOCAD. `http://www.pdgm.com/products/skua-gocad`, 2015.

[Sna] Snapsculpt. `https://www.snapsculpt.com`.

[Son] Sony : Générateur 3D. https://www.sonymobile.com/fr/products/phones/xperia-xz1/3d-creator.

[Spa66] Edwin H. Spanier : *Algebraic topology*. McGraw-Hill series in higher mathematics. McGraw-Hill, 1966.

[Sys] Dassault Systèmes : Catia - logiciel de conception assisté par ordinateur. https://www.3ds.com/catia.

[Ter] Sàrl Terranum : Coltop3D. https://www.terranum.ch/produits/coltop3d/.

[TGM+09] Olivier Terraz, Guillaume Guimberteau, Stéphane Mérillou, Dimitri Plemenos et Djamchid Ghazanfarpour : 3Gmap L-systems : an application to the modelling of wood. *The Visual Computer*, 25(2):165–180, 2009.

[TM83] Mitsuo Takeda et Kazuhiro Mutoh : Fourier transform profilometry for the automatic measurement of 3-D object shapes. *Appl. Opt.*, 22(24):3977–3982, Dec 1983.

[VD] Frédéric Vidil et Guillaume Damiand : Moka. http://moka-modeller.sourceforge.net.

[Vin83] Andrew Vince : Combinatorial maps. *J. Comb. Theory, Ser. B*, 34(1):1–21, 1983.

[Wei85] Kevin Weiler : Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, Jan 1985.

[Wei88] Kevin Weiler : The radial edge structure : A topological representation for non-manifold geometric boundary modeling. *In* M.-J. Wozny, H.-W. McLaughlin et J.-L. Encarnac„o, éditeurs : *Geometric Modeling for CAD Applications : Selected and Expanded Papers from the Ifip Wg 5.2 Working Conference*, pages 3–36. Elsevier Science, 1988.

[Whe58] Harry E. Wheeler : Time-stratigraphy. pages volume 42, no 5, pages 1047–1063, jan 1958.

[Wil14] A. Williams : *OpenSCAD for 3D Printing*. Createspace Independent Pub, 2014.

[WS05] Peter Worth et Susan Stepney : Growing music : musical interpretations of L-Systems. *In* F. Rothlauf *et al.*, éditeurs : *EvoMUSART workshop, EuroGP 2005, Lausanne, Switzerland, March 2005*, volume 3449 de *LNCS*, pages 545–550. Springer, 2005.

Annexe A

Grammaire BNF du langage utilisé pour les scripts de règle et les expressions de plongement

```
##################
####  TOKENS ####
##################

^ --> XOR
+ --> PLUS
- --> MINUS
* --> MULT
/ --> DIV
% --> MOD
and --> AND
&& --> AND
or --> OR
|| --> OR
++ --> INC
-- --> DEC
< --> LT
<= --> COMP
> --> GT
>= --> COMP
== --> COMP
not --> NOT
! --> NOT
!= --> COMP
null --> NULL
NULL --> NULLUPPER
@ --> ALPHA
$ --> DOLLAR
| --> PIPE
=> --> ASSIGN
new --> NEW
delete --> DELETE
```

```
try --> TRY
catch --> CATCH
finally --> FINALLY
throw --> THROW
& --> AMPERSAND
continue --> CONTINUE
if --> IF
else --> ELSE
while --> WHILE
for --> FOR
foreach --> FOREACH
step --> STEP
( --> LPAR
) --> RPAR
{ --> LBRACE
} --> RBRACE
= --> AFFECT
[ --> LBRACKET
] --> RBRACKET
; --> SEMICOLON
: --> COLON
:: --> STATICOP
return --> RETURN
in --> IN
. --> DOT
, --> COMMA
_ --> UNDER
@print --> PRINT
# --> SHARP
@ebd --> EBD
@"[gG]"map --> GMAP
@rule --> RULE
@op --> RULE
@collect --> COLLECT
@modeler --> MODELER
@dimension --> DIMENSION
@"[lL]"eft"[pP]"attern --> \nERN
@"[rR]"ight"[pP]"attern --> RIGHTPATTERN
@mark --> MARK
@unmark --> UNMARK
@is"[mM]"arked --> ISMARKED
@is"[nN]"ot"[mM]"arked --> ISNOTMARKED
@lang --> LANG
@header --> HEADER
( --> LPAR
) --> RPAR
{ --> LANGDEF
{ --> PLAINCODE

#################
### GRAMMAIRE ###
#################

<all_instr> ::= <all_instr> <instr>
```

```
 | <instr>
;
<script_instr_list> ::= <script_instr_list> <script_instr>
 | <script_instr>
;
<instr> ::= <script_instr>
 | <func_declar>
;
<script_instr> ::= <rule_application> SEMICOLON
 | <rule_alternativ> SEMICOLON
 | <assoc_rule_param> SEMICOLON
 | <expr_instr>
;
<expr_instr> ::= <test_condition>
 | <loop>
 | <assoc_rule_param>
 | <try_catch>
 | <specific_lang>
 | <header_lang>
 | <expr_without_semicolon> SEMICOLON
;
<expr_without_semicolon> ::= <assignement>
 | <declaration>
 | PRINT LPAR<arg_list> RPAR
 | <expr_no_previous>
 | DELETE<IDENT>
 | BREAK
 | RETURN<affectable_value>
 | <undo_mark>
 | <do_mark>
 | CONTINUE
 | THROW<expr_no_previous>
;
<test_condition> ::= IF<condition> <block>
 | IF<condition> <block> ELSE<block>
 | IF<condition> <error> ELSE
 | IF<error> <block>
 | IF<error> <block> ELSE<block>
;
<loop> ::= <loop_while>
 | <loop_for>
 | <loop_foreach>
 | <do_while>
;
<loop_while> ::= WHILE LPAR<condition> RPAR<block>
 | WHILE<error>
;
<do_while> ::= DO<block> WHILE LPAR<condition> RPAR SEMICOLON
 | DO<error>
;
<loop_for> ::= FOR<IDENT> IN<expr_calc> DOT DOT<expr_calc> <block>
 | FOR<IDENT> IN<expr_calc> DOT DOT<expr_calc> STEP<const_number> <block
   >
 | FOR LPAR<var_type> <IDENT> AFFECT<expr_calc> SEMICOLON<condition>
```

```
    SEMICOLON<assignement> RPAR<block>
  | FOR<error>
;
<loop_foreach> ::= FOREACH LPAR<var_type> <IDENT> COLON<expr_calc> RPAR<
    block>
  | FOREACH LPAR<var_type> <IDENT> COLON<collects> RPAR<block>
  | FOR    LPAR<var_type> <IDENT> COLON<expr_calc> RPAR<block>
  | FOR  LPAR<var_type> <IDENT> COLON<collects> RPAR<block>
  | FOREACH<error>
;
<block> ::= LBRACE  RBRACE
  | LBRACE<all_instr> RBRACE
  | <instr>
  | LBRACE<all_instr> <error>
;
<func_declar> ::= <var_type> <func_name> LPAR<func_declar_arg> RPAR<
    block>
  | VOID<func_name> LPAR<func_declar_arg> RPAR<block>
;
<func_declar_arg> ::= <var_type> <IDENT>
  | <func_declar_arg> COMMA<var_type> <IDENT>
;
<try_catch> ::= TRY<block> <catch_expr> FINALLY<block>
  | TRY<block> <catch_expr>
;
<catch_expr> ::= CATCH LPAR<var_type> <IDENT> RPAR<block>
  | CATCH LPAR RPAR<block>
  | CATCH LPAR<var_type> <IDENT> RPAR<block> <catch_expr>
;
<rule_application> ::= <rule> LPAR RPAR
  | <rule> LPAR<rule_arg_list> RPAR
  | <rule> LPAR<error> RPAR
;
<rule_alternativ> ::= <rule_application> PIPE<rule_application>
  | <rule_alternativ> PIPE<rule_application>
;
<func_name> ::= <IDENT>
;
<func_call> ::= <func_name> LPAR<arg_list> RPAR
  | <func_name> LPAR RPAR
  | <func_name> LPAR<error>
;
<collects> ::= <collect_nodes>
  | <collect_ebds>
;
<collect_nodes> ::= COLLECT<orbit> <expr_calc> <orbit>
  | <orbit> UNDER<orbit> LPAR<expr_calc> RPAR
  | LT<orbit_dim_list> PIPE<orbit_dim_list> GT LPAR<expr_calc> RPAR
  | LT<orbit_dim_list> GT LPAR<expr_calc> RPAR
;
<collect_ebds> ::= COLLECT<orbit> UNDER<IDENT> <expr_calc> <orbit>
  | <orbit> UNDER<IDENT> LPAR<expr_calc> RPAR
;
<orbit_dim_list> ::= <orbit_dim_list> COMMA<orbit_dim>
```

```
 | <orbit_dim_list> SEMICOLON<orbit_dim>
 | <orbit_dim_list> <orbit_dim>
 | <orbit_dim>
 | <error>
;
<orbit_dim> ::= <INT>
 | <IDENT>
;
<ruleName> ::= <IDENT>
 | <STRING>
;
<orbit> ::= LT<orbit_dim_list> GT
 | LT GT
 | LT<error> GT
;
<assignement> ::= <assignable> AFFECT<affectable_value>
 | <assignable> <INC>
 | <assignable> <DEC>
;
<declaration> ::= <var_type> <IDENT>
 | <var_type> <IDENT> AFFECT<affectable_value>
 | <var_type> <IDENT> LPAR RPAR
 | <var_type> <IDENT> LPAR<arg_list> RPAR
 | <var_type> <error> AFFECT<affectable_value>
 | <var_type> <error>
;
<assignable> ::= <IDENT>
 | <assignable> LBRACKET<expr_calc> RBRACKET
;
<affectable_value> ::= <condition>
 | <collect_nodes>
 | <rule_application>
 | <rule_alternativ>
 | <collect_ebds>
 | <orbit>
;
<constructor_basic> ::= EBD LT<IDENT> GT LPAR RPAR
 | EBD LT<IDENT> GT LPAR<arg_list> RPAR
 | <IDENT> LPAR<arg_list> RPAR
 | <IDENT> LPAR RPAR
;
<constructor> ::= NEW<constructor_basic>
 | <constructor_basic>
;
<rule_arg_list> ::= <rule_arg_list> COMMA<rule_arg>
 | <rule_arg>
;
<rule_arg> ::= <IDENT> ASSIGN<expr_arg>
 | <condition>
;
<assoc_rule_param> ::= <rule> DOT<IDENT> AFFECT<condition>
;
<condition> ::= TRUE
 | FALSE
```

```
 | <expr_calc>
 | LPAR<condition> RPAR
 | <expr_calc> <COMP> <expr_calc>
 | <expr_calc> LT<expr_calc>
 | <expr_calc> GT<expr_calc>
 | <expr_calc> DIFF<expr_calc>
 | <expr_calc> EQ<expr_calc>
 | NOT<condition>
 | <condition> <binary_bool_op> <condition>
 | <test_mark>
;
<rule> ::= RULE LT<ruleName> GT
 | RULE
 | RULE LT<error> GT
;
<expr_arg> ::= <condition>
 | <orbit>
 | <collect_nodes>
 | <collect_ebds>
;
<sharp_argument> ::= <INT>
 | <IDENT>
 | LPAR<expr_calc> RPAR
;
<expr_topo> ::= MODELER
 | DIMENSION
 | <rule>
 | GMAP
 | GMAP SHARP SHARP
 | GMAP SHARP<sharp_argument>
 | \nERN
 | \nERN  SHARP<sharp_argument> SHARP<sharp_argument>
 | \nERN  SHARP<sharp_argument> LBRACKET<sharp_argument> RBRACKET
 | \nERN  LBRACKET<sharp_argument> COMMA<sharp_argument> RBRACKET
 | RIGHTPATTERN
;
<static_class_attribute> ::= <var_type_common> STATICOP<
    static_class_attribute>
 | <var_type_specific> STATICOP<static_class_attribute>
 | <expr_var>
;
<expr_no_previous> ::= <expr_topo>
 | <expr_no_previous> DOT<expr_var>
 | <constructor>
 | <rule> DOT<func_call>
 | <static_class_attribute>
;
<expr_var> ::= <IDENT>
 | <func_call>
;
<expr_calc> ::= <expr_no_previous>
 | <const_value>
 | LPAR<expr_calc> RPAR
 | <expr_calc> DOT<expr_var>
```

```
 | <expr_calc> <MOD> <expr_calc>
 | <expr_calc> <MULT> <expr_calc>
 | <expr_calc> <XOR> <expr_calc>
 | <expr_calc> <DIV> <expr_calc>
 | <expr_calc> <PLUS> <expr_calc>
 | <expr_calc> <MINUS> <expr_calc>
 | <MINUS> <expr_calc>
 | <PLUS> <expr_calc>
 | MULT<expr_calc>
 | MULT LPAR<expr_no_previous> RPAR
 | AMPERSAND<expr_no_previous>
 | AMPERSAND LPAR<expr_no_previous> RPAR
 | <expr_calc> SHARP<sharp_argument>
 | <expr_calc> ALPHA<sharp_argument>
 | <expr_calc> LBRACKET<expr_calc> RBRACKET
;
<cast_expr> ::= LPAR<var_type> RPAR<expr_calc>
;
<const_value> ::= <const_number>
 | <CHAR>
 | <STRING>
 | NULL
 | NULLUPPER
 | EBD LT<IDENT> GT DOT<IDENT>
;
<const_number> ::= <const_integer>
 | <FLOAT>
 | <DOUBLE>
;
<const_integer> ::= <INT>
;
<arg_list> ::= <arg_list> COMMA<expr_arg>
 | expr_arg: e
;
<var_type_common> ::= <IDENT>
;
<var_type_specific> ::= EBD LT<IDENT> GT
 | EBD LT<IDENT> GT DOT<STRING>
 | EBD
 | LIST LT<var_type> GT
;
<var_type_templated> ::= <IDENT> LT<var_type_list> GT
;
<var_type> ::= <var_type_common>
 | <var_type_specific>
 | <var_type_templated>
 | <var_type> STATICOP<var_type>
;
<var_type_list> ::= <var_type_list> COMMA<var_type>
 | <var_type>
;
<binary_bool_op> ::= <OR>
 | <AND>
;
```

```
<specific_lang> ::= LANG  LPAR<LANGDEF> RPAR<PLAINCODE>
 | LANG<PLAINCODE>
;
<header_lang> ::= HEADER  LPAR<LANGDEF> RPAR<PLAINCODE>
 | HEADER<PLAINCODE>
;
<do_mark> ::= MARK  LPAR<expr_calc> COMMA<IDENT> RPAR
 | MARK   LPAR<collect_nodes> COMMA<IDENT> RPAR
;
<undo_mark> ::= UNMARK  LPAR<expr_calc> COMMA<IDENT> RPAR
 | UNMARK   LPAR<collect_nodes> COMMA<IDENT> RPAR
;
<is_marked> ::= ISMARKED  LPAR<expr_calc> COMMA<IDENT> RPAR
 | ISMARKED   LPAR<collect_nodes> COMMA<IDENT> RPAR
 | <expr_calc> DOLLAR<IDENT>
;
<is_not_marked> ::= ISNOTMARKED LPAR<expr_calc> COMMA<IDENT> RPAR
 | ISNOTMARKED  LPAR<collect_nodes> COMMA<IDENT> RPAR
;
<test_mark> ::= <is_marked>
 | <is_not_marked>
;
```

Résumé

La modélisation géométrique est utilisée dans de nombreux domaines pour la construction d'objets 3D, l'animation ou les simulations. Chaque domaine est soumis à ses propres contraintes et nécessiterait un outil dédié. En pratique, un même outil est utilisé pour plusieurs domaines, en factorisant les caractéristiques communes. Ces modeleurs fournissent un ensemble d'opérations types, que l'utilisateur compose pour construire ses objets. Pour des opérations plus spécifiques, les outils actuels offrent des API.

La plate-forme Jerboa propose un outil de génération d'opérations géométriques personnalisées. Elles sont définies graphiquement par des règles de transformations de graphes. Des vérifications automatiques de préservation de la cohérence des objets sont faites lors de l'édition qui peuvent être enrichies par des propriétés métiers. Notre contribution a consisté à étendre le langage par des *scripts*, pour composer les règles et réaliser des opérations complexes. Nous avons étendu les vérifications automatiques, en particulier pour assurer la cohérence géométrique. Enfin, nous avons modifié le processus d'application des opérations pour augmenter les possibilités de contrôle.

Pour valider cette approche, nous avons développé un modeleur dédié à la géologie, pour la représentation du sous-sol, en collaboration avec l'entreprise Géosiris. Nous avons défini un flux d'activité avec Géosiris en suivant des contraintes spécifiques à la géologie. Grâce à la rapidité de développement des opérations dans Jerboa, nous avons pu prototyper et tester rapidement plusieurs algorithmes de reconstruction du sous-sol, pour les appliquer sur des données réelles fournies par l'entreprise.

Mots-clés: Modélisation géométrique, topologie, transformation de graphe, vérifications, compilation, DSL , géologie

Abstract

Geometric modeling is used in various scopes for 3D object construction, animation or simulations. Each domain must cope with its constraints and should have its dedicated tool. In fact, several common characteristics of different domains are factored in a single tool. These modelers contain sets of basic operations that the user composes to build his objects. For more specific operations, current common tools offer API.

Jerboa's platform allows to generate personalized geometrical operations. These are defined by graph transformation rules. During their design, many automated verifications are done for the preserving of object consistency. They also be enriched with additional properties. Our contribution consists in extending the Jerboa language with *scripts* to compose rules and define complex operations. We also extended automated verifications, in particular to ensure geometric consistaency. Finally, we modified operations application process, in order to increase user control possibilities.

To validate this approach, we have implemented a geological dedicated modeler for subsoil modeling, in collaboration with Geosiris Company. We defined a workflow with Geosiris that follows specific geological reconstruction constraints. Thanks to the Jerboa rapid prototyping mecanism, we developped and quickly tested several subsoil reconstruction algorithms, and apply them to real data provided by the company.

Keywords: Geometric modeling, topology, graph transformation, verifications, compilation, DSL, geology