Chapitre 7

Travaux connexes et développements

Durant ce travail de thèse, nous avons eu l'occasion d'aborder plusieurs projets liés à notre travail de recherche. Nous présentons ici quelques-uns de ces travaux. Le premier projet, présenté section 7.1, concerne le développement d'un noyau géométrique à base topologique. Ce noyau s'intègre dans le cadre du projet RNTL[1] *Nogémo*. Le but de ce projet est de concevoir et de développer un noyau géométrique modulaire pour modeleurs et simulateurs. Ce noyau est au cœur de différents travaux. Nous avons travaillé sur deux d'entre eux, un modeleur géométrique à base topologique, *Moka*, présenté section 7.2, et un projet d'analyse d'images pour une aide à la détection de tumeurs cérébrales présenté section 7.3.

Section 7.4 nous présentons une étude sur l'intégration de la carte topologique dans un processus de segmentation markovienne. Ce travail montre que la carte topologique peut être intégrée dans des algorithmes de segmentation. Nous avons également travaillé sur quelques opérations de modification de la carte topologique 2d et 3d, principalement les opérations de fusion et de coupure. En effet elles constituent les opérations de base pour plusieurs algorithmes de segmentation. De plus, nous avons étudié une opération intéressante, le *coraffinement*, qui permet d'envisager l'implantation d'algorithmes de segmentation en parallèle sur différents « morceaux » d'images, l'image totale étant ensuite recomposée au moyen de cette opération. Ces opérations sont présentées brièvement section 7.5. Enfin, nous avons étudié différentes possibilités pour implanter efficacement la carte topologique, tout en conservant un programme modulaire et extensible. Cette réflexion est présentée section 7.6.

7.1 Un noyau générique de 3-G-cartes

Ce noyau est la base du projet RNTL *Nogémo*. Il a été initialement développé par Allan Fousse et Daniel Menevaux, maîtres de conférences au laboratoire IRCOM-SIC. Nous l'avons ensuite entièrement repris, afin de le stabiliser, de l'homogénéiser, de l'optimiser

[1]Réseau National des Technologies Logicielles.

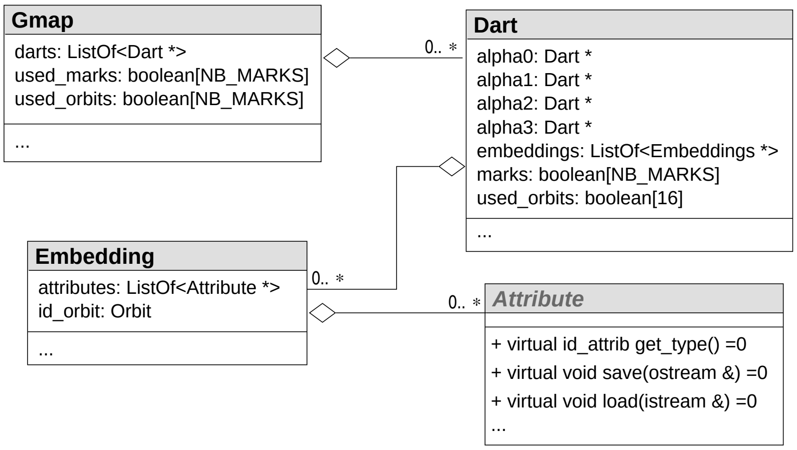


Fig. 7.1 – Le schéma UML du noyau de 3-G-carte, représentation partielle.

mais également de le valider. Ce noyau étant au centre d'un projet de plate-forme logicielle, il se doit d'être le plus générique possible. En effet, les différentes applications pouvant l'utiliser ont toutes des besoins spécifiques différents. Dans sa version actuelle, ce noyau est basé sur les 3-G-cartes. Il permet donc la modélisation de quasi-variétés orientables ou non de dimension inférieure ou égale à trois. Il a été développé en C++, afin d'obtenir un code facilement modifiable, et extensible.

Nous expliquons brièvement la structure globale de ce noyau. Il comprend trois classes principales : la classe *Gmap* est la classe de base permettant de déclarer une 3-G-carte, la classe *Dart* représente un brin, et enfin la classe *Attribute* représente un attribut associé à une orbite particulière. Cet attribut peut être géométrique, comme les coordonnées d'un point 3d que nous associons à une orbite sommet, ou une 2-G-carte que nous associons à un plongement face, mais également d'autres attributs de couleur, texture... Ce noyau permet d'associer à n'importe quelle orbite de la 3-G-carte n'importe quel type d'attribut. De plus, quel que soit le plongement et le type d'attribut, nous avons implanté les coutures et décousures afin qu'elles mettent à jour ces attributs lorsque cela est nécessaire.

Un attribut spécifique est associé à une orbite particulière. Plusieurs attributs différents peuvent être associés à une même orbite. Par exemple, nous pouvons associer à une face une surface 2d, une couleur et une texture. Ces attributs sont regroupés dans la classe *Embedding*. Nous pouvons voir figure 7.1 le schéma représentant ces classes, et la manière dont elles sont reliées. Sur ce schéma UML, nous avons représenté seulement les principaux champs afin de ne pas entrer dans les détails techniques.

La classe *Gmap* est principalement composée d'un ensemble de brins, mais possède également d'autres attributs. Le champ *used_orbits* est un tableau de 16 booléens, qui permet pour chaque orbite de savoir si elle est plongée ou non. Cela permet de ne pas effectuer de parcours afin de chercher un plongement, lorsque nous savons que cette orbite n'est pas plongée. Enfin, le champ *used_marks* donne les marques booléennes qui sont en cours d'utilisation. La classe *Gmap* possède des méthodes permettant de demander et de réserver une marque libre, et de libérer une marque. Ces deux méthodes vont utiliser et mettre à jour ce tableau. Cette classe possède un grand nombre de méthodes, permettant de créer ou supprimer des brins, de coudre ou découdre ces brins, avec ou sans mise à jour des plongements, des méthodes pour affecter, supprimer ou récupérer un *Attribut* pour une orbite particulière. D'autres méthodes permettent de tester si un brin est marqué, de le marquer ou le démarquer. Ce sont les principales méthodes que l'utilisateur peut appeler. Il existe bien entendu plusieurs autres méthodes, certaines étant privées et utilisées en interne pour par exemple mettre à jour les plongements, ou tester si deux brins appartiennent à une même orbite...

La classe *Dart* possède quatre pointeurs afin de représenter les quatre involutions α et une liste d'*Embedding* portées par ce brin. Chaque *Embedding* correspond à une orbite particulière. Le tableau de booléens *used_ orbit* permet de savoir en *O(1)* si ce brin porte un *Embedding* pour une orbite. Cela évite de parcourir la liste *Embedding* inutilement. Enfin le tableau *marks* contient les marques booléennes de ce brin.

La classe *Embedding* contient un champ *id_ orbit* correspondant à un identificateur de l'orbite correspondant à cet *Embedding*. Elle contient ensuite la liste des *Attribute* contenu dans cet *Embedding*. La classe *Attribute* est une classe virtuelle pure. L'utilisateur désirant un attribut particulier va créer sa classe dérivant de *Attribute*, et fixer son « comportement ». Il faut, entre autres, donner un identifiant différent à chaque *Attribut*, définir les méthodes *Save* et *Load*... Cela permet par exemple d'écrire une méthode de sauvegarde générique dans la classe *Gmap*, qui va utiliser la méthode *Save* des classes *Attribut* redéfinies par l'utilisateur. Nous avons également implanté toutes les classes *Coverage* permettant de parcourir les brins de chaque orbite possible. Ces classes sont implantées « à la STL », comme des itérateurs. Cela permet par exemple d'utiliser l'opérateur ++ pour passer au brin suivant du parcours.

Nous ne détaillons pas plus les fonctionnalités de ce noyau. Son principal atout est d'être très générique, nous n'avons fixé aucune contrainte sur son utilisation, à l'exception de la dimension. De plus, il gère de manière transparente l'utilisation de n'importe quel plongement, ce qui permet à un utilisateur non spécialiste de l'utiliser sans se préoccuper de cet aspect. Ses deux défauts principaux sont sa lenteur et l'espace mémoire occupé. En effet, de par sa généricité, il effectue beaucoup de tests afin de mettre à jour tous les éventuels plongements, et la structure en liste de listes pour les attributs est coûteuse en espace mémoire. Mais ce noyau peut être considéré comme un prototype. Lorsqu'une application spécifique nécessite un plongement particulier, il est possible de spécialiser le noyau pour tenir compte de ce plongement, et ainsi supprimer des tests, et les listes d'attributs. Nous obtenons alors un noyau spécialisé, pour lequel nous ne pouvons plus plonger n'importe quelle orbite, mais moins coûteux en temps d'exécution et en espace mémoire. Ce changement de noyau pourra être effectué sans aucune modification des sources de l'application, simplement en conservant la même interface entre la version générale et la version spécialisée.

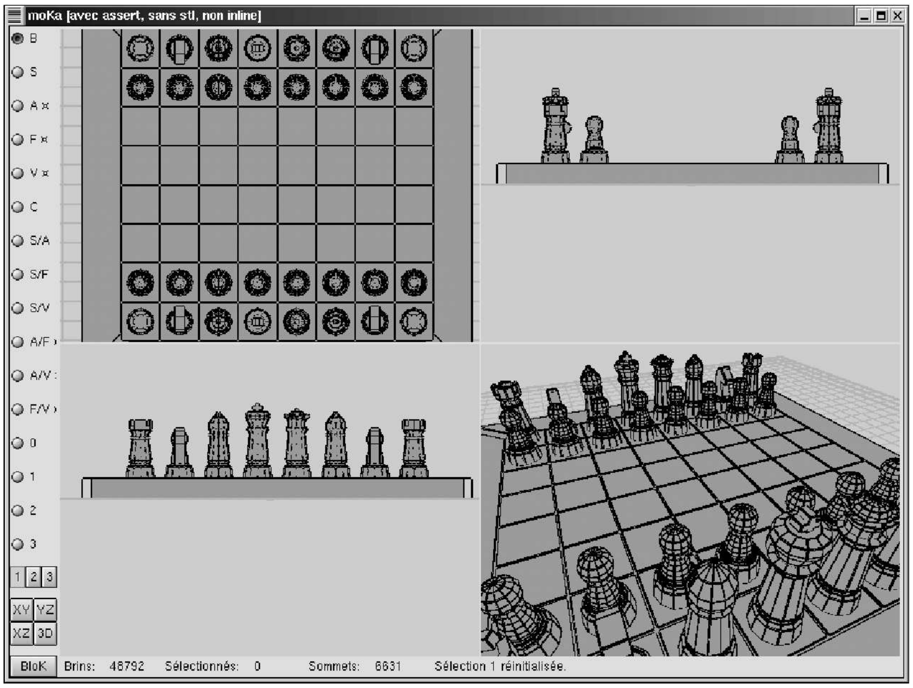


FIG. 7.2 – Une photo d'écran de Moka.

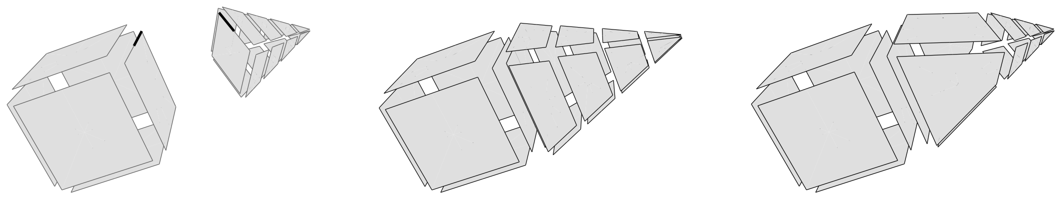
7.2 Moka : un modeleur géométrique à base topologique

Moka[2] est un modeleur géométrique à base topologique. Il a été développé par Frédéric Vidil lors de plusieurs stages que nous avons encadrés et co-encadrés. Durant ces mois de travail, nous avons participé au développement de certaines fonctions spécifiques, comme par exemple les opérations de fusion, bouchage, triangulation ou l'export xfig[3].

Ce modeleur est basé sur le noyau présenté section précédente. De ce fait, nous avons pu profiter de l'ensemble des fonctions déjà existantes, et nous consacrer ici uniquement à la couche application et aux fonctions de haut niveau. La figure 7.2 présente une photo d'écran de Moka. Nous pouvons voir que ce modeleur possède 4 fenêtres de visualisation différentes : 3 vues 2d sur les différents plans possibles, et une vue 3d. Il est possible de basculer l'affichage sur une seule de ces vues. L'utilisateur a à sa disposition un certain nombre d'objets qu'il peut créer, combiner et déformer à sa guise. Ces objets de base sont de dimension 2, par exemple des lignes brisées, des polygones réguliers ou non, ou des objets 3d comme des cubes, sphères, tores, pyramides, cônes. Tout les différents paramètres de ces objets sont paramétrables, comme par exemple le nombre de subdivisions de la sphère sur les parallèles et les méridiens, la position du centre, la direction...

Il existe de nombreuses opérations, les deux principales étant bien entendu la couture et la décousure. Il est possible d'effectuer ces deux opérations pour toutes les dimensions,

[2]Pour modeleur de cartes.
[3]La majorité des figures de cette thèse sont d'ailleurs réalisées avec ce modeleur et cette fonction.



a. Deux objets. b. Après la couture géométrique. c. Après la couture topologique.

Fig. 7.3 – La couture topologique et géométrique.

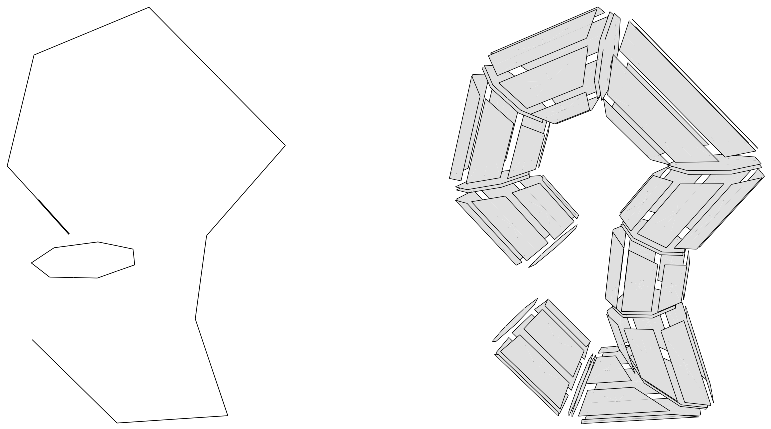
mais également de manière « intuitive » où la dimension est calculée automatiquement, suivant les dimensions déjà cousues. La couture peut être réalisée de manière topologique ou géométrique, comme nous pouvons voir figure 7.3. La couture géométrique déforme le deuxième objet, par translation, rotation et homothétie, afin que la face cousue soit de la même géométrie que la face du premier objet. Nous voyons figure 7.3.b que cette couture permet de « plaquer » les deux objets. La couture topologique (cf. figure 7.3.c) n'effectue aucune modification géométrique. La face cousue du deuxième objet récupère la géométrie de la face du premier objet de manière automatique, car les orbites sommets sont fusionnés, mais le reste de l'objet ne subit aucune déformation géométrique.

Quelques-unes des autres opérations existantes sont le bouchage, la triangulation, la quadrangulation, la fusion, l'insertion de cellules, différents maillages, différentes extrusions... La plupart de ces opérations fonctionnent pour n'importe quelle dimension. Par exemple la triangulation est développée en dimension 1, ce qui revient à couper un segment en deux, en dimension 2 pour trianguler une face, et en dimension 3 pour trianguler un volume en plusieurs tétraèdres. Nous pouvons voir figure 7.4 un exemple d'extrusion d'une face le long d'un chemin. Cette opération permet de construire un volume en faisant « courir » la face le long du chemin.

Ce modeleur continue a être développé, afin d'intégrer d'autres opérations plus complexes, dont certaines sont primordiales dans le monde des CAO : principalement le chanfreinage et les opérations booléennes, mais également les plongements splines, nurbs... De plus, c'est ce modeleur qui va être la base de travail pour la réalisation du projet Nogémo évoqué section 7.1. Mais la version actuelle est déjà opérationnelle, validée d'ailleurs par sa grande utilisation dans cette thèse, et possède suffisamment d'opérations pour modeler des scènes élaborées.

7.3 Projet de détermination du volume tumoral cérébral

Ce projet a été réalisé par Cyrius Cayrous [Cay01] lors de son stage de DESS que nous avons co-encadré. L'objectif de ce stage était la réalisation d'une interface intégrant des outils de segmentation et de visualisation volumique sur des coupes IRM de cerveau. La réalisation de cette application s'inscrit dans le cadre d'un projet de détermination du



a. Une face et un chemin. b. Après l'extrusion.

Fig. 7.4 – L'opération d'extrusion d'une face le long d'un chemin.

volume tumoral et de ses variations lors du suivi évolutif. L'outil informatique à développer a pour but d'aider le médecin dans son choix de stratégie thérapeutique.

Nous avons tout d'abord intégré des algorithmes de segmentation [CAFMF01] au cœur d'une interface graphique permettant de charger un ensemble de coupes scanner, et de les segmenter en autorisant bien entendu le réglage de tous les paramètres de cette segmentation. Nous avons également développé certaines opérations en 2d, comme la possibilité de visualiser l'ensemble des coupes sous forme de mosaïque, de modifier les paramètres de couleur et contraste, afin de faire ressortir certaines régions de l'image...

Puis, nous avons intégré l'algorithme optimal d'extraction de la carte de niveau n à partir d'une image segmentée, présenté section 5.6. Étant donné le faible temps imparti, nous avons implanté uniquement le niveau 3 de simplification. En effet, le nombre de traitements différents à écrire pour traiter les précodes de ce niveau est beaucoup moins important que pour la carte topologique. Cette application montre l'avantage de nos différents niveaux de simplification qui offrent le choix aux utilisateurs de travailler avec un niveau particulier. Ce développement est également basé sur le noyau de 3-G-cartes présenté section 7.2. De ce fait, il nous a fallu étudier les traitements des précodes sur les G-cartes, étant donné que le travail présenté section 5.6 utilise les cartes combinatoires. Mais cette conversion s'effectue sans difficulté particulière. Après avoir construit cette carte, nous avons ensuite implanté des opérations en dimension 3, principalement différentes possibilités de visualisation.

La figure 7.5 montre une capture d'écran du logiciel développé. Cette figure présente le mode 3d, sur lequel nous voyons la carte des frontières reconstruite à partir des coupes IRM préalablement segmentée. Nous avons sélectionné sur une de ces coupes une région, qui est présentée en faces pleines en 3d. Nous pouvons indifféremment visualiser seulement cette région, en faces pleines ou non, ou toutes les régions et cette région sélectionnée en faces pleines. Ce dernier mode de visualisation permet de visualiser la position de cette région par rapport à l'image totale. Cet affichage représente pour le moment les faces en

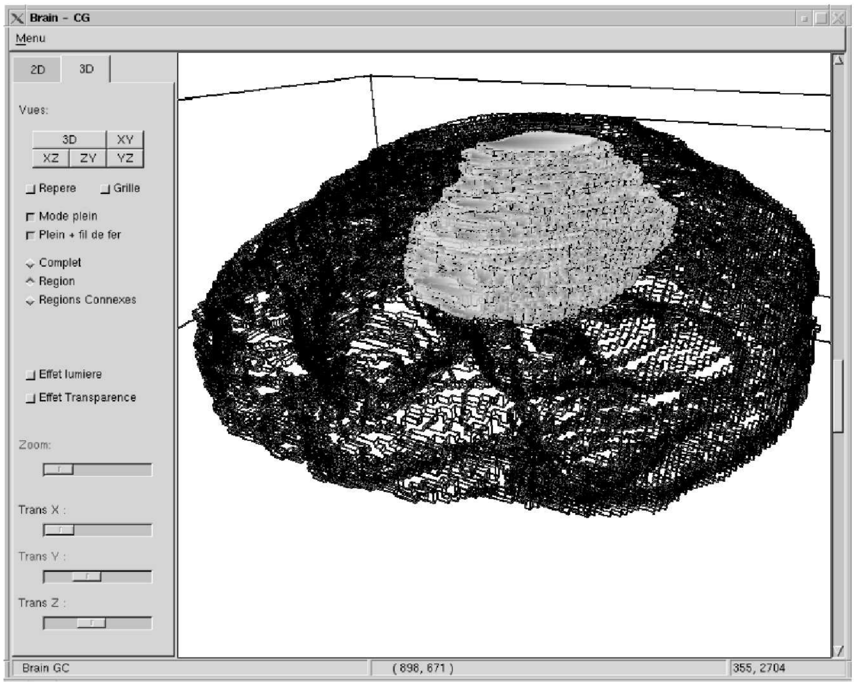


FIG. 7.5 – Capture d'écran du logiciel de détermination du volume tumoral cérébral.

intervoxel, ce qui explique le rendu « cubique » des objets. Nous envisageons l'implantation d'algorithmes de lissage afin d'obtenir un meilleur rendu pour l'affichage.

Ce travail peut être considéré comme un prototype, permettant de montrer quelques fonctionnalités en 3d utilisant la carte combinatoire reconstruite. Mais nous envisageons de nombreuses autres opérations afin de faciliter le travail du praticien, par exemple la modification interactive de la segmentation lorsqu'il n'est pas satisfait de cette dernière. Pour cela, il faut pouvoir fusionner certaines régions, ou au contraire relancer la segmentation seulement sur certaines régions en modifiant les paramètres. De plus, nous devons définir les opérations permettant le calcul volumétrique d'une région sélectionnée de manière précise. En effet, la version actuelle se contente de compter les voxels de cette région, et d'approximer le volume en connaissant les caractéristiques de l'examen. Même si cela permet une première estimation, ce calcul n'est pas très précis et mérite d'être amélioré.

7.4 Intégration de la carte topologique 2d dans un processus de segmentation markovienne

Ce travail est le résultat du stage de recherche de DEA effectué par Pascal Bourdon [Bou01] que nous avons co-encadré. La segmentation par champs de Markov est une des approches souvent utilisées dans le monde du traitement des signaux et des images. C'est une méthode dite *stochastique*, qui consiste à considérer l'image comme un processus aléatoire discret paramétrable dont nous tentons d'estimer les paramètres associés à chaque région. Cette approche permet d'intégrer facilement les critères d'appartenance à une région sous forme de *potentiels*.

L'idée de ce travail de recherche consiste à intégrer la carte topologique dans le processus de segmentation markovienne, afin de pouvoir ainsi définir des potentiels calculés sur cette carte. Pour cela, nous avons tout d'abord développé l'algorithme classique de segmentation markovienne, sans utiliser la carte topologique. Afin d'intégrer ensuite ces dernières, nous avons implanté l'algorithme optimal d'extraction de la carte topologique à partir d'une image. La partie importante de recherche a été la définition de nouveaux potentiels pouvant se calculer sur cette carte. De plus, il nous a fallu étudier la manière dont ces nouveaux potentiels peuvent s'intégrer aux potentiels classiques.

Nous avons défini un potentiel *taille* permettant de défavoriser la création de région de petite taille. Ce potentiel peut se calculer très simplement et efficacement sur la carte topologique. Les résultats obtenus sont encourageants et montrent que l'intégration de ce potentiel améliore la segmentation, en limitant effectivement la création de petites régions, qui sont souvent le résultat d'une sur-segmentation. Les différents problèmes techniques rencontrés lors de ce travail nous ont empêchés de tester d'autres potentiels. Mais ces premiers résultats montrent la faisabilité de cette nouvelle approche, tant pour le domaine du traitement des signaux et des images que pour celui de l'infographie ou de l'imagerie combinatoire. Nous envisageons de poursuivre ce travail lors d'un second stage de DEA. Nous pouvons maintenant étudier directement les potentiels calculés sur la carte topologique, étant donné que les problèmes techniques ont déjà été résolus. Les possibilités sont nombreuses : nous envisageons des potentiels de rugosité, de forme, de relation de voisinages... Le calcul de ces critères sera plus facile et efficace grâce à la carte topologique.

7.5 Quelques opérations sur la carte topologique 2d et 3d

Ce travail est le résultat du stage de recherche de DEA effectué par Patrick Resch [Res01a, Res01b] que nous avons encadré. Le but de ce stage était de définir les opérations de fusion et de coupure sur la carte topologique 3d. Ces opérations sont en effet la base de nombreux algorithmes de segmentations en dimension 2, et donc de futurs algorithmes de segmentations 3d. De plus, ces deux opérations sont également importantes dans une optique interactive de contrôle d'une segmentation. En effet, la plupart des outils de segmentation doivent autoriser le contrôle, et éventuellement la modification interactive d'une segmentation automatique. Un expert doit en effet obligatoirement valider la segmentation dans les domaines sensibles, comme par exemple l'imagerie médicale, et pouvoir fusionner deux régions s'il juge qu'elles ont été sur-segmentées, ou au contraire couper une région lorsqu'il le désire.

Au cours de ce travail, nous nous sommes également intéressé à une autre opération, dans une optique de parallélisation de l'algorithme de segmentation : le *coraffinement*. L'idée consiste à couper une grosse image en plusieurs petits blocs, segmenter ensuite chaque bloc séparément en parallèle, puis reconstruire la segmentation de l'image globale en « recollant » chaque morceau d'image. L'opération effectuant ce recollement est le coraffinement. Ce principe permet d'envisager la segmentation de grosses images. En effet, les algorithmes de segmentation sont souvent gourmands en temps de calcul, et la taille des données à traiter est très importante en 3d.

Afin de définir ces trois opérations en dimension 3, nous avons tout d'abord commencé à les étudier en dimension 2, de manière similaire à la démarche utilisée pour ce travail de thèse. En effet, les problèmes de visualisation de la carte topologique 3d freinent considérablement sa compréhension. Le fait de repartir en 2d permet de bien comprendre les problèmes des différentes opérations, et de mieux envisager ces mêmes problèmes en passant à la dimension supérieure. De plus, nous avons réutilisé certaines techniques 2d pour certains traitements spécifiques 3d.

Nous avons défini ces opérations de manière locale, chaque fois que cela était possible. Cette technique permet de limiter le nombre de cas à traiter, et simplifie donc les algorithmes. De plus, nous avons également préféré une légère perte en complexité au profit d'une plus grande simplicité des algorithmes, afin d'en faciliter la compréhension. En effet, de nombreux points sont assez techniques et l'écriture optimale de l'opération devient vite incompréhensible. Nous ne détaillons pas ici ces algorithmes car cela demanderait un chapitre complet, mais le lecteur intéressé pourra se référer à [Res01a, Res01b, DR02]. Nous allons maintenant étudier comment intégrer ces algorithmes dans un processus de segmentation 3d.

7.6 Implantation optimisée de la carte topologique 2d et 3d

Afin d'implanter la carte topologique ainsi que les différents algorithmes d'extraction, nous avons développé un programme permettant de la représenter et de la manipuler. Étant donné les plongements différents qu'il est possible d'utiliser, ce programme devait être modulaire afin de pouvoir changer facilement de plongement sans trop de problème. Nous avons pour cela choisi le langage C++ afin de réaliser un modèle objet facilement extensible. Mais nous avons un problème d'optimisation, tant au niveau de l'espace mémoire que du temps d'exécution. En effet, de par la taille importante des données à traiter, principalement en dimension 3, il faut minimiser l'espace mémoire utilisé par l'application afin de pouvoir travailler sur des images de taille raisonnable. Mais nous avons également un problème de complexité en temps d'exécution. Si l'algorithme d'extraction demande trop de temps, il ne sera pas utilisable dans des domaines applicatifs traitant les images dans des délais assez courts. Ces trois problèmes sont souvent en opposition. Un code C++ modulaire est souvent plus lent en temps d'exécution qu'un code C totalement optimisé, et l'amélioration du temps d'exécution se fait souvent au détriment de l'espace mémoire. Nous avons donc essayé de trouver un compromis entre ces trois contraintes, mais en privilégiant légèrement l'espace mémoire car nous avons jugé que c'était la contrainte la plus importante.

Nous ne détaillons pas précisément cette étude, mais en donnons seulement les idées principales. Le lecteur intéressé pourra se reporter au rapport de recherche [DF99]. La première constatation que nous avons faite est que l'utilisation des méthodes virtuelles est coûteuse en temps d'exécution et en espace mémoire. En effet, chaque classe possédant une méthode virtuelle réserve un pointeur vers la table des méthodes virtuelles afin de résoudre les appels à ces méthodes de manière dynamique. Une classe ayant une méthode virtuelle a 4

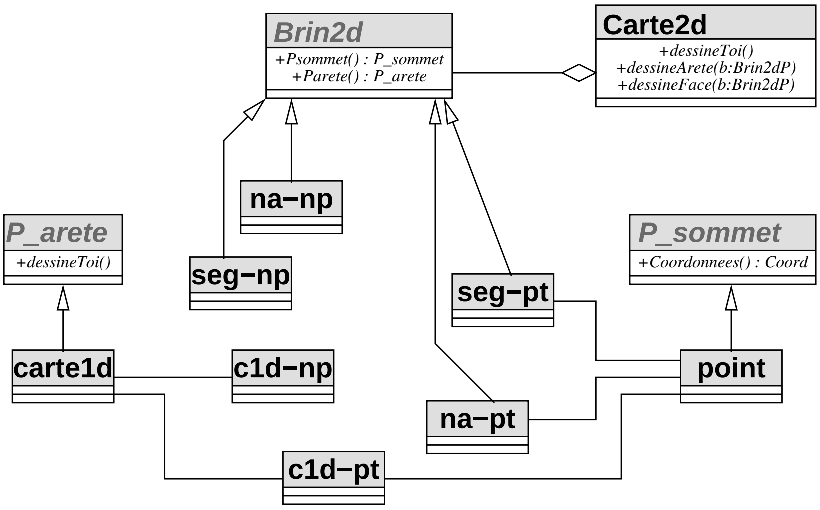


FIG. 7.6 – Le modèle objet de notre implantation de la carte topologique 2d.

octets de plus que la même classe sans méthode virtuelle. De plus, les appels aux méthodes virtuelles sont résolus dynamiquement, ce qui entraîne une indirection supplémentaire, mais surtout interdit le mécanisme d'*inline*. Une méthode *inline* est recopiée à l'endroit où elle est appelée, ce qui évite un appel de fonction ainsi que l'empilement des paramètres. De ce fait, un appel à une méthode *inline* est moins coûteux en temps qu'un appel à une méthode non *inline*. Ces deux raisons font qu'un appel à une méthode virtuelle prend plus de temps qu'un appel à une méthode non virtuelle. Mais les méthodes virtuelles sont les seules possibilités, en C++, d'obtenir un code modulaire. Il faut donc jongler entre les deux aspects, et définir certaines méthodes virtuelles lorsqu'elles sont susceptibles d'être redéfinies, et les autres non virtuelles afin qu'elles soient moins coûteuses en temps d'exécution. Les différents tests que nous avons effectués entre une implantation où toutes les méthodes sont virtuelles, et une deuxième où seules un petit nombre le sont montrent que le temps d'exécution de la première solution est en moyenne deux fois plus important que pour la deuxième implantation.

Nous avons ensuite étudié la manière de représenter les brins par rapport aux différents plongements que nous utilisons. En dimension 2, nous avons implanté le plongement « arête ouverte et sommet » présenté section 4.3, afin de ne pas représenter un même sommet géométrique plusieurs fois. L'implantation directe de ce plongement consiste à ajouter un pointeur par brin pour le plongement arête, et un pointeur par brin pour le plongement sommet. Mais un seul brin par orbite sommet désigne effectivement le plongement correspondant, et un seul brin par orbite arête désigne le plongement arête. Les autres brins n'utilisent pas ces pointeurs, et l'espace mémoire correspondant est inutile. Afin de résoudre ce problème de perte d'espace mémoire, nous avons utilisé pleinement les mécanismes objets, et implanté le modèle objet présenté figure 7.6.

La classe *Carte2d* est un ensemble de *Brin2d*. Cette classe *Brin2d* est virtuelle pure et n'a aucun attribut. Elle possède des méthodes permettant de récupérer le plongement arête et le plongement sommet qui sont virtuelles pures. Les deux classes virtuelles pures *P_arete* et *P_sommet* représentent les plongements arêtes et sommets. Le fait d'avoir ces classes génériques autorise le développeur à choisir la manière dont il désire réaliser ces plongements. Nous plongeons les sommets par un point géométrique, et les arêtes par des cartes 1d. Nous dérivons ensuite 6 classes de brins différentes de la classe *Brin2d*. Ces six classes s'expliquent par l'ensemble des possibilités de plongement d'un brin. Il peut porter un plongement sommet ou non, porter un plongement arête ou non. Dans ce dernier cas, soit le plongement arête est un segment, soit c'est une courbe ouverte. Nous différencions ces deux cas car une arête ayant pour plongement un segment ne pointe pas vers une 1-carte de plongement, étant donné que ces cartes sont ouvertes. Ces six classes ont pour nom sur le modèle objet de la figure 7.6 *seg-np*, *seg-pt*, *c1d-np*, *c1d-pt*, *na-pt* et *na-np*. La première partie du nom concerne le plongement arête : *seg* pour plongé segment, *c1d* pour plongé carte 1d, *na* pour non plongé arête ; et la deuxième partie concerne le plongement sommet : *pt* pour plongé sommet et *np* pour non plongé sommet. Six est donc le nombre de possibilités de combiner un des trois plongements arêtes avec un des deux plongements sommets. Grâce à ces six classes, chaque brin de la carte possède exactement les attributs qui lui sont nécessaires, il n'y a plus aucun attribut inutile et l'espace mémoire utilisé est donc optimal.

Ces six classes de brins définissent chacune les méthodes permettant de récupérer le plongement sommet et le plongement arête, qui suivant les cas (que nous ne détaillons pas ici) retourneront l'attribut porté par le brin, ou appelleront la méthode d'un autre brin appartenant à la même orbite. Cette implantation privilégie l'espace mémoire au détriment d'une légère perte en temps d'exécution. En effet, les méthodes permettant d'affecter un plongement à un brin devront, suivant le type de ce brin, créer un nouveau brin d'un autre type afin de pouvoir effectuer cette affectation. Cela entraîne donc une allocation mémoire suivie d'une désallocation. Ces modifications de type de brins se font de manière transparente sans que l'utilisateur ait besoin de les gérer. De plus, suivant les besoins en espace ou en temps, il est possible d'effectuer des compromis afin de faire moins de changements de type de brins en regroupant certaines classes de brins, ce qui entraîne donc une légère perte en espace mémoire. Nous avons comparé cette implantation avec l'implantation classique de ce même modèle, ayant une seule classe brin avec deux pointeurs vers les éventuels plongements. Les résultats obtenus montrent un gain de l'ordre de 30% en espace mémoire, ce qui est très important pour de grosses images. De plus, la perte en temps d'exécution est minime, de l'ordre de 1%.

Pour l'implantation de la carte topologique 3d, nous n'avons pas développé le même modèle objet car nous n'avons pas implanté le plongement surface ouverte, arête ouverte et sommet. Pour réaliser cette implantation, nous aurions alors 18 classes de brins représentant toutes les combinaisons possibles entre les trois plongements. Mais comme pour la dimension 2, il est envisageable de regrouper certaines de ces classes, afin de diminuer leur nombre ainsi que les modifications de type, mais au prix d'une perte en espace mémoire. Pour implanter le plongement uniquement face, nous avons seulement deux types de brin suivant s'il pointe vers un brin d'une 2-G-carte ou non. Dans ce deuxième cas, c'est que le brin lui étant β_3-cousu porte cette information, que nous pouvons alors retrouver sans problème. Afin de représenter les 2-G-cartes de plongements, nous avons utilisé le même principe que pour la carte topologique 2d, et implanté les 6 classes de brins afin que ces cartes soient minimales en espace mémoire.

Nous ne détaillons pas plus les différents tests que nous avons effectués afin de comparer différentes implantations, des solutions intermédiaires, avec ou sans méthodes virtuelles, avec des allocateurs par blocs... Cette étude permet de comprendre un peu mieux les mécanismes objets du C++ et permet d'implanter la carte topologique de manière efficace dans ce langage. Bien entendu cette implantation ne peut pas être aussi efficace qu'une implantation « brutale » en langage machine, mais nous obtenons un bon compromis entre efficacité en espace, en temps et modularité. C'est bien notre objectif initial, afin d'obtenir un programme facilement extensible et modifiable.

7.7 Conclusion

Nous avons présenté dans ce chapitre quelques-uns de nos différents travaux, réalisés dans des domaines proches de notre cadre de recherche. Ces travaux ouvrent tous des perspectives de recherche ou des possibilités de développement logiciel. De plus, certains de ces travaux ont été réalisés en collaboration avec des chercheurs en traitement du signal, domaine que nous ne connaissions que très peu, et qui s'est avéré très enrichissant.

Moka, bien que n'étant pas totalement achevé, est opérationnel et laisse entrevoir de nombreuses possibilités d'évolutions. De plus, la plate forme logicielle qui va être développée autour du noyau doit intégrer de nombreuses applications différentes. Des logiciels de CAO, inspirés plus ou moins directement de Moka, des logiciels d'analyses d'images 3d, des logiciels de segmentation utilisant le modèle des cartes topologiques, un modeleur discret...

Pour la partie opérations sur la carte topologique, nous avons jusqu'ici étudié trois opérations de modification, mais nous envisageons maintenant de nombreuses autres possibilités. Ce travail est nécessaire afin d'offrir de nombreux outils aux utilisateurs éventuels de ce modèle. Une perspective nous tenant particulièrement à cœur est le développement d'une branche de Moka travaillant avec des cartes topologiques. Cela pose quelques problèmes, étant donné que nous avons défini ce modèle dans le cadre de la représentation d'images segmentées, ce qui n'est plus le cas si nous travaillons maintenant en interactif. Mais les modifications à apporter doivent être pas trop importantes et ce projet doit donc être réalisable.

Notre étude sur l'implantation de la carte topologique est primordiale afin de pouvoir obtenir un programme efficace en temps d'exécution et n'occupant pas un espace mémoire « trop » important, tout en étant modulaire et extensible. Ces trois problématiques sont en opposition, mais notre solution consiste en des compromis entre ces trois aspects. Suivant les besoins spécifiques d'une application, il est possible de privilégier la complexité en temps ou en espace mémoire, en utilisant des solutions intermédiaires.