



HAL
open science

Introduction d'une vue textuelle synchronisée avec la vue géométrique primaire dans Cabri-II

Valérie Bellynck

► **To cite this version:**

Valérie Bellynck. Introduction d'une vue textuelle synchronisée avec la vue géométrique primaire dans Cabri-II. Education. Université Joseph-Fourier - Grenoble I, 1999. Français. NNT : . edutice-00000206

HAL Id: edutice-00000206

<https://theses.hal.science/edutice-00000206>

Submitted on 14 Nov 2003

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE
présentée par

Valérie BELLYNCK

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
(ARRÊTÉS MINISTÉRIELS DU 5 JUILLET 1984 ET
DU 30 MARS 1992)

Spécialité
INFORMATIQUE

INTRODUCTION D'UNE VUE TEXTUELLE SYNCHRONISÉE
AVEC LA VUE GEOMETRIQUE PRIMAIRE
DANS LE LOGICIEL CABRI-II

29 octobre 1999

Jury :

M. Jean-Pierre	PEYRIN	Président
M. Richard	ALLEN	Rapporteur
Mme Monique	GRANDBASTIEN	Rapporteur
M. Nicolas	BALACHEFF	Examinateur
M. Bertrand	IBRAHIM	Examinateur
M. Vincent	QUINT	Examinateur
M. Jean-Marie	LABORDE	Directeur

THÈSE PRÉPARÉE AU SEIN DU LABORATOIRE LEIBNIZ (IMAG, UJF, INPG & CNRS)

Remerciements

La « page » de remerciements est quelque chose de difficile pour moi, encore plus aujourd'hui. Trouver le bon ton, en dire un peu mais pas trop, l'élan du cœur quand on s'en va et qu'on dit au revoir pour ne pas dire adieu.

Pourquoi s'expliquer avant de remercier et pas simplement remercier ?

Par ces quelques mots, je voudrais remercier chacun, avec une pensée particulière. Mais il est des moments de la vie où toutes les choses sont si difficiles qu'il est même gênant pour ceux qui sont remerciés de recevoir ces remerciements par quelqu'un que trop de problèmes accablent.

Dans ces quelques années, j'ai trouvé des amis, de véritables amis, autant que la vie peut en donner en quelques années, et c'est une grande chance. Certains le sont devenus résolument. D'autres hésitent, tout comme moi peut-être un peu frileuse. Il n'y a pas que ceux qui m'ont offert leur amitié et que je ne pourrais jamais assez remercier, mais aussi tous ceux qui ont simplement été présents, honnêtes avec eux-mêmes et avec les autres, faisant de leur mieux, chacun selon ses affinités et capacités. À tous ceux-là, je voudrais dire merci, car c'est grâce à eux que j'ai pu avancer et finalement aboutir. Et puis il y a ceux qu'on ne peut pas remercier, pour rien, rien, rien. Donc on n'en parlera pas.

Alors moi, j'ai à remercier :

Mon directeur de thèse, Jean-Marie Laborde, concepteur du logiciel Cabri-géomètre, dont la confiance m'a encouragée tout au long de ces trois années, tout en me laissant libre d'explorer.

Le responsable de mon équipe de travail, Nicolas Balacheff, toujours disponible et prêt à apporter son aide.

Richard Allen et Monique Grandbastien, qui ont accepté la lourde tâche de rapporter sur mon manuscrit.

Bertrand Ibrahim, Jean-Pierre Peyrin et Vincent Quint, qui ont bien voulu participer à ce jury.

Je voudrais aussi que tous mes collègues et anciens collègues sachent que j'ai une pensée sincère pour chacun d'entre eux.

Je voudrais remercier plus particulièrement deux d'entre eux qui comptent aujourd'hui parmi mes meilleurs amis, et une autre amie d'avant et d'ailleurs. Ils m'ont accompagnée dans mes moments si difficiles que je ne peux en parler plus ici. La chaleur de leur amitié m'a aidé à ne pas plier.

Et puis il y a ma famille, l'ancienne, originelle, à qui je dois tout ce que je suis ; celle que j'ai construite, réduite à mes deux garçons que je voudrais tant porter vers le monde mais dont l'absence m'est si cruelle ; et celle de demain, dont chacun, je crois, m'a fait le présent de m'accepter.

Enfin il y a Christian, grâce à qui ma vie d'aujourd'hui est faite du meilleur alors qu'elle aurait pu n'être faite que du presque pire.

septembre 1999

-0-0-0-0-0-0-0-0-0-

Introduction

Cabri-géomètre est un logiciel de géométrie dynamique constituant un micromonde d'apprentissage [Laborde 95]. Ce logiciel permet de concevoir des activités didactiques favorisant l'assimilation de concepts mathématiques [Laborde 89, 95, Bellemain 92]. Il constitue un excellent support pour explorer et manipuler des objets mathématiques abstraits.

Schématiquement, l'utilisateur définit une organisation des objets abstraits en introduisant un à un les objets et en spécifiant les relations entre chaque nouvel objet et les objets déjà construits. Ces relations sont appelées des contraintes. Pour définir cette organisation, l'utilisateur manipule directement le résultat de ses constructions, c'est-à-dire les représentations graphiques des objets qu'il a déjà construits. Il peut déformer les objets en modifiant les derniers degrés de liberté de la figure construite, simplement par manipulation directe des objets, sans devoir effectuer une action préalable spécifique pour activer l'outil spécialisé pour l'animation.

Au départ, mon sujet de recherche était d'intégrer au logiciel de nouvelles manipulations pour redéfinir les contraintes entre les objets de façon dynamique, même lorsque ces objets ne sont pas des objets de base du logiciel. Par exemple, on aimerait que l'utilisateur puisse redéfinir un carré basé sur les deux extrémités d'un de ses côtés en raccrochant la construction à deux points d'une droite, ou redéfinir un cercle contraint par un point de sa circonférence en un cercle contraint par une tangence. On pourrait aussi rajouter de nouvelles contraintes, obligeant par exemple un point à appartenir à trois objets en même temps, ou un cercle déjà contraint par son centre et un point de sa circonférence, à être en plus tangent à un autre cercle. Il s'agissait donc de définir des raccourcis par rapport à l'activation explicite de l'outil de redéfinition des contraintes, de définir de nouvelles contraintes, et de mettre en œuvre l'ajout de contraintes, explicitement par l'introduction d'un nouvel outil, et implicitement par la définition des raccourcis correspondants.

J'étais très intéressée par l'aspect « définition de macro-constructions », car elles laissent le plus longtemps possible l'utilisateur maître de l'état du micromonde constitué par Cabri. Or, un prolongement de cette approche consiste à trouver et à mettre au point un moyen pour que l'utilisateur puisse définir lui-même ses « raccourcis ». Il pourrait ainsi spécifier la proposition du logiciel de redéfinir les points de base du carré de l'exemple précédent, et celle de rajouter une contrainte de tangence pendant la déformation d'un cercle l'amenant à s'approcher d'une droite. Dans cette optique, une idée est de permettre à l'utilisateur de redéfinir des contraintes dans les macros, et ainsi d'utiliser le concept des macros pour définir des raccourcis.

Deux impératifs apparaissent :

- il faut que le concept des macros puisse supporter l'intégration des redéfinitions de contraintes ;
- il faut qu'on puisse vérifier que le résultat obtenu est bien le résultat attendu.

Il est nécessaire de fournir un moyen d'accès immédiat à l'organisation effective des objets abstraits. De plus, pour mettre au point ses macro-constructions, l'utilisateur a besoin d'être informé sur l'état courant des dépendances entre les objets et sur le contenu exact des macro-constructions qu'il définit. Il est donc nécessaire de lui fournir un moyen d'accès à la vue structurelle de la construction géométrique qu'il a mise en œuvre.

C'est pourquoi l'objectif de la thèse a évolué vers une problématique d'intégration d'une vue textuelle de la figure et des macros, munie des fonctionnalités adaptées à la mise au point du programme de construction.

Finalement, on arrive à l'idée de considérer le logiciel comme un environnement de développement pour la programmation visuelle de figures. Cette fonctionnalité s'adresse à des utilisateurs non familiers de quelque technique de programmation que ce soit, et sans motivation pour le devenir. La familiarisation doit s'acquérir de façon immédiate, par l'évidence.

Ces exigences conduisent à la mise en place d'une vue textuelle des figures, équivalente à la vue graphique et dynamique, dans le sens où le programme se construit en même temps que la figure, et où l'ubiquité des objets dans les deux vues synchrones permet un apprentissage implicite du langage de Cabri-programmation. La « qualité dynamique » de la géométrie dans la figure sera alors traduite par la « qualité formelle » du langage induit.

L'intégration d'une vue textuelle de programme dans une interface qui permet de commander des constructions géométriques par manipulation directe est une approche opposée à l'approche classique, où on part d'un langage de programmation usuel pour lui ajouter un couche graphique. Nous avons rencontré ici des problèmes originaux.

Cette problématique conduit à une recherche théorique sur les moyens à mettre en œuvre pour aider des non-informaticiens dans des tâches de programmation, et à une recherche pratique sur les mécanismes à utiliser pour arriver à fournir le support désiré.

L'organisation du document reflète cette décomposition.

Dans la première partie (Manipulation directe, programmation visuelle, programmation textuelle), nous présentons les différents aspects qui rapprochent l'activité de construction de figures géométriques dans le logiciel Cabri-géomètre d'une activité de programmation.

Nous présentons les difficultés rencontrées par l'utilisateur du fait de la forme et du mode opératoire des différentes activités de programmation, liées au paradigme de programmation supporté par le logiciel. Ensuite, pour choisir la forme de support adaptée aux buts que nous nous sommes fixés, nous faisons un tour d'horizon des différents supports utilisés pour la programmation au cours de l'histoire de l'informatique, puis nous étudions les différents logiciels existants qui offrent, comme Cabri, des possibilités de programmation. Cette étude nous permet d'énumérer des critères primordiaux dans le contexte du domaine d'application de Cabri. Enfin, nous présentons ce que l'approche envisagée pourrait apporter, indépendamment de ses contraintes de réalisation.

La seconde partie (De la manipulation à la programmation) est consacrée à la présentation de la réalisation du prototype.

Nous commençons par présenter les différents facteurs qui nous ont conduite aux différents choix, celui du langage de Cabri-programmation pour spécifier complètement la structure, et celui du sous-langage de géométrie formelle pour décrire la dynamique de la géométrie dynamique.

Nous présentons aussi les différents artefacts (statiques et dynamiques) attendus pour donner accès à la sémantique des programmes (notations secondaires...), et les contraintes liées au contexte de l'intégration.

Ensuite, nous décrivons les problèmes rencontrés et les solutions choisies, qui conduisent aux spécifications internes. Puis le prototype est décrit dans son aspect extérieur, avec le décalage entre les désirs et la réalité : ce qui est fait et comment c'est fait, et ce qui n'est pas fait. Nous terminons par un bilan sur la méthode choisie et une discussion sur ce qu'il faudrait modifier pour dépasser les limitations de notre prototype, et concluons par quelques éléments de réflexion sur les nouvelles études que le prototype permet d'aborder.

-o-o-o-o-o-o-o-o-

Partie I

Manipulation directe, programmation visuelle, programmation textuelle

Axé sur l'amélioration des fonctionnalités programmatoires d'un logiciel de manipulation directe, ce travail s'inscrit dans le récent domaine de recherche constitué de l'étude des systèmes interactifs offrant des activités de programmation par manipulation de l'interface. Ce domaine vise à simplifier la tâche de programmation, jusqu'à la rendre abordable par des novices en programmation, ou même par des utilisateurs non-programmeurs.

Le logiciel concerné est Cabri-géomètre, le Cahier de BRouillon Interactif qui se présente sous la forme d'une feuille blanche accompagnée d'outils de construction d'objets graphiques, principalement géométriques, mais aussi numériques et textuels. La construction finale est élaborée incrémentalement comme juxtaposition de constructions élémentaires. Chaque construction élémentaire est le résultat d'une action de l'utilisateur sur les objets déjà construits, avec la sensation de manipulation directe de ces objets. Ces constructions élémentaires peuvent être regroupées et définir de nouveaux outils qui constituent des sous-programmes de construction applicables à d'autres objets. Le résultat obtenu constitue une figure déformable sous réserve de vérification des contraintes qui, définies par les actions de construction, lient ses différents constituants.

L'activité de construction de figure et de définition de nouveaux outils de construction supportée par les manipulations de l'interface de Cabri s'apparente à l'activité de programmation. Elle permet d'aborder le logiciel sous son aspect d'environnement interactif de programmation.

Les systèmes interactifs offrant des possibilités de programmation à des non-programmeurs constituent un domaine de recherche récent. Les moyens utilisés pour que l'utilisateur spécifie sa tâche de programmation, les niveaux d'interaction supportés par le système et les supports graphiques utilisés pour informer l'utilisateur de l'état du système, conduisent à classer ces systèmes et à distinguer la programmation visuelle de la programmation par démonstration ou par l'exemple.

La catégorie dans laquelle le logiciel Cabri-géomètre s'inscrit est fortement marquée par les choix et les principes fondamentaux qui le caractérisent. Outre le choix du mode opératoire à manipulation directe des objets graphiques, toutes les fonctionnalités suivent principalement deux principes : (1) allègement de la tâche de l'utilisateur par résolution automatique de tous les choix pouvant être ressentis comme implicites grâce à la manipulation directe des objets (réduction des distances sémantiques et articulatoires) , et (2) maîtrise la plus fine possible du contenu du logiciel (de l'état de la mémoire) par l'utilisateur afin de laisser le logiciel ouvert (flexibilité). Ce second principe ne constitue pas une relégation des choix délicats à l'utilisateur, qui pourrait être vue comme un principe de "facilité". Au contraire, il a pour conséquence d'imposer à l'implémenteur la difficile contrainte de fournir à l'utilisateur des accès aux données mémorisées dans l'état courant.

Dans une utilisation au premier niveau, la fonctionnalité programmatoire est séquentielle. Au second niveau, elle permet aussi l'encapsulation (hiérarchisation) par définition de macro-commandes. Le premier des deux principes nous incite à rendre l'apprentissage de la programmation de sous-programmes (ou macro-constructions) le plus simple possible, simplement par spécification a posteriori parmi tous les objets de la figure d'exemple (ou d'apprentissage) des objets initiaux et terminaux de la construction. L'effet des activités de programmation est immédiatement visible sur les représentants graphiques des objets manipulés.

Le second principe peut sembler quelque peu opposé au premier, puisque les objets graphiques manipulés directement par l'utilisateur sont les instances de variables du programme de construction. Ils constituent le résultat d'une exécution sur des valeurs particulières, les valeurs des objets de base de la construction. C'est sur leurs représentants graphiques que l'utilisateur agit et ressent exécuter des actions. L'image obtenue (résultat graphique) ne contient pas de trace explicite des causes, c'est-à-dire des actions de l'utilisateur : on n'y trouve que la représentation graphique de leurs effets, c'est-à-dire les conséquences des choix.

Cela rend les contraintes indiscernables des propriétés et conduit à des difficultés de mise au point des programmes. La vue des résultats, donc des conséquences, n'est pas une vue de la structure interne du programme, qui elle, donnerait accès aux choix et donc aux causes.

Comment y remédier ? Parmi les supports visuels donnant accès à la structure interne, le choix textuel offre principalement l'avantage d'une lisibilité immédiate. Ce recours à une vue textuelle, bien que permettant la programmation textuelle, n'est pas un retour aux langages de commandes : ce n'est pas le résultat du texte qui est exécuté, mais ce qui est exécuté qui est visualisé par un texte, qui est conséquence des manipulations et non initiateur des commandes.

Cette spécificité d'approche de la programmation textuelle est rendue particulièrement sensible par l'utilisation de la dimension temporelle, avec la perception dynamique apportée par la réactivité et la synchronisation des vues. Les différentes vues sont des supports de manipulation des objets, des média de communication entre le logiciel et l'utilisateur. Chacune d'elles véhicule des informations complémentaires.

Le titre de cette partie, "Manipulation directe, programmation visuelle et programmation textuelle", résume le processus de raisonnement précédent, prenant son contenu dans les points d'attraction du discours. En bref, le but de cette partie est de répondre aux questions suivantes : d'où on part, ce qui y manque et ce qu'on propose.

Le premier chapitre est donc consacré à une présentation du contexte, c'est-à-dire du logiciel Cabri-géomètre et des activités programmatoires qu'il permet. L'accent est mis sur les difficultés propres à ce type d'activité dans cet environnement, ce qui fait apparaître la nécessité de l'accès à la structure logique du programme construit au travers des manipulations effectuées.

Dans le deuxième chapitre, nous montrons les différentes formes utilisées pour présenter les programmes au cours de l'histoire de l'informatique, et dans les approches actuelles voulant faire programmer des non-informaticiens. Il en ressort que les approches textuelles et graphiques ont chacune des avantages et des inconvénients.

Le troisième chapitre met en relation certaines spécificités du logiciel avec celles d'autres environnements qui possèdent un des quatre aspects fondamentaux suivants, présents conjointement dans Cabri : primauté graphique, caractère programmatoire, manipulation directe, et public non-informaticien.

Le dernier chapitre présente les apports potentiels résultant des augmentations de fonctionnalités possibles. Ces apports concernent Cabri lui-même, ses utilisateurs, son développement, et le domaine de recherche concerné, avec complémentarité des trois aspects cités dans le titre (quel type d'information concerne chacun d'eux).

Cabri-géomètre est un logiciel qui, grâce à la manipulation directe, permet l'exploration de figures géométriques [Laborde 85]. Des enseignants de lycée et de collège, ainsi que de nombreux mathématiciens, l'utilisent pour l'enseignement (dès la sixième) et pour leurs travaux personnels. Issu de la recherche de l'équipe EIAH du laboratoire Leibniz de l'IMAG, Cabri-géomètre est le fruit d'une collaboration de longue haleine entre mathématiciens, informaticiens, didacticiens, et enseignants quotidiennement en contact avec les élèves [Laborde 89, 95, Bellemain 92].

Ce logiciel plonge l'utilisateur dans un micromonde intelligent et constitue ainsi un environnement d'apprentissage pour le cas de la géométrie [Laborde & Laborde 91, Bellemain 96]. L'utilisateur peut construire des figures géométriques, explorer le champ des animations et déformations de la construction globale, élaborer de nouveaux outils avec des macro-constructions, et régler¹ son environnement pour des tâches spécifiques en y intégrant éventuellement ses outils personnels. Tous les objets manipulés sont saisis directement par l'utilisateur, et réagissent en temps réel, même sur la calculatrice TI-92.

Ces possibilités de déformation de figures vérifiant un certain nombre de contraintes définissent la géométrie dynamique et ont conduit à la distinction des concepts de figure et de dessin [Arsac 89, Parzysz 88, Laborde & Capponi 94].

Initialement conçu pour un usage en didactique des mathématiques, Cabri-géomètre a vu le champ de ses domaines d'application s'agrandir. Partant de la géométrie euclidienne dans le plan, il s'est étendu à la géométrie descriptive dans l'espace, puis aux géométries non-euclidiennes [Laborde & Laborde 96, Lister 98] et à l'étude des coniques [Cuppens 95, Guillerault 96, Trgalová J. 95], pour arriver aujourd'hui à l'algèbre [Capponi & al. 98] et aux probabilités [Coutindo 99], tant en didactique [Capponi & Laborde 94, Charrière 96] qu'en recherche.

Les niveaux d'études concernés vont de l'école primaire à l'université. En physique, les possibilités de simulation offertes par la géométrie dynamique conduisent à utiliser le logiciel pour modéliser des phénomènes d'optique, des machines électromagnétiques, et des mouvements mécaniques de pièces.

¹ Nous préférons « régler » au français « profiler », terme technique en aviation et non en informatique.

domaine	discipline	cadre d'application
enseignement	mathématiques	géométrie euclidienne géométries non euclidiennes coniques géométrie analytique, vectorielle fonctions systèmes dynamiques probabilités
	physique	optique mécanique électricité électromagnétisme
	chimie, biologie	
	logique	
	...	
recherche	didactique	mathématiques géométries vecteurs, transformations, angles,... analyse fonctions, systèmes dynamiques
	philosophie	histoire des mathématiques
	mathématiques	
	...	
autre	CAO	
	art	art mauresque rosaces ludique
	...	

Le domaine de base de Cabri, à savoir la géométrie dynamique, et l'aisance manipulative de son interface, lui confèrent de remarquables qualités exploratoires. En effet, la manipulation directe des objets permet de les déplacer sans la gêne provoquée par la nécessité d'une action préalable, un retour d'informations rassurant accompagne les tentatives d'utilisation des outils, et le retour en arrière sur la dernière action est toujours possible et immédiat.

L'ensemble de ces propriétés permet des activités de type boîte noire, ou problème ouvert. D'autres activités trouvent un terrain propice dans Cabri : il s'agit d'activités de réalisation d'une tâche de construction définie par un énoncé mathématique, et de simulation de machines géométriques appelées systèmes articulés.

Le tableau ci-dessous récapitule les différentes applications de Cabri-géomètre, avec la façon dont le logiciel est utilisé et les capacités de l'utilisateur qui sont sollicitées. Les liens horizontaux ne sont pas représentés, mais on peut lire par exemple : dans les applications de Cabri à la mécanique, sur la base d'une figure préalablement construite, on trouve des activités de simulation, pour lesquelles la tâche est une tâche d'expérimentation. Les fonctionnalités utilisées sont des animations avec prises de mesures et les capacités sollicitées chez l'utilisateur sont perceptives.

Ainsi, la première colonne liste les différents domaines d'application du logiciel. Les deux colonnes suivantes, "activité" et "tâche", décrivent les buts de l'utilisation qui en est faite, du côté de l'objectif. La colonne "activité" correspond plutôt au fond et la colonne "tâche" à la forme de l'utilisation. Les deux dernières colonnes décrivent ce qui est utilisé : les "fonctionnalités" sont relatives au logiciel, et les "facultés" à l'utilisateur.

application	activité	type de tâche	fonctionnalité	faculté
géométrie euclidienne	explorations	résolution de problème ouvert	animation	psychomotricité
géométries non euclidiennes			prise de mesures	perception
coniques		modélisation	résolution de boîte noire	création
géométrie analytique	simulation			observation
fonctions		simulation	expérimentation	destruction
systèmes dynamiques	construction			programmation
probabilité		prototypage	présentation	
optique	prototypage			investigation
mécanique		...		
électricité				
électromagnétisme				
CAO				
graphisme mauresque				
rosaces				
ludique				
...				

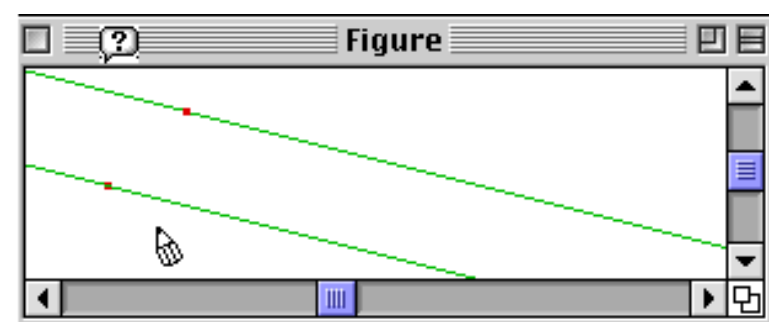
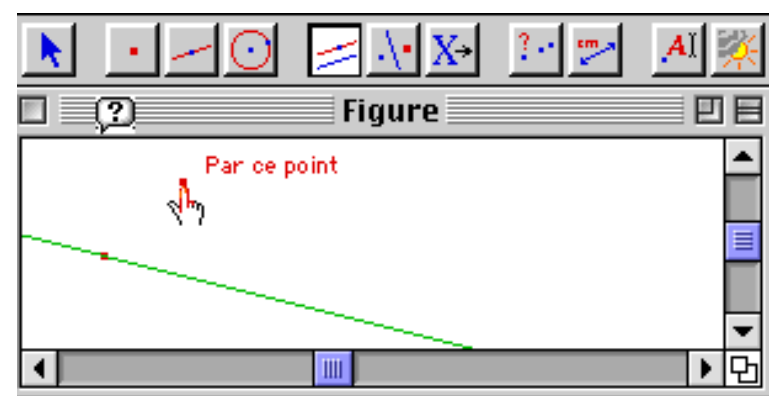
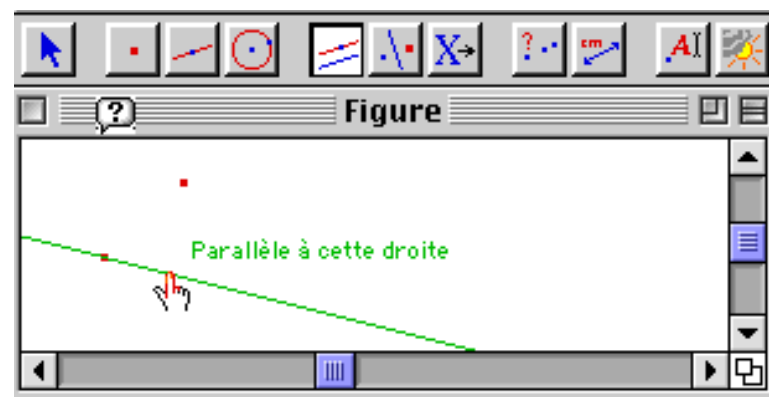
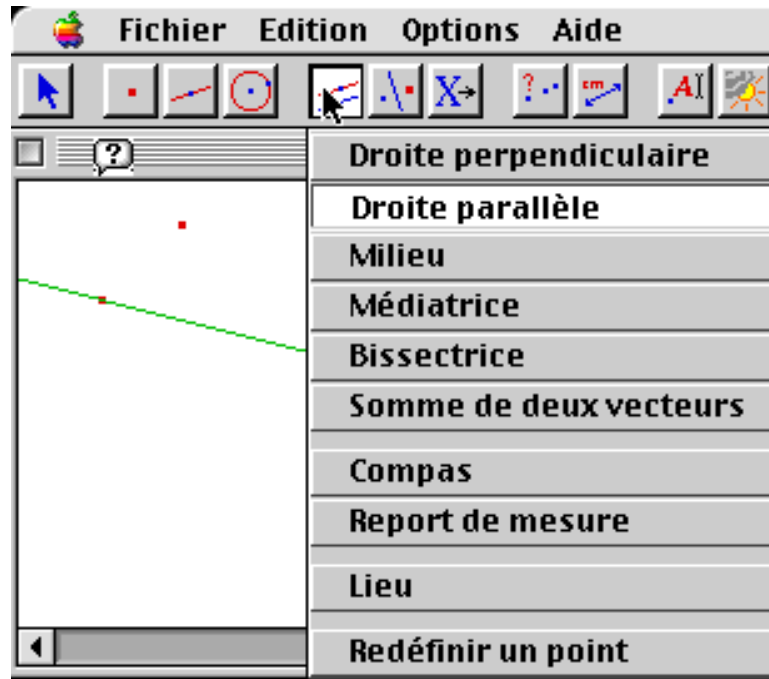
Cabri possède le caractère ludique des jeux de construction, avec l'animation en plus pour compenser la manipulation réelle. Aussi la tentation est-elle proche de l'appeler "meccano virtuel animé", en prolongeant la métaphore du meccano empruntée à Pierre-Marie Charrière [Charrière 96].

Dans ce chapitre, nous nous limiterons aux activités de construction et au caractère programmatoire des tâches, dans l'utilisation de Cabri.

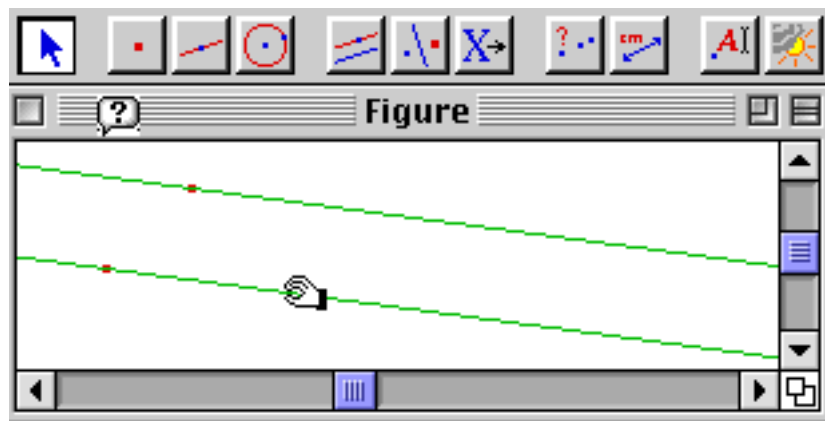
A) Vers une activité de programmation

Actuellement, pour construire une figure géométrique avec Cabri, l'utilisateur doit agir objet par objet. Chaque objet est défini par le choix d'une propriété de dépendance qui le lie à d'autres objets précédemment construits. Le nouvel objet est construit "sur" des objets parents en respectant la contrainte géométrique choisie.

Par exemple, pour construire une droite parallèle à une autre et passant par un point, l'utilisateur choisit l'outil de construction "Droite parallèle" puis spécifie la droite et le point de référence pour cette construction. Les quatre illustrations suivantes montrent les différentes étapes manipulatoires de cette construction.



Dans la fenêtre suivante, l'utilisateur manipule la droite de base. La droite construite comme une droite parallèle à cette droite, suit le mouvement de la droite dont elle dépend.



Qu'est-ce qu'un caractère programmoire pour une activité ? D'après le dictionnaire, une activité est une fraction spéciale de l'ensemble des actes coordonnés et des travaux de l'être humain. En se rapportant à la théorie de l'activité, cette fraction est spécialisée par son orientation vers un objet qui coordonne les actes et constitue sa raison d'être. Ainsi, les activités sont concrétisées par la réalisation d'une tâche ou d'un ensemble de tâches. Une tâche est orientée vers la réalisation d'un objectif, l'atteinte de cet objectif étant mesurable.

Les mots qui apparaissent dans la colonne "activité" du tableau précédent ne constituent pas obligatoirement l'objet de l'activité. Ils regroupent plutôt des caractéristiques de l'activité en cours. Par exemple, une activité de prototypage a à la fois des caractères de modélisation et de programmation. Un caractère programmoire pour une activité est un caractère de l'activité que l'on retrouve dans toute activité de programmation.

Quelles sont les tâches qui sont spécifiques d'une activité de programmation ? Pour répondre à cette question, rappelons la structure du cycle de vie d'un logiciel, qui se décompose principalement en les phases suivantes :

- spécification du résultat attendu,
- traduction du modèle mental du résultat attendu en un plan compatible avec l'ordinateur (conception),
- transmission de ce plan à l'ordinateur (expression),
- confrontation du résultat obtenu avec le résultat attendu (validation).

Une activité de construction de figure géométrique est typique d'une activité de programmation, puisqu'elle passe par toutes les étapes de ce cycle de vie.

Une activité d'exploration d'une figure déjà construite a pour objectif que l'utilisateur construise sa propre idée du résultat "contenu" dans l'ordinateur, par un jeu d'aller/retour entre l'image apparente et l'image supposée. Elle est donc proche d'une étape de validation.

D'autres utilisations de Cabri-géomètre peuvent faire appel à différents aspects programmatoires du logiciel.

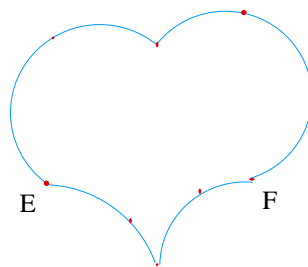
Pour illustrer plus concrètement ces différents aspects, voici un exemple d'école qui couvre un assez large éventail d'activités types (ou de types de tâches), puisqu'il s'agit d'un exemple de construction.

1°) D'une programmation séquentielle simple...

Le but est d'aboutir à la réalisation d'une construction qui réponde à la spécification suivante : tracer un cœur déformable mais qui reste toujours un cœur. Cet exemple met en œuvre des tâches d'investigation car la méthode n'est pas dirigée : aucune directive n'est donnée concernant la méthode. L'activité est constituée de l'ensemble des actions et des tâches effectuées dans le cadre de la réalisation du travail demandé. L'approche utilisée, pour faire ressortir le caractère programmatoire de cette activité, est de s'attacher à l'étude d'un chemin de raisonnement emprunté parmi tous les chemins possibles, en notant le type et le caractère de chaque tâche ou action exécutée.

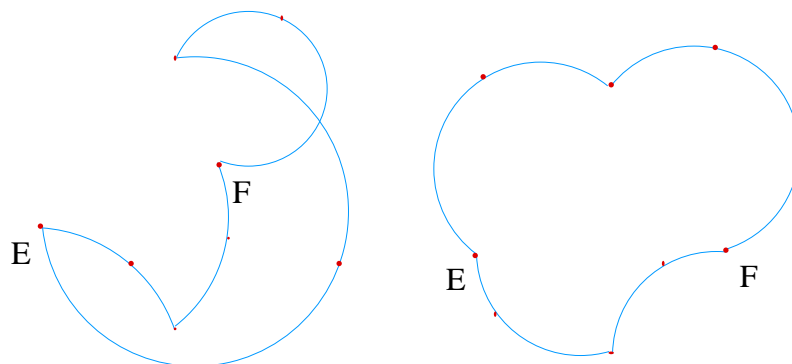
a. Tâche d'investigation

Après une phase d'exploration des outils, qui constitue une activité de découverte des constituants de l'environnement informatique, une première approche peut consister en l'utilisation directe des outils de construction d'arcs. Le résultat obtenu est illustré par la figure suivante.



Chaque point est placé au jugé, à l'aide du retour d'information fourni par l'interface, qui visualise l'arc en cours de création. Ainsi, cette figure est obtenue uniquement en utilisant les capacités perceptives.

La validation de la figure passe par une étape d'expérimentation par animation, de façon à vérifier la deuxième partie de l'énoncé : "déformable mais en restant toujours un cœur". Les points de base des arcs ayant été positionnés librement, leur déplacement est totalement libre, ce qui permet des déformations telles que la figure ne représente plus un cœur. D'autre part, les deux arcs issus des points E et F perdent leur propriété de tangence mutuelle.



Ces observations conduisent à un réajustement de l'interprétation de la consigne et à une remise en cause de la construction. La tâche devient celle de réaliser une construction perceptivement proche de la première, mais dont les seules déformations possibles sont l'agrandissement et la réduction ou du moins sont telles que les arcs de cercle en E et F restent toujours tangents.

b. Recherche d'une stratégie, parallèle avec les activités de programmation

Pour cette deuxième approche, l'échec de la tentative précédente conduit à moins de précipitation et à une approche plus analytique du problème. La distance entre l'énoncé du problème et sa solution informatique impose une prise en compte de sa complexité. Pour y faire face, une méthode, présente dans toute activité de programmation, consiste en :

- l'analyse des causes de l'échec de la méthode,
- le choix d'une stratégie de remédiation,
- la décomposition du problème en sous-problèmes indépendants.

Ainsi, l'approche prend le tournant suivant : comme la seule création des arcs ne suffit pas, il faudra construire d'autres objets et les cacher ensuite ; par contre, l'introduction d'un axe de symétrie pour la figure réduira la tâche de moitié. Il reste alors à :

- déterminer les objets à ajouter,
- choisir parmi tous les objets ceux qui permettront de déformer la figure,
- déterminer l'ordre dans lequel introduire les objets de la figure et les contraintes qui les relient,

de façon à ce que la figure résultante réagisse comme le spécifie l'énoncé.

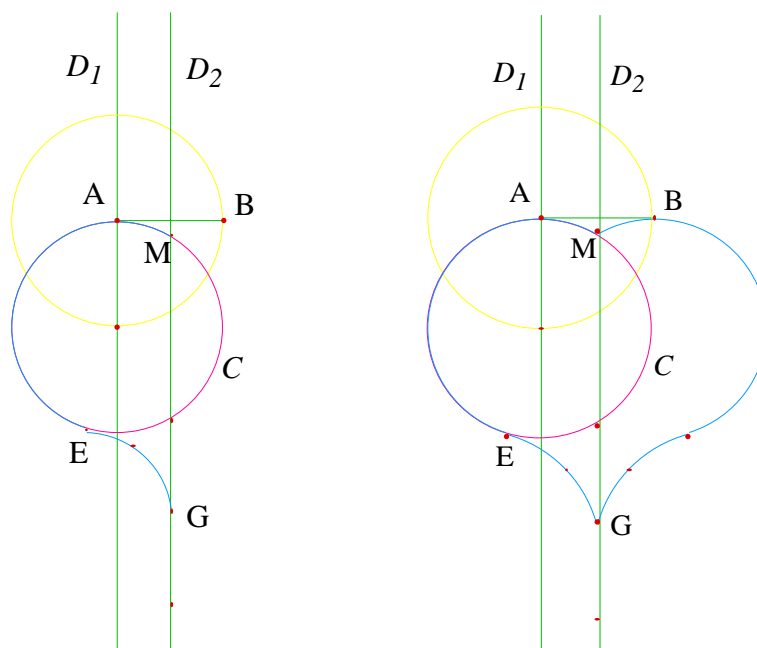
D'abord, trois objets supplémentaires semblent indispensables : un cercle pour porter chacun des arcs de la première moitié du cœur, et la droite qui sert d'axe de symétrie. Les autres objets sont déterminés à mesure des besoins, en fonction du contexte proche.

Nous appellerons « objets maîtres » les points de base qui commandent les déformations de la figure.

Pour choisir les objets maîtres, nous décidons de ne considérer que des points qui appartiennent aux éléments visibles sur la figure finale. Deux points sont choisis pour leurs rôles apparemment équivalents : les deux points les plus hauts du cœur. L'extrémité commune des deux arcs constitue un troisième point, car il permet d'ajuster la forme du cœur a posteriori. L'ordre d'introduction des objets relève de la mise en place d'une stratégie (plan, programme), dont le choix précis découle d'une décomposition en sous-tâches, auxquelles sont associés des sous-produits intermédiaires :

1. déterminer comment prendre en compte l'aspect symétrique de la figure, c'est à dire quels objets construire directement et quels objets construire par symétrie ;
2. dans la moitié à construire directement, déterminer comment lier au segment [AB] le cercle sur lequel est "posé" le "gros" arc ;
3. À partir de l'esquisse atteinte, représenter le deuxième arc et la moitié symétrique afin de visualiser la tâche restante.

Les deux figures suivantes montrent une proposition de solution sur le demi-cœur et sur le cœur complet, respectivement, dans laquelle tous les objets construits sont laissés visibles.



Les objets maîtres sont donc A, B et E. A et B sont libres dans tous leurs déplacements. Le déplacement d'un des deux provoque une déformation du segment $[AB]$, avec changement de direction.

La droite D_1 étant construite comme contrainte à rester perpendiculaire au segment $[AB]$, sa direction suit l'évolution de celle de $[AB]$. De même D_2 , médiatrice du segment $[AB]$, tourne avec $[AB]$. D_2 constitue l'axe de symétrie du cœur et, dans la deuxième figure, contraint la deuxième moitié à être symétrique de la première.

Le cercle de centre A est construit afin de déterminer la position du centre du cercle C. Il est contraint à passer par le point B, aussi les déplacements de A et B modifient le rayon de ce cercle.

E n'est pas un point complètement libre, puisqu'il est pris comme un point du cercle C, mais E reste libre dans ses déplacements sur le cercle C.

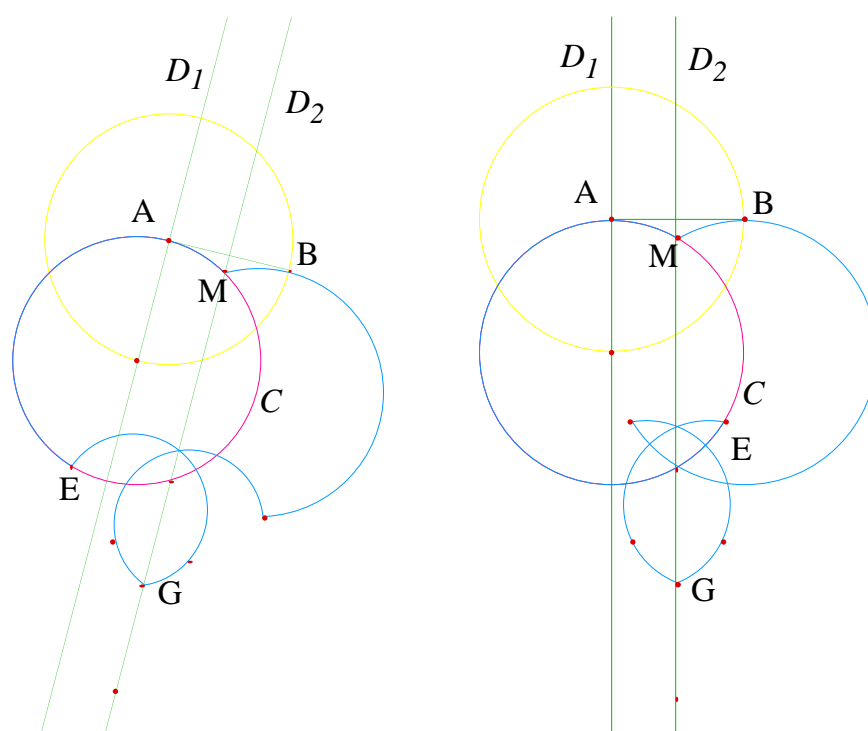
La fin de la construction est la suivante : M est, parmi les deux points d'intersection de C avec la médiatrice de $[AB]$, celui qui est le plus proche du segment lui-même, et G est le milieu du bipoint constitué de l'autre point d'intersection et du symétrique de M par rapport à ce deuxième point d'intersection. Les deux arcs laissés visibles dans la construction finale sont d'une part l'arc de cercle qui passe par M, A et E, et d'autre part celui qui passe par E, un point libre et G.

Cette paraphrase de la figure complète le dessin. Non seulement elle décrit son dynamisme, c'est à dire ses réactions aux déplacements des points de base, mais aussi elle veut informer sur les causes des réactions. Ainsi, elle compense l'effet statique de la vue "papier" et donne les moyens de comprendre intuitivement les déformations potentielles. C'est cette paraphrase que nous formalisons plus loin en un « Cabri-programme ».

La figure obtenue est une séquence de constructions élémentaires ré-exécutée pour chaque déplacement, au moins pour les objets touchés par le déplacement, c'est-à-dire les objets dépendant des objets déplacés. Le mot "ré-exécuté" est introduit ici pour différencier ce que Cabri fait pendant la construction, où le logiciel mémorise les méthodes de calcul, de ce qu'il fait pendant l'affichage, où il applique les méthodes mémorisées.

Par exemple, lors de la construction d'une droite passant par deux points, la création donne lieu au calcul de la dépendance des coefficients de l'équation de la droite en "fonction" des coordonnées des deux points. Ces deux points pouvant être déplacés directement ou par le biais des objets dont ils sont issus, les coefficients obtenus ne sont pas mémorisés en tant que valeurs, mais en tant que méthodes de calcul appelées schématiquement "fonctions". C'est lors de l'affichage que les calculs de ces coefficients sont résolus, c'est-à-dire appliqués aux valeurs courantes des coordonnées des deux points.

Pour la suite, le scénario de la résolution de la troisième sous-tâche est similaire à une reprise de celui-ci, depuis la recherche de remédiation à la première approche, dite approche "directe". En effet, le résultat de certains déplacements des points A et E montre l'insuffisance de la définition du point E et la non adéquation de l'utilisation d'un point libre comme point intermédiaire du deuxième arc (voir les deux figures suivantes).



Ces deux figures font apparaître qu'il s'agit, pour finir, de construire le cercle porteur du dernier arc, de sorte qu'il contraigne le point E à être le point de tangence des deux cercles. Le problème ainsi constitué peut être assimilé au type de tâche appelé problème ouvert.

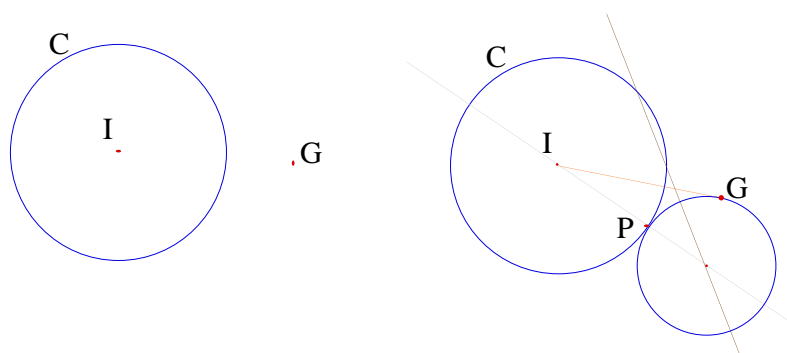
L'énoncé obtenu par l'extraction et la formalisation de cette sous-tâche en un problème ouvert conduit à l'exemple suivant. Une figure peut être associée à ce problème : c'est celle qui résulte de l'allègement des constructions annexes dans la figure plus complexe.

c. Tâche de type "problème ouvert"

Exemple de problème ouvert.

Déterminer le cercle tangent au cercle C qui passe par G.

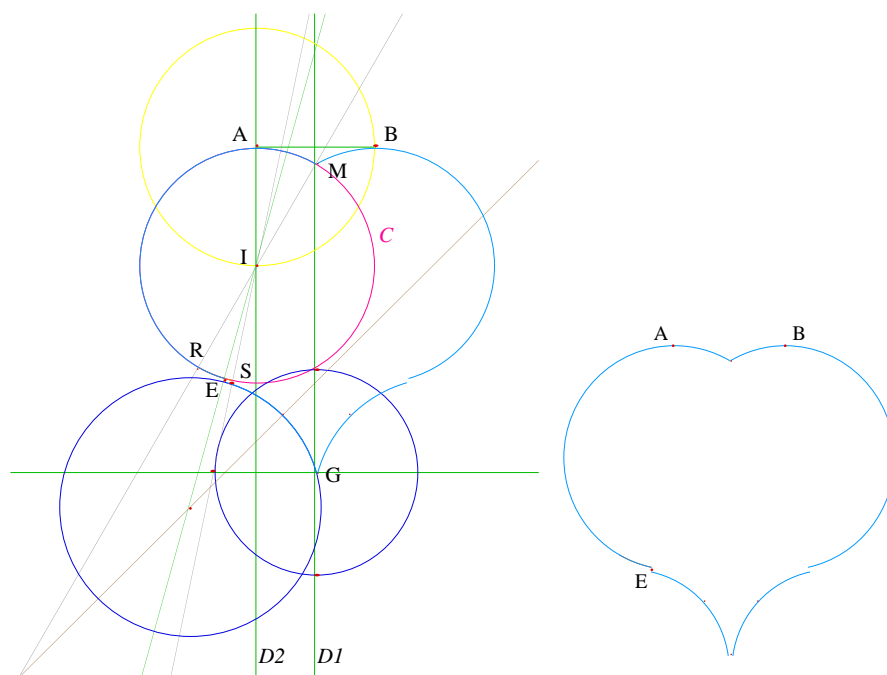
Les deux figures suivantes représentent le problème et sa solution :



Même a posteriori, les animations de cette figure ne permettent pas facilement de ressentir la contrainte de définition de la droite qui est en fait la médiatrice de [PG].

d. Réintégration et recomposition du résultat

Les deux figures suivantes représentent une solution avec tous les objets utilisés visibles, et la même solution allégée du tracé des objets géométriques non pertinents pour le résultat. Par rapport à la réintégration directe du résultat de la macro, ont été rajoutées une contrainte sur l'appartenance de E à l'arc RS et la redéfinition des arcs symétriques.



Dans ce scénario, les fonctionnalités du logiciel utilisées ont été des fonctionnalités d'animation et de construction, avec retour en arrière sur les tentatives infructueuses.

Les activités programmatoires qui y ont eu lieu sont ressenties dans les tâches de construction, qui demandent d'appliquer un outil formel à un ou plusieurs objets (programmation fonctionnelle), de confronter le résultat avec l'idée a priori qu'on en avait, ainsi que d'effectuer les tâches de structuration de la stratégie (élaboration d'un plan) et du choix des variables, rôle joué en quelque sorte par les objets cachés à la fin.

Ce scénario illustre bien la mise en place d'une méthodologie partant d'une analyse descendante du problème. Cette forme d'analyse est concrétisée par les choix des objets "maîtres" de la figure, et par l'analyse et la décomposition de la tâche, depuis l'observation d'une image du résultat attendu jusqu'à l'extraction de sous-problèmes validables indépendamment.

Ainsi, les tâches d'investigation (de prospection), ainsi que toutes les tâches qui nécessitent d'élaborer une figure dans le logiciel Cabri-géomètre, apportent à l'activité un caractère programmatoire.

Au premier niveau, la construction est augmentée de manière incrémentale, c'est-à-dire en spécifiant incrémentalement les objets et les contraintes qui les lient. C'est de la programmation séquentielle simple. L'aspect séquentiel est principalement ressenti par la correspondance directe entre les liens de dépendance des objets entre eux et l'ordre dans lequel ils ont été introduits.

2°) ...à une programmation plus évoluée

Mais plusieurs aspects conduisent à une programmation plus évoluée.

a. Élimination d'objets

Dans le fonctionnement séquentiel vu ci-dessus, une erreur sur le choix de l'outil, et donc sur le choix des propriétés que vérifient chacun des objets construits, oblige à supprimer l'objet insatisfaisant et à recommencer. En conséquence, les constructions dépendant de cet objet sont aussi détruites. Or, les objets supprimés ne sont pas nécessairement, et sont même rarement contigus dans l'ordre de leur création.

Il s'ensuit un réajustement de la liste des objets par rapport au souvenir de l'utilisateur de sa construction incrémentale, dès la suppression d'un premier objet dans une figure complexe. Ce réajustement devient de plus en plus important au fur et à mesure qu'on fait des tentatives de construction. Or, ce dont l'utilisateur a besoin, ce sur quoi il se base pour poursuivre sa construction, est la dépendance entre les objets (on peut dire l'ordre de dépendance entre les objets), et un des moyens mnémotechniques fréquemment (facilement) utilisé est la mémoire de l'ordre des actions.

b. Redéfinition de contraintes

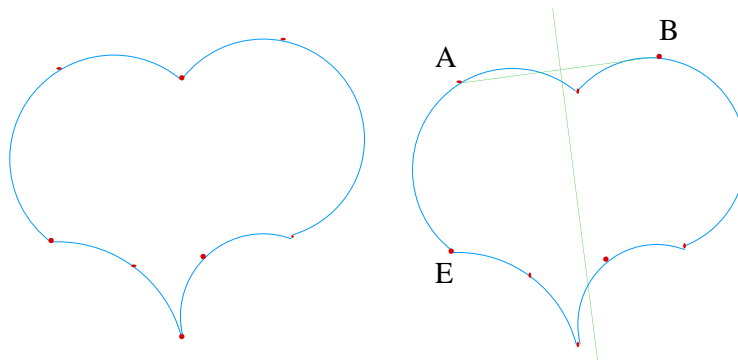
Le blocage consécutif au retour en arrière dans la stratégie de construction d'une figure contribue aux vertus didactiques du logiciel, puisqu'il motive si fortement qu'il oblige à élaborer préalablement une stratégie, un programme.

Cependant, toutes les utilisations de Cabri n'ont pas pour objectif de mettre l'utilisateur dans une situation d'apprentissage. C'est pourquoi Cabri offre la possibilité de modifier les contraintes entre les objets par l'intermédiaire d'outils de redéfinition. L'utilisation de ces outils libère l'activité d'exploration d'un choix préalable de stratégie puisqu'il devient possible d'ajuster la stratégie a posteriori, sans nécessité de refonte totale.

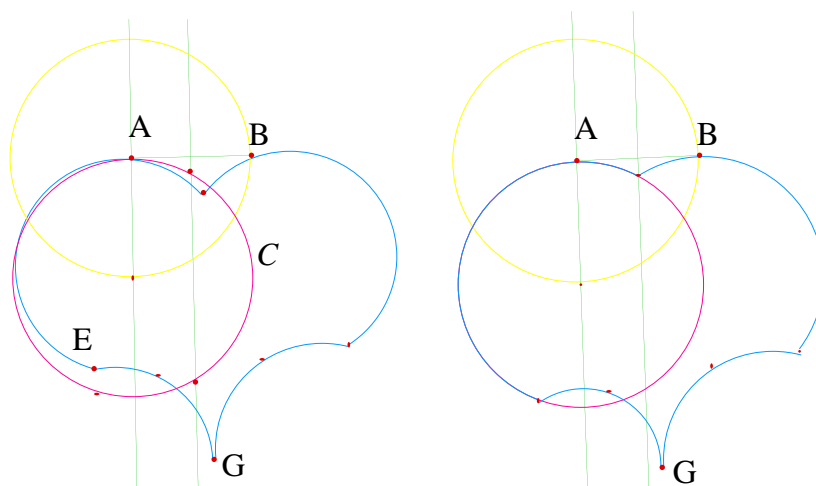
Les propositions de solutions peuvent être construites directement sur l'ébauche, offrant ainsi la possibilité d'une confrontation directe des comportements. En reprenant l'exemple du cœur, le choix des rôles des points A, B et E peut être fait depuis l'ébauche, et chaque validation de sous-tâche peut être suivie d'une étape de redéfinition d'un minimum d'objets pour réajuster la figure par morceaux.

Les outils de redéfinition des contraintes permettent de rattraper des erreurs de l'utilisateur dans la construction d'une figure. En effet, lors de la phase d'exploration d'un problème géométrique, un utilisateur peut construire des morceaux de la figure, qu'il considère par la suite comme valides pour son problème. Il lui est donc fastidieux de recommencer ces constructions, alors qu'une greffe dans une nouvelle construction lui suffirait.

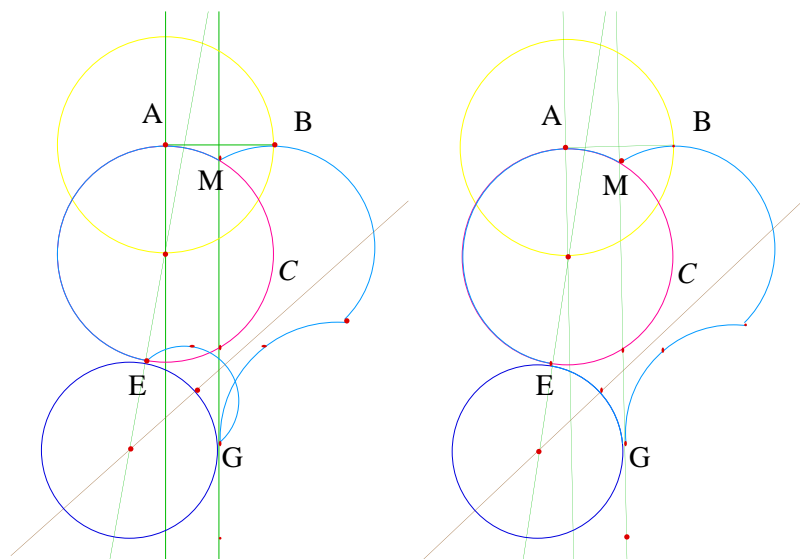
Les illustrations ci-dessous résument sous forme de film l'évolution de la figure selon la stratégie décrite précédemment, dans laquelle on "profite" au maximum de l'utilisation de l'outil de redéfinition des points.



Entre les deux illustrations ci-dessus, des points ont été nommés, le segment $[AB]$ a été construit ainsi que sa médiatrice. Sur figure suivante, le cercle de centre A et la parallèle à la médiatrice qui passe par A définissent le centre du cercle C comme l'un de ses points d'intersection.



Entre les deux illustrations ci-dessus, le point E a été redéfini comme un point sur le cercle C. Sur la figure suivante, le cercle qui passe par G et E a pour centre le point de la médiatrice de $[EG]$ qui est aussi sur la droite portée par le rayon du cercle C qui passe par E.



Mais ces outils de redéfinition agissent sur l'ordre de la construction : la programmation perd alors son caractère séquentiel, car l'ordre induit des contraintes entre les objets constituants de la figure ne respecte plus l'ordre de leur introduction.

Certaines contraintes effectives entre les objets deviennent alors plus difficiles à appréhender. L'annexe A1 rassemble les divers supports qui illustrent les conséquences des deux redéfinitions de points de la deuxième ligne du tableau précédent, dans l'ordre induit par les contraintes effectives qui lient les objets (et qui reste le plus proche de l'ordre historique, au sens que le minimum d'objets est déplacé).

La redéfinition des contraintes qui lient les objets entre eux rompt le caractère séquentiel de l'activité de programmation, puisque l'ordre des constructions ne respecte plus l'ordre des manipulations que l'utilisateur a en tête. L'ordre des contraintes correspond au réordonnement minimal qui respecte les relations de dépendance des objets.

Par certains aspects, l'activité qui en découle est plus naturelle, car la stratégie (le "plan") est plus libre (constructions investigatrices sans plan préalable et possibilité d'ajustement de la stratégie sans nécessité de refonte totale), ce qui contribue à la liberté d'exploration de constructions offerte par Cabri.

En effet, pour être à même d'appréhender l'évolution effective des contraintes entre les objets et de garder la maîtrise des réactions de la construction, l'utilisateur doit se baser sur un modèle minimal du fonctionnement du logiciel : la mémorisation directe de l'historique de la construction n'est pertinente que si elle est réajustée par les conséquences imaginées des redéfinitions utilisées, selon le modèle.

c. Spécification et utilisation de macro-constructions

Un autre aspect, qui renforce l'importance de l'activité de programmation dans l'utilisation du logiciel, résulte de la possibilité de définir des macro-constructions.

Cette fonctionnalité apporte modularité et abstraction à l'activité de programmation. Les macro-constructions peuvent être réutilisées (modularité) dans d'autres contextes (abstraction). L'activité programmatoire peut utiliser des raisonnements descendants (top-down design) grâce à cette possibilité de concrétiser l'analyse descendante de l'activité programmatoire.

L'utilisation de macros joue un rôle important dans les activités de programmation, permettant d'aborder des applications plus importantes en nombre d'objets, et de réutiliser les travaux effectués d'une application à l'autre (modularité).

Elle modifie aussi la méthodologie utilisée dans les activités de programmation, en faisant prendre en compte, en souci premier, le souhait de rendre potentiellement réutilisable tout investissement et tout aboutissement (à la réalisation) d'une tâche. Le choix même de la décomposition s'en trouve influencé. De nombreux articles vantent les mérites de la modularité (voir l'extrait suivant de [Pane & Myers 96], p. 37 et 38).

"Modular programming is often taught through a discipline known as top-down design, where the program is described at an abstract, high level, then refined into a modular hierarchy [Wirth 83]. However, hierarchically designed programs are not always easy to develop and comprehend [Curtis 89, Perkins 89]. Top-down strategies are difficult for novices because their spontaneous plans are based on concrete mental execution; action-oriented rather than object-oriented [Rogalski 90]. And, taking modularity and abstraction to the extreme can interfere with locality and visibility (see "Locality and Hidden Dependencies" on page 8) [Green 96]. One problem with comprehension of modular programs is that novices do not yet have the expert strategy of reading a program in a top-down order of execution manner — instead they read the program like a book [Gellenbeck 91-b, Jeffries 82, Wiedenbeck 86-b]. Novices focus on the very literal and concrete, rather than the abstract, hierarchical, general view used by experts [Onorato 86]. An environment is described to make very effective use of modularity [Miller 94].

A modular program can be modified faster than an equivalent non-modular program when at least one of the following conditions hold [Korson 86]:

modularity has been used to promote information hiding, which localizes changes;

existing modules provide a suite of useful generic functions that can be composed to implement a new functionality;

the modification requires an extensive understanding and modification of the existing code.

However, modularity did not help in other cases, such as adding a new feature to a program.

Another kind of abstract thinking that is difficult for novices is writing a general solution to a problem rather than a solution that is specific to the situation (e.g. a program that sorts a list). This requires students to make a shift from value processing to variable processing; and to elaborate some of the control decisions that are not consciously made in solving a specific problem [Hoc 90]. Also, a lack of abstraction is evident in the tendency of novice to code loops as sequential actions, unrolled [Hoc 89]. Examples and analogies play an important role in learning and understanding, and explanations help learners to generalize the examples [Lewis 87].

[...]

Examples

The programming environment can support modularity and reduce its negative attributes by providing multiple views, such as call graph, outline, and class hierarchy views [Miller 94, Roberts 88].

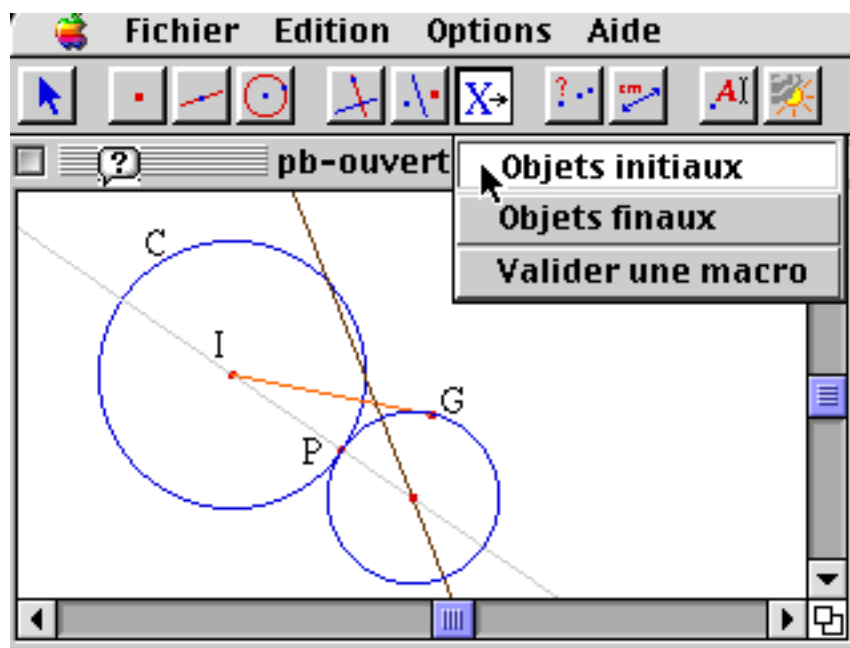
Spreadsheets do not support modularity and abstraction very well. Abstraction is presented at a fixed level and hierarchical representations are not supported [Lewis 87]

KidSim uses programming by demonstration to address the concrete vs. general abstraction problem. It allows users to create an abstract rule by demonstrating what that rule should do in a specific (concrete) situation. The system then automatically generalizes the specific rule into a more abstract one [Cypher 95]."

Ces macro-constructions sont donc utilisées pour reproduire des processus de construction déjà introduits par l'utilisateur pour aboutir à la construction de certains objets.

Pour définir une macro-construction, il suffit de choisir a posteriori ses arguments (objets initiaux) et ses résultats (objets finals). Par exemple, lors de l'introduction de la tâche du problème ouvert extrait du scénario de l'exemple, l'utilisateur peut choisir de définir une macro qui répond à l'énoncé de cette sous-tâche, c'est-à-dire qui construit le cercle tangent à un cercle et passe par un point déterminé. Il peut spécifier sa macro-construction depuis une figure "d'école", c'est-à-dire construite et utilisée seulement afin de servir de support à la résolution du problème ouvert dans un premier temps, puis à la définition de la macro. Il peut la spécifier également depuis la figure globale.

Par exemple, le problème ouvert du §I.1.A.1°c peut donner lieu à la définition d'une macro. Cette macro aurait pour objets initiaux ceux de la figure initiale (celle de gauche) et construirait le cercle cherché, c'est-à-dire le cercle qui passe par G. l'illustration suivante montre le contenu du menu qui gère la définition des macros.



Une macro-construction n'est pas simplement un enregistrement (nommé) d'une séquence d'actions de construction, mais le résultat d'un calcul sur cette séquence.

En effet, l'utilisateur peut avoir construit des objets annexes non utiles à la construction des objets terminaux de la macro ; ces constructions ne seront pas extraites de la figure et ne seront pas ré-exécutées lors des utilisations de la macro.

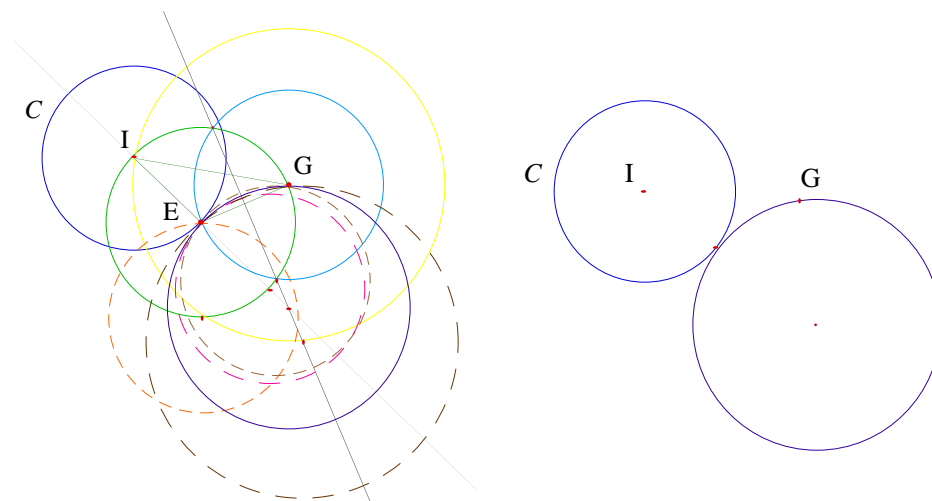
Typiquement, dans le cas de la définition d'une macro depuis une figure "d'école" construite pour la résolution d'un problème ouvert, cette figure aura servi de terrain d'investigation, avec des tentatives et des renoncements à ces tentatives. Les outils de redéfinition de contraintes peuvent même avoir été utilisés.

La liste des constructions résultante est la liste minimale des constructions utiles à la construction des objets terminaux depuis la spécification des objets initiaux. L'ordre de ces constructions respecte l'ordre interne de la figure mais pas l'ordre historique.

Ainsi, le logiciel Cabri-II offre à l'utilisateur la possibilité d'apprendre au logiciel des méthodes de construction. Ces méthodes sont mémorisées dans des "Macros", qui regroupent en un seul outil la liste des constructions minimalement nécessaires pour construire des objets finals en partant d'objets initiaux.

Dans le corps d'une macro, n'interviennent que des "primitives" (outils de base) de Cabri. Si une macro a été appelée pour la construction d'objets intervenant dans une nouvelle macro, celle-ci n'apparaît pas explicitement dans le corps de la macro englobante. En fait, l'utilisateur n'a pas accès aux constructions internes d'une macro, même en cours de construction. Il n'a même pas accès à la construction de la macro : il spécifie les objets initiaux et les objets finals, et le logiciel extrait de la construction globale (i.e. de la figure) la liste des constructions à effectuer pour construire les objets finals depuis les objets initiaux. Les constructions non nécessaires ne sont pas gardées.

Les figures ci-dessous illustrent, dans le cas de l'exemple du problème ouvert, le processus d'extraction des constructions utiles mis en œuvre dans Cabri. Elles représentent la figure "d'école" et un exemple d'appel de la macro résultante, sur une configuration similaire.



Dans la figure "d'école", des constructions annexes sont intercalées parmi les constructions utiles.

Dans la figure "d'appel", les objets internes de la macro ne sont pas visibles. Ils ne sont pas non plus accessibles. C'est par ce choix que les macros peuvent être considérées comme des boîtes noires. Les contraintes qui lient des objets résultant de l'appel ne sont retrouvables que par l'observation des propriétés de la figure : ils font partie des invariants de la figure en réponse aux animations qui la déforment.

En dehors de cette application didactique, dans les activités de programmation, cette spécificité allège la figure, conduisant à considérer les macros comme des outils de base (à identifier les macros à des outils de base), banalisant ainsi leur utilisation. Cela incite à une hiérarchisation de la programmation, en indifférenciant le comportement des outils, qu'ils soient de base ou construits (par l'utilisateur qui a défini les macros ou par d'autres utilisateurs). L'annexe A2 rassemble les divers supports qui illustrent ces deux figures, selon le même plan que l'annexe A1.

B) Nécessité de la vision et de la manipulation textuelle

L'activité programmatrice supportée par les manipulations de l'interface du logiciel Cabri-géomètre est plus évoluée que de la programmation séquentielle.

Les trois fonctionnalités qui permettent de dépasser ce niveau de programmation sont les possibilités d'éliminer des objets, de redéfinir des contraintes et de fabriquer de nouveaux outils par spécification de macro-commandes.

La section précédente (§I.1.A.) a présenté ces fonctionnalités et ce qu'elles apportent au logiciel. La présente section a pour but d'exposer leurs limites dans l'état actuel du logiciel et ce qui aiderait à les dépasser.

1°) Limites au cas pas cas

a. Élimination d'objets

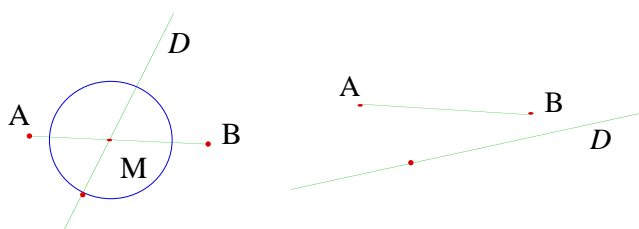
Un utilisateur peut être amené à reprendre une figure, soit parce qu'elle a été construite par quelqu'un d'autre, soit parce qu'un certain laps de temps s'est écoulé depuis sa dernière utilisation. Dans ce cas, s'il sélectionne un objet et le détruit, il peut être surpris par le nombre d'objets qui disparaissent. Il peut revenir en arrière sur son action à l'aide de la commande Défaire/Refaire la plupart du temps, sauf s'il est allé trop vite et a déjà entamé une autre tâche.

Une question se pose : quels sont les moyens pour l'utilisateur de deviner, depuis la figure seulement, les conséquences d'une destruction d'objet ?

Lorsque l'objet est libre, ou du moins qu'il lui reste un degré de liberté, il suffit à l'utilisateur d'animer cet objet pour repérer tous les objets qui en dépendent. Il existe cependant un cas de réserve, celui des constructions conditionnelles. En effet, la configuration d'un dessin affiché à l'écran peut représenter une figure dans laquelle certains objets n'existent pas.

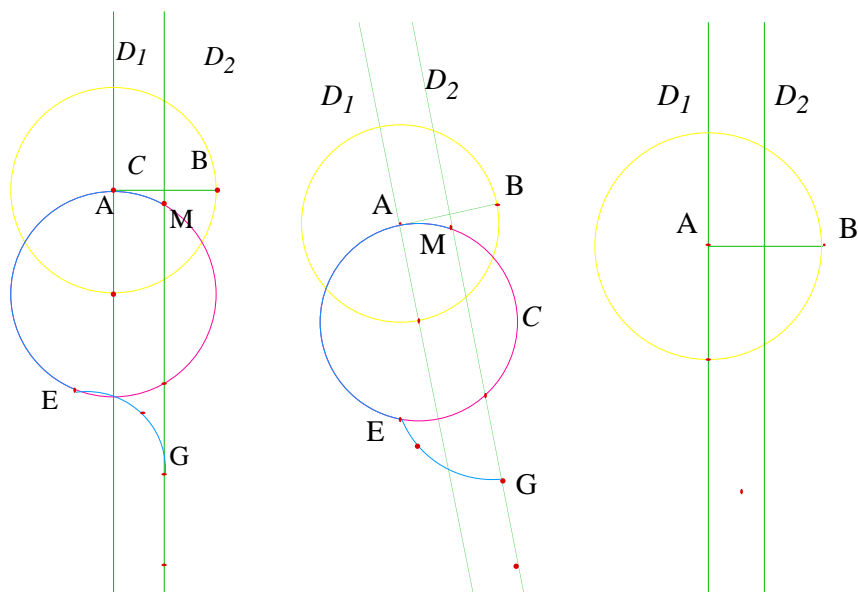
Par exemple, une droite et un segment n'ont pas toujours de point d'intersection, mais il est possible de lier des objets au point d'intersection depuis un cas de figure où la droite et le segment se coupent. Les constructions issues (dépendantes) de ce point sont alors conditionnelles, comme ce point lui-même. Si l'utilisateur détruit le segment ou la droite dans un cas où ce point n'est pas visible, ce point sera détruit aussi ainsi que toutes les constructions qui en dépendent.

La figure suivante montre un cas où le point d'intersection, M, existe (premier dessin), et un cas où il n'existe pas (deuxième dessin). M est utilisé comme centre d'un cercle ; ce cercle n'apparaît donc aussi que dans le premier cas.



Lorsque l'objet est lié, il faut que l'utilisateur détermine un objet qui le lie. Il peut commencer par essayer chacun des points de base de la figure. Mais l'animation de cet objet a pour conséquence le déplacement ou la déformation de tous les objets qui dépendent de lui, dont celui qui l'intéresse, mais pas seulement.

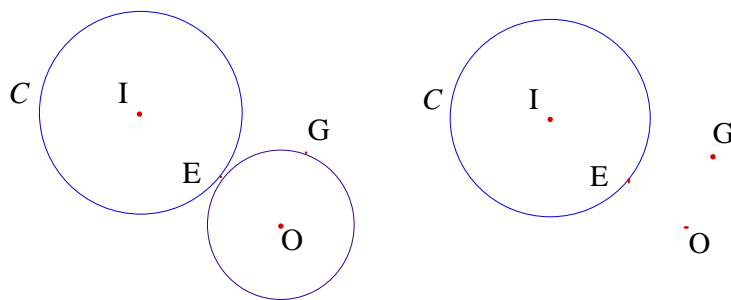
Dans la figure suivante, les conséquences de l'élimination du cercle C ne sont pas prévisibles, si ce n'est par essai et retour en arrière. C est entièrement contraint par les constructions dont il est issu, il ne peut être déplacé car il ne lui reste plus un seul degré de liberté.



Pour déterminer les objets qui dépendent de C, le point A est un objet de base dont C dépend ; il peut être déplacé. D1 et D2 sont alors modifiés, tout comme le point G. Mais ni D1 et ni D2 ne dépendent de C c'est pourquoi ils ne sont pas détruits consécutivement à C. Par contre, G l'est.

Lorsque l'objet détruit est un objet final d'une macro, mais que cette macro construit d'autres objets, alors ces autres objets ne sont pas détruits, et les objets intermédiaires, internes à la macro et non accessibles, ne sont pas non plus détruits.

La figure ci-dessous illustre ce phénomène sur l'exemple de la macro définie à partir du problème ouvert. Le cercle de centre O est résultat de l'appel à cette macro, mais aussi le point E. La destruction de ce cercle n'a pas d'influence sur le point E.



Un utilisateur non "averti" (expert) ne peut maîtriser les conséquences de la destruction d'objets dans tous les cas.

En effet, en dehors des objets visibles dans l'état courant de la figure, le logiciel ne lui donne pas d'information sur les changements effectifs consécutifs aux destructions d'objets. L'utilisateur n'a pas les moyens d'estimer si ces destructions détruiront d'autres objets, dépendant de ceux-ci, et de réfléchir à (d'envisager) d'autres alternatives si ces destructions ne sont pas désirées. Une information sur l'état courant de la figure remédierait à ce problème et aux difficultés qui en découlent.

b. Redéfinition de contraintes

Les outils de redéfinition des contraintes entre les objets agissent sur l'ordre dans lequel le système constitué par chacune des contraintes peut être résolu.

En effet, la séquence des constructions qui constitue une figure peut être vue comme un graphe dont les nœuds représentent les constructions et les arcs les dépendances entre chacun des objets construit par chacune d'elles.

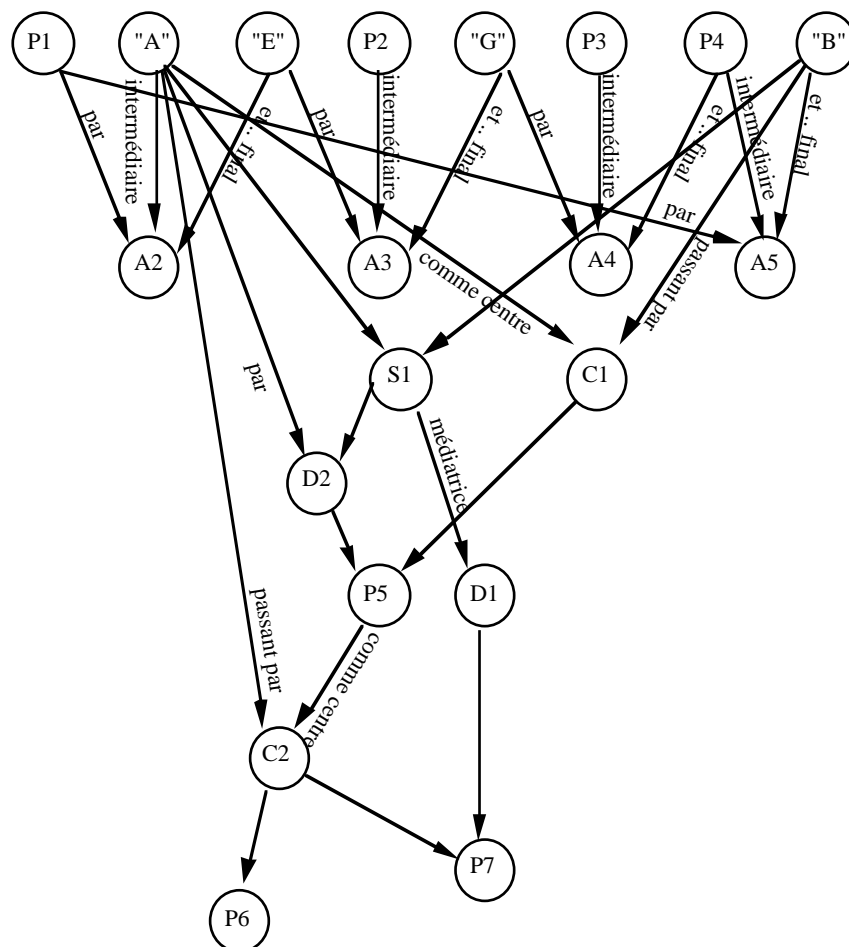
Les nœuds sont assortis de systèmes de contraintes, traduites en équations pour les objets géométriques. Les liens entre les objets orientent la propagation des résultats le long de chemins du graphe des dépendances.

Lorsqu'un nouvel objet est construit, le système de contraintes est paramétré par les objets dont il dépend. Lorsqu'il est affiché, ce paramétrage est réalisé (instancié) à l'aide des valeurs effectives prises par ces objets. Un objet ne peut être "réalisé" (résolu) que si les objets dont il dépend le sont.

Il en résulte un "ordre" (en fait un "pré-ordre" est un ordre total sur chaque chemin d'un graphe orienté) de propagation sur le graphe. Ce n'est pas un ordre "total", car certains objets peuvent être intervertis (échangés), mais l'histoire des constructions le respecte toujours, tant que l'outil de redéfinition des contraintes n'a pas été utilisé. Il s'agit d'un tri topologique.

Les objets qui peuvent être intervertis sont des objets qui n'appartiennent pas au même chemin dans le graphe orienté des dépendances. L'ordre de mémorisation des objets dans Cabri (ordre "logique") est l'ordre assujéti au minimum de modifications depuis l'ordre historique tout en respectant l'ordre de propagation des contraintes.

Le diagramme ci-dessous visualise le graphe des relations de dépendance qui lient les objets du demi-cœur dans son état sur la troisième "photo" du film.



Dans la version Mac sur laquelle nous avons travaillé, les outils de redéfinition des contraintes fournis sont des outils de redéfinition de points : redéfinir un point comme un point "sur" un objet, comme point d'intersection de deux objets, ou l'identifier avec un autre point. Dans la version synhro actuellement disponible, l'outil de redéfinition a été étendu à d'autres types d'objets que les points (droite...)

Par exemple et schématiquement, le troisième outil (identifier avec un autre point) modifie cet ordre de la manière suivante :

1. Toutes les constructions dépendant du point à redéfinir sont identifiées : toutes les constructions qui l'utilisent sont repérées, ainsi que celles qui dépendent des objets construits par ces constructions, et cela, en cascade jusqu'à la fin de la figure, constituant ainsi une chaîne.
2. Au fur et à mesure de cette "propagation", chacune de ces constructions est ré-appliquée au nouveau point.
3. Une fois toute la chaîne reproduite, la chaîne modèle est détruite.

Pour les deux premiers outils, un point est préalablement construit comme spécifié par l'utilisateur.

Dans le cas "point sur", le point est contraint à appartenir à l'objet désigné et donc à vérifier son équation. De plus, il a une position proche de la position du curseur lors du clic sur la souris.

Dans le cas "point d'intersection de deux objets", le point construit vérifie les deux équations des deux objets dont il dépend, et, si ces objets ont plusieurs points d'intersection, le point considéré est l'intersection la plus proche de la position désignée. Ce point est ajouté en dernier dans l'ordre "logique". La suite de la procédure est la même que dans le cas du troisième outil.

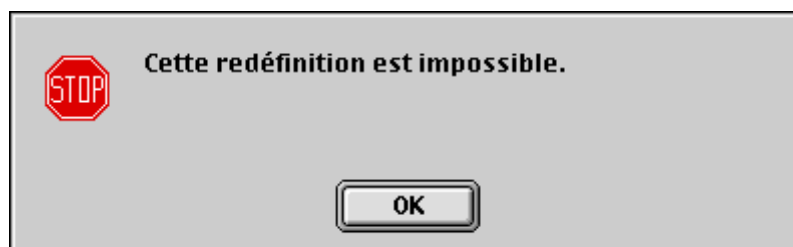
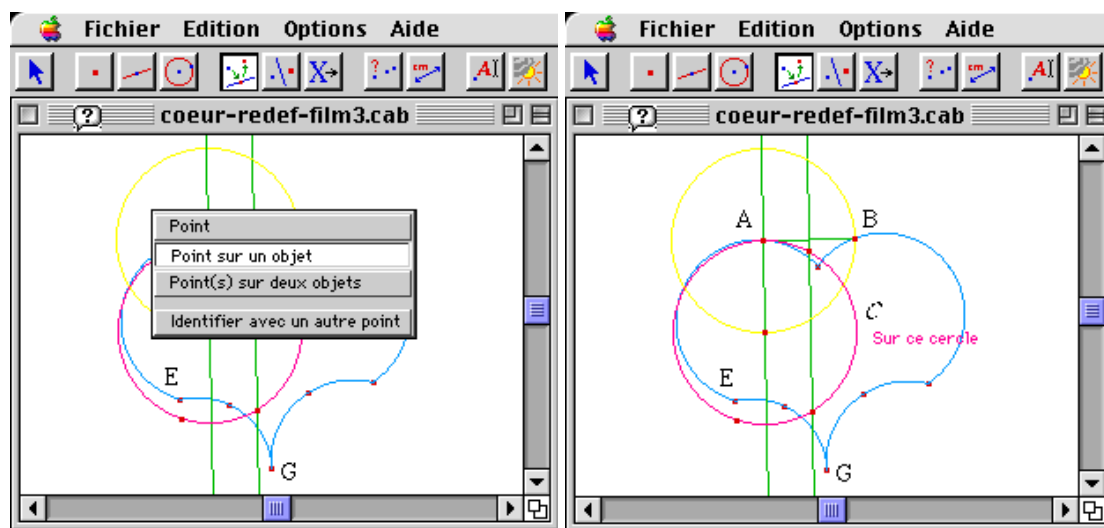
La version Windows supporte la redéfinition des autres types d'objets. Par exemple, la redéfinition d'une droite se décline au moins en cinq outils : redéfinir une droite comme passant par un point et de direction "libre", redéfinir une droite comme passant par deux points, redéfinir une droite comme parallèle à une autre droite et passant par un point, idem avec perpendiculaire, et enfin identifier une droite avec une autre. Le fonctionnement est similaire au fonctionnement de ces outils pour les points, en particulier en ce qui concerne les modifications dans l'ordre de la construction.

L'annexe A.1 montre l'effet des redéfinitions de deux points sur les deux derniers points créés entre les troisième et quatrième figures du "film" (§I.1.A.2°.b).

Les redéfinitions de points demandées dans cet exemple ne posent pas de problèmes, comme toutes celles qui utilisent l'outil "identifier avec un autre point" lors qu'aucune construction n'est basée sur le nouveau point.

Par contre, il n'est pas possible de redéfinir un point sur un objet si cet objet dépend en fait de ce point. Par exemple, il n'est pas possible dans la "troisième" figure du film, de redéfinir le point A comme un point sur le cercle C. Le graphe de dépendance doit être acyclique sinon le tri topologique n'est pas possible. Une telle tentative conduit le logiciel à refuser la redéfinition et à produire une boîte de dialogue pour signifier l'impossibilité.

L'illustration suivante présente la séquence de cette tentative.



Aucune autre information n'est fournie à l'utilisateur sur les causes de ce refus. C'est à lui de se rendre compte, par l'expérience, des circonstances qui conduisent à ce type de refus. Il doit donc se construire un modèle de fonctionnement pour le logiciel dans son ensemble, et plus particulièrement pour ces outils délicats.

Le manuel de référence indique, dans sa présentation des menus, pour l'item de redéfinition : "Si l'on tente de redéfinir un point comme étant sur un objet qui dépend lui-même du point initial, la création échoue évidemment."

Avec un modèle fiable, ces impossibilités sont identifiables lors des premières redéfinitions. Mais, dès l'enchaînement de plusieurs utilisations de ces outils, l'utilisateur, même expert, ne sait plus exactement les contraintes d'ordonnancement des résolutions de l'ensemble des contraintes de la figure qu'il manipule. Dans ce cas aussi, un retour d'information du système sur l'état de la figure avec laquelle il travaille est nécessaire.

c. Macro-commandes

La troisième fonctionnalité supportée par Cabri qui confère aux activités qui l'utilisent un caractère programmatoire, est la possibilité de définir des macro-constructions.

Comme nous l'avons vu précédemment, l'utilisation de macros permet d'aborder les activités de construction sous un aspect plus modulaire, en offrant un support de validation des sous-tâches. Pour définir une macro, il suffit de sélectionner sur un exemple d'école, d'une part les objets à partir desquels doit être extrait le modèle, les "objets initiaux", et d'autre part, les objets auxquels les constructions extraites doivent aboutir, les "objets finals".

D'autre part, lorsqu'une macro-construction a été utilisée pour l'élaboration d'une figure, cette figure peut être utilisée pour définir une nouvelle macro. Les constructions produites sont extraites de la figure résultante, gommant toute différence entre les constructions résultant de l'appel à la macro précédente, des autres. Ainsi, lorsque tous les objets de la première macro ne sont pas utiles pour la nouvelle macro, les constructions associées ne sont pas mémorisées dans la méthode en cours de définition.

Les macros de Cabri suivent les mêmes principes de base que les macros introduites dans certains langages d'assemblage :

[Augier 97]

"Une version plus évoluée d'assembleur permet d'utiliser des macro-commandes.

La notion de macro est très importante puisqu'elle permet de symboliser par un seul mnémonique un ensemble de calculs qui apparaissent régulièrement dans les programmes. Une macro peut contenir des paramètres, ils seront substitués au moment de l'assemblage par les valeurs positionnées dans le source.

Remarque : il ne faut pas confondre macro et sous-programme. Dans le cas du sous-programme c'est une portion de code qui est appelée, le déroulement séquentiel du programme est donc interrompu pour se brancher à l'adresse du sous-programme. Une fois le sous-programme terminé, le déroulement normal du programme reprend. Dans le cas des macros, le code généré par la macro est inséré dans la séquence du programme et au moment de l'exécution il n'y a pas d'interruption de la séquence. De plus il n'est plus possible dans le code généré d'identifier ce qui a été codé par macro et ce qui ne l'a pas été."

Cependant, le processus de définition est plus évolué que le simple enregistrement de toutes les primitives exécutées entre la construction de objets initiaux et celle des objets finals.

Pour illustrer ce processus d'extraction des constructions utiles pour une méthode, le diagramme ci-dessous illustre le graphe de dépendance des objets de l'exemple du paragraphe précédent (§I.1.A.2°.c). Ici, les arêtes ne sont pas étiquetées pour ne pas alourdir l'illustration.

La macro-construction à extraire de ce diagramme doit reproduire la construction qui détermine le cercle tangent au cercle C qui passe par G, c'est-à-dire C9. Le sous-graphe est visualisé en couleur, les objets initiaux en vert, leurs objets de base implicites en vert foncé comme les centres des cercles, les extrémités des segments, etc, les objets terminaux en orange, leurs points de base en marron. Les degrés de liberté restant sont schématisés par des flèches bleues sans origine.

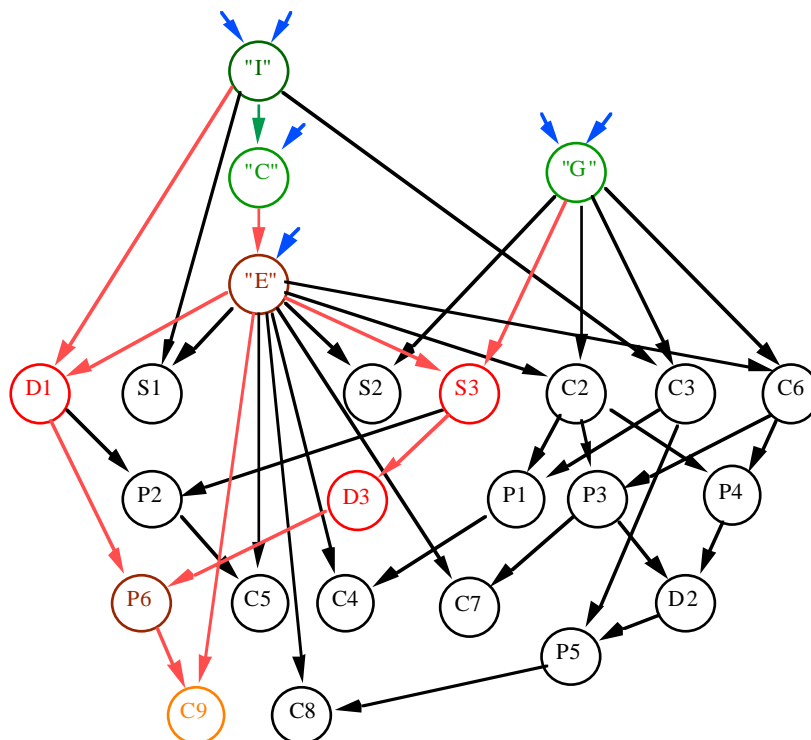
Un point libre, dans un espace à deux dimensions, a deux degrés de liberté, un par direction ; un point sur cercle a un degré de liberté ; un cercle défini seulement par son centre a encore son rayon comme degré de liberté.

Dans cet exemple, le point E est généré par la macro, alors que tous ces degrés de liberté ne sont pas résolus. En effet, Cabri permet de définir des macro-constructions capables de développer des constructions de sous-figures, en choisissant arbitrairement les positions de points définis par l'outil "Point sur un objet".

Ainsi, les méthodes de construction qui utilisent cet outil peuvent être mémorisées comme les autres outils par le processus de validation des macros. Il n'est pas nécessaire à l'utilisateur de leur imposer un traitement spécifique comme une obligation à sélectionner les points ainsi définis comme des objets initiaux.

Pour choisir ces points, de façon interne aux macro-constructions, le logiciel utilise un générateur de nombres aléatoires, et transcrit le résultat en la génération d'un point au hasard sur le support de l'objet.

Dans les diagrammes comme le diagramme suivant, ces points sont repérables par la flèche bleue qui y arrive et par leur couleur rouge ou marron.



Le nombre de constructions mémorisées grâce à ce calcul sur la séquence de toutes les constructions de la figure, est minimisé. La différence dans le cas de l'exemple du diagramme est considérable (9 objets sont concernés par la macro, alors que la figure en compte 24). Ce choix est fondamental pour respecter les contraintes de place mémoire et a permis d'intégrer le logiciel dans une calculatrice, la TI-92.

Ce diagramme ne fait pas apparaître l'ordre effectif des calculs, puisqu'il ne donne aucune indication sur cet ordre pour les contraintes de même niveau dans le graphe. Il représente la structure logique de la figure, mais pas l'ordre "logique".

Les macro-commandes ont un caractère magique et impénétrable pour l'utilisateur car les objets internes aux macros ne sont pas accessibles, alors que le processus d'apprentissage automatique est de concept élaboré.

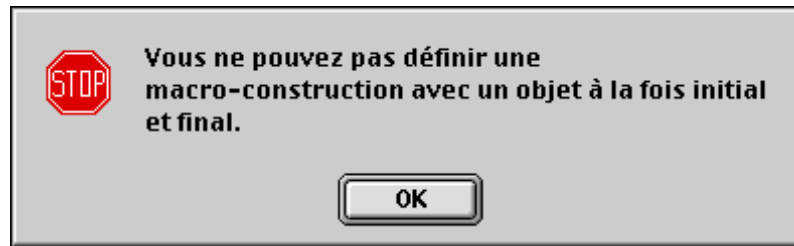
L'utilisateur n'a donc pas de moyen interactif pour s'informer sur le contenu exact des macros en termes de constructions et d'objets construits. Une première raison à ce choix d'inaccessibilité des macros est lié à leur utilisation pédagogique, comme des boîtes noires. Une autre est l'objectif d'uniformiser leur comportement avec celui des outils de base, de les banaliser afin de favoriser leur réutilisation, même quand elles ont été définies par d'autres. Un professeur peut fabriquer ses propres outils, dont les élèves ne peuvent distinguer le comportement de celui des autres outils.

Cependant, en phase de mise au point d'une macro, l'utilisateur ne peut pas vérifier a posteriori que ses manipulations sont bonnes. Deux niveaux d'échec peuvent se présenter :

- refus de validation par le logiciel,
- comportement du résultat en désaccord avec les désirs de l'utilisateur.

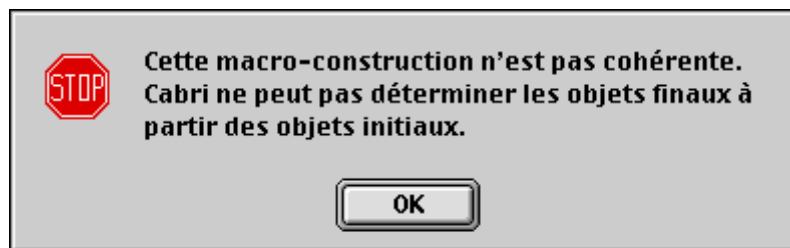
Dans le premier cas, le problème provient d'un mauvais choix des paramètres de la macro. Les moyens de l'utilisateur pour y remédier sont de retrouver les dépendances entre les objets. En cas de récurrence ou de nouvelle instance du problème, il peut refaire la construction "à la main" sur une figure d'école, et spécifier ses paramètres depuis cette nouvelle figure.

Sur l'exemple précédent, l'utilisateur ne peut spécifier sur cette figure la macro qui construirait le segment [IE] (=S1) et le point G depuis le cercle bleu C6 de centre G. S'il essaye, il aboutit à l'affichage de la boîte de dialogue :



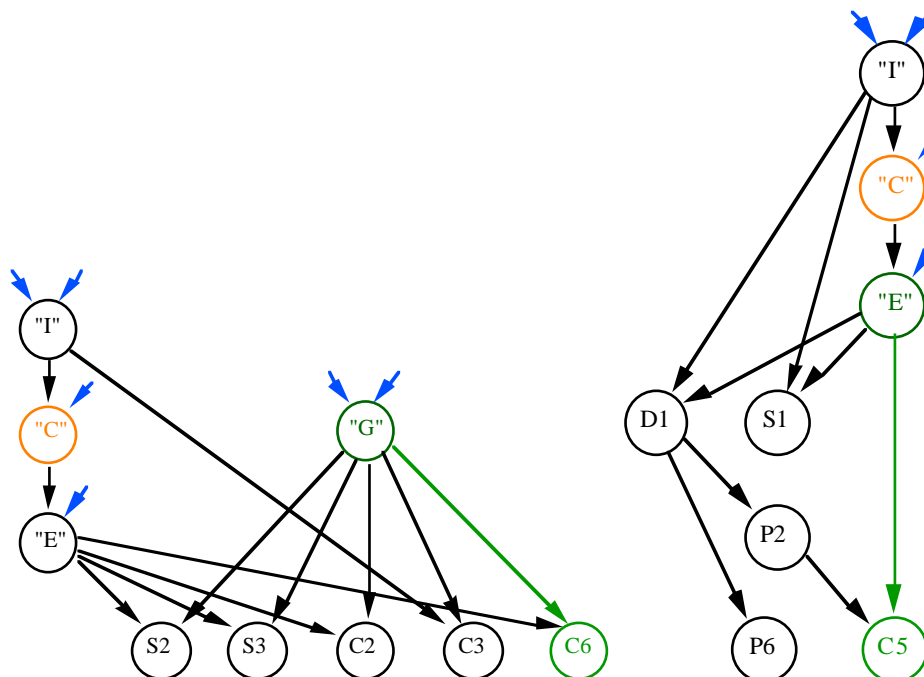
car le point G est à la fois objet initial, car spécifié explicitement comme tel, et final en tant que paramètre implicite de C.

Dans d'autres tentatives, la boîte de dialogue de refus est la suivante :



comme par exemple, le choix du cercle bleu, C6, de centre G comme objet initial, ou celui de C5, le cercle rose, dont le centre est à l'intersection de la droite (IE) et du cercle de centre E (comme objet initial), et du cercle C comme objet final.

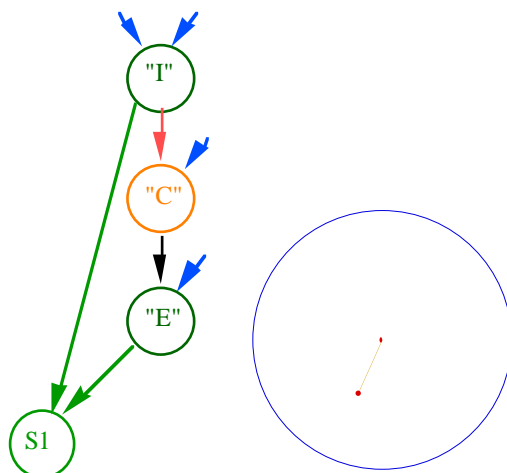
Les diagrammes suivants montrent que la relation de dépendance entre les objets demandés est inadaptée ou inversée : il n'y a pas moyen de remonter depuis les objets résultats, le long des arêtes du graphe, et de s'arrêter sur les objets initiaux ou les objets de base implicites, par tous les chemins entrepris.



Par contre, Cabri valide la macro qui construit le cercle C depuis le segment [IE], alors que E a été construit contraint à être sur C lui-même, donc dépendant de C.

En fait, ce qui se passe ici, c'est que I et E sont pris par le logiciel comme des objets initiaux car ce sont les extrémités implicites du segment S1. Dans la macro résultante, le cercle est contraint à avoir l'origine du segment comme centre, mais n'est aucunement contraint par son extrémité.

L'illustration suivante montre un extrait du diagramme précédent avec ses couleurs mises à jour par rapport à la situation de cette définition d'un part, et le résultat d'un exemple d'appel d'autre part. L'arête en rouge est la seule relation de dépendance mémorisée dans la macro.



Dans tous ces cas, un retour d'information efficace sur les causes effectives du refus, expliquant par la structure logique de la figure le refus, comme dans ces explications, serait suffisant pour prévenir l'utilisateur des impossibilités. Il permettrait de comprendre les contraintes logiques de définition des macro-constructions depuis la figure de travail de l'utilisateur.

Concernant l'inadaptation de la macro résultante aux désirs de l'utilisateur, un moyen immédiatement disponible est un test direct des comportements d'appel de la macro sur des objets initiaux liés entre eux comme dans la figure d'école, et sur des objets initiaux liés différemment.

Lorsqu'une macro-construction est appelée pour la construction d'objets d'une figure, chacune des constructions de la liste des constructions qui constituent la méthode adaptée au type des objets sélectionnés, est appliquée à ces objets, et la figure est augmentée de la liste formée par chacun des objets résultants.

Il n'y a pas, dans le processus d'extraction des constructions, de vérification sur les propriétés des objets initiaux. Dans le cas de l'exemple, le fait que G appartienne ou non au cercle C n'est pas une condition d'application et ne peut être spécifié comme tel.

De même, en imaginant une construction élaborée sur un objet de base de type polygone, mais qui est en fait un carré (par construction), une macro définie avec ce carré comme objet initial pourra s'appliquer à tous les objets de type polygone, indépendamment du fait qu'ils sont ou non des carrés (par leur propriétés comme par les constructions qui les ont définis). Même le nombre de sommets du polygone ne peut être spécifié comme un prérequis.

La figure complète est constituée par l'ensemble des contraintes qui lient tous ses objets, ceux qui proviennent d'appels de macro et ceux qui proviennent d'appels d'outils de base, indifféremment.

C'est cette figure, cette structure globale, que l'utilisateur observe et anime, c'est grâce à elle qu'il confronte son idée a priori avec le comportement obtenu.

Les macro-constructions contenant en général plusieurs constructions, la complexité de la figure est augmentée d'autant. Alors la tâche de mise au point de macro est rendue plus difficile par l'inaccessibilité des constructions intermédiaires.

Non seulement l'utilisateur n'a pas accès directement au contenu exact de la macro, mais en plus il ne peut la tester que dans un contexte d'appel. Il devra différencier parmi les propriétés de la figure celles qui proviennent des contraintes "initiales" de celle qui proviennent de l'utilisation de la macro.

De plus, à l'appel, les objets que la macro a effectivement construits ne sont pas toujours ceux qu'elle aurait pu construire dans d'autres circonstances. En effet, si les constructions à produire mémorisées dans une macro contiennent une construction déjà existante, il n'est pas nécessaire que le logiciel effectue cette construction, car cela reviendrait à dupliquer l'objet.

Et de fait, il ne le fait pas plus que directement dans la figure : par exemple si un point est construit comme intersection de deux droites, on ne peut reconstruire l'intersection de ces deux mêmes droites.

Par contre, si l'utilisateur construit une autre droite sur une première, en passant par deux points pris "sur " cette première, alors il peut construire l'intersection de cette nouvelle droite avec la seconde, qui de fait sera confondue avec le premier point d'intersection.

Cependant, ces deux points ne sont pas identiques de par les relations qui les lient aux autres objets de la figure : ils sont logiquement différents, mais confondus physiquement. Leur coïncidence est une propriété, et non une contrainte explicite : elle n'est pas visible sur le programme, seulement sur la représentation géométrique.

Lorsque le comportement obtenu n'est pas le comportement attendu, l'utilisateur modifie la figure et doit redéfinir la macro.

Un seul autre moyen est envisageable : sauvegarder la figure sous forme textuelle, éditable par tout éditeur classique, et rouvrir le fichier modifié dans l'environnement de Cabri. Mais c'est un moyen périlleux sous la forme actuelle de la version officielle du logiciel, du fait de son état de lisibilité.

La mise au point des macro-constructions est une activité complexe qui peut être résumée comme la résolution incrémentale de boîtes noires avec refonte totale entre chaque étape. Elle consiste en une recherche de structure logique de figure adaptée d'une part au comportement désiré, et d'autre part au respect des exigences spécifiques du processus d'extraction des macros. Dans cette activité, l'utilisateur manipule cette structure logique sans y avoir un accès direct.

Il n'est pas étonnant que ces trois situations typiques où le caractère programmatoire des activités supportées par ce logiciel n'est plus de la programmation séquentielle simple, rende impérieux le besoin d'un retour d'information sur la structure logique effective des figures et des macros manipulées. Par ces activités programmatoires, c'est justement sur cette structure logique que l'utilisateur agit.

2°) Accès à la structure logique

Pour résumer ce qui précède, les activités programmatoires se concrétisent en des tâches de manipulation de la structure logique des figures et des macros, et la vue géométrique ne fournit pas un accès direct à la structure logique de la figure. En effet, si les objets construits peuvent être manipulés directement, les relations entre eux sont introduites implicitement par les spécifications des outils puis des objets auxquels ils s'appliquent.

Un retour d'information éphémère est fourni sous le curseur au moment de la sélection de chacun des objets. De plus, ces relations ont un effet sur les réactions des objets construits et donc leurs propriétés. En dehors de ces deux réactions de l'interface, les seules informations persistantes sont accessibles soit par l'outil "Revoir la construction" qui fournit un magnétophone permettant de voir se reconstruire la figure pas à pas (sauf pour les objets intermédiaires des macro-constructions), soit par la sauvegarde lisible de la figure sous un format textuel.

L'impossibilité de déboguer graphiquement, et la nécessité de mettre au point des macros et des constructions "imaginaires", ainsi que de prendre conscience de la structure de la construction pour la redéfinition des contraintes et la destruction d'objets, conduisent donc à la nécessité de fournir une forme reflétant le "programme" qui a construit la figure.

Que faut-il visualiser ?

Dans le cas des redéfinitions de contraintes, l'historique de la construction ne représente pas non plus la structure logique, et ne permet pas de voir toutes les contraintes effectives entre les objets. L'historique mémorise une information suffisante pour qu'un utilisateur qui maîtrise le fonctionnement interne du logiciel puisse présumer de la structure logique effective. En effet, l'historique permet de recalculer le résultat des manipulations, puisque toutes les actions y sont mémorisées. Mais l'information utile pour l'utilisateur qui construit une figure, est justement ce résultat des manipulations, c'est-à-dire ce que le logiciel visualise graphiquement. Cependant, l'historique garde son intérêt d'espion pour l'utilisateur-professeur (mais ce n'est pas notre objet ici).

Plusieurs supports sont envisageables, des graphes comme ceux qui illustrent le paragraphe précédent et ceux qui sont utilisés en programmation visuelle, ou des textes comme le fichier de sauvegarde textuel de figure, mais il faut alors qu'ils soient manipulables directement depuis le logiciel, à l'instar des langages de commande ou de programmation.

Ce support doit être non seulement lisible, mais aussi compréhensible et utilisable par les utilisateurs, qui dans le cas de Cabri sont des utilisateurs "tout public", ensemble hétérogène contenant non seulement des utilisateurs novices, mais aussi des experts, et possédant tous les profils, du chercheur à l'ingénieur en passant par le professeur et l'élève de tout niveau.

La section suivante (a) a pour objectif de faire le point sur tout ce que peut apporter un tel support, d'abord en tant que tel, c'est-à-dire inerte. Elle s'attache aux aspects spatiaux. La section (b) s'attachera ensuite aux mérites de l'aspect animation, donc aux aspects temporels utilisables pour donner accès à la structure logique des programmes.

a. Fichiers textuels d'enregistrement

Dans l'état actuel de Cabri-géomètre, il est possible d'enregistrer les figures et les macros sous une forme textuelle. Des exemples de ces fichiers sont donnés dans les annexes A.1 et A.2. Ils mettent en correspondance les fichiers obtenus avant et après une redéfinition de contrainte, et une validation de macro respectivement.

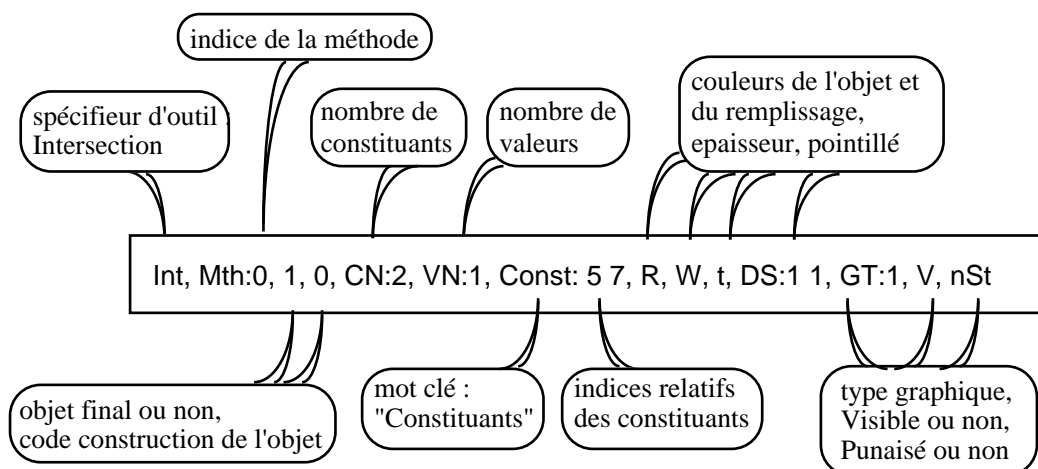
Le langage utilisé est décrit dans [Tessier 98]. Les choix qui l'ont régi sont très liés à la contrainte "calculatrice", au détriment de la lisibilité.

Forme des fichiers d'enregistrement

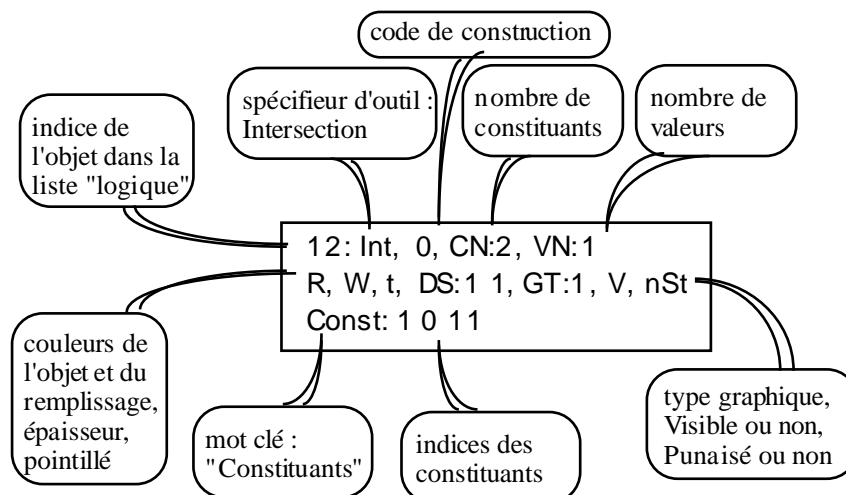
Après quelques lignes d'en-tête, ces fichiers sont constitués de deux parties : la première décrit les macros utilisées dans la figure enregistrée, et la deuxième la figure enregistrée elle-même.

Les descriptions des macros commencent par une en-tête, spécifiant les types des paramètres, l'icône et d'autres caractères globaux de la macro, puis pour chaque méthode, la liste des différentes constructions, en consacrant une ligne par objet à construire.

Quelques indices de lecture du codage utilisé, sont présentés à l'aide de l'extrait de la figure d'école de la macro de l'annexe A.2 qui concerne la construction d'un point d'intersection. Dans la macro, le format de la ligne correspondante est décrit par l'illustration suivante :



La figure est décrite objet par objet, dans l'ordre "logique" mémorisé. Les objets internes aux macros n'apparaissent pas explicitement dans la description de la figure, mais ils conduisent à un "saut" dans les indices des objets correspondant au nombre de leurs objets. Dans la figure, le format des lignes correspondant aux objets accessibles est décrit par l'illustration suivante :

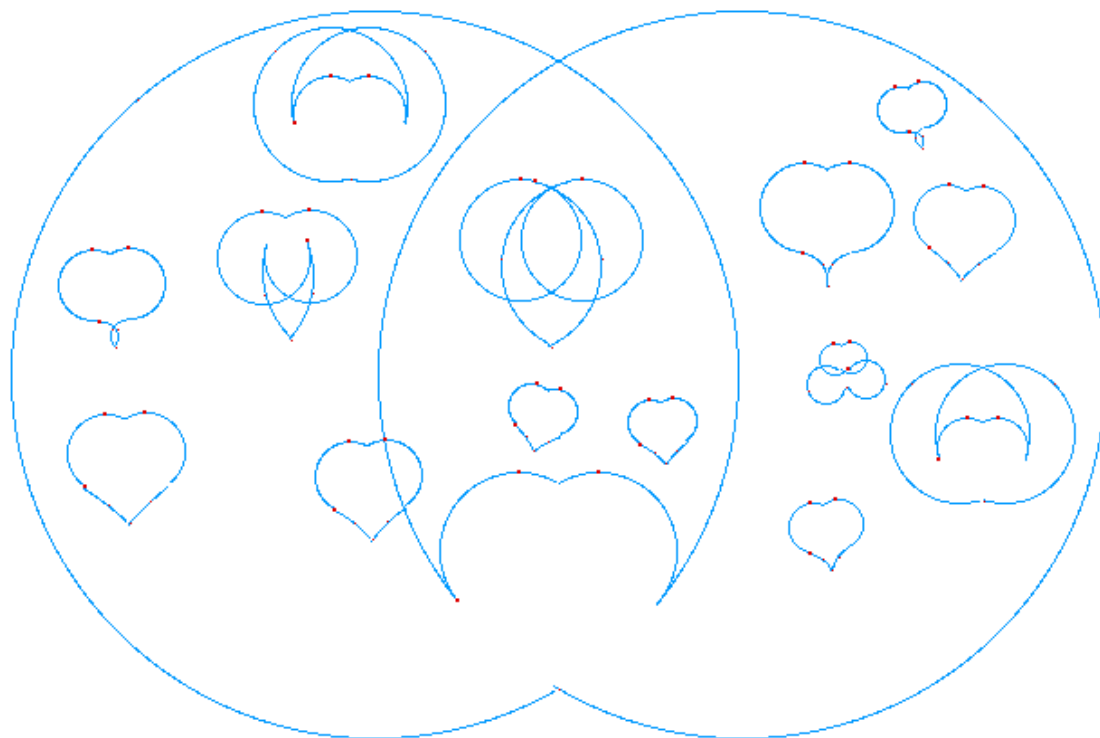


Comment l'utiliser ? assimilation du contenu, débogage

Considérons l'exemple de la mise au point d'une macro au comportement inadapté. Il s'agit de la définition de la macro qui dessine un cœur depuis deux sommets (cf §I.1.A.1°.d).

D'après cette figure, le point E a l'air d'être contraint à rester sur l'arc RS. Cependant, les résultats des appels à la macro résultante ne semblent pas avoir les mêmes propriétés que la sous-figure de la figure d'école. Il semble même que cette macro ait un comportement aléatoire, puisque les différents cas de figure suivants sont atteints tour à tour. Et même, lors des animations du point E, différentes figures résultant de l'appel de la même macro ne se comportent pas de la même manière.

L'illustration suivante montre plusieurs appels de cette macro.



Déterminer les raisons de ce résultat consiste exactement en une activité de débogage. La figure d'école a le comportement désiré, mais la figure d'appel a un comportement aléatoire.

L'examen des fichiers d'enregistrement des deux figures montre que le contenu extrait de la figure d'école pour la macro correspond exactement à celui de la figure d'école. Un examen plus fin, c'est-à-dire l'inspection un à un des outils utilisés, fait apparaître l'utilisation de l'outil "Point sur objet" pour construire le point intermédiaire de l'arc RS. Le comportement aléatoire de cette macro vient de cette utilisation.

Ses limitations (identification immédiate des objets)

Cette forme n'est pas satisfaisante dans son état actuel pour plusieurs raisons : d'abord elle est très difficile d'accès pour tous les utilisateurs, quel que soit leur degré de familiarité avec l'informatique. En tout état de cause, elle profiterait d'une reconsidération de son langage.

Cependant, indépendamment du codage, un simple enregistrement textuel ne suffit pas. En effet, même s'il rend accessibles les objets imaginaires, il ne permet pas de comprendre aisément la structure logique du programme de construction de la figure et n'est que d'une aide limitée pour déboguer les macros.

En effet, le contenu pur du texte, c'est-à-dire de la quantité et de la qualité d'information fournie par le support, est complet puisque la seule donnée d'un tel fichier permet au logiciel de reconstruire la figure.

Mais en dehors de ce contenu, un tel texte inerte n'est pas efficace pour répondre aux besoins de visualisation de programme et de débogage. Dans son efficacité pour ces tâches intervient d'une part l'identification des objets du programme, et l'adéquation des moyens utilisables pour les manipuler, et d'autre part les mêmes éléments pour les relations qui les relient.

Par exemple, en reprenant la question de la différenciation des propriétés et des contraintes : deux objets peuvent être logiquement différents, mais confondus physiquement. Leur coïncidence est une propriété, et non une contrainte explicite : elle n'est pas visible sur le programme, seulement sur la représentation géométrique.

Donc, deux objets peuvent se confondre sur la figure géométrique et se différencier sur le support structurel (ou "logique"). Ainsi une difficulté apparaît dans la mise en correspondance des bons objets entre eux.

Dans l'autre sens, beaucoup de constructions peuvent résulter localement du même type de manipulation d'objets géométriques. Dans le cas de l'extraction de la macro de l'annexe A.2, six cercles passent par le point E, et trois points sont des points d'intersection du cercle C2 avec un autre cercle.

Le tableau ci-dessous rassemble les extraits du fichier d'enregistrement de cette figure pour ces cercles et ces points, ainsi que quelques autres. Les différences entre chacun des points d'une part, et chacun des cercles d'autre part, ne sont pas frappantes, aussi l'identification avec leurs représentants géométriques n'est pas immédiate.

utilisation de l'outil Point(s) sur deux objets	utilisation de l'outil Cercle
12: Int, 0, CN:2, VN:1 R, W, t, DS:1 1, GT:1, V, nSt Const: 10 11	13: Cir, 0, CN:2, VN:2 O, W, t, DS:5 8, GT:0, V, nSt Const: 12 6
14: Int, 256, CN:2, VN:1 R, W, t, DS:1 1, GT:1, V, nSt Const: 9 10	15: Cir, 0, CN:2, VN:2 P, W, t, DS:8 14, GT:0, V, nSt Const: 14 6
18: Int, 32896, CN:2, VN:1 R, W, t, DS:1 1, GT:1, V, nSt Const: 10 17	17: Cir, 0, CN:2, VN:2 IBI, W, t, DS:1 1, GT:0, V, nSt Const: 5 6
20: Int, 32897, CN:2, VN:1 R, W, t, DS:1 1, GT:1, V, nSt Const: 10 17	19: Cir, 0, CN:2, VN:2 Br, W, t, DS:5 8, GT:0, V, nSt Const: 18 6
2: Int, 256, CN:2, VN:1 R, W, t, DS:1 1, GT:1, V, nSt Const: 21 11	23: Cir, 0, CN:2, VN:2 dBr, W, t, DS:8 14, GT:0, V, nSt Const: 22 6
25: Int, 0, CN:2, VN:1 R, W, t, DS:1 1, GT:1, V, nSt Const: 9 24	26: Cir, 0, CN:2, VN:2 V, W, t, DS:1 1, GT:0, V, nSt Const: 25 6

Ce problème est lié à l'aspect "extérieur" de l'enregistrement par rapport à l'application : il résulte de l'impossibilité de matérialiser la relation bi-univoque entre les objets du support et les objets géométriques, que le support soit textuel sous la forme actuelle ou qu'il soit un tout autre support visuel extérieur.

L'utilisateur doit mettre en relation "à la main" les objets logiques, c'est-à-dire les indices dans le cas présent, avec les objets "réels" représentés sur la figure.

Une autre limitation de l'utilisation d'un support extérieur est lié au degré d'interaction dont il bénéficie. Comme il n'est manipulable que depuis l'extérieur de Cabri, toute modification conduit à une réintroduction de la figure complète, et le logiciel doit alors fonctionner comme un compilateur.

Par contre, un intérêt important du support extérieur textuel, est justement qu'il est manipulable avec n'importe quel logiciel de traitement de texte. Sa caractéristique de complétude vis à vis de la figure, donne accès à toutes les libertés à l'utilisateur téméraire. Le texte est complet puisqu'il permet à lui seul au logiciel de reconstruire la même figure que celle qui a été enregistrée, et dans la même configuration de dessin.

Pour peu que l'utilisateur ait bien acquis le langage et bien repéré son fonctionnement, il peut utiliser à profit les spécificités manipulatoires des textes, copiant et collant des morceaux contigus de code, et ajustant a posteriori les indices des objets pour les adapter à son objectif.

b. Vues textuelles ou graphiques intégrées (=> identification immédiate des objets)

Deux supports peuvent visualiser la structure logique du programme : à base de texte ou à base de graphe. La raison d'être du support est de remédier aux difficultés qui apparaissent dans l'activité programmatoire de l'environnement de visualisation mathématique constitué par Cabri. Le concept de vue est adapté pour un tel support.

Qu'est-ce qu'une vue ?

Dans le domaine de l'édition structurée des documents, une vue est définie de façon à répondre à des exigences de visualisation spécifiques, liés au type de traitements requis par l'utilisateur.

"Une vue n'est pas une présentation différente du document, mais plutôt un filtre qui permet de n'afficher que certaines parties du document.[...] Pour chaque vue, on définit la visibilité des éléments structurels du document." [Quint 87] (p.123)

Plus généralement, une définition de la notion de vue dans le cadre de l'approche cognitive des mécanismes de l'interaction Homme-Machine est établie par [Nanard 90] (p.32) :

"Une vue est la représentation d'une partie d'un ensemble d'entités existantes (l'observé), organisée en fonction d'un modèle mental (le point de vue)."

La notion de vue est associée à la nécessité d'un point de vue particulier de l'utilisateur.

"La notion de point de vue sépare les différentes facettes d'une interface. La possibilité d'ajuster individuellement chacune des facettes favorise la mise au point des relations de correspondance (sémantique / articulatoire / opératoire) qui réalise l'interface et donc l'adaptation de celle-ci à la tâche et au point de vue de l'usager. La possibilité de contrôler l'évolution du point de vue permet de suivre la couverture de la tâche." [Nanard 90] (p.136)

Dans le cadre des manipulations de structures logiques de constructions, l'"observé" est la construction elle-même, les différentes facettes à rendre observables sont d'une part le résultat de la construction, c'est-à-dire la figure lorsque les seules constructions sont géométriques, et d'autre part la structure logique de la figure, c'est-à-dire les contraintes effectives entre les constituants de la figure qui régissent son comportement et son dynamisme.

Quels formes de vues sont envisageables dans notre contexte ?

Les différentes vues possibles doivent correspondre à un type de tâche particulier : elles doivent être adaptées à la recherche d'efficacité de l'interface vis-à-vis des manipulations requises pour ces types de tâches.

Une vue peut être interprétée non pas seulement comme un filtre, mais comme une projection. Par un filtre, tous les éléments ne sont pas accessibles, tandis que par une projection, c'est leur représentation qui est incomplète.

Dans le cadre des constructions géométriques, les manipulations des éléments requises agissent soit sur l'état du dessin, soit sur les "lois" qui le régissent.

Les modifications de l'état du dessin prennent effet sur les attributs graphiques des éléments, leur état manipulatoire ou les paramètres libres correspondant à leurs derniers degrés de liberté. Les vues capables de supporter ces catégories de manipulation peuvent être qualifiées de "aspectuelles".

Les modifications des "lois" de la figure sont atteintes par création et destruction d'éléments de la figure, redéfinition de contraintes entre certains éléments, et modification du contenu des macro-constructions : si les macros sont modifiables a posteriori, il est attendu que leurs modifications soient répercutées sur leurs appels. Dans ce cas, les vues sont qualifiées de "structurelles".

Cette classification des manipulations requises met en évidence quatre facettes utiles et séparables des constructions : trois sont aspectuelles (attributs graphiques, état manipulatoire et valeurs des paramètres libres) et une structurelle.

La facette structurelle peut être portée par deux formes de vue : les formes graphiques et textuelles. Ces deux formes répondent différemment aux manipulations de la structure logique de la construction. Chacune d'elle réalise différemment les relations de correspondance sémantique, articulatoire et opératoire [Hutchins & all. 85] (p.321, 329), [Carbonneaux 98] (p.20).

La différenciation de ces relations permet d'évaluer plus finement leur influence sur l'assimilation par l'utilisateur de la relation entre ses actions au travers de l'interface, et leurs effets sur les données (éléments) manipulées.

L'aspect sémantique se rapporte au sens des actions effectuées, i.e. à la relation entre le domaine des manipulations (c'est-à-dire le lieu des actions, donc sur l'interface) d'une part, et le domaine sur lequel elles s'appliquent (donc pour nous le programme de construction) d'autre part.

L'aspect articulatoire se rapporte au mouvement, au rapport entre le mouvement nécessaire sur le support de l'interface et le mouvement qui provoquerait la même réaction dans le domaine représenté par la représentation graphique. Typiquement, il s'agit par exemple de l'opposition entre le mouvement exercé pour déplacer la souris et le déplacement correspondant d'un objet sélectionné parmi les objets du domaine visualisé.

Un autre exemple est la sélection de deux points pour le tracé de la droite passant par ces deux points. Les mouvements exécutés sont différents des mouvements exécutés pour cette même tâche sur une feuille de papier, puisque le crayon reste appuyé entre les deux points pour laisser la trace de la droite sur le papier, alors que cliquer sur la souris demande de relâcher la pression.

L'aspect opératoire est lié à l'ordre des sous-actions correspondant à la réalisation d'une tâche. Il demande de considérer l'exécution d'une tâche comme la réalisation d'une opération.

Les vues graphiques et textuelles de la structure logique du programme de construction diffèrent selon ces aspects sémantique, articulatoire et opératoire. En voici quelques exemples.

1. Dans un texte, un déplacement de la souris dans l'état abaissé provoque la sélection de suite de caractères "survolés". Cette sélection est visualisée par l'affichage en inversion vidéo de la chaîne concernée. Dans un programme de construction, un tel mouvement a pour effet la sélection d'éléments contigus du texte du programme.

2. Un simple clic est différencié du double ou du triple clic, et a pour effet des sélections automatiques d'entités textuelles imbriquées : le caractère dans le mot, le mot dans la ligne, la ligne dans le paragraphe.

3. Dans un graphe, des nœuds sont habituellement sélectionnés par l'intermédiaire de la mise en évidence dans un premier temps d'un rectangle de sélection, puis d'une répercussion visuelle de l'état particulier "sélectionné" sur les représentations des nœuds contenus dans cette zone. Les double et triple clics peuvent trouver dans la manipulation des graphes, leurs "pendants" aux manipulations textuelles : le sous-graphe à la distance de un lien, de deux liens...

4. Les zones textuelles sélectionnées peuvent être déplacées et positionnées toutes ensemble ailleurs dans le texte. Elles peuvent être remplacées globalement par d'autres. Leur lien avec les autres éléments du texte est établi implicitement dans la séquence constituée par le texte.

5. Les nœuds ou les sous-graphes sélectionnés peuvent être déplacés tous ensemble, mais leurs positions et leurs liens avec le reste du graphe nécessitent une spécification plus complexe parce qu'elle dépend de choix plus nombreux. Il n'y a pas seulement un précédent et un suivant comme dans le cas du texte, mais les liens de chaque nœud sélectionné avec chaque nœud restant sont possibles.

6. Dans les documents structurés, les éléments sélectionnés peuvent être déplacés dans la structure logique du document, par des commandes spécifiques : un titre peut être transformé en sous-titre d'un élément de structure de niveau inférieur, tout le contenu du paragraphe correspondant peut être déplacé simultanément (implicitement).

Voir et manipuler un programme comme un document structuré

Les manipulations de documents structurés permettent d'atteindre directement des manipulations de la structure du texte. Chaque entité textuelle n'est pas considérée seulement comme une chaîne de caractères, mais aussi comme un composant textuel avec un rôle bien spécifique dans la structure du document.

Un document structuré est un document respectant une décomposition logique. Les différents éléments de la structure logique véhiculent des informations sur leur position dans la décomposition en plus de leur contenu propre.

Par exemple, un titre de sous-paragraphe est une chaîne de caractères constituant le titre d'un élément de structure sous-paragraphe, et un sous-paragraphe est contenu dans un élément de structure paragraphe, lui-même contenu dans un document.

Un programme est un document structuré. Le problème qui se pose quand on veut considérer une vue d'un programme comme un document structuré, est l'adéquation de la structure du texte du programme avec la structure logique de la construction qu'il spécifie.

Ce problème apparaît dans la correspondance à réaliser entre les manipulations des deux structures : à quoi correspond la sélection d'un élément de chacune de ces structures, quelles sont les modifications qu'elle peut subir, à quoi correspondent ses déplacements ?

Voir et manipuler un programme comme un graphe

Les graphes et les sous-graphes peuvent être manipulés de façons spécifiques aux graphes par des outils structurels pour modifier la structure du graphe, des outils graphiques pour modifier l'aspect du graphe, des outils visuels pour une meilleure visualisation et appréhension du graphe.

Les fonctionnalités adaptées aux manipulations des graphes sont par exemple l'ajout ou la suppression de sommets ou d'arêtes, la modification d'une des extrémités d'une arête, la visualisation de sous-graphes répondant à certaines spécificités, la contraction de sous-graphe changeant de niveau d'abstraction dans la représentation du graphe, la décontraction d'un graphe, son éclatement, le déplacement d'un sous-graphe vers un autre point du graphe [Carbonneaux 98] (p.214).

L'utilisation d'une clé, c'est-à-dire les manipulations de la souris simultanées à la pression d'une touche du clavier, permettent de changer de mode opératoire en cours de manipulation. Tant que la touche est abaissée, l'interface répond différemment aux manipulations de la souris. Cette différenciation permet de manipuler les données sous des angles différents, comme par l'intermédiaire des vues.

Passer d'une forme de vue et de manipulation à l'autre

L'intérêt des vues intégrées est la possibilité de visualiser les objets simultanément dans les différentes vues, et d'utiliser la dynamique de l'interface commune pour concrétiser la correspondance entre les différents représentants des mêmes objets.

La manipulation d'un objet par l'utilisateur à travers une vue, peut être accompagnée d'un retour d'information de l'interface sur les autres vues de l'objet. Toutes les vues peuvent visualiser simultanément, mais à leur manière, l'état sélectionné ou non des données qu'elles affichent. C'est le principe de l'ubiquité des objets.

Chaque vue peut être adaptée à une activité prenant place dans le logiciel.

L'activité d'exploration des propriétés géométriques de la construction d'une figure s'appuie en général plus sur une vue géométrique.

L'activité programmatrice nécessitant des manipulations de la structure logique de la construction, pour évoluer dans cette activité, l'utilisateur peut s'appuyer sur une vue qui d'une part visualise les informations sur lesquels il agit, et d'autre part supporte les manipulations dont il a besoin.

L'activité de présentation du résultat final de la construction, afin de dégager certains objets géométriques et d'en cacher d'autres, est une autre activité qui ne nécessite pas forcément la même vue qu'une activité de programmation, qui, elle, ne s'intéresse pas à l'aspect des objets, mais à leur rôle dans la structure logique de la construction.

Ainsi, les vues peuvent être considérées comme des projections de l'état du logiciel. Ces projections sont orientées par une direction pragmatique liée à l'activité.

D'autre part, certaines vues sont complémentaires du point de vue de l'information qu'elles visualisent, et peuvent même recouvrir l'ensemble des informations concernant chacune des données mémorisées dans le logiciel. Il s'agit de la notion de complétude.

De plus, bien que les vues puissent être considérées comme des projections de l'état du logiciel selon certaines directions liées à leur qualité manipulative, les différentes vues peuvent être combinées pour devenir des supports autonomes qui véhiculent toutes les informations sur l'état du logiciel.

L'adjectif "structurel" peut être utilisé pour qualifier la vue choisie. La forme du support peut être textuelle ou graphique.

Pour notre part, il ne s'agit pas de choisir entre la programmation textuelle et la programmation visuelle, notre objectif étant de fournir une vue pour le programme déjà existant. Un des principes de base de la conception de Cabri-géomètre est de laisser autant et aussi loin que possible l'utilisateur maître de l'état du logiciel. Cette qualité est à rapprocher de la sensation d'engagement direct de l'utilisateur dans ses actions au travers de l'interface, par la relation entre le sentiment de responsabilité et la confiance en sa connaissance du fonctionnement du logiciel.

De plus, tous les profils d'utilisateurs existent autour de Cabri : utilisateur-élève, utilisateur-professeur, utilisateur-chercheur ou indépendant. Ils diffèrent aussi par leur niveau de familiarité avec Cabri d'une part, et plus généralement avec toute activité programmatrice. Chacun d'eux doit pouvoir travailler avec les représentations qui lui correspondent le mieux.

Le choix d'une vue textuelle plutôt que graphique pourrait être laissé à l'utilisateur. Il pourrait aussi désirer travailler avec une vue textuelle et une vue graphique simultanément ou en alternance. En effet, les deux formes de vues sont complémentaires, de par la qualité de manipulation des objets propres à la représentation des objets, et de par la nature des informations supportées par la description et leur rapport à la réalité.

Dans le cas de l'intégration des vues, ce sont les mêmes objets qui sont représentés dans chaque vue. Il n'y a pas de gestion simultanée de différentes structures de données pour supporter les manipulations de chacune d'elles. Toute modification de l'état d'un objet depuis une vue se répercute sur les autres vues : les objets n'existent qu'en un seul exemplaire.

L'intégration des vues structurelles contribue à la qualité de l'interface selon l'impression de localité [Pane & Myers 96]. Enfin, la proximité entre les entités d'une même donnée n'est pas seulement mesurée en termes de distance métrique, mais aussi par la simultanéité des animations.

c. Nécessité de la vue textuelle dynamique et synchronisée (modification "in extenso", apprentissage du langage par "bain linguistique")

La vue constitue un support (une forme) dynamique et synchronisé. Sa qualité de synchronisme est caractérisée par le fait qu'elle réagit en même temps que la figure vis-à-vis des créations et destructions d'objets, et des redéfinitions de contraintes. Sa qualité dynamique est caractérisée par le fait qu'elle peut être utilisée seule et possède des fonctionnalités d'édition et d'animation adaptées à sa forme et à sa qualité. Les deux intérêts principaux fournis par ces qualités sont les suivants :

1°) La possibilité d'éditer la figure seulement sur la vue structurelle pour certaines manipulations pour lesquelles la vue choisie est plus efficace. Par exemple, le déplacement de "blocs de programmes" est plus efficace sur une vue structurelle que sur la vue "résultat" (ou géométrique), que ce soit à l'intérieur d'un programme ou d'un programme à un autre. Les modifications de branchements d'une vue graphique sont plus efficaces dans la vue graphique, justement. Elles correspondent à des redéfinitions particulières de contraintes. Il est possible de rendre perceptible à l'utilisateur que les liens qu'il manipule refusent de se brancher sur certains types d'objets, et acceptent de se brancher sur certains autres, en modifiant la nature du lien. Cela correspond dans la vue résultat au refus du système de "voir" les objets survolés dont le type ne convient pas.

2°) L'allègement de la nécessité de l'apprentissage de la fonctionnalité par l'utilisateur, puisqu'il peut être aidé par un apprentissage implicite, du style "bain linguistique". En effet, les manipulations de la figure elle-même suffisent aux constructions, donc la synchronisation des effets sur les deux vues permet une imprégnation des réactions de la vue structurelle en réponse aux mêmes manipulations. Il restera l'apprentissage de l'aspect articulatoire pour les manipulations de la vue structurelle elle-même : comment sélectionner un objet dont la forme est présentée différemment.

Un autre aspect qui contribue à rendre la vue dynamique est la mise à disposition de fonctionnalités dynamiques spécifiques à la vue. Par exemple, depuis la vue structurelle, deux fonctionnalités sont judicieuses :

- l'accessibilité aux vues aspectuelles des éléments de la figure : attributs graphiques, manipulateurs et de position (valeur de derniers degrés de liberté),
- la possibilité de déplacer la vue structurelle entre différents niveaux d'abstraction : les macro-commandes peuvent être pliées ou dépliées. L'appel d'une macro-commande est plié quand son affichage est réduit au minimum d'information concernant cet appel : identificateur de la macro et paramètres. L'appel d'une macro-commande est déplié lorsque la liste des commandes effectivement exécutées est affichée.

L'accessibilité du programme de construction à différents niveaux d'abstraction est une qualité très importante pour l'activité de programmation. C'est un des critères de conception des environnements de programmation répertoriés par Pane et Myers dans leur article d'étude des solutions d'usage choisies pour les spécifications de systèmes pour programmeurs novices : [Pane & Myers 96] (p.37)

"Abstraction of functionality into modules is a powerful programming concept. It can promote information hiding, reduce the amount of code that must be understood in detail, and provide a suite of primitive that can be composed to implement new functionality. When programmers understand code at an abstract level, they are more likely to reuse that code in other appropriate places, and that reuse is more likely to be by invoking the code (making a procedure call), which is a more efficient form of reuse than making a copy of the code in the new context . But novices are not ready to use the abstraction tools which are emphasized by modern languages."

D'où l'objectif de l'intégration d'une vraie vue structurelle que nous choisirons textuelle dans le cadre des développements de cette thèse, se réservant la possibilité de doubler ultérieurement cette vue textuelle par une vue à base de graphes..

Chapitre 2

Du raisonnement humain à l'activité informatique

L'objectif de ce deuxième chapitre est de mettre en relation les comportements du logiciel avec ceux d'autres environnements qui possèdent, comme Cabri, un des aspects fondamentaux suivants : primauté graphique, caractère programmatoire, manipulation directe et public non informaticien.

Le fil conducteur de ce chapitre est l'examen des supports physiques utilisables pour aider le programmeur. Le choix de ce fil conducteur est lié à la nature des aspects fondamentaux cités précédemment et à la nécessité de choisir un tel support pour Cabri. En effet, le support utilisé pour donner accès aux données et comme moyen de communication entre l'utilisateur et l'ordinateur, est primordial dans une interface à manipulation directe. Puisqu'il s'agit d'une activité au caractère programmatoire, le programmeur même s'il est non informaticien, est considéré comme un utilisateur de méthodologie pouvant être guidé grâce à la manipulation directe et la primauté graphique et animatrice.

Partant d'une étude de l'évolution des méthodes utilisées pour aider les activités de programmation à travers une petite histoire de l'informatique, les différents courants de recherche visant à proposer des environnements pour effectuer ces activités sont abordés, en s'intéressant principalement aux supports proposés soit pour orienter vers une méthodologie, soit pour servir de média de communication entre le programmeur et l'ordinateur. L'approche visuelle est justifiée par une compensation a priori de difficultés inhérentes à l'utilisation de langages informatiques formels pour des utilisateurs novices. Mais les limites de ces approches conduisent aux études cognitives actuelles dont l'objectif est de déterminer les processus les plus naturels afin d'adapter les environnements aux utilisateurs plutôt que le contraire.

Ce chapitre nous permet, au passage, de montrer en quoi l'activité de programmation a évolué avec le progrès des techniques, et en quoi cela conduit à une diversification du "métier" d'informaticien, dans ses différentes tâches. Il nous permet aussi de rappeler ce qui définit le concept de manipulation directe, et de résumer les différentes approches utilisées pour concevoir des interfaces homme-machine.

Examinons d'abord comment, depuis les débuts de l'informatique, on a cherché à aider les intervenants, experts ou non, à programmer.

A) Une petite histoire de l'informatique

Cette petite histoire de l'informatique est axée sur l'évolution de l'activité de programmation, en passant par l'utilisation des langages de programmation, de langages de commandes, la programmation visuelle et la programmation par démonstration.

[Mathelot 69] définit la programmation de la manière suivante :

"Il s'agit de transposer les problèmes que l'on veut faire traiter à la machine sous une forme qui lui soit assimilable, c'est-à-dire de faire de la programmation (ou écrire des programmes). Qu'est-ce qu'un programme ? Les ordinateurs ne pouvant effectuer que des opérations élémentaires, l'ensemble des instructions destinées à faire effectuer toute une suite d'opérations constitue ce qu'on appelle un programme.[...] Le dialogue avec l'ordinateur peut se faire en langage machine qui est constitué d'instructions très élémentaires sous forme numérique, binaire ou 'binarisée'."

L'histoire de l'informatique a vu quelques phases d'oscillation entre programmation à base de graphes et programmation directement par langage informatique textuel.

programmation exclusivement binaire ou cartes à trous, avant 1950,	=> organigrammes.
langages comme basic ou fortran	=> organigrammes.
études théoriques sur les langages	=> perfectionnement des langages et utilisation de langages de description => expression fonctionnelle et expression actionnelle des programmes => méthodes d'analyse (descendante, par cas, récurrente)
arrivée du génie logiciel	=> recherche de réutilisation du code produit, étude des architectures des logiciels, approche "objet" (comportement polymorphe) => évolution du point de vue du programmeur vis à vis des programmes qu'il conçoit => langages spécifiques : LISP, PROLOG, Impératif / Déclaratif => méthodes de conception : ADELE, MERISE
apparition des interfaces homme-machine	=> élaboration de méthodes de conception (PAC) => et de modèles utiles pour la conception (modèles des tâches), => utilisation et mise au point de bibliothèques d'outils d'interfaces

1°) De la programmation binaire aux premiers langages algorithmiques

Dans sa description du passage du calculateur à l'ordinateur, [Augier 97] rappelle que :

"Les premiers calculateurs électroniques, comme celui de Von Neumann de l'institut des études avancées de Princeton, devaient être programmés intégralement en langage machine.[...] à cette époque, programmer signifiait souvent câbler directement les opérations dans les circuits mémoire de la machine même.[...] Les utilisateurs, qui étaient alors aussi les concepteurs de la machine, devaient planifier eux-mêmes leurs travaux et le partage des ressources, et cela de manière tout à fait empirique et surtout manuelle."

Les périodes suivantes ont vu l'apparition des cartes perforées (ou à trous) dans les années 50, et l'arrivée du transistor. Ensuite commence l'histoire proprement dite des langages de programmation. D'après [Augier 97] (p.78),

"par langage de programmation, il faut comprendre deux choses :

La première est la définition logique du langage : d'une part les symboles pour l'écriture, et d'autre part l'ensemble des règles qui constituent le langage, un peu comme le vocabulaire et la grammaire pour un langage humain.

La deuxième chose est plus tangible. Il s'agit des logiciels qui permettent de traduire le langage symbolique (compréhensible et donc utilisé par le programmeur) en code binaire (compréhensible et exécuté par l'ordinateur). Par comparaison avec les langages humains, c'est cette fois au travail d'un interprète-traducteur qu'il faut faire référence."

Les langages de première génération sont tout simplement le code binaire utilisé par les processeurs.

a. Les assembleurs et macro-assembleurs

C'est vers 1950 que sont définis les langages de deuxième génération : les langages d'assemblage ou assembleurs. Ils sont très proches du code binaire de l'ordinateur. Il s'agit de la possibilité d'utiliser des mnémoniques pour représenter les codes binaires du processeur, d'utiliser des étiquettes pour représenter des adresses en mémoire et des symboles pour représenter des valeurs particulières.

Vers 1955, l'idée de définir des macro-instructions constituées de séquences d'instructions paramétrables conduit à la définition des macro-assembleurs.

Les langages d'assemblage permettent de programmer les comportements d'un processeur. Aujourd'hui encore des langages d'assemblage sont définis pour tous les processeurs.

À un niveau plus bas, des langages appelés micro-codes sont utilisés pour programmer les cartes : ainsi certaines cartes très simples (ou épurées) sont capables d'en émuler d'autres plus sophistiquées (ou plus riches). Une carte 68000 peut émuler une carte IBM 370 ou 390 sur les PC/AT-370, puis des PS-2/370.

Remarque : contrairement aux macros du logiciel Cabri-géomètre, les macro-assembleurs permettent la récursivité vraie.

b. Les assembleurs structurés

Ils ont donné le jour aux premiers langages évolués dits de troisième génération : les langages algorithmiques. Ces langages permettent de créer des programmes avec des termes compréhensibles pour les mots-clés du langage, d'utiliser des variables de différents types : entiers, réels, caractères, booléens, etc., d'utiliser des instructions conditionnelles pour décrire la structure du programme et même des sous-programmes.

Les premiers, les langages LP, sont dépendants de la machine et sont appelés assembleurs structurés. En 1960, Wirth définit PL360 pour la machine IBM 360. Pour Bull sont définis LP70 et LP80 (machine IRIS 80), etc.

Ce sont ces langages qui donnent naissance au langage C vers 1976-1977 [Kernighan & Ritchie 78].

c. Les premiers langages algorithmiques universels

Les langages de troisième génération les plus évolués sont des langages algorithmiques universels, c'est-à-dire indépendants de la machine. Les plus anciens, vers 1955, sont COBOL pour les applications de gestion, FORTRAN pour les applications scientifiques et techniques, et BASIC à vocation générale.

d. Méthodologie utilisée

Les programmes étaient alors préparés à part sur papier. Les utilisateurs utilisaient des organigrammes pour planifier leur programmation et visualiser le processus ordonné à l'ordinateur.

Par le terme organigramme,

« on représente des sortes de boîtes noires dont on ne se soucie pas encore de savoir comment elles fonctionnent mais dont on a précisément identifié les paramètres en entrée et en sortie, ainsi qu'une fonction ou action élémentaire à exécuter sur ces paramètres ou sur l'environnement global du programme. » [Augier 97] (p.73)

Trois types de boîtes sont représentées : les boîtes d'action (rectangle) qui symbolisent la réalisation d'un traitement, les boîtes de test (losange) ayant deux points de sortie pour une entrée, et les boîtes d'entrée/sortie (ovales) qui matérialisent les points d'entrées et de sortie du traitement.

Le flot des données entre les différentes boîtes est porté par des liens qui les unissent. (p. 91) *« C'est une des premières méthodes qui a permis de décrire graphiquement un algorithme, c'est-à-dire le déroulement d'un programme. [...] À l'aide de ce schéma, on peut représenter un problème, son analyse et sa résolution. »* Dans ces langages, les branchements conditionnels sont explicites et le programme peut être calqué sur l'organigramme.

« L'utilisation d'un organigramme a été et est toujours un sujet de controverses. Il est exact que, mal rédigé, son utilité est pauvre. En revanche, pour peu que l'on suive certaines règles simples, il oblige à réfléchir au problème à résoudre avant de commencer la programmation. Parmi ces règles simples, on pourrait citer :

- *toujours utiliser des flèches qui vont de haut en bas et de gauche à droite. Cela implique un déroulement séquentiel sans débranchements intempestifs ;*
- *ne jamais faire couper une ligne de liaison par une autre ;*
- *éviter les cercles de rappel. Limiter leur usage au changement de page.*

Le risque est de contourner les deux premières règles par l'utilisation de cercles. [Augier 97] (p. 91) »

Un risque similaire existe dans les langages comme BASIC et FORTRAN, conduisant à des règles de programmation correspondantes sur l'usage des instructions de branchement.

2°) Langages algorithmiques structurés (3^{ème} génération)

L'évolution s'est poursuivie avec les langages de troisième génération, i.e. les langages algorithmiques structurés et à sémantique bien définie, comme ALGOL-60, ALGOL-W, ALGOL-68, PL1, PASCAL, ADA, C, etc...

Les améliorations de ces langages portent principalement sur un choix plus large des types des variables ainsi que des instructions conditionnelles pour élaborer des tests plus sophistiqués, supprimant la nécessité de l'usage de l'instruction de branchement explicite. Avec ces langages de programmation, il devient possible d'écrire un programme avec un logiciel de traitement de texte qui autorise le programmeur à effectuer toutes les corrections, ajouts de commentaires nécessaires pour une écriture et une lecture aisées et rapides.

Le texte source du programme est entré dans l'ordinateur à l'aide d'un éditeur, traitement de texte dédié au langage qui propose éventuellement des indentations automatiques, une aide en ligne, une vérification syntaxique permanente... Ensuite ce texte source est traité par un ou plusieurs logiciels spécifiques au langage de programmation utilisé pour produire un langage exécutable par l'ordinateur.

Un compilateur est un logiciel qui lit le source du programme, généralement en plusieurs passages. À chacun de ces passages correspond une tâche particulière : analyse syntaxique, résolution des références et traduction proprement dite. À l'issue de ces traitements, un code objet est produit, qui doit être lié à d'autres codes objets (bibliothèques externes) et positionné (adresses de chargement et d'exécution) par un éditeur de liens pour constituer le programme réellement exécutable.

Historiquement, le premier de ces langages est ALGOL60, sorti en 1960. C'est aussi le premier dont la description syntaxique a utilisé une grammaire hors-contexte. En 1964, environ, Wirth définit ALGOL W, dont les qualités de clarté lui confèrent un rôle important dans l'enseignement de l'informatique.

Ensuite sortent des langages plus complexes et complets, PL/I chez IBM. ALGOL68 chez les universitaires. Ce dernier a une syntaxe définie par une "W-grammaire" (grammaires de van Wijngarden).

PASCAL (1974), encore de Wirth, est déduit d'ALGOL60, et est à vocation générale. Il est surtout utilisé pour l'enseignement. Dans la même lignée apparaissent ensuite Modula, puis Simula.

ADA est défini environ en 1980 et sort en 1985. Il est utilisé pour les gros projets, la modélisation et la conception de logiciels. C++ est une extension du langage C définie en 1990 pour supporter la programmation à objets.

Et enfin Java a été conçu en 1990, initialement sous le nom de « oak », pour la programmation d'appareillages., *« Le but principal de cette démarche était de concevoir un langage de programmation portable, sûr et également ouvert sur les technologies réseau et multimédia. L'apport principal de ce nouveau venu fut la possibilité de développer les applications pour le World Wide Web avec une déconcertante facilité grâce à la notion d'applet mise en œuvre dans le langage [...]. Le langage Java se présente comme un langage à objets à la fois compilé et interprété. Le code source de Java, dont la syntaxe est similaire à celle d'un C++ épuré, est traduit après la compilation en un code binaire universel destiné à une machine virtuelle Java. C'est ainsi qu'est réalisée la portabilité du langage qui devient optimale puisque tout code Java compilé devient exécutable sur n'importe quelle machine virtuelle Java, quelle que soit la machine sur laquelle cette dernière s'exécute [machine cible]. Cette machine virtuelle se présente sous la forme d'un interpréteur qui supporte l'exécution du pseudo-code Java compilé. »* [Keramane 97]

Methodologie

Quelles conséquences cela a-t-il sur la manière d'aborder les activités programmatoires ? Les organigrammes n'apparaissent plus en tant que tels dans les livres de cours universitaires d'informatique. La conception et la spécification des programmes s'appuient sur l'utilisation de formalismes de description d'algorithmes, comme le « pseudo-Pascal » des manuels de Hopcroft & Ullman.

Cependant ces formalismes ne sont pas des langages au sens propre de l'informatique, car ils ne sont pas traduisibles tels quels en un langage "compréhensible" par l'ordinateur. Il s'agit plutôt d'un compromis entre l'imposition de la rigueur liée à l'utilisation d'un formalisme strict, et la liberté discursive du langage naturel (et maternel). Les spécifications s'appuient sur des expressions "fonctionnelles", ou "actionnelles" des problèmes.

Les illustrations suivantes sont des adaptations des extraits de [Scholl & al. 93] à la construction d'un carré. La première montre un algorithme décrit (semi-formellement) sous forme fonctionnelle, par un "schéma fonctionnel".

Algorithme qui construit un carré depuis un de ses côtés.

Forme fonctionnelle

Spécifications descendantes de types

Segment : un type	
LeSegment : deux points -> un Segment	{constructeur}
LesExtrémités : un Segment -> deux points	{sélecteur}
LaDirection : un Segment -> une droite	{sélecteur}
Polygone : le type BasePol ajouterP (Polygone)	{constructeur}
{ ajouterP est la fonction qui ajoute un sommet à un polygone }	
LaBasePol : deux points -> une BasePol	{constructeur}
EstBasePol? : un Polygone -> un booléen	{testeur}
{ indique si le polygone considéré n'a que deux points (élément de base de la définition récurrente) }	
LePointInitial : un BasePol -> un point	{sélecteur}
LeSecondPoint : un BasePol -> un point	{sélecteur}
dernierSommet : un Polygone -> un point	{sélecteur}
premiersSommets : un Polygone -> un Polygone	{sélecteur}
DernierCôté : un Polygone -> un Segment	{sélecteur/constructeur}
LeNombreDeCôtés : un Polygone -> un entier	{sélecteur}

Spécifications descendantes des fonctions**Carré** : un Segment -> un Polygone

{ Posons $\langle C, D \rangle = \text{Carré}(S)$ où
 si A et B sont les 2 points ou les extrémités du segment S,
 alors C et D sont les deux autres sommets du polygone carré ABCD }

3eSommetTriangleIsoRect : deux points -> un point

{ Posons $P = \text{3eSommetTriangleIsoRect}(M, N)$ où
 P est le troisième sommet du triangle MNP rectangle en N
 tel que M et P soient à égale distance de N }

DroitePerpendiculaire : une droite et un point -> une droite

{ Posons $R = \text{DroitePerpendiculaire}(D, M)$ où
 R est la droite perpendiculaire à (DM) }

IntersectionsDroiteCercle : une droite et un cercle -> deux points

{ Posons $\langle A, B \rangle = \text{IntersectionsDroiteCercle}(D, C)$ où
 A et B sont les points d'intersection (réels ou imaginaires)
 de la droite D et du cercle C }

Réalisations ascendantes des fonctions**(1) 3eSommetTriangleIsoRect (M, N) :**

soit $\langle P, Q \rangle = \text{IntersectionsDroiteCercle}$
 (**DroitePerpendiculaire** (**LaDirection** (**LeSegment** (M, N)), N),
 Cercle (N, M)) dans P

(2) Carré (S) :

soit $\langle A, B \rangle = \text{LesExtrémités}(S)$
 dans **ajouterP**(**ajouterP**(**LaBasePol**(A, B),
 3eSommetTriangleIsoRect (A, B)),
 3eSommetTriangleIsoRect (B, C))

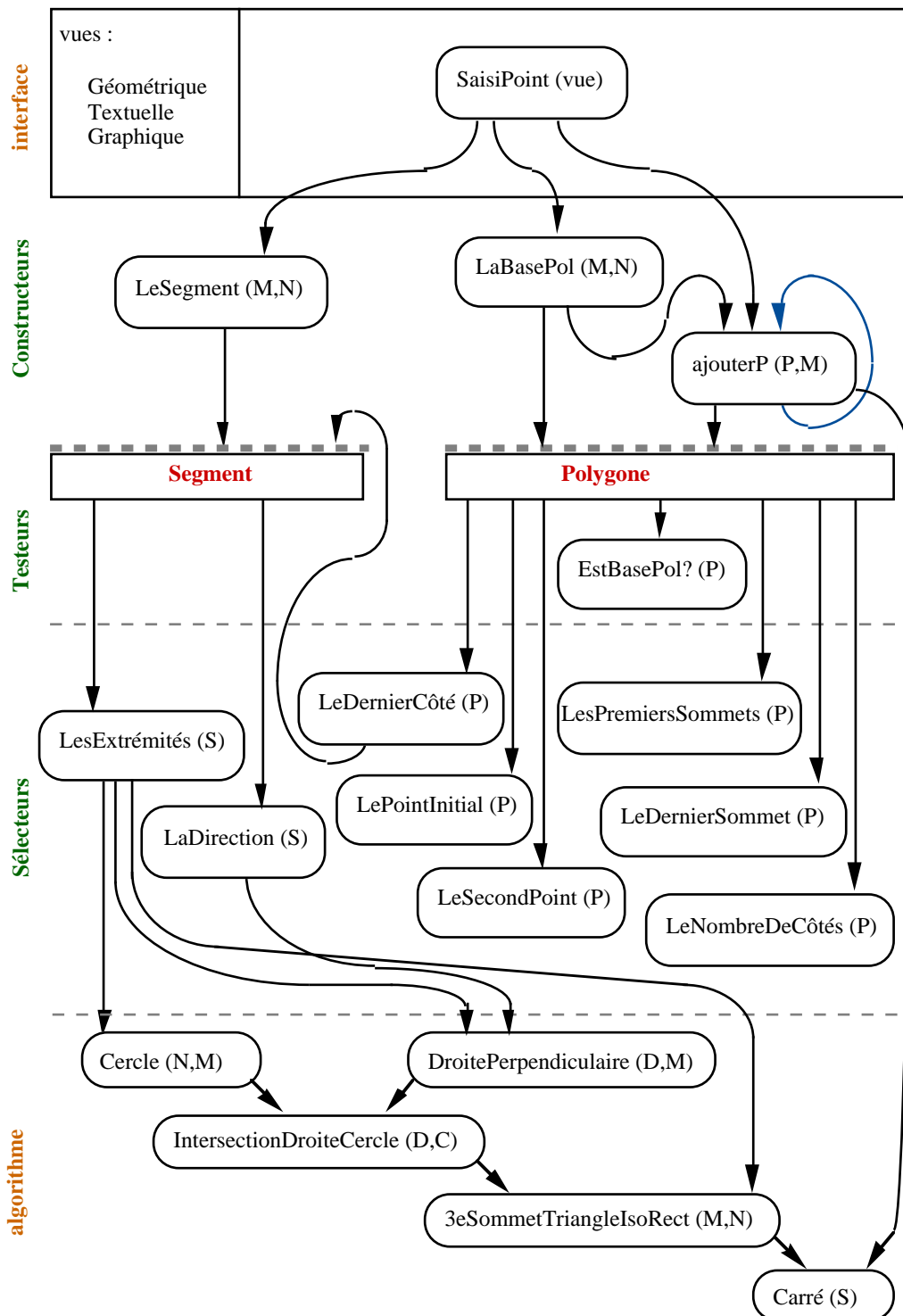
La suivante montre la description d'un algorithme proche sous forme actionnelle. Ce n'est pas exactement l'algorithme décrit précédemment, puisque dans celui-ci, l'approche choisie est basée sur une itération à exécuter deux fois : à partir du côté précédent, l'algorithme construit le segment suivant.

Forme actionnelle

lexique	{l'ensemble des noms introduits pour résoudre le problème}
P1, P2 : deux points	{les points du côté de base du carré}
P3, P4 : deux points	{ les points qui "finissent" le carré}
Saisir :	
une action (le résultat : un point)	{saisie d'un point dans la fenêtre graphique }
ConstruirePointSuivant :	
une action (la donnée : deux points, le résultat : un point)	{construit le troisième sommet d'un triangle rectangle isocèle}
Relier :	
une action (la donnée : une séquence de points)	{relie (trace) en boucle les points dans la fenêtre graphique par un polygone}
ConstruireDroite :	
une action (la donnée : deux points, le résultat : une droite)	{construit la droite qui passe par les deux points donnés}
ConstruireDroitePerpendiculaire :	
une action (la donnée : une droite, le résultat : une droite)	{construit la droite perpendiculaire à la droite donnée}
ConstruireCercle (M,N) :	
une action (la donnée : deux points, le résultat : un cercle)	{construit le cercle de centre le premier point donné passant par le second}
ConstruirePointsIntersectionDroiteCercle	
une action (la donnée : deux points, le résultat : deux points)	{ "construit" les points d'intersection du cercle et de la droite donnée, même s'ils sont "imaginaires" }
algorithme	
	{état initial : aucun point n'a été construit}
	Ecrire ("Choisir les positions des deux sommets du côté de base");
Saisir (P1) ;	{ "démarrer" : définition de l'"origine" du polygone}
Saisir (P2) ;	{ et du côté de base du polygone}
i parcourant [3...4] :	
ConstruirePointSuivant (Pi-1,Pi,Pi+1) ;	
{Pi+1 est le troisième sommet du triangle Pi-1PiPi+1 rectangle et isocèle en Pi }	
	{état final : le carré est construit}
	Relier (P1,P2,P3,P4)
Saisir (X) :	
lexique { local à l'action (vide ici)}	
algorithme	
Ecrire("+");	{affiche le curseur "attente de position"}
Lire(X);	
ConstruirePointSuivant (M,N,P) :	
lexique { local à l'action }	
D1, D2 : deux droites	
C : un cercle	
Q : un point	
algorithme	
ConstruireDroite (M, N, D1);	
ConstruireDroitePerpendiculaire (D1, D2);	
ConstruireCercle (M, N, C);	
ConstruirePointsIntersectionDroiteCercle(D2, C, P, Q);	

Ainsi, les textes des algorithmes sont des emboîtements de textes, reflétant la structure d'analyse.

Le diagramme suivant visualise les relations de dépendance entre les diverses fonctions décrites et les types construits dans l'exemple du carré. La définition récursive du type polygone est mise en évidence par la couleur bleue de la flèche qui boucle autour du constructeur récursif de ce type : ajoutP. Une couche supplémentaire est ajoutée pour introduire les points de base de la construction. Elle simule la couche de communication entre l'utilisateur du logiciel et l'ordinateur, c'est-à-dire le support de l'interface.

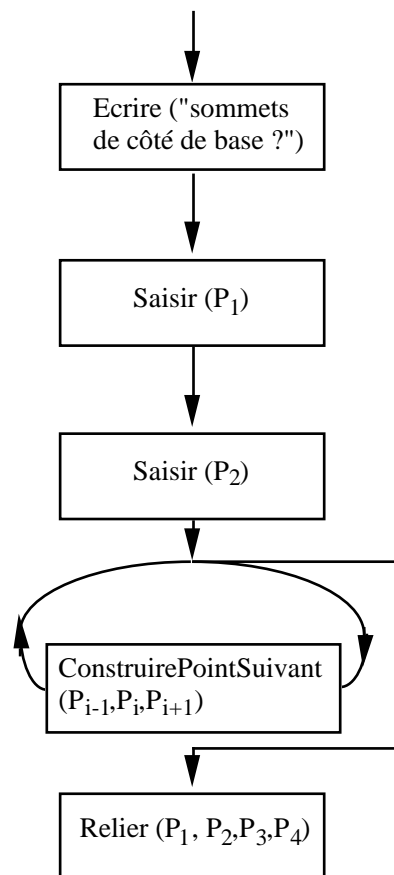


Les divers chemins d'exécution d'un algorithme décrit sous une forme actionnelle, peuvent être visualisés à l'aide d'un graphe. Le schéma obtenu est appelé "schéma d'exécution". Son niveau d'abstraction correspond au niveau de décomposition des actions.

Comme dans les organigrammes, les flèches expriment l'ordre d'exécution. La composition conditionnelle est visualisée par une répartition des flèches selon les différents cas ("conditions d'entrée") des actions plus élémentaires qu'elles représentent.

La composition itérative est représentée par deux flèches en boucle tournant autour d'une action et une flèche pour marquer (et brancher vers) la sortie. Les points possibles d'observation de l'état du système au niveau d'abstraction du schéma peuvent y être représentés.

L'illustration ci-dessous représente le schéma d'exécution d'un algorithme de construction du carré.



3°) Langages de 4^{ième} génération

Les langages L4G sont des langages applicatifs, c'est-à-dire dédiés à un but précis. Par exemple, ce sont

"des langages évolués qui prennent en compte les traitements des bases de données. Ils intègrent dans des fonctions simples le traitement et l'accès à des bases de données." [Augier 97] (p. 83).

Comme pour les expressions fonctionnelles et actionnelles, les "schémas des relations" suivent un formalisme textuel. Comme dans le formalisme fonctionnel, les relations sont d'abord spécifiées, puis associées.

Dans le cas relationnel, les relations ne sont pas associées à une réalisation mais à un prédicat. Ce prédicat est écrit à l'aide d'opérations ensemblistes et de quantificateurs. La notation relationnelle permet de décrire les accès aux données (projections sur des attributs, sélections) ou aux relations (opérations sur les ensembles).

La problématique de l'informatique aborde la formulation de problèmes types, les méthodes d'analyse, les situations où intervient l'informatique. La résolution progressive et raisonnée des problèmes s'appuie sur l'identification des niveaux d'abstraction successifs et l'utilisation de langages appropriés. Les démarches d'analyse et de raisonnement contiennent spécification, décomposition, analyse par cas, récurrence, organisation modulaire, etc.

Le principe d'analyse descendante est un principe de décomposition du problème considéré. Le problème spécifié au départ est décomposé en problèmes intermédiaires indépendants qui à leur tour sont spécifiés. La solution du problème initial est exprimée en supposant les problèmes intermédiaires résolus. Ce processus est réitéré pour exprimer les solutions de ces sous-problèmes.

L'analyse par cas est une forme particulière de décomposition d'un problème. Le moyen utilisé est la recherche et le choix d'une partition du domaine dans lequel le problème doit être résolu, énumérant ainsi un ensemble des cas : un cas est la restriction du problème à un sous-domaine. Il apparaît une nécessité de recouvrement de tout le domaine, mais aussi un risque d'incohérence si l'intersection de deux sous-domaines n'est pas vide.

L'analyse récurrente conduit à se placer dans une optique constructive. En effet elle impose de chercher à résoudre un problème portant sur une donnée d'une certaine dimension par la résolution du même problème sur des données de dimensions inférieures. La reconstitution de la solution finale est une construction basée sur les solutions du "niveau" inférieur, selon le processus déterminé. Par exemple, le calcul de $n!$ est ramené à celui de $(n-1)!$. Plus précisément, une analyse récurrente comporte une analyse par cas mettant en évidence des cas de base et des cas généraux, pour lesquels on peut fournir des relations de récurrence.

4°) Arrivée du génie logiciel et langages 5^{ème} génération

Les logiciels se composent en des applications de plus en plus importantes. Les temps de conception et développement des logiciels professionnels actuellement sur le marché se comptent en "homme-années". La participation de différents participants conduit à la tâche de synchronisation des personnes ainsi qu'à la standardisation des développements.

L'organisation modulaire et la réintégration d'éléments logiciels conduisent à la considération de l'architecture du logiciel. Cette nouvelle tâche consiste en l'étude et la spécification des articulations entre les divers éléments.

"Ce qui n'était au début qu'une série de techniques plus ou moins empiriques a été maintenant quantifié, regroupé et a permis de passer de la programmation au génie logiciel."
[Augier 97] (p.97)

a. Programmation par objets

Le souci de réutilisation de programmes déjà écrits, la vue à plus long terme de l'évolution des activités de conception de logiciels ont conduit à l'approche "objet". Les langages à objets permettent de définir des traitements pour des classes entières d'objets et d'en spécialiser certains aux besoins plus spécifiques de sous-classes.

Conceptuellement, ces langages ont un comportement polymorphe, c'est-à-dire qu'ils sont capables d'attribuer des comportements (opérations) similaires à des ensembles d'objets distincts (polymorphisme). Les objets appartenant à un ensemble peuvent être utilisés comme s'ils appartenaient à un sous-ensemble de celui-ci (polymorphisme par inclusion) : l'opération ne possède alors qu'une seule réalisation (segment de code commun) à laquelle les objets se réfèrent.

D'autre part des ensembles d'objets non liés par une relation d'inclusion peuvent réaliser des opérations conceptuellement identiques (polymorphisme opérationnel), chacun d'eux déterminant une relation spécifique (recouvrement) ou bien une réalisation standard leur étant uniformément appliquée (généricité). Ils demandent au programmeur de réfléchir à la décomposition des tâches qu'il veut commander à l'ordinateur en termes de regroupement d'objets en classes selon la similarité des traitements qu'ils doivent subir, avec une décomposition de plus en plus fine selon le degré de spécialisation des objets.

Le choix du terme objet marque le décalage du centre de préoccupation (point d'attention) : il ne représente plus une simple donnée, valeur d'une variable d'un type composite particulier, dont le caractère "obligé" est lié au transfert entre les procédures, mais il commande aux traitements qu'il peut subir. Les données manipulées ne sont plus "prétexte" des manipulations, mais ce sont les manipulations qui deviennent un prétexte à la "vie" des objets.

Les méthodes de programmation évoluent vers une représentation autonome des objets dans des contextes de situations, le programme n'est plus ressenti ni visualisé dans son ensemble comme une séquence. Le programmeur ne peut plus vérifier l'état de la machine à chaque instant de la séquence des calculs qu'elle effectue. Il doit faire "confiance" aux modules qu'il n'a pas écrits, n'ayant le plus souvent même pas accès au code.

b. Langages pour l'IA

D'autre part, des langages spécifiques ont été définis pour les besoins de l'Intelligence Artificielle. Dans ce cadre, ce ne sont pas des valeurs que les variables doivent manipuler, mais des symboles. Ce niveau d'abstraction supplémentaire est lié au caractère formel des données traitées.

Parmi ces langages, on peut citer LISP (de Mac Carthy en 1960) et PROLOG (en 1972 à Marseille). LISP est dit "fonctionnel" ou encore applicatif. Par exemple utilisé en algèbre, il permet d'effectuer des calculs dans le corps des polynômes, comme de programmer des divisions de polynômes, de calculer formellement leurs dérivées sans calculer leurs valeurs pour des valeurs des variables etc. PROLOG permet de faire des calculs sur des prédicats et de décrire un problème par un ensemble de règles. La machine parcourt l'ensemble des règles, effectue les substitutions de termes spécifiées et utilise un mécanisme d'unification accompagné du « backtracking », pour résoudre le système. Elle explore entièrement le graphe des solutions, sauf si l'utilisateur a explicitement demandé un parcours incomplet. le « backtracking » donne la capacité d'itération à un langage déclaratif.

Ces deux langages (LISP et PROLOG) peuvent être interprétés, c'est-à-dire qu'ils ne nécessitent pas l'usage d'un compilateur mais d'un interpréteur. Le programme source est introduit directement dans l'interpréteur qui considère une à une chaque nouvelle ligne, appelée "commande". Chaque ligne à son tour est analysée, traduite et son exécution répercutée sur les données manipulées. Dans le cas présent, ces données sont des symboles. En cas d'erreur, on obtient exactement la ligne sur laquelle s'est produite l'erreur : la dernière exécutée. Il existe aussi des compilateurs pour LISP et PROLOG. Les qualités d'efficacité et de compacité de LISP en font un langage très utilisé aujourd'hui, en particulier pour toutes les applications dans lesquelles les constructions et manipulations de types jouent un rôle important.

c. PROLOG et la géométrie dynamique

Le terme de 5^{ième} génération est particulièrement approprié, puisque PROLOG a justement été le langage choisi par les japonais pour leur fameux « 5th generation project » dans les années 1985-1990.

PROLOG est utilisé pour résoudre des problèmes de construction "à la règle et au compas" de figures géométriques, et couplé avec Cabri, pour visualiser les solutions. Une figure dans Cabri est une solution élaborée par un utilisateur pour un problème particulier de géométrie. Un problème de construction géométrique est considéré comme une recherche de solution pour la vérification conjuguée d'un ensemble de propriétés géométriques.

Ce type de problème n'admet pas toujours de solution. Par exemple, tous les nombres réels qui peuvent s'écrire comme des combinaisons linéaires de racines carrées sont constructibles en ne faisant usage que d'une règle et d'un compas, mais pas le nombre Π .

Lorsque l'utilisateur a élaboré une solution, cette solution est le reflet de la stratégie suivie pour effectuer ses différents choix de construction. Les relations qui relient les éléments géométriques de la figure suivent un ordre fixe, et l'introduction de chaque nouvel élément est spécifiée par le respect d'une contrainte. Cette contrainte est une propriété particulière qui relie ce nouvel élément à d'autres éléments précédents dans cet ordre.

Pour un ensemble de propriétés géométriques à vérifier conjointement, il peut y avoir plusieurs solutions. Par exemple, la construction d'un carré est généralement basée sur un segment qui constitue un de ses côtés. Les quatre côtés du carré jouent exactement le même rôle dans la définition d'un carré, mais pas dans la construction. Il y a donc au moins les quatre solutions correspondant à chacun des choix d'un côté, sans parler des solutions qui reposent sur une stratégie toute différente comme celles qui seraient basées sur une diagonale.

Les propriétés demandées pour la construction sont les axiomes (règles ?) de PROLOG, les éléments géométriques sont les termes du domaine.

Cette méthode a été utilisée pour orienter des élèves dans leur recherche de solution d'un problème de construction par un système tuteur [Desmoulins 95, Desmoulins 94].

L'illustration suivante montre l'exemple de la déclaration de la figure dynamique de géométrie qui représente un carré, selon le formalisme de GéoSpécif. L'approche choisie est décrite comme une approche prenant en compte la spécification logique de la figure de manière non déterministe (au sens de PROLOG) et incrémentale. [Bouhineau 95, Bouhineau 97] (p.184)

"L'algorithme utilisé opère en deux temps. Tout d'abord, il complète automatiquement la spécification avec des contraintes géométriques vérifiées par la figure. Ensuite, il effectue une traduction algébrique du problème et tente de résoudre le système d'équations algébriques obtenu."

La spécification d'une figure sous une approche déclarative est structurée en deux parties : la première "crée" les éléments de la figure, la deuxième déclare les contraintes entre ces éléments. $|AB|$ représente la longueur du segment $[AB]$, et $(A, B) \perp (B, C)$ déclare que les droites (A, B) et (B, C) sont perpendiculaires.

point (A), point (B), point (C), point (D), polygone (A, B, C, D),
(A, B) \perp (B, C) AB = BC (B, C) \perp (C, D) BC = CD

d. Les langages de commande et LOGO

Un autre mode de transmission de tâches à un ordinateur se fait par l'intermédiaire de langages de commande. Ce sont des langages interprétés qui permettent de manipuler les données par l'intermédiaire de commandes (d'actions).

Parmi ces langages, on trouve des langages de manipulation (gestion) des systèmes, permettant d'espionner et de commander des applications depuis le système de l'ordinateur (les langages de scripts comme les shells d'Unix, AppleScript).

Certaines applications sont basées entièrement sur de tels langages : elles sont conçues pour appliquer les instructions communiquées par l'utilisateur à des données, et leur soumettre un comportement.

Par exemple, le langage LOGO ([Papert 80, Avram 84]) a été défini pour l'enseignement de la programmation. Il permet d'aborder les notions de récursivité, de procédure et de liste. Une ligne LOGO est une suite d'instructions. Ces instructions peuvent être des primitives du langage ou encore être définies par l'utilisateur.

Le logiciel signifie qu'il est prêt à recevoir une nouvelle commande par l'affichage d'un symbole d'invite (*prompt*). Chaque ligne commence par une commande. Ensuite suivent les données attendues pour cette commande, et l'utilisateur conclue en signifiant au logiciel que la commande est complète : il appuie sur la touche de fin de ligne (Entrée). L'environnement LOGO permet de programmer les dessins laissés par la trace des mouvements d'une tortue.

"L'algorithmique [apparaît] comme l'art de s'imposer une attitude méthodique dans la construction de programmes, en se plaçant au bon niveau d'abstraction, en choisissant les moyens d'expression adéquats et en réutilisant des algorithmes fondamentaux." [Scholl 93].

Le seul moyen de communication avec l'ordinateur, au départ ou du moins après les cartes perforées, est textuel. Avec l'arrivée des interfaces graphiques, ces moyens sont augmentés des manipulations d'icônes et des représentants graphiques des données dans les supports graphiques, par le biais du guidage des déplacements d'un curseur à l'aide d'une souris [Myers 96]. De nouveaux utilisateurs peuvent de plus en plus tirer profit de l'informatique et de la programmation, alors que le coût des services d'un professionnel rendait inabordable l'investissement requis.

5°) Apparition des IHM

[Augier 97] (p. 95) "Les interfaces graphiques ne sont pas véritablement un outil de conception mais plutôt un environnement de travail. Les IG ont été développées pour améliorer l'ergonomie des machines. Cela d'abord sur les micro-ordinateurs (à la suite des travaux des laboratoires de Xerox une IG a été pour la première fois mise en place sur l'Apple LISA, puis sur le Macintosh qui est à l'origine de la large diffusion des IG que l'on connaît aujourd'hui)."

Répandues sur tous les supports informatiques aujourd'hui, elles permettent à l'utilisateur novice comme à l'utilisateur expert en informatique, d'effectuer des tâches au travers de l'ordinateur.

Cabri-géomètre est un logiciel dont l'IHM est tout aussi importante que le contenu informationnel et algorithmique.

Trois aspects de l'apparition des interfaces graphiques sont fondamentaux.

a. Leur raison d'être

Le premier aspect concerne leur conception avec leur raison d'être. *"Leur cible était d'abord les utilisateurs finals des ordinateurs, c'est-à-dire les personnes qui voulaient profiter de la puissance de ces machines sans pour autant être docteur en informatique. Un architecte par exemple qui va pouvoir dessiner plus vite, calculer plus vite ses plans en utilisant un ordinateur n'y verrait aucun intérêt s'il devait passer le temps ainsi gagné à faire fonctionner sa machine."* [Augier 97]

Les interfaces graphiques permettent aux utilisateurs des ordinateurs (micro-ordinateurs seulement, au départ, et maintenant aussi les stations de travail) d'avoir accès aux ressources de l'ordinateur, c'est-à-dire aux données qu'il mémorise, mais aussi aux fonctionnalités que le logiciel d'interface propose. Ces accès sont graphiques, basés sur l'exploitation d'icônes, de fenêtres et de menus déroulants. Ainsi, les interfaces graphiques présentent de manière synthétique et graphique les travaux que l'utilisateur accomplit et peut accomplir : il spécifie ses tâches en s'appuyant sur les retours d'informations affichées par le logiciel d'interface à ses manipulations de la souris, du clavier, et éventuellement d'autres périphériques d'entrée (voix, écran tactile, crayon...).

b. Leur manière d'être

Le second aspect concerne toujours la conception des interfaces graphiques, mais du point de vue de leur manière d'être. Les logiciels devant constituer des environnements graphiques sont basés sur une boucle infinie d'attente d'événements. Ces événements proviennent des périphériques d'entrées de l'ordinateur (souris, clavier, micro, connexion, ...). Des traitements spécifiques sont associés à chaque type d'événement. Les événements sont différenciés par leur type. Ces types sont liés au support de provenance de l'événement. Ainsi parmi ces événements se trouvent les pressions sur des touches du clavier, sur un bouton de la souris, leur relâchement, les reprises de "main" du système pour lui-même ou d'autres applications, les demandes extérieures de remise à jour de l'affichage graphique, etc.

Schématiquement

La boucle d'attente d'événements constitue le cœur du programme principal. Dans une organisation modulaire, le programme principal est contenu dans le module contrôleur de l'ensemble de la structure du logiciel, celui qui centralise les différents traitements. Il contient le démarrage du logiciel ainsi que les traitements à effectuer pour terminer l'exécution du programme, comme de libérer l'ordinateur de toute la place mémoire utilisée pendant la session d'utilisation du logiciel. La forme du curseur est adaptée à la situation courante depuis cette boucle. Ce module constitue le noyau fonctionnel de l'application.

Conception

La spécification des interfaces Homme-Machine constitue une étape cruciale et délicate pour laquelle différentes approches sont envisagées. Une interface homme-machine fait figurer des outils sous forme iconique et des représentations graphiques pour les données. Au cours des manipulations de l'interface, les différentes représentations graphiques doivent évoluer.

La conception de l'interface consiste d'une part à spécifier et choisir les représentations externes (graphiques), et d'autre part à définir leur évolution (dynamique de l'interface).

Les recherches en IHM ont permis de déterminer des méthodes de conception. Une des approches consiste à présenter, par niveau d'abstractions, les représentations externes utilisées et la cohérence interne ou abstraite de ce qu'il représente.

[Coutaz 90] définit le modèle PAC qui a pour but d'obliger le concepteur à séparer en trois composantes sa spécification, chacune représentant respectivement le niveau de présentation, d'abstraction ou de contrôle. Il décrit les mécanismes internes du fonctionnement de l'interface en termes d'interactions entre agents (modules) logiciels (de présentation, d'abstraction ou de contrôle).

[Ziegler & al. 88] sépare les niveaux conceptuel, de dialogue et celui qui se rapporte aux entrées/sorties.

[Bernat 94] préconise le niveau du domaine, le niveau conceptuel, le niveau de présentation et le niveau visuel.

Exemple niveau conceptuel

[Norman 90] et [Gritzman & al 95] mettent en avant l'importance d'orienter l'interface vers les tâches que l'utilisateur aura à accomplir. Leur approche constitue un point de vue conceptuel pour les IHM, c'est-à-dire qu'elle permet de cadrer les réflexions des concepteurs pour la définition d'une IHM. D'après [Zacklad 97],

"Dans la plupart des réflexions méthodologiques, une distinction assez nette est établie entre des cadres de modélisation conceptuels et de conception même si la terminologie est assez fluctuante. Chez Coutaz et Ménadier, on présente d'une part des modèles "issus des sciences cognitives" ou de "spécification" (conceptuelle, sémantique, syntaxique et lexicale[par exemple]) et d'autre part des modèles d'"architecture logicielle" souvent liés à des générateurs d'IHM."

En modélisation conceptuelle, l'utilisation de différents points de vue peut donner accès à toutes les facettes d'une IHM, sans rien oublier, mais en minimisant la redondance.

Des travaux comme [Zacklad 97] s'intéressent à cette recherche de dimensions pour la modélisation d'artefacts (une IHM étant un cas particulier d'artefact, et aussi une composition structurée d'un ensemble d'artefacts), et préconise cinq dimensions.

Deux dimensions hiérarchiques structurent la modélisation depuis les deux extrémités opposées : Partie-tout (chaque partie étant considérée comme un tout à décomposer) et Généralisation (les parties étant regroupées par caractère généralisable).

Les autres dimensions sont dites "optiques" pour marquer leur contribution transversale. La dimension Abstraction contribue à une progression en boucle entre l'abstraction inhérente à la description de buts et la concrétisation traduite par une approche de leur mise en œuvre. La dimension Cybernétique oblige à définir ce que le système est (structure), ce qu'il fait (fonction) et ce qu'il devient (comportement). Enfin, la dimension Interaction s'appuie sur une modélisation "agent", issue des travaux des sciences cognitives et d'intelligence artificielle.

Le modélisateur "considère le système comme un agent doté d'une certaine autonomie et agissant de façon réflexe, agent réactif, ou en fonction de représentation "mentales", agent cognitif." Les différents agents concernés par la résolution d'un problème particulier d'interaction sont considérés comme les constituants un groupe. Trois types d'interaction entre l'agent et le "monde" (les autres agents sont considérés comme appartenant à un groupe) sont différenciés : interactions objectales (relations directes), interactions instrumentales (relations à travers un artefact) et interactions sociales (relations entre agents).

Pour analyser les interactions instrumentales, le modèle des Situations d'Activité Instrumentées [Rabardel 95] étudie les contributions des différents points de vue associés aux activités entre les trois intervenants, à savoir le sujet, l'instrument et l'objet.

Exemple dimension Interaction

Une méthode de conception des interfaces homme-machine est la conception d'un modèle de tâches comme MAD [Scapin 89], DIANE+ [Tarby & Barthet 96], UAN et GOMS. Un tel modèle est utilisé pour une modélisation d'IHM basée sur l'analyse des interactions de type instrumental.

D'après [Zacklad 97], MAD est bien adapté pour décrire le point de vue de l'activité du sujet sur l'objet, et GOMS le point de vue du sujet sur l'instrument. Le mot "modèle" sous-entend une formalisation, et "tâche" est associé à un travail déterminé à exécuter. MAD et GOMS permettent de représenter de façon rigoureuse une décomposition des tâches en sous-tâches de granularité plus fine et l'ordonnement de ces sous-tâches entre elles. Un modèle de tâche constitue un formalisme, rigoureux et complet pour spécifier toutes les fonctionnalités d'une IHM.

Ces méthodes obligent le concepteur à se plier au caractère rigoureux dû à l'introduction d'une structure rigide et à l'élaboration d'un modèle et ainsi à se rapprocher des exigences de l'informatique. Elles constituent un support de communication pour la mise au point entre les demandes des ergonomes et les contraintes des développeurs [Balbo & al 98].

Si le concepteur est tenté de rapprocher le développement de l'interface d'un processus automatisable basé sur l'utilisation d'un modèle de tâche, il ne peut spécifier par l'analyse de la tâche que des contraintes s'y rapportant. Il en découle que le contenu attendu dans le modèle des tâches semble plus important que celui qui est porté par la structure seule. Ce manque est compensé par une place laissée dans le modèle pour une description moins formelle des tâches, exprimée en langage courant (annotations textuelles). Cependant, des critères de choix peuvent être extraits des modèles, en appliquant des règles négatives : si le modèle de tâche présente telle et telle spécificité, alors tel type de concept sera proscrit dans l'IHM.

Deux paradigmes opposés permettent de structurer l'interaction. Ils concernent l'ordre de sélection des éléments graphiques de l'interface : soit le choix préalable de l'outil restreint les objets auxquels il peut être appliqué, soit la sélection préalable des objets limite les outils qui peuvent leur être appliqués. On parle de "mode opératoire" verbe/nom ou nom/verbe [Carbonneaux 98].

Ce choix a une influence importante sur l'effort d'adaptation nécessaire à un utilisateur pour maîtriser le fonctionnement de l'interface et aboutir aisément à ses objectifs.

"Le contrôle de l'action pour la gestion ou la prévention des erreurs est identique dans les deux modes. Une différence notable est dans le type d'aide que le type de mode opératoire fournit. Elle est contextuelle pour le mode verbe/nom (par exemple, une liste d'objets peut être affichée dans une barre d'état, pour avertir l'utilisateur des objets à sélectionner [capable d'être utiles pour l'application de] la commande), tandis que l'aide est une aide en ligne pour le mode nom/verbe. Le mode verbe/nom favorise donc une exploration plus libre du logiciel que le mode nom/verbe, mais il se peut qu'il ne réponde pas entièrement au principe de cohérence."

Toute l'interface peut ne pas suivre le même paradigme.

Implémentation

Dans la phase d'implémentation des interfaces homme-machine, les développeurs utilisent des bibliothèques de procédures (boîte à outils, Toolkit en anglais). L'article [Myers 95] est très utile pour bien comprendre ce qui caractérise les "outils" utilisés par les développeurs :

"This paper will try to define these terms more specifically, and use the general term "user interface tool" for all software aimed to help create user interfaces. Note that the word "tool" is being used to include what are called "toolkits", as well as higher-level tools, such as interface Builders, that are not Toolkits".

"Les services offerts par les outils sont calqués sur un modèle d'organisation de l'interface, plus ou moins évolué selon les outils. Réciproquement, un modèle d'architecture prend toute sa force lorsqu'il est mis en œuvre par un outil de construction qui en prouve l'utilité. [...]"

On trouve plusieurs catégories d'outils : boîtes à outils, squelettes d'application, langages de description et éditeurs graphiques. Les deux premiers sont utilisables depuis un langage de programmation traditionnel, les deux derniers permettant de construire une interface avec peu ou pas de programmation.

Les boîtes à outils les plus connues sont la X Toolkit et Motif qui en dérive, XView, et InterViews. Le squelette d'application le plus célèbre est MacApp. Parmi les langages de description, on peut citer UIL et Tk. Les éditeurs d'interfaces les plus répandus sont sans doute Visual Basic, Interface Architect, Interface Builder et en France Aïda/Masai et XFaceMaker." [Chatty 93]

Les boîtes à outils sont programmées en langages à objets. Les fonctionnalités d'interaction sont supportées au travers d'objets interactifs (widgets en anglais). Chacun de ces objets est spécifié par son aspect visuel (présentation), son comportement en réaction aux actions de l'utilisateur (interface d'entrée), son lien avec le reste de l'application (interface de sortie).

Mise au point

Une conséquence de l'usage des boîtes à outils sur l'activité de programmation est que le développeur n'est plus maître de tout le code du logiciel qu'il produit. Il doit s'imprégner des principes sous-jacents aux procédures auxquelles il peut faire appel d'une part, et d'autre part se placer, pour sa conception, au niveau d'abstraction adapté à ses besoins.

Par exemple pour définir le contrôle, les boîtes à outils reposent sur l'utilisation d'un des mécanismes suivants, allant du plus procédural au plus déclaratif : séquence, programmation impérative ou fonctionnelle (structures de contrôles classiques), automates à états finis, programmation réactive, réseaux de Pétri, programmation concurrente (divers mécanismes de synchronisation), flots de données ou programmation logique.

La variété des choix conduit à une évolution de la tâche du programmeur. L'article [Accot 98] présente quatre interview d'intervenants dans des activités de conceptions d'interfaces homme-machine, et fait état des difficultés et de l'évolution des tâches. En particulier, certaines limites des boîtes à outils actuelles sont discutées.

Le problème de la mise au point des programmes est, pour le développeur, de déterminer les raisons pour lesquelles le logiciel en cours de conception ne se comporte pas exactement comme désiré, et d'y remédier. Une première tâche, particulière dans le cas des organisations modulaires, est la localisation d'une zone de programme qui contribue à ce décalage. [John & al. 92]

"Software development is a pervasive and central task in computer science whose nature is changing rapidly. The traditional practice of coding systems from scratch, of maintaining those systems in term of that code, and of relying on the virtuosity of individual, skilled programmers has not scaled to the needs of large, complex system development. In its place we can expect future software development to center around component-based software construction, reusable software design framework (or architectures) and, distributed, cooperative interaction between developers."

Pour les logiciels constituant des interfaces graphiques, la localisation de la zone erronée du programme n'est pas évidente à cerner, parce que le sentiment de l'entrée du système dans l'état courant et de l'histoire qui l'y a conduit n'est pas accessible séquentiellement. En effet, un moyen utilisé classiquement pour la mise au point de programmes, est de "pister" la séquence constituée par le programme à étudier. Ce pistage est accéléré (aidé, guidé) par la mise en place de points d'arrêt dans une exécution dirigée (bridée) du programme : le programme, au lieu de se dérouler tout seul, c'est-à-dire en autonomie, est contraint par l'intermédiaire d'un débogueur.

Quel est le rôle d'un débogueur et quels sont les besoins particuliers dans le cadre des interfaces graphiques ? Le débogueur permet de dérouler le programme pas à pas où jusqu'à certains "points" d'exécution que le développeur juge pertinents ou cruciaux. Le problème des points d'arrêts mêlés aux interfaces graphiques est que ces points d'arrêt constituent des événements qui modifient donc le cours recherché (pisté) du programme.

Il est difficile de mettre un point d'arrêt dans la boucle d'attente d'événement qui ne s'arrête pas pour les événements "système" demandant une interruption, alors que justement un point d'arrêt du débogueur est reçu par le logiciel comme un événement en provenance du système.

Aussi le développeur éprouve-t-il quelques difficultés à obliger le logiciel à s'arrêter à des endroits particuliers, avec l'assurance qu'il a exécuté un certain nombre de tâches. Quand le programme est arrêté sur un point d'arrêt particulier, en dehors de la boucle d'attente d'événement, on ne peut savoir exactement ce que le programme a exécuté.

Par exemple, on n'a pas accès au nombre d'itérations effectuées par la boucle d'attente d'événements principale, ni à la liste des événements de cette boucle que le logiciel a déjà gérés. Le programme a perdu de son aspect séquentiel. La mise au point des programmes s'en trouve modifiée.

c. Leur influence sur le monde informatique

Le dernier aspect concerne l'influence des interfaces graphiques sur le monde de l'informatique : elles modifient les rôles des utilisateurs des moyens informatiques, faisant même apparaître de nouvelles catégories d'utilisateurs. [Augier 97]

"Les développeurs sont eux aussi une catégorie particulière d'utilisateurs finals. Même si leur tâche consiste à travailler sur un ordinateur, ils ont eux aussi une mission particulière, en fait différente de la gestion au jour le jour de leur machine. Pour cette raison, ils ont voulu profiter de ces nouveaux environnements et vu qu'ils étaient aussi ceux qui les développaient... Un développeur qui utilise une IG est d'une part un utilisateur avec les exigences décrites précédemment, et d'autre part, en tant que développeur véritablement c'est avoir accès à un ensemble de bibliothèques qui vont lui permettre de réaliser simplement les objets graphiques de l'IG, pour pouvoir les mettre en œuvre dans l'application qu'il développe."

L'arrivée des interfaces graphiques fait apparaître une nouvelle catégorie : les utilisateurs non informaticiens. Elles leur rendent abordables les activités de programmation, implicitement ou explicitement.

B) De nouveaux moyens pour aborder la programmation : l'apport des interfaces graphiques

L'utilisation des langages de programmation classiques est réservée à des spécialistes informaticiens. Avec l'introduction des interfaces graphiques et l'amélioration des interfaces Homme-Machine, l'univers informatique est devenu plus abordable par le grand public.

1°) Du concept de manipulation directe aux interfaces démonstratives

Le concept de manipulation directe permet à tout utilisateur de manipuler directement les données avec lesquels il travaille.

[Baulac 84] (p.100) *"La manipulation directe des objets de l'application est le fait de pouvoir appeler directement un objet sans le recours à la description de cet objet dans un langage quelconque. Les objets de l'application sont soit des objets de contrôle (fenêtres, procédures, boutons, ...), soit des objets propres au domaine de l'application (des objets géométriques dans le cas de Cabri-géomètre)".*

[Nanard 90] (p.42), [Bellemain 92] (p.100-103), [Carbonneaux 98] (p.6)" *Les principes de la manipulation directe ont été jusqu'à présent plutôt mis en œuvre dans la conception d'interfaces graphiques permettant la manipulation, souvent nécessaire, d'objets informatiques à l'intérieur d'environnements informatiques. Ces objets ne sont pas représentés mais plutôt symbolisés par des images, et la manipulation du symbole représente pour l'utilisateur la manipulation de l'objet lui-même. Ces manipulations sont destinées plutôt à simplifier la mise en œuvre de notions informatiques intervenant dans l'utilisation d'un système qu'à permettre la construction de connaissances relatives à ces notions. Dans le cas d'environnements destinés à l'apprentissage, la manipulation directe s'applique à des représentations d'objets mathématiques et non pas aux objets eux-mêmes.* "

Concrètement, une interface est dite de manipulation directe si elle possède les propriétés suivantes :

1. *"Les objets concernés ont une représentation permanente,*
2. *Des actions physiques directes ou l'appui sur des boutons étiquetés sont utilisés au lieu de commandes textuelles de syntaxe compliquée,*
3. *Le résultat des actions sur les objets est immédiatement visible,*
4. *Les opérations sont rapides, incrémentales et réversibles."* [Shneiderman 82] (p. 239)
5. *"Les transformations sont contrôlables en temps réel".* [Laborde 95] (p. 36)
6. *Le déplacement des objets concernés doit être continu.*
7. *Tous les objets représentés doivent être manipulables.*
8. *La création des objets s'effectue naturellement et directement.*
9. *Aucun ordre sur les éléments des procédures."* [Carbonneaux 98] (p.23)

Avec des interfaces respectant ce mode de manipulation, l'utilisateur est plus à même de définir les comportements attendus de ces données, ou plutôt leur réactions en fonction de ses manipulations.

L'idée est apparue de donner à tous les utilisateurs, quels que soient leurs profils, les moyens de définir eux-mêmes les tâches qu'ils désirent ordonner aux ordinateurs, en leur donnant les moyens de communiquer directement avec l'ordinateur. [Myers 90] définit les interfaces démonstrationnelles comme des interfaces capables de générer de nouveaux outils utilisables dans l'interface elle-même et définis par démonstration, où le terme "démonstration" ayant ici le sens de « présentation ».

2°) Une diversification des intervenants et une évolution des tâches

Auparavant, dans toutes les activités informatiques, l'utilisateur était différencié du programmeur. L'évolution de l'usage de l'informatique conduit à modifier cette différenciation des types d'utilisateur de l'ordinateur [Myers 95].

Considérant les moyens informatiques non plus comme un magasin (un serveur) d'applications mais comme un environnement permettant d'en fabriquer, le programmeur devient tout autant client de services informatiques que l'utilisateur final, celui pour lequel sont prévues ces applications, et qui est éventuellement non informaticien.

De même que le programmeur devient client des logiciels qui constituent les environnements de programmation, l'utilisateur final commence à être à même d'utiliser des outils de fabrication ou d'ajustement de ces environnements. Apparaît une évolution du statut de l'utilisateur, mais tout aussi bien du programmeur, et même une nouvelle catégorie de protagonistes : ceux dont le travail est la mise au point et la conception d'activités propres au domaine d'application du logiciel.

Ainsi, autour d'un environnement donné (en plus de l'ergonome) "collaborent" trois types de personnes : celles qui fabriquent l'environnement, celles qui utilisent l'environnement pour mettre au point des activités du domaine (les experts du domaine) et les utilisateurs finals.

Considérons un tableur comme Excel. On différencie les concepteurs du logiciel lui-même, des concepteurs de fiches, des utilisateurs des fiches conçues.

Les besoins de l'utilisateur final sont des besoins de "client" par rapport aux fiches : il ne peut qu'y entrer des données. L'utilisateur concepteur de fiches est par exemple un expert en comptabilité. Il prépare des tableaux dans lesquels seules certaines cellules sont modifiables. Les autres cellules s'ajustent automatiquement par les calculs qu'il a prévus, ou restent identiques. Les concepteurs du logiciel sont ceux qui ont prévu le fonctionnement global du logiciel. Ils ont su trouver les solutions permettant à l'ordinateur de supporter les fonctionnalités intéressantes pour la fabrication de telles fiches, différencier les niveaux de fonctionnements adaptés aux différents types d'utilisateurs aval du logiciel.

La chaîne de vie du logiciel se trouve modifiée : au lieu de boucler entre le programmeur et l'utilisateur, le logiciel est ajusté entre le programmeur et l'expert du domaine. C'est l'expert qui est à même de sonder l'adaptation du logiciel et de ces "fiches" aux attentes de l'utilisateur final.

Dans les EIAH (Environnements Informatisés pour les Apprentissages Humains), le même type de différenciation apparaît : les concepteurs de l'environnement (programmeurs et ergonomes), les concepteurs des activités (professeurs) et les utilisateurs (exécuteurs de ces activités : les élèves) interviennent dans le cycle de vie du logiciel avec des fonctions distinctes.

Dans les environnements graphiques de programmation, les concepteurs de l'environnement sont des programmeurs aidés d'ergonomes, les concepteurs des applications sont des utilisateurs de l'environnement de programmation, éventuellement non informaticiens, et les utilisateurs des applications sont a priori non informaticiens. Ces derniers sont appelés les utilisateurs finals (end users en anglais).

La différence principale avec les environnements de programmation classique comme CodeWarrior, est la qualité (profil) des utilisateurs supposée par l'environnement : dans le cas classique, ce sont des informaticiens, alors que dans le cas graphique, tous les niveaux de familiarité sont attendus.

L'informaticien voit une évolution de sa tâche : ce n'est plus l'utilisateur des ressources informatiques qui doit s'adapter au seul moyen de communication possible avec l'ordinateur, mais c'est ce moyen de communication qui doit être adapté à tous les types d'utilisateurs et qui doit être capable de s'adapter aux besoins de l'utilisateur. La tâche du programmeur, souvent aidé d'un ergonome, est alors de mettre en place les environnements. Elle n'est cependant pas déconnectée du domaine d'application, puisque les choix d'environnement doivent être liés aux besoins des activités, et des conceptions de ces activités.

Les nouvelles tâches du programmeur prennent essentiellement deux directions :

- Définir la structure de l'interface et son comportement,
- Construire des outils pour la réalisation d'interfaces (cf. paragraphe précédent).

Pour ouvrir la possibilité à des non-informaticiens de fabriquer leurs propres applications, les environnements doivent remédier aux principales difficultés inhérentes à la programmation textuelle par langage informatique pour les non informaticiens.

"Spreadsheets and visual programming languages raise a challenge for existing schema-based models of programming knowledge, which have been scarcely been applied outside Pascal-like languages." [Green 95].

Ces difficultés proviennent essentiellement de trois sources :

- la nécessité de raisonner de façon abstraite pour programmer des abstractions d'actions (noms de fonctions ou d'actions) agissant sur des abstractions de valeurs (variables),

- l'obligation de suivre une structure rigide qui contraint l'expression textuelle naturelle selon le seul carcan formel compréhensible par l'ordinateur,
- l'obligation d'utiliser la forme langagière comme média de communication alors qu'elle ne constitue pas nécessairement toujours ni pour tous le support adéquat des mécanismes de raisonnement humain engagés dans la tâche.

Le premier point concerne les difficultés liées à la programmation en aveugle, c'est-à-dire sans aucun moyen pour identifier les actions avec leurs effets sur les variables et ces dernières avec les valeurs en jeu. Il est atténué par l'utilisation de la manipulation directe.

Le second point ne remet pas en cause la forme textuelle du support. Il est lié à l'utilisation d'un langage formel et non pas naturel.

Par contre, le dernier point met en doute le choix de la forme textuelle du support si la volonté d'aider l'utilisateur à formaliser ses raisonnements est pris en considération.

3°) La place des diagrammes comme aide au raisonnement

L'intérêt de l'utilisation de diagrammes pour le raisonnement humain est un sujet récurrent de l'histoire des sciences. Avec l'informatique et les interfaces graphiques, il réapparaît tant pour les domaines d'application comme les mathématiques, que pour l'activité informatique elle-même.

Dans les domaines d'application comme les mathématiques, la question du support du raisonnement humain est abordée régulièrement. Par exemple [Barwise & Etchemendy 91] étudient l'intérêt de l'utilisation de diagrammes pour des activités de démonstration en mathématique, et plus particulièrement en géométrie, théorie des ensembles et logique. Ils argumentent sur la légitimité d'inférences hétérogènes (visuelles et textuelles) en prenant comme arguments l'universalité de la représentation linguistique et les dangers de la représentation visuelle :

"despite the obvious importance of visual images in human cognitive activities, visual representation remains a second-class citizen in both the theory and practice of mathematics.[...] The main reason for the low repute of diagrams and other forms of visual representation in logic is the awareness of a variety of ill-understood mistakes one can make using them: witness the fallacies that have arisen from the misuse of diagram in geometry. By contrast, it is felt that we have a fairly sophisticated semantic analysis based on reasoning."

Ils établissent ainsi la raison d'être de leur logiciel "Hyperproof" : permettre à des étudiants de résoudre des tâches de raisonnement déductif à l'aide d'une combinaison intégrée d'énoncés et de diagrammes.

Pour l'activité informatique, le sujet réapparaît avec les interfaces graphiques. D'abord dans le cas d'applications graphiques à des domaines particuliers où le raisonnement est important, de nouvelles possibilités sont explorées pour visualiser les phénomènes. Dans le cadre de la géométrie et du logiciel Cabri-géomètre, [Laborde C. 93] fait état du double rôle de la visualisation en géométrie : ce logiciel permet d'explorer des constructions et de valider des conjectures. Une figure peut-être considérée comme une expérience graphique destinée à vérifier visuellement une propriété géométrique ou à supporter le raisonnement. En fait, le retour d'information procuré par le programme n'est pas seulement de nature perceptive, mais aussi de nature conceptuelle : c'est parce que les étudiants ont une idée a priori de quelques invariants de la figure qu'ils seront convaincus de l'inexactitude de leur solution. Dans le cas de Cabri-géomètre, ce n'est pas seulement la composante graphique qui agit pour apporter de l'aide au raisonnement, mais son animation et sa réponse aux manipulations, donc sa composante dynamique.

Ensuite, apparaît la question du support graphique et de ce qu'il apporte à l'activité informatique elle-même : au travers de ces interfaces, les technologies de manipulations maîtrisées aujourd'hui apportent la possibilité de définir graphiquement les programmes soit à l'aide de la souris, soit plus récemment par interprétation de tracés effectués à main "levée".

Les publications regroupées dans [Addis & al. 98], le recueil d'articles qui a servi de base à *"Thinking with diagram"* en 1998, témoignent des différents aspects de ce sujet. Par exemple y sont discutées :

- l'apport des diagrammes aux raisonnements humains liés en particuliers à l'utilisation de machines [Olivier 97],
- l'usage fait des diagrammes selon le niveau d'expertise de l'utilisateur [Scaife & al. 98]
- L'influence des places respectives des diagrammes et des programmes dans la communication avec l'ordinateur : programmer par le biais de diagrammes ou voir des programmes au travers de diagrammes [Blackwell & al. 98],

[Botshernitsan & Downes 97] rappelle ainsi les premières motivations des langages de programmation visuelle :

« From cave printings to hieroglyphics to paintings of Campbell's soup cans, human have long communicated with each other using images. The field of visual programming languages asks: why, then, do we persist in trying to communicate with our computer using textual programming languages? Would we not be more productive and would the power of modern computers not be accessible to a wider range of people if we were able to instruct a computer by simply drawing for it the images we see in our mind's eye when we consider the solutions to particular problems? Obviously, proponents of Visual Programming Languages (VPLs) argue that the answer to both these questions is yes. »

L'apport de la composante visuelle produit principalement deux palliatifs pour atténuer les difficultés inhérentes à l'usage exclusif de langages formels pour toute activité de programmation. La programmation à l'aide d'un langage visuel propose la spécification de la structure logique du programme par l'intermédiaire de graphes, un peu comme les organigrammes. La programmation par démonstration s'appuie sur l'apprentissage automatique des processus utilisés par l'utilisateur lors d'activités de conception sur ordinateur.

C) Environnements pour non-programmeurs

1°) Définition de la programmation visuelle

a. D'où elle vient

"Il s'agit de remplacer la nature textuelle des programmes par l'utilisation à plus ou moins grande de l'image" [Girard 92].

Les premières classifications s'attachent à l'aspect visuel. [Chang 86] distingue les possibilités de représentation des objets manipulés (visualisation ou représentation) et le choix de la forme de représentation pour la structure logique du programme (textuel — linéaire — ou associée à des représentations graphiques).

[Shu 86] introduit la contribution de l'interaction en différenciant les environnements de programmation visuelle des visualiseurs de programmes.

Dans ses divers travaux [Myers 86] [Myers 90] [Myers 96], Myers analyse le domaine de la programmation visuelle et établit une taxonomie basée sur une classification de toutes les approches de la programmation selon trois critères : compilé/interprété, place de la visualisation graphique des programmes ou des données, accès visuel à un exemple d'exécution du programme se déroulant parallèlement à sa construction.

[Girard 92] (p.23-30) présente, en français, la démarche de Myers, expose et illustre les principes de sa démarche.

Myers définit la programmation visuelle comme une programmation qui fait usage au moins de deux dimensions. Il exclut ainsi les systèmes qui ne gèrent qu'une visualisation graphique des programmes ou des données, puisque ceux-ci sont définis classiquement (de façon linéaire comme par langage textuel) ainsi que les langages textuels manipulant des données visuelles. Ainsi, la programmation visuelle contient différentes approches graphiques permettant de spécifier des programmes.

b. Les différentes branches

Soit les données manipulées au travers du système sont les données sur lesquelles le programme agit, soit ces données sont les éléments («tokens») d'un langage de programmation visuel, soit encore, il s'agit des nœuds et des liens d'un graphe qui représente le programme. Ainsi, il y a deux grandes classes de systèmes de programmation visuelle :

- ceux qui sont basés sur une manipulation des données à travers l'interface, avec génération (extraction) automatique et adaptée de programmes ou spécification du programme en relation directe avec les données du programme
- ceux qui sont basés sur la manipulation de la structure logique du programme, représentée par des graphes. Ces graphes représentent le programme à travers une grammaire à formalisme graphique. Les approches diffèrent selon l'information portée par les nœuds et les liens du graphe : le flot de contrôle (représentation des changements d'état de l'ordinateur) ou le flot de données pour une machine virtuelle abstraite (par exemple l'encapsulation peut être représentée par une forme englobante, l'addition par un nœud en forme de triangle,...).

Les langages de programmation visuelle (*Visual Programming Language -VPL*) s'appuient sur les manipulations d'expressions visuelles pour construire un programme. Les expressions visuelles sont transmises à l'ordinateur par l'utilisateur qui "manipule" interactivement des éléments graphiques ou iconiques en fonction d'une grammaire spatiale spécifique. Elles sont associées à une interprétation sémantique.

SketchPad [Sutherland 63] conçu au MIT est réputé avoir été la première application graphique par ordinateur [Botshernitsan & Downes 97]. Le paradigme sous-jacent est la spécification visuelle de contraintes et la programmation orientée par les contraintes. Il s'agit d'une spécification interactive du graphe de contrôle du programme, puisque les nouveaux objets sont "calculables" dès que ceux dont ils dépendent ont été calculés.

Le langage LOGO [Papert 80], dans le domaine de la géométrie est considéré comme une des premières contributions à la programmation visuelle. Un vocabulaire de simples commandes alphanumériques permet à l'utilisateur de commander les déplacements d'une tortue graphique. L'environnement de la tortue constitue un micromonde d'apprentissage. Il conduit à un domaine important d'applications, à savoir les approches ludiques de la programmation. L'objectif de ces environnements est de permettre à des enfants et à des non-programmeurs de définir interactivement des jeux ou des simulations. Le langage LOGO oblige le programmeur à s'imaginer à la place de la tortue pour spécifier ses changements relatifs de direction, de position et d'état.

D'autres environnements sont basés sur un paradigme de programmation à base d'icônes, comme Pygmalion [Smith 75], KidSmith [Cypher & Smith 95], Cocoa et StageCast.

L'approche consiste en une spécification de règles de changement d'état d'objets iconiques évoluant dans une scène. Ces règles sont mémorisées dans une liste et peuvent être regroupées et organisées pour former un programme structuré. Elles sont définies par manipulation directe : pour définir une règle décrivant le comportement d'un objet, l'utilisateur spécifie l'état initial par la sélection du contexte de l'objet (un voisinage immédiat) et l'état final par l'évolution de l'objet dans ce contexte. À chaque instant, l'environnement essaye d'exécuter une règle possible. Pour cela, il parcourt le programme défini par l'utilisateur programmeur, c'est-à-dire la liste des règles associées à chacun des objets présents dans la scène. Dans ces approches, l'utilisateur est considéré comme observateur extérieur du comportement des icônes, et sa tâche est de leur inculquer des comportements.

Le paradigme de programmation supportés dans les tableurs est réinvesti en programmation visuelle, et conduit à l'approche dite à base de formulaires ou genre tableur ("*spreadsheet like*"). L'exemple le plus connu en est Forms/3 [Hays & Burnett 95].

Le principe des cellules et des formules associées est repris des tableurs, ainsi que l'utilisation de références relatives ou absolues. Cela permet d'offrir un style de programmation simple et concret associé à un rendu immédiat.

Schématiquement, Forms/3 permet de définir des formes, qui sont l'équivalent des cellules des tableurs. À ces cellules sont associées des formules pour calculer leurs valeurs. Les paramètres des formules peuvent être des références à des valeurs d'autres cellules. Ils sont dynamiquement remplacés par les résultats des calculs des formules associées à ces cellules. Ces définitions permettent de définir des types de cellules, les formulaires sont les instances de ces cellules.

Les types peuvent même être récursifs. Pour cela, il suffit de définir une cellule dont la formule dépend de la valeur de la cellule elle-même. L'exemple donné dans la référence citée ci-dessus pour illustrer les définitions récursives, est le calcul du $n^{\text{ième}}$ nombre de la suite de Fibonacci.

D'autres approches permettent de décrire directement les changements d'état du système à travers la spécification du diagramme de transition. Ces approches reposent sur le paradigme de flot de contrôle pour une machine virtuelle abstraite. Il apparaît aujourd'hui une évolution introduisant l'utilisation du paradigme de flot de données [Ibrahim 98].

"Visual programming has recently undergone a paradigm shift: historically, many notations were based on control flow, while many current languages are based on the notion of data flow". [Blackwell & al. 98]

Le développement des systèmes à programmation visuelle conduit à des études théoriques sur les langages visuels, c'est-à-dire les langages dont le formalisme peut être représenté en deux dimensions.

Dans ce cadre, on peut citer les travaux de Erwig [Erwig 97] et la collection d'articles réunis dans [Marriott & Meyer 98]. Rekers transcrit les techniques classiques de compilation des langages textuels en leurs correspondants visuels [Rekers 92], afin d'associer édition et compilation de graphes, puis poursuit cette idée en proposant un pendant textuel avec les éditeurs syntaxiques [Rekers 94]. L'organisation d'icônes dans le plan constitue une phrase visuelle, et est un homologue en deux dimensions de l'arrangement à une dimension des "*tokens*" des langages de programmation textuels conventionnels.

Plus largement, l'état de l'art de [Botshernitsan & Downes 97] présente le domaine de la programmation visuelle comme "*grown from a marriage of work in computer graphics, programming languages, and human-computer interaction.*".

La typologie établie par Burnett permet de classer les différentes approches ainsi que la bibliographie qui s'y rapporte [Burnett & Baker 94]. La rubrique consacrée à la classification des langages (VPL-II-A : *Language Classifications*) rassemble les articles qui traitent des paradigmes utilisés, comme par exemple les langages basés sur les contraintes (*Constraint-based languages*), sur des manipulations de formulaires et de formules comme dans les tableurs (*Form-based and Spreadsheet-Based Languages*), les langages de programmation par démonstration, ou les langages impératifs. La rubrique VPL-II-B (*Visual Representations*) est consacrée aux articles traitant des représentations visuelles : diagrammes, icônes ou séquences de dessins statiques. Les caractéristiques langagières sont discutées dans les articles de la rubrique VPL-III (*Language Features*) et plus particulièrement les thèmes traitant des informations sur des données abstraites (*Data abstraction*), les types de données et les structures (*Data types and structures*). La rubrique VPL-IV-A (*Language Implementation Issues*) s'intéresse aux solutions informatiques envisagées : les approches informatiques (comme la prise en compte d'une initiative liée à la demande ou par les données), les analyses et décompositions grammaticales, et les transducteurs (interpréteurs et compilateurs). Il y a enfin la rubrique portant sur la théorie des VPLs.

Les quelques rubriques présentées ici sont juste données à titre d'illustration et de précision de termes, mais cette liste n'est pas complète. Ce ne sont pas les systèmes (ou produits) qui y sont classés, mais les articles parlant soit de ces produits selon l'aspect du système qui y est traité, soit établissant une taxonomie selon certains points de vue, soit traitant de problématiques plus théoriques.

Certains systèmes sont basés sur une programmation hétérogène visuelle et textuelle [Erwig 95]. Ils permettent à des experts de domaines d'application, sans connaissance en informatique, de maîtriser plus finement la conception des applications qu'ils doivent spécifier. Par exemple, le système DIVA permet la mise au point de didacticiels [Ibrahim 89, Ibrahim 95].

Ces systèmes sont prévus pour faire appel à différents intervenants, qui contribuent, en coopération, à l'élaboration du logiciel, les différentes phases du travail étant aidées par des outils adaptés.

Les experts du domaine définissent graphiquement la structure logique du didacticiel, intégrant des branchements conditionnels entre les différents composants logiciels décrits en langage naturel. Les programmeurs codent les composants logiciels à l'aide d'un éditeur capable de les relier à leurs spécifications, par animation visuelle (dynamisme). Ainsi, la forme graphique et la composition de texte proche du langage naturel sont combinées pour rendre abordable à des non-programmeurs la tâche de spécification du didacticiel et de vérification de la bonne adéquation du résultat obtenu.

c. L'apport effectif de la dimension visuelle à la programmation

Cependant, l'apport effectif de la caractéristique "visuelle" des langages est controversé. [Blackwell & al. 98] émet un doute sur la part de succès de produits commerciaux comme LabVIEW due à l'aspect visuel plutôt qu'au niveau de langage proposé.

En effet, l'objectif d'offrir la possibilité de programmer à des non-programmeurs est supposé atteint grâce à l'usage d'une syntaxe visuelle, mais LabVIEW fournit à l'utilisateur des bibliothèques bien gérées de composants logiciels réutilisables.

Une autre question concerne l'adaptation des langages visuels à différentes tailles d'applications [Burnett 95] : une technique efficace pour résoudre des problèmes d'une certaine dimension ne l'est plus obligatoirement en changeant d'échelle de problème.

[Blackwell 96] part à l'assaut des préjugés sur nos fonctionnements métacognitifs en programmation. Pour cela, il fait une analyse statistique d'articles sur la programmation visuelle, de façon à rechercher le bien fondé des arguments proposés pour privilégier la programmation visuelle au dépend de la programmation textuelle (croyances et convictions métacognitives).

Il considère douze affirmations. Pour chacune d'elles, il fait la part des choses entre les arguments "pour" et les arguments "contre". Par exemple, sur l'expression de la structure, il mentionne deux articles qui suggèrent que les systèmes très complexes ont peu de chance de se révéler trivialement faciles à comprendre lorsqu'ils sont présentés visuellement. Dès qu'un système est assez complexe pour que l'encapsulation devienne nécessaire, les dépendances cachées sont aussi problématiques avec les langages textuels qu'avec les langages visuels.

Les affirmations sont listées dans la deuxième colonne du tableau ci-dessous, et les conclusions apportées dans la troisième.

mots clés	affirmation suspecte	conclusion
1: <i>Straightforward advantages</i>	La programmation visuelle est naturellement supérieure à d'autres formes de programmations.	Il y existe des activités de programmation où les langages visuels sont inférieurs aux langages textuels, mais différentes représentations sont adaptées à des choses différentes.
2: <i>Improved productivity</i>	Qualité intrinsèque des langages visuels : leur appréhension est innée, ils ne demandent aucun apprentissage. Ils améliorent la productivité.	Peu de travaux étudient la capacité innée à la programmation visuelle, mais les dimensions cognitives de Green et Petre forment un outil de comparaison.
3: <i>Abstraction versus concreteness</i>	Les solutions de problèmes sont plus facilement perceptibles lorsque les informations abstraites sont converties sous une forme concrète.	Il n'y a pas de consensus sur le lien entre vision et abstraction, mais les modèles cognitifs abstraits sont plus efficaces dans les problèmes bien structurés et les modèles concrets sont meilleurs pour les problèmes semi-structurés. Certains formats de diagrammes accroissent la spécificité de la représentation d'un problème, rendant ainsi plus accessible la solution qu'avec des notations formelles.
4: <i>Expression of structure</i>	Les relations sont plus explicitement représentées et facilement reconnues dans des dessins que dans du texte.	La restriction plane réduit un certain degré de la structure de systèmes très complexes. L'application de la dimension cognitive aux dépendances cachées suggère que les systèmes très complexes auront peu de chances d'être faciles à comprendre à partir d'une représentation visuelle. Lorsque l'encapsulation devient nécessaire, les dépendances cachées deviennent aussi problématiques dans les langages visuels que textuels. Il est aussi possible qu'une notation visuelle exprime mieux la structure de l'ordinateur lui-même, de sorte que l'on puisse comprendre son état interne (modèle mental des

		machines)
5: <i>Comparision to natural language</i>	La communication par l'intermédiaire d'images ou de schémas (idéogrammes chinois, hiéroglyphes égyptiens, icônes de la vie courante) est naturelle et constitue un support universel de communication, indépendant de la langue maternelle : la programmation visuelle possède un caractère aculturel.	La nécessité pour des non-anglophones de se familiariser avec quelque douzaines de mots-clés anglais conduit au défaut qu'est l'utilisation de faux-amis par les anglophones. Les images peuvent compléter d'autres formes de communication, ainsi que constituer un palliatif pour certains handicapés, les illettrés ou les enfants pré-lettrés.
6: <i>Expressivity of pictures</i>	Une illustration contient plus d'information qu'un texte. Les gens préfèrent généralement les images aux mots.	Les illustrations donnent un accès global à l'information, l'esprit d'un journal a plus de chances d'être assimilé par elles qu'à partir d'un contenu d'information.
7: <i>Mental imagery</i>	Les langages visuels sont plus proches de l'image mentale qu'un texte. L'identification des structures perceptives et conceptuelles constitue un pont au dessus d'un fossé sémantique.	Les problèmes à contexte spatial sont mieux présentés sous une forme spatiale, comme par exemple la programmation de l'apparence d'une IHM : les environnements graphiques encouragent un modèle mental plus proche du domaine visé.
8: <i>Resemblance of the real world</i>	Les langages visuels sont plus proches du monde réel que les langages textuels puisque nous évoluons dans un monde visuel où les images abondent.	Des représentations visuelle communes peuvent être réutilisées sous la forme de métaphores pour la programmation, les autres "entrées" (son), étant plus linéaire.
9: <i>Cognitive resources</i>	Les formes sont plus faciles à imaginer que les mots puisque le cerveau humain est optimisé pour traiter les images et la vision.	On est ici influencé fortement par la métaphore informatique de la pensée.
10: <i>Intuition and Naturalness of pictures</i>	L'information graphique est plus intuitive que le texte, autorisant l'appel à l'"intelligence innée" et une "interaction intuitive".	Les spécialistes s'appuient sur des schémas pour maîtriser les situations complexes, la référence pour cela étant que les formes de diagrammes standard par les ingénieurs informatiques sont considérées comme étant familières ou encore traditionnels. Si la programmation visuelle ne requérait que des moyens naturellement émergents, il ne serait plus nécessaire d'apprendre, ni de passer du temps à un labeur intensif, frustrant.
11: <i>Syntax versus semantics</i>	Le langage de programmation intuitif idéal ne nécessiterait aucun travail pour écrire un programme : l'utilisateur exprimerait ce qu'il veut faire naturellement et le travail de programmation serait automatique. La façon de le réaliser par les langages visuels est exprimée	Tout langage a une syntaxe, sinon ce ne serait pas un langage, ceux de programmation visuelle n'allègent la contrainte syntaxique que pour les tâches triviales, pas plus que les éditeurs textuels dirigés par la syntaxe. La sémantique ne peut être

	comme une différence entre syntaxe et sémantique, le programmeur n'a pas besoin d'être concerné par la syntaxe, qui est le problème de l'ordinateur, c'est la sémantique du programme, ce qu'il est supposé faire qui est important.	exprimée sans la syntaxe, pas plus qu'aucun système de notation
12: <i>Direct icon manipulation</i>	Généralisation de la métaphore du bureau pour expliquer en quoi les icônes sont meilleures pour les tâches de programmation.	L'intérêt de la manipulation directe est incontestable. Il est peu probable que les icônes soient plus porteuses de sens intuitif que les mots pour beaucoup de constructions de programmes, mais la position comme mécanisme de codage a un support plus psychologique. Des icônes représentatives sont plus faciles à retenir si elles sont abstraites durant les premières confrontations avec un environnement, mais avec une familiarité croissante, la position est plus utile pour retrouver des fonctions particulières.

Le premier critère (1: *Straightforward advantages*) révèle le parti pris pour le visuel contre le textuel de certains argumentaires écrits pour justifier certains travaux de recherche en programmation visuelle. En dehors de ce premier critère, la liste des critères de Blackwell exprime implicitement les qualités désirées par les programmeurs pour les environnements de programmation. Les affirmations suspectes constituent presque toutes autant de revendications pour un environnement idéal. Ainsi, l'utilisateur aimerait que son environnement :

1. améliore sa productivité, l'allégeant des étapes d'adaptation et d'apprentissage (2 : *Improved productivity*),
2. se rapproche de son expression naturelle (5 : *Comparison to natural language*), au delà de sa matérialisation par sa forme ou le langage, en lui offrant une communication en relation directe avec ses processus cognitifs (7 : *Mental imagery*, 10 : *Intuition and Naturalness of pictures*, 11 : *Syntax versus semantics*), et en optimisant son temps d'accès à ses ressources cognitives (9 : *Cognitive resources*),
3. lui permette d'utiliser son expérience du monde réel par l'usage de métaphores (8 : *Resemblance to the real world*),
4. permette de manipuler directement les objets du domaine plutôt que de passer par une étape explicite de désignation (12 : *Direct icon manipulation*),
5. offre un pouvoir d'expression adapté à la représentation de ses problèmes (4 : *Expression of structure*), qui permette de les concrétiser (8 : *Resemblance to the real world*), minimise le fossé sémantique (7 : *Mental imagery*) et rende perceptible les solutions (3 : *Abstraction versus concreteness*), et
6. permette de travailler sur des représentations complètes à la fois (alternativement ou à la demande) détaillées et synthétiques (6 : *Expressivity of pictures*).

Brad Myers fait état de l'évolution de l'argumentaire se rapportant au support textuel pour la programmation [Myers 98] :

"Many people argue that programming is difficult because it requires the precise use of a textual language, and that a system that eliminates language will be inherently easier to use . However, to date nobody has been able to prove that a visual language is superior to textual languages for all tasks [Green 91]. To the contrary, often textual languages are superior to visual languages. It seems that visual languages might be better for small tasks, but often break down for large tasks, and we want to make sure that the proposed language will be appropriate for creating complete applications. The good news is that it is not always difficult to learn to program in a textual language. In fact, the most successful end-user programming system is the spreadsheet, which is text-based . "

Ainsi programmation visuelle et textuelle sont complémentaires, et le choix le plus pertinent de l'une ou de l'autre dépend du contexte. Quand la situation des problèmes à résoudre requiert un pouvoir d'expression accessible par l'une et l'autre, elles peuvent être utilisées conjointement.

2°) Programmation par l'exemple, par démonstration

La programmation par l'exemple regroupe un certain nombre d'approches envisagées pour fabriquer des programmes en donnant des exemples de leurs comportements ou effets. Toutes ces approches prônent un travail sur des exemples concrets plutôt que la description de procédures dans l'abstrait [Cypher & al. 93].

Contrairement aux systèmes de programmation visuelle, le but explicite de ces systèmes n'est pas de fournir à l'utilisateur des moyens pour effectuer consciemment des activités de programmation, mais plutôt de lui offrir des fonctionnalités pour faciliter l'accomplissement de ses tâches, lorsqu'elles peuvent prendre la forme de programmes. Ainsi l'utilisateur n'a pas nécessairement conscience d'effectuer une activité de programmation : le système le fait à sa place, et lui propose d'utiliser le résultat de cette activité.

a. Définition

Une définition de la programmation par démonstration est donnée dans [McDaniel 97] :

"One method for simplifying the programming process has been programming-by-demonstration (PBD). Instead of using a textual notation, the software author builds the program by providing examples of the intended interactions between the user and the application. Examples are demonstrated using the same interface normally used to create and manipulate the application's data. The system uses the examples to infer the author's intention and assembles code to execute the program."

[Maulsby 93] fait une récapitulation historique du domaine de recherche que constitue la programmation par démonstration, collectant les systèmes, techniques et idées dont la contribution est significative.

b. Concrètement

La première motivation de la programmation par démonstration est d'éviter aux utilisateurs l'exécution de tâches répétitives à la main. La répétition de tâches est commune dans les applications informatiques.

Dans l'exemple de [Sugiura 96], lors de l'édition de textes, un utilisateur peut vouloir changer la police de caractère du mot "ordinateur" chaque fois qu'il apparaît. Une macro pour cette tâche peut lui éviter de devoir rechercher chacune des instances du mot et d'invoquer la commande de modification de police à chaque fois. Cependant, comme un certain degré de capacité à programmer est requis pour créer des macros, les utilisateurs peu habiles ou peu expérimentés doivent effectuer à la main ces tâches répétitives.

La programmation par démonstration est une méthode pour convertir une "démonstration" de l'utilisateur en un code exécutable. En PBD (*Programming By Demonstration*), l'utilisateur démontre simplement comment une macro devrait se comporter, et le système génère un programme correspondant à la démonstration de l'utilisateur.

Ainsi l'utilisateur n'est pas censé avoir besoin d'apprendre quelque technique de programmation que ce soit. Mais la mise au point des macros se révèle souvent elle-même consommatrice de temps et d'attention. Par exemple, les systèmes qui imposent une phase d'enregistrement explicite, encadrée d'opérations spéciales pour démarrer et arrêter l'enregistrement, obligent l'utilisateur à anticiper sur la pertinence de la définition d'une macro pour la tâche qu'il exécute. Même lorsque l'utilisateur a été capable d'anticiper ce besoin, il doit établir un plan pour sa démarche de "démonstration" et l'exécuter avec attention et précision.

Les techniques utilisées vont du mode enregistrement explicite (Metamouse [Maulsby 98], Peridot [Myers 93], SketchPad [Sutherland 63]), à l'initiative automatique de définition de macro (*macro auto-definition* : EAGER [Cypher 93], Dynamic Macro [Masui & Nakayama 94], DemoOffice [Sugiura 96]), en passant par l'extraction manuelle ou automatique, d'une suite d'actions de l'historique, c'est-à-dire des actions que l'utilisateur a effectuées pour créer une donnée spécifique (*action slicing* : Chimera [Horwitz & al. 93], Cabri-géomètre [Bellemain 92]).

c. Application "géométrique"

En dehors des applications en édition textuelle ou en géométrie impérative, une application importante de la programmation par démonstration est la Conception Assistée par Ordinateur (CAO).

L'objectif de la CAO est de fournir un système grâce auquel des utilisateurs non-programmeurs sont capables de décrire des objets réels avec leur topologie et leurs particularités. La topologie d'un objet correspond à la notion de figure en géométrie.

Tous les objets d'une même famille peuvent être définis par une topologie spécifique : cette topologie engendre la famille. Une topologie est constituée d'entités géométriques et de contraintes entre ces entités. Les particularités de l'objet dans la famille sont aussi décrites par des contraintes entre les entités géométriques qui constituent sa topologie, mais ces contraintes gardent un caractère modifiable.

De même qu'un dessin est une instance particulière d'une figure, un objet peut-être vu, dans une certaine mesure, comme une instance particulière de la topologie décrivant la famille de l'objet. La différence réside dans la possible inconsistance du système de contraintes : il peut être surcontraint ou sous-contraint selon que les contraintes sont surabondantes ou insuffisantes.

Cette nécessité de décrire "à part" des contraintes supplémentaires conduit à l'introduction d'une composante déclarative dans la description des objets en CAO. Quant à leur topologie, elle peut être définie de manière impérative (non variable).

"Un système de programmation est dit impératif si la structure de contrôle de l'exécution de l'algorithme résultant du programme est décrite par le programmeur. Il est dit déclaratif si la structure de contrôle d'exécution est fabriquée par le système à partir de relations entre objets utilisées par le programmeur." [Girard 92]

Ce caractère déclaratif ou impératif conduit, par la possibilité de sous-contrainte de l'exemple sur lequel est "démontrée" la conception, à une considération de l'ambiguïté et à la notion d'inférence [Halbert 84].

d. Notion d'inférence

Plus précisément, l'assertion suivante définit la programmation basée sur l'exemple : "*Un système de programmation est dit basé sur exemple si l'utilisateur peut utiliser les valeurs d'un exemple d'exécution pour définir les objets sur lesquels porte le programme à construire*" [Girard 92].

Le rapport de l'utilisateur et du système à l'exemple conduit à des précisions. Le critère discriminant lié à la notion d'inférence apparaît dans [Halbert 84]. Il est repris par Myers [Myers 86].

Ce critère différencie les systèmes capables de déduire un programme des actions de l'utilisateur selon la nécessité du recours à une réambiguïsation par dialogue ou non. Les premiers, caractérisés par Halbert par la formule "*Do What I Mean*" ("Fais ce que je veux dire"), supportent la programmation par l'exemple et sont dits "avec inférence plausible" (abduction). Les seconds, caractérisés par Halbert par la formule "*Do What I Did*" ("Fais ce que j'ai fait"), supportent la programmation avec exemple et sont dits "sans inférence plausible".

Les deux approches caractérisées par Halbert sont précisées par la nature impérative ou déclarative du système, ce qui conduit à une classification de ces systèmes selon trois méthodes de programmation [Potier 95]. Les trois méthodes de programmation qui distinguent les systèmes de CAO existants sont la programmation traditionnelle, la programmation inférentielle et la programmation interactive.

La programmation traditionnelle repose sur une définition textuelle des programmes.

La programmation inférentielle exige que le système soit capable d'inférer des programmes à partir d'un cas particulier utilisé comme modèle (systèmes "variationnels"). Elle cherche à inférer une méthode de construction des différents objets d'une famille à partir d'un objet particulier sans s'intéresser au processus de conception de ce dernier. La topologie de la famille est identifiée au système expert constitué de l'ensemble des règles logiques qui traduisent chacune ses contraintes.

Grâce à la dissociation entre la topologie (fixe) et les particularités (contraintes variables), la génération d'un nouvel objet de la famille s'effectue par propagation de nouvelles règles. Le formalisme de propagation est soit algébrique, lorsqu'il est effectué par la résolution d'un système d'équations non linéaires, soit déductif, lorsqu'il est effectué par l'activation du moteur d'inférence à partir des faits nouveaux constitués par ces nouvelles règles. C'est une approche déclarative.

Enfin, la programmation interactive consiste à ce que le système génère un programme au fur et à mesure de la définition interactive d'un objet de la famille (systèmes "paramétriques"). Basée sur un exemple, elle vise à capturer le processus de conception de l'utilisateur pour cet exemple en tant qu'objet particulier d'une famille et à en déduire le processus de tout autre objet de la famille. C'est une approche impérative : elle reflète la démarche de conception adoptée par l'utilisateur.

e. Accès aux programmes

Même si le but initial de ces systèmes n'est pas d'offrir à des non-programmeurs des moyens pour assimiler des concepts informatiques théoriques, il n'en reste pas moins que leurs utilisateurs doivent remédier à des difficultés spécifiques de programmation.

Dans les approches de programmation par démonstration, ce n'est pas l'utilisateur qui apprend à programmer, mais l'environnement qui lui propose des programmes extraits automatiquement de ses constructions.

En CAO, [Heydon & Nelson 94] fait état de deux artefacts destinés à accompagner l'utilisateur dans sa tâche de conception : un retour d'information graphique pour identifier les contraintes définies entre les objets, et l'affichage et l'édition du programme de construction grâce à des commandes. Ils justifient leur préférence pour le deuxième :

« *It can be difficult to design an interface that allows users to determine and control what constraints are imposed on the drawing. Some systems have dealt with this problem by making visible the constraints in the drawing, but this solution can be visually busy and confusing. We think double-view editing is a better approach: the constraints are visible in the program view, and can be modified or deleted in that view with ordinary text-editing operations.* »

Ce dernier argument est aussi pertinent dans le cas de Cabri-géomètre.

Ainsi dans Juno-2, un éditeur à double vue affiche la vue de l'image et simultanément le programme en langage Juno-2.

Dans toutes ces approches, les tâches de conception sont illustrées simultanément dans une vue graphique du résultat, ce qui constitue un moyen de contrôle pour l'utilisateur.

3°) Approches cognitives

Issus du domaine de la psychologie de la programmation, les travaux de Green et Fitter en 1979 abordent les facteurs cognitifs qui interviennent dans l'utilisation de notations. Au travers de l'évolution des articles [Green 89, Green 91, Green & Petre 96], Thomas Green définit un cadre d'analyse basé sur les "dimensions cognitives des notations".

Partant d'une précision sur la différence entre des données brutes et ce qui en fait de l'information, — citons-le :

"Data must be presented in a usable form before it becomes information, and the choice of representation affects the usability. But usability is not simply 'better' or 'worse'; how good a representation is depends on what you want to use it for."

il relativise les avantages de l'usage de diagramme ou de texte par le problème à résoudre.

Les effets de la structure de notation sont résumées par deux maximes :

"Every notation highlights some kinds of information at the expense of obscuring other kinds [...]. Corollary : part of the notation design problem is to make the obscured information visible."

et

"When seeking information, there must be a cognitive fit between the mental representations and the external representation. [...] if you think iteratively, recursion will be hard".

a. Des dimensions cognitives

Les dimensions cognitives des notations ont pour objectif de décrire les rapports entre le programmeur et l'environnement de programmation. Elles sont constituées d'un certain nombre de termes choisis pour leur caractère suggestif qui facilitent la compréhension par des non-spécialistes, tout en recouvrant le domaine de la programmation. Ces dimensions sont interdépendantes et l'amélioration d'un artefact de l'interface de l'environnement informatique selon l'une d'elle, conduit à un compromis pour les autres. En voici la liste :

- gradient d'abstraction (un gradient correspond à une étendue et est associé aux possibilités d'évolution à travers cette étendue), *"Abstraction Gradient: What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?"*
- proximité entre le monde du problème et le monde du programme, *"Closeness of mapping: What 'programming games' need to be learned?"*
- consistance (possibilité de réutiliser une généralisation de comportement d'un outil sur un autre, aidant à une intuition du comportement), *"Consistency: When some of the language has been learnt, how much of the rest can be inferred?"*

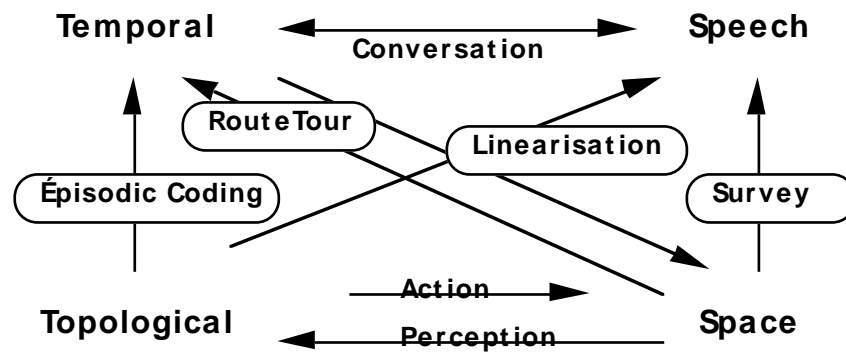
- caractère diffus des notations (répandues dans toutes les directions, en opposition à la concision), "*Diffuseness: How many symbols or graphic entities are required to express a meaning?*"
- susceptibilité d'induire en erreur (risque d'erreur), "*Error-proneness: Does the design of notation induce 'careless mistakes'?*"
- difficulté des opérations mentales (au niveau des notations, comme au niveau de leur enchevêtrement), "*Hard mental operations: are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?*"
- dépendances cachées (perceptives ou symboliques) et localité, "*Hidden dependencies: is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?*"
- engagement prématuré, "*Premature commitment: do programmers have to make decisions before they have the information they need?*"
- évaluation progressive (possibilité d'exécution partielle, par exemple en cours d'édition), "*Progressive evaluation: can a partially-complete program be executed to obtain feedback on 'How am I doing' ?*"
- expressivité des rôles (lisibilité des rôles d'un composant dans le tout - identité porteuse de sens, modularité bien structurée, ..), "*Role-expressiveness: can a reader see how each component of a program relates to the whole?*"
- notations secondaires (notations ne faisant pas strictement partie de la structure syntaxique, mais introduites pour aider à la perception de la sémantique ; ex : commentaires, indentation), "*Secondary notations: can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the 'official' semantics of the language?*"
- viscosité (quel est l'effort nécessaire pour le moindre changement ?), "*Viscosity: how much effort is required to perform a single change?*"
- visibilité (vue globale et complète possible), "*Visibility: is every part of code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?*".

Ces dimensions constituent une clé pour évaluer les qualités des environnements de programmation. Ils permettent de comparer des environnements de programmation textuelle et des environnements de programmation visuelle.

b. Vers la programmation naturelle

Blackwell utilise les dimensions de Green pour ses travaux, qui portent principalement sur l'apport des diagrammes à l'activité de programmation [Blackwell 97]. Il s'attache à tout ce qui a trait aux schémas (diagrammes) : ce qui les caractérise, ce qui motive leur introduction (ressemblance, métaphores...), ce qu'ils apportent (localité et correspondance — identité, pouvoir d'expression et spécificité...). Il étudie aussi la contribution du visuel dans toute activité intellectuelle, en particulier en modélisation, et l'interaction du visuel et du verbal.

Le schéma suivant, des supports internes / externes de pensées (topologique et temporel / verbal et spatial) et de leur transfert, permet d'étudier en quoi les différentes interactions représentées peuvent influencer sur les raisonnements. "*Interaction between verbalisation and images can compromise diagrammatic reasoning*".



Blackwell met en doute les arguments donnés par les partisans de la programmation visuelle qui tentent de justifier la suprématie naturelle du raisonnement visuel, pour faire ressortir la contribution effective de la composante visuelle dans l'aide au raisonnement et à la modélisation.

Les arguments de Blackwell ont montré une nouvelle approche pour offrir à des non-programmeurs des capacités de programmation, et l'approche cognitive de la programmation lui a ouvert la voie. Il s'agit dans cette approche, d'adapter l'environnement informatique aux processus naturels des utilisateurs, plutôt que d'obliger les utilisateurs à apprendre à programmer dans l'environnement du logiciel qu'ils utilisent, c'est-à-dire à s'adapter à l'environnement de programmation du logiciel et à assimiler le paradigme de programmation qu'il impose.

[Pane & Myers 96] donnent un état de l'art de la recherche sur les programmeurs novices, répertoriant le vocabulaire utilisé dans la bibliographie pour classer les différentes approches. Cette étude contribue à la recherche sur la « programmation naturelle ».

L'objectif de cette recherche est l'élaboration d'environnements permettant à des non-programmeurs d'aborder des tâches de programmation sans apprentissage préalable.

Cette approche est basée sur l'étude des processus cognitifs menée par [Pane & al 98], et en particulier l'observation du discours et de la structure des solutions déterminées par des enfants et des non-programmeurs confrontés à des concepts de programmation [Pane & al 97].

L'objectif est de cerner les difficultés potentielles et de suggérer des approches alternatives plus proches de la manière dont les personnes pensent. Dans cette approche, l'activité de programmation est vue comme un processus de transformation d'un plan mental décrit en termes familiers vers un plan compatible avec l'ordinateur.

L'approche cognitive contribue de deux manières à notre argumentaire :

- informations utiles pour les activités de programmation,
- en montrant que les nouvelles approches pour aider à la programmation, d'une part ont pour objectif de rapprocher les environnements des utilisateurs plutôt que de d'obliger les utilisateurs à s'adapter aux nécessités des ordinateurs, et d'autre part s'intéressent au langage et aux notions naturellement utilisés par les programmeurs en recensant des critères pertinents pour mettre en place ou consolider des artefacts dans les environnements interactifs.

Il semble donc que la tendance actuelle se tourne vers les environnements multimodaux, c'est-à-dire des environnements supportant des formes alternatives d'expression.

D) Conclusion

Au cours de ce chapitre, nous avons vu l'évolution des supports au raisonnement utilisés pour définir des programmes au cours de l'histoire de l'informatique, ainsi que l'apport des interfaces homme-machine dans ce domaine.

La programmation est une activité de traduction du raisonnement humain en tâches exécutables par la machine. Grâce aux interfaces Homme-Machine à manipulation directe, le support graphique n'est plus seulement un support visuel d'information puisqu'en sens inverse, l'utilisateur peut transmettre à l'ordinateur une information sur ses désirs. Dans le sens de l'humain vers la machine, la communication graphique s'effectue aujourd'hui principalement à travers le médiateur qu'est la souris qui suffit pour spécifier les choix d'outils et sélectionner les données. Le support textuel est gardé pour l'entrée de textes. Dans le sens de la machine à l'humain (on parle de "rendu" [Bastide & al. 98]), les supports allient graphismes, textes et sons. Une interface graphique qui constitue un environnement de programmation est un support d'expérimentation manuelle, c'est-à-dire un support de tentative et de validation.

Trois approches ont pour but commun d'offrir à des utilisateurs non-programmeurs la possibilité d'aborder des activités de programmation. Elles diffèrent par le public visé et la motivation de ce public vis à vis de ces activités. Dans ces trois approches, les trois façons de considérer le programmeur conduisent à des principes différents de conception des environnements.

- La première approche présentée ici est la programmation visuelle. Elle a pour objectif d'utiliser le support visuel pour programmer. Dans cette approche, l'utilisateur doit s'adapter à l'outil informatique qu'on lui fournit, et acquérir des compétences sur les concepts informatiques. Le public est supposé motivé par l'apprentissage de l'informatique. Une de ses applications est l'apprentissage de la programmation à des utilisateurs élèves novices en programmation.
- La deuxième approche est la programmation par démonstration. Les utilisateurs ne sont particulièrement motivés pour apprendre la programmation. Celle-ci s'inscrit plutôt comme une nécessité vis à vis de la tâche qu'ils ont à accomplir. L'objectif de cette approche est d'offrir des fonctionnalités de programmation les moins lourdes et les plus discrètes possibles tout en restant les plus efficaces. L'utilisateur utilise et valide les programmes proposés par l'environnement.
- La troisième approche est la programmation naturelle. Cette approche est la plus récente. Elle repose sur le constat des difficultés rencontrées dans les approches tout-visuel. Elle est issue des recherches en sciences cognitives et vise à adapter l'environnement informatique aux prédispositions naturelles du raisonnement humain plutôt qu'à obliger l'humain à faire le pas principal d'apprentissage. Dans ce cas, ce n'est pas exactement la façon de considérer l'utilisateur qui diffère, mais le point de vue du concepteur de l'environnement de programmation, et par là, de l'environnement lui-même.

De la première approche ressort une liste de critères désirés par les utilisateurs pour les aider dans leur tâche de programmation. Elle contribue à montrer la complémentarité des programmations visuelles et textuelles. La deuxième approche propose des techniques de programmation automatique capables de soulager et d'orienter les utilisateurs. Elle contribue à justifier l'intérêt de la simultanéité de l'édition du programme et de la réalisation de ses effets. La dernière approche, l'approche cognitive, a permis de mettre en évidence un certain nombre de critères pour comparer les supports de programmation. Ces critères permettent de justifier l'importance de certaines spécificités des environnements qui supportent des activités de programmation dans des domaines particuliers, comme la géométrie dynamique.

Pour notre part, nous considérons que les vues textuelles et visuelles véhiculent des informations qui peuvent être complémentaires. Ces vues permettent des manipulations des objets qui leur sont propres. Ces manipulations d'objets peuvent être adaptées aux tâches à exécuter. Au-delà de la visualisation seule, la manipulation de données suit des caractéristiques liées et limitées par la vue qui y donne accès.

De même que certaines personnes qualifient leur façon d'accéder à leur propre mémoire de plutôt visuelle ou de plutôt auditive, il n'existe pas un mode de pensée meilleur pour tous les utilisateurs. C'est à l'utilisateur de choisir la vue qui lui convient le mieux, c'est-à-dire celle dans laquelle il réussit le mieux à mettre en correspondance ce qu'il programme et ce qu'il a dans la tête. Nous rejoignons la remarque de [Green & Petre 96], vis à vis de l'encadrement du travail de programmation : *"the order of working should be left to the programmer, not prescribed by the environment."*

Une dernière dimension peut être considérée comme une vue : il s'agit de la contribution dynamique. En effet l'animation et la réponse aux actions de l'utilisateur facilitent la compréhension et donnent un accès à d'autres informations. Cette contribution prend place dans la synchronisation des effets visuels entre les outils et les différentes vues. Elle est contenue dans l'association de l'action de l'utilisateur à un retour d'information pertinent appliqué à chacune des sorties de l'IHM.

La composante dynamique est caractérisée par ce qui est nommé « interaction » dans les conceptions d'IHM. La spécification de l'interaction est une difficulté fondamentale de la spécification des interfaces homme-machine.

L'intégration de plusieurs vues au sein d'un même logiciel conduit à la spécification d'un environnement dans lequel l'interaction de l'utilisateur est équivalente dans toutes les vues. L'action définie par une interaction dans n'importe quelle vue peut produire des résultats dans toutes les vues.

Nous proposons d'intégrer au logiciel Cabri-géomètre, une vue textuelle du programme qui soit corrélée à la vue de son exécution. Le choix de la forme textuelle de cette vue dans une première version, est imposée à l'utilisateur, mais les mécanismes mis en place doivent s'étendre à l'adjonction d'une autre vue de forme graphique, et l'utilisateur devra pouvoir en choisir une et naviguer de l'une à l'autre. Ainsi, l'approche considérée consiste à synchroniser la vue textuelle du programme avec la vue interactive de son exécution, en préparant l'intégration future d'une vue graphique. Chacune des vues doit pouvoir être activée ou désactivée par l'utilisateur, indifféremment.

Pour définir les artefacts à mettre en place autour de la vue textuelle, le chapitre suivant propose un examen de la façon dont les différents dimensions et critères établis pour aider à programmer (vus dans le présent chapitre) sont mis en œuvre.

Chapitre 3

D'autres logiciels permettant de programmer

L'objectif de ce troisième chapitre est de regarder de quelle manière sont proposés et gérés des artefacts permettant des activités de type programmatoire dans d'autres logiciels que Cabri-géomètre.

Les différents aspects dont l'intérêt a été justifié dans le chapitre précédent sont observés plus précisément.

L'approche suivie consiste à considérer ces logiciels, qui permettent donc tous des activités de programmation, en partant des plus éloignés du domaine et du contexte de Cabri-géomètre, pour arriver à ceux qui s'en rapprochent le plus.

A) Ceux qui sont très éloignés du contexte

1°) Logiciels de dessin

Dans des logiciels de dessin comme MacDraw ou ClarisWorks (vectoriel), il n'y a en fait pas ou très peu de véritable possibilité de programmation. Même s'il est possible de regrouper des objets, il n'y a pas de lien direct entre les éléments du dessin : chacun se réfère en premier lieu à la feuille où il est dessiné.

Dans MacDraw, il n'y a pas de macros. Par contre, il y en a dans ClarisWorks, mais elles constituent seulement un enregistrement des actions effectuées, réexécutées exactement sur la même position qu'à leur définition. Quelques autres défauts de fonctionnement majeurs découlent des choix minimaux d'implémentation de ces macros : par exemple, l'imbrication de deux définitions provoque une exécution en boucle infinie.

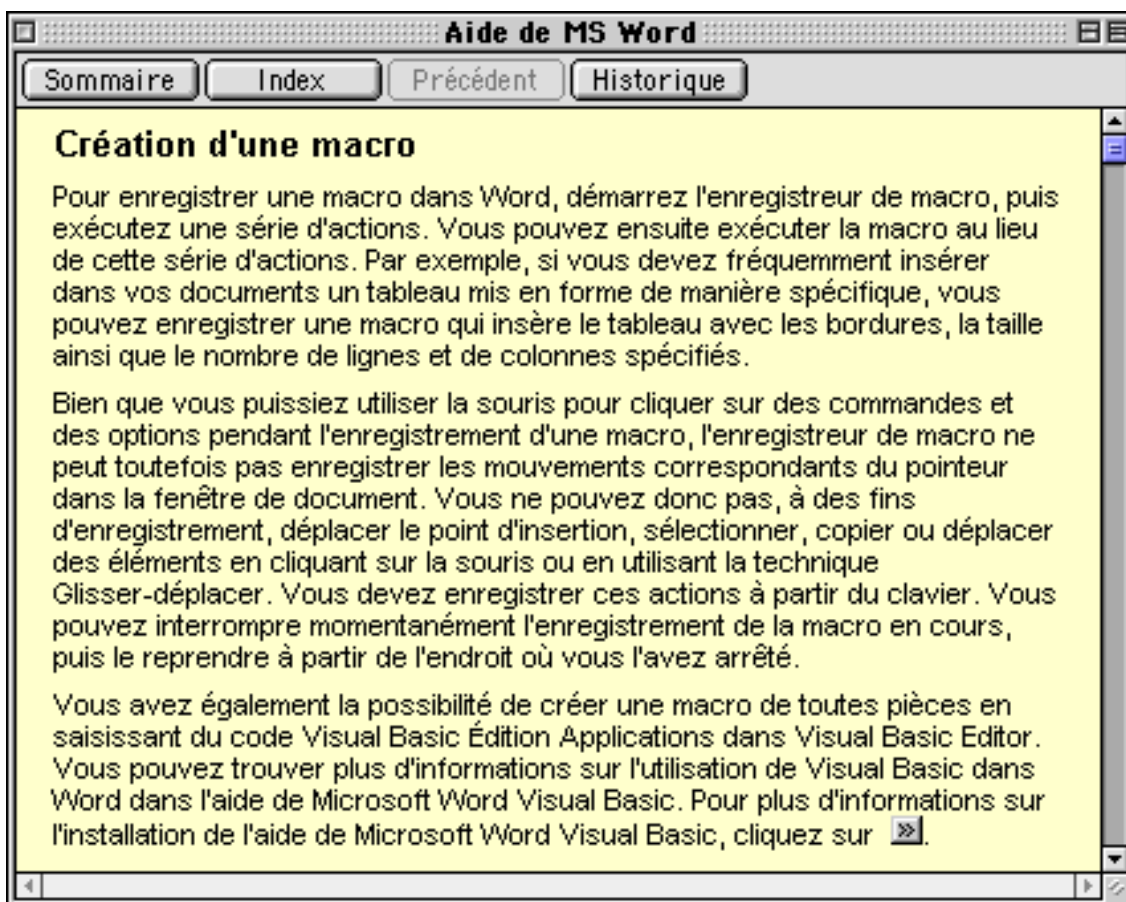
L'animation permise n'est pas modifiable ni spécifiable : le comportement des productions est toujours celui qui est prévu dans le logiciel. Il n'est pas spécifiable par l'utilisateur : tous les objets ne sont contraints que par leur type et il n'y a pas de type d'objet les rendant dépendants d'autres objets, en dehors de l'outil d'« association ». Avec cet outil, les objets associés peuvent être considérés par l'utilisateur comme formant un seul objet dont les seules animations possibles sont des animations de déplacement global, de symétrie ou de déformation homothétique. Il n'y a pas non plus de déformations relatives.

2°) Éditeurs textuels

a. Éditeurs classiques

Les tableurs comme Excel, et les éditeurs de documents comme Word98, offrent à l'utilisateur la possibilité de définir des macros pour automatiser des tâches.

L'illustration suivante montre un extrait de l'aide de Word98.



Seuls les choix des commandes sont enregistrés, ainsi que tout ce qui peut être spécifié par l'intermédiaire de caractères : la dimension de l'interaction n'est pas prise en compte au-delà du clavier.

Le logiciel reconnaît qu'une commande a été utilisée, mais l'enregistrement ne retient aucune méthode de calcul relative à la position de la souris lors de la définition. Une macro est exactement équivalente au texte du programme qui peut être édité en saisissant du code pour Visual Basic.

Pour tous les éléments de Word, les objets disponibles sont par exemple les champs, les signets, les documents et les caractères. Les commandes sont exprimées à l'aide des propriétés et des méthodes relatives aux objets.

Sans l'édition textuelle, les macros ne sont modifiables que par prolongation de la dernière commande après son exécution. Le logiciel ne fournit aucune aide pour la mise au point des macros. La seule aide qui traite d'un problème de bogue dans une macro définie par l'utilisateur, informe qu'il n'est possible d'y remédier que juste après l'action non désirée.

b. Éditeurs de documents structurés

Grif et Thot sont des éditeurs de documents structurés [Quint 87, Bonhomme 97]. La différence avec les éditeurs classiques est qu'ils permettent à l'utilisateur de manipuler les documents à travers leur structure, qui correspond à l'organisation du document en parties, chapitres, titres, références et illustrations, etc.

Par exemple, dans Thot, chaque document est organisé selon une structure spécifique. La structure générique définit la manière dont les structures spécifiques peuvent être construites. Une classe de documents est caractérisée par une structure générique : tous les documents d'une classe sont construits en accord avec la même structure générique.

Les structures génériques ne décrivent que l'organisation logique des documents, pas leur présentation. Une présentation graphique peut être dérivée du modèle de document sans être incluse dans le document lui-même, en associant une présentation générique à la structure générique. Des règles de présentation spécifient la présentation de chaque élément de la structure logique du document, relativement à son contexte dans cette structure.

Entrée de la structure du document

L'utilisation de Thot s'effectue en deux temps, par éventuellement deux utilisateurs de profils différents. Le premier a pour tâche de spécifier la structure générique et la présentation associée. Son travail s'apparente à de la programmation, mais ses compétences sont celles du domaine du contenu des documents qui seront produits. Il prépare le travail du second utilisateur, et fait partie de la catégorie constituée des experts du domaine, présentée dans le paragraphe §I.2.B.2°. Comme en CAO, cet expert pourrait apprécier de disposer de fonctionnalités qui l'aideraient dans sa tâche. Par contre, le concept de macro ne semble pas pertinent dans le contexte de la définition de la structure générique, car les nécessités ne sont pas les mêmes.

En effet, la structuration des documents a justement pour objectif de préparer le travail d'édition qui suivra, avec un degré d'ajustement tel qu'il n'y aura aucun « rafistolage » répétitif ensuite, même s'il sera encore possible de définir des annotations. Ainsi, la définition de la structure générique a des chances de ne pas être elle-même répétitive, puisque la décomposition hiérarchique du document vise à regrouper les éléments du document dont le rôle est identique, et d'hériter les discriminations de manière progressive selon l'importance de ce rôle. De plus, la quantité d'éléments manipulés ne semble pas a priori tellement importante et l'activité de création d'une nouvelle structure générique peut s'appuyer sur la définition d'une autre.

La structure générique et la présentation des documents sont décrites à l'aide de deux langages spécifiques, appelés respectivement langage S et langage P. L'expert spécifie ses choix par deux programmes écrits dans ces langages. Il s'agit donc d'une programmation langagière à laquelle l'expert doit se former.

Le logiciel compile chacun des programmes, les lie et produit une application dédiée à l'édition de documents respectant la forme désirée.

Entrée du document

L'édition du document est le travail du second utilisateur. Pour réaliser sa tâche, il utilise l'éditeur dédié généré par Thot à partir des spécifications de l'expert. Cet utilisateur construit son document et définit une structure spécifique en naviguant entre les différents niveaux de la structure générique grâce aux menus et aux outils générés par le logiciel. À chaque niveau, et pour chaque élément de la structure spécifique, l'éditeur visualise la zone courante d'édition possible selon les contraintes décrites par la présentation, et la rend sensible.

Place de la manipulation

Nous avons décrit la place de la programmation dans les tâches à effectuer pour se servir de ce logiciel. Mais quelle est la place de la manipulation dans la réalisation des tâches de programmation ?

Les programmes en langage S et P peuvent être édités dans n'importe quel éditeur, cependant l'édition textuelle d'une structure générique peut être aidée par deux éditeurs de documents structurés dédiés, générés par le logiciel lui-même. En effet, pour cela il suffit de lier la structure générique de l'éditeur dédié aux programmes en langage S à la syntaxe du langage S, et celle de l'éditeur dédié aux programmes en langage P à la syntaxe du langage P. Ainsi, l'expert sera guidé par l'animation associée aux fonctionnalités de chacun des éditeurs, comme le futur utilisateur de l'éditeur en cours de construction sera aidé par l'éditeur une fois conçu.

Il n'y a pas d'interface spécifique de type programmation par démonstration. On peut imaginer que, dans une telle interface, la structure générique serait décrite à l'aide d'un graphe, et la présentation, par manipulation directe de boîtes (position, silhouette et dimensions).

L'utilisateur ne dispose pas de rendu immédiat de sa programmation dans une vue « résultat » synchronisée avec l'état courant de l'éditeur, sur laquelle s'appuyer, même pour définir la présentation.

3°) Éditeurs de programmes

Quelquefois, les éditeurs de programmes sont intégrés à des environnements de programmation. Leurs usagers sont des informaticiens, puisqu'ils les utilisent justement pour programmer. Ils sont donc familiers des spécificités de la programmation textuelle. Aucun de ces éditeurs ne supporte de programmation visuelle, ni de programmation par démonstration.

a. Éditeurs intimement liés à un langage de programmation

Emacs est originellement l'éditeur intégré des environnements de programmation en Lisp. Programmé lui-même en lisp (souvent spécifique de Emacs), il est possible à l'utilisateur de l'enrichir par ses propres programmes, et de définir des « modes » pour d'autres langages que Lisp.

Fred est une version plus évoluée de Emacs. C'est l'éditeur intégré de l'environnement de programmation en MCL (Macintosh Common Lisp). La seule différence avec Emacs est l'interaction directe avec MCL et l'ajout du « look and feel » du Macintosh. Donc l'utilisateur peut le programmer en MCL.

Xedit est un éditeur programmable en Rexx/Xedit, anciennement en Exec/Xedit. Exec puis Rexx est le langage de scripts de plusieurs systèmes d'exploitation d'IBM (OS/VSE, VM/CMS...). Xedit et Rexx tournent aussi sur PC (Windows et OS/2).

D'une façon générale, tous les langages de scripts permettent de piloter les éditeurs depuis l'extérieur, puisqu'ils permettent de commander toutes les applications en simulant les commandes d'un utilisateur.

vi est l'éditeur de base sous UNIX. La saisie se fait par appel de commandes du type : « ! ligne de commande ». Dans ce cas, l'éditeur n'est pilotable que depuis l'extérieur, comme avec les scripts. Et comme tous les scripts, les commandes peuvent être organisées selon un programme arbitrairement complexe écrit dans le langage des scripts supporté par le système d'exploitation de l'environnement.

b. Éditeurs de programmes seulement liés à une syntaxe

Dans certains environnements, l'éditeur intégré n'est qu'un éditeur syntaxique comme les éditeurs générés par Thot. L'utilisateur peut les paramétrer selon la syntaxe désirée, mais il ne peut commander l'éditeur lui-même à l'aide des programmes qu'il écrit dans l'environnement.

Par exemple, CodeWarrior (CW) permet de programmer en C, en Pascal, en C++ et en JAVA. Il est configurable de façon à guider l'utilisateur par la syntaxe et des notations secondaires (indentation...). Il permet de préparer l'édition de liens et la compilation ainsi que la gestion du projet, et fournit une aide au débogage. Mais l'étape de configuration ne constitue pas réellement une tâche de programmation.

B) Ceux qui ont un rapport direct fort

1°) Logiciels de programmation visuelle

Comme nous l'avons dit dans le paragraphe §I.2.4, les systèmes de programmation visuelle sont basés sur les manipulations d'expressions visuelles pour construire un programme. Il ne s'agit pas seulement d'afficher un programme spécifié par un texte, mais de proposer des outils pour saisir le programme par interaction avec l'interface.

Nous présentons dans ce paragraphe deux environnements caractérisés par des paradigmes de programmation très différents. Le premier, Cocoa, est basé sur un paradigme proche des langages classiques (ou éducatifs, du style Pascal). Il appartient à la même famille que KidSim et Gamut, StageCast étant le nom d'une version plus évoluée. Le deuxième, Forms/3, est issu d'un paradigme spécifique des interfaces Homme-Machine, puisqu'il s'agit d'une généralisation des tableurs. Ces deux environnements proposent un rendu immédiat des programmes en cours, sur des exemples d'appel.

a. Cocoa et Stagecast

Un environnement comme Cocoa permet de définir des scénarios, c'est-à-dire des scènes et leur animation. Ce sont Allen Cypher et David C. Smith qui sont à l'origine de ce logiciel. Cocoa et son successeur « Stagecast » sont développés en Java. Ils offrent une interface épurée qui permet de créer des pièces (et personnages) dans un paysage, et de les programmer pour qu'ils exécutent des tâches.

Le public visé est composé d'enfants. Il s'agit pour l'enfant de concevoir un scénario. La programmation s'effectue à deux niveaux. Au premier niveau, l'utilisateur définit, par manipulation directe des pièces concernées, les règles élémentaires de changement d'état des pièces dans leur voisinage proche.

L'illustration suivante montre la boîte de dialogue qui accompagne la présentation de la scène après la définition de la règle « si clic et rocher, monte d'une case » dans Stagecast.



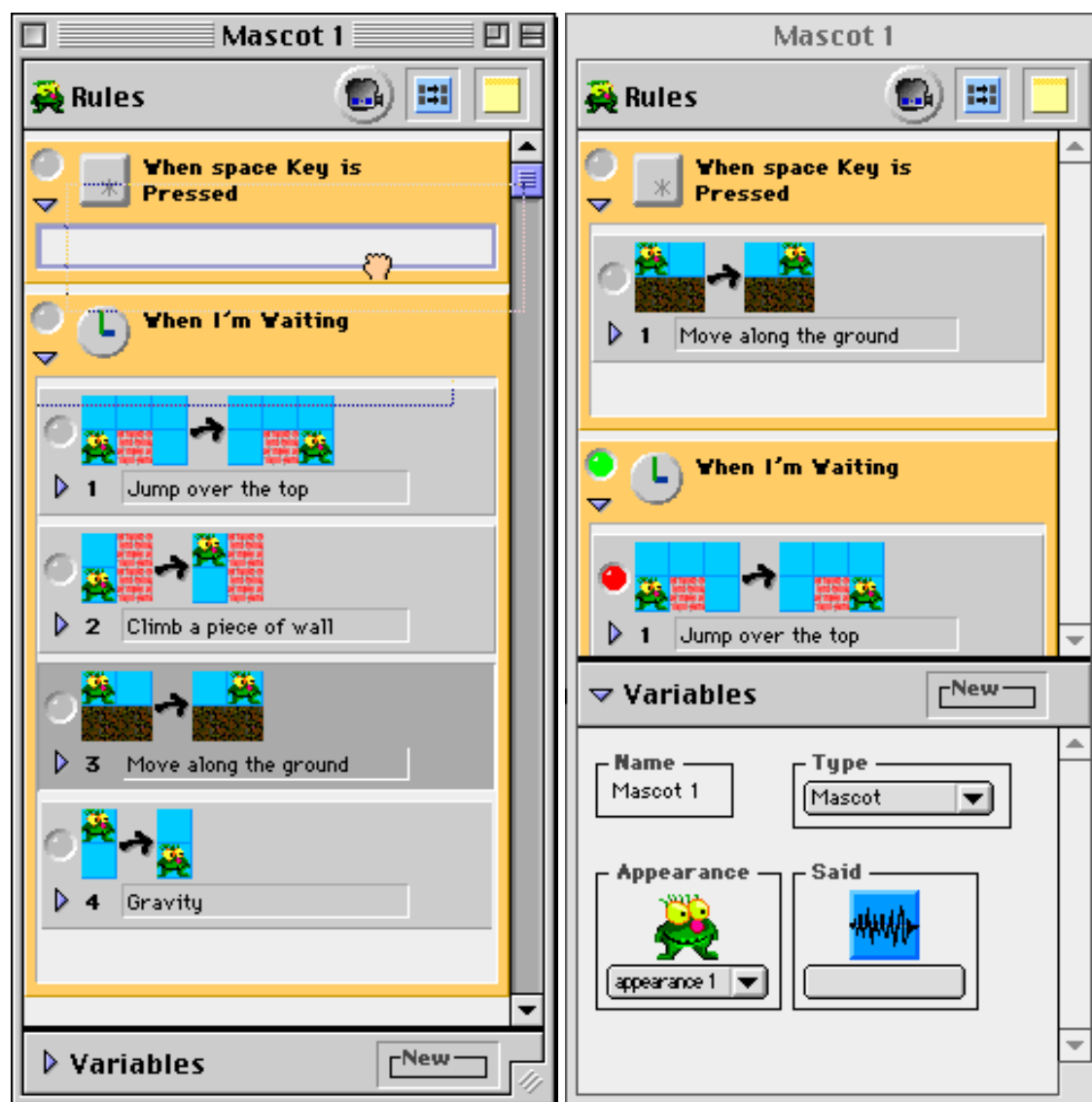
Pour définir une règle, l'utilisateur sélectionne les pièces concernées, le voisinage touché, puis déplace les pièces dans ce contexte. Il peut faire disparaître ou apparaître des représentants des pièces par sélection directe d'un aspirateur qu'il passe sur les pièces non désirées. De même, lorsqu'il veut définitivement effacer une des règles qu'il a définies, il lui suffit de l'aspirer.

Au second niveau, il définit le scénario d'exécution. Il existe deux types de contrôles :

- les contrôles concernant l'articulation générale du programme. L'utilisateur a le choix entre : faire en premier, faire en rond, tirer au hasard pour le faire et passer au suivant, les faire tous, et continuer.
- les instructions conditionnelles. L'utilisateur peut définir des règles conditionnelles à la pression d'une touche du clavier, à des actions sur la souris, ou à des valeurs de variables. La condition encapsule un sous-programme (une « liste » de règles).

Ensuite, il peut réorganiser son programme en déplaçant les règles. Il lui suffit de cliquer sur une règle pour la sélectionner et ensuite la positionner à l'emplacement voulu.

L'illustration suivante montre le déplacement de la règle « Move along the ground » dans l'instruction conditionnelle « When space Key is pressed », dans Cocoa. La fenêtre de gauche montre l'état avant le déplacement, on voit la main prête à « lâcher » la règle sous la condition. La fenêtre de droite montre la nouvelle organisation du programme. Dans cette fenêtre, nous avons déplié la description des variables associées à l'objet « Mascot 1 ».



Les différentes règles peuvent aussi être déplacées et regroupées, ce qui permet de structurer le scénario. La navigation entre les différents niveaux, constitués par les regroupements effectués, est liée aux manipulations du triangle classique à deux positions : la position horizontale pour l'affichage condensé et la position verticale pour l'affichage expansé.

L'utilisateur peut suivre pas à pas l'exécution du programme en observant simultanément l'effet de sa progression sur la liste des règles et dans la scène. L'utilisateur est informé de la règle utilisée par l'allumage d'une « ampoule » verte. L'échec de la tentative d'utilisation d'une règle est annoncé par le passage au rouge de la même ampoule. Pour la mise au point du scénario, chaque règle peut être temporairement désactivée. L'ampoule est alors barrée.

Ainsi, Cocoa permet de définir des programmes et introduit à certaines notions fondamentales de la programmation. Il s'agit par exemple du contrôle du programme, avec les itérations et les instructions de contrôle, de l'utilisation de types et de variables et de la structuration du programme. Le paradigme de programmation est orienté vers les objets de la scène : les procédures sont naturellement associées aux objets dont elles décrivent le comportement. Cocoa offre un affichage multivue synchrone des procédures associées aux objets et de l'exécution du programme.

Dans la vue programme, bien que les différents éléments soient décrits par des illustrations, le programme lui-même est représenté linéairement. Il semble donc que le langage visuel utilisé puisse être mis en correspondance directe avec un langage textuel.

Les avantages de Stagecast

L'illustration suivante montre le texte enregistré pour décrire le scénario. Cette fonctionnalité est introduite dans Stagecast et inexistante dans Cocoa. Certains champs de la description du monde ne sont pas instanciés dans le monde décrit par l'utilisateur. Dans ce cas, le logiciel affiche des commentaires par défaut pour le guider afin qu'il les remplace par des commentaires utiles.

```

new world
My Name
3 avr. 99
What it does (What is the purpose of your world ? If it is a game, what is the goal ?) What to do (What controls
are available to people ? Can they click on anything ? Do any of the keys do anything ?) What to look for
(What do you particularly want people to notice about your world ?) How it works (Describe interesting
characters, rules and variables.) Ideas for changes (Is there anything that you particularly want people to
experiment with or change ?)
2 character types ; 1 more can be created in this trial version
3 rules ; 7 more can be created in this trial version
1 stage ; 1 more can be created in this trial version
Characters
    Bungee, rock
Bungee
    1.faire 2 d'abord

    2.si clic et rocher, descend d'un pas
    And if
    my appearance is Facing Right
    the mouse is clicked
    do
    move Bungee

    3.faire 4 ensuite

    4.si clic et rocher, monte d'un pas
    And if
    my appearance is Facing Right
    the mouse is clicked
    do
    move Bungee

    5.si clic, avance d'un pas
    And if
    my appearance is Facing Right
    the mouse is clicked
    do
    move Bungee
    move Bungee
    move Bungee
    Bungee appearances
    Facing Right, Facing Left
rock
    rock appearances
    left, right
Stages
    stage 2
Jars
    Jar 1
Global
Global variables
    variable 1 : Bungee
    follow me : Bungee
Stagecast Creator Version 1.0
Apple Computer, Inc. Java version 1.1.7, Mac OS 8

```

b. Forms/3

Un environnement comme Forms/3 étend le principe de cellules des tableurs à la définition de formes [Hays & Burnett 95]. Ce logiciel est écrit en Lisp et accessible seulement sous Unix. Son abord nous semble plus difficile pour les utilisateurs novices. Il est présenté ici parce que le paradigme de programmation sous-jacent est particulier, et qu'il est possible d'y définir des types récurifs.

Les formulaires sont les données manipulées par l'utilisateur. Une forme est l'association d'un groupe de cellules et d'une formule. Une cellule est représentée par une boîte modifiable par les manipulations de ses poignées avec la souris. Les représentations des cellules dépendent de leurs attributs graphiques. Une formule est une expression saisie grâce à une combinaison d'entrées de caractères (opérateurs, nombres et noms de fichiers) et de sélections d'autres cellules déjà définies. La valeur résultante de la formule est affichée à l'intérieur de la cellule. Son expression est accessible par une zone sensible située sous la cellule.

Ce formalisme permet de définir des types et même des types récurifs.

La définition graphique d'un type s'appuie sur un affichage (ou « rendu ») particulier lié au paradigme de définition supporté par Forms/3. Les types ainsi définis sont appelés types de données abstraits visuels (Visual Abstract Data Types -VADTs).

Un type abstrait est composé de cellules regroupées à l'intérieur d'une cellule. L'accès à chacun des composants d'un type défini graphiquement se fait par l'appel d'une fonction, qui permet d'utiliser un des composants. Les composants sont ainsi les correspondants des champs des structures de données de Pascal ou C, et des éléments des listes de Lisp.

À l'intérieur de ce type abstrait, les différents champs peuvent être conjugués dans des formules dont les références peuvent éventuellement se croiser.

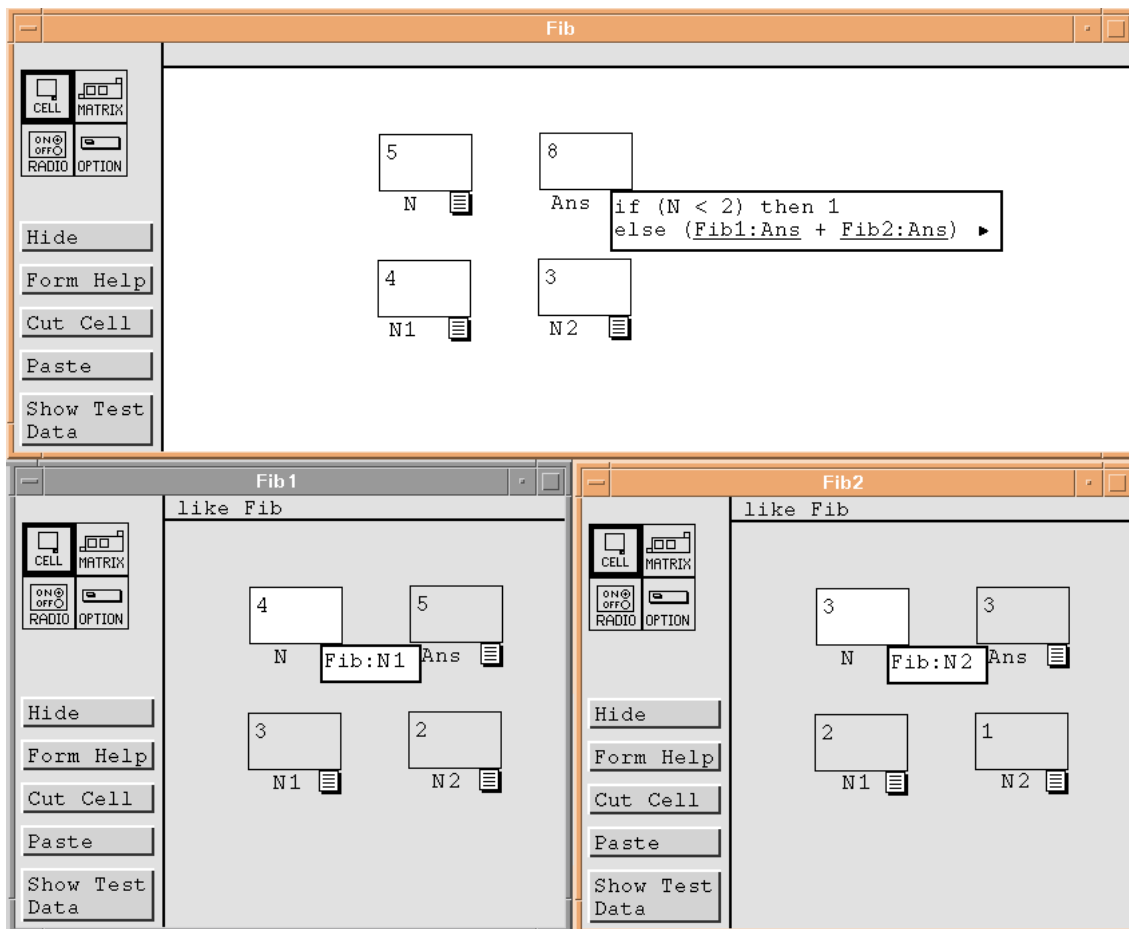
Lorsqu'on définit des formulaires comme des copies d'autres formulaires, ces copies respectent les formules de l'original, mais sont libres pour les valeurs qui ne sont pas définies par des formules. Un formulaire peut être fabriquée pour spécifier concrètement une abstraction réutilisable ; c'est ce principe qui permet de définir des types récurifs.

L'illustration suivante montre un formulaire capable de calculer le Nième nombre de la suite de Fibonacci.

La première fenêtre présente le formulaire nommé Fib. Elle est composée de 4 formulaires. La première composante, dont le nom est N, ne contient pas de formule : elle est « libre ». Deux autres composantes, de noms N1 et N2, contiennent les deux prédécesseurs de N : elles sont définies respectivement par les formules N-1 et N-2. La dernière composante est prévue pour le Nième nombre de la suite de Fibonacci à calculer.

Les deux autres fenêtres présentent deux formulaires nommés Fib1 et Fib2. Ce sont deux copies du formulaire Fib. À l'intérieur de ces deux formes, les quatre composantes sont liées par les mêmes formules que dans le formulaire original : par exemple dans Fib1, N1 contient la valeur du N moins un. Par contre, une nouvelle formule a été définie pour évaluer N : N a la même valeur que le N1 de Fib, le formulaire original. De même une formule a été définie pour évaluer le N de Fib2 avec la même valeur que le N2 de Fib.

La formule de la dernière composante du formulaire de base Fib est alors une formule conditionnelle : Si N est inférieur à 2, c'est 1, sinon c'est la somme des résultats du calcul des formules correspondantes dans les deux formulaires qui sont des copies.



« *Form-based visual programming languages provide a declarative approach to programming, characterized by a dependence-driven, direct-manipulation working model. Users of form-based languages create cells and define formulas for those cells. These formulas reference values contained in other cells and use them in calculations. When a cell's formula is defined, the underlying evaluation engine calculates the cell's value, and those of other affected cells (at least those that are visible to the user) and displays new results. Form-based visual programming languages include, as a subclass, commercial spreadsheet systems* » [Rothermel 98]

Ainsi un formulaire permet à l'utilisateur de mettre ensemble des calculs liés, dans des abstractions réutilisables, répondant aux mêmes motivations que les procédures, les fonctions et les définitions de types dans les autres langages.

Le logiciel ne permet pas d'obtenir une vue textuelle du programme généré, ce qui n'est pas franchement étonnant si c'est proche de Lisp : ce n'est pas vraiment un langage trivial (ni naturel) pour les non-informaticiens !

2°) Logiciels de constructions géométriques type CAO

Les logiciels de conception assistée par ordinateur (CAO) intègrent de la programmation par démonstration (cf. §I.2.B.2.c).

Au niveau du support visuel, deux types de rendu sont proposés selon les systèmes : ceux qui sont basés sur une édition double-vue (programme et rendu-immédiat), et ceux qui proposent un rendu symbolique directement sur le graphique en cours de réalisation.

Nous présentons deux systèmes pour la première approche : Juno-2 et EPB, et Apollonius pour la deuxième approche.

a. Juno-2

Juno-2 est un « experimental, constraint-based, double-view drawing editor. » [Heydon & Nelson 94]

Il est basé sur l'utilisation d'un langage d'extension aussi bien impératif que déclaratif. Ce langage puissant permet de définir des contraintes géométriques à deux dimensions sans être limité ni à ces deux dimensions ni à la géométrie. L'utilisateur peut écrire des programmes en Juno-2 pour définir de nouvelles opérations de dessin (usage impératif) ainsi que de nouvelles contraintes (usage déclaratif).

« Constraints allow you to specify locations in your drawing declaratively. For exemple, to draw an equilateral triangle, you first draw an arbitrary triangle and then constraint its sides to be equal ; Juno-2 will adjust the vertices to make the triangle equilateral. Moreover, the constraints are maintained whenever part of the picture is changed, so constraints make it easier to maintain a picture in the face of modifications. » [Heydon & Nelson 94]

Le logiciel exécute ces programmes pour produire des figures, de la même manière qu'en PostScript : elles sont calculées puis dessinées.

Un éditeur double-vue affiche le résultat en image et simultanément le programme en langage Juno-2 (cf. citation §I.2.C).

Plus précisément, le processus standard pour élaborer une figure dans cet environnement, comporte trois phases : l'esquisse, l'ajout de contraintes puis l'ajustement.

Pour l'esquisse, l'utilisateur fait d'abord appel à l'outil de création, qui ne crée que des points de base, puis il commande le tracé en leur appliquant des fonctions accessibles depuis un menu. Ces fonctions prédéfinies sont par exemple l'affectation d'une position courante, le déplacement en ligne droite depuis la position courante vers une nouvelle position, ou selon une courbe de Bezier guidée par deux points et son « but », la fermeture d'une trace, le remplissage, le changement de couleur du chemin courant, etc. Ces fonctions sont regroupées dans différents modules dont un module PostScript, qui fait appel à de fonctions équivalentes aux fonctions de base du langage PostScript.

Pour la spécification des contraintes, quelques contraintes géométriques sont prédéfinies :

- Pour deux points, l'alignement sur une même horizontale (HOR) ou sur une même verticale (VER) ;
- Pour deux couples de points, l'égalité des distances (CONG) ou le parallélisme des droites portées par chacun des couples.

Toutes les contraintes associées sont coordonnées avec le terme AND.

Pour l'ajustement final, deux outils permettent de déformer la figure tout en respectant sa spécification. Drag permet de déplacer les points directement dans la figure, et Adjust permet de modifier la position d'un point de base, individuellement.

L'exemple suivant est extrait de [Heydon & Nelson 94]. Il montre le programme associé à la construction d'un triangle équilatéral.

```

VAR
  a ~ (179.5, 197.1),
  b ~ (271.2, 223.6),
  c ~ (202.4, 289.8),
IN
  (a, b) CONG (b, c) AND
  (b, c) CONG (c, a) ->
  PS.Moveto(a) ;
  PS.LineTo(b) ;
  PS.LineTo(c) ;
  PS.Close() ;
  PS.Fill() ;
END

```

b. les systèmes Like, EBP et le projet GIPSE

Le système EBP est un système de CAO conçu à l'université de Poitiers.

La conception de ce système a été historiquement basée sur une spécification textuelle. L'évolution vers l'intégration de fonctionnalités de manipulation directe pour le rendre plus abordable aux non-programmeurs est toujours en cours de recherche, comme en témoignent les sujets de thèse.

[Girard 92] a étudié les différentes possibilités offertes aux non-programmeurs pour paramétrer des systèmes de CAO. Il a spécifié les besoins de définition interactive de structures de contrôle (sous-programmes, alternatives et répétitions) et réalisé un prototype, le système Like.

Pour sa part, [Potier 95] a étudié les systèmes qui offrent de la programmation par l'exemple. Sa contribution au système se situe essentiellement dans l'intégration d'interactions graphiques pour définir le programme à travers un exemple d'exécution. Le système EBP constitue une greffe d'environnement de programmation sur un autre système, le système Like. Il effectue une capture des interactions de l'utilisateur par espionnage des commandes et des désignations effectuées. Mais la plupart des actions interrompent l'utilisateur par des questions auxquelles il doit répondre. De telles boîtes de dialogue provoquent une rupture d'engagement direct.

Enfin, [Patry 98] a étudié l'interaction et les dialogues capables d'aider à la désambiguïsation. Dans le projet GIPSE, trois approches sont utilisées pour remplacer les dialogues :

« L'adjonction d'un mécanisme d'écho à plusieurs niveaux permet d'offrir à l'utilisateur une représentation complète de l'état du système, quel que soit le nombre de tâches et sous-tâches entrant en jeu. L'utilisation de la manipulation directe en complément des formes pré-existantes de dialogue structuré autorise l'utilisateur à employer la stratégie de dialogue qui lui semble la plus efficace pour réaliser une tâche donnée. Enfin, la possibilité pour l'utilisateur de créer de nouvelles fonctions de façon simple permet une adaptation du système à ses besoins et habitudes. »

Dans le système résultant, l'affichage textuel du programme reste un support indispensable pour le suivi de la programmation.

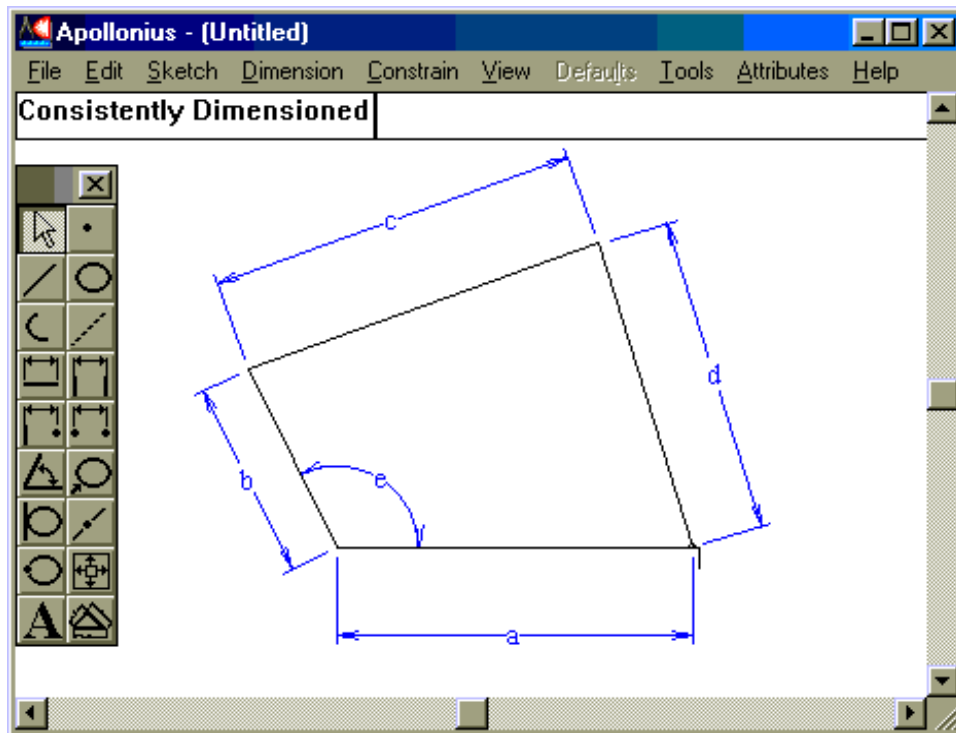
c. Appolonius

Ce logiciel est un analyseur de dessins basé sur des croquis. Les dessins sont conçus comme des esquisses précisées ensuite par l'ajout de contraintes. Les contraintes sont symbolisées directement dans la figure et les objets sont manipulés directement aussi bien pour définir les créations d'objets que les contraintes qui les lient.

L'esquisse est définie par manipulation directe, le mode opératoire étant le mode verbe/nom. Elle ne contient que des points, des segments, des cercles et des arcs, reliés par leurs extrémités. L'esquisse est un ensemble de chemins. Cette esquisse est ensuite contrainte. Les contraintes fournies sont de deux types : soit elles portent sur des valeurs et contraignent donc les dimensions associées à l'esquisse, soit elles portent sur des propriétés.

Les premières constituent les items du Menu « Dimension ». Par exemple il y en a qui fixent la longueur d'un segment ou la distance entre deux points, la distance entre un point et une droite, la mesure d'un angle ou le rayon d'un cercle). Les secondes constituent les items du Menu « Constrain ». Par exemple on peut forcer la tangence d'une ligne à un cercle, l'appartenance d'un point à une droite ou à un cercle, la position d'un point sur l'origine ou l'horizontalité pour une droite.

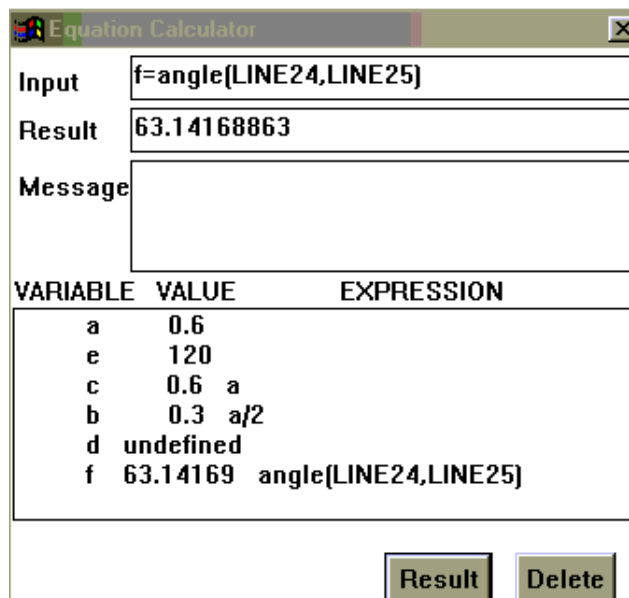
L'illustration suivante présente une copie d'une fenêtre du logiciel.



Au fur et à mesure de l'ajout de contraintes, le logiciel vérifie la consistance du système obtenu, c'est-à-dire que les contraintes demandées sont compatibles entre elles et détermine si la figure possède encore ou non des degrés de liberté. Dans le « bon » cas, il résout le système correspondant et propose une solution en l'affichant.

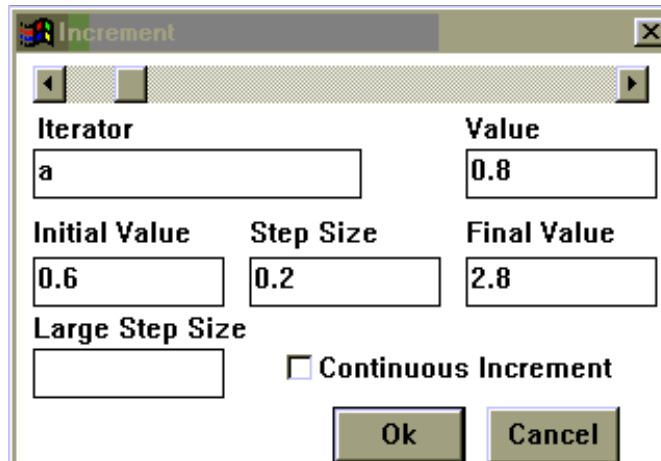
Certaines dimensions peuvent être liées à des variables plutôt qu'à des valeurs et les variables et les valeurs peuvent être utilisées dans des formules.

L'illustration suivante montre la fenêtre qui décrit les variables de la figure précédente, plus une nouvelle variable, non définie comme une dimension (f).



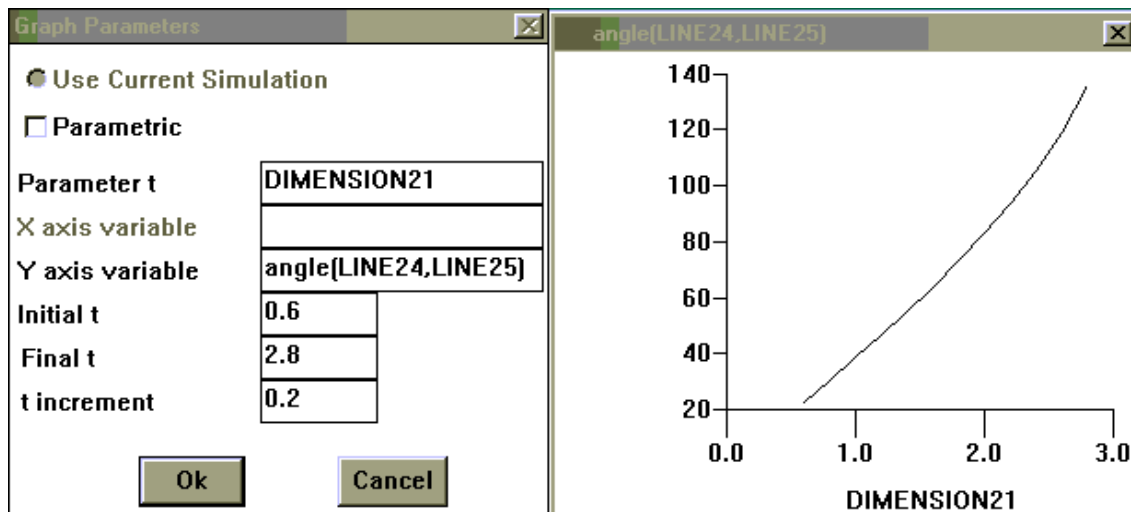
L'utilisateur peut modifier la configuration atteinte de l'esquisse par manipulation directe de ses points, mais alors le système de contraintes n'est plus vérifié. Tout se passe comme si le logiciel imbriquait deux modes de fonctionnement : un pour l'édition de l'esquisse, l'autre pour la résolution des contraintes.

L'animation de la figure « sous contraintes » n'est pas commandée par manipulation directe, mais par l'intermédiaire de boîtes de dialogue. Différents types d'« animations » sont proposés, conduisant à différentes formes de rendu. Un rendu dynamique est piloté par un incrémenteur : associée à un itérateur (variable partant d'une valeur initiale jusqu'à une valeur finale, avec un pas ou en « continu »), une boîte de dialogue permet de visualiser les différentes configurations de l'esquisse qui respectent l'ensemble des contraintes avec la valeur courante de l'itérateur.



Deux rendus statiques produisent soit une table soit un graphe. Il s'agit d'une trace d'une variable de la figure selon les valeurs d'un itérateur. Cette trace est présentée dans les différentes lignes d'une table, ou représentée graphiquement par une fonction qui lie les deux dimensions.

L'illustration suivante montre un tel graphe et la boîte de dialogue qui l'a commandé.



Contrairement à Juno-2, aucun texte du programme n'est accessible. Cependant, le fichier de sauvegarde des figures est lisible par les éditeurs de texte classiques, mais difficilement interprétable. Cette possibilité n'est même pas indiquée dans le manuel d'utilisation du logiciel.

Ce logiciel n'intègre pas non plus de possibilité de définition de sous-programmes. Il n'est possible de structurer la construction qu'en reprenant des éléments de construction déjà définis par ailleurs.

En fait, ce logiciel est orienté vers la simulation et le relevé de mesure.

3°) Logiciels éducatifs constituant des micro-mondes de géométrie

Parmi les logiciels éducatifs qui intègrent de la géométrie dynamique, nous présentons SketchPad, Cinderella et GéoPlan. Ces trois logiciels offrent des fonctionnalités plus évoluées que les autres logiciels du domaine, en ce qui concerne soit l'approche supportée de programmation par démonstration (macros ou scripts), soit l'édition multi-vue dont une vue textuelle.

Pour chaque logiciel, l'attention est focalisée sur les points suivants :

- cadre
- principes de fonctionnement et de manipulation : modes opératoires
- aide à l'utilisation des outils
- outils de programmation
 - accès à la structure logique du programme de construction
 - destruction des objets
 - redéfinition de contraintes
 - nouveaux outils par script ou macro
 - extraction des paramètres
- multi-vues
 - corrélation des animations entre les vues (sélection et déplacement) :
synchronisation des vues
 - équivalence des vues : chaque vue est-elle dynamique, suffisante pour décrire toute la figure ?
- vue textuelle
 - existence d'une sauvegarde textuelle
 - informations fournies et identification des objets
 - proximité du langage avec les manipulations de l'interface
 - respect de la langue et de la culture (symbolique conventionnelle)

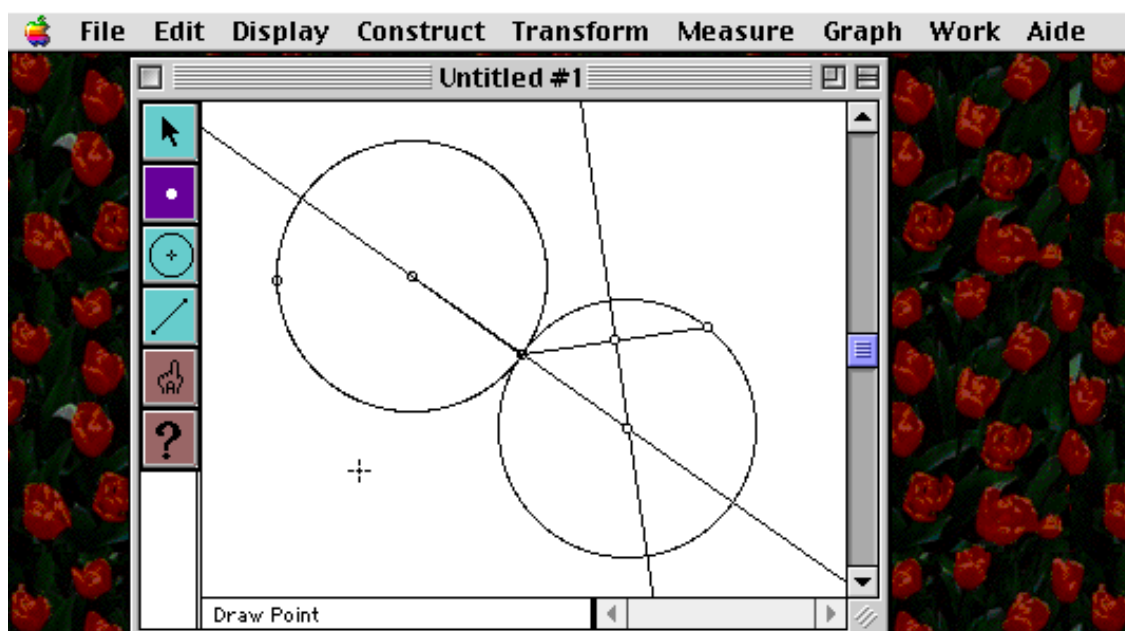
a. Sketchpad

SketchPad constitue un micro-monde d'apprentissage pour la géométrie, comme Cabri. Il fonctionne sur Mac et PC, et les figures sont animables depuis le Web grâce à une « Applet » JAVA.

Principes de fonctionnement

Le mode opératoire est le mode nom/verbe pour tous les outils accessibles depuis la barre de menus système (barre générale, située dans la partie haute de l'écran, en dehors de la fenêtre), et verbe/nom pour les quelques outils associés aux icônes de la boîte à outils (barre de menu située dans la fenêtre).

Les outils de la barre générale s'appliquent sur les objets sélectionnés au préalable. Toutes les constructions possibles pour le jeu d'objets sélectionnés sont créées en une seule action de l'utilisateur.



Les outils situés dans les fenêtres géométriques (boîte à outils) réagissent selon le mode opératoire dual : le choix d'un outil courant est visualisé par l'inversion vidéo de l'icône qui le représente, les créations d'objets correspondantes se font à la volée au fur et à mesure des sélections de l'utilisateur dans la zone géométrique.

Aide

L'aide est une aide en ligne. Il n'y a pas de retour d'information dans la fenêtre graphique, si ce n'est un ajustement de la forme du curseur lui-même. Dans l'illustration précédente, l'aide en ligne est la chaîne « Draw Point » et le curseur est visible sous le premier cercle ; il est représenté par une petite croix verticale. Il s'agit du curseur de construction. Son aspect change lorsqu'il passe au-dessus d'un point : il devient une petite croix oblique entourée d'un rond. Si l'outil était le pointeur, le curseur prendrait différentes formes de flèches. Il n'est jamais accompagné de texte. Le texte correspondant à celui du curseur de Cabri est affiché dans la zone de la fenêtre dédiée à l'aide en ligne.

Accès et manipulation de la structure logique

Il n'y a pas de vue textuelle du programme de construction de la figure en cours. Cependant, sa structure logique est rendue sensible grâce à deux outils particuliers, le sélecteur des parents et le sélecteur des enfants. Ces outils donnent accès à cette structure lorsqu'ils informent ponctuellement sur les liens de parenté entre les objets. Ils n'informent pas sur la nature exacte du lien, mais la plupart du temps l'utilisateur peut la deviner (l'induire).

Il est possible de détruire tout objet sélectionné. Mais la conséquence de cette destruction sur les autres objets n'est pas annoncée explicitement. Tout ce que peut faire l'utilisateur, c'est d'appliquer le sélecteur des enfants jusqu'à ce qu'il ne donne plus de résultat, c'est-à-dire qu'aucun objet ne soit plus sélectionné.

La fonctionnalité de redéfinition de contrainte n'est pas offerte. Toute erreur oblige donc à reprendre les étapes de la construction, au moins depuis cette erreur.

Par contre, il est possible de définir de nouveaux outils, par l'intermédiaire d'enregistrements (Script en anglais). Un script est le résultat de l'enregistrement des commandes utilisées après avoir ouvert une fenêtre spécifique et avoir actionné le bouton de démarrage de l'enregistrement. L'enregistrement s'arrête quand on clique sur le bouton d'arrêt, caractérisé par le dessin de la chaîne « stop ». Il s'agit donc d'une fonctionnalité modale.

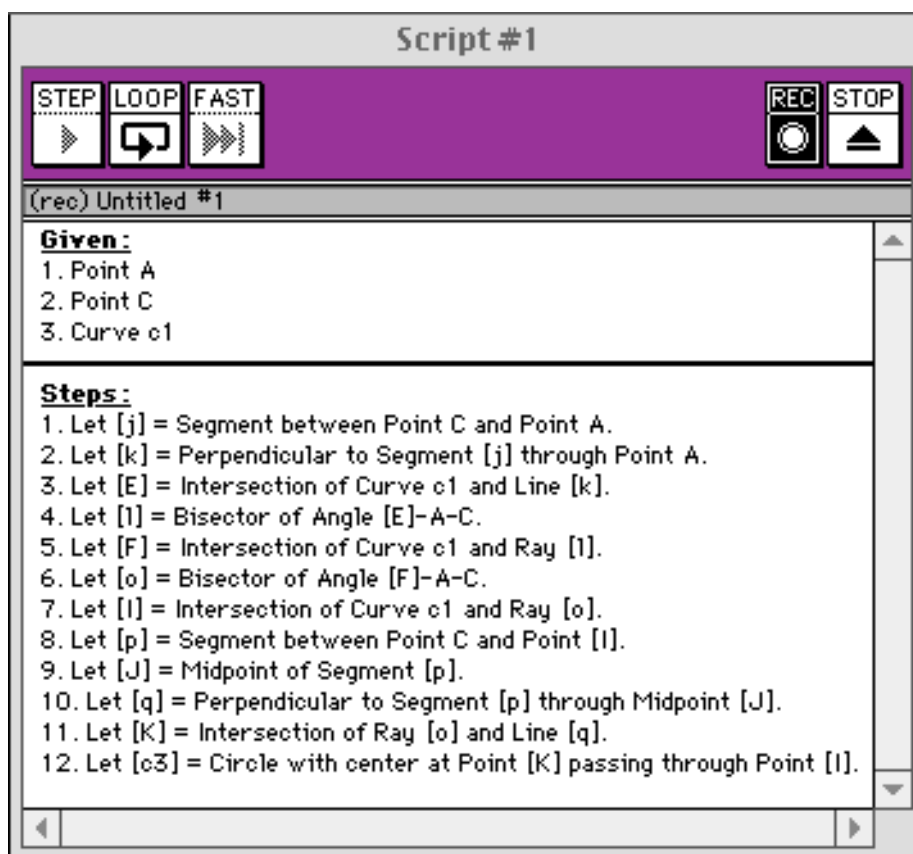
Vue textuelle du script

L'enregistrement d'un script est accompagné d'une fenêtre textuelle qui informe l'utilisateur de son contenu courant. Lors de l'enregistrement d'un script, l'utilisateur peut revenir en arrière plusieurs fois grâce au défaire/refaire. La dernière action est alors annulée et s'il s'agit d'une construction, son enregistrement est effacé. Par contre, s'il détruit un objet, la destruction n'est pas enregistrée.

D'autre part, le défaire/refaire efface la dernière construction mémorisée. Ainsi, en cas d'erreur importante, l'utilisateur doit recommencer tout l'enregistrement. Il ne peut éditer le texte par saisie directe au clavier. La vue textuelle n'est pas dynamique, c'est-à-dire modifiable directement pas les manipulations des éléments du texte par l'utilisateur. Elle est seulement synchronisée avec la vue du résultat.

L'illustration suivante montre une fenêtre de script en cours d'enregistrement. Cette fenêtre est constituée de trois zones, et son contenu dépend de son état : « en cours d'enregistrement » ou « prêt à servir ».

État : « en cours d'enregistrement »



La première zone contient les boutons de début et de fin d'enregistrement « Rec » et « Stop », deux boutons désactivés, et un bouton appelé « Loop ». C'est grâce à ce bouton que les scripts peuvent s'appeler eux-mêmes. Ainsi, les scripts des SketchPad peuvent s'auto-référencer et permettent de définir des scripts récursifs.

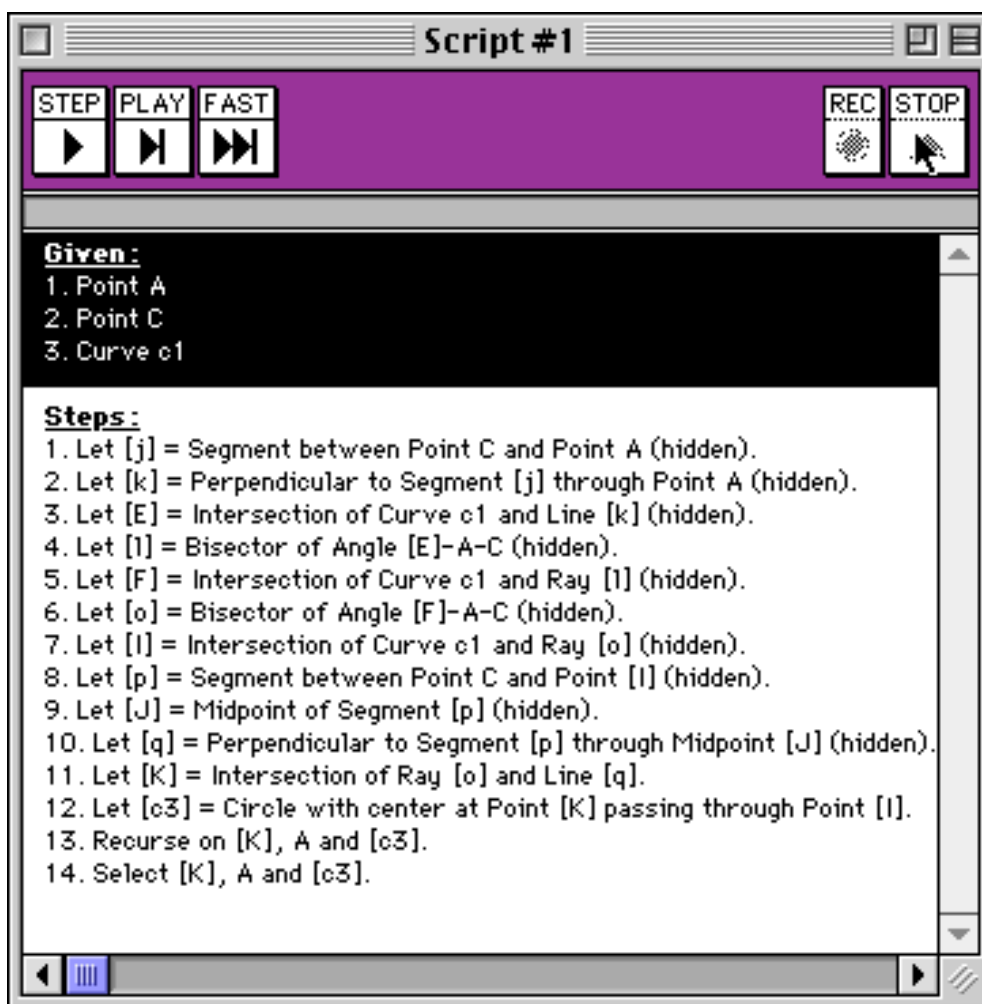
La deuxième zone de cette fenêtre présente les objets initiaux. L'utilisateur est informé de la destination de cette zone par son titre, affiché sur sa première ligne : « Given ». Les objets initiaux sont extraits automatiquement de l'enregistrement en cours de création : dès que la construction d'un nouvel objet dépend de l'existence d'un autre, dont la création n'a pas été enregistrée, ce dernier est ajouté. Les objets initiaux constituent la donnée du script. Pour appliquer ce script, il suffira à l'utilisateur d'avoir sélectionné suffisamment d'objets pour chaque type d'objets de la donnée.

Les actions enregistrées apparaissent dans la troisième zone. Sa première ligne est son titre et informe l'utilisateur que ce qui suit est la liste des étapes du script : « Steps ». Toutes les constructions d'objets sont introduites par le terme « Let », qui définit le nom utilisé pour cet objet, ainsi que les contraintes qui le régissent. Tous les objets dont les constructions sont enregistrées dans cette liste sont des objets finals. L'utilisateur peut décider de cacher des objets avant de mettre un terme à l'enregistrement.

Sur l'exemple d'appel, les correspondants de ces objets seront visibles pendant la phase de construction et cachés dès la fin de l'application du script. Les appels récursifs sont enregistrés par une ligne introduite par le terme « Recurse on ... ». Ils requièrent les mêmes paramètres que l'appel du script lui-même, c'est-à-dire au minimum des paramètres capables de « réaliser » la donnée.

Si des objets sont sélectionnés à la fin de l'enregistrement, leurs correspondants seront aussi sélectionnés après l'exécution du script. L'enregistrement de cette commande est caractérisée par le terme « Select ».

État : « prêt à servir »



Dans la première zone, les boutons de début et de fin d'enregistrement sont désactivés, le bouton « loop » a disparu et à sa place figure le bouton d'exécution ralentie. Les deux autres boutons commandent l'exécution pas à pas et l'exécution complète du script.

La deuxième zone a le même contenu que dans l'état précédent, mais elle est affichée en inversion vidéo.

Dans la troisième vue, l'état éventuellement caché des objets est précisé (« hidden ») et la ligne « Select » termine l'enregistrement.

C'est lors de l'utilisation des scripts récursifs que sont précisées les profondeurs d'appel des récursions (dans l'arbre des appels). À cette occasion, le logiciel précise le nombre d'objets que cet appel générera.

Le langage

Le langage affiché est constitué de la liste numérotée des actions enregistrées, une ligne étant réservée pour chaque action. Dans la phase d'enregistrement, seules les actions de construction sont mémorisées, à l'appel récursif près. Les lignes correspondant aux constructions suivent une forme spécifique, différente des autres outils. Une telle ligne est décomposée en deux parties :

La première, avant le signe « = », définit le nom utilisé pour l'objet dont la construction est décrite par cette ligne.

La deuxième décrit la construction, c'est-à-dire le lien entre cet objet et l'objet auquel il est lié. Cette description est proche du langage naturel avec une composition des mots-clés visibles par l'utilisateur dans les menus déroulants, des prépositions, conjonctions, et des noms d'objets. Par morceaux, ces compositions sont proches de l'aide en ligne.

Pour les autres commandes, essentiellement les commandes de récursion et de sélection, les premiers termes sont suivis de la liste des objets concernés, séparés par des virgules et la conjonction « and ».

En dehors de l'appel récursif, le langage de construction ne contient pas d'instructions de contrôle, et en particulier pas d'instructions conditionnelles. L'arrêt des appels récursifs est fixé au premier appel, par la profondeur choisie au travers d'une boîte de dialogue. L'utilisateur est alors informé du nombre d'objets géométriques correspondant à son choix de profondeur d'appel.

Désignation des objets

Les noms des objets du script sont imposés par le logiciel, pour tous les objets « enregistrés ». Les noms des objets initiaux ne contiennent que des lettres de l'alphabet, en majuscules, affectées dans l'ordre alphabétique et doublées au-delà de 26 noms (etc.). Les noms des autres objets sont en minuscules, encadrés des caractères « [» et «] », et l'usage de chaque lettre est numérotée.

L'adaptabilité à la culture

Les points sont représentés par de petits disques évidés, selon l'usage américain. Contrairement aux autres types d'objets, leur aspect n'est pas modifiable.

La seule version est anglaise, aussi tous les textes prennent leurs termes en anglais. Il n'y a donc pas d'adaptation à différentes cultures langagières.

Divers

Une utilisation annexe des scripts est l'enregistrement de toute une session de travail. Cependant, comme ce script est un simple enregistrement, il n'est pas représentatif de l'état réel. On doit effectuer une correction mentale pour répercuter les effets des destructions.

b. Cinderella

Cinderella est écrit en Java, et donc directement intégrable dans des pages Web. Il constitue un micro-monde d'apprentissage pour la géométrie, comme Cabri et Sketchpad.

Principes de fonctionnement

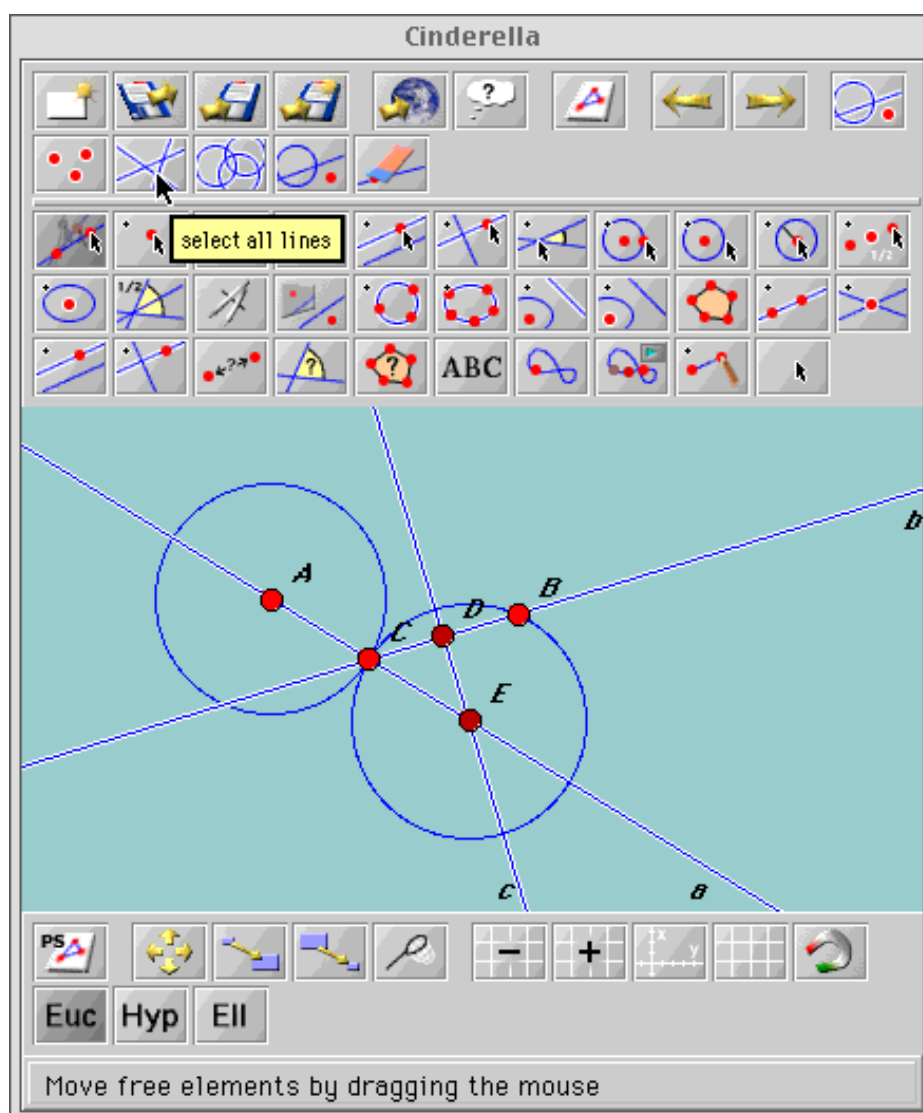
Le mode opératoire est le mode verbe/nom.

Tous les outils sont représentés dans les barres d'icônes, mais ils sont aussi accessibles depuis la barre de menu générale (système). Dans cette situation, ils sont regroupés sous le menu « mode » et leur sélection provoque une mise à jour de l'aide en ligne.

La fenêtre « figure » est scindée en quatre zones. La première, située dans la partie haute de la fenêtre, regroupe les outils de communication avec le système (enregistrer, imprimer, etc.), les outils de déplacement dans la structure logique de la construction (refaire/défaire), les outils de sélection globale (tout sélectionner, sélectionner toutes les droites, tout désélectionner, etc.), l'outil de destruction. Le refaire/défaire permet de revenir pas à pas jusqu'au début de la construction.

La seconde zone située en dessous de la précédente mais au-dessus de la zone graphique, regroupe les autres outils de construction et les outils de sélection unitaire. Les outils de sélection pour déplacer sont distincts des outils de destruction ; ils figurent respectivement en premier et en dernier.

L'illustration suivante montre un exemple de fenêtre Cinderella.



La zone suivante est une zone graphique ; elle contient la représentation de la configuration courante de la figure en cours de construction.

L'avant-dernière zone est située dans la partie basse de la fenêtre. Elle regroupe des outils de modification de la vue graphique : déplacement de la feuille vue à travers la fenêtre, zoom, affichage du repère, d'un quadrillage, d'échelle (bien que cet outil n'influe pas sur les valeurs des mesures), choix du repère (euclidien, hyperbolique ou elliptique).

La dernière zone contient l'aide en ligne.

Aide

L'aide est une aide en ligne, et un retour d'information apparaît sous le curseur sous la forme d'une chaîne pour avertir l'utilisateur de l'effet de l'utilisation de l'outil survolé, mais cela au moment du survol de l'outil. Il n'y a pas de retour d'information sous le curseur dans la zone graphique pour aider l'utilisateur pendant l'utilisation de l'outil, contrairement à Cabri.

Accès et manipulation de la structure logique


Nous avons vu l'existence des boutons défaire/refaire qui permettent de revoir la construction, ainsi que toutes les actions (déplacement, ...). Il s'agit d'un accès indirect à la structure logique du programme. Cinderella fournit de plus un accès direct, puisque la structure logique de la construction peut être exhibée sous une forme textuelle.

Il n'est pas possible de définir de nouveaux outils. Le logiciel ne supporte ni le concept de macro ni celui de script. Le logiciel ne fournit pas non plus d'outil « calculatrice ». La fonctionnalité de redéfinition de contrainte n'est pas offerte.

Vue textuelle

Cette forme est constituée de quatre colonnes. La première représente par un icône le type de l'objet de la ligne. L'illustration suivante montre la fenêtre textuelle qui décrit les figures de l'illustration précédente, dans laquelle les trois droites (a, b et c) sont sélectionnées.

	Who?	What?	Where?
●	A	Point(-2,91 3,31)	(-2,91 3,31)
●	B	Point(1,75 3,03)	(1,75 3,03)
○	C0	Circle(A;2,18)	$(x + 2,91)? + (y - 3,31)? = 2,18?$
●	C	PointOn(C0;-31,53°)	(-1,05 2,17)
/	a	Join(A;C)	$y = -,61x + 1,52$
/	b	Join(C;B)	$y = ,3x + 2,5$
●	D	Mid(C;B)	(,35 2,6)
/	c	Perpendicular(b;D)	$y = -3,26x + 3,75$
●	E	Meet(a;c)	(,84 1,01)
○	C1	Circle(E;C)	$(x - ,84)? + (y - 1,01)? = 2,22?$

PS 

Move free elements by dragging the mouse

La seconde colonne sert à identifier cet objet et affiche son nom dans la figure. Les noms des objets sont choisis par le logiciel et ne sont pas modifiables.

La troisième contient une description des relations avec les autres éléments qui contraignent l'objet ou des valeurs pour les degrés de liberté restants.

La dernière colonne affiche une représentation analytique de l'objet : coordonnées pour les points, équation affine pour les droites, équation pour les cercles, etc.

Dans les équations des cercles, les puissances « carré » sont symbolisées par des points d'interrogation. D'autres formats sont utilisables pour chaque type d'objet.

Le repère est fixé (la position de son origine et ses unités), mais trois types de repère sont proposés : euclidien, hyperbolique ou elliptique. On peut considérer que le repère et l'échelle sont des éléments implicites, vis à vis desquels les valeurs des degrés de liberté rendent explicites les contraintes.

La corrélation de la vue textuelle avec la vue géométrique est rendue sensible par la synchronisation des répercussions des utilisations des outils. La sélection des objets est signifiée par l'inversion vidéo. Les mêmes objets sont représentés dans leur état « sélectionné » dans les deux illustrations précédentes.

Cet effet est actif en cours de construction et synchronisé avec les créations et les destructions d'objets. Mais Cinderella ne prévient pas l'utilisateur des objets qui seront détruits consécutivement à la destruction des objets sélectionnés : seuls les objets sélectionnés sont mis en relief, les objets qui en dépendent ne le sont pas. Le déplacement des objets de base est répercuté sur les valeurs de la vue textuelle. Cette animation rend sensible la relation de dépendance entre les autres objets et l'objet déplacé.

En dehors de ces fonctionnalités de synchronisation, la vue textuelle est presque complètement statique : à part pour l'outil « Intersection », la seule qualité dynamique de la vue textuelle est d'ajouter un echo des commandes définies à travers la manipulation active d'une vue graphique de la figure. Elle ne possède pas de qualité dynamique liée aux manipulations directes de l'interface : même lorsque la fenêtre textuelle est la fenêtre active, les déplacements et les manipulations de la souris ne provoquent pas de réaction du curseur. Il n'est possible de sélectionner un objet depuis la fenêtre textuelle que pour l'outil « Intersection ». Une fois l'outil « Intersection » sélectionné au travers de la fenêtre graphique, l'utilisateur peut sélectionner les objets concernés dans la fenêtre textuelle. Le point d'intersection résultant est construit dans les deux fenêtres.

La vue textuelle n'est pas non plus enregistrable en dehors d'une forme PostScript, adaptée uniquement à l'impression. Les seules manipulations textuelles supportées sont la saisie et la destruction de caractères. Le déplacement global d'un bloc sélectionné ou sa duplication ne sont pas supportés, et encore moins la recherche automatique d'un nom.

Cette vue peut servir à poursuivre la construction, mais ne peut pas servir de support pour des tâches de réorganisation structurelle du programme de construction. En fait, elle constitue seulement un écho textuel de la vue géométrique. Lorsque l'utilisateur ferme la fenêtre géométrique, il peut ajouter de nouveaux éléments, en sélectionner, les positionner différemment, mais les manipulations de la forme textuelle restent limitées.

Le langage

Le langage induit par les manipulations de l'interface a une forme fonctionnelle, c'est-à-dire que chaque nouvel objet est le résultat de l'application d'une fonction de construction à des objets géométriques qui constituent ses arguments. Le nom de la fonction est le nom de l'outil utilisé pour la définition de l'objet. Mais c'est le seul endroit où ces noms sont accessibles : ce n'est pas le même nom que celui qui apparaît dans le menu « mode ». Par exemple, la droite a été définie par l'icône qui raccourcit l'appel par le menu Mode, et qui comprend le choix dans le premier menu de l'item « Line », et dans le menu alors ouvert en cascade de l'item « By two points ».

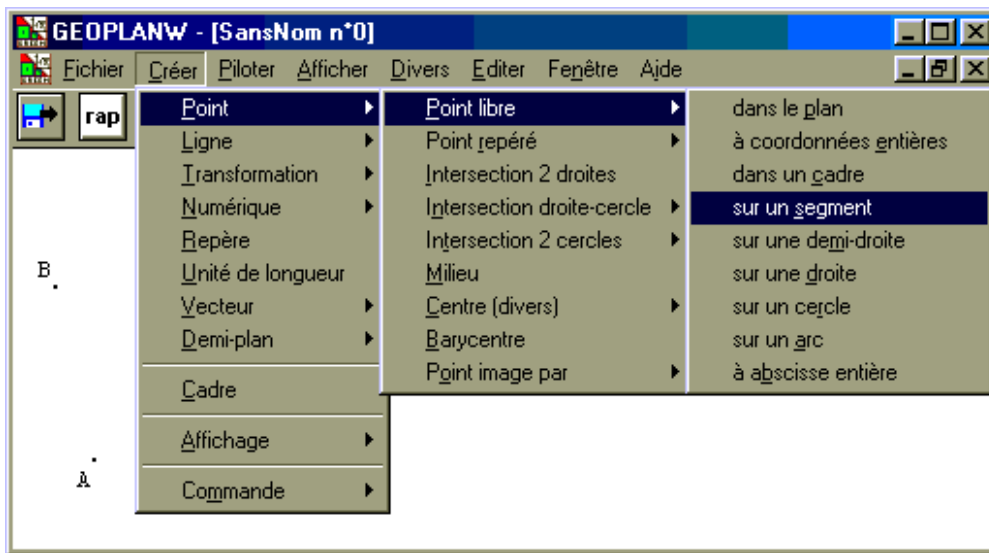
Multi-vues

D'autres vues de la figure sont utilisables. Parmi ces vues, le logiciel fournit les vues hyperbolique et sphérique. Ces deux vues sont dynamiques puisqu'elles peuvent être utilisées indifféremment pour poursuivre une construction. Mais certains objets sont impossibles à saisir dans ces vues du fait même de leur nature géométrique. Toutes les vues sont synchronisées.

c. GéoPlan

GéoPlan est un logiciel de géométrie dynamique basé sur l'intégration d'artefacts manipulateurs sur un interpréteur de commandes. Il fonctionne sur PC, sous Windows.

L'illustration suivante montre une fenêtre de l'application lors du choix d'un outil.

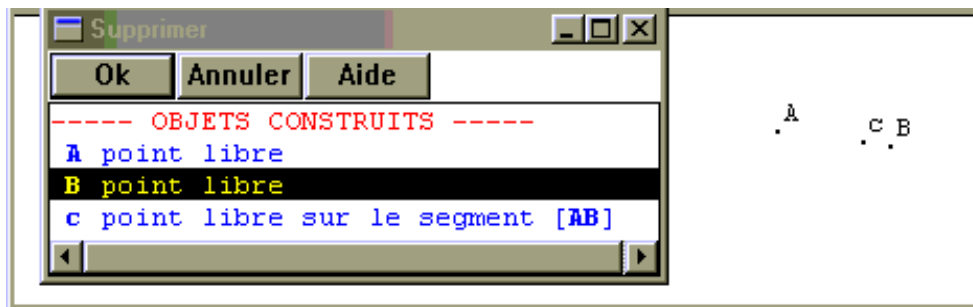


Principes de fonctionnement

Le mode opératoire est le mode verbe/nom, on peut même dire que le mode opératoire est modal. En effet, d'une part, les outils ne supportent pas de polymorphisme, aussi toutes les variantes d'appel forcent l'utilisateur à choisir explicitement l'outil adapté. Ce choix s'effectue à travers une cascade de menus. D'autre part, le choix d'un outil conduit à l'ouverture d'une boîte de dialogue (qui provoque une rupture d'engagement direct). Chaque boîte de dialogue, spécifique de l'outil choisi, attend le remplissage de ses différents champs, et ne disparaît qu'après un abandon ou une acceptation. Les différents champs peuvent être saisis au clavier ou désignés dans la figure. Dans ce dernier cas, c'est le logiciel qui se charge d'insérer les noms des objets désignés dans le champ actif de la boîte de dialogue à remplir.

Par exemple pour la création d'un point dans le plan, l'utilisateur doit d'abord choisir un nom, et le logiciel génère un point dans le plan en tirant au hasard sa position dans l'espace visible par l'utilisateur, puis l'affiche ainsi que son nom (étiquette).

Pour la destruction d'un objet, la boîte de dialogue affiche la liste des objets de la figure. L'utilisateur sélectionne les lignes décrivant les objets qu'il désire supprimer, puis actionne le bouton « OK ».



Aide

L'aide fournie est liée principalement au guide « serré » du dialogue, et aucun retour d'information préventif n'est produit. Par exemple, en dehors de l'activation de tout outil, le curseur est une flèche orientée vers le haut à gauche. Son apparence reste identique lorsqu'il passe au-dessus d'objets, qu'ils soient animables ou non. L'utilisateur ne peut le savoir qu'en essayant de cliquer sur la souris pendant son passage aux environs de l'objet.

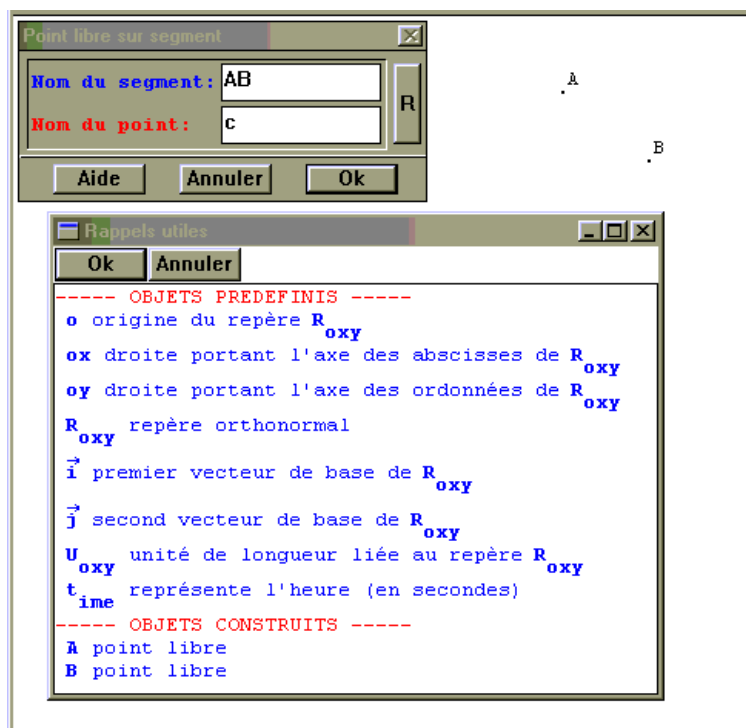
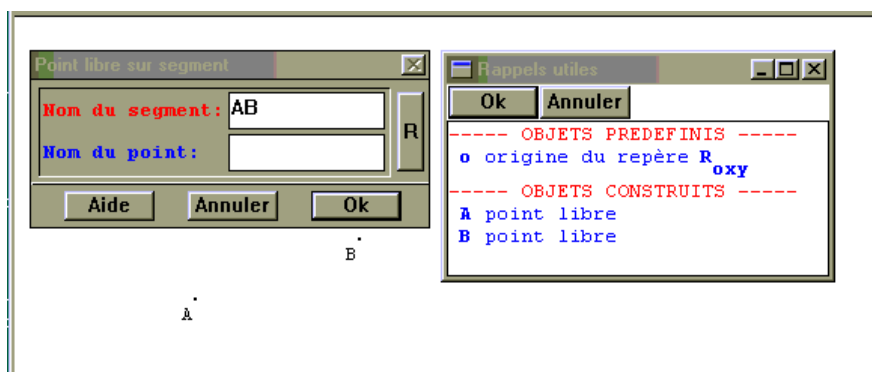
Alors seulement le curseur change : il devient une main qui semble saisir l'objet désigné et le déplacer.

Accès et manipulation de la structure logique

Le défaire/refaire n'est pas infini, contrairement à Sketchpad et Cinderella, mais il ne permet de revenir que d'un pas, puis de revenir sur ce pas, justement. Mais plusieurs autres possibilités sont offertes pour revoir les contraintes qui lient les objets.

Des informations sur les objets sont accessibles depuis les boîtes de dialogue associées aux outils (bouton R) ou avec la touche « Alt », par sélection souris dans la fenêtre graphique.

Les deux illustrations suivantes montrent le contenu de la fenêtre graphique pendant l'utilisation de l'outil **Point libre sur un segment**. Le contenu de la fenêtre des rappels utiles change entre la désignation de paramètres et le choix du nom de l'objet à construire.



Ensuite, il y a l'outil **Historique** qui permet de revoir une à une chacune des commandes utilisées dans la figure. La progression dans la liste des commande est liée à leur re-exécution dans la fenêtre graphique. Le texte affiché pour chaque commande est le même que celui de l'information sur chaque objet.

De plus, le programme de construction est accessible depuis une vue textuelle.

Vue textuelle

Le texte est éditable au clavier dans une fenêtre textuelle. Lorsqu'il est saisi sans faute, il peut être exécuté et le résultat de son exécution remplace la figure précédente.

Dans l'autre sens, la figure n'est pas éditable quand l'édition textuelle est active : comme on peut le voir dans la copie d'écran suivante, tous les boutons de construction sont désactivés.



En plus de la saisie au clavier, les menus de l'éditeur contiennent les outils systèmes minimaux, comme l'enregistrement et l'impression, les classiques couper/copier/coller, ainsi que des outils de recherche et de remplacement.

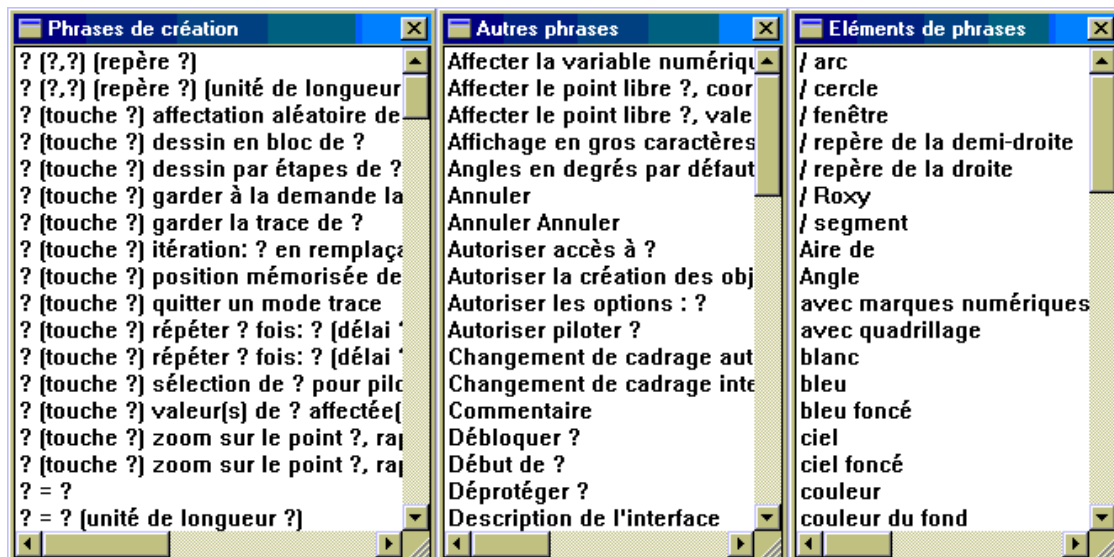
Ainsi l'éditeur textuel, sans être un éditeur syntaxique, est doté de fonctionnalités adaptée à la saisie.

Il ne fournit cependant qu'une vue statique du programme de construction.

Cette vue n'est pas synchronisée, mais est autonome.

Le langage

Le langage est basé sur un vocabulaire et une syntaxe français. Pour le rendre plus abordable par des non-programmeurs, des articles sont introduits entre les mots. En fait, les unités de langage sont des expressions de quelques mots. De plus une aide permet de récupérer la liste de chaîne de caractères reconnues. L'utilisateur peut copier/coller celles qui lui sont utiles, ce qui minimise la saisie et limite les fautes de frappe.



Après l'en-tête, la description de la figure commence. Elle est constituée de la liste des objets construits, avec leur identification (nom et type et description formelle de leur méthode de construction), et la liste des valeurs effectives de leurs paramètres (analytiques : coordonnées des points...).

C) Bilan sur la programmation dans les micro-mondes de géométrie

a. Résumé

Nous avons vu dans le premier chapitre la nécessité d'un accès à la structure logique des programmes de construction dans Cabri. Dans le second chapitre, nous avons étudié la question du support de cette information. Dans le présent chapitre, nous avons ensuite montré plus précisément comment les supports aidaient l'activité de programmation, en nous intéressant plus particulièrement aux manipulations permises dans les différentes vues et au langage utilisé pour décrire les programmes.

GéoPlan permet une édition textuelle avec un éditeur spécifique doté de fonctionnalités propres à la manipulation de textes. Mais cet éditeur n'est pas étroitement lié à l'éditeur graphique.

Cinderella propose plusieurs vues synchronisées et dynamiques, dont une vue textuelle. Mais d'une part, il ne permet pas de structurer les programmes de construction puisque le logiciel n'intègre aucun concept hiérarchique (sous-programme, procédure, macro, script) et, d'autre part, le langage met sur le même plan les différents niveaux d'abstraction, de la géométrie dynamique à la géométrie analytique (coordonnées des points).

Enfin, Sketchpad propose une vue textuelle de ses enregistrements, mais pas de toute la construction. De plus le concept choisi ne peut pas intégrer la hiérarchisation.

Donc parmi les logiciels proches de Cabri, il n'y a rien de comparable avec les possibilités de programmation de Cabri, en particulier aucun ne permet de structuration hiérarchique des programmes. Mais chacun apporte des « idées » intéressantes.

Les logiciels des domaines connexes permettent de compléter la liste des « nécessités ».

Les autres logiciels qui permettent des activités de programmation sans être dédiés à son enseignement, montrent l'intérêt d'une vue textuelle comme support d'information sur la structure logique des programmes. Cette vue doit être liée à l'exécution du programme.

La proximité entre le monde du problème et le monde du programme est réalisée dans le domaine de la géométrie. L'usage d'un support graphique dynamique à la Cabri donne un accès immédiat aux effets du programme sur le modèle : la structure du programme n'est pas visualisée explicitement, mais son interprétation est exécutée en temps réel. Une telle démarche est envisageable dans le domaine de la géométrie, et dans quelques autres domaines accessibles par Cabri - voir §I.1-, ou encore lorsque les objets du domaine sont représentables dans leur état vis-à-vis des actions de l'utilisateur.

Les logiciels dédiés à l'enseignement de la géométrie montrent la nécessité de proposer à l'utilisateur des moyens pour naviguer entre différents niveaux d'abstraction.

Les niveaux d'abstraction sont liés aux possibilités de hiérarchisation : même si, dans Cabri, les macros ne sont pas réellement des procédures, il est tout de même possible de structurer la construction des figures, et même le processus de construction de nouvelles macros, comme si elles pouvaient s'imbriquer. Comme l'utilisateur n'a pas accès explicitement au contenu des macros, mais peut décider de définir une macro a posteriori, cette encapsulation est implicite. Si dans les faits, le logiciel, lui, ne conserve pas de trace de l'encapsulation, mais seulement des constructions extraites, l'utilisateur, n'ayant pas accès au contenu effectif de ses macros, peut imaginer un modèle de fonctionnement du logiciel dans lequel les macros sont imbriquées. Si le comportement des macros est cohérent, son modèle n'est pas mis en défaut.

Cependant il serait préférable de donner les moyens à l'utilisateur de se construire un modèle réaliste, c'est-à-dire en accord avec la réalité. Deux possibilités s'offrent pour Cabri : soit gérer des macros dont la structure hiérarchique est effective à tous les niveaux (plus de remplacement de l'appel par l'extrait utile de son contenu), soit informer continuellement les utilisateurs du contenu effectif des macros.

Un dernier aspect, qui permet de rester fidèle aux principes mis en œuvre dans Cabri, est de laisser à l'utilisateur le plus de choix possibles, lui laissant la possibilité d'évoluer entre différents niveaux d'abstraction et supports, et le supposant capable de choisir celui qui lui convient le mieux pour chaque tâche particulière.

En ce qui concerne la navigation entre les différents niveaux d'abstraction, c'est l'utilisateur qui doit pouvoir décider de rester à un niveau formel ou de passer à un niveau analytique, de rester au niveau de la règle et du compas (outils de base) ou de passer à un niveau abstrait (outils ad hoc).

Quant à la forme du programme, sa manipulation doit pouvoir s'appuyer ou non sur l'éditeur structurel, qu'il soit textuel ou graphique, selon ses besoins immédiats.

Il s'agit d'un principe de non-choix pour le programmeur du logiciel, et de tout choix pour son utilisateur.

b. Critères d'évaluation de logiciels de géométrie dynamique

De toute cette analyse, nous tirons maintenant une liste de critères utiles et pertinents pour catégoriser et évaluer les logiciels de géométrie dynamique, orientée vers l'évaluation de l'aide que ces logiciels apportent à leurs utilisateurs pour des activités au caractère programmatoire.

Cette liste est issue de la confrontation de l'étude théorique du chapitre I-2, qui nous a permis de dégager les notions importantes pour les systèmes qui offrent des capacités de programmation à des utilisateurs non-programmeurs, avec l'examen des logiciels des domaines connexes et des logiciels de géométrie.

Cette liste est divisée en deux parties principales : la première concerne les caractéristiques de l'interface, qui sont indépendantes du domaine d'application (la géométrie), et la deuxième permet d'examiner plus précisément les fonctionnalités offertes par les logiciels.

Critères d'interface :

• Ergonomie

• Ergonomie visuelle

Mode de représentation de l'état « sélectionné » (clignotement, poignées, sur-brillance, sous-brillance)

Ubiquité des objets et des outils en multivue

• Ergonomie manipulateur

Modes de saisie (clavier, souris, son) adaptés aux formes des données manipulées

• Accès à la sémantique

• Manipulation directe,

Retour d'information en mode transformation (déformation, déplacement) et en mode construction

Contrôle des transformations (déformation, déplacement...) en temps réel

Mode opératoire adapté

Polymorphisme (pas d'ordre sur les éléments des procédures)

Opérations rapides, incrémentales et réversibles

• Qualité des métaphores (réalité augmentée permettant de tirer parti de la familiarité extérieure des fonctionnalités /versus/ fonctionnement modal),

Prise en compte d'un modèle de conception graphique des icônes

Quels icônes pour les instructions conditionnelles ?

Expressivité des rôles, susceptibilité d'induire en erreur

Nommage automatique des objets

• Prise en compte de la culture de l'utilisateur (langue maternelle, notations et symboles usuels)

• Possibilités de notations secondaires (adaptées en cas de « multivue » : indentations — présentation ajustable et commentaires pour la vue structurelle)

• Difficulté des opérations mentales

• Notion d'objets complexes

• Comportement

• Engagement direct (contre-exemple : dialogue fantôme obligeant à une réponse pour rendre la main),

• Petit nombre ou absence d'engagements prématurés — Green — (mode opératoire adapté et opérations réversibles)

• Consistance et uniformité (c'est-à-dire le même comportement pour les objets),

Polymorphisme (prise en compte équivalente de tous les types d'objets conceptuellement utilisables pour l'outil, gestion de l'ambiguïté)

Anticipation sur les désirs (prise en compte de l'implicite par application d'outils préemptifs : proposition de changement d'outil à la volée)

- Cohérence (c'est-à-dire le même schéma),
- Viscosité (mesure de l'effort pour aboutir à un changement),

Critères liés aux fonctionnalités

• Richesse.

- Redéfinition (cohérence de cet outil, et donc même degré de polymorphisme que les autres outils)
- Modifications d'attributs graphiques
 - Remplissage
 - Accès à l'ordre
 - Choix (couleur, hachurage, transparence)
- Étiquetage
 - Autoritaire /libre /mixte
 - Position de l'étiquette
 - Longueur bornée ou non de la chaîne
- Macros
 - Choix a priori/a posteriori (« engagement prématuré »)
 - Visibilité du contenu effectif (« dépendances cachées perceptibles ou symboliques »)
 - Possibilité de modifier sans tout refaire (« engagement prématuré »)
 - Évaluation incrémentale (dynamique)
 - Possibilité de paramétrisation « globale »
 - Respect de la cohérence pour les outils résultants
 - Possibilité de définir des macros qui respectent le même degré de polymorphisme que les outils de base de l'interface et donc éventuellement plusieurs méthodes pour une même macro
 - Possibilité d'intégrer cette macro dans les outils préemptifs
 - Macros récursives
 - Itération / condition d'arrêt
- Multi-vue
 - Choix euclidien, hyperbolique, textuel...
 - Synchronisation entre les vues
 - Ubiquité des objets
 - Simultanéité des modifications
 - Dynamisme (autonomie possible de chaque vue)
 - Recouvrement, complémentarité et complétude de l'ensemble des vues (« visibilité »)
- Vue textuelle
 - Fichier et /ou fenêtre
 - Figure et macros
 - Caractère diffus ou concis

Structuration possible du « programme » (« gradient d'abstraction »)
 Avertissement des conséquences de destruction d'objets (« engagement prématuré »)

- Pertinence.

- Manipulation du texte dans la vue textuelle
- Arrêt des appels récursifs en géométrie dynamique (profondeur dynamique)

- Qualité de la réalisation.

- Banc d'essais
 - Coût à plate-forme fixe (multi-vue /uni-vue)
 - Nombre d'objets avant borne mémoire
 - Rapport temps de chargement d'une figure

- Qualité du micro-monde réalisé

- Extensibilité
 - Paramétrabilité du micro-monde avec intégration des outils de l'utilisateur
- Macros
 - Répétition de séquences
 - Extraction de constructions.

D) Conclusion

Un environnement supportant des activités programmatoires doit permettre à l'utilisateur de maîtriser deux facettes transversales de l'ensemble des données manipulées : une vue résultat et une vue programme. Par exemple, les objets du domaine doivent être manipulables par leur valeur (instance) et par leur rôle dans l'ensemble des données.

Les vues résultats sont susceptibles de prendre toutes les formes permettant d'appréhender différemment le domaine (en géométrie, représentation euclidienne, hyperbolique, sphérique ; en IHM une horloge peut avoir une représentation classique ou numérique). Cela correspond au module « présentation » du modèle PAC [Coutaz 90]. La structure logique n'est pas explicite.

Les vues programmes sont susceptibles de prendre toutes les formes permettant de représenter toutes les informations qui concernent la structure logique du programme.

Dans le cas de Cabri, trois formes peuvent coexister avec deux qualités. Il s'agit des formes : géométrique, textuelle et graphique, et des qualités : synchronisée et dynamique.

Le principe consistant à laisser la maîtrise des choix à l'utilisateur est un principe fondamental du logiciel Cabri-géomètre. Ainsi, toutes les vues doivent pouvoir être utilisées indépendamment et conjointement.

Le respect de ce principe conduit au problème de la gestion simultanée et autonome des artefacts de manipulation spécifiques à chaque forme de support. En effet, les différents supports portent des vues synchronisées et dynamiques dans lesquelles les formes de manipulation des données sont fondamentalement distinctes, dans leur structure (dessin / texte), comme dans leur relation (structure logique de programme : relations sémantiques entre les données du programme et relations syntaxiques entre les éléments textuels).

Apports potentiels d'une vue textuelle dynamique et synchronisée

Dans le premier chapitre, nous avons montré en quoi certaines activités effectuées par les utilisateurs dans l'environnement de Cabri-géomètre sont proches d'activités de programmation. Nous avons montré que, pour ce type d'activité, l'utilisateur pouvait ressentir le besoin d'intervenir directement dans la structure logique du programme.

Dans le deuxième chapitre, nous avons exposé les formes des différents supports de programmation présents dans les systèmes actuels qui permettent des activités de programmation, ainsi que ce que chaque forme apporte. Indépendamment de la motivation de ces activités de programmation (soit que leur objectif premier soit justement l'enseignement de la programmation, soit que la programmation soit inhérente à l'activité de ses utilisateurs non nécessairement programmeurs, soit encore que, comme dans Cabri et ses concurrents, la programmation soit pertinente dans le contexte d'application du logiciel), l'accès aux structures logiques des programmes est une nécessité.

Le troisième chapitre a été consacré à l'examen des fonctionnalités de différents logiciels particuliers vis-à-vis des activités de programmation, et nous avons conclu qu'aucun des concurrents de Cabri n'offre des possibilités de programmation aussi évoluées que lui. Nous y avons aussi proposé des critères d'évaluation de l'aide à la programmation fournie par l'interface homme-machine des micro-mondes de géométrie.

Dans ce quatrième et dernier chapitre, nous présentons les apports directs attendus de l'introduction de fonctionnalités d'édition de la structure du programme ainsi que les apports à plus long terme, qui nécessiteraient des développements supplémentaires. Ces fonctionnalités apportent des éléments dans deux directions : pour les utilisateurs actuels de Cabri et pour l'enseignement de la programmation.

A) Apports aux utilisateurs de Cabri

1°) Mise au point de grosses figures

Un certain nombre de chercheurs et d'enseignants utilisent Cabri dans des domaines connexes à la géométrie. Il s'agit de l'optique, de la physique, des statistiques,... (cf. tableau §I.1.). Ces utilisations nécessitent souvent des constructions et des mises au point de figures de plusieurs centaines d'objets.

Comme nous l'avons vu dans le chapitre I, une des difficultés liées à la construction de grosses figures est que les objets construits sont éloignés dans le temps : l'utilisateur doit se rappeler les contraintes exactes que chaque objet respecte par construction, ainsi que ses liens avec les objets dont il dépend.

Une vue synchrone de la structure logique du programme de construction, avec ubiquité des objets, permettrait de rétablir la correspondance entre les occurrences de la variable qui représente l'objet géométrique et l'instance de cet objet représenté graphiquement (géométriquement). Ainsi, l'utilisateur posséderait plusieurs modes d'accès pour chacune des données, et pourrait poursuivre la construction indifféremment sur chacune des vues, du moment que la vue est dynamique. L'utilisateur pourrait choisir le mode d'accès qui est le plus immédiat dans le contexte de la situation où il se trouve. Par exemple, dans une vue de la structure logique, chaque objet serait accessible et sélectionnable sans ambiguïté.

Les définitions de macro-constructions permettent de structurer la construction, et de récupérer l'investissement dans la mise au point de certaines de ses parties en les réutilisant dans de nouvelles constructions. La vue structurelle donnerait accès au contenu même des macros.

De plus, si nous la munissons de fonctionnalités de dépliement et de repliement des macros, elle permettra d'obtenir une vue synthétique du programme, et de choisir un niveau d'abstraction adapté à sa compréhension et à sa mise au point. Donc le niveau d'abstraction du programme sera adaptable en pliant ou dépliant les macros définies.

Une vue structurelle permet donc de répondre à chacun des points développés dans le chapitre I-1.

Le choix d'une forme textuelle pour cette vue permet de plus d'éditer la figure depuis n'importe quel support textuel, même sur le papier, et de fixer les étapes (liste ordonnée d'actions) à accomplir pour construire la figure.

2°) Enseignement avec Cabri

a. Préparation des exercices et illustrations de cours

Quand un professeur prépare des exercices, il a un rôle d'utilisateur particulier. Ses impératifs ne sont pas les mêmes que ceux des utilisateurs élèves. Il n'a pas de raison de se poser des questions qui contribueraient à l'aider à assimiler des notions mathématiques, en dehors d'une phase de test de ses exercices. Et pour la phase d'édition du support-figure (ou macro) de l'exercice, il a besoin d'efficacité.

L'accès à la structure logique de la figure y contribuera, ainsi que les possibilités de modifications des macros pour leur mise au point.

b. Evaluation des connaissances et des acquis

L'utilisation d'un logiciel dans l'enseignement conduit au problème du contrôle des connaissances acquises par l'élève et donc à l'évaluation de ses travaux. Pour cela, le professeur peut regarder les figures construites par les élèves. Mais, d'une part, ces figures ne contiennent pas toute l'information sur les tentatives des élèves, et d'autre part l'information concernant les contraintes qui régissent les figures ne sont pas immédiatement accessibles depuis la figure.

L'accès à la structure logique répondra à la deuxième attente, et pour la plupart des tâches d'évaluation du résultat, cela suffit.

Par contre, en ce qui concerne l'évaluation de la stratégie d'exploration du problème par l'élève, la vue structurelle ne répondra pas aussi bien au besoin du professeur que l'historique des actions de l'élève.

La vue servira donc moins à ajuster a posteriori les énoncés des problèmes qu'à vérifier que les contraintes désirées auront bien été respectées par l'élève.

3°) Applications de Cabri à de nouveaux domaines

a. Recherche en mathématiques

En recherche en mathématiques, Cabri et la géométrie dynamique offrent un support de réflexion permettant de deviner, d'imaginer des propriétés. En géométrie non euclidienne (§I-1) ou avec les coniques, par exemple, l'environnement constitué par le micro-monde de Cabri est très apprécié, car il permet d'appréhender des propriétés non perceptibles sans animation.

Les figures construites pour ce type d'activités sont de grosses figures et les chercheurs fabriquent des outils personnels en écrivant des macros très complexes qu'ils réutilisent d'une recherche à une autre.

Contrairement à l'utilisation de Cabri pour l'enseignement pur, les utilisateurs-chercheurs investissent plus de temps dans l'appropriation de l'outil informatique. Ils veulent faire fructifier leur investissement en réutilisant ce qu'ils ont déjà construit. Ils intègrent leurs macro-constructions dans la barre de menus, et la spécialisent selon leurs besoins (barre de géométrie hyperbolique...). Ils travaillent avec le logiciel à un niveau d'abstraction plus élevé que celui des outils de base.

Ils veulent aussi réutiliser et partager des macro-constructions entre chercheurs. Or, sans forme textuelle, le seul moyen pour connaître le contenu effectif des macros est le commentaire prévu par son créateur ou une analyse des effets de son application à des objets. Comme nous l'avons vu dans le chapitre I-1, l'enregistrement textuel est difficilement décriptable et donc réutilisable, même pour des experts. On ne peut donc corriger ou modifier une grosse macro qu'avec un risque important de ne pas aboutir.

Ainsi, les chercheurs en mathématiques utilisateurs de Cabri sont, a priori, gros consommateurs de fonctionnalités spécifiques des environnements de programmation, comme la hiérarchisation. Le besoin d'une vue textuelle, accompagnée d'outils de navigation appropriés, est particulièrement important pour eux.

b. Prototypage graphique

Dans d'autres domaines comme la conception assistée par ordinateur, ou la modélisation en 3 dimensions, le concept des macros de Cabri permet de fabriquer des prototypes d'outils et ainsi d'évaluer la pertinence de différentes approches pour manipuler les objets de ces domaines à travers une interface graphique.

Depuis l'origine, les systèmes de CAO sont essentiellement basés sur la programmation textuelle, et les recherches actuelles tentent d'y intégrer des approches de manipulation directe des objets géométriques (§I-2-C-2°-c, §I-3-B-2°-b).

Le public visé par ces systèmes est constitué d'utilisateurs non-programmeurs, mais les tâches qu'ils ont à accomplir sont souvent répétitives. Elles peuvent être optimisées par la définition de programmes. Aussi, l'approche de la programmation par l'exemple (cf. §I.2.C), et la définition automatique des macros, est bien adaptée à ce domaine. Une des spécificités de ce domaine est le grand nombre d'objets à construire, ce qui conduit à une gestion critique de l'ambiguïté.

Dans ces domaines où la programmation textuelle est première, la programmation directe commande l'édition d'un texte qui, lui, en tant que programme de construction, commande la construction. Le rôle du programme textuel reste prédominant.

L'interface de Cabri-géomètre pourrait être très utile pour ces domaines, en particulier grâce au concept de macro qu'il supporte. La conjugaison de la manipulation directe des objets que Cabri-géomètre fournit, et la possibilité de définir des macro-commandes, permettrait l'exploration et l'évaluation d'idées nouvelles.

Dans un tel contexte, les prototypages en CAO nécessiteraient la mise au point de macros Cabri complexes, difficiles à mettre au point sans accès (textuel) à leur contenu effectif. De plus, l'utilisation synchronisée de la vue textuelle peut constituer un moyen pour remédier aux problèmes liés à l'ambiguïté.

Imaginer un tel apport de Cabri à un domaine comme la CAO nous conduit, réciproquement, à dégager des améliorations possibles de Cabri lui-même. Par exemple, nous voyons l'intérêt de définir des paramètres globaux pour les macros, et d'intégrer de véritables structures de contrôle qui sont indispensables en CAO [Girard 95].

La première version mise en œuvre dans le cadre de ce travail répond à certains de ces désirs, mais pas encore à tous.

B) Apports au domaine de la programmation

1°) Enseignement de la programmation

Il se trouve que l'outil Cabri peut être propice à la découverte et à l'enseignement de la programmation dans un cadre dont l'utilisateur est déjà familier.

a. Où l'usage de Cabri est-il pertinent ?

La programmation passe par l'acquisition des compétences suivantes :

1. Analyse d'un problème et formulation abstraite de la solution,
2. Algorithmisation au niveau d'objets informatiques abstraits (par exemple, les objets géométriques de Cabri),
3. Assimilation des outils algorithmiques (structures de contrôle et de données) du langage et de l'environnement,
4. Savoir-faire en débogage (détection de dysfonctionnement et planification de remédiation, d'ajustement de la stratégie) par passages de raffinements successifs entre les étapes 2 et 3.

D'abord, l'usage de Cabri est pertinent par son domaine d'application, qui est l'enseignement des mathématiques. En effet, un des objectifs fondamentaux de l'enseignement de mathématiques est l'apprentissage de la résolution de problèmes.

Ensuite, dans les activités de construction, la stratégie mise en œuvre procède du même processus que pour toute activité de programmation. Comme nous l'avons montré dans le chapitre I-1, l'utilisateur a une idée (imposée ou libre) du résultat qu'il veut atteindre et il doit concevoir le programme de sa construction.

L'analogie entre formalisation de solution de problèmes et programmation de solution, est classique. L'élaboration d'une démonstration mathématique nécessite le choix d'un plan. Le logiciel, par ses qualités didactiques, contribue à convaincre les élèves qui l'utilisent de l'intérêt des preuves, en donnant du sens à ce type d'activité. Un argument comme "ce n'est pas parce qu'une propriété est apparente sur un dessin qu'elle est vraie dans la figure" conduit à douter de la généralisation de la perception statique au comportement dynamique, et ainsi à motiver l'élaboration d'une preuve des propriétés perceptives.

La stratégie suivie pour élaborer une démonstration consiste à vérifier visuellement les propriétés cherchées, puis à rechercher des contraintes utiles à ces propriétés parmi les constructions imposées par les choix des contraintes, et enfin à rechercher des inférences basées sur des théorèmes connus, à mettre en œuvre à partir des contraintes pour conduire à ces propriétés. Lorsqu'une propriété est un peu éloignée des contraintes de construction de la figure, l'élève doit choisir des étapes et planifier sa démonstration. Ainsi, Cabri contribue à rapprocher le raisonnement et les ressources cognitives (entre la résolution de problèmes et les moyens accessibles par l'élève).

Ainsi, l'analogie classique entre résolution de problème et programmation d'une solution montre que Cabri est très utile pour l'assimilation du point 1, fondamentalement par son domaine d'application, mais aussi plus directement par le type d'activité qu'il permet.

De plus, dans le micro-monde constitué par Cabri, la manipulation directe des objets abstraits et le retour d'information, donné préventivement pour guider l'utilisation des outils, minimise la distance entre les points 2 (manipulation d'objets informatiques abstraits) et 3 (assimilation d'outils algorithmiques). Cette distance n'est donc ni trop grande ni trop petite pour favoriser l'acquisition de compétences en débogage (étape 4). Cabri-géomètre fournit un support pour tester et déboguer, dans lequel les problèmes rencontrés par les utilisateurs pour la mise au point de leurs figures et macros s'expriment au niveau d'abstraction du micro-monde que constitue Cabri. Cela rejoint la remarque de [Pane & Myers 96] (p.34)

"Testing and debugging are areas of difficulty for novices [du Boulay 89-a]. As mentioned above (see "Support Incremental Running and Testing with Immediate FeedBack" on page 12), [du Boulay 89-b] claims that the computational machine should reveal its internal working in terms of the language itself. This can be interpreted as a call for a source-level debugger and tracer with data visualization. "

Dans Cabri, non seulement les programmes sont exécutés au fur et à mesure de leur édition dans la vue géométrique, mais en plus les bogues de l'utilisateur apparaissent comme des décalages entre les comportements désirés des objets qu'il a construits et ceux qui sont révélés par les animations de sa construction. Il s'agit d'une conséquence du concept de micro-monde : Cabri constitue un micro-monde de programmation.

L'environnement de Cabri-géomètre est donc bien adapté à l'enseignement de la mise au point de programmes.

b. En quoi notre version prototype peut y contribuer

Notre version prototype est une extension du logiciel Cabri-géomètre qui intègre une vue textuelle synchronisée et dynamique de la figure en cours de construction.

Les vues textuelle et géométrique sont synchronisées avec sélection simultanée des objets dans la forme correspondant à chaque vue, ce qui permet un apprentissage implicite du langage de Cabri-programmation. On parle d'ubiquité d'un objet car sa présence est matérialisée en plusieurs lieux à la fois. La vue textuelle est équivalente à la vue géométrique : elle permet d'accéder à toutes les informations et elle est réactive (dynamique) puisque le programme se construit en même temps que la figure. La qualité « dynamique » de la géométrie dans la figure est traduite par une qualité « formelle » du langage induit. Considérer un dessin, c'est-à-dire une configuration particulière de la figure, correspond à considérer une instance particulière de l'expression formelle du programme. Une vue instantannée d'un objet est un accès temporaire à son état manipulatoire courant (caché, punaisé, à l'infini...), ses attributs graphiques (aspects) et ses valeurs effectives (analytiques).

Le besoin de maîtriser la stratégie (plan de conception) de construction de la figure géométrique constitue une des principales motivations à l'extension du logiciel par l'accès à la structure logique du programme.

La vue textuelle permet à l'utilisateur de naviguer entre différents niveaux d'abstraction. Au niveau le plus bas, l'utilisateur peut connaître toutes les informations mémorisées sur chacun des objets géométriques. Au plus haut niveau, l'utilisateur atteint une vue synthétique de toute la construction grâce à l'affichage plié des macro-constructions utilisées. Il peut extraire de cette vue la stratégie sous-jacente au programme. Il peut voir aussi le contenu effectif des figures après l'utilisation des macros, la redéfinition des contraintes et l'élimination d'objets, et en particulier la structure logique de ses macros.

Notre version va améliorer l'utilisabilité de Cabri pour apprendre à programmer et déboguer.

En effet, de même que pour la rédaction d'un document, l'élaboration d'un programme

"constitue un processus dynamique qui implique le passage à travers différentes phases dans lesquelles les [programmeurs] réalisent des activités spécifiques. [...] L'ensemble des phases du processus d'édition constitue un modèle cognitif [d'activité de programmation]" [Romero Salcedo 98].

La vue textuelle du programme de construction peut aider l'utilisateur dans différentes étapes du modèle cognitif d'activité de programmation. Un tel modèle inclut les phases suivantes (les points 1 et 2 sont analogues à ceux de la section précédente, les autres non) :

1. Délimitation de l'objectif. L'objectif peut être décrit par un énoncé (par exemple un cahier des charges, avec des désirs et des restrictions) ou par une projection sur le résultat attendu (par exemple un scénario). Il reste néanmoins une intersection à déterminer entre cet objectif et le champ des résultats atteignables, restreint par le pouvoir d'expression du langage.
2. Structuration de l'objectif et de la résolution. Ici aussi le langage de programmation a une influence. Par exemple, selon que le langage de programmation est déclaratif ou impératif, la décomposition de l'objectif sera différente. Dans le cas déclaratif, les objets construits sont déclarés ainsi que les contraintes qui les relient. Toutes les contraintes sont « égales » : il n'y a pas d'ordre subjectif dans leur établissement. Dans le cas impératif, les constructions d'objets sont ordonnées avec les contraintes qui les relient aux autres objets déjà construits. Ainsi l'ordre de création des objets n'est pas libre (cf. §I-2-A-3°-c, §I-2-C-2°-c).
3. Planification du développement. Une fois l'objectif décomposé, le programmeur doit déterminer l'ordre dans lequel développer les différents éléments. Cette décomposition étant plus ou moins arborescente, la méthode de programmation peut être assujettie au choix de finir chacun des éléments avant de passer au suivant, dans un ordre de parcours spécifique de l'arbre— profondeur ou largeur d'abord, ou mixte — ou de progresser sur plusieurs éléments en même temps à des fins de mise au point incrémentale.
4. Traduction dans le langage de programmation. Schématiquement, on peut dire que l'utilisateur doit s'adapter au langage compris par l'ordinateur. Il doit écrire un programme syntaxiquement et sémantiquement correct qui réalise son objectif.

5. Mise au point par vérification de l'adéquation du résultat par rapport à l'objectif.

Dans l'environnement Cabri, l'utilisateur tend à répondre à ses problèmes de programmation par l'expérimentation. Cela lui permet de s'imprégner du problème à résoudre dans le cadre de l'environnement où il doit être résolu, et ainsi de cadrer l'objectif, indépendamment de tout problème de formulation.

Une fois l'objectif cadré, l'utilisateur peut poursuivre sa recherche de solution, directement par l'expérimentation. Il s'agit d'une continuité du processus en cours. Il peut aussi, arrivé à un certain point de la résolution, décider de structurer sa méthode, de façon à éviter de recommencer des étapes qu'il a déjà résolues. Pour cela, un moyen offert par le logiciel est la définition a posteriori de macro-constructions.

La remise en cause des processus utilisés dans l'étape d'expérimentation, conduit à une planification du développement. En effet, en cas de refonte totale ou partielle, toutes les tentatives expérimentées ne seront pas reconduites, mais seulement les actions utiles. L'utilisateur choisit donc parmi ses diverses tentatives celles qui font finalement partie du plan, de la stratégie. Il peut aussi choisir de récupérer, par définition de macros, quelques parties des constructions qu'il a effectuées.

Ainsi, les contributions aux trois premiers points ne sont pas directement liées à la vue textuelle. Mais dans les phases de remédiation (deux derniers points), la vue textuelle est recherchée et sollicitée, comme par exemple en cas d'échec de l'outil de validation des macros. La vue textuelle permet alors un apprentissage et une découverte par l'exemple de ce que constitue un programme. Le passage à cette vue permet à l'utilisateur de s'imprégner de concepts associés aux langages informatiques par observation de l'exemple du programme courant qui représente sa construction.

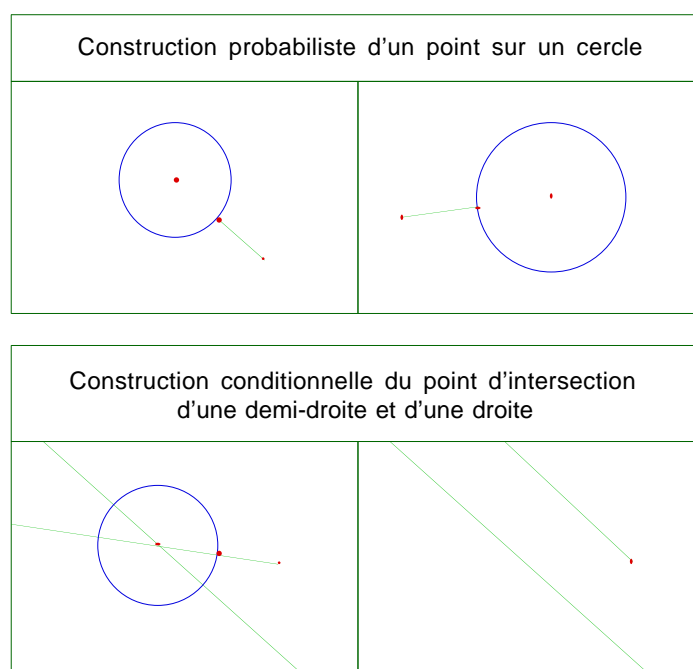
Contribution à l'assimilation du concept de langage informatique

Ces concepts informatiques sont :

- Ce qu'est un langage informatique, avec sa syntaxe et sa sémantique. Le respect de la syntaxe est dirigé et implicite. Il disparaît complètement du souci de l'utilisateur, tant que l'utilisateur ne modifie que la vue textuelle par ses manipulations du logiciel, et non pas par un éditeur extérieur ou par le clavier. La correspondance sémantique est induite de la synchronisation des « rendus » géométrique et textuel, c'est-à-dire résultat et programme. L'accès à la sémantique résulte des réactions aux manipulations et du retour d'information du curseur.
- Ce qu'un langage informatique particulier permet de faire : quel est son pouvoir d'expression, quel type de problème un langage permet-il de résoudre ? l'observation des comportements du résultat de la construction courante permettent à l'utilisateur de remettre en cause sa stratégie, et aux limites de toutes les tentatives, de cerner le domaine des comportements accessibles et par opposition de donner une idée de ce qui ne l'est pas. Par exemple une inversion de l'ordre des contraintes n'est pas possible dans Cabri sans reconstruction ou redéfinition des contraintes. Le prototype permet de s'imprégner de l'effet des redéfinitions de contraintes.
- Ce sur quoi le langage permet d'agir. Un langage informatique permet de manipuler des données organisées en structures. Cette organisation est basée sur des définitions de types, capables de représenter des variables qui sont instanciées par des valeurs (constantes). Le prototype permet de manipuler les données en agissant directement sur des images de leurs réalisations ou sur leurs identificateurs. La relation entre ces deux aspects concret (ou du moins avec une réalité graphique) et abstrait, est matérialisée par une animation synchronisée.

Par quel moyen ? Les utilisateurs peuvent aborder la notion de procédure et de paramètres, ainsi que la structuration de la programmation, en examinant les macro-constructions élaborées automatiquement. L'édition synchronisée graphique et textuelle liée aux manipulations de l'interface d'un logiciel dans lequel la construction automatique des macros est très proche des procédures, permet aux utilisateurs d'apprendre et de découvrir par l'exemple et l'exposé. La différence entre paramètre et variable est perceptible dans le programme par la forme de l'identificateur et par son identification multiple. Par opposition aux identificateurs de variable, un identificateur de paramètre est précédé par le caractère '#'.

Par exemple, « P », « A », P3 peuvent être des noms de points, les noms entre guillemets étant des noms donnés par l'utilisateur, les noms sans guillemets mais indicés étant attribués par le logiciel. Ces noms identifient les variablesinstanciées par les valeurs de leurs coordonnées. Dans les macros, tous les noms de points sont de la forme #P1, #P2... Ils identifient les paramètres de la macro, qui seront remplacés lors de l'appel par des variables comme « P », « A » ou P3. Chaque variable est en relation avec un seul objet concret. Les paramètres sont des variables particulières propres aux macros. À chaque utilisation d'une macro, ses paramètres sont identifiés avec les variables spécifiées par l'appel. Ainsi, selon l'appel, un paramètre est mis en correspondance avec plusieurs variables, et donc avec plusieurs objets concrets. Cette correspondance est matérialisée par l'animation associée au rendu de la synchronisation.



Avec quel contrôle ? Ni la version commerciale de Cabri, ni le prototype ne propose d'instruction de contrôle. Par contre, Cabri permet des constructions conditionnelles et des constructions probabilistes. Les constructions conditionnelles sont liées à chacun des points d'intersection de deux objets, indépendamment de leur existence dans le cas de la configuration courante de la figure. Il en résulte des réalisations graphiques conditionnelles à l'existence de certains points d'intersection, qui peuvent, à un certain niveau d'abstraction, simuler des instructions conditionnelles (cf. illustration du §I-1-B-1°-a).

La vue textuelle permet de percevoir l'alternative associée à l'existence ou non de ces points : dans le programme (vue textuelle), ces points existent toujours, mais ils ne sont réalisés dans la vue graphique que dans le cas de leur existence. Les constructions probabilistes sont liées à des instances aléatoires de variables internes de macros (cf. §I-1-B-1°-c). La vue textuelle permet de manipuler les objets internes aux macros, et donc les valeurs affectées aléatoirement à ces variables.

Ainsi, la traduction dans le langage de programmation est automatique, ce qui soulage l'utilisateur d'un premier niveau de formulation, et le familiarise par l'exemple et l'imitation, au respect des contraintes syntaxiques. L'utilisateur agit par des actions qui sont directement intégrées au programme. La réponse à ses actions dans la vue géométrique informe l'utilisateur sur la sémantique qui leur est associée.

Aide à la transposition

Le dynamisme de la vue structurelle contribue à la décontextualisation, au sens didactique du terme, c'est-à-dire à la transposition : l'objet à enseigner est extrait des éléments de son contexte qui ne contribuent pas directement au savoir enseigné.

En effet, l'enseignement d'une syntaxe particulière n'est pas l'objectif de l'enseignement de la programmation, mais une obligation pour expérimenter l'apprentissage dans le contexte d'un environnement particulier. Par contre, l'élève doit apprendre que pour programmer, il doit respecter des syntaxes, que ces syntaxes transcrivent des règles de formulation, et qu'à ces règles sont associées des sémantiques qui transcrivent les formulations en résultat obtenu par le programme.

Enfin, arrivé à un certain niveau de mise au point, l'utilisateur est obligé de passer par une étape de formulation du programme pour se remémorer la stratégie en cours et effectuer des raisonnements. La formulation devient nécessaire à l'utilisateur, comme support de raisonnement. C'est de là que vient le besoin de la forme structurelle : la mise au point passe par la formulation des dysfonctionnements, nécessaire pour planifier des remédiations de la stratégie mise en œuvre.

Ainsi le choix d'une vue textuelle relie Cabri à l'usage des langages de programmation textuels, et ainsi permet d'apporter à l'enseignement classique de la programmation le bénéfice de ses qualités manipulatoires.

De plus, l'approche suivie pour la réalisation du prototype (depuis un logiciel multilingue permettant des activités programmatoires basées sur des manipulations d'interface et non pas basé sur un langage de commandes textuelles transcrites en artefacts manipulatoires d'IHM), conduit à un langage de programmation dont les termes et la syntaxe respectent automatiquement la langue de dialogue choisie par l'utilisateur (français, anglais, allemand...), ce qui rare et intéressant.

2°) Pour la recherche sur les environnements de programmation

L'environnement constitué par cette version de Cabri dotée des fonctionnalités d'édition textuelle intimement liée à la vue géométrique, fournit un cadre d'étude et de comparaison des pouvoirs d'expression <graphique + animation> versus <texte + animation> en programmation.

L'extrait suivant souligne l'intérêt actuel pour l'étude des contributions de différentes vues pour mobiliser les capacités cognitives dans les activités de programmation.

"Multiple Representations

What are the benefits (if any) of using multiple representations in programming? [Schneiderman 77] has suggested that if one of the representation is well-known and understood, then additional ones will be redundant. However, this may not necessarily be true, given that:

- *different representations will highlight different types of information at the expense of others;*
- *understanding of a notation is often not complete and/or correct (e.g. when learning a new language, novel application of the language).*

We know of no empirical work in the field which has addressed the use of multiple representations in programming. Could work on multiple representations for problem solving in other domains (e.g. Cox's (1996) work on constraint satisfaction problems with SwitchER, or Ainsworth & al's work on computational estimation (1996)) be extended to programming ?" [Blackwell & al.98-b].

Les différentes vues peuvent être considérées comme des dimensions transversales ou des facettes des programmes.

Dans le domaine de la géométrie, indépendamment de l'activité de programmation, Cinderella utilise plusieurs vues du résultat. Dans les différentes vues, les objets géométriques, du fait même de leur nature, ne répondent pas aux manipulations de la même manière : dans la vue sphérique, les points à l'infini représentant les "extrémités" sont impossibles à "saisir", obligeant à l'utilisation du texte ou de la vue euclidienne pour poursuivre la figure en utilisant ces points. Mais la vue sphérique doublée de la vue euclidienne donne un accès perceptif à des propriétés abstraites dans la vue euclidienne et "réaliste" ou plutôt concrète (ou manipulable mais non palpable) à ces propriétés.

Dans le domaine de la programmation, notre prototype produit deux vues synchronisées : l'une du résultat et l'autre du programme. L'information portée par l'animation peut constituer, dans un certain sens, une autre vue du programme, puisqu'elle contribue à informer l'utilisateur sur l'effet du programme. Le choix de laisser ouvert l'intégration d'une vue graphique du programme, a été gardé en permanence lors de la conception de cette version. La version prototype constitue un cadre pour évaluer l'apport des différentes vues, leur complémentarité, et pour les comparer.

C) Conclusion

Transformer un environnement logiciel permettant des activités programmatoires en un environnement de programmation tourné vers le tout-public, présente plusieurs difficultés :

- Synthèse des spécificités des techniques manipulatoires selon la forme des données manipulées (texte, graphe, représentant graphique du domaine (métaphore ou objet) -dans Cabri : géométrie)
- Recherche des techniques manipulatoires utiles pour la mise au point de logiciels par des programmeurs de tous profils : du novice à l'expert, du professeur à l'élève, du chercheur à l'industriel.
- Intégration, dans une architecture logicielle déjà existante, des éléments logiciels qui gèrent les artefacts manipulatoires adaptés aux différentes formes de données présentées dans les différentes vues, tout en coordonnant ces différentes vues.

Nous donnerons nos solutions dans la partie suivante.

Conclusion de la partie I

Notre travail participe de plusieurs domaines de recherche : les interfaces homme-machine de manipulation directe, la programmation visuelle ou par démonstration, la manipulation directe appliquée aux éditeurs textuels de programmes, l'édition multivue synchronisée voire multimodale, et les environnements de programmation visuelle pour non-programmeur.

Notre travail contribue à la recherche sur les environnements formant des micro-mondes de programmation interactive ouverts au tout public. En effet :

- l'approche est inversée par rapport à l'approche historique du domaine, puisqu'au lieu de s'appuyer sur un langage de commandes et de le rendre manipulable par l'intermédiaire des fonctionnalités d'une interface, le langage de programmation est induit par les manipulations possibles d'une interface de manipulation directe déjà existante.
- le contexte de Cabri offre un support d'étude et d'évaluation de l'intérêt d'une vue textuelle dans les environnements de programmation visuelle, ou de l'exploitation des qualités complémentaires des différentes vues par leurs manipulations simultanées et synchronisées.

Dans le chapitre I-1, nous avons montré que les activités des utilisateurs du logiciel Cabri-géomètre s'apparentent à des activités de programmation. Nous avons fait ressortir les difficultés rencontrées par l'utilisateur lors d'actions particulières qui touchent à la structure profonde du programme de construction de la figure en cours. Ces difficultés sont dues à ce que l'utilisateur ne maîtrise pas la structure logique du programme qu'il saisit. D'où la nécessité d'intégrer dans ce logiciel une vue dynamique et synchronisée à la vue géométrique, qui donne accès à la structure logique du programme de construction en cours.

Dans le chapitre I-2, nous avons examiné les différents supports utilisés pour aider les programmeurs dans leurs activités de programmation au cours de l'histoire de l'informatique, et plus particulièrement dans le cadre des environnements actuels qui mettent à profit toutes les capacités graphiques des écrans des ordinateurs. Deux formes principales peuvent donner accès à cette structure : une vue textuelle ou une vue basée sur des graphes. Les apports effectifs de la programmation visuelle sont actuellement sujets à controverse.

Dans le chapitre I-3, nous avons comparé les approches mises en œuvre concrètement dans des environnements particuliers, ainsi que la forme choisie pour supporter les structures logiques des programmes. Les informations portées par les vues elles-mêmes, leurs possibilités d'animation et de manipulation, donnent à ces deux formes des intérêts différents, complémentaires par certains aspects et non uniformément adaptés à tous les types d'utilisateurs, que ce soit par leur degré d'expérience ou leur forme d'esprit. De cette étude, nous avons déduit une liste de critères utiles pour évaluer les possibilités programmatoires offertes par les logiciels et fournir des éléments à introduire dans le cahier des charges du prototype à réaliser.

Le chapitre I-4 a été consacré à un examen de toutes les applications attendues du prototype.

Les besoins de navigation entre les différents niveaux d'abstraction sont accentués par le choix conceptuel de laisser l'initiative aux utilisateurs tout en leur donnant les moyens de maîtriser les informations qu'ils manipulent. Ce sont les utilisateurs qui sont le plus en mesure d'effectuer le choix du support de la structure logique ainsi que du niveau d'abstraction qui leur permettent de cerner au mieux la sémantique du programme.

Les principales difficultés proviennent de la nécessité que les différentes vues réagissent simultanément à chaque action associée à une manipulation de l'interface, et cela de manière adaptée aux contraintes particulières de la vue, et même de façon optimale par rapport à ses spécificités.

Il s'agit entre autres de déterminer une forme de manipulation directe applicable aux éléments de texte des éditeurs textuels de programmes, par la mise en évidence de fonctionnalités adaptées aux objets textuels en relation directe avec une vue graphique.

La vue graphique est une exécution du programme de construction affichée dans la vue textuelle. Donc la vue textuelle est une vue exécutée dynamiquement : les modifications du programme doivent être immédiatement transcrites sur le résultat de son exécution (la vue graphique), sans passer par une phase de compilation.

Les parties suivantes présentent la spécification et la réalisation du prototype d'éditeur textuel corrélé à l'interface déjà existante du logiciel.

Partie II

De la manipulation à la programmation

Notre objectif est d'apporter au logiciel Cabri-géomètre des fonctionnalités d'environnement de développement pour les programmes de construction de figures géométriques.

Dans la partie I, nous avons montré la nécessité d'introduire, dans ce logiciel, un éditeur textuel lié aux manipulations de l'interface, de façon à le doter de fonctionnalités d'environnement de programmation. Ainsi l'éditeur textuel doit être un éditeur de programme muni de fonctionnalités de débogueur.

L'objectif de cette partie est de cerner les attentes et les moyens de le réaliser. Par exemple, le comportement attendu d'un débogueur est de visualiser les instances des variables, et de permettre une exécution pas à pas ou rapide avec points d'arrêt. L'éditeur textuel doit intégrer des artefacts manipulatoires adaptés à la forme textuelle des données, tout en étant lié à la vue des résultats (dite vue géométrique), donc à une manipulation directe de représentations graphiques des données. Au même titre que la vue géométrique, la vue textuelle doit constituer une image de l'état courant du noyau fonctionnel de l'application.

Cette partie II est décomposée en quatre chapitres.

Le premier chapitre présente ce qu'on voudrait, c'est-à-dire les désirs (spécification externe) et les contraintes (internes et externes). Nous y définissons un langage de géométrie formelle, représentatif de la géométrie dynamique supportée par le logiciel.

Le deuxième chapitre est consacré aux spécificités du domaine et du logiciel qui obligent à des résolutions particulières, et à la présentation de la mise en œuvre.

Le troisième chapitre présente le résultat, c'est-à-dire le prototype réalisé, son évaluation et ses limitations.

Enfin, le dernier chapitre est consacré aux prolongements et perspectives : passage d'une version linéaire à une version hiérarchique, uniformisation de tous les traitements de manipulation directe, extension de la notion de macros et amélioration de leurs outils de débogage, enrichissement des structures de contrôle, etc.

Le premier aspect à aborder est le choix du support à mettre en place pour la vue de la structure logique d'une figure géométrique construite dans le logiciel Cabri-géomètre. Nous avons vu dans la partie I que cette vue doit être dynamique et synchronisée avec la vue géométrique, que l'édition du programme et l'exécution doivent matérialiser conjointement les réactions de l'interface vis-à-vis des manipulations de l'utilisateur. Nous avons vu aussi que la vue textuelle doit permettre à l'utilisateur de naviguer entre différents niveaux d'abstraction selon la tâche courante qu'il veut effectuer.

Dans ce cadre et en respectant le principe de fournir à l'utilisateur des moyens pour exercer ses propres choix plutôt que de lui imposer ceux des développeurs du logiciel, l'approche choisie consiste

- à effectuer une première réalisation avec un support textuel,
- à réaliser notre implémentation de sorte que ce choix ne soit pas irréversible.

Le choix d'un support textuel, justifié dans la partie I, est basé principalement sur le fait qu'un texte est immédiatement lisible. Ainsi le problème du choix des informations à montrer est allégé de celui du choix des formes de représentation de chacune d'elles. De plus, cette approche permet d'imaginer de saisir le texte à l'aide d'un éditeur textuel extérieur.

Cependant avec l'objectif de pouvoir, à terme, proposer à l'utilisateur de choisir la forme qui lui convient le mieux, les composants ajoutés doivent être échangeables (ou mis en double) avec ceux qui donneraient accès à une vue graphique. Ultérieurement, un moyen pour le "doublage" de la vue textuelle avec une vue basée sur des graphes, sera de coupler Cabri-géomètre avec un éditeur de graphes comme Cabri-graphe.

Dans ce chapitre, nous spécifions le contenu et le fonctionnement d'une vue textuelle dynamique et synchronisée intégrée dans un logiciel constituant un micro-monde de géométrie dynamique.

La première section est consacrée à la présentation des désirs considérés indépendamment du contexte matériel de l'intégration. Il s'agit de spécifications externes.

Les contraintes liées au résultat font l'objet de la deuxième section. Il s'agit des contraintes externes, qui recadrent le prototype dans la réalité de ses utilisations possibles.

Les contraintes associées au contexte particulier du logiciel Cabri-géomètre, et à la manière dont il est actuellement conçu et réalisé, sont décrites dans la dernière section, consacrées aux contraintes internes.

A. Spécificité de l'interface désirée

Quelle apparence externe attendons-nous de notre prototype ? Deux aspects doivent être spécifiés : statique et dynamique.

Notre vue textuelle doit permettre à l'utilisateur d'être informé sur l'état courant du programme de construction qui est exécuté et dont la vue géométrique présente le résultat. Le support de cette information est donc un espace de communication entre le logiciel et l'utilisateur.

Quelle est l'information mémorisée dans un programme de construction ? Il s'agit, parmi toutes les informations mémorisées par le logiciel sur chacun des objets géométrique, de déterminer celles qui sont utiles aux tâches de programmation. Cet aspect concerne plutôt une contribution statique à la communication et constitue le sujet de la première sous-section.

Comment présenter cette information ? Il s'agit de déterminer sa forme avec un langage induit des manipulations de l'interface. Le choix de la forme et du langage constitue le sujet de la deuxième sous-section.

La troisième sous-section est consacrée à l'aspect dynamique de l'environnement, avec l'évolution de la forme, les outils de navigation dans les différents niveaux d'abstraction et les artefacts utiles pour aider au débogage des figures et des macros.

1°) Informations à produire

Le logiciel, au départ, visualise les résultats des manipulations de son interface sur un support graphique qui modélise une feuille de papier de 1 m^2 . Le noyau fonctionnel du logiciel permet d'interpréter les manipulations effectuées par l'utilisateur et de répercuter ces interprétations dans une figure géométrique. Les actions sont liées au mode opératoire et aux désignations des éléments de l'interface et des objets de la figure. L'interprétation est la transcription de ces actions en une suite de commandes du noyau géométrique. La répercussion de ces commandes est matérialisée par leurs réalisations. Le concept de manipulation directe impose que toute répercussion soit rendue perceptible à l'utilisateur.

Par exemple, la répercussion d'une action de construction d'un objet consiste en l'application de toutes les étapes allant de la création à l'affichage de la représentation graphique de cet objet. Un objet dépend d'une part de son type, pour sa description statique, et d'autre part des contraintes qui le relient aux autres objets, pour sa description dynamique, c'est-à-dire la description qui permet de l'animer. Sa création consiste en la préparation des calculs pour la partie dynamique avec mémorisation de la méthode qui lie l'objet aux autres objets, et en la mémorisation de l'équivalent des équations capables de décrire sa représentation graphique, pour la partie statique.

Son animation, et donc la répercussion d'une action d'animation appliquée à l'objet, consiste à mettre à jour les « équations » du graphisme correspondant, puis à répercuter l'animation sur tous les objets qui en dépendent.

La répercussion d'une destruction, quant à elle, se décompose récursivement en la destruction de tous les objets dont la méthode de construction utilise l'objet à détruire : chacun des objets est effacé et la place qu'il occupait en mémoire est libérée.

Ainsi, le noyau fonctionnel est considéré comme un interpréteur de langage de commandes graphiques, ou encore, puisque le logiciel Cabri-géomètre permet des activités de programmation, comme un compilateur à la volée du langage de programmation par démonstration. La résolution syntaxique correspond à l'interprétation des actions de l'utilisateur. La résolution sémantique correspond à la répercussion des actions.

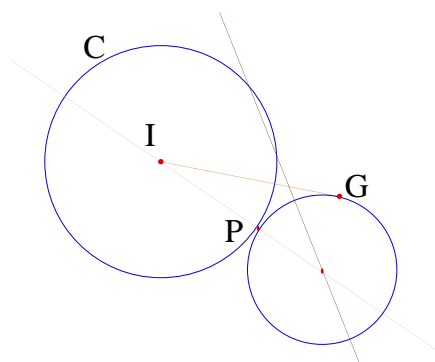
Le principal objectif du texte (ou du graphe) à produire, est de fournir un support pour remédier aux difficultés inhérentes aux activités de programmation dans le logiciel. Les fonctionnalités à apporter concernent principalement le suivi de l'état de la structure logique du programme construit après élimination d'objets, redéfinition de contraintes ou utilisation de macro-constructions.

L'information à fournir doit représenter l'état courant du programme de construction mémorisé par le logiciel, c'est-à-dire la liste des objets qui est parcourue par le noyau fonctionnel pour afficher toute la figure, appels des macros compris. Ce contenu comprend :

- la structure logique de la figure et des macros, c'est-à-dire toutes les relations de dépendance entre les objets créés : les méthodes de construction utilisées pour chacun d'eux, les dépendances résultant des méthodes de construction, ainsi que la relation d'ordre induite par l'ordre de création, et la dépendance vis-à-vis d'une macro-construction (création de cet objet par un appel de macro, quelle macro et quelle commande de cette macro).
- la description analytique de chaque objet de la figure, utilisée pour son affichage, et complétée de ses attributs graphiques (épaisseur, couleur, etc.) et de son état (visible, punaisé, à l'infini, etc.).

a. Structure logique

Figures



La figure suivante présente un exemple utilisé pour définir la macro du problème ouvert du § I.A.1^o.c.

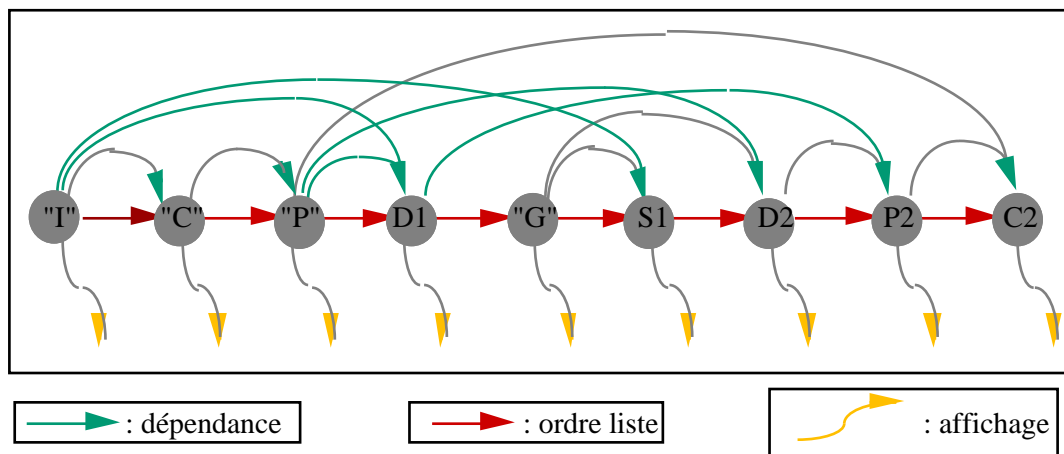
Pour réaliser cette figure, l'utilisateur a d'abord construit le cercle C, définissant son centre I à la volée. Ensuite, il a considéré un point du cercle, qu'il a nommé P, et a tracé en gris la droite (IP). Il a créé le point G, et a tracé en orange le segment [IG] (inutile pour la macro à définir), et en marron la médiatrice des deux points G et P. Enfin, le cercle tangent cherché est défini comme le cercle de centre le dernier point défini (le point d'intersection) et qui passe par G.

La création à la volée d'un objet conduit à insérer une action de création dans la liste des objets construits, comme si l'outil correspondant avait été actionné par l'utilisateur dans l'ordre correspondant. Il en découle que la liste des actions effectuées par l'utilisateur est équivalente à l'algorithme suivant, présenté sous une forme actionnelle (cf. § I.2.A.2^o). Dans cette présentation, les identificateurs entre doubles quotes sont ceux qui ont été choisis par l'utilisateur.

Lexique
 "I", "P", "G", P2 : quatre points
 "C", C2 : deux cercles
 S1 : un segment
 D1, D2 : deux droites

Algorithme
 ConstruirePoint ("I")
 ConstruireCercle ("I", "C")
 ConstruitePointSur ("C", "P")
 ConstruireDroite ("I", "P", D1)
 ConstruirePoint ("G")
 ConstruireSegment ("G", "I", S1)
 ConstruireMédiatrice ("G", "P", D2)
 ConstruirePointIntersection (D1, D2, P2)
 ConstruireCercle (P2, "P", C2)

L'information mémorisée par le logiciel pour cette figure peut être représenté par :



Le graphe représente la structure logique de la figure, avec les relations de dépendance liées aux contraintes définies entre ses différents éléments (les flèches situées au dessus des noms des objets) et à l'ordre de construction (les flèches horizontales). L'extraction des informations utilisées pour l'affichage graphique est représentée par les flèches situées sous les objets.

Une relation directe apparaît entre la liste des actions de l'algorithme et ce schéma. Ainsi, l'expression actionnelle de l'algorithme décrit la structure logique de la figure. Pour que l'information soit complète, il faut lui ajouter les informations d'affichage.

Macros

L'information à fournir doit aussi présenter le contenu des macro-constructions définies. Dans l'exemple précédent, la macro définie par le cercle "C" et le point "G" comme objets initiaux et le cercle C2 comme objet final, est déduite par le logiciel, par abstraction de la partie utile extraite de la figure (cf. §I-1-A-2°-c).

Remarque : le point "I" est l'origine implicite du cercle C1 et les points P1 et P2 sont des objets finals implicites de la macro. En effet, l'utilisateur ne les a pas explicitement désignés comme étant des paramètres de la macro, mais le logiciel les a considérés comme tels : ce sont des points de base de la construction d'un autre paramètre (C1 pour "I" et C2 pour P1 et P2) (cf. §I-1-B-1-c).

Pour déterminer cette partie utile et visualiser le traitement effectué, il suffit de :

- marquer les objets initiaux explicites et les objets initiaux implicites à l'aide d'un marqueur (#),
- propager les dépendances depuis les objets finals à l'aide d'un autre marqueur (§), en arrêtant cette propagation sur les objets initiaux, marqués de (#).

Lexique

#"I", §"P", #"G", §P2 : quatre points
 #"C", §C2 : deux cercles
 S1 : un segment
 §D1, §D2 : deux droites

Algorithme

ConstruirePoint ("I")
 ConstruireCercle ("I", "C")
 ConstruitePointSur ("C", §"P")
 ConstruireDroite ("I", §"P", §D1)
 ConstruirePoint ("G")
 ConstruireSegment ("G", #"I", S1)
 ConstruireMédiatrice ("G", §"P", §D2)
 ConstruirePointIntersection (§D1, §D2, §P2)
 ConstruireCercle (§P2, §"P", §C2)

On obtient alors l'algorithme suivant :

La partie utile est la liste des actions dont le résultat est marqué d'un §. Les actions qui construisent des objets initiaux n'ont pas besoin d'être enregistrées, puisqu'elles ne seront pas à exécuter lors des appels de la macro, ni les actions qui construisent des objets inutiles pour la construction de l'objet final (le segment S1). Elle est décrite par :

Lexique

"I", "P", "G", P2 : quatre points
 "C", C2 : deux cercles
 D1, D2 : deux droites

Algorithme

ConstruirePointSur ("C", "P")
 ConstruireDroite ("I", "P", D1)
 ConstruireMédiatrice ("G", "P", D2)
 ConstruirePointIntersection (D1, D2, P2)
 ConstruireCercle (P2, "P", C2)

Dans cet algorithme, les actions sont appliquées aux objets définis dans la figure. L'abstraction de cette partie par le logiciel est une formalisation (généralisation) qui permet de considérer les objets sur lesquels sont effectués les actions comme des objets formels. Les objets formels sont soit les paramètres des macros, soit des objets internes aux macros.

Dans l'illustration suivante, on présente le texte de la macro elle-même. Les noms formels des objets sont précédés du caractère dièse, et ceux qui représentent des paramètres externes de la macro (objets initiaux, implicites ou finals) sont désignés dans le lexique par le mot « paramètre » qui précède leur type. Cette description n'étant présentée ici qu'à des fins d'illustration, nous avons ici supprimé les doubles quotes des noms et rajouté l'indice 1 au nom « P ». On n'a donc plus que des identificateurs de forme #chaîne.

Lexique

#C1, #C2 : deux paramètres cercles
 #I, #P1, #G, #P2 : quatre paramètres points
 #D1, #D2 : deux droites

Algorithme

ConstruirePointSur (#C1, #P1)
 ConstruireDroite (#I, #P1, #D1)
 ConstruireMédiatrice (#G, #P1, #D2)
 ConstruirePointIntersection (#D1, #D2, #P2)
 ConstruireCercle (#P2, #P1, #C2)

Appel d'une macro

Sur la figure suivante, l'utilisateur a construit un nouveau cercle C et un nouveau point G, puis il a appelé la macro décrite précédemment sur ces objets. La liste des actions effectuées par l'utilisateur est décrite par l'algorithme suivant, dans lequel la macro définie précédemment est appelée « AppelerMacro ». Les deux droites #D1 et #D2 n'étant plus construites par l'utilisateur, nous les avons ôtées du lexique.

Lexique

"I", "P", "G", P2 : quatre points
 "C", C2 : deux cercles

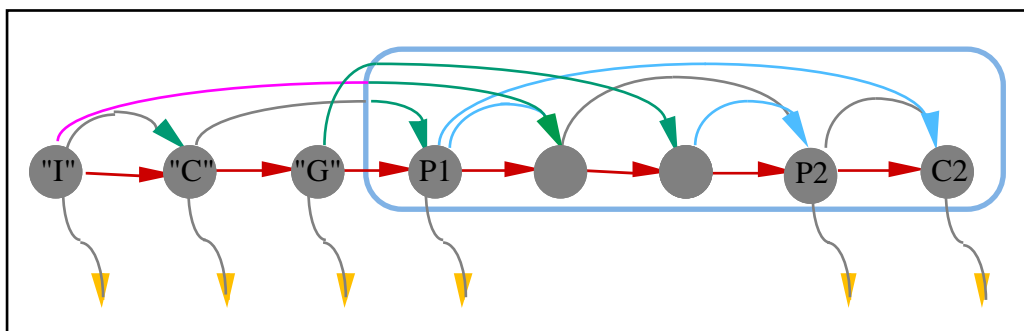
Algorithme

ConstruirePoint ("I")
 ConstruireCercle ("I", "C")
 ConstruirePoint ("G")
 AppelerMacro ("C", "G", C2)

La liste des actions exécutées par le logiciel pour cet appel de macro est obtenue en remplaçant respectivement les paramètres #C1 par "C", #I par "I" et #C2 par C2 dans l'algorithme de la macro, ce qui donne :

ConstruirePointSur (C1, P1)
 ConstruireDroite ("I", P1, D1)
 ConstruireMédiatrice ("G", P1, D2)
 ConstruirePointIntersection (D1, D2, P2)
 ConstruireCercle (P2, P1, C2)

L'information mémorisée par le logiciel pour cette figure est représentée par le schéma suivant.



Dans ce graphe, les objets construits par la macro sont entourés d'un cadre bleu. Les deux droites D1 et D2, internes à la macro, sont inaccessibles à l'extérieur de l'appel et de l'exemple d'apprentissage, c'est pourquoi leurs noms sont écrits en gris clair.

On retrouve dans cette représentation l'algorithme de construction décrit précédemment. Il est porté par le graphe défini par les flèches situées au-dessus des noms des objets, en associant une action à chaque objet quand il n'est pas construit par une macro, et au regroupement des objets construits par elle quand il l'est.

Réciproquement, depuis l'expression actionnelle, on retrouve la structure logique en dépliant l'appel de la macro, c'est-à-dire en remplaçant son appel par la liste des constructions effectuées.

Le niveau d'abstraction utilisé dans les algorithmes précédents permet de décrire les figures géométriques, sans tenir compte des spécificités d'affichage. Il donne accès à la géométrie dynamique supportée par le logiciel.

Nous avons défini la géométrie formelle comme la spécification statique de la géométrie dynamique. Un langage de géométrie formelle doit donc permettre de décrire les figures géométriques en laissant possible leur animation : l'ensemble des instances possibles des variables formelles du langage est exactement l'ensemble des dessins (configurations) possibles pour une figure donnée. Ainsi, un langage de géométrie formelle permet de définir la structure de la figure, sans s'occuper du "bruit" généré par les informations inutiles à sa structure.

L'expression actionnelle qui décrit formellement une figure peut être mise en relation directe avec la liste des actions effectuées par l'utilisateur pour construire ses objets. Les algorithmes décrits sous cette forme peuvent être traduits en un langage de géométrie formelle, qui peut donc être déduit du langage de commandes « induit » de l'interface, c'est-à-dire du langage d'interface. Nous parlons de langage induit de l'interface car Cabri n'est pas une transcription interactive d'un langage interne de commandes : au contraire, nous avons induit ce langage à partir du fonctionnement interne de Cabri.

b. Génération graphique

La description formelle d'une figure ne spécifie pas les aspects graphiques de ses objets. Elle ne permet donc pas de spécifier complètement le dessin de chaque objet.

La modification des attributs graphiques passe par l'usage de commandes, et donc par des actions de l'utilisateur. Mais ces actions n'ont pas de sens vis-à-vis de la structure logique du programme de construction. C'est pourquoi le langage de géométrie formelle cherché ne peut être induit de la liste de toutes les commandes (c'est-à-dire de l'utilisation de tous les outils) mais seulement de certaines.

La classification suivante nous permettra de dégager les commandes utiles pour la définition du langage cherché. Les outils du logiciel se répartissent entre :

- ceux qui ont toujours un sens pour la géométrie formelle :
 - les outils de création d'objets de base, comme les points d'une part, et les objets géométriques élémentaires, d'autre part, dont les méthodes de construction sont basées seulement sur des points ;
 - les outils de construction d'objets géométriques, dont les méthodes de construction peuvent être basées sur des objets géométriques élémentaires, et pas seulement des points (construction simple ou transformation) ; l'outil de marquage d'un angle est considéré comme l'outil de construction d'un angle ;
 - les outils de mesure, qui associent des valeurs munies d'unités à certains types d'objets géométriques ;
 - les outils de calcul sur les mesures ou sur des nombres, qui construisent des valeurs munies d'unités cohérentes ;

- ceux qui ont parfois un sens :
 - les outils de propriété, qui associent les propriétés relatives de certains types d'objets à ces objets (sous la forme d'une alternative de deux textes éditables) ;
 - les outils de définition de textes (ou commentaires) et de tables ;
 - l'outil équation ;
- ceux qui n'ont jamais de sens :
 - les outils de définition de Macro-constructions, avec les spécifications des objets initiaux et finals et l'outil de validation de Macro-constructions ;
 - les outils de redéfinition des contraintes, qui remplacent la méthode de construction utilisée pour un objet en une autre, basée éventuellement sur d'autres objets ;
 - l'outil permettant de nommer des objets ;
 - les outils d'affichage (ou de présentation), qui permettent de modifier l'apparence (rendu) des objets ;
 - les outils d'animation (déplacement, marquage).

Par exemple, les outils d'affichage, ou l'outil d'affectation d'un nom aux objets, ne permettent pas d'agir sur la structure logique du programme de construction. Leur usage ne doit donc pas apparaître dans la description recherchée.

Ni les attributs d'un objet, ni son nom ne sont considérés comme des objets : un langage de géométrie formelle ne doit donc pas permettre de les manipuler. Par contre, les mesures prises sur les objets sont utilisées par certains outils de construction d'objets géométriques, comme la rotation selon la valeur d'un angle. Les catégories transcriposables en géométrie formelle contiennent les outils de création d'objets de base, et les outils de construction d'objets géométriques, les outils de mesure, et les outils de propriété.

Les outils de définition de textes (ou commentaires) et de tables n'ont pas nécessairement de sens du point de vue de la géométrie formelle, selon l'utilisation qui en est faite. S'ils étaient pris en compte, ils le seraient de la même manière que les objets géométriques. Mais il n'y a d'intérêt à les prendre en compte que si leur utilisation est supérieure à celle de simples commentaires ou collecteurs de résultats, c'est-à-dire s'ils sont utilisés dans d'autres outils pour servir à construire d'autres objets.

Par exemple, pour une utilisation du logiciel à des fins de simulation pour des traitements de statistiques, les tables peuvent porter des mesures sur lesquelles sont calculés des moyennes, écarts-types et autres mesures statistiques, et les valeurs obtenues peuvent être utilisées géométriquement.

Lorsque l'utilisateur utilise les relations de dépendance définies par les outils de construction de texte, table et nombre, pour influencer sur la dynamique du résultat, sur ses possibilités d'animation, les capacités du logiciel qu'il recherche ne sont plus seulement celles d'un outil de géométrie (Cabri-géomètre), mais plutôt celles d'un visualiseur de phénomènes mathématiques dynamiques (Cabri-« étendu »). En conséquence, considérer ces outils dans le langage de géométrie formelle peut être laissé à l'utilisateur par définition de préférence, par exemple, puisque lui seul sait ce qu'il veut faire avec ce logiciel.

Actuellement, le comportement des objets de type non exclusivement géométrique n'est pas homogène dans Cabri : des objets textes ou tables sont créés pour contenir les objets-nombres et leur servir de support d'affichage et de sélection à travers la vue graphique. De plus, ni un objet texte ni un objet nombre ne peut être choisi comme objet final de Macro.

Les autres catégories d'outils ne concernent pas la géométrie formelle.

Cependant, les informations graphiques font partie des informations mémorisées par le logiciel, et pour certaines tâches, il peut être utile de connaître les valeurs effectives des objets géométriques, le rendu qui a été choisi pour eux, ou encore de connaître directement leurs possibilités d'animation, sans avoir à rechercher dans l'"histoire" de ces objets les causes de ces possibilités. Par exemple, pour discuter à plusieurs des propriétés d'une figure, pour dégager des propriétés, ou modifier les valeurs d'objets inaccessibles, il est utile que l'utilisateur puisse lire et modifier ces informations.

Dans les schémas précédents, l'utilisation des informations utiles seulement à l'affichage est schématisée par les flèches situées en dessous des noms des objets. Ces informations ne sont pas contenues dans le niveau d'abstraction choisi pour les descriptions actionnelles présentées.

Donc le langage à définir ne sera pas le langage directement induit des actions de l'utilisateur : certaines actions seront filtrées. Les actions qui ne produisent pas d'objet mais modifient leur apparence, sont signalées à l'utilisateur seulement par leurs effets, i.e. par l'information de l'évolution des valeurs analytiques ou graphiques des objets qu'elles affectent. Pour ce qui est des actions qui produisent des objets, on peut laisser à l'utilisateur le soin d'adapter le niveau d'abstraction du langage de géométrie formelle au degré d'abstraction de ses raisonnements.

2°) Forme à donner à l'information

Nous avons vu dans le §I-2-B-2° que différents profils d'utilisateurs devaient être pris en compte pour les EIAH. Nous avons vu aussi que, dans le cas du logiciel Cabri-géomètre, cette différenciation était particulièrement importante à cause de l'étendue de ses applications possibles (§I-4-1-A). En effet les utilisateurs vont de l'élève au chercheur en didactique, en passant pas le chercheur en mathématiques.

Pour la spécification de toute nouvelle fonctionnalité, il est nécessaire de tenir compte des motivations et des besoins de tous les utilisateurs.

Cependant, la motivation par rapport à la réalisation d'activités de programmation est différente selon les profils des utilisateurs. L'utilisateur chercheur est client d'une programmation automatique « de confort ». L'utilisateur professeur l'est aussi quand il s'agit de préparer ses exercices (comme les utilisateurs des environnements de CAO §I-2-C-2°). Par contre, s'il prépare un exercice basé sur les spécificités des macros, il s'intéresse à ce qu'est cette programmation, pour lui et ses élèves.

Toutes ces motivations différentes conduisent malgré tout à des besoins similaires, principalement pour trois raisons :

- la confrontation de tous les utilisateurs à des activités de programmation,
- leur culture mathématique,
- leur caractère « non-programmeur ».

Ainsi, tous les utilisateurs sont intéressés par l'intégration de capacités d'environnement de développement pour les aider dans la résolution des problèmes de programmation auxquels ils sont confrontés. Nous avons énuméré dans la première partie (§I-2-C-1°) les qualités désirées par les programmeurs pour les environnements de développement.

La culture mathématique commune des utilisateurs a une conséquence notable sur l'utilisation d'implicites. Ainsi, l'identification textuelle des objets est influencée par les usages concernant les noms des droites ou des segments. Ces usages dépendent des pays. Cette identification textuelle permet en particulier :

- de désigner des objets non définis explicitement comme les côtés d'un polygone ou la direction d'un vecteur, ou

- de nommer un objet sans lui avoir affecté de nom, comme une droite qu'on désigne grâce à deux points qui lui appartiennent, ou un segment, par les noms de ses extrémités.

a. Distance sémantique

Le langage de programmation proposé doit minimiser les apprentissages spécifiques requis. Pour limiter certains apprentissages non cruciaux (évitables), l'accès à la sémantique des programmes peut être facilité par le choix du langage et par l'animation introduite dans l'environnement. Ainsi, un des objectifs principaux de cette extension du logiciel à un environnement de programmation, est de minimiser l'adaptation des utilisateurs aux exigences de son langage (première qualité répertoriée dans le §I-2-C-1°).

La distance sémantique entre les manipulations de l'interface (donc le langage d'interface) et sa réalisation textuelle doit être minimisée, vu le type d'utilisateur attendu. Une méthode possible est de choisir des éléments de texte familiers à l'utilisateur pour présenter le contenu du programme.

Ainsi, pour le choix du lexique, nous avons extrait les mots qui apparaissent dans l'interface et ont un sens au niveau de la structure logique : les noms des outils sont exactement ceux qui figurent dans les boîtes à outils, les noms des objets sont ceux que l'utilisateur leur a donnés quand il les a nommés.

Pour le choix de la syntaxe, nous avons choisi une syntaxe tenant compte de l'ordre des manipulations de l'utilisateur et donc du mode opératoire. La spécification des paramètres des outils est accompagnée d'un sucre syntaxique pour préciser la manière dont ce paramètre sera pris en compte par l'outil. Il est constitué de la chaîne de caractères affichée sous le curseur pendant la construction active sur la figure graphique, et donnée comme retour d'information à l'utilisateur pour le guider dans l'utilisation des outils. Cette chaîne de caractères est construite dans la langue choisie par l'utilisateur, ce qui rapproche de l'expression naturelle de l'utilisateur (deuxième qualité répertoriée dans le §I-2-C-1°).

Par exemple, la construction d'un cercle de centre le point nommé « O » par l'utilisateur et passant par le point « A », est décrit par :

Cercle (« O » comme centre, passant par « A »)
--

Cette familiarité du lexique et de la syntaxe du langage de programmation permet l'économie de son apprentissage. La syntaxe du langage est automatiquement respectée, puisque les constructions (et donc le programme) ne sont introduites qu'au travers des manipulations permises par l'interface. Sa sémantique est directement accessible par l'effet immédiat de l'appel de chaque commande, et même avec "prévenance", puisque les constructions sont dirigées par les retours d'information proposés par le curseur.

b. Identification

Dans une figure géométrique dessinée par Cabri-géomètre, les objets sont identifiés par leur réalisation et désignés par leur dessin ou leur nom quand ils ont été nommés par l'utilisateur. Tous les objets ne sont pas visibles. L'utilisateur peut en avoir caché certains, d'autres peuvent être des objets internes à des appels de macros. L'utilisateur peut atteindre les objets cachés en changeant leur état, mais il ne peut pas atteindre les objets internes aux macros.

Dans le programme de construction, tous les objets sont accessibles. Ils sont identifiés depuis la figure géométrique comme décrit ci-dessus, ou depuis le programme par du texte, et donc par un nom. Donc tous les objets doivent être nommés. Chaque nom doit identifier un objet unique dans le programme, même si l'utilisateur a choisi de donner un même nom à deux objets distincts de la figure. Mais si l'utilisateur a nommé un objet dans la figure, ce nom doit lui permettre d'identifier aussi l'objet dans le programme. À défaut de la spécification du nom par l'utilisateur, l'éditeur doit lui "inventer" un nom.

Ainsi trois types de noms cohabitent : les noms inventés par le logiciel (noms automatiques), les noms donnés par l'utilisateur, accompagnés d'une différenciation en cas d'usage multiple, et les noms implicites. Cela conduit à des problèmes de cohabitation de syntaxes différentes.

Un moyen classique pour remédier à ce genre de problème, est de réserver l'usage d'un caractère particulier comme séparateur. Dans le cas de deux syntaxes, ce moyen n'est pas trop restrictif, mais avec trois syntaxes, le problème devient plus compliqué. La réflexion développée à ce sujet est présentée dans les problèmes spécifiques (§II-2).

Pour la suite de l'exposé de cette section, nous prendrons comme convention que les noms utilisateurs sont encadrés de doubles quotes et indicés à partir d'un deuxième usage, et que les noms automatiques sont indicés à partir de la première lettre du type de l'objet à nommer.

c. Le langage de Cabri-programmation

Le langage de géométrie formelle choisi est donc essentiellement induit des manipulations de l'interface pour les actions de construction qui ont permis de créer chacun des objets mémorisés par la liste parcourue par le noyau fonctionnel pour afficher toute la figure. Des notations secondaires sont introduites, déduites du retour d'information fourni par l'interface pour porter des informations sémantiques. Cette représentation est proche de la forme actionnelle, à deux choses près :

- les paramètres sont accompagnés d'un sucre syntaxique constitué de l'information textuelle fournie par le curseur afin d'indiquer la façon dont ils seront utilisés,
- les paramètres résultats des actions sont séparés des autres paramètres, afin d'être cohérent avec les manipulations effectuées par l'utilisateur : les résultats des actions ne sont pas sélectionnés par les utilisateurs avec les autres paramètres des outils.

Ainsi, une zone fournit la liste des constructions mémorisées par le logiciel et une zone représente la liste des objets construits. Cette deuxième zone contient, en fait, le déclaration des objets et de leurs types. La correspondance entre une action et son résultat est matérialisée par l'alignement horizontal d'un tableau. Ainsi le langage de géométrie formelle est un langage de couples.

L'exemple ci-dessous reprend la description de la construction de la figure donnée dans le paragraphe §II-1-A-1°-a.

Point	<u>point "I"</u>
Cercle ("I" comme centre)	<u>cercle "C"</u>
Point sur un objet (Sur "C")	<u>point "P"</u>
Droite (Par "I", et "P")	<u>droite D1</u>
Point	<u>point "G"</u>
Segment ("G", "I")	<u>segment S1</u>
Médiatrice (Ce 1er point "G", Ce 2ème point "P")	<u>droite D2</u>
Point(s) sur deux objets (D1, D2)	<u>point P2</u>
Cercle (P2 comme centre, passant par "P")	<u>cercle C2</u>

L'exemple ci-dessous reprend la description de la construction de la figure qui utilise la macro définie (§II-1-A-1°-b).

Point	<u>point "I"</u>
Cercle ("I" comme centre)	<u>cercle "C"</u>
Point	<u>point "G"</u>
cercle-TangentAUnCercle ("C", "G")	<u>point P1, point P2, cercle C2</u>

L'exécution de cette macro a conduit à la réalisation des actions décrites ci-dessous :

Point sur un objet (Sur "C")	<u>point</u> P1
Droite (Par "I", et P1)	<u>droite</u> D1
Médiatrice (Ce 1er point "G", Ce 2ème point P1)	<u>droite</u> D2
Point(s) sur deux objets (D1, D2)	<u>point</u> P2
Cercle (P2 comme centre, passant par P1)	<u>cercle</u> C2

Pour saisir un programme depuis un éditeur externe, les informations d'affichage non décrites par le langage formel peuvent être spécifiées dans une troisième colonne du tableau. Cette troisième colonne est composée de l'expression analytique de la liste des attributs graphiques et des états (punaisé, caché...) de l'objet décrit dans une ligne.

Pour spécifier le changement de colonne, on peut comme pour Excel, utiliser un séparateur paramétrable comme le caractère de tabulation (tab).

Le langage de triplets permet de décrire complètement les figures. Il est appelé le langage de Cabri-programmation.

Le programme ainsi représenté correspond à la liste des actions minimales qui permettraient à l'utilisateur de reproduire la même figure que celle qui est dessinée par le logiciel, dans son état courant.

L'ordre des actions de cette liste n'est véritablement contraint que par le pré-ordre issu du tri-topologique associé aux constructions des objets et par l'ordre des remplissages et recouvrement apparents des objets entre eux.

3°) Animation des programmes

Quels fonctionnements introduire dans l'environnement ?

De plus en plus, l'animation est utilisée dans les environnements de développement. Cette animation permet de fournir des informations sémantiques à l'utilisateur dans des artefacts éphémères. Elle est donc particulièrement importante pour des utilisateurs non-programmeurs. L'intérêt du caractère éphémère de ces artefacts est que, contrairement aux boîtes de dialogue, ils ne rompent pas l'impression d'engagement direct.

Nous avons montré dans le §I-2-C-1°, qu'on attend des environnements de développement qu'ils concrétisent la représentation des problèmes et qu'ils permettent de travailler sur des représentations à la fois détaillées et synthétiques. Ces points complètent les deux déjà abordés dans le §II-1-A-2°-a. L'animation de l'environnement peut contribuer à améliorer ces qualités.

a. Vues et navigation

L'animation permet de naviguer entre les différents niveaux d'abstraction qui portent chacun sur un niveau de détail adapté au travail de l'utilisateur. Elle permet à l'utilisateur d'utiliser des représentations à la fois détaillées et synthétiques, selon ses besoins.

Quelle forme de vue pour quelle information ?

Nous avons vu dans le Chapitre I-1 (tableau de tâches et activités), que selon les types d'activités, les tâches à effectuer diffèrent.

- Une activité d'exploration conduit à différentes tâches parmi lesquelles des tâches de construction, d'observation et de présentation, de résolution de boîtes noires.
- Une activité de modélisation conduit à des tâches de résolution de problèmes ouverts, de construction, de présentation puis d'expérimentation.

- Une activité de simulation (prise dans un sens différent de la modélisation par son rapport au sujet : simuler un phénomène n'est pas modéliser un objet) conduit à des tâches d'expérimentation et d'observation.
- Une activité de construction conduit à des tâches de programmation, d'investigation et de présentation.
- etc.

Les tâches ne sont pas toutes effectuées conjointement, même si, par exemple, des tâches de programmation peuvent être alternées avec des tâches de présentation.

Dans toutes ces activités et dans les tâches associées, les raisonnements effectués par l'utilisateur se situent à différents niveaux d'abstraction. Par exemple, pour les tâches de présentation, on peut modifier la forme de la figure ou son aspect.

Pour modifier la forme de la figure, un utilisateur doit accéder aux paramètres qui spécifient les valeurs des derniers degrés de liberté de la figure. En travaillant directement sur une vue géométrique, l'utilisateur voit immédiatement les effets de ses ajustements. En effet, il peut saisir les objets qui portent ces degrés de liberté et les déplacer. Grâce au concept de manipulation directe, la répercussion de ses actions est immédiate et « continue ».

Depuis le texte du programme, l'utilisateur doit aussi pouvoir modifier les valeurs « libres ». Pour cela, le logiciel doit l'informer d'une part des valeurs courantes, et d'autre part de la possibilité de les modifier. Mais, comme pour les attributs de présentation (couleur, épaisseur...), ces informations ne peuvent être prises en compte par la géométrie formelle, qui étant justement formelle, ne spécifie pas les valeurs effectives. De plus, on peut considérer ces modifications comme des ajustements de présentation, et les attributs correspondants doivent donc être accessibles par l'utilisateur en même temps et sur le même support que les attributs de présentation.

Depuis la figure, les objets non accessibles (internes aux macros) ne sont plus accessibles en dehors de la destruction d'un de leurs parents (objet dont ils dépendent). Par contre, les objets cachés peuvent être remis dans l'état « non-caché ». Un objet « à l'infini » comme le point d'intersection de deux droites parallèles, n'est accessible qu'en modifiant la propriété des objets dont il dépend, qui le met à l'infini (en supprimant le parallélisme des deux droites). Tous les objets sont accessibles depuis le programme de construction. Un dernier état permet d'agir sur les animations possibles de la figure, il s'agit de l'état « punaisé » / « dépunaisé ». L'ensemble de ces propriétés décrit l'état manipulatoire des objets.

Dans l'éditeur textuel, l'ensemble de ces informations sur chaque objet ne fait pas partie de la géométrie formelle, et leur accès dans le cadre des activités de programmation qui sont la source de la motivation de la vue programme, n'est que consultatif ou temporaire.

Ainsi, le prototype doit donner à l'utilisateur les moyens d'accéder à toutes les informations mémorisées par le logiciel sur chacun des objets géométriques, mais pas nécessairement en même temps.

Les différentes informations utiles pour une tâche particulière sont liées aux niveaux d'abstraction correspondant à la tâche.

Pour le cas des tâches de programmation, les différents niveaux d'abstraction sont les suivants :

- Résultat,
- Programme (du squelette à la liste complète des commandes),
- Expression analytique (algébrique),
- Apparence,
- État manipulatoire.

Un programme peut être manipulé selon différents niveaux d'abstraction. Par exemple, selon que l'utilisateur utilise Cabri-géomètre pour faire des figures à la règle et au compas, ou qu'il a défini des outils plus sophistiqués, lui permettant par exemple de tracer directement un carré connaissant un de ses côtés, ses raisonnements s'appuient sur un niveau d'abstraction plus élevé (dans l'exemple, une abstraction de la notion du carré). Aux deux extrémités, le programme peut être présenté sous la forme d'un squelette, ou il peut décrire la liste complète des objets construits par le logiciel. Nous avons vu aussi que l'utilisateur doit pouvoir décider de filtrer certains objets construits selon qu'ils ont ou non un sens vis-à-vis de l'utilisation qu'il fait de Cabri.

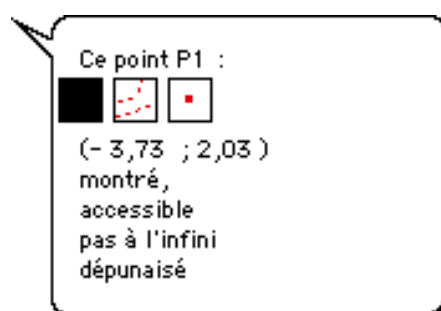
Le langage induit par les manipulations de l'interface doit décrire complètement la figure et les macros auxquelles elle fait appel, en proposant ces différents niveaux d'information grâce à différents supports de manipulation offerts par l'environnement :

- La vue programme de géométrie formelle, dans laquelle les objets géométriques sont spécifiés par les contraintes qui les lient sous la forme de propriétés géométriques, doit permettre une navigation hiérarchique dans les appels aux macros.
- La vue géométrique doit continuer à révéler le résultat de l'exécution du programme de géométrie formelle associé.
- La vue de présentation doit visualiser la description analytique de chaque objet (l'instance de son équation, exprimée en fonction des valeurs des objets de base de la figure), son apparence (les rendus des dessins des objets sont spécifiés par leurs attributs graphiques) et son état manipulateur.

Durées de vie des éléments d'interface

L'environnement de programmation doit permettre à l'utilisateur de naviguer entre ces différents niveaux de vue, de façon à accéder à toutes les informations utiles, et de la manière la plus adaptée à ses besoins. Le mode d'accès à chacune des vues doit être le moins coûteux possible pour l'utilisateur.

C'est pourquoi les informations sur le graphisme (analytique et apparence) peuvent être volatiles : dans les tâches de programmation, l'utilisateur a essentiellement besoin d'accéder à la structure logique du programme, et ce n'est qu'épisodiquement qu'il va devoir modifier l'apparence d'un objet ou ajuster « à la main » son expression analytique. Ainsi, un support adapté doit être éphémère comme une bulle d'aide, mais de plus éditable.



Par contre, les navigations entre les différents niveaux de la hiérarchie du programme (macros pliées/dépliées) doivent être plus durables : soit l'utilisateur travaille au niveau des outils de base ou à un niveau d'outils « fiables », soit il s'intéresse au contenu d'une macro pour savoir ce qu'elle fait ou la mettre au point. Ce choix doit donc être durable, et l'association d'un mode d'affichage à chaque appel de macro est adaptée. La présentation classique aujourd'hui d'un triangle à deux positions a l'intérêt, par la permanence de sa position, d'informer constamment l'utilisateur sur le mode utilisé. Cette notation secondaire peut être associée à une indentation, qui rappelle le niveau hiérarchique de chacune des lignes du programme.

L'ensemble de ces informations est présenté sous la forme suivante, qui est un exemple statique du texte de programme généré par notre éditeur textuel.

<p>Figure Point Cercle ("I" comme centre) Point cercle-TangentAUnCercle ("C", "G") > Point sur un objet (Sur "C") > Droite (Par "I", et P1) > Médiatrice (Ce 1er point "G", Ce 2ème point P1) > Point(s) sur deux objets (D1, D2) > Cercle (P2 comme centre, passant par P1) Fin de la figure</p> <p>Macro : cercle-TangentAUnCercle Objets initiaux> Objets finaux> Centre implicite (cercle #C1) > > Point sur un objet (Sur #C1) > Droite (Par #P1, et #P3) > Médiatrice (Ce 1er point #P2, Ce 2ème point #P3) > Point(s) sur deux objets (#D1, #D2) > Cercle (#P4 comme centre, passant par #P3) Fin de la macro : cercle-TangentAUnCercle</p>	<pre>-> point "I" -> cercle "C" -> point "G" -> point P1, point P2, cercle C2 -> point P1 -> droite D1 -> droite D2 -> point P2 -> cercle C2</pre> <pre>cercle #C1, point #P2, point #P3, point #P4, cercle #C2, point #P1, -> point#P3 -> droite#D1 -> droite#D2 -> point#P4 -> cercle#C2</pre>
--	---

b. Concrétisation des problèmes

Pour concrétiser les problèmes, le moyen privilégié est la synchronisation des animations. La synchronisation agit à deux niveaux : elle permet d'une part d'identifier les variables avec leurs instances, et d'autre part de mettre en relation les actions avec leurs effets.

Une variable prend un sens concret pour l'utilisateur lorsqu'il connaît l'objet qu'elle représente. Ainsi, il doit mettre en relation le nom et l'objet représenté par ce nom, et avoir une idée du comportement de cet objet.

En effet, dans le cas particulier de la géométrie formelle, l'animation du programme doit signaler à l'utilisateur la dépendance entre les objets. Elle doit permettre de répondre aux besoins spécifiques qui proviennent des particularités de la programmation de figures en géométrie dynamique. Un moyen pour cela est de matérialiser la sélection d'un objet dans la vue résultat et dans la vue programme, avec un rendu particulier dans le programme de façon à mettre en évidence toutes les actions dans lesquelles cet objet a une influence.

Par exemple quand on sélectionne un objet, dans l'éditeur graphique ou dans l'éditeur textuel indifféremment, toutes les utilisations de cet objet peuvent être affichées en couleur ou clignoter. Pour prévenir des conséquences de la destruction d'un objet, il faudrait même que les objets construits par des actions dépendant de l'objet sélectionné apparaissent récursivement dans un état « un-peu-sélectionné ».

La mise en relation des actions avec leurs effets peut être matérialisée par deux artefacts : le premier associe l'exécution de toute nouvelle action avec son résultat par synchronisation (association temporelle), le deuxième relie l'exécution pas à pas pilotée par le magnétophone de Cabri à un rendu dans le programme de construction.

L'animation permet de synchroniser l'exécution pas à pas du programme avec l'image de son résultat. L'exécution pas à pas pilotée par les boutons du magnétophone de Cabri doit être associée à un rendu textuel qui matérialise des déplacements de la position courante dans le parcours de la liste des actions.

L'animation permet de manipuler directement et indifféremment les objets du domaine ou leurs identificateurs, en concrétisant leur manipulation par une animation synchronisée du programme et du résultat. C'est pourquoi

- la sélection d'un objet depuis chacune des vues doit être matérialisée dans l'autre vue comme si l'objet avait été sélectionné dans celle-ci, même en cours de construction, c'est-à-dire pendant le choix des paramètres à associer à l'outil courant.
- La construction d'un nouvel objet est matérialisée dans la vue résultat et dans la vue programme simultanément.

B) Limites cruciales (espace/temps)

L'objectif de cette section est de recadrer les spécifications du prototype dans les limites liées aux contraintes matérielles de son existence effective.

Ces considérations ont pour conséquence de limiter le projet par son coût en espace et en temps, et par ses perspectives d'utilisation en tenant compte du contexte de l'existence commerciale de Cabri-géomètre.

Le dernier aspect, qui joue un rôle important pour les différents choix de conception, est la prise en compte de l'insertion du prototype dans le contexte d'un « projet » d'équipe. En effet, le logiciel Cabri-géomètre définit un cadre propice aux investigations dans des recherches nouvelles et plusieurs personnes travaillent simultanément à son évolution.

Examinons donc les contraintes liées au surcoût matériel, au produit commercial et enfin au projet Cabri-géomètre.

1°) Surcoût matériel

a. Espace

La version officielle de Cabri II tient sur une disquette (1,4 Mo), et une version existe sur une calculatrice, la TI-92. Ces qualités doivent être préservées : l'intégration de nouvelles fonctionnalités dans ce logiciel doit consommer un minimum de place mémoire supplémentaire. Pour estimer le surcoût tolérable, on peut s'appuyer sur quelques principes issus du sens commun.

L'augmentation du code due à l'intégration de nouvelles fonctionnalités dans un logiciel doit être en rapport avec l'importance de ces fonctionnalités dans l'étendue des fonctionnalités déjà supportées par le logiciel.

Dans le cas de Cabri, la programmation n'est pas le domaine pour lequel le logiciel a été conçu. Même si nous avons montré que des activités de programmation résultaient de l'utilisation de ce logiciel, son domaine d'application et le principal centre d'intérêt de ses utilisateurs est la visualisation de notions mathématiques ou la modélisation de phénomènes physiques par des outils mathématiques dynamiques.

Lors de l'intégration d'une nouvelle vue des données, le surcoût doit être réparti entre les différentes vues potentielles, mais l'éloignement des traitements nécessaires par rapports aux traitements déjà implémentés peut mener à un surcoût plus important. La vue structurelle constitue un nouvel accès aux données, déjà accessibles directement (pour la plupart) par la vue géométrique. Les traitements nécessaires à son intégration ne doivent pas prendre plus de place que les traitements purement géométriques.

L'estimation du surcoût toléré peut être basée sur un examen de la répartition des coûts dans le code du logiciel (en %). Le surcoût peut être estimé dans chaque rubrique, en fonction de l'éloignement des besoins à mettre en œuvre pour la vue structurelle, des traitements déjà codés.

Le tableau ci-dessous présente la répartition du code en fonction des fonctionnalités qu'il gère.

Répartition	Description	Code		Données
Librairies	récupération de code extérieur	115 Ko	10 %	24 Ko
Ressources	communication avec le système	0 Ko	0 %	4 Ko
Squelette d'IHM	Programme principal, gestion des menus, des fichiers, des fenêtres, des dialogues, des préférences, du défaire/refaire et du magnétophone	338 Ko	30 %	24 Ko
Gestion fine du graphique et de l'impression	écran avec l'« allumage » des pixels et l'équivalent papier	114 Ko	10 %	5 Ko
Gestion de la structure de donnée	noyau fonctionnel, constructions, initialisations, sauvegarde et sauvegarde textuelle	126 Ko	12 %	17 Ko
Calculs pour les objets géométriques « simples »	Objet	214 Ko	19 %	7 Ko
Coniques	Coniques	103 Ko	9 %	4 Ko
Calculatrice	Rpn	32 Ko	3 %	14 Ko
Texte, Table, Nom		32 Ko	3 %	5 Ko
Macros	définir, appliquer, sauvegarder	25 Ko	2 %	3 Ko
TI-92		18 Ko	2 %	3 Ko

Ainsi, le squelette devra prendre en charge la gestion d'une « sortie » supplémentaire en réponse à un nouvel item d'un menu. Les boutons du magnétophone devront être complétés par des boutons de gestion des déplacements dans le programme.

Le prototype aura besoin d'une gestion spécifique pour les supports de la vue textuelle : fenêtres et bulles d'aide. Les déplacements de la feuille sous la fenêtre sont à implémenter pour des feuilles de texte, car les feuilles déjà gérées sont uniquement graphiques. La gestion des ascenseurs, le zoom, etc. requièrent une gestion spécifique, aussi coûteuse que celle qui est déjà implémentée. Les transpositions textuelles des déplacements dans l'« historique » de la figure sont aussi entièrement nouvelles. La première version du prototype ne permet pas d'imprimer le programme généré.

L'affichage du texte préparé dans la fenêtre et des attributs graphiques conduit à un surcoût lié principalement à la spécificité des supports de ces « sorties ».

L'insertion de nouvelles structures de donnée est nécessaire pour mémoriser des informations sur la représentation textuelle des objets. Ces nouveaux supports d'information doivent permettre essentiellement de gérer les notations secondaires et de synchroniser les animations de la vue textuelle avec les manipulations directes des objets géométriques.

Le noyau fonctionnel devra être alourdi du traitement de l'identification des objets, de la préparation du texte du programme qui représente une figure, et de la construction dynamique du texte. De plus, pour les quelques types de données qui ont un comportement spécifique (les polygones, avec prise en compte de leurs côtés, les formules, les nombres munis d'unité, les textes et les tables dont les traitements spécifiques pour les coniques, la calculatrice et les objets « non-géométriques »), il est nécessaire de définir une transcription textuelle à supporter par le noyau fonctionnel.

La quantité de code liée au traitement des macros est a priori plus importante dans le prototype que dans la version officielle, puisque les macros sont aussi importantes que les figures dans le programme.

Pour ce prototype, nous n'avons pas considéré le portage sur la TI-92.

Cependant, la réalisation d'un prototype ne subit pas aussi strictement ces contraintes que la réalisation d'un produit. Les attentes du prototype peuvent être décomposées en plusieurs niveaux d'exigence sur le résultat. Le surcoût toléré est alors lié à ce niveau de qualité atteint.

En conséquence, les choix de mise en œuvre du prototype doivent permettre de

- récupérer tout investissement possible en recherchant dans le code existant si des procédures similaires à celles qui s'avèrent nécessaires ne sont pas déjà intégrées,
- minimiser les ressources spécifiques, par exemple en faisant appel principalement à des chaînes de caractères déjà mémorisées.

b. Temps

Cabri-géomètre gère un retour d'information pour guider l'utilisateur dans ses choix. Pour cette gestion, chaque donnée déjà introduite par l'utilisateur est considérée dans la boucle d'attente d'événements de l'IHM, de façon à calculer le retour d'information à lui associer.

L'introduction d'une nouvelle vue pour donner accès à ces données implique d'insérer de nouveaux calculs et traitements pour gérer la nouvelle forme des données.

En entrée, seule la vue active est concernée pour calculer si chacune des données est concernée par l'action courante. Par contre, en sortie, toutes les vues synchronisées doivent supporter le retour d'information qui leur est adapté, et ce pour chacune des données.

Par conséquent, toute insertion de retour d'information dans une vue synchronisée conduit à une augmentation du temps d'exécution de cette boucle, à cause du code introduit dans la boucle générale d'attente d'événements.

Le risque, dans le cadre de la programmation en C, est de voir apparaître un scintillement du curseur.

Les structures de données définies dans le logiciel sont optimisées pour le traitement de données géométriques. La gestion d'une autre forme d'affichage des données conduit à la gestion de nouvelles structures de données associées aux structures initiales et donc à un coût supplémentaire pour leurs traitements.

Cela conduit à un compromis entre la modification des structures porteuses des données et la mise au point de traitements plus compliqués. En effet, ces traitements devraient alors compenser l'inadéquation des structures initiales avec la forme des données nécessaire afin de les supporter efficacement.

Les risques encourus sont :

- une augmentation perceptible du temps de chargement du logiciel ou d'une figure,
- un temps de latence apparaissant avant la réalisation des rendus lorsque la vue textuelle est active.

Un moyen pour vérifier si le prototype répond bien à ces exigences, est de réaliser des bancs d'essais (BenchMark), c'est-à-dire des expériences qui dégagent des valeurs quantitatives (voir §III-3-A-1°).

2°) Prolongement commercial

Notre prototype doit conduire à des réalisations intégrées dans les prochaines versions commerciales du logiciel. Les utilisateurs sont habitués à un certain nombre de spécificités du logiciel, qui ont même contribué à leur choix de Cabri. Ces spécificités sont fondamentales et le prototype doit les respecter.

a. Même philosophie

Cabri est muni d'une interface épurée qui permet cependant une très large gamme de possibilités. Comme pour les jeux de société, les logiciels les plus intéressants sont le plus souvent ceux dont les règles sont simples et peu nombreuses, mais dont la combinatoire laisse la place à l'initiative la plus libre pour l'utilisateur.

Cette spécificité conduit à deux principes de conception pour le prototype :

- minimum de « boutons » supplémentaires,
- choix textuel non bloqué.

Le premier point a pour objectif de préserver l'interface épurée sans l'alourdir de choix permanents visibles en dehors quand ils n'ont pas de rapport avec l'activité en cours de l'utilisateur.

Le second point a pour objectif de laisser l'utilisateur libre de choisir le support qui lui convient le mieux pour effectuer les tâches à réaliser.

b. Mêmes prestations

Les prestations d'un logiciel correspondent à l'étendue des fonctionnalités qu'il propose, dans chacune des directions supportées par son noyau fonctionnel.

Font partie des prestations :

- la liste des outils associés au domaine,
- la liste des caractéristiques graphiques paramétrables,
- la liste des langues de dialogue des interfaces...

Si on intègre une nouvelle vue, son comportement doit être cohérent avec celui des autres vues, et par conséquent toutes les fonctionnalités supportées dans les autres vues et qui ont un sens pour la forme des données dans cette vue doivent y être étendues. La vue programme du prototype doit donc avoir un comportement équivalent au comportement des outils supportés par la vue résultat.

Ainsi, la gestion des attributs graphiques doit être maîtrisable depuis la vue programme, et le multilinguisme de l'interface doit être prolongé au niveau du texte du programme de construction.

c. Niveau de qualité des fonctionnalités

Il s'agit là encore principalement d'une conséquence de la préservation de la cohérence. Par exemple, pour la gestion du multilinguisme, les énoncés proposés par le logiciel sont proches de l'expression naturelle dans chacune des langues offertes. La désignation par des démonstratifs accordés, l'accord des adjectifs au genre des noms propres, l'ordre des mots, etc. respectent (dans une certaine mesure) les caractéristiques des langues. La qualité proposée dans la version officielle du logiciel doit être conservée dans le prototype.

Un autre point est que l'intégration d'une vue programme ne doit pas se faire au détriment de la qualité. Si la version officielle respecte subtilement les principes de la manipulation directe, le prototype ne doit pas alourdir les artefacts qui y contribuent (§II-C-2°-a). Il doit aussi respecter les principes de manipulation directe dans chacune des vues, et par conséquent proposer un correspondant satisfaisant pour la forme des données présentées.

3°) Insertion dans un projet

Différents travaux de maintenance, de recherche et de développements des nouvelles fonctionnalités, sont effectués en parallèle sur le logiciel par plusieurs développeurs et chercheurs, effectuant leurs travaux indépendamment les uns des autres. Il en découle un certain nombre de principes de conception parmi lesquels :

- sortir du code commun pour aller dans une zone personnelle de développement par appel de procédure,
- implémenter en prévoyant les évolutions ultérieures du logiciel, pour que les autres développeurs n'aient pas à se soucier de la maintenance de nos fonctionnalités nouvelles,
- réutiliser au maximum les procédures déjà écrites et maintenues,
- croiser les tests de toutes les procédures définies en définissant de nouvelles utilisations,
- préserver la portabilité en réutilisant du code déjà porté,
- prendre en considération l'avenir pour effectuer les choix,
- minimiser les choix irréversibles.

C) Influence de l'implémentation de Cabri

Quelle est l'influence de l'implémentation et des choix de conception du logiciel sur la spécification du prototype ?

Cabri-géomètre étant un produit commercial, il n'est pas possible de décrire trop précisément ses principes de conception et leur réalisation sans risquer de découvrir ses secrets d'implémentation. C'est pourquoi nous nous restreignons à quelques idées générales et aux points qui ont un rapport direct avec nos besoins de développement.

1°) Comment Cabri est-il structuré ?

Cabri est une IHM, donc le programme principal est basée sur une boucle qui attend les événements, gère ceux qui la concernent, et répercute leurs effets sur les données (§I-2-A-4°-b).

Cabri est codé en C, avec passage de procédures par les champs des structures de données. Cette organisation permet d'associer des comportements aux données et ainsi de les rapprocher du concept d'objet (§I-2-A-3°-a). Plusieurs procédures sont ainsi associées à chaque objet, dont : création, affichage, distance (entre l'objet et la position du curseur) et destruction. Les créations et les destructions sont génériques : elles sont identiques pour tous les objets puisqu'une seule et même structure de données permet de décrire tous les objets.

Lorsqu'un objet est créé, plusieurs niveaux de mémorisation permettent de gérer les animations. La classe de l'objet dépend de la contrainte qui le relie aux autres objets : par exemple une droite perpendiculaire à un autre objet. La méthode de construction de cet objet (parmi les constructions possibles d'objets dans cette classe) dépend du type des objets dont il dépend effectivement. Par exemple, dans le cas d'une droite perpendiculaire à un objet, la méthode dépend du fait que cet autre objet est un segment ou un côté d'un triangle. Ce concept de méthode précisant une classe permet de supporter le polymorphisme des outils de Cabri : les classes de construction ont pour rôle d'identifier la méthode de construction adaptée au choix des paramètres effectué par l'utilisateur.

Au niveau de la classe de construction, on mémorise :

- toutes les méthodes possibles pour l'outil courant,
- la procédure de sélection des objets sur lesquels l'outil courant peut être appliqué,
- la gestion des retours d'information fournis à l'utilisateur pour l'aider dans son choix des objets à sélectionner.

La création d'un objet consiste en :

- la mémorisation de la classe de construction dont il est issu,
- l'identification de la méthode de construction et la mémorisation de son identificateur parmi les méthodes mémorisées dans la classe,
- la mémorisation de la liste des constituants de l'objet, c'est-à-dire des objets par rapport auxquels cet objet est effectivement contraint.

La méthode de construction permet de calculer les valeurs des paramètres algébriques de l'objet. Ces paramètres algébriques sont : dans le cas d'un point, ses coordonnées, dans le cas d'une droite, les coefficients de son équation, etc. Ils sont obtenus par résolution des contraintes décrites par la méthode mémorisée et utilisation des paramètres algébriques des constituants de l'objet. Par exemple, les coefficients de l'équation d'une droite sont déterminés en fonction des coordonnées des deux points qui la définissent, en résolvant le système d'équations associé :

Si les deux points sont A (x_A, y_A) et B (x_B, y_B),
 et si la droite a pour équation $ax + by - c = 0$,
 alors le système obtenu est le classique système de deux équations à deux inconnues :

$$\begin{cases} a x_A + b y_A = c \\ a x_B + b y_B = c \end{cases}$$

Le calcul effectué pour résoudre ce système met à jour les valeurs caractéristiques de la représentation graphique de l'objet. Ainsi, c'est au niveau de la méthode de construction que sont mémorisées les informations utiles pour paramétrer la procédure de sélection des objets, la procédure qui reconnaît que deux objets sont identiques (au sens que les contraintes qui les définissent sont basées sur les mêmes objets), mais aussi les procédures utilisées pour les animations.

Dans l'exemple ci-dessus, les coefficients de la droite sont obtenus (dans le cas général, on peut se ramener à $c=1$) en fonction des coordonnées du point A, fixe, et du point B, animé, par :

$$a = \frac{y_A - y_B}{x_B y_A - x_A y_B}$$

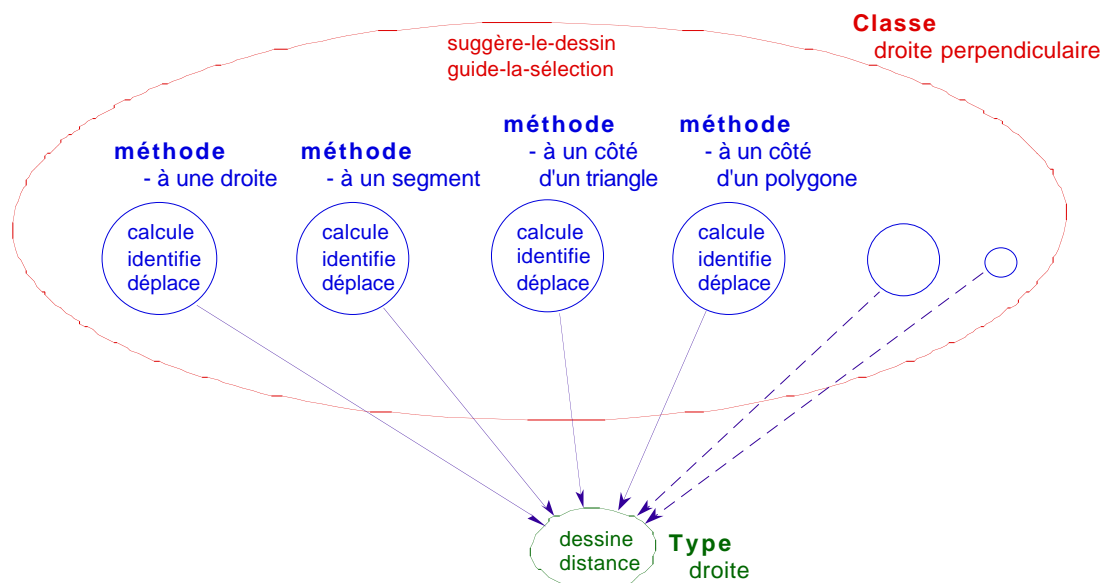
$$b = \frac{x_B - x_A}{x_B y_A - x_A y_B}$$

Remarque : dans l'exemple ci-dessus, nous n'avons pas considéré les cas particuliers qui posent problème, car le but de cette explication est seulement de donner une idée des principes d'implémentation du logiciel.

La création d'un objet est immédiatement suivie d'un premier affichage de l'objet dans l'état courant de la figure telle qu'elle est au moment de la création, avec

- application de la méthode de construction aux objets dont il dépend, avec leurs positions et leurs formes courantes, pour obtenir les valeurs algébriques de l'objet,
- mémorisation de ces valeurs, caractéristiques de l'objet, avec son type,
- utilisation de ces valeurs pour l'affichage, qui ne dépend plus que du type de l'objet.

Ensuite, à chaque tour de la boucle principale d'attente d'événements, la distance entre la position du curseur et l'objet, est calculée en fonction des valeurs algébriques. Ce calcul ne dépend aussi que du type de l'objet. Lorsque cette distance est inférieure à un seuil, l'objet en question est considéré comme « survolé » par le curseur, et le logiciel génère le retour d'information associé à cet objet pour l'outil courant.



Ainsi, l'affichage de l'objet et le calcul de sa distance à tout point de la feuille sont définis par des procédures que ne dépendent que de ses valeurs caractéristiques et du type de l'objet concerné. De même, la manière dont un objet en contraint un autre ne dépend que de ses valeurs et de son type, à travers une méthode de construction. Ainsi, les procédures de dessin et de calcul de distance sont associées aux types des objets.

Cabri-géomètre est très bien adapté aux objets qu'il permet de manipuler : graphiques et même géométriques. Ses structures de données sont conçues spécialement pour gérer la manipulation directe des données et permettre la géométrie dynamique.

2°) Conséquences sur l'insertion de l'édition textuelle dans l'existant

Ajouter une vue textuelle revient à ajouter d'autres modes d'affichage et d'édition à des données déjà existantes, de façon à pouvoir exercer les différents modes d'affichage et de saisie indifféremment depuis la vue programme ou depuis la vue résultat. De plus, l'éditeur textuel doit permettre de modifier la présentation du programme.

L'objectif est de garder les principes de développement, mais de les appliquer à la manipulation d'autres formes de données que les données du domaine, même si les techniques de manipulation qui sont efficaces pour ces nouvelles formes de donnée sont radicalement différentes des précédentes (existantes).

a. Quelles sont les manipulations attendues ?

En tant qu'éditeur textuel, le prototype doit permettre d'accéder aux données et au programme, et guider l'utilisateur dans sa saisie du programme. En tant qu'environnement de programmation, il doit de plus lier la saisie du programme à son exécution, et permettre de revoir la construction pas à pas.

Accès aux données

D'une manière générale, lorsqu'un utilisateur manipule des données, deux motivations peuvent intervenir dans ses actions : agir sur la présentation et agir sur les données elles-mêmes. La vue textuelle doit gérer ces deux aspects pour elle-même et « à distance » pour la vue résultat.

Accès aux objets géométriques

L'édition graphique d'un point est effectuée par sélection directe d'un point de la fenêtre guidée par le retour d'information fourni par le curseur. Un point déjà construit peut être saisi et déplacé quand il possède au moins un degré de liberté. Il possède deux degrés de liberté s'il est un point libre et un degré de liberté s'il est un point sur un autre objet. On peut alors dire qu'il peut être animé. Dans les autres cas, le point est lié et contraint par les autres objets. La mise à jour de ses valeurs algébriques est gérée par la méthode de construction.

Le dessin d'un point constitue son identificateur géométrique, le logiciel reconnaît que le curseur le désigne lorsque la distance entre la position du curseur et celle du dessin est inférieure à un certain seuil. Lorsque deux points sont situés au même endroit (points confondus dans l'état de la figure représentée par le dessin), l'ambiguïté est résolue par le retour d'information fournie sous le curseur : le logiciel affiche la chaîne : « quel objet ? » et déroule le menu constitué des différentes possibilités si l'utilisateur appuie sur la souris. La sélection et le choix se font au moment du relâchement de la souris.

Dans l'éditeur textuel, un point et son identificateur textuel sont dissociés et l'identificateur apparaît en différents endroits du programme : une fois dans la liste des objets construits, et autant de fois qu'il est utilisé pour d'autres constructions, dans la liste des constructions mémorisées par le logiciel. On parle d'ubiquité de l'identificateur et de l'objet dans la vue programme, et entre la vue géométrique et la vue programme. L'identificateur textuel est « multiple ». L'éditeur textuel doit reconnaître que le curseur désigne l'objet lorsqu'il survole n'importe quelle occurrence de son identificateur, c'est-à-dire que la position du curseur est contenue dans une zone « silhouette » d'une des occurrences de l'objet.

Pour que la vue textuelle soit dynamique, un point doit pouvoir être construit depuis l'éditeur textuel. L'utilisateur doit pouvoir entrer, par saisie au clavier, les paramètres algébriques « libres » des objets animables. Depuis la vue textuelle, le schéma de fonctionnement des outils de construction peut d'abord être identique à celui de la vue géométrique : l'objet est créé par l'action de l'outil sur des objets sélectionnés dans le programme (avec le même guide fourni par le curseur), le texte du programme de construction s'affiche. Mais ensuite, avec l'affichage des objets construits, plusieurs variantes sont envisageables.

- Soit le logiciel présente directement la bulle de présentation des informations graphiques (cf. §II-1-A-3°-a) et la bloque en attendant la saisie au clavier des paramètres algébriques non calculables. Ce blocage provoque une rupture d'engagement direct, un peu comme dans Géoplan, mais moins forte si la bulle est reliée à l'objet qu'elle décrit, et si elle ne cache pas l'occurrence de définition de l'objet et cache le moins possible de texte du programme.
- Soit il initialise les paramètres algébriques en fonction de la position du curseur, comme si la fenêtre textuelle était la fenêtre géométrique. Mais cela conduit à un décalage entre le rendu géométrique et le mouvement du curseur.
- Soit il initialise les paramètres algébriques avec des valeurs aléatoires (en laissant les objets accessibles dans la fenêtre), ce qui ramène le problème de l'initialisation à celui de la modification, et qui donc peut constituer un choix temporaire de développement.

La position de l'objet géométrique doit être modifiable au clavier en entrant des valeurs nouvelles pour ses coordonnées : depuis les bulles de présentation, depuis la troisième colonne du programme quand l'utilisateur l'a affichée, ou depuis la vue géométrique. Le fait que les valeurs soient ou non modifiables doit être signalé à l'utilisateur, par exemple en utilisant la convention des menus : les valeurs modifiables en noir, les valeurs fixées en gris.

Ainsi, le prototype doit permettre d'éditer les paramètres algébriques (et de la même manière les attributs graphiques). Nous choisissons la troisième solution proposée (valeurs aléatoires) avec les possibilités de modifications depuis les bulles d'aide. Cependant, nous n'avons eu le temps d'implémenter cette fonctionnalité.

Présentation du programme et notations secondaires

L'utilisateur doit aussi pouvoir modifier la position d'une occurrence de l'identificateur, par exemple en insérant un passage à la ligne ou en modifiant l'ordre de la construction.

Lors des appels de macros en particulier, la liste des objets construits peut être trop longue pour tenir sur une ligne. L'éditeur textuel doit donc permettre d'insérer des retours à la ligne. Mais lorsqu'il plie et déplie une macro, il doit retrouver son programme dans l'état de présentation dans lequel il l'avait mis.

Il ne s'agit pas de notations secondaires, puisque celles-ci ont pour objet d'insister sur la syntaxe et la sémantique du texte du programme, et doivent donc être générées par le prototype. Il s'agit plutôt d'ajustements ponctuels de présentation.

Dans le formalisme textuel choisi, le retour à la ligne n'a de valeur sémantique qu'en fin de description de construction. Dans la partie gauche du texte, la fin de la primitive est alignée sur la liste des déclarations des objets qu'elle construit, contenue dans la partie droite du texte. Dans la partie droite du texte, la fin de la liste des objets construits est marquée par le début de la partie gauche de la ligne suivante, qui décrit la construction suivante.

L'artéfact qui contient l'information sémantique est donc l'alignement et non le simple retour-charriot.

Accès à la structure du programme

L'outil de redéfinition des contraintes agit sur l'ordre des constructions (§II-1°-A). Mais cette évolution n'est pas maîtrisable par l'utilisateur. Or, le remplissage des objets fermés (polygones, cercles) est actuellement consécutif à cet ordre dans Cabri. Cet ordre est déterminant pour les recouvrements des objets. Il n'y a pas actuellement de gestion séparée des remplissages et des contraintes de constructions.

L'éditeur textuel doit donner accès aux modifications possibles de cet ordre, c'est-à-dire aux modifications qui respectent les dépendances entre les objets issus des contraintes et l'ordre des remplissages.

Si, ultérieurement, Cabri permet de gérer séparément les contraintes topologiques et les contraintes de remplissage, le paradigme de programmation sous-jacent à l'utilisation de Cabri pourra devenir plus déclaratif. En effet, la déclaration d'un objet, c'est-à-dire l'élément du programme de la deuxième colonne qui déclare un objet, pourra alors remonter dans l'historique de la construction en entraînant avec elle la partie droite correspondante, jusqu'à la dernière primitive qui a servi à définir un objet dont dépend la primitive déplacée. De même, elle ne pourra descendre que jusqu'à la première commande basée sur un des objets qu'elle construit. Les déplacements possibles seront beaucoup plus libre que dans l'état actuel du logiciel où l'ordre des remplissages doit aussi être respecté.

Dans les éditeurs textuels habituels, il est conventionnel que les éléments de texte sélectionnés soient affichés en inversion vidéo. De plus, lorsque l'utilisateur laisse la souris appuyée, il peut déplacer le bloc de texte sélectionné et le lâcher quand la position proposée par le logiciel lui convient.

Ensuite, deux possibilités sont envisageables :

- Soit le retour d'information proposé par le logiciel bloque le déplacement sur la dernière position possible, (celle qui précède l'utilisation de l'objet construit comme objet initial de la commande),
- Soit après un court délai, le déplacement de la ligne se met à entraîner le déplacement des lignes filles.

En pratique, l'utilisateur pourra régler lui-même le délai infini pour le premier cas, sinon court, moyen ou long.

Guider l'utilisateur

Le prototype doit associer étroitement un éditeur textuel aux manipulations de l'interface. Élément essentiel de l'environnement de programmation ainsi constitué, non seulement l'éditeur textuel doit être un éditeur syntaxique (§I-3-A-3°-b), mais aussi il doit être muni d'effets visuels pour aider l'utilisateur à assimiler le paradigme de programmation sollicité. Ces artefacts sont attendus pour les constructions de nouveaux objets, les destructions d'objet, les redéfinitions de contraintes et les définitions de macros.

Construction

L'éditeur textuel doit adapter les retours d'information fournis dans la vue géométrique pour guider l'utilisateur dans son choix des objets sur lesquels il peut appliquer chaque outil.

Pour cela, le texte du programme peut être construit au fur et à mesure des sélections, en proposant de considérer l'objet survolé de la manière indiquée sous le curseur. L'utilisateur valide la proposition en sélectionnant l'objet survolé, et l'éditeur passe au paramètre suivant de l'action associée à l'outil.

En cas de renoncement à la construction entamée, l'éditeur textuel doit effacer toute la ligne de commande.

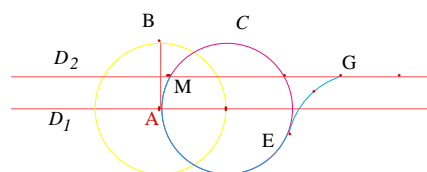
Pour informer de façon permanente l'utilisateur de l'état « en cours », et donc immédiatement réversible ou abandonnable, la ligne de programme correspondante peut être affichée dans une autre couleur.

Des constructions doivent pouvoir être insérées. Il faut donc que l'utilisateur puisse se replacer entre deux éléments de la liste des constructions. Les constructions précédentes doivent être accessibles, mais pas les suivantes. L'utilisateur doit être informé de cet état des constructions, sans pour autant perdre la suite du programme. Pour cela, l'éditeur textuel peut utiliser des couleurs.

Destruction

Contrairement à un éditeur textuel classique (pas syntaxique), l'effacement d'éléments de texte peut avoir des conséquences éloignées de la zone de texte manipulée. L'éditeur textuel doit mettre en évidence les objets qui seront détruits en propageant par exemple l'affichage en couleur de toutes les occurrences de leurs identificateurs à travers la relation de dépendance.

L'illustration ci-dessous reprend l'exemple de la destruction du point C de la figure du §I-1-B-1°.



Le premier tableau représente le programme de cette construction.

Figure	
> Point	-> point "A"
> Point	-> point "B"
> Segment ("A", "B")	-> segment S1
> Droite perpendiculaire (Par "A", Perpendiculaire à S1)	-> droite "D1"
> Cercle ("A" comme centre, passant par "B")	-> cercle C1
> Point(s) sur deux objets ("D1", C1)	-> point P1
> Cercle (P1 comme centre, passant par "A")	-> cercle "C"
> Médiatrice (Médiatrice de S1)	-> droite "D2"
> Point sur un objet (Sur "C")	-> point "E"
> Point(s) sur deux objets ("D2", "C")	-> point "M"
> Point(s) sur deux objets ("D2", "C")	-> point P2
> Symétrie centrale (Symétrique de "M", par rapport à ce centre P2)	-> point P3
> Milieu (Milieu de P3, et de P2)	-> point "G"
> Arc (Par "M", "A" intermédiaire, et "E" final)	-> arc A2
> Point	-> point P4
> Arc (Par "E", P4 intermédiaire, et "G" final)	-> arc A3
Fin de la figure	

Le deuxième tableau met en évidence les constructions affectées par la destruction du point C.

Figure	
> Point	-> point "A"
> Point	-> point "B"
> Segment ("A", "B")	-> segment S1
> Droite perpendiculaire (Par "A", Perpendiculaire à S1)	-> droite "D1"
> Cercle ("A" comme centre, passant par "B")	-> cercle C1
> Point(s) sur deux objets ("D1", C1)	-> point P1
> Cercle (P1 comme centre, passant par "A")	-> cercle "C"
> Médiatrice (Médiatrice de S1)	-> droite "D2"
> Point sur un objet (Sur "C")	-> point "E"
> Point(s) sur deux objets ("D2", "C")	-> point "M"
> Point(s) sur deux objets ("D2", "C")	-> point P2
> Symétrie centrale (Symétrique de "M", par rapport à ce centre P2)	-> point P3
> Milieu (Milieu de P3, et de P2)	-> point "G"
> Arc (Par "M", "A" intermédiaire, et "E" final)	-> arc A2
> Point	-> point P4
> Arc (Par "E", P4 intermédiaire, et "G" final)	-> arc A3
Fin de la figure	

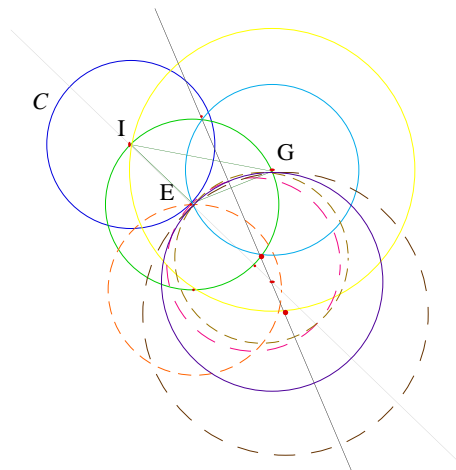
Le troisième tableau contient le programme après destruction.

Figure	
> Point	-> point "A"
> Point	-> point "B"
> Segment ("A", "B")	-> segment S1
> Droite perpendiculaire (Par "A", Perpendiculaire à S1)	-> droite "D1"
> Cercle ("A" comme centre, passant par "B")	-> cercle C1
> Point(s) sur deux objets ("D1", C1)	-> point P1
> Médiatrice (Médiatrice de S1)	-> droite "D2"
> Point	> point P4
Fin de la figure	

Définition de macros

Des effets visuels doivent visualiser par simulation le processus suivi par le logiciel pour extraire les commandes de la macro. Le prototype doit permettre à l'utilisateur de remonter dans la dépendance des objets construits depuis les objets terminaux et en s'arrêtant sur les objets initiaux. Les constructions ainsi parcourues doivent rester affichées en couleur avant d'être recopiées une à une dans le corps de la macro.

L'illustration ci-dessous reprend l'exemple du §I-1-B-1°-b.



Le tableau ci-dessous représente le programme de la construction de cette figure, déjà illustré par le graphe du § I-1-B-1°-c.

Figure	
> Point	-> point "I"
> Cercle ("I" comme centre)	-> cercle "C"
> Point	-> point "G"
> Point sur un objet (Sur "C")	-> point "E"
> Segment ("E", "I")	-> segment S1
> Segment ("I", "G")	-> segment S2
> Droite (Par "I", et "E")	-> droite D1
> Cercle ("E" comme centre, passant par "G")	-> cercle C2
> Cercle ("G" comme centre, passant par "I")	-> cercle C3
> Point(s) sur deux objets (C2, C3)	-> point P1
> Cercle (P1 comme centre, passant par "E")	-> cercle C4
> Point(s) sur deux objets (D1, C2)	-> point P2
> Cercle (P2 comme centre, passant par "E")	-> cercle C5
> Segment ("E", "G")	-> segment S3
> Cercle ("G" comme centre, passant par "E")	-> cercle C6
> Point(s) sur deux objets (C2, C6)	-> point P3
> Cercle (P3 comme centre, passant par "E")	-> cercle C7
> Point(s) sur deux objets (C2, C6)	-> point P4
> Droite (Par P4, et P3)	-> droite D2
> Point(s) sur deux objets (D2, C3)	-> point P5
> Cercle (P5 comme centre, passant par "E")	-> cercle C8
> Médiatrice (Médiatrice de S3)	-> droite D3
> Point(s) sur deux objets (D1, D3)	-> point P6
> Cercle (P6 comme centre, passant par "E")	-> cercle C9
Fin de la figure	

Dans le texte du programme de cette construction, nous avons mis en évidence les constructions à extraire à l'aide de rectangles de « sélection », avec la même convention de couleurs que pour le graphe du § I-1-B-1°-c. Les flèches du graphe trouvent donc comme transposé textuel ces rectangles de « sélection ». Le résultat est constitué des constructions sélectionnées en rose. Le contenu de macro est décrit dans le tableau suivant :

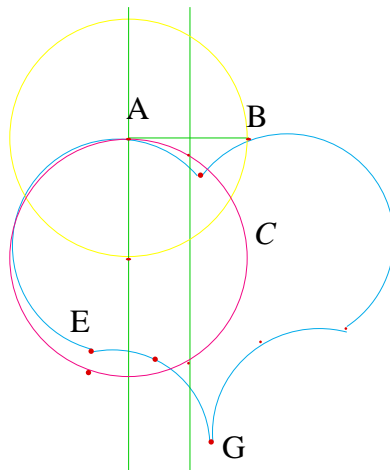
Macro : cercleTgt	
Objets initiaux >	point #P1, cercle #C1
Objets finaux >	point #P3, point #P4, cercle #C2
Centre Implicite >	point #P2
> Point sur un objet (Sur #C1)	-> point #P3
> Droite (Par #P2, et #P3)	-> droite #D1
> Segment (#P3, #P1)	-> segment #S1
> Médiatrice (Médiatrice de #S1)	-> droite #D2
> Point(s) sur deux objets (#D1, #D2)	-> point #P4
> Cercle (#P4 comme centre, passant par #P3)	-> cercle #C2
Fin de la macro : cercleTgt	

Redéfinition des contraintes

Deux aspects doivent être implémentés :

- Comme pour la définition des macros, le prototype doit visualiser par simulation le processus suivi. Lorsqu'un objet a été sélectionné pour être redéfini, tous les objets qui en dépendent doivent être affichés en gris clair, selon la convention de désactivation des items d'un menu, et le curseur ne doit « réagir » que lorsque des objets pertinents sont survolés.

- Il faut visualiser par simulation le processus suivi par le logiciel pour redéfinir les contraintes directement depuis la vue textuelle en manipulant les données ou à partir d'un outil, et le logiciel doit proposer l'échange s'il est pertinent. Pour éviter que l'état de la construction ne change de façon inattendue pendant une modification, l'éditeur textuel doit informer l'utilisateur du changement d'état de la construction en cours de modification, tant que la modification n'est pas complète.



Pour illustrer ce schéma de fonctionnement, nous reprenons l'exemple de la redéfinition du point E de la troisième image du « film » du §I-1-A-2°-b.

Figure	
> Point	-> point P1
> Point	-> point "A"
> Point	-> point "E"
> Arc (Par P1, "A" intermédiaire, et "E" final)	-> arc A2
> Point	-> point P2
> Point	-> point "G"
> Arc (Par "E", P2 intermédiaire, et "G" final)	-> arc A3
> Point	-> point P3
> Point	-> point P4
> Arc (Par "G", P3 intermédiaire, et P4 final)	-> arc A4
> Point	-> point "B"
> Arc (Par P4, "B" intermédiaire, et P1 final)	-> arc A5
> Segment ("A", "B")	-> segment S1
> Médiatrice (Médiatrice de S1)	-> droite D1
> Cercle ("A" comme centre, passant par "B")	-> cercle C1
> Droite perpendiculaire (Par "A", Perpendiculaire à S1)	-> droite D2
> Point(s) sur deux objets (D2, C1)	-> point P5
> Cercle (P5 comme centre, passant par "A")	-> cercle "C"
> Point(s) sur deux objets (D1, "C")	-> point P6
> Point(s) sur deux objets (D1, "C")	-> point P7
> Point sur un objet (Sur "C")	-> point P8
Fin de la figure	

Le tableau ci-dessous représente le programme de la construction de cette figure, déjà illustré par le graphe du § I-1-B-1°-b. Le point E doit être redéfini en le point P8.

L'effet de la redéfinition conduit au programme suivant :

Figure	
> Point	-> point P1
> Point	-> point "A"
> Point	-> point P2
> Point	-> point "G"
> Point	-> point P3
> Point	-> point P4
> Arc (Par "G", P3 intermédiaire, et P4 final)	-> arc A4
> Point	-> point "B"
> Arc (Par P4, "B" intermédiaire, et P1 final)	-> arc A5
> Segment ("A", "B")	-> segment S1
> Médiatrice (Médiatrice de S1)	-> droite D1
> Cercle ("A" comme centre, passant par "B")	-> cercle C1
> Droite perpendiculaire (Par "A", Perpendiculaire à S1)	-> droite D2
> Point(s) sur deux objets (D2, C1)	-> point P5
> Cercle (P5 comme centre, passant par "A")	-> cercle "C"
> Point(s) sur deux objets (D1, "C")	-> point P6
> Point(s) sur deux objets (D1, "C")	-> point P7
> Point sur un objet (Sur "C")	-> point P8
> Arc (Par P8, P2 intermédiaire, et "G" final)	-> arc A2
> Arc (Par P1, "A" intermédiaire, et P8 final)	-> arc A3
Fin de la figure	

L'effet visuel à implémenter dans le prototype est de donner l'illusion que la construction de l'objet à redéfinir pousse toutes les constructions d'objets qui en dépendent, et finit par se superposer à celle de l'objet qui le redéfinit. Le remplacement des identificateurs dans les dépendances peut être effectué après un court délai.

Dans le cas de la manipulation directe du texte, le remplacement des identificateurs est effectué après que l'utilisateur a relâché la souris.

Revoir la construction pas à pas

L'éditeur textuel doit être accompagné de fonctionnalités d'environnement de développement. Parmi ces fonctionnalités se trouve l'exécution pas à pas. Le logiciel fournit déjà la possibilité de revoir la construction de la figure par l'action des boutons d'un magnétophone. Il s'agit donc, dans le prototype, de définir son correspondant textuel.

Une des raisons d'être du prototype est d'aider l'utilisateur à assimiler les relations entre les différents éléments de sa figure. Si l'utilisateur éprouve la nécessité de revoir sa construction, c'est justement pour se remémorer ces relations. Il s'agit donc, pendant l'exécution pas à pas, de visualiser ces liens. Une possibilité est de simuler la sélection des objets, c'est-à-dire de produire le rendu généré par le logiciel pour la construction de cette figure. Il ne s'agit cependant pas de reproduire le film de la construction effectuée, mais seulement ses animations utiles.

b. Conséquences sur le code à insérer

Ce qu'il contient

Pour obtenir une vue programme dynamique et synchronisée avec la vue résultat, il faut ajouter quelques traitements supplémentaires et « doubler » quelques appels de procédures spécifiques de la vue géométrique par les appels à leurs transpositions textuelles.

Il s'agit :

1. de préparer et afficher le texte du programme qui décrit la construction des objets, en récupérant les éléments de texte affichés par l'interface dans la vue résultat,
2. de guider l'utilisateur et définir un retour d'information adapté à la saisie du programme, pendant les constructions et les destructions d'objets,
3. de repérer quand une occurrence est survolée et gérer la demande d'affichage des attributs,
4. de gérer l'affichage et les modifications des attributs graphiques, algébriques et manipulatoires,
5. de gérer un curseur textuel lié aux manipulations du magnétophone, et à l'insertion de constructions,
6. de déplacer des constructions « à la main » dans le programme (redéfinition de contrainte),
7. de simuler visuellement le processus de validation de macros,
8. de permettre l'introduction d'ajustements ponctuels de présentation du programme,
9. de permettre deux modes d'affichages pour les appels des macros : plié et déplié.

Où l'insérer

Préparer et afficher

Les procédures qui préparent les données pour dessiner les objets dans la vue géométrique dépendent des méthodes de construction. Dans le cas du texte du programme, la classe de construction mémorise l'outil, et la méthode l'information sémantique associée à chaque paramètre. Cependant, la préparation du texte du programme (1-a) ne dépend de la méthode et même de la classe que comme d'un paramètre. Les textes qui décrivent les objets sont construits selon le même schéma, quel que soit le type des objets.

Les procédures qui dessinent les objets géométriques sont spécifiques du type de l'objet concerné (schéma du §II-1-B-1°), mais pas leurs transposées textuelles (1-b). Ces dernières doivent juste afficher le texte préparé pour chaque objet, dans son contexte du programme complet, c'est-à-dire dans l'ordre induit par l'historique de la construction, et en prenant en compte les notations secondaires à produire pour les objets internes aux macros.

De plus, lorsque des objets sont issus de macros, le prototype doit aussi afficher le contenu des macros utilisées.

Guider

Pour guider la sélection (2), les traitements pour l'édition textuelle peuvent être introduits dans la procédure qui est attachée à la classe de construction. Cela permet de récupérer et d'adapter la chaîne générée pour la vue géométrique.

Repérer

La procédure qui calcule la distance était utilisée pour comparer cette distance à un seuil et décider si le curseur survole un objet. Dans le cas du texte (3), elle doit être remplacée par une procédure différente, puisque l'objet y est « multiple ».

Attributs graphiques

L'affichage des attributs graphiques, algébriques et manipulateurs (4) doit répondre à un appel classique d'aide. Dans beaucoup de logiciels sur Macintosh (sous Mac OS), il suffit à l'utilisateur de cliquer deux fois sur la représentation de l'objet sur lequel il veut obtenir des informations. Les bulles peuvent donc être accessibles depuis la sélection des objets. Le double clic permet de différencier la sélection simple de la demande d'informations. Les traitements correspondants doivent donc être insérés dans les procédures qui gèrent les sélections d'objets, au niveau de la boucle principale d'attente d'événements.

La procédure « déplace » est transposée en une modification des attributs algébriques. Son correspondant textuel dépend du type de l'objet, puisqu'un segment par exemple n'est animable, donc modifiable, que par ses points d'extrémité.

L'état de ce texte est même dépendant de la méthode de construction. Par exemple, le point libre est toujours modifiable et ses coefficients affichables, alors que le point d'intersection n'est jamais modifiable. Cependant, ses coordonnées peuvent être affichées pour information.

Un autre exemple est constitué par les méthodes associées à la construction du cercle. Lorsqu'un cercle est défini par son centre et par un point de sa circonférence, il n'est modifiable que par l'intermédiaire de ces deux points et on ne peut donner accès directement aux attributs algébriques, c'est-à-dire aux coefficients de son équation, sans avoir à définir de quelle manière ces deux points en sont affectés, alors que dans le cas d'un cercle défini par son centre et son rayon, la modification dépend de la liberté du rayon.

Dans le cas d'un segment, on peut même se demander ce qu'il est judicieux de considérer comme attribut algébrique : l'équation de la direction, ou les coordonnées des extrémités, et comment informer l'utilisateur sur les possibilités de modification. Dans le cas d'un vecteur ou d'un polygone, apparaît en plus le problème de la direction et de l'ordre des points respectivement.

Magnétophone et curseur textuel

Les procédures de gestion du magnétophone (5) peuvent être simplement complétées par la gestion d'un « curseur » textuel. Ce curseur peut être matérialisé par une progression de couleurs dans le texte du programme de façon à simuler les manipulations de l'interface.

Manipulation directe du texte

Le déplacement d'éléments de programme qui représentent des constructions du programme (6), requiert de mettre en œuvre des artefacts pour les sélectionner « à la main », puis de générer un rendu adapté pour indiquer à l'utilisateur l'impossibilité de descendre la sélection plus bas, ou l'entraînement d'autres sélections. Ce déplacement doit se faire tant que la souris est appuyée. Son traitement doit donc être intégré à la gestion du curseur.

Notations secondaires

Les ajustements ponctuels de présentation du programme doivent être proposés de façon fugitive à l'utilisateur, pour ne pas alourdir le mode opératoire de l'introduction d'un mode. Ainsi, l'utilisateur sera informé, par le retour d'information associé au curseur, des manipulations permises dans l'état courant de l'environnement. Ses apparitions étant liées aux manipulations du curseur, il est naturel que l'événement attendu pour réaliser ce que l'interface propose soit un clic sur la souris. De plus, ce fonctionnement est cohérent avec le fonctionnement de la vue textuelle dans l'interface de Cabri.

Les insertions de passages à la ligne (8) dépendent à peine de la sémantique du texte survolé : peu importe que ce texte soit un identificateur de classe de construction ou une chaîne accompagnant un paramètre, aucun mot ne peut être coupé. Par contre, tous les caractères blancs peuvent être considérés comme des boutons dont l'action provoque un passage à la ligne. Dans l'autre sens, la fin de la ligne peut être considérée comme un bouton dont l'action provoque une remontée de la ligne suivante, lorsqu'il ne s'agit pas d'une nouvelle commande. Il suffit donc que le prototype génère un retour d'information sous le curseur pour informer l'utilisateur de sa réactivité (par la métaphore du bouton de RC par exemple). Ces insertions de sauts de ligne doivent être mémorisées au niveau de chaque commande, de façon à pouvoir « suivre » leur déplacement sans gestion supplémentaire.

Appels de macros

Les deux modes d'affichage des macros (plié et déplié (9)), doivent être accessibles par l'utilisateur depuis le texte qui représente la commande d'appel de chaque macro. Comme il s'agit d'ajustements de présentation, la proposition de changement de mode doit être fugitive, comme pour les passages à la ligne. L'état choisi doit être associé aux objets et non pas aux macros, car une même macro peut être utilisée plusieurs fois dans la même figure.

c. Conclusion

En dehors des modifications des paramètres algébriques, les nouveaux traitements sont indépendants du type de l'objet, de la classe, et même de la méthode, autrement qu'en tant que paramètre du processus de traitement à effectuer : seules les valeurs qu'elles mémorisent sont utiles, la sémantique n'a pas d'influence.

Ces valeurs ne doivent donc pas être mémorisées dans les champs des structures de ces données, ce qui conduirait à dupliquer le même code dans toutes les procédures qui spécialisent les classes, les méthodes ou les types.

C'est pourquoi les procédures relatives au traitement du texte du programme ne doivent pas être passées en paramètres des données. Et les structures de données nécessaires aux manipulations textuelles ne peuvent être intégrées dans les structures de données de Cabri.

Il est donc nécessaire d'introduire des structures de données spécifiques pour les manipulations des éléments du texte du programme.

La nécessité d'insérer une vue textuelle synchronisée et dynamique dans l'éditeur géométrique défini par Cabri conduit à :

- ajouter à l'implémentation des « transposées textuelles » de certaines procédures géométriques,
- mettre en œuvre des artefacts spécifiques d'un environnement de programmation, et
- définir des structures de données minimales capables de mémoriser l'information nécessaire pour regrouper les traitements similaires (si possible par hiérarchisation de façon à garder une certaine homogénéité dans les principes de développement de l'ensemble du code obtenu).

Chapitre 2

Problèmes spécifiques et solutions choisies

L'intégration de fonctionnalités d'environnement de développement dans un logiciel dans lequel la manipulation des objets est ajustée à son domaine d'application, conduit à des problèmes délicats d'implémentation et de choix.

Ces problèmes sont d'abord présentés indépendamment des solutions envisageables. Ensuite, nous mettons en relation les problèmes qui peuvent être résolus avec les solutions mises en œuvre dans d'autres contextes pour des problèmes similaires. Enfin, nous décrivons les problèmes qui ne peuvent être résolus que par la mise en œuvre de solutions spécifiques.

A) Les problèmes

Ces problèmes sont de trois ordres :

- problèmes de conception : les choix de synchronisation et de dynamisme de la vue textuelle du programme contraignent l'architecture du prototype. Ils conduisent à l'obligation d'intégrer des méthodes dans le noyau fonctionnel de Cabri qui doit continuer de gérer l'ensemble des fonctionnalités.
- problèmes théoriques : le noyau fonctionnel du logiciel n'est pas conçu comme un compilateur de langage informatique. La transcription textuelle des manipulations de l'interface conduit à l'utilisation de différentes syntaxes dans un même texte.
- problèmes directement issus des contraintes : les contraintes internes et externes exposées dans le §II-1-C conduisent à des choix de mise en œuvre, pour minimiser les coûts de réalisation et de maintenance, ainsi que les surcoûts en termes de place mémoire et de temps d'exécution.

1°) Piloter l'édition textuelle : problèmes de conception

La forte corrélation entre les vues résultat et programme et le choix de la forme textuelle pour la vue du programme conduisent à des problèmes d'insertion fine des différents traitements à implémenter.

a. Piloter

Les langages de scripts sont conçus pour piloter les applications depuis le système de l'ordinateur. Ces langages permettent de simuler les actions d'un utilisateur et de communiquer avec l'application à travers un programme. Les programmes transmettent des événements aux applications et interrogent le système sur l'effet de ces événements. Ces langages sont très liés au système de l'ordinateur. Ils permettent de programmer des actions conditionnelles et des boucles. Ils sont très utiles pour tester les applications.

Un langage comme AppleScript permet de paramétrer son lexique par un dictionnaire. De plus, sa syntaxe gère quelques artefacts pour faciliter la lisibilité des scripts, comme les articles et les accords. Ainsi, le programme peut utiliser des mots issus de la langue maternelle et, par certains aspects, se rapproche de l'expression naturelle. Pour qu'une application soit scriptable, il faut que ses concepteurs aient défini les événements communiqués, et aient éventuellement paramétré le dictionnaire.

Une version particulière du logiciel Cabri-II est « scriptable ». Cette application, Cabri-scriptable, est utilisée par exemple par des didacticiens pour « espionner » les travaux des élèves, car elle permet de mémoriser toutes les actions effectuées par eux. Comme Sketchpad, Cabri-scriptable permet de mémoriser une séquence d'actions. De plus, l'utilisateur peut ensuite modifier cette séquence comme un texte, à l'aide de n'importe quel éditeur textuel. Il peut introduire des boucles et des tests, puis « rejouer » la séquence mémorisée dans le logiciel.

Cependant, un script ne permet pas d'accéder aux structures de données de l'application, mais seulement d'espionner les commandes, c'est-à-dire le « dialogue » effectué au travers du système de l'ordinateur. Il ne peut donc pas révéler l'état interne du programme lorsque celui-ci effectue des traitements sur ses données. Ainsi, l'application Cabri-scriptable ne peut décrire la structure courante de la figure. Pour cela, il faudrait reprogrammer une application qui, à partir du script, recalculerait les traitements de Cabri sur la structure de la construction lors de l'élimination d'objets ou de la redéfinition de contraintes. Elle ne peut pas non plus donner accès au contenu des macros.

Ainsi, pour le prototype, il ne suffit pas de piloter la vue structurelle depuis les actions de l'utilisateur dans l'interface existante, il faut aussi modifier l'interface pour que l'application révèle son état courant.

Nous ne pouvons donc pas partir du code de Cabri-scriptable.

b. Édition textuelle

La forme choisie pour la vue structurelle est la forme d'un texte. La conception du prototype requiert donc d'intégrer des fonctionnalités de traitement de texte au logiciel. La complexité des traitements augmente avec l'étendue et la qualité de ces fonctionnalités.

Un visualiseur de programme est une application capable seulement d'afficher le programme, et ne permet que des ajustements locaux de présentation. On peut considérer son résultat comme une « impression écran ». Dans un visualiseur de programme, un programme peut être mémorisé "comme une suite de caractères imprimables auxquels se mêlent des caractères de contrôle de présentation physique" : le modèle du document programme est linéaire.

Un éditeur basé sur ce type de modèle ne permet que de l'édition en ligne et des ajustements de présentation. Par exemple, avec un modèle de document linéaire, la mémorisation de références à des éléments de texte n'est pas compatible avec le déplacement d'une suite de caractères d'une position à une autre.

Les éditeurs de documents structurés sont basés sur un modèle plus élaboré [Quint 87] :

"Le modèle représentant le document comme une suite de caractères imprimables auxquels se mêlent des caractères de contrôle de présentation physique est trop limité pour autoriser des traitements puissants. Un modèle d'un plus haut niveau d'abstraction est nécessaire."

Le modèle de document des éditeurs de documents structurés considère le document à travers sa structure logique. La structure logique d'un document est la manière dont le document est composé. Dans le cas d'un texte en langage courant, c'est sur la structure logique que repose le « plan » du document. Dans le cas du texte d'un programme, la structure logique est déduite de la grammaire du langage.

Dans les éditeurs-débogueurs actuels des environnements de développement, le texte semble de plus en plus intelligent : les mots semblent posséder un sens pour la souris. En effet, on peut considérer que le sens d'un mot dans un programme informatique, est une forme de connaissance ou d'assimilation de l'objet qu'il représente (la variable et son instance). L'impression d'intelligence est produite par l'éditeur, lorsqu'il propose à l'utilisateur des comportements qui rendent perceptible cette connaissance. Par exemple, l'éditeur peut proposer d'utiliser cet objet quand l'utilisateur actionne un outil pouvant l'utiliser, informer sur l'état de l'objet ou fournir d'autres informations utiles à l'utilisateur concernant cet objet dans la tâche en cours.

Les fonctionnalités désirées pour l'éditeur textuel intégré dans Cabri sont variées et particulièrement proches des données représentées par le texte du programme. Comme nous l'avons vu au chapitre précédent (§II-1-C-2°-a), elles comprennent :

- une édition pilotée par les manipulations des boutons de l'interface et des représentations graphiques des objets déjà construits dans le programme en cours d'édition,
- des références « réciproques » (entre les déclarations des objets de la colonne des objets construits, et les utilisations de ces objets comme paramètres de construction de la colonne des constructions),
- plusieurs formes de sélection et de visualisation de blocs de texte (contigus ou non, visualisés par une inversion vidéo pour les actions sur la structure du programme, ou par un affichage en couleur pour montrer les relations de dépendance entre les objets),
- différents modes d'affichage (condensé ou expansé pour les appels des macros, éphémère ou permanent pour les attributs graphiques, algébriques et manipulateurs),
- des déplacements ou effacement de blocs de textes non contigus (pour les redéfinitions de contraintes, et pour la réorganisation du programme afin d'ajuster les remplissages),
- des animations (pour simuler l'exécution pas à pas, la destruction d'objet, la définition des macros et la redéfinition de contraintes).

Un logiciel comme Thot (§I-3-A-2°-b) permet de générer un éditeur lié à la syntaxe d'un langage de programmation (§I-3-A-3°-b) comme un éditeur de documents structurés. On pourrait donc imaginer d'utiliser un tel éditeur. Mais l'éditeur résultant ne pourrait pas être intégré directement dans Cabri, puisque le noyau fonctionnel du prototype doit rester le noyau fonctionnel de Cabri (§II-2-A-1°-a). Cette idée ne conduit donc pas à une solution acceptable.

Par contre, les principes de conception des éditeurs de documents structurés peuvent être suivis pour choisir les structures de données adaptées à l'édition et aux traitements désirés pour le prototype. Il est même utile de spécifier l'éditeur désiré en Thot : cela permet d'identifier les comportements attendus de l'éditeur en termes de structure de document. Par exemple, le rapprochement du concept d'ubiquité des objets (dans notre prototype) avec celui des références (dans un document structuré) conduit à chercher, pour gérer l'ubiquité, une structure de données inspirée de celle qui est utilisée par Thot.

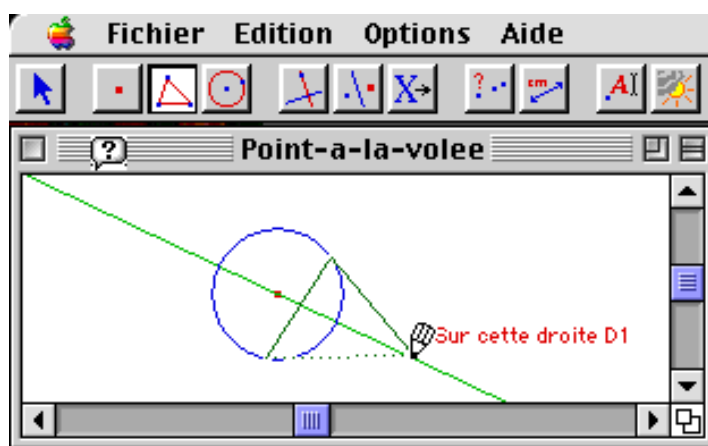
c. Exemple

Nous présentons ici un exemple pour lequel la résolution de la conjugaison des deux problèmes précédents est particulièrement critique. Il s'agit de la prise en compte des spécificités de l'interface avec la gestion de raccourcis manipulateurs.

Le logiciel permet de prendre en compte les désirs de l'utilisateur avant même que celui-ci ne les ait formulés. Il s'agit d'un fonctionnement proche du polymorphisme. Nous l'appelons un raccourci. Un raccourci est une capacité du logiciel à proposer des actions différentes de l'outil sélectionné lorsque ces actions peuvent être utiles ou peuvent avoir un sens pour l'utilisateur.

Définir un raccourci, c'est spécifier les circonstances dans lesquelles certains outils sont capables de proposer leurs "services" à l'utilisateur et à l'outil courant, par proposition de choix sous la forme d'un retour d'information potentiellement actif, c'est-à-dire capable d'insérer une action différente dans (ou avant) l'action en cours.

Un des points délicats conséquence de ce concept, est la gestion du « défaire ». Lorsque l'utilisateur abandonne la tâche qu'il s'était affectée, le logiciel doit revenir en arrière aussi sur ses « propres initiatives ».



Par exemple : l'interface permet des créations de points (point libre, point "sur", point d'intersection) à la volée, en remplaçant temporairement l'outil de construction courant par l'outil « Point » (ou « Point sur » ou « Point à cette intersection »). Si l'utilisateur abandonne la construction en cours, le logiciel doit détruire ses constructions de points à la volée. L'éditeur textuel doit rendre compte de la création de points à la volée, et de leur destruction en cas d'abandon de l'outil de construction courant par l'utilisateur.

Sur cette copie d'écran, on voit que l'outil courant est l'outil « Triangle », alors que le curseur propose de construire un point.

Un autre exemple (non supporté encore par Cabri) serait une proposition de l'interface de rendre tangent un cercle (une conique) à une droite quand l'utilisateur le déforme en passant "au dessus" d'une droite. L'outil courant « Pointeur » serait alors remplacé temporairement par l'outil de redéfinition. Dans ce cas, on peut considérer que l'utilisateur accepte la redéfinition s'il relâche l'animation de la figure dans une position où l'interface propose une redéfinition. Mais l'éditeur textuel ne doit prendre acte de la redéfinition qu'en cas d'acceptation de celle-ci par l'utilisateur.

Ces manipulations constituent des accès à la structure logique des programmes de construction. Le pilotage interne de la vue textuelle et l'utilisation d'une structure de données adéquate permettent de décorréler les problèmes de gestion de l'un et de l'autre. Ainsi la gestion du rendu des raccourcis dans le texte du programme doit résulter directement des traitements effectués par le noyau fonctionnel de Cabri sur la structure logique du programme. Si chaque objet géométrique est représenté par un seul élément textuel, contenant toutes les informations pour permettre son affichage dynamiquement, l'affichage du texte pourra ajuster les positions relatives de ses différents éléments.

2°) Imbriquer plusieurs syntaxes : problèmes théoriques

La récupération d'une grammaire, induite des manipulations de l'interface, pour la description formelle d'un programme de construction, la description de sa "projection" dans la fenêtre graphique, qui autorise l'utilisateur à manipuler directement des instances de chacun des objets qu'il définit, le mécanisme d'identification de ces objets, et la prise en compte des spécificités de certains outils de l'interface comme la calculatrice, ou de la forme de polymorphisme des outils (permettant de prendre comme paramètres des composants d'objets comme un côté d'un triangle), nous conduisent à juxtaposer plusieurs syntaxes différentes.

a. Identifier les objets

L'identification des objets a déjà été abordée au §II-1-A-2°-b. Elle conduit à plusieurs problèmes difficiles.

Trois formes de noms doivent coexister : les noms générés automatiquement par le logiciel, les noms donnés par l'utilisateur, et les noms implicites.

Noms automatiques : indiçage et redéfinition

Une première idée pour cette identification consiste à nommer systématiquement les objets non déjà nommés par l'utilisateur, ces noms étant construits en indiçant les objets à partir de la première lettre du type de l'objet.

L'avantage de cet indiçage est que les indices croissent moins vite qu'avec un indiçage sur tous les objets indépendamment de leur type. De plus, cela permet à l'utilisateur de profiter de la correspondance entre l'ordre de construction des objets et leurs indices pour mémoriser leurs noms.

L'inconvénient d'utiliser une telle procédure de nommage est une contrepartie de son avantage mnémotechnique : après des redéfinitions de contraintes, qui modifient l'ordre de la construction, l'ordre induit par les indices des objets n'est plus en relation avec l'ordre de création de ces objets par l'utilisateur.

De plus, si le nom affecté aux objets pour la vue textuelle n'est pas mémorisé avec le reste des informations de la figure, la reprise d'une figure enregistrée ne conduit pas toujours aux mêmes affectations pour les noms. Par exemple, si l'utilisateur a détruit des objets pendant la construction d'une figure, la nouvelle affectation de noms ne pourra pas indiquer les objets détruits. L'information sémantique constituée par la première lettre du type de l'objet minimise légèrement ce problème.

L'indiçage des noms des objets conduit donc à des sensations de "sauts" dans l'identification des objets par l'utilisateur.

Place des « noms utilisateur »

Pour éviter la sensation de saut, l'utilisateur peut nommer lui-même les objets.

Des quotes pour différencier les provenances des noms

On peut utiliser deux fois le même nom, ce qui peut par exemple servir pour montrer, sur une même figure, différentes situations d'un même problème à des élèves. Dans ce cas, le prototype doit différencier les noms utilisateur identiques, par exemple encore en les indiçant. Par exemple, si un utilisateur a nommé deux droites D, le processus d'indiçage les distinguera en les nommant D1 et D2. Si l'utilisateur avait déjà utilisé le nom D1, cela nous ramène au même problème.

Donc le nom affiché dans la vue textuelle ne peut pas être seulement le nom utilisateur. Le prototype doit signaler la provenance du nom : utilisateur ou automatique.

L'utilisateur ne doit pas assumer les problèmes liés à l'implémentation du prototype. Il ne doit donc pas être contraint par les nécessités dues à l'informatique, en particulier pour le nommage de ses objets. L'idéal serait que tous les caractères soient permis, sans restriction.

Il est cependant plus naturel d'entourer de quotes les noms utilisateur que les noms automatiques, qui peuvent être considérés par l'utilisateur comme des noms machine, dans sa représentation mentale du logiciel.

Éviter les noms déjà utilisés

Si un utilisateur a nommé une droite D1, il n'est pas judicieux que le logiciel génère aussi le nom D1, même si le nom utilisateur est entouré de quotes dans le texte du programme.

Donc il est nécessaire que le prototype évite les noms déjà utilisés.

Cependant, si l'utilisateur choisit de nommer une droite D1, alors que le logiciel a déjà utilisé le nom D1, ce nom automatique doit-il être modifié, ou est-il préférable d'interdire à l'utilisateur l'usage de ce nom ? Nous avons vu qu'il est préférable de ne pas restreindre l'utilisateur, mais nous avons vu aussi que les "sauts" dans le comportement du logiciel, comme le renommage de la droite D1 en par exemple D2 (si D2 n'a pas été utilisé) provoque une rupture de l'engagement direct. Nous arrivons donc à un choix conflictuel entre deux nécessités.

De plus, un utilisateur peut détruire le nom d'un objet. Le processus de nommage doit alors lui affecter un nom automatique.

En fait, la prise en compte de considérations aussi fines de chacune des possibilités conduit au risque d'une perte de cohérence du processus de nommage des objets. On ne peut éviter cette perte de cohérence qu'en mettant en place un processus simple, préservant l'intuition du comportement par l'utilisateur (§I.2.C.3°.a).

La procédure la plus simple est l'affectation d'un nom automatique à tout objet, qu'il soit ou non aussi nommé par l'utilisateur. Le nom utilisé dans le programme est alors ce nom automatique, suivi éventuellement du nom utilisateur entre quotes.

Ambiguïté des noms implicites

Quelques conventions d'usage en géométrie autorisent à nommer des objets géométriques sans les avoir définis dans le contexte. Leur définition provient de la définition d'un concept mathématique.

Prenons l'exemple d'un quadrilatère défini par quatre points A, P, I et V. La convention veut que l'on nomme ce quadrilatère AIVP, si ces points se suivent dans cet ordre lorsque l'on parcourt le périmètre du quadrilatère. On peut alors le nommer aussi indifféremment IVPA, VPAI et PAIV. Le sens de parcours peut être ou ne pas être imposé selon le pays, quand cela a un sens.

Nous disons qu'un tel nom est "implicite".

Un autre exemple est celui de la droite: il n'est pas nécessaire d'avoir donné un nom à la droite passant par A et B pour pouvoir la nommer. Mais son nom (AB) ou (BA) en France est AB ou BA aux USA, alors que AB ou BA en France représente la distance entre les points A et B.

Ces noms implicites sont en fait très conventionnels et dépendant de la culture. Ce sont des noms d'usage, apparus par nécessité de désignation dans les contextes restreints d'exercices. Mais leur usage conduit à des ambiguïtés, qui ne sont pas bloquantes dans les situations réelles où elles peuvent être levées au coup par coup.

Dans le contexte de l'identification informatique, leur usage systématique cumulé avec la gestion de noms automatiques et de noms utilisateur conduit à des ambiguïtés encore plus fréquentes. Par exemple, on obtiendrait :

- ["P1"1"I"], pour le segment d'origine le premier point P1 (nommé par l'utilisateur) et d'extrémité le point I (nommé par l'utilisateur) nommé implicitement par le logiciel,
- "[P1I]"1, pour le premier segment nommé par l'utilisateur [P1I] et dont on peut penser que son origine est P1 et son extrémité I,
- [P1"I"]1, pour le premier segment d'origine P1 (nom automatique) et d'extrémité un point nommé I par l'utilisateur.

Par contre, un avantage à l'utilisation de noms implicites est leur robustesse vis-à-vis des destructions d'objets, des redéfinitions de contraintes et autres modifications de la structure logique du programme.

Trois syntaxes

La coexistence de ces trois syntaxes conduit à des problèmes d'ambiguïté. Le choix de résoudre ces problèmes dans certaines situations (pour certains types d'objets) fait perdre de sa cohérence au processus de nommage.

Dans la première version, nous ne gérons que l'indigage des noms des objets non déjà nommés par l'utilisateur, sans ajustement a posteriori de l'indice donné. Nous ne gérons pas les noms implicites, en dehors des implicites déjà gérés par l'interface, comme la désignation du côté d'un polygone, et cela seulement selon la convention française.

b. Prise en compte de « sous-objets »

Par exemple, lorsque l'outil « Droite perpendiculaire » est activé, l'interface propose de construire les droites perpendiculaires aux côtés des polygones sans que ceux-ci n'aient été matérialisés par des segments ou des droites. Le texte du programme doit permettre d'identifier cette nuance dans le passage de paramètres : le paramètre de l'outil n'est pas seulement le polygone, mais un sous-objet non construit du polygone. Le côté d'un polygone ne peut donc pas être identifié par un nom automatique.

Nous avons vu que les noms implicites étaient générés à partir des constituants d'un objet. Comme un polygone ne peut exister que si ses sommets ont déjà été construits, le texte peut faire référence à ces points.

Nous avons vu aussi que les noms implicites dépendent de la culture. Nous ne considérerons ici que la convention française, l'influence de la culture n'ayant de conséquences que sur la forme du rendu. Il restera à paramétrer cette forme en fonction de la langue de dialogue choisie.

Pour spécifier le côté concerné par cette utilisation particulière du polygone, il faut donc repartir des traitements implémentés dans le code de Cabri pour identifier le côté concerné. Ce côté est identifié par son indice dans l'ordre des côtés portés par deux points de base consécutifs du polygone. Il dépend du nombre de sommets du polygone.

Les traitements à mettre en œuvre pour accéder aux sous-objets manipulables par l'interface sont à déterminer de manière ad hoc dans chacune des situations rencontrées.

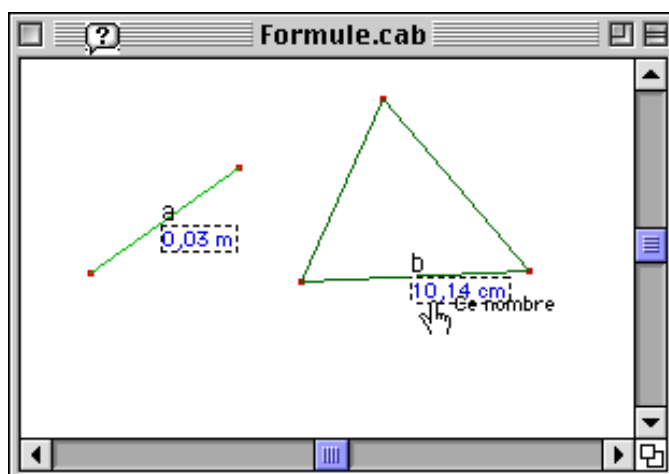
c. Prise en compte de « sur-objets »

Le logiciel permet aussi d'effectuer des calculs sur des mesures associées à des objets, et d'utiliser leurs résultats comme paramètres d'outils de transformation (rotation, report de mesure...). Par exemple, il est possible de construire l'image d'un point par une rotation dont l'angle est le double de la mesure d'un des angles d'un triangle.

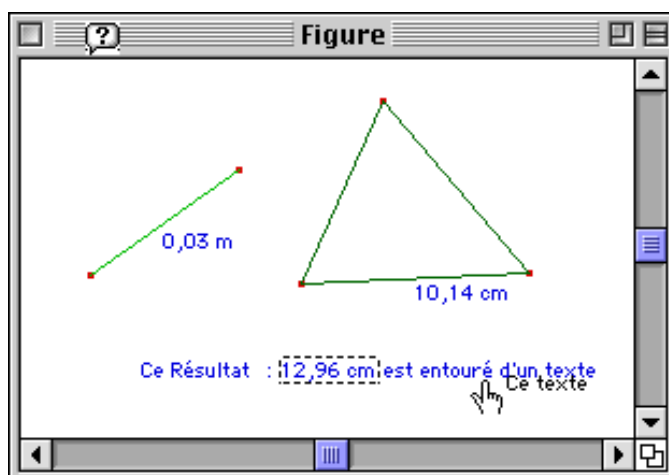
La spécification des formules se fait dans Cabri grâce à l'outil Calculatrice qui donne accès à une boîte de dialogue dédiée à l'édition des formules. Les calculs ne peuvent être basés que sur des nombres ou des mesures. La calculatrice gère la cohérence des unités : par exemple si l'on ne peut additionner des cm et des cm², le logiciel prévient que les unités sont incompatibles entre elles ou avec la fonction.

Les nombres

On peut choisir l'unité pour exprimer les mesures de longueurs en cm, en km, etc. les mesures d'angles en degrés, en radians, etc. et on peut additionner des cm et des m. Les nombres peuvent d'ailleurs être considérés comme des mesures sans unité. Les mesures (donc) sont les seuls objets sur lesquels le curseur réagit lorsque l'outil courant est l'outil Calculatrice.



L'outil Calculatrice utilise une dénomination temporaire des nombres pour les désigner : au fur et à mesure de leur utilisation dans la formule, les nombres sont « numérotés » par les lettres de l'alphabet (au delà de 26, les lettres sont doublées).



Le fonctionnement de cet outil est particulier, mais les nombres ne sont pas des objets géométriques, mais plutôt des abstractions d'objets géométriques, puisqu'ils ne sont obtenus depuis d'autres objets (vraiment) géométrique qu'après application d'une fonction (de mesure). C'est pourquoi, ces nombres abstraits ne peuvent être concrétisés qu'à travers un support qui les rend lisibles. Ce support est constitué par un objet de type texte.

Les formules

Pour entrer une formule, l'utilisateur sélectionne les mesures sur lesquelles il veut effectuer ses calculs et insère les opérateurs. Le logiciel visualise la sélection de ces nombres dans la figure par un rectangle de sélection auquel est accrochée une étiquette portant le nom temporaire de cet objet. Simultanément, la formule est construite dans la calculatrice.

Les formules qui apparaissent dans la calculatrice sont par exemple de la forme :

$a+b$	$a^2+b^2-c^2$
$\cos(a*2)$	$\ln(a/b+c/b)$

Ces formules doivent être modifiées pour la vue textuelle. En effet, les nombres sont déjà identifiés par un nom dans le programme, il faut donc remplacer les noms temporaires utilisés par la calculatrice par les noms effectifs dans le texte.

La formule utilisée par un appel de l'outil Calculatrice peut être considérée comme un paramètre composite. Ses composants sont les objets nombres sur lesquels est basé le calcul, et son texte est constitué de la formule transformée. Par exemple, en reprenant les quatre formules précédentes, on aboutit à :

Calculatrice (N2+N3)	-> nombre N4
Calculatrice (N1^2+N2^2-N3^2)	-> nombre N4
Calculatrice (cos(N3*2))	-> nombre N4
Calculatrice (ln(N2/N1+N3/N1))	-> nombre N4

Dans tous les cas, cet appel à l'outil de construction d'un nombre est suivi de la construction du texte qui doit le porter.

Texte (Inclure N4)	-> texte T2
--------------------	-------------

Les traitements à implémenter pour supporter les spécificités de l'outil calculatrice obligent donc à récupérer le texte de la formule mémorisée avec chaque nombre construit par cet outil, et à remplacer dans ce texte les noms temporaires des paramètres par les noms utilisés dans le texte du programme pour les constituants du nombre. Il faut aussi traiter l'outil Texte avec sa « méthode » d'inclusion de nombres.

Pour « compiler » un texte de programme saisi par un éditeur extérieur à Cabri, il faut intégrer un analyseur spécifique qui vérifie la syntaxe des formules. De même, pour permettre de modifier les formules dans le texte, l'éditeur textuel du prototype doit reconnaître que l'élément modifié est une formule et qu'il doit vérifier une syntaxe spécifique.

Les unités représentent un état d'affichage des mesures. Elles peuvent être considérées comme des attributs associés aux objets nombres. Elles peuvent donc être intégrées aux bulles avec les autres attributs.

3°) S'intégrer dans un code déjà existant : problèmes de contraintes

Nous avons vu (§II-1-C-3°) qu'il fallait éviter de dupliquer du code et donc essayer de reprendre systématiquement les procédures déjà codées, pour les utiliser dans le contexte des besoins du prototype. Mais ces procédures ont été implémentées pour un usage différent, et pour le prototype, on ne peut pas se limiter à les appeler. Les contraintes de maintenance obligent à ne pas modifier leur contenu, donc les seules possibilités consistent à adapter le contexte. Cependant, on ne peut pas non plus modifier l'état interne de la figure pour le décrire. Il faut donc éviter de rajouter des objets ou de modifier leur description mémorisée. Ce problème, avec ses deux contraintes antagonistes, conduit à des choix de compromis. En effet, les solutions possibles sont :

- dupliquer le code en remplaçant les références aux données nécessaires par celles du nouveau contexte,
- ajouter des données temporaires pour remplacer (patcher) ces références,
- transformer ces données en paramètres des procédures.

Illustrons ces trois solutions sur l'exemple de la présentation des attributs graphiques, algébriques et manipulateurs associés à un objet. Cette présentation conduit aux trois besoins suivants :

- récupérer les équations calculées par l'outil de l'interface "Coordonnées et équation", sans créer de nouveaux objets dans la figure,
- récupérer la gestion des palettes graphiques pour les appeler depuis les bulles d'aide,
- adapter le vocabulaire utilisé dans les menus, en respectant le multilinguisme.

a. Les équations

L'outil de Cabri « Coordonnées et équation » permet de construire les objets nécessaires pour porter les coordonnées d'un point ou l'équation d'une droite, d'un cercle ou d'une conique. Les deux coordonnées sont portées chacune par un objet de type nombre et le couple des deux nombres est porté par un objet de type texte. Quant aux équations, elles sont portées par un objet de type texte. Donc, dans le cas d'un point, cet outil génère trois objets : deux nombres et un texte, et dans les cas d'une droite, d'un cercle ou d'une conique, cet outil génère un seul objet.

Si le code relatif aux coordonnées d'un point est simple et relativement court, le code relatif au calcul de l'équation d'une conique est complexe et long. La duplication du code qui gère les coordonnées des points n'est pas plus coûteuse que le code nécessaire à la création de trois objets temporaires créés par le code initial dans une zone mémoire « à part », mais le code du calcul d'une conique ne doit pas être dupliqué. En effet, non seulement cela allonge considérablement le code et donc la place mémoire nécessaire au logiciel, mais les hypothétiques efforts de maintenance de ce code devraient aussi être dupliqués, en gardant bien en tête que chaque modification d'un côté de ce code devrait aussi être répercutée dans l'autre.

La création temporaire d'objets conduit à un autre problème : comme afficher l'état d'un objet dans la vue textuelle répond à un besoin purement consultatif, il ne faut pas modifier l'état courant du logiciel, ni au niveau de l'outil courant avec la méthode en cours, ni au niveau du programme de la figure qui ne doit pas faire état de ces objets temporaires. Après cet affichage, l'utilisateur doit retrouver le logiciel dans l'état d'exécution de son programme précédent : si une construction était en cours, l'outil courant et les objets déjà sélectionnés doivent rester inchangés.

La troisième solution est de transformer le code en modifiant le statut des données critiques des procédures utiles : les variables globales dont les valeurs sont affectées ou utilisées peuvent être déclarées comme des variables de la procédure. Il en découle un allongement de la liste des paramètres, et donc au-delà de la perte de lisibilité du programme, un encombrement de la mémoire vive et un temps d'exécution plus important.

b. Les palettes

Depuis les bulles d'aide, il est plus immédiat pour l'utilisateur d'identifier le schéma suggestif d'une icône qui décrit l'épaisseur du trait ou la couleur, que d'interpréter le texte constitué par l'adjectif correspondant. L'affichage de l'état graphique est donc plus adapté sous une forme icônique que sous une forme textuelle. Remarquons que le choix est différent pour la troisième composante, dans l'impression ou la sauvegarde textuelle du texte du programme.

Depuis l'icône représentée dans la bulle d'aide, un comportement cohérent de l'interface est de proposer le même mode de modification que depuis les autres places où cette icône est affiché : depuis la barre de menu des attributs, ou depuis chacun des items du menu de modification des attributs de la barre des outils de Cabri.

Il est donc nécessaire de récupérer le code déjà mis en œuvre dans Cabri pour gérer les palettes. Plusieurs étapes doivent être réalisées :

- choisir le schéma de fonctionnement.
- récupérer l'affichage de l'icône de l'attribut associé à l'objet et l'ouverture de la palette graphique à partir de cette position et non plus à partir des menus.
- répercuter la modification de l'attribut comme depuis l'outil du menu graphique.

Schéma de fonctionnement.

Pour ouvrir une bulle d'aide, il suffit à l'utilisateur de double-cliquer sur l'occurrence du nom d'un objet. Tant que le bouton de la souris est abaissé, la bulle reste affichée. L'utilisateur peut alors déplacer la souris (bouton abaissé) pour placer le curseur dans la bulle d'aide. S'il positionne son curseur sur une des icônes qui représentent l'attribut graphique associé à l'objet, il peut espérer pouvoir modifier cet attribut. Pour cela, on peut imaginer qu'alors la palette graphique correspondant à l'icône survolée s'ouvre, et que le nouvel attribut est celui qui est survolé lors du relâchement de la souris.

Affichage de l'icône de l'attribut associé à l'objet

Tous les types d'objets ne supportent pas les mêmes attributs. Nous avons vu que les nombres sont affectés d'une unité qui doit faire partie des attributs de l'objet. Plus particulièrement pour les attributs graphiques, la taille du point n'a de sens que pour les points, alors que l'épaisseur du trait et les pointillés n'ont pas de sens pour eux.

Les affichages des icônes d'attributs graphiques dépendent donc du type de l'objet. Il faut récupérer dans le code de Cabri, la procédure qui met en correspondance les types d'objets et les attributs. Ensuite, il faut afficher l'icône correspondant à la valeur de chacun des attributs concernés par l'objet.

Il faut donc trouver la valeur de l'attribut associée à l'objet, et afficher cette valeur dans une icône dessinée à l'ouverture de la bulle d'aide.

Gestion de la palette graphique à partir de cette position

Les palettes graphiques sont considérées comme des menus particuliers. Elles sont gérées par les mêmes procédures que les autres menus.

L'implémentation des menus est programmée en C, avec passage de procédure par les champs des structures des données pour permettre une programmation à objet. En fait, selon que ce menu a la forme d'un simple menu déroulant, ou d'une palette, la procédure appelée n'est pas la même. En conséquence, une procédure est écrite dans le logiciel pour chaque type de menu. Pour les palettes, il s'agit d'une procédure nommée `palette_menu_proc`.

Il faut ajuster l'appel de ces procédures pour tenir compte du décalage des positions et de la spécificité du mode de manipulation des bulles d'aide. En effet, l'utilisateur ne doit pas relâcher la souris tant qu'il veut voir la bulle d'aide. Puisque le choix de l'item ne peut être communiqué par un relâchement de la pression sur la souris, la seule possibilité est de faire intervenir le facteur temps.

Le changement de la valeur d'un attribut doit être dynamique, et ne se fixer par exemple qu'au bout de deux secondes sur le même item. Le changement de valeur est alors répercuté dans l'objet et sur l'icône affichée dans la bulle.

c. Le multilinguisme

Les morceaux de phrase proposés sous le curseur pour aider l'utilisateur ne sont composés que de démonstratives, pour désigner l'objet en évitant toute ambiguïté. Pour cela, le type de l'objet désigné est précisé, ainsi que son nom utilisateur s'il en a un, le tout affiché dans la même couleur que sa représentation graphique.

Dans le contexte du programme, le problème du multilinguisme est plus important. En effet, les éventuels défauts sont moins bien compensés par un recoupement des informations de formes d'expression diverses (textuelle et graphique), aidant ainsi l'utilisateur à distinguer, puisqu'il faut trier dans un texte plus long et plein d'informations, ce qui était différencié par le changement de forme. D'où l'importance de la présentation et des notations secondaires.

L'état manipulateur décrit les possibilités de manipulation de l'objet. Nous avons vu (§II-A-3°-a) que les objets peuvent être :

- montrés ou cachés, selon le choix de l'utilisateur, par l'usage de l'outil "cacher/montrer" ;
- accessibles ou non, selon qu'ils sont internes à des macros ou résultat de constructions conditionnelles ;
- à l'infini ou pas à l'infini, comme le point d'intersection de deux droites parallèles ;
- punaisés ou dépunaisés, selon le choix de l'utilisateur par l'usage de l'outil "punaiser/dépunaiser", afin de fixer des positions de points libres dans des animations.

Dans les bulles, ces états doivent être décrits par du texte.

Pour les états spécifiés par l'action d'un outil ("Cacher/montrer" et "Punaiser/dépunaiser"), des verbes correspondant aux états sont déjà utilisées dans le logiciel pour nommer ces outils. Ils sont donc déjà mémorisés dans les ressources du logiciel, et ceci dans toutes les langues de dialogue supportées. Il faut cependant ajuster la forme de ces termes, afin qu'ils s'accordent au genre du type de l'objet, comme des adjectifs.

Par contre pour les autres états (accessible ou non accessible, à l'infini ou pas à l'infini), il faut ajouter les termes requis. De plus, il faudra gérer l'accord, selon les nécessités du langage, comme dans le cas précédent. Dans le cas du français, il n'y a rien de particulier à faire puisqu'accessible est aussi bien féminin que masculin, et que "à l'infini" ne s'accorde pas.

B) Ce qui est récupérable

Pour supporter les fonctionnalités qui permettent une édition des programmes guidée par la syntaxe, qui rendent réactive la vue textuelle, et qui de plus aident l'utilisateur à mettre au point ses figures, nous avons vu au §II-2-A-1°-b que le modèle de document adapté est un modèle de document structuré. Nous avons vu aussi que nous ne pouvons intégrer directement un éditeur de document structuré dans le prototype puisque le noyau fonctionnel de Cabri supporte déjà la saisie du programme par manipulation directe des objets résultant de son exécution.

Si on ne peut intégrer directement dans Cabri un éditeur de documents structurés, ni récupérer le code d'un tel éditeur, par contre les principes doivent être utilisés.

Et pour cela, l'utilisation d'un logiciel comme Thot, qui est capable de générer un éditeur de Cabri-programme, permet de valider la correspondance entre les éléments des structures logiques de documents et les éléments du langage. La description de la structure logique des programmes est donnée en Annexe B.

On peut remarquer au passage qu'une approche duale à celle qui fait l'objet de cette vue textuelle, consisterait à insérer des fonctions d'édition d'objets géométriques dans le noyau fonctionnel généré par Thot.

Nous proposons de s'inspirer du modèle des documents structurés pour transposer la structure logique des Cabri-programmes en celles des documents, pour assimiler les relations entre les objets aux références et pour rapprocher les insertions de notations secondaires des attributs.

1°) Description de la structure d'un Cabri-programme par un modèle de documents structurés

Le passage de la simple visualisation à l'édition rend nécessaire de mettre en relation les structures des programmes de construction avec les structures des textes de programme. Dans le cas d'un langage de programmation, la structure du document est la structure d'un programme. Elle est portée par la grammaire du langage.

Nous avons vu que les éditeurs de documents structurés étaient basés sur une description hiérarchique de leur composition (§I-3-A-2°-b). Dans le cas du langage de Cabri-programmation, tout programme est constitué d'une description de la figure et des macros qu'elle utilise. Une macro est composée d'une en-tête pour spécifier ses paramètres, et d'un corps qui contient la liste des constructions qu'elle doit réaliser. Une figure est décrite par une liste de constructions, chacune respectant la syntaxe du langage de géométrie formelle. Ces constructions sont précisées par les attributs graphiques et manipulateurs des objets qu'elles construisent. Ces attributs graphiques sont une liste d'états.

De même, le corps d'une macro est décrit par la liste des constructions que la macro contient, mais en général, seuls les attributs qui décrivent l'état manipulateur sont utiles. Les attributs qui décrivent l'aspect de l'objet n'ont de sens que pour les objets externes de la macro : l'utilisateur peut changer l'aspect d'un objet initial ou choisir un aspect différent des valeurs par défaut pour un objet final. Par contre, les valeurs algébriques ne peuvent être fixées (à moins de considérer des variables globales).

Thot est un générateur d'éditeurs de documents structurés basé sur une notion de métastructure des documents. Cette métastructure spécifie la manière dont les différents éléments des documents peuvent être assemblés dans des structures capables de décrire des documents complets à saisir par l'éditeur désiré. Au plus bas niveau, les éléments sont appelés des éléments de base. Aux niveaux supérieurs, ce sont des éléments construits à l'aide d'un constructeur qui assemble des éléments d'un niveau inférieur.

Pour nous, les éléments de base sont les chaînes de caractères qui constituent le texte du programme, avec les noms des outils et les sucres syntaxiques, mais aussi les noms donnés aux objets ou aux macros par les utilisateurs, et quelques mots clés pour préciser le sens des paramètres des macros. Les autres éléments de base sont les identificateurs d'objets.

Les éléments construits sont par exemple la description d'une construction ou d'un objet construit, la description d'une macro ou encore un paramètre entouré du texte qui accompagne les sélections dans la fenêtre géométrique.

Les constructeurs gérés par Thot sont la liste, l'agrégat, le choix, l'unité et la référence. Tous les éléments d'une liste sont du même type, alors que les différents éléments d'un agrégat peuvent être de différents types, comme les différents champs d'une « structure » du langage Pascal ou du langage C. Le contenu d'une figure est une liste de constructions, mais une figure est un agrégat constitué d'une en-tête, d'une liste de constructions et d'une fin.

Le choix permet de prévoir plusieurs possibilités. Une construction est soit une construction simple, c'est-à-dire résultant de l'utilisation d'un outil de base, soit une macro-construction.

Quant à l'unité, elle permet d'intégrer des structures définies différemment (non astreintes à la même logique de document). L'unité est utile pour supporter les formules de l'outil Calculatrice.

La référence permet de supporter les renvois à d'autres parties du document, comme le renvoi à une section, à un chapitre, à une citation bibliographique, ou à une figure. Dans Thot, la référence est bidirectionnelle : par exemple, un renvoi fait référence à un chapitre et ce chapitre est référencé par le renvoi.

Dans notre contexte, les paramètres des commandes, c'est-à-dire les appels aux objets utilisés par les commandes, sont des références aux déclarations des objets, un peu plus que bidirectionnelles. Dans un sens, l'occurrence de l'identificateur d'un objet entouré de sucre syntaxique dans la liste des constructions, renvoie à l'identificateur de cet objet dans la liste des objets construits. Dans l'autre sens, l'identificateur de l'objet dans la liste des objets construits renvoie à toutes les utilisations de cet objet. Et plus loin encore, si l'objet résulte de l'utilisation d'une macro, son identificateur doit renvoyer aussi à la variable de la macro qu'il instancie et à ses utilisations formelles dans le corps de la macro.

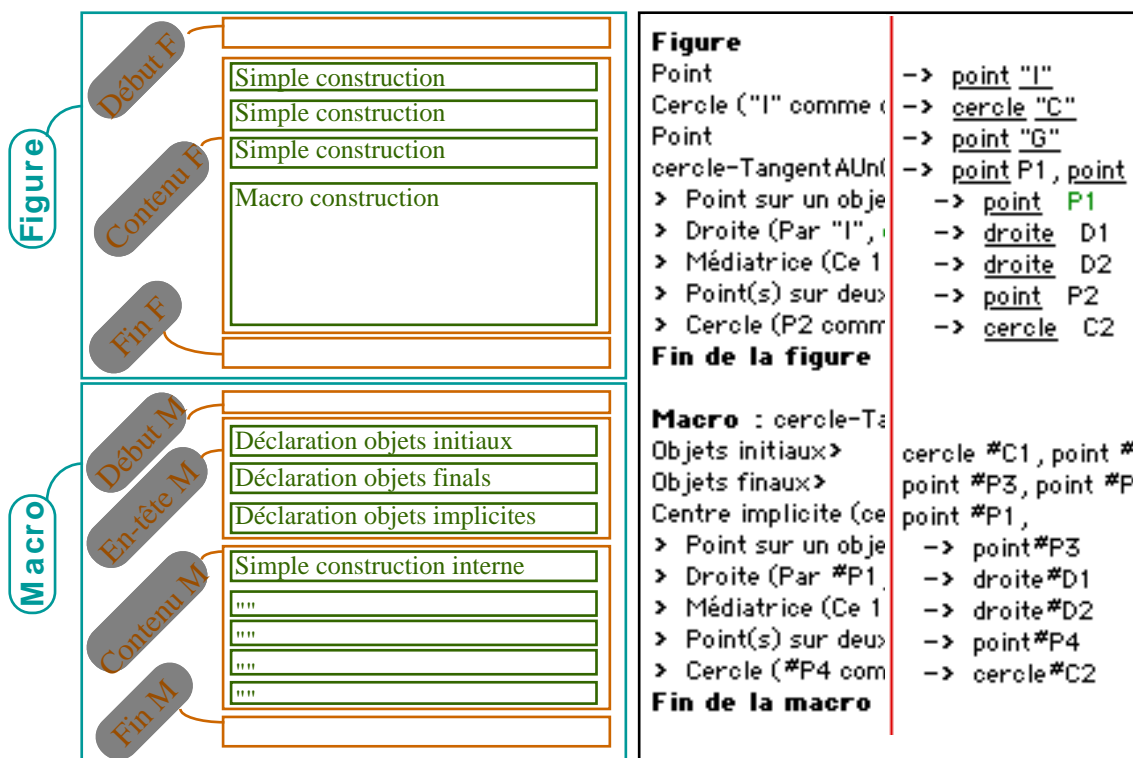
De même que tout l'aspect logique des documents n'est pas entièrement décrit par la structure, les ajustements de présentation pour l'insertion de sauts de lignes et le repliement des macros doivent pouvoir être associés à certains éléments (ou à chaque élément). Pour cela, Thot utilise la notion d'attribut. Il s'agit d'une information attachée à un élément structurel qui s'ajoute au type de l'élément et en précise la fonction dans le document.

Ces attributs peuvent aussi servir à marquer des informations temporaires. Ils permettent de modifier les couleurs de certains éléments, ou de les afficher en inversion vidéo. C'est la notion d'attribut qui, associée au constructeur de références, est sous-jacent aux artefacts qui signalent les objets sélectionnés et les dépendances entre les objets.

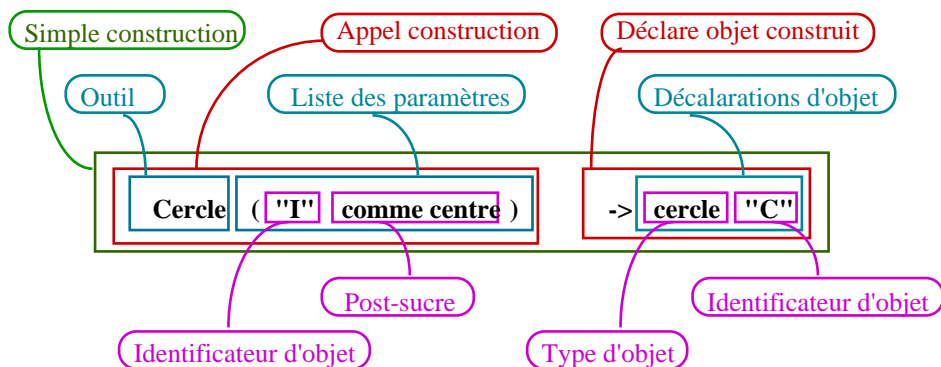
Des attributs attachés à aux simples constructions, aux macro-constructions et aux simples constructions internes (constructions formelles) contenues dans le corps des macros, informent sur l'état d'avancement dans la reconstruction d'une figure ou dans l'exécution d'une macro (curseur textuel), et permettront de simuler les processus utilisés par le logiciel lors des destructions d'objets, des redéfinitions de contraintes et les extractions de constructions utiles pour les macros. De même les attributs peuvent porter des styles pour prévenir de l'état de manipulation des objets. Par exemple, la couleur gris clair pourrait signifier qu'un objet est inutilisable pour l'outil courant.

Exemple

Reprenons l'exemple de la figure d'appel de la macro qui trace un cercle tangent à un autre cercle. Nous arrêtons la décomposition au niveau des constructions dans le premier schéma. Pour décrire ce texte jusqu'au niveau des simples constructions, il faudrait encore décomposer l'élément Macro-construction, ce que nous n'avons pas fait pour ne pas surcharger le schéma.



Le deuxième schéma poursuit la décomposition de la deuxième simple construction de la figure précédente, c'est-à-dire de la construction du cercle initial. Ce schéma est un peu simplifié pour ne pas imbriquer trop de niveaux de décomposition : sous cete forme, il ne peut décrire les listes de paramètres séparés par des virgules, ni les listes d'objets construits par une seule construction.



Ces illustrations ne schématisent ni les constructeurs de référence, ni les attributs.

2°) Comment le mettre en œuvre

La mise en œuvre des éditeurs de documents structurés est basée sur un concept de boîtes gigognes. Chaque élément de la structure est représenté dans une boîte qui correspond à son encombrement (sa silhouette). La position et l'encombrement d'une boîte dépend en partie du constructeur associé à l'élément qu'elle porte, ainsi que des positions et des encombrements de chacune des boîtes qui portent un des éléments que ce constructeur organise.

Le constructeur définit donc une fonction pour calculer la position et l'encombrement de la boîte associée au même élément, à partir des positions et des encombrements de toutes les boîtes des éléments qui le composent.

Par exemple, pour un élément organisé en une liste d'éléments, le coin haut gauche de la boîte est le coin haut gauche de la première boîte, et l'encombrement est la « somme » des encombrements (où la fonction « somme » peut tenir compte de silhouettes non rectangulaires, et peut être spatiale plutôt que seulement verticale).

Cette conception impose de mémoriser toutes les informations nécessaires aux calculs de boîte pour tous les éléments utilisés pour décrire un document. Dans le cas du Cabri-programme de l'exemple précédent (simplifié), cela conduit à mémoriser 26 éléments pour décrire sa structure jusqu'aux simples constructions, et environ à 120 (=94+26) éléments pour décrire complètement sa structure, soit environ neuf éléments en moyenne par simple construction. La complexité est en $n \log(n)$ où n est le nombre d'éléments de base, sans tenir compte du surcoût lié à la gestion des références et des attributs.

Une alternative à ce coût mémoire consiste à simuler l'utilisation des boîtes par des calculs d'encombrement de chaînes de caractères.

En fait, ces deux méthodes opposées peuvent être implémentées : soit les données portent toutes les informations utiles aux calculs des boîtes et les procédures n'ont qu'à consulter ces données et effectuer le minimum de calcul de répercussion globale des valeurs relatives, soit les données ne contiennent qu'un minimum d'information (leur texte) et les procédures doivent tout faire.

Un compromis consiste à arrêter la description hiérarchique physique (les boîtes imbriquées) à un niveau de la décomposition d'un document qui n'est pas le niveau le plus bas, et à simuler l'utilisation des boîtes au-delà. Le choix associé à ce compromis est basé sur la recherche du niveau le plus performant : tous les intermédiaires sont possibles.

Mais, étant donné que le coût de l'implémentation des structures de données pour supporter les boîtes est complet dès qu'on l'utilise, nous n'avons pas implémenté ce mode de représentation dans la première version du prototype qui est présentée dans ce manuscrit.

Ainsi, nous avons choisi une structure de donnée linéaire pour stocker les informations nécessaires à l'édition du texte. Par contre, nous avons accédé aux données en imaginant remplacer plus tard les accès linéaires par des accès hiérarchiques ou mixtes.

Le texte du programme est donc actuellement mémorisé dans deux éléments de texte : l'un pour la liste des constructions, l'autre pour la liste des objets construits.

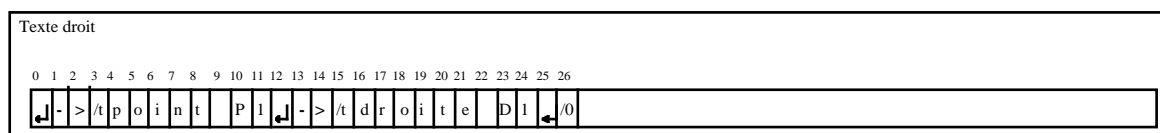
C) Ce qui n'est pas récupérable

Dans cette section, nous décrivons les méthodes utilisées pour résoudre les problèmes obligeant à des résolutions particulières, ainsi que les solutions mises en œuvre.

1°) Comment est gérée l'ubiquité des objets

Dans nos textes linéaires, l'accès aux données se fait comme dans un tableau, par l'indice du premier et du dernier caractère.

Texte gauche																																																							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50					
F	i	g	u	r	e	.	>			P	o	i	n	t	.	>			D	r	o	i	t	e		(P	a	r		P	l)	.		F	i	n		d	e		l	a		F	i	g	u	r	e	.	/	0	



Pour chaque objet, l'information nécessaire à l'accès à chacune de ses occurrences est mémorisée : d'abord l'occurrence de définition, ensuite, la liste des occurrences correspondant à chacun des appels.

La liste de toutes ces informations est parcourue pour repérer le moment où le curseur survole une occurrence de l'identificateur. La procédure qui effectue cette tâche est ajoutée à côté de celle qui, dans la fenêtre graphique, vérifie si la distance entre le point défini par le curseur et la représentation graphique de l'objet est inférieure à un seuil.

Pour la mise en place du rendu, quatre cas sont à considérer, selon la fenêtre active (géométrique ou textuelle) et la fenêtre où le produire (idem).

Par exemple, dans le cas où la fenêtre active est la fenêtre textuelle, le rendu à générer dans la fenêtre géométrique est le même que dans la version de base, sauf que la position du curseur est relative à la fenêtre textuelle, donc sa transposition dans la fenêtre géométrique doit être « inventée » pour que le texte apparaisse près de la représentation graphique de l'objet. De plus, le texte généré pour ce rendu « géométrique » doit être adapté pour calculer le rendu « textuel » à produire. En effet, ces groupes déictiques (à fonction de désignation) ne nomment pas les objets dans la vue géométrique si ceux-ci n'ont pas de nom. Nous devons intégrer les identificateurs d'objets utilisés dans la vue textuelle pour que l'utilisateur puisse mettre en relation les objets avec leurs identificateurs sans ambiguïté. Cette intégration devrait tenir compte des contraintes de la langue de dialogue, mais nous ne l'avons pas encore fait. Par exemple, en français on dit « ce point P1 », alors qu'en anglais on ne dit pas « This point P1 ».

Lorsque l'outil courant est un outil de construction, l'éditeur textuel doit proposer de prendre en compte les objets survolés dont le type convient, avec le même texte que la vue géométrique.

2°) Gestion des notations secondaires

Dans ce texte réactif mémorisé par une structure linéaire, la gestion des insertions de blancs et "retours-chariot" nécessite des traitements particuliers.

Nous avons choisi de permettre le remplacement de tout caractère blanc par un retour-chariot par un simple clic sur la souris lorsque le curseur survole un caractère blanc ou un retour chariot. L'intérêt de ce choix est que la reconnaissance du caractère survolé par le curseur est immédiate.

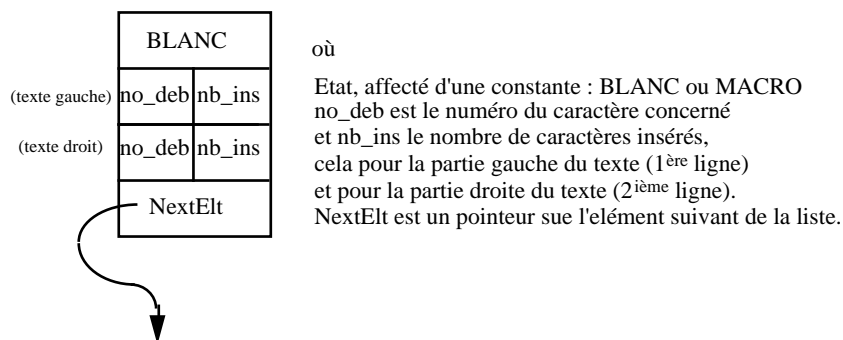
Les défauts sont qu'il faut deux caractères pour mémoriser les retours-chariot, et qu'en conséquence toutes les références aux occurrences ultérieures des identificateurs de noms se retrouvent décalées.

De plus, un problème apparaît à cause de la séparation du texte sur deux colonnes et de la nécessité de garder un alignement horizontal. Un passage à la ligne dans la partie gauche oblige à insérer un retour-chariot dans la partie droite, cette insertion devant être effectuée au début de la déclaration des objets construits correspondant au texte de la construction affecté. Depuis l'autre partie, un passage à la ligne dans la partie droite oblige à insérer le retour-chariot correspondant dans la partie gauche, à la fin du texte décrivant la construction concernée. Inversement, le retour d'un retour-chariot en un blanc pose aussi des problèmes spécifiques, liés aux multiples possibilités d'échanges dont ce blanc peut être issu.

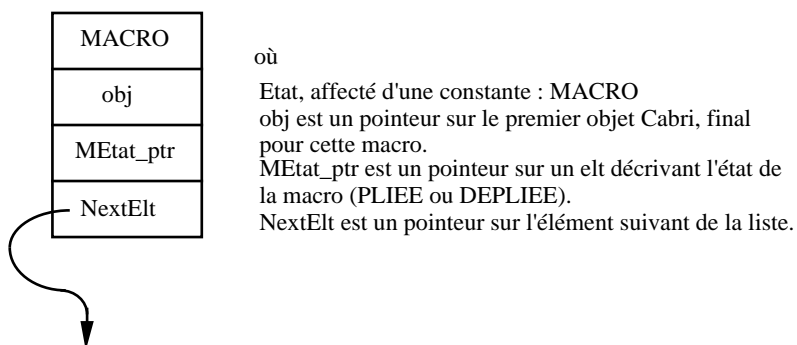
De plus, les notations secondaires doivent pouvoir être cumulées : des retours-chariot doivent pouvoir être insérés dans le corps des macros dépliées, et leur gestion doit résister aux deux modes de présentation.

Nous ne décrivons ici que brièvement les structures de données utilisées. La mise au point de ces traitements a été très délicate.

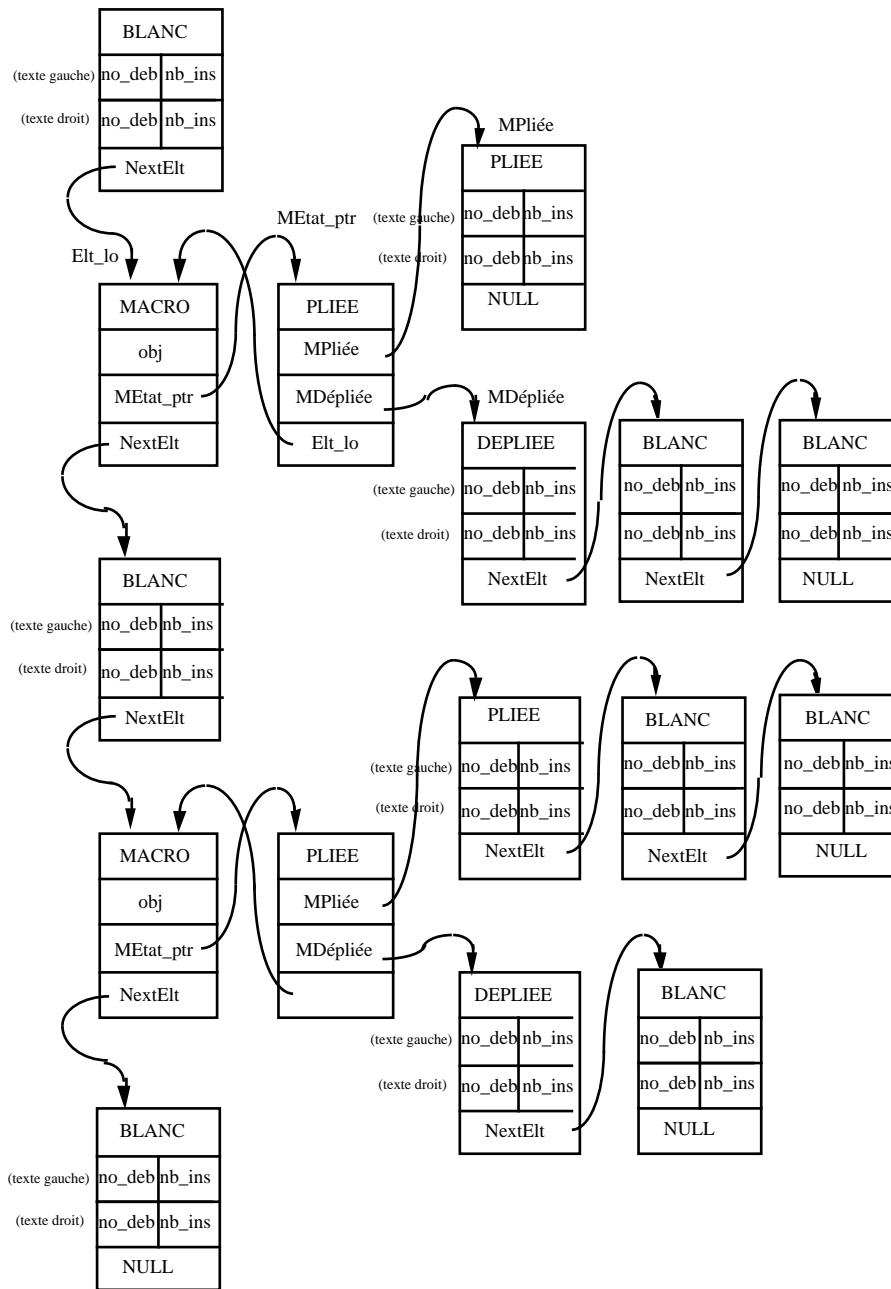
Nous avons choisi de mémoriser les indices et le type des remplacements effectués dans une structure adaptée à accès rapide. L'indice du premier caractère est 0. Pour chaque échange d'un blanc avec un RC, ou pour toute insertion de RC, on ajoute un élément dans une liste. Cette liste est ordonnée par indices croissants. Chaque élément est représenté par la structure de la figure suivante.



Les macros peuvent être appelées sous les deux formes pliée et dépliée, et les blancs insérés sont mémorisés pour chaque forme. De plus, les tailles en nombre de caractères pour les deux formes sont distinctes, et il faut rattraper les décalages sur les calculs conséquents des indices. Pour éviter le problème d'avoir deux indices égaux pour 2 objets différents, la forme sur laquelle sont calculés les indices est la forme la plus encombrante. Nous avons introduit un nouveau type d'élément dans la liste, supporté par la même structure, mais avec un "cast" :



Une liste est donc constituée de l'alternance de ces deux types d'éléments :



Cette implémentation serait allégée par une gestion plus générique des attributs de présentation et le passage à une édition semi-structurée, par exemple si la décomposition structurée s'arrêtait aux simples constructions (un niveau en dessous du niveau hiérarchique de la première illustration de l'exemple, mais dans sa version non simplifiée : 1 par objet construit et 1 par encapsulation possible).

3°) Traitements particuliers pour les sous-objets

Les termes "sous-constituants" ou sous-objets virtuels » sont utilisés ici pour parler d'objets non construits en tant que tels, mais auxquels on permet d'accéder comme à des objets « réels ». Il s'agit par exemple des côtés d'un polygone, traités comme des directions. Le terme n'est pas parfaitement adapté, parce qu'on a l'impression qu'après avoir accédé aux constituants effectifs que sont les points, sommets du polygone, on considère des pseudos sur-constituants que seraient les segments du polygone si on allait jusqu'au bout de leur construction en tant qu'objets (ou plutôt juste le segment nécessaire). Une méthode possible pour supporter ces sous-constituants, serait la création effective du sous-objet désigné dans un objet, mais à la volée.

Nous avons vu que l'utilisation de sous-objets (ou objets virtuels) ou l'utilisation de sur-objets (comme l'accès aux formules résultant de l'utilisation de la calculatrice) conduisait à imbriquer plusieurs syntaxes.

Dans cette section, nous nous limitons au cas des sous-objets de type « côté de polygone ». Pour traiter un côté d'un polygone, il faut :

- retrouver les informations dans la mémorisation des objets, c'est-à-dire le nombre de constituants variables et l'accès direct à des "sous-constituants",
- générer le rendu adapté à la forme textuelle.

Tous les objets de type « Polygone » n'ont pas le même nombre de sommets : le nombre de sommets d'un polygone est une variable de son type. En fait, ce nombre de sommets est le nombre de constituants du polygone. Une fois le polygone construit, son nombre de constituants est fixé : l'utilisateur ne peut lui rajouter un nouveau sommet.

En conséquence, chacun des sommets de ce polygone est formellement accessible, et nous pouvons l'utiliser pour constituer le texte du programme qui base une construction sur un de ses côtés. Pour cela, nous combinons le sucre syntaxique constitué du retour d'information fourni sous le curseur, avec le nom implicite du segment. Il nous suffit de récupérer le code qui permet d'accéder au nombre de constituants du polygone, et celui qui spécifie le côté concerné par la construction de la droite perpendiculaire.

Par contre, lorsqu'un polygone est un objet initial de macro, tout polygone peut lui être affecté, indépendamment de son nombre de sommets. Le problème est donc plus compliqué.

Lorsque le curseur survole la représentation graphique d'un objet, le logiciel « reconnaît » que cet objet a été survolé. Il filtre ensuite cet objet pour vérifier si son type est compatible avec une méthode de construction possible de l'outil courant. C'est pendant ce traitement que les sous-objets sont considérés. Pour que le filtre « accepte » un sous objet, il faut que l'outil courant contienne une méthode applicable à un type d'objet qui peut être porté par un sous-ensemble des constituants de l'objet, comme une direction : une direction peut être portée par tous les côtés d'un triangle, par un vecteur ou par un segment. La mémorisation de ce paramétrage dans la construction d'un objet contient une désignation du sous-objet concerné effectivement par l'appel. Cette désignation est mémorisée dans l'objet construit, par le même support que celui qui est utilisé pour les formules des calculs.

Pour décrire la construction du « rendu », nous prenons l'exemple de la construction de la droite parallèle au cinquième côté d'un polygone. Le texte à générer est par exemple la ligne suivante :

> Droite parallèle (Par P6, Parallèle à ce côté du polygone P5 : [P2, P3]) -> Droite D1

Le sucre syntaxique préparé par la procédure de filtrage était : "Par ce point", puis "Parallèle à cette direction". La deuxième expression doit être ajustée lors du remplacement du nom du type de l'objet qui y apparaît ("cette direction") par l'identificateur de l'objet dans le texte du programme (qui ici n'est pas seulement "P5", mais "ce côté du polygone P5 : [P2, P3]").

La stratégie qui consiste à mémoriser dans l'objet construit la désignation du sous-objet concerné, permet de n'utiliser qu'une seule méthode de construction pour supporter un nombre variable de constituants du même type. Elle pourrait être étendue à la gestion des fractales. On peut considérer que le type « direction » est utilisé comme un « cache ». Lorsqu'une méthode de construction utilise un objet de type cache, l'accès aux paramètres n'est pas direct, mais nécessite un calcul. Nous aborderons ce point dans le §II-4-B.

Présentation du prototype et évaluation

Ce chapitre est consacré à la présentation du prototype. Nous commençons par montrer comment il fonctionne avec une sorte de manuel d'utilisation de la vue textuelle, puis nous présentons comment nous avons fait, avec les structures de données utilisées et les différentes méthodes utilisées pour répondre à chacune des attentes (§II-1-C-2°-a).

Nous évaluons ensuite le résultat. Pour cela, nous examinons d'abord le langage de géométrie formelle, puis la méthode utilisée pour assurer la simultanéité des réactions dans les vues géométrique et textuelle. Enfin, nous présentons la stratégie choisie pour prendre en charge les effets d'animation et évaluons la qualité dynamique de la vue textuelle.

A) Présentation du prototype

Nous avons vu au chapitre II-1 les motivations générales de la définition du langage de Cabri-programmation. Nous avons aussi vu quelques spécificités à prendre en compte, comme des contreparties textuelles de la qualité de manipulation directe supportée par le logiciel. Dans cette section, nous synthétisons ces différentes exigences par une présentation du résultat.

1°) Langage de géométrie formelle

Le langage de Cabri-programmation est un langage de triplets qui décrit chaque objet construit dans la figure courante, dans un ordre issu des dépendances entre les objets, et est induit des manipulations de l'interface. Chaque triplet représente la commande qui spécifie les relations de l'objet qu'il décrit avec les autres objets déjà construits (contraintes), son type et son identificateur textuel (déclaration), et ses attributs de représentation graphique (affichage).

Dans le prototype, la vue textuelle présente de façon permanente seulement deux composantes, la troisième apparaissant dans les « bulles ». Nous donnons ici la syntaxe externe des deux premières composantes seulement.

La syntaxe des troisièmes composantes a été donnée précédemment. Chacune n'est constituée que d'une liste d'états, et des valeurs algébriques pour les objets libres. Ces troisièmes composantes ne servent que pour entrer des figures complètes à partir d'un fichier texte ou pour stocker de façon permanente la vue textuelle, en complétant sa définition formelle par les valeurs algébriques des objets libres, les attributs graphiques des objets pour ajuster la présentation et les attributs d'animation.

Les deux premières composantes caractérisent les possibilités d'animation de la figure, par un langage de géométrie formelle.

Schématiquement, chaque construction d'un objet à l'aide d'un outil provoque l'écriture d'une ligne d'instruction dans la fenêtre textuelle. Ces instructions suivent la syntaxe suivante¹ :

'>' nom-de-l-outil '(' {oi ';' } oi ')'	'->' {of ';' } of
---	-------------------

où oi et of spécifient respectivement les objets initiaux et les objets finals.

Nous décrivons maintenant plus précisément le langage de géométrie formelle par son lexique et sa syntaxe.

a. Lexique

Le lexique est choisi en français, mais il est remplacé par sa version anglaise, allemande ou espagnole, selon la langue de dialogue choisie par l'utilisateur. La liste suivante énumère les noms des pré-terminaux utilisés pour les noms d'outils. Elle ne contient ni les outils d'animation ni les outils d'aspect.

Point	Droite perpendiculaire	Aligné ?
Point sur un objet	Droite parallèle	Parallèle ?
Point(s) sur deux objets	Milieu	Perpendiculaire ?
	Médiatrice	Equidistant ?
Droite	Bissectrice	Appartient ?
Demi-droite	Somme de deux vecteurs	
Vecteur	Compas	Distance & longueur
Triangle	Report de mesure	Aire
Polygone	Lieu	Pente
Polygone régulier	Redéfinir un point	Mesure d'angle
		Coord. & Equation
Cercle	Symétrie axiale	Calculatrice
Arc	Symétrie centrale	Table
Conique	Translation	Texte
	Rotation	
Marquer un angle	Homothétie	
	Inversion	

Remarque : les outils "marquer un angle" et "texte" ne sont pas des outils d'aspect, puisqu'il définissent des objets de type Marque (d'angle).

Les types des objets de Cabri apparaissent dans deux chaînes de caractères : le nom du type, et le désignateur typé utilisé pour compléter la chaîne du retour d'information du curseur dans lequel l'article est remplacé par un démonstratif.

nom du type	désignateur typé
le point	ce point
la droite	cette droite
la demi-droite	cette demi-droite
le vecteur	ce vecteur
le triangle	ce triangle
le polygone	ce polygone

¹ Nous utiliserons les notations suivantes :

$\langle \alpha \rangle = \varepsilon + \alpha$	$\{ \alpha \} = \alpha^*$	$[\alpha] = \alpha^+$
---	---------------------------	-------------------------

Les chaînes de caractères entre simples quotes et écrites en gras sont des terminaux.

le polygone régulier	ce polygone régulier
le cercle	ce cercle
l'arc	cet arc
la conique	cette conique
le texte	ce texte
le nombre	ce nombre
l'angle	cet angle
la marque	cette marque
l'unité	–
l'équation	cette équation
le repère	ces axes

Cette liste est à compléter par les noms des sous-types définis par l'utilisateur pour désigner le premier objet final d'une macro.

Les chaînes de caractères qui servent à produire le retour d'information affiché sous le curseur et qui sont utilisées comme sucre syntaxique dans le langage de géométrie formelle sont données dans la table suivante :

pré-sucres		post-sucres
sur ...	rotation de comme centre
par ...	selon de l'arc
et par ...	autour de est-elle parallèle ?
passant par ...	homothétique de est-elle perpendiculaire ?
d'origine ...	dans
perpendiculaire à ...	à ...	
parallèle à ...	circonférence de ...	
milieu de ...	équation de ...	
médiatrice de ...	éditer ...	
symétrique de ...	et ...	
par rapport à	
translater ...		
de ...		

Un pré-sucre supplémentaire est ajouté pour l'éditeur textuel : il s'agit d'une chaîne utilisée pour préciser la désignation d'un nouveau point d'intersection, c'est-à-dire d'un point d'intersection de deux objets, différent d'un point déjà construit. En français, cette chaîne est : « différent de... »

b. Syntaxe

La syntaxe du langage de géométrie formelle est décrite par la liste des règles suivantes¹ :

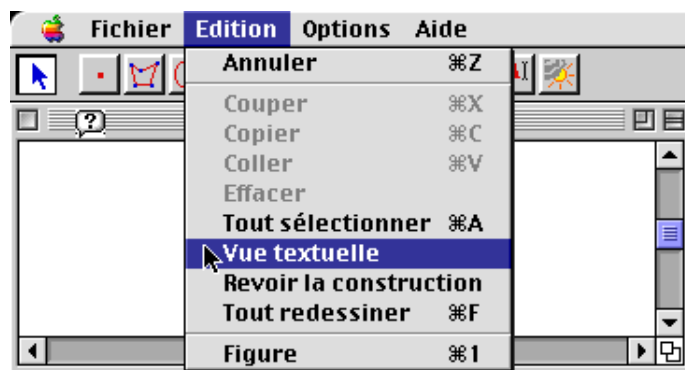
Programme_Cabri =	Figure &RETOUR { Macro &RETOUR }	
Figure =	' Début de la figure'	&RETOUR
	[Construction_simple Construction_macro]	
	' Fin de la figure'	&RETOUR
Construction_simple =	Outil { '(' Liste_paramètres ')' }	'->' Déclarations_objets &RETOUR
Construction_macro =	NOM_MACRO '(' Liste_paramètres ')' '->' Déclarations_objets	&RETOUR
	&TAB [Construction_simple &RETOUR]	
Liste_paramètres =	Paramètre_ensucré { &VIRGULE Paramètre_ensucré }	
Paramètre_ensucré =	Pré_sucré Ident_Objet Post_sucré	
Déclarations_objets =	Déclaration_objet { &VIRGULE Déclaration_objet }	
Déclaration_objet =	Type_objets Ident_Objet	
Macro =	' Début de la macro' NOM_MACRO	&RETOUR
	Déclarations_macro [Construction_Msimple]	
	' Fin de la macro' NOM_MACRO	&RETOUR
Déclarations_macro =	' Objets initiaux >'	Déclarations_Mobjets &RETOUR
	' Objets finaux >'	Déclarations_Mobjets &RETOUR
	{ Objets_implicites }	
Construction_Msimple =	Outil { '(' Liste_Mparamètres ')' }	'->' Déclarations_Mobjets &RETOUR
Liste_Mparamètres =	MParamètre_ensucré { &VIRGULE MParamètre_ensucré }	
MParamètre_ensucré =	Présucré Ident_MObjet Postsucré	
Déclarations_Mobjets =	Déclaration_Mobjet { &VIRGULE Déclaration_Mobjet }	
Déclaration_Mobjet =	Type_objets Ident_MObjet	
Objets_implicites =	Objets_implicites ' >'	Déclarations_Mobjets &RETOUR
Ident_Objet =	Nom_utilisateur Nom_automatique	
Nom_utilisateur =	"ES NOM_OBJET "ES Indice	
Nom_automatique =	NOM_AUTO	
Ident_MObjet =	'# NOM_AUTO	

2°) Appel et fonctionnement de la vue textuelle

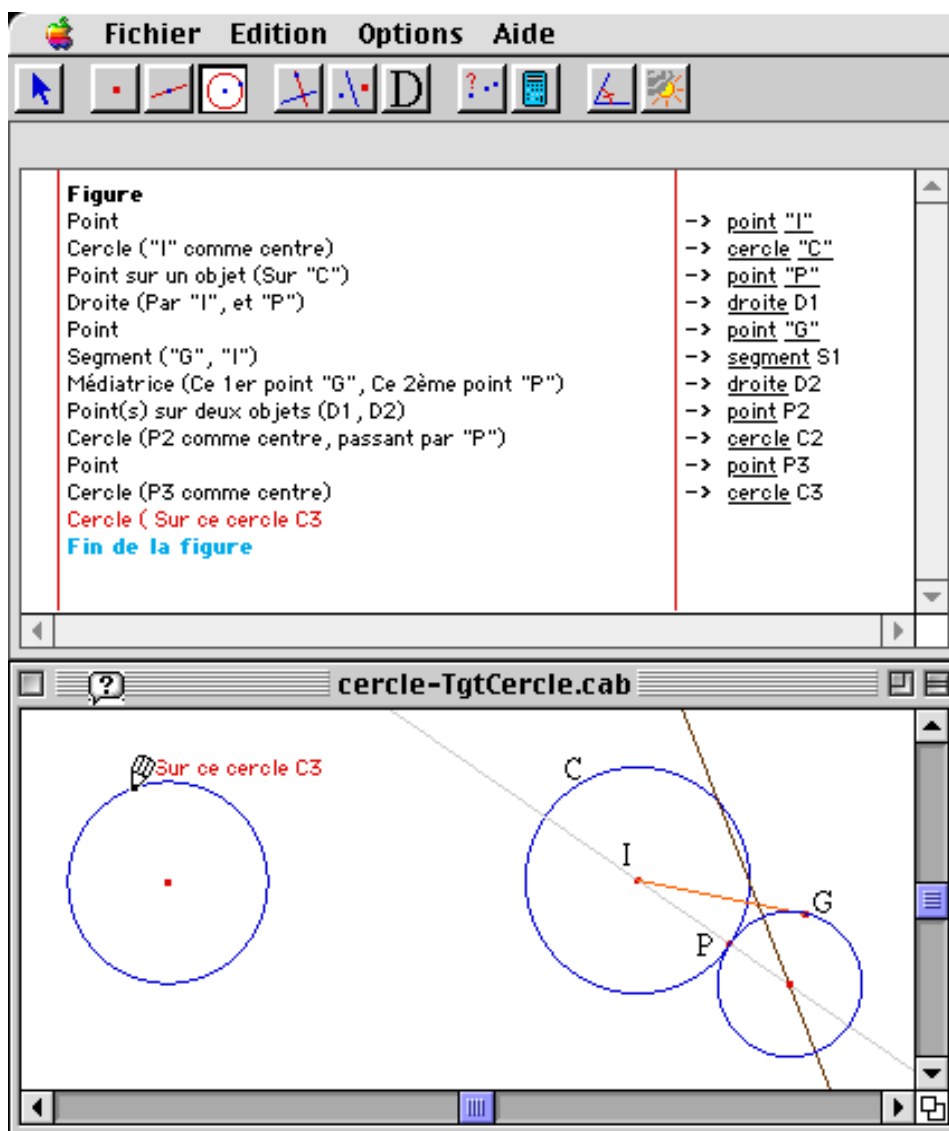
L'ouverture de l'édition textuelle se fait depuis l'item **Vue Textuelle** du menu **Edition**, juste avant **Revoir la construction**.

¹ En plus des notations précédentes, pour les caractères spéciaux, nous prenons la convention utilisée en html :

le retour chariot est noté &RETOUR, la virgule, &VIRGULE,
les doubles quotes, "ES, et la tabulation, &TAB.

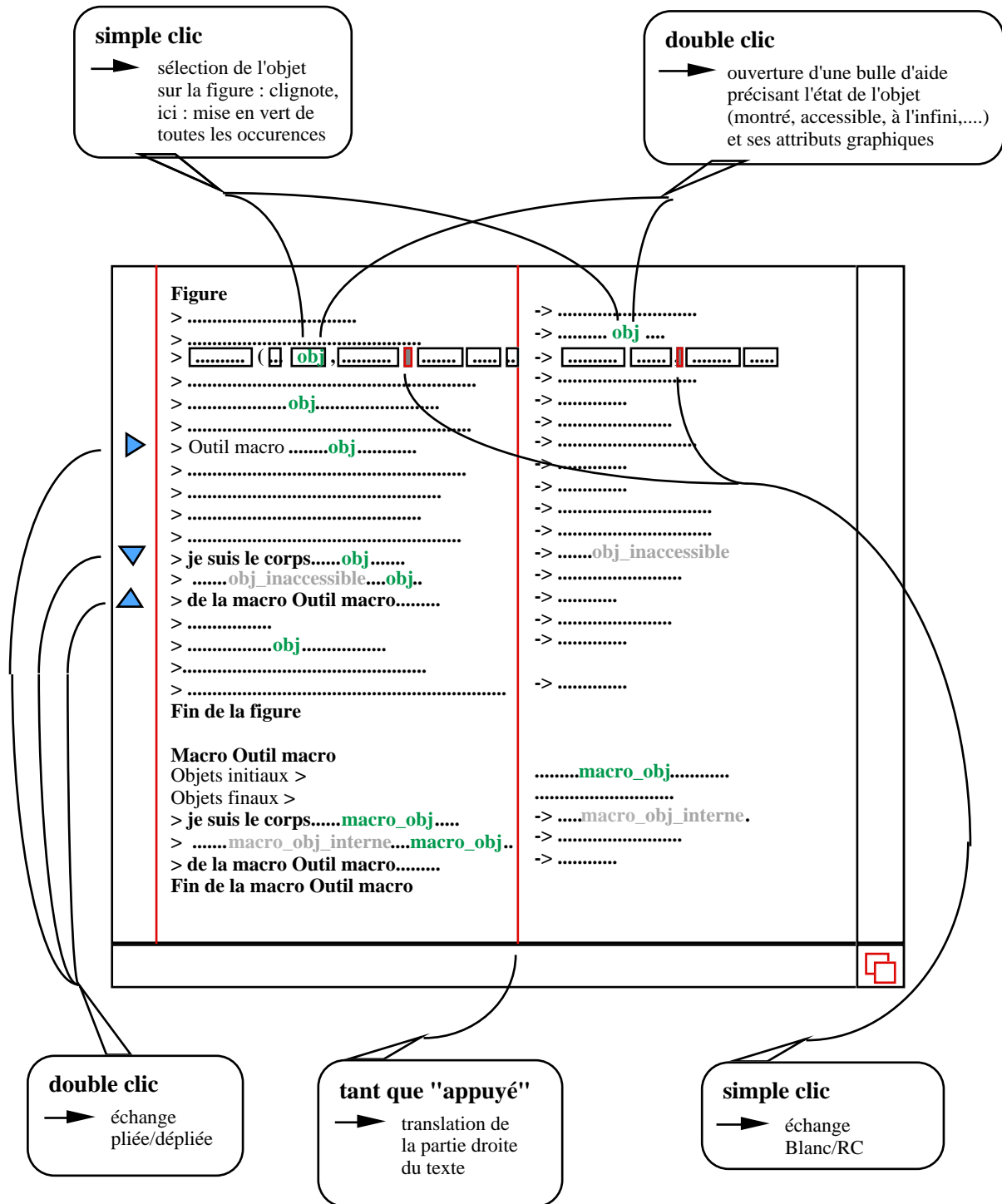


On peut continuer à construire la figure depuis la vue géométrique avec répercussion (écho) des constructions effectuées dans la vue textuelle, ou poursuivre la construction depuis la vue textuelle avec répercussion des constructions dans la vue géométrique.



Le seul problème de notre version prototype est qu'elle ne permet pas de détruire des objets quand la vue textuelle est active. Le palliatif est de fermer la vue textuelle pour détruire des objets, et de la rouvrir ensuite pour s'informer de l'effet de cette destruction.

Le schéma ci-dessous synthétise les diverses manipulations permises et résume leurs effets.



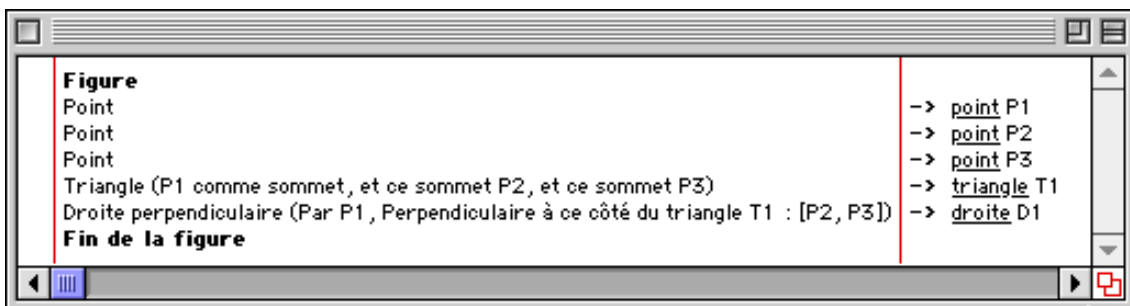
En reprenant la liste des désirs énumérés dans la section II-1-C-2°-b, nous présentons le fonctionnement général de la vue textuelle. Dans la suite, nous notons entre parenthèses l'indice dans cette liste de la fonctionnalité concernée par l'explication courante.

a. Le texte

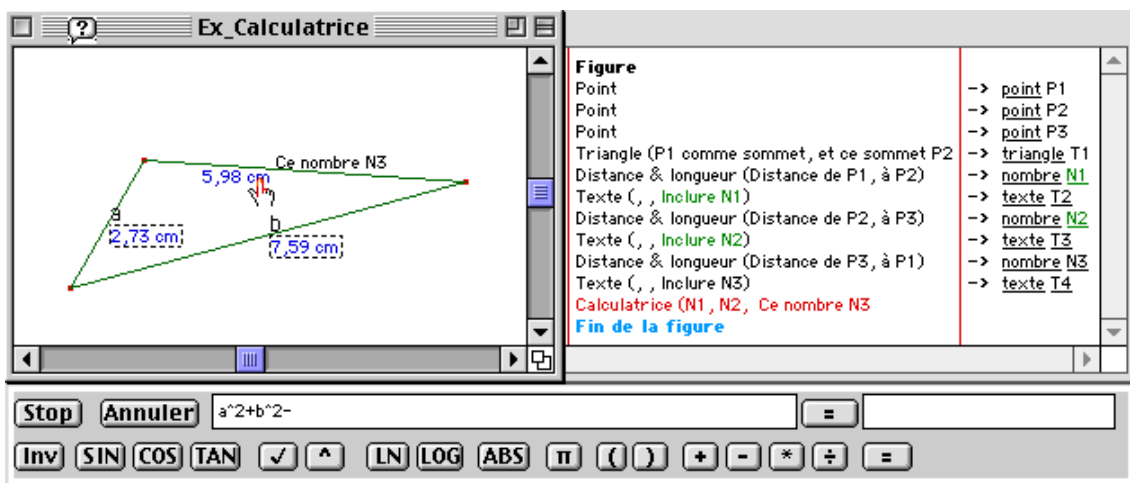
La première fonctionnalité attendue (1) est la préparation et l’affichage du texte du programme qui décrit la construction d’une figure et le contenu des macros utilisées. La méthode suivie consiste à récupérer les éléments de texte affichés par l’interface dans la vue résultat, et cela tout en gérant des spécificités comme les côtés des triangles ou des polygones, et les formules pour l’outil Calculatrice.

La dernière illustration du §II-1-A-3°-a est une copie de la fenêtre textuelle qui décrit l’exemple de la construction et de l’appel de la macro qui construit un cercle tangent à un autre cercle (§II-1-A-1°-a).

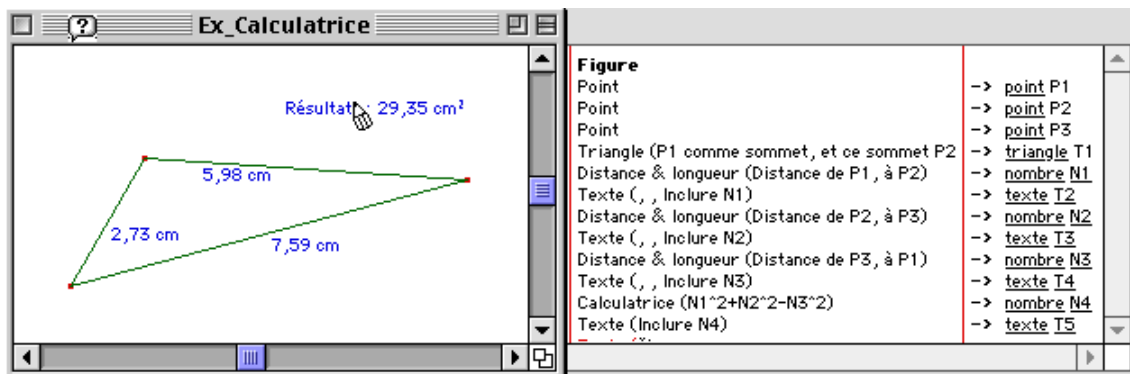
L’exemple suivant montre le texte résultant de la construction d’un triangle et de la perpendiculaire à un de ses côtés.



Nous montrons ci-dessous deux étapes de la construction d’un nombre par l’outil Calculatrice. La première copie d’écran présente l’état du logiciel pendant la saisie de la formule.



La deuxième copie d’écran montre le programme et la figure lorsque le nombre qui porte le résultat du calcul est « construit ».

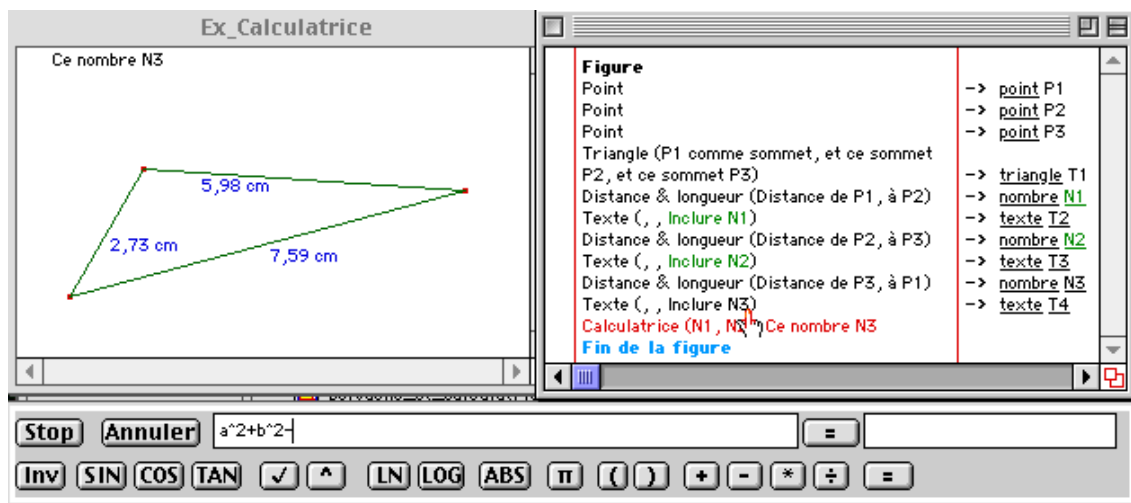


b. Gguidage de l'utilisateur

(2a) Le prototype répercute dans le programme en cours de saisie, le guidage de l'utilisateur pendant les constructions d'objets.

Dans la vue géométrique, Cabri-géomètre signale à l'utilisateur les objets déjà sélectionnés pour l'outil courant : leurs représentations graphiques clignotent et l'objet en cours de construction est suggéré par un tracé en pointillé qui suit les mouvements de la souris. Dans la vue textuelle, le prototype construit la ligne de programme au fur et à mesure des sélections et propose de prendre en compte les objets survolés possibles pour l'outil courant. Les réactions des deux vues sont synchronisées et les sélections des objets peuvent être effectuées indifféremment depuis l'une ou l'autre des deux vues.

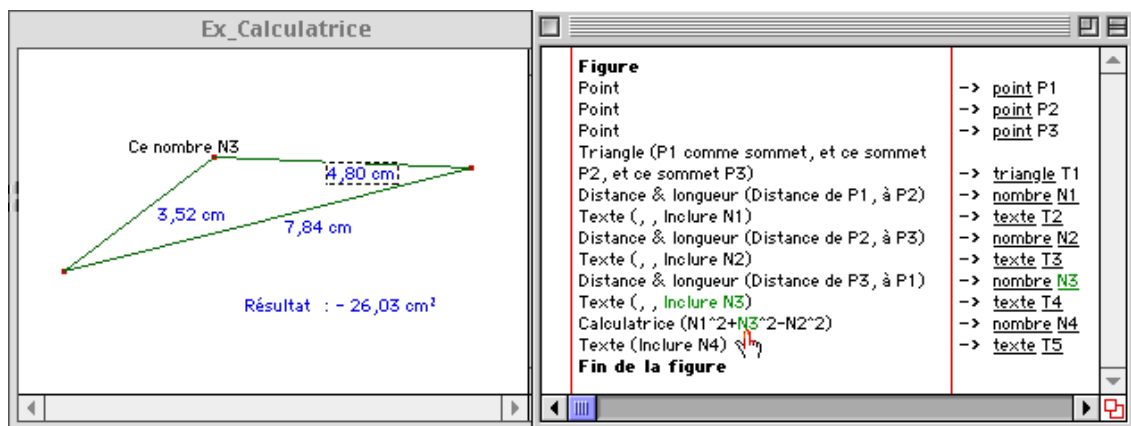
Dans le premier exemple sur la calculatrice (ci-dessus), la fenêtre géométrique est la fenêtre active, et la fenêtre textuelle répond passivement par écho.



La copie d'écran ci-dessus présente le schéma inverse : la vue textuelle est la vue active et la vue géométrique répond en écho.

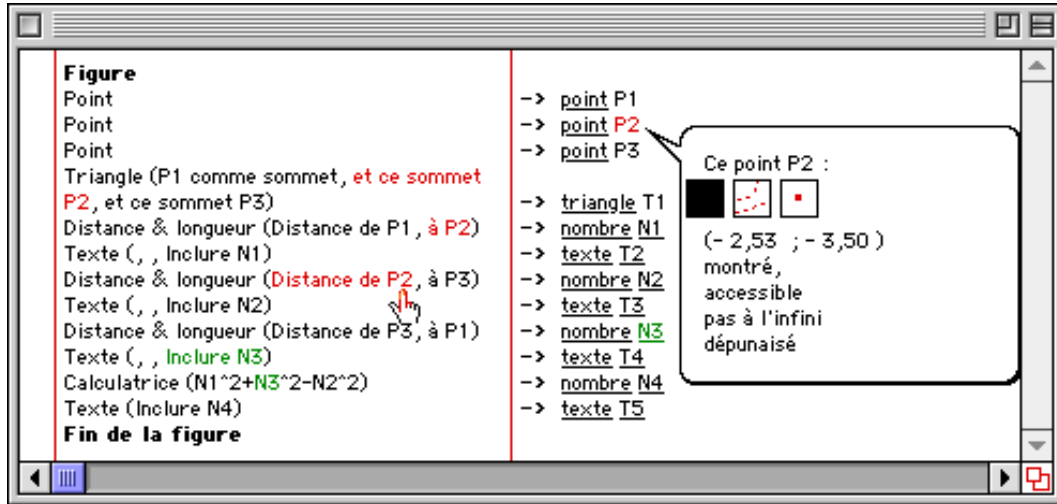
c. Réactivité des identificateurs des objets

(3) Les outils de construction et l'outil « pointeur » réagissent sur toutes les occurrences de chaque objet, de la même manière que dans la vue géométrique : lorsque le curseur survole une occurrence de l'identificateur d'un objet sur lequel la construction courante peut se baser, le retour d'information prévu est affiché dans la vue géométrique, près de la représentation graphique de l'objet.



Si l'utilisateur sélectionne cet élément de texte, par un clic sur la souris, alors toutes les occurrences de son identificateur sont affichées en vert, et la représentation graphique de cet objet se met à clignoter.

d. Affichage des attributs



(4a) Le prototype affiche les attributs graphiques et manipulateurs d'un objet lorsque l'utilisateur clique deux fois sur une occurrence quelconque de son identificateur. Pour les points, il affiche aussi ses coordonnées. Toutes les occurrences de cet objet sont alors affichées en orange, et l'état des autres objets peut ne pas être modifié.

Avec le comportement général de la vue textuelle présenté dans cette section, nous avons principalement décrit les fonctionnalités d'édition, c'est-à-dire les fonctionnalités qui permettent de saisir un programme de construction en voyant son texte s'afficher.

3°) Fonctionnalités aidant au débogage

En dehors de la sélection simultanée dans les vues programme et résultat, il reste quelques fonctionnalités implémentées dans le prototype et utiles pour aider l'utilisateur à maîtriser ce que ses programmes produisent effectivement, et à mettre au point ses figures. Comme nous l'avons vu au §II-1-C-2°-b, ces fonctionnalités concernent essentiellement les notations secondaires (8) et la navigation dans la structure logique du programme : navigation horizontale (5a) liée à l'ordre de l'exécution et navigation verticale (9) pour accéder à une représentation plus ou moins abstraite du programme.

Pour illustrer ce paragraphe, nous reprenons l'exemple de la macro qui construit un cercle tangent à un autre cercle.

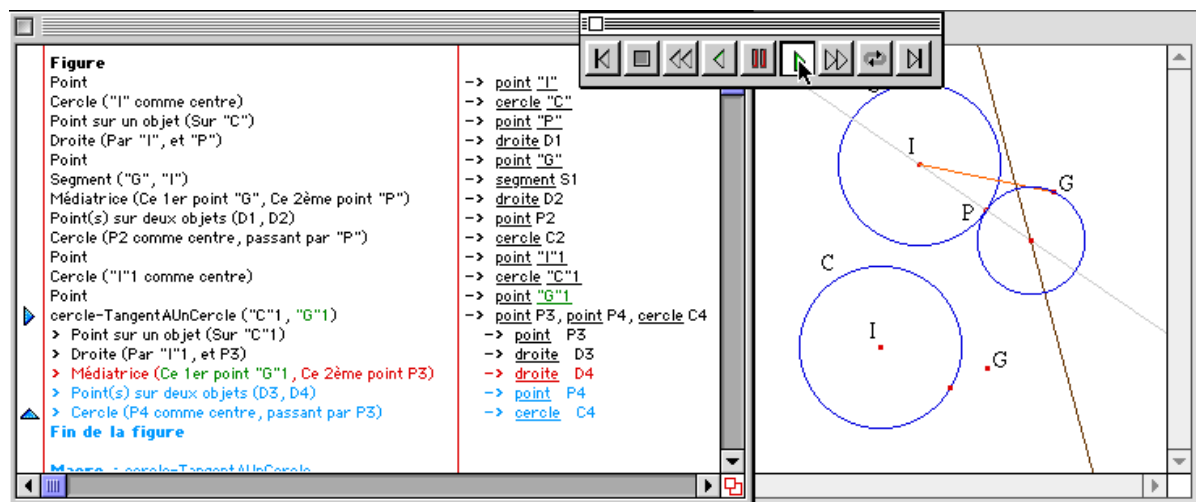
a. Revoir la construction et curseur textuel

La vue textuelle peut être utilisée en même temps que l'outil **Revoir la construction**. Cet outil permet de se déplacer dans la liste des constructions mémorisée par le logiciel, c'est-à-dire dans le programme qui spécifie la construction. Les déplacements sont commandés par des pressions sur les boutons d'un magnétophone.



On peut ainsi provoquer une « exécution » pas à pas de la figure, une avance rapide (de dix pas) ou complète, et un retour au début, un retour d'un pas, ou un retour rapide. Dans la fenêtre graphique, l'utilisateur voit apparaître les objets un à un selon les boutons utilisés.

Le prototype présente un curseur textuel lié aux manipulations du magnétophone. Ce curseur simule les manipulations de l'utilisateur pour construire la figure : la progression des sélections des objets est associée à la progression des couleurs dans le texte du programme.

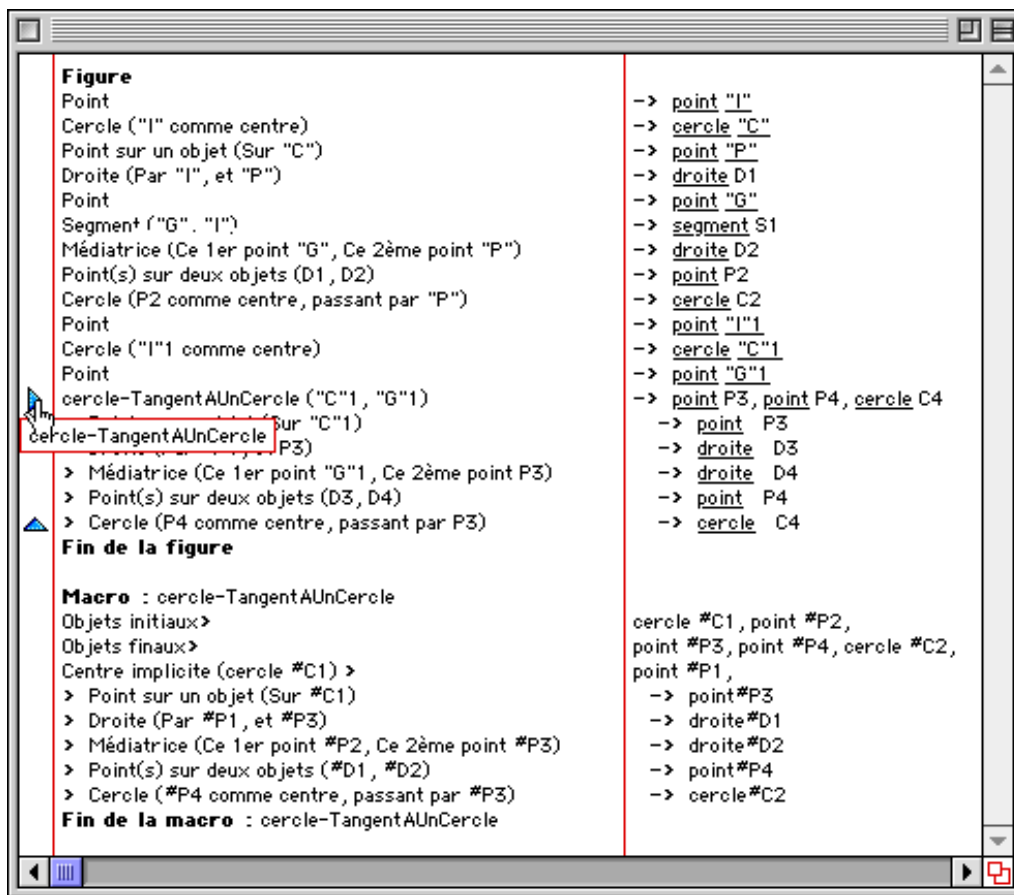


L'image ci-dessus montre l'état de l'écran pendant la simulation de la sélection du deuxième point G pour construire la médiatrice D4, interne à l'exécution de la macro. Le texte du programme écrit en noir correspond aux constructions déjà exécutées, le texte en rouge à la construction en cours, et le texte en bleu à ce qui restera à exécuter.

b. Pliage et dépliage des macros

Le prototype gère deux modes d'affichage pour les appels des macros : plié et déplié. Dans l'image précédente, la macro était dépliée. Pour la plier, il suffit à l'utilisateur de cliquer sur l'une des petites flèches situées sur la gauche de la fenêtre. Lorsque le curseur survole l'une d'elles, le prototype lui accroche une étiquette portant le nom de la macro concernée. Ce retour d'information est utile pour relier la flèche de fin du dépliement de la macro avec la macro concernée, en particulier dans l'éventualité que le logiciel permette ultérieurement l'imbrication des macros. Il permet aussi d'informer l'utilisateur d'une action possible.

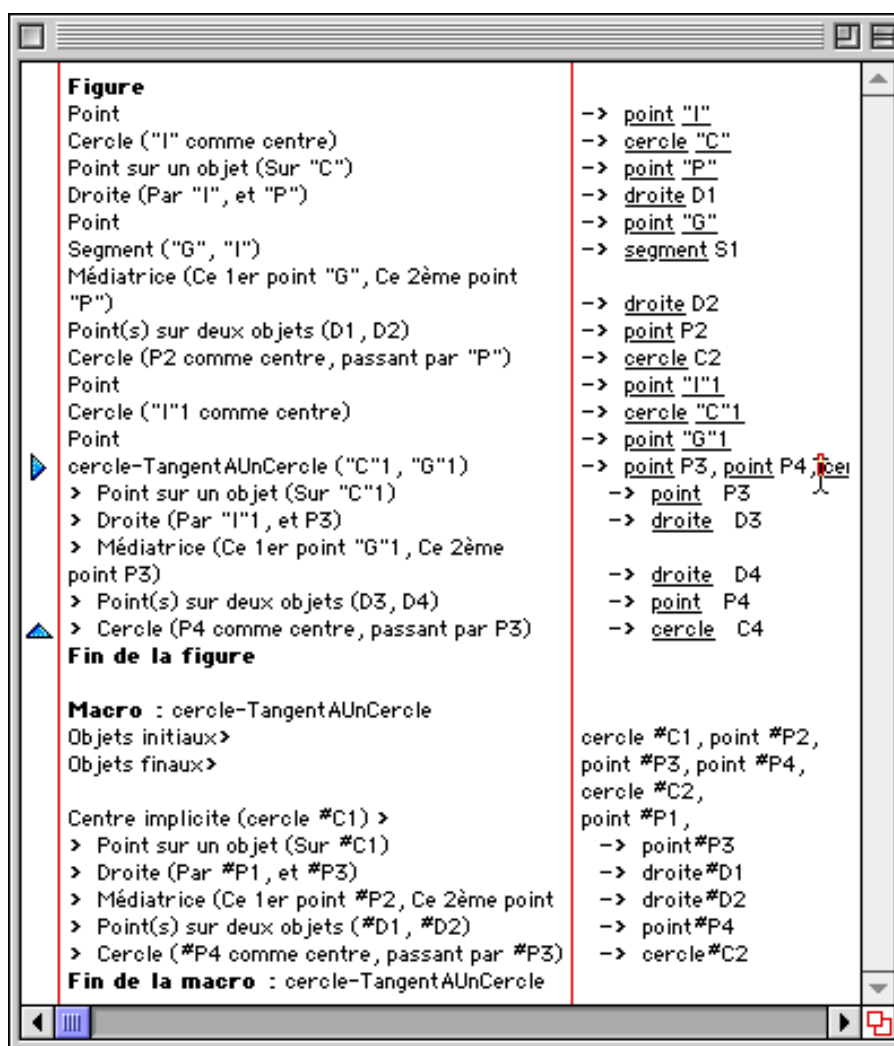
L'illustration ci-dessus montre cette action de l'utilisateur.



c. Notations secondaires

Deux principales raisons interviennent dans le choix de la présentation du programme.

La première raison est de mettre en évidence la structure hiérarchique du programme, par l'utilisation d'une indentation, mais on peut considérer que le repliement des macros en est aussi un moyen. La présentation utilisée doit être systématique pour être efficace, c'est pourquoi cette tâche est attribuée à l'éditeur lui-même. Dans le prototype, nous avons implémenté une indentation dans les deux zones de programme : la liste des constructions et la liste des objets construits.



Lorsque les lignes (du support) ne sont pas assez longues pour que toute une ligne de programme soit visible, l'utilisateur a besoin d'insérer des passages à la ligne qui n'ont pas de rapport avec la syntaxe. Ces ajustements ponctuels de présentation du programme, avec la rupture de lignes constituent la deuxième raison de nos choix.

Par exemple, nous avons utilisé le passage à la ligne pour couper la ligne de construction du triangle dans les illustrations sur la transposition textuelle de l'outil Calculatrice (§II-3-A-2°). Dans l'illustration ci-dessus, nous montrons l'aspect du curseur lorsqu'il survole des caractères blancs où peuvent être insérés les sauts de ligne. Lorsque le saut de ligne est inséré dans la partie gauche du texte, la liste des objets construits dans la partie droite descend aussi d'une ligne.

Ces insertions doivent être cohérentes avec les autres accès de l'utilisateur à la présentation : par exemple, lorsque l'utilisateur a inséré des sauts de lignes dans une macro, il doit retrouver sa présentation lorsqu'il plie et déplie cette macro, une fois ou plusieurs.

B) Évaluation

L'évaluation présentée ici doit contribuer à mesurer l'apport des fonctionnalités introduites dans le logiciel pour répondre aux besoins présentés au §I-1. Elle n'a pas pour objectif d'évaluer le prototype de manière absolue, mais relativement à la version officielle du logiciel.

Dans la section §I-3-D, nous avons regroupé les critères en deux groupes, selon le centre d'intérêt de l'évaluation : l'interface indépendamment du domaine, et l'adéquation de l'interface au domaine en examinant plus précisément les fonctionnalités offertes.

Comme nous n'introduisons pas de nouvelles fonctionnalités liées directement au domaine d'application du logiciel, le prototype ne contribuera à la qualité du logiciel vis-à-vis des critères du deuxième groupe que par l'extension de ses fonctionnalités et par la pertinence des choix associés. Nous avons décrit ces aspects au §II-1 pour les désirs et au §II-3-A pour les réalisations effectives.

Dans cette section, nous examinons les critères d'interface généraux, puis nous nous concentrons sur les limitations fonctionnelles, et nous terminons par une évaluation quantitative.

1°) Évaluation selon les critères d'évaluation généraux

Revenons sur les critères d'interface présentés dans la première partie (§I-3-D). Parmi les critères qui caractérisent une IHM, nous n'avons gardé que ceux qui permettent aux utilisateurs de programmer et les y aident. Il restait trois catégories de critères, caractérisant respectivement l'ergonomie, les accès à la sémantique des programmes construits, et le comportement de l'interface.

a. Ergonomie

Les nouvelles possibilités introduites pour manipuler les données sont l'usage de la souris pour sélectionner les objets, pour plier et déplier les macros, et pour insérer des sauts de ligne. Le choix de bulles d'aide, pour donner accès aux attributs qui sont associés aux objets, a aussi des conséquences sur les possibilités de manipulation.

Toutes les occurrences des identificateurs d'objets sont réactives. Dans la liste des constructions, la zone sensible recouvre l'identificateur et le sucre syntaxique. Dans la liste des objets construits, elle ne recouvre que l'identificateur, et pas le type de l'objet. Peut-être serait-il préférable d'étendre cette dernière zone de sélection, et ce serait immédiat à implémenter. Mais, dans cette première version, nous avons recherché la concision.

Dans la version du prototype présentée ici, nous n'avons pas encore implémenté les manipulations du texte du programme décrites au §II-1-C.

Parmi les critères d'ergonomie visuelle figurent les modes de représentation de l'état sélectionné. Dans le prototype, nous avons choisi différents modes de sélection selon la tâche exécutée par l'utilisateur.

Les sélections éphémères utilisées à des fins consultatives sont concrétisées par un changement de couleur du texte concerné. Différentes couleurs sont utilisées lorsque les fonctionnalités peuvent être utilisées en même temps. Par exemple, les objets sélectionnés pour les constructions sont écrits en vert, et les objets sélectionnés pour consultation de leurs attributs sont écrits en orange.

Dans le prototype, ce sont les seules sélections déjà implémentées. Des sélections plus durables sont prévues pour les manipulations plus conséquentes du texte, c'est-à-dire qui agissent sur la structure du programme (destruction d'objets, extraction de constructions pour les macros, redéfinition des contraintes). Les deux modes de sélection pourront être conjugués.

b. Accès à la sémantique

Le premier aspect aidant à l'assimilation de la sémantique est la qualité de la manipulation directe. Nous avons vu les caractéristiques de ce mode d'interaction dans les IHM. Dans le cadre de l'insertion de nouvelles représentations des données, les exigences de ce mode d'interaction demandent un soin particulier pour l'implémentation des fonctionnalités de manipulation des données.

Comme la possibilité de modifier les valeurs algébriques des objets n'a pas encore été introduite, les animations et déformations de figures ne peuvent être pilotées par la vue textuelle des figures. Par contre, le retour d'information fourni en mode construction est très proche de celui qui est produit dans la vue résultat.

Les nouvelles métaphores introduites sont :

- les petits triangles qui symbolisent le repliement des macros, matérialisant leur début et leur fin,
- l'étiquette qui s'accroche à ces petits triangles,
- la barre de séparation des deux zones de texte, et le curseur de déplacement de cette barre.

Ces symboles sont aujourd'hui classiques et utilisés dans différentes interfaces. Par contre, nous avons introduit une invite devant chaque liste d'objets construits. Cette invite est une flèche, symbole choisi pour signaler que ce qui suit résulte de l'exécution de ce qui précède.

N'ayant pas introduit d'instructions conditionnelles, nous n'avons pas eu besoin de choisir une représentation pour elles, donc pas non plus de métaphore associée.

Le nommage automatique des objets laisse l'utilisateur libre de nommer les objets sans contrainte tant que le texte du programme n'est pas saisi par un éditeur externe. Il sera alors nécessaire d'introduire un séparateur entre la syntaxe du programme et la syntaxe des noms d'objets choisis par l'utilisateur. Le nommage automatique des objets est ajusté pour répondre à un compromis (§II-2-A-2°-a). Le type de l'objet est rappelé à l'utilisateur par la première lettre des noms automatiques des objets.

Le langage de programmation tient compte de la culture de l'utilisateur, puisqu'il utilise la langue de dialogue choisie par celui-ci.

Pour aider l'utilisateur à assimiler ce que font effectivement ses programmes de construction de figure, le prototype gère l'introduction de notations secondaires, avec des indentations et une présentation ajustable par insertion de sauts de lignes. Il ne gère cependant pas encore les commentaires pour la vue structurelle.

La difficulté des opérations mentales est liée au paradigme de programmation. Dans les micro-mondes constituants des environnements d'apprentissage, le paradigme choisi a pour motivation d'obliger l'utilisateur à rechercher des solutions en mettant à contribution certaines ressources cognitives. Le critère n'est donc pas ici d'évaluer les difficultés des opérations mentales, mais de choisir celles qui sont utiles à l'utilisateur. L'objectif de la vue textuelle d'un programme de construction est de donner à l'utilisateur des moyens pour identifier les décalages entre ses désirs et ses réalisations.

Dans ce cadre, les aspects qui contribuent à minimiser la difficulté des opérations mentales sont ceux qui concrétisent les effets de chaque instruction du programme, ou qui libèrent l'utilisateur d'efforts de mémoire. Le prototype répond à ce critère par l'ubiquité des objets dans la vue programme et dans la vue résultat, ainsi que par la mémorisation et l'explicitation des relations de dépendance entre les objets.

c. Comportement

En ce qui concerne les modifications du comportement du prototype par rapport au logiciel, nous n'avons pas introduit de dialogue, ni aucun autre artefact qui aurait pour conséquence de déconcentrer l'utilisateur de sa tâche en cours pour répondre à des exigences de l'interface. Nous n'avons donc pas introduit de rupture d'engagement direct.

Avec la vue textuelle comme elle a été implémentée, certaines opérations ne sont plus immédiatement réversibles : le schéma du processus à exécuter par l'utilisateur pour mettre à jour le texte après des destructions d'objets est composé de plusieurs étapes. Nous avons donc introduit des engagements prématurés.

Parmi les fonctionnalités introduites qui contribuent à satisfaire le critère de consistance et d'uniformité des outils de l'interface, il y a les rendus choisis pour marquer les objets sélectionnés. Ces rendus sont consistants par rapport à l'activité de la fonctionnalité : selon que la fonctionnalité utilisée est passive ou active par rapport à la structure du programme, le rendu est un changement de couleur du texte ou une inversion vidéo.

Le critère de cohérence est aussi satisfait, c'est-à-dire qu'une même suite d'actions de l'utilisateur aboutit au même type de réaction du logiciel. En effet, comme le prototype est piloté par le noyau fonctionnel du logiciel, nous n'avons pas introduit d'incohérence.

Dans cette version du prototype, la destruction des objets contribue négativement à l'interface par rapport au critère de viscosité. En effet, l'effort pour aboutir à la mise à jour de la vue textuelle est important : il faut fermer la vue avant puis la rouvrir, alors que l'utilisateur ne devrait pas avoir à intervenir.

2°) Limitations fonctionnelles

Nous ne présentons pas ici les dysfonctionnements (bogues répertoriés) du prototype, mais les limitations fonctionnelles, c'est-à-dire les limitations dues au fait que certains des désirs présentés dans la section II-1-C-2°-b ne sont pas supportés par le prototype. Mentionnons :

- la destruction d'objets (2b), en dehors du palliatif décrit au §II-3-A-1°,
- la modification des attributs (4b),
- les trois simulations décrites au §II-1-C pour prévenir des destructions d'objets (2b), visualiser l'extraction des commandes pour la définition des macros (7a) et pour la redéfinition des contraintes (7b),
- les déplacements d'éléments textuels (6).

On peut rajouter à ces fonctionnalités manquantes, l'impression et la sauvegarde du texte produit, ainsi que l'interprétation des textes de programmes saisis avec un éditeur externe.

Le prototype ne permet pas de détruire directement les objets depuis la vue textuelle, et la mise en œuvre de cette fonctionnalité nécessite une refonte des structures de données principales utilisées.

En effet, ce prototype a été initialement réalisé pour fournir rapidement un moyen d'accéder à la structure du programme de construction mémorisée par le logiciel ainsi qu'au contenu des macro-constructions.

En conséquence, un choix « d'urgence » a été de préparer le texte du programme dans deux éléments de texte seulement, un pour la liste des constructions, l'autre pour la liste des objets construits, et de gérer les accès à leurs caractères comme l'accès aux éléments d'un tableau. Ce choix a l'intérêt de permettre un affichage immédiat des deux textes dans leurs zones respectives par l'usage d'une seule procédure prédéfinie, et une gestion simple des déplacements de feuille.

À cause de ce choix, la gestion des ajustements de présentation concrétisées par les possibilités de repliement de macros et l'insertion de sauts de lignes a donné lieu à des traitements très sophistiqués et difficiles à mettre au point. Les procédures mises en œuvre sont incompatibles avec les destructions d'objets.

Ce point est d'autant plus difficile que la destruction d'un objet est elle-même assez compliquée, car tous les objets qui dépendent de cet objet doivent aussi être détruits, et les caractères correspondants ne sont pas contigus dans les éléments de texte. De plus, l'effacement d'un caractère a des conséquences sur les références à tous les caractères situés après lui. La mise au point de ces calculs serait très délicate, et il est plus que vraisemblable qu'ils augmenteraient considérablement le temps de réponse pour l'effacement d'un objet.

Notre méthode d'indexage absolu des caractères qui constituent le texte n'est donc pas la plus adaptée pour le futur. Dans le cadre de ce travail, elle nous a cependant permis de simplifier l'affichage dans la fenêtre.

Le modèle de document adapté à l'interface est un modèle de document structuré, et il faut que la structure de données utilisée pour la gestion de ce texte reflète sa structure au moins jusqu'au niveau des instructions. Les principes des modifications à implémenter sont présentés plus loin au §II-4-A-1°.

3°) Espace – temps

Dans le §II-1-B-1°-a, nous avons présenté les contraintes du prototype liées aux limitations en place mémoire et en temps d'exécution. À cet effet, nous avons présenté un tableau de répartition des coûts en place mémoire du logiciel. Nous reprenons ici ce tableau en y insérant le surcoût dû aux différentes insertions de code pour l'intégration de la vue textuelle. Nous n'avons conservé que les rubriques donnant lieu à un surcoût.

Répartition	Modules	Surcoûts (code compilé)	
		Valeur exacte	Répartition
Squelette d'IHM	Main, menus,	7,1 Ko	26 %
	Fenêtres (ascenseurs...),	10,6 Ko	
	magnétophone	0,7 Ko	
	pas de dialogue ni de préférence supplémentaire	0 Ko	
Gestion graphique du texte	Affichage du texte d'une figure,	7,5 Ko	23 %
	des attributs graphiques	8,7 Ko	
	Édition structurée		
	impression papier du programme non encore implémentée	?	
Gestion de la structure de données	pour les notations secondaires	9,6 Ko	15 %
	pour les références « croisées »	1 Ko	
	rien de plus dans le noyau fonctionnel, constructions, initialisations,	0 Ko	
	sauvegarde du texte du programme non encore implémentée	?	
Gestion des objets	Construction dynamique du texte	3,6 Ko	25 %
	Construction statique du texte	11 Ko	
	Nommage de tous les objets, dont coniques, calculatrice, texte, table et noms	2,9 Ko	
Macros	Construction du texte d'appel, du contenu	7,4 Ko	11 %

La place mémoire requise par le prototype est 1,16 Mo de code et 131 Ko de données, alors que la version initiale prenait 1,09 Mo pour son code et 107 Ko pour ses données. L'augmentation est donc de 70 Ko de code et de 24 Ko de données, soit un surcoût de 7% de code et de 22% de données.

Le surcoût plus important en place des données provient de la gestion d'une structure de données « parallèle » à la structure de données qui mémorise les objets. Ce choix temporaire provient de la volonté d'effectuer tous les traitements relatifs à la gestion du texte du programme en dehors du code déjà existant, et ainsi, pendant la phase de prototypage, de minimiser les risques de « mélange » des sources d'erreurs. Il est donc lié à la contrainte « projet ».

Le tableau suivant présente le pourcentage de code ajouté dans le prototype selon la répartition précédente.

Répartition	Version officielle	Prototype	Proportion de code en plus
Squelette d'IHM	338	18,4	5%
Gestion des sorties	114	16,2	14%
Gestion des structures de donnée	126	10,6	8%
Noyau fonctionnel	381	17,5	5%
Macros	25	7,4	30%

Prolongements, potentiel et perspectives

Dans ce chapitre, nous présentons :

- ce qu'il faudrait faire pour passer du prototype à une version plus finalisée dans laquelle les choix fondamentaux sont fixés.
- ce qu'on pourrait imaginer pour augmenter les applications du prototype, et les retombées de l'approche utilisée sur la conception de nouvelles fonctionnalités pour le logiciel Cabri-géomètre.

Nous présentons d'abord quelques prolongements à réaliser pour passer du prototype à une version plus robuste (qui résiste aux évolutions des désirs), plus complète (qui réponde à toutes les attentes présentées au chapitre §II-1) et plus « orthogonale » (qui permette une édition textuelle à manipulation directe et ne soit plus limitée à une visualisation).

Nous explorons ensuite les potentiels de l'approche et ce que le prototype et l'approche permettent d'envisager pour une prochaine version du logiciel Cabri-géomètre. Dans cette optique, nous présentons nos réflexions sur l'introduction de macros récursives, l'uniformisation des comportements des objets indépendamment de leur type et l'introduction d'une vue basé sur un graphe de la structure logique de la construction. Pour les deux premières propositions, le prototype constitue un support de réflexion pour des améliorations du logiciel et de son noyau fonctionnel lui-même.

A) Prolongements

1°) Passer d'une version linéaire à une version hiérarchique

Nous avons vu que la structure de données choisie pour mémoriser le texte (deux énormes chaînes de caractères pour les deux composantes du langage de géométrie formelle) n'est pas adaptée au déplacement d'éléments de programme.

Avec ce choix, les informations à mémoriser pour supporter l'ubiquité des objets consistent en une liste des indices du premier et du dernier caractère de chaque occurrence réactive de chaque objet. La gestion de notations secondaires constitue un point délicat de la mise au point de cette version du prototype.

Nous avons aussi vu que la technique à implémenter pour remédier à ce problème est d'utiliser une structure des données plus hiérarchique (§II-2-B-1). La raison pour laquelle nous ne l'avons pas fait directement dans cette première version est qu'une décomposition entièrement hiérarchique prendrait trop de place mémoire pour chaque programme, et qu'en conséquence il fallait se tourner vers une version mixte.

Les priorités de développement nous ont alors conduite à implémenter une première version avec des fonctionnalités de manipulation minimale, mais permettant de valider l'approche et d'obtenir un outil concret pour déterminer tous les autres choix pertinents à mettre en œuvre.

Quelles sont les modifications à apporter à la structure de données pour rapprocher notre implémentation de celle des documents structurés, afin de mieux gérer l'édition du programme et l'ubiquité des objets ?

D'abord, le texte du programme est construit à partir des différents mots-clés qui sont mémorisés dans le logiciel et qui sont utilisés dans le contexte de la construction de chacun des objets par un outil. Au plus bas niveau, le prototype génère automatiquement des noms pour désigner les objets.

Bien que la structure du texte soit linéaire, les traitements mis en œuvre pour sa construction sont regroupés selon le niveau du texte qu'ils gèrent.

Pour la partie gauche, celle qui décrit les constructions des objets,

- un traitement est spécialisé dans la construction du sucre syntaxique,
- un autre insère les identificateurs des objets concernés,
- un autre encore construit la liste des paramètres entourés de leur sucre et mémorise les informations utiles à son accès,
- un dernier préfixe le tout par le nom de l'outil utilisé.

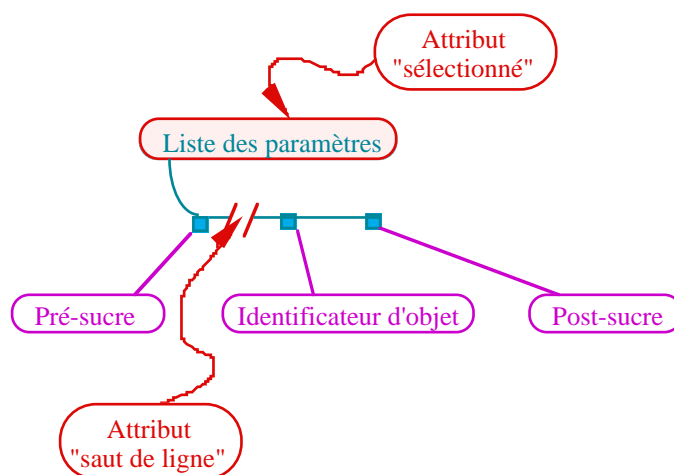
De même, la composition du texte de la partie droite est structurée. Les procédures que nous avons implémentées correspondent donc exactement à des éléments de la structure du document constitué par le programme.

Nous avons simulé une composition structurée des programmes pour l'accès aux identificateurs des objets, en mémorisant l'information utile au calcul de leur encombrement et de leur position. Ensuite, la notion de lien entre les différentes occurrences des identificateurs des objets est proche du concept de *référence* de Thot. Les animations statiques peuvent être basées sur des affectations de valeurs temporaires à des attributs attachés à certains éléments. Dans cette utilisation, l'attribut est attaché à n'importe quel élément de la structure générique du document (i.e. à l'instanciation de la structure logique du document, sans le contenu du document lui-même), et modifie sa représentation.

Le niveau de décomposition choisi impose des calculs compliqués pour retrouver les manipulations résultant de l'insertion des notations secondaires. Il n'est pas possible de trouver une gestion robuste des notations secondaires en gardant ce choix de modèle linéaire pour le texte de programme.

Nous n'avons pas mémorisé les informations utiles pour délimiter les constructions, car, de toutes façons, permettre leur manipulation tout en continuant à permettre les échanges entre caractères blancs et retours-chariot est ingérable. En effet, ces insertions doivent être locales et relatives aux éléments qu'elles affectent pour « suivre » les manipulations sans effort.

Un attribut attaché à n'importe quel élément de la structure du document pourrait mémoriser la modification de présentation qui doit affecter cet élément, avec l'information nécessaire pour spécifier comment : les échanges entre blancs et retour-chariots affectent les constructeurs, par exemple en agissant entre le second et le troisième élément de la liste.



Un compromis intéressant pour définir le niveau de décomposition d'une version plus robuste sera d'utiliser une structure de données hiérarchique jusqu'au niveau des constructions simples, et linéaire ensuite.

Il y aura à cela plusieurs avantages :

- cette décomposition est adaptée pour déplacer des constructions ou en détruire, simuler le processus d'extraction des macros,
- les insertions de sauts de ligne seront relatives aux constructions, et ne remettent pas en cause les ajustements Partie droite/Partie gauche,
- les calculs de boîtes sont simplifiés, car celles-ci n'ont qu'une composante verticale,
- la mémorisation des informations nécessaires pourra être directement intégrée dans les structures de données de Cabri car elle est associée à chaque objet construit,
- la gestion des références devra seulement être étendue de façon à identifier la construction qui utilise les identificateurs.

Le choix d'implémentation est alors d'utiliser un élément textuel pour chacune des parties : un pour décrire la construction, l'autre pour le texte de déclaration de l'objet.

Les attributs qui servent à gérer les insertions de sauts de ligne ne peuvent agir qu'à l'intérieur des constructions.

Les attributs qui servent à gérer le dépliement et le repliement des macros agissent sur le constructeur des appels de macros, qui peut être un constructeur spécifique.

2°) Traiter uniformément toutes les manipulations directes

Avec ce choix de nouvelles structures de données pour mémoriser le texte du programme, les manipulations « directes » du texte de la structure, décrites dans la section §II-1-C-2-a, pourront être implémentées en associant un attribut à chaque "simple construction" de Cabri, c'est-à-dire à chaque élément de base choisi pour la structure effective du document. Cet attribut permettra de modifier l'affichage de tout l'élément et donc de toute une construction, par l'application d'un fond coloré.

Dans l'environnement Cabri, ce rôle de contrôle des manipulations permises sur la structure est exercé par le noyau fonctionnel de Cabri lui-même. En effet, il existe déjà dans la version de base l'outil de redéfinition des contraintes, et la possibilité d'éliminer les objets. Il s'agit d'implémenter le rendu textuel de ses manipulations.

Pour la destruction des objets, il restera à insérer des modifications des attributs associés aux constructions des objets à détruire, et de retarder la boucle générale d'attente d'événements par une attente d'un laps de temps, dans cette même procédure. Pour l'extraction de macro, la stratégie peut être la même que pour simuler les destructions.

Par contre, la redéfinition d'une contrainte demandera l'ajout d'une nouvelle fonctionnalité au noyau fonctionnel. En effet, lorsque l'utilisateur a sélectionné une construction, tant qu'il ne relâche pas la souris, le logiciel peut suivre les déplacements de la souris et déplacer une ombre de la construction sélectionnée, en suggérant par un trait horizontal les endroits où l'utilisateur peut insérer la construction sans redéfinir de contraintes, et par une identification des objets construits pour redéfinir une construction. Le logiciel doit calculer continuellement si la nouvelle position est possible, c'est-à-dire si elle est compatible avec l'ordre induit par les contraintes qui relient les objets référencés par la construction manipulée et les autres constructions.

3°) Modifier les macros

Dans la version actuelle du prototype, les macros ne sont pas modifiables, alors que cet objectif constituait une des motivations de sa conception. Nous n'avons pas non plus présenté de scénario de fonctionnement et d'utilisation pour qu'une extension de Cabri-géomètre supporte les modifications du contenu des macros.

Pour définir une macro, l'utilisateur sélectionne, sur une figure servant d'exemple d'école, les objets dont il veut que la construction parte et ceux auxquels il veut qu'elle aboutisse. Nous avons décrit le processus d'extraction des constructions utiles au §I-1-B-1°-c et exposé les causes des difficultés pour les utilisateurs pour mettre au point leurs programmes de construction.

La version du prototype implémentée permet de visualiser le contenu des macros, c'est-à-dire la liste des constructions extraites. De plus, si l'utilisateur appelle une macro-construction dans la construction d'une nouvelle figure, sa description dans la vue textuelle peut être dépliée, et chaque construction exécutée décrite. La vue textuelle permet donc à l'utilisateur de mettre en relation des comportements de sa figure géométrique (donc de son programme) avec les contraintes qui relient les objets qui la constitue.

Bien que la vue textuelle fournisse un nouveau support de réflexion pour l'utilisateur, comme les objets internes aux macro-constructions ne sont pas représentés dans la vue géométrique, le prototype implémenté ne permet pas à l'utilisateur d'identifier exactement les constructions mal choisies lorsque le comportement des objets qu'il construit n'est pas celui qu'il désire.

Il faudrait donc offrir à l'utilisateur un mode « débogage » dans lequel tous les objets construits par les macros seraient visibles. Le risque est que ces objets encombrant la figure et que plus rien ne soit facilement identifiable. On peut imaginer que, pour éviter ce problème, le passage dans ce mode cache les objets inutiles à cet appel de macro, donc tous les objets différents des paramètres choisis et des constructions effectuées par la macro.

L'association de ce mode à l'activation de la vue textuelle du programme, serait un premier pas pour permettre de donner à l'utilisateur des moyens pour remédier aux défauts de ses macros, en modifiant leur contenu.

L'accès à un mode est habituellement associé à l'activation d'un menu qui désactive les items des autres menus non accessibles dans ce mode, comme pour l'item « Revoir la construction ».

Pour simplifier, nous présentons l'exemple du débogage d'une seule macro, basée sur une réalisation de cette macro concrétisée par un appel. La conjugaison de plusieurs débogages simultanés conduit d'ailleurs certainement à des problèmes de mise en correspondance des modifications et des macros.

Dans le mode « débogage » ou « révision d'une macro », tous les objets utiles à l'appel d'une macro doivent être accessibles dans le texte et dans la figure, les autres doivent être non accessibles. La convention classique pour les textes, dans les menus, est de griser les objets non accessibles et de laisser bien nets les autres. L'utilisateur peut alors effectuer les manipulations qu'il désire sur les objets accessibles de son programme, et les modifications correspondantes sont répercutées dans le texte du contenu de la macro. Ce schéma de fonctionnement permet de remédier aux problèmes internes des macros.

Lorsque les modifications à apporter nécessitent l'utilisation d'objets externes aux macros, ces objets doivent être rajoutés dans les objets initiaux. Pour réaliser cette action, deux schémas sont envisageables :

- tout objet non accessible antérieur à l'appel de la macro est sélectionnable. Il devient alors un objet initial et passe dans l'état accessible. Par souci de cohérence, on peut alors aussi imaginer que tout objet postérieur à l'appel de la macro soit sélectionnable, mais qu'alors il devienne un objet final, le processus d'extraction étant alors complété par la simulation de l'extraction des constructions complémentaires.
- avant de passer dans le mode de révision d'une macro, les objets sélectionnés par l'utilisateur sont accessibles. Si ces objets sont utilisés lors des modifications, ils seront ajoutés aux objets initiaux.

Ces deux schémas sont compatibles. Leur mise en œuvre est basée sur un filtre placé entre les textes de construction et les fonctionnalités déjà vues pour manipuler les structures logiques des figures.

B) Potentiel et perspectives

Dans le chapitre I-4, nous avons présenté les apports potentiels d'une vue textuelle donnant accès au programme d'une figure, pour toutes les applications du logiciel. Cette présentation était basée sur une abstraction du concept, puisqu'elle était détachée de la forme effective de la vue et des contraintes de réalisation.

Cette section est plus attachée à montrer les potentiels de notre réalisation, c'est-à-dire comment ce que nous avons réalisé permet d'envisager de nouveaux développements pour le logiciel, en perspective à plus long terme, d'aborder de nouveaux problèmes.

1°) Structures de contrôle et macros récursives

Nous présentons dans une même section quelques réflexions sur l'introduction des structures de contrôle et des macros récursives, car c'est uniquement dans le cadre de la spécification des conditions d'arrêt des appels récursifs que les structures de contrôle sont nécessaires.

En effet, pour les autres usages de conditionnelles, les si-géométriques sont suffisants. De plus, ils sont très proche de l'intuition que les non-informaticiens ont des instructions conditionnelles. A contrario, l'apprentissage des cellules conditionnelles dans les tableurs est quelque chose de difficile à assimiler pour les non-informaticiens.

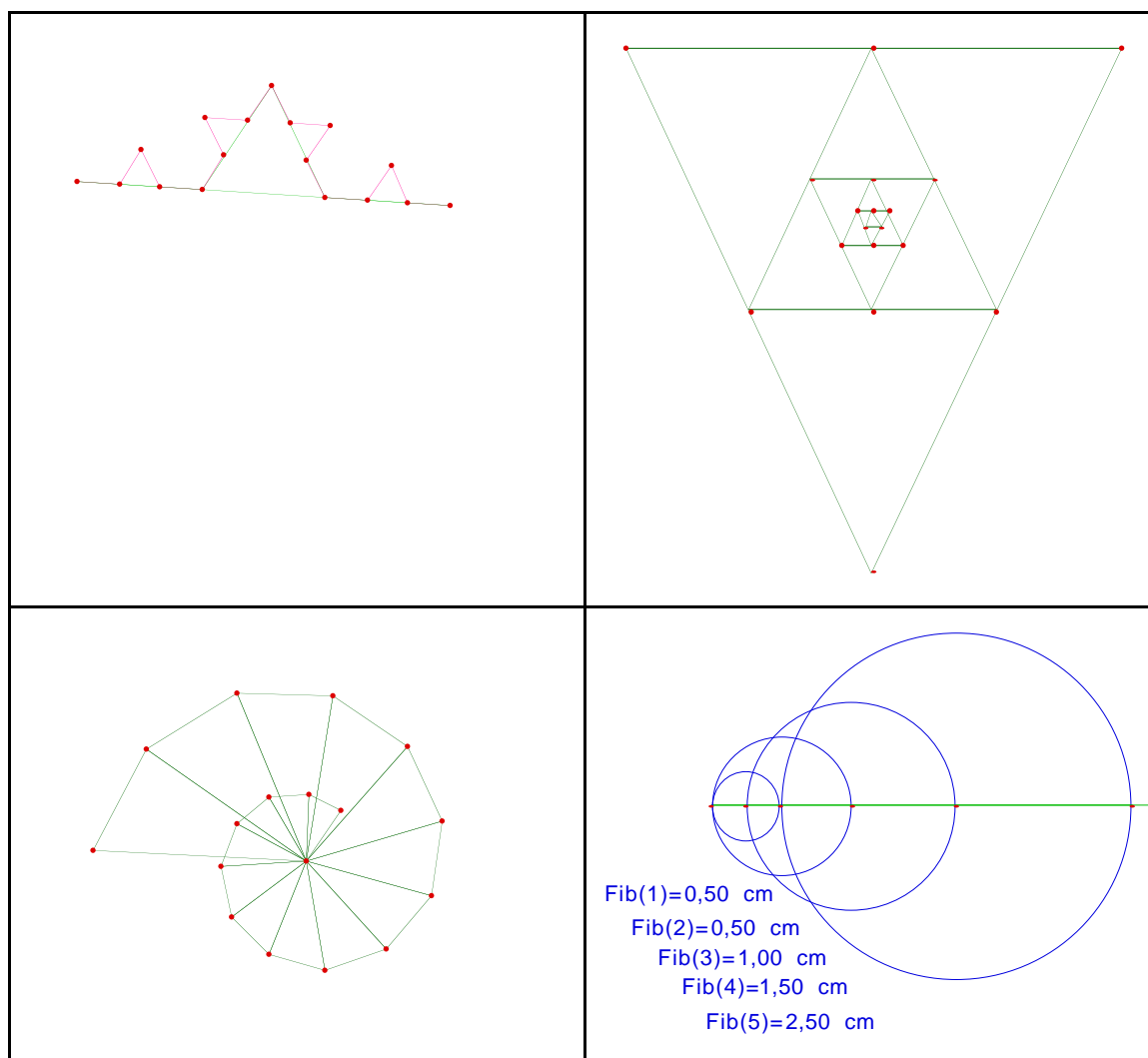
a. La récursivité dans Cabri en géométrie : vers des fractales dynamiques

Comme le dit Douglas Hofstadter dans "Gödel, Escher, Bach, les Brins d'une Guirlande Eternelle" [Hofstadter 79],

"QU'EST-CE QUE LA RÉCURSION ?" [...] l'emboîtement et des variations sur l'emboîtement. C'est là un concept que l'on retrouve fréquemment : histoires à l'intérieur d'histoires, films à l'intérieur de films, tableaux dans les tableaux, poupées gigognes (et même remarques entre parenthèses (me voici !) à l'intérieur de remarques entre parenthèses).[...]

Les fractales sont des constructions géométriques récursives. Ce concept a été inventé par Mandelbrod.

En voici quelques exemples réalisés avec Cabri, bien sûr "à la main", puisque Cabri ne supporte pas la récursion.



En géométrie, tous les objets créés sont conservés, contrairement au calcul récursif (comme celui de $n!$), où tous les résultats intermédiaires, une fois transmis, n'ont plus besoin d'être conservés. En conséquence, le nombre des objets à mémoriser est exponentiel. Les limites de tout ordinateur sont vite atteintes, et il faut prévoir une protection et un dialogue pour prévenir l'utilisateur.

Les conditions d'arrêt de la descente dans la récursion peuvent être liées à la profondeur des appels dans l'arbre de la construction, ou à un seuil sur la dimension d'un des objets générés.

Dans le cadre de la géométrie dynamique, la profondeur doit être modifiable dynamiquement. Pour une condition d'arrêt basée sur une spécification explicite de la profondeur, l'interface doit fournir à l'utilisateur des boutons « + » et « - » ou un ascenseur. Mais l'arrêt par choix d'une dimension seuil est plus pertinent.

Le comportement attendu quand on combinera la géométrie dynamique avec les fractales, est que, lorsque l'utilisateur modifiera les dimensions de l'objet de base, des constructions nouvelles apparaîtront ou des constructions trop petites ou trop grandes disparaîtront. Les objets ne devront être construits effectivement dans la structure du programme que lorsqu'une fractale précise sera demandée : les gérer comme les « si-géométriques » obligerait à mémoriser une infinité d'objets !

De plus, si l'utilisateur construit des objets basés sur des objets générés par la fractale, et si, par animation, ces derniers objets disparaissent, l'utilisateur devra les voir réapparaître s'il effectue une animation inverse.

Ce fonctionnement n'est pas exactement le même que celui d'un « si-géométrique », puisque l'objet sur lequel est basée la construction peut ne pas exister, auquel cas on ne peut pas l'utiliser, alors que tous les objets liés à un « si-géométrique » existent tout le temps en mémoire, qu'ils aient ou non une représentation physique.

Pour l'implémenter, on pourra mémoriser la méthode de construction, comme on mémorise le contenu d'une macro, en utilisant un mécanisme de désignation de l'objet sur lequel cette méthode s'applique, mais le processus d'extraction sera différent.

Dans la vue simultanée du programme, les constructions liées à une modification de la profondeur d'appel devront aussi apparaître et disparaître. Par contre, les constructions basées sur des sous-objets des fractales pourront être décrites en suivant un formalisme similaire à celui des macros, par exemple à la fin du programme.

Cela conduit à trois objectifs importants :

- introduire un nouveau paradigme de programmation avec l'introduction d'une instruction de contrôle conditionnelle,
- gérer la création dynamique d'objets dans un environnement de programmation où tout le « code » est expansé,
- ne pas perdre les constructions basées sur des objets construits par une macro récursive.

b. Fonctionnement

Le premier objectif conduit à déterminer un formalisme et un mode d'interaction avec l'utilisateur permettant de spécifier une condition qui soit une véritable instruction conditionnelle au sens des langages de programmation usuels, et pas un simple « si-géométrique ». Cette structure de contrôle étant la première dans le logiciel, elle nécessitera une véritable innovation, et il sera délicat de trouver un schéma cohérent avec les autres outils. Il n'est même pas évident qu'il conviendra de lui donner le statut d'un outil.

De plus, la mise au point conduira à des problèmes difficiles. D'ailleurs, la conditionnelle n'est gérée dans aucun logiciel de géométrie dynamique.

Sketchpad gère les appels en boucles à profondeur fixée. Nous avons vu au §II-3-CB-3°-a, que l'utilisateur peut enregistrer la suite de ses constructions et les voir par l'intermédiaire d'un texte. Une fenêtre spécifique est fournie pour visualiser les scripts et pour donner accès à quelques outils dédiés aux scripts. Parmi ces outils se trouve un outil permettant de spécifier les appels récursifs. Lorsque l'enregistrement d'un script est terminé, l'utilisateur peut activer cet outil, qui lui permet de sélectionner dans la figure les objets sur lesquels le script devra s'appliquer à nouveau à l'issue de l'exécution des constructions mémorisées.

La deuxième illustration du §II-3-CB-3°-a montré ce que devient le script après un tel usage. Lorsque l'utilisateur appelle un script autoréférent, une boîte de dialogue l'oblige à choisir une profondeur d'appel en le prévenant de l'encombrement mémoire consécutif à son choix. Sketchpad ne gère donc pas de récursivité dynamique, et n'a pas d'instruction de contrôle. Nous ne pouvons nous en inspirer ni pour définir le concept ni pour l'implémenter.

Dans Cabri, un scénario d'utilisation pourrait être le suivant :

Au cours de la validation d'une macro, l'utilisateur choisira de définir une macro récursive. La figure d'école redeviendra active, avec l'outil de sélection comme outil courant. L'utilisateur sélectionnera alors une liste d'objets de types compatibles avec ceux des paramètres de la macro, même si ce ne sont pas des objets dont les constructions seront extraites par le processus de validation des macros.

Ces objets porteront les appels récursifs. Si l'utilisateur veut une condition d'arrêt liée à une dimension, il choisira un des outils d'un nouveau menu qui regroupera des outils de comparaison. Il devra alors sélectionner des objets de types convenables.

Par exemple, pour l'outil « Plus grand que », il faudra deux nombres. Le résultat de la condition d'arrêt sera une propriété, qui apparaîtra dans la boîte de dialogue, par exemple "N1 est plus grand que N2", ou "N1 n'est pas plus grand que N2".

Par cohérence avec tous les autres outils, cet outil construira un objet, donc un objet de type "propriété de comparaison".

Cette liberté de choix de paramètres pour les outils de comparaison permet d'imaginer une étape de validation qui compléterait l'étape de validation des macros, au cours de laquelle on rechercherait (de manière nécessairement incomplète à cause des théorèmes d'indécidabilité bien connus) à déterminer si le choix de l'utilisateur conduit à coup sûr à une boucle infinie.

Il faut aussi gérer le nombre d'objets générés par l'appel, d'une façon cohérente : lorsque l'utilisateur anime la figure, le logiciel doit vérifier que le nombre d'objets à générer est raisonnable, de façon à arrêter les appels récursifs dès qu'il n'y a plus assez de place en mémoire pour tous les objets de la profondeur demandée. On peut aussi ajouter un nombre maximum d'objets constructibles dans les préférences.

Dans ce scénario d'interaction, la vue textuelle n'est pas indispensable. Par contre, le langage de géométrie formelle défini pour cette vue textuelle est utile pour réfléchir sur l'implémentation à mettre en œuvre.

c. Implémentation

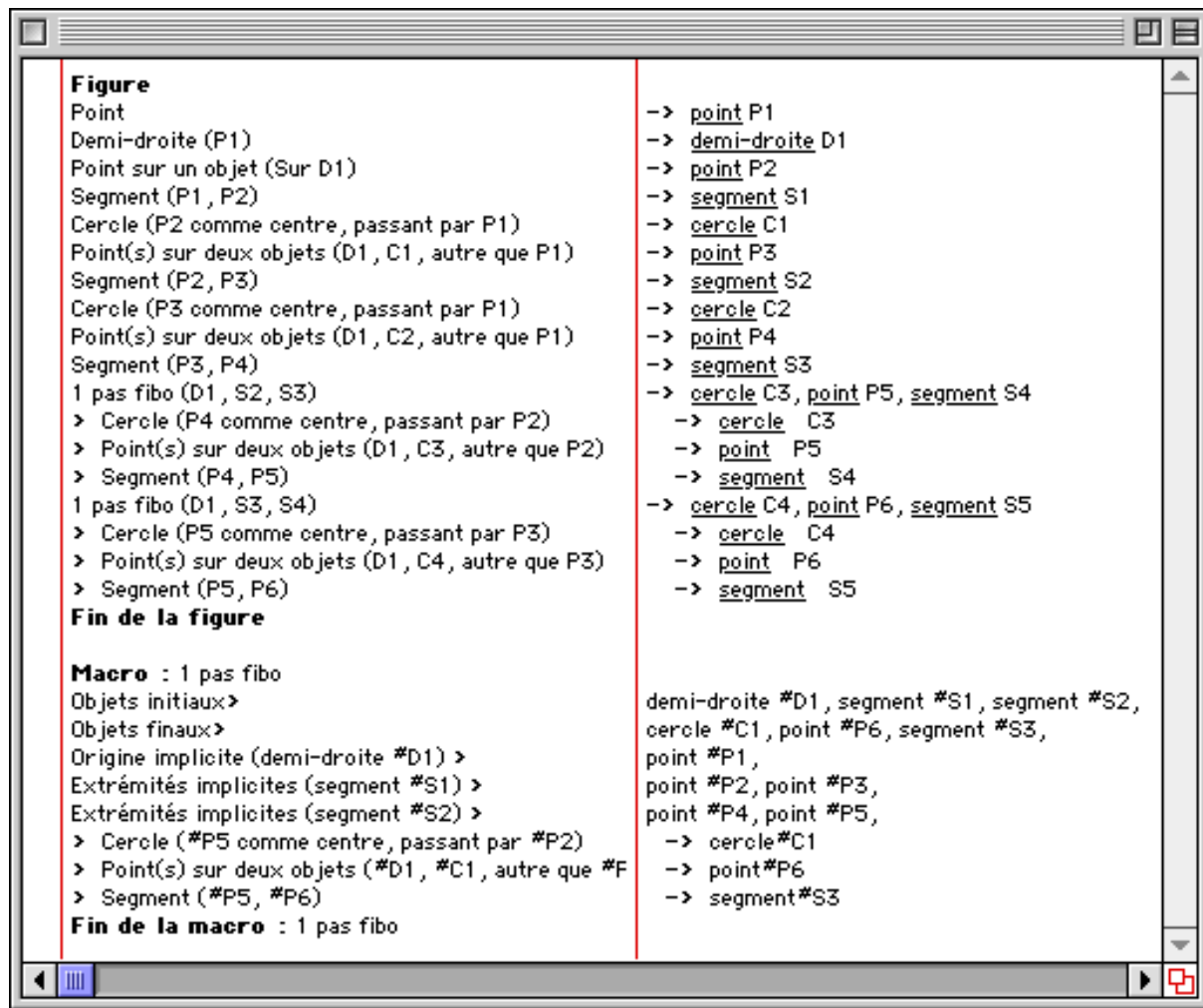
Du point de vue de l'implémentation, deux problèmes difficiles doivent être résolus :

- déterminer les structures de données capables de supporter les fractales dynamiques,
- déterminer un processus pour désigner les sous-objets des fractales dynamiques.

Dans la recherche d'une solution à ces problèmes, le texte du programme est utile, avec sa grammaire formelle, car il permet de concrétiser les solutions envisagées.

Par exemple, pour obtenir la dernière figure ci-dessus, qui est une simulation géométrique de la suite de Fibonacci jusqu'au troisième appel de la formule récursive, nous avons défini une macro-construction qui regroupe les constructions d'un pas d'itération. Cette macro pourrait constituer la base de la fractale.

Nous obtenons le texte suivant :



Pour supporter les macros récursives il faudrait ajouter au langage la déclaration des objets sur lesquels l'appel récursif est basé. La chaîne de caractères associée pourrait être, en français : « Appels récursifs sur », et le texte de la macro deviendrait :

<pre>Macro : 1 pas fibo Objets initiaux > Objets finaux > Origine implicite (demi-droite #D1) > Extrémités implicites (segment #S1) > Extrémités implicites (segment #S2) > Appels récursifs sur > > Cercle (#P5 comme centre, passant par #P2) > Point(s) sur deux objets (#D1, #C1, autre que #P2) > Segment (#P5, #P6) Fin de la macro : 1 pas fibo</pre>	<pre>demi-droite #D1, segment #S1, segment #S2, cercle #C1, point #P6, segment #S3, point #P1, point #P2, point #P3, point #P4, point #P5, segment #S2, segment #S3 -> cercle #C1 -> point #P6 -> segment #S3</pre>
--	--

La figure suivante correspond à des appels récursifs de cette macro, jusqu'à la profondeur 3. Si les macros sont pliées, le texte deviendrait :

<pre>Figure Point Demi-droite (P1) Point sur un objet (Sur D1) Segment (P1, P2) Cercle (P2 comme centre, passant par P1) Point(s) sur deux objets (D1, C1, autre que P1) Segment (P2, P3) 1 pas Fibo (D1, S1, S2) Fin de la figure</pre>	<pre>-> point P1 -> demi-droite D1 -> point P2 -> segment S1 -> cercle C1 -> point P3 -> segment S2 -> cercle C2, point P4, segment S3, cercle C3, point P5, segment S4 cercle C4, point P6, segment S5</pre>
---	---

En dépliant l'appel de la macro, on obtiendrait le texte suivant :

<pre>Figure Point Demi-droite (P1) Point sur un objet (Sur D1) Segment (P1, P2) Cercle (P2 comme centre, passant par P1) Point(s) sur deux objets (D1, C1, autre que P1) Segment (P2, P3) 1 pas Fibo (D1, S1, S2) > Cercle (P3 comme centre, passant par P1) > Point(s) sur deux objets (D1, C2, autre que P1) > Segment (P3, P4) > 1 pas Fibo (D1, S2, S3) Fin de la figure</pre>	<pre>-> point P1 -> demi-droite D1 -> point P2 -> segment S1 -> cercl C1 -> point P3 -> segment S2 -> cercle C2, point P4, segment S3, -> cercle C2, -> point P4, -> segment S3, -> cercle C3, point P5, segment S4 cercle C4, point P6, segment S5</pre>
--	---

et en dépliant une nouvelle fois, on obtiendrait

<pre>Figure Point Demi-droite (P1) Point sur un objet (Sur D1) Segment (P1, P2) Cercle (P2 comme centre, passant par P1) Point(s) sur deux objets (D1, C1, autre que P1) Segment (P2, P3) 1 pas Fibo (D1, S1, S2) > Cercle (P3 comme centre, passant par P1) > Point(s) sur deux objets (D1, C2, autre que P1) > Segment (P3, P4) > 1 pas Fibo (D1, S2, S3) > Cercle (P4 comme centre, passant par 2) > Point(s) sur deux objets (D1, C3, autre que P2) > Segment (P4, P5) > 1 pas Fibo (D1, S3, S4) Fin de la figure</pre>	<pre>-> point P1 -> demi-droite D1 -> point P2 -> segment S1 -> cercl C1 -> point P3 -> segment S2 -> cercle C2, point P4, segment S3, -> cercle C2, -> point P4, -> segment S3, -> cercle C3, point P5, segment S4 -> cercle C3, -> point P5, -> segment S4 -> cercle C4, point P6, segment S5</pre>
---	---

Une instruction conditionnelle utilisée pour l'arrêt doit être supportée par une propriété. La condition pourrait être exprimée par une déclaration paramétrée par une propriété comparative sans partie droite. La chaîne de caractères associée pourrait être : « Tant que » . Le résultat serait le texte suivant :

<pre>Figure Point Demi-droite (P1) Point sur un objet (Sur D1) Segment (P1, P2) Cercle (P2 comme centre, passant par P1) Point(s) sur deux objets (D1, C1, autre que P1) Segment (P2, P3) Distance & longueur(S2) Texte (inclure N1) 1 pas Fibo (D1, S1, S2) Fin de la figure Macro : 1 pas fibo Objets initiaux > Objets finaux > Origine implicite (demi-droite #D1) > Extrémités implicites (segment #S1) > Extrémités implicites (segment #S2) > Appels récursifs sur > Tant que (#P1 est vraie) > Cercle (#P5 comme centre, passant par #P2) > Point(s) sur deux objets (#D1, #C1, autre que #P2) > Segment (#P5, #P6) > Distance & longueur(#S2) > Texte (inclure #N2) > Calculatrice (#N2/ 10) > Texte (inclure #N3) > Comparaison (#N3, est plus petit que #N1) Fin de la macro : 1 pas fibo</pre>	<pre>-> point P1 -> demi-droite D1 -> point P2 -> segment S1 -> cercl C1 -> point P3 -> segment S2 -> nombre N1 -> texte T1 -> cercle C2, point P4, segment S3, cercle C3, point P5, segment S4 cercle C4, point P6, segment S5 demi-droite #D1, segment #S1, segment #S2, nombre #N1 cercle #C1, point #P6, segment #S3, point #P1 point #P2, point #P3, point #P4, point #P5, segment #S2, segment #S3 -> cercle #C1 -> point #P6 -> segment #S3 -> nombre #N2 -> texte #T1 -> nombre #N3 -> texte #T2 -> propriété #P1</pre>
--	---

2°) « Raccourcis » et polymorphisme

Nous avons parlé plus haut (§II-2) des "raccourcis" (manipulateurs implicites). Notre travail permet d'en envisager quelques extensions.

a. Extension à d'autres outils comme Droite Parallèle et aux autres outils qui construisent des droites

Une question apparaît pour l'arrêt des propositions des raccourcis. Par exemple, si la construction d'une droite parallèle est proposée à la volée, peut-elle, elle aussi, être préemptée par l'outil Point Sur ?

Dans ce cas, le souci de cohérence oblige à supporter le même schéma dans l'autre sens : l'outil Point Sur peut s'appliquer à une droite constructible à la volée comme une droite parallèle à toute droite survolée, mais l'outil Droite Parallèle peut s'appliquer à un point ("passant par ce point") constructible à la volée par l'outil Point Sur.

On arrive donc à des appels récursifs du schéma d'exécution dans lesquels l'utilisateur a beaucoup de risque de se perdre, ainsi que le logiciel.

Dans l'exemple pris, les deux outils diffèrent. Mais le phénomène est possible avec le seul outil Droite Parallèle : pendant la construction d'une droite parallèle à une autre droite, et plus précisément pendant le choix de l'argument de type « droite » de cet outil, le curseur passe au dessus d'une droite. L'outil Droite Parallèle étant préemptif, le logiciel détecte qu'il peut proposer l'utilisation de cet outil pour construire une autre droite parallèle, qui sera le paramètre de la construction explicite choisie initialement par l'utilisateur.

Ainsi, le logiciel pourra mettre en route temporairement une nouvelle fois cet outil pour construire une droite parallèle à celle survolée, et ce sera sur le résultat de cette construction "à la volée" que sera basée la construction de la droite parallèle initialement cherchée par l'utilisateur. Sur ce deuxième appel, le schéma devra encore pouvoir se reproduire, d'où la récursivité de ces schémas et le problème de l'arrêt.

b. Extension à d'autres outils que les outils de construction

Si l'outil courant est le pointeur, les objets sélectionnés peuvent être déformés selon les mouvements de la souris déplacée avec bouton enfoncé.

On peut imaginer que l'outil de redéfinition puisse aussi devenir préemptif. En effet, supposons que l'objet "maître" de la déformation, c'est-à-dire celui qui est tiré par la souris, soit un point. Alors la composante Redéfinir un Point comme un Point Sur de l'outil de redéfinition peut proposer de redéfinir ce point à la volée.

La version de Cabri qui supporte la redéfinition de tous les objets peut permettre de redéfinir une droite en cours de rotation. Selon que la droite a été définie par une direction ou par un deuxième point, elle peut être redéfinie comme une droite parallèle ou perpendiculaire à une droite survolée, ou comme passant par un point survolé, ou comme passant par un point à construire sur l'objet survolé, par exemple par le point situé de la droite survolée situé exactement sous le curseur.

c. Extension à plusieurs outils ayant mêmes types d'objets initiaux et terminaux

L'extension de ce concept à plusieurs outils ayant mêmes types d'objets initiaux et terminaux, comme l'ensemble des transformations géométriques, conduit à des ambiguïtés.

Prenons l'exemple de l'outil Symétrie axiale (d'un point par rapport à une droite). Cet outil est basé sur la connaissance de deux objets : une droite et un point. Il construit un objet final de type Point. Il pourrait donc être proposé dès qu'un point est requis par l'outil courant choisi par l'utilisateur, « Cercle », par exemple.

Chaque fois que le curseur survolera un point, au lieu que le curseur propose "ce point comme centre", il proposera "le symétrique — miroir — de ce point (comme centre)". Nous mettons "comme centre" entre parenthèses parce que, dans les propositions actives dans la version actuelle, le correspondant n'est pas affiché. Si le curseur survole une droite, rien ne se passera car l'outil Cercle ne peut être basé sur une droite.

De même, pour l'outil préemptif Symétrie Centrale, il proposera "le symétrique — symétrie centrale — de ce point (comme centre)". On aboutit donc à un problème de gestion de l'ambiguïté ou plutôt de choix. De plus, les "phrases" précises sont très longues (perte en concision).

d. Extension aux macros

Ce concept peut encore être étendu aux macros, qui pourraient être préemptives ou non, au gré de l'utilisateur. Le logiciel proposerait donc d'appliquer une macro préemptive chaque fois qu'il en trouverait l'opportunité.

Dans le cas d'un outil de base préemptif, le logiciel actuel propose l'exécution d'un programme constitué d'une seule instruction. Dans le cas d'une macro, le programme serait constitué de plusieurs instructions, et créerait ainsi non seulement l'objet désiré, mais aussi toute une liste d'autres objets.

En conclusion, il faut :

- limiter à 1 la profondeur d'application récursive de ce schéma (minimiser la difficulté des opérations mentales),
- étendre l'usage des raccourcis aux redéfinitions,
- réfléchir au problème des outils qui construisent le même type d'objets et sont basés sur un objet de même type (ambiguïtés) afin de déterminer une stratégie. Ce concept est-il pertinent pour tous les outils ? Doit-on l'introduire par famille d'outils, pour préserver la consistance du logiciel, etc. ?
- associer à chaque outil et à chaque objet initial de chaque outil une chaîne de caractères efficace (discriminante, concise, et éventuellement intégrable dans une chaîne englobante),
- ne pas appliquer de raccourci pour les objets intermédiaires des macros.

Ce concept rapprocherait le fonctionnement de Cabri d'autres logiciels comme Cocoa. Ce logiciel vérifie à chaque itération d'une boucle infinie s'il existe une règle applicable parmi les règles d'une liste définie par l'utilisateur, et applique la première qu'il trouve. Le problème de l'ambiguïté est ainsi résolu.

Mais, dans Cabri, il n'est pas pertinent de gérer l'équivalent de la liste de ces règles, alors qu'elle n'aura pas de sens pour l'utilisateur en dehors de cet effet finement perceptible.

Comment ce schéma d'utilisation pourrait-il être mis en œuvre ? Pour anticiper sur les désirs de l'utilisateur, la proposition de l'application d'un outil préemptif doit être introduite dans la gestion du curseur : c'est au moment du choix du curseur à afficher en fonction de la position du curseur et de l'outil courant, qu'on peut estimer les possibilités de préemption.

3°) Intégration d'une vue liée à un éditeur de graphes

a. Motivations et historique

Dans la première partie, nous avons présenté une étude sur les supports utilisés habituellement par les programmeurs pour concevoir et représenter leurs programmes. Les interfaces graphiques permettent aujourd'hui de proposer des environnements de programmation basés sur un langage informatique à deux dimensions. Ces approches sont particulièrement étudiées dans le cadre des environnements orientés vers les non-programmeurs.

Notre choix d'une forme textuelle pour la vue implémentée dans le prototype devait pouvoir être remis en cause, et la vue textuelle pourrait être complétée par une autre vue non principalement textuelle. L'environnement résultant serait alors doté de trois vues du programme de construction :

- une vue géométrique, qui représenterait le résultat de l'exécution du programme,
- une vue textuelle, qui donnerait accès à la structure du programme sous la forme classique d'un langage de programmation,
- une vue graphique, qui permettrait de visualiser et d'éditer la structure logique du programme représentant les relations de dépendance entre les objets sous la forme d'un graphe.

L'intégration de la vue graphique nécessite des compétences en édition de graphes, domaine qui se trouve justement abordé depuis un certain temps par l'équipe de laquelle Cabri-géomètre est issue. L'histoire du logiciel Cabri-géomètre a d'ailleurs commencé avec un éditeur de graphes, et ce sont les concepts mis en œuvre pour ce premier éditeur qui ont permis de concevoir le logiciel Cabri-géomètre. Depuis, une nouvelle version de Cabri-graphe a été développée, et les recherches nécessaires à son amélioration ont fait l'objet d'une thèse [Carbonneaux 98].

Cabri-graphe permet de spécifier toutes sortes de graphes par manipulation directe. Les types de graphes supportés vont des graphes simples aux graphes multiples orientés, et le logiciel donne accès à des outils capables de faire des opérations sur les graphes (produit fibré...). Une grande variété de représentations des arêtes est proposée, allant des simples segments aux courbes de Bézier, en passant par les lignes brisées. Le logiciel est capable d'ajuster la présentation pour minimiser le nombre de croisements. Il permet de plus de contracter des sous-graphes d'un graphe, sur plusieurs niveaux, et de les décontracter en retrouvant la présentation initiale.

b. Quel graphe choisir ?

Dans ce mémoire de thèse, nous avons utilisé deux formes de graphe pour représenter les figures de Cabri. La première forme (§I-1-B-1°-b) met l'accent sur les relations de dépendance qui définissent les contraintes entre les objets. La deuxième forme (§II-1-A-1-a) donne en plus accès à l'ordre dans lequel les objets géométriques sont mémorisés. Cet ordre est perceptible lors du remplissage d'objets géométriques dont la représentation graphique constitue une courbe fermée. Ces deux formes peuvent être utiles, selon la tâche à effectuer par l'utilisateur.

Dans les deux formes, les nœuds représentent les objets, et les arêtes les contraintes qui relient les objets entre eux. Le repliement des macros correspond à une contraction d'un sous-graphe.

La redéfinition des contraintes peut prendre une forme proche de celle qui est prévue dans la vue textuelle, par déplacement d'un sommet et identification avec n'importe quel autre sommet qui représente un objet du même type.

La vue graphique peut être complétée par le même artefact pour donner accès aux attributs graphiques, algébriques et manipulatoires des objets.

Nous avons présenté une troisième forme qui schématise l'expression actionnelle des algorithmes (§1-2-A-2°), mais le passage à une telle forme demande plus de travail, car elle est plus éloignée de la structure logique du programme.

c. Quelle adaptation ?

Pour intégrer une vue graphique, il faut insérer dans le code du programme de Cabri les traitements dédiés aux manipulations de graphes.

Pour une forme de graphe proche de la structure logique du programme, ces traitements doivent être ajoutés aux mêmes endroits que ceux qui produisent la vue textuelle.

Ainsi, la production du texte du programme qui décrit chaque construction doit être complétée par la construction du sous-graphe correspondant. De même, les traitements mis en œuvre pour supporter les manipulations de la structure logique du programme, doivent être complétés des manipulations du graphe correspondantes.

Il faut donc ajouter les manipulations structurelles spécifiques au graphe à celles du programme textuel, les mettre en double.

d. Quels nouveaux problèmes ?

Cette intégration conduit cependant à de nouveaux problèmes. En effet, pour intégrer cette vue graphique, il faut ajouter les structures de données utilisées dans Cabri-graphe aux structures de données de Cabri-géomètre.

On se trouve confronté au même problème que pour l'intégration de la vue textuelle, problème qui nous a empêchée de reprendre directement un éditeur de documents structurés : les fonctions d'édition et de manipulation du graphe sont supportées par le noyau fonctionnel de l'éditeur, alors que le noyau fonctionnel qui doit garder la maîtrise de la structure logique de la figure géométrique est celui de Cabri-géomètre.

Par exemple, le problème de la mémorisation de la configuration choisie par l'utilisateur, et la robustesse de la vue graphique par rapport aux manipulations du graphe est un problème du même ordre que le problème que nous avons rencontré pour prendre en compte les notations secondaires dans le texte. Ce problème est même encore plus difficile dans le cas des graphes, du fait de l'ajout d'une dimension. Or, cet aspect est entièrement géré par le noyau fonctionnel de l'éditeur de graphes. Son intégration est donc plus compliquée que la simple adaptation d'un outil.

Conclusion de la partie II

Dans cette partie, nous avons justifié les choix effectués pour la réalisation du prototype, décrit le résultat de cette réalisation, et présenté ses perspectives d'évolution et d'application.

Nous devons apporter des fonctionnalités d'environnement de programmation à Cabri-géomètre, qui est un logiciel orienté vers le tout-public et permet de définir des constructions de figures géométriques par manipulation directe.

À partir de la recherche théorique des qualités et critères pertinents à mettre en œuvre, présentée dans la première partie de ce mémoire, nous avons défini la forme et le comportement attendus, qui constituent les spécifications externes du logiciel étendu. Un langage de géométrie formelle a ainsi été défini, et nous avons décrit les manipulations à mettre en œuvre.

Les contraintes liées au contexte de la réalisation nous ont ensuite conduite aux différents choix et aux spécifications internes.

Les fonctionnalités à apporter devant être intégrées dans un logiciel existant, il en est ressorti quelques principes de conception, comme le respect du mode de manipulation directe des objets. Nous avons aussi dû tenir compte des limitations des coûts espace et temps liés à la réalité opérationnelle.

L'ensemble des contraintes et des choix effectués nous ont posé des problèmes pour lesquels nous avons dû rechercher des solutions spécifiques. En particulier, nous avons mis en évidence le parallélisme entre les besoins découlant de l'objectif consistant à aboutir à un texte de programme manipulable structurellement, et les principes utilisés pour définir les éditeurs de documents structurés.

Nous avons alors décrit les solutions pratiques que nous avons adoptées, et le résultat obtenu. Cette description peut-être vue comme l'ébauche d'un manuel d'utilisation de la vue textuelle. Nous avons aussi présenté les limitations de notre prototype et déterminé leurs causes.

Enfin, nous avons présenté les prolongements de cette réalisation, qui permettraient de passer du prototype à une réalisation plus complète, répondant mieux aux désirs exprimés au départ.

Sur deux exemples, nous avons montré que le langage défini pour décrire les manipulations de la structure logique des programmes de construction peut constituer un bon support de réflexion pour apporter de nouvelles fonctionnalités au logiciel Cabri-géomètre.

Enfin, nous avons décrit ce qu'il faudrait encore faire pour associer une autre forme de représentation à la structure logique des programmes de construction, en construisant une vue similaire à la vue textuelle par ses caractères dynamique et synchronisé. Cette vue serait basée sur l'intégration des fonctionnalités de l'éditeur de graphes Cabri-graphe, qui seraient utilisées pour les manipulations de la structure logique du programme.

Les difficultés principales ont été :

- du point de vue conceptuel, la définition d'un langage de programmation textuel, caractérisant non pas les manipulations de l'interface, mais le résultat d'un calcul effectué par le noyau fonctionnel du logiciel sur les données mémorisées, définies et modifiées par les interactions de l'utilisateur ;

- du point de vue technique, la mise en place de nouvelles structures de données, d'une part pour supporter l'ubiquité des objets nécessaire pour aider l'utilisateur à assimiler les contraintes qui les relient, et pour mettre en relation leurs identificateurs textuels avec leurs représentations graphiques, et d'autre part pour permettre à l'utilisateur d'introduire une forme de notations secondaires (des sauts de ligne « à volonté ») compatible avec la navigation dans la hiérarchie de la structure du programme (par dépliement et repliement des macros).

Ce travail nous a conduite à étudier assez en détail les IHM et leur implémentation, et à assimiler des concepts issus de l'édition des documents structurés.

Il nous a aussi menée à réfléchir sur le concept de type, et à proposer d'introduire des types construits, tout en préservant l'homogénéité des schémas d'utilisation de ces types, le polymorphisme des outils permettant de considérer des sous-objets ou des sur-objets.

Enfin, l'introduction de macros récursives a fait l'objet d'une étude assez détaillée, car, si le principe général est clair, sa mise en œuvre mène à des problèmes nombreux et passablement compliqués, si on veut arriver à une manipulation directe et naturelle de la vue textuelle.

En fait, tous ces sujets participent d'un seul et même thème général, à savoir la recherche de solutions théoriques et pratiques permettant d'étendre les moyens qu'a l'utilisateur de spécifier ses désirs, et de le libérer de la nécessité, purement informatique, de choisir un outil adapté au type d'objet qu'il manipule pour la tâche qu'il désire effectuer.

Conclusion générale et perspectives

L'intégration d'une vue textuelle de programme dans une interface qui permet de commander des constructions géométriques par manipulation directe est une approche opposée à l'approche classique, mais qui présente de nombreux aspects intéressants.

Nous en avons montré plusieurs :

- Une programmation basée sur une extraction optimisée des constructions (ou commandes) utiles entraîne pour l'utilisateur des problèmes de mise au point très difficiles, dus au manque d'accès à la structure logique du programme obtenu.
- La production et la visualisation a posteriori du texte du programme comme une vue du résultat de la construction conduit à définir un langage complexe, devant rendre compte de toutes les spécificités des manipulations permises sur les données.

Dans les perspectives, on peut imaginer de poursuivre cette recherche suivant deux axes :

- un axe proche de l'application Cabri : il s'agirait de mettre en évidence la nécessité d'un langage décrivant les manipulations d'une interface, et d'envisager une étape de « lissage » dont on peut imaginer qu'elle contribuerait à rendre l'interface plus cohérente et plus robuste.
- un axe concernant toutes les applications : en "induisant" le langage textuel à partir des manipulations de l'interface d'un logiciel supportant la manipulation directe jusqu'à un niveau très pointu, avec les traitements résultant de chaque spécificité de manipulation des données, on pourrait généraliser les variantes langagières nécessaires pour supporter ces spécificités.

On pourrait aussi imaginer de les prévoir dans l'approche duale, c'est-à-dire, réciproquement, de partir d'une spécification textuelle de la dimension interaction à implémenter dans une interface, et de construire ensuite les fonctionnalités de manipulation directe.

Cependant, cela ne serait possible que parce qu'il s'agirait d'un "retour de balancier", la première étape ayant été absolument nécessaire pour découvrir les fonctionnalités indispensables. Mais ce retour ne serait pas qu'une pure réingénierie : il permettrait, partant d'une conception plus orthogonale des différents concepts, de croiser toutes les possibilités et par suite d'offrir de nouvelles fonctionnalités, impossibles ou extrêmement lourdes à mettre en œuvre actuellement à cause du "poids de l'histoire" de cette première étape.

-o-o-o-o-o-o-o-o-o-

Bibliographie

[Accot 98]

Accot Johnny, Chatty Stéphane, Jestin Yannick, Sire Stéphane (1998), "Conception des interfaces : et si nous analysions enfin la tâche du programmeur ?", IHM'98, Nantes, septembre 1998.

[Addis & al. 98]

Tom Addis, Alan Blackwell, Paul Brna, David Gooding, Jim Kyle, Patrick Olivier, Mike Scaife, Keith Stenning, "Thinking with Diagrams", discussion électronique, 1998.
<http://www.mrc-cbu.cam.ac.uk/projects/twd/Workshop.html> .

[Arsac 89]

Arsac G. (1989), "La construction du concept de figure chez les élèves de 12 ans", Proceedings of the thirteenth conference of International Group for Psychology of Mathematics Education, Éd. GR didactique et acquisitions des connaissances scientifiques, 46 rue Saint Jacques, 75005 Paris, pp. 85-92.

[Augier 97]

Marc Augier (1997), "Les logiciels", Que sais-je ? Presses Universitaires de France, août 1997.

[Avram 84]

Doris Avram, Tristan Savatier, Michèle Weidenfeld (1984), "LOGO - Manuel de référence", S.O.L.I., Systèmes d'Ordinateur Logo International, 33 rue de Poissy, 75005 Paris, 1984. ISBN 2-7124-0533-1

[Balbo & al 98]

Balbo S., Ozkan N., Lu S., Paris C. (1998), "Task models in industrial context : how do they fit ?", 10èmes journées sur l'ingénierie des interfaces homme-machine (IHM98), Nantes, 1998.

[Barwise & Etchemendy 91]

Jon Barwise, John Etchemendy (1991), "Visual information and Valid Reasoning", Visualisation in Teaching and Learning Mathematics, ed. W.Zimmerman, S.Cunningham, 1991.

[Bellemain 92]

Bellemain F. (1992) "Conception, réalisation et utilisation d'un logiciel d'aide à l'enseignement de la géométrie : Cabri-géomètre", Thèse, Université Joseph Fourier, Grenoble.

[Bellemain & Capponi 92]

Bellemain F., Capponi B. (1992) "Spécificité de l'organisation d'une séquence d'enseignement lors de l'utilisation de l'ordinateur". Educational Studies in Mathematics 23 (1) 59-97.

[Bellemain 96]

Bellemain F. (1996) "Micromonde, manipulation directe et enseignement de la géométrie : un éclairage pour comprendre l'évolution de Cabri-géomètre vers Cabri-géomètre II", Université d'été "Cabri-géomètre" de l'ordinateur à la calculatrice, De nouveaux outils pour l'enseignement de la géométrie, ed. IREM de Grenoble, Grenoble, p.171-202, 1996.

[Blackwell 96]

Alan F. Blackwell (1996), "Metacognitive Theories of Visual Programming: What do we think we are doing?", In Proceedings IEEE Symposium on Visual Languages, pp. 240-246, September 1996.
<http://www.mrc-cbu.cam.ac.uk/projects/twd/mypapers/VL96.html>

[Blackwell & al. 98]

Alan Blackwell, Kirsten Whitley, Judith Good, Marian Petre, "Programming in Picture, Pictures of Programs", Thinking with Diagrams, discussion électronique, 1998
<http://www.mrc-cbu.cam.ac.uk/projects/twd/discussion-papers/programming.html>.

[Blackwell & al. 98-b]

Bibliographie

- Alan Blackwell, Kirsten Whitley, Judith Good, Marian Petre (1998), "Discussion Paper: Programming", in Proceedings of Thinking with Diagrams 98: Is there a science of diagrams?.
<http://www.mrc-cbu.cam.ac.uk/projects/twd/discussion-papers/programming.html>
- [Blackwell 98]
Blackwell, A.F. and Engelhardt, Y. (1998). "A taxonomy of diagram taxonomies", in Proceedings of Thinking with Diagrams 98: Is there a science of diagrams?, pp. 60-70.
<http://www.mrc-cbu.cam.ac.uk/projects/twd/mypapers/TwD98.html>
- [Botshernitsan & Downes 97]
Marat Boshernitsan, Michael Downes (1997), "Visual Programming Languages: a Survey", Rapport de Recherche, CS 263 Final Project, Université de Californie, Berkeley, décembre 1997.
<http://www.cs.berkeley.edu/~maratb/cs263/paper/paper.html>
- [Bouhineau 97]
Denis Bouhineau (1997), "Constructions automatiques de figures géométriques et Programmation Logique avec Contraintes", Thèse, Université Joseph Fourier, Grenoble, juin 1997.
- [Bouhineau 95]
Denis Bouhineau (1995), "Vers une approche déclarative pour les logiciels de dessins géométriques", Environnements Interactifs d'Apprentissage avec Ordinateur, Eyrolles, Paris, 1995.
- [Brihault 93]
Yannick Brihault, Marcel Duboué (1993), "Aspects cognitifs d'une interface pour l'apprentissage de la programmation", Environnement Interactifs d'Apprentissage avec Ordinateur, Eyrolles, Paris, 1993.
- [Burnett & Baker 94]
Burnett Margaret M. and Marla J. Baker, A Classification System for Visual Programming Languages, Journal of Visual Languages and Computing, 287-300, September 1994.
- [Burnett 94]
Burnett Margaret, Richard Hossli, Timothy Pulliam, Brian VanVoorst, and Xiaoyang Yang, "Toward Visual Programming Languages for Steering in Scientific Visualization: a Taxonomy", IEEE Computational Science & Engineering, 44-62, Winter 1994.
- [Carbonneaux 98]
Yves Carbonneaux, "Conception et réalisation d'un environnement informatique sur la manipulation directe d'objets mathématiques, l'exemple de Cabri-graphs", Thèse, Université Joseph Fourier, Grenoble, janvier 1998 .
- [Cardelli 88]
Cardelli Luca (1988), "Building User Interfaces by Direct Manipulation", SRC Research Report 22, Digital Equipment Corporation, October 2, 1987, appears in: ACM Siggraph Symposium on User Interface Software, pp. 152-166, ACM Press, 1988.
- [Capponi & al. 98]
Bernard Capponi, Bernard Geneves, Veronica Hoyos (1998), "Simulation of Drawing Machines on Cabri-II and its dual Algebraic Symbolisation : Descartes' Machine & Algebraic Inequality", CERME 1, Osnabrueck, Germany, août 1998.
- [Capponi & Laborde 94]
Bernard Capponi, Colette Laborde (1994), "Cabri-classe. Collège", Ed. Archimède, Argenteuil, ISBN 2-9506960-5-8, 1994.
- [Chang 86]
S.K. Chang, "Languages" in Visual Languages, Plenum Press, New-York, 1986, pp.1-7.
- [Charrière 96]
Pierre-Marie Charrière (1996), "Apprivoiser la géométrie avec CABRI-GÉOMÈTRE", Ed. Monographie du CIP, Genève, 1996.
- [Chatty 93]
<http://www.cenatls.cena.dgac.fr/divisions/PII/toccatata/toccatata.html>
- [Coutaz 90]

- Coutaz Joëlle (1990), Interfaces homme-ordinateur, Conception et réalisation, Dunod informatique, Bordas, Paris 1990.
- [Coutindo 99]
Coutindo Cilheda, (1999), « Cabri et la simulation d'expériences aléatoires », actes de CabriWorld, PUC São Paulo, Brésil, 1999.
- [Cuppens 95]
Cuppens Roger (1995), « Faire de la géométrie en jouant avec Cabri-géomètre », 2 tomes, Ed. Association des Professeurs de Mathématiques de l'Enseignement Public, 1995.
- [Cypher & al. 93]
Cypher A., and al., "Watch What I Do: Programming by Demonstration", publish. by Cypher A., in MIT Press, 1993.
- [Cypher 93]
Cypher A., "Eager: Programming Repetitive Tasks by Demonstration", in [Cypher & al. 93], pp. 205-218, 1993.
- [Cypher & Smith 95]
Cypher A., and Smith D.C. (1995) "KIDSIM: End User Programming of Simulation", in Proceedings of CHI '95 (Denver CO, May 1995), ACM Press.
- [Desmoulins 95]
Cyrille Desmoulins (1995), "La détection de solutions particulières dans TALC : un approche logique basée sur des extensions de l'énoncé du problème", Environnements Interactifs d'Apprentissage avec Ordinateur, Eyrolles, Paris, 1995.
- [Desmoulins 94]
Cyrille Desmoulins (1994), "Étude et réalisation d'un système tuteur pour la construction de figures géométriques", Thèse, Université Joseph Fourier, Grenoble, février 1994.
- [Erwig 95]
Martin Erwig & Bernd Meyer (1995), "Heterogeneous Visual Languages - Integrating Visual and Textual Programming", in 11th IEEE Symp. on Visual Languages, Darmstadt, 1995, pp. 318-325.
- [Erwig 97]
Martin Erwig (1997), "Abstract Visual Syntax ", in 2th IEEE Int. Workshop on Theory of Visual Languages, 15-25, 1997.
- [Girard 92]
Patrick Girard (1992), "Environnement de programmation pour non-programmeurs et paramétrage en conception assistée par ordinateur : le système Like" ,Thèse, Université de Poitier, France, juillet 1992.
- [Green 98]
Green T R G and Blackwell A F (1998), "A tutorial on cognitive dimensions",
- [Green 97]
Green, T. R. G. (1997), "Cognitive approaches to software comprehension: results, gaps and limitations", extended abstract of talk at workshop on Experimental Psychology in Software Comprehension Studies 97, University of Limerick, Ireland.
<http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/LimerickTalk1997/LimerickTalk.html>
- [Green 96]
Green, T. R. G. and Petre, M. (1996), "Usability analysis of visual programming environments: a 'cognitive dimensions' framework", J. Visual Languages and Computing, 7, 131-174.
<http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/index.html>
- [Green 95]
Green, T. R. G. and Navarro, R. (1995), "Programming plans, imagery, and visual programming", In Nordby, K., Gilmore, D. J., and Arnesen, S. (1995) INTERACT-95. London: Chapman and Hall (pp. 139-144).
- [Guillerault 96]

Bibliographie

Guillerault M. (1996) "Le point de vue de Cabri-géomètre II sur les coniques", Université d'été "Cabri-géomètre" de l'ordinateur à la calculatrice, De nouveaux outils pour l'enseignement de la géométrie, ed. IREM de Grenoble, Grenoble, p. 127-170, 1996.

[Halbert 84]

D. Halbert, "Programming by example", PhD. Thesis, Berkeley, California, 1984.

[Hays 95]

Hays, Judith G. and Margaret M. Burnett, "A Guided Tour of Forms/3", Oregon State University, Dept. of Computer Science, TR 95-60-6, June 1995.

<http://www.cs.orst.edu/~burnett/Forms3/Tour/tour.html>

[Heydon & Nelson 94]

Allan Heydon, Greg Nelson (1994), "The Juno-2 Constraint-Based Drawing Editor", RR 131a, digital System Resaerch Center, Palo Alto, California, décembre 1994.

[Hoftsadter 79]

Douglas Hoftsadter, "Gödel, Escher, Bach, les Brins d'une Guirlande Eternelle", (Ed.) InterEditions, 1985, idée originale publiée aux Etats Unis sous le titre "Gödel, Escher, Bach : an Eternal Golden Braid", par BasicBooks, inc., Publishers, New York, 1979.

[Horwitz & al. 93]

Horwitz S., Reps T., Binkley D., "Interprocedural Slicing Using Dependence Graphs", ACM Trans. on Programming Languages and Systems, vol. 12, N° 1, pp. 26-60, 1993.

[Ibrahim 89]

Bertrand Ibrahim , Alain Aubord, Birgit Laustsen, Michael Tepper (1989), "Techniques de Génie Logiciel pour l'EAO", Conference on "Enseignement et Apprentissage avec l'Ordinateur", Martigny, November 23-24, pp. 120-129.

[Ibrahim 95]

Bertrand Ibrahim (1995), "Une nouvelle étape dans la convivialité: les logiciels auto-éducatifs", Revue Informatique et Technologies modernes dans l'Enseignement et de la Formation, Paris, No 77, mars 1995, ISSN:1254-3985, pp. 137-142.

[Ibrahim 98]

Ibrahim Bertrand, "Diagrammatic representation of data types and data manipulations in a combined data-and control-flow language", 1998 IEEE International Symposium on Visual Languages, Halifax, Canada, September 1998.

[Jarke 99]

Matthias Jarke, "Scenarios for Modelling", in Communications of the ACM, Vol. 42, N°1, janvier 1999.

[John & al. 92]

Bonnie E. John, Philip L. Miller, Brad A. Myers, Christine M. Neuwirth, Steven A. Shafer, "Human-Computer Interaction in the School of Computer Science", Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.

<http://reports-archive.adm.cs.cmu.edu/anon/1996/CMU-CS-92-193.ps>

[Kheramane 97]

Chérif Kheramane (1997), "Spécification de présentations multimédia structurées interactives", Thèse, Institut National Polytechnique de Grenoble (INPG), Grenoble, novembre 1997.

[Kernighan & Ritchie 78].

Brian W. Kernighan, Dennis M. Ritchie, "The C programming language", Ed. Prentice-Hall, Software Series, Englewood Cliffs, New Jersey, 1978.

[Klint 93]

P. Klint (1993), "A meta-environment for generating programming environments", *ACM Transactions on Software Engineering and Methodology*, 2(2):176--201, 1993.

[Laborde C. 93]

Colette Laborde (1993), "Learning from Computers: Mathematics Education and Technology", Ed. C. Keitel & K. Ruthven, *Nato ASI Serie F*, Vol. 121, Springer-Verlag, Paris, 1993.

[Laborde C. & Capponi 94]

Colette Laborde, Bernard Capponi (1994), "Cabri-géomètre constituant d'un milieu pour l'apprentissage de la notion de figure géométrique", *Recherche en didactique des mathématiques* Vol. 14, 1-2, 1994.

[Laborde C. & Laborde JM. 91]

Laborde C., Laborde J.-M. (1991), "Micromondes et environnements d'apprentissage", in : Bellissant C. (ed.) *Actes des XIII Journées francophones sur l'informatique*. Grenoble : IMAG & Université de Genève. 157-177.

[Laborde JM.96]

Laborde Jean-Marie (1996) "Explorations en géométries non euclidiennes", Université d'été "Cabri-géomètre" de l'ordinateur à la calculatrice, De nouveaux outils pour l'enseignement de la géométrie, ed. IREM de Grenoble, Grenoble, p.105-126, 1996.

[Laborde JM. 95]

Laborde C., Laborde J.-M. (1995), "Des connaissances abstraites aux réalités artificielles, le concept de micromonde Cabri", *Environnement Interactifs d'Apprentissage avec Ordinateur (tome 2)*, Eyrolles Paris, pp. 29-41.

[Laborde JM. 89]

Laborde Jean-Marie (1989), "Intelligent Microworlds and Learning Environments", in *Intelligent Learning Environments: The Case of Geometry*, edited by J.-M. Laborde, *NATO Serie F: Computer and Systems Sciences*, (1995) vol. 117, pp. 113-132.

[Lister 98]

Lister Tim, « Hyperbolic geometry with Cabri », [LIEN ???](#), (dernière visite 6 juillet 1999), mai 1998.

[Marriott & al. 96]

Marriott K., Meyer B., Wittenburg K. (1996), "A survey of Visual Language Specification and Recognition", *Workshop on Theory of Visual Languages*, 1996.

[Marriott 96]

Marriott K., Meyer B. (1996), "Towards a Hierarchy of Visual Language", *IEEE Symp. on Visual Languages*, 1996.

[Marriott & Meyer 98]

Marriott K., Meyer B. , "Visual Language Theory." (collection of papers), Springer Verlag, 1998.
<http://www.springer-ny.com/catalog/np/nov97np/DATA/0-387-98367-8.html>

[Masui & Nakayama 94]

Masui T., Nakayama K., "Repeat and Predict - Two Keys to Efficient Text Editing", *Proceedings of CHI'94*, pp. 118-123, 1994.

[Mathelot 69]

Pierre Mathelot (1969), "L'informatique", *Que sais-je ? Presses Universitaires de France*, mai 1995.

[Maulsby 93]

David Maulsby & Alan Turransky. (1993), Appendix A : "A programming by Demonstration Chronology : 23 years of examples", in Cypher A. (Ed.), *Watch What I Do : Programming by Demonstration*, MIT Press, 1993.
<http://www.dnai.com/~cypher/WatchWhatIDo/BackMatter/Chronology.html>

[Maulsby 98]

D. Maulsby, "Including Procedures Interactively: Adventures with Metamouse", *Masters thesis. Research Rept. 88/335/47*, University of Calgary, December 1998.

Bibliographie

[McDaniel 97]

Richard G. McDaniel, Brad A. Myers, "Improving Demonstration Using Better interaction Techniques", CMU-CS-97-103, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213, 1997.

[MG-IT 97]

MG-IT : Pôle productique Rhône-Alpes (1997), "Structure d'un environnement de Conception Multi-vues et Multi-représentations", Journées "Modeleurs Géométriques", Grenoble, septembre 1995.

[Myers 86]

Brad A. Myers and William Buxton (1986), "Creating Highly-Interactive and Graphical User Interface by Demonstration", SIGGRAPH'86, Dallas, August 18-22, Vol. 20, N° 4, pp. 249-259, 1986.

[Myers 90]

Myers Brad A. (1990), "Demonstration Interface : A Step Beyond Direct Manipulation", IEEE Computer, août 90, pp. 61-73.

[Myers 93]

Myers Brad A. (1993), "Peridot : Creating User Interfaces by Demonstration", in Cypher A. (Ed.), Watch What I Do : Programming by Demonstration, MIT Press, 1993, pp. 125-153.

[Myers 95]

Myers Brad A. (1995), "UIMs, Toolkits, Interface Builders", révision de "User Interface Software Tools", ACM Transactions on Computer-Human interaction, Vol. 2, No. 1, pp. 64-103, mars 1995.

[Myers 96]

Myers Brad A. (1996), "A Brief History of Human Computer Interaction Technology", rapport de recherche CMU-CS-96-163 et CMU-HCII-96-103 de l'université de Carnegie Mellon, Pittsburg, décembre 1996.

<http://reports-archive.adm.cs.cmu.edu/anon/1996/CMU-CS-96-163.ps>

[Nanard 90]

Nanard Jocelyne (1990), La manipulation directe en interface homme-machine, thèse de doctorat d'état, Université des Sciences et Techniques du Languedoc, Montpellier II, 1990.

[Nielson 92]

H.R. Nielson and F. Nielson (1992), "Semantics with Applications: a formal introduction", John Wiley & Sons, 1992.

[Nielson 93]

Jakob Nielsen (1993), "Noncommand User Interfaces", Communications of the ACM V36, N°4 , pp. 83-99, avril 1993.

[Olivier 97]

Patrick Olivier (coordinateur), "Diagrams and Machine Reasoning", Thinking with Diagrams, 1997.

<http://www.mrc-cbu.cam.ac.uk/project/twd/discussion-papers/history.html>

[Pane & al 98]

J.F. Pane, C.A. Ratanamahatana, and B.A. Myers (1998), "Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems", submitted for publication, 1998.

<http://www.cs.cmu.edu/~pane/StudiesPaperSubmission.html>

[Pane & Myers 96]

J.F. Pane and B.A. Myers (1996), "Usability Issues in the Design of Novice Programming Systems", Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-96-132, Pittsburgh, PA, August 1996, 85 pages.

<http://reports-archive.adm.cs.cmu.edu/anon/1996/CMU-CS-96-132.ps>

<http://www.cs.cmu.edu/~pane/cmu-cs-96-132.html>

[Parzys 88]

Parzys B. (1988), "Knowing vs Seeing, problems of the plane representation of space geometry figures", Educational Studies in Mathematics 19. 1, 79-92.

[Petre 97]

Petre, M., Blackwell, A. F. and Green, T. R. G. (1997), "Cognitive questions in software visualisation", in Stasko J., Domingue, J., Price, B., Brown, M. (Eds.) *Software Visualization: Programming as a Multi-Media Experience*. MIT Press.

<ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/tg/CogQuesSoftVis.ps.gz>

[Potier 95]

Jean-Claude Potier (1995), "Contribution à la notion de programmation par démonstration. Conception sur exemple, mise au point et génération de programmes portables de géométrie paramétrée dans le système EBP", Thèse, Poitiers, France, juillet 1995.

[Quint 87]

Vincent Quint (1987), "Une approche de l'édition structurée des documents", Thèse de doctorat d'État, Université Joseph Fourier, Grenoble I, France, 1987.

[Quint 90]

Quint Vincent, Nanard Marc, André Jacques (1990), "Towards document engineering", Rapport IRISA, PI 536, Rennes (FR), 05/1990, et Rapport INRIA, RR 1244, Le Chesnay (FR) , 06/1990.

[Rabardel 95]

P. Rabardel, "Les hommes & les technologies, Approche cognitive des instruments contemporains", Série Psychologie, Armand Colin, 1995.

[Rekers 92]

J. Rekers, "Parser Generation for Interactive Environments, PhD Thesis, Amsterdam, 1992.

[Rekers 94]

J. Rekers, "On the use of Graph Grammars for defining the Syntax of Graphical Languages", Colloquium on Graph Transformation and its application in Computer Science, Palma de Majorca, Spain, Mars 1994.

<ftp://ftp.wi.leidenuniv.nl/pub/cs-techreports/tr94-11.ps.gz>

[Robert 83]

Paul Robert (1983), "Micro-Robert de poche : dictionnaire du français promordial", Ed. S.N.L. Dictionnaire LE ROBERT, Paris, 1983.

[Romero Salcedo 98]

Manuel Romero Salcedo (1998), "Alliance sur l'internet : support pour l'édition coopérative de documents structurés sur un réseau à grande distance", Thèse de l'INPG, Grenoble, 1998.

[Rothermel 98]

Gregg Rothermel, Lixin Li, Christopher Dupuis, Margaret Burnett, "What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs", in *Proceedings of the 20th Int'l Conference on Software Engineering*, pp. 198-207, Kyoto, Japan, April 1998.

[Scaife & al. 98]

Mike Scaife, Peter Cheng, Ric Lowe, Yvonne Rogers, Duska Rosenberg, "TwD : in the Head or in the World ?", contribution des sciences cognitives à "Thinking with Diagrams", discussion électronique, 1998.

<http://www.mrc-cbu.cam.ac.uk/projects/twd/discussion-papers/cognitive-science>.

[Scapin 89]

Scapin D. (1989), MAD: "Une méthode analytique de description des tâches", Colloque ingénierie des interfaces homme-machine, 1989.

[Scholl & al. 93]

P.-C. Scholl, M.-C. Fauvet, F. Lagnier; F. Maraninchi (1993), "Cours d'informatique : langages et programmation", Ed. Masson, Paris, 1993.

[Shu 86]

N. Shu, "Visual programming language, a perspective and a dimensional analysis", in *Visual Languages*, Plenum Press, New-York, 1986, pp.10-34.

[Shneiderman 82]

B. Shneiderman (1982), "The future of interactive systems and the emergence of direct manipulation", *Behaviour and Information Technology* (1), pp. 237-256.

Bibliographie

[Smith 75]

Smith D. C. (1975), "PYGMALION: A Creative Programming Environment", Ph.D. dissertation, Stanford University, 1975.

[Sugiura 96]

Atsushi Sugiura, Yoshiyuki Koseki (1996), "Simplifying Macro Definition in Programming by Demonstration", UIST'96, pp. 173-182, Seattle, november 6-8, 1996.

[Sutherland 63]

I.B. Sutherland, "Sketchpad, a man-machine graphical communication system", in Proceedings of the Spring Joint Computer Conference, pp. 329-346, 1963.

[Tarby & Barthet 96]

Tarby J.-C., Barthet M.-F. (1996), "The DIANE+ Method", CADUI'96, Namur, 1996.

[Tessier 98]

Sylvie Tessier (1998), "Les fichiers de Cabri-II", publication interne, 1998.

[Trgalová J. 95]

Trgalová J. (1995), « Étude historique et épistémologique des coniques et leur implémentation informatique dans le logiciel Cabri-géomètre. », Thèse, Université Joseph Fourier, Grenoble, 1995.

[Wagner 97]

Tim A. Wagner, (1997), "Incremental Analysis for Real Languages", PLDI'97 ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, Nevada, 15-18 June 1997.

<http://http.cs.berkeley.edu/~twagner/glr.ps>

[Yang 95]

Yang S., Burnett Margaret M., DeKoven E. & Zloof M., "Representation design benchmark: a design-time aid for VPL navigable static representations", D.C.S. Tech Rpts 95-60-3, Oregon State University, Corvallis, 1995.

[Zacklad 96]

Manuel Zacklad (1996), "Cinq dimensions pour la modélisation des interfaces homme-machine : P.A.G.I.C", ERGO-IA'96, Biarritz, 1996.

[Ziegler & al 88]

J.E. Ziegler, K.-P. Fähnrich (1988), "Direct Manipulation", in Handbook of Human-Computer Interaction, pp.123-133, M. Helander (Eds.), Elsevier Science Publishers B. V., North-Holland, 1988.

Annexes

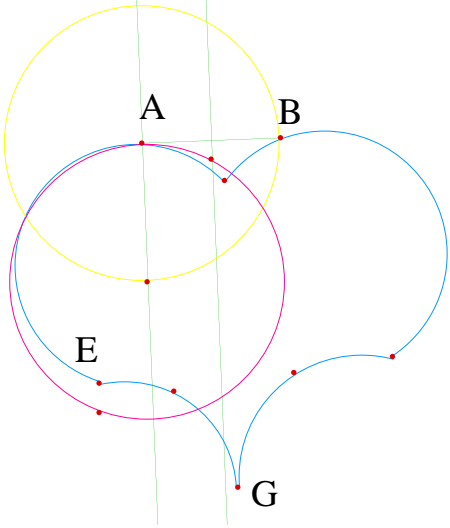
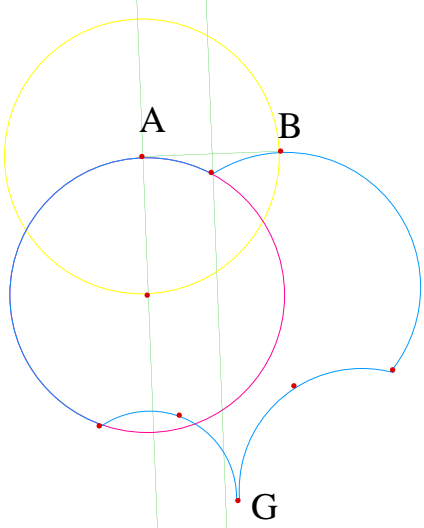
Annexes	223
Annexe A	225
Exemples d'enregistrement textuel	225
1°) Exemples d'enregistrement textuel de figure.....	225
2°) Exemples d'enregistrement textuel de macro	229
Annexe B	235
Un Cabri-programme vu comme un document structuré	235
1°) Structure logique du document programme, exprimée en langage S.....	235
2°) Présentation du document programme, exprimée en langage P.	238

Exemples d'enregistrement textuel

Dans cette annexe, nous présentons deux textes d'enregistrement de Cabri-II, et leur description en langage de géométrie formelle. Le premier décrit une figure, le second une macro.

1°) Exemples d'enregistrement textuel de figure

Le tableau ci-dessous illustre l'évolution de l'ordre induit par les contraintes entre les objets d'une figure lors de redéfinitions de contraintes. Les contraintes modifiées sont des redéfinitions de points sur des points déjà construits. La première colonne donne l'état interne de la figure avant les deux redéfinitions, et le deuxième, après. Le contenu de chaque colonne est une image de la figure, l'enregistrement sous forme textuelle de la figure et l'affichage statique de notre vue textuelle.

3e figure du film illustrant la redéfinition	4e figure du film illustrant la redéfinition
	
<p>Figure Cabri-II vers. MacOS 1.1.5</p> <p>1: Pt, 0, CN:0, VN:1 R, W, t, DS:1 1, GT:1, I, nSt Val: 0 0</p>	<p>Figure Cabri-II vers. MacOS 1.1.5</p> <p>Window center x: 0 y: 1</p> <p>1: Pt, 0, CN:0, VN:1 R, W, t, DS:1 1, GT:1, I, nSt</p>

Annexe A

2: Axes, 1, CN:1, VN:3
 Y, W, t, DS:1 1, GT:0, I, nSt
 Const: 1, Val: 1 0, 0 1

3: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 0.86_ 2.36_

"A", NP: -7, -106, NS: 17, 15
 4: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: -0.16_ 2.83_
 p: 0, Times, S: 14 C: 15 Fa: 0

"E", NP: -31, -15, NS: 16, 15
 5: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: -0.7 -0.16_
 p: 0, Times, S: 14 C: 3 Fa: 0

6: Arc, 0, CN:3, VN:5
 IBl, W, t, DS:1 1, GT:0, V, nSt
 Const: 3 4 5

7: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 0.23_ -0.26_

"G", NP: 35, 39, NS: 17, 15
 8: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.03_ -1.46_
 p: 0, Times, S: 14 C: 15 Fa: 0

9: Arc, 0, CN:3, VN:5
 IBl, W, t, DS:1 1, GT:0, V, nSt
 Const: 5 7 8

10: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.73_ -0.03_

11: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 2.96_ 0.16_

12: Arc, 0, CN:3, VN:5
 IBl, W, t, DS:1 1, GT:0, V, nSt
 Const: 8 10 11

"B", NP: 45, -104, NS: 16, 15
 13: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.56_ 2.9
 p: 0, Times, S: 14 C: 15 Fa: 0

14: Arc, 0, CN:3, VN:5

Val: 0 0

2: Axes, 1, CN:1, VN:3
 Y, W, t, DS:1 1, GT:0, I, nSt
 Const: 1, Val: 1 0, 0 1

"A", NP: -7, -106, NS: 17, 15
 3: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: -0.16_ 2.83_
 p: 0, Times, S: 14 C: 15 Fa: 0

4: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 0.3 -0.4

"G", NP: 36, 39, NS: 17, 15
 5: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.03_ -1.46_
 p: 0, Times, S: 14 C: 15 Fa: 0

6: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.73_ -0.03_

7: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 2.96_ 0.16_

8: Arc, 0, CN:3, VN:5
 IBl, W, t, DS:1 1, GT:0, V, nSt
 Const: 5 6 7

"B", NP: 45, -104, NS: 16, 15
 9: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.56_ 2.9
 p: 0, Times, S: 14 C: 15 Fa: 0

10: Seg, 0, CN:2, VN:2
 G, W, t, DS:1 1, GT:0, V, nSt
 Const: 3 9

11: PBiss, 0, CN:1, VN:2
 G, W, t, DS:1 1, GT:0, V, nSt
 Const: 10

12: Cir, 0, CN:2, VN:2
 Y, W, t, DS:1 1, GT:0, V, nSt
 Const: 3 9

13: Perp, 0, CN:2, VN:2
 G, W, t, DS:1 1, GT:0, V, nSt
 Const: 3 10

14: Int, 0, CN:2, VN:1

IBI, W, t, DS:1 1, GT:0, V, nSt
Const: 11 13 3

15: Seg, 0, CN:2, VN:2

G, W, t, DS:1 1, GT:0, V, nSt
Const: 4 13

16: PBiss, 0, CN:1, VN:2

G, W, t, DS:1 1, GT:0, V, nSt
Const: 15

17: Cir, 0, CN:2, VN:2

Y, W, t, DS:1 1, GT:0, V, nSt
Const: 4 13

18: Perp, 0, CN:2, VN:2

G, W, t, DS:1 1, GT:0, V, nSt
Const: 4 15

19: Int, 0, CN:2, VN:1

R, W, t, DS:1 1, GT:1, V, nSt
Const: 18 17

"C", NP: 51, -41, NS: 16, 15

20: Cir, 0, CN:2, VN:2

P, W, t, DS:1 1, GT:0, V, nSt
Const: 19 4

p: 0, Times, S: 14 C: 15 Fa: 2

21: Pt/, 0, CN:1, VN:3

R, W, t, DS:1 1, GT:1, V, nSt
Const: 20, Val: -0.687547840255783 -
0.532077334043842

22: Int, 256, CN:2, VN:1

R, W, t, DS:1 1, GT:1, V, nSt
Const: 16 20

Figure

Point
Point
Point
Arc (Par P1, "A" intermédiaire, et "E" final)
Point
Point
Arc (Par "E", P2 intermédiaire, et "G" final)
Point
Point
Arc (Par "G", P3 intermédiaire, et P4 final)
Point
Arc (Par P4, "B" intermédiaire, et P1 final)
Segment ("A", "B")
Médiatrice (Médiatrice de S1)
Cercle ("A" comme centre, passant par "B")
Droite perpendiculaire (Par "A", Perpendiculaire à S1)
Point(s) sur deux objets (D2, C1)
Cercle (P5 comme centre, passant par "A")
Point sur un objet (Sur C2)
Point(s) sur deux objets (D1, C2)
Fin de la figure

-> point P1
-> point "A"
-> point "E"
-> arc A2
-> point P2
-> point "G"
-> arc A3
-> point P3
-> point P4
-> arc A4
-> point "B"
-> arc A5
-> segment S1
-> droite D1
-> cercle C1
-> droite D2
-> point P5
-> cercle C2
-> point P6
-> point P7

R, W, t, DS:1 1, GT:1, V, nSt
Const: 13 12

"C", NP: 51, -40, NS: 16, 15

15: Cir, 0, CN:2, VN:2

P, W, t, DS:1 1, GT:0, V, nSt
Const: 14 3

p: 0, Times, S: 14 C: 15 Fa: 2

16: Pt/, 0, CN:1, VN:3

R, W, t, DS:1 1, GT:1, V, nSt
Const: 15, Val: -0.687547840255783 -
0.532077334043842

17: Arc, 0, CN:3, VN:5

IBI, W, t, DS:1 1, GT:0, V, nSt
Const: 16 4 5

18: Int, 256, CN:2, VN:1

R, W, t, DS:1 1, GT:1, V, nSt
Const: 11 15

19: Arc, 0, CN:3, VN:5

IBI, W, t, DS:1 1, GT:0, V, nSt
Const: 7 9 18

20: Arc, 0, CN:3, VN:5

IBI, W, t, DS:1 1, GT:0, V, nSt
Const: 18 3 16

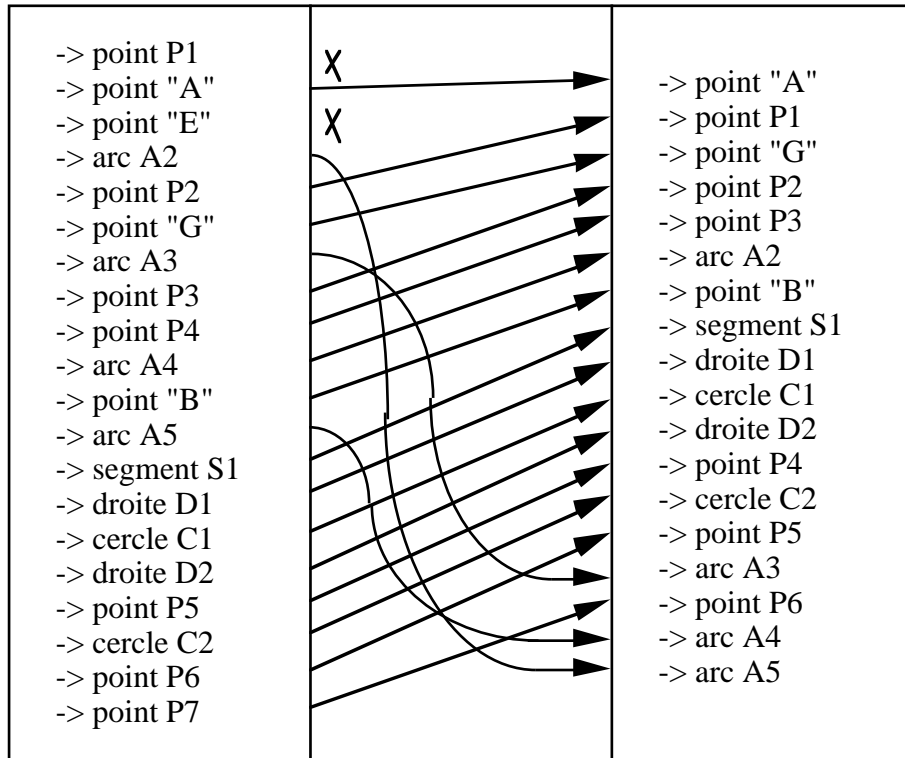
Figure

Point
Point
Point
Point
Point
Arc (Par "G", P2 intermédiaire, et P3 final)
Point
Segment ("A", "B")
Médiatrice (Médiatrice de S1)
Cercle ("A" comme centre, passant par "B")
Droite perpendiculaire (Par "A", Perpendiculaire à S1)
Point(s) sur deux objets (D2, C1)
Cercle (P4 comme centre, passant par "A")
Point sur un objet (Sur C2)
Arc (Par P5, P1 intermédiaire, et "G" final)
Point(s) sur deux objets (D1, C2)
Arc (Par P3, "B" intermédiaire, et P6 final)
Arc (Par P6, "A" intermédiaire, et P5 final)
Fin de la figure

-> point "A"
-> point P1
-> point "G"
-> point P2
-> point P3
-> arc A2
-> point "B"
-> segment S1
-> droite D1
-> cercle C1
-> droite D2
-> point P4
-> cercle C2
-> point P5
-> arc A3
-> point P6
-> arc A4
-> arc A5

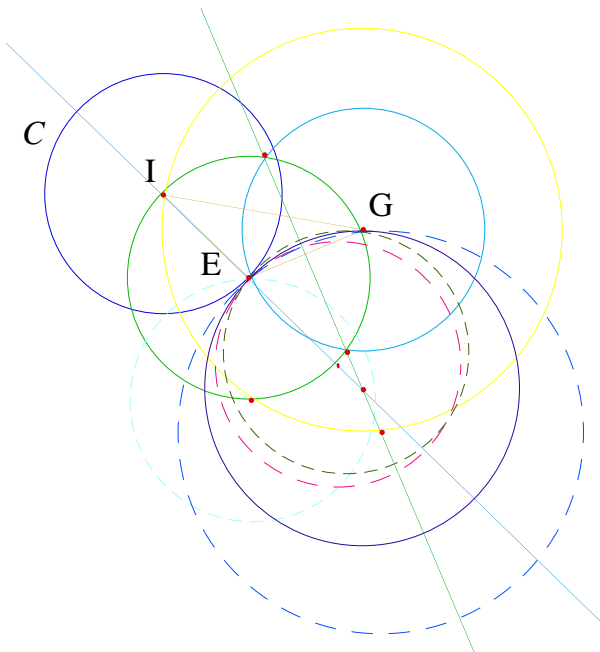
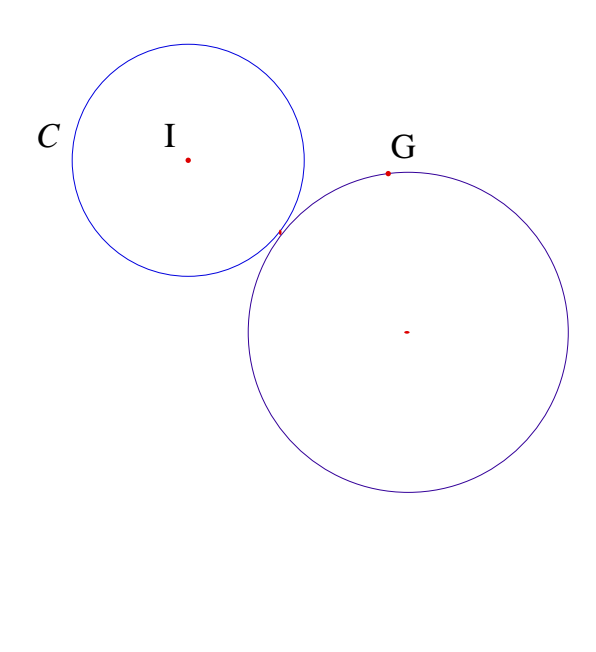
Le tableau suivant illustre l'évolution de l'ordre des objets depuis un extrait des deux vues textuelles statiques.

Les points P1 et "E" sont respectivement redéfinis en P7 et P6 qui deviennent P5 et P6 dans la nouvelle liste (à cause de la renumérotation des objets : le premier point non nommé par l'utilisateur est P1), ce qui a pour effet le déplacement de toutes les constructions qui dépendaient de P1 à la suite de la définition de P6, le déplacement de toutes les constructions qui dépendaient de "E" à la suite de P7, et l'élimination de P1 et de "E".



2°) Exemples d'enregistrement textuel de macro

Dans l'exemple présenté ici, une seule méthode de construction a été définie : celle qui "part" des points E et G et trace la bissectrice par construction et non en utilisant l'outil pré-défini. Une autre méthode pourrait être introduite dans la même macro afin de supporter le polymorphisme, c'est-à-dire des paramètres de types différents. Cette méthode pourrait tracer par exemple la médiatrice du segment [EG] directement à l'aide de l'outil médiatrice. Cette introduction d'une nouvelle méthode se fait par "sur charge" de la macro, c'est à dire en définissant une autre macro et en lui donnant le même nom.

Figure "d'école", support de la définition	Figure "d'appel"
	
<p>Figure Cabri-II vers. MacOS 1.1.5</p> <p>Window center x: 0.96_ y: -0.6</p> <p>1: Pt, 0, CN:0, VN:1 R, W, t, DS:1 1, GT:1, I, nSt Val: 0 0</p> <p>2: Axes, 1, CN:1, VN:3 Y, W, t, DS:1 1, GT:0, I, nSt Const: 1, Val: 1 0, 0 1</p> <p>"I", NP: -32, -21, NS: 12, 15 3: Pt, 0, CN:0, VN:1 R, W, t, DS:1 1, GT:1, V, nSt Val: -0.8 0.13_ p: 0, Times, S: 14 C: 15 Fa: 0</p>	<p>Figure Cabri-II vers. MacOS 1.1.5</p> <p>Used macro(s):</p> <p>cercle tangent à un autre, no name</p> <p>Icon:</p> <pre>0066000000000000 0600600000000000 60000600AAA00000 600006AA000A3300 06006A0000003300 0066A000000000A0 000A00000000000A 000A00000000000A 00A0000000000000 00A0000000000000 00A0000000000000 00A0000000000000</pre>

"C", NP: -84, -38, NS: 16, 15
 4: Cir, 0, CN:1, VN:2
 Bl, W, t, DS:1 1, GT:0, V, nSt
 Const: 3, Val: 1.66_
 p: 0, Times, S: 14 C: 15 Fa: 2

"G", NP: 61, -7, NS: 17, 15
 5: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.96_ -0.36_
 p: 0, Times, S: 14 C: 15 Fa: 0

"E", NP: -10, 18, NS: 16, 15
 6: Pt/, 0, CN:1, VN:3
 R, W, t, DS:1 1, GT:1, V, nSt
 Const: 4, Val: 0.394543218181842 -
 1.02892493300576
 p: 0, Times, S: 14 C: 15 Fa: 0

7: Seg, 0, CN:2, VN:2
 dG, W, t, DS:1 1, GT:0, V, nSt
 Const: 6 3

8: Seg, 0, CN:2, VN:2
 dG, W, t, DS:1 1, GT:0, V, nSt
 Const: 3 5

9: Line, 0, CN:2, VN:2
 lGr, W, t, DS:1 1, GT:0, V, nSt
 Const: 3 6

10: Cir, 0, CN:2, VN:2
 G, W, t, DS:1 1, GT:0, V, nSt
 Const: 6 5

11: Cir, 0, CN:2, VN:2
 Y, W, t, DS:1 1, GT:0, V, nSt
 Const: 5 3

12: Int, 0, CN:2, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Const: 10 11

13: Cir, 0, CN:2, VN:2
 O, W, t, DS:5 8, GT:0, V, nSt
 Const: 12 6

14: Int, 256, CN:2, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Const: 9 10

15: Cir, 0, CN:2, VN:2
 P, W, t, DS:8 14, GT:0, V, nSt
 Const: 14 6

16: Seg, 0, CN:2, VN:2

000A000000000000A
 000A000000000000A
 0000A000000000A0
 00000A0000000A00
 000000AA000AA000
 Mth: 0
 CN:2, ON:6, FN:3, PO:5
 CT:
 circle, CS 1, Bl, W, t, DS:1 1, GT:0, V, nSt
 point, CS 0, R, W, t, DS:1 1, GT:1, V, nSt
 Const:
 Pt/, Mth:3, 1, 0, CN:1, VN:3, Const: 2, R, W, t,
 DS:1 1, GT:1, V, nSt
 Line, Mth:1, 0, 0, CN:2, VN:2, Const: 1 4
 Seg, Mth:0, 0, 0, CN:2, VN:2, Const: 4 3
 PBiss, Mth:1, 0, 0, CN:1, VN:2, Const: 6
 Int, Mth:0, 1, 0, CN:2, VN:1, Const: 5 7, R, W, t,
 DS:1 1, GT:1, V, nSt
 Cir, Mth:1, 1, 0, CN:2, VN:2, Const: 8 4, V, W, t,
 DS:1 1, GT:0, V, nSt

Figure description:

Window center x: 1.2 y: -2.1

1: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, I, nSt
 Val: 0 0

2: Axes, 1, CN:1, VN:3
 Y, W, t, DS:1 1, GT:0, I, nSt
 Const: 1, Val: 1 0, 0 1

"I", NP: -12, 0, NS: 12, 15
 3: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: -0.03_ -0.56_
 p: 0, Times, S: 14 C: 15 Fa: 0

"C", NP: -62, 0, NS: 16, 15
 4: Cir, 0, CN:1, VN:2
 Bl, W, t, DS:1 1, GT:0, V, nSt
 Const: 3, Val: 1.57938103206428
 p: 0, Times, S: 14 C: 15 Fa: 2

"G", NP: 79, 4, NS: 17, 15
 5: Pt, 0, CN:0, VN:1
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 2.63_ -0.73_
 p: 0, Times, S: 14 C: 15 Fa: 0

Ma: cercle tangent à un autre, Const: 4 i: 0 5 i: 0
 6: Ma R, F No1, VN:3
 R, W, t, DS:1 1, GT:1, V, nSt
 Val: 1.24703806308995 -1.49137934186126

10: Ma R, F No2, VN:1

dG, W, t, DS:1 1, GT:0, V, nSt
Const: 6 5

17: Cir, 0, CN:2, VN:2
IBl, W, t, DS:1 1, GT:0, V, nSt
Const: 5 6

18: Int, 32896, CN:2, VN:1
R, W, t, DS:1 1, GT:1, V, nSt
Const: 10 17

19: Cir, 0, CN:2, VN:2
Br, W, t, DS:5 8, GT:0, V, nSt
Const: 18 6

20: Int, 32897, CN:2, VN:1
R, W, t, DS:1 1, GT:1, V, nSt
Const: 10 17

21: Line, 0, CN:2, VN:2
Gr, W, t, DS:1 1, GT:0, V, nSt
Const: 20 18

22: Int, 256, CN:2, VN:1
R, W, t, DS:1 1, GT:1, V, nSt
Const: 21 11

23: Cir, 0, CN:2, VN:2
dBr, W, t, DS:8 14, GT:0, V, nSt
Const: 22 6

24: PBiss, 0, CN:1, VN:2
dGr, W, t, DS:1 1, GT:0, V, nSt
Const: 16

25: Int, 0, CN:2, VN:1
R, W, t, DS:1 1, GT:1, V, nSt
Const: 9 24

26: Cir, 0, CN:2, VN:2
V, W, t, DS:1 1, GT:0, V, nSt
Const: 25 6

R, W, t, DS:1 1, GT:1, V, nSt

11: Ma R, F No3, VN:2
V, W, t, DS:1 1, GT:0, V, nSt

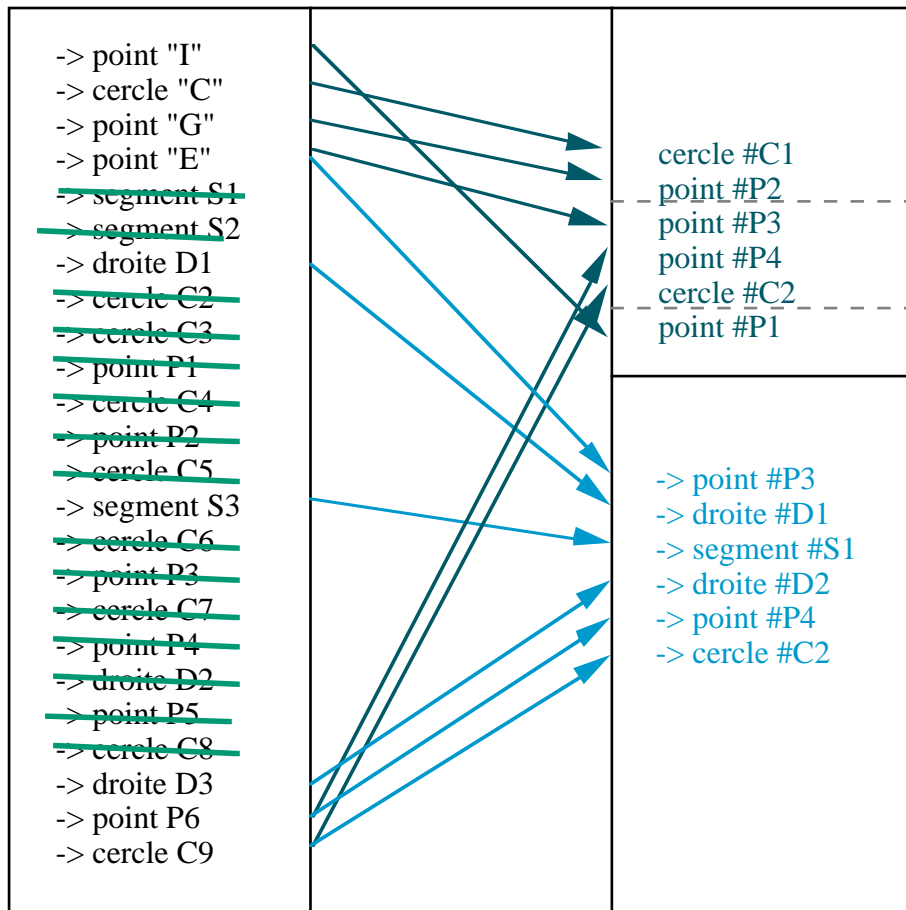
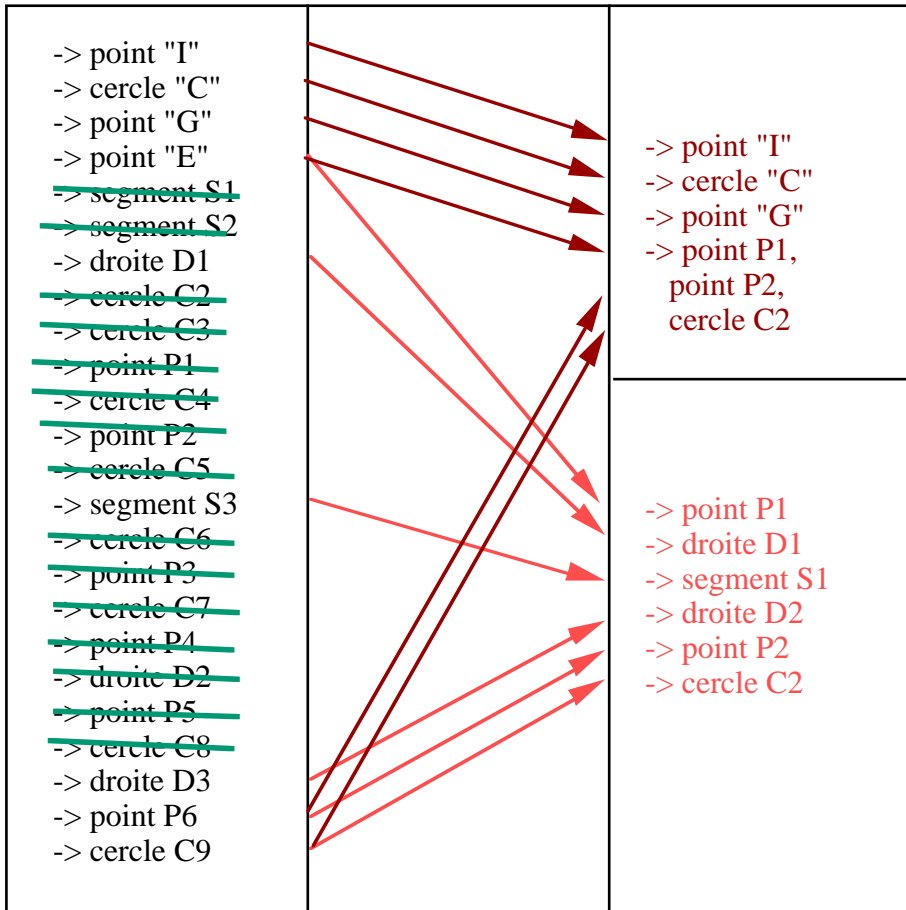
<p>Figure Point Cercle ("I" comme centre) Point Point sur un objet (Sur "C") Segment ("E", "I") Segment ("I", "G") Droite (Par "I", et "E") Cercle ("E" comme centre, passant par "G") Cercle ("G" comme centre, passant par "I") Point(s) sur deux objets (C2, C3) Cercle (P1 comme centre, passant par "E") Point(s) sur deux objets (D1, C2) Cercle (P2 comme centre, passant par "E") Segment ("E", "G") Cercle ("G" comme centre, passant par "E") Point(s) sur deux objets (C2, C6) Cercle (P3 comme centre, passant par "E") Point(s) sur deux objets (C2, C6) Droite (Par P4, et P3) Point(s) sur deux objets (D2, C3) Cercle (P5 comme centre, passant par "E") Médiatrice (Médiatrice de S3) Point(s) sur deux objets (D1, D3) Cercle (P6 comme centre, passant par "E") Fin de la figure</p>	<p>-> <u>point "I"</u> -> <u>cercle "C"</u> -> <u>point "G"</u> -> <u>point "E"</u> -> <u>segment S1</u> -> <u>segment S2</u> -> <u>droite D1</u> -> <u>cercle C2</u> -> <u>cercle C3</u> -> <u>point P1</u> -> <u>cercle C4</u> -> <u>point P2</u> -> <u>cercle C5</u> -> <u>segment S3</u> -> <u>cercle C6</u> -> <u>point P3</u> -> <u>cercle C7</u> -> <u>point P4</u> -> <u>droite D2</u> -> <u>point P5</u> -> <u>cercle C8</u> -> <u>droite D3</u> -> <u>point P6</u> -> <u>cercle C9</u></p>	<p>Figure Point Cercle ("I" comme centre) Point cercle tangent à un autre ("C", "G") > Point sur un objet (Sur "C") > Droite (Par "I", et P1) > Cercle (P1 comme centre, passant par "G") > Cercle ("G" comme centre, passant par P1) > Point(s) sur deux objets (C2, C3) > Point(s) sur deux objets (C2, C3) > Droite (Par P2, et P3) > Point(s) sur deux objets (D1, D2) > Cercle (P4 comme centre, passant par P1) Fin de la figure Macro : cercle tangent à un autre Objets initiaux> Objets finaux> Centre implicite (cercle #C1) > > Point sur un objet (Sur #C1) > Droite (Par #P1, et #P5) > Cercle (#P5 comme centre, passant par #P2) > Cercle (#P2 comme centre, passant par #P5) > Point(s) sur deux objets (#C2, #C3) > Point(s) sur deux objets (#C2, #C3) > Droite (Par #P3, et #P4) > Point(s) sur deux objets (#D1, #D2) > Cercle (#P6 comme centre, passant par #P5) Fin de la macro : cercle tangent à un autre</p> <p>-> <u>point "I"</u> -> <u>cercle "C"</u> -> <u>point "G"</u> -> <u>point P1, point P4, cercle C4</u> -> <u>point P1</u> -> <u>droite D1</u> -> <u>cercle C2</u> -> <u>cercle C3</u> -> <u>point P2</u> -> <u>point P3</u> -> <u>droite D2</u> -> <u>point P4</u> -> <u>cercle C4</u> cercle #C1, point #P2, point #P5, point #P6, cercle #C4, point #P1, -> point#P5 -> droite#D1 -> cercle#C2 -> cercle#C3 -> point#P3 -> point#P4 -> droite#D2 -> point#P6 -> cercle#C4</p>
--	---	--

Les tableaux suivants illustrent l'extraction des constructions de la figure vers le contenu ou "corps" de la macro, depuis des extraits des deux vues textuelles statiques, d'une part sur le résultat d'une utilisation de la macro (ou appel), et d'autre part sur l'expression formelle du contenu à construire par la macro.

Dans cette deuxième part (deuxième tableau), les noms des objets sont précédés de "#" pour rappeler constamment leur caractère formel, c'est à dire qu'ils sont à remplacer par les variables effectives de la figure.

La première partie de ce tableau (lignes non précédées des caractères "->") constitue l'entête de la macro, par analogie aux procédures. Ses trois parties spécifient les paramètres d'entrée, les paramètres résultats, et les paramètres implicites respectivement.. Le point #P1 est le centre implicite de cercle : ce n'est pas un objet spécifié comme un objet initial, mais il peut être implicitement utilisé dans le "corps" de la macro, comme paramètre d'entrée des constructions internes.

Dans le premier tableau, la dernière colonne est décomposée en deux zones : la partie supérieure montre la liste des objets construits accessibles par l'utilisateur, et la zone inférieure "déplie" l'appel de la macro qui a construit comme résultat P1, P2 et C2, en montrant la liste des objets construits lors de son appel. Dans les deux tableaux, les éléments non gardés sont barrés dans la première colonne.



Un Cabri-programme vu comme un document structuré

Spécification d'un programme Cabri, vue comme un document structuré en Thot.

1°) Structure logique du document programme, exprimée en langage S.

Les identificateurs d'objets ne sont créés qu'au moment de la spécification des objets construits. Dans les appels, les occurrences des identificateurs d'objets sont en fait des références aux identificateurs effectifs. Dans la figure suivante, ces liens sont représentés en bleu.

Dans les macros les identificateurs d'objets sont des identificateurs formels. Toutes les occurrences d'identificateurs d'objets instanciant un identificateur formel pointent sur l'identificateur, comme une référence par allumage. Dans la figure suivante, ces liens sont représentés en jaune.

Cette figure représente une fenêtre contenant un exemple filtré de programme Cabri, constitué d'une figure et d'une macro. Les constructions internes à l'appel de la macro sont indentées (Commandes et Résultats). Les occurrences d'un identificateur d'objet (IdObj) sont mises en évidence en vert, ainsi que les occurrences de l'identificateur formel qu'il instancie dans la macro.

```
STRUCTURE ProgrammeCabri; { Ce schéma définit la classe ProgrammeCabri }

DEFPRES ProgrammeCabriP; { Le schéma de présentation par défaut est ProgrammeCabriP }

ATTR { Définition des attributs globaux }
  EtatExécution = (Fait, EnCours, AFaire)
  { Tous les éléments de la structure peuvent passer par chacun de ces états }
  { L'état Fait induit un affichage en noir, l'état AFaire en bleu et l'état EnCours en rouge }
  { La langue n'est pas considérée comme un attribut ici, puisqu'elle est gérée ailleurs }

STRUCT { Définition de la structure générique }
  ProgrammeCabri = { Un ProgrammeCabri a une structure d'agrégat : c'est une figure suivie des macros
définies }
    BEGIN
    Figure = Constructions;
    Macros = LIST [0..*] OF (Macro); { Zéro ou plusieurs Macro(s) }
    END; { fin de la liste des Macros }

Constructions = LIST [0..*] OF ( Commande =
```

```

CASE OF { soit SimpleCommande, soit MacroCommande }
  SimpleCommande = { Une SimpleCommande est une commande de base }
  BEGIN
    AppelSimpleCommande = Text;
    ParamétrageCommande = LIST [0..*] OF (ParamCommande);
    RésultatCommande = LIST [0..*] OF (ObjetConstruit);
  END; { fin de la SimpleCommande }
  MacroCommande = { Une MacroCommande est un commande d'appel d'une macro }
  BEGIN
    AppelMacroCommande = Text;
    ParamétrageCommande;
    RésultatCommande;
    CorpsMacroCommande;
  END; { fin de la MacroCommande }

END );

CorpsMacroCommande (ATTR AffAppelMacro = (Déplié, Plié) ) =
CorpsMacroCommandeDéplié;
CorpsMacroCommandeDéplié = Constructions WITH AffAppelMacro = Déplié;
{* Paramètres *}

ParamCommande (ATTR Mode = (Sélection, DéfinitionMacro, ÉtalageÉtatObjet)) = CASE
OF { soit ParamDirect, soit ParamContextué, soit Formule }
  ParamDirect;
  ParamContextué;
  Formule = Expression;
END; { fin de ParamCommande }

ParamDirect = { Appel Direct d'un paramètre }
BEGIN
  PréSucre = Text;
  Ref_IdentificateurObjet;
  PostSucre = Text;
END;

ParamContextué = CASE OF { Appel indirect des paramètres effectifs qui sont des constituants du
paramètre direct }
BEGIN
  ParamDirect;
  PréSucre = Text;
  Parents = LIST [0..*] OF ( Ref_IdentificateurObjet );
  PostSucre = Text;
END;

Expression = LIST OF ( { minimale }
  Construction = CASE OF
    Text;
    Ref_IdentificateurObjet;
    BlocParenthèse =
      BEGIN
        Expression;
      END;
  END;
);
{* Résultats *}

```

ObjetConstruit = { Résultat de l'appel d'une commande : l'ObjetConstruit, instancié dans la figure, formel dans les macros }

```

BEGIN
TypeObjet = Text;
IdentificateurObjet (ATTR StatutObjet = Instancié, Formel) = Text;
ÉtatObjet;
END;

```

ÉtatObjet = { État d'un objet }

```

BEGIN
Manipulateur = Text;
Affichage = Text;
Valeur = CASE OF
    EquationObjet = Expression;
    dim2 =
        BEGIN
            X = Expression;
            Y = Expression;
        END;
END;

```

{* Macro *}

Macro = { Une Macro }

```

BEGIN
EntêteMacro =
    BEGIN
        SpecifObjInit = SpecifParam;
        SpecifObjFin = SpecifParam;
        SpecifObjGlob = SpecifParam;
        SpecifObjImplicLiés;
    END;
CorpsMacro = ProgrammeConstruction;
END;

```

SpecifParam = { Spécificateur de paramètre des macros }

```

BEGIN
TypeDépédance = Text; { initiaux, terminaux (dits finaux) ou globaux }
RésultatCommande;
END;

```

SpecifObjImplicLiés = LIST [Ø..*] OF (**SpecifObjImplicLié** =

```

BEGIN
LienObjImplic = Text; { par exemple Centre, pour un cercle }
ParamCommande;
RésultatCommande;
END;
);

```

ASSOC { Définition des éléments associés }

```

IdentificateurObjet = Text;
CréationMacroObjet = Text;

```

UNITS { Les éléments exportés vers les objets inclus }

```

Réfer = CASE OF { Toutes les références possibles }
    Ref_IdentificateurObjet = REFERENCE (IdentificateurObjet);

```



```

Ref_CréationMacroObjet = REFERENCE (CréationMacroObjet);
END;

Vers_ObjetConstruit = REFERENCE (ObjetConstruit);

END

```

L'animation du document programme est supportée par les concepts d'attribut et de références.

Un seul attribut global est défini dans la structure générique. « ÉtatExécution » est utilisé pour informer sur l'état du programme. Il contrôle un affichage en couleur des éléments du document (structure logique et présentation), lié à l'utilisation de l'outil « Revoir la construction » du logiciel. Cet outil permet de se déplacer dans le programme de construction de la figure à travers la manipulation d'un magnétophone. Tous les éléments de la structure peuvent passer par chacun des états « Fait », « EnCours » et « AFaire » induisant les couleurs d'affichage noir, rouge et bleu respectivement.

Un attribut local, « EtatExécution », est défini pour l'élément de la structure logique « ParamCommande ». Les trois modes de cet attribut induisent l'affichage en couleur de l'élément correspondant (si ParamDirect) ou de l'identificateur du constituant effectif (si ParamContextué ou Formule). En mode « Sélection » l'élément est affiché en vert, en mode « DéfinitionMacro » l'élément est affiché en vert ou en jaune selon que l'objet correspondant dans Cabri est choisi pour devenir un objet initial ou final respectivement, et en mode « ÉtalageÉtatObjet » en rouge pour le moment.

Un autre attribut local, « StatutObjet », est défini pour l'élément de la structure logique « IdentificateurObjet ». Cet attribut peut prendre deux valeurs : « Instancié » et « Formel ». Sous l'attribut « Formel », l'élément « IdentificateurObjet » peut constituer une référence pour des éléments « IdentificateurObjet » issus d'éléments « Macro ». Sous l'attribut « Instancié », l'élément peut faire référence à un élément IdentificateurObjet issu d'un élément « Macro ».

Un dernier attribut local, « AffAppelMacro, associé à l'élément de structure « CorpsMacroCommande », est défini afin de rendre modifiable l'état d'affichage des appels de macro : « Plié » ou « Déplié ».

2°) Présentation du document programme, exprimée en langage P.

```

PRESENTATION ProgrammeCabriP; { Le schéma de présentation par défaut est ProgrammeCabriP }

VIEWS
  TextInteractif_view; { vue du programme en Cabri-langage formel, avec accès interactif aux valeurs et
  dépliage-repliage dynamique des macros }
  TextToutAPlat_view; { vue in extenso du programme en Cabri-langage étendu, avec affichage
  complet des valeurs et toutes les macros nécessaires dépliées }

CONST
  Const_Virgule = Text ',';
  Const_ParenthOuvrante = Text '(';
  Const_ParenthFermante = Text ')';
  Const_Espace = Text ' ';

DEFAULT { Règles de présentation par défaut }

```

```

BEGIN
HorizRef : Enclosed . HRef;
VertRef : * . Left;
Width : Enclosing . Width;
Height : Enclosed . Height;
VertPos : Top = Previous . Bottom;
HorizPos : Left = Enclosing . Left;
Justify : Enclosing =;
LineSpacing : Enclosing =; { jamais besoin de "justifié" }
Visibility: Enclosing =;
Font : Enclosing =;
Style : Enclosing =;
Size : Enclosing =;
Adjust : Enclosing =;
Depth : 0;
Indent : Enclosing =;
UnderLine : Enclosing =;
Thickness : Enclosing =;
LineStyle : Enclosing =;
LineWeight : Enclosing =;
Background : Enclosing =;
Foreground : Enclosing =;
FillPattern : Enclosing =;
LineBreak : Yes;
PageBreak : Yes;
Hyphenate : Enclosing =;
END;

```

BOXES { boites de présentation spécifiques (d'articulation et de mise en page) }

{ boites de présentation d'articulation }

BVirgule: { boite virgule }

```

BEGIN
Content : Const_Virgule;
Width : Enclosed . Width;
VertPos : Bottom = Creator . Bottom;
HorizPos : Left = Previous . Right;
Style : Bold;
END;

```

BParenthOuvrante : { boite parenthèse ouvrante }

```

BEGIN
Content : Const_ParenthOuvrante;
Width : Enclosed . Width;
VertPos : Bottom = Creator . Bottom;
HorizPos : Left = Previous . Right;
Font : Creator =;
Foreground : Creator =;
END;

```

BParenthFermante : { boite parenthèse fermante }

```

BEGIN
Content : Const_ParenthFermante;
Width : Enclosed . Width;
VertPos : Bottom = Creator . Bottom;
HorizPos : Left = Previous . Right;
Font : Creator =;
Foreground : Creator =;

```

```

        END;

{ boites de présentation de mise en page }
Bespace : { boite espace }
    BEGIN
        Content : Const_Espace;
        Width : Enclosed . Width;
        VertPos : Bottom = Creator . Bottom;
        HorizPos : Left = Previous . Right;
        Font : Creator =;
        Foreground : Creator =;
    END;

Zg : { zone de gauche }
    BEGIN
        Height : Enclosing . Height + 2;
        Width : Enclosing . Width + 6;
        HorizPos : Left = Enclosing . Left - 3;
    END;

Zd : { zone de droite }
    BEGIN
        Height : Enclosing . Height + 2;
        Width : Enclosing . Width + 6;
        HorizPos : Left = Enclosing . Left + Enclosing . Width + 6;
    END;

Bulle : { Bulle }
    BEGIN
        Height : Enclosing . Height + 2;
        Width : Enclosing . Width + 6;
    END;

Page : { modèle de page }
    BEGIN
        Height : Window . Height;
        Width : Window . Width;
        Column (Zg, Zd);
        Vanishable (Bulle);
    END;

RULES { * règles de présentation des éléments du schéma de structure d'un programme Cabri * }
ProgrammeCabri :
    BEGIN
        Justify : No;
        Size : 12 pt;
        Adjust : Left;
        Width : Enclosing . Width - 1 cm;
        HorizPos : Left = Enclosing . Left + 0.5 cm;
        VertPos : Top = Enclosing . Top + 0.5 cm;
        Hyphenate: Yes;
    END;

Constructions :
    BEGIN
        Justify : No;
        Size : 12 pt;
    
```

Adjust : Left;
Width : Enclosing . Width - 1 cm;
HorizPos : Left = Enclosing . Left + 0.5 cm;
VertPos : Top = Enclosing . Top + 0.5 cm;
Hyphenate: Yes;
 END;

{* Paramètres *}

ParamCommande :

BEGIN
 Justify : No;
 Size : 12 pt;
 Adjust : Left;
 Width : Enclosing . Width - 1 cm;
 HorizPos : Left = Enclosing . Left + 0.5 cm;
 VertPos : Top = Enclosing . Top + 0.5 cm;
 Hyphenate: Yes;
 END;

ParamDirect :

BEGIN
 Justify : No;
 Size : 12 pt;
 Adjust : Left;
 Width : Enclosing . Width - 1 cm;
 HorizPos : Left = Enclosing . Left + 0.5 cm;
 VertPos : Top = Enclosing . Top + 0.5 cm;
 Hyphenate: Yes;
 END;

ParamContextué :

BEGIN
 Justify : No;
 Size : 12 pt;
 Adjust : Left;
 Width : Enclosing . Width - 1 cm;
 HorizPos : Left = Enclosing . Left + 0.5 cm;
 VertPos : Top = Enclosing . Top + 0.5 cm;
 Hyphenate: Yes;
 END;

Expression :

BEGIN
 Justify : No;
 Size : 12 pt;
 Adjust : Left;
 Width : Enclosing . Width - 1 cm;
 HorizPos : Left = Enclosing . Left + 0.5 cm;
 VertPos : Top = Enclosing . Top + 0.5 cm;
 Hyphenate: Yes;
 END;

{* Résultats *}

ObjetConstruit :

BEGIN

```
Justify : No;  
Size : 12 pt;  
Adjust : Left;  
Width : Enclosing . Width - 1 cm;  
HorizPos : Left = Enclosing . Left + 0.5 cm;  
VertPos : Top = Enclosing . Top + 0.5 cm;  
Hyphenate: Yes;  
END;
```

```
ÉtatObjet :  
BEGIN  
Justify : No;  
Size : 12 pt;  
Adjust : Left;  
Width : Enclosing . Width - 1 cm;  
HorizPos : Left = Enclosing . Left + 0.5 cm;  
VertPos : Top = Enclosing . Top + 0.5 cm;  
Hyphenate: Yes;  
END;
```

```
{* Macro *}
```

```
Macro :  
BEGIN  
Justify : No;  
Size : 12 pt;  
Adjust : Left;  
Width : Enclosing . Width - 1 cm;  
HorizPos : Left = Enclosing . Left + 0.5 cm;  
VertPos : Top = Enclosing . Top + 0.5 cm;  
Hyphenate: Yes;  
END;
```

```
SpecifParam :  
BEGIN  
Justify : No;  
Size : 12 pt;  
Adjust : Left;  
Width : Enclosing . Width - 1 cm;  
HorizPos : Left = Enclosing . Left + 0.5 cm;  
VertPos : Top = Enclosing . Top + 0.5 cm;  
Hyphenate: Yes;  
END;
```

```
SpecifObjImplicLiés :  
BEGIN  
Justify : No;  
Size : 12 pt;  
Adjust : Left;  
Width : Enclosing . Width - 1 cm;  
HorizPos : Left = Enclosing . Left + 0.5 cm;  
VertPos : Top = Enclosing . Top + 0.5 cm;  
Hyphenate: Yes;  
END;
```

```
END
```

Table des matières

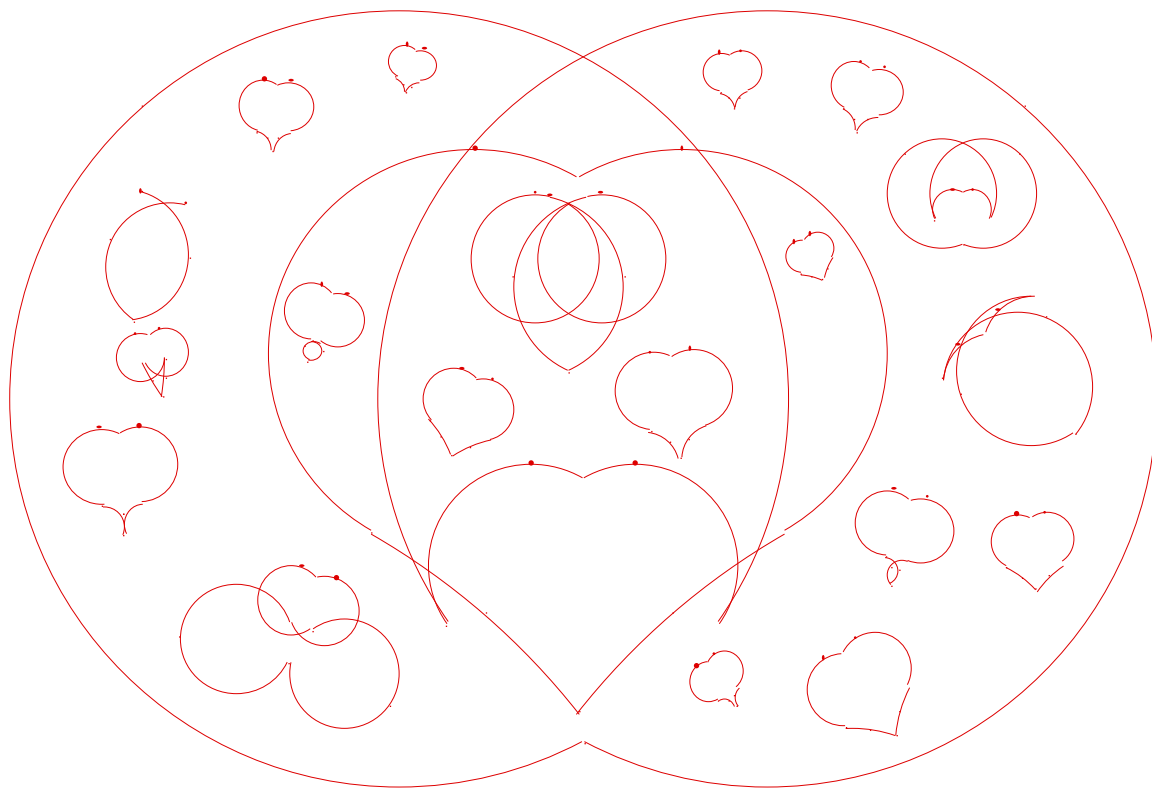
Remerciements	1
Introduction	3
Partie I	5
Manipulation directe, programmation visuelle, programmation textuelle	5
Chapitre 1	7
Le contexte : Cabri	7
A) Vers une activité de programmation.....	9
1°) D'une programmation séquentielle simple.....	12
a. Tâche d'investigation.....	12
b. Recherche d'une stratégie, parallèle avec les activités de programmation.....	13
c. Tâche de type "problème ouvert"	15
d. Réintégration et recomposition du résultat	16
2°) ...à une programmation plus évoluée	17
a. Élimination d'objets	17
b. Redéfinition de contraintes	17
c. Spécification et utilisation de macro-constructions	19
B) Nécessité de la vision et de la manipulation textuelle.....	22
1°) Limites au cas pas cas.....	23
a. Élimination d'objets	23
b. Redéfinition de contraintes	25
c. Macro-commandes	27
2°) Accès à la structure logique.....	32
a. Fichiers textuels d'enregistrement.....	33
b. Vues textuelles ou graphiques intégrées (=> identification immédiate des objets).....	37
c. Nécessité de la vue textuelle dynamique et synchronisée (modification "in extenso", apprentissage du langage par "bain linguistique").....	41
Chapitre 2	43
Du raisonnement humain à l'activité informatique	43
A) Une petite histoire de l'informatique	43
1°) De la programmation binaire aux premiers langages algorithmiques	44
a. Les assembleurs et macro-assembleurs	45
b. Les assembleurs structurés	45
c. Les premiers langages algorithmiques universels.....	45
d. Méthodologie utilisée	46
2°) Langages algorithmiques structurés (3 ^{ième} génération).....	46
Méthodologie.....	47
3°) Langages de 4 ^{ième} génération.....	51
4°) Arrivée du génie logiciel et langages 5 ^{ième} génération.....	52
a. Programmation par objets.....	52
b. Langages pour l'IA.....	53

c. PROLOG et la géométrie dynamique	54
d. Les langages de commande et LOGO	55
5°) Apparition des IHM	55
a. Leur raison d'être	56
b. Leur manière d'être	56
c. Leur influence sur le monde informatique	60
B) De nouveaux moyens pour aborder la programmation : l'apport des interfaces graphiques	60
1°) Du concept de manipulation directe aux interfaces démonstratives	60
2°) Une diversification des intervenants et une évolution des tâches	61
3°) La place des diagrammes comme aide au raisonnement	63
C) Environnements pour non-programmeurs	64
1°) Définition de la programmation visuelle	64
a. D'où elle vient	64
b. Les différentes branches	65
c. L'apport effectif de la dimension visuelle à la programmation	67
2°) Programmation par l'exemple, par démonstration	71
a. Définition	71
b. Concrètement	71
c. Application "géométrique"	72
d. Notion d'inférence	73
e. Accès aux programmes	73
3°) Approches cognitives	74
a. Des dimensions cognitives	74
b. Vers la programmation naturelle	75
D) Conclusion	77
Chapitre 3	79
D'autres logiciels permettant de programmer	79
A) Ceux qui sont très éloignés du contexte	79
1°) Logiciels de dessin	79
2°) Éditeurs textuels	80
a. Éditeurs classiques	80
b. Éditeurs de documents structurés	81
3°) Éditeurs de programmes	82
a. Éditeurs intimement liés à un langage de programmation	82
b. Éditeurs de programmes seulement liés à une syntaxe	83
B) Ceux qui ont un rapport direct fort	83
1°) Logiciels de programmation visuelle	83
a. Cocoa et Stagecast	83
b. Forms/3	87
2°) Logiciels de constructions géométriques type CAO	88
a. Juno-2	89
b. les systèmes Like, EBP et le projet GIPSE	90
c. Appolonius	90
3°) Logiciels éducatifs constituant des micro-mondes de géométrie	93
a. Sketchpad	93
b. Cinderella	97

c. GéoPlan.....	101
C) Bilan sur la programmation dans les micro-mondes de géométrie.....	104
a. Résumé.....	104
b. Critères d'évaluation de logiciels de géométrie dynamique.....	105
D) Conclusion.....	108
Chapitre 4.....	109
Apports potentiels d'une vue textuelle dynamique et synchronisée	109
A) Apports aux utilisateurs de Cabri.....	109
1°) Mise au point de grosses figures.....	109
2°) Enseignement avec Cabri.....	110
a. Préparation des exercices et illustrations de cours.....	110
b. Evaluation des connaissances et des acquis.....	110
3°) Applications de Cabri à de nouveaux domaines.....	111
a. Recherche en mathématiques.....	111
b. Prototypage graphique.....	111
B) Apports au domaine de la programmation.....	112
1°) Enseignement de la programmation.....	112
a. Où l'usage de Cabri est-il pertinent ?.....	112
b. En quoi notre version prototype peut y contribuer.....	113
2°) Pour la recherche sur les environnements de programmation.....	117
C) Conclusion.....	118
Conclusion de la partie I	119
Partie II	121
De la manipulation à la programmation	121
Chapitre 1.....	123
Désirs et contraintes	123
A. Spécificité de l'interface désirée.....	124
1°) Informations à produire.....	124
a. Structure logique.....	125
b. Génération graphique.....	129
2°) Forme à donner à l'information.....	131
a. Distance sémantique.....	132
b. Identification.....	132
c. Le langage de Cabri-programmation.....	133
3°) Animation des programmes.....	134
a. Vues et navigation.....	134
b. Concrétisation des problèmes.....	137
B) Limites cruciales (espace/temps).....	138
1°) Surcoût matériel.....	138
a. Espace.....	138
b. Temps.....	140
2°) Prolongement commercial.....	141
a. Même philosophie.....	141
b. Mêmes prestations.....	141
c. Niveau de qualité des fonctionnalités.....	142

3°) Insertion dans un projet.....	142
C) Influence de l'implémentation de Cabri.....	142
1°) Comment Cabri est-il structuré ?.....	143
2°) Conséquences sur l'insertion de l'édition textuelle dans l'existant.....	145
a. Quelles sont les manipulations attendues ?.....	145
b. Conséquences sur le code à insérer.....	152
c. Conclusion.....	155
Chapitre 2	157
Problèmes spécifiques et solutions choisies	157
A) Les problèmes	157
1°) Piloter l'édition textuelle : problèmes de conception.....	157
a. Piloter.....	158
b. Édition textuelle.....	158
c. Exemple.....	159
2°) Imbriquer plusieurs syntaxes : problèmes théoriques	161
a. Identifier les objets.....	161
b. Prise en compte de « sous-objets ».....	163
c. Prise en compte de « sur-objets ».....	163
3°) S'intégrer dans un code déjà existant : problèmes de contraintes	165
a. Les équations.....	166
b. Les palettes.....	166
c. Le multilinguisme.....	168
B) Ce qui est récupérable.....	168
1°) Description de la structure d'un Cabri-programme par un modèle de documents structurés.....	169
Exemple.....	170
2°) Comment le mettre en œuvre.....	171
C) Ce qui n'est pas récupérable	172
1°) Comment est gérée l'ubiquité des objets.....	172
2°) Gestion des notations secondaires.....	173
3°) Traitements particuliers pour les sous-objets.....	176
Chapitre 3	179
Présentation du prototype et évaluation	179
A) Présentation du prototype	179
1°) Langage de géométrie formelle.....	179
a. Lexique.....	180
b. Syntaxe.....	182
2°) Appel et fonctionnement de la vue textuelle	182
a. Le texte.....	185
b. Gguidage de l'utilisateur.....	186
c. Réactivité des identificateurs des objets.....	186
d. Affichage des attributs.....	187
3°) Fonctionnalités aidant au débogage.....	187
a. Revoir la construction et curseur textuel.....	187
b. Pliage et dépliage des macros.....	188
c. Notations secondaires.....	189

B) Évaluation	191
1°) Évaluation selon les critères d'évaluation généraux	191
a. Ergonomie.....	191
b. Accès à la sémantique.....	192
c. Comportement.....	193
2°) Limitations fonctionnelles	193
3°) Espace – temps.....	194
Chapitre 4	197
Prolongements, potentiel et perspectives	197
A) Prolongements.....	197
1°) Passer d'une version linéaire à une version hiérarchique	197
2°) Traiter uniformément toutes les manipulations directes	199
3°) Modifier les macros	200
B) Potentiel et perspectives	201
1°) Structures de contrôle et macros récursives	201
a. La récursivité dans Cabri en géométrie : vers des fractales dynamiques.....	202
b. Fonctionnement.....	203
c. Implémentation.....	204
2°) « Raccourcis » et polymorphisme.....	207
a. Extension à d'autres outils comme Droite Parallèle et aux autres outils qui construisent des droites.....	207
b. Extension à d'autres outils que les outils de construction.....	207
c. Extension à plusieurs outils ayant mêmes types d'objets initiaux et terminaux	207
d. Extension aux macros.....	208
3°) Intégration d'une vue liée à un éditeur de graphes.....	209
a. Motivations et historique.....	209
b. Quel graphe choisir ?.....	209
c. Quelle adaptation ?.....	210
d. Quels nouveaux problèmes ?.....	210
Conclusion de la partie II	211
Conclusion générale et perspectives	213
Bibliographie	215
Annexes	223
Annexe A	225
Exemples d'enregistrement textuel	225
1°) Exemples d'enregistrement textuel de figure.....	225
2°) Exemples d'enregistrement textuel de macro	229
Annexe B	235
Un Cabri-programme vu comme un document structuré	235
1°) Structure logique du document programme, exprimée en langage S.....	235
2°) Présentation du document programme, exprimée en langage P.	238
Table des matières	243



Version Cabri de
L'arbre des cœurs
de Kinkas (Brésil)