Novembre 2001

Université de Limoges

Laboratoire d'Informatique pour la Commande Numérique

Mémoire de thèse de :

Isabelle BLASQUEZ

isabelle.blasquez@unilim.fr

Tél : 05 55 43 43 99

Généralisation du z-buffer étendu :

Application à la simulation d'usinage

Remerciements

J'adresse mes plus vifs remerciements à Monsieur Richard BERLAND, Professeur à l'Université de Limoges, Directeur du Laboratoire d'Informatique pour la Commande Numérique pour m'avoir accueillie au sein de son équipe, pour la confiance qu'il m'a accordée et surtout pour la disponibilité dont il a toujours fait preuve pour me guider.

Je remercie tout particulièrement Monsieur Jean-François POIRAUDEAU, Maître de conférences à l'Université de Limoges, à l'origine de cette recherche, pour son encadrement, son soutien, ses remarques judicieuses et son aide précieuse dans la rédaction de ce mémoire.

J'exprime ma reconnaissance à Messieurs Michel MÉRIAUX, Professeur à l'Université de Poitiers et René SOENEN, Professeur à l'Université Claude Bernard de Lyon, qui ont accepté d'être rapporteurs de cette thèse et de juger ce travail.

Je remercie également Messieurs Dominique MEIZEL, Professeur à l'Université de Technologie de Compiègne et Dimitri PLEMENOS, Professeur à l'Université de Limoges, qui ont accepté d'examiner ce travail et de participer à ce jury.

Je tiens à remercier Monsieur Jean-François GUENAL, Directeur du Département de Génie Mécanique de l'IUT de Limoges, pour son aide au cours de ces recherches.

Un remerciement tout particulier à Messieurs Serge ROUX et Daniel TEXIER, Ingénieurs d'études au LICN, pour leur esprit d'équipe et leur bonne humeur.

Je remercie Messieurs Robert COUDERC et George FANNECHÈRE, Maîtres de conférences à l'Université de Limoges, pour leurs aides et leurs encouragements.

Un grand merci aux bibliothécaires, au personnel du Département de Génie Mécanique et du Département Informatique de l'IUT de Limoges pour leur sympathie, leur enthousiasme et leur soutien moral.

Enfin, je remercie tous mes proches pour leurs encouragements, et en particulier Laurent pour sa confiance et l'aide inestimable qu'il m'a apporté au cours de ces dernières années.

Table des matières

Table des figures

Liste des tableaux

Introduction

Dans la mise en œuvre de tout processus et en particulier des processus de fabrication par enlèvement de matière (usinage), la simulation possède de multiples avantages :

- sur le plan de l'action, la simulation aide à la prévision, à la décision et à l'élaboration des produits, en autorisant de nombreux essais qui permettent de choisir les caractéristiques adéquates.
- sur le plan de la sécurité, la simulation facilite la mesure des tolérances techniques de fonctionnement et l'expérimentation, sans courir le moindre risque, des procédures de sécurité.
- sur le plan financier, la simulation permet de réaliser des économies substantielles en temps et en coût : elle évite nombre d'expériences tout en permettant une meilleure évaluation des conséquences.
- sur le plan de la formation professionnelle, la simulation développe en quelque sorte le savoir faire.

En conception et fabrication assistée par ordinateur (CFAO), on a longtemps présenté abusivement comme simulation d'usinage une simulation coupée des réalités de l'atelier. Effectuée en amont dans la chaîne des données qui définissent le processus d'usinage [1], sur un modèle insuffisamment précis de la machine et de son environnement, cette simulation nécessitait en fait que soient effectuées dans de nombreux cas des opérations complémentaires de mise au point et d'essai du programme pièce avant que son introduction en production ne soit possible :

- il s'agissait tout d'abord de tester si la trajectoire globale de l'outil de coupe était faisable hors collisions, de détecter les collisions qui pouvaient avoir lieu entre les éléments de la machine-outil (outil, porte outil, ...) et ceux de l'environnement (pièce, éléments de bridage, ...).
- il s'agissait aussi d'optimiser la fabrication en éliminant tout ce qui contribuait

à augmenter le temps de cycle (trajectoires générées trop au large par crainte des collisions, profondeur de passe insuffisante, stratégie d'usinage pouvant être améliorée, ...)
- il s'agissait enfin de contrôler la dérive entre les surfaces usinées et nominales, de vérifier si les surfaces obtenues répondaient aux critères géométriques du cahier des charges, si le programme réalisait bien la pièce à usiner avec les spécifications liées à celle-ci.

Cette phase de mise au point sur site coûtait (et coûte) très cher, en charges directes comme le prix de réalisation des pièces prototypes, les dommages éventuels aux équipements de production, mais aussi en coûts indirects comme la non-utilisation de la machine-outil à des tâches de production ou l'allongement du cycle global de conception et de production d'un produit. Ce qui représente, par conséquent, une perte de productivité considérable.

Ce sont ces coûts élevés qui ont amené très tôt les chercheurs à se consacrer à ce problème selon deux approches différentes :

- d'une part rendre plus aisée la génération de programmes de qualité, et ceci
  - soit dans le cadre traditionnel de la programmation d'usinage,
  - soit en cherchant à modifier en profondeur les concepts hérités d'un autre âge sur lesquels repose cette programmation
- d'autre part fournir des outils informatiques efficaces de validation hors ligne des programmes.

Tout en restant dans le cadre de la programmation d'usinage traditionnelle, la **première démarche** a entraîné des progrès continus dans les algorithmes utilisés par les systèmes de FAO.

En se limitant à l'usinage de poches qui sert à illustrer notre travail, nous pouvons citer les travaux [2, 3] qui ont présenté des algorithmes pour la génération de trajectoires pour l'usinage en zigzag et en contours parallèles. Dans [2] est présenté un algorithme pour la génération des trajectoires de l'outil selon la stratégie en zigzag. Cet algorithme consiste à minimiser le nombre d'engagements de l'outil de coupe dans la matière en rendant plus efficace le calcul des trajectoires en zigzag. De plus, l'algorithme propose la modification des conditions de coupe tout au long de la trajectoire globale.

Dans [3], une méthode de génération de trajectoires d'outil est présentée pour la stratégie en contours parallèles, accompagnée de techniques pour l'élimination

des singularités. Il s'agit des interférences entre les trajectoires telles que les auto-intersections et les points de rebroussement des trajectoires.

Certains travaux se sont intéressés à l'application des méthodes de la géométrie algorithmique à la génération automatique des trajectoires [4, 5]. Dans [4], est présenté un algorithme pour la génération des trajectoires de l'outil utilisant les diagrammes de Voronoi. La poche est divisée en plusieurs zones monotones qui seront usinées par la stratégie des contours parallèles. La méthode des enveloppes convexes est utilisée dans [5] afin de résoudre les interférences des trajectoires de l'outil pendant la génération de celles-ci, pour le cas de l'usinage en 5 axes.

D'autres travaux se sont intéressés à la qualité de surface obtenue et au temps d'usinage. Les objectifs de ces travaux étaient de répondre aux deux besoins suivants :

- éviter d'usiner plusieurs fois des régions de la poche car l'usinage répété d'une région diminue la qualité de surface et augmente le temps d'usinage.
- diminuer le nombre excessif de rétraction d'outil de coupe pendant l'usinage de la poche car cette opération augmente fortement le temps d'usinage.

Il est proposé notamment dans [6] un algorithme général pour la génération de trajectoires en usinage de poches en zigzag, qui consiste à rendre plus efficace l'étape de génération des trajectoires en minimisant le nombre de rétractions de l'outil de coupe.

Toujours dans le cadre de ce que nous avons défini comme une première démarche possible pour la recherche, mais sur un tout autre plan, les conditions se sont trouvées réunies pour que puisse être envisagée comme possible à moyen sinon à court terme la portabilité des programmes pièces [7]. Essentiellement sous l'influence des industries aéronautiques et automobiles, cette notion est désormais une exigence du e-manufacturing. La pièce à usiner et le programme-pièce peuvent en effet avoir été conçus sur un continent et les usinages effectués dans des ateliers situés sur d'autres continents. Elle est plus complexe que celle de la portabilité d'un code puisqu'elle est aussi fonction des caractéristiques géométriques, mécaniques, (en statique et dynamique ...) des Machines Outils à Commande Numérique (MOCN) utilisées.

C'est à vrai dire la structure même de la chaîne allant de la conception à la fabrication qui, après de multiples tentatives qui furent autant d'échecs, va connaître de profondes modifications pour aboutir à une véritable intégration de ces activités autour d'un modèle unique de la pièce à obtenir.

En effet, tout programme d'usinage en Commande Numérique (CN) utilise à l'heure actuelle des notions définies dans les années 50 à 60 : le programme à l'exécution ne fait qu'enchaîner des séquences de mouvements définis par rapport aux

axes de la machine (codes G) et des fonctions binaires (codes M) prises en charge par son automate sans que l'on puisse attribuer aux primitives qu'il utilise une sémantique quelconque liée à un modèle de la pièce à usiner. L'expression "code ISO" fait référence à la norme certes bien acceptée ISO 6983 mais celle-ci ne définit que la syntaxe et un noyau minimum de fonctions du langage, chaque fabricant de machine l'accroissant de façon anarchique de ses propres extensions avec l'appui des fabricants de Commande Numérique par Calculateur (CNC : ensemble du directeur de CN et de son automate associé) ; c'est pourquoi nous préférons parler de "code G".

Dans la suite logique de l'adoption par de grandes entreprises d'un modèle neutre et standardisé de données STEP (STandard for the Exchange of Product model data, ISO 010303) et de protocoles applicables à la description géométrique de pièces et des caractéristiques de fabrication, est en cours d'adoption un ensemble de normes et protocoles (projet européen STEP NC, ISO DIS 14649). Ceux-ci définissent un modèle de données *STEP compliant* à l'interface entre les systèmes de CFAO et la commande numérique, et un nouveau paradigme conduisant à concevoir un programme pièce comme une séquence de tâches d'usinage élémentaires (*workingsteps*), chaque tâche décrivant une opération unique de fabrication utilisant un outil donné, comme un trou, une rainure, une ébauche de poche, etc .

Ceci ne concerne pas l'interface opérateur, les données d'usinage pouvant lui être présentées sous une forme quelconque selon le choix fait pour l'interface homme machine de la CNC.

Les objectifs de cette ambitieuse réforme sont les suivants :

- établir enfin un standard largement accepté pour la transmission des données CN jusqu'à l'atelier. Lui même basé sur un standard de représentation générique, il réduira tout besoin de conversion de ces données.
- supprimer de ce fait tout format intermédiaire, à commencer par le *Cutter Location file*, et la nécessité des post-processeurs.
- autoriser des modifications aisées des données CN sur la machine par l'opérateur, souvent le mieux à même de juger de leur intérêt.
- permettre le *feedback* de ces données modifiées à l'atelier vers les bureaux des méthodes et d'études, maintenant ainsi l'unicité du modèle de la pièce malgré des flux bidirectionnels entre ces entités.

On ne peut que se réjouir de voir se profiler une telle révolution qui résoudra des problèmes majeurs et *peu orthogonaux* de la CN et devrait mettre un terme à la nécessité d'utiliser des émulateurs spécifiques de CNC pour la validation hors ligne

des programmes. Cependant, outre que cette solution que notre laboratoire avait préconisée en 1993 aura été utilisée pendant une décennie si l'adoption de STEP NC par les grands donneurs d'ordres se déroule comme on peut actuellement le prévoir (mais aucune CNC conforme n'existe encore sur le marché), on notera que deux évolutions intéressantes se dessinent qui confortent les options prises en leur temps par le laboratoire pour mettre en doute l'élimination annoncée par certains de l'opérateur, et s'avèrent intéressantes dans le cadre de ce travail.

Ces informations sont déduites des scénarios hypothétiques [8] établis en mai 2001 par l'ISO pour illustrer différents cas d'applications prévisibles pour la nouvelle norme.

- il n'est pas question, malgré la disparition annoncée de toute possibilité d'extension anarchique au nouveau standard, malgré la mise en place d'un modèle cohérent, de contester la nécessité de la vérification préalable des programmes, et l'intérêt de l'effectuer si possible hors ligne [1].
- il est désormais admis qu'un opérateur expérimenté est le mieux placé dans l'atelier pour effectuer de façon interactive des modifications éventuelles (au moins technologiques) au programme pièce, valider des choix définitifs suite à ses essais, ce qui d'ailleurs ne posait problème auparavant qu'en raison des difficultés du *feedback*.

La **seconde démarche** ouverte aux chercheurs est de travailler sur la vérification des programmes d'usinage. Elle a conduit à la conception de logiciels spécifiques de simulation permettant validation et optimisation hors ligne, activité dont quelques firmes dans le monde (CGTech, SPRING Technologie...) se sont fait une spécialité. Certains de ces logiciels fonctionnent de façon autonome, d'autres ont été intégrés, au fil des restructurations des entreprises, à des systèmes de CFAO mais permettent ce que nous avons toujours considéré comme essentiel [1], une vérification externe au processus de génération du programme.

Le présent travail s'inscrit dans le cadre de cette seconde démarche, l'amélioration des performances d'outils logiciels de validation et d'optimisation, dont on vient de voir que l'usage était actuellement de plus en plus intégré à la pratique des ateliers et que l'intérêt qu'ils présentent pour les utilisateurs ne serait nullement mis en cause

[1] Avec le développement de l'usinage à grande vitesse (UGV) [9, 10], la simulation est devenue un passage quasi obligatoire. La vérification des trajectoires outils avant leur mise en application est indispensable car les vitesses d'avance pratiquées en UGV rendent bien souvent inopérantes les interventions d'urgence. En simulant le programme de la pièce, on souhaite prévoir les accidents de parcours.

par les standards à venir.

En introduisant en 1988 le concept d'émulation logicielle des commandes numériques de machine-outil [11], validé par l'implantation ultérieure de son logiciel LI-CN[2], notre laboratoire a été un des tous premiers à permettre que soit accepté en entrée d'un simulateur vérificateur le programme spécifique, en code G intégral, d'une MOCN. Pour être plus proche des préoccupations des utilisateurs (PME-PMI) auprès desquels les membres du laboratoire ont pris conscience du verrou technologique majeur que constituait pour eux la validation des programmes d'usinage, ce logiciel a été originalement conçu comme un outil d'atelier et non de bureau d'études ou de méthodes comme c'est le cas pour les systèmes de CFAO.

Il nous est apparu que trois types de difficultés pouvaient freiner l'introduction d'un logiciel de CAO/CFAO dans une entreprise de petite taille :

- **des difficultés d'ordre économique** qui regroupent l'investissement initial élevé en matériels, logiciels, la formation du personnel, l'embauche éventuelle de spécialistes, et l'évaluation précise a priori des gains de productivité qui reste souvent difficile.
- **des difficultés d'ordre technique** lorsqu'il faut suivre l'évolution rapide des matériels et logiciels.
- **des difficultés d'ordre pratique** qui sont dues à un manque de souplesse des logiciels, à une absence d'intégration des divers logiciels utilisés pour les phases successives du travail de conception, a un manque de liaison avec le processus de fabrication en aval, à la complexité du mode opératoire, et à un manque de convivialité de l'interaction homme machine.

L'un des obstacles les plus importants à la diffusion de ces logiciels dans le monde de l'industrie réside dans la difficulté citée ci-dessus de prouver la rentabilité d'un lourd investissement. Ainsi, d'un côté de très grandes entreprises peuvent se permettre d'utiliser des logiciels de CAO/CFAO de haut de gamme dont les prix dépassent plusieurs millions de francs et exploitables seulement sur des stations de travail avec des cartes graphiques spécialisées. D'un autre côté, les PME ou les PMI souhaitent aussi exploiter des logiciels performants mais plus simples, s'adaptant sur des micro-ordinateurs standard et permettant de bénéficier pleinement des connaissances et des compétences techniques des opérateurs (*skilled operator*). La convivialité de la simulation vis à vis de l'opérateur est aussi l'un des facteurs dominant de son acceptation dans les ateliers.

[2]Le logiciel LI-CN est une marque déposée de l'Université de Limoges.

Dans cette thèse nous proposons une amélioration du z-buffer étendu appliqué à la simulation d'usinage en atelier. Cette structure de données graphique a été proposée initialement par Van Hook [12] pour visualiser en simulation un usinage, et est bien adaptée aux programmes composés de nombreuses trajectoires et aux conditions exigeantes de l'atelier.

Le premier chapitre est consacré aux différentes méthodes de modélisation dans les espaces $2D$ et $3D$, et plus particulièrement à la modélisation $2D\frac{1}{2}$ des trajectoires outils.

La présentation générale de la technique du z-buffer étendu est détaillée dans le deuxième chapitre. Elle est ensuite suivie de l'application de cette technique dans le cadre de la simulation d'usinage. Toutefois, la technique du z-buffer étendu ne permet pas de garder en mémoire l'historique de l'usinage de la pièce. Pour revenir en arrière dans la simulation, il est nécessaire de rejouer celle-ci depuis le début. Dans la suite du mémoire, nous proposons des solutions à ce problème.

Dans le troisième chapitre, nous proposons un algorithme pour une simulation interactive en introduisant deux nouvelles fonctionnalités que nous appelons respectivement trace en Z et trace en temps [13]. La trace en Z permet de conserver le modèle de la pièce usinée à un instant quelconque de la simulation et ainsi de pouvoir apporter des modifications éventuelles dans le programme d'usinage. La trace en temps permet de réafficher une scène quelconque de la simulation.

Pour implémenter ces deux traces, nous nous sommes intéressé dans le quatrième chapitre aux structures de données adéquates. Nous proposons tout d'abord l'utilisation de *Skip List*, structure de données de caractère probabiliste, puis introduisons l'*Interval Treap*, nouvelle structure de données basée sur les arbres binaires d'intervalles [14].

Dans le cinquième chapitre, une évaluation expérimentale suivie d'une interprétation théorique montre que l'usage de ces deux structures est compatible avec les caractéristiques des micro-ordinateurs au standard actuel et permet la mise en place d'une simulation d'atelier conviviale.

Nous concluons notre mémoire par le sixième chapitre où après reconnaissance de la stratégie d'usinage, nous évaluons la faisabilité de la simulation en procédant à une estimation des ressources matérielles nécessaires pour qu'elle puisse être exécutée dans de bonnes conditions.

Chapitre 1

Modélisation géométrique et Simulation d'usinage

La mise en place de toute simulation de processus dans un espace en trois dimensions nécessite tout d'abord une étape de modélisation qui aboutit à la construction d'une *maquette virtuelle* calculée géométriquement par rapport à un repère tridimensionnel. Cette maquette peut ensuite être visualisée dans une orientation donnée, ce qui revient à positionner l'oeil de l'observateur en un point précis de l'espace. La modélisation géométrique et la visualisation sont a priori deux étapes distinctes pour la simulation. Cependant, dans d'autres applications, comme le calcul de structures, la modélisation géométrique peut parfois se passer de visualisation, et dans certains cas seule l'étape de visualisation suffit pour répondre au problème posé.

1.1 Modélisation géométrique

Dans son sens le plus général, le terme *modélisation* désigne la conception et l'élaboration d'un modèle théorique. Un modèle géométrique représente un ensemble de données mathématiques caractérisant un objet physique ou un de ses éléments dans l'espace. La modélisation géométrique regroupe donc l'ensemble des théories et des techniques permettant de représenter mathématiquement et de traiter jusqu'aux formes complexes. En général, le modèle se comporte comme une *maquette virtuelle* d'un objet qui n'a pas encore d'existence réelle. Il peut contenir des données de nature très diverses comme les caractéristiques géométriques, topologiques et technologiques de l'objet [15]. De nombreux modèles peuvent traduire la forme que l'on cherche à concevoir. La qualité d'un modèle est sa capacité à représenter le plus fidèlement possible l'objet réel en intégrant du mieux possible les contraintes diverses liées à cet objet. Ainsi, les modèles sont caractérisés par :

- leur degré de précision. Ils modélisent l'objet de façon exacte ou approchée. En effet, l'utilisation de modèles dans le domaine de la géologie demande une précision du modèle à une dizaine de centimètres près, alors que l'utilisation de modèles pour la mécanique nécessite une précision au micron près. Cette précision dépend bien sûr de la taille de l'objet à modéliser par rapport à son environnement. Dans le cas de la simulation d'usinage, les déplacements de l'outil de l'ordre du micron sont à l'origine des modifications du modèle représentant la pièce usinée.
- leur souplesse d'utilisation. Les modèles doivent être facilement créés ou modifiés.
- leur richesse sémantique, c'est à dire leur capacité à conserver le plus d'information possible sur l'objet à modéliser.
- leur adaptation à une application donnée.

La modélisation géométrique porte donc principalement sur la mise en forme de chacune des composantes de la maquette virtuelle et sur leur assemblage. Plusieurs techniques ont été développées. Il est possible de les regrouper en quatre classes : la modélisation 2D, la modélisation 3D filaire, la modélisation 3D surfacique et la modélisation 3D volumique.

1.1.1 La modélisation 2D

La construction du modèle est basée sur la méthode traditionnelle du dessin technique réalisé par le dessinateur industriel. Celui-ci construit les différentes vues planes de l'objet, véhiculant ainsi des données technologiques tout en définissant la morphologie du produit. Mais le modèle est pauvre car il n'existe aucune relation entre ces différentes vues. Ainsi toute modification à réaliser sur le modèle doit se faire sur toutes les vues concernées.

1.1.2 La modélisation 3D

La modélisation géométrique 3D est l'étape dans laquelle sont définies de manière numérique les diverses caractéristiques géométriques d'un objet ou d'une scène tridimensionnelle. L'outil employé, appelé modeleur, a pour but de créer un fichier regroupant toutes ces caractéristiques. Ainsi, l'objet ou la scène 3D pourront être transférés dans d'autres logiciels. Plus précisément ces fichiers de description géométrique sont utilisés dans deux domaines principaux qui sont la CFAO (Conception et Fabrication Assistées par Ordinateur) et la synthèse d'image.
La construction d'un modèle tridimensionnel permet de passer rapidement d'une vue à l'autre (par exemple du plan à la coupe). Il existe une cohérence entre toutes les vues, c'est-à-dire que toute modification dans une vue est automatiquement répertoriée dans les autres vues. On distingue habituellement trois types de modélisation 3D : filaire, surfacique et volumique. Leur différence est basée sur l'utilisation d'entité différentes : points, segments, droites, cercles, courbes, surfaces, polyhèdres, volumes.

1.1.2.1 La modélisation fil de fer (wireframe)

Dans ce modèle, le premier conçu, seules les coordonnées des sommets et les arêtes les joignant sont conservées. La face en tant que telle n'est pas connue. C'est un modèle simple qui présente certains avantages comme la création et la visualisation rapide du modèle, une faible utilisation du processeur, une faible capacité mémoire et une modification aisée des points et des arêtes. Mais cette modélisation

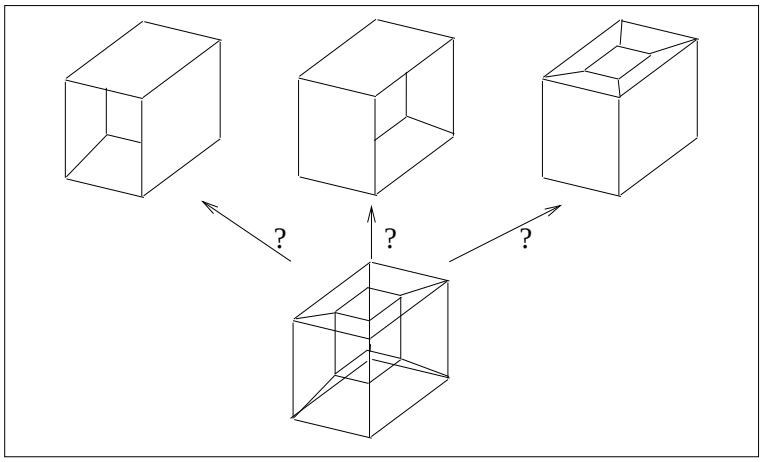

FIG. 1.1 – Ambiguïtés de la modélisation *fil de fer*

présente aussi de nombreux inconvénients. En effet, les objets ainsi modélisés sont transparents et ne permettent pas l'effacement des parties cachées. La modélisation reste ambiguë car plusieurs objets différents peuvent conduire à la même représentation puisque rien ne permet de distinguer le vide du plein (figure 1.1, page 11).

En CAO mécanique, le modèle filaire sert essentiellement à définir des points et des lignes de construction (brouillon), ainsi que des surfaces élémentaires, points et lignes servant à la définition des surfaces complexes et des volumes.

1.1.2.2 La modélisation surfacique

Ce type de modélisation considère la notion de surface limite des objets en associant au modèle précédent la notion de face. Il permet donc de définir des objets en associant des éléments de surface à des contours délimités par des arêtes. En général, les objets sont représentés soit sous la forme d'un maillage polygonal (réseau de facettes polygonales planes), soit sous la forme de surfaces paramétriques qui regroupent les surfaces de type analytiques (plane, de révolution, ...) et les surfaces de type synthétiques (Coons, Bézier, B-spline, β-spline, Nurbs [16, 17]).
La modélisation surfacique permet de définir une forme par un ensemble de surfaces, mais ces surfaces restent indépendantes les unes des autres, comme des peaux infiniment fines dont on ne verrait pas qu'elles délimitent un volume. Ainsi le modèle surfacique, s'il permet de déterminer très précisément les trajectoires d'un outil en utilisant directement la définition biparamétrique des surfaces, ne permet pas de définir la notion d'intérieur et d'extérieur.

Néanmoins, bien que la modélisation surfacique pose des problèmes de cohérence puisque chaque surface est décrite indépendamment des autres, elle a tout de même de nombreuses applications dans plusieurs secteurs industriels comme l'aéronautique et l'automobile. En effet la CAO, en tant que telle, a débuté à la fin des années soixante avec la modélisation des surfaces complexes ou formes libres et leur usinage. Tous les grands constructeurs automobiles et aérospatiaux ont développé en interne dans ces années là, des logiciels de simulation dans trois buts : faire de la *simulation d'usinage* de type bureau d'études, effectuer des calculs d'aérodynamique, juger l'esthétique à partir d'images de synthèse. Actuellement, la plupart des modeleurs géométriques utilisent en plus d'une représentation volumique un noyau surfacique destiné à la formulation géométrique des formes libres dont on ne connaît pas de formulation analytique. Le choix entre ces deux modélisations se pose lorsque les surfaces des objets sont définies à partir de réseaux de points (balayage d'une surface).

1.1.2.3 La modélisation volumique

La modélisation volumique est apparue au début des années soixante-dix. Ce type de modélisation permet de représenter sans ambiguïté les objets tout en conservant leurs caractéristiques physiques : les arêtes et les faces de l'objet, ainsi que la matière sont modélisés. La modélisation volumique se distingue de la modélisation surfacique notamment par les notions d'intérieur et d'extérieur. Elle est composée d'une enveloppe fermée et peut représenter facilement un objet physique. Une telle représentation de l'objet est aussi appelée représentation complète de l'objet. Cette modélisation est proche de la réalité, elle permet d'effectuer des calculs sur les propriétés physiques de l'objet (surface, volume, masse, centre de gravité, centre d'inertie) et de soumettre ainsi l'objet à différentes analyses, traditionnellement effectuées sur les prototypes.

Pour mettre en place la modélisation volumique, on utilise la géométrie des solides. La génération de solides ou volumes est très importante dans le domaine de la CAO comme dans celui des images de synthèse et de l'animation en 3D. Comment ne considérer une pièce mécanique que du point de vue des surfaces alors qu'il faut être capable de faire des ouvertures ou de combiner des volumes ? Les méthodes volumiques se basent sur des volumes élémentaires, tels que des primitives géométriques (comme les parallélépipédes, les cylindres ou les sphères) ou des volumes simples, puis combinent ces volumes de base pour créer de nouveaux volumes. En effet, il est plus facile d'utiliser des opérations ensemblistes (telles que l'union, l'intersection, la

différence) avec des solides plutôt qu'avec des surfaces.

On distingue généralement quatre méthodes principales de modélisation d'objets 3D : CSG, représentation par les frontières (BRep), énumération spatiale, décomposition en cellules.

1.1.3 Méthode de représentation d'objets solides

Les objets 3D manipulés dans le domaine de la modélisation géométrique volumique sont des sous-ensembles de l'espace euclidien R^3. Pour modéliser un tel objet, il suffit de définir l'ensemble des points qui décrivent le volume de l'espace occupé par l'objet.
En mathématique, la représentation d'un ensemble peut être réalisée de deux manières : soit en donnant une expression mathématique qui caractérise les éléments de l'ensemble, soit en donnant la liste complète de ces éléments. Ces deux types de représentation d'un ensemble se retrouvent au travers des deux types de modélisation volumique suivantes : la modélisation volumique continue qui regroupe des méthodes analytiques exactes de représentation de l'objet 3D et la modélisation volumique discrète qui regroupe des méthodes approximatives de représentation de l'objet 3D.

1.1.3.1 Association de solides

1. **Définitions**

   Un *objet* est un ensemble de points partagés en *points intérieurs* et *points frontières*. Les points de l'espace qui ne sont ni des points intérieurs de l'objet, ni des points frontières de l'objet sont des *points extérieurs* à l'objet.

   Les *points frontières* sont les points dont la distance entre l'objet et le complément de l'objet est nulle. Un *ensemble fermé* contient tous les points frontières, alors qu'un *ensemble ouvert* n'en contient aucun.

   La *frontière* d'un ensemble fermé est l'ensemble de ses points frontières.

   L'*intérieur* d'un ensemble fermé est le complément de la frontière par rapport à l'objet. Il est donc constitué par tous les points de l'ensemble, les points frontières exclus.

La *clôture* d'un ensemble est l'union de l'ensemble avec l'ensemble des points frontières. La clôture est un ensemble fermé.

Un *solide* est un objet à trois dimensions. Son intérieur est aussi à trois dimensions, mais sa frontière n'est qu'à deux dimensions.

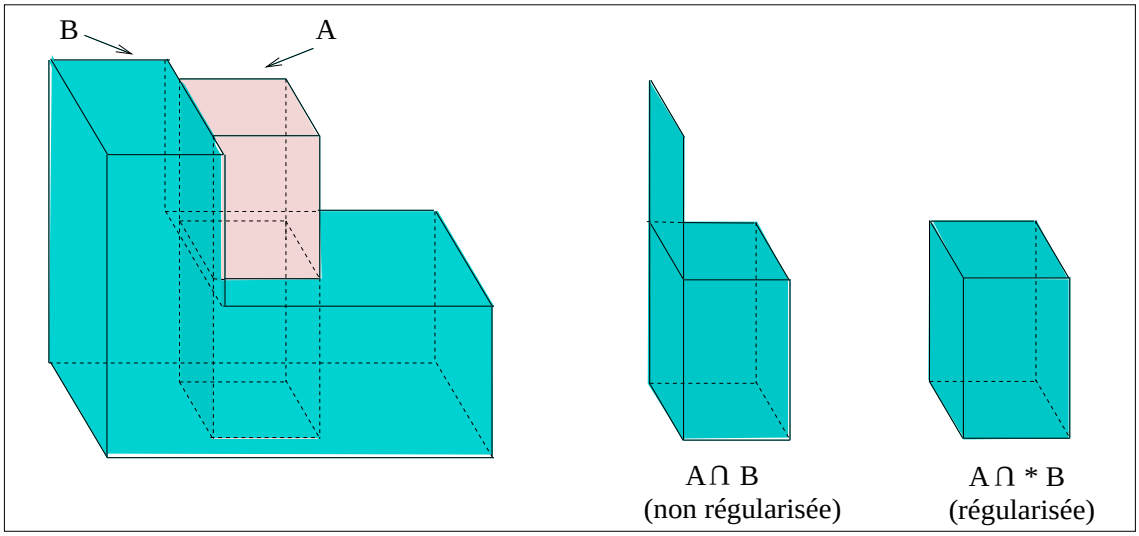

FIG. 1.2 – Opération booléenne non régularisée et régularisée

2. **Opérateurs booléens régularisés**
   Les opérations booléennes ensemblistes telles que l'union ($\cup$), l'intersection($\cap$) et la différence ($-$) sont utilisées pour combiner des objets. En appliquant une opération booléenne à deux solides A et B, nous nous attendons à obtenir comme résultat un objet solide. Mais ce n'est pas toujours le cas, comme l'illustre la figure 1.2, page 14. En effet, dans cette figure les deux solides A et B ont un bout de face en commun. Le résultat de l'intersection (classique) entre ces deux solides à trois dimensions se décompose en un parallélépipède (trois dimensions) et en un rectangle (deux dimensions). Le résultat ainsi obtenu ne peut pas être assimilé à un objet à trois dimensions. Les opérations booléennes de régularisation vont donc être utilisées pour éliminer l'objet de dimension inférieure. L'idée de la régularisation est simple et se décompose en trois étapes :
   – Tout d'abord, l'opération booléenne entre les deux solides est effectuée normalement, générant si le cas se présente des éléments de dimensions inférieures dans l'objet résultat.

- Puis, l'intérieur de l'objet résultat est calculé. Cette étape consiste à supprimer de l'objet résultat tous les éléments de dimensions inférieures. Le résultat obtenu est donc un solide sans frontière.
- Enfin, la dernière étape consiste à calculer la clôture sur le solide résultat obtenu dans l'étape précédente. En calculant la clôture, nous rajoutons alors la frontière aux points intérieurs du solide précédent.

Requicha [18] introduit les opérateurs booléens régularisés, notés $\cup^*$, $\cap^*$, $-^*$ qui assurent que les opérations booléennes sur des solides conduisent toujours à des solides. D'après ce qui précède, les opérations booléennes régulières peuvent être définies à partir des opérations booléennes ordinaires $\cup$, $\cap$, $-$ de la manière suivante :

$$\begin{aligned} X \cup^* Y &= r(X \cup Y) \\ X \cap^* Y &= r(X \cap Y), \\ X -^* Y &= r(X - Y) \end{aligned}$$

où r désigne l'opérateur de régularisation où la régularisation d'un ensemble fermé est définie comme la clôture de ses points intérieurs :

$$r(...) = \mathit{clôture}(\mathit{intérieur}(...))$$

En utilisant cette définition, le résultat de l'intersection régulière entre les deux solides A et B de la figure 1.2, page 14 donne maintenant comme résultat un solide (un parallélépipède).

1.1.3.2 La modélisation volumique continue par les méthodes exactes

1. **La modélisation solide (CSG) :**
   La modélisation par la composition arborescente de solides (*Constructive Solide Geometry* ou *CSG*) consiste à combiner des solides élémentaires (parallélépipède, prisme, cylindre, cone, sphère, tore, ...) à l'aide d'opérateurs booléens réguliers (union, intersection, différence). L'objet modélisé est ainsi décrit par la succession des opérations qui l'ont généré. La conception d'un solide complexe revient à créer un arbre binaire dit arbre CSG dont les noeuds correspondent aux opérateurs booléens réguliers et les feuilles correspondent aux solides élémentaires (figure 1.3, page 16). Pour retrouver l'objet final, il suffit de parcourir l'arbre suivant un parcours postfixé. Du point de vue mathématique, l'objet peut être représenté par une expression algébrique dont les opérandes sont des volumes de forme simple et les opérateurs des combinaisons

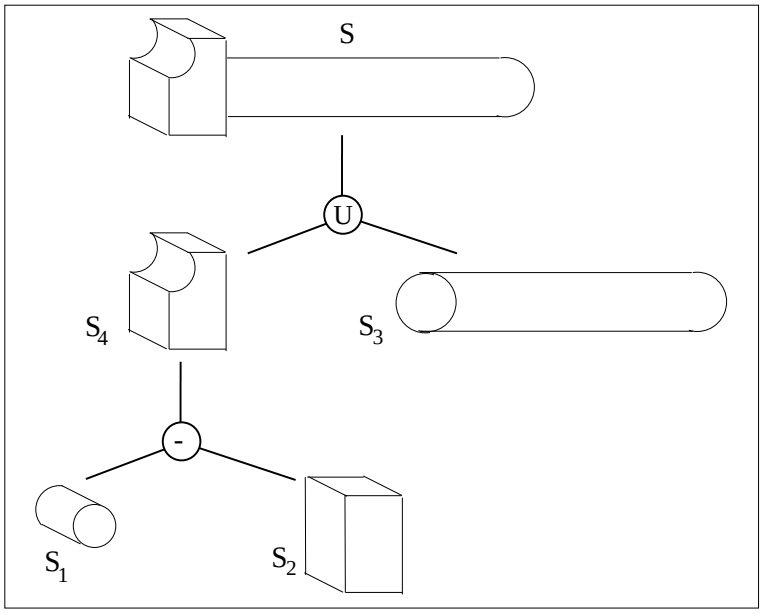

FIG. 1.3 – Exemple d'objet obtenu par une modélisation CSG

booléennes choisies pour sa construction. Par exemple, le solide S (marteau) de la figure 1.3 de la page 16 a pour équation : $S = (S_1 - S_2) \bigcup S_3$.
Un modèle géométrique exact est donc obtenu par la méthode CSG. Ce type de représentation a l'avantage d'être peu coûteux en espace mémoire : le stockage des données est compact puisque seules les définitions analytiques des solides élémentaires sont mémorisées. Le modèle CSG permet aussi de conserver l'historique des opérations effectuées et présente des facilités de modification, puisqu'il est aisé de transformer les dimensions ou la position d'un solide élémentaire. Il suffira alors de reparcourir l'arbre pour recalculer le nouvel objet.
Les avantages supplémentaires par rapport à un modèle surfacique sont directement issus de la notion de matière. Plusieurs méthodes peuvent être utilisées. La plus simple consiste à tester si un point est à l'intérieur ou à l'extérieur d'un solide complexe. Prenons par exemple le solide S (marteau) de la figure 1.3 de la page 16, *un point P est à l'intérieur de ce solide* se traduit par la vérification de l'intériorité et de l'extériorité des solides élémentaires ou de la manière suivante :
$[P \in ((S_1 - S_2) \bigcup S_3)] \Longleftrightarrow [((P \in S_1) \text{ ET } (P \notin S_2)) \text{ OU } (P \in S_3)]$
Grâce à la notion de matière, les résultats massiques (masse, centre d'inertie, ...) sont désormais calculables.
L'inconvénient majeur de la modélisation CSG est le temps d'exécution. Pour

obtenir l'objet final, il faut parcourir entièrement l'arbre CSG, ce qui peut être très coûteux en temps. En effet, le temps nécessaire à l'obtention d'un objet est raisonnable si le nombre de volumes élémentaires est réduit, mais ce temps augmente rapidement avec la complexité du solide (puisque le temps de mise à jour est directement proportionnel au nombre d'opérations rencontrées au cours de l'élaboration du solide final).
De plus, la modélisation CSG ne permet ni de concevoir des formes dites naturelles comme des minéraux ou des végétaux, ni de modéliser des surfaces gauches, ce qui ne l'empêche pas d'être largement utilisée dans le domaine de la CFAO, puisque les combinaisons booléennes telles que l'union et la différence correspondent respectivement aux procédés de soudure, collage d'une part, et d'enlèvement de matière d'autre part qui sont utilisés dans les processus industriels de fabrication.

2. **La méthode par représentation des frontières (BRep)**
   La méthode par représentation des frontières (*Boundary Representation* ou *BRep*) permet de représenter un objet par un ensemble de surfaces le délimitant. Alors qu'un solide CSG s'apparente à un *assemblage de briques*, le modèle BRep s'apparente à un *assemblage de peaux surfaciques* qui seraient cousues entre elles pour former une *gourde* étanche : le volume [19]. L'objet est alors représenté par sa frontière.

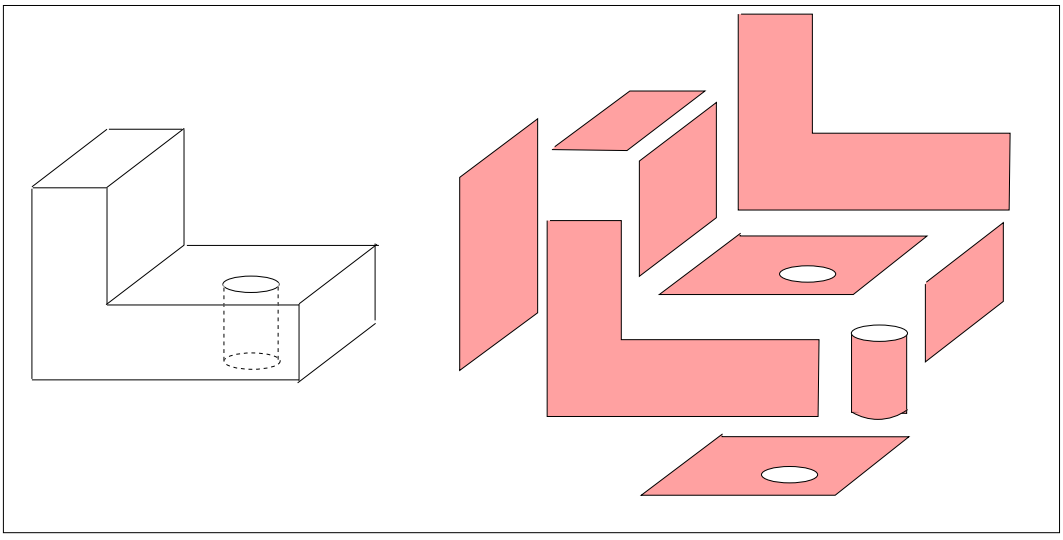

FIG. 1.4 – Les faces d'un solide en modélisation BRep

La BRep mémorise :
- **les caractéristiques géométriques** de l'objet comme les coordonnées des points caractéristiques de l'enveloppe du solide (les sommets), les équations des arêtes et des faces. Il est intéressant de remarquer que la notion d'intérieur et d'extérieur est associée à toutes les faces du solide.
- **les caractéristiques topologiques** de l'objet comme le nombre de faces, de trous, ...
- **les caractéristiques complémentaires** comme la couleur de la face, le degré de transparence de la face, ou les propriétés physiques telles que la densité, la résistance, ...

Pour les objets courbes, la notion intuitive de face disparaît (figure 1.4, page 17).

La relation d'Euler est valable pour tout polyhèdre régulier et complexe :

$$V - E + F = 2 \tag{1.1}$$

où V est le nombre de sommets, E le nombre d'arêtes et F le nombre de faces. Cette loi peut être généralisée au solide [16] :

$$V - E + F - H = 2(C - G) \tag{1.2}$$

où H est le nombre de trous dans les faces, G le nombre de trous qui passent au travers de l'objet et C le nombre de composants disjoints (parties) de l'objet (figure 1.5, page 18).

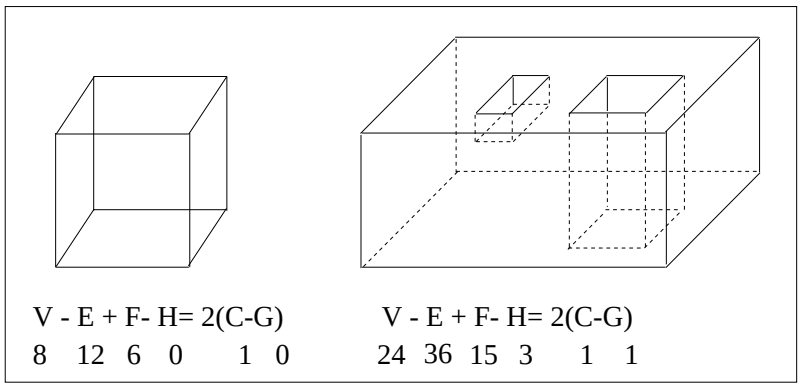

FIG. 1.5 – Equation d'Euler et polyhèdres

Pour éviter les incohérences topologiques (comme une face oubliée, par exemple), et garantir la validité des solides, les opérateurs d'Euler sont également utilisés. Chaque fois qu'un opérateur d'Euler est appliqué, l'équation précédente

reste valide. Ces opérations consistent à retirer ou à ajouter des sommets, des arêtes ou des faces tout en continuant à satisfaire la formule d'Euler généralisée.

L'avantage du modèle BRep est de pouvoir être modifié complètement ou partiellement, sans engendrer une réévaluation complète de celui-ci. De plus, les modifications peuvent être des modifications plus ou moins locales des frontières de l'objet comme par exemple la translation d'un sommet (les arêtes correspondantes s'adaptent aussi) ou la déformation locale d'une face (décalage parallèle, bombé ou creux), ...

Les représentations par les frontières ont été les premières représentations utilisées en CAO.

1.1.3.3 La modélisation volumique discrète par les méthodes approximatives

1. **Décomposition spatiale en cellules**
   La décomposition spatiale en cellules [16] consiste à découper l'espace tridimensionnel en cellules élémentaires. Chaque objet est alors décrit dans cet espace par une liste de cellules occupées. Ces cellules ne sont pas arrangées sur une grille régulière. Chaque cellule peut être décomposée pour offrir une précision supérieure. L'avantage de cette méthode est de pouvoir décomposer des formes complexes, mais elle n'est pas très précise, notamment sur le définition des contours de l'objet. Cette décomposition spatiale en cellules n'est pas nécessairement unique comme le montre la figure 1.6, page 19.

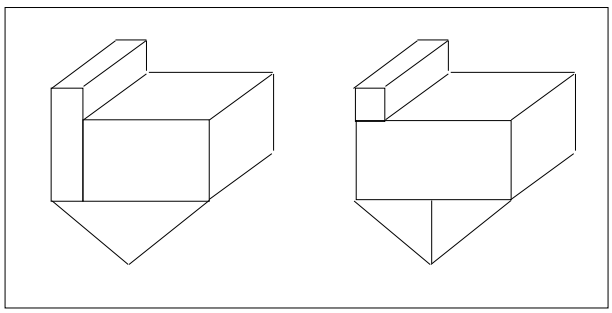

FIG. 1.6 – Décomposition spatiale en cellules d'un solide de deux façons différentes

2. **Enumération spatiale**
   L'énumération spatiale est un cas particulier de la décomposition en cellules identiques arrangées en une grille fixe et régulière (figure 1.7, page 20). Ces cellules sont appelées voxels (éléments en volume) par analogie au pixel. Les voxels sont des cellules spatiales régulières occupées par le solide, ce sont généralement des cubes de taille fixe. Plus les dimensions du cube sont petites, plus la description des objets est précise. Pour savoir si une cellule est occupée par un solide ou pas, nous utilisons le centre de la cellule. Le solide est ainsi défini par la liste des coordonnées des centres des cellules.
   L'avantage de cette méthode est la simplicité d'accès à un point donné, mais cette méthode est coûteuse en terme de mémoire.

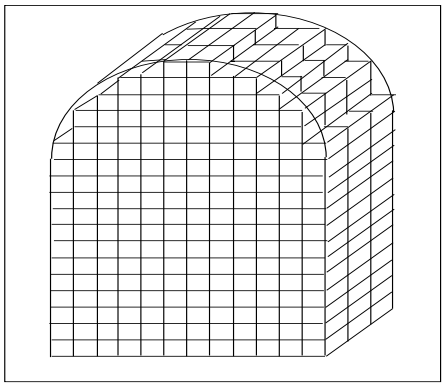

FIG. 1.7 – Enumeration spatiale

3. **arbre octal** (*octree*)
   L'arbre octal est une généralisation de l'énumération spatiale : les cubes peuvent avoir des tailles différentes à une puissance de deux près. Un arbre octal est basé sur la subdivision récursive d'un cube (ou voxel) en huit cubes (ou voxels) de même taille (octants).
   Le principe de construction est le suivant. Tout d'abord, l'objet à modéliser est recouvert par un premier cube. Ensuite, à chaque stade du processus récursif deux cas sont possibles. Soit l'objet ne remplit pas complètement ce cube, il est alors subdivisé en 8 nouveaux octants. Soit l'octant est complètement plein ou vide, la subdivision est alors stoppée et l'octant est marqué comme étant plein ou vide.
   L'octree est une extension de l'arbre quaternaire *quadtree* en deux dimensions qui est alors basé sur la subdivision récursive d'un carré en quatre carrés de même taille (quadrants).

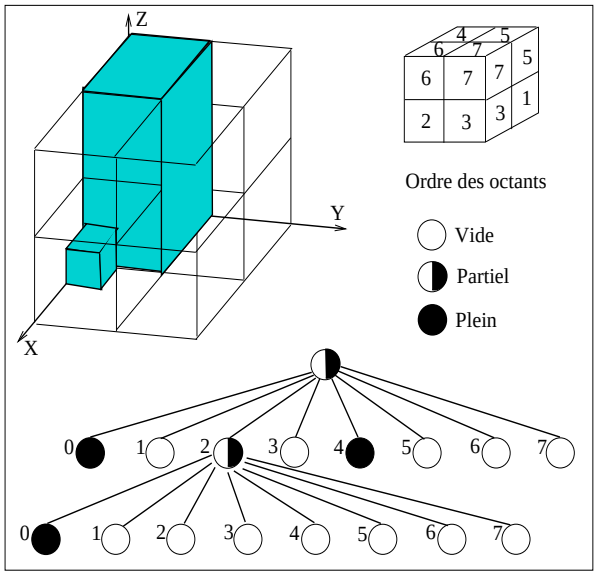

FIG. 1.8 – Représentation d'un objet sous forme d'octree

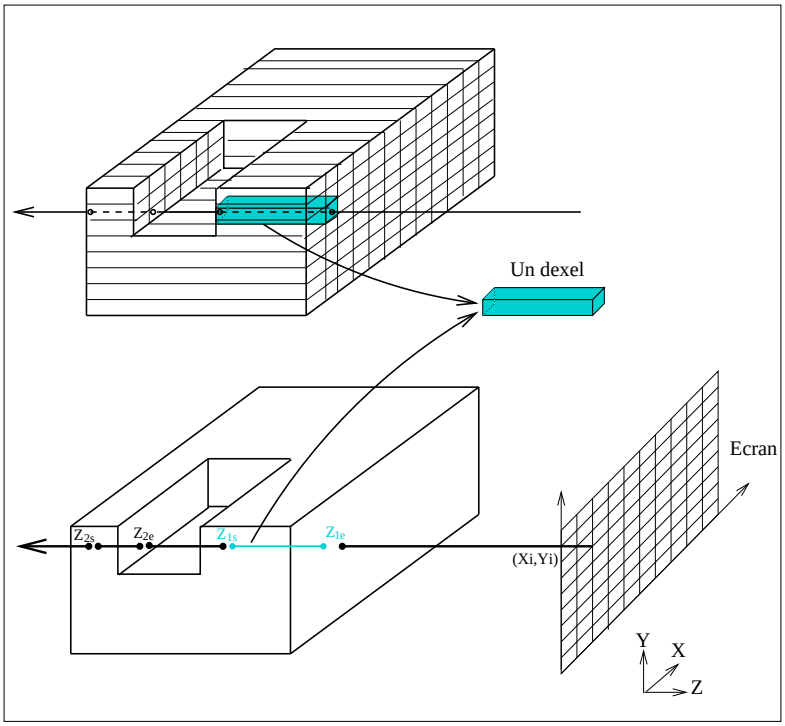

FIG. 1.9 – Méthode du z-buffer étendu

4. **z-buffer étendu**
   Le z-buffer (tampon en Z ou tampon de profondeur) étendu est une variante de l'énumération spatiale : les cellules élémentaires ne sont plus des voxels (cubes identiques), mais des dexels (parallélépipèdes dont une dimension est variable). Un dexel représente une partie du solide derrière un pixel et d'une façon plus imagée, un dexel peut être identifié à une *frite* [20] (figure 1.9, page 21). Cette approche par dexels peut être comparée à un codage par plage (*run length encoding*). De plus, comme cette méthode est destinée à définir une image matricielle de la pièce, une liste de dexels est associée à chaque pixel (élément de la matrice). La technique du z-buffer étendu sera étudiée dans les paragraphes suivants.

1.2 Choix d'une méthode pour la simulation d'usinage

Nous nous présentons brièvement dans ce paragraphe les caractéristiques de la simulation d'usinage par enlèvement de matière en fraisage (voir annexe).

1.2.1 La modélisation dans la simulation d'usinage

1.2.1.1 L'usinage en commande numérique par fraisage

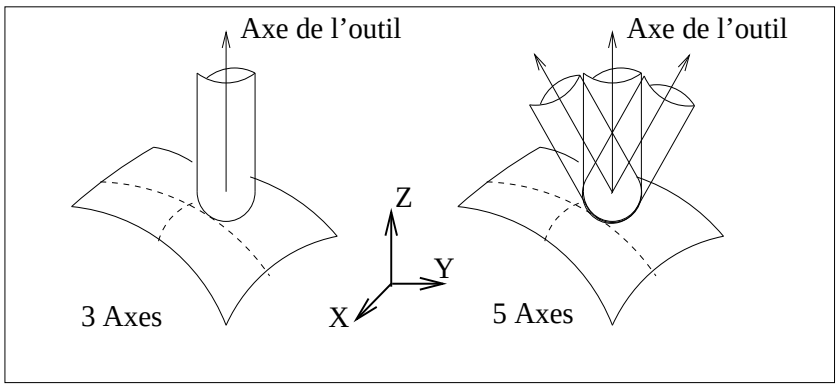

FIG. 1.10 – Comparaison entre l'usinage à 3 axes et l'usinage à 5 axes

L'espace à trois dimensions est défini par un système de référence de trois axes. Le repérage d'un solide s'y fait à partir de sa position, définie par les coordonnées d'un point de référence, et de son orientation, définie par les angles par rapport

aux axes de référence. La plupart des pièces industrielles sont usinées avec 3 degrés de liberté : x, y, z. Les valeurs de x et y décrivent le mouvement de l'outil dans le plan de travail. En usinage 3 axes, l'axe de l'outil est aligné avec l'axe z et garde une orientation fixe. Ainsi, il n'existe qu'une unique position de l'outil tangente à la surface en une position donnée. Un cas particulier très fréquent est celui de l'usinage dit en $2D\frac{1}{2}$, à profondeur constante. L'usinage à 5 axes offre deux degrés de liberté supplémentaires par rapport à l'usinage à 3 axes. Ces deux degrés de liberté (angles de rotation autour de l'axe de l'outil) permettent d'orienter l'outil (figure 1.10, page 22). En usinage 5 axes, il existe donc une infinité de positions de l'outil tangentes à la surface du point de contact. L'usinage en 5 axes permet de réaliser des surfaces ayant un seul point de contact ou une ligne (usinage par flanc) de contact entre l'outil et la surface nominale. Les exemples pris reposent principalement sur des usinages en $2D\frac{1}{2}$ en raison des contraintes d'implémentation de notre algorithme sur la version actuelle de LI-CN. Nous montrons dans la suite que ces contraintes ne limitent pas l'intérêt de la méthode proposée.

1.2.1.2 Modélisation des trajectoires outils

L'usinage complet de la pièce met en oeuvre une succession de trajectoires élémentaires de l'outil. Chacune est caractérisée par le positionnement du centre de l'outil dans l'espace (point de départ et point d'arrivée) ainsi que son mode de déplacement (interpolation linéaire ou circulaire). Une trajectoire élémentaire est orientée du *point de départ* (point courant) qui correspond au centre outil au début du segment de trajectoire vers le *point d'arrivée* qui correspond au centre outil à la fin du segment de trajectoire. La simulation consiste à reproduire le mouvement de l'outil en déterminant le volume de matière enlevé au brut par l'outil au cours d'une trajectoire. La modélisation des volumes est donc une étape importante de la simulation. Pour chaque nouvelle trajectoire, le volume de la pièce (le brut) et le volume de l'outil sont comparés, et le volume du brut est modifié. La simulation d'usinage peut se résumer à une succession d'opérations booléennes régulières de différence puisque les volumes successifs balayés par l'outil sont soustraits au volume restant du brut. Durant la simulation, une représentation graphique est affichée.

Le *volume balayé* par l'outil sur une trajectoire élémentaire est l'union des surfaces balayées par l'outil sur la même trajectoire élémentaire, et cela sur la hauteur de l'outil. Le volume balayé par l'outil peut aussi se décomposer comme l'union de trois volumes : $V_1 \bigcup V_2 \bigcup V_3$ où V_1 représente le volume de l'outil dans sa position finale, V_2 représente le volume de l'outil dans sa position initiale, V_3 représente le volume intermédiaire défini par l'ensemble des positions de l'outil (hors volume de

l'outil dans la position initiale et la position finale de la trajectoire), ce volume est délimité par les surfaces balayées par les bords de l'outil.

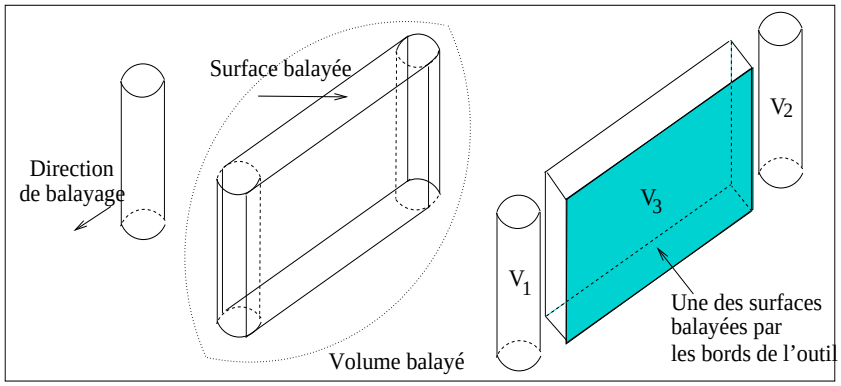

FIG. 1.11 – Volume balayé

1.2.2 Les différentes méthodes possibles

Dans la littérature, plusieurs méthodes sont présentées dans le domaine de la simulation d'usinage. Nous pouvons distinguer trois principales catégories : les méthodes basées sur la modélisation permettant d'obtenir une représentation complète de la pièce, les méthodes d'aide à la vérification du programme d'usinage et les méthodes basées sur la visualisation.

1.2.2.1 Les méthodes basées sur la modélisation

Elles ont été les premières méthodes utilisées dans la simulation d'usinage. Tout d'abord, Sungurtekin et Voelcker [21] se sont intéressés à l'utilisation du modèle CSG pour la simulation des programmes d'usinage sur des machines à commande numérique. La première étape consiste à construire le volume balayé par l'outil à l'aide d'opérations booléennes (différence, union, intersection) sur des volumes primitifs tels que des cylindres, des sphères, des parallélépipèdes (voir figure 1.12, 25). Pour modéliser l'enlèvement successif de matière, les volumes balayés sont soustraits à la pièce brute par l'opération booléenne de différence. Un modèle géométrique est construit sous forme d'un arbre CSG, où les noeuds sont les opérations booléennes et les feuilles sont les volumes primitifs (figure 1.13, page 26).

La représentation sous cette forme (arbre CSG) conserve l'historique de construction ; une opération *défaire* peut avoir lieu. Cependant, la visualisation graphique

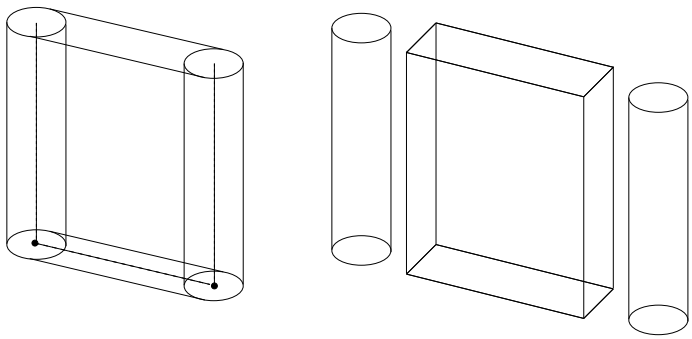

FIG. 1.12 – Modélisation du volume balayé.

nécessite d'autres algorithmes d'évaluation de l'arbre CSG, mais elle permet de visualiser le modèle sous n'importe quel point de vue.

Il existe d'autres méthodes qui font intervenir d'autres modélisations comme la représentations par frontières [22] ou les Graftrees [23]. Les Graftrees sont basées sur les octrees, et les noeuds de l'octree sont des éléments construits en CSG.

Quoi qu'il en soit, les approches basées sur la modélisation fournissent un modèle géométrique exact et un temps d'exécution raisonnable si le nombre de trajectoires que compte le programme d'usinage est faible. Cependant, ce temps s'accroît rapidement avec la complexité de la pièce à usiner. En effet, pour chaque trajectoire, un calcul précis entre le volume balayé de l'outil et le volume de plus en plus complexe de la pièce doit être exécuté. Un usinage complet peut comprendre plusieurs dizaines de milliers de trajectoires. Ainsi, les intersections à évaluer deviennent de plus en plus nombreuses, ce qui entraîne des calculs de plus en plus complexes et un temps d'exécution qui croît considérablement. En effet, une simulation s'exécute en un temps en $O[n^4]$, où n est le nombre de trajectoires du programme d'usinage.
Dans ce cas, les méthode basées sur la modélisation apparaissent trop lentes pour pouvoir effectuer dans l'atelier, une simulation sur des pièces complexe [1]. Pour réduire ce temps d'exécution, il est nécessaire de disposer aussi de méthodes basées sur la visualisation.

[1] Il nous parait significatif que les exemples fournis par les promoteurs de STEP NC ne portent que sur des pièces prismatiques relativement simples

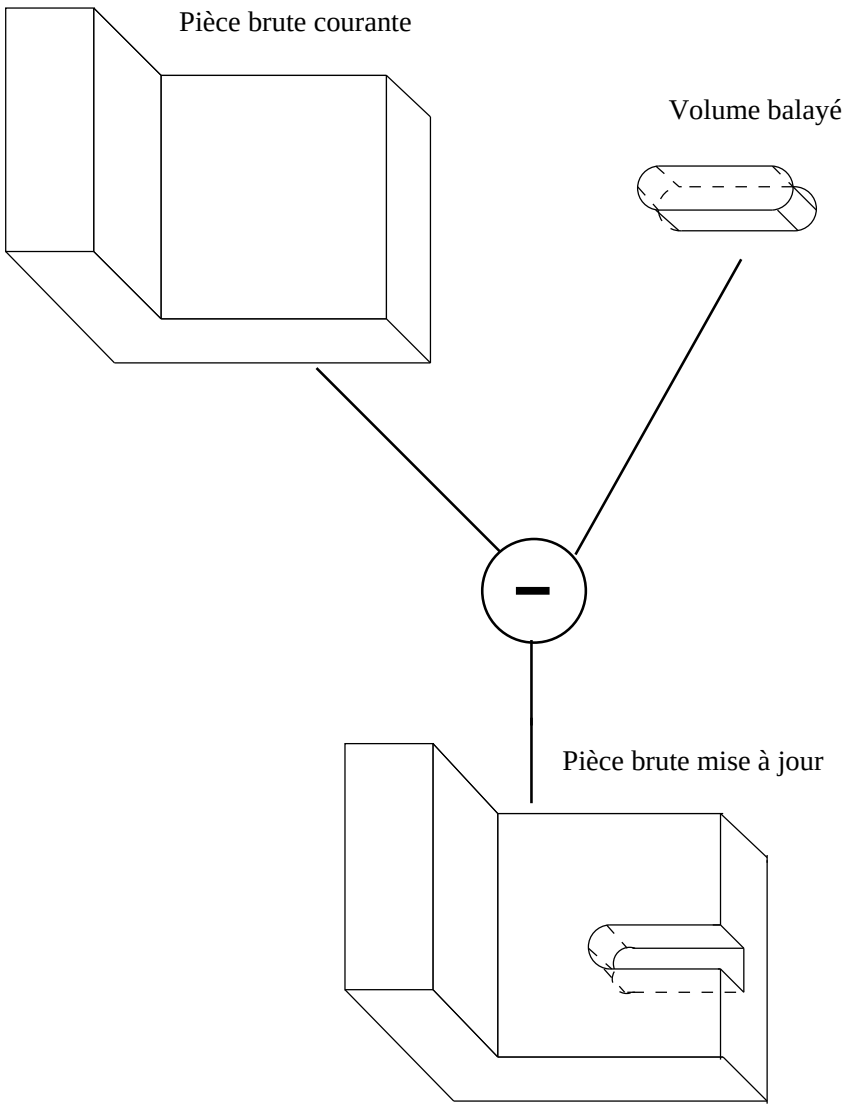

FIG. 1.13 – Actualisation du modèle géométrique de la pièce brute.

1.2.2.2 Les méthodes d'aide à la vérification

Une autre approche de la simulation d'usinage introduite par Jerard [24] est appelée intersection des vecteurs discrets (ou approche *lawn moving*). Une première étape de discrétisation transforme la surface conçue initialement en une distribution de points. A chaque point est associé un vecteur normal ou tout autre vecteur choisi arbitrairement. Un calcul d'intersections permet de déterminer la distance directe (valeur de coupe) entre chaque point de la surface conçue initialement et les points des surfaces balayées par l'outil en réduisant la taille des vecteurs initiaux. Cette approche est destinée à la vérification du programme d'usinage en évaluant ainsi l'erreur en chaque point de la surface. Toutefois, seul un affichage graphique de l'erreur d'usinage peut être envisagé.

1.2.2.3 Les méthodes basées sur la visualisation

Van Hook [12] a proposé en 1986 une nouvelle technique pour visualiser en temps réel une simulation d'usinage. Il utilise un z-buffer étendu pour décrire le brut suivant une direction de vue fixée, et un z-buffer étendu pour l'outil. Cette méthode basée sur la visualisation décompose le solide en une succession d'éléments de base (les dexels), et ramène le temps d'exécution à $O(n)$ [24]. Le temps de simulation croît donc linéairement avec le nombre de mouvements d'outil. L'avantage de cette méthode est sa rapidité d'exécution par rapport aux méthodes précédentes. La simulation mise en place par Van Hook avait un taux de mise à jour élevé (par rapport aux méthodes existantes) de 10 opérations de coupe par seconde. Une opération de coupe, qui correspond aussi à un mouvement de l'outil, regroupe toutes les opérations booléennes régularisées nécessaires pour soustraire dans une position donnée, un outil composé de 100×100 pixels au brut. Van Hook pouvait ainsi visualiser un enlèvement *continu* de matière et obtenir une simulation réaliste.

Ces méthodes dépendent du point de vue choisi, mais elle sont plus rapides que les méthodes utilisant une représentation complète et ne nécessitent pas de matériel particulier pour mener à bien les calculs d'intersection de volume. Nous développerons plus précisément la technique du z-buffer étendu dans les chapitres suivants.

Chapitre 2

Le z-buffer étendu

2.1 Présentation de la technique du z-buffer étendu

2.1.1 Retour sur le z-buffer

La méthode du z-buffer est une méthode d'élimination des parties cachées d'un modèle surfacique [25]. L'image produite est représentée par une matrice de pixels, appelée z-buffer ou *depth-buffer*. Pour chaque pixel, la méthode gère un couple de valeurs (z, c) : une profondeur et une couleur. La profondeur est la distance à l'oeil de l'objet vu depuis ce pixel.
Les profondeurs sont initialisées à la plus grande valeur possible, puis tous les objets de la scène sont traités les uns après les autres dans un ordre quelconque. Pour chaque nouvel objet de la scène, la profondeur est calculée en chacun des pixels de l'objet. Si elle est inférieure à la profondeur actuelle en ce pixel, alors la nouvelle surface est plus proche de l'oeil que la surface précédemment analysée : la profondeur et la couleur de la nouvelle surface sont affectées au pixel correspondant du z-buffer (figure 2.1, page 29 où le niveau de gris représente sa couleur et la profondeur correspond au nombre associé à chaque pixel).

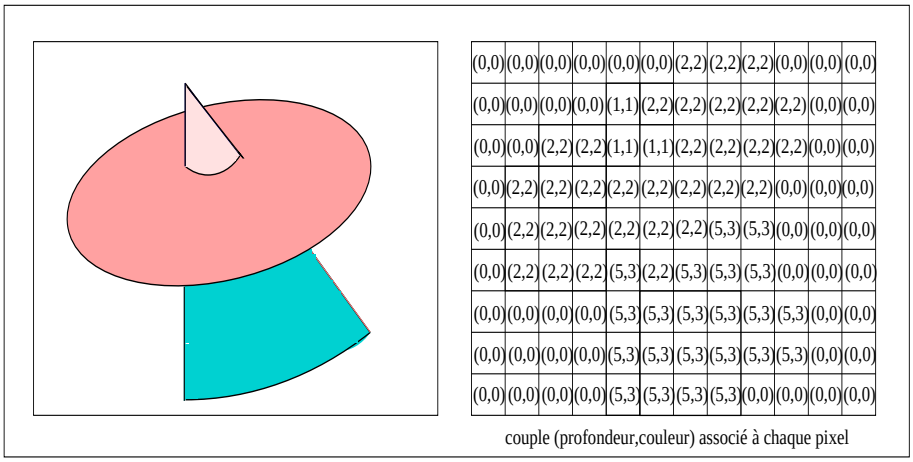

FIG. 2.1 – Une image et son z-buffer asssocié.

2.1.2 Modélisation d'un solide par la technique du z-buffer étendu

Le z-buffer étendu est une représentation 3D qui permet d'approximer la représentation géométrique d'un solide comme un ensemble de parallélépipèdes rectangles, généralement appelés éléments en profondeur ou dexels (*depth elements*) copiés sur

pixels (*picture elements*).
Van Hook [12] a été le premier à fixer un point de vue dans l'espace image et à utiliser un z-buffer étendu pour la simulation. Dans cette méthode, on associe à chaque pixel une liste contenant des informations en profondeur sur l'objet à modéliser dans une direction donnée. Une liste ordonnée de dexels est ainsi associée à chaque pixel de l'écran (figure 2.2, page 30).
Un procédé d'intersection de rayons avec le solide est utilisé pour convertir le solide en une représentation par dexels. Des rayons parallèles sont lancés à partir des points de la grille (écran) vers le solide. Le résultat de l'intersection entre le solide et ces rayons issus de la grille donne un ensemble de segments qui permet alors de générer un ensemble de dexels pour représenter l'objet. Géométriquement un dexel peut être interprété comme un segment dans une direction de vue donnée. Les dexels sont en fait des segments *dans* le solide.

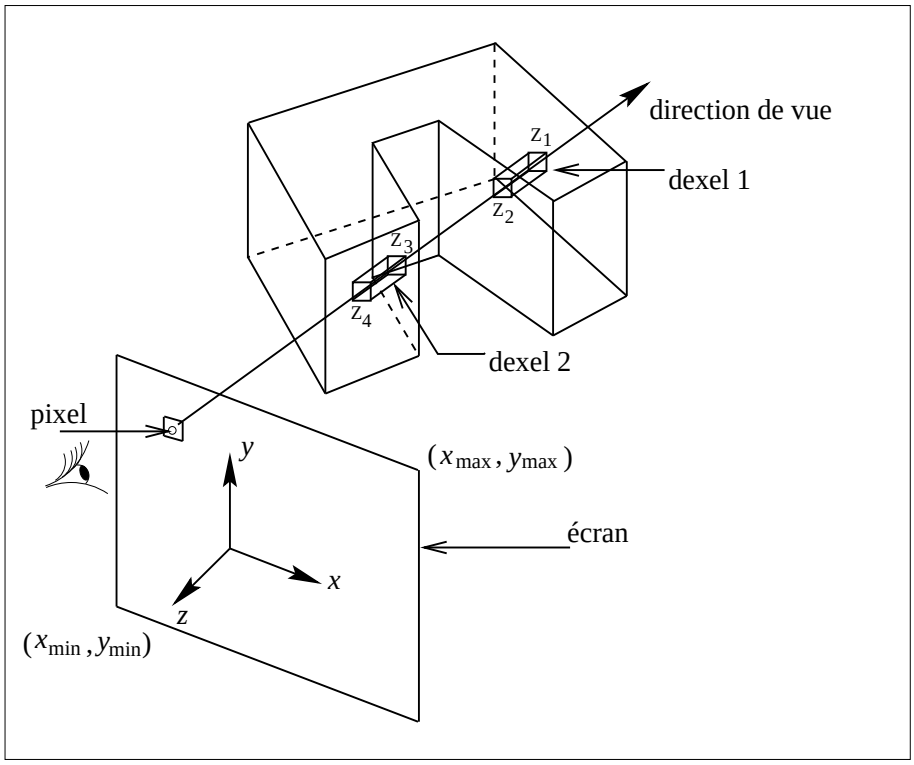

FIG. 2.2 – Le z-buffer étendu.

Un dexel représente un parallélépipède rectangle aligné sur l'axe des Z, il correspond ainsi à une partie du solide derrière un pixel et suivant une direction donnée.

Il contient des informations de type spatial (valeurs en Z) et de type graphique (couleur) de l'objet à modéliser. La structure d'un dexel est construite suivant la méthode du z-buffer, à l'exception près que la valeur en Z la plus proche comme la valeur en Z la plus éloignée sont mémorisées dans chaque dexel.
Soit une fenêtre graphique de taille définie par ses pixels extrêmes (x_{min}, y_{min}) et (x_{max}, y_{max}). Un dexel est défini de la manière suivante :

$$\text{dexel}(x, y, z_{min}, z_{max}, c_{min}, c_{max}, S) \text{ où}$$

- x et y sont des entiers qui déterminent la position du pixel ($x \in [x_{min}, x_{max}]$ et $y \in [y_{min}, y_{max}]$).
- z_{min} et z_{max} sont les valeurs en Z minimale et maximale du dexel. Ces valeurs sont aussi appelées $NearZ$ (pour la plus proche valeur en profondeur) et $FarZ$ (pour la plus lointaine valeur en profondeur).
- c_{min} et c_{max} correspondent aux couleurs du dexel respectivement en z_{min} et z_{max}.
- (x, y, z_{min}) et (x, y, z_{max}) sont les extrémités d'un segment *dans* le solide S.

Une liste ordonnée de dexels est ensuite associée à chaque pixel. Comme les dexels sont tous distincts et ne se superposent pas, cette liste correspond à un ensemble ordonné de dexels disjoints. On note ListeDexel(x, y, S) la liste dans le solide S associée au pixel de coordonnées (x, y).

$$\begin{aligned} ListeDexel(x, y, S) = [&dexel(x, y, z_{min_0}, z_{max_0}, c_{min_0}, c_{max_0}, S), \\ &\dots \\ &dexel(x, y, z_{min_i}, z_{max_i}, c_{min_i}, c_{max_i}, S), \\ &dexel(x, y, z_{min_{i+1}}, z_{max_{i+1}}, c_{min_{i+1}}, c_{max_{i+1}}, S), \\ &\dots \\ &dexel(x, y, z_{min_n}, z_{max_n}, c_{min_n}, c_{max_n}, S)] \end{aligned}$$

où

- $dexel(x, y, z_{min_i}, z_{max_i}, c_{min_i}, c_{max_i}, S)$ représente le ième dexel de la liste.
- $z_{max_{i-1}} < z_{min_i} < z_{max_i} < z_{min_{i+1}}$
- $0 \leq i \leq n - 1$
- $n + 1$ est le nombre de dexels de la liste du solide S.

Une représentation sous forme de dexels pour un solide S est une collection de listes de dexels construites pour une direction de vue fixée, elle est notée de la manière suivante :

$$\begin{aligned} D(S) = [&d : d = ListeDexel(x, y, S), \\ &\text{pour tout } x \in [x_{min}, x_{max}] \text{ et tout } y \in [y_{min}, y_{max}]] \end{aligned}$$

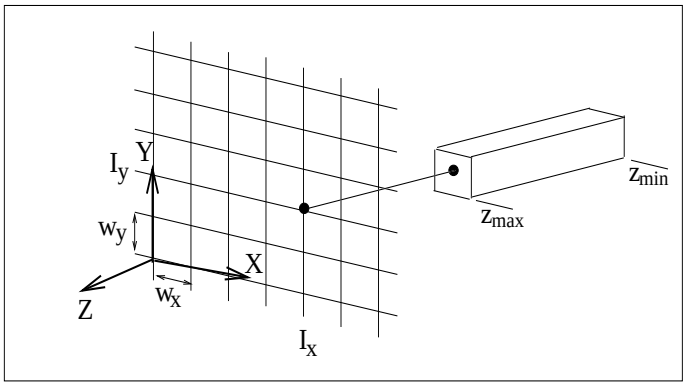

FIG. 2.3 – Un dexel du z-buffer étendu.

Le lieu des dexels est défini par une grille à deux dimensions dans le plan (x, y). Chaque point de la grille est repéré par une paire d'entiers (I_x, I_y). Les points sont uniformément répartis sur la grille selon l'axe des X et l'axe des Y, et sont régulièrement espacés par les distances w_x et w_y. Les coordonnées des points sur la grille sont $(I_x * w_x, I_y * w_y)$. Pour simplifier les opérations, on considère que $w_x = w_y = w = P_{ech}$ qui correspond au pas d'échantillonnage.
Chaque dexel est représenté par un solide rectangle localisé autour d'un point de la grille et s'étendant le long de l'axe des Z (figure 2.3, page 32). Les dimensions X et Y de chaque dexel sont ainsi fixées et la longueur du dexel dépend d'une paire de valeurs en Z ($NearZ$ et $FarZ$) qui représente la plus proche et la plus lointaine valeur en Z.
La résolution de la structure dexel dépend de la résolution de l'écran. La précision de la représentation de n'importe quel objet est déterminée par la taille w du dexel et par l'orientation de la direction de vue, ce qui fait de cette méthode une méthode approximative.

L'ensemble de ces dexels fournit alors assez d'informations pour afficher l'objet dans une direction donnée. En effet, une image de l'objet est obtenue si on affiche à l'écran la couleur correspondant au Z_{max} (ou $FarZ$) du dernier dexel (figure 2.2, page 30). Il est évident que ce dexel cache tous les autres dexels pour la direction donnée, néanmoins, les informations stockées dans les autres dexels sont essentielles pour modéliser l'objet.

2.1.3 Combinaison de deux solides

Nous avons vu précédemment que la représentation 3D d'un objet par la méthode du z-buffer étendu est composée de listes de dexels associées à chaque pixel de l'écran. Ainsi, l'application d'une opération booléenne régularisée (obr) entre deux solides A et B utilisant cette représentation revient à effectuer des opérations booléennes régularisées sur chacune des listes des dexels des objets A et B (figure 2.4, page 33) :

$$D(A) \ \ obr \ \ D(B) = [d : d = ListeDexel(x, y, A) \ \ obr \ \ ListeDexel(x, y, B),$$
$$\text{pour tout } x \in [x_{min}, x_{max}] \text{ et tout } y \in [y_{min}, y_{max}]]$$

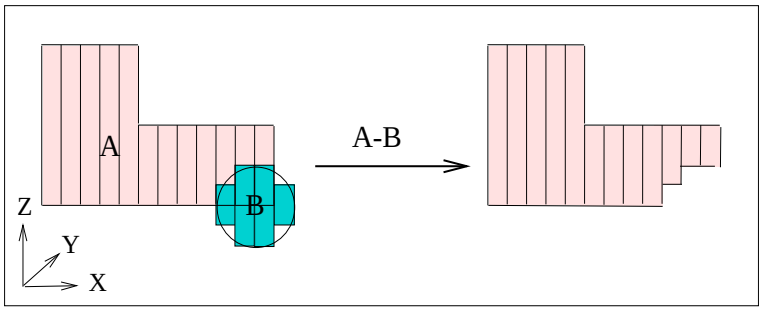

FIG. 2.4 – Combinaison de deux solides dans le plan (X,Z)

Comme l'opération booléenne régularisée est effectuée sur une liste de dexels associée à un même pixel, x et y sont invariants au cours de cette opération. Les seules variables susceptibles d'être modifiées sont les Z_{min_i} et Z_{max_i} de chaque dexel. La technique du z-buffer étendu simplifie l'utilisation des opérations booléennes régularisées en ramenant les opérations booléennes régularisées entre dexels dans un espace euclidien à trois dimensions (E^3) à des opérations booléennes régularisées sur des intervalles dans un espace euclidien à une dimension (E^1).

Une opération booléenne entre deux listes de dexels peut se décomposer en deux étapes. Premièrement, les nouveaux dexels sont obtenus à partir d'une opération booléenne entre deux dexels, un de chaque liste. Puis, une nouvelle liste de dexels est construite avec les nouveaux dexels (il faut préalablement s'assurer que les dexels de la nouvelle liste sont bien tous disjoints). En pratique ces deux étapes se font simultanément puisque nous utilisons l'ordre naturel des dexels dans les listes.

2.1.3.1 Opérations booléeennes entre deux dexels

Soit une opération booléenne entre deux dexels a et b. Le dexel a est un élément d'une liste de dexels associée à un objet A et le dexel b est un élément d'une liste

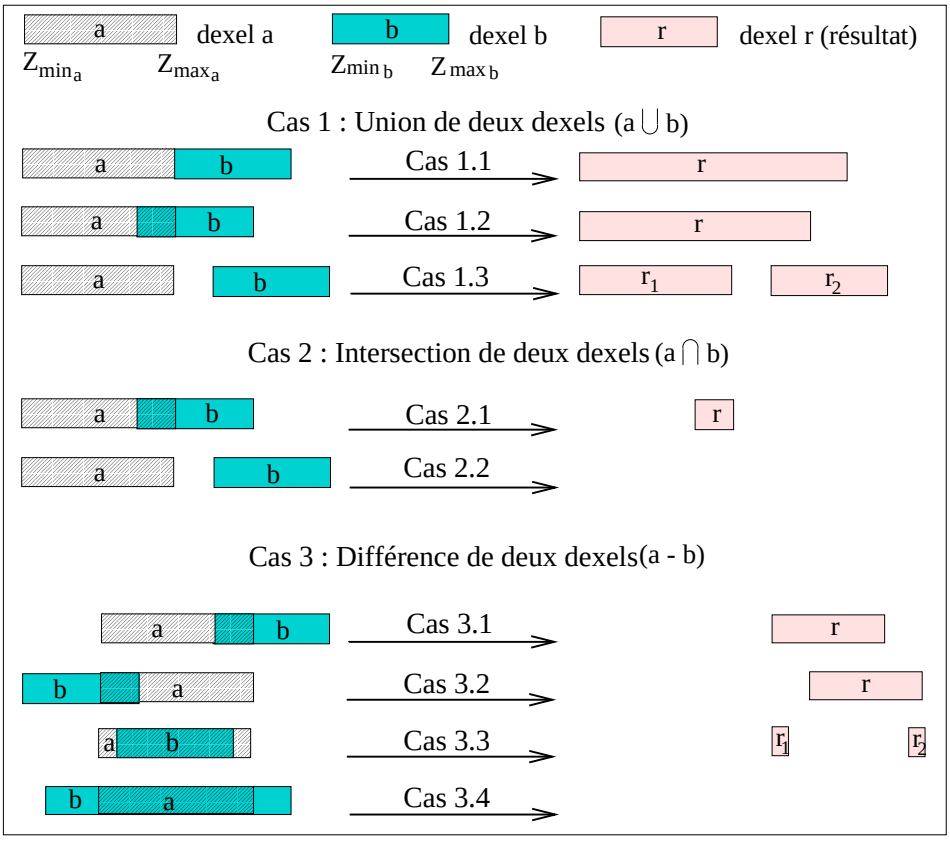

FIG. 2.5 – Différentes opérations booléennes entre deux dexels.

de dexels associée à un objet B. Notons les valeurs minimale et maximale en Z de chaque dexel par Z_{min} et Z_{max} [26].

Pour une opération booléenne régularisée de type **union** (cas 1 de la figure 2.5, page 34), la valeur Z_{min} du nouveau dexel correspond à la valeur minimale des Z_{min} de a et de b. La valeur Z_{max} du nouveau dexel correspond à la valeur maximale des Z_{max} de a et de b.

Pour une opération booléenne régularisée de type **intersection** (cas 2 de la figure 2.5, page 34), seule la partie commune entre les deux dexels est conservée. La valeur Z_{min} du nouveau dexel correspond à la valeur maximale des Z_{min} de a et de b. La valeur Z_{max} du nouveau dexel correspond à la valeur minimale des Z_{max} de a et de b.

Pour une opération booléenne régularisée de type **différence** entre deux dexels (cas 3 de la figure 2.5, page 34), aucun, un ou deux dexels peuvent être créés, suivant le positionnement relatif des dexels comparés. La troisième partie de la figure 2.5

montre les quatre cas possibles qui apparaissent lors de la différence entre deux dexels a et b, le dexel b est soustrait au dexel a.

- Le cas 3.1 de la figure 2.5 représente la *réduction du dexel a par l'arrière*, c'est à dire la modification de son Z_{max}. Ainsi, si $Z_{min_a} < Z_{min_b}$ et $Z_{max_a} < Z_{max_b}$, les valeurs minimale et maximale du nouveau dexel sont alors :
  $Z_{min} = Z_{min_a}$
  $Z_{max} = \min(Z_{max_a}, Z_{min_b})$
- Le cas 3.2 de la figure 2.5 représente la *réduction du dexel a par l'avant*, c'est à dire la modification de son Z_{min}. Ainsi, si $Z_{min_a} > Z_{min_b}$ et $Z_{max_b} < Z_{max_a}$, les valeurs minimale et maximale du nouveau dexel sont alors :
  $Z_{min} = \max(Z_{min_a}, Z_{max_b})$
  $Z_{max} = Z_{max_a}$
- Le cas 3.3 de la figure 2.5 entraîne l'*apparition d'un nouveau dexel* à partir du dexel a. Ainsi, si $Z_{min_a} < Z_{min_b} < Z_{max_b} < Z_{max_a}$, deux dexels r_1 et r_2 résultent de la différence entre le dexel a et le dexel b. Les valeurs minimale et maximale de ces nouveaux dexels sont :
  $Z_{min_1} = Z_{min_a}$
  $Z_{max_1} = Z_{min_b}$
  $Z_{min_2} = Z_{max_b}$
  $Z_{max_2} = Z_{max_a}$
- Le cas 3.4 de la figure 2.5 représente la *suppression d'un dexel*. Ainsi, si $Z_{min_b} < Z_{min_a} < Z_{max_a} < Z_{max_b}$, alors un dexel vide est obtenu lorsque le dexel b est soustrait au dexel a.

Grâce à l'utilisation d'opérations booléennes régularisées, un dexel dont les valeurs minimale et maximale ($Z_{min} = Z_{max}$) sont égales, est considéré comme un dexel nul. Il est alors immédiatement supprimé de la liste de dexels.

2.1.3.2 Opérations booléeennes entre une liste de dexels : Mise à jour de la liste

Le paragraphe précédent s'intéresse aux opérations booléennes entre deux dexels à x et y constants. Dans une opération booléenne entre deux objets, les listes des deux objets sont comparées et modifiées afin d'obtenir de nouvelles listes représentant le nouvel objet. Soient deux listes de dexels, une provenant d'un objet A, l'autre provenant d'un objet B. Un dexel de l'objet A (dexel a) est extrait de la liste, il devient le dexel courant, et il est mis à jour en le comparant un à un avec les dexels de l'objet B. Deux cas peuvent apparaître :

- **cas où les dexels se superposent**
  Lorsque les valeurs minimale et maximale de deux dexels se superposent alors

un, deux ou aucun dexel est généré selon le cas envisagé (cas 3.1, 3.2, 3.3, 3.4 de la figure 2.5, page 34). Si un dexel est créé, alors il devient le dexel courant. Si deux dexels sont créés, le premier dexel (dexel r_1 de la figure 2.5, page 34) est intégré à la liste et le second dexel (dexel r_2 de la figure 2.5, page 34) devient le dexel courant. Dans les deux cas, le dexel suivant de la liste correspondant à l'objet B est *extrait* et une procédure de comparaison entre ce dexel et le dexel courant est réengagée. Si aucun dexel n'est généré, un dexel suivant est extrait de l'objet B est extrait et la procédure est également répétée.

* **cas où les dexels ne se superposent pas**
  Lorsque les valeurs minimale et maximale de deux dexels ne se superposent pas alors deux cas sont possibles : soit le dexel b est devant le dexel courant a soit le dexel b est derrière le dexel courant a. Dans les deux cas précédents, le dexel b n'entraîne aucune modification pour les opérations booléennes de différence ou d'intersection. Par contre, dans une opération booléenne d'union entre le dexel a et le dexel b, le dexel b est ajouté à la liste de dexels, un dexel b suivant est extrait et la procédure est répétée.

2.2 Présentation de la technique du z-buffer étendu dans le cadre de la simulation d'usinage

2.2.1 Le z-buffer étendu et la simulation d'usinage

Van Hook a mis au point une méthode qui permet de visualiser l'usinage d'un solide par un outil tout au long de sa trajectoire. Cette visualisation se fait dans l'espace image en utilisant des opérateurs booléens réguliers de différence.
La simulation d'usinage consiste à modéliser un brut, puis à visualiser l'évolution de ce solide. Grâce à la méthode de Van Hook, la pièce est modélisée et visualisée directement tout au long de la simulation d'usinage.
Dans la technique du z-buffer étendu, le vecteur en profondeur du système de coordonnées des dexels (axe des Z) est aligné avec le vecteur de vue. La représentation 3D d'un solide sous forme de dexels peut être directement affichée puisque la couleur de la face lointaine (Z_{max} ou $FarZ$) du dernier dexel de chaque liste correspond à la couleur visible pour le pixel associée à cette liste (figure 2.2, page 30). Si aucun dexel n'est présent dans la liste, une couleur par défaut (la couleur du fond) est affichée.
Dans la simulation d'usinage, seule la couleur des faces lointaines des dexels joue un rôle dans la visualisation du modèle, nous définissons un dexel du solide S de la simulation de la manière suivante :

$$\textbf{dexel}(x, y, NearZ, FarZ, C, S)$$

où

- x et y sont des entiers qui déterminent la position du pixel dans la fenêtre graphique de taille définie par ses pixels extrêmes (x_{min}, y_{min}) et (x_{max}, y_{max}).
- $NearZ$ et $FarZ$ sont les valeurs en Z minimale et maximale du dexel. La matière à l'intérieur du solide est alors bornée par le $NearZ$ et par le $FarZ$.
- C correspond à la couleur du dexel. Cette couleur correspond à la face $FarZ$ du dexel.

Comme les dexels sont alignés avec le vecteur de vue, seules les vues de face et de derrière du modèle peuvent être affichées. Pour afficher un objet basé sur une représentation 3D par dexels dans d'autres directions que la direction de vue avec la méthode de Van Hook, il faut reconstruire entièrement la représentation dans la direction de vue souhaitée, ce qui entraîne des limites quant à l'utilisation de cette technique. Nous reviendrons sur ces limites dans les paragraphes suivants.

Du point de vue algorithmique, un dexel est un enregistrement qui contient des

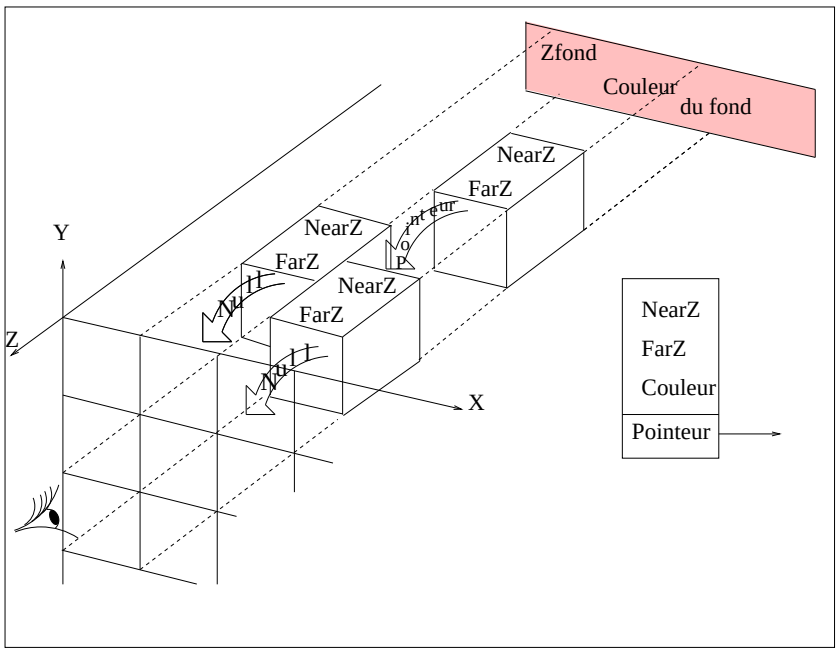

FIG. 2.6 – Structure de donnée du z-buffer étendu.

informations sur sa plus proche valeur en Z ($NearZ$), sa plus lointaine valeur en Z ($FarZ$), sa couleur et un lien sur le dexel suivant que nous représenterons sur la figure 2.6 de la page 37 comme un élément d'une liste chaînée. La comparaison de deux z-buffers étendus se traduit donc par des opérations sur des listes ordonnées.

2.2.2 Modélisation du brut et de l'outil

En simulation d'usinage, seul le volume géométrique de l'outil ou le volume balayé par l'outil est soustrait au brut. Dans la technique du z-buffer étendu, l'outil est représenté par son volume décomposé en dexels. Le brut est également représenté par un z-buffer étendu, ce qui permet de comparer pour chaque position de l'outil, les dexels de l'outil aux dexels du brut. Nous travaillons donc sur la structure de données du z-buffer étendu du brut car nous considérons que seul l'outil enlève de la matière au brut. L'opération booléenne utilisée pour comparer ces deux z-buffers étendus est une opération booléenne régularisée de différence. Le z-buffer étendu s'exécute donc en $O(n)$ où n est le nombre de positions de l'outil puisque la technique du z-buffer étendu nous permet de ramener les opérations booléennes de différence dans un espace à 3 dimensions à des intersections d'intervalles dans un espace euclidien à une dimension (E^1) [24].

2.2.3 Opération booléenne de différence entre deux dexels.

L'usinage est équivalent à une opération booléenne régularisée de différence entre les dexels de l'outil et les dexels du brut. L'opération de coupe est simplifiée en comparant les valeurs maximales et minimales des deux dexels. La figure 2.7, page 2.7 répertorie les 5 cas qui peuvent apparaître lors de la différence de deux dexels :

- cas 1 : le dexel de l'outil emporte la partie avant du dexel de la pièce.
- cas 2 : le dexel de l'outil emporte la partie arrière du dexel de la pièce.
- cas 3 : le dexel de l'outil coupe le dexel de la pièce en deux parties.
- cas 4 : le dexel de l'outil supprime complètement le dexel de la pièce.
- cas 5 : le dexel de l'outil ne recouvre pas le dexel de la pièce. Aucune intersection n'est possible, le dexel de la pièce n'est pas modifié.

2.2.4 A propos de l'outil

2.2.4.1 Dépôt de la couleur de l'outil

Dans la simulation d'usinage, le brut est considéré comme un solide *positif* alors que l'outil est considéré comme un solide *négatif*, puisque l'outil est soustrait au brut. La différence entre la structure du brut et de l'outil tient dans la couleur du dexel. Pour le brut, la couleur du dexel est celle de la surface la plus proche de l'oeil de l'utilisateur (face $FarZ$ d'après la figure 2.6, page 37, tandis que pour l'outil, la couleur mémorisée sera celle de la surface la plus lointaine de l'oeil de l'utilisateur (face $NearZ$ des dexels de l'outil) puisque lorsque l'outil est soustrait au brut, la surface la plus lointaine de l'outil (par rapport à l'utilisateur) devient la surface du

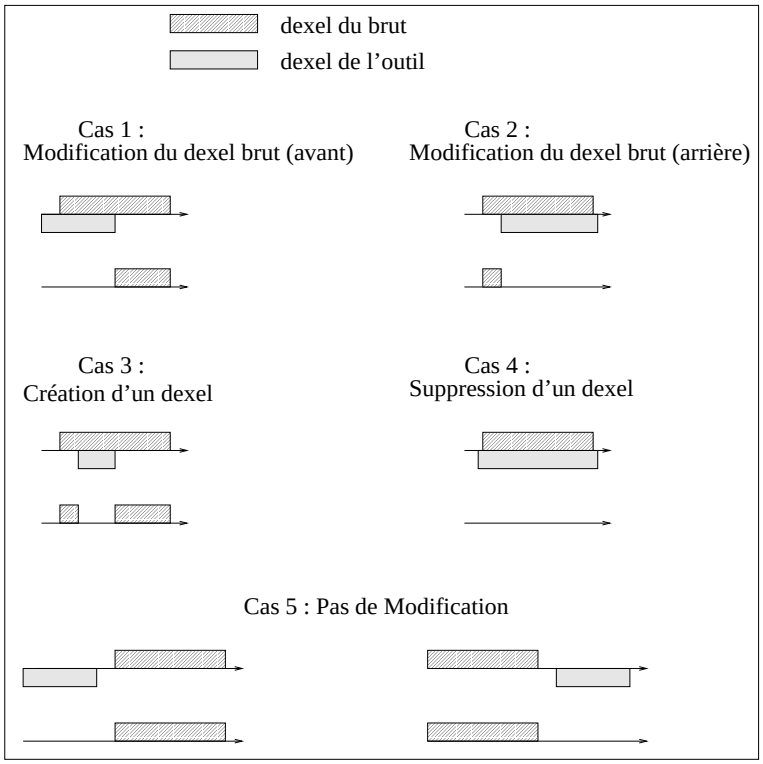

FIG. 2.7 – Opération booléenne régularisée de différence entre un dexel du brut et un dexel de l'outil.

brut la plus proche (par rapport à l'utilisateur) et donc l'outil dépose la couleur de sa plus lointaine face (face $NearZ$) sur le brut.

2.2.4.2 Problème dû à la discrétisation de l'outil

Dans la technique du z-buffer étendu appliquée à la simulation d'usinage, l'outil doit être échantillonné pour être décomposé en dexels. A chaque position élémentaire de l'outil, les dexels du brut sont comparés aux dexels de l'outil. La figure 2.8, page 40 montre comment dans le plan (X,Z) l'outil, initialement assimilé à un cercle, est échantillonné. En fonction du pas d'échantillonnage, le contour de l'outil devient plus ou moins grossier dans le plan (X,Z). Cette *perte d'information* sur les contours de l'outil se traduira par une perte d'information dans la simulation d'usinage, plus le pas d'échantillonnage est petit et plus la perte d'information est minime.

Le choix du pas d'échantillonnage P_{ech}, qui détermine la taille de la face de chaque dexel, influe directement sur la précision de la représentation. Comme le montre la figure 2.9 de la page 40, le pas d'échantillonnage peut être calculé à partir du rayon

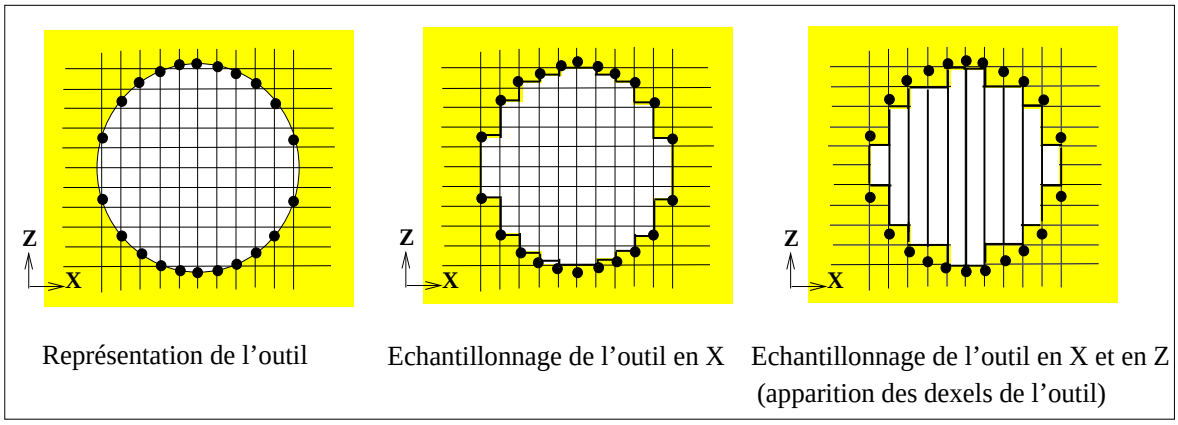

FIG. 2.8 – Discrétisation de l'outil

R de l'outil et d'une tolérance d'erreur E fixée par l'utilisateur :

$$P_{ech} = 2 \times R - 2\sqrt{R^2 - E^2}$$

A partir de cette équation, il est possible de calculer P_{ech} afin de visualiser au mieux l'outil dans l'espace image. Le rapport $\frac{E}{R}$ caractérise la précision relative de l'outil dans la représentation [27].

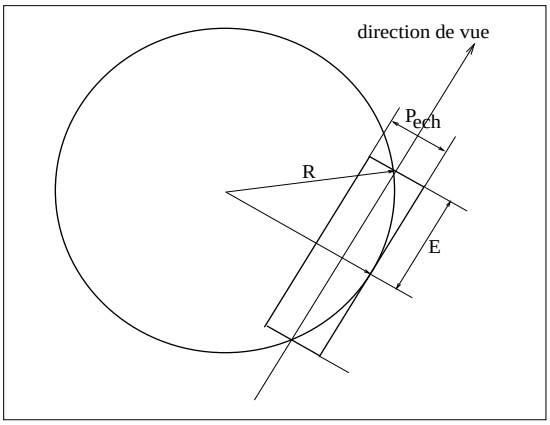

FIG. 2.9 – Calcul du pas d'échantillonnage à partir du rayon de l'outil

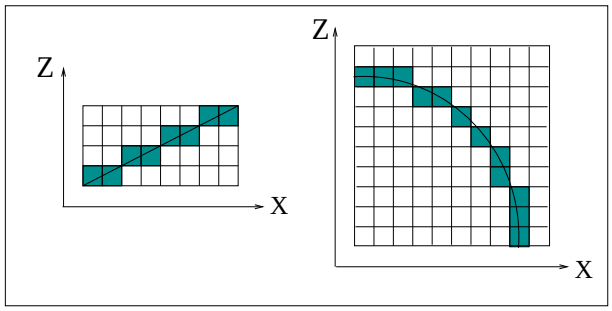

FIG. 2.10 – Segment de droite et cercle par l'algorithme de Bresenham

2.2.5 A propos des trajectoires de l'outil

2.2.5.1 Mise en place des trajectoires de l'outil

La simulation consiste à suivre le mouvement de l'outil en déterminant le volume de matière enlevé au brut par l'outil au cours d'une trajectoire. Pour obtenir une simulation réaliste, nous avons choisi un enlèvement *continu* de matière. Nous devons discrétiser la trajectoire en *positions élémentaires* qui représentent les différentes positions du centre de l'outil.
Les différentes *étapes* de la simulation d'usinage sont définies par les positions élémentaires successives de l'outil.
Les positions élémentaires sont obtenues à partir d'une généralisation de l'algorithme de Bresenham dans un espace en 3D (3D DDA *Digital Differential Analyser*) [28](figure 2.10, page 41). Ces positions seront d'autant plus nombreuses que la résolution choisie sera grande.

2.2.5.2 Problème dû à la discrétisation des trajectoires de l'outil

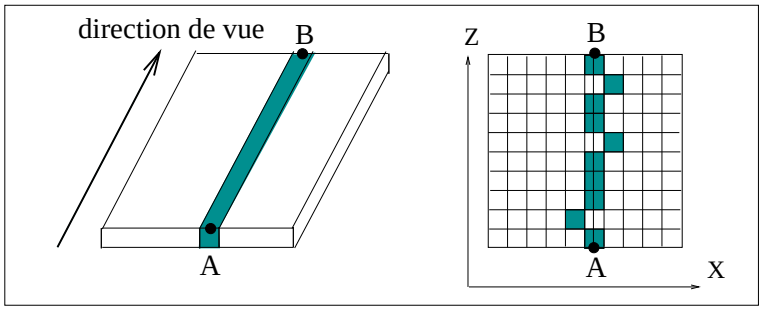

FIG. 2.11 – Discrétisation d'une trajectoire dans la direction de vue

La discrétisation d'une trajectoire outil en positions élémentaires par l'algorithme de Bresenham peut entraîner des imprécisions dans la représentation de l'usinage, notamment si la trajectoire de l'outil est parallèle à la direction de vue comme sur la figure 2.11 de la page 41.

2.3 Améliorations et variantes du z-buffer étendu

- **Améliorations du z-buffer étendu :**
  Le z-buffer étendu permet d'afficher un solide dans une orientation donnée pour un point de vue fixé. A la fin de la simulation, le point de vue reste fixe et la pièce ne peut pas être visualisée dans une autre orientation. Si nous désirons changer de point de vue, il faut réexécuter une nouvelle simulation. En 1994, Huang et Oliver [29] ont proposé une extension de la technique du z-buffer étendu de Van Hook pour pallier ce problème. Pour éviter que la construction de la pièce sous forme de dexels ne soit limitée qu'à une seule direction de vue, ils ont développé une méthode d'affichage de contours et ont introduit un système de coordonnées propre au dexel et indépendant du point de vue. Ainsi, l'ensemble des dexels est converti en un ensemble de contours linéaires, qui peuvent être visualisés depuis n'importe quel point de vue. Une estimation de l'erreur de la technique du z-buffer étendu de Van Hook est alors envisageable.
  En 1994, Hui [26] a développé une méthode pour accélerer la simulation en calculant directement les volumes balayés dans l'espace image.

- **Variantes du z-buffer étendu :**
  En 1991, Saito et Takahashi [30] ont proposé une variante de la technique du z-buffer étendu : la technique du G-buffer (ou buffer géométrique) également utilisé pour la simulation d'usinage. Le G-buffer contient des informations sur la profondeur comme pour le z-buffer étendu, mais aussi sur le vecteur normal à la primitive géométrique au point d'intersection, ce qui permet d'appliquer à la scène (pièce) des techniques de rendu réaliste (comme l'ombre, la texture...) Le G-buffer a longtemps été utilisé pour un usinage en 3 axes, grâce à Chiou et Lee il peut maintenant être utilisé indirectement pour un usinage en 5 axes [31].
  En 1997, Glaeser et Gröller [32] ont utilisé des techniques de géométrie différentielle pour générer le volume balayé par l'outil en mouvement. Les calculs d'intersections ont été réalisés dans une structure de données appelée Γ-buffer.
  En 1995, Menon a proposé une variante du z-buffer étendu, appelée Ray-

Representation (ray-reps) [33, 34]. Une ray-rep n'est autre qu'une méthode plus générale pour représenter un solide. Dans le procédé du lancer de rayon

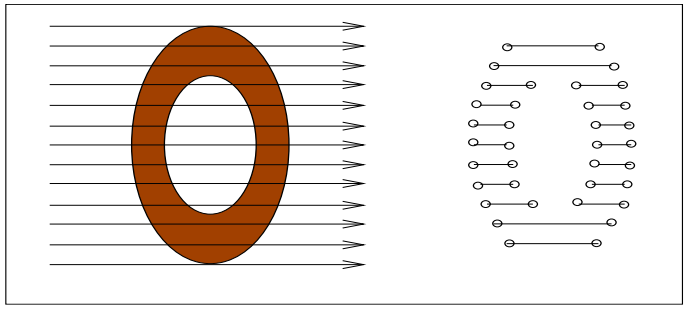

FIG. 2.12 – Ray Representation

(ray-casting) [16], les rayons sont lancés à travers des solides et le passage de la lumière au travers de cet objet est étudié. Dans une ray-rep, une grille de rayons parallèles est lancée au travers d'un solide. L'intersection entre un rayon et l'objet solide en 3 dimensions peut être nulle, ou correspondre à un ou plusieurs segments. Ces segments sont alors délimités par les points d'entrée et de sortie du rayon dans l'objet. Ces segments y compris leurs extrémités constituent la Ray Representation qui est représentée sur la figure 2.12, page 43. A partir de cette représentation, toutes les opérations booléennes peuvent être calculées en effectuant un calcul parallèle à l'aide d'un *raycasting engine*. La ray-rep n'est vraiment efficace que si un calcul parallèle est implémenté. Il est important de remarquer que la Ray Representation est, dans un premier temps, une méthode de modélisation d'un solide, pour visualiser cette modélisation, il faut dans un second temps, utiliser des techniques d'affichage. La méthode du z-buffer étendu permet quant à elle de modéliser et de visualiser un solide dans le même temps. C'est pour cette raison que la Ray Representation ne convient pas pour notre application. Cependant, il est intéressant de remarquer que cette méthode trouve d'autres applications, notamment en biologie moléculaire pour déterminer par exemple l'espace vide entre les atomes [35].

Chapitre 3

Les traces : De nouvelles fonctionnalités pour le z-buffer étendu

Par le biais de la simulation, l'opérateur souhaite découvrir le parcours de l'outil et les défauts éventuels qui pourraient survenir lors d'un usinage réel, il veut donc savoir où, quand et comment ces défauts apparaissent, pour pouvoir mesurer leur impact et trouver des solutions. Pour cela, il est souhaitable de réaliser une simulation interactive où l'opérateur visualise non seulement la pièce dans son état final, mais aussi les différentes phases de l'usinage. Pour suivre l'évolution du brut et visualiser un moment précis de l'usinage, deux possibilités s'offrent à nous. La première solution consiste à réexécuter la simulation du début jusqu'au moment souhaité, ce qui est assez coûteux en temps. La seconde solution consiste à *recoller de la matière* à l'aide de copeaux virtuels conservés en mémoire au cours de la simulation.
La méthode du z-buffer étendu de Van Hook telle qu'elle a été définie précédemment ne possède pas de mémoire et ne peut donc pas revenir à un moment précis de la simulation. Dans un premier temps, seule la première solution est envisageable, puisque la méthode du z-buffer étendu modélise et affiche chaque étape de la simulation, mais ne garde aucune trace de l'historique de cette simulation. En effet, lorsqu'un dexel est supprimé du z-buffer étendu, cette suppression est définitive et les informations contenues dans ce dexel ne sont plus désormais accessibles en mémoire. De même lorsqu'un dexel est modifié, son champ NearZ ou FarZ est modifié, mais l'ancienne valeur n'est plus directement accessible. Un retour en arrière dans la simulation paraît difficile à mettre en oeuvre, puisque dans ces conditions certaines données devront être recalculées, et ceci sera coûteux en temps.
Cependant, pour visualiser rapidement un moment précis de la simulation, nous nous sommes intéressés à la seconde solution qui consiste à conserver en mémoire les différentes phases de l'usinage. Pour cela, il a été nécessaire de définir de nouvelles fonctionnalités au z-buffer étendu que nous avons appelées *traces* et qui permettent de sauvegarder au cours de la simulation des données essentielles, pour rejouer ensuite une scène quelconque de cette simulation [13].

3.1 Vers une animation pour la technique du z-buffer étendu

3.1.1 Liste de dexels d'un solide S dans une position statique

Dans le chapitre précédent, nous avons défini un dexel d'un solide S de la manière suivante :

$$\textbf{dexel}(x, y, NearZ, FarZ, C, S)$$

Un dexel ainsi défini permet de représenter le solide S *en position statique*. Pour représenter la liste de dexels associées au pixel de coordonnées (x, y) de laquelle est extraite le dexel ci-dessus, nous utilisons une représentation réduite du dexel. En effet, à partir du moment où nous avons désigné une liste déterminée de dexels, les coordonnées x et y sont identiques pour tous les dexels de cette liste, nous choisissons de ne pas les faire apparaître dans la notation réduite du dexel, car elles seront stockées implicitement dans un tableau :

$$\textbf{dexel\_réduit}(NearZ, FarZ, C)$$

La liste ordonnée de dexels du solide S *en position statique* associée au pixel de coordonnées (x, y) est définie de la manière suivante :

$$\begin{aligned}\textbf{ListeDexel}(x, y, S) = [&\text{dexel\_réduit}(NearZ_0, FarZ_0, C_0),\\ &\dots\\ &\text{dexel\_réduit}(NearZ_j, FarZ_j, C_j),\\ &\text{dexel\_réduit}(NearZ_{j+1}, FarZ_{j+1}, C_{j+1}),\\ &\dots\\ &\text{dexel\_réduit}(NearZ_n, FarZ_n, C_n)]\end{aligned}$$

où

- dexel_réduit$(NearZ_j, FarZ_j, C_j)$ représente le $(j+1)$ème dexel de la liste.
- $NearZ_j < FarZ_j < NearZ_{j+1} < FarZ_{j+1}$

D'autre part, pour une position statique du solide, le dexel réduit peut aussi être représenté par le couple de ses deux points extrêmes (P_k,P_{k+1}), de coordonnées en Z respectives $NearZ$ et $FarZ$ et ayant tous deux comme attribut commun la couleur C du dexel (figure 3.1, page 47).

$$\textbf{dexel\_réduit}(NearZ, FarZ, C) = (P_{2j}(NearZ, C), P_{2j+1}(FarZ, C))$$

La liste ordonnée des dexels du solide S *en position statique* associée au pixel de coordonnées (x, y) peut aussi être modélisée par une suite de points (figure 3.1, page 47) tels que :

$$\begin{aligned}\textbf{ListeDexel}(x, y, S) = [&P_0(NearZ_0, C_0), P_1(FarZ_0, C_0),\\ &\dots\\ &P_{2j}(NearZ_j, C_j), P_{2j+1}(FarZ_j, C_j),\\ &P_{2j+2}(NearZ_{j+1}, C_{j+1}), P_{2j+3}(FarZ_{j+1}, C_{j+1}),\\ &\dots\\ &P_{2n}(NearZ_n, C_n), P_{2n+1}(FarZ_n, C_n)]\end{aligned}$$

où $P_{2j}(NearZ_j, C)$ et $P_{2j+1}(FarZ_j, C)$ sont les points extrêmes du $(j+1)$ème dexel.

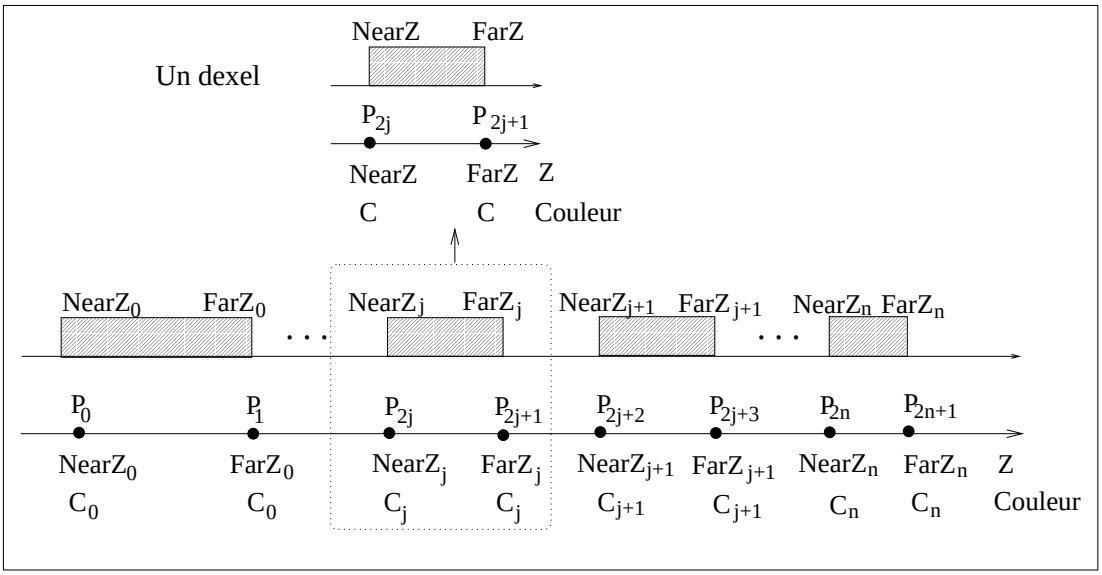

FIG. 3.1 – Un dexel et une liste de dexels associée à un pixel de coordonnées (x,y)

3.1.2 Evolution d'une liste de dexels d'un solide S au cours d'une simulation

En 1997, Dubreuil et Lienhardt [36] proposent "de ne plus considérer l'animation comme étant l'application d'un mouvement ou d'une déformation sur un objet tridimensionnel (3D), mais [directement] comme un objet 4D". Pour cela, ils introduisent une dimension temporelle qu'ils considèrent à l'égal des dimensions spatiales. Dans le cas du z-buffer étendu, l'objet 3D est réduit au préalable à un objet 1D, mais il est intéressant de conserver ce principe et d'introduire au moins une nouvelle dimension temporelle pour modéliser l'animation.

Dans la figure 3.2 de la page 49, nous avons représenté un exemple de séquence d'usinage depuis son étape initiale 0 jusqu'à son étape finale 7. Pour chaque étape de la séquence, un dexel de l'outil est soustrait au(x) dexel(s) du brut. Nous visualisons donc l'évolution d'une liste de dexels du brut associé à un pixel particulier de l'écran (x et y sont fixes au cours de la séquence d'usinage). Pour des questions de lisibilité, nous avons représenté dans la colonne de gauche, l'outil en dessous du brut. En représentant cette séquence d'usinage étape par étape, nous avons réalisé une coupe de l'objet spatio-temporel (la liste de dexels dans le temps) et nous pouvons ainsi visualiser l'animation subie par la liste de dexels. Cette liste de dexels devient une liste *dynamique* qui évolue au cours du temps et qui peut être définie à partir de

la liste de dexels *statique* précédemment définie à laquelle on associe une variable temps qui représente les différentes étapes de la simulation.
Ainsi, nous pouvons représenter la séquence d'usinage (c'est à dire la liste de dexels dynamique) comme le produit cartésien d'un ensemble D_1 fini ayant comme unique élément la liste statique de dexels et d'un ensemble D_2 fini ayant comme éléments les différentes étapes d'usinage [1] :

$$\textbf{ListeDexelsDynamique}(x, y, S) = \textbf{SequenceUsinage}(x, y, S) = D_1 \times D_2$$

Par exemple, la liste suivante correspond à la liste dynamique de dexels de la séquence d'usinage de la figure 3.2 de la page 49 :

$$\begin{array}{ll} \textbf{ListeDexelsDynamique}(x, y, S) = < & (\text{ListeDexelsStatique}(x, y, S),\ 0), \\ \text{ou } \textbf{SequenceUsinage}(x, y, S) & (\text{ListeDexelsStatique}(x, y, S),\ 1), \\ & (\text{ListeDexelsStatique}(x, y, S),\ 2), \\ & (\text{ListeDexelsStatique}(x, y, S),\ 3), \\ & (\text{ListeDexelsStatique}(x, y, S),\ 4), \\ & (\text{ListeDexelsStatique}(x, y, S),\ 5), \\ & (\text{ListeDexelsStatique}(x, y, S),\ 6), \\ & (\text{ListeDexelsStatique}(x, y, S),\ 7) > \end{array}$$

L'objet de base sur lequel nous travaillons est le dexel. Il faut donc répercuter les modifications subies par la liste de dexels au cours de la simulation, sur la structure même des dexels. Nous avons donc repris dans la seconde partie de la figure 3.2 de la page 49, l'évolution de la séquence d'usinage en ne représentant que les points extrêmes des dexels. Au cours de la simulation de l'étape 0 à l'étape 7, de nouveaux dexels apparaissent, d'autres évoluent (ils sont rabotés par l'avant, par l'arrière) ou sont même supprimés. Les différentes opérations possibles pour modifier la liste de dexels sont reprises en détail dans la figure 3.3 de la page 50. Cela se traduit

[1] Un *domaine* dans un modèle de données est défini comme l'ensemble homogène des valeurs qui peuvent être prises par une propriété d'un objet. Par exemple, dans notre application, le domaine des entiers de 0 à 32 767 constitue l'ensemble des valeurs qui peuvent être prises par la coordonnée NearZ (propriété) d'un dexel (objet considéré dans l'exemple).

Le *produit cartésien* de deux domaines D_1 et D_2 est l'ensemble des couples $< v_1, v_2 >$ tels que v_1 appartienne à D_1 et v_2 appartienne à D_2. Soit par exemple, les domaines $D_1 = \{NearZ, FarZ\}$ et $D_2 = \{C_1, C_2, C_3\}$, le produit cartésien $D_1 \times D_2 = \{< NearZ, C_1 >, < NearZ, C_2 >, < NearZ, C_3 >, < FarZ, C_1 >, < FarZ, C_2 >, < FarZ, C_3 >\}$

Le produit cartésien de n domaines D_i est l'ensemble des *n-uplets* ou *tuples d'ordre n* $< v_1, v_2, ..., v_n >$ tels que, $v_i \in D_i$ pour $1 \leq i \leq n$

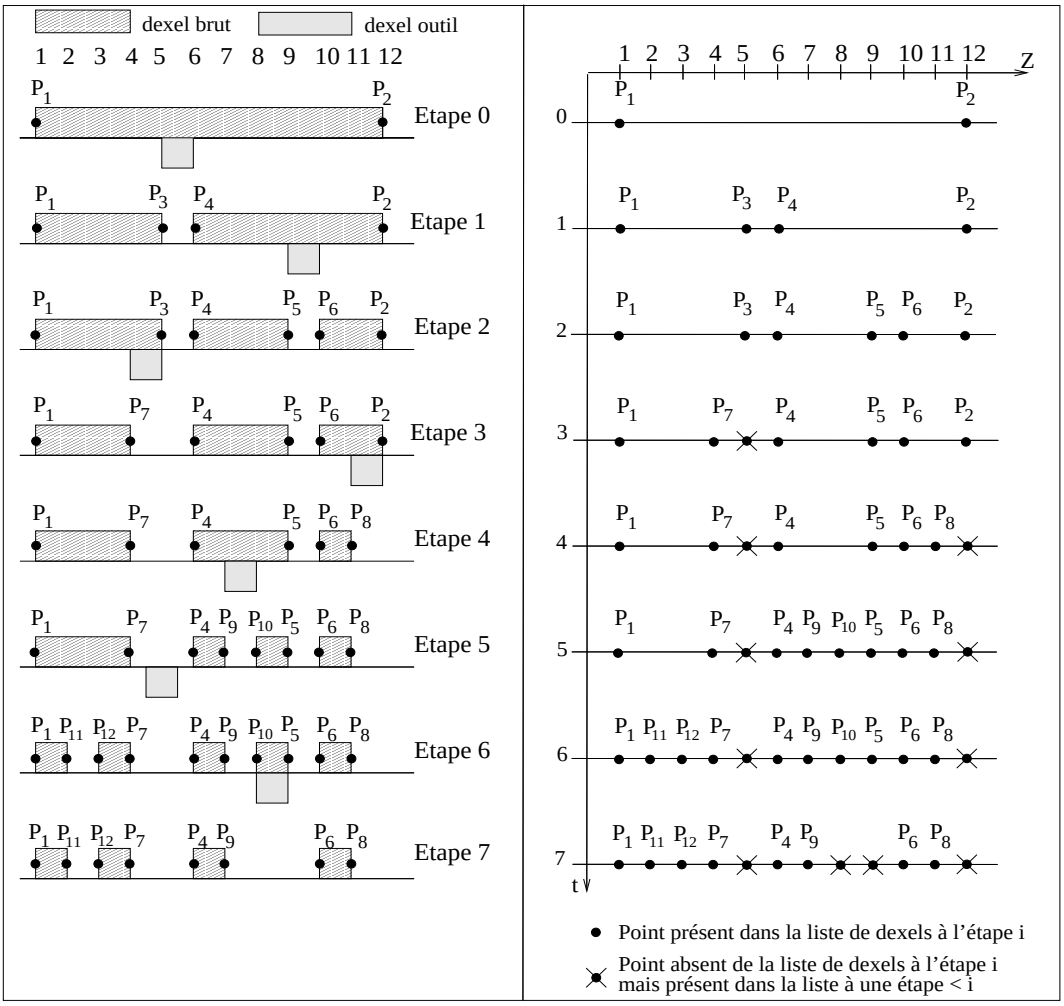

FIG. 3.2 – L'animation de la simulation au travers d'une séquence d'usinage

dans la liste de dexels par la création ou la disparition de points. Lorsqu'un point disparaît de la simulation, il est barré par une croix sur la figure précédente. Il est donc nécessaire d'introduire deux nouvelles dimensions temporelles pour *dater* ces modifications. Ces nouvelles données mémorisent l'instant de création (t_{crea}) et l'instant de disparition (t_{disp}) d'un point de la liste.
Chaque élément de la liste est maintenant caractérisé par trois coordonnées qui permettent de déterminer l'évolution de cet élément (et par conséquent du dexel associé) dans la simulation : une coordonnée en Z, et deux coordonnées temporelles (t_{crea} et t_{disp}). Pour chaque élément de la liste, il faut également ajouter la couleur aux dimensions précédentes. Nous avons donc transformé une liste dynamique de

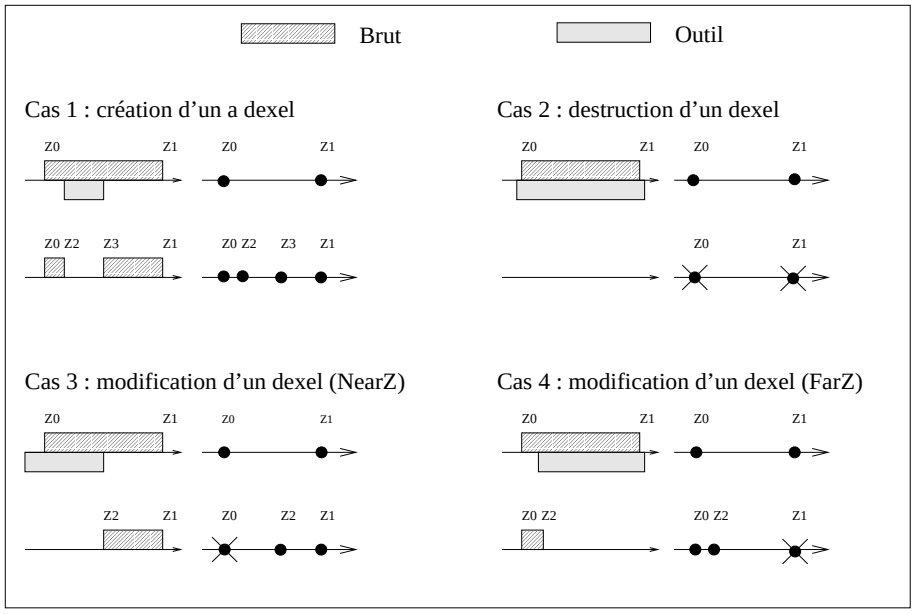

FIG. 3.3 – Modification de la liste de dexels en fonction de l'opération effectuée

dexels en une liste d'éléments composés de quatre coordonnées $(z, t_{crea}, t_{disp}, c)$, dont les deux coordonnées temporelles sont à l'origine de l'animation. Chaque élément est unique. L'ensemble des éléments de la liste peut être représenté par un sous-ensemble du produit cartésien suivant : $Z \times C \times T \times T$
où

- Z est le domaine des coordonnées en z.
- C est le domaine des couleurs.
- T est le domaine des étapes de la simulation.

Nous parlons de sous-ensemble du produit cartésien, puisque pour une simulation $t_{crea} < t_{disp}$.

Les éléments de ce produit cartésien seront désormais les éléments de base des traces. Nous allons voir dans les paragraphes suivants quelles coordonnées utiliser et mettre à jour pour créer des traces qui permettent d'obtenir de manière plus ou moins complète un moment précis de la simulation.

3.2 Amélioration de la technique du z-buffer étendu par les traces

Ainsi, nous définissons de nouvelles fonctionnalités pour le z-buffer, les *traces* qui mémorisent l'historique du z-buffer étendu au cours de la simulation. La trace complète (trace en Z ou z-trace) et la trace en temps (t-trace) permettent de reconstruire le modèle géométrique du z-buffer étendu à un moment quelconque de la simulation et/ou d'afficher une vue de ce modèle géométrique à l'écran. La z-trace et la t-trace sont indépendantes. Elles sont construites pendant le simulation, et mémorisent au fur et à mesure l'évolution du z-buffer étendu.

3.2.1 Trace en Z ou trace complète

Pour une étape donnée, la trace en Z permet de recréer une scène qui sera identique à la scène produite lors de la simulation d'usinage. Les informations de la trace seront définies lors de l'actualisation du z-buffer étendu au cours de la simulation. Nous allons définir l'algorithme utilisant ces informations.

3.2.1.1 Caractéristiques de la trace en Z

L'élément de base de la trace en Z, appelé *élément en Z*, dérive de la structure du dexel. Les éléments en Z sont donc directement définis à partir des éléments de la liste dynamique de dexels. Ils font partie du sous-ensemble cartésien : $Z \times C \times T \times T$ défini précédemment où $t_{crea} < t_{disp}$. Un élément en Z(z, c, t_{crea}, t_{disp}) contient donc les informations suivantes : une valeur en Z, une couleur, une étape de création, une étape de disparition, et un lien vers l'élément suivant.
De même que pour le z-buffer étendu et les dexels, nous associons une liste d'éléments en Z à chaque pixel de l'écran.

3.2.1.2 Mise en place de la trace en Z

1. **Initialisation**
   Elle crée la trace en Z qui décrit le brut à l'étape 0.

2. **Mise à jour**
   L'étude des différentes opérations booléennes entre deux z-buffers étendus, rappelées dans la figure 3.4 de la page 52, permet la mise à jour de la trace.

Pour une étape n de la simulation, nous comparons un dexel du brut à un dexel de l'outil. Le brut est délimité par un NearZ (Z0) qui est apparu à l'étape i de la simulation et par un FarZ (Z1) qui a été créé à l'étape j de la simulation. Les étapes i et j sont des étapes quelconques de la simulation qui se sont déroulées avant l'étape courante n. Ainsi, la trace en Z se construit en même temps que le z-buffer étendu se modifie.

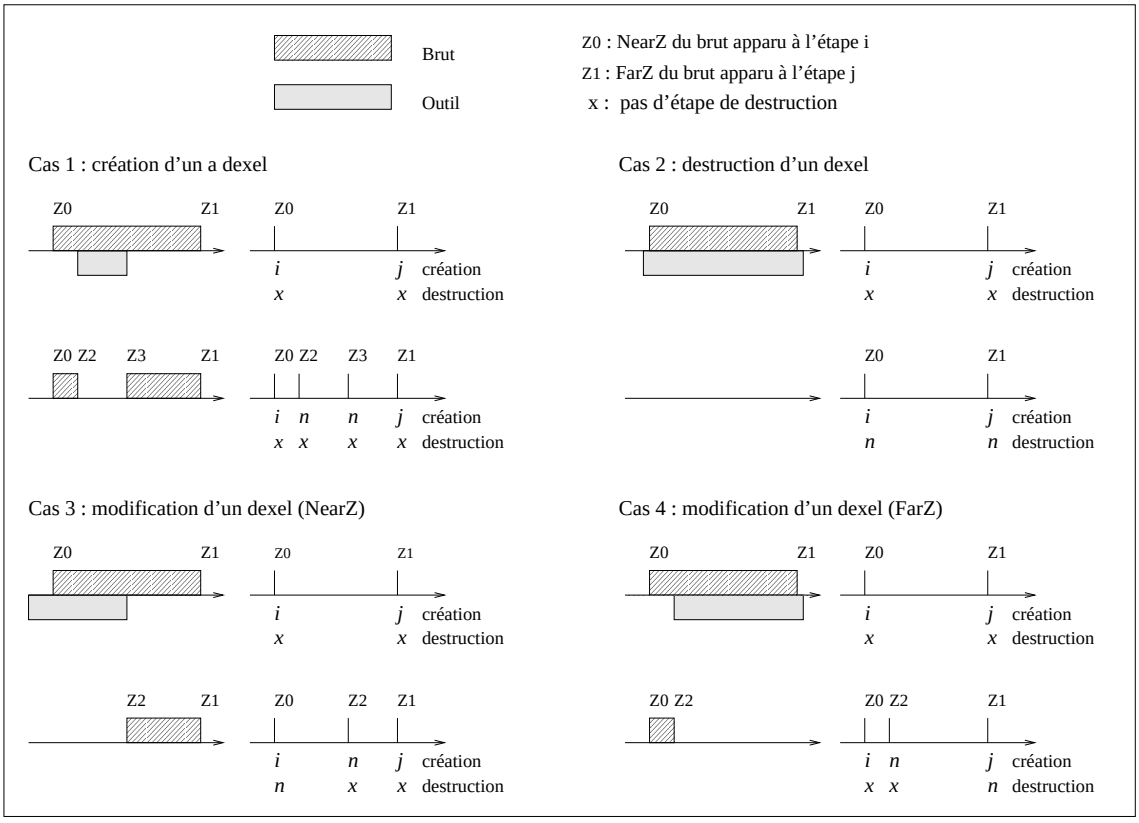

FIG. 3.4 – Operations booléennes sur les dexels à l'étape n

- **Création d'un dexel (cas 1 de la figure 3.4, page 52)**
  Lorsqu'un nouveau dexel est créé dans le z-buffer étendu, deux nouvelles valeurs de Z apparaissent. Cette apparition se répercute sur la trace : deux nouveaux éléments en Z sont créés. Ils sont classés dans la liste ordonnée suivant les Z croissants et il faut mettre à jour les informations suivantes : valeur en Z, couleur, étape de création.

- **Suppression d'un dexel (cas 2 de la figure 3.4, page 52)**
  Lorsqu'un dexel est supprimé du z-buffer étendu, les informations concernant ce dexel sont perdues dans le z-buffer étendu. La trace en Z garde en mémoire toutes les opérations effectuées sur le z-buffer étendu. La disparition du dexel entraîne la mise à jour de l'étape de disparition dans la trace.

- **Modification d'un dexel (cas 3 et 4 de la figure 3.4, page 52)**
  Si un dexel est réduit par l'avant (cas 3), alors la valeur de son NearZ est modifiée. Si un dexel est réduit par l'arrière (cas 4), alors la valeur de son FarZ est modifiée. Dans les deux situations, la trace subira deux modifications : une mise à jour d'un élément en Z déjà existant dans la trace (l'étape de disparition est actualisée) et la création d'un nouvel élément en Z dont nous connaissons la valeur, la couleur et l'étape de création. De par cette analyse, nous pouvons déduire les règles suivantes :
  Règle 1. Dans la trace en Z, toutes les modifications du z-buffer étendu sont répertoriées.
  Règle 2. Lorsqu'un dexel (deux nouvelles valeurs de Z) est ajouté dans le z-buffer étendu alors deux éléments en Z sont créés dans la trace en Z.
  Règle 3. Lorsqu'un dexel (deux valeurs de Z déjà existantes) est supprimé dans le z-buffer étendu alors deux éléments en Z sont mis à jour dans la trace en Z.
  Règle 4. Si et seulement si un NearZ est modifié dans le z-buffer étendu alors dans la trace en Z un seul élément en Z est ajouté et un autre élément en Z est mis à jour.
  Règle 5. Si et seulement si un FarZ est modifié dans le z-buffer étendu alors dans la trace en Z un seul élément en Z est ajouté et un autre élément en Z est mis à jour.

3.2.1.3 Reconstruction de la scène à l'étape n :

A partir de la trace en Z (figure 3.5a, page 54), nous allons reconstruire le z-buffer étendu tel qu'il était à l'étape n de la simulation.

1. Dans un premier temps, il est nécessaire de sélectionner les éléments en Z (figure 3.5b, page 54) qui vérifient les inégalités suivantes :

   $$\begin{array}{c}(\ \textbf{étape de création} \leq \textbf{étape } \boldsymbol{n}\ )\\ \textbf{ET}\\ (\ \textbf{étape de disparition} > \textbf{étape } \boldsymbol{n}\ )\end{array}$$

   Nous obtenons ainsi une liste ordonnée suivant les Z croissants, dans laquelle les NearZ et les FarZ sont alternés. D'après la règle 4, si l'étape de disparition d'un seul NearZ est mise à jour dans la trace, alors un nouveau NearZ est créé. Ainsi, si nous construisons le z-buffer étendu à partir de la trace, nous aurons toujours une liste ordonnée de Z où un NearZ suit un FarZ. Il en est de même avec la règle 5 et les FarZ. Les règles 2 et 3 nous montrent aussi que si deux éléments de la trace sont modifiés (étape de disparition mise à jour) ou créés, il s'agit obligatoirement d'un NearZ et d'un FarZ, ce qui permet lors de la reconstruction du z-buffer étendu de préserver l'alternance entre les NearZ et les FarZ.

2. Il est nécessaire de regrouper les éléments en Z (cf. figure 3.5c) par deux. Ils forment alors un dexel, dont le NearZ prend la valeur du premier élément en Z, et le FarZ prend la valeur du second élément en Z.

3. Finalement, nous retrouvons le z-buffer étendu souhaité en chaînant les dexels (cf. figure 3.5d).

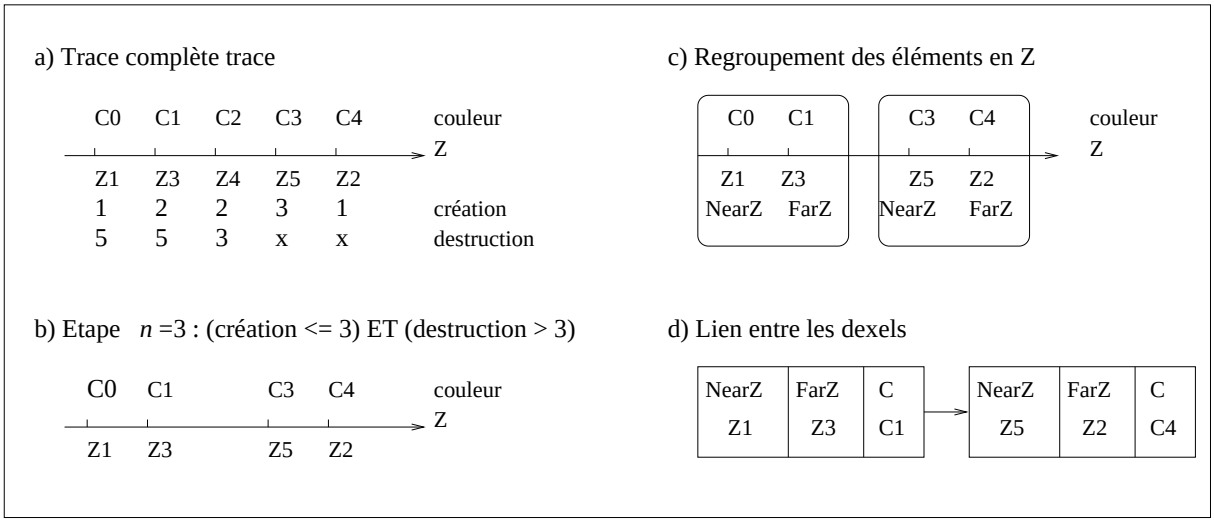

FIG. 3.5 – Reconstruction de scène à l'étape $n = 3$

3.2.2 Trace en temps ou trace rapide

Pour une étape donnée, la trace en temps permet d'afficher une scène. Les informations de la trace seront définies lors de l'actualisation du z-buffer étendu au cours de la simulation. Nous allons définir l'algorithme utilisant ces informations. La trace en Z et la trace en temps restent deux traces indépendantes.

3.2.2.1 Caractéristiques de la trace en temps

Lorsque nous travaillons sur le z-buffer étendu la direction de vue se situe dans la direction des Z croissants. Lorsque nous affichons un pixel à l'écran, il faut considérer le dernier Z du z-buffer étendu (ce sera donc un FarZ).
Règle 6. Le pixel affiché à l'écran représente le dernier FarZ du z-buffer étendu.
La trace en temps évolue lorsque la valeur du dernier FarZ est modifiée. Comme pour le z-buffer étendu, dans la trace en temps, nous associons à chaque pixel de l'écran, une liste d'*éléments en temps*. L'élément en temps est donc l'élément de base qui possède les informations suivantes : la valeur et la couleur du dernier FarZ, l'étape de création qui repère le moment où le dernier FarZ apparaît dans la trace, et un pointeur sur l'élément suivant (figure 3.6 de la page 55).

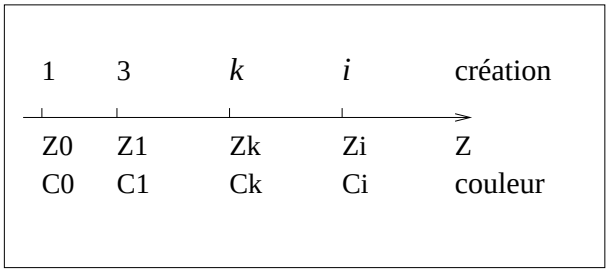

FIG. 3.6 – Représentation de l'historique en temps

Chaque élément en temps de la trace en temps est caractérisé par trois coordonnées : (z, c, t_{crea}). Les éléments de la trace en temps font donc partie du sous-ensemble cartésien : $Z \times C \times T$, sous-ensemble du sous-ensemble cartésien défini précédemment $Z \times C \times T \times T$ où $t_{crea} < t_{disp}$. La trace en temps est donc un ensemble plus restrictif que la trace en Z. Une conséquence directe de cette restriction se retrouve dans les possibilités des traces, puisque la trace en temps permet d'afficher seulement une scène de la simulation alors que le modèle géométrique de la même scène peut être reconstruit avec la trace en Z.

3.2.2.2 Mise en place de la trace en temps

1. **Initialisation**
   Elle crée la trace en temps qui décrit le brut à l'étape 0.

2. **Mise à jour**
   La mise à jour de la trace en temps n'est possible qu'en analysant les différentes opérations booléennes sur un intervalle entre deux z-buffers étendus présentées dans la figure 3.4 de la page 3.4.

   - **Création d'un dexel (cas 1 de la figure 3.4) :**
     Cette opération n'est pas prise en compte pour la mise à jour de la trace, puisqu'aucune modification n'a lieu sur le dernier FarZ.

   - **Suppression d'un dexel (cas 2 de la figure 3.4) :**
     Cette opération est prise en compte pour la mise à jour de la trace si et seulement si le dexel supprimé est le dernier dexel du z-buffer étendu. Deux cas peuvent se présenter. Dans le premier cas, le dernier dexel est le seul dexel du buffer. Un nouvel élément en temps est créé : son étape de création est l'étape courante n. Sa valeur de Z et sa couleur sont les données par défaut correspondant au fond (background). Dans le second cas, le dexel supprimé a un dexel qui le précède, un nouvel élément en temps est créé : son étape de création est l'étape courante n. Sa valeur de Z et sa couleur sont celles du FarZ du dexel précédent.

   - **Modification d'un dexel (cas 3 et 4 de la figure 3.4) :**
     Si le dexel est réduit par l'avant (cas 3) alors la valeur de son NearZ est modifiée. Cela ne modifie en aucun cas la trace en temps qui, d'après la règle 6, est mise à jour dès lors que le dernier Far Z est modifié. Si le dexel est réduit par l'arrière (cas 4) alors la valeur de son FarZ est modifiée. D'après la règle 6, la trace sera mise à jour si et seulement si le FarZ modifié est le FarZ du dernier dexel du z-buffer étendu. Un nouvel élément en temps est créé : nous connaissons son étape de création, la valeur du nouveau FarZ, et sa couleur. La trace est alors ordonnée en temps suivant les étapes de création.

La mise à jour est donc active dans les deux cas suivants uniquement :
- lorsque le FarZ du dernier dexel est modifié (cas 4)
- lorsque le dernier dexel est supprimé.

3.2.2.3 Reconstruction de la scène à l'étape n

A partir de la trace en temps (figure 3.6, page 55), nous voulons retrouver la couleur de chaque pixel de l'image telle qu'elle était à l'étape n de la simulation.

Il suffit de parcourir la trace en temps jusqu'à une étape s qui vérifie l'inégalité suivante :

$$\textbf{étape de création } \boldsymbol{s} \geq \textbf{étape } \boldsymbol{n}$$

- **Si étape de création** $s =$ **étape** n alors nous avons retrouvé la valeur de Z et la couleur associée au pixel de l'étape n (figure 3.7 de la page 58).

- **Si étape de création** $s >$ **étape** n alors la valeur de Z et la couleur du pixel de l'étape n sont celles de l'élément en Z qui précède l'élément en Z de l'étape s, puisque la nouvelle modification s'est produite après l'étape n (figure 3.8 de la page 58).

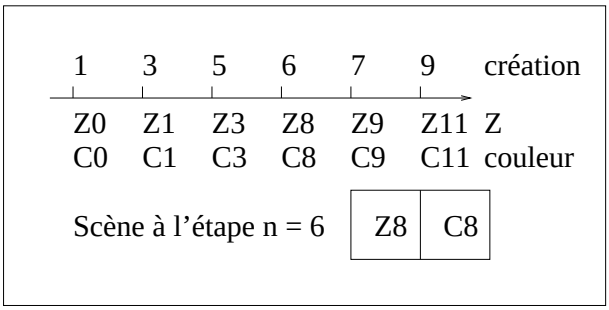

FIG. 3.7 – Reconstruction d'une scène avec la trace en temps. Cas où l'étape cherchée se trouve dans la trace

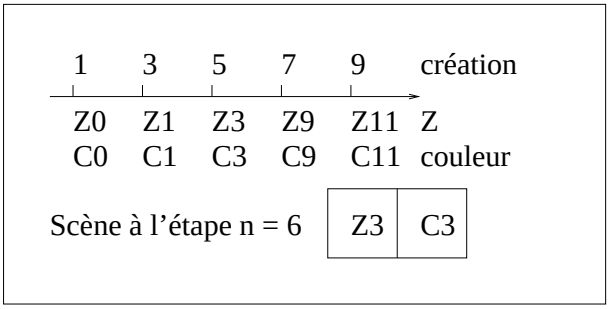

FIG. 3.8 – Reconstruction d'une scène avec la trace en temps. Cas où l'étape cherchée ne se trouve pas dans la trace

Chapitre 4

Structures de données pour le z-buffer étendu

4.1 Le z-buffer étendu ramené au problème du dictionnaire

Le z-buffer étendu évolue au cours de la simulation, des dexels sont créés, éventuellement modifiés et/ou supprimés. De plus, l'affichage d'une scène quelconque de la simulation par la trace en temps consiste à retrouver directement pour chaque pixel de l'écran la valeur du dernier FarZ à l'étape cherchée et à afficher sa couleur. Ainsi, que ce soit pour la construction du z-buffer étendu et/ou pour l'utilisation des traces, nous recherchons une valeur dans un ensemble ordonné de données. Nous pouvons alors nous ramener à un *dictionnaire* qui regroupe les opérations d'insertion, de suppression et de recherche d'un élément dans un ensemble [37]. Les structures de données classiques pour représenter un dictionnaire sont : les listes, les arbres de recherche et plus récemment les *skip lists*. Plusieurs facteurs influent sur le choix d'une structure pour un algorithme précis : le temps d'exécution, l'occupation mémoire ou la simplicité de programmation.

4.2 Etat de l'art des différentes structures de données pour un dictionnaire

4.2.1 Listes chaînées

Les listes chaînées apparaissent comme la structure de données la plus simple à implémenter pour le z-buffer étendu, et la seule à notre connaissance présentée dans la littérature. En effet, nous retrouvons rapidement un dexel cherché en nous déplaçant d'élément en élément. La facilité d'insertion (ou de suppression) d'un élément dans la liste est un des principaux avantages des listes chaînées. Cette implémentation présente cependant quelques inconvénients comme le temps de recherche d'un élément dans une liste chaînée de n éléments qui est dans le pire des cas en $O(n)$: pour atteindre l'élément cherché, il faut alors parcourir un à un tous les éléments le précédant dans la liste.
Dans notre application, le z-buffer étendu est composé d'au maximum cinq ou six dexels. La structure de liste chaînée est bien adaptée à ce cas là. En revanche, les traces peuvent atteindre des tailles plus importantes et comprendre jusqu'à cent cinquante éléments. Pour rechercher plus rapidement un élément dans une telle trace, nous allons nous intéresser aux arbres binaires et aux skip lists, structures de données qui deviennent intéressantes dans ce type de problème.

4.2.2 Arbres binaires

Dans les arbres binaires de recherche, le temps d'exécution pour la recherche est dans tous les cas $O(h)$, où h est la hauteur de l'arbre et dans le pire des cas en $O(n)$, où n est le nombre de noeuds de l'arbre. Pour accélérer ces temps d'exécution, on utilise des arbres binaires de recherche équilibrés. Un arbre binaire de recherche de n noeuds est équilibré si sa hauteur est en $O(\log n)$, ce qui réduit le temps d'exécution à $O(\log n)$ dans le pire des cas. Plusieurs implémentations permettent d'obtenir des arbres binaires équilibrés : les arbres AVL, les arbres 2-3 , les arbres rouges et noirs [38]. Pour rechercher un élément le plus rapidement possible dans un dictionnaire, la solution la plus courante consiste à utiliser des arbres binaires de recherche équilibrés. Malheureusement, le rééquilibrage rend la construction de la structure coûteuse en temps. L'avantage des listes chaînées est qu'une fois l'étape n trouvée, il suffit de passer directement à l'élément suivant pour obtenir l'étape $n+1$, ce qui permet d'animer la simulation rapidement. Nous souhaitons donc utiliser la rapidité de recherche des arbres binaires et la simplicité des listes chaînées.

4.2.3 Amélioration des listes chaînées par les skip lists

Une *skip list* est une structure de données probabiliste simple et efficace introduite par Pugh en 1990 [39]. Les skips lists sont donc une généralisation relativement récente des listes chaînées simples : elles atteignent les performances des arbres binaires de recherche en conservant la structure des listes chaînées. Pugh appelle *skip lists* des listes chaînées qui permettent de *sauter* au-dessus d'éléments intermédiaires pour atteindre rapidement l'élément cherché. Par exemple, si on modifie une liste de n éléments de manière à ce que chaque noeud ait un pointeur sur le noeud deux positions plus loin dans la liste, dans le pire des cas, il faut traverser $(\frac{n}{2}+1)$ éléments de cette liste.

Chaque élément de la skip list est représenté par un noeud (on parle aussi de tour) contenant un tableau de plusieurs pointeurs sur les noeuds suivants. La hauteur ou le niveau d'un noeud correspond au nombre de pointeurs que contient le noeud (figure 4.1, page 62). A un noeud de niveau k, on associe un tableau de k pointeurs.
Dans une liste chaînée simple, les noeuds sont tous de niveau 1. Dans une skip list, le $i$ème pointeur d'un noeud de niveau k $(i \leq k)$ avec une valeur clé v pointe sur le prochain noeud possédant au moins i niveaux dont la valeur clé est supérieure ou égale à v (figure 4.2, page 63).
Le premier noeud d'une skip list est appelé la *tête* de liste. Sa hauteur est maximale. Il contient donc un tableau de pointeurs depuis le niveau 1 jusqu'au niveau maximum. Le dernier noeud d'une skip list est le noeud NIL. Sa hauteur est également

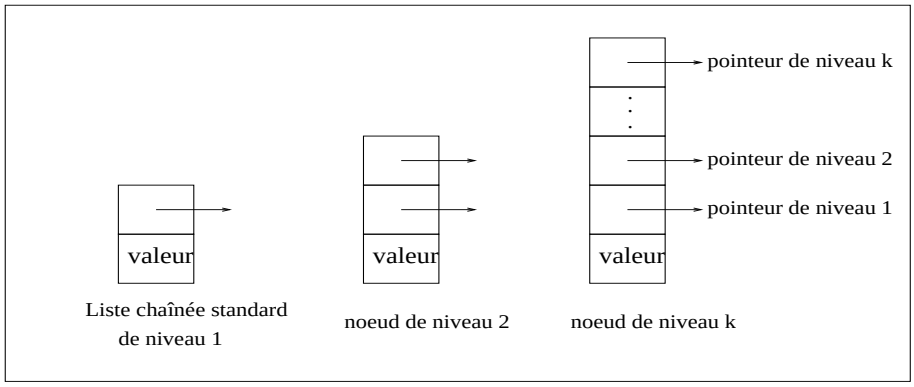

FIG. 4.1 – Noeud d'une skip list

maximale (figure 4.2, page 63).

Les skips lists qui subissent des mises à jour (insertion ou suppression d'un noeud) sont des structures de données dynamiques [40] et sont construites de manière probabiliste. En effet, lorsqu'on insère un nouveau noeud dans la skip list, son niveau est fixé par un générateur de nombres aléatoires. Dans de telles skip lists, nous trouvons en moyenne 50 % de noeuds de niveau 1, 25 % de noeuds de niveau 2, 12,5 % de noeuds de niveau 3, etc. Une description détaillée et une analyse des algorithmes d'insertion, de suppression et de recherche d'éléments dans une skip list probabiliste est proposé dans [41]. D'autres études portent sur l'analyse analytique des temps de recherche [42] et sur les performances des skips lists [43]. Les algorithmes d'insertion d'éléments dans une skip list prennent plus de temps que dans le cas d'une liste chaînée, mais la recherche d'un élément est très rapide. Le temps moyen de recherche est de $O(\log n)$ et dans le pire des cas on atteint $O(n)$ [1]. Nous avons décidé d'utiliser les skips lists pour profiter de la rapidité de ce temps de recherche.
Dans les skip lists probabilistes, la forme de la liste dépend d'un générateur de nombres aléatoires. Pour les skip lists déterministes [44], la forme de la liste est connue. Par exemple, pour la skip list déterministe 1-2-3, chaque saut est de taille 1, 2, puis 3 (figure 4.2, page 63). Les skip lists sont un sous ensemble des structures de données dynamiques ; elles sont plus performantes dans la recherche que dans l'insertion d'un élément.
Pour la recherche d'un élément de valeur v_f dans la skip list, nous utiliserons l'algorithme proposé par Pugh. La recherche s'effectue en traversant la liste de la tour de tête de liste vers la tour de fin de liste (tour NIL) sur les différents niveaux de pointeurs. Le premier noeud courant est la tête de la liste. On se place sur son

[1] Pour un dictionnaire de 250 éléments, il y a une chance sur un million pour que le temps de recherche mettent trois fois plus de temps que le temps moyen attendu [39].

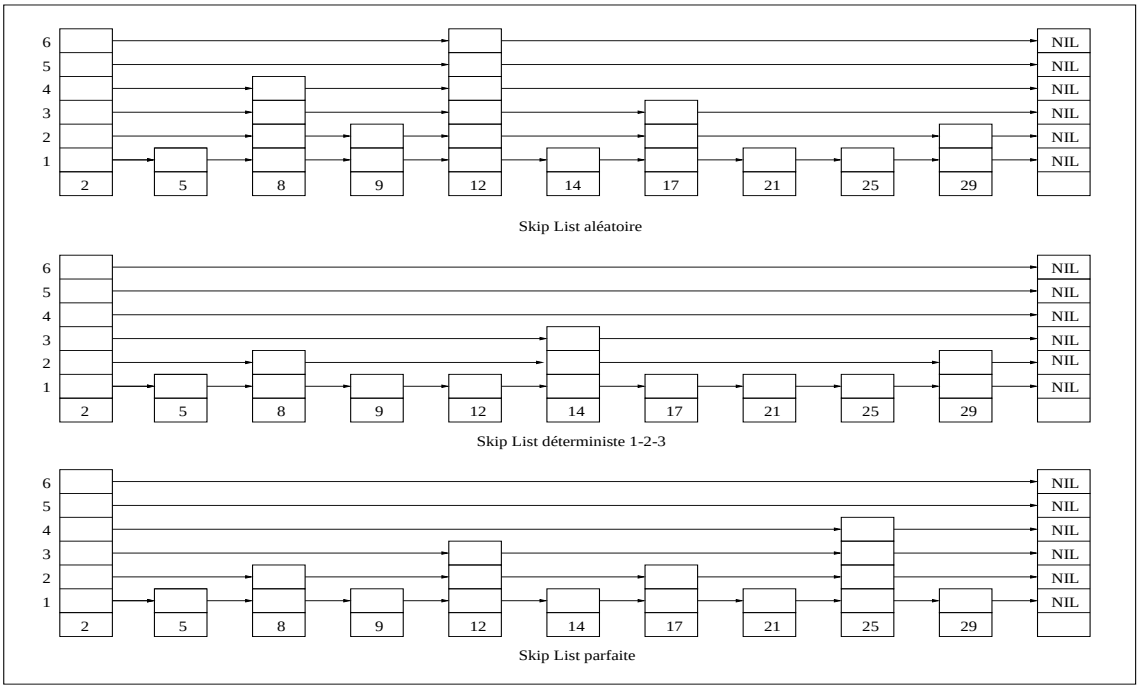

FIG. 4.2 – SkipList

pointeur de niveau le plus élevé, et on traverse la liste sur ce niveau. Pour le noeud suivant de même niveau et de valeur clé v, on compare v à la valeur v_f cherchée :

- Si $v_f \geq v$, on se déplace sur le noeud suivant de même niveau qui devient le nouveau noeud courant, et on continue d'explorer la liste sur le même niveau.
- Si $v_f < v$, on reste sur le noeud courant, et on continue l'exploration de la skip list sur le niveau inférieur.

Lorsqu'on atteint le niveau 1, on se trouve soit devant le noeud cherché, soit ce noeud n'est pas dans la liste. En effet, si le noeud suivant contient la valeur v cherchée, alors cette valeur appartient effectivement à la skip list et nous avons trouvé sa position, sinon la recherche a échoué et la valeur v_f ne se trouve pas dans la skip list. Nous avons adapté la fin de cet algorithme à notre application. Pour reconstruire une scène de la simulation à l'étape n, nous cherchons dans la skip liste un noeud dont l'étape de création serait n. Lorsque le niveau 1 est atteint, le noeud courant n_c pointe sur le noeud suivant n_s. Si l'étape de création s de n_s est égale à l'étape n cherchée, les données du noeud suivant correspondent à la valeur en Z et à la couleur du pixel de l'étape n. Si l'étape de création s de n_s est supérieure à l'étape n cherchée, les données du noeud courant correspondent à la valeur en Z et à la couleur du pixel de l'étape n, puisque la prochaine modification n'aura lieu qu'après l'étape n.

L'algorithme de reconstruction d'une scène à partir d'une skip list est le suivant :

ReconstructionScene (EtapeCreationCherchee)
x :=TourTeteDeListe
pour i := NiveauMax .. 2 faire
tant que (x.suivant[i].EtapeCreation < EtapeCreationCherchee) faire
x :=x.suivant[i]
fin tant que
fin pour
si (x.suivant[1].EtapeCreation=EtapeCreationCherchee)
alors retourne x.suivant[1].CouleurPixel
sinon retourne x.CouleurPixel
fin si
fin ReconstructionScene

4.3 Choix d'une structure de données adaptée au z-buffer étendu et les traces

4.3.1 Structures de données pour z-buffer étendu

4.3.1.1 Evolution du z-buffer étendu au cours de la simulation

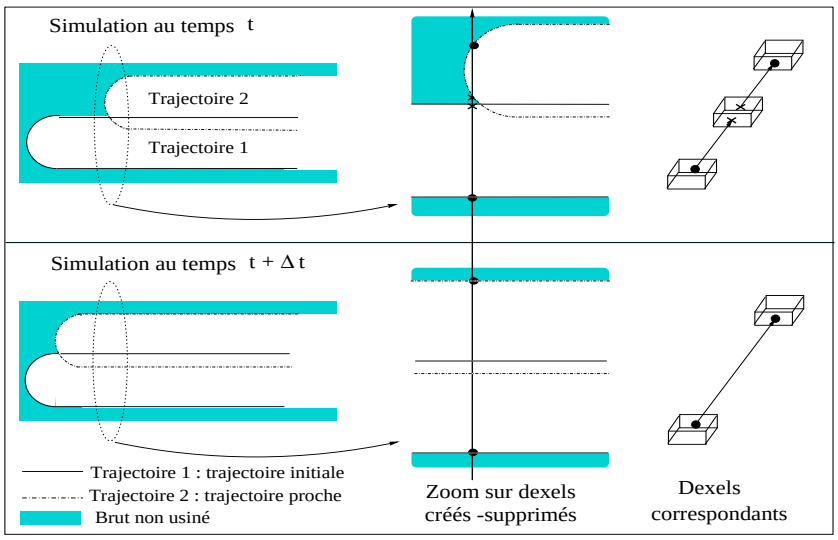

FIG. 4.3 – Visualisation des dexels créés, puis supprimés pour une trajectoire proche

Une trajectoire est rarement isolée, les pièces sont habituellement usinées par un ensemble de trajectoires suivant une stratégie. Mais quelle que soit la stratégie d'usinage (zigzag ou en contour parallèle), les trajectoires isolées initiales (trajectoire de type 1 sur la figure 4.3, page 64) sont souvent suivies de trajectoires proches (trajectoire de type 2 sur la figure 4.3, page 64) qui recouvrent partiellement ces dernières. La distance entre les deux trajectoires est appelée *distance entre passes*. Lors de la simulation, les trajectoires sont échantillonnées, l'outil avance donc de pixel en pixel à chaque étape de la simulation. Comme l'outil est circulaire, de nouveaux dexels sont créés dans certains cas à chaque nouvelle position élémentaire et supprimés lorsque l'outil continue sa progression sur la trajectoire. En effet, si la distance entre passes est inférieure au rayon de l'outil, la trajectoire de type 2 n'engendre pas de nouveaux dexels lors de son passage avec un recouvrement partiel de la trajectoire de type 1. Par contre si la distance entre passes est supérieure au rayon de l'outil alors de nouveaux dexels sont créés lorsque la trajectoire de type 2 est parallèle à la trajectoire de type 1, comme le montre la figure 4.3. Les dexels sont créés au début du passage de l'outil, puis supprimés lorsque l'outil poursuit sa trajectoire.

4.3.1.2 Choix d'une structure de données adaptée au z-buffer étendu

Au cours de la simulation, le z-buffer étendu évolue et de nombreux dexels sont créés, puis supprimés peu de temps après. Des opérations de recherche, d'insertion et de suppression d'éléments sont utilisées. L'utilisation des skip lists n'est pas appropriée dans ce cas puisque seule la construction du z-buffer est prise en compte, et une fois la simulation achevée, aucune recherche d'élément n'est a priori effectuée sur le z-buffer étendu. La construction du z-buffer étendu consiste à comparer les dexels de l'outil et du brut entre eux, à insérer, et à supprimer ou à modifier ces nouveaux dexels. Or, l'utilisation d'une skip list devient intéressante lorsqu'on a des éléments à insérer ou à supprimer dans une liste. Lors de la construction du z-buffer étendu, nous travaillons sur des intervalles, et même si le nombre d'éléments insérés, mais surtout supprimés, au cours de la simulation est important, le nombre de dexels associé à un pixel et composant un z-buffer étendu à un instant donné de la simulation est peu important, c'est pourquoi il est préférable d'utiliser une simple liste chaînée.

4.3.2 Structures de données pour les traces

Les traces utilisent uniquement les opérations d'insertion et de recherche d'un élément. Ces opérations interviennent tout d'abord lors de la construction des traces,

puis lors de l'utilisation des traces pour reconstruire et/ou afficher le modèle géométrique à un moment quelconque de la simulation.

4.3.2.1 Structures de données pour la trace en Z

Pour la trace en Z, nous rechercherons une structure de données qui permette de réduire le temps d'exécution lors de la construction de la trace en Z uniquement. Nous avons vu précédemment que lors d'une reconstruction du modèle géométrique avec la trace en Z, tous les éléments de la trace en Z doivent être examinés les uns après les autres, la structure de données de la liste chaînée sera tout aussi bien envisagée que la structure de données sous forme de skip list aléatoire. Par contre, il est souhaitable que la structure de données choisie garde une certaine linéarité des éléments. Par exemple, il serait difficile de reconstruire le z-buffer étendu à partir d'une trace en Z sous forme d'arbre binaire équilibré, puisque le rééquilibrage de l'arbre aurait détruit le caractère alterné des NearZ et FarZ.

4.3.2.2 Structures de données pour la trace en temps

A priori, lors de la construction des traces, une simple liste chaînée permet d'obtenir un temps d'exécution plus rapide, mais lors de l'utilisation des traces pour la reconstruction et/ou l'affichage du modèle géométrique, le choix entre une liste simplement chaînée ou une skip list sera discuté ultérieurement.

4.4 Interval Treap : une nouvelle structure de données complète pour le z-buffer étendu

4.4.1 Vers la nouvelle structure de données : Interval Treap

Pour obtenir une simulation interactive, nous proposons d'implémenter le z-buffer étendu sous forme d'*Interval Treap* (Treap : Tree-heap). Cette nouvelle structure de données, basée sur les arbres binaires d'intervalles et la propriété de *heap-order* (ou propriété de *tas* où la valeur d'un noeud est strictement supérieure à la valeur de son père), contient maintenant les paramètres nécessaires pour construire, puis afficher et reconstruire le z-buffer étendu tel qu'il était à un moment précis de la simulation [14].
Comme le nombre de données mémorisées au cours de la simulation est important, la structure de données d'arbre binaire de recherche est intéressante car elle permet un accès rapide : en effet, le temps nécessaire pour atteindre un élément x de l'arbre est proportionnel à la profondeur de x dans l'arbre. Dans le pire des cas, le temps

de recherche d'un élément dans un arbre équilibré de n éléments est en $O(\log n)$. Pour garder en mémoire toutes les opérations subies par le z-buffer étendu, il est nécessaire de stocker deux types de données : des données relatives à la position qui permettront de mémoriser les différentes valeurs en Z (NearZ et FarZ), et des données relatives au temps qui permettront de mémoriser l'instant d'apparition de ces valeurs en Z dans la structure de données. Comme le z-buffer étendu est constitué de dexels modélisés dans un espace à une dimension par des intervalles de données [NearZ,FarZ], nous avons tout d'abord pensé à utiliser une structure de données d'arbres d'intervalles. Une représentation possible est le *span space* proposé par Livnat [45], qui utilise un *k-d tree* pour substituer des points aux intervalles. Le k-d tree est une structure de données utilisée pour la décomposition spatiale qui permet de partitionner l'espace pour une répartion discrète de points. Les points eux-mêmes sont à l'origine des frontières entre les différents sous-espaces. Dans un espace à deux dimensions, on utilise des arbres binaires de recherche à deux dimensions (ou *two-dimensionnal binary search tree, 2-d tree*) [46]. Le plan est alors divisé successivement par des lignes verticales et par des lignes horizontales. Chaque noeud contient un élément de l'ensemble et on lui associe une ligne qui passe par cet élément. Pour les niveaux pairs (à partir du niveau 0 de la racine) ce sera une ligne verticale et pour les niveaux impairs ce sera une ligne horizontale (voir figure 4.4, page 67). Dans un espace à k dimensions, on parle de k-d tree (arbres binaires de recherche à k dimensions, le plan est alors divisé successivement par k hyperplans passant par les points de l'ensemble).

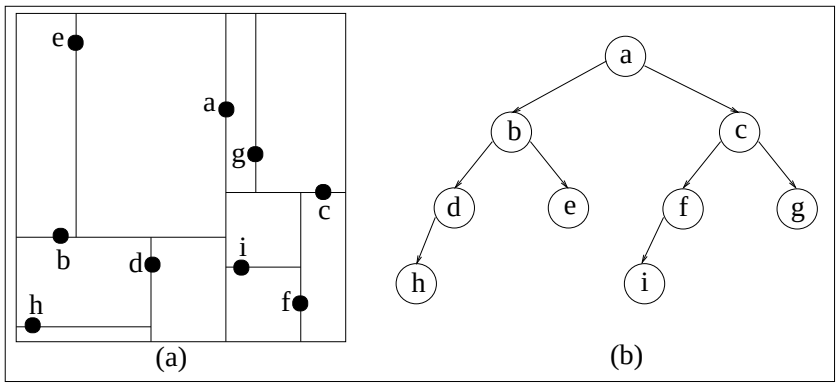

FIG. 4.4 – (a) Décomposition spatiale du plan par des lignes verticales et horizontales et (b) k-d tree correspondant

Dans le span space, un intervalle $I = [a_i, b_i]$ avec $a_i \leq b_i$ est représenté par un point dans un espace à deux dimensions avec a_i comme abscisse et b_i comme ordonnée.

Trouver les intervalles qui contiennent la valeur q revient à chercher les points tels que $x \leq q$ et $y \geq q$. L'inconvénient de cette méthode est la perte d'informations relatives au temps et notamment du moment où apparaît le dexel dans le z-buffer étendu.

Nous souhaitons donc travailler sur une structure de données qui garde en mémoire deux types de valeurs : dans l'espace et dans le temps. Dans les arbres cartésiens introduits par Vuillemin [47], chaque noeud contient deux clés (x,y). La valeur de la clé x vérifie la propriété des arbres binaires de recherche : la valeur de la clé x d'un noeud père est plus grande que la valeur de la clé x de son fils gauche et plus petite que la valeur de la clé x de son fils droit. La valeur de la clé y vérifie la propriété de *heap-order* [48] : la valeur de la clé y d'un noeud père est toujours plus petite que la valeur de la clé y de ses noeuds fils. Ces arbres binaires de recherche ne sont pas équilibrés. En moyenne, le temps d'exécution des opérations de base d'insertion et de recherche d'un élément dans un arbre cartésien de n éléments est en $O(\log(n))$, et dans le pire des cas, si l'arbre est complètement déséquilibré, cela peut atteindre $O(n)$. Pour obtenir des arbres cartésiens équilibrés, il suffit de remplacer la valeur clé y par une valeur aléatoire qui permet d'équilibrer l'arbre lors de sa construction. Nous obtenons alors des Treaps [49, 50] qui sont des arbres binaires de recherche équilibrés à construction aléatoire. Dans notre application, nous n'introduisons pas de valeurs aléatoires, mais nous mémorisons un intervalle [NearZ, FarZ] en stockant deux valeurs clés en Z et l'instant d'apparition d'un nouvel élément dans la structure de données en stockant une valeur indexée en temps. Nous créons ainsi une nouvelle structure de données appelée *Interval Treap* qui permet de conserver à la fois des intervalles de données et des instants d'apparition afin de mémoriser toutes les modifications subies par le z-buffer au cours de la simulation.

4.4.2 Présentation de l'Interval Treap

Nous définissons un *Interval Treap* comme un arbre binaire d'intervalles indexé en temps mais non équilibré. Il garde en mémoire les informations sur de nouveaux intervalles crées à partir d'opérations successives sur un intervalle initial : l'Interval Treap est donc composé d'*intervalles superposés*. L'élément de base d'un Interval Treap est un noeud qui représente un intervalle apparu à une étape i. Il contient les champs suivants : deux valeurs clés en Z, et une valeur en temps. Une valeur clé en Z appelée Z_{min} borne l'intervalle de données par sa valeur minimale : pour la représentation graphique du noeud, cette valeur sera placée à gauche (figure 4.5, page 69). Une valeur clé en Z appelée Z_{max} borne l'intervalle de données par sa valeur maximale : pour la représentation graphique du noeud, cette valeur sera placée à droite (figure 4.5, page 69). La valeur en temps correspond au mo-

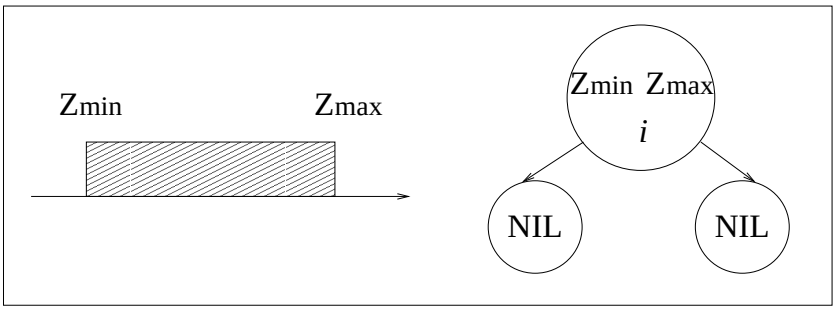

FIG. 4.5 – Correspondance entre l'intervalle créé à l'étape i et le noeud de l'Interval Treap

ment où apparaît l'intervalle dans la structure de donnée, nous l'appelons l'étape de création. Dans la représentation graphique du noeud, cette valeur est placée sur la figure 4.5 de la page 69 en-dessous des valeurs clés en Z.
Montrons que l'Interval Treap est un arbre binaire d'intervalles. En effet, un noeud ne peut avoir au plus que deux fils car les modifications ne peuvent avoir lieu que sur Z_{min} ou sur Z_{max} et lorsqu'un intervalle est modifié, nous ajoutons des fils au noeud représentant cet intervalle. Si le Z_{min} est modifié, un nouveau fils gauche sera créé, si le Z_{max} est modifié, ce sera un nouveau fils droit. Comme dans tout arbre binaire, un noeud a deux pointeurs vers des noeuds fils. Si un fils n'existe pas, le noeud pointe sur NIL. Une feuille est un noeud dont les deux pointeurs fils sont à NIL.
Nous allons étudier la mise à jour de l'Interval Treap en n'analysant que les opérations booléennes de différence entre deux intervalles qui sont les seules utilisées en usinage.

- **Modification d'un intervalle par son Z_{min} (cas 1 de la figure 4.6, page 70)**
  Si un intervalle est réduit par l'avant, alors la valeur de son Z_{min} est modifiée. Par convention, le Z_{min} est la valeur clé placée à gauche dans le noeud. Pour symboliser la disparition du Z_{min} de l'intervalle père, le fils gauche du noeud de l'intervalle père pointe désormais sur NIL. Un nouveau Z_{min} apparaît dans la structure de données créant ainsi un *nouvel* intervalle. Pour représenter ce nouvel intervalle dans la structure de données, un nouveau noeud fils est créé. Il s'agit d'un fils droit. Sa valeur clé Z_{min} est la nouvelle valeur Z_{min}. Sa valeur clé Z_{max} est la valeur Z_{max} du noeud père. Sa valeur en temps est celle de l'étape présente n.

- **Modification d'un intervalle par son Z_{max} (cas 2 de la figure 4.6, page 70)**
  Ce cas est le symétrique du précédent où Z_{min} est remplacé par Z_{max} et Z_{max}

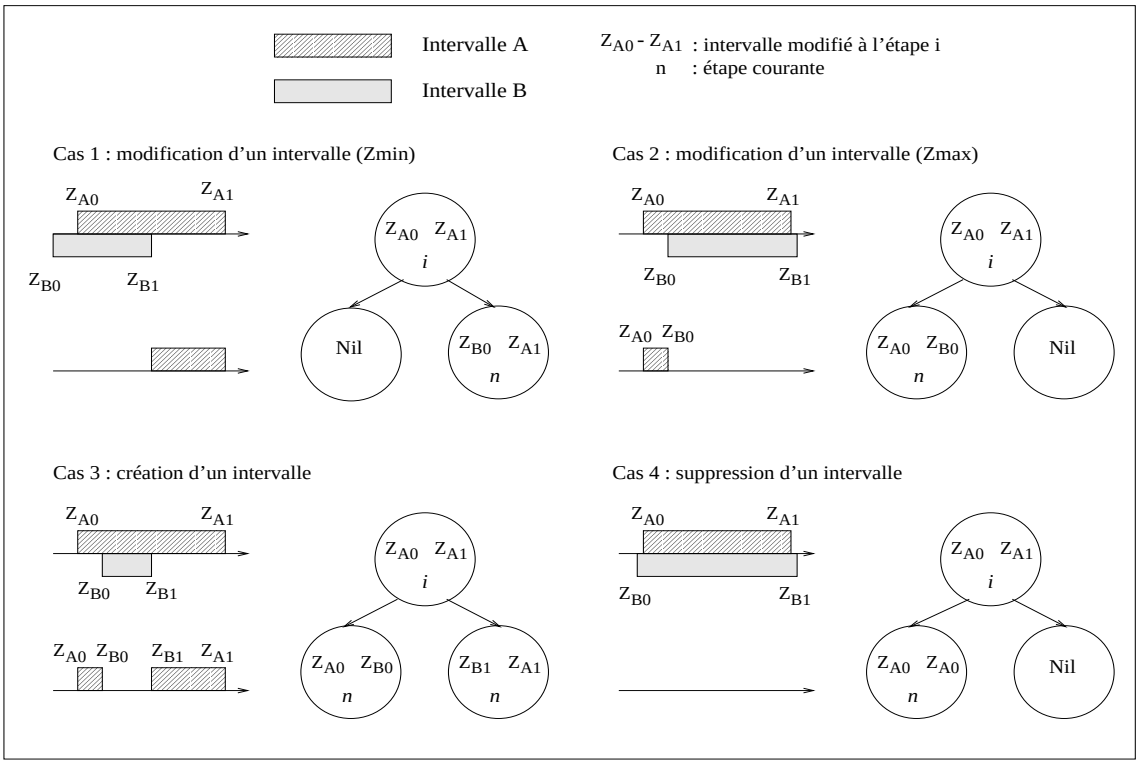

FIG. 4.6 – Opérations booléennes de différence entre deux intervalles : influence sur l'Interval Treap

remplacé par Z_{min}.

- **Création d'un intervalle (cas 3 de la figure 4.6, page 70)**
  Lorsqu'un nouvel intervalle apparaît dans la structure de données, un nouveau Z_{min} et un nouveau Z_{max} sont créés. Les bornes Z_{min} et Z_{max} de l'intervalle père ne sont pas altérées. La création d'un nouvel intervalle se répercute sur la structure de données par la création de deux nouveaux noeuds fils : un fils droit et un fils gauche dont les valeurs sont respectivement celles du cas 1 et et du cas 2 précédents.

- **Suppression d'un intervalle (cas 4 de la figure 4.6, page 70)**
  La suppression d'un intervalle est une étape plus délicate à modéliser. Pour montrer qu'un intervalle est supprimé à l'étape i, il suffit qu'un seul noeud fils soit créé pour représenter cette étape i. Par convention, le noeud créé sera un fils gauche qui aura les mêmes valeurs pour les clés Z_{min} et Z_{max}. Ce noeud est

appelé *feuille vide* de l'Interval Treap. En effet, il n'y aura plus de modification possible à partir de ce noeud c'est donc une feuille que nous qualifions de vide puisque la valeur du Z_{min} et celle du Z_{max} sont égales.

4.4.3 Utilisation de l'Interval Treap avec le z-buffer étendu

4.4.3.1 L'Interval Treap : une structure de données complète pour le z-buffer étendu

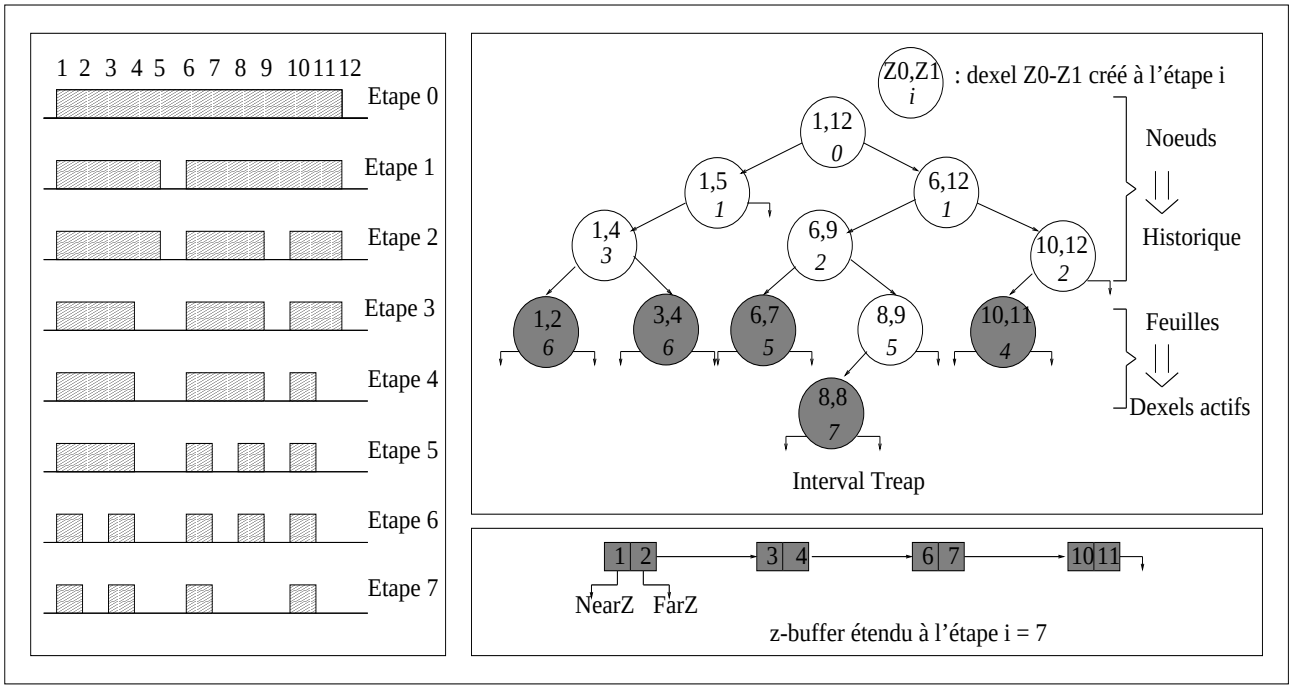

FIG. 4.7 – Exemple Interval Treap et z-buffer étendu

L'Interval Treap est une structure de données bien adaptée au z-buffer étendu. L'élément de base de l'Interval Treap contient alors les informations relatives aux dexels. La valeur clé Z_{min} devient la valeur clé NearZ. La valeur clé Z_{max} devient la valeur clé FarZ. La valeur clé en temps représente l'instant d'apparition du dexel dans le z-buffer étendu au cours de la simulation.
Lorsqu'un dexel est modifié, nous ajoutons des noeuds fils au noeud père modélisant le dexel à modifier. Ces fils deviennent alors les nouvelles feuilles de l'arbre. *Les dexels actifs du z-buffer étendu sont les feuilles de l'arbre Interval Treap.* Les feuilles de l'arbre Interval Treap sont *ordonnées*. Si nous parcourons les feuilles non vides

de l'Interval Treap de gauche à droite, nous retrouvons les dexels du z-buffer étendu dans le même ordre (figure 4.7, page 71). Les noeuds intermédiaires dans l'Interval Treap font partie de l'historique du z-buffer étendu car à un moment donné de la simulation ces noeuds ont eux-aussi été des feuilles de l'arbre donc des dexels actifs du z-buffer étendu.

L'Interval Treap mémorise toutes les modifications subies par le z-buffer au cours de la simulation. Il occupe donc en mémoire plus de place qu'une simple liste chaînée, mais ses fonctionnalités regroupent à la fois celles du z-buffer étendu et des traces. Les simulations montreront que la mémoire occupée par un Interval Treap est finalement moins importante que le cumul de la mémoire occupée par deux listes chaînées, une modélisant le z-buffer étendu et l'autre modélisant la trace associée à ce même z-buffer étendu.

4.4.3.2 Mise à jour de l'Interval Treap

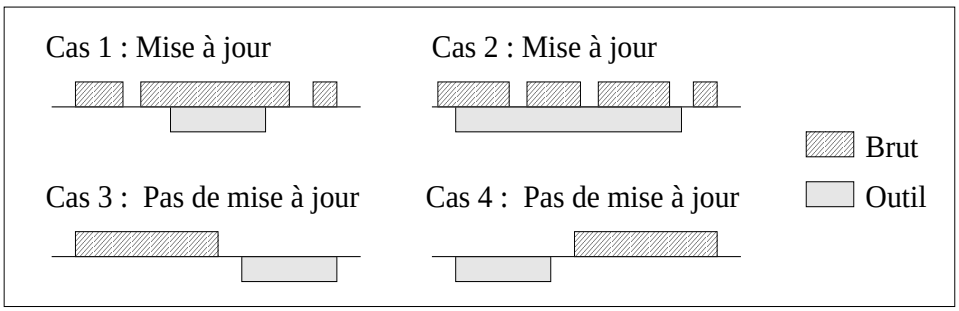

FIG. 4.8 – Mise à jour d'un dexel

Lors de la mise à jour de l'Interval Treap, il faut retrouver la feuille (cas 1 de la figure 4.8, page 72) ou les feuilles (cas 2 de la figure 4.8, page 72) correspondant aux dexels à modifier. Comme les feuilles de l'Interval Treap sont *ordonnées*, **l'exploration de l'arbre s'effectue suivant un parcours préfixe ou préordre.** Pour éviter de parcourir tout l'arbre à chaque nouvelle étape, nous avons mené l'étude suivante. Tout d'abord, nous avons répertorié deux cas (cas 3 et 4 de la figure 4.8, page 72) pour lesquels le dexel ne subit pas de modifications.

- cas 3 : le dexel outil précède le dexel du brut, sans le chevaucher. Cette condition peut se résumer par l'inégalité suivante : $FarZ\ Outil \leq NearZ\ Brut$
- cas 4 : le dexel du brut précède le dexel de l'outil, sans le chevaucher. Cette condition peut se résumer par l'inégalité suivante : $FarZ\ Brut \leq NearZ\ Outil$

Un dexel sera modifié si et seulement si le cas 3 ET le cas 4 ne sont pas vérifiés. Un dexel ne sera pas modifié si et seulement si le cas 3 OU le cas 4 est vérifié. Un dexel pourra être modifié si et seulement si, il vérifie la condition suivante :

$$\textbf{FarZOutil} > \textbf{NearZBrut} \qquad \textbf{ET} \qquad \textbf{FarZBrut} > \textbf{NearZOutil} \tag{4.1}$$

Comment cette condition peut-elle modifier l'exploration de l'arbre Interval Treap ?
Tous les noeuds de l'arbre Interval Treap ont été à un moment donné une feuille, c'est-à-dire un dexel actif du z-buffer étendu. Il faut donc pour chaque noeud vérifier si la condition (1) est respectée.

- si (1) est VRAIE : nous continuons l'exploration car les noeuds suivants (dexels) sont susceptibles d'être modifiés.
- si (1) est FAUSSE : nous arrêtons *momentanément* l'exploration de cette branche de l'Interval Treap. En effet, si un noeud (dexel) ne vérifie pas la condition (1) alors ses noeuds fils (*dexels fils*) ne vérifieront pas non plus la condition (1).

Ensuite, nous voulons déterminer dans quel cas l'exploration est interrompue de manière momentanée ou définitive. A chaque étape, nous comparons le z-buffer étendu de l'outil au z-buffer étendu du brut dexel par dexel. Les modifications sur les dexels du brut ne seront plus possibles lorsque :

$$\textbf{FarZBrut} \geq \textbf{FarZOutil} \tag{4.2}$$

Pour le z-buffer étendu sous forme d'arbre Interval Treap, nous aurons un *arrêt définitif* du parcours de l'arbre dans l'un ou l'autre des cas suivants :

- si une feuille vient d'être créée et Nouveau FarZ Brut $\geq$ FarZ Outil.
- si la condition (1) est FAUSSE et si (2) est VRAIE.

4.4.3.3 Utilisation de l'Interval Treap

Affichage de la couleur en cours à l'écran
La couleur visible à l'écran est la couleur du dernier FarZ du z-buffer étendu. Lorsque le z-buffer est sous forme d'Interval Treap, nous nous intéressons donc à la dernière feuille non vide (NearZ $\neq$ FarZ), c'est-à-dire la feuille la plus à droite dans l'arbre. Pour cette raison, lors de l'affichage de la couleur du z-buffer étendu en cours, **le parcours de l'arbre sera de type postordre** (fils droits vers fils gauches). Si toutes les feuilles sont vides, la couleur recherchée est la couleur du fond.

Utilisation de l'Interval Treap comme historique du z-buffer étendu
Grâce à cette structure de données, nous pouvons reprendre l'usinage à partir de

n'importe quelle étape de la simulation. Pour reconstruire l'Interval Treap du z-buffer étendu tel qu'il était à une étape i de la simulation, il suffit de *couper* l'arbre correctement. Nous pouvons indifféremment choisir un parcours préordre ou post-ordre, puisque toutes les branches de l'arbre doivent être examinées. Pour chaque ramification de l'arbre, nous stoppons l'exploration dans l'un ou l'autre des cas suivants :

- si le noeud courant vérifie les inégalités suivantes :
  (Etape de création du fils gauche > Etape i) ET
  (Etape de création du fils droit > Etape i)
  Nous initialisons alors les pointeurs de ce noeud vers les fils droits et gauches à NIL.
- si le noeud courant est une feuille (noeud sans fils droit, ni fils gauche).

Une fois l'Interval Treap reconstruit, deux applications peuvent être envisagées. A partir de cette structure de données, il est possible d'afficher à l'écran la couleur du pixel de l'étape i (voir paragraphe précédent). Nous pouvons aussi faire subir de nouvelles opérations au z-buffer étendu et jouer une autre simulation.

Chapitre 5

Evaluation Experimentale

5.1 Mise en place de l'évaluation expérimentale

Pour évaluer les algorithmes, nous avons utilisé un PC de 300 Mhz, de 512 Ko de mémoire cache de niveau 2 et de 128 Mo de RAM. Le logiciel de simulation a été développé en C avec le compilateur icc d'Intel version 1.1A et utilise des accès directs à une carte graphique standard de type VGA, et l'affichage se réalise actuellement en point par point. Les parcours préordre et postordre de l'Interval Treap ont été programmés en utilisant des procédures récursives.

Pour étudier la structure de données la mieux adaptée à chaque problème, nous avons choisi deux fichiers d'usinage significatifs, c'est à dire avec un nombre important de trajectoires (et donc de positions élémentaires) qui répondent à deux stratégies d'usinage différentes :

- Le premier fichier d'usinage est programmé suivant une stratégie d'usinage en zigzag (voir annexe). Il permet de réaliser la simulation de l'usinage d'une petite pièce de moulage en forme de voiture (figure 5.1, page 77). L'usinage de cette pièce nécessite 17 108 trajectoires de $0,5$ mm de long pour la plupart. Il s'effectue sur un brut de dimensions $107 \times 72 \times 33$ mm avec un outil de 3 mm de rayon. L'usinage réel de cette pièce sur une fraiseuse prend environ 1 h 30 min avec une fraiseuse à trois axes.
- Le second fichier d'usinage est programmé suivant une stratégie d'usinage en contour parallèle. Il permet de réaliser la simulation de l'usinage d'une petite pièce de moulage en forme de cendrier (figure 5.2, page 77). L'usinage de cette pièce nécessite 369 trajectoires. Il s'effectue sur un brut de dimensions $50 \times 50 \times 16$ mm avec un outil de 1 mm de rayon.

Nous avons travaillé sur des images de 640 colonnes et 480 lignes (307 200 pixels). Pour la vue considérée :

- un pixel représente 0,26 mm dans le cas de la voiture et
- un pixel représente 0,18 mm dans le cas du cendrier.

Nous avons choisi pour orientation des pièces les angles $\varphi = 45°$ et $\theta = -45°$ (la figure 5.3 de la page 78 donne la correspondance entre les angles d'orientation et la visualisation de la pièce).

Fichiers de simulation

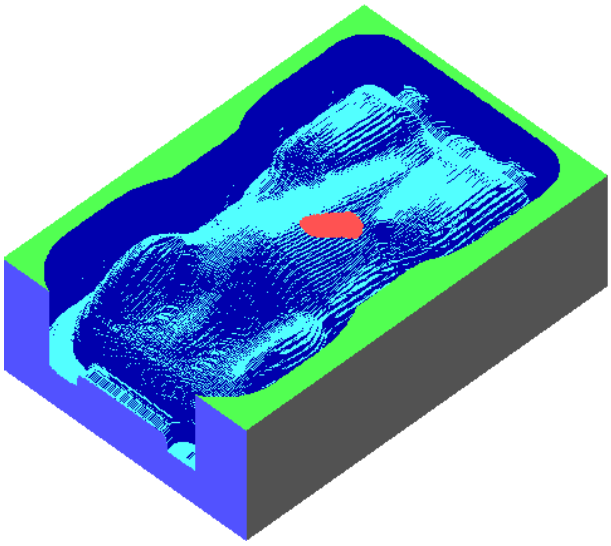

FIG. 5.1 – Usinage d'une petite pièce de moulage en forme de voiture (usinage zigzag)

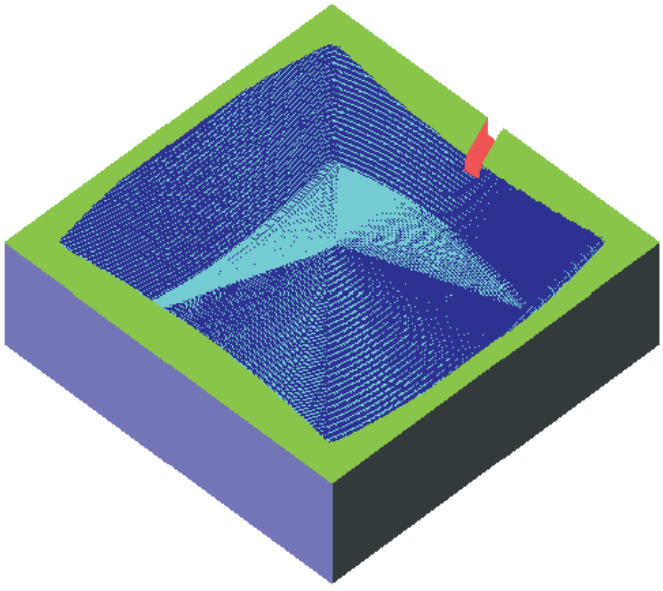

FIG. 5.2 – Usinage d'une petite pièce de moulage en forme de cendrier (usinage contour parallèle)

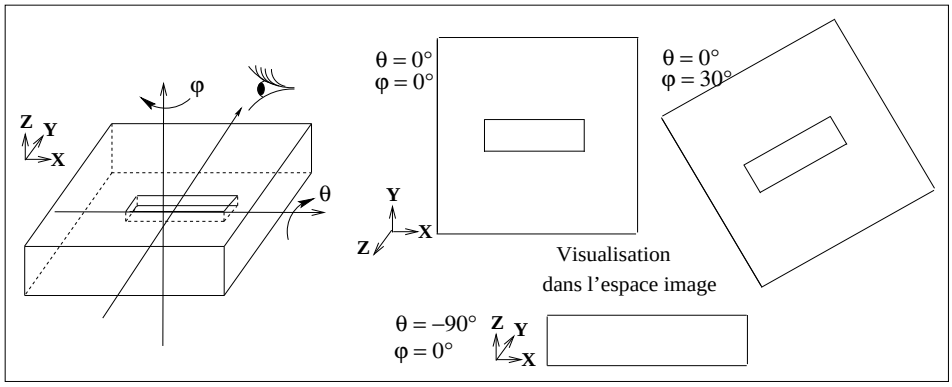

FIG. 5.3 – Angles d'orientation et visualisation correspondante de la pièce dans l'espace image

5.2 Création des structures de données

Tout d'abord, nous nous sommes intéressés à la mémoire et au temps nécessaires pour créer ces quatre structures de données. Nous avons relevé les résultats relatifs :

- à la construction d'un z-buffer étendu sous forme de liste chaînée,
- à la construction d'un z-buffer étendu et d'une trace en Z sous forme de liste chaînée,
- à la construction d'un z-buffer étendu et d'une trace en temps sous forme de liste chaînée,
- et à la construction du z-buffer étendu sous forme d'Interval Treap.

Les résultats de simulation liés à l'occupation mémoire sont regroupés dans les tableaux 5.1 et 5.2 de la page 80. L'allocation mémoire est réalisée sous forme de pointeurs. A l'initialisation, ces structures de données comptent 307 200 dexels, éléments en Z, éléments en temps. Un dexel et un élément en temps occupent 12 octets dont 4 octets pour le pointeur. Un élément en Z et un noeud de l'Interval Treap occupent 16 octets dont 4 octets pour le pointeur de l'élément en Z et 16 octets pour le pointeur du noeud de l'Interval Treap. Pour une image de taille donnée, le nombre d'éléments varie ensuite suivant plusieurs paramètres :

- le brut, de par sa taille et sa forme, occupe une partie plus ou moins importante de l'espace image, ce qui permet de déterminer le nombre de pixels actifs,
- le programme d'usinage intervient de deux manières selon le nombre de trajectoires à simuler, et selon la stratégie d'usinage utilisée,
- l'angle de vue.

Par exemple, le nombre d'éléments en temps représente 72,7% du nombre d'éléments en Z pour la voiture, et seulement 48,2% du nombre d'éléments en Z pour le cendrier.

Cette fraction plus ou moins importante du nombre d'éléments en temps est une illustration de l'influence de la stratégie d'usinage.

Les résultats de simulation liés à la création du z-buffer étendu et la mise en place des traces sous forme de listes chaînées et la mise en place du z-buffer étendu sous forme d'Interval Treap sont regroupés dans le tableau 5.3 de la page 80 pour la voiture et pour le cendrier. Ces temps ne tiennent pas compte des temps d'affichage, seule la création du z-buffer étendu et des traces nous intéresse ici.

5.3 Le z-buffer étendu et les traces

5.3.1 Reconstruction d'une scène quelconque de la simulation

Tout d'abord, nous avons étudié les temps de reconstruction des scènes sans affichage. Vu le grand nombre de positions élémentaires de la simulation (44 084 pour la voiture), nous nous sommes intéressés dans le tableau 5.4 de la page 81 à des scènes de la simulation correspondant à des positions élémentaires arbitraires. Par exemple, dans la première colonne, nous avons étudié la scène dans laquelle la pièce a déjà subi un dixième de la simulation. Lors de la construction du z-buffer étendu, il s'agit de la scène obtenue à la position 4 408 pour la voiture. Pour simplifier, nous appelons désormais cette étape de la simulation : *étape* $\frac{1}{10}$. La figure 5.4 de la page 79 représente l'étape $\frac{1}{2}$ dans laquelle la pièce a déjà subi la moitié de la simulation.

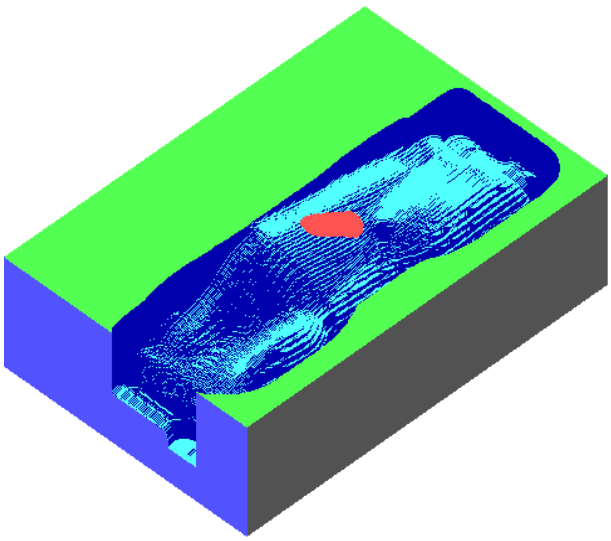

FIG. 5.4 – Etape $\frac{1}{2}$ de la simulation d'usinage de la pièce en forme de voiture

Résultats concernant la création des structures de données

	Trajectoires (Positions)	Pixel du brut	z-buffer étendu		
			Dexels créés	Dexels supprimés	Dexels en tout
Voiture	17 108 (44 084)	121 067	83 217 *0,96Mo*	52 427 *0,60Mo*	390 417 *4,46Mo*
Cendrier	369 (39 563)	80 769	141 718 *1,62Mo*	131 369 *1,50Mo*	448 918 *5,13Mo*

TAB. 5.1 – Occupation mémoire pour le z-buffer étendu simple sous forme de liste chaînée

	Trace en Z	Trace en temps	Interval Treap
	Éléments en Z	Eléments en temps	Noeuds
Voiture	2 991 055 *45,64Mo*	2 174 530 *24,88Mo*	2 922 415 *45,66Mo*
Cendrier	2 082 348 *31,77Mo*	1 004 534 *11,49Mo*	2 132 861 *32,54Mo*

TAB. 5.2 – Occupation mémoire pour les nouvelles fonctionnalités

	z-buffer étendu (liste chaînée)	trace en Z	trace en temps	z-buffer étendu (Interval Treap)
Voiture	99,42	103,04	101,23	341,13
Cendrier	32,46	34,99	33,86	89,05

TAB. 5.3 – Temps de création des structures de données du z-buffer étendu et des traces (en secondes).

Résultats concernant la reconstruction d'une scène de la simulation

		Étape $\frac{1}{10}$	Étape $\frac{1}{4}$	Étape $\frac{1}{2}$	Étape $\frac{3}{4}$	Dernière Étape
Voiture	z-buffer[3]	10,88	26,03	50,14	74,20	99,42
	z-trace[3]	0,56	0,57	0,57	0,58	0,57
	t-trace[4]	0,06	0,09	0,17	0,25	0,31
Cendrier	z-buffer[3]	3,35	8,19	16,37	24,28	32,46
	z-trace[3]	0,39	0,40	0,41	0,40	0,41
	t-trace[4]	0,06	0,08	0,10	0,12	0,13

TAB. 5.4 – Reconstruction d'une scène de la simulation pour une étape donnée (temps en secondes).

Résultats concernant l'animation de la simulation

	$[0,\frac{1}{10}]$	$[\frac{1}{10},\frac{1}{4}]$		$[\frac{1}{4},\frac{1}{2}]$		$[\frac{1}{2},\frac{3}{4}]$		$[\frac{3}{4}$,fin]		[0,fin]
	$[0,\frac{1}{10}]$	$[0,\frac{1}{10}]$	$]\frac{1}{10},\frac{1}{4}]$	$[0,\frac{1}{4}]$	$]\frac{1}{4},\frac{1}{2}]$	$[0,\frac{1}{2}]$	$]\frac{1}{2},\frac{3}{4}]$	$[0,\frac{3}{4}]$	$]\frac{3}{4}$,fin]	[0,fin]
Voiture	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(b)
z-buffer	11,21	10,71	16,31	25,81	27,36	50,81	27,24	76,02	26,86	108,86
t-trace	1,49	0,55	2,03	0,61	7,19	0,66	7,19	0,71	7,53	24,44
Cendrier	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(b)
z-buffer	4,29	3,85	5,65	8,78	8,57	16,91	8,68	25,05	9,01	36,47
t-trace	1,60	0,55	1,64	0,60	2,03	0,61	2,42	0,65	1,92	9,73

TAB. 5.5 – Animation pour le cas de la voiture (temps en secondes).

[3] Pour une simulation.
[4] Moyenne pour 100 simulations.

Dans le tableau 5.4 de la page 81, nous avons relevé le temps nécessaire pour une étape quelconque : pour la construction du z-buffer étendu, pour la reconstruction de la scène avec la trace en Z, et pour la reconstruction de la scène avec la trace en temps.

Pour commencer, nous nous intéresserons aux résultats de simulation concernant la simulation de la voiture et répertoriés dans le tableau 5.4 de la page 81. Ces résultats mettent en évidence l'utilité et l'efficacité des traces. La trace en Z est construite en 103,04 s (tableau 5.3 de la page 80) soit seulement 3,6% de plus que la seule construction du z-buffer étendu. Mais la trace en Z nécessite 45,64 Mo soit 921,5% de plus que le z-buffer étendu. La construction de la trace en Z ne nécessite donc qu'un supplément de temps négligeable par rapport au temps de construction du z-buffer étendu, en contrepartie, l'occupation mémoire est bien plus importante dans le cas de la trace que pour le seul z-buffer étendu.
Pour reconstruire une scène quelconque, la trace en Z n'utilise dans le pire des cas que 5,2% du temps nécessaire pour construire la même scène avec le z-buffer étendu. Nous devons donc faire un choix entre la mémoire et le temps.
Nous pouvons de même examiner le cas de la trace en temps. Cette dernière est construite en 101,23 s soit seulement 1,8% de plus que la seule construction du z-buffer étendu. Elle occupe 24,88 Mo soit 456,9% de plus que le z-buffer étendu.
Pour reconstruire une scène quelconque dans le pire des cas, la trace en temps n'utilise dans le pire des cas que 0,5% du temps nécessaire pour construire la même scène avec le z-buffer étendu.

Bien que la trace en Z et la trace en temps soient totalement indépendantes, la trace en temps est plus rapide à mettre en place que la trace en Z. En effet, elle n'est actualisée que dans certains cas (cf. règle 6 de la page 55), alors que la trace en Z suit toutes les étapes d'actualisation du z-buffer étendu. D'autre part, dans la trace en temps, les éléments en temps sont déjà ordonnés lors de leur création, alors que pour la trace en Z, les éléments Z sont à ordonner dans la liste. En utilisant la trace en temps la scène est reconstruite plus rapidement. En effet, nous n'avons pas besoin d'explorer toute cette trace pour retrouver la valeur souhaitée (la couleur), contrairement à la trace en Z où tous les éléments en Z doivent être parcourus pour vérifier s'ils respectent les inégalités permettant la reconstruction de la scène, et cela quelque soit l'étape choisie. La trace en temps occupe moins de mémoire que la trace en Z. Elle paraît donc plus adaptée pour reconstruire rapidement une scène, mais la trace en Z a néanmoins l'avantage de recréer un z-buffer étendu, ce qui permet d'aller au-delà des observations et de recommencer la simulation avec des trajectoires différentes. Comme pour la trace en Z, il faut choisir entre le temps

et les possibilités d'actions.

Les résultats obtenus pour la simulation d'usinage du cendrier (tableau 5.4 de la page 81) confirment les observations et remarques précédentes. En effet, dans ce cas, la trace en Z est construite en seulement 7,8% de temps de plus que le temps nécessaire à la seule construction du z-buffer étendu. La trace en Z nécessite aussi 499,4% de mémoire de plus que le z-buffer étendu.
Pour reconstruire une scène quelconque, la trace en Z n'utilise dans le pire des cas que 4,3% du temps nécessaire pour construire la même scène avec le z-buffer étendu. Quant à la trace en temps, elle est construite en un temps de 4,3% de plus que le temps nécessaire à la construction du z-buffer étendu, et occupe 123,9% de mémoire de plus que ce dernier. Pour reconstruire une scène quelconque dans le pire des cas, la trace en temps n'utilise dans le pire des cas que 1,8% du temps nécessaire pour construire la même scène avec le z-buffer étendu.

Les résultats précédents montrent l'efficacité des structures de données : la reconstruction d'une étape quelconque d'une trace (en temps ou en Z) prend moins de temps dans tous les cas que la construction du z-buffer étendu jusqu'à cette étape, c'est pourquoi nous allons utiliser la trace en temps (la plus rapide) pour réaliser une simulation interactive et revisualiser la simulation de la pièce entre deux étapes quelconques choisies par l'utilisateur.

5.3.2 Animation de la simulation

Nous avons étudié l'animation de deux manières, en affichant successivement toutes les étapes à l'écran : tout d'abord, la simulation est exécutée en utilisant le z-buffer étendu, ensuite la trace en temps est utilisée. L'animation est réalisée entre deux étapes E_n et E_m, elle est notée par l'intervalle $[E_n,E_m]$ dans le tableau 5.5 de la page 81. Par exemple, pour la colonne qui contient l'intervalle $[\frac{1}{10},\frac{1}{4}]$, l'animation commence par une image correspondant à un dixième des trajectoires ($E_n=\frac{1}{10}$) et elle se termine lorsque le quart des trajectoires à été traité ($E_m=\frac{1}{4}$).
Nous avons donc rejoué la simulation d'une étape E_n à une étape E_m. Lorsque $E_n \neq 0$, une phase préliminaire est nécessaire. Elle permet de repérer dans la structure de données la première étape de la simulation (E_n). A partir de là, l'animation proprement dite (ce que voit l'utilisateur) peut commencer et se dérouler de l'étape E_n à l'étape E_m. C'est pourquoi nous avons relevé les temps d'exécution dans les deux phases suivantes :

- de 0 à E_n (si $E_n \neq 0$) (colonne (a)). Pour la première ligne, le temps d'exécution inclue la construction du z-buffer étendu jusqu'à l'étape E_n et l'affichage de

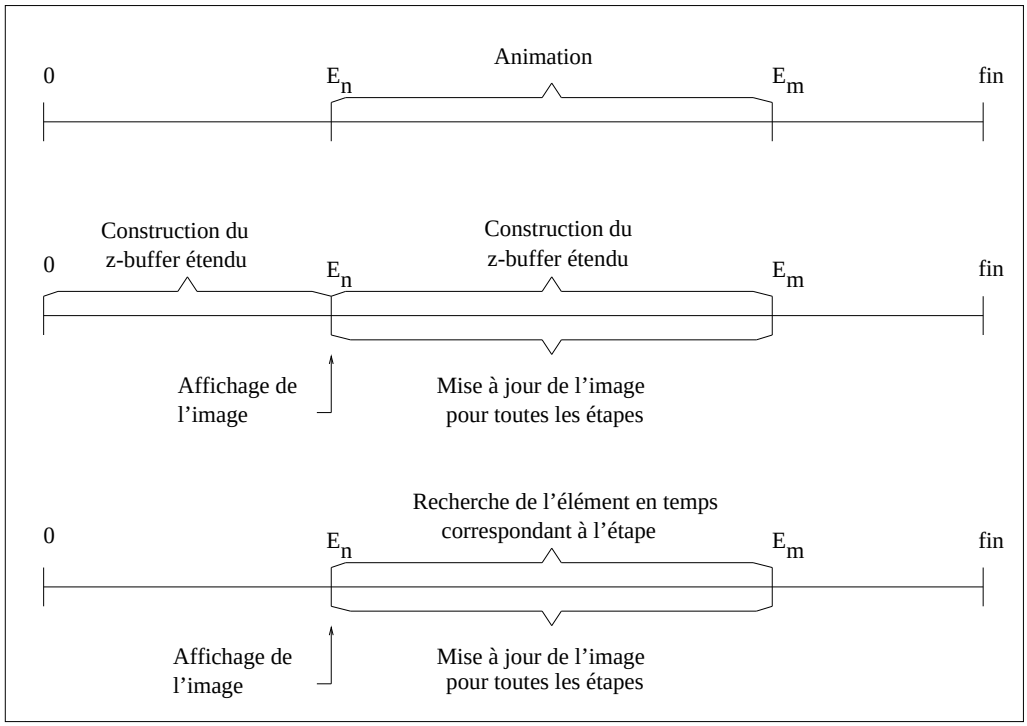

FIG. 5.5 – Mise en place d'une animation d'une étape E_n à une étape E_m

l'image correspondant à l'étape E_n. Dans la seconde ligne, le temps d'exécution correspond à la recherche dans la trace de l'élément en temps conforme à l'étape E_n et à l'affichage de la couleur de cet élément et cela pour chaque pixel.

– de E_n à E_m (colonne (b)). Pour les étapes E_i telles que $E_n < E_i \leq E_m$, le temps d'exécution de la première ligne comprend l'évolution du z-buffer étendu de l'étape E_n à l'étape E_m et la mise à jour de l'image pour toutes les étapes E_i. Dans la seconde ligne, le temps d'exécution tient compte de la recherche dans la trace de la position de l'élément en temps correspondant aux étapes E_i et de l'affichage de la mise à jour de l'image. Si E_n=0 alors nous affichons au début de la simulation l'image correspondant à l'étape 0, c'est à dire le brut.

Remarquons que si nous totalisons les colonnes étiquetées (b) en excluant la dernière, nous trouvons un résultat cohérent avec celui de la dernière colonne qui correspond à une simulation complète.

Ces résultats mettent en évidence l'utilité d'une trace en temps pour la mise en place d'une animation. En effet, l'animation obtenue à partir de la trace en temps est quasiment dix fois plus rapide pour les étapes du début d'usinage que celle obtenue à partir du z-buffer étendu. Si nous resimulons l'usinage dans son intégralité, la trace en temps est 4,4 fois plus rapide qu'avec le z-buffer étendu pour la voiture et 3,7 fois plus rapide pour le cendrier.
Dans le cas de la voiture par exemple, le z-buffer étendu est généré par 17 108 trajectoires en 108,86 s, soit 6,36 ms par trajectoire. La structure de données de la trace en temps garde les informations nécessaires à la simulation des 17 108 trajectoires qui est alors exécutée en 24,44 s, soit 1,43 ms par trajectoire. La trace en temps permet d'obtenir une animation plus rapide, plus efficace et donc plus conviviale pour l'opérateur. Dans le tableau 5.5 de la page 81, les scènes ont été rejouées de manière chronologique, mais l'animation pourrait également *remonter le temps* et *rembobiner* la trace : elle commencerait alors par des scènes de fin d'usinage et se finirait par des scènes de début d'usinage. En remontant ainsi pas à pas dans la simulation, on pourrait repérer précisément les étapes de la simulation où une erreur d'usinage se produit. La solution algorithmique consiste à transformer les listes ordonnées chaînées d'éléments en temps associées à chaque pixel en listes ordonnées doublement chaînées avec un léger surcoût en temps lors de la construction et un surcoût de mémoire de 33,3% par rapport à la liste simplement chaînée dû à l'introduction d'un pointeur supplémentaire.

5.4 Le z-buffer étendu sous forme d'Interval Treap

5.4.1 Création de l'Interval Treap et des Traces

Pour la voiture, la création du z-buffer étendu s'effectue en 99,42 s, tandis que la construction de l'Interval Treap nécessite 341,13 s. Ces temps ne tiennent pas compte des temps d'affichage. La structure de donnée de l'Interval Treap met donc 3,43 fois plus de temps pour être mise en place que la structure de liste chaînée. Pour le cendrier, la structure de donnée de l'Interval Treap met que 2,74 fois plus de temps pour être mise en place que la structure de liste chaînée.

Les résultats de simulation liés à l'occupation mémoire sont regroupés dans les tableaux 5.1-5.2 de la page 80. A la fin de la simulation de la voiture, l'Interval Treap occupe 10 fois plus d'espace mémoire que tous les dexels qui ont été nécessaire pour le mise en place du z-buffer étendu sous forme de liste chaînée. Mais en associant la z-trace et la t-trace au z-buffer étendu, nous pouvons reconstruire le z-buffer étendu et/ou afficher rapidement une scène quelconque de la simulation. Nous atteignons alors une occupation mémoire de $4,46 + 45,64 + 24,88$ soit 74,98 Mo alors que l'Interval Treap qui permet aussi de reconstruire et/ou d'afficher une scène quelconque

de la simulation n'occupe que 45,66 Mo, soit un gain mémoire de 40 %. Pour le cendrier, le gain mémoire est de 32,7 %, puisque l'occupation mémoire de l'Interval Treap seul représente 32,54 Mo alors que l'occupation mémoire combinée du z-buffer et des traces représente $5,13 + 31,77 + 11,49$ soit 48,39 Mo.

5.4.2 Reconstruction de scènes quelconques de la simulation

Ensuite, comme précédemment, nous nous sommes intéressés à des scènes de la simulation correspondant à des positions élémentaires arbitraires. Lors de la construction du z-buffer sous forme de liste chaînée, les étapes $\frac{1}{10}$, $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$ et la dernière étape nécessitent respectivement 10,88 s, 26,03 s, 50,14 s, 74,20 s et 99,42 s pour la voiture et 3,35 s, 8,19 s, 16,37 s, 24,28 s et 32,46 s pour le cendrier, c'est à dire de l'ordre de plusieurs dizaines de secondes. Nous avons relevé sur les figures 5.6 et 5.7 de la page 87 le temps nécessaire pour la mise en place de l'affichage d'une scène quelconque par la t-trace et l'Interval Treap (IT(aff)) et la reconstruction d'une scène quelconque par la z-trace et l'Interval Treap (IT(rec)) pour la simulation de la voiture et du cendrier. Ces temps ne représentent plus que des dixièmes de seconde.

Pour afficher une scène quelconque de la simulation de la voiture, l'Interval Treap met, dans le pire des cas, 3,7 fois plus de temps que la t-trace, ce qui ne représente que 2,2% du temps nécessaire pour construire la même scène avec le z-buffer étendu. L'affichage d'une scène avec un Interval Treap est moins rapide que l'affichage de la même scène avec la t-trace car il faut tenir compte de l'exploration des feuilles vides de l'Interval Treap qui peuvent être plus ou moins nombreuses suivant le point de vue choisi.
Pour reconstruire une scène quelconque, l'Interval Treap met dans le meilleur des cas 30% de moins que le temps nécessaire pour reconstruire la même scène avec la z-trace et dans le pire des cas, l'Interval Treap ne met que 40% plus de temps que la z-trace. L'exploration des *feuilles vides*, qui n'apparaissent pas dans la structure de la z-trace, explique pourquoi les temps de reconstruction avec l'Interval Treap sont plus importants pour les scènes de fin de simulation. Mais, dans le pire des cas, l'Interval Treap n'utilise que 2,0% du temps nécessaire pour construire la même scène avec le z-buffer étendu. Notons qu'il suffit de recréer seulement 4 scènes de fin de simulation pour valider la structure de l'Interval Treap.

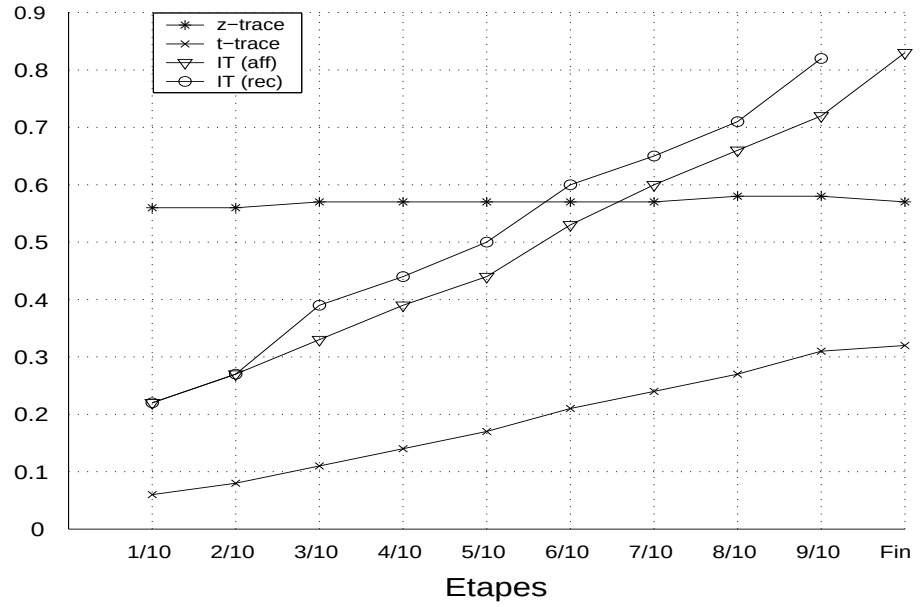

FIG. 5.6 – Reconstruction d'une scène pour la voiture (en secondes)

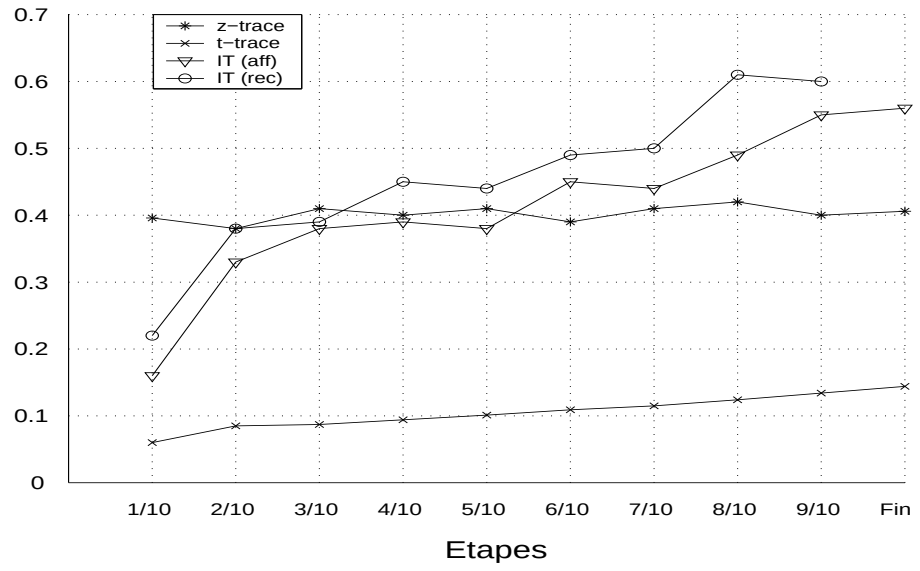

FIG. 5.7 – Reconstruction d'une scène pour le cendrier (en secondes)

La simulation du cendrier confirme les observations et remarques précédentes. Pour afficher une scène quelconque de la simulation du cendrier, l'Interval Treap met, dans le pire des cas, 4,7 fois plus de temps que la t-trace, ce qui ne représente que 3,8% du temps nécessaire pour construire la même scène avec le z-buffer étendu. Pour reconstruire une scène quelconque, l'Interval Treap met dans le meilleur des cas 40% de moins que le temps nécessaire pour reconstruire la même scène avec la z-trace et dans le pire des cas, l'Interval Treap ne met que 50% plus de temps que la z-trace. Mais, dans le pire des cas, l'Interval Treap n'utilise que 6,6% du temps nécessaire pour construire la même scène avec le z-buffer étendu. Notons qu'il suffit de recréer seulement 3 scènes de fin de simulation pour valider la structure de l'Interval Treap.

Modèles d'Interval Treap potentiels au cours d'une simulation

Nous nous sommes ensuite intéressés aux différentes formes d'arbres pouvant être obtenus au cours d'une simulation. Ils sont représentés sur les figures 5.8 et 5.9 de la page 89. Le nombre de noeuds, la hauteur moyenne, la hauteur maximale permettent de définir l'équilibre de l'arbre[1]. L'équilibre d'un Interval Treap dépend des paramètres de simulation, notamment de l'angle de vue (orientation de la pièce dans l'espace image) et de la stratégie d'usinage. Par exemple, l'arbre non équilibré dit de modèle 1 est obtenu pour un usinage en zigzag. Par symétrie, on obtient également un arbre non équilibré dit de modèle 3, obtenu aussi pour un usinage en zigzag. Un usinage de type contour parallèle produirait un arbre se rapprochant d'un Interval Treap équilibré de modèle 2. Les Interval Treap non équilibrés de modèle 1 sont obtenus pour un usinage en zigzag, et c'est sur cet exemple qui n'est pas le plus favorable pour notre structure de données que nous avons étudié les performances de cette nouvelle implémentation pour le z-buffer étendu. Ce déséquilibre est mis en évidence par des temps très proches pour l'affichage et la reconstruction d'une scène à l'aide d'un Interval Treap (IT). En effet, pour reconstruire un IT à une étape i de la simulation à partir d'un IT équilibré de modèle 2, il faut parcourir toutes les branches de cet arbre, ce qui nécessite de nombreux retours en arrière, contrairement à l'affichage de la même scène i qui ne nécessite qu'une exploration partielle de l'arbre jusqu'au noeud cherché (qui est en fait la dernière feuille de l'IT tel qu'il était à l'étape i). Pour les arbres déséquilibrés de modèle 1 et 3, les retours en arrière sont

[1] La hauteur d'un noeud n d'un arbre est la longueur du plus long chemin entre la racine de l'arbre et le noeud n. La hauteur moyenne est la moyenne de toutes les hauteurs de l'arbre. La hauteur maximale, appelée aussi hauteur de l'arbre, est la plus grande hauteur que peut avoir un noeud quelconque de l'arbre
Un arbre binaire est dit équilibré si pour chaque noeud, les hauteurs des sous-arbres gauche et droit différent d'un au plus. [38, 51]

nettement moins nombreux, l'exploration partielle de l'arbre et l'exploration totale ne diffèrent que de très peu de branches. Ainsi, pour la reconstruction d'une scène, et notamment pour les étapes de début et de milieu de simulation, l'exploration d'un Interval Treap déséquilibré de modèle 1 ou 3 peut être plus rapide que celle d'un Interval Treap équilibré, puisqu'il y aura moins de branches à explorer.

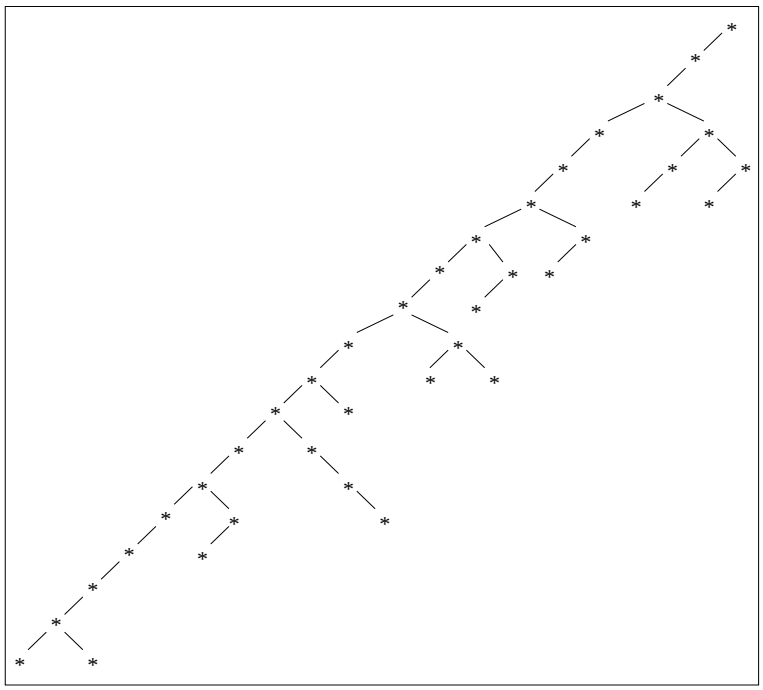

FIG. 5.8 – Interval Treap non équilibré obtenu au cours de la simulation dit de modèle 1

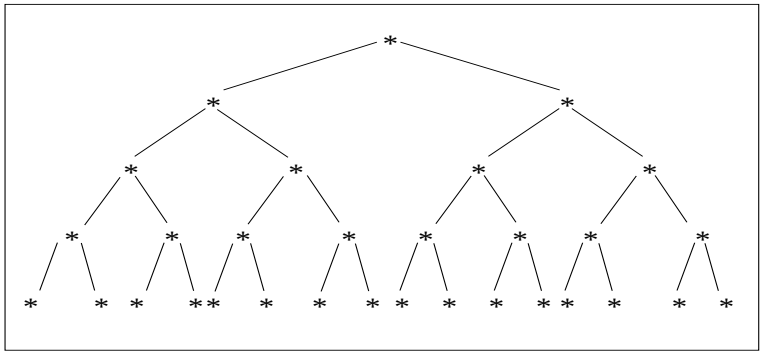

FIG. 5.9 – Interval Treap équilibré obtenu au cours de la simulation dit de modèle 2

5.4.3 Etude Analytique

L'analyse d'algorithme fait référence à deux conceptions différentes pour l'étude des performances d'un programme. La première approche cherche à déterminer l'ordre de grandeur des performances d'un algorithme en considérant le pire des cas. Nous prouvons alors que pour n'importe quel algorithme résolvant un problème particulier, il existe une *borne inférieure*, cette borne coincide avec les performances de l'algorithme dans le pire des cas. C'est cette approche qui nous permet de dire que dans le pire des cas la recherche d'un élément dans une liste chaînée de N éléments est en $O(N)$. La seconde approche consiste à caractériser de façon rigoureuse les performances d'un algorithme en les analysant dans le meilleur des cas, le cas moyen et le pire des cas avec des méthodes permettant d'affiner le précision. Cette approche repose sur le dénombrement précis des différentes configurations d'une structure de données. Sedgewick [52] utilise le fait que les algorithmes peuvent généralement être formulés en termes de procédures récursives ou itératives pour aboutir à des récurrences permettant soit de décrire le cas moyen, soit de borner les performances dans le meilleur ou le pire des cas. Comme les relations de récurrences peuvent être résolues à l'aide de séries génératrices (transformée en Z), Sedgewick utilise maintenant ces séries comme un outil de dénombrement systématique d'objets combinatoires, où l'objet combinatoire représente une structure de données manipulée par un algorithme.
Nous nous intéressons ici à cette dernière approche puisque notre étude porte sur la calcul et l'évaluation de la hauteur moyenne dans un Interval Treap afin de caractériser et de classifier au mieux l'équilibre de ces arbre de recherche. Dans le meilleur des cas, l'arbre de N noeuds est équilibré, sa hauteur maximale est en $\log_2 N$ et les opérations d'insertion et de recherche d'un élément peuvent être implémentées en un temps $\log_2 N$. Dans le pire des cas, l'arbre est complètement déséquilibré (un noeud par niveau), sa hauteur maximale est en N et les opérations d'insertion et de recherche d'un élément peuvent être implémentées en un temps N. Ainsi, le caractère équilibré de l'arbre a une influence sur le temps de construction de l'Interval Treap et le temps nécessaire pour l'affichage ou la reconstruction d'une scène de la simulation qui se ramène à la recherche d'éléments dans l'IT. Prenons par exemple l'arbre de modèle 1 de la figure 5.8 de la page 89 qui est un Interval Treap obtenu au cours de la simulation de la voiture. Cet arbre a 38 noeuds ce qui est voisin du nombre moyen de noeuds des Interval Treap de la simulation. Dans le meilleur des cas, l'arbre de recherche est équilibré. La hauteur moyenne pour un arbre de recherche équilibré de 38 noeuds est de $H_{equilibre} = 3,5$. Dans le pire des cas, l'arbre de recherche est complètement déséquilibré. La hauteur moyenne pour un arbre de recherche complètement déséquilibré de N noeuds est de $H_{non\ equilibre} = \frac{N-1}{2}$, donc

pour un arbre de 38 noeuds $H_{non\ equilibre} = 18,5$. Pour le cas moyen, Sedgewick [52] a étudié les arbres binaires de recherche obtenus à partir d'un tirage aléatoire de leurs clés. Il en a déduit la formule suivante pour la hauteur moyenne d'un arbre de N noeuds : $H_{aleatoire} = 4,311 \times \ln(N) + o(\ln(N))$, donc pour un arbre de 38 noeuds $H_{aleatoire} = 15,68$. La hauteur moyenne calculée sur l'Interval Treap de modèle 1 de la figure 5.8 de la page 89 est : $H_{IT1} = 9,11$. Cette valeur est à peu près deux fois celle d'un arbre complètement déséquilibré, elle se rapproche de la hauteur moyenne d'un arbre équilibré. L'utilisation de l'Interval Treap comme nouvelle structure de données du z-buffer étendu est donc intéressante, même dans un cas comme celui de l'exemple qui est défavorable.

5.5 Utilisation des Skip lists

Comme nous l'avons proposé dans le chapitre précédent, nous allons étudier l'influence des skip lists uniquement sur les traces, puisque dans le cas du z-buffer étendu sous forme de liste chaînée, il s'agit de retrouver un intervalle et non un point.

5.5.1 Structures de données pour la trace en Z

L'utilisation de skip lists peut être intéressante lors de la construction de la trace en Z. En effet, lors de la construction de cette trace, un certain nombre d'éléments en Z sont créés, puis mis à jour mais aucun élément n'est supprimé. Dans le tableau 5.6 de la page 91, nous avons relevé les temps de construction des traces en Z sous forme de liste chaînée simple et de skip list aléatoire.

Pièce usinée	Type d'usinage	z-trace liste chaînée	z-trace skip list
Voiture	zigzag	103,04	104,97
Cendrier	contour parallèle	34,99	33,49

TAB. 5.6 – Comparaison liste chaînée et skip list pour la trace en Z (en secondes)

La trace en Z sous forme de skip list est donc plus rapide pour traiter des fichiers d'usinage écrits à partir d'une stratégie de type contour parallèle et la trace en Z sous forme de liste chaînée est plus efficace pour les fichiers d'usinage écrits à partir d'une stratégie de type zigzag.

5.5.2 Structures de données pour la trace en temps

L'utilisation d'une skip list pour la trace en temps intervient au moment de l'affichage d'une scène grâce à cette nouvelle fonctionnalité. En effet, lors de la construction de la trace en temps, l'élément de base de la trace (le dernier FarZ modifié du z-buffer étendu) vient toujours s'insérer en bout de liste. L'insertion de ce nouvel élément est facilement réalisable si on mémorise toujours le dernier élément courant de la trace. C'est pourquoi, nous avons choisi de construire tout d'abord la trace en temps sous forme de liste chaînée simple. Ce n'est qu'après la totale construction de la trace en temps que nous modifierons la structure de données.
L'opération de recherche d'un élément dans la trace en temps est la seule opération utilisée lors de l'affichage d'une scène. Les skips lists sont des structures de données plus performantes que les listes chaînées pour ce type d'opération. C'est pourquoi, nous avons choisi d'implémenter la trace en temps sous forme de skip list déterministe presque parfaite et de skip list aléatoire.

5.5.2.1 Utilisation de skip lists déterministes presque parfaites

Pour étudier les performances de la trace en temps sous forme de skip lists déterministes, nous avons choisi de répartir les sauts uniformément le long de la skip list. Nous obtenons de manière déterministe une skip list parfaite où un noeud de niveau i pointe sur le (2^i)ème noeud suivant. Les tests ont été effectués sur des skip lists presque parfaites de hauteur maximale allant de 1 à 6. La skip list parfaite de hauteur maximale $h = 1$ correspond à une simple liste chaînée. La skip list parfaite de hauteur maximale $h = 6$ saute au plus 32 noeuds de la skip list (soit $2^{h-1} = 2^5$ noeuds). Les skip lists obtenues au cours de la simulation sont en fait des skip lists *presque* parfaites car elles contiennent un nombre quelconque de noeuds, pas obligatoirement égal à une puissance de deux. Dans une telle skip list de n éléments, nous obtiendrons ($\lfloor x \rfloor$ correspond à la partie entière de x par défaut) :

Mise en place d'une trace en temps
sous forme de skip lists presque parfaites (voiture)

	Noeuds L_1	Noeuds L_2	Noeuds L_3	Noeuds L_4	Noeuds L_5	Noeuds L_6
Ttrace L_6	950 901	475 623	237 294	118 630	60 289	331 793
Ttrace L_5	950 901	475 623	237 294	118 630	392 082	–
Ttrace L_4	950 901	475 623	237 294	510 712	–	–
Ttrace L_3	950 901	475 623	748 006	–	–	–
Ttrace L_2	950 901	1 223 629	–	–	–	–
Ttrace L_1	2 174 530	–	–	–	–	–

TAB. 5.7 – Répartition du nombre de noeuds par niveaux pour la voiture

	Mémoire Statique	Mémoire Dynamique
Ttrace L_6	69 584 960 (*soit 66,36Mo*)	38 919 248 (*soit 37,12Mo*)
Ttrace L_5	60 886 840 (*soit 58,07Mo*)	37 592 076 (*soit 35,85Mo*)
Ttrace L_4	52 188 720 (*soit 49,77Mo*)	36 023 748 (*soit 34,35Mo*)
Ttrace L_3	43 490 600 (*soit 41,47Mo*)	33 980 900 (*soit 32,41Mo*)
Ttrace L_2	34 792 480 (*soit 33,18Mo*)	30 988 876 (*soit 29,55Mo*)
Ttrace L_1	26 094 360 (*soit 24,88Mo*)	26 094 360 (*soit 24,88Mo*)

TAB. 5.8 – Occupation mémoire des skip lists pour la voiture (en octets)

Ttrace L_1	Ttrace L_2	Ttrace L_3	Ttrace L_4	Ttrace L_5	Ttrace L_6
–	0,38	0,49	0,61	0,72	0,82

TAB. 5.9 – Construction la trace en temps sous forme de skip list presque parfaites pour la voiture (en secondes)

Mise en place d'une trace en temps
sous forme de skip lists presque parfaites (cendrier)

	Noeuds L_1	Noeuds L_2	Noeuds L_3	Noeuds L_4	Noeuds L_5	Noeuds L_6
Ttrace L_6	359 370	179 153	88 841	42 164	19 159	315 487
Ttrace L_5	359 370	179 153	88 841	42 164	335 006	–
Ttrace L_4	359 370	179 153	88 841	377 710	–	–
Ttrace L_3	359 370	179 153	466 011	–	–	–
Ttrace L_2	359 370	645 164	–	–	–	–
Ttrace L_1	1 004 534	–	–	–	–	–

TAB. 5.10 – Répartition du nombre de noeuds par niveaux pour le cendrier

	Mémoire Statique	Mémoire Dynamique
Ttrace L_6	32 145 088 (*soit 30,66Mo*)	20 599 680 (*soit 19,64Mo*)
Ttrace L_5	28 126 952 (*soit 28,82Mo*)	19 347 812 (*soit 18,45Mo*)
Ttrace L_4	24 108 816 (*soit 22,99Mo*)	18 007 788 (*soit 17,17Mo*)
Ttrace L_3	20 090 680 (*soit 19,16Mo*)	16 499 108 (*soit 15,73Mo*)
Ttrace L_2	16 072 544 (*soit 15,33Mo*)	14 635 064 (*soit 13,96Mo*)
Ttrace L_1	12 054 408 (*soit 11,49Mo*)	12 054 408 (*soit 11,49Mo*)

TAB. 5.11 – Occupation mémoire des skip lists pour le cendrier (en octets)

Ttrace L_1	Ttrace L_2	Ttrace L_3	Ttrace L_4	Ttrace L_5	Ttrace L_6
–	0,16	0,22	0,27	0,33	0,38

TAB. 5.12 – Construction la trace en temps sous forme de skip list presque parfaites pour le cendrier (en secondes)

- $l_6 = \lfloor \frac{n}{2^5} \rfloor$ noeuds pour le niveau 6,
- $l_5 = \lfloor \frac{n}{2^4} \rfloor - l_6$ noeuds pour le niveau 5,
- $l_4 = \lfloor \frac{n}{2^3} \rfloor - l_6 - l_5$ noeuds pour le niveau 4,
- $l_3 = \lfloor \frac{n}{2^2} \rfloor - l_6 - l_5 - l_4$ noeuds pour le niveau 3,
- $l_2 = \lfloor \frac{n}{2} \rfloor - l_6 - l_5 - l_4 - l_3$ noeuds pour le niveau 2,
- $l_1 = \lfloor \frac{n}{2} \rfloor$ noeuds pour le niveau 1.

Tout d'abord, dans les tableaux 5.7 et 5.10, nous avons relevé la répartition des différents noeuds pour chaque skip list construite. Nous appelons *Noeuds* L_k, le nombre de noeuds de niveau k présents dans la skip list. Nous appelons *Ttrace* L_k, une trace en temps mise sous la forme d'une skip list presque parfaite de hauteur maximale k. La Ttrace L_1 correspond en fait à la liste chaînée standard.

En ce qui concerne l'occupation mémoire, rappelons qu'un élément en temps de la trace en temps simple sous forme de liste chaînée occupe 12 octets, et un pointeur supplémentaire occupe 4 octets. Deux méthodes peuvent alors être envisagées pour transformer une t-trace en skip list presque parfaite. La première méthode consiste à allouer la mémoire de manière *dynamique*. Pour chaque nouveau pointeur, la mémoire est incrémentée de 4 octets. La seconde méthode consiste à allouer la mémoire de manière *statique*. Pour une skip list presque parfaite de niveau k, tous les noeuds occuperont : $(12 + 4 \times k)$ octets. A partir des tableaux 5.7 et 5.10, nous pouvons calculer l'occupation mémoire et reporter ces résultats dans les tableaux 5.8 et 5.11. Pour notre application, nous avons choisi une implémentation statique de la mémoire, ce qui permet de construire plus rapidement les skip lists. En effet, pour un nouveau niveau, seul le champ correspondant au pointeur de ce niveau doit être mis à jour, il n'y a pas de nouvelle allocation mémoire à mettre en oeuvre au cours de l'obtention de la skip list. Une skip list de niveau k, construite de manière *statique* occupera $\frac{2+k}{3}$ fois plus de mémoire que la liste chaînée simple.

Ensuite, nous nous sommes intéressés au temps supplémentaire nécessaire pour transformer la liste chaînée en skip list. Ces résultats sont regroupés dans les tableaux 5.9 et 5.12. Pour la voiture, la création du z-buffer nécessite 99,42 secondes, tandis que la création du z-buffer et la création de la trace en temps nécessite 101,23 secondes. On en déduit que la création de la trace en temps s'effectue en 1,81 secondes. Dans le meilleur des cas, la transformation de la trace en temps de liste

chaînée en skip list de niveau 2 s'effectue en 0,38 seconde, soit 21 % du temps nécessaire pour créer cette trace en temps. Dans le pire des cas, la transformation de la trace en temps de liste chaînée en skip list de niveau 6 s'effectue en 0,82 seconde, soit 45,3 % du temps nécessaire pour créer cette trace en temps.
Pour le cendrier, la création du z-buffer nécessite 33,86 secondes, tandis que la création du z-buffer et la création de la trace en temps nécessite 32,46 secondes. On en déduit que la création de la trace en temps s'effectue en 1,40 seconde. Dans le meilleur des cas, la transformation de la trace en temps de liste chaînée en skip list de niveau 2 s'effectue en 0,16 seconde, soit 15,7 % du temps nécessaire pour créer cette trace en temps. Dans le pire des cas, la transformation de la trace en temps de liste chaînée en skip list de niveau 6 s'effectue en 0,38 seconde, soit 27,1 % du temps nécessaire pour créer cette trace en temps.
Quelle que soit la pièce usinée, la mise en place de la transformation de la trace en temps en skip lists presque parfaites est très rapide.

Ensuite, nous avons étudié les temps de reconstruction des scènes sans affichage. Pour la recherche d'un élément de valeur v_f dans la skip list, nous avons optimisé l'algorithme présenté à la page 64. En effet, on remarque qu'entre deux noeuds de niveau i, on trouve toujours :

- 1 noeud de niveau i-1,
- 2 noeuds de niveau i-2,
- 4 noeuds de niveau i-3,
- ...,
- 2^{j-1} noeuds de niveau i-j $(1 < j < i)$

En effet, considérons que le $n$ème noeud de la skip list soit un noeud de niveau i. Appelons ce noeud N_1. Le noeud suivant de niveau i dans la skip list est donc le $n + 2^i$ ème noeud de la skip list. Appelons ce noeud N_3. Le noeud N_1 est aussi un noeud de niveau $i-1$ dans la skip list. Le noeud suivant de niveau $i-1$ est le $n+2^{i-1}$ ème noeud de la skip list. Appelons ce noeud N_2. Le noeud suivant de niveau $i-1$ est le $(n + 2^{i-1}) + 2^{i-1}$ soit $n + 2^i$ ème noeud de la skip list, c'est à dire le noeud N_3, qui est aussi un noeud de niveau i.
Nous pouvons en déduire la règle suivante sur la skip list *presque* parfaite :
Dans une skip list presque parfaite, entre deux noeuds de niveau i, il y aura au plus un noeud de niveau i-1.

Recherchons une étape de création dans la skip list et supposons que l'exploration des noeuds de niveau i est terminée et intéressons nous au niveau $i-1$. D'après la règle précédente, pour le niveau $i-1$, la consultation de l'étape de création d'un seul noeud est nécessaire pour se déplacer dans la skip list. En effet, si on examinait

l'étape de création de deux noeuds de niveau i-1, cela reviendrait à *sauter* deux noeuds de niveau i-1, donc à se retrouver sur un noeud de niveau i dont l'étape de création a déjà examinée lors de l'exploration du niveau i. L'algorithme précédent dû à Pugh peut être optimisé.
Pour le parcours de la skip list sur le niveau maximum, il faut *sauter* un certain nombre de noeuds pour se positionner correctement au plus près du noeud contenant la valeur v_f cherchée. Le nombre de noeuds *sautés* est compris entre 1 et N_{max} où $N_{max} = \lfloor \frac{n}{2^{h_{max}-1}} \rfloor$ est le nombre de noeuds de niveau maximum (h_{max}) dans une skip list de n noeuds.
Pour les parcours de la skip list sur les niveaux i inférieurs au niveau maximum, il suffit juste de tester l'étape de création du noeud suivant de niveau $i-1$. Si cette étape de création est inférieure à l'étape de création cherchée, ce noeud devient le nouveau noeud courant.

ReconstructionScene (EtapeCreationCherchee)
x :=TourTeteDeListe
tant que (x.suivant[NiveauMax].EtapeCreation < EtapeCreationCherchee) **faire**
x :=x.suivant[NiveauMax]
fin tant que
pour i := (NiveauMax-1) à 2 **faire**
si (x.suivant[i].EtapeCreation < EtapeCreationCherchee)
alors x :=x.suivant[i]
fin si
fin pour
si (x.suivant[1].EtapeCreation=EtapeCreationCherchee)
alors retourne x.suivant[1].CouleurPixel
sinon retourne x.CouleurPixel
fin si

Nous pouvons alors examiner dans le pire des cas le temps de recherche d'un noeud dans la skip list. Supposons que ce noeud soit le $p^{\text{ème}}$ noeud dans une skip list de hauteur maximale 6 et considérons que la tour de tête occupe la position 0. Nous pouvons décomposer p :

$$p = a \times 2^5 + b \times 2^4 + c \times 2^3 + d \times 2^2 + e \times 2^1 + f \times 2^0$$

D'après l'analyse précédente, dans le pire des cas et pour chaque niveau inférieur au niveau 6, on aura un seul noeud à explorer. Si $a = 0$ alors, il faudra quand même explorer un noeud de niveau maximum 6, sinon il faudra explorer $(a+1)$ noeuds de niveau maximum dans le pire des cas. En fait dans le pire des cas, il faut explorer effectuer : $a + 1 + 1 + 1 + 1 + 1 = a + 5$ noeuds. On peut généraliser à une skip

list presque parfaite de niveau maximum h, dans le pire des cas, la recherche d'un noeud à la position p telle que : $p = a_{h-1} \times 2^{h-1} + a_{h-2} \times 2^{h-2} + \ldots + a_0$, et obtenue en parcourant au plus : $a_{h-1} + (h-1)$ noeuds.

Les temps de reconstruction ont été relevés pour différentes étapes de la simulation. Il s'agit d'un temps moyen calculé sur 100 simulations consécutives. Les figures 5.10 et 5.12 des pages 106 et 107 regroupent ces résultats respectivement pour la voiture et le cendrier. Sur l'axe des ordonnées, on retrouve les temps de reconstruction et sur l'axe des abscisses, on retrouve les différentes scènes de la simulation.

Pour les étapes de début de simulation, les skip lists les plus efficaces sont les skip lists presque parfaites de hauteurs maximales les plus petites. La liste chaînée simple ou la skip list de niveau 2 sont alors les structures de données les plus rapides pour reconstruire une scène. Pour les étapes de fin de simulation, les skip lists les plus efficaces sont les skip lists presque parfaites de hauteurs maximales les plus grandes. Les skip lists de niveau 5 ou 6 sont alors les structures de données les plus rapides pour la voiture alors que la skip list de niveau 4 est la structure de données la plus efficace pour le cendrier.
Nous constatons aussi que plus les skip lists ont une hauteur maximale élevée, plus les temps de reconstruction des scènes sont proches. Les points des séries de mesures des figures 5.10 et 5.12 des pages 106 et 107 montrent un certain alignement. Nous souhaitons déterminer la valeur de la grandeur représentée par la pente de cette droite. Si la pente est importante, les temps de reconstructions seront distants, sinon les temps de reconstruction seront proches les uns des autres.
Pour justifier cette proximité, nous allons utiliser la méthode des moindres carrés. Nous disposons d'un ensemble de n points expérimentaux : M_i=(x_i ;y_i) avec i=1...n. Nous souhaitons ajuster la distribution des n points expérimentaux M_i par une droite d'équation :
$y = a + b \times (x - x_{moy}) = f(x)$ dite droite de régression de y en x.
$x_{moy} = \frac{\sum_{i=1}^{n} x_i}{n}$ est la moyenne des n points expérimentaux, b est la pente de la droite de régression et a l'ordonnée du point d'abscisse x_{moy}.
La méthode de moindres-carrés consiste à choisir a et b de telle sorte que la somme des carrés des écarts en ordonnées $\sum_{i=1}^{n} (y_i - y'_i)^2$ soit minimale (avec M point de la droite de régression de coordonnées $(x_i ; y'_i)$ où $y'_i = f(x_i)$). D'après [53], les coefficients a et b répondent aux équations suivantes :

$$a = y_{moy} = \frac{\sum_{i=1}^{n} y_i}{n} \qquad\qquad b = \frac{\sum_{i=1}^{n} (x_i - x_{moy}) y_i}{\sum_{i=1}^{n} (x_i - x_{moy})^2}$$

Les valeurs b des pentes des droites de régression linéaire ont été calculées dans le tableau 5.20 de la page 108. Nous avons également calculé dans le tableau 5.21 de la page 108 les coefficients de corrélation. D'après [53], le coefficient de corrélation répond à l'équation suivante :

$$R = \frac{\sum_{i=1}^{n}(x_i - x_{moy})(y_i - y_{moy})}{(n-1)s_x s_y} \text{ avec } s_x^2 = \frac{\sum_{i=1}^{n}(x_i - x_{moy})^2}{n-1} \text{ et } s_y^2 = \frac{\sum_{i=1}^{n}(y_i - y_{moy})^2}{n-1}$$

Il permet de vérifier la validité de l'ajustement par une droite. La valeur absolue du coefficient de corrélation est toujours comprise entre 0 et 1. Si le coefficient de corrélation est égal à 1, les points expérimentaux sont parfaitement alignés avec les points de la droite de régression linéaire. Plus le coefficient de corrélation se rapproche de 1, meilleur est l'ajustement linéaire. On estime que l'ajustement est valable pour un coefficient de corrélation compris entre 0,7 et 1. En ce qui concerne la voiture, les coefficients de corrélation sont tous supérieurs à 0,93, les résultats concernant la droite de régression peuvent être exploités sans problème. En ce qui concerne le cendrier, seuls les coefficients de corrélation des traces en temps de niveau 5 et de niveau 6 posent problème. En s'intéressant plus précisément aux temps de reconstruction des scènes, on constate que ces temps varient peu, on s'attend donc à une pente de la droite de régression faible, ce qui est tout à fait en accord avec la valeur de pente calculée. On considère donc que la pente de la droite de régression est un bon indicateur pour justifier de la proximité des temps de reconstruction.
Pour la liste chaînée simple, la pente de la droite de régression est de 0,3021 pour la voiture et 0,0824 pour le cendrier. Pour la skip list presque parfaite de niveau 6, la pente de la droite de régression est de 0,0158 pour la voiture et 0,0028 pour le cendrier. Les temps de reconstruction des scènes sont plus homogènes pour les skip lists de hauteur maximale élevée.

Nous avons calculé dans le tableau 5.13 de la page 100 le nombre de noeuds à explorer dans le pire des cas (en utilisant la formule précédente) pour retrouver un noeud à une position souhaitée. Nous avons envisagé trois cas :

- (1) : le pire des cas pour a=2, ce qui revient à chercher des noeuds entre la 32ème et la 95ème position. En effet, pour la voiture, la skip list la plus grande de la simulation est composée de 87 éléments ($a = 2$, $b = 1$, $c = 0$, $d = 1$, $e = 1$, $f = 1$). Pour le cendrier, la skip list la plus grande de la simulation est composée de 94 éléments ($a = 2$, $b = 1$, $c = 0$, $d = 1$, $e = 1$, $f = 0$).
- (2) : le pire des cas pour $b = 1$ ($a = 0$), ce qui revient à chercher des noeuds entre la 16ème et la 31ème position. En Effet, pour la voiture, les skip lists ont en moyenne 24 éléments ($a = 0$, $b = 1$, $c = 1$, $d = 0$, $e = 0$, $f = 0$) Pour le cendrier, les skip lists ont en moyenne 30 éléments ($a = 0$, $b = 1$, $c = 0$, $d = 1$, $e = 1$, $f = 0$).

	a=2 (entre 64ème et 95ème position) (1)	b=1 (entre 16ème et 31ème position) (2)	e=1 (entre 1ème et 3ème position) (3)	(1)-(3)	(2)-(3)
Ttrace L_6	7	6	6	1	0
Ttrace L_5	9	5	5	4	0
Ttrace L_4	14	6	4	10	2
Ttrace L_3	25	9	3	22	6
Ttrace L_2	48	16	2	46	14
Ttrace L_1	95	31	3	92	28

TAB. 5.13 – Nombre de noeuds à explorer dans le pire des cas pour des skip list de hauteurs différentes

- (3) : le pire des cas pour e=1 ($a = 0$, $b = 0$, $c = 0$, $d = 0$), ce qui revient à chercher des noeuds entre la 3ème et la 1ème position, et qui correspond à des étapes de début de simulation.

Ensuite, nous avons calculé dans la quatrième colonne du tableau la différence du nombre de noeuds à explorer entre le cas (1) et le cas (3) et dans la dernière colonne, la différence du nombre de noeuds à explorer entre le cas (2) et le cas (3). Grâce à l'étude de ces différences, nous allons retrouver les variations de la pente de la droite de régression concernant les temps de reconstruction de scènes par des skip lists de niveaux différents (tableau 5.20 de la page 108). Nous pouvons considérer que le nombre de noeuds à explorer pour les étapes de début de simulation (colonne (3)) et pour les étapes de fin de simulation (colonne (2) pour la plupart des skip list de la simulation, et colonne (1) parfois) est proportionnel au temps de reconstruction des scènes et donc à la pente de la droite de régression linéaire correspondante.

D'après les deux dernières colonnes du tableau, la différence entre le nombre de noeuds à explorer dans le pire des cas entre les étapes de début de simulation et les étapes de fin de simulation augmente lorsque le niveau de la skip list diminue. Cela confirme les remarques précédentes concernant les pentes des droites de régression. Plus les skip lists ont une hauteur maximale élevée, plus la pente de la droite de

régression linéaire correspondante est faible et plus les temps de reconstruction des scènes sont homogènes.

Dans les deux exemples, la skip list la mieux adaptée au problème de reconstruction d'une scène est la skip list de niveau 4 qui saute au plus 8 noeuds. Cette skip list occupe 2 fois plus de mémoire qu'une simple liste chaînée, la pente de la droite de régression sur les temps de reconstruction des scènes est de 0,0709 pour la voiture et 0,0129 pour le cendrier.
Pour la voiture, la skip list de niveau 4 met 21,1 % plus de temps que la liste simple chaînée pour reconstruire la scène de l'étape $\frac{1}{10}$, mais 26,2 % moins de temps que la skip list de niveau 6 qui correspond ici au pire des cas. Pour reconstruire une étape de fin de simulation, la skip list de niveau 4 met 14,2 % plus de temps que la skip list de niveau 6, mais 57,5 % moins de temps que la liste chaînée simple.
Pour le cendrier, la skip list de niveau 4 est la structure de données la plus rapide que ce soient pour les étapes de début de simulation ou de fin de simulation. Elle met 3,3 % moins de temps que la liste simple chaînée pour reconstruire la scène de l'étape $\frac{1}{10}$, et 25,8 % moins de temps que la skip list de niveau 6 qui correspond ici au pire des cas. Pour reconstruire une étape de fin de simulation, la skip list de niveau 4 met 50,7 % moins de temps que la skip list de niveau 6, et 5,5 % moins de temps que la liste chaînée simple.
La skip list de niveau 4 paraît donc être un bon compris pour atteindre rapidement les scènes de début de simulation, de fin de simulation tout en occupant la mémoire de manière raisonnable.

5.5.2.2 Utilisation de la dichotomie pour la trace en temps

En s'intéressant de plus près à l'algorithme optimisé pour la reconstruction d'une scène par des skip lists presque parfaites présenté à la page 97, nous constatons que cet algorithme se rapproche du principe de la dichotomie. Nous avons donc appliquer le principe de la dichotomie à la trace en temps.

- **Choix de la méthode de la dichotomie**
  L'algorithme de la dichotomie effectue la recherche d'un élément dans un tableau trié en réduisant à chaque fois l'intervalle de recherche de moitié. Comme la dichotomie utilise le fait que le tableau ait été trié au préalable, elle permet de limiter le nombre de lectures à $\log(N) + 1$, et de diminuer ainsi l'espace de recherche. C'est une méthode de recherche très efficace.
  Dans notre application, la trace en temps est construite suivant des valeurs en temps croissantes. Il est donc aisé de transformer la trace en temps sous forme

de tableau trié et de lui appliqué le principe de la dichotomie présenté ci-après.

- **Mise en place de la méthode de la dichotomie**
  Nous rappelons le principe général de la dichotomie : étant donné un tableau T trié en ordre croissant, et un élément de valeur v_f, on cherche à savoir où v_f apparaît dans T. Pour commencer, on compare la valeur cherchée v_f à l'élément central du tableau, si ce n'est pas la bonne, un test permet de trouver dans quelle moitié du tableau on trouvera la valeur v_f. On continue récursivement jusqu'à un sous-tableau de taille 1, ce qui revient à réitérer le processus jusqu'à trouver la valeur cherchée ou jusqu'à ce que les bornes inférieure et supérieure de l'intervalle de recherche se croisent. Ainsi, à chaque itération, la taille de la sous suite de valeurs à explorer est divisée par deux.
  Côté programmation, il vaut mieux implémenter cet algorithme de manière itérative, car la fonction est exécutée jusqu'à trouver la position désirée. Les dépilages sont ensuite effectués mais nous n'avons plus besoin des états intermédiaires mémorisés par la récursivité puisque le problème est résolu.
  Dans le cas de la trace en temps v_f représente l'étape de création. Si cette étape n'est pas présente dans le tableau trié de la trace en temps, nous considérons comme pour les skip lists que l'étape de création cherchée est la première étape de création inférieure à v_f et présente dans la trace en temps. Pour reconstruire une scène par cette méthode, nous utilisons l'algorithme de dichotomie classique.

- **Analyse des résultats obtenus par la méthode de la dichotomie**
  A partir de la trace en temps obtenue au cours de la simulation, nous créons un tableau trié qui permet d'effectuer la recherche dichotomique. Comme le nombre d'éléments de la trace en temps est connu, nous allouons le tableau trié de manière statique et rangeons dans ce tableau toutes les traces en temps de la simulation, les unes à la suite des autres. En réalité, nous devons créer deux structures de données pour mener à bien la recherche dichotomique :
  - la première structure de données est le tableau trié présenté précédemment. Il regroupe toutes les traces en temps de la simulation. Chaque élément du tableau contient la valeur en Z de l'élément considéré, sa couleur et son étape de création, ce qui correspond à 8 octets par élément du tableau.
  - la seconde structure de données est également un tableau qui contient l'indice correspondant au premier élément de chaque trace dans le tableau trié précédent. Comme à chaque pixel de l'écran est associé une trace, à la fin de la simulation, nous obtenons pour cette seconde structure de données un tableau de 307 200 éléments pour nos exemples de simulation, où chaque

élément représente un indice du tableau trié, ce qui correspond à 4 octets soit au total 307200 × 4 soit 1 228 800 octets (1,17Mo).

Le tableau 5.14 de la page 103 donne l'occupation mémoire utilisée par la structure de données de la trace en temps dans le cas d'une recherche dichotomique. Cette structure occupe moins d'espace mémoire que toutes les structures précédemment étudiées pour la trace en temps. En effet, pour la voiture, cette structure n'occupe que 71,4% de l'espace mémoire occupé par la liste chaînée simple (comme nous utilisons un tableau trié, une économie est réalisée sur les pointeurs des éléments suivants) et 35,7% de l'espace mémoire occupé par la skip list de niveau 4. De même, pour le cendrier, cette structure n'occupe que 76,8% de l'espace mémoire occupé par la liste chaînée simple et 38,4% de l'espace mémoire occupé par la skip list de niveau 4.

Le temps nécessaire pour créer la nouvelle structure de données est répertorié dans la tableau 5.15 de la page 103. Cette nouvelle structure de données met à peu près le même temps à être créée que la skip list de niveau 4 que ce soit pour le cas de la voiture ou du cendrier.

Voiture	18 625 040 (*soit 17,76Mo*)
Cendrier	9 265 072 (*soit 8,84Mo*)

TAB. 5.14 – Occupation mémoire de la structure de la trace en temps utilisé pour la recherche dichotomique

	Trace L_4	Dichotomie
Voiture	0,61	0,62
Cendrier	0,27	0,28

TAB. 5.15 – Construction de la trace en temps pour une recherche dichotomique (en secondes)

Les figures 5.11-5.13 des pages 106 et 107 permettent de comparer les temps de reconstruction des scènes sans affichage.
Nous nous intéresserons tout d'abord à la voiture. Pour les étapes de début de simulation, les skip lists sont plus efficaces, alors que pour les étapes de fin de simulation

la recherche dichotomique est plus performante. La dichotomie met 36,3% plus de temps que la liste chaînée simple et 8,3% plus de temps que la skip list de niveau 4 pour reconstruire la scène à l'étape $\frac{1}{10}$. Par contre, dans le meilleur des cas, elle ne met que 13,9% du temps nécessaire à la liste chaînée simple et 35,1% du temps nécessaire à la skip list de niveau 4 pour reconstruire une scène de fin de simulation. Pour le cendrier, les skip lists sont également un peu plus efficaces pour les étapes de début de simulation, en fin de simulation la dichotomie est bien plus performante. La dichotomie met à peu près le même temps que la liste chaînée simple pour reconstruire la scène à l'étape $\frac{1}{10}$ et 3,12% plus de temps que la skip list de niveau 4 pour reconstruire la même scène. Par contre, dans le meilleur des cas, elle ne met que 27,4% du temps nécessaire à la liste chaînée simple et 53,9% du temps nécessaire à la skip list de niveau 4 pour reconstruire une scène de fin de simulation .

Les pentes des droites de régression concernant les temps de reconstruction sont regroupées dans le tableau 5.20 de la page 108. Cette pente est meilleure pour la recherche par dichotomie dans le cas de la voiture et meilleure pour la skip list de niveau 4 dans le cas du cendrier.
A la vue de ces résultats, nous en déduisons que pour notre simulation où les skip lists sont au maximum de niveau 6, plus la trace en temps est importante, et plus la dichotomie est efficace (car cela évite d'effectuer des sauts inutiles sur plusieurs niveaux dans le cas des skip lists). L'angle de vue et la stratégie d'usinage ont aussi une influence sur l'efficacité de la recherche par dichotomie, puisqu'ils ont une influence sur le nombre d'éléments en temps, et donc sur la longueur de chaque trace en temps de la simulation.

5.5.2.3 Utilisation d'une skip list aléatoire pour la trace en temps

Pour compléter l'étude des structures de données pour la trace en temps, nous avons étudié les performances d'une skip list aléatoire. Les résultats sont regroupés dans les tableaux 5.16-5.19 de la page 105.
La skip list aléatoire met 8,2% fois plus de temps à être créée que la skip list presque parfaite de niveau 4 dans le cas de la voiture et 3,6% plus de temps dans le cas du cendrier. Par contre la skip list aléatoire n'occupe que 78,1% de l'espace mémoire occupée par la skip list de niveau 4 dans le cas de la voiture et 88,9% dans le cas du cendrier. En ce qui concerne le temps de reconstruction des scènes, la skip list aléatoire n'est pas une structure de données intéressante pour la trace en temps par rapport aux structures de données étudiées précédemment. En effet, comme le montre les tableau 5.11 et 5.13 des pages 106-107, la skip list aléatoire est, dans les deux exemples, toujours moins efficace que la skip list presque parfaite de niveau

4 et que la dichotomie. Pour les étapes de fin de simulation, la skip list aléatoire est tout de même plus efficace que la liste chaînée simple, elle ne met que 51,4% du temps nécessaire par la liste chaînée simple dans le cas de la voiture, et 68,3% du temps nécessaire par la liste chaînée simple dans le cas du cendrier pour reconstruire une scène.

	Voiture	Cendrier
Temps de création	0,66	0,28

TAB. 5.16 – Création d'une skip list aléatoire pour la trace en temps (en secondes)

	Nodes L_1	Nodes L_2	Nodes L_3	Nodes L_4	Nodes L_5	Nodes L_6
Voiture	806 352	581 815	163 672	121 462	111 624	389 605
Cendrier	295 132	216 918	71 131	42 380	31 748	347 225

TAB. 5.17 – Répartition des noeuds dans la skip list aléatoire

	Mémoire Statique	Mémoire Dynamique
Voiture	40 766 624 (*soit 38,87Mo*)	38 919 248 (*soit 37,12Mo*)
Cendrier	21 452 156 (*soit 20,46Mo*)	20 599 680 (*soit 19,64Mo*)

TAB. 5.18 – Occupation mémoire d'une skip list aléatoire (en octets)

		Étape $\frac{1}{10}$	Étape $\frac{1}{4}$	Étape $\frac{1}{2}$	Étape $\frac{3}{4}$	Dernière étape
Voiture	Liste chaînée	0,06	0,09	0,17	0,25	0,31
	SL aléatoire	0,11	0,12	0,14	0,15	0,16
Cendrier	Liste chaînée	0,06	0,08	0,10	0,12	0,13
	SL aléatoire	0,09	0,10	0,10	0,10	0,09

TAB. 5.19 – Affichage d'une scène par la trace en temps sous forme de liste chaînée et de skip list aléatoire pour la voiture et le cendrier (en secondes)

Reconstruction d'une scène par la trace en temps pour la voiture

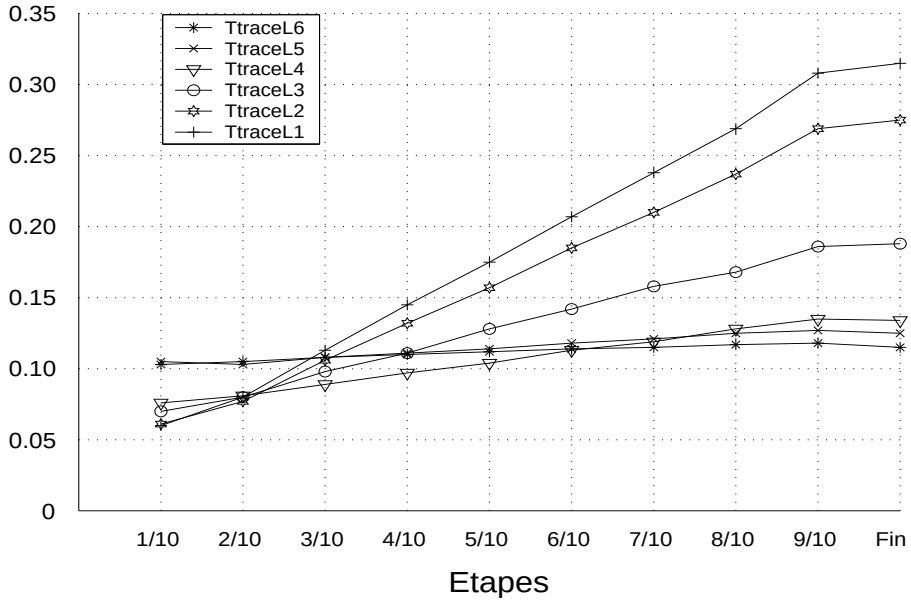

FIG. 5.10 – Trace en temps sous forme de skip lists presque parfaites pour la voiture

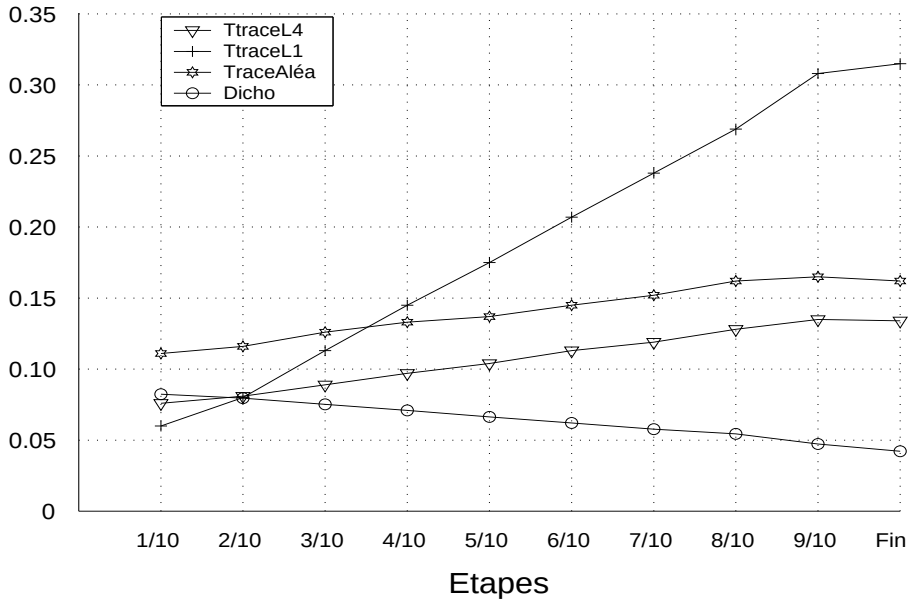

FIG. 5.11 – Trace en temps sous forme de skip lists déterministe, aléatoire et utilisation de la dichotomie pour la voiture

Reconstruction d'une scène par la trace en temps pour le cendrier

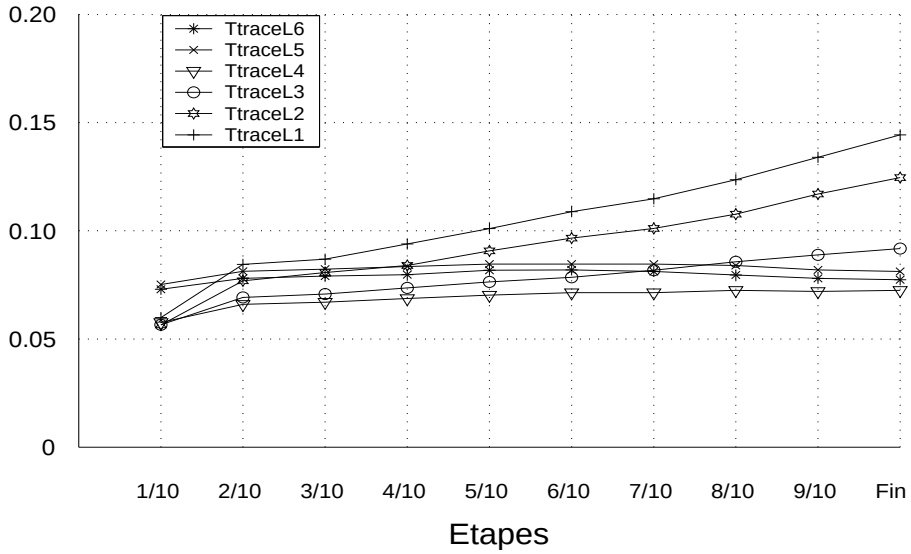

FIG. 5.12 – Trace en temps sous forme de skip lists presque parfaites pour le cendrier

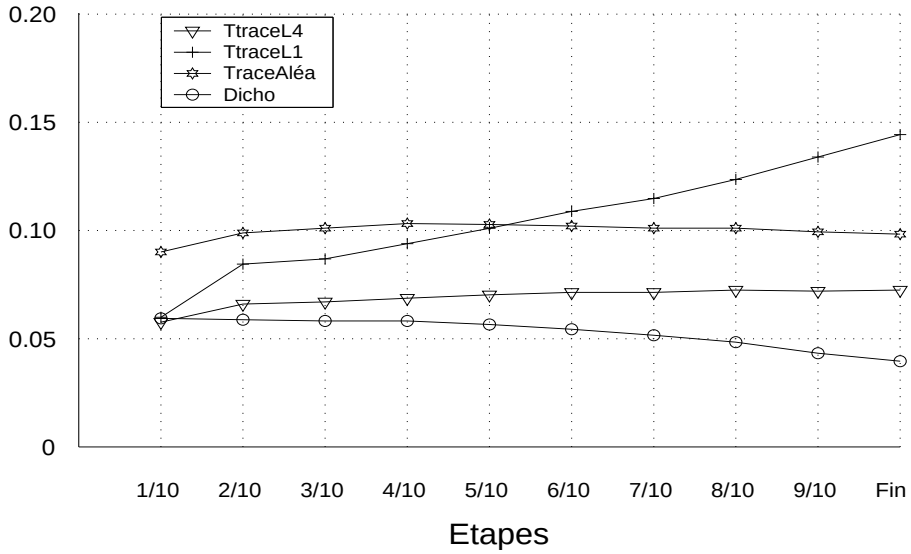

FIG. 5.13 – Trace en temps sous forme de skip lists déterministe, aléatoire et utilisation de la dichotomie

	Trace L_1	Trace L_2	Trace L_3	Trace L_4	Trace L_5	Trace L_6	Dicho	Aléa
Voiture	0,3021	0,2539	0,1399	0,0709	0,0283	0,0158	-0,0445	0,0635
Cendrier	0,0824	0,0658	0,0337	0,0129	0,0042	0,0028	-0,0217	0,0043

TAB. 5.20 – Pente de la droite de régression linéaire obtenue par la méthode des moindres carrés sur les temps de reconstruction d'une scène pour toutes les structures étudiées pour la trace en temps

	Trace L_1	Trace L_2	Trace L_3	Trace L_4	Trace L_5	Trace L_6	Dicho	Aléa
Voiture	0,9977	0,9976	0,9963	0,9938	0,9758	0,9431	0,9971	0,9848
Cendrier	0,9863	0,9833	0,9736	0,8505	0,4511	0,3252	0,9386	0,3417

TAB. 5.21 – Coefficient de corrélation pour la droite de régression linéaire obtenue par la méthode des moindres carrés sur les temps de reconstruction d'une scène pour toutes les structures étudiées pour la trace en temps

5.5.2.4 Conclusion sur les méthodes pour implémenter la trace en temps

D'après les études précédentes, les méthodes les mieux adaptées à la reconstruction d'une scène par la trace en temps semblent être la skip list presque parfaite de niveau 4 ou l'utilisation de la méthode de la dichotomie. Le temps nécessaire pour afficher une scène quelconque de la simulation est à peu près constant, ce qui permet de mettre en place une animation conviviale de la simulation en assurant ainsi un débit régulier d'images. Pour choisir la structure adéquate au problème, il faut tenir compte de la stratégie d'usinage et de l'angle de vue, et trouver le bon compromis entre le temps de construction de la nouvelle structure de données, son occupation mémoire et son efficacité pour optimiser les temps de reconstruction des scènes (c'est à dire sa faculté à rechercher plus ou moins rapidement un élément dans la trace).

Chapitre 6

Evaluation des ressources nécessaires

Pour s'assurer que la simulation d'usinage est compatible avec les contraintes de l'utilisateur et utilise au mieux les ressources matérielles, il convient d'effectuer des prédictions de performances à partir du programme d'usinage.
Nous allons évaluer la faisabilité de la simulation en analysant les ressources (mémoire, temps de calcul) nécessaires à l'exécution d'un programme pièce. Tout d'abord, nous proposons un algorithme destiné à la reconnaissance la stratégie d'usinage, ce qui nous permet ensuite d'évaluer la mémoire nécessaire au bon déroulement de la simulation.

6.1 Reconnaissance de la stratégie d'usinage

Dans un fichier d'usinage, nous disposons :

- d'une table d'outils qui contient les dimensions de chaque outil (rayon et hauteur).
- d'une succession de trajectoires de l'outil. Pour chaque trajectoire de l'outil (appelée aussi trajectoire centre outil), nous disposons des coordonnées cartésiennes de son point de départ et de son point d'arrivée. En transformant ces coordonnées cartésiennes en coordonnées polaires, nous obtenons :
  - la longueur de la trajectoire notée l_i pour la i^{eme} trajectoire.
  - l'angle de la trajectoire avec l'axe (Ox) noté α_i pour la i^{eme} trajectoire.

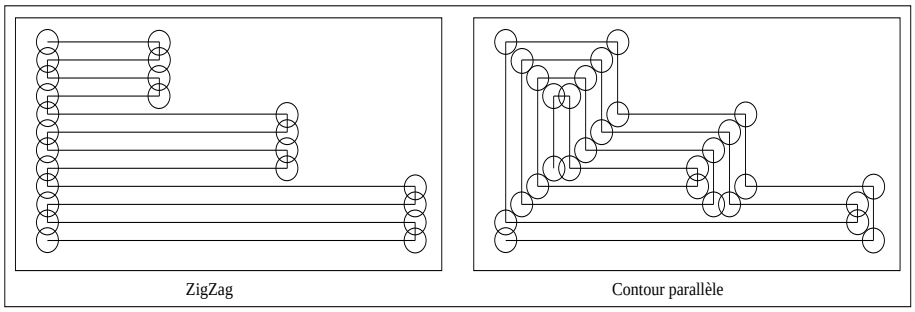

FIG. 6.1 – Usinage en zigzag et contours parallèles par une représentation de l'outil en début et fin de trajectoires

Deux stratégies d'usinage sont principalement utilisées : en directions parallèles ou zigzag et en contours parallèles ou spirale (figure 6.1 de la page 110). Nous rappelons que la distance entre deux trajectoires de l'outil est appelée *distance entre passes*. Elle est choisie par l'opérateur et dépend du rayon de l'outil et est au maximum égale à $2 \times (Rayon\ de\ l'Outil)$.

6.1.1 Analyse de la stratégie d'usinage

Le programme d'usinage est composé d'une succession de trajectoires qui se répètent que nous appellerons *motif de base*. Pour un usinage en zigzag, ce motif de base est prévisible, puisque les segments de trajectoire sont de suite parallèles entre eux. Par contre pour un usinage en contours parallèles, les trajectoires suivent le contour décalé de la pièce, les segments sont parallèles entre eux au bout d'un certain temps correspondant au contour de la pièce ou à un usinage de surfaces monotones. Ces segments sont alors décalés entre eux de la distance entre passes. Bien que le motif soit répétitif, il n'est pas prévisible et est spécifique à chaque pièce pour ce type d'usinage.
Pour reconnaître une stratégie d'usinage, nous allons donc essayer de retrouver en premier dans les trajectoires du programme d'usinage un éventuel motif de base de type zigzag. Nous allons maintenant étudier plus en détail les caractéristiques d'un tel motif.

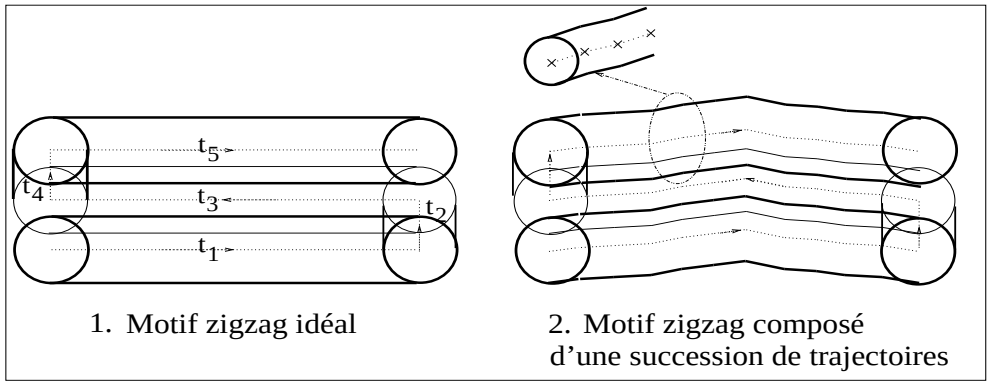

FIG. 6.2 – Motif répétitif dans un usinage en zigzag

L'usinage en zigzag est basé sur le motif idéal présenté sur la figure 6.2.1 de la page 111. Ce motif est caractérisé par :

- *Caractéristique 1* : deux *grandes* trajectoires opposées de π (par exemple, t_1 et t_3 de la figure 6.2.1). La longueur de ces trajectoires dépend du contour à obtenir.
- *Caractéristique 2* : deux *petites* trajectoires qui suivent le contour de la pièce. Elles permettent à l'outil de se replacer d'un zig vers un zag. Ces trajectoires sont dites *petites* car en principe, elles ont une longueur inférieure à $2 \times (Rayon\ de\ l'Outil)$ (par exemple, t_2 et t_4 de la figure 6.2.1).

L'usinage de la plupart des formes simples que l'on rencontre actuellement sur des

pièces mécaniques correspond à un déplacement du centre outil dans un plan : c'est un usinage de type 2D$\frac{1}{2}$. Cependant, lorsque les surfaces à réaliser deviennent plus complexes, il est indispensable de réaliser un vrai usinage 3D et de déplacer l'outil suivant 3 axes simultanément. La représentation géométrique de ces surfaces fait appel à un maillage composé d'un ensemble de carreaux constitués par des fonctions de Bézier ou des fonctions NURBS. Les trajectoires d'usinage en zigzag sont alors définies par des courbes isoparamétriques qui sont approchées par des petits segments linéaires qui respectent un motif conforme à un zigzag (figure 6.2.2 de la page 111). Ce motif est alors appelé *zigzag approché*.

6.1.2 Classement des trajectoires

Nous allons répartir les trajectoires en deux classes basées sur la longueur des trajectoires. La valeur seuil permettant de passer d'une classe à l'autre est égale à $2 \times$ (*Rayon de l'Outil*) qui est la valeur maximale de la distance entre passes fixée lors de la génération du programme d'usinage.
Il suffit alors de compter le nombre de grandes trajectoires et le nombre de petites trajectoires. Si ces deux ensembles sont sensiblement égaux en nombre de trajectoires (moyennant une erreur relative que nous avons estimée après les tests à 25%), nous supposons que l'usinage utilisé est du type zigzag. Mais une vérification s'impose. En effet, si nous nous trouvons dans le second cas de la figure 6.2 de la page 111 (c'est-à-dire dans le cas d'un usinage 3D), où les grandes trajectoires du motif zigzag sont en fait composées d'une multitude de petites trajectoires, le résultat du classement des trajectoires ne s'avérera pas compatible avec la stratégie réellement utilisée dans un tel usinage.
Pour valider le résultat précédent et/ou compléter l'étude précédente, nous avons envisagé deux autres approches de reconnaissance de la stratégie d'usinage.

6.1.3 Méthode du vecteur V_k

Pour reconnaître un usinage de type zigzag d'un usinage de type contours parallèles, nous allons étudier une méthode qui s'intéresse au sens de parcours des trajectoires successives[1]. Chaque trajectoire i est représentée par un vecteur $\overrightarrow{v_i}$ de longueur l_i et d'angle α_i ($0 \leq \alpha_i \leq 2\pi$: angle de la trajectoire avec l'axe Ox). Définissons le vecteur $\overrightarrow{V_k}$: l'origine du vecteur $\overrightarrow{V_k}$ est le point de départ de la première trajectoire $\overrightarrow{v_1}$ et l'extrémité du vecteur $\overrightarrow{V_k}$ est le point d'arrivée de k^{eme} trajectoire

[1] Nous supposons que toutes les trajectoires de la séquence d'usinage sont des trajectoires linéaires, les éventuelles trajectoires circulaires ayant été au préalable décomposées en trajectoires linéaires

$\overrightarrow{v_k}$. Le vecteur $\overrightarrow{V_k}$ est donc le vecteur résultant de la somme vectorielle des trajectoires : $\overrightarrow{V_k} = \sum_{i=1}^{k} \overrightarrow{v_i}$. On lui associe une longueur L_k et un angle θ_k (angle du vecteur $\overrightarrow{V_k}$ avec l'axe Ox). Pour déterminer le type d'usinage, nous nous intéressons à l'évolution de l'angle θ_k du vecteur $\overrightarrow{V_k}$.

Nous représenterons l'angle et la fonction angulaire de chaque vecteur $\overrightarrow{V_k}$ en fonction du temps. En effet, comme nous nous déplaçons sur les trajectoires $\overrightarrow{v_i}$ de la séquence d'usinage à vitesse constante, nous repérons à la fin de chaque trajectoire i l'instant d'apparition t_k d'un nouveau vecteur $\overrightarrow{V_k}$. Nous noterons donc par :

- $\theta(t_k)$ l'angle du vecteur $\overrightarrow{V_k}$.
- $\varphi(t_k)$ la fonction angulaire cumulée associée au vecteur $\overrightarrow{V_k}$.
- $\alpha(t_{i-1})$ l'angle de la trajectoire $\overrightarrow{v_i}$ commencée à l'instant t_{i-1}, la trajectoire $\overrightarrow{v_i}$ étant définie sur l'intervalle de temps $[t_{i-1}, t_i[$.

La fonction angulaire cumulée est définie par Otterloo [54] à partir de l'angle entre 2 trajectoires successives : $\Delta\alpha(t_i) = \alpha(t_i) - \alpha(t_{i-1})$. Chaque terme $\Delta\alpha(t_i)$ est transformé en $\Delta\varphi(t_i)$ qui vérifie l'inégalité suivante : $-\pi \leq \Delta\varphi(t_i) \leq \pi$:

$$\Delta\varphi(t_i) = \left| \begin{array}{llll} \Delta\alpha(t_i) + 2\pi & \text{pour} & -2\pi & < \Delta\alpha(t_i) < -\pi \\ \Delta\alpha(t_i) & \text{pour} & -\pi & \leq \Delta\alpha(t_i) \leq \pi \\ \Delta\alpha(t_i) - 2\pi & \text{pour} & \pi & < \Delta\alpha(t_i) < 2\pi \end{array} \right.$$

$\Delta\varphi(t_i)$ a une valeur positive pour des angles convexes et une valeur négative pour des angles concaves. La fonction angulaire cumulée $\varphi(t_k)$ est alors égale à :

$$\varphi(t_k) = \sum_{i=1}^{k-1} \Delta\varphi(t_i) \text{ avec } k > 1$$

Otterloo utilise la fonction angulaire cumulée pour identifier une courbe simple fermée (polygone) à partir de l'étude de son contour. La représentation paramétrique d'une telle courbe est une fonction périodique :

$$z(t_k) = z(t_k + n \times 2\pi) \qquad \text{pour } t_k \in [0, 2\pi] \text{ et } n \in Z$$

En étudiant la trajectoire d'un point qui se déplace à vitesse constante sur le contour de la courbe, il est possible de calculer et de suivre l'évolution de la fonction angulaire cumulée sur la courbe. A cause de la périodicité de la représentation paramétrique, la fonction angulaire cumulée doit elle aussi devenir une fonction périodique de la forme :

$$\varphi(t_k + 2\pi) = \varphi(t_k) + 2\pi \qquad \text{pour } t_k \in [0, 2\pi]$$

La fonction angulaire cumulée $\varphi(t_k)$ est donc égale à 2π en fin de parcours du contour d'un polygone simple fermé, en considérant que les côtés du polygone soient parcourus dans le sens trigonométrique direct. Si le polygone est parcouru dans le sens anti trigonométrique, $\varphi(t_k) = -2\pi$. Au cours du parcours dans le sens trigonométrique direct d'un polygone simple fermé, $0 \leq \varphi(t_k) < 2\pi$.

Nous considérons qu'une courbe γ est parcourue dans le sens trigonométrique direct (ou sens positif), si en se déplaçant le long de cette courbe γ, l'intérieur de γ, c'est-à-dire la région limitée par γ reste à gauche de la courbe γ. Ainsi, lorsque deux trajectoires successives d'une courbe polygonale sont parcourues dans le sens anti trigonométrique, $\Delta\varphi(t_k) > 0$ et la fonction angulaire cumulée $\varphi(t_k)$ augmente. Lorsque deux trajectoires successives d'une courbe polygonale sont parcourues dans le sens trigonométrique inverse, $\Delta\varphi(t_k) < 0$ et par conséquent $\varphi(t_k)$ décroît.
Dans une séquence d'usinage en contours parallèles, les trajectoires successives sont toujours parcourues dans le même sens. Supposons que ce sens de parcours soit le sens trigonométrique direct, $\varphi(t_k)$ ne se limite plus à 2π, mais à chaque première trajectoire d'un nouveau contour (puisque pour chaque trajectoire $\Delta\varphi(t_k) > 0$) :

$$\varphi(t_k) = n \times 2\pi \qquad \text{avec } n \in N$$

n est appelé le *niveau* du contour, il correspond à :

$$n = \lfloor \frac{\sum_{i=1}^{k-1} \Delta\varphi(t_i)}{2\pi} \rfloor \text{ où } (\lfloor\ X\ \rfloor \text{ représente la partie entière de } X).$$

Le *niveau n* est positif si les trajectoires sont parcourues dans le sens trigonométrique direct, négatif sinon. Le *niveau n* est plus ou moins important suivant la taille de la pièce usinée et la distance entre passes. Dans un usinage en contours parallèles, le *niveau n* maximal est toujours supérieur à 1 ou inférieur à -1. Le *niveau 0* correspond au premier contour d'usinage de la pièce.
Précédemment, nous avons caractérisé le vecteur $\overrightarrow{V_k}$ par sa longueur L_k et par son angle θ_k (correspondant à $\theta(t_k)$). L'angle $\theta(t_k)$ repère la position du vecteur $\overrightarrow{V_k}$ par rapport à l'axe (Ox). Désormais, nous appelons $\theta_{[-\pi;\pi]}(t_k)$, l'angle $\theta(t_k)$ qui vérifie l'inégalité suivante : $-\pi \leq \theta(t_k) \leq \pi$. Nous définissons l'angle $\theta_{[niveau]}(t_k)$ qui tient compte du *niveau n* du contour sur lequel se situe la trajectoire k associée au vecteur $\overrightarrow{V_k}$ par :

$$\theta_{[niveau]}(t_k) = \theta_{[-\pi;\pi]}(t_k) \ + \ n \times \ 2\pi$$

Un usinage de type contours parallèles est identifié dès que la fonction angulaire cumulée $\varphi(t_k) > 2\pi$ ou $\varphi(t_k) < -2\pi$. L'angle $\theta_{[niveau]}(t_k)$ maximum, noté $\theta_{max[niveau]}$ vérifie dans ce cas une des inégalités suivantes :

$$\theta_{max[niveau]} > 2\pi \text{ ou } \theta_{max[niveau]} < -2\pi.$$

Calculons maintenant la fonction angulaire cumulée pour un motif zigzag de base. Un motif zigzag de base est composé de deux grandes trajectoires opposées de π : $\overrightarrow{v_1}(l_1, \alpha_1)$ et $\overrightarrow{v_3}(l_3, \alpha_1 + \pi)$ et de deux petites trajectoires permettant à l'outil de se déplacer d'un zig vers un zag : $\overrightarrow{v_2}(l_1, \alpha_2)$ et $\overrightarrow{v_4}(l_4, \alpha_2)$. La trajectoire $\overrightarrow{v_5}(l_5, \alpha_1)$ recommence un nouveau motif. Nous supposons un motif idéal comme celui de la séquence d'usinage en zigzag de la figure 6.3 de la page 116 où $\alpha_4 = \alpha_2$. Les trajectoires $\overrightarrow{v_1}$ et $\overrightarrow{v_2}$ sont des trajectoires consécutives : $0 < \alpha_2 - \alpha_1 < \pi$ pour un parcours des trajectoires dans le sens trigonométrique direct, $-\pi < \alpha_2 - \alpha_1 < 0$ sinon. Etudions l'évolution de la fonction angulaire cumulée le long du motif zigzag pour $0 < \alpha_2 - \alpha_1 < \pi$:

$$\begin{array}{llllll}
\Delta\alpha(t_2) = \alpha_2 - \alpha_1 & \longrightarrow & \Delta\varphi(t_2) = \alpha_2 - \alpha_1 & >0 & \longrightarrow & \varphi(t_2) = \alpha_2 - \alpha_1 \\
\Delta\alpha(t_3) = (\alpha_1 + \pi) - \alpha_2 & \longrightarrow & \Delta\varphi(t_3) = \alpha_1 - \alpha_2 + \pi & >0 & \longrightarrow & \varphi(t_3) = \pi \\
\Delta\alpha(t_4) = \alpha_2 - (\alpha_1 + \pi) & \longrightarrow & \Delta\varphi(t_4) = \alpha_2 - \alpha_1 - \pi & <0 & \longrightarrow & \varphi(t_4) = \alpha_2 - \alpha_1 \\
\Delta\alpha(t_5) = \alpha_1 - \alpha_2 & \longrightarrow & \Delta\varphi(t_5) = \alpha_1 - \alpha_2 & <0 & \longrightarrow & \varphi(t_5) = 0
\end{array}$$

Pour un usinage en zigzag direct, $0 \leq \varphi(t_k) \leq \pi$, en effet $\Delta\varphi(t_k)$ change de signe au passage d'un zig vers un zag ce qui permet à $\varphi(t_k)$ de ne pas dépasser 2π. Généralisons. Dans une séquence d'usinage en zigzag, les trajectoires se situent toujours sur le contour de *niveau 0*, la fonction angulaire cumulée $\varphi(t_k)$ et l'angle $\theta_{[niveau]}(t_k)$ du vecteur $\overrightarrow{V_k}$ vérifient les inégalités suivantes quel que soit la trajectoire d'une séquence d'usinage en zigzag :

$$-\pi \leq \varphi(t_k) \leq \pi$$
$$-\pi \leq \theta_{[niveau]}(t_k) \leq \pi$$

Prenons maintenant les exemples des séquences d'usinage représentées sur la figure 6.3 de la page 116 pour illustrer la méthode du vecteur $\overrightarrow{V_k}$.
Dans le premier cas de la figure 6.3 de la page 116, nous avons représenté une succession de 8 trajectoires correspondant à une séquence d'usinage de type zigzag où nous pouvons suivre l'évolution du vecteur $\overrightarrow{V_k}$ (avec $1 \leq k \leq 8$) et de son angle $\theta(t_k)$. La fonction angulaire cumulée correspondant à ce type d'usinage est représentée sur la figure 6.4.1 de la page 116. Nous vérifions au travers de la figure 6.4.1 l'inégalité suivante : $-\pi \leq \varphi(t_k) \leq \pi$, caractérisant un usinage de type zigzag.

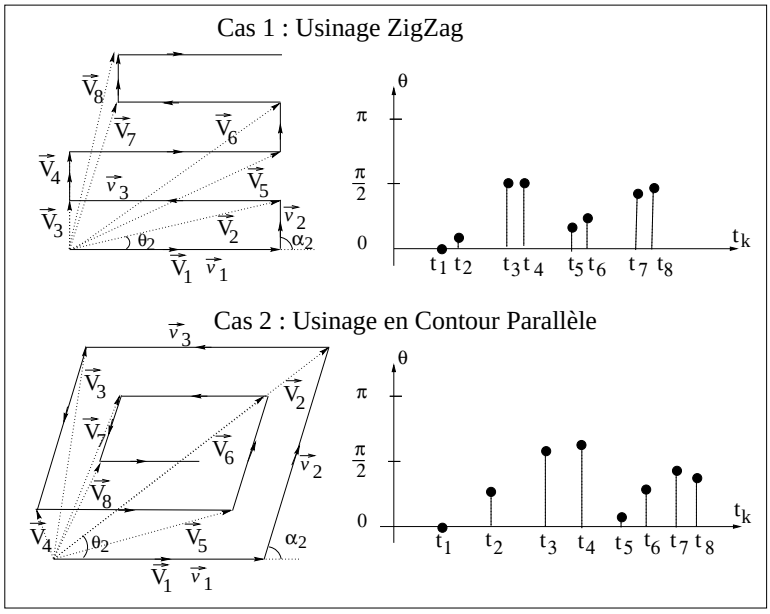

FIG. 6.3 – Vecteur : Somme vectorielle des vecteurs de trajectoires

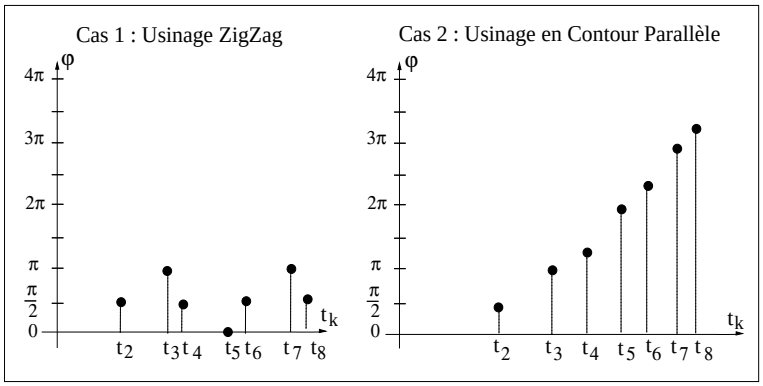

FIG. 6.4 – Fonction angulaire cumulée

Dans le second cas de la figure 6.3, nous avons représenté une succession de 8 trajectoires correspondant à une séquence d'usinage de type contours parallèles. La fonction angulaire cumulée correspondant à ce type d'usinage est représentée sur la figure 6.4.2 de la page 116, elle vérifie bien $\varphi(t_k) > 2\pi$. Nous pouvons retrouver le *niveau* du contour de chaque trajectoire à partir de la fonction angulaire cumulée :

- $0 < \varphi(t_k) < 2\pi$ pour $2 \leq k \leq 4$ alors
  - $n = 0$.
  - les trajectoires ($\overrightarrow{v_1}$, $\overrightarrow{v_2}$, $\overrightarrow{v_3}$, $\overrightarrow{v_4}$) appartiennent au contour de *niveau 0*, les vecteurs $\overrightarrow{V_k}$ correspondants ($\overrightarrow{V_1}$, $\overrightarrow{V_2}$, $\overrightarrow{V_3}$, $\overrightarrow{V_4}$) pointent sur le contour de *niveau 0*.
  - $0 < \theta_{[niveau]}(t_k) < \pi$ pour $2 \leq k \leq 4$.
- $2\pi \leq \varphi(k) < 4\pi$ pour $5 \leq k \leq 8$ alors
  - $n = 1$.
  - les trajectoires ($\overrightarrow{v_5}$, $\overrightarrow{v_6}$, $\overrightarrow{v_7}$, $\overrightarrow{v_8}$) appartiennent au contour de *niveau 1*, les vecteurs $\overrightarrow{V_k}$ correspondants ($\overrightarrow{V_5}$, $\overrightarrow{V_6}$, $\overrightarrow{V_7}$, $\overrightarrow{V_8}$) pointent sur le contour de *niveau 1*.
  - $2\pi \leq \theta_{[niveau]}(t_k) < 3\pi$ pour $5 \leq k \leq 8$.

Pour visualiser la *signature* de l'usinage (zigzag ou contours parallèles) nous utilisons la *carte de Gauss* qui permet de représenter des courbes (surfaces) dans un cercle (sphère) unitaire à partir des vecteurs normaux ([55], [56]). Prenons l'exemple d'une courbe γ (figure 6.5 de la page 117), et associons à chaque point P de γ, le point Q sur le cercle unité S^1 qui est à l'intersection du vecteur normal unitaire en P ramené au centre du cercle S^1 et de S^1 lui-même. Lorsque P' se rapproche de P sur la courbe γ, l'image gaussienne Q' de P' se rapproche de l'image gaussienne Q de P.

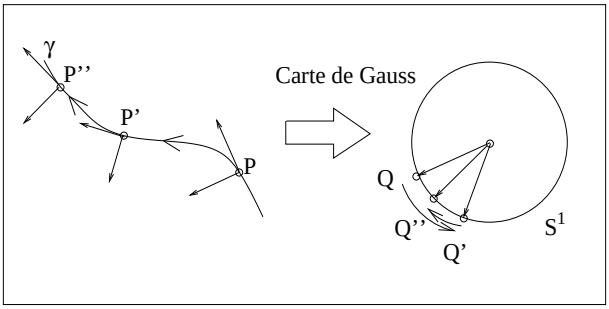

FIG. 6.5 – Carte de Gauss

La carte de Gauss peut également être définie à partir du vecteur tangent. Dans ce cas-là, le point Q est à l'intersection du vecteur tangent en P ramené au centre du cercle S^1 et du cercle lui-même. Les deux représentations ne sont équivalentes que pour les courbes du plan.

La carte de Gauss permet de d'étudier la courbure. Par convention, la courbure

est dite positive, lorsque le centre de courbure se trouve du même côté de la courbe que l'extrémité du vecteur normal orienté et négative lorsque les deux points sont de part et d'autre de la courbe. Inverser l'orientation d'une courbe revient aussi à inverser le signe de sa courbure. Considérons un point se déplaçant le long de la courbe γ, et intéressons-nous au déplacement correspondant pour l'image gaussienne de ce point. Le sens de rotation donné par le déplacement de l'image gaussienne Q sur S^1 change, lorsque la courbure s'inverse (voir figure 6.5, page 117). En comparant le sens de parcours d'arcs de cercle définis entre deux images gaussiennes, la carte de Gauss permet d'identifier la courbure de la courbe correspondante. Lors de la polygonalisation d'une courbe, il existe une corrélation entre la courbure et le sens de parcours des trajectoires (interpolations linéaires issues de la polygonalisation). En effet, parcourir les trajectoires dans le sens trigonométrique direct revient à définir une courbure positive, sinon la courbure est négative.

Pour notre application, nous avons choisi de représenter le vecteur $\overrightarrow{V_k}$ sur la carte de Gauss. La *signature* de l'usinage dépend donc de l'évolution de l'image gaussienne du vecteur $\overrightarrow{V_k}$:

- Pour un usinage de type zigzag, le vecteur image de $\overrightarrow{V_k}$ sur la carte de Gauss effectue un va-et-vient sur une partie du cercle, sans jamais parcourir le cercle dans son intégralité. Ce va-et-vient correspond aux changement du sens de parcours des trajectoires lors du passage d'un zig vers un zag.
- Pour un usinage de type contours parallèles, le vecteur image de $\overrightarrow{V_k}$ sur la carte de Gauss tourne toujours dans le même sens autour du cercle unitaire S^1 puisque les trajectoires sont toujours parcourues dans le même sens (courbure de même signe tout au long de la séquence).

Utiliser la carte de Gauss en comparant les parcours d'arcs de cercle décrits par l'extrémité de l'image gaussienne du vecteur $\overrightarrow{V_k}$ revient donc à identifier la stratégie d'usinage de la simulation.

Sur la figure 6.6 de la page 119, nous avons représenté l'image $\overrightarrow{V_k^1}$ sur la carte de Gauss du vecteur $\overrightarrow{V_k}$ pour les séquences d'usinage de la figure 6.3 de la page 116. Nous vérifions bien que pour un usinage de type zigzag, le vecteur image $\overrightarrow{V_k^1}$ effectue un va-et-vient sur une partie du cercle (figure 6.6.1) et que pour un usinage de type contours parallèles, le vecteur image $\overrightarrow{V_k^1}$ parcourt plusieurs fois le cercle dans son intégralité (figure 6.6.2). Intéressons-nous plus particulièrement au vecteur image $\overrightarrow{V_5^1}$ issu d'un usinage de type contours parallèles. Ce vecteur est calculé à partir des 5 premières trajectoires de la figure 6.3.2. La 5^{eme} trajectoire fait partie du contour de *niveau 1* de la séquence d'usinage et, comme le montre la figure 6.6.2, l'angle $\theta_{5[niveau]}$ est strictement supérieur à 2π.

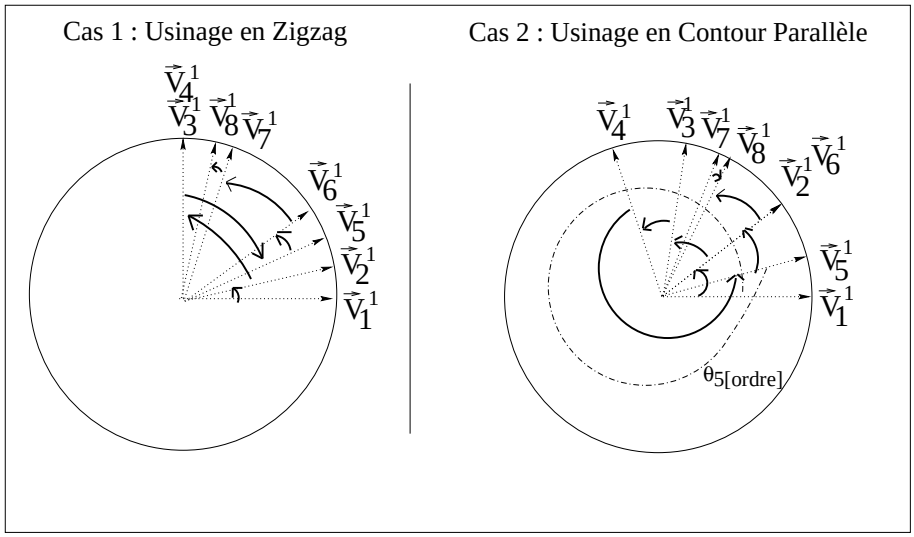

FIG. 6.6 – Déplacement du vecteur ramené à un vecteur unitaire et angle $\theta_{k[niveau]}$ $(k = 5)$

6.1.4 Méthode des Trajectoires proches

Pour reconnaître un motif zigzag, nous proposons une nouvelle approche qui s'intéresse également au sens de parcours des segments de trajectoires. Dans l'approche précédente, nous avons étudié le sens de parcours d'une succession de trajectoires, dans cette nouvelle approche, nous nous intéresserons au sens de parcours de deux trajectoires non successives, mais proches et de même inclinaison. Ainsi, d'après la *Caractéristique 1* de la définition d'un usinage en zigzag, les deux grandes trajectoires du motif zigzag sont opposées de π. Ce ne sont pas des trajectoires consécutives, mais des trajectoires *proches* et de même inclinaison. En effet, la distance entre ces deux trajectoires dépend de la distance entre passes (choisie lors de la génération du programme d'usinage) qui est au maximum égale à $2 \times (Rayon\ de\ l'Outil)$. Ceci nous amène à la définition suivante : deux trajectoires de même inclinaison sont dites *trajectoires proches* si et seulement si elles sont distantes d'une longueur inférieure ou égale à deux fois le rayon de l'outil.
Pour un usinage de type contours parallèles, nous remarquons que les trajectoires proches sont parcourues dans le même sens, puisque le motif est répétitif et les trajectoires suivent le contour décalé de la pièce. La règle suivante est à la base de notre démarche :

Règle :

- Dans un motif zigzag, la plupart des trajectoires proches sont parcourues en

sens inverse.

- Dans un motif contours parallèles, la plupart des trajectoires proches sont parcourues dans le même sens.

6.1.4.1 Reconnaissance de trajectoires proches

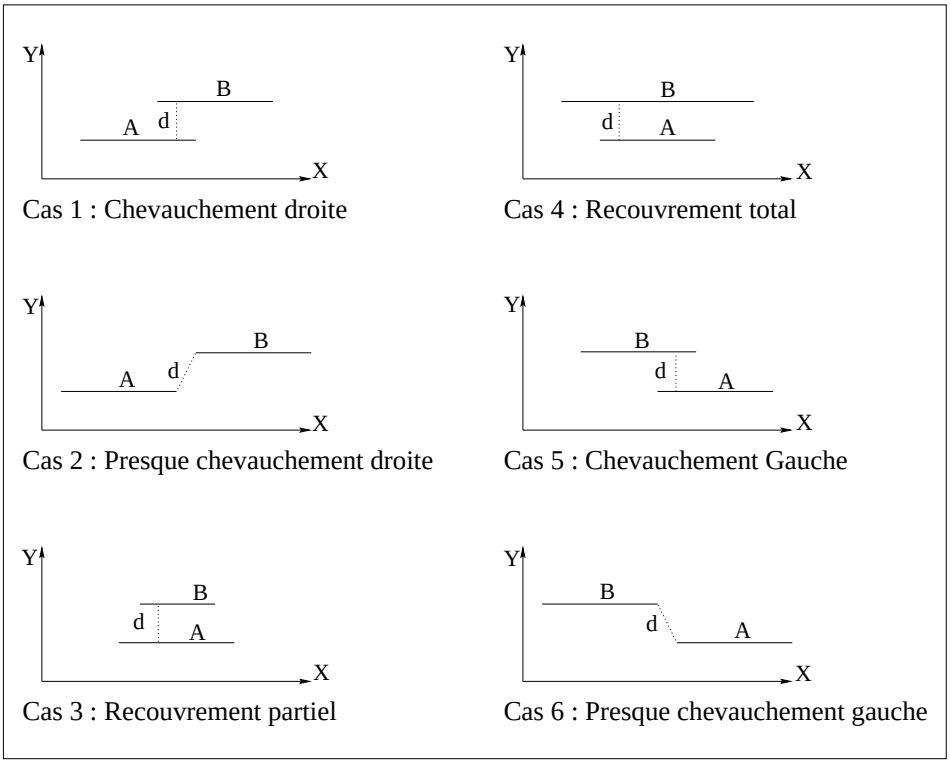

FIG. 6.7 – Trajectoires Proches de même orientation

Tout d'abord, il est nécessaire de pouvoir reconnaître deux trajectoires *proches* pour deux trajectoires données de même inclinaison. Sur la figure 6.7 de la page 120, nous avons répertorié les différentes positions entre deux trajectoires A et B de même inclinaison. Nous supposons que la trajectoire A est la trajectoire d'ordonnée inférieure ($Y_A \leq Y_B$) et nous notons par d la distance que nous souhaitons évaluer entre les deux segments de trajectoires A et B. La distance d entre deux segments A et B est calculée ainsi :
$d = \min_{x \in A, y \in B} \mid x - y \mid$, où $\mid x - y \mid$ est la distance euclidienne entre les points x et y.
Nous nous sommes volontairement placés dans un cas simple où la direction des trajectoires est parallèle à l'axe des X. Si les trajectoires étaient inclinées de $\alpha_{[0;\pi]}$

$(0 \leq \alpha_{[0;\pi]} \leq \pi)$ par rapport à l'axe des X, il suffirait de se ramener au cas simple par une rotation de $(-\alpha_{[0;\pi]})$ sur les trajectoires.

Dans les cas 1, 3, 4 et 5 de la figure 6.7, la trajectoire B chevauche la trajectoire A sur l'axe des X partiellement ou totalement. La distance $d = Y_B - Y_A$ entre ces deux trajectoires correspond à la différence des ordonnées des trajectoires A et B. D'après la définition précédente, ces trajectoires sont des trajectoires proches si et seulement si elles sont distantes d'au plus $2 \times (Rayon\ de\ l'Outil)$, c'est à dire si et seulement si $d \leq 2 \times (Rayon\ de\ l'Outil)$.

Dans les cas 2 et 6 de la figure 6.7, la trajectoire B ne chevauche pas la trajectoire A sur l'axe des X. Néanmoins, nous pouvons considérer que ces trajectoires sont proches si et seulement la nouvelle distance d reste inférieure ou égale à $2\times(Rayon\ de\ l'Outil)$. La nouvelle distance d correspond à la distance entre les deux extrémités les plus proches. Pour une trajectoire A donnée, nous décidons d'appeler ses extrémités X_{G_A} et X_{D_A} avec $X_{G_A} < X_{D_A}$.

- Le cas 2 peut correspondre à un chevauchement à droite. Il faut donc évaluer la distance entre l'extrémité droite de la trajectoire A et l'extrémité gauche de la trajectoire B, ce qui revient à calculer : $d = \sqrt{\Delta Y^2 + \Delta X^2}$ avec $\Delta Y = Y_B - Y_A$ et $\Delta X = X_{G_B} - X_{D_A}$
- Le cas 6 peut correspondre à un chevauchement à gauche. Il faut donc évaluer la distance entre l'extrémité gauche de la trajectoire A et l'extrémité droite de la trajectoire B, ce qui revient à calculer : $d = \sqrt{\Delta Y^2 + \Delta X^2}$ avec $\Delta Y = Y_B - Y_A$ et $\Delta X = X_{G_A} - X_{D_B}$

6.1.4.2 Recherche de trajectoires proches dans le fichier d'usinage

Supposons que le fichier d'usinage soit composé d'une succession de n trajectoires. Toutes les $n-1$ premières trajectoires du fichier vont être examinées les unes après les autres. Intéressons nous à la trajectoire t_i $(1 \leq i \leq n-1)$ d'inclinaison $\alpha_{i[0;\pi]}$. Les $i-1$ èmes trajectoires précédentes ont déjà été examinées, nous souhaitons donc trouver la première trajectoire de même orientation et proche de la trajectoire i parmi les $n-i$ trajectoires suivantes.
La premiere étape consiste à comparer l'inclinaison $\alpha_{t[0;\pi]}$ $(\alpha_{i+1[0;\pi]} \leq \alpha_{t[0;\pi]} \leq \alpha_{n[0;\pi]})$ des trajectoires restantes avec l'orientation $\alpha_{i[0;\pi]}$ de la trajectoire de *base*. Si ces orientations sont compatibles $(\alpha_{i[0;\pi]} - \Delta\alpha \leq \alpha_{t[0;\pi]} \leq \alpha_{i[0;\pi]} + \Delta\alpha)$, la proximité des trajectoires est alors testée en utilisant la méthode présentée dans la section précédente.
Si t_i et t_p sont des trajectoires proches de même inclinaison, nous allons *marquer*

la trajectoire i (puisque nous savons qu'il existe une trajectoire proche t_p associée à cette trajectoire t_i). C'est en examinant le sens de parcours des deux trajectoire grâce aux angles α_i ($0 \leq \alpha_i \leq 2\pi$) associés préalablement aux trajectoires, que nous préciserons le *sens de parcours* des trajectoires :

- Si ces trajectoires sont parcourues dans le même sens (même angle α à une erreur $\Delta\alpha$ près), la trajectoire i est marquée comme étant dans le *Même Sens* que sa trajectoire proche.
- Si ces trajectoires sont parcourues dans le sens inverse (angle α déphasé de π à une erreur $\Delta\alpha$ près), la trajectoire i est marquée comme étant dans le *Sens Inverse* de sa trajectoire proche.

Il est important de remarquer que :

- Pour pour un motif zigzag *idéal*, deux trajectoires proches de même inclinaison ne sont jamais consécutives puisqu'elles appartiennent à deux motifs d'usinage différents, ce qui est vérifié par exemple pour les trajectoires t_1 et t_3 de la figure 6.2.1 de la page 111.
- Pour un motif zigzag *reconstitué* comme celui de la figure 6.2.2 de la page 111, une grande trajectoire d'angle α n'est pas *franche*, mais est décomposée en *trajectoires réduites* d'angle $\alpha_{[0;\pi]}(\pm\Delta\alpha)$. Or si la longueur des *trajectoires réduites* est inférieure à $2 \times (Rayon\ de\ l'Outil)$, il peut s'avérer que la première trajectoire proche de même inclinaison que la trajectoire de base appartient en fait à la même grande trajectoire non franche que la trajectoire de base et donc au même motif. Pour éviter ce problème, nous considérerons que la recherche d'une trajectoire proche et de même inclinaison $\alpha_{i[0;\pi]}$ que la trajectoire t_i commencera non pas à partir de la première trajectoire suivante t_{i+1} mais à partir de la première trajectoire suivante t_j ($j \geq i+1$) qui vérifie $\alpha_{j[0;\pi]} > \alpha_{i[0;\pi]} + \Delta\alpha$ ou $\alpha_{j[0;\pi]} < \alpha_{i[0;\pi]} - \Delta\alpha$.

Reconnaissance de la stratégie d'usinage :
L'interprétation des résultats consiste à comparer le nombre de trajectoires marquées de *Même Sens* au nombre de trajectoires marquées en *Sens Inverse*. En effet si le nombre de trajectoires marquées de *Même Sens* représente plus de 80 % du nombre total des trajectoires marquées, nous considérons que la séquence d'usinage correspond à une séquence d'un usinage de type contours parallèles, sinon la séquence d'usinage correspond à une séquence d'usinage de type zigzag puisque la plupart des trajectoires proches sont parcourues en *Sens Inverse*, ce qui correspond au sens de parcours des grandes trajectoires d'un motif zigzag.

6.1.5 Cas particulier de l'usinage zig

L'usinage zig est un usinage unidirectionnel parfois appelé usinage en aller simple. A la fin de chaque trajectoire, l'outil de coupe est dégagé du brut et ramené par un déplacement rapide au point de départ de la trajectoire suivante. L'usinage en zig est donc composé de trajectoires linéaires parallèles toujours parcourues dans le même sens, ce qui, contrairement à un usinage de type zigzag, minimise la flexion de l'outil et permet de minimiser les erreurs d'usinage.

L'usinage zig dans les 3 méthodes précédentes
Nous allons étudier maintenant le comportement de l'usinage zig dans chacune des trois méthodes proposées précédemment.

- D'après la définition précédente, un usinage zig est essentiellement composé de *grandes* trajectoires. Dans la méthode dite du *classement des trajectoires*, la quasi égalité du nombre de petites et grandes trajectoires laisse supposer que la stratégie d'usinage employée est de type zigzag. L'inégalité du nombre de petites et grandes trajectoires peut correspondre soit à une stratégie d'usinage de type contours parallèles, soit à une stratégie d'usinage de type zig.
- Un usinage zig est essentiellement composé de grandes trajectoires *unidirectionnelles*. Le parcours de deux trajectoires successives s'effectue toujours dans le même sens, ce qui entraînera pour la méthode des trajectoires proches un résultat semblable à un usinage en contours parallèles.
- De plus, les trajectoires sont *parallèles* dans un usinage zig. Ainsi, deux trajectoires proches sont parcourues dans le même sens. L'extrémité de l'image gaussienne du vecteur $\overrightarrow{V_k}$ reste sur une partie du cercle, sans jamais parcourir le cercle dans son intégralité. La fonction angulaire cumulée φ reste égale à zéro comme les trajectoires sont parallèles entre elles. L'angle réel maximum $\theta_{max[niveau]}$ vérifie l'inégalité suivante : $-2\pi < \theta_{max} < 2\pi$ pour une stratégie d'usinage de type zig, ce qui était aussi le cas pour une stratégie d'usinage de type zigzag.

L'étude précédente montre que pour la méthode du classement des trajectoires et des trajectoires proches, une succession de trajectoires correspondant à un usinage zig donne les mêmes résultats qu'une succession de trajectoires correspondant à un usinage de type contours parallèles. Nous allons donc proposer une méthode qui permet de différencier ces deux types d'usinages.

Différence entre usinage de type zig et usinage de type contours parallèles
Pour différencier rapidement un usinage de type zig d'un usinage de type contours parallèles, nous utilisons l'angle α associé à chacune des trajectoires ($0 \leq \alpha \leq 2\pi$).

En effet, pour un usinage de type zig, les trajectoires sont unidirectionnelles, donc un seul angle α est extrait des données du programme d'usinage. Par contre, pour un usinage de type contours parallèles, les trajectoires suivent le contour de la pièce. Comme un contour fermé est composé au minimum de trois trajectoires d'orientation différente, un usinage de type contours parallèles est composé d'au moins 3 directions d'orientation α distinctes. Ainsi, nous supposons qu'une stratégie d'usinage de type contours parallèles est reconnue dès lors qu'en parcourant les trajectoires du fichier d'usinage 3 directions d'orientation α sont mises en évidence. Si toutes les trajectoires du fichier d'usinage sont parcourues et si moins de 3 directions d'orientation α sont extraites, on suppose que la stratégie d'usinage employée est alors de type zig.

6.1.6 Vers une reconnaissance de la stratégie d'usinage

6.1.6.1 Evaluation experimentale

Nous avons testé nos méthodes de reconnaissance d'usinage sur différents fichiers dont nous connaissons au préalable la stratégie d'usinage. Les fichiers peuvent être composés de plusieurs sous-parties d'usinage. Chaque sous-partie d'usinage est composée d'une succession de trajectoires comprises entre chaque changement d'outil. Une rétraction de l'outil de la matière peut également marquer la fin d'une sous-partie d'usinage. Pour ces fichiers, nous avons seulement analysé la première sous-partie d'usinage.

Nous avons travaillé sur six fichiers :

- les trois premiers fichiers utilisent une stratégie d'usinage en zigzag :
  - ZZIdéal : correspond à la totalité de la séquence d'usinage d'un fichier composé de 29 trajectoires formant des motifs idéaux en zigzag.
  - ZZIdéalPartie : correspond à la première sous-partie d'usinage (62 trajectoires formant des motifs ideaux en zigzag) de la séquence d'usinage d'un fichier composé de plusieurs sous-partie.
  - Voiture : correspond au premier fichier d'usinage présenté dans le chapitre 5 concernant l'évaluation expérimentale et composé de 17 108 trajectoires. Il est intéressant de rappeler que pour ce fichier d'usinage, les motifs zigzag sont des motifs approchés.

- les trois derniers fichiers utilisent une stratégie d'usinage en contours parallèles :
  - CPIdéal : correspond à la totalité de la séquence d'usinage d'un fichier composé de 45 trajectoires. A la fin de cette séquence d'usinage, la pièce usinée est identique à la pièce usinée par la séquence d'usinage du fichier ZZIdéal.
  - CPIdéalPartie : correspond à la première sous-partie d'usinage (76 trajectoires) de la séquence d'usinage d'un fichier composé de plusieurs sous-partie.
  - Cendrier : correspond à la première sous-partie d'usinage (composé des 59 premières trajectoires) du second fichier d'usinage présenté dans le chapitre 5 concernant l'évaluation expérimentale.

Les résultats issus de la méthode du classement des trajectoires et de la méthode des trajectoires proches sont donnés dans les tableaux ci-après.

	Classement des trajectoires		
Nom du fichier	PETITES traject.	GRANDES traject.	Erreur Relative
ZZIdéal	14	15	6,66%
ZZIdéalPartie	31	31	0,00%
Voiture	17 107	1	99,99%
CPIdéal	11	36	69,44%
CPIdéalPartie	31	45	31,11%
Cendrier	6	53	86,67%

	Méthode des trajectoires proches		
Nom du fichier	Nb. de Traject. Même Sens.	Nb. de Traject. Sens Inverse	% de Traject. Même Sens
ZZIdéal	8	10	44,44%
ZZIdéalPartie	13	13	50,00%
Voiture	17 084	1	99,99%
CPIdéal	28	0	100,00%
CPIdéalPartie	52	0	100,00%
Cendrier	54	0	100,00%

Pour la méthode du vecteur $\overrightarrow{V_k}$, nous trouvons :
- pour les trois fichiers utilisant une stratégie d'usinage en zigzag, l'angle maximum $\theta_{max[niveau]}$ tel que $-\pi \leq \theta_{max[niveau]} \leq \pi$.
- les trois fichiers utilisant une stratégie d'usinage en contours parallèles, l'angle maximum $\theta_{max[niveau]}$ tel que $\theta_{max[niveau]} > 2\pi$.

Nous avons également mesuré le temps d'exécution des trois méthodes. Pour le fichier de la voiture qui contient plus de 17 000 trajectoires, nous pouvons donner les ordres de grandeurs suivants :
- La méthode du classement des trajectoires s'exécute en quelques micro-secondes.
- La méthode du vecteur $\overrightarrow{V_k}$ s'exécute en quelques dizaines de micro-secondes.
- La méthode des trajectoires proches s'exécute en quelques centaines de micro-secondes.

Ces simulations montrent que la méthode du classement des trajectoires est la plus rapide des trois méthodes, mais aussi la moins robuste. La méthode du vecteur $\overrightarrow{V_k}$ et des trajectoires proches sont très robustes. Si nous devions n'utiliser qu'une seule méthode pour reconnaître la stratégie d'usinage d'un fichier, la méthode du vecteur $\overrightarrow{V_k}$ semble être la méthode la mieux adaptée car elle a un temps d'exécution minimal et est très robuste. Les temps d'exécution dépendent aussi de la complexité de l'usinage et du nombre croissant de trajectoires.

6.1.6.2 Mise en place de la reconnaissance d'usinage

Tout d'abord, nous avons remarqué expérimentalement que si le nombre de trajectoires à analyser est inférieur à 10, il est difficile de reconnaître une stratégie d'usinage. En effet, on ne peut parler de stratégie d'usinage que pour un nombre suffisant de trajectoires, le premier test consiste donc à comptabiliser le nombre de trajectoires du programme. Ensuite, nous pouvons utiliser les méthodes présentées précédemment.

Pour reconnaître la stratégie d'usinage, la méthode du vecteur $\overrightarrow{V_k}$ est la plus fiable, puisque l'angle maximum $\theta_{max[niveau]}$ du vecteur $\overrightarrow{V_k}$ est indépendant de la constitution des trajectoires : l'angle maximum $\theta_{max[niveau]}$ est identique que le programme d'usinage comporte une trajectoire franche ou la même trajectoire constituée de petites trajectoires. Cette méthode est donc plus robuste que la méthode du classement des trajectoires (qui se base uniquement sur la longueur des trajectoires). La méthode du vecteur $\overrightarrow{V_k}$ s'intéresse à l'enchaînement des trajectoires. Elle est assez rapide, puisqu'un seul traitement est effectué par trajectoire. Pour un programme d'usinage composé de n trajectoires, le temps d'exécution de cette méthode est donc

en $O(n)$. Elle sera donc utilisée en premier pour reconnaître la stratégie d'usinage dans un programme donné.

La méthode du classement des trajectoires et la méthode des trajectoires proches peuvent ensuite être utilisées pour une vérification de la stratégie d'usinage. La méthode du classement des trajectoires est la moins robuste, puisque la mise en évidence d'une stratégie de type zigzag devient impossible lorsque le programme d'usinage n'est constitué que par une succession de trajectoires réduites.

La méthode des trajectoires proches est quant à elle relativement robuste. Pour un programme d'usinage composé de n trajectoires, la première trajectoire est comparée au $(n-1)$ èmes trajectoires suivantes dans le pire des cas, la deuxième trajectoire est comparée au $(n-2)$ èmes trajectoires suivantes dans le pire des cas. Le temps d'exécution de cette méthode est donc dans le pire des cas en $O(n^2)$, ce qui rend cette méthode plus lente que les méthodes précédentes, et ceci dans tous les cas.

Ainsi en étudiant les caractéristiques des différentes stratégies utilisées pour l'usinage d'une pièce, nous avons mis au point trois méthodes distinctes et indépendantes qui permettent d'analyser et de traiter les données des trajectoires contenues dans le fichier d'usinage pour reconnaître la stratégie utilisée. L'utilisation d'une méthode ou la combinaison de 2 ou des 3 méthodes permet de reconnaître de manière relativement fiable la stratégie d'usinage employée ce qui nous permettra d'analyser plus facilement par la suite le programme d'usinage. En décomposant le programme initial en plusieurs sous-parties d'usinage, toutes les stratégies utilisées dans un même programme d'usinage sont mises en évidence en appliquant les méthodes de reconnaissance de la stratégie d'usinage sur chaque sous-partie du programme.

6.2 Evaluation mémoire en $2D\frac{1}{2}$

6.2.1 Présentation du problème

A partir des paramètres contenus dans le fichier d'usinage, nous souhaitons estimer le nombre de dexels et d'éléments en Z qui vont être créés au cours de la simulation. Cette estimation permet d'évaluer l'espace mémoire nécessaire au bon déroulement de la simulation et d'adapter éventuellement la taille de l'image (contenant la simulation d'usinage) à l'espace mémoire disponible sur la machine.

Pour évaluer le nombre de dexels et d'éléments en Z, nous allons nous placer dans un cas simple d'usinage. Notre étude s'appuie sur les conditions d'usinage suivantes :

1. L'usinage est un usinage de poche de type $2D\frac{1}{2}$, c'est à dire à profondeur constante.
2. Nous ne modifions qu'un seul degré de liberté (angle d'orientation φ) parmi les angles d'orientation de la pièce φ et θ (voir figure 5.3 de la page 78 du chapitre précédent). En effet, en fixant l'angle θ à $-90°$, nous conservons une profondeur constante en Z dans l'espace image (nous travaillons à profondeur constante et nous ne tolérons pas d'inclinaison de la pièce pour pouvoir calculer par la suite pour chaque déplacement de l'outil, une estimation du volume de matière enlevé sans tenir compte de projections).
3. Une seule stratégie d'usinage est utilisée. Elle peut être de type zigzag ou contours parallèles.

En effet, si la pièce est composée de plusieurs sous-parties d'usinage, nous traiterons chaque sous-partie séparément les unes après les autres.

6.2.2 Principe utilisé pour l'évaluation du nombre de dexels

6.2.2.1 Classification des dexels

Pour une image de dimensions données, nous devons réserver à l'initialisation un nombre $D_{initial}$ de dexels correspondants aux dimensions de l'image ($D_{initial}$ = Nombre de Lignes $\times$ Nombre de Colonnes). Les $D_{initial}$ dexels sont de deux types :

1. des dexels de type *fond de l'image*. Ils contiennent les valeurs par défaut pour NearZ et FarZ.
2. des dexels de type *brut*. Ils contiennent les valeurs appropriées pour NearZ et FarZ qui permettent de modéliser le brut non usiné.

Lors de la simulation d'usinage, D_c nouveaux dexels sont créés, D_d dexels sont supprimés. A la fin de la simulation, le nombre total de dexels utilisés est donc de $D_{total} = D_{initial} + D_c$. Le nombre final de dexels utilisés pour représenter la pièce en fin de simulation pour une image de dimensions données est donc de $D_{final} = D_{initial} + D_c - D_d$. Evaluer toute la mémoire nécessaire à la bonne exécution de la simulation revient à estimer, à partir des paramètres du programme d'usinage, le nombre de dexels créés D_c.[2]

6.2.2.2 Principe envisagé pour l'estimation du nombre de dexels utilisés

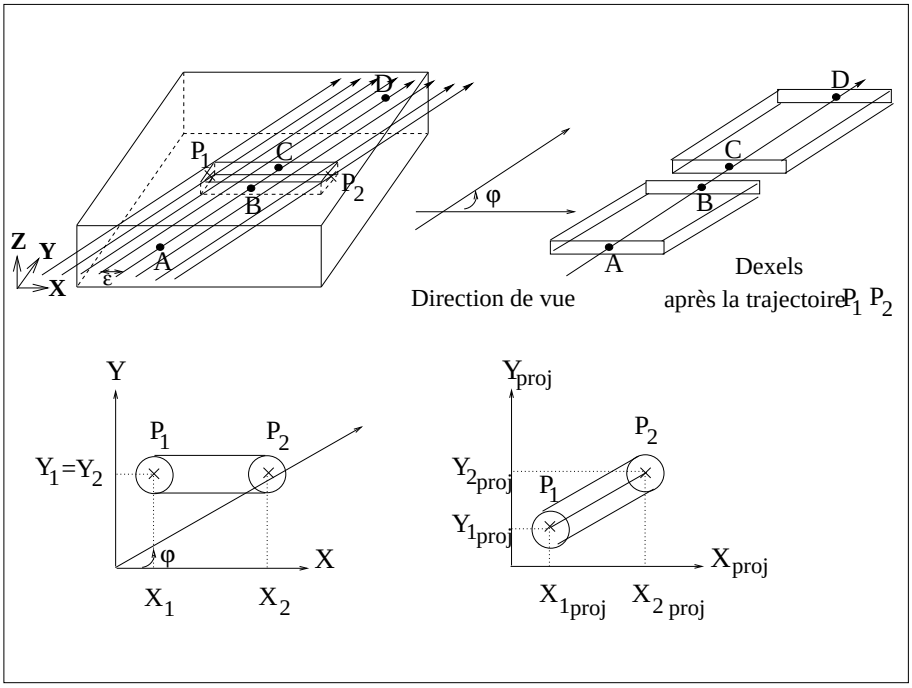

FIG. 6.8 – Visualisation des dexels créés pour une trajectoire isolée dans le brut

Pour estimer le nombre de dexels utilisés dans la simulation nous allons évaluer le nombre de points d'intersection des trajectoires de l'outil avec une famille de droites parallèles entre elles. Un résultat dû à Steinhaus [57] permet d'estimer la longueur

[2]Nous ne pouvons pas prévoir à l'avance à quel moment de la simulation les dexels seront supprimés. Ils peuvent être supprimés au fur et à mesure de leur création comme le montre la figure 4.3 de la page 64, ou à un moment quelconque de la simulation. C'est pour cela que nous allons évaluer le nombre total de dexels créés afin d'allouer un espace mémoire suffisant sans avoir besoin de *récupérer* la mémoire libérée par les dexels supprimés

d'une courbe Γ. Désignons par $M(\epsilon, \theta)$ le nombre de points d'intersection de Γ avec des lignes parallèles séparées d'une distance ϵ et de direction $\theta + \frac{\pi}{2}$; alors $\epsilon M(\epsilon, \theta)$ donne une estimation de la mesure de la projection de Γ sur une droite de direction θ où chaque point est compté avec sa multiplicité.

Dans notre application, on connaît la longueur de la courbe et nous cherchons le nombre de points d'intersection de rayons parallèles avec les trajectoires de l'outil. On obtient $M(\epsilon, \theta) = \frac{Longueur\ Projetee\ de\ \Gamma}{\epsilon}$. Malheureusement cette formule ne peut pas être appliquée directement car nous disposons de courbes d'une part d'épaisseur constante non nulle et d'autre part discrétisées. Nous devrons donc traiter les cas relatifs aux différentes stratégies d'usinage.

6.2.2.3 Evaluation du nombre de dexels pour une trajectoire isolée

Tout d'abord, nous souhaitons évaluer le nombre de dexels nécessaires à la mise en place d'une trajectoire outil isolée dans le brut. Nous avons représenté la trajectoire outil sur la figure 6.8 de la page 130 par ses points de départ $P_1(X_1, Y_1)$ et d'arrivée $P_2(X_2, Y_2)$. La direction de vue (orientation φ de la pièce) est représentée sur la figure 6.8 par une succession de droites parallèles orientées dans le même direction φ (par rapport à l'axe X) et espacées de ϵ. Ces droites représentent les rayons lancés à partir de chaque pixel, et ϵ représente la distance entre deux pixels, c'est à dire le pas d'échantillonnage.
Au moment de l'initialisation, un dexel est associé à chaque pixel du brut. Après la mise en place de la trajectoire $[P_1\ P_2]$, un nouveau dexel est créé pour chaque pixel se situant sur cette trajectoire. Ainsi, évaluer le nombre de nouveaux dexels créés par une trajectoire isolée revient à évaluer le nombre de points d'intersection de cette trajectoire avec les droites orientées dans la direction φ.
Projetons tout d'abord le segment $[P_1\ P_2]$ sur l'axe des X : $X_{1proj} = X_1 * \cos\varphi - Y_1 * \sin\varphi$ et $X_{2proj} = X_2 * \cos\varphi - Y_2 * \sin\varphi$, donc $LongueurProjetee_{[P_1P_2]} = |X_{2proj} - X_{1proj}|$. Le nombre de points d'intersection est alors de :$\lceil \frac{LongueurProjetee_{[P_1P_2]}}{\epsilon} \rceil$ (où $\lceil X \rceil$ est la partie entière supérieure de X).
Pour obtenir le nombre réel de points d'intersection des droites orientées avec la trajectoire, il faut se ramener à l'espace (X,Y,Z). Comme la profondeur reste constante (usinage en $2D\frac{1}{2}$) pour un angle θ fixé, le nombre de points d'intersection des droites avec la trajectoire de l'outil est égal à :
$D_c = \lceil \frac{LongueurProjetee_{[P_1P_2]}}{\epsilon} \rceil \times \lceil \frac{Profondeur}{\epsilon} \rceil$

6.2.2.4 évaluation du nombre de dexels pour deux trajectoires proches

Une trajectoire est rarement isolée car une pièce est usinée par un ensemble de trajectoires suivant une stratégie. La trajectoire isolée initiale est suivie de trajec-

toires proches qui la recouvrent partiellement. La distance entre passes est notée D_{EP} . Lors de la simulation, les trajectoires sont échantillonnées, l'outil avance donc de pixel en pixel à chaque étape de la simulation. Comme l'outil est circulaire, dans certains cas, de nouveaux dexels peuvent être créés pour une nouvelle position d'outil et supprimés lorsque l'outil continue sa progression sur la trajectoire. Si la distance entre passes est inférieure au rayon de l'outil, la trajectoire 2 ne créera pas de nouveaux dexels lors de son passage partiel sur de la trajectoire 1. Par contre si la distance entre passes est supérieure au rayon de l'outil alors de nouveaux dexels seront créés lorsque la trajectoire 2 est parallèle la trajectoire 1, comme le montre la figure 4.3 de la page 64. Les dexels sont créés au debut du passage de l'outil, puis supprimés lorsque l'outil poursuit sa trajectoire. Il faut donc comptabiliser ces dexels comme des dexels créés lors de la simulation. Le nombre de nouveaux dexels créés est donc égal à celui du nombre de dexels créés pour une trajectoire isolée c'est à dire : D_c

6.2.3 Evaluation du nombre de dexels pour une succession de trajectoires en zigzag

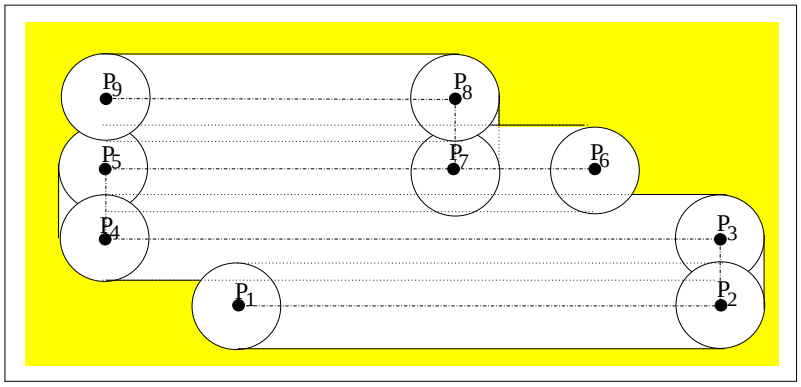

FIG. 6.9 – Exemple de trajectoires en zigzag

Dans le cas d'un usinage en zigzag comme celui présenté sur la figure 6.9 de la page 132, les trajectoires proches sont de grandes trajectoires.

– Si la distance entre passes D_{EP} est supérieure au rayon de l'outil alors les nouveaux dexels créés pour une grande trajectoire proche $[P_i\ P_{i+1}]$ se calculent comme précédemment, c'est à dire : $D_c = \lceil \frac{LongueurProjetee_{[P_i\ P_{i+1}]}}{\epsilon} \rceil \times \lceil \frac{Profondeur}{\epsilon} \rceil$.

– Si la distance entre passes D_{EP} est inférieure au rayon de l'outil alors de nouveaux dexels seront créés si les trajectoires proches ne se "superposent" pas totalement (comme dans un cas similaire aux trajectoires $[P_1\ P_2]$ et $[P_3\ P_4]$). Les nouveaux dexels créés correspondent à la différence de longueur projetée entre les deux trajectoires comme $[P_1\ P_4]$ sur la figure 6.9 de la page 132, il sont donc au nombre de : $D_c = \lceil \frac{LongueurProjetee_{[P_1\ P_4]}}{\epsilon} \rceil \times \lceil \frac{Profondeur}{\epsilon} \rceil$.

6.2.4 Evaluation du nombre de dexels pour une succession de trajectoires en contours parallèles

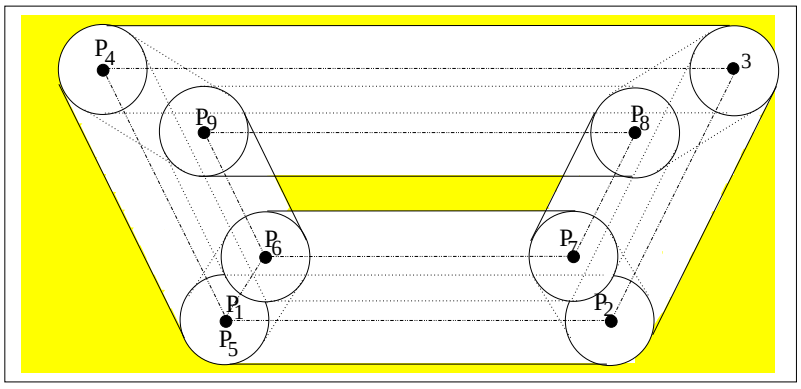

FIG. 6.10 – Exemple de trajectoires en contours parallèles

Dans le cas d'un usinage en contours parallèles comme celui présenté sur la figure 6.10 de la page 133, les trajectoires du premier contour sont des trajectoires isolées ($[P_1\ P_2]$, $[P_2\ P_3]$, $[P_3\ P_4]$, $[P_4\ P_5]$), et les trajectoires des autres contours sont considérées comme des trajectoires proches. Si la distance entre passes D_{EP} est supérieure au rayon de l'outil alors les nouveaux dexels créés pour une grande trajectoire proche $[P_i\ P_{i+1}]$ se calculent comme précédemment, c'est à dire : $D_c = \lceil \frac{LongueurProjetee_{[P_i\ P_{i+1}]}}{\epsilon} \rceil \times \lceil \frac{Profondeur}{\epsilon} \rceil$.

6.2.5 Evaluation du nombre d'éléments en Z

Nous souhaitons évaluer approximativement le nombre d'éléments en Z présents dans la trace en Z en fin de simulation pour une pièce usinée en $2D\frac{1}{2}$, et pour un point de vue fixé à $\theta = -90°$ et φ variable.
Mais pour commencer, nous allons évaluer le nombre d'éléments en Z nécessaires à

la construction d'une trace en Z correspondant à la création d'une trajectoire isolée dans le brut.

6.2.5.1 Evaluation du nombre d'éléments en Z pour une trajectoire

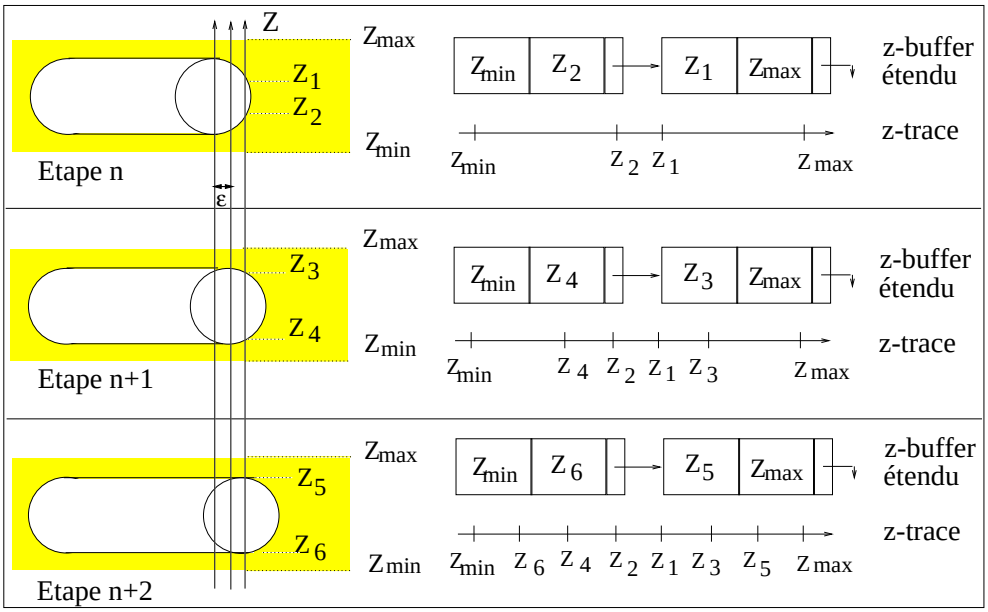

FIG. 6.11 – Evaluation du nombre d'éléments en Z

Comme le montre la figure 6.11 de la page 134, à chaque étape de la simulation, le NearZ et le FarZ du z-buffer étendu sont modifiés ce qui revient à rajouter deux nouveaux éléments en Z dans la z-trace. Ainsi en fin de trajectoire, l'enveloppe et l'*intérieur* (endroit du brut où l'outil attaque la matière) de la trajectoire sont mémorisés par un grand nombre d'éléments en Z.

La figure 6.12 de la page 135 permet de visualiser dans l'espace image la pièce et la trajectoire usinée. Les éléments en Z de la z-trace sont créés suivant l'axe des Z. A priori, le nombre de ces éléments en Z correspond au volume défini par l'enveloppe de la trajectoire, puisqu'à chaque nouvelle étape de la simulation l'outil change de position et *grignote* un peu plus de l'*intérieur* de la trajectoire. A priori, ce volume est constant quel que soit l'angle de vue φ choisi. Expérimentalement, nous avons voulu vérifier que le nombre d'éléments en Z de la z-trace pour un angle de vue φ de 0° est équivalent au nombre d'éléments en Z de la z-trace pour un angle de vue φ de 90°. Mais nous ne retrouvons pas le même nombre d'éléments en Z dans les deux cas. En réalité, il faut tenir compte de l'échantillonnage de l'outil. Dans la

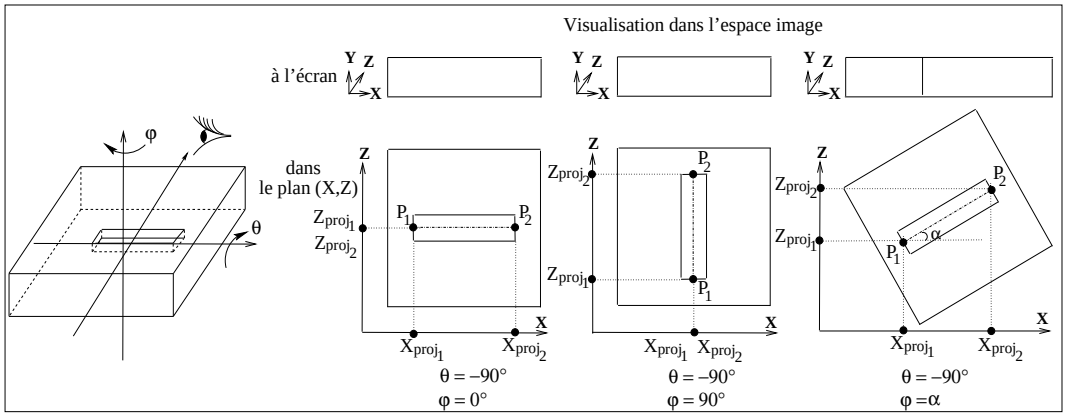

FIG. 6.12 – Visualisation de la trajectoire dans l'espace image

technique du z-buffer étendu appliquée à la simulation d'usinage, l'outil doit être échantillonné et décomposé en dexels afin de comparer les dexels du brut aux dexels de l'outil. La première partie de la figure 6.13 de la page 136 montre comment dans le plan (X,Z) l'outil, initialement assimilé à un cercle, est échantillonné. En fonction du pas d'échantillonnage, le contour de l'outil devient plus ou moins grossier dans le plan (X,Z). Cette *perte d'information* sur les contours de l'outil se traduira par une perte d'information dans la z-trace. La seconde partie de la figure 6.13 de la page 136 montre ce phénomène. Nous supposons que la trajectoire usinée est parallèle à l'axe des X et nous observons l'évolution d'une z-trace Z_{t_n} sur cette trajectoire à différentes étapes de la simulation. A l'étape i de la simulation, la z-trace commence à être modifiée, deux éléments en Z sont créés (Z_5 et Z_7). A l'étape $i+4$ de la simulation, la z-trace subit sa dernière modification, elle est alors composée de 8 éléments en Z (Z_1, Z_2, Z_3, Z_5, Z_7, Z_9, Z_{10} et Z_{11}). Le nombre d'éléments en Z attendu est alors de 11. Lorsque nous évaluerons le volume de la trajectoire, il faudra tenir compte de ces 3 éléments en $Z_{Manquants}$ sur chacune des z-traces.
Si la trajectoire usinée est parallèle à l'axe des Z , tous les Z attendus seront présents dans la z-trace finale : à chaque nouvelle étape, l'outil se déplace d'un Z, et donc ce Z est mémorisé dans la z-trace.

L'évaluation du nombre d'éléments en Z va dépendre de l'orientation α de la trajectoire dans le repère image.
Si la trajectoire est parallèle à l'axe des X ($\alpha = 0°$), il faut tenir compte des $Z_{Manquants}$.
Si la trajectoire est parallèle à l'axe des Z ($\alpha = 90°$), il n'y a pas de Z manquant.
Si la trajectoire est orientée dans une direction α quelconque (par rapport à l'axe

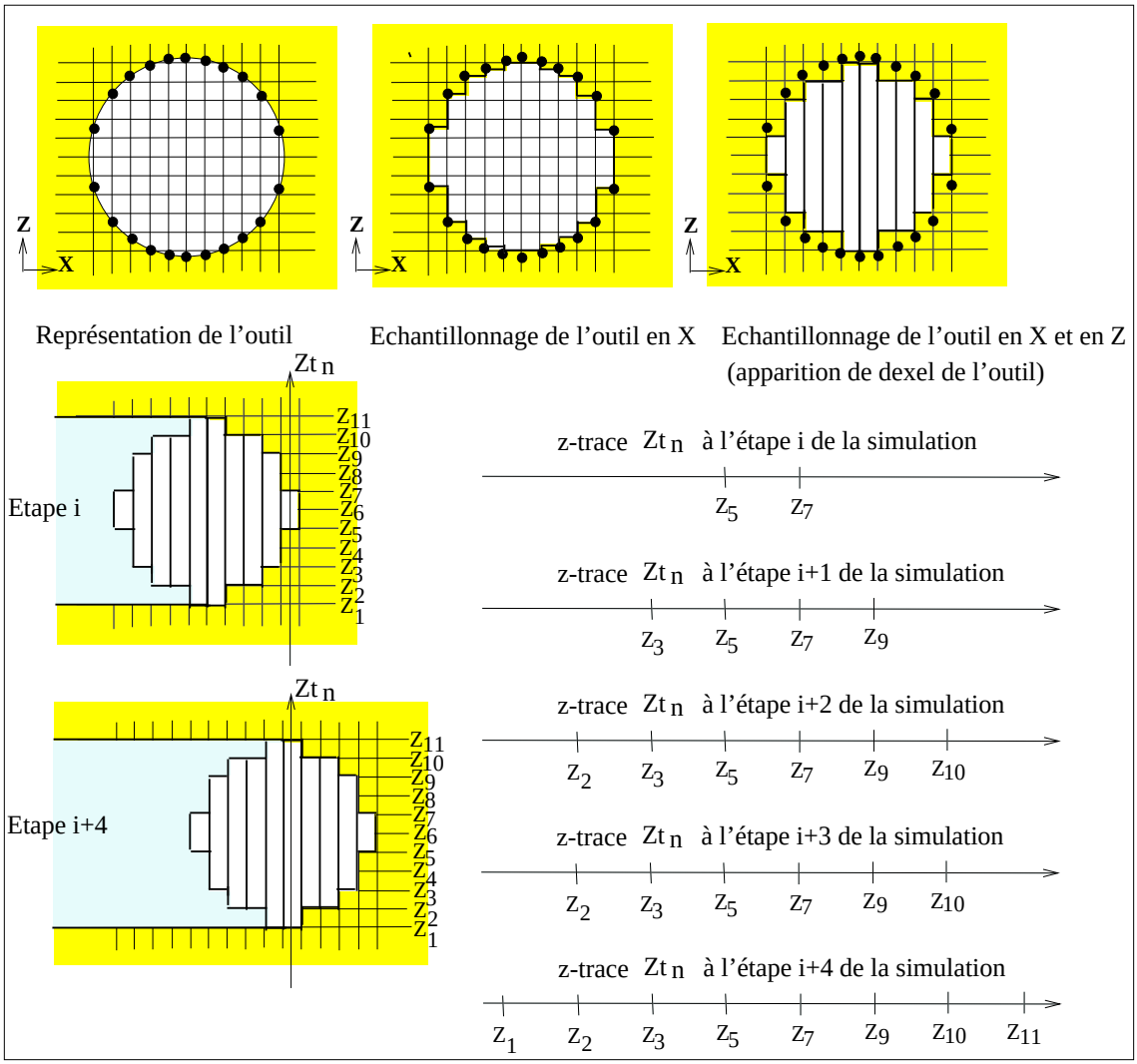

FIG. 6.13 – Échantillonnage de l'outil

des X) dans l'espace image (X,Z) , il faut décomposer le problème. Revenons à la figure 6.12 de la page 135, si $P_1(X_{proj_1},Y_{proj_1})$ est le début de la trajectoire et $P_2(X_{proj_2},Y_{proj_2})$ la fin de la trajectoire, la direction de la trajectoire dans le plan image (X, Z) est donnée par : $\alpha = \arctan(\frac{(Y_{proj_2}-Y_{proj_1})}{(X_{proj_2}-X_{proj_1})})$. La longueur de la trajectoire projetée sur X est : $L_X = \|X_{proj_2} - X_{proj_1}\|$ et la longueur de la trajectoire projetée sur Z est : $L_Z = \|Z_{proj_2} - Z_{proj_1}\|$. Notons ϵ, le pas d'échantillonnage, le nombre d'échantillons sur X correspondant à L_X est $Nb_X = \lceil \frac{L_X}{\epsilon} \rceil$ et le nombre d'échantillons sur Z correspondant à correspondant à L_Z est $Nb_Z = \lceil \frac{L_Z}{\epsilon} \rceil$.

Pour se déplacer de P_1 à P_2, nous utilisons l'algorithme de Bresenham. Grâce à Bresenham, nous effectuons soit des déplacements horizontaux (parallèle à l'axe des X), soit des déplacements verticaux (parallèle à l'axe des Z). Pour une trajectoire (P_1 P_2) de direction α, nous considérons que le nombre de déplacements horizontaux représente $(100 * \frac{|\cos\alpha|}{|\cos\alpha|+|\sin\alpha|})\%$ du nombre total de déplacements et le nombre de déplacements verticaux représente $(100 * \frac{|\sin\alpha|}{|\cos\alpha|+|\sin\alpha|})\%$ du nombre total de déplacements

6.2.5.2 Evaluation du nombre d'éléments en $Z_{Manquants}$ dus à l'échantillonnage de l'outil

Le nombre de $Z_{Manquants}$ dépend du rayon de l'outil R et du pas d'échantillonnage ϵ choisi. Pour déterminer le nombre de $Z_{Manquants}$ lors de l'échantillonnage de l'outil, il faut comparer pour chaque point d'échantillonnage en cours sur X, la hauteur en Z de ce point d'échantillonnage avec la hauteur en Z du point sur X de l'échantillonnage précédent. Si la différence de hauteur est nulle ou égale à 1, aucune information ne sera perdue lorsque l'outil avancera d'un élément en X, mais si cette différence de hauteur est supérieure à 1, la z-trace au final perdra $Z_{Manquants}$ éléments en Z de plus. Le nombre de $Z_{Manquants}$ est indépendant de la trajectoire, il doit être calculé uniquement à chaque changement d'outil ou d'échelle. L'algorithme suivant permet d'évaluer le nombre de $Z_{Manquants}$:

EvaluationNombreZManquants
$NbPoints_{Rayon_X} = \lceil \frac{RayonOutil}{\epsilon} \rceil$;
$HauteurZ_{Precedente}$=0 ;
$Z_{Manquants}$=0 ;
pour i := 1 .. $NbPoints_{Rayon_X}$ faire
$\cos\gamma = \frac{(NbPoints_{Rayon_X} - i) \times \epsilon}{R}$;
$\gamma = \arccos\gamma$;
$HauteurZ_{Courante} = \lfloor \frac{R \times \sin\gamma}{\epsilon} \rfloor$;
si $|HauteurZ_{Courante} - HauteurZ_{Precedente}| > 1$
alors $Z_{Manquants} = Z_{Manquants} +$
$2 \times (|HauteurZ_{Courante} - HauteurZ_{Precedente}| - 1)$;
fin si
$HauteurZ_{Precedente} = HauteurZ_{Courante}$
fin pour

Evaluation du nombre d'éléments en Z pour une trajectoire $(P_1\ P_2)$ **d'orientation** α **quelconque dans le plan (X,Z) image**

Nous évaluons tout d'abord le volume correspondant à une trajectoire parallèle à l'axe des X. Nous considérons que la longueur de cette trajectoire est la longueur projetée en X de la trajectoire $(P_1\ P_2)$. Comme l'outil effectue uniquement des déplacements horizontaux le long de cette trajectoire, il ne faut pas oublier de soustraire le volume dû aux $Z_{Manquants}$. Au final, le volume engendré par la trajectoire $(P_1\ P_2)$ sur X est égal à Vol_X :
$Vol_X = (\frac{L_X \times 2 \times RayonOutil \times Profondeur}{\epsilon \times \epsilon \times \epsilon}) - (\frac{L_X \times Z_{Manquants} \times Profondeur}{\epsilon \times \epsilon})$

Pour calculer le volume correspondant à un déplacement de l'outil parallèle à l'axe des Z, nous travaillons sur la trajectoire de longueur L_Z. Pour un tel déplacement, il n'y a pas de $Z_{Manquants}$, le volume engendré par la trajectoire $(P_1\ P_2)$ sur Z est égal à Vol_Z :
$Vol_Z = (\frac{L_Z \times 2 \times RayonOutil \times Profondeur}{\epsilon \times \epsilon \times \epsilon})$

Comme la trajectoire T_i $(P_1\ P_2)$ a une orientation α, il faut tenir compte de Bresenham pour évaluer le nombre d'éléments en Z au total dans la z-trace :
$Z_{Total_{T_i}} = \frac{(Vol_X \times |\cos\alpha|) + (Vol_Z \times |\sin\alpha|)}{|\cos\alpha| + |\sin\alpha|}$

6.2.5.3 Evaluation du nombre d'éléments en Z pour un ensemble de trajectoires suivant une stratégie d'usinage en zigzag ou en contours parallèles

La distance entre passes D_{EP} est un paramètre choisi par l'utilisateur lors de l'élaboration du fichier d'usinage. Il est commun à l'ensemble des trajectoires d'une même stratégie d'usinage et il permet de définir les trajectoires de telle manière qu'elles se "recouvrent" partiellement les unes sur les autres et cela quelle que soit la stratégie choisie (zigzag ou contours parallèles). Le recouvrement est total si D_{EP} est nul, le regroupement est partiel si $0 < D_{EP} < 2 \times RayonOutil$, et il n'y a pas de recouvrement lorsque $D_{EP} > 2 \times RayonOutil$. Nous supposons par la suite que D_{EP} a été correctement choisi $(0 < D_{EP} < 2 \times RayonOutil)$.
Tout d'abord, prenons un cas idéal composé de deux trajectoires proches T_1 et T_2 de même longueur et l'une rigoureusement au-dessus de l'autre. Elles ont aussi la même largeur puisque la largeur est fixe et égale à $2 \times RayonOutil$. Nous remarquons que ces trajectoires proches se recouvrent à 50% lorsque $D_{EP} = RayonOutil$. Ainsi pour un D_{EP} quelconque, la trajectoire T_2 recouvre $(100 - (\frac{50 \times D_{EP}}{RayonOutil}))\%$ de la trajectoire T_1. Si nous évaluons séparément l'ensemble des éléments en Z des deux trajectoires $Z_{Total} = Z_{Total_1} + Z_{Total_2}$, nous comptabilisons en double les éléments en Z communs

aux deux trajectoires. Le nombre d'éléments en Z réellement utilisés dans la z-trace est donc seulement de $(\frac{50\times D_{EP}}{RayonOutil}))\%$ des Z_{Total}.
Si maintenant, nous nous intéressons au cas réel, les trajectoires proches T_1 et T_2 ne sont pas forcément ni de même longueur, ni l'une rigoureusement au-dessus de l'autre. Le recouvrement peut avoir lieu, soit sur toute la trajectoire T_1, soit sur une partie. En réalité, nous ne recherchons pas le nombre exact d'éléments en Z réellement utilisés dans la z-trace, mais plutôt une approximation. De plus, un ensemble de n trajectoires, lorsqu'elles suivent une stratégie, usine généralement une poche, c'est pourquoi nous considérons que le nombre d'éléments en Z approchés correspond à $(\frac{50\times D_{EP}}{RayonOutil}))\%$ des Z_{Total} de toutes les trajectoires, même si les trajectoires T_1 et T_2 ne sont pas exactement de même longueur. Pour majorer cette approximation, et réserver assez de mémoire, nous majorons le nombre d'éléments en Z ainsi trouvés de 20%.
Ainsi, pour un ensemble de n trajectoires suivant une stratégie d'usinage définie à partir d'une distance de passes D_{EP}, nous pouvons approximer le nombre d'éléments en Z nécessaires à la construction de la z-trace correspondante en appliquant la formule suivante :
$$Z_{Total_{Approx}} = 1,2\times(\sum_{i=1}^{n} Z_{Total_{T_i}})\times(\frac{50\times D_{EP}}{100\times RayonOutil}) = 1,2\times(\sum_{i=1}^{n} Z_{Total_{T_i}}\times(\frac{D_{EP}}{2\times RayonOutil}))$$

6.2.6 Evaluation expérimentale

6.2.6.1 Mise en place de l'évaluation expérimentale

Nous avons testé l'évaluation du nombre de dexels et celle du nombre d'éléments en Z sur le fichier ZZIdéal qui correspond à un usinage en zigzag semblable à celui proposé sur la première partie de la figure 6.1. Dans ce fichier, l'angle α des grandes trajectoires du motif zigzag est de 90°.

Nous avons travaillé avec trois échelles (ϵ) différentes :
- $\epsilon = 0,38$ qui correspond à une image de 640 colonnes et 480 lignes.
- $\epsilon = 0,57$ qui correspond à une image de 320 colonnes et 480 lignes.
- $\epsilon = 0,76$ qui correspond à une image de 320 colonnes et 240 lignes.

Tout d'abord, nous avons relevé le nombre réel de dexels utilisés par le z-buffer étendu au cours de la simulation (notés D_r) et le nombre réel d'éléments en Z créés par la z-trace (notés Z_r). Ensuite, une estimation du nombre de dexels (notés D_e) et du nombre d'éléments en Z (notés Z_e) a été effectuée. Ces simulations ont été réalisées pour une orientation de la pièce allant de $\varphi = 0°$ à 180°. La pièce étant symétrique, les orientations de $\varphi = 180°$ à 360° donnent les mêmes résultats que les

orientations précédentes.

6.2.6.2 Evolution du nombre réel de dexels en fonction de la direction de vue et de l'orientation des trajectoires

Le graphe 6.15 montre l'évolution du nombre de dexels en fonction de l'angle de vue.

- **Pour une image de 640 colonnes et 480 lignes**, nous constatons que D_r est minimal pour les vues où l'angle φ a la même valeur que l'angle α des grandes trajectoires (8 100 pour $\varphi = 90°$). Dans ce cas-là, l'outil ne crée pas de nouveau dexel à chacun de ses déplacements le long d'une même grande trajectoire puisqu'il se déplace suivant une direction de vue : un seul et même dexel est retouché (*raboté*) tout au long de ce parcours.
  Par contre, pour les directions de vue dont l'angle φ est décalé de 90° (modulo 180°) par rapport aux angles α des grandes trajectoires, nous observons un nombre maximal de dexels (91 013 pour $\varphi = 90°$). Pour expliquer ce phénomène nous devons revenir sur l'échantillonnage de l'outil qui est représenté sur la figure 6.14.

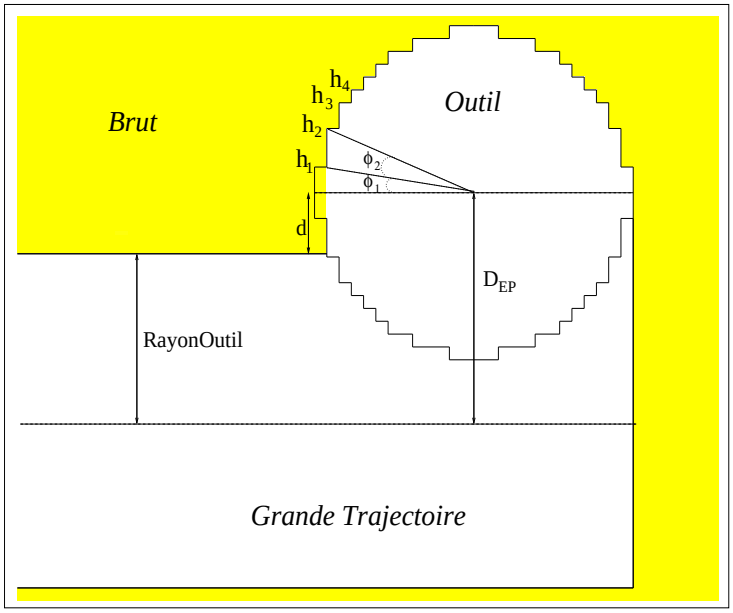

FIG. 6.14 – Echantillonnage de l'outil pour une grande image de 640 colonnes et 480 lignes

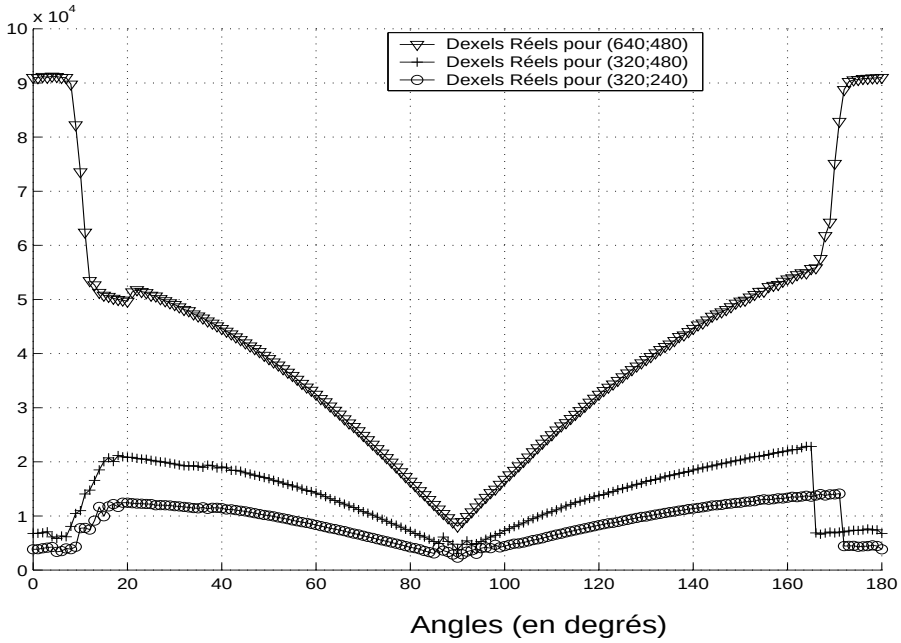

FIG. 6.15 – Evolution du nombre réel de dexels en fonction de l'angle de vue.

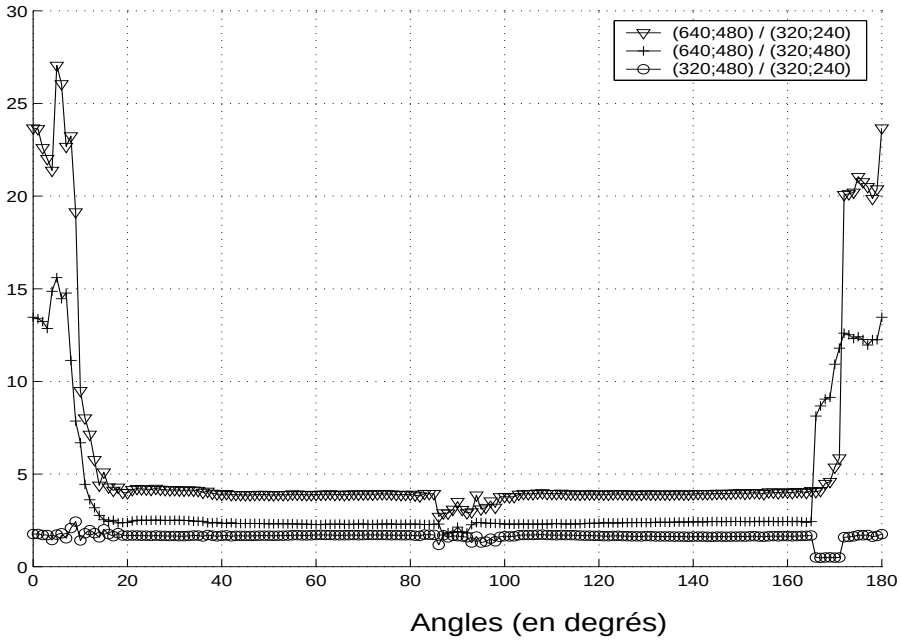

FIG. 6.16 – Evolution du rapport du nombre réel de dexels pour deux échelles différentes et trois dimensions d'images en fonction de l'angle de vue.

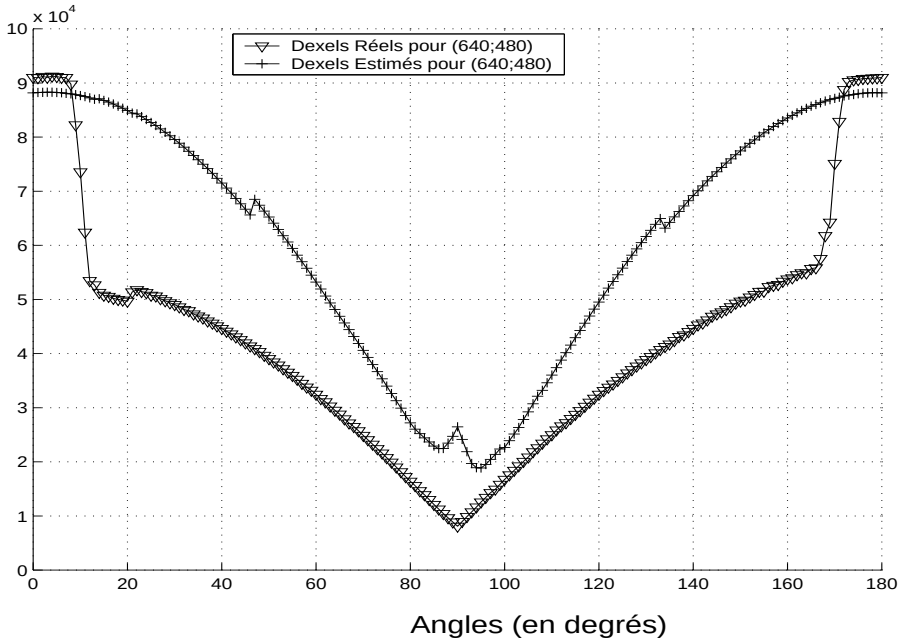

FIG. 6.17 – Evolution de D_r et D_e en fonction de l'angle de vue pour une image de 640 colonnes et 480 lignes.

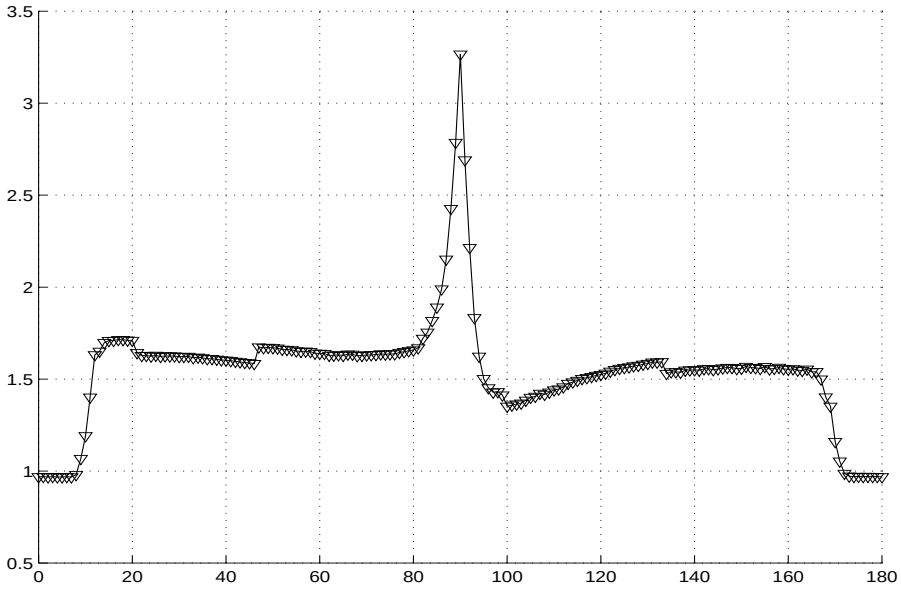

FIG. 6.18 – Evolution du rapport $\frac{D_e}{D_r}$ pour une image de 640 colonnes et 480 lignes.

Notons h_1 la valeur du premier palier d'échantillonnage, h_2 la valeur du deuxième palier d'échantillonnage. Nous avions vu dans le chapitre sur les structures de données (figure 4.3 de la page 64) que des dexels peuvent être créés puis supprimés entre deux trajectoires proches. Cela dépend de la position de l'outil par rapport à la grande trajectoire précédente, c'est à dire de la distance entre passes D_{EP}. Notons d la distance telle que :

$$d = D_{EP} - RayonOutil$$

- Si $d > h_1$ alors à chaque avancée de l'outil pour un angle de vue décalé de 90° par rapport à l'angle α des grandes trajectoires, un dexel supplémentaire est créé pour ensuite être supprimé
- Si $d \leq h_1$ alors aucun dexel supplémentaire n'est créé.

Dans notre simulation (pour $\epsilon = 0.38$), $d > h_1$, ce qui explique que D_r soit maximal pour un angle de vue décalé de 90° par rapport à l'angle des grandes trajectoires (soit $\varphi = 0°$ modulo 180°). Ce nombre de dexels est constant jusqu'à $\varphi = 8°$, puis il chute brusquement pour les angles de vue φ de 8° à 20°. Cette brusque diminution du nombre de dexels correspond au passage du premier palier d'échantillonnage (h_1 où $\Phi_1 = 8°$) au deuxième palier d'échantillonnage (h_2 où $\Phi_2 = 20°$). Le nombre de dexels diminue ensuite plus uniformément jusqu'à son minimum pour les raisons suivantes :

- Lorsque l'angle de vue φ se rapproche de l'angle α des grandes trajectoires, le phénomène des dexels supplémentaires créés et supprimés s'annule.
- Pour les angles de vue, où ce phénomène est toujours présent, les paliers d'échantillonnage sont plus réguliers, il n'y a plus de variation brusque du nombre de dexels. Nous pouvons toutefois noter une nouvelle variation autour de $\varphi = 30°$, ce qui correspond au passage du palier h_2 au palier h_3.

- **Pour une image de 320 colonnes et 480 lignes**, D_r est toujours minimal pour les vues où l'angle φ a la même valeur que l'angle α des grandes trajectoires.
  Par contre, pour les directions de vue dont l'angle φ est décalé de 90° (mod 180°) par rapport aux angles α des grandes trajectoires, nous n'observons pas un nombre maximal de dexels. En effet, pour cette image, nous nous trouvons dans le cas où $d \leq h_1$, aucun dexel supplémentaire n'est créé, le nombre de dexels n'est donc pas maximal. Au premier changement de palier d'échantillonnage, nous retrouvons la variation brusque du nombre de dexels comme précédemment.

- **Pour une image de 320 colonnes et 240 lignes**, nous observons la même tendance que la courbe précédente, puisque qu'avec une telle échelle nous nous trouvons toujours dans le cas où : $d \leq h_1$.

Le graphe 6.16 montre l'évolution du rapport du nombre réel de dexels pour deux échelles différentes et trois dimensions d'image en fonction de l'angle de vue. Mis à part l'angle de vue décalé de 90° par rapport aux grandes trajectoires (soit $\varphi = 0°$ modulo 180°), le rapport du nombre de dexels est à peu près constant.
Le rapport entre D_r pour une image de 640 colonnes par 480 lignes et D_r pour une image de 320 colonnes par 480 lignes est 2. En effet, comme le nombre de colonnes est divisé par 2, le nombre de dexels est aussi divisé par 2.
On constate les mêmes variations de rapport pour les autres tailles d'images lors de la division du nombre de colonnes et/ou de lignes.

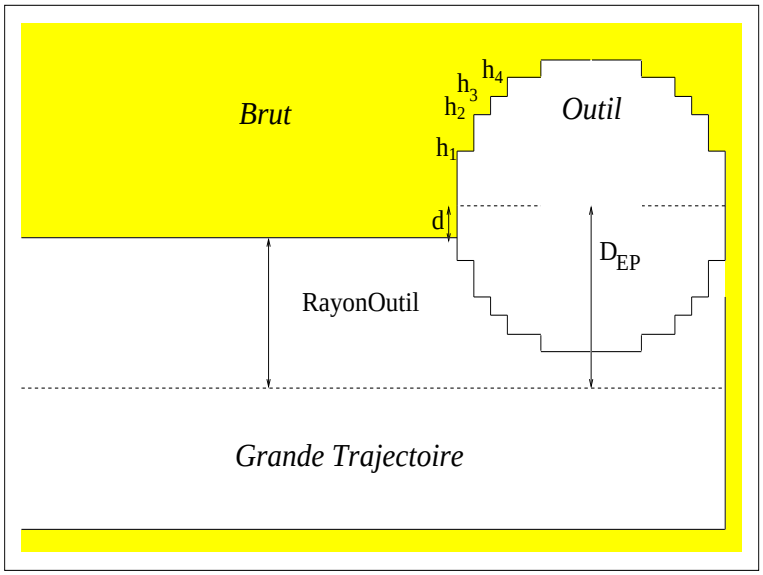

FIG. 6.19 – Echantillonnage de l'outil pour une image de 320 colonnes et 240 lignes.

6.2.6.3 Analyse des courbes correspondant à l'estimation du nombre de dexels

Les figures 6.17 et 6.18 permettent de comparer D_r et D_e. Ces deux courbes suivent la même tendance. La différence entre le nombre de dexels réel et estimé s'explique par la prise en compte ou non des dexels supplémentaires éventuellement créés.

L'évolution du rapport du nombre estimé de dexels pour deux échelles différentes et trois dimensions d'image en fonction de l'angle de vue est constant et dépend de la taille de l'image choisie. Il est identique au rapport du nombre réel de dexels pour deux échelles différentes et trois dimensions d'image.
Le rapport entre D_e pour une image de 640 colonnes par 480 lignes et D_e pour une image de 320 colonnes par 480 lignes est 2.
On constate les mêmes variations de rapport pour les autres tailles d'images lors de la division du nombre de colonnes et/ou de lignes.

6.2.6.4 Evolution du nombre réel d'éléments en Z en fonction de la direction de vue et de l'orientation des trajectoires

Le graphe 6.20 montre l'évolution du nombre réel d'éléments en Z en fonction de l'angle de vue. Quelle que soit l'image choisie, l'évolution de Z_r suit la même tendance. Z_r est maximal pour les vues où l'angle φ a la même valeur que l'angle α des grandes trajectoires (1 818 837 pour $\varphi = 90°$). Pour comprendre pourquoi le nombre d'éléments en Z est maximal pour une direction de vue dans la même direction que les grandes trajectoires, il est nécessaire de revenir sur l'explication des $Z_{Manquants}$ (voir page 137). Lorsque l'angle α de la trajectoire est le même que l'angle de vue φ (modulo 180°), la trajectoire est parallèle à l'axe des Z dans l'espace image. On ne tient plus compte des $Z_{Manquants}$ puisque tous les Z sont présents dans la z-trace finale. En effet, à chaque nouvelle étape, l'outil se déplace d'un Z, et donc ce Z est mémorisé dans la z-trace, le nombre d'éléments en Z est donc maximal.
Par conséquent, Z_r est minimal pour les angles de vue où φ est décalé de 90° (mod 180°) par rapport aux angles α des grandes trajectoires (1 066 735 pour $\varphi = 0°$), là où les $Z_{Manquants}$ sont les plus nombreux.

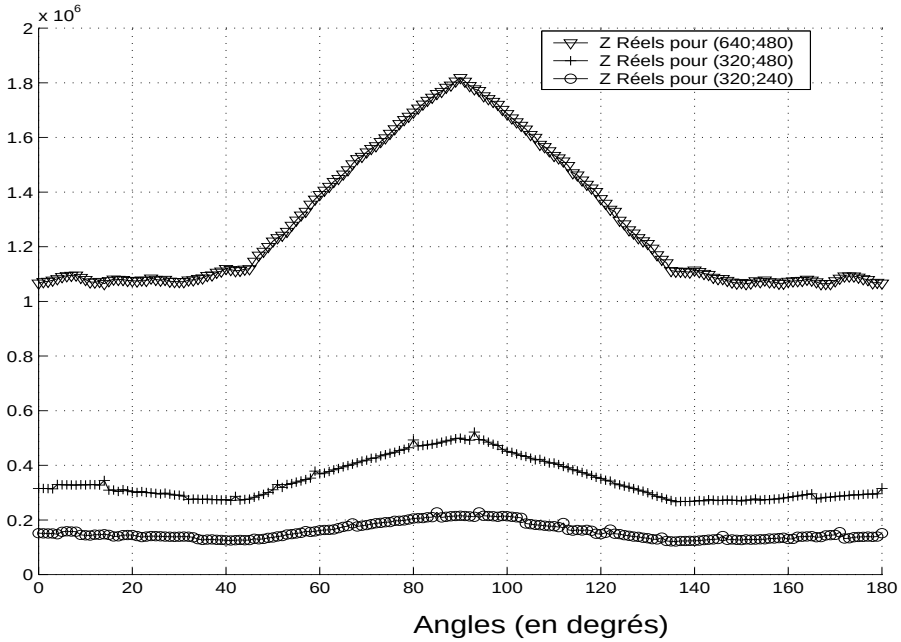

FIG. 6.20 – Evolution du nombre réel d'éléments en Z en fonction de l'angle de vue.

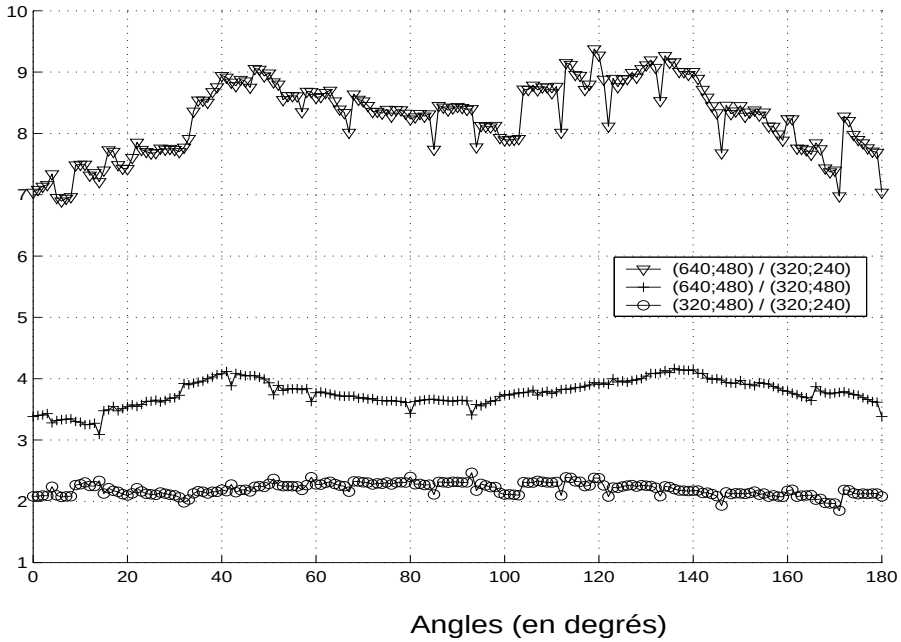

FIG. 6.21 – Evolution du rapport du nombre réel d'éléments en Z pour deux échelles différentes et trois dimensions d'image en fonction de l'angle de vue.

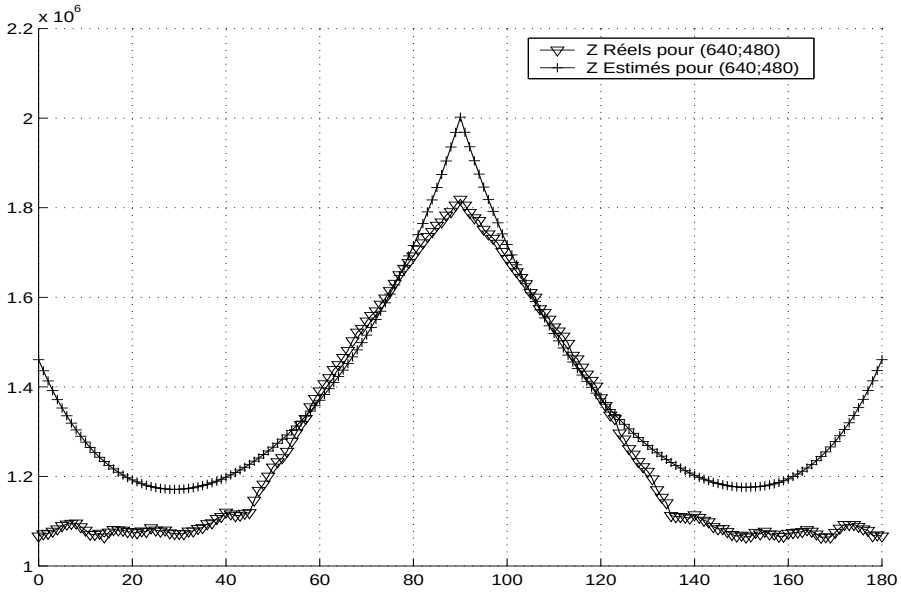

FIG. 6.22 – Evolution de Z_r et Z_e en fonction de l'angle de vue pour une image de 640 colonnes et 480 lignes.

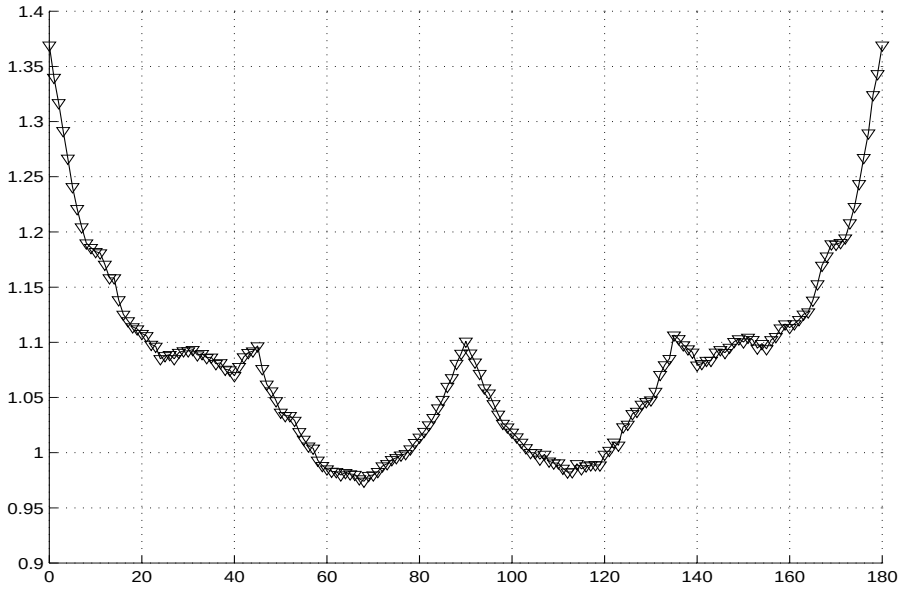

FIG. 6.23 – Evolution du rapport $\frac{Z_e}{Z_r}$ pour une image de 640 colonnes et 480 lignes.

Le graphe 6.21 montre l'évolution du rapport du nombre réel d'éléments en Z pour deux échelles différentes et trois dimensions d'image en fonction de l'angle de vue. Ce rapport ne peut plus être directement calculé à partir du nombre de colonnes et de lignes de chaque image. En effet, l'échantillonnage de l'outil entre en compte et il faut prendre en considération les $Z_{Manquants}$.

6.2.6.5 Analyse des courbes correspondant à l'estimation du nombre d'éléments en Z

La figure 6.22 permet de comparer le nombre réel d'éléments en Z nécessaires à la simulation et le nombre d'éléments en Z estimés. L'allure de la courbe représentant le nombre estimé d'éléments en Z suit l'allure de la courbe représentant le nombre réel d'éléments en Z. Le rapport entre les deux courbes est représenté sur la figure 6.23. Il est voisin de 1.

6.2.6.6 Temps de simulation

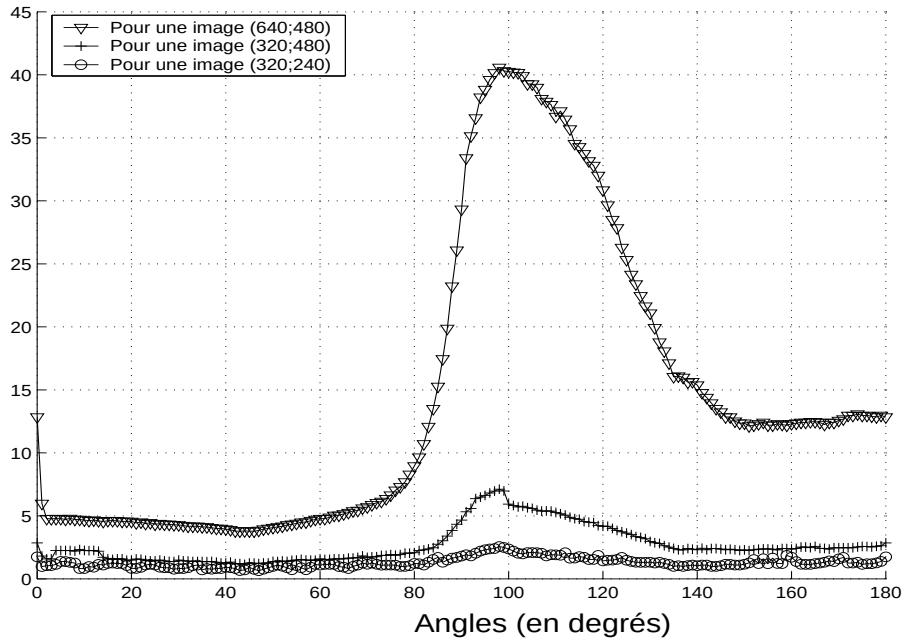

FIG. 6.24 – Evolution du temps de simulation en fonction de l'angle de vue (en secondes).

Le graphe 6.24 représente l'évolution du temps nécessaire pour mener à bien une simulation. Ce temps est de l'ordre de quelques secondes à quelques dizaines de secondes. Les vues où l'angle φ a la même valeur que l'angle α des grandes trajectoires ont un temps de simulation 8 fois plus important pour une image de 640 colonnes et 480 lignes que les vues où l'angle de vue est décalé de 90° par rapport à l'angle α des grandes trajectoires. Pour de telles vues ($\varphi = 90$), le nombre d'éléments en Z est maximal, les dexels seront donc très fréquemment modifiés ce qui influe sur le temps de simulation.

En revanche, l'estimation du nombre de dexels et du nombre d'éléments en Z est très rapide et ne nécessite que quelques millisecondes. Il sera donc intéressant de réaliser une estimation de la mémoire nécessaire en fonction de l'angle de vue et de déterminer la direction privilégiée des trajectoires. Ceci nous permet de réserver rapidement assez de mémoire pour mener à bien la simulation sans monopoliser toutefois trop de ressources, et de choisir la meilleure résolution et le meilleur angle de vue pour visualiser au mieux et au plus vite la simulation.

Conclusion

Avec l'évolution qui se dessine en CFAO (STEP NC, ISO 14649), la nécessité de la simulation d'usinage ne sera pas remise en cause ni, dans bien des cas, le rôle d'un opérateur qualifié pour vérifier et optimiser de façon interactive les programmes pièces.
Le travail présenté dans le cadre de cette thèse porte sur l'amélioration du z-buffer étendu, technique bien adaptée pour représenter les objets dans le domaine de la simulation d'usinage pratiquée dans les petits ateliers de mécanique.
Dans sa version initiale, le z-buffer étendu ne garde pas en mémoire l'historique de la construction de la pièce finale : pour revenir en arrière dans la visualisation, il faut reprendre la simulation depuis le début. L'introduction de deux *traces* comme nouvelles fonctionnalités du z-buffer étendu nous a permis d'envisager une simulation interactive :

- La trace en Z est une trace complète qui permet de reconstruire le modèle à un instant précis de la simulation
- La trace en temps affiche seulement un instant précis de la simulation, mais s'avère plus rapide.

Pour optimiser les performances des traces, nous avons étudié différentes structures de données. Nous avons d'une part utilisé les Skip Lists et nous avons d'autre part introduit les *Interval Treap*. Ces structures permettent une consultation rapide des données mémorisées entraînant ainsi une simulation conviviale au débit d'images régulier.
Nous avons également montré que l'évaluation préalable des ressources nécessaires au bon déroulement de la simulation peut s'effectuer très rapidement.
L'introduction des traces doit permettre une amélioration significative de la convivialité et des performances de la simulation d'usinage telle qu'elle peut se pratiquer dans les ateliers.

Ces méthodes sont coûteuses en mémoire mais restent compatibles avec les cartes graphiques, la capacité mémoire et les vitesses des processeurs des micro-ordinateurs utilisés aujourd'hui dans ces ateliers. De plus, l'évolution de la technologie des micro-

ordinateurs en ce qui concerne d'une part l'affichage graphique vers le transfert de matrices de pixels de la mémoire centrale vers la mémoire graphique ne peut être que bénéfique ; d'autre part, la mémoire disponible à un coût de plus en plus faible permet de rendre de moins en moins pénalisante la consommation élevée nécessaire .

Nous pensons que le z-buffer étendu avec les améliorations que nous proposons est une structure simple, robuste, qui au prix d'une perte d'information certaine peut traiter les nombreuses trajectoires actuellement incompatibles avec l'emploi d'un modèle exact.

Dans un proche avenir nous implanterons une procédure de calcul du taux d'enlèvement de matière, ce qui permettra de prolonger notre travail à un début de vérification des conditions de coupe.
La principale limitation de la méthode du z-buffer étendu provient de la nécessité de choisir un angle de vue avant la simulation, ce qui diminue l'interactivité et affecte la précision du modèle et la visibilité de certaines erreurs. Nous estimons que le découplage de la modélisation et de la visualisation dans la structure du z-buffer étendu en utilisant simultanément plusieurs directions de vue (orthogonales ou non) permettrait à moindre coût une meilleure précision. L'usage de l'algorithme des *marching cubes* rendrait certes plus complexe l'étape de modélisation en nécessitant le traitement de nombreux cas particuliers mais permettrait une précision accrue et une meilleure interactivité. Les améliorations que nous proposons s'intégreraient bien à cette nouvelle approche, que ce soit pour l'usage des traces ou pour l'estimation des ressources nécessaires à la simulation.

Annexe

A.1 L'usinage par enlèvement de matière

A.1.1 Généralités sur le fraisage

Les machines-outils sont classées principalement selon la nature de l'opération effectuée : enlèvement de matière, déformation , découpage, soudage, ... L'enlèvement de matière lui-même peut être obtenu par différents procédés physiques : enlèvement de copeaux, abrasion, électro-érosion, qui donnent lieu à l'existence d'autant de machines différentes. On trouve beaucoup de machines classiques de tournage et de fraisage, mais aussi, de plus en plus, des machines spécialisées *métiers* avec des architectures mécaniques diversifiées.

L'usinage par enlèvement de copeaux (tournage, fraisage), est un procédé de fabrication qui permet de réaliser une pièce par enlèvement progressif de matière. Cet enlèvement de matière est généré sur la base de la formation d'un copeau par mouvement relatif d'un élément, possédant une géométrie particulière (outil de coupe), par rapport à un autre élément, la matière à usiner (brut) (figure A.25, page 152).

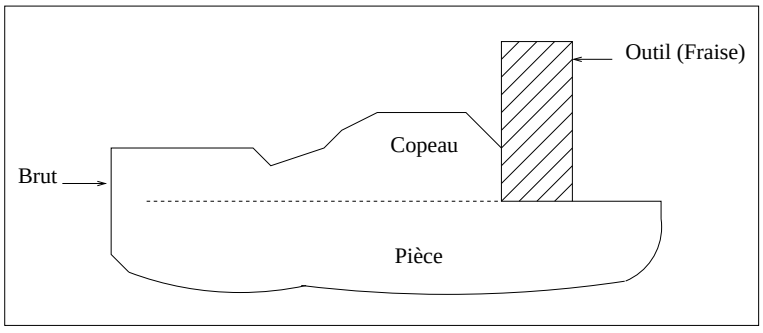

FIG. A.25 – Définition de l'usinage (fraisage).

La partie active de l'outil pénètre dans la matière pour séparer la partie à obtenir (copeau) de celle à récupérer (pièce à usiner). Le mouvement relatif, dans le cas du fraisage, résulte du déplacement de l'outil par rapport à la pièce solidaire de la table de la machine-outil (mouvement d'avance) et de la rotation de l'outil (mouvement de coupe). L'usinage d'une pièce est donc réalisé par la combinaison de ces deux mouvements (le mouvement d'avance appliqué à la pièce et le mouvement de coupe appliqué à l'outil).

Dans la suite de cette présentation, nous nous intéresserons plus particulièrement au fraisage. Une fraiseuse à trois axes est une machine qui usine principalement des pièces prismatiques (contrairement au tour qui permet d'usiner des pièces de révolution). La pièce est fixée dans une installation de bridage (table porte-pièce).

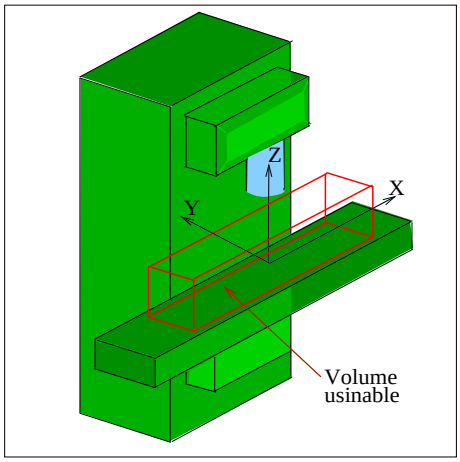

FIG. A.26 – Fraiseuse

L'outil de coupe est bloqué sur un système porte-fraise, lui-même fixé dans la broche de la machine-outil. Lors du fraisage, l'outil est mis en rotation, une lame d'outil (arête coupante) pénètre dans la matière et enlève un copeau. L'outil continue à suivre sa trajectoire qui interfère avec la pièce à usiner. Ces mouvements sont assurés par les mouvements constitutifs de la machine outil. Pour définir la trajectoire de l'outil au cours de l'usinage, un système d'axes est normalisé [58]. Ce système d'axes est défini à partir des normes NF Z 68-020 et ISO 841 qui précisent la désignation des axes et le sens de déplacements sur les axes. Les axes XYZ forment un repère direct (figure A.26, page 153) :

- **l'axe Z** correspond toujours à l'axe de la broche. C'est l'axe de rotation de l'outil pendant l'usinage.
- **l'axe X** correspond à l'axe perpendiculaire à Z qui permet le plus grand

déplacement de la table de la fraiseuse.
- **l'axe Y** correspond à l'axe perpendiculaire à X et à Z.
  Le **plan XY** représente donc le plan de table de la machine-outil.

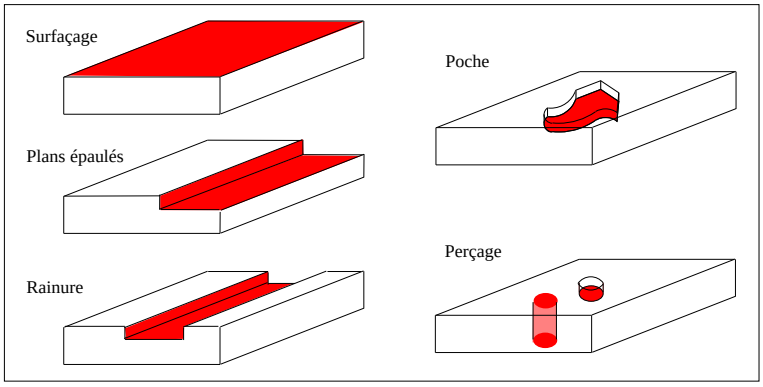

FIG. A.27 – Formes simples usinables en fraisage

Le déplacement de l'outil suivant les axes définis précédemment permet d'usiner une pièce de différentes manières. En réalité, l'outil se déplace uniquement suivant l'axe Z, les translations suivant les axes X et Y sont assurées par le déplacement de la table porte-pièce. Les principales formes usinables ou actions pour le fraisage dit en $2D\frac{1}{2}$ sont représentées[3] sur la figure A.27 de la page 154 :
- le surfaçage qui consiste à usiner un plan,
- les plans épaulés qui sont caractérisés par l'association de deux plans perpendiculaires,
- la rainure qui est caractérisée par l'association de trois plans perpendiculaires,
- la poche qui est une forme creuse dans la pièce, délimitée par des surfaces quelconques (plan, cylindre).
- le perçage qui s'identifie à des trous débouchants ou borgnes.

L'usinage d'une pièce se réalise en plusieurs étapes. La pièce, avant usinage (le brut), est pourvue d'une surépaisseur. La valeur de cette surépaisseur est fonction de la gamme d'usinage et des différentes contraintes technologiques. En fonction de cette surépaisseur et de l'état de surface demandé, trois étapes peuvent (ou non, selon les cas) être distinguées :
- **L'ébauche** : c'est la première étape dans l'usinage d'une poche. Elle consiste en un enlèvement de matière important en éliminant l'excédent de matière afin de préparer l'étape de finition.

[3] à profondeur de coupe constante

- **La demi-finition** : cette étape consiste à réaliser la surface à usiner avec une faible surépaisseur. Elle corrige les défauts résultant d'une grosse ébauche.
- **La finition** : c'est l'étape finale de l'usinage. Elle consiste à réaliser la surface à usiner en respectant toutes les spécifications imposées par le dossier de définition de la pièce. A l'issue de cette étape, la forme, la rugosité et les dimensions désirées sont obtenues pour les surfaces usinées.

La géométrie de l'outil influe directement sur les formes usinables. Les outils les plus couramment utilisés sont les fraises cylindrique, torique et sphérique (figure A.28, page 155). La fraise cylindrique est en général réservée pour l'ébauche. Les fraises sphérique et torique sont utilisées pour réaliser des surfaces quelconques. Cependant, la fraise sphérique ne permet pas de réaliser un usinage en $2D\frac{1}{2}$ car la vitesse au point O est nulle lorsque la fraise est en mouvement. D'autres fraises sont réservées à des usinages particuliers (rainures,...), elles sont appelées les fraises de formes.

Suivant le nombre d'arêtes tranchantes, on distingue les fraises à une, deux ou trois tailles.

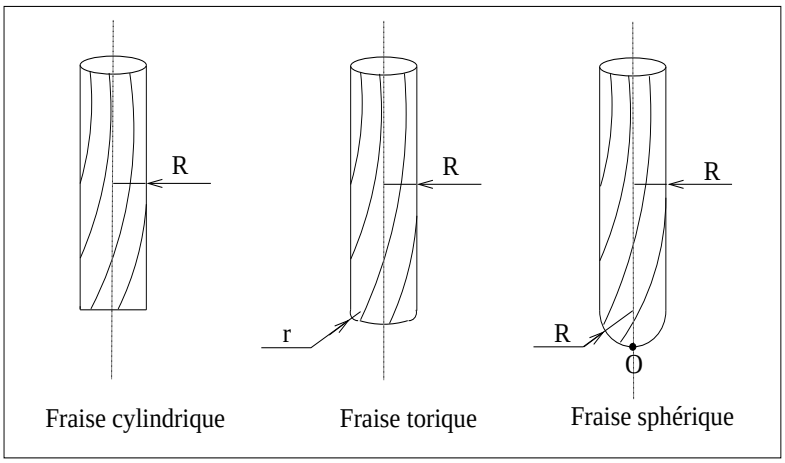

FIG. A.28 – Outils utilisés.

Le fraisage peut s'effectuer en deux modes de travail de la fraise comme le montre la figure A.29, page 156 :

- **Fraisage en opposition** : l'attaque d'une dent de la fraise se fait avec une épaisseur de copeau nulle. Chaque dent attaque une surface écrouie par la dent précédente.
- **Fraisage en concordance**, ou *en avalant* : l'attaque de la fraise se fait sur un copeau épais. Les dents attaquent directement sur une grande largeur de copeau.

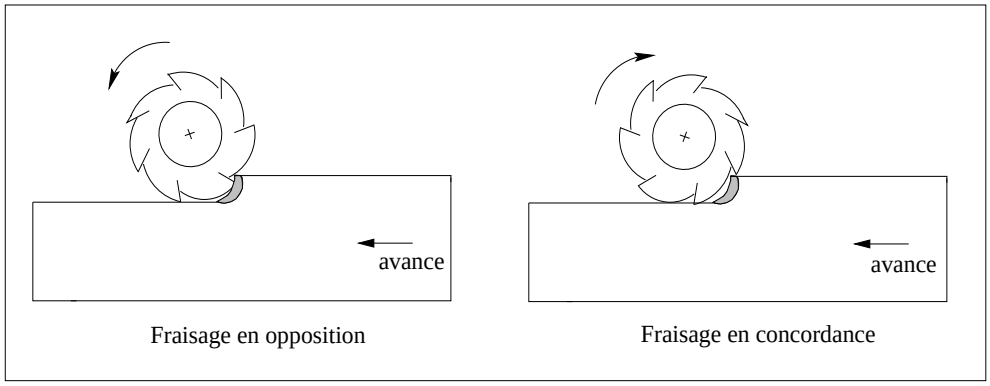

FIG. A.29 – Modes de fraisage.

Pour arriver à usiner une pièce dans de bonnes conditions (bon état de la surface usinée, rapidité de l'usinage, usure modéré de l'outil,...), l'opérateur doit tenir compte de divers paramètres tels que le type et la puissance de la machine, la matière usinée (acier, aluminium), la matière de l'outil, le type de l'opération. Pour cela, il faut régler des paramètres spécifiques appelés paramètres de coupe représentés sur la figure A.30 de la page 157 :

- **la vitesse de coupe** (V_c) qui correspond au déplacement de l'arête de coupe par rapport à la pièce.
- **la vitesse d'avance** (F) qui correspond au déplacement de l'outil sur la trajectoire d'usinage.
- **la profondeur de passe** (a) qui permet de déterminer le volume de copeau

A.1.2 Définition de l'usinage de poche

Une *poche* (figure A.31, page 157), est considérée comme une surface plane délimitée par un *profil extérieur* fermé et qui ne présente pas d'auto-intersubsection. Elle peut contenir en son intérieur des parties de matière non usinées appelés *îlots*

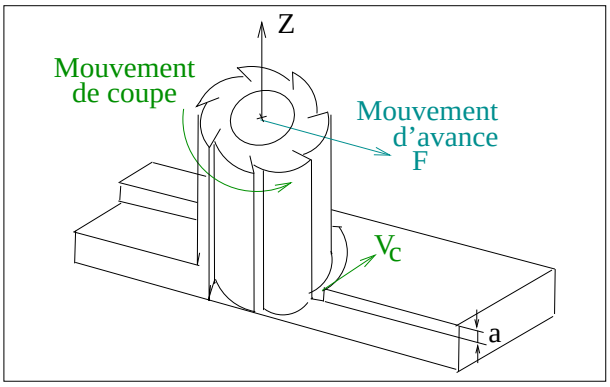

FIG. A.30 – Paramètres de coupe en fraisage.

positifs, délimités eux aussi par des *contours intérieurs*. Les profils extérieurs et les contours intérieurs sont construits par des segments de droite et/ou des arcs de cercles.

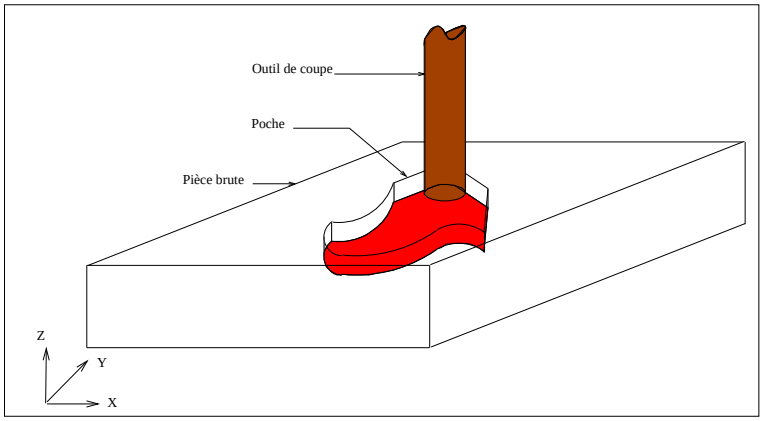

FIG. A.31 – Usinage de poches en $2D\frac{1}{2}$.

L'usinage d'une poche est effectué à profondeur de coupe constante. L'évidement de la poche consiste à enlever toute la matière contenue à l'intérieur du profil extérieur tout en laissant celles délimitées par les îlots positifs. Les poches peuvent être usinées par plusieurs passes d'usinage et avec différents outils.

L'usinage de nombreuses pièces mécaniques comporte l'évidement de poches en $2D\frac{1}{2}$ [59]. L'usinage de poche en $2D\frac{1}{2}$ est réalisé par des trajectoires linéaires et circulaires dans des plans parallèles au plan XY.

Afin de réduire le temps d'usinage, la longueur totale des trajectoires de l'outil de coupe doit être minimale pour réaliser une ébauche. Considérant ce fait, la génération de trajectoires est réalisée selon une certaine stratégie d'usinage. En pratique, il existe trois stratégies principales :

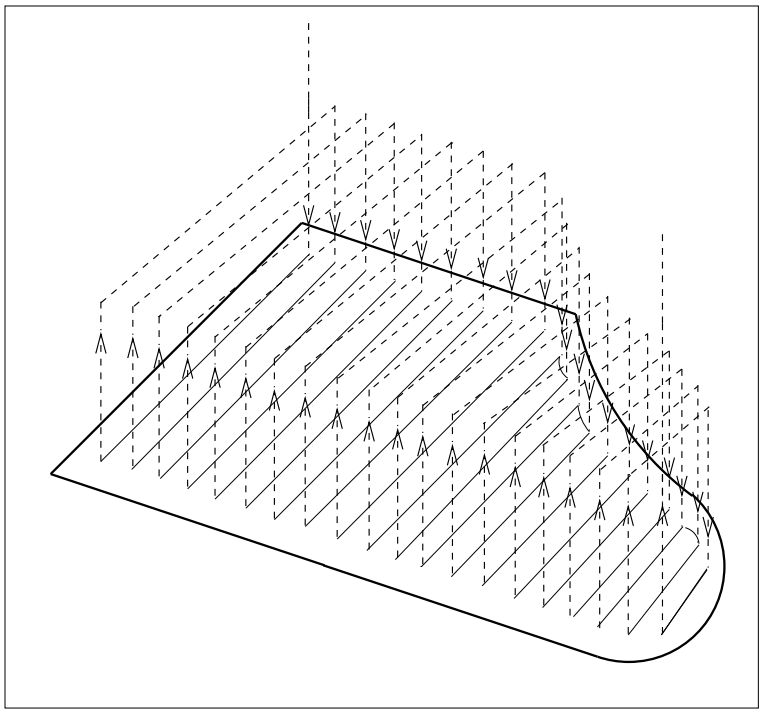

FIG. A.32 – Usinage unidirectionnel.

- **L'usinage unidirectionnel** : appelé encore usinage en aller simple (figure A.32, page 158), est réalisé par des trajectoires linéaires parallèles mais avec dégagement de l'outil de coupe pour chaque trajectoire de retour. Ce type d'usinage présente beaucoup d'opérations de dégagement, retour rapide et engagement, ce qui provoque un temps hors usinage important pour une production en série.
  La distance entre deux trajectoires de l'outil est appelée *distance entre passe*. Elle est choisie par l'opérateur et dépend du rayon de l'outil. En effet, elle est au maximum égale à $2 \times Rayon\ de\ l'Outil$, afin que toute la matière soit usinée entre deux grandes trajectoires.
- **L'usinage en zig-zag** : appelé encore usinage en aller-retour (figure A.33, page 159), est réalisé par des trajectoires linéaires parallèles. L'origine et l'ex-

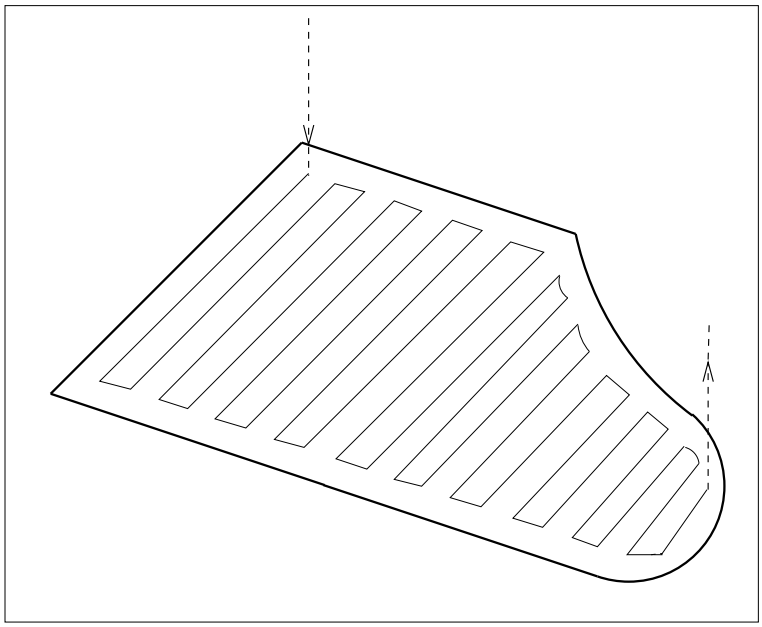

FIG. A.33 – Usinage en zig-zag.

trémité d'une trajectoire sont déterminées suivant la forme du profil de la poche.

– **L'usinage en contours parallèles** : les trajectoires de l'outil sont des contours décalés à une certaine distance par rapport au profil extérieur de la poche (figure A.34, page 160).

Selon la forme du profil extérieur de la poche et l'éventuelle présence d'îlots positifs, la poche peut être divisée en plusieurs zones. L'usinage de la poche se poursuit alors zone après zone. L'usinage de chaque zone débute par un perçage et se poursuit selon la stratégie choisie pour l'usinage de la poche. En usinage en contours parallèles, ces zones sont appelées zones monotones. Dans chaque zone, l'usinage se poursuit sans rétraction de l'outil de coupe.

A.2 Usinage par commande numérique

La CN reçoit les informations codées concernant le programme d'usinage de la pièce, les paramètres d'usinage (comme les dimensions de l'outil) et la modulation éventuelle des vitesses d'avance et des vitesses de coupe. Elle doit ensuite interpréter ces données et envoyer les signaux de commande à la machine outil tout en contrôlant en permanence les déplacements des divers organes mobiles de la machine outil

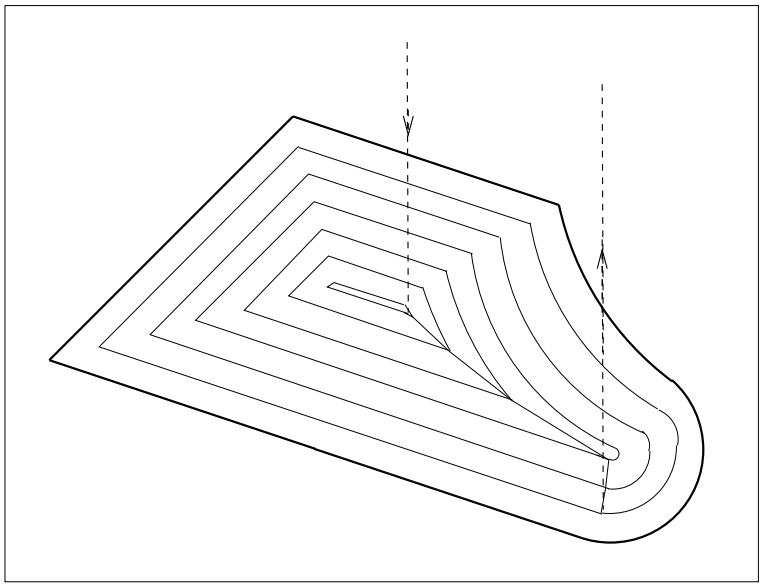

FIG. A.34 – Usinage en contours parallèles.

en vitesse et en position pour mener à bien l'usinage désiré.

Les déplacements que l'on est appelé à rencontrer sur une MOCN peuvent prendre des formes diverses. Nous distinguons deux types de trajectoires de l'outil :

- **Trajectoire fonctionnelle** :
  Ce type de trajectoire contribue à la réalisation de la surface à usiner. L'outil de coupe reste en contact avec la surface pendant son mouvement. Cette trajectoire se réalise en vitesse de travail (vitesse qui tient compte du matériau, de la profondeur de passe,...).
- **Trajectoire non-fonctionnelle** :
  Cette trajectoire n'intervient pas dans la réalisation de la surface à usiner. Elle permet le déplacement de l'outil pour atteindre cette surface. Ce type de trajectoire s'effectue généralement en vitesse rapide.

A.2.1 Le programme pièce, quelques notions actuelles

1. **Structure**

   Les instructions programmées doivent contenir toutes les données nécessaires à la commande et au séquencement des opérations à réaliser pour assurer

l'usinage de la pièce sur la machine.
Elles regroupent :
- **les données géométriques** qui indiquent la forme et la dimension de la pièce à usiner et permettent à la CN de calculer les positions successives de l'outil par rapport à la pièce pendant les différentes phases d'usinage.
- **les données technologiques** qui précisent, compte tenu des caractéristiques et des performances de la machine outil, les conditions de coupe optimales pour l'usinage.

Les instructions d'un programme sont écrites dans un langage codé appelé **langage machine** qui devraient se conformer à la norme internationales ISO 6983 (NF Z 68-035, NF Z 68-036 et NF Z 68-037). Toutefois, malgré les nombreux efforts de normalisation, il présente des spécificités selon le directeur de commande numérique dont est équipée la machine : la norme ne définit qu'un noyau sémantique commun minimal, c'est pourquoi on parle plus justement de **code G** (et **M**).

Un programme pièce définit une succession d'actions, ligne par ligne, chaque ligne constitue un **bloc d'information** et correspond à une étape particulière du processus d'usinage. Chaque bloc, ou séquence d'usinage, comporte des **mots** qui sont la combinaison de lettres d'identification appelées **adresses**. Les mots débutent par une lettre-adresse qui donne un sens physique aux données numériques qui suivent. Les lettre-adresses assurent sans ambiguïté l'identification de l'information et la séparation des mots. Les lettres-adresses usuelles sont :
- **N** : pour les mots *numéro de bloc :*
  Ces mots figurent obligatoirement en début de chaque bloc
- **G** : pour les mots *fonction préparatoire :*
  Ces fonctions définissent le déroulement de certaines fonctions de commande et préparent la CN à exécuter une action bien précise. Ce sont généralement des ordres de déplacement, de décalage, d'appels de cycles spécifiques d'usinage, ... Un bloc d'information peut contenir plusieurs fonctions préparatoires G si elles ne sont pas contradictoires.

  Ces fonctions définissent l'appel du programme résidant dans le directeur de commande en vue d'exécuter une action bien définie. Elles sont appelées par la lettre-adresse **G** suivie d'un numéro (0 à 99, selon la norme, en fait davantage). Par exemple, la fonction **G**01 (ou **G**1) demande l'exécution d'une interpolation linéaire, alors que l'interpolation circulaire est mise en oeuvre

par **G**2 ou **G**3 suivant le sens de parcours (le sens antitrigonométrique ou le sens trigonométrique).

- pour les mots de *dimensions ou d'ordre de déplacement*, composés d'une adresse accompagnée de sa valeur formatée :

  X,Y,Z : pour les coordonnées principales du point à atteindre.

  A,B,C : pour les coordonnées angulaires.

  U,V,W : pour les déplacements secondaires parallèles à **X,Y,Z**.

  I,J,K : pour les coordonnées du centre d'interpolation.

- pour les mots correspondant aux *fonctions diverses* :

  R : pour la programmation d'un cercle par son rayon en interpolation circulaire.

  S : pour la vitesse de rotation de la broche.

  F : pour la vitesse d'avance.

  T : pour désigner le numéro de l'outil de coupe.

- **M** : pour les mots *fonctions auxiliaires*
  Ces fonctions servent à définir des interruptions de programme et des actions gérées par l'automate. On peut citer parmi ces fonctions :
  - La fonction **M**05 qui assure l'arrêt de la broche.
  - La fonction **M**06 qui provoque le changement de l'outil de coupe en déclenchant une action physique.
  - La fonction **M**02 qui réinitialise le système et efface les registres. C'est la fin du programme.

La plupart des machines acceptent des blocs à format variable dans lequel ne figurent que seules les instructions nécessaires à leur exécution. Chaque constructeur spécifie dans le manuel de programmation la façon d'écrire les données numériques allouées aux différentes lettre-adresses et l'organisation du bloc d'informations [60]. Un exemple d'organisation d'un bloc retenu par le constructeur **NUM** est donné par la figure A.35, page 163.

S'agissant de l'usinage de poches, ce même constructeur propose les fonctions préparatoires **G45** et **G46**, en fait des macros dites *cycles de poches* permettant l'usinage :

- **G45** : de poches simples circulaires, oblongues, rectangulaires,...
- **G46** : d'une ou plusieurs poches de formes variées avec îlots.

On doit noter toutefois que :

- la stratégie d'usinage n'est pas précisée,
- des problèmes apparaissent à l'usinage ; ainsi, il peut arriver qu'un étranglement (zone de rétrécissement entre une poche et un îlot, entre deux îlots,...)

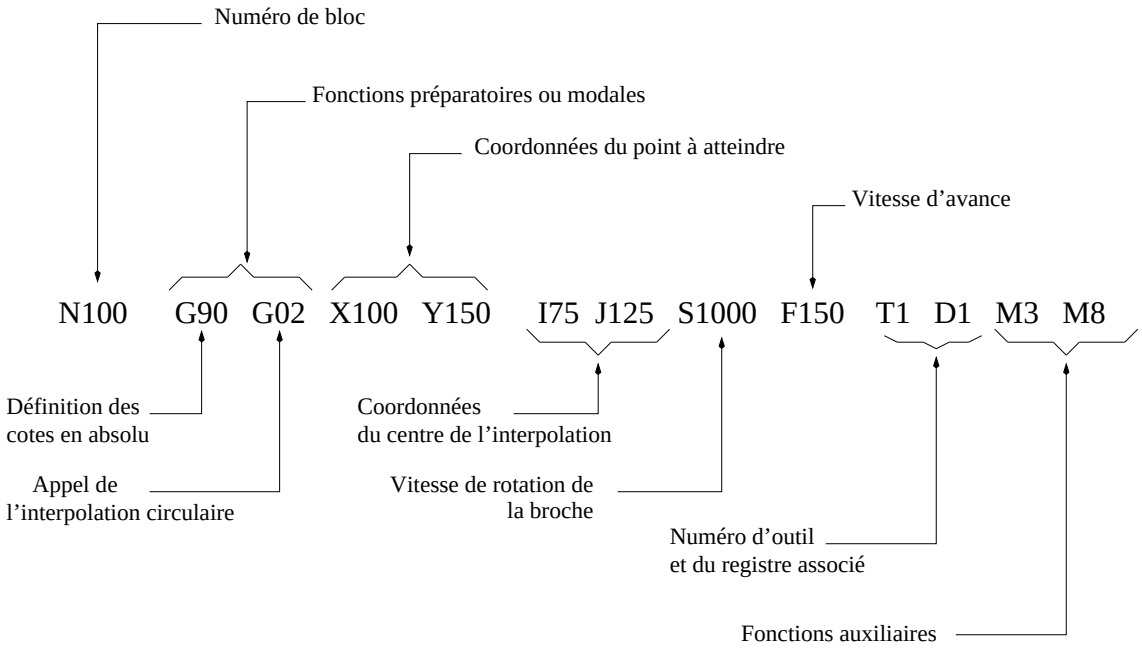

FIG. A.35 – Organisation d'un bloc.

ne soit pas usiné alors que sa dimension offre pourtant le passage du diamètre de la fraise.

2. Elaboration

Les pratiques rencontrées pour la réalisation d'un programme de commande numérique [60, 1] peuvent être regroupées essentiellement en deux catégories adaptées à la nature des pièces à réaliser (usinages géométriques simples - numériquement très importants-, ou usinages de surfaces complexes).

(a) **La programmation d'atelier (Workshop programming)** :
La programmation manuelle consistait à écrire, ligne par ligne, les étapes successives nécessaires à l'élaboration d'une pièce donnée. Après décomposition du cycle de travail, le programmeur calculait les coordonnées des points intermédiaires, définissait tous les déplacements pour chaque passe d'usinage et réalisait lui-même la codification des instructions en respectant le format spécifique prévu, ce qui demandait une bonne connaissance des techniques d'usinage et du langage G. A l'origine, l'analyse, le calcul

des trajectoires outils et la rédaction du programme s'effectuaient sur papier à partir du dossier de définition de la pièce reçu du bureau d'études. Ce programme pouvait être alors saisi directement au clavier ou édité sur un support en fonction de la machine-outil utilisée, pour enfin être chargé sur celle-ci.

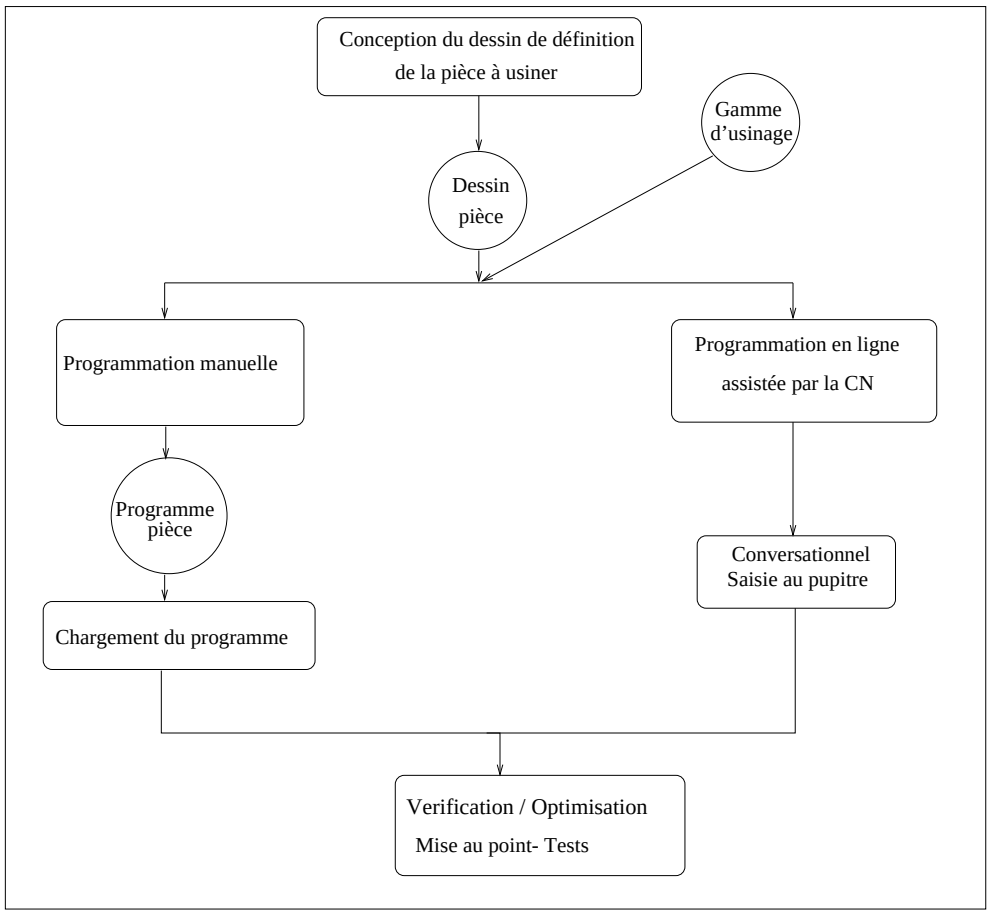

FIG. A.36 – Démarches de programmation d'atelier.

Des logiciels d'aide à la programmation sont désormais couramment intégrés dans les directeurs de commande. Ces logiciels proposent entre autres :

- **la programmation géométrique de profils** à partir d'éléments géométriques simples (segments de droites et arcs de cercles), en laissant la CN calculer les points de raccordements entre ces éléments.
- **la définition graphique de profils** qui guide l'opérateur en lui pro-

posant une visualisation instantanée et dynamique des profils en cours de réalisation. L'utilisateur ne se préoccupe plus de la codification ISO.
- **la programmation conversationnelle** qui permet à un opérateur de créer un programme pièce directement, sans avoir recours au langage machine codé en ISO. L'élaboration de la géométrie de la pièce et la génération des trajectoires font appel à des fonctions graphiques et des menus déroulants. Une interactivité se crée entre les données entrées par l'utilisateur et les réponses de la CN. Une fois la pièce dessinée, la CN convertit en langage I.S.O toutes les informations qui ont été programmées en mode conversationnel.

Une fois élaborés (figure A.36, page 164), les programmes en code G présentent l'avantage d'être relativement aisés à mettre en oeuvre et permettent une utilisation directe sur les machines dont ils peuvent exploiter au mieux les possibilités. Ils sont compréhensibles par l'opérateur, se prêtent bien à une mise en oeuvre pas à pas, à l'introduction de données liées au contexte opératoire et de ce fait un opérateur averti peut effectuer sur site des corrections diverses.

(b) **La programmation automatique ou assistée (FAO/CFAO)** :
L'utilisation de systèmes informatiques rend automatique la quasi totalité des tâches de réalisation du programme d'usinage. Lorsque la définition de l'usinage devient trop complexe, on fait appel à un langage de programmation spécialisé qui comporte généralement deux phases de traitement des programmes (figure A.37, page 167) :
- **le programme processeur** calcule les coordonnées de tous les points définissant la forme de la pièce, puis détermine les trajectoires outils en tenant compte de certaines données technologiques d'usinage (vitesse, avance, profondeur de passe en fonction des matières usinées et de l'outil utilisé, ...).
  Il crée ainsi un fichier dénommé CLFILE (Cutter Location File) contenant les coordonnées des positions successives de l'outil, appelées aussi CLDATA (Cutter Location Data). Ce fichier est indépendant de la Machine Outil et de la CN.
- **le programme post-processeur** traduit ce fichier en langage I.S.O (code G), en tenant compte des caractéristiques de la machine et de la CN.

Les systèmes de FAO (Fabrication Assistée par Ordinateur) suivent un

processus similaires. Ils assurent en plus la reprise automatique des données de définition de profils de contournage générés par des logiciels de CAO (figure A.37, page 167).

A.2.2 Eléments de comparaison entre la situation actuelle et celle à venir du ISO 14649

Ils sont représentés sur la figure A.38.

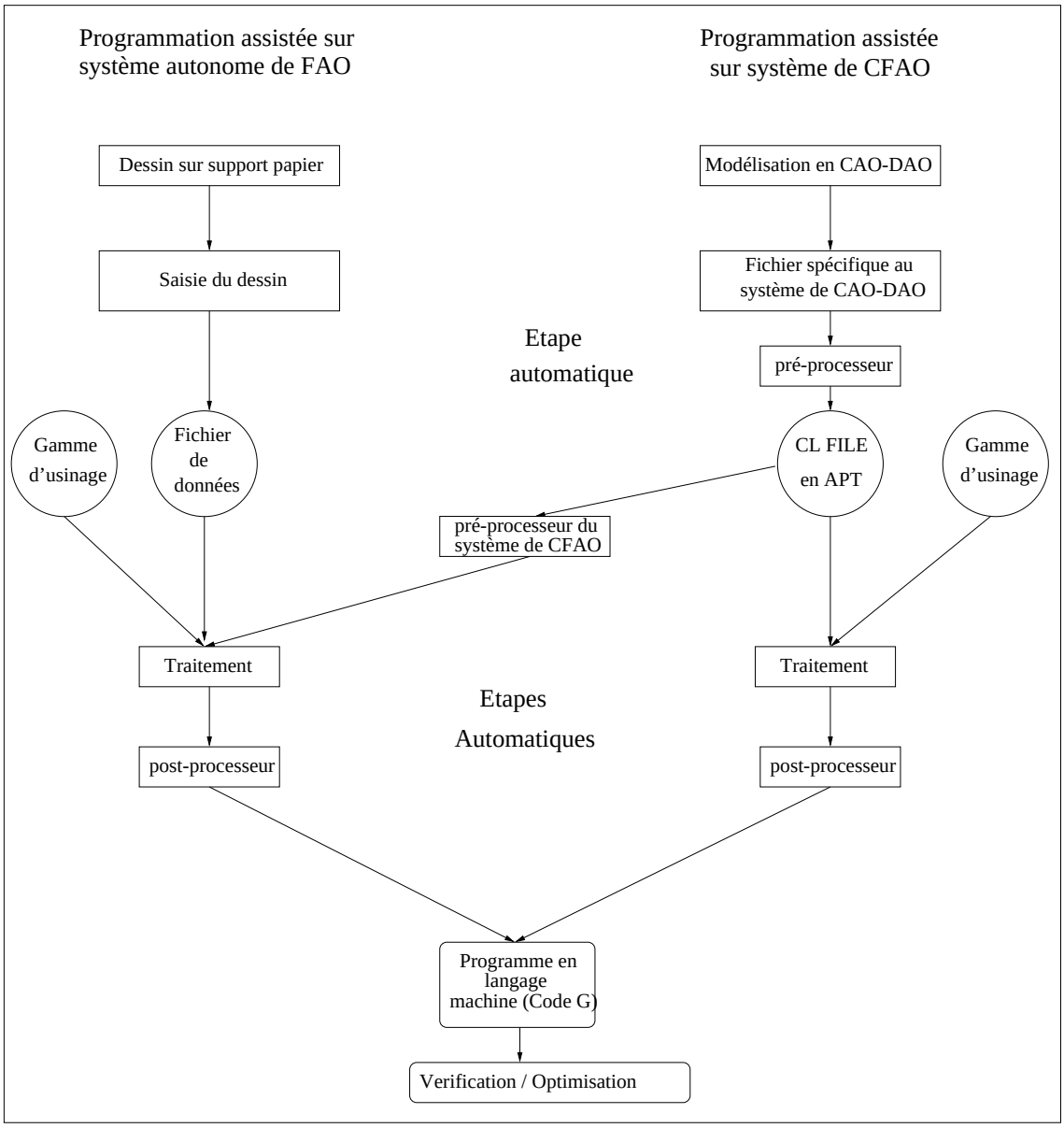

FIG. A.37 – Démarches de programmation assistée.

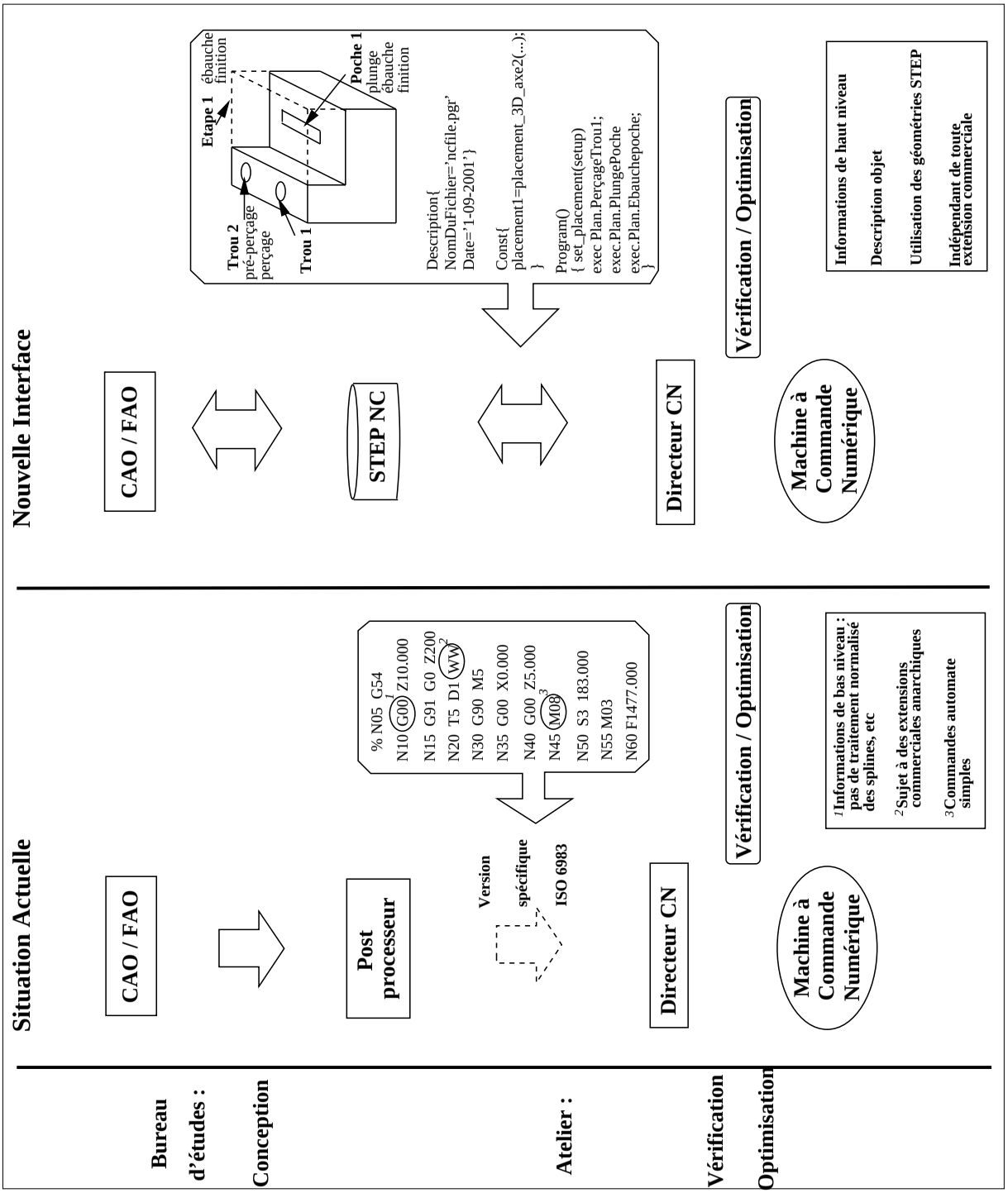

FIG. A.38 – Situation Actuelle et Nouvelle Interface pour les machines à commande numérique.

Bibliographie

[1] R. Berland, R. Couderc, G. Fannechère, J.-F. Guénal, J.-F. Poiraudeau, S. Roux, and D. Texier. Émulation des commandes numériques de machines-outils. Vers des M.O.C.N. virtuelles. *Revue d'automatique et de productique appliquées*, 6(3) :291–302, 1993.

[2] T. Kramer. Pocket Milling with Tool Engagement Detection. *Journal of Manufacturing Systems*, 11(2) :114–123, 1991.

[3] A. Hatna, C. Marty, and A.Belaidi. Evidement de poches complexes à fond plat par la technique des décalages successifs. *Revue internationale de CFAO et d'infographie*, 11(3) :261–288, 1996.

[4] M. Held, G. Lukacs, and L. Andor. Pocket Machining Based on Contour-Parallel Tool Paths Generated by Means of Proximity Maps. *Computer-Aided Design*, 26(3) :189–203, March 1994.

[5] Y. S. Lee and T. C. Chang. 2-Phase approach to global tool interference avoidance in 5-axis machining. *Computer-Aided Design*, 27(10) :715–729, October 1995.

[6] E. M. Arkin, M. Held, and C. L. Smith. Optimization Problems Related to Zigzag Pocket Machining. Technical report, New York, November 1995.

[7] M. Albert. STEP NC - The End of G-Codes ? *Modern Machine Shop*, July 2000.

[8] F. Proctor, S. Kamatsu, and F. Glantschig. Iso Draft International Standard 14 649-1. 2001.

[9] T. Beard. Programming For High Speed Machining. *Modern Machine Shop*, August 1997.

[10] A.-L. Defretin and G. Levaillant. Usinage à grande vitesse. *Techniques de l'Ingénieur, Traité de Génie Mécanique*, BT 2 :BM 7180 1–26, 1999.

[11] H. Conraux. Émulation des CNC : application à la programmation des MOCN. *Rapport de DEA, Université de Limoges*, 1988.

[12] T. Van Hook. Real–Time Shaded NC Milling Display. *Computer Graphics (Proc. SIGGRAPH'86)*, 20(4) :15–20, 1986.

[13] I. Blasquez and J-F. Poiraudeau. Improving the Extended Z-Buffer, Z-Trace, T-Trace and Animation. *Proceedings of the IASTED International Conference Computer Graphics and Imaging*, pages 81–87, 1999.

[14] I. Blasquez and J-F. Poiraudeau. The Interval Treap a Complete Data Structure for the Extended Z-Buffer. *Proceedings of Spring Conference on Computer Graphics and Its Applications*, pages 247–255, Mai 3-6, 2000.

[15] P. Couffin, J.-Y. Herbin, and M. Puzenat. *CAO en mécanique*. Armand Colin, Paris, 1989.

[16] J.D. Foley, A. Van Dam, S.K. Feiner, and J.F. Hugues. *Computer Graphics Principles and Practice*. Addison-Wesley Publishing Company, USA, 1992.

[17] J.-C. Léon. *Modélisation et construction de surfaces pour la CAO*. Hermès, Paris, 1991.

[18] A.A.G. Requicha. Representation for Rigid Solids : Theory, Methods and Systems. *Computing Surveys*, 12(4) :437–464, 1980.

[19] M. Mekhilef and B. Yannou. Conception intégrée assistée par ordinateur. *Techniques de l'Ingénieur, Traité de Mécanique*, BD 1 :BM 5006, 1998.

[20] B. Péroche, J. Argence, D. Ghazanfarpour, and D. Michelucci. *La synthèse d'images*. Hermès, 1990.

[21] U.A. Sungurtekin and H.B. Voeckler. Graphical simulation and automatic verification of NC programs. *Proc. IEEE International Conference on Robotics and Automation*, 1 :156–165, 1986.

[22] B. Chazelle. Application Challenges to Computational Geometry. Technical report, 521-96, Princeton University, 1996.

[23] Y. Kawashima, K. Itoh, T. Ishida, S. Nonaka, and K. Ejiri. A flexible quantitative method for nc machining verification using a space division based solid model. *The Visual Computer*, 7 :149–157, 1991.

[24] R.B. Jerard, S.Z. Hussaini, and R.L. Drysale. Approximate methods for simulation and verification of numerically controlled machining programs. *The Visual Computer*, 5 :329–348, 1989.

[25] E. Catmull. Computer Display of Curved Surfaces. *Proc. IEEE Conf. on Computer Graphics, Pattern Recognition, and Data Structures*, May 1975.

[26] K.C. Hui. Solid sweeping in image space – Application in NC simulation. *The Visual Computer*, 10 :306–316, 1994.

[27] Y. Huang and J.H. Oliver. Integrated Simulation, Error Assesment, and Tool Path Correction for Five-Axis NC Milling. *Journal of Manufacturing Systems*, 14(5) :331–344, 1995.

[28] D. F. Rogers. *Procedural Elements for Computer Graphics*. Mc Graw-Hill Publishing Company, USA, 1985.

[29] Y. Huang and J.H. Oliver. NC milling error assessment and toolpath correction. *Computer Graphics (Proc.SIGGRAPH'94)*, pages 287–294, 1994.

[30] T. Saito and T. Takahashi. NC machining with G-buffer method. *Computer Graphics (Proc. SIGGRAPH'91)*, 25(4) :207–216, 1991.

[31] C.-J. Chiou and Y.-S. Lee. A shape-generating for multi-axis machinig G-buffer models. *Computer-Aided Design*, 31 :761–776, 1999.

[32] G. Glaeser and E. Gröller. Efficient Volume-Generation During the Simulation of NC-Milling. Technical report, Institute of Computer Graphics, University of Technology, Vienna, 1997.

[33] J.P. Menon and D.M. Robinson. Advanced NC Verification via Massively Parallel Raycasting. *Manufacturing Review*, 6(2) :141–154, 1993.

[34] J.P. Menon and H.B. Voelcker. On the Completeness and Conversion of Ray Representations of Arbitrary Solids. Technical report, IBM RC 19935, T.J. Watson Research Center Yorktown Heights, 1995.

[35] M.S. Head, G. Kedem, and M.G. Prisant. Application of the Ray Representation and a Massively Parallel Special Purpose Computer to problems of Protein Structure and function : I. Methodology for Calculation of Molecular Contact Surface, Volume and Internal Free Space. Technical report, CS-1994-31, Departements of Chemistry and Computer Science,Duke University, Durham, North Carolina, 1994.

[36] N. Dubreuil and P. Lienhardt. Utilisation du produit cartésien en modélisation 4d pour l'animation. *5èmes Journées AFIG 97*, pages 91–100, 3–5 Décembre 1997, Rennes.

[37] A. Aho and J. Ullman. *Concepts fondamentaux de l'informatique*. Dunod, Paris, 1993.

[38] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique*. Dunod, Paris, 1994.

[39] W. Pugh. Skip Lists : A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33 :668–676, 1990.

[40] R. Tamassia. Data Structures. *ACM Comput. Surv.*, 28(1) :23–26, 1996.

[41] W. Pugh. A Skip List Cookbook. Technical Report CS-TR-2286, Dept. of Computer Science, University of Maryland, 1989.

[42] T. Papadakis , J.I. Munro, and P. Poblete. Average Search and Update Costs in Skip List. *BIT*, 32 :316–332, 1992.

[43] P. Kirschenhofer, C. Martinez, and H. Prodinger. Analysis of an optimized search algorithm for skip lists. *Theoritical Computer Science*, 144 :199–220, 1995.

[44] J.I. Munro, T. Papadakis, and R. Sedgewick. Deterministic Skip Lists. *ACM - SIAM Symposium on Discrete Algorithms (SODA)*, pages 367–375, 1992.

[45] Y. Livnat, H. Shen, and C.R. Johnson. A Near Isosurface Extraction Algorithm Using The Span Space. *IEEE Trans. on Visualization and Computer Graphics*, 2(1) :73–84, 1998.

[46] M. De Berg, M. Van Kreveld, M. Overmars, and O Schwartskopf . *Computational Geometry - Algorithms and Applications*. Springer-Verlag, 1997.

[47] J. Vuillemin. A Unifying Look at Data Structures. *Communications of the ACM*, 23 :229–239, 1980.

[48] M.A. Weiss. *Data Structure and Algorithm Analysis in C++*. Addison Wesley, 1999.

[49] R. Seigel and C.R. Aragon. Randomized Search Trees. *Algorithmica*, 16 :464–497, 1996.

[50] G.E. Blelloch and M. Reid-Miller. Fast Set Operation Using Treaps. *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 16–26, 1998.

[51] M. Divay. *Algorithmes et structures de données*. Dunod, Paris, 1999.

[52] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, 1996.

[53] P.G. Hoel. *Statistique mathématique*. Armand Colin, Paris, 1984.

[54] P.J. Van Otterloo. *A Contour-Oriented Approach to Shape Analysis*. Prentice-Hall International, UK, 1991.

[55] T.M. Olano and T.S. Yoo. Precision Normals (Beyond Phong). Technical report, 93-021, UNC Computer Science, University of North Carolina, Chapel Hill, 1993.

[56] D.A. Forsyth and J. Ponce. *Computer Vision : a Modern Approach*. Prentice Hall, 2001.

[57] C. Tricot. *Curves and fractal dimension.* Springer-Verlag, New York, 1995.

[58] B. Méry. *Machines à commande numérique.* Hermès, Paris, 1997.

[59] M. Held. *On the Computational Geometry of Pocket Machining.* Springer-Verlag, Berlin Heidelberg, 1991.

[60] C. Marty, C. Cassagnes, and P. Marin. *La pratique de la commande numérique des machines-outils.* Technique et Documentation-Lavoisier, Paris, 1993.