Amélioration des performances du z-buffer étendu à l'aide des skip lists

Isabelle BLASQUEZ, Jean-François POIRAUDEAU
LICN-IUT, Université de Limoges
Allée André Maurois, 87065 LIMOGES Cedex
poiraudeau@unilim.fr

Résumé

Dans cet article, nous proposons une amélioration du z-buffer étendu. Cette technique est bien adaptée pour représenter les objets dans le cas de la simulation d'usinage pratiquée dans les ateliers, mais elle ne garde pas en mémoire l'historique de la construction de la pièce finale : pour revenir en arrière, il faut rejouer la simulation depuis la première étape. Pour obtenir une simulation interactive, nous avons proposé une nouvelle structure de données appelé « trace ». La « trace en temps » permet d'afficher un moment précis de la simulation. Nous exposons ici la structure de données la mieux adaptée pour consulter rapidement cette trace. L'évaluation expérimentale montre que l'utilisation des « skip lists » est compatible avec les performances des micro-ordinateurs standard d'aujourd'hui, tout en améliorant la consultation de la trace, permettant ainsi de mettre en place une animation plus conviviale de la simulation.

Mots clés : z-buffer étendu, skip list, commande numérique, simulation d'usinage, modélisation géométrique.

1 Introduction

Nous nous intéressons dans ce papier à la simulation de l'exécution de programmes d'usinage telle qu'elle est pratiquée dans les ateliers sur des micro-ordinateurs standard[1]. Par le biais de la simulation, l'opérateur souhaite découvrir le parcours de

[1] Cette étude s'inscrit dans les recherches du laboratoire LICN (Laboratoire d'Informatique pour la Commande Numérique) sur les machines-outils virtuelles. Ces recherches ont mené au développement d'un logiciel d'émulation des directeurs de commande numérique des machines-outils appelés LI-CN (marque déposée de l'Université de Limoges). Ce logiciel est actuellement utilisé par plus de 150 entreprises françaises et par la plupart des centres techniques spécialisés dans la commande numérique en France.

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 préférable 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. Dans un premier temps, il est possible de 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.

Ainsi, pour visualiser rapidement un moment précis de la simulation, nous nous sommes intéressés à la seconde solution qui conserve les différentes phases de l'usinage dans deux structures de données. Nous avons introduit la notion de « traces » dans le z-buffer étendu [Blasquez 1999]. La « z-trace » est une « trace complète » construite suivant une direction de vue donnée. Elle permet de modéliser le brut à un moment donné et de continuer la simulation à partir d'un programme d'usinage modifié. La « trace en temps » ou « trace rapide » permet de se replacer très rapidement à un moment précis de la simulation d'usinage et de réaliser une animation. Le but de cet article est d'améliorer la structure de données de la trace en temps afin d'accélérer la reconstruction de la scène.

2 Simulation d'usinage par la technique du z-buffer étendu

L'usinage complet de la pièce met en œuvre une succession de trajectoires de l'outil. Chacune est caractérisée par le mode de déplacement du centre de l'outil (interpolation linéaire ou circulaire), ainsi que son positionnement dans l'espace (point de départ et point d'arrivée). 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 d'apparence continu de la 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). Ces positions seront d'autant plus nombreuses que la résolution choisie sera grande. Pour définir la partie du brut à enlever pendant la simulation, un calcul d'intersections de volumes entre le brut et chaque position de l'outil est réalisé grâce à des opérations booléennes de soustraction. 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 longs. Durant la simulation, une représentation graphique est affichée. Pour représenter le brut et l'outil, plusieurs méthodes ont été développées dans le domaine de la simulation, nous pouvons en distinguer deux types :

- Les méthodes basées sur les modèles 3D comme la méthode *Constructive Solid Geometry* (CSG) et la *Boundary Representation* (Brep) qui nécessitent un temps

d'exécution en $O(n^3)$. Elles demandent une grande capacité de calcul et ne permettent pas une visualisation en temps réel réaliste.

- Les méthodes basées sur la visualisation comme le *z-buffer étendu* [VanHook1986] et la *Ray Representation* (RayRep).

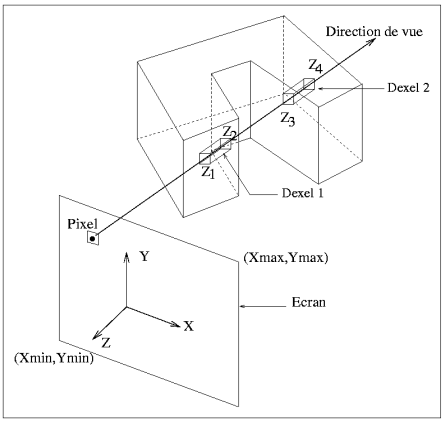

Figure 1. *z-buffer étendu*

Dans cet article nous adopterons la méthode du z-buffer étendu qui de par sa rapidité rend la simulation conviviale. Van Hook a été le premier à fixer un point de vue dans l' « espace image » et à utiliser un z-buffer étendu composé d'éléments de base en profondeur appelés dexels. Une liste de dexels est ensuite associée à chaque pixel de l'écran. Un dexel représente un parallélépipède, il correspond ainsi à une partie du brut derrière un pixel et suivant une direction donnée. Le z-buffer étendu permet donc d'afficher un objet suivant une orientation donnée. Le brut et l'outil sont représentés par des z-buffer étendus. Notre structure de travail est le z-buffer étendu du brut car en usinage, on considère que seul l'outil enlève de la matière au brut. Les opérations booléennes s'effectuent entre les deux z-buffers étendus [Hui 1994] : pour chaque position élémentaire, il faut comparer, dexel par dexel, le z-buffer étendu du brut au z-buffer étendu de l'outil. Comme on associe une liste de dexels à chaque pixel de l'écran, on peut interpréter géométriquement un dexel comme un segment sur un rayon donné orienté dans la direction de vue. La technique du z-buffer étendu nous permet de ramener les opérations booléennes de soustraction à des intersections d'intervalles dans l'espace euclidien E^1. Le z-buffer étendu s'exécute donc en $O(n)$. Un dexel contient des informations de types graphique et spatial sur l'objet à modéliser, notamment sur sa plus proche valeur en profondeur (NearZ), sa plus lointaine valeur en profondeur (FarZ), sa couleur et un pointeur sur le dexel suivant. La matière est alors bornée par le NearZ et le FarZ. Du point de vue algorithmique, la comparaison des z-buffers étendus se traduit par des opérations sur des listes ordonnées.

3 Trace en temps ou trace rapide

Nous avons récemment introduit la notion de trace en temps (t-trace) qui affiche une scène pour une étape donnée de la simulation. Seul l'affichage de la scène nous intéresse, nous ne voulons pas intervenir dans cette scène. Les données stockées dans la trace sont définies lors de l'actualisation du z-buffer étendu au cours de la simulation.

3.1 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. Lorsqu'on affiche un pixel à l'écran, il faut considérer le dernier Z du z-buffer étendu (ce sera donc un FarZ).
Règle 1 : Le pixel affiché à l'écran représente le dernier FarZ du z-buffer étendu.
La t-trace évolue lorsque la valeur du dernier FarZ est modifiée. Comme pour le z-buffer étendu, dans la t-trace, on associe 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 du dernier FarZ, 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.

3.2 Mise en place de la trace en temps

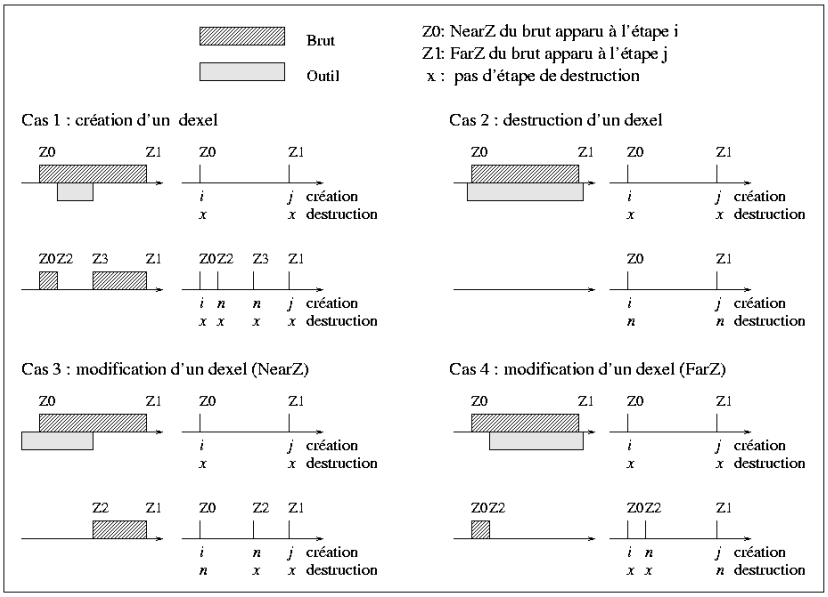

Figure 2. *Opérations booléennes entre deux dexels à l'étape n*

3.2.1 Initialisation et mise à jour

L'initialisation crée la t-trace qui décrit le brut à l'étape 0.
La mise à jour de la t-trace n'est possible qu'en analysant les différentes opérations booléennes sur un intervalle entre deux z-buffers étendus. Pour une étape *n* de la simulation, on compare un dexel du brut à un dexel de l'outil (figure 2). 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*. Nous avons montré précédemment [Blasquez 99] que la mise à jour de la t-trace est uniquement active si le FarZ du dernier dexel est modifié (cas 4) ou si le dernier dexel est supprimé.

3.2.2 Reconstruction de la scène à l'étape n

A partir de la t-trace, nous voulons retrouver la couleur de chaque pixel de l'ïmage telle qu'elle était à l'étape n de la simulation. Il suffit de parcourir la t-trace jusqu'à une étape *s* qui vérifie l'inégalité suivante: **étape *s*** ≥ **étape *n***.
Si étape *s* = étape *n* alors nous avons retrouvé la valeur de Z et la couleur associée au pixel de l'étape *n*. Sinon étape *s* > étape *n*, la valeur de Z et la couleur associée au pixel de l'étape *n* sont celles de l'élément en Z qui précède l'élément en Z de l'étape *s*, puisque la prochaine modification aura lieu après l'étape *n*.

4 Étude des structures de données possibles pour la trace en temps.

Reconstruire une scène *n* consiste à retrouver pour chaque pixel de l'écran la valeur du dernier FarZ et sa couleur telles qu'elles étaient à l'étape *n*. Comme la t-trace mémorise ces données dans un élément en temps, reconstruire une scène avec la t-trace revient à rechercher un élément le plus rapidement possible dans un ensemble de données. On se ramène alors à un « dictionnaire » qui regroupent les opérations d'insertion, de suppression et de recherche d'un élément dans un ensemble. Les structures de données classiques pour implémenter un dictionnaire sont : les listes et les arbres de recherche. Plusieurs facteurs influencent le choix d'une structure pour un algorithme précis : le temps d'exécution, l'occupation mémoire et la simplicité de programmation. Pour les listes, l'implémentation la plus couramment utilisée est la liste chaînée. Dans le pire des cas, la recherche d'un élément dans une liste chaînée se fait en O(*n*). Ce temps d'exécution n'est pas aussi bon que pour d'autres structures, mais cela est compensé par la simplicité d'implémentation. 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 nœuds de l'arbre. Pour accélérer ces temps d'exécution, on utilise des arbres binaires de recherche équilibrés. Un arbre binaire de recherche de *n* nœuds est dit équilibré si sa hauteur est log(*n*), ce qui réduit le temps d'exécution à *O*(log(*n*)) dans le pire des cas.

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. Les *skips lists* sont une généralisation relativement récente (1990) des listes chaînées simples : elles atteignent les performances des arbres binaires de recherche en conservant la structure des listes chaînées.

5 Skip List

Une *skip list* (SL) est une structure de données probabiliste simple et efficace introduite par Pugh [Pugh 1990] . Il est facile d'effectuer des opérations d'insertion et de suppression sur des listes chaînées, mais lorsqu'il s'agit de rechercher un élément, dans le pire des cas, il faut traverser la liste jusqu'à cet élément. Pugh appelle *skip lists* des listes chaînées qui permettent de « sauter » au-dessus des éléments intermédiaires pour atteindre rapidement l'élément cherché.

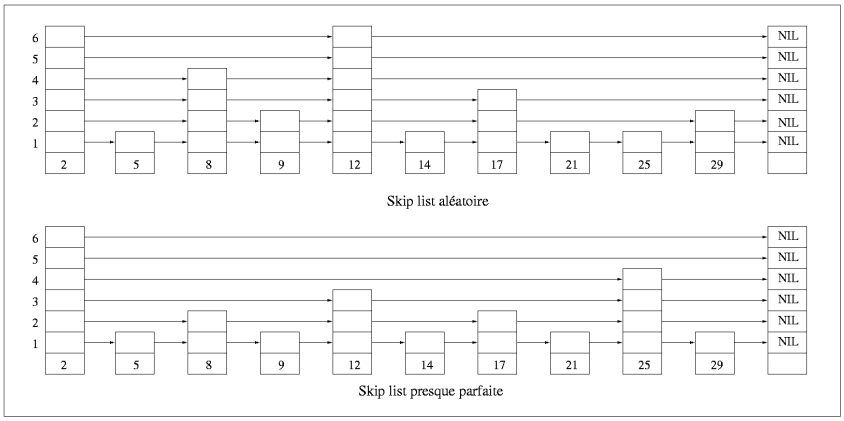

Figure 3. *Skip list*

Chaque élément de la SL est représenté par un nœud (appelé aussi tour) contenant un tableau de plusieurs pointeurs sur les nœuds suivants. La hauteur ou le niveau d'un nœud correspond au nombre de pointeurs du nœud . A un nœud de niveau *k*, on associe un tableau de *k* pointeurs. Dans une SL, le $i^{ème}$ pointeur d'un nœud de niveau *k* ($i \leq k$) avec une valeur clé *v* pointe sur le prochain nœud possédant au moins *i* niveaux dont la valeur clé est supérieure ou égale à *v* (figure 3). Les algorithmes d'insertion d'élément dans une SL 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 $O(\log n)$ [Kirschenhofer 1995] et dans le pire des cas on atteint $O(n)$[2]. Nous avons décidé d'utiliser les SL pour profiter de la rapidité de ce temps de re-

[2] Pour un dictionnaire de 250 éléments, il y a une chance sur un million pour que le temps de recherche soit trois fois plus rapide que le temps moyen attendu [Pugh 1990].

cherche. Dans les SL probabilistes, la forme de la liste dépend d'un générateur de nombres aléatoires, tandis que pour les SL déterministes [Munro 1992], la forme de la liste est fixée.

Dans notre application, la liste chaînée simple obtenue à la fin de la simulation sera transformée en SL qui ne servira que pour la recherche d'un élément, c'est une structure de données statique. Nous avons choisi de répartir les sauts uniformément le long de la SL. Nous obtenons de manière déterministe une SL presque parfaite où un nœud de niveau *i* pointe sur le $2^{\text{ième}}$ nœud suivant de niveau maximal supérieur ou égal à *i* (figure 3). Nous avons limité la hauteur maximale des nœuds à $h = 6$. Au maximum, le nombre de nœuds sautés sera donc de $2^{h-1} = 2^5$ soit 32 nœuds. Les SL obtenues au cours de la simulation sont en fait des SL *presque* parfaites car elles contiennent un nombre quelconque de nœuds, pas obligatoirement égal à une puissance de deux. Dans une telle SL de *n* éléments, nous obtiendrons ($\lfloor x \rfloor$ correspond à la partie entière de x par défaut) : $l_6 = \lfloor n/2^5 \rfloor$, $l_5 = \lfloor n/2^4 \rfloor - l_6$, $l_4 = \lfloor n/2^3 \rfloor - l_6 - l_5$, $l_3 = \lfloor n/2^2 \rfloor - l_6 - l_5 - l_4$, $l_2 = \lfloor n/2 \rfloor - l_6 - l_5 - l_4 - l_3$, $l_1 = \lfloor n/2 \rfloor$ respectivement pour les nœuds de niveau 6 à 1.

Pour la recherche d'un élément de valeur v_f dans la SL, nous utilisons l'algorithme proposé par Pugh mais optimisé. En observant la structure d'une SL presque parfaite, nous pouvons en déduire la règle 2 : dans une SL presque parfaite, entre deux nœuds de niveau *i*, il y aura au plus un nœud de niveau *i*-1.

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 sans jamais dépasser le nœud qui contient la valeur souhaitée. Pour le parcours de la SL sur le niveau maximum, il faut « sauter » le nombre de nœuds pour se positionner correctement au plus près du nœud contenant la valeur v_f cherchée. Pour les parcours de la SL sur les niveaux *i* inférieurs au niveau maximum, il suffit juste de tester l'étape de création du nœud suivant de niveau *i-1*. Si cette étape de création est inférieure à l'étape de création cherchée, ce nœud devient le nouveau nœud courant.

ReconstructionScèneOptimisée (ÉtapeCréationCherchée)

```
x := TourTêteDeListe
tant que (x.suivant[NiveauMax].EtapeCréation < ÉtapeCréationCherchée) faire
    x := x.suivant[NiveauMax]
fin tant que
pour  i := (NiveauMax-1) .. 2 faire
    si (x.suivant[i]. ÉtapeCréation < ÉtapeCréationCherchée) faire
        alors x := x.suivant[i]
fin pour
si  (x.suivant[1]. ÉtapeCréation = ÉtapeCréationCherchée)
   alors retourne x.suivant[1].CouleurPixel
   sinon retourne x.CouleurPixel
```

Nous pouvons alors examiner dans le pire des cas le temps de recherche d'un nœud dans la SL. Supposons que ce nœud soit le $p^{ième}$ d'une SL de hauteur maximale 6 et considérons que la tour de tête occupe la position 0. Nous pouvons décomposer p en $p = a*2^5 + b*2^4 + c*2^3 + d*2^2 + e*2 + f$. D'après l'analyse précédente, dans le pire des cas et pour chaque niveau inférieur au niveau 6, on aura un seul nœud à explorer. Si a=0 alors, il faudra quand même explorer un nœud de niveau maximum 6, sinon il faudra explorer (a+1) nœuds de niveau maximum dans le pire des cas, soit : a+6 nœuds à parcourir. Pour une SL presque parfaite de niveau h, dans le pire des cas, la recherche d'un nœud à la position p telle que : $p = a_{h-1}*2^{h-1} + a_{h-2}*2^{h-2} + \ldots + a_0$, est obtenue en parcourant au plus : $(a_{h-1} + h)$ nœuds.

6 Évaluation expérimentale

Prenons l'exemple de la simulation de l'usinage d'une petite pièce de moulage. L'usinage réel de cette pièce sur une fraiseuse prendra environ 1 h 30. Elle nécessite 17,108 pas de trajectoires de 0,5 mm pour la plupart avec un brut de dimensions 107x72x33 mm et un outil de 3 mm. Un PC de 300 Mhz, de 512 Ko de mémoire cache de niveau 2 et de 128 Mo de RAM a été utilisé. Le logiciel de simulation a été développé en C et utilise des accès directs à une carte graphique standard. Nous avons travaillé sur une image de 640 colonnes et 480 lignes (soit 307 200 pixels)[3]. A l'initialisation, ces structures de données comptent 307 200 dexels, éléments en temps. Pour une image de taille donnée, le nombre d'éléments varie suivant plusieurs paramètres : la taille et la forme du brut, le nombre de trajectoires, la stratégie d'usinage utilisée et l'angle de vue. Un dexel et un élément en temps d'une liste chaînée simple occupe 12 octets. Un pointeur supplémentaire occupe 4 octets de plus.

Pour étudier les performances de la structure de données des SL dans le cas de la t-trace, des SL presque parfaites de différentes hauteurs maximales (de 1 à 6) ont été mises en place. La SL parfaite de hauteur maximale h=1 correspond à une simple liste chaînée. La SL parfaite de hauteur maximale h=6 saute au plus $2^{h-1}=2^5$ soit 32 nœuds de la SL. La création du z-buffer nécessite 99,42 s. La création du z-buffer et la mise en place de la t-trace sous forme de liste chaînée simple nécessite 101,23 s. La création et la mise en place de la t-trace s'effectue en 1,81 s.

Pour des raisons d'efficacité, nous transformerons la liste chaînée simple en SL presque parfaite en allouant la mémoire de manière « statique ». Pour une SL de niveau h, tous les nœuds occuperont : $(8 + 4*h)$ octets, soit au total $(2+h)/3$ fois plus de mémoire que pour la liste chaînée simple. Le coût de mémoire est important, mais non prohibitif et la construction des SL est rapide. Le tableau 1 donne le temps nécessaire pour transformer la liste chaînée en SL et l'occupation mémoire des SL . Nous appelons « Ttrace L_h », une t-trace mise sous la forme d'une SL presque parfaite de hauteur maximale h.

[3] pour la vue considérée, 1 pixel représente 0,26 mm.

Ttrace L_1	Ttrace L_2	Ttrace L_3	Ttrace L_4	Ttrace L_5	Ttrace L_6
--	0,38 (s)	0,49 (s)	0,61 (s)	0,72 (s)	0,82 (s)
24,88 (Mo)	33,18 (Mo)	41,47 (Mo)	49,77 (Mo)	58,06 (Mo)	66,36 (Mo)

Tableau 1. *Construction de la SL (secondes) et occupation mémoire (Mo).*

Dans le meilleur des cas, la transformation de la t-trace de liste chaînée en SL de niveau 2 s'effectue en 0,38 secondes, soit 21% du temps nécessaire pour créer cette t-trace. Dans le pire des cas, la transformation de la t-trace de liste chaînée en SL de niveau 6 s'effectue en 0,82 secondes, soit 45,3% du temps nécessaire pour créer cette t-trace.

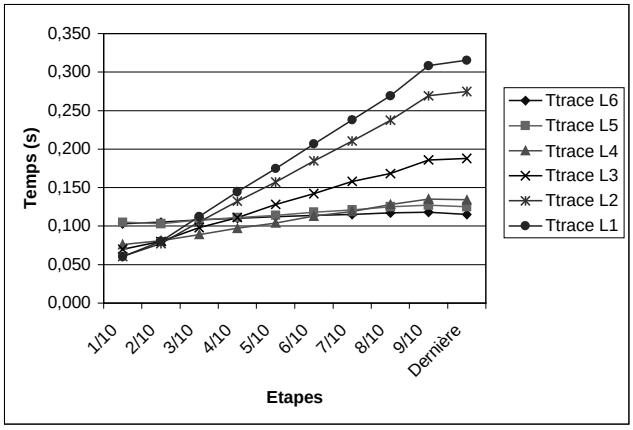

Figure 4. *Reconstruction d'une scène*

Puis, nous avons relevé les temps de reconstruction des scènes sans affichage. Vu le grand nombre de positions élémentaires de la simulation (44 084), nous nous sommes intéressés à des scènes de la simulation correspondant à des positions élémentaires arbitraires. Lors de la construction du z-buffer étendu, la scène dans laquelle la pièce a déjà subi un dixième de la simulation est la scène obtenue à la position 4 408. Pour simplifier, nous appelons cette étape de la simulation : « étape 1/10 ». Ces temps de reconstruction ont été relevés pour différentes étapes de la scène. Il s'agit d'un temps moyen calculé sur 100 simulations consécutives. La figure 4 regroupe ces résultats. Pour les étapes de début de simulation, les SL les plus efficaces sont les SL presque parfaites de hauteur maximale les plus petites. La liste chaînée simple ou la SL de niveau 2 sont alors les structures de données les plus rapides pour reconstruire une scène. Pour les étapes de fin de simulation, les SL les plus efficaces sont les SL presque parfaites de hauteur maximale les plus grandes. Les SL de niveau 5 ou 6 sont alors les structures de données les plus rapides. Par ailleurs, le temps de reconstruction d'une scène quelconque de la simulation est constant pour les SL de hauteur maximale élevée, ce qui permet d'obtenir un débit d'images régulier. La SL la mieux adaptée à ce problème est la SL de niveau 4, qui sautent au plus 8 nœuds. Cette SL occupe 2 fois plus de mémoire qu'une simple liste chaînée, l'écart type entre les temps de reconstruction des scènes est de 0,021 s. La

SL de niveau 4 met 27% plus de temps que la liste simple chaînée pour reconstruire la scène de l'étape 1/10, mais 36% fois moins de temps que la SL de niveau 6 qui correspond ici au pire des cas. Pour reconstruire une étape de fin de simulation, la SL de niveau 4 met 1,16 fois plus de temps que la SL de niveau 6, mais 2,35 fois moins de temps que la liste chaînée simple. La SL de niveau 4 paraît donc être un bon compromis et permet d'atteindre rapidement les scènes de début de simulation, de fin de simulation tout en occupant la mémoire de manière raisonnable.

7 Conclusion

Pour afficher une scène quelconque d'une simulation d'usinage utilisant le z-buffer étendu, nous avons proposé précédemment une trace en temps qui garde en mémoire l'historique de cette simulation. Ici, nous proposons d'utiliser différentes structures de données afin d'optimiser les performances de cette trace. Les *skip lists* sont des listes chaînées améliorées qui permettent de sauter au-dessus d'un ou plusieurs éléments de la liste accélérant ainsi la recherche. Grâce aux *skip lists*, 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. Cette méthode reste toujours compatible avec les cartes graphiques, la capacité mémoire et la vitesse des processeurs utilisés dans les micro-ordinateurs utilisés aujourd'hui dans les ateliers.

8 Bibliographie

[Blasquez 1999] Blasquez I., Poiraudeau J.-F., Improving the Extended Z-Buffer, Z-Trace, T-Trace and Animation , *Proceeding of the IASTED International Conference Computer Graphics and Imagings*, Palm Springs CA, p. 81-87, 1999.

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

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

[Munro 1992] Munro J.I., Papadakis T., Sedgewick R., Deterministic Skip Lists, *SODA*, 1992.

[Pugh 1990] Pugh W., Skip Lists : A Probabilistic Alternative to Balanced Trees, *Communications of the ACM*, 33 :663-676,1990.

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