The Interval Treap
A complete data structure for the extended z-buffer

I. Blasquez and J.-F. Poiraudeau
LICN–IUT,Université de Limoges
E-mail : poiraudeau@unilim.fr

Abstract

In this paper, we propose an improvement of the extended z-buffer. This object representation is well adapted to the NC machining simulation as practiced in the workshops. The data structure most often used to implement the extended z-buffer is the linked list, but it does not keep in memory the history of the construction of the workpiece. Thus, to go back in the visualization process, it is necessary to replay the simulation from the first stage of the machining. Also, to obtain an interactive simulation, we propose to implement the extended z-buffer as an "Interval Treap" (Treap: tree-heap). This new data structure, based on the interval binary search trees, contains all the information necessary to display and to reconstruct again the extended z-buffer at any given moment of the simulation. The experimental and theoretical evaluation shows how the use of Interval Treaps is compatible with the memory capacity and the processing speed of standard PC hardware, by offering new prospects to the extended z-buffer like the setting up of an animation.

Keywords: extended z-buffer, binary search tree, NC milling simulation, geometric modelling, performance study.

1 Introduction

In this paper, we aim to improve the data structure of the extended z-buffer. The latter is used in NC milling simulation practiced in the workshops with the standard PC graphics hardware. With the simulation, the machine operator tries to find out the errors which may appear during a real machining. He needs to know where, when and how these errors appear so as to measure their impact and thus find adequate solutions. The best way of achieving this is to perform an interactive simulation where the operator can visualize not only the finished part but also the different phases of the machining. In order to follow the actual material removal and to visualize a specific moment of the machining, two possibilities are available. It is first possible to execute again the simulation from the beginning to the chosen moment, however, this is rather time consuming. The second solution consists "in gluing" together the virtual part and virtual chips obtained and memorized during the simulation. To visualize faster a specific moment of the machining, we choose to focus on the second solution which allows the different machining phases to be stored. Several methods have already been developed in the field of machining simulation. Some methods are based on 3 D models like the CSG method and the B-Rep which both require at least an $O(n^3)$ execution time [1]. However, they are time-consuming and also unusable when simulating the machining of many thousand tool positions. Other methods based on image space such as the extended z-buffer [2] and the Ray-Representation [3] are well adapted to this application. In this paper, we focus on the extended z-buffer technique which takes only an expected time in $O(n)$ and gives a user-friendly animation [1].

In a previous paper, we already introduced two data structures, the z-trace and the t-trace, to come back to any specific moment of the simulation [4]. We introduce now the notion of "Interval Treap" in the extended z-buffer. This new data structure memorizes all the intervals obtained during the whole simulation. The "Interval Treap" allows not only to go back quickly to a specific moment of the simulation but also to model the part, the simulation can then be continued from a modified part. The originality of our work consists in using this new single data structure to implement the extended z-buffer in the field of computer graphics, and also in studying the performances of such a data structure not only in relation to the characteristics of the standard PC hardware, but also from an analytical point of view.

2 Different possible data structures for the extended z-buffer

Van Hook [2] was the first scientist to fix a view point in the image space and to define an extended z-buffer which contains depth elements, called dexels. A sorted list of dexels is then associated with each pixel. A dexel represents a rectangular solid, and corresponds to a part of the part behind a pixel for a fixed view point. With the extended z-buffer an object can be displayed in a specified direction [5]. The part and the tool are represented by extended z-buffers. The extended z-buffer structure of the part, and not that

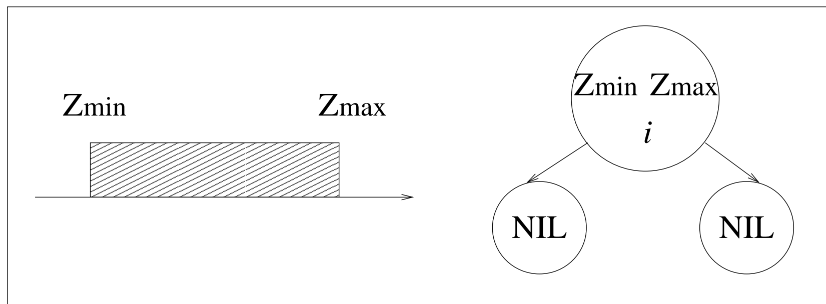

Figure 1: Correspondence between the interval created at stage i and the node of the Interval Treap

of the tool, will be modified as the tool merely removes material from the part during machining. The boolean operations require two extended z-buffers [6]: at each elementary position the extended z-buffer of the part and the extended z-buffer of the tool have to be compared dexel by dexel. As a list of dexels is associated with each pixel, a dexel can be interpreted geometrically as a segment of a given ray in the specified viewing direction. With the extended z-buffer technique, boolean subtraction operations can be simply performed on one-dimensional line segment intersections. A dexel contains graphic, spatial and modeling information about its near depth value (NearZ), its far depth value (FarZ), its color and a pointer to the following dexel. The material is bounded by the NearZ and the FarZ. Algorithmically, the comparison between the different extended z-buffers is performed by operations on sorted lists. The linked lists seem to be the easiest data structure available to implement the extended z-buffer, and the single data structure used usually in the literature. Indeed, the required dexel is quickly found by moving from one element to another. However, this implementation has some drawbacks such as the search time of an element in a linked list of n elements which is in the worst case in $O(n)$: to reach a required element, it is thus necessary to go through all the elements preceding it in the list one by one.

Moreover, the extended z-buffer proposed by Van Hook does not keep in memory the history of the simulation. Indeed, when a dexel is deleted from the extended z-buffer, it is also deleted from the linked list. This deletion is definitive and the information about this dexel will not be henceforth present any more in memory. In the same way, when a dexel is modified, its NearZ or FarZ is modified, but the former value is not directly accessible any more. It might also be difficult to go back to a specific moment of the simulation, because some data must be recomputed, and this will be time consuming.

In a recent paper, we defined the z-trace and the t-trace as an extension of the extended z-buffer. The z-trace keeps in memory every Z (NearZ and FarZ) and their parameters. It allows to recreate any given scene i of the simulation by reconstructing the extended z-buffer as it was at this scene i. As the displayed pixel is the last FarZ of the extended z-buffer, the t-trace memorizes only the last FarZ and its parameters. It allows to display a scene of simulation by finding the color of the pixels. The z-trace and the t-trace are independent. They are built during simulation, and thus memorize the evolution of the extended z-buffer step by step. The linked list is the data structure used for the z-trace and for the t-trace. The originality of this paper consists in using a new single data structure which stores all the data calculated during simulation, and in speeding up the processing of the simulation compared with an implementation by linked list .

The binary search tree (BST) data structure allows a fast access to a specific element among a great number of elements memorized during simulation: indeed, the time necessary to reach an element x of the tree is proportional to the depth of x in this tree [7]. In the worst case, the search time of an element in a balanced tree of n elements is $O(\log n)$. To keep in memory all the modifications of the extended z-buffer, it is necessary to store two types of data: first, the space data corresponding to the Z-values (NearZ and FarZ), then the time data corresponding to the creation stage of the Z-values in the data structure. As the extended z-buffer can be simply modeled as a segment [NearZ,FarZ] in one-dimensional space, we first thought of using an interval tree data structure. A possible representation is the "span space" proposed by Livnat, Shen and Johnson, who use a kd-tree and replace intervals by points [8]. An interval $I = [a_i, b_i]$ with $a_i \leq b_i$ is represented by a point in a two-dimension space with a_i as X-coordinate and b_i as Y-coordinate. By looking for the points such as $x \leq q$ and $y \geq q$, all the intervals which contain the q value are found. With this method, however, the notion of time is forgotten, especially the moment the dexel appears in the extended z-buffer cannot be memorized.

We would like to use a data structure which memorizes two types of data: in space and in time. In the Cartesian trees introduced by Vuillemin, every node has two keys (x,y) [9]. The value of the x-key satisfies the property of the binary search trees: the value of the x-key of a parent node is bigger than the value of the x-key of his left child and smaller than the value of the x-key of his right child. The value of the y-

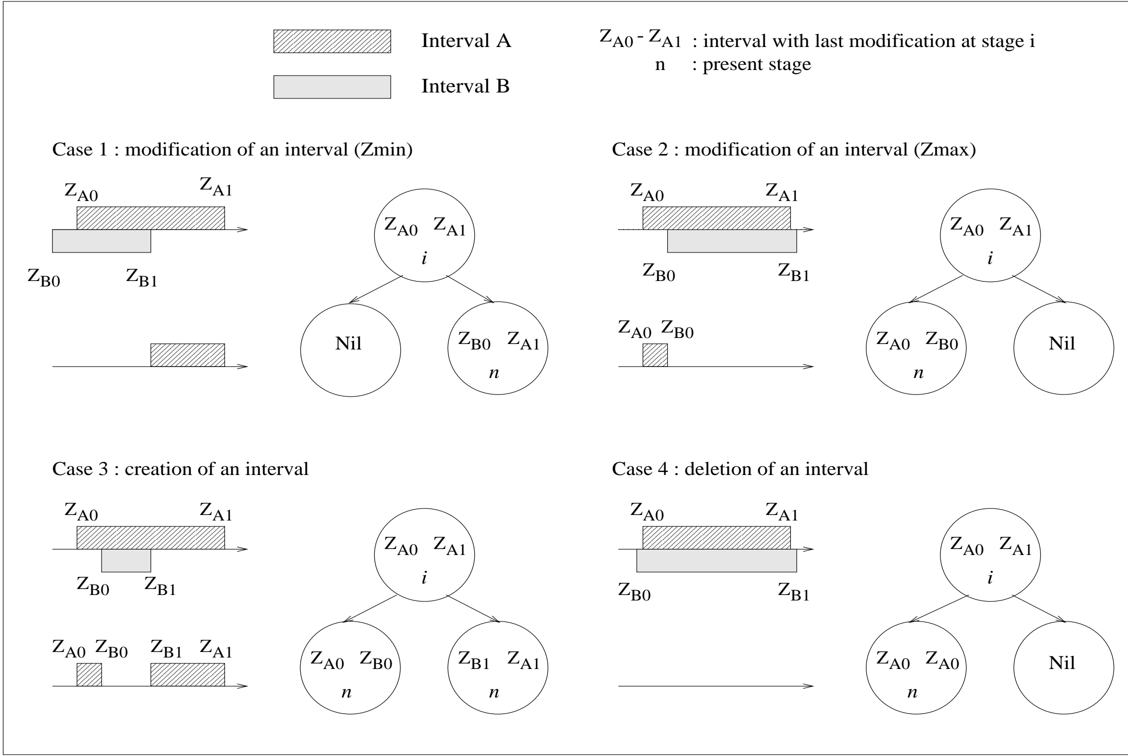

Figure 2: Boolean difference operation between two intervals: influence on the Interval Treap

key satisfies the property of "heap-order": the value of the y-key of a parent node is always smaller than the value of the y-key of his children nodes [7]. These binary search trees are unbalanced. On average, the execution time of the basic operations of insertion and search for an element in a Cartesian tree of n elements is $O(\log(n))$, and in the worst case, if the tree is completely unbalanced, it can achieve $O(n)$. To obtain balanced Cartesian trees, it is possible to replace y-key value by a random value which allows to balance the tree during its setting-up. Treaps (tree and heap) are obtained, they are balanced binary search trees constructed randomly [10, 11]. In our application, we do not have random values, but we memorize an interval [NearZ, FarZ] by storing two Z-values key and one value key indexed in time which memorize the moment of appearance of a new interval in the data structure. We thus create a new data structure called "Interval Treap" which contains an interval values and a time-value and allows to memorize all the modifications undergone by the z-buffer during the simulation.

3 Interval Treap: a new data structure

We define an *Interval Treap* (IT) as a interval binary search tree indexed in time but unbalanced. It keeps in memory information about new intervals created from successive operations on an initial interval: Interval Treap is thus composed of "superimposed intervals". The basic element of Interval Treap is a node which represents an interval that appeared at stage i. It contains the following data: two Z-values, and one time-value. A Z-value called Z_{min} limits the interval by its minimal value: in the graphic representation of the node, this value is put on the left (figure 1). A Z-value called Z_{max} limits the interval by its maximal value: in the graphic representation of the node, this value is put on the right (figure 1). The time-value corresponds to the moment when the interval appears in the data structure (called creation stage). In the graphic representation of the node, this value is put on figure 1 below the Z-values.

The Interval Treap is an interval binary tree. Indeed, a node can have at most two children because modifications can take place only on a Z_{min} or on a Z_{max}, and when an interval is modified, children are added to the node representing this interval. If the Z_{min} is modified, a new left child will be created, if the Z_{max} is modified, it will be a new right child. If a child does not exist, the node points towards NIL. A leaf is a node where both children pointers are NIL.

The Interval Treap can be updated by studying the different substraction boolean operations between two intervals which are the boolean operations used in machining.

Modification of an interval by its Z_{min} (case 1 in figure 2)

If an interval is reduced by its front, then the Z_{min}-value is modified. By convention, the Z_{min} is the value put on the left in the node. To show the dele-

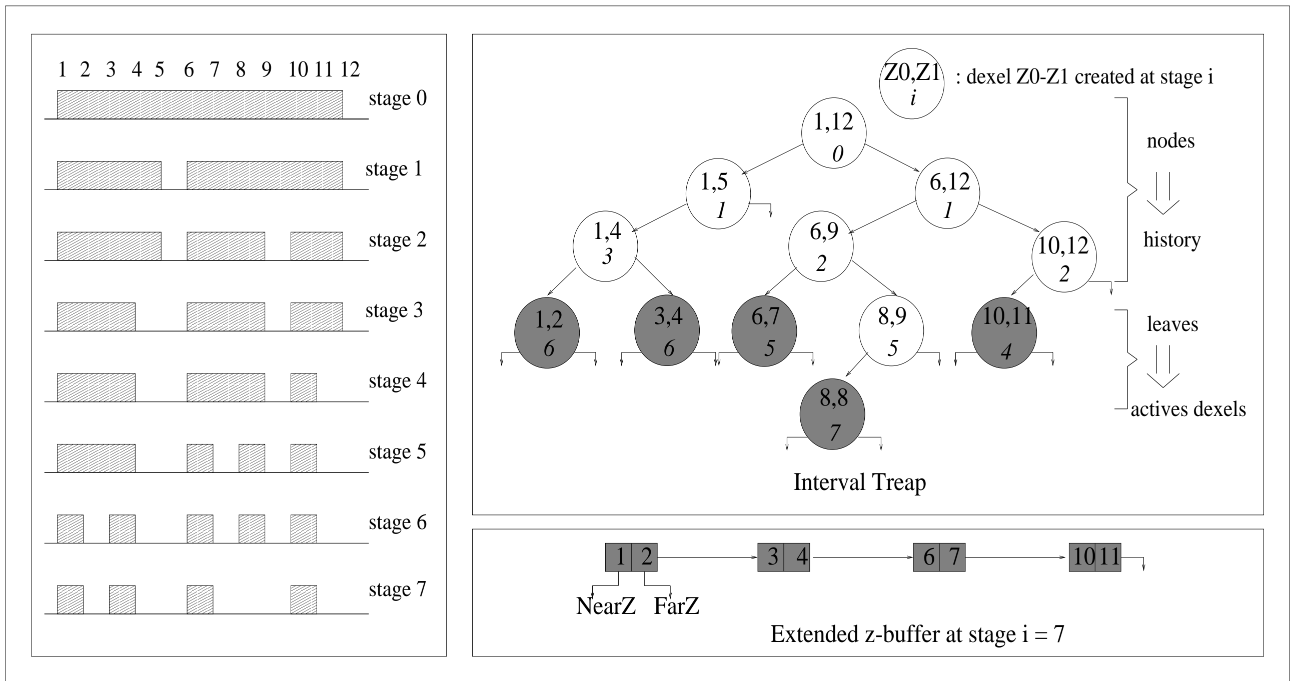

Figure 3: Example of Interval Treap and extended z-buffer

tion of Z_{min} from the parent interval, the pointer of its left child is now towards NIL. A new Z_{min} appears in the data structure and a new interval is created. To represent this new interval in the data structure, a new right child node is created. Its Z_{min}-value is the new Z_{min}-value. Its Z_{max}-value is the Z_{max}-value of the parent node. Its time-value is also the present stage n.

Modification of an interval by its Z_{max} (case 2 in figure 2)
This is the same case as previously where Z_{min} is replaced by Z_{max}, and Z_{max} replaced by Z_{min}.

Creation of an interval (case 3 in figure 2)
When a new interval appears in the data structure, a new Z_{min} and a new Z_{max} are created. The boundaries of the parent interval are not changed. The creation of a new interval is reflected on the data structure by the creation of two new children nodes: a right child and a left child whose values are respectively those of the previous case 1 and case 2.

Deletion of an interval (case 4 in figure 2)
The deletion of an interval is a difficult stage to model. To show that an interval is removed at stage i, only one child node must be created to represent this stage i. By convention, the node created will be a left child which will have the same values for Z_{min} and Z_{max}. This node is called “empty leaf” of the Interval Treap. Indeed, there will be no more possible modification from this node, it is thus a leaf which is described as empty because the Z_{min}-value and the Z_{max}-value are equal.

4 Using the Interval Treap with the extended z-buffer

4.1 The Interval Treap: a complete data structure for the extended z-buffer

The Interval Treap is a data structure well adapted for the extended z-buffer. The information about the dexel is contained in the basic element of the Interval Treap. The key value NearZ corresponds to the Z_{min}-value. The value FarZ corresponds to the Z_{max}-value. The time-value defines the moment a dexel appears in the extended z-buffer during the simulation.

When a dexel is modified, children nodes are added to the father node modelling the dexel. These children then become the new leaves of the tree. *The active dexels of the extended z-buffer are the leaves of the Interval Treap.* The leaves of an Interval Treap are arranged by order. To traverse the non-empty leaves of an Interval Treap from left to right is equivalent to traversing the dexels of the extended z-buffer extended in the same order(figure 3). The intermediate nodes of the Interval Treap are a part of the history of the extended z-buffer, because at a given time of simulation these nodes were also the leaves of the tree, so they were active dexels of the extended z-buffer.

An Interval Treap memorizes all the modifications undergone by the extended z-buffer during the simulation. It thus occupies in memory more space than a simple linked list, but its functionalities are at the same time those of the extended z-buffer and the traces. Simulations will show that the memory oc-

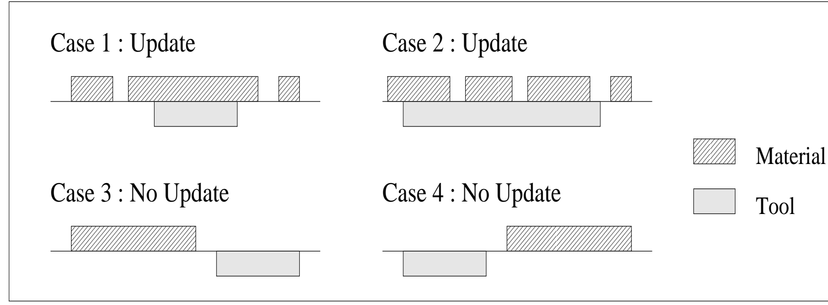

Figure 4: Update-No Update of a dexel

cupied by any Interval Treap is finally less significant than the whole memory occupied by two or three linked lists; one modelling the extended z-buffer, and the other(s) modelling the trace(s) associated with the extended z-buffer.

4.2 Updating of the Interval Treap

During the update of an Interval Treap, it is necessary to find the leaf (case 1 in figure 4) or the leaves (case 2 of the figure 4) corresponding to the dexels to be modified. As the leaves of the Interval Treap are arranged by order, **a postorder or preorder traversal is used.** To avoid traversing the whole tree for every new stage, we listed two cases (case 3 and 4 in figure 4) for which the dexel does not have any modifications.

- case 3: if the dexel of the tool precedes the dexel of the part without overlapping it. This condition can be summarized by the following inequality:
  FarZ of the tool $\geq$ *NearZ of the part*
- case 4: if the dexel of the part precedes the dexel of the tool without overlapping it. This condition can be summarized by the following inequality:
  FarZ of the part $\geq$ *NearZ of the tool*

A dexel will be modified if and only if case 3 AND case 4 are not satisfied. A dexel will not be modified if and only if case 3 OR case 4 is satisfied. Thus, a dexel could be modified if and only if it verifies the following condition (1):

$$\begin{gathered}\textit{FarZ of the tool} > \textit{NearZ of the part}\\ \textit{AND}\\ \textit{FarZ of the part} > \textit{NearZ of the tool}\end{gathered}$$

How can this condition modify the traversal of the Interval Treap?
At a given moment of the simulation, all the nodes of the Interval Treap have been a leaf, i.e an active dexel of the extended z-buffer. It is necessary for every node to verify if condition (1) is observed.

- if (1) is TRUE: the traversal continues because the following nodes (dexels) are susceptible to be modified.
- if (1) is FALSE: the traversal of this branch of the Interval Treap is stopped *momentarily.* Indeed, if a father node (dexel) does not verify condition (1) then its children nodes ("children dexels") will not verify condition (1) either.

Then, we want to determine in which case the traversal is interrupted momentarily or definitely. At each stage, the extended z-buffer of the part and the extended z-buffer of the tool have to be compared dexel by dexel. The updating on the dexels of the extended z-buffer of the part will not be possible any more when:

$$\textit{Far Z of the part} \geq \textit{Far Z of the tool} \quad (2)$$

For the extended z-buffer as an Interval Treap, the traversal of the tree will be stopped *definitely* in either of the following cases:

- if a leaf has just been created and:
  New FarZ of the part $\geq$ *FarZ of the tool.*
- if condition (1) is FALSE and if (2) is TRUE.

4.3 Use of the Interval Treap

4.3.1 Display of the current color on the screen

The visible color on the screen is the color of the last FarZ of the extended z-buffer. When the z-buffer uses the data structure of Interval Treap, only the last non empty leaf (NearZ $\neq$ FarZ), that is the leaf on the right in the tree, is interesting. That is why, for the display of the color of the current extended z-buffer, **the traversal of the tree will be a postorder traversal** (from right children to left children). If all the leaves are empty, the required color is the color of the background.

Toolpaths (Positions)	Pixels of the part	z-buffer			z-trace	time trace	interval treap
		Created dexels	Deleted dexels	All the dexels	Z elements	Time elements	IT nodes
17,108 (44,084)	121,067	83,217 *0.96Mo*	52,427 *0.60Mo*	390,417 *4.46Mo*	2,991,055 *45.64Mo*	2,174,530 *24.88Mo*	2,922,415 *45.66Mo*

Table 1: Memory occupation.

4.3.2 Use of Interval Treap as a history of the extended z-buffer

Thanks to this data structure, the machining can start again from any stage of the simulation. To reconstruct the Interval Treap of the extended z-buffer as it was at stage i of the simulation, it is necessary to "cut" correctly the tree. A preoder or postorder can be indifferently chosen because all the branches of the tree must be examined. For each ramification of the tree, the traversal is stopped in either of the following cases:

- if the current node verifies the following inequalities:

  (creation stage of the left child > stage i)
  AND
  (creation stage of the right child > stage i)

  The pointers of the current node towards its right and left children are initialized with NIL.

- if the current node is a leaf (node with neither right nor left child).

Once the Interval Treap is reconstructed, two applications can be envisaged. Thanks to this data structure, it is first possible to display on the screen the color of the pixel at stage i (see previous paragraph). Then, the extended z-buffer can undergo new operations in order to play another simulation.

5 Experimental evaluation of the algorithm

Let's take the example of the machining simulation of a small mould. Machining this mould requires 17,108 toolpaths - most of them being 0.5 mm long - as well as a 107×72×33 mm stock and a 3 mm tool. The machining takes 1 h 30 min with a three-axis milling machine. To evaluate the algorithms, a 300 Mhz PC is used. It has a level-two 512 Ko cache memory and a 128 Mo RAM. The simulation software uses direct access to a standard graphics board. The preorder and postorder traversals of the Interval Treap use recursive algorithms. A 640×480 image (307,200 pixels) [1] is used. The data structures have 307,200 dexels, Z-elements and time-elements when initialized. For an image of a given size, the number of elements varies according to several parameters: the size and the shape of the stock, the number of toolpaths and the machining strategy used, and finally the view point. A dexel and a time-element use 12 octets

First, we study the memory and the time required to create the Interval Treap. The creation of the extended z-buffer takes 99.42 s. The construction of the Interval Treap requires 341.13 s. These times do not take into account the display time. The Interval Treap needs 3.43 times more time to be set up than the data structure of the linked list.

The results related to the memory used are presented in table 1. At the end of the simulation, the Interval Treap occupies 10 times more memory space than all the necessary dexels for the implementation of the extended z-buffer as linked list. But if we consider the z-trace and the time-trace, the extended z-buffer can be reconstructed and/or rapidly displayed for any given stage of the simulation. The memory occupation is also *4.46Mo* + *45.64Mo* + *24.88Mo*, so *74.88Mo*. With the Interval Treap, the memory occupation is only *44.59Mo*, and any given stage can also be reconstructed and/or displayed and we save 40 % memory off.

We now study the reconstruction times of scenes without access to the graphics board. Considering the large number of elementary positions present in the simulation (44,084), we list simulation scenes which correspond to arbitrary elementary positions. For example, the scene in which the part has already undergone a tenth of the simulation is listed in the first column. This corresponds to position 4,408 when the extended z-buffer is constructed. To simplify matters, let's say that this simulation stage is called "stage $\frac{1}{10}$". For the construction of the extended z-buffer, the time required for stages $\frac{1}{10}$, $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$ and the last stage is respectively 10.88 s, 26.03 s, 50.14 s, 74.20 s and 99.42 s, that is equivalent to several ten seconds.

[1] For this specified view, one pixel represents a 0.26 mm length on the part

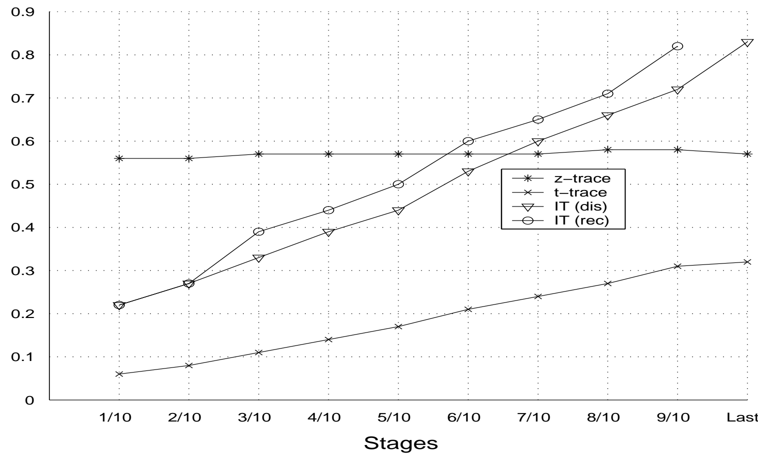

Figure 5: Reconstruction of the scene (time in seconds)

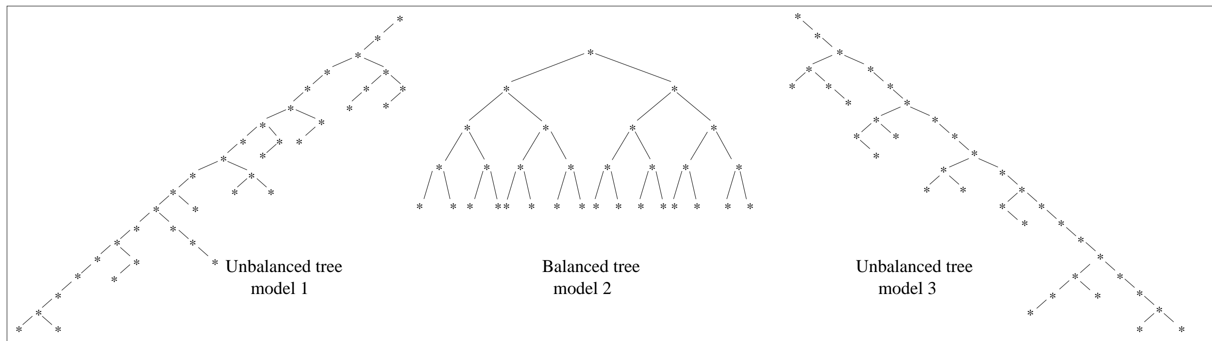

Figure 6: Different Interval Treaps

In figure 5, the time required is listed for any given stage: for the setting up of the display of the scene with the time-trace and the Interval Treap (IT(dis)), and for the reconstruction of the scene with the z-trace and the Interval Treap (IT(rec)). These times are now equivalent only to tenths of second. To display any given scene, the Interval Treap uses in the worst case 3.7 more time than the t-trace, that is only 2.2% more than the time necessary to construct the same scene with the extended z-buffer. The display of the scene with the Interval Treap is less quick than with the t-trace: in fact, the traversal of empty leaves either decreases or increases the time, depending on the specified point of view.

To reconstruct any given scene, the Interval Treap uses in the best case 30% less time than the reconstruction of the same scene with the z-trace. The Interval Treap uses in the worst case only 40% more than the z-trace. The z-trace data structure has no empty leaves, that is why the reconstruction with the Interval Treap is more time consuming for the scenes at the end of the simulation. Although, the Interval Treap uses in the worst case only 2.02% of the time necessary to construct the same scene with the extended z-buffer.

It is interesting to notice that only 4 reconstructions of any scene at the end of the simulation are necessary to validate the Interval Treap data structure.

In figure 6, we listed different possible models for Interval Treap. The balance of an Interval Treap depends on several parameters, especially the point of view (orientation of the workpiece in the image space) and the machining strategy used. For example, the unbalanced trees of models 1 and 3 are obtained with the zigzag machining strategy. The balanced trees of model 2 could be created with a machining strategy such as parallel contour. Let's take the example of the unbalanced Interval Treap of model 1. We now study the performances of this new implementation for the extended z-buffer on this tree which is not the best case for our data structure. The times for the reconstruction and the display of any given scene by an IT are very close. The unbalance is also shown by these times. Indeed, to reconstruct an IT at stage i of the simulation from a balanced IT of model 2, all the branches of this tree have to be traversed, so many comebacks must be executed. On the contrary, for the display of the same scene i of the simulation, only a partial traversal is necessary up to the right node (which was the last leaf of the IT at stage i). For the unbalanced trees of models 1 and 3, the comebacks are fewer but most branches can be seen in the whole traversal compared to the partial traversal. For the

reconstruction of any given scene, especially for the scene at the beginning and at the middle of the simulation, the traversal of an unbalanced Interval Treap of model 1 or 3 can be faster than the traversal of a balanced Interval Treap because fewer branches must be treated.

6 Analysis of the algorithm

The analysis of the algorithm refers to two different designs for the performance study of a program. The first approach consists in determining the performances of any algorithm by considering the worst case. For any given algorithm solving a specific problem, a lower limit can be found, and this limit corresponds to the performances of the algorithm in the worst case. Thanks to this approach, it is possible for instance to say that the search time for a linked list of N elements is in $O(N)$ in the worst case. The second approach consists in characterizing in a rigorous way the performances of an algorithm by analyzing the best case, the average case and the worst case with methods making it possible to refine the precision. This approach relies on the precise enumeration of the different configurations of a data structure.

We adopt the second approach. Thanks to the calculation and the evaluation of the average height (or level), the balance of the Interval Treap can be characterized and classified as well as possible. Indeed, in the best case, a tree of N nodes is balanced. Its maximal height is $\log_2(N)$ and the operations of insertion and search can be implemented in $\log_2(N)$ time. In the worst case, the tree is completely unbalanced (one node per level). Its maximal height is N, and the operations of insertion and search can be implemented in N time, this worst case structure is no more efficient than a regular linked list. So, the time required for the construction of the IT, and the time required for the display and the reconstruction of a specific scene of the simulation depend on the balance of the Interval Treap. Let's take the example of the model 1 tree in figure 6: it is an Interval Treap obtained during the simulation. This tree has 38 nodes, this is about the average number of nodes of the Interval Treap used in the simulation. In the best case, the tree could be balanced. The average height for a balanced search tree of 38 nodes is $H_{balanced} = 3.5$. In the worst case, the tree could be completely unbalanced. The average height for a completely unbalanced search tree of N nodes is $H_{unbalanced} = \frac{N-1}{2}$, so for a tree of 38 nodes $H_{unbalanced} = 18.5$. For the average case, Segewick studied the binary search trees constructed from a random drawing for their key values [12]. He obtained the following formula about the average height for such a tree of N nodes: $H_{rand} = 4.311 \times \ln(N) + o(\ln(N))$, so for a tree of 38 nodes $H_{rand} = 15.68$. The average height calculated on the Interval Treap of model 1 in figure 6 is: $H_{IT1} = 9.11$. This value is about twice as much as for a completely unbalanced tree, and it approaches the average height of a balanced tree. Using the Interval Treap as a new data structure for the extended z-buffer is thus interesting, even in a case such as the unfavourable example studied in the previous paragraph.

7 Conclusion

To recreate a given scene in NC milling simulation using the extended z-buffer, we have presented a new single data structure based on interval binary trees: the Interval Treap. These search trees can quickly display and/or reconstruct any given scene of the simulation. We have also studied the performances of this new data structure for the extended z-buffer in two different ways. The experimental evaluation showed that even if the setting-up of Interval Treaps is somewhat time consuming, this extra time is recovered if several scenes at the end of the simulation are reconstructed successively. Moreover, this method is compatible with the standard graphics board, the memory capacity and the processing speed of PCs assembled today. Thanks to the analysis of the algorithm, it is then possible to characterize the balance of the Interval Treap by studying the average heights. The latter are close to the best case of balanced trees. The Interval Treap data structure seems to be well adapted for a complete use of the extended z-buffer.

References

[1] 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.

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

[3] 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.

[4] 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.

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

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

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

[8] 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.

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

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

[11] 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.

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