TABLE-DRIVEN QUADTREE TRAVERSAL ALGORITHMS

by

Mark R. Lattanzi

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

for partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Dr. Clifford A. Shaffer, Chairman

Dr. James D. Arthur

Dr. Lenwood S. Heath

May, 1989

Blacksburg, Virginia

TABLE DRIVEN QUADTREE TRAVERSAL ALGORITHMS

by

Mark R. Lattanzi

Dr. Clifford A. Shaffer, Chairman

Computer Science

(ABSTRACT)

Two quadtree algorithms are presented that use table driven traversals to reduce the time complexity required to achieve their respective goals. The first algorithm is a two step process that converts a boundary representation of a polygon into a corresponding region representation of the same image. The first step orders the border pixels of the polygon. The second step fills in the polygon in $O(B)$ time where B is the number of border pixels for the polygon of interest. A table propagates the correct values of upcoming nodes in a simulated traversal of the final region quadtree. This is unique because the pointer representation of the tree being traversed does not exist. A linear quadtree representation is constructed as this traversal proceeds.

The second algorithm is an update algorithm for a quadtree (or octtree) of moving particles. Particle simulations have had the long-standing problem of calculating the interactions among n particles. It takes $O(n^2)$ time for direct computation of all the interactions between n particles. Greengard [Gree87, Carr87] has devised a way to approximate these calculations in linear time using a tree data structure. However, the particle simulation must still rebuild the particle tree after every iteration, which requires $O(n \log n)$ time. Our algorithm updates the existing tree of particles, rather than building a new tree.

It operates in near linear time in the number of particles being simulated. The update algorithm uses a table to store particles as they move between nodes of the tree.

Acknowledgements

I would like to express my thanks to Dr. Clifford A. Shaffer for his unending patience and support during the research and writing of this thesis. His advice and suggestions have been invaluable. I would also like to thank Dr. James D. Arthur and Dr. Lenwood S. Heath for serving as members of my committee. Thanks are also due to

for all her help producing the figures in this document.

CONTENTS

List of Illustrations

1. INTRODUCTION

1.1. Overview

Choosing the correct data structure can significantly decrease the time complexity of a task. The quadtree [Same84b, Same89a, Same89b] has become a popular data structure for applications in computer cartography, computer graphics, and image processing as well as other related disciplines. The quadtree is a hierarchical data structure that stores planar regions. It recursively subdivides a square region into four smaller squares (quadrants) until a certain criterion is met. Changing this criterion gives rise to many varieties of quadtrees. One such variant of the quadtree, called the Point Region (PR) quadtree [Same84b, Oren82], stores at most one point in each node (quadrant) of the tree (Figure 1). If two points ever come to be in one node, the node is split into four equal child nodes each having one fourth the area of the original node. Splitting occurs until each point resides in a quadrant by itself. The *region* quadtree, which stores regions instead of points [Klin79], splits nodes until every quadrant in the image is homogeneous (Figure 2). An octtree [Meag82a] is the three dimensional analog of the quadtree. Instead of planes decomposing into four quadrants, volumes are split into eight octants. The main idea is unchanged. Henceforth, "quadtree" will be used as a generic term inferring to any of these data structures.

Presented below are two related quadtree based algorithms. The first one converts a boundary representation of a polygon to an equivalent region representation for that polygon. This conversion is often needed in geographic information systems (GIS) [Peuq84, Same84a] since both methods of storing polygons may be used in such systems. Conversion algorithms of this type operate in two phases. Phase one constructs a region quadtree

using only the border pixels; phase two fills in the region by determining the values of the unassigned nodes in the quadtree. Our algorithm uses a unique and potentially versatile method for constructing a linear quadtree by simulating a traversal of the pointer-based quadtree for the region of interest. Phase one of our algorithm has been reduced to a simple sort, and our filling phase takes $O(B)$ time where B is the number of border pixels on the boundary of the region. Previously reported methods [Same80, Webb84, Atki86] perform the filling phase in $O(n \cdot B)$ time where n is the depth of the PR quadtree.

The second algorithm removes a significant bottleneck from particle system simulations: updating the storage structure of the particles. Particle simulations are used in many ways, including astrophysical models [Appe85], flock simulations [Reyn86], and fluid mechanics [Hock88]. A recurring problem is how to efficiently store and maintain a system of moving particles. For large n (the number of particles), the cost of computing these interactions as well as the expense of maintaining the data structure to store the particles becomes prohibitively large. Earlier hierarchical algorithms have time complexity $O(n \log n)$ [Appe85, Barn86, Mill68, Mill70]. Greengard has devised a way to approximate particle-particle interactions by using a quadtree to aggregate the masses of the particles at internal nodes [Gree87, Carr87]. His algorithm is $O(n)$ where n is the number of particles. Since these forces can be found in $O(n)$ time, the bottleneck of a moving particle simulation now becomes the rebuilding of the tree to store the newly positioned particles. To insert n particles into a quadtree requires $O(n \log n)$ effort. The algorithm presented below (hereafter, the fast update algorithm) updates the existing tree of particles in near linear time, thus making the total particle simulation algorithm complexity $O(n)$. This is a significant improvement over the naive algorithm since n is usually on the order of tens of thousands of particles [Hock88].

The relationship between these two algorithms is that both use tables to improve the efficiency of their tree traversals. Other table-supported traversal algorithms may be found in [Same85b, Shaf87a, Shaf87b, Shaf88]. Both algorithms perform constant time or near constant time operations at each node during a tree traversal to accomplish their respective goals. Other similarities exist and will be discussed later.

1.2. Background and Definitions

Before the algorithms can be explained, several terms pertaining to quadtrees must be defined. Quadtrees contain two types of nodes: *internal* nodes and *leaf* nodes. The leaf nodes often hold the actual data to be stored in the tree; internal nodes serve as pathways to the leaves of the tree. The *root* node of a quadtree represents the entire image. For non-trivial quadtrees, it is an internal node. In quadtrees, nodes split into four *quadrants* traditionally labeled NW, NE, SW, SE (Figure 2). In three dimensions, octtree nodes split into eight *octants* labeled OCT0 through OCT7 (Figure 3). When an octant splits, eight smaller volumes (cubes) are formed. When a node of a quadtree or octtree splits, the newly formed nodes are called *children*; the original node is called their *parent. Siblings* are nodes that have the same parent node. *Ancestors* of a node are all of the nodes on a path from the current node to the root node of the tree.

Quadtree nodes have an associated *depth* in the tree. The depth of a node is the distance from the root of the tree to the node. The root node has a depth of zero. For a $2^N \times 2^N$ image, the maximum depth for a node is N. Depth is normally used when referring to the whole quadtree, while *level*, another characteristic of quadtree nodes, will be used when referring to specific nodes in the quadtree. The level of a node is related to the node's size. In two dimensions, a node that is $2^n \times 2^n$ pixels has a level of n. For

a quadtree, the smallest possible node (level 0) corresponds to a pixel ($2^0 \times 2^0$). For a $2^N \times 2^N$ image, the root node is at level N. The root node therefore represents the entire image.

Two storage representations for quadtrees are the *pointer-based* quadtree and the *linear* quadtree. A pointer-based quadtree uses the traditional tree structure to represent the block decomposition hierarchy, complete with pointers linking a parent node to its children (as shown in Figure 2d). The linear quadtree [Garg82, Abel83] stores only the leaf nodes of the tree along with the size and position of each node. These leaf nodes correspond to the blocks illustrated by the image decomposition in Figure 2c. Some practitioners [Garg86, Atki86] define the linear quadtree as the list of all the BLACK leaf nodes, since the WHITE nodes can be interpolated as necessary. Following [Shaf87a], we prefer to include all of the leaf nodes in the linear quadtree. Each node in the linear quadtree has an address associated with it called a *Morton code* [Mort66]. A Morton code is a locational code produced by interleaving the bits of the (X,Y) values of the upper left corner of the given node, first Y, then X (Figure 4). An equivalent code can be used to address octtree nodes using the Z coordinate as the most significant coordinate. The origin of each two dimensional images is defined to be in the upper left corner.

A *traversal* of a linear quadtree is the process of visiting every node within the tree in sorted order. Traversing the leaves of a quadtree or octtree in Morton order is equivalent to a postorder traversal of the pointer-based tree structure. Morton order for a quadtree is NW, NE, SW, SE; Morton order for an octtree is OCT0, OCT1, . . . , OCT7 (Figure 3).

Pointer-based quadtrees are normally used when storing the quadtree completely in core memory would be advantageous [Shaf86, Shaf87a]. Particle simulations need to be fast. The entire tree needs to be readily available for quick, easy access. Therefore,

pointer-based structures are most often used to implement particle simulations. Linear quadtrees are typically used for disk-based applications for organizational reasons [Shaf86, Shaf87a]. When disks are used, disk fetches need to be minimized; linear quadtrees store no internal nodes, so separating the nodes into pages becomes an easier task.

A *boundary representation* of a polygon describes its perimeter and shape. One structure often used to represent boundaries is the *chaincode* [Free74, Dyer80, Same80]. A chaincode is a set of ordered directional codes that when followed, will produce a list of the border pixels on the polygon described. A border pixel is a pixel in the image that lies on the boundary of the polygon. Typically, a starting point is given with a chaincode, although, this is not required [Same80].

The two algorithms below will work for quadtrees and octtrees, linear or pointer-based. Our filling algorithm uses a linear quadtree (2 dimensions) while a pointer-based octtree (3 dimensions) is used by the fast particle update algorithm. This should give the reader a flavor for both type of tree algorithms.

1.3. Document Description

Below we describe the two algorithms in greater detail. Chapter 2 is devoted to the conversion algorithm. It covers previous work, describes the algorithm, analyzes it, and draws conclusions. Chapter 3 similarly describes the fast update algorithm. Pseudocode is presented for each algorithm. Examples of the algorithms are also provided. Chapter 4 is devoted to overall conclusions and ideas for future work in these two areas.

2. THE CONVERSION ALGORITHM

2.1. Previous Work

Four previous algorithms for chaincode to quadtree conversion have been presented (related work may also be found in [Hunt79a, Hunt79b, Garg84]). The first [Same80] converts a chaincode to a pointer-based quadtree representation. This algorithm first builds a quadtree by inserting each border pixel derived from the chaincode as a BLACK node. The tree is constructed by progressing along the chain of boundary pixels, using neighbor finding operations [Same82, Same85a] as the chaincode is processed. A second phase then traverses this quadtree in order to fill the polygon bordered by the chaincode. The order of complexity for this algorithm is $O(n \cdot B)$ where B is the number of border pixels on the chaincode and n is the depth of the resulting tree for a $2^n \times 2^n$ image.

The second algorithm [Atki86] begins with a list of linear quadtree nodes corresponding to the border pixels derived from the chaincode, and constructs a linear quadtree. This list of nodes is not initially sorted by Mort[illegible] code. Each border pixel contains information about which edges of the pixel are adjacent to WHITE pixels (i.e., those edges of the border pixels that are adjacent to the chaincode border). The initial node list is processed by a recursive procedure that splits the list into four bins depending on which quadrant of the image each node is in. The procedure is then re-invoked for each of the resulting sub-lists, which are in turn subdivided based on which subquadrant of the image the corresponding pixel appears in. After n subdivisions for a $2^n \times 2^n$ image, each sublist now contains at most four nodes, corresponding to siblings within a 2×2 sub-block of the image. This process can be viewed as a recursive radix sort on the border node list. The values of these sibling nodes and their border states determine the values of any missing

siblings, and also determine if the four siblings may be merged to form a single node. The resulting node sublist is then passed back as the recursion unwinds, allowing even larger missing nodes to be added and, if possible, allowing larger siblings to be merged. The final result is an unsorted list of nodes corresponding to the linear quadtree for the image with the interior nodes of each polygon set to BLACK. The order of complexity for this algorithm is $O(n \cdot B)$ since each border pixel participates in n constant time operations. A final sorting phase is required to generate a node list ordered by Morton code.

The third algorithm [Mark85a, Mark85b] parallels Samet's algorithm [Same80] except it allows for more generalized input. Instead of a list of border pixels, Mark and Abel's algorithm works for any vector representation of a polygon. Thus, the first step of their algorithm is to compute the list of pixels lying on the boundary of the polygon of interest. This step takes $O(B)$ time. However, the input to step two of the algorithm, i.e., the polygon fill phase, is a sorted list of pixels by Morton code. So, the pixel list must be sorted before being passed to step two. The main theorem underlying this algorithm states that given a sorted list of border pixels, the absent nodes between any two border pixels must be either all the same color, or change colors only one time. In this second case, the intermediary nodes will be in two contiguous sets. The first set will be one color, and the second set will be the other color. The importance of this theorem is that a limited amount of information is needed to determine the colors of all the absent nodes. Mark and Abel show that at most one ancestor find and between two and four neighbor finds are performed per node. A detailed analysis of this process was presented by Mark and Abel in an earlier paper [Mark85c]. As with the algorithm of [Same80], the overall time complexity of this algorithm is dominated by their polygon fill operation: $O(n \cdot B)$ where B is the number of border pixels and n is the depth of the final tree.

Webber [Webb84] improved the conversion algorithm of Samet [Same80] by using a method called *path length balancing.* The main idea is to treat the resulting quadtree as two one-dimensional objects: a horizontal one and a vertical one. By strategically positioning the polygon in the quadtree, the total cost of the neighbor find operations can be minimized. Thus, the border pixels can be inserted into the quadtree, and the interior of the region can be filled in time proportional to the number of nodes in the quadtree, or equivalently, the number of pixels obtained from the chaincode. Although Webber's result is linear, it is constrained by the fact that the region representation of the polygon must be placed at a certain optimal position within the tree to achieve a linear time complexity. Sometimes this is not desirable, so path length balancing cannot be used, and this algorithm's time complexity will be reduced to $O(n \cdot B)$.

2.2. General Description of our Conversion Algorithm

Our new algorithm converts a boundary representation to a linear quadtree in Morton code order. The boundary representation may be either a chaincode or a vector representation of the polygon's perimeter, but it must be convertible to a list of the boundary pixels. We assume for simplicity that all images are binary, although our algorithm would work equally well for multicolor images. Nodes interior to the polygon boundary are labeled BLACK after the conversion process is complete; nodes exterior to the polygon are labeled WHITE.

Our algorithm works in the spirit of Samet's algorithm [Same80] for producing a pointer-based quadtree. Both algorithms take as input a list of the border pixels generated from a chaincode. Both algorithms operate in two phases. However, our algorithm reduces the complexity of Samet's initial construction phase to a simple sort of the border pixels

by Morton code. As demonstrated in [Webb84], Samet's construction phase has worst case time cost $O(n \cdot B)$. Our construction phase is a sort of all the border pixels requiring at worst $O(B \log B)$ time. For long lists of pixels, these times are asymptotically the same; however for short perimeters (as might normally be expected), our algorithm is faster.

The polygon filling phase of our algorithm corresponds closely to Samet's second phase, but this portion of our algorithm is performed in time $O(B)$ instead of $O(n \cdot B)$. This linear time bound is achieved by using a simple table to record the expected values of neighboring nodes yet to be visited during the simulated traversal of the tree. Since a tree traversal visits every node one time, and we do constant processing at each node, our polygon filling step is $O(n)$ where n is the number of nodes in the tree. By the theorem presented in [Hunt79a], the number of nodes in the resulting quadtree is directly proportional to the number of border pixels for the polygon, so this step takes $O(B)$ time overall. The total cost of our algorithm will therefore be $O(B \log B)$, dominated by the cost of the initial sorting step. Alternatively, we could use the initial construction phase of [Webb84] yielding an overall time complexity of $O(B)$ with the constrained positioning of the polygon within the tree.

2.3. Detailed Description of our Conversion Algorithm

As with the algorithm of [Mark86], the first step in our chaincode to quadtree conversion process is to generate a sorted list of linear quadtree node records representing the border pixels. Border pixels are defined to be those pixels with at least one edge on the chaincode boundary. The initial border pixel list is created by following the directional codes of the chaincode from pixel to pixel. The second (filling) step of our algorithm takes as input a set of sorted border pixels. The border pixels may be four or eight connected.

Polygon boundary descriptions are restricted as follows.

1) Each polygon represented must be closed (i.e., it must begin and end on the same pixel).

2) Polygons may not be self-intersecting, nor may two polygons in the image intersect.

As the chaincode is processed, a linear quadtree record for each border pixel is created that contains the Morton code, the pixel's x and y coordinates, and a border code telling which sides of the pixel are adjacent to WHITE pixels (i.e., which borders of the pixel actually touch the chaincode boundary). The Morton code duplicates the x and y coordinates for the pixel; however, presentation of the algorithm is simplified if all operations are done in terms of x and y coordinates. Final processing of the node list can remove this redundant information if desired. After all border pixels have been generated, they must be sorted by Morton code, and passed to the filling phase of this algorithm.

The second phase processes the sorted list, simulating a traversal on the corresponding pointer-based quadtree. The purpose of this traversal is to fill in the interior of the polygons whose borders are described by the boundary pixel list. To aid filling of the polygons during this traversal, a table is maintained that indicates the expected color of adjacent nodes yet to be processed. This color table is updated as the node list is processed to reflect the current state of the traversal.

The polygon filling algorithm begins by finding the first *chunk* in the image. A chunk is defined to be all pixels whose Morton codes lie between two successive border pixels (that is, successive in terms of Morton code). Each chunk in the image must eventually be broken into quadtree nodes. A chunk's constituent nodes are determined during the traversal phase and, based on the color of each such node, our table is updated. As nodes are created during the traversal, they are passed to a routine that compares the node with its siblings to determine if all four siblings are the same color and thus may be merged

into a single node. This routine simply keeps a list of consecutive nodes of the same color. When a node of a second color is processed, it may not merge with any nodes preceding it; thus, stored nodes may be output. Four consecutive siblings of the same color are merged to form a single node. At most $3n$ nodes need to be stored at one time. For further details on the output node process, see [Shaf86]. The traversal continues until all the chunks have been processed and the resulting nodes output.

If the first border pixel of the image is not at position (0, 0), then maximal WHITE nodes are output until the first border pixel is reached. Similarly, after the last border pixel has been processed, WHITE nodes are output until the lower right corner of the image has been reached.

2.4. The Simulated Traversal

This section describes in detail the table driven simulated traversal that gives the filling phase linear time complexity. When the initial border pixel list is created, a border code is associated with each pixel (similar to the border code used in [Same80, Atki86]). This border code describes which of the pixel's edges may be adjacent to WHITE pixels. The code can be viewed as a 4 bit map with bits 0, 1, 2, and 3 corresponding to directions N, E, S, and W, respectively. Alternatively, the code can be viewed as a four bit value generated by summing values of N = 1, E = 2, S = 4, and W = 8 for each edge on a border. For example, a pixel in the SE corner of a region has a border code of 2+4 = 6. Note that the color for all border pixels is BLACK, since by definition border pixels are within the polygon defined by the chaincode.

After sorting the border pixel list into Morton code order, the complete linear quadtree is generated by processing the pixel list in Morton order while simulating a traversal

of the corresponding pointer-based quadtree. To aid this traversal, information is stored in the *color table.* This table is a two dimensional array with n rows for a $2^n \times 2^n$ image, i.e., one row for each level of the corresponding tree (except for the root at level n). Each row contains four columns, representing quadrants NW, NE, SW, and SE. Thus, each entry in the table corresponds to a quadrant at a particular level of the tree. The value stored at each entry is the expected color for the next node of that level and quadrant in the tree. Node levels are labeled from 0 (corresponding to nodes of size $2^0 \times 2^0$, i.e., pixels) to $n-1$ (corresponding to nodes of size $2^{n-1} \times 2^{n-1}$, i.e., children of the root node). Before processing begins, the color table is initialized so that all entries contain the value WHITE.

The traversal of the node list begins by setting the current level to be n, corresponding to visiting the root of the non-existent pointer-based tree. The initial position in the traversal is (0, 0), which is the upper left corner of the image (as well as the root node). At each stage of the traversal, we maintain the current level in the tree, the current (x, y) position in the tree, and four values that describe the four corner pixels of the current node. Each corner pixel in the current node may also be the corner pixel of some larger node. The *maximum corner value* for each corner is the level of the largest possible node for which that pixel is the corresponding corner. For example, pixel (0, 0) is the NW corner of a node corresponding to the entire image, while pixel (4, 0) is the NE corner of a 4×4 pixel node, regardless of the actual level for the current node. We maintain this corner information in order to allow updates to the color table in constant time. During the traversal process, corner information for the current node is easily derived from the corner information associated with the pixel's parent.

Processing of the pixel list is simply a matter of processing the collection of pixels that lay between each pair of border pixels (i.e., a chunk). As the nodes within a chunk are

generated and output, the color table is updated for each node. The first node in a chunk is the border pixel itself, since this border pixel (which must be the NW corner of the chunk) may have a chaincode border along its E or S edge. (If necessary, this pixel will later be merged with its siblings.) The next node in the chunk is the largest quadtree block with the current pixel position as its NW corner that does not include the next border pixel. After a node is output, a new (x, y) position is calculated – this will be the next pixel in Morton order following the node just processed. If the location of this pixel is the same as the location of the next border pixel on the border node list, then the current chunk is complete and processing of the next chunk may begin. This continues until the entire list of border pixels has been processed. The final chunk after the last border pixel contains the remainder of the image. All the nodes in the last chunk are WHITE.

Procedure TRAVERSE of Section 2.5 is the heart of our algorithm. It simulates a pointer-based quadtree traversal while simultaneously traversing the list of border pixels. If the node at the current level does not fit into the current chunk (determined by function CHECK_NODE_SIZE), then the algorithm recursively processes the node's four children. For each of these children, the NW corner's coordinates and the maximum corner values are computed. A maximum corner value is the level of the largest node with that pixel in the corresponding corner. The largest eastern neighbor for a node N will therefore be at the same level as N's maximum NE corner value. Similarly, the largest southern neighbor for N is its maximum SW corner value. If the block corresponding to the current node in the quadtree traversal fits within the current chunk, then the color of the current node is determined and the node is passed to the node merging routine ORDER_INSERT. (Further details on ORDER_INSERT may be found in [Shaf86, Shaf88].) If the current node is a border pixel, then its color is BLACK. If the current node is not a border pixel, then the

color of the current node is found in the color table. The entry examined in the color table is at the level and quadrant of the maximum corner value for the NW pixel of the current node. After the node's color is determined, this color may then be used to update the color table.

If the current node is a NW, NE, or SW child of its parent, then the color table is updated. For SE children, no update is required since other nodes will later update those color table entries. The color table is updated as follows. For the current node, the sizes for the largest adjacent eastern neighbor and the largest adjacent southern neighbor are calculated. They are obtained from the maximum corner values for the current node. If the current node is a border pixel, the pixel's border code determines what color to use when updating the table. For example, if the pixel has stored a border along the eastern edge, then the largest quadrant directly east of the current pixel is set to be WHITE. This is accomplished by setting the entry in the color table for the largest eastern neighbor (level and quadrant). If the current node is not a border pixel, then the neighbor receives the same value as the current node. The quadrant of the neighbor is determined by a simple function based on the current node's location and the level passed down from the current node's parent in the variable *maxne*. If no eastern border exists, the color table entry is set to BLACK. The southern direction is processed in a similar way, using the value *maxsw*.

When the table is used to determine a node's color, the level of the largest node that has a common northwest corner pixel with the current node is used for the table lookup. This insures that the correct color will still be in the table, since no nodes will have been processed that could have changed this value in the table.

2.5. Conversion Algorithm Pseudocode

This section contains a Pascal-like pseudocode description for the third phase of our chaincode to quadtree conversion algorithm. This pseudocode contains four procedures; START_TRAVERSE is called first. START_TRAVERSE initializes the table values and begins the simulated pointer-based quadtree traversal by calling TRAVERSE, which performs the actual traversal. The last two procedures are CHECK_NODE_SIZE, a function that determines if the current node will fit into the current chunk, and UPDATE_TABLE, a procedure that updates the color table based on the level and quadrant passed to it.

The pseudocode below calls several primitive procedures that did not merit a formal description. BIT_SET(*border*, *bnum*) returns TRUE if bit *bnum* from the right is set in the integer *border*, and FALSE otherwise. The function FINDQUAD(x, y, *level*) returns the quadrant of the node containing the point (x, y) at *level* with respect to the node's parent. ORDER_INSERT(*level*, *color*, x, y) outputs a *color* node of size $2^{level} \times 2^{level}$ whose NW corner pixel is (x, y). This procedure merges siblings if they are all the same color. GET_NEXT and TEST_NEXT both deal with the input list of border pixels. GET_NEXT(x, y, *border*) takes the next border pixel off the top of the border pixel list and stores its coordinate and border information in x, y, and *border*. TEST_NEXT(x, y) checks if the next pixel on the list has the coordinates x and y. GET_X() and GET_Y() return the x and y coordinates of the next border pixel, respectively.

```
  { Declarations of types and global variables. }
type
  QUADTYPE = (NW, NE, SW, SE);
&BCODE = integer; { border code type: 1 = N, 2 = E, 4 = S, 8 = W. }
  NODE = record { record type for border pixel list }
    x, y : integer;
```

```
    border : BCODE { border code for each border pixel }
  end;
var
  colortable : array[0..MAXLEVEL, NW..SE] of integer;
  border : BCODE; { border code of node currently being processed }
  currlvl : integer; { level of node currently being processed }
  currquad : QUADTYPE; { quadrant of node currently being processed }

procedure UPDATE_TABLE(maxnw, maxne, maxsw, x, y : integer);
  { Updates the color table based on the current node just outputted. }
var
  width : integer; { Width of the current node. }
  eastquad, southquad : QUADTYPE; { Largest quadrant to E or S of current one. }
begin
  width := 2^currlvl;
  case currquad of
    NW: begin
      eastquad := NE;
      southquad := SW;
      if BIT_SET(border, 2) then { If eastern border exists }
        colortable[currlvl][eastquad] := WHITE
      else colortable[currlvl][eastquad] := color;
      if BIT_SET(border, 3) then { If southern border exists }
        colortable[currlvl][southquad] := WHITE
      else colortable[currlvl][southquad] := color
    end
    NE: begin
      eastquad := FINDQUAD(x + width, y, maxne);
      southquad := SE;
      if BIT_SET(border, 2) then { If eastern border exists }
        colortable[maxne][eastquad] := WHITE
      else colortable[maxne][eastquad] := color;
      if BIT_SET(border, 3) then { If southern border exists }
        colortable[currlvl][southquad] := WHITE
      else colortable[currlvl][southquad] := color
    end
    SW: begin { Eastern neighbor already set by NE sibling }
      southquad := FINDQUAD(x, y + width, maxsw);
      if BIT_SET(border, 3) then { If southern border exists }
        colortable[maxsw][southquad] := WHITE
      else colortable[maxsw][southquad] := color
    end
    SE: { Do nothing, neighbors will be set later on }
  end{ case currquad of ... }
end; { UPDATE_TABLE }

function CHECK_NODE_SIZE(x, y : integer) : boolean;
```

```
  { Returns whether or not the quadrant of size (2^currlvl × 2^currlvl) whose NW corner is
    (x, y) is contained within the current chunk. }
var size : integer;
begin
  if (border <> 0) and (currlvl <> 0) then { If current node is a border pixel }
    return (FALSE);
  size := 2^currlvl − 1;
  if (x + size >= GET_X()) and (y + size >= GET_Y()) then
    return (FALSE)
  else return (TRUE)
end; { CHECK_NODE_SIZE }

procedure TRAVERSE(maxnw, maxne, maxsw, x, y : integer);
  { This recursive procedure imitates a pointer-based tree traversal in order to fill in the
    polygon's area. Its parameters are the levels of largest quadrants that the current node
    is a northwest, northeast, and southwest corner of. Also passed along are the current
    x, y coordinates. }
var color : integer;
begin
  if CHECK_NODE_SIZE(x, y) then
    begin { If current node fits into the current chunk, output current node }
      if border = 0 then { if current node is NOT a border pixel }
        color := colortable[maxnw][FINDQUAD(x, y, maxnw)];
      UPDATE_TABLE(maxnw, maxne, maxsw, x, y)
        end;
      ORDER_INSERT(currlvl, color, x, y);
      border := 0 { Other nodes in chunk have no border }
    end
  else { Recurse down a level into four children }
    begin
      currlvl := currlvl − 1;
      for currquad in NW, NE, SW, SE do
        begin
          case currquad of
            NW: begin
              TRAVERSE(maxnw, currlvl, currlvl, x, y);
              x := x + 2^currlvl
            end
            NE: begin
              TRAVERSE(currlvl, maxne, currlvl, x, y);
              x := x − 2^currlvl;    y := y + 2^currlvl
            end
            SW: begin
              TRAVERSE(currlvl, currlvl, maxsw, x, y);
              x := x + 2^currlvl
            end
            SE: TRAVERSE(currlvl, currlvl, currlvl, x, y)
```

```
            end; { end case }
            if TEST_NEXT(x, y) then GET_NEXT(x, y, border)
          end; { End for loop }
        currlvl := currlvl + 1
      end{ End recursive case }
end;

procedure START_TRAVERSE;
begin
  currlvl := MAXLEVEL;
  if TEST_NEXT(x, y) then
    begin
      x := 0;    y := 0;    border := 0
    end
  else
    GET_NEXT_BORDER_PIXEL(x, y, border);
  TRAVERSE(MAXLEVEL, MAXLEVEL, MAXLEVEL, x, y)
end;
```

2.6. Example of the Conversion Algorithm

Figure 5a shows an example polygon with all border pixels shaded and their borders shown with a heavy outline. Figure 5b numbers each border pixel by order of its Morton code. Figure 5b also labels with letters those other blocks of the image that will be generated by the traversal phase. Note that nodes A and B are not part of the border list; however, they are interior to the polygon. Also note that blocks 1, 2, 3, and A must eventually be merged to form a single quadtree node. Each border pixel has a border code associated with it. This code indicates which sides of the border pixel are adjacent to WHITE pixels.

Recall that a chunk is the collection of pixels whose Morton codes lie between the Morton codes of two consecutive border pixels. For example, the chunk defined by border pixels 3 and 4 of Figure 5b has two nodes: the first is the level 0 (pixel-sized) node containing pixel 3. The second is the node labeled A. When pixel 4 is reached, a new border

pixel (pixel 5) is read off the list, and a new chunk is calculated. Note that every chunk begins with a level 0 node containing the border pixel at the beginning of that chunk. The final chunk defined by pixel 9 at the end of the image contains nodes 9, E, F, G, H, I, and J.

During processing, each node has three corner values associated with it, indicating the maximum level for the node containing the corresponding corner pixel. For node 1, the maximum corner values corresponding to the (NW, NE, SW) corners are (3, 0, 0). Pixel B has values (0, 0, 1). Pixel 5 has values (0, 2, 0).

The traversal process begins with the color table initially containing the value WHITE for all entries. Figure 6 follows the color table as it is updated by the various nodes. Pixel 1 is the first node used to update the color table. After pixel 1 is processed, the color table is modified as shown in Figure 6a (the initialized WHITE values are not shown for clarity). Entries updated by pixel 1 are underlined. The largest eastern and southern neighbors of pixel 1 are at level 0, so corresponding entries at level 0 in the color table are set to BLACK. After pixel 2 is processed, the color table is as shown in Figure 6b. Since the largest eastern neighbor of pixel 2 is a 2×2 quadrant that is a NE child of its parent, row 1, column NE in the color table is set to BLACK. Figure 6c shows the color table after processing pixel 3. Since node A is not a border pixel, the color table is accessed to determine node A's color. Node A is at level 0, and is a SE child. Thus, its value as indicated by the color table is BLACK. Node A does not update the table as it is a SE child. Figures 6e and 6f show the color table after processing pixels 4 and 5, respectively. Since pixel 5 has an eastern border and its maximum NE corner value is 2, the table entry (level = 2, quad=NE) is set to WHITE. Pixel B is a SW child, so it updates the table for its largest southern neighbor, of size 2×2. This entry in the table is set to BLACK,

as shown in Figure 6f. Pixel 6 is a SE quadrant so no update is performed. This process continues until the whole image has been processed or equivalently, the entire tree has been traversed. Figure 5c shows the final block decomposition for the image after all processing and merging has been performed.

2.7. Analysis of the Conversion Algorithm

As mentioned previously, both [Same80] and [Atki86] present conversion algorithms that are $O(n \cdot B)$. Samet's algorithm individually inserts B border pixels into a quadtree at level n, while Atkinson *et al*'s algorithm performs n passes through a node list of length B. Since Mark and Abel perform between two and four neighbor find operations at all n nodes of the quadtree, their algorithm is also $O(n \cdot B)$. Thus, all three of these earlier works have a depth factor (n) in their time complexity. The algorithm of [Webb84] is $O(B)$ for the initial construction of the pixel node lists, but it imposes a restriction by limiting the placement of the image within the image plane (the quadtree). Furthermore, the actual polygon fill phase is not mentioned. If one of the previously discussed algorithms is used, the overall time complexity of this algorithm would be $O(n \cdot B)$ in the worst case.

Our algorithm consists of two phases. First, a chaincode (or any other boundary representation) is processed, generating a list of linear quadtree nodes corresponding to the border pixels for the polygons. For a list of B border pixels, the time for this phase is $O(B)$. The list is then sorted by Morton code, requiring $O(B \log B)$ time. The output of the first phase is a sorted list of the border pixel nodes. An image of size $2^n \times 2^n$ has 2^{2n} pixels, giving an upper bound to the size of B as 2^{2n}. The depth of the corresponding quadtree is n since the image is decomposed into four quadrants at each step. Thus, $2n$ is an upper bound for $\log B$. When B is significantly smaller than the total number of

pixels in the corresponding raster image (which is usually the case), B log B is much less than $n \cdot B$. If the polygon border is exceedingly complex or winding, B can get large. For larger B values, $n \cdot B$ may be less than B log B by a constant factor. O(B log B) is asymptotically the same as O($n \cdot B$). However, our first step has been reduced to a simple sort while the other algorithms [Atki86, Mark85a, Same80, Webb84] have a more complex initial phase.

An important aspect of our new algorithm is that our filling phase can be performed in time O(B). By Hunter's theorem [Hunt79a], the number of nodes in the quadtree is O(B), since B is the length of the perimeter for the polygons represented. Our traversal process (named TRAVERSE in the pseudocode) simulates a traversal of the corresponding pointer-based quadtree, which contains O(B) nodes with each node processed once. Each node (i.e., each part of a chunk) is processed in constant time, since all subroutines other than TRAVERSE operate in constant time. Thus, the total time complexity of our filling phase is O(B).

2.8. Conclusions

This algorithm accomplishes three goals. First, it presents a different and potentially useful approach for converting border codes to a region quadtree. Second, it is the first boundary to region conversion algorithm (for linear or pointer-based quadtrees) that is O(B) for sorted data (border pixels) and O(B log B) for lists of unsorted border pixels. All previous algorithms [Atki86, Mark85a, Same80, Webb84] have worst case time complexity of O($n \cdot B$) for an image resolution of $2^n \times 2^n$. Third, it is the first linear quadtree algorithm that we are aware of that utilizes the algorithmic technique of simulating a pointer-based quadtree traversal during processing of the linear quadtree node list. While

we have engaged in discussions with researchers on the theoretical implications of such an approach, it is the first actual presentation of such an implementation. As the quadtree representation is utilized for more applications, a wide variety of algorithm techniques will prove increasingly useful to researchers and practitioners.

This algorithm could be easily modified to generate a pointer-based quadtree in $O(B \log B)$ time. However, the initial quadtree construction phase must not operate by inserting the border nodes into the quadtree as done in [Same80]. Instead, we would retain our initial phase in which we sort the border pixels by Morton code, and then modify our traversal phase to actually construct the pointer-based quadtree rather than simulate its traversal. Our algorithm can also be easily modified to operate in three dimensions. The color table must be expanded so as to store eight octants for each level, and the border codes require six bits (one for each face of the voxel). The table is still quite small, and the operation of the algorithm is essentially unchanged.

3. A FAST PARTICLE SYSTEM UPDATE ALGORITHM

3.1. Introduction and Previous Work

Our second algorithm deals with simulating a moving system of particles. The most widely publicized uses for particle system simulations are in the field of astrophysics. There are many unanswered questions about star cluster and galaxy evolution where particle simulations may be helpful [Appe85]. Celestial mechanics studies often use particle simulations [Hock88]. Fluids can be modeled as streams of interacting particles, so applications exist in fluid mechanics, plasma physics, and related fields [Hock88]. Molecular dynamics can be studied using particle simulations as well [Hock88]. Even animal behavior studies (such as flocking and schooling behavior [Reyn87]) could benefit from faster particle simulation algorithms.

Modeling a large system of moving particles requires two distinct steps. The first step calculates or approximates the interactions between particles. Direct calculation takes $O(n^2)$ time if every particle affects every other particle in the system. After all of the interactions have been calculated for each particle and all of the particles' new locations have been calculated, the next step of the simulation is to update the data structure that the particles are stored in. Historically, the particles were simply stored on lists, requiring only adjustments to each particle's position and velocity. Recent developments in this area [Appe85, Barn86, Carr87] have called for using tree data structures to store the system of particles for reasons explained below. The naive approach when using a tree is to construct a new tree for each cycle. To rebuild this tree requires that each particle be inserted into the tree at its new destination (modifying the tree as necessary). Each insertion is, on average, an $O(\log n)$ operation. It must be done for all n particles resulting in an overall

time complexity of $O(n \log n)$ for this approach.

Similar ideas were proposed by Appel [Appe85] and Barnes and Hut [Barn86] to approximate the particle interactions in a system using a tree data structure. Appel suggests four ways for speeding up a particle simulation. Three deal with coding aspects of the algorithm rather than the logic. The fourth way uses a better force calculation algorithm. He proposes to break the system into clumps. Each clump interacts with every other clump. Clumps that are too dense are further split into smaller clumps and so on, until accurate results can be obtained. This occurs when no two particles from two different clumps are closer together than two particles taken from the same clump. Appel's clumping algorithm has a time complexity of $O(n \log n)$. His coding improvements together with his clumping algorithm result in a faster implementation of a particle simulation than the brute force $O(n^2)$ algorithm. Appel's algorithm initially uses a k-d tree [Fink74]. The internal nodes of the tree store data about the clumps of particles. Particle motion results in an unbalanced tree after just a few cycles. Consequently, Appel's algorithm is hard to analyze and has no rigorous error bound.

Barnes and Hut [Barn86] implemented an $O(n \log n)$ algorithm using a PR octtree [Oren82, Same84b] which produces an image space decomposition. A PR octtree has regular subdivisions so a more rigorous analysis of the aggregated clumps is possible. The internal clump structure is not neglected in the center of mass calculations as in [Appe85]. Barnes and Hut rely on the internal nodes in a hierarchical data structure (an octtree) to store clumps of particles; however, they minimize force calculation errors that are introduced when the centers of mass of a group of particles are used instead of the individual particles. Both Appel, and Barnes and Hut concentrated on the interaction calculations and did little work on the tree update step. Both groups developed $O(n \log n)$ force calcu-

lating algorithms , so even when using the naive tree update algorithm, their entire particle simulation complexity would be $O(n \log n)$.

For gravitational particle systems, Greengard *et al* [Gree87, Carr87] have proposed a method to approximate the forces on each particle due to gravity in linear time with respect to the number of particles (n). The forces acting on any given particle can be divided into three parts: the component created by the particles close by, the component created by the particles far away, and an external component that is independent of the number of particles. The near particle interactions must each be computed separately for each pair of particles; however, the effect of a group of far particles can be modeled by a single point at the center of mass of the cluster. Carrier *et al* [Carr87] implement this idea with an octtree. However, they do not present an algorithm to update this octtree after all the particles have moved. Presumably, a new tree could be built costing $O(n \log n)$ time, and then the old tree could be discarded.

Reynolds [Reyn87] uses particle system simulation for a different application. He models the interactions among groups of animals like flocks of birds or schools of fish. Reynolds is more concerned with animating bird behavior than fast particle simulations, so all of his computations were done using naive $O(n^2)$ algorithms. The number of birds per flock that could be modeled was therefore limited. Using a faster calculation algorithm and a fast tree update algorithm, the maximum number of animals in a flock could be increased significantly.

The above works [Appe85, Barn86, Gree87, Carr87] all focus on bettering the first step of a particle simulation, i.e., the time required for the interaction computations, because historically, this was the expensive step of a simulation. The second step of a particle simulation requires that the data structure storing the particles also be updated efficiently.

Simply rebuilding the tree takes $O(n \log n)$ time. This does not change the overall time complexity for Appel, and Barnes and Hut, but for Greengard's work this update becomes the expensive step. If this procedure could be performed in linear time, then the entire particle simulation would be linear in the number of particles being simulated. In this chapter, we present an algorithm that updates the positions of a moving system of particles stored in an octtree in nearly linear time. The algorithm assumes that the particles are initially represented in an octtree. This assumption means that prior to the particle simulation, an octtree must be initialized, an $O(n \log n)$ task. However, this step is performed only one time. Typical particle simulations have from hundreds to millions of cycles depending on the application ([Appe85], [Hock88]). All particle simulation algorithms using tree structures have this initial building cost.

3.2. General Description of the Update Algorithm

The basic idea behind our algorithm is to update an existing octtree, rather than build a new one from scratch after every time step. Two passes (traversals) will update all of the particles in the octtree. Both passes perform the same operations; however, they each work on a different subset of the particles in the system. The first pass traverses the octtree in ascending Morton order. Recall that a Morton code is an address for a node formed by interleaving the bits of the coordinates of the upper left corner of the node. For octtrees, this is still true except that all three coordinates are used (x, y, z). During the first pass, all of particles that are traveling from a lower to a higher Morton code are moved (updated) with splitting and merging occurring as necessary. The rationale behind this is that only particles moving to nodes that have not been visited can be updated. Pass two traverses the octtree in descending Morton order, updating particles that are moving

from a higher to a lower Morton code. Both passes are postorder traversals of the particle octtree, but during the backward pass, the children of each node are traversed in reverse (descending) order. Since all of the particles must either move forward or backward in Morton code, each particle will be moved one time; the final tree will be accurate and up to date.

During each pass of the algorithm, particles may move into and out of nodes, and splitting and merging of nodes may occur. One of two events can happen when a particle moves. 1), it can move and still be in the same node, or 2) a particle can cross a node boundary. The first event requires no extra work since the tree is not modified.

To efficiently process the particles that cross node boundaries, a table (2D array) of lists is used. This table structure is similar in concept to the table used in our conversion algorithm. As particles cross out of nodes, they are put onto a list in the table. As nodes are processed, the appropriate lists are checked to see if the top particles on them belong within the current node. If so, they are added to the octtree at the current node, with splits occurring as necessary. In this way, the particles crossing node boundaries are temporarily moved out of the tree, and then subsequently replaced when their destination node is visited in the traversal. Merging may occur if particles pass out of nodes, thus leaving them less full.

The heart of this algorithm lies in the structure containing the set of lists. There is a list for each level of the octtree (the rows of the table), and a list for each face of a node: four sides, the front face, and the back face (Figure 7). The faces correspond to the columns of the lookup table. When a particle leaves a node at level L out of the front face, it is added to the end of the list labeled [L, front]. Particles get added to lists based on the level of their source node and which face they leave their source node through. The

algorithm requires that each of the lists in the table must remain sorted by Morton code, ascending for the forward pass and descending for the backward pass. This will insure that the topmost particle on each list will be added back into the storage tree before every particle below it on the list. Because of this, only the top of each list needs to be checked when looking for particles entering the current node. In general, when a particle is added to a list, its correct position is at the end of the list. This is due to the nature of a Morton order traversal, the fact that there are different lists for each one of the faces of the node, and that the particles are moved in two passes. If a particle does not belong at the end of a list, then it will be inserted into the list from the end until it reaches its proper position (Figure 8). Thus, at all times, the particles on the lists are in sorted order.

Particles travel up and down the lists in the table as the traversal progresses. When a link in the tree is traversed upward, that is, when a node's children have been processed, all particles still on the lists corresponding to these children are moved up to the lists at the current level (Figure 9). This action corresponds to a particle moving out of its source node's parent. The particle did not move into a sibling of its original node, so its destination must be later in the traversal. Therefore, the particle should temporarily be placed on a list at one level higher than its original node's level. Another way to think of this action is that the level of the current node of the traversal is increasing by one, so all particles not yet placed must follow along with the traversal until their destination node is encountered. When particles move up lists, they must be sorted onto the list at the next higher level. Again, this usually means just adding the particles to the end of the list. Before the current node has finished processing, the node's children are checked to see if merging can occur and the algorithm acts accordingly.

Similar to the action taken for uptraversals is the action taken for downtraversals.

Before a node's children are processed, all particles on lists at the parent node's level are moved to lists at the current level. Note that since the lists in the table are sorted, once a particle from the source list is found to reside at the end of the destination list, the rest of the source list can be concatenated onto the end of the destination list. So, as the traversal occurs, particles will be added to the lists and move within the table of lists. Note that particles never move from one *face* list to another. Only the level of the list they reside on will change. This movement can be thought of as a tracing of the path from the source node of the particle through the nodes of the tree until the particle's destination node is visited during the traversal. All particles on the lists are doing this concurrently.

3.3. Improvements to the Algorithm

Notice that during the forward pass, particles only leave an octant through the east, south, and back faces (equivalent to crossing an *active border* as defined in [Shaf86, Same85b]). Because of this, only three lists in the table will be used during the forward pass. Similarly, the backward pass will only add particles to the north, west, and front lists. Furthermore, after a pass is completed, no particles will remain in the lists. An obvious optimization is to only use three lists, and vary what they store during each pass. This optimization will save some space, and simplify the program code.

Traditionally, PR quadtree leaf nodes contain at most one point [Oren82, Same84b]. Whenever a second point enters a node, the node is decomposed. Since the table structure in this algorithm requires that the lists of particles remain sorted, improvements that help maintain this sorted order should be implemented. A *swap* is when a particle must be inserted into a list at some place other than the end. Each particle that the inserted particle goes in front of (above) constitutes one swap. Swapping occurs whenever necessary

to maintain sorted order. Geometrically, a swap occurs whenever two particles cross paths while also crossing a node boundary during the same time step. By allowing two particles to be stored in a leaf node instead of only one, most particles will cross paths within nodes, thus eliminating most of the swapping. Since particles that cross tend to be closer to each other than to any other particle, they will tend to reside in the same node. So, they will never be added to the table, and therefore, never cause a swap. Putting three particles in each node has not been explored; however, the added expense would probably not justify any potential savings [Nels86].

3.4. Update Algorithm Pseudocode

This section contains the Pascal-like pseudocode for the more complex update routines. The simpler routines are described within this section, but no pseudocode is given. The main procedure is called UPDATE_TREE. It is passed the number of iterations to be simulated and the root of the tree containing the particle system. UPDATE_TREE first calls INITIALIZE which initializes the lookup table pointers to NULL, and sets all global variables to their proper values.

The fast update algorithm's movement phase is one iteration of the *while* loop found in UPDATE_TREE. This loop calls TRAVERSE_TREE(*toptree*, FORW) for the forward pass through the tree of particles and TRAVERSE_TREE(*toptree*, BACK) for the backwards pass through the tree.

During each of the two traversals of the octtree that comprise one update cycle, six major steps occur at each node. First the lists at level $L + 1$, (where L is the level of the current node) are examined to see if their top particles belong in the current node. If so, then the whole list is merged with the level L list below it in the table. This is a down

traversal in the octtree. The second step checks to see if the current node has children, in which case the traversal of the octtree continues descending into each child node. After all children have been processed, the third step is to move all the particles that reside within the current node. The fourth step checks all of the lists in the table at the current level L for incoming particles to add to the current node. Only the top particle on each list is checked. The fifth step, the uptraversal, deals with the table of lists. It is the reverse of the downtraversal step. Before a node is completely processed and returns control to its parent, all particles still on lists at the current level $L-1$ are moved up a list because they have crossed over the current node's boundary (Figure 9). The last major step of the node processing is to check for possible merging.

TRAVERSE_TREE begins the processing of these six steps by looking into the lookup table (hereafter called S for storage) to see if there are any particles stored within it on lists at the current node level $L+1$ for all three faces (lists). Each column in the table corresponds to a face of the current node. For the forward pass, the FACE1 list is the east face. FACE2 is the south face, and FACE3 is the back face. For the backward pass, the lists correspond to the west, north, and front faces respectively. If particles are found on the S[$L+1$, *face_i*] lists, where *face_i* is FACE1, FACE2, or FACE3 that are destined for the current node, then DOWNTRAVERSAL(*node*, *face_i*, *pass*) is called. DOWNTRAVERSAL merges the particles on the S[$L+1$, *face_i*] list onto the list of particles stored on the S[L, *face_i*] list. This process happens for all three *face* lists. Note that when particles move down lists, they must be sorted onto the new list in ascending order during the forward pass and descending order during the backward pass. The simple procedure INSERT_SORT(*source_list*, *dest_list* $\uparrow$*.tail*, *pass*) adds the particle from the top of the *source_list* to the *dest_list* in the proper position. The search for the proper position

starts from the tail of the destination list, and works up to the head of the list. Note that if two lists are being merged together (as in DOWNTRAVERSAL and UPTRAVERSAL), and the top particle on the source list belongs on the bottom of the destination list, the procedure CONCAT_LIST(*source_list*, *dest_list*) is called, and the two lists are added together in constant time. CONCAT_LIST will return a NULL pointer in *source_list* and a pointer to the top of the merged list in *dest_list*. CONCAT_LIST works because the two lists being merged were in each already sorted.

After these three list checks are performed for FACE1, FACE2, and FACE3, the next step of TRAVERSE_TREE is to process the current node's children. This is done by recursively calling TRAVERSE_TREE for all eight child octants of the parent node. When the current node is a leaf node, its two slots (*objt*[1], *objt*[2]) are checked for particles moving in the direction of the current traversal. If this is the case, and the particle has not already been moved for this iteration, then MOVEMENT(*node*, *pass*, *pnt*) is called and the appropriate particle is moved. Note that *node* is the current node of traversal, *pass* is either FORW or BACK, and *pnt* is a pointer to the description of the particle in the node to be moved. The procedure PART_DEST(*pnt*) returns the Morton code of the particle's next position, its destination. This determines during which pass the particle will be moving.

The procedure MOVEMENT updates the particle's position, and if the current pass is FORW, then the object's MOVED field is set to TRUE. Otherwise, the particle's MOVED field is reset to FALSE. INQUAD(*node*, *particle*) determines whether *particle* is still within the current *node* after it has moved, or if not, which face it left through. INQUAD returns STILL, FACE1, FACE2, or FACE3. INQUAD tests each coordinate x, y, z for inclusion within the range of the current node. If the x test fails, the particle left

the node through FACE1, that is, the east or west face. Similarly, for the y coordinate and FACE2, and the z coordinate and FACE3, respectively. If the particle leaves the current node, MOVEMENT calls INSERT_SORT which adds the particle in the appropriate place on the appropriate list in the table.

After all particles in the current node have been moved, TRAVERSE_TREE makes three calls to CHECKLIST(*node*,*face_i*) to check the tops of all the lists (FACE1, FACE2, FACE3) at the current level L for incoming particles. These lists are checked until a particle is found not to be destined for the current node. Again, this holds because the lists are sorted in Morton order. The procedure ADDTONODE(*node*, *source_list*) removes the top particle from the source list and puts it into the current node. If there is no more room, i.e., a full level 0 node, RESOLVE_COLLISION is called to fix the problem. However, this rarely occurs. UPTRAVERSAL(*node*,*face_i*), the reverse of DOWNTRAVERSAL, is called three times moving all particles left on the lists one below the current level up to the current level lists. This step moves all particles leaving the current subtree up to the appropriate level list. Particles moving up lists have not yet reached their destination nodes. These nodes will be visited later in the traversal.

Lastly, the current node is checked for a possible merging of its children. Merging will occur if two or fewer particles exist in the children of the current node. Note that this value is two, because we allow each node to store at most two particles. MERGE(*node*) performs this simple operation.

The process described above occurs for every node in the octtree, and for both the forward and backward passes. After both passes have been performed, every particle will have moved once, all particles will have new destinations computed for them, and every particle's MOVED field will again be FALSE. The next iteration of the simulation is ready

to begin.

```
  { Declarations of types and global variables. }
type
  DIRTYPE = (FACE1, FACE2, FACE3, STILL);
  PASSTYPE = (FORW, BACK);
  OBJECTPTR = ↑OBJECT;
  OBJECT = record { record to hold one particle }
    dx, dy, dz : real; { velocity as 3 scalars }
    x, y, z : real; { current position of the particle }
    moved : boolean; { boolean flag for used for pass two }
    xyz : longint; { Morton code for current position }
    prev, next : OBJTPTR { pointers to other nodes in doubly linked list }
  end;
  LISTNODE = record { record to hold particles on lists in table }
    head, tail : OBJECTPTR ; { pointer to top and bottom of list }
  end;
  NODEPTR = ↑OCTNODE;
  OCTNODE = record { record holding one octtree node }
    level : integer; { level of the node in the tree }
    x1,y1,z1 : real; { coordinate for front upper left hand corner of node }
    objt: array [1..NUMSLOTS] of OBJECTPTR; { pointers to particles in the node }
    oct : array [0..7] of NODEPTR { pointers to children }
  end;
var { global variables }
  { table of lists for temporary storage of moving particles }
  S : array [0..MAXLEVEL, FACE1..FACE3] of LISTNODE;
  toptree : NODEPTR; { root node of particle octtree }

procedure MOVEMENT(node : NODEPTR; pass : PASSTYPE; pnt : OBJECT);
  { Moves the particle pnt in node to a new location or puts it into the table of lists. }
var dir : DIRTYPE;
begin
  if pass = FORW then pnt↑.moved := TRUE;
  pnt↑.x := pnt↑.x + pnt↑.dx; pnt↑.y := pnt↑.y + pnt↑.dy; pnt↑.z := pnt↑.z + pnt↑.dz;
  pnt↑.xyz := COMPUTE_XYZ(pnt↑.x, pnt↑.y, pnt↑.z);
  dir := INQUAD(node, pnt);
  if dir <> still then { if particle left the current node }
    begin
      INSERT_SORT(pnt, S[dir, node↑.level]↑.tail, pass);
      pnt := NULL { Remove particle from current node. }
    end
end;

procedure DOWNTRAVERSAL(node : NODEPTR; dir : DIRTYPE;
```

```
                pass : PASSTYPE);
  { Merges appropriate source list particles with the destination list in sorted order. The
    source list is at a higher level than the destination list. }
var source_list : OBJECTPTR;
  dest_list : LISTNODE;
begin
  source_list := S[dir, node↑.level + 1]↑.head;
  dest_list := S[dir, node↑.level];
  { While the source particle list is not empty }
  while source_list <> NULL do
    begin
      if source_list ↑.xyz > dest_list ↑.tail then CONCAT_LIST(source_list, dest_list)
      else begin
          INSERT_SORT(source_list, S[dir, node ↑.level]↑.tail, pass);
          source_list := source_list↑.next
        end
    end;
  S[dir, node↑.level]↑.head := source_list
end;

procedure UPTRAVERSAL(node : NODEPTR; dir : DIRTYPE; pass : PASSTYPE);
  { Merges appropriate source list particles with the destination list in sorted order. The
    source list is at a higher level than the destination list. }
var source_list : OBJECTPTR;
  dest_list : LISTNODE;
begin
  source_list := S[dir, node↑.level - 1]↑.head;
  dest_list := S[dir, node↑.level];
  { While the source particle list is not empty }
  while source_list) <> NULL do
    begin
      if source_list ↑.xyz < dest_list ↑.tail then CONCAT_LIST(source_list, dest_list)
      else begin
          INSERT_SORT(source_list, S[dir, node ↑.level]↑.tail, pass);
          source_list := source_list↑.next
        end
    end;
  S[dir, node↑.level]↑.head := source_list
end;

procedure CHECKLIST(node : NODEPTR; dir : DIRTYPE);
{ Adds particles from the lists bound for the current node. }
var source_list : OBJECTPTR;
begin
  source_list := S[dir, node↑.level]↑.head;
  while (source_list <> NULL) and (INQUAD(node, source_list) = STILL) do
    begin
```

```
      if (node↑.objt[0] = NULL) or (node↑.objt[1] = NULL) or (node↑.level > 0) then
        ADDTONODE(node, source_list)
      else RESOLVE_COLLISION { full node at level 0 }
      source_list := source_list↑.next
    end;
  S[dir, node↑.level]↑.head := source_list
end;

procedure TRAVERSE_TREE(node : OCTNODE; pass : PASSTYPE);
  { Performs the traversal of the particle tree doing six constant time operations at each
    node. }
var child, pnum : integer;
begin
  { Checks for particles belonging in the current node from higher leveled lists. }
  for F in {FACE1, FACE2, FACE3 } do
    if INQUAD(node, S[F, node↑.level + 1]↑.head)
      then DOWNTRAVERSAL(node, F, pass);
  if not LEAF(node) then { if current node has children }
    if pass = FORW then
      { Traverse children in ascending Morton order. }
      for child := 0 to 7 do TRAVERSE_TREE(node↑.oct[child]);
    else
      { Traverse children in descending Morton order. }
      for child := 7 to 0 do TRAVERSE_TREE(node↑.oct[child]);
  { Move all particles in the current node that haven't moved yet. }
  for pnum := 1 to NUMSLOTS do
    if (node↑.objt[pnum] <> NULL then
      if ((node↑.objt[pnum]↑.xyz <= PART_DEST(node↑.objt[pnum])) and
          (pass = FORW)) or
      (node↑.objt[pnum]↑.xyz > PART_DEST(node↑.objt[pnum])) and
          (pass = BACK)) then
        if node↑.objt[pnum]↑.moved then
          node↑.objt[pnum]↑.moved = FALSE
        else MOVEMENT(node, pass, node↑.objt[pnum]);
  { Check for incoming particles from the current level lists. }
  CHECKLIST(node, FACE1);
  CHECKLIST(node, FACE2);
  CHECKLIST(node, FACE3);
  { Move all particles not placed in the current node to higher leveled lists. }
  for F in {FACE1, FACE2, FACE3 } do
    if (node↑.level > 0) and (S[F, node↑.level − 1]↑.head <> NULL) then
    UPTRAVERSAL(node, F, pass);
  if not LEAF(node) then MERGE(node)
end;

procedure UPDATE(maxit : integer; toptree : NODEPTR);
var count : integer;
```

```
begin
  INITIALIZE;
  for count := 1 to maxit do
    begin { Loop for one time step (iteration). }
      TRAVERSE_TREE(toptree, FORW);
      TRAVERSE_TREE(toptree, BACK);
    end
end;
```

3.5. Example of the Algorithm

The following example is a walkthrough of one iteration of the update algorithm. This example is presented in two dimensions for clarity and ease of visualization. Figure 10a shows the space containing a particle system. Figure 10b shows the corresponding PR quadtree along with arrows representing each particle's velocity (final destination). Figure 10c shows the numbered leaf nodes of the initial quadtree. Figure 11 will be referenced throughout the simulation. It is an illustration of the table of lists. Initially, this table is empty.

The update phase consists of two nearly identical traversals of the tree. Note from Figure 10c, particles A, B, C, D, E, F, G, and H will move during the forward traversal while particles I and J will move during the backward traversal. Recall that as a traversal progresses, six major jobs are done for each node:

1) Check for DOWNTRAVERSALS of particles in the table of lists.
2) If the current node has children, visit each child.
3) MOVE the particles in the current node.
4) Check the table (3 lists) for particles entering the current node.
5) Perform any necessary UPTRAVERSALS.
6) MERGE the node if necessary.

During the forward traversal through the tree, the first node visited will be the root node. Since the lists are empty, no downtraversals occur. Next, the children of the

root are visited in ascending Morton order. TRAVERSE_TREE is recursively called on the northwest child. Nothing happens here either so again, a recursive call is made on the northwest child: node 1 of the tree. Now, the six steps are performed for this node. Figure 11a shows the lookup table after node 1 has been processed. Particle A has been removed from the tree, and now resides on a level 2 FACE1 (east) list. This happened because particle A's source node was at level 2, and particle A passed out of it through the eastern edge. No other particles were added to the node, and no splitting or merging occurred. Merging can only occur after the processing of a SW node. Before this postorder traversal continues with nodes 2, 3, 4, and 5, their parent node checks for downtraversals. Since the current level is 2, level 2 lists in S are examined. Particle A is found on list S[2, FACE1], but it does not belong in this subtree, so no downtraversal occurs. After node 4 has been processed, the table is as shown in Figure 11b. Particles B, C, and D have been added to lists; they are all in transit to their destination nodes. Note that when particles C and D were added to the lists, they had to be sorted to their proper positions because particle C was added to the list first, but particle D will be leaving first. This is an illustration of a swap. Node 5's interesting step is the call to CHECKLIST. Particle B's destination is node 5, so particle B is removed from the table and added back into the quadtree at node 5. When the parent of nodes 2, 3, 4, and 5 calls UPTRAVERSAL, all particles still in the table on level 1 lists are pushed up a list Simply put, all particles still on level L-1 lists need to move up to level L lists, because they did not move to a sibling of their source node. The uptraversal happens again after nodes 6 and 7 have been processed. Merging occurs of nodes 2, 3, 4, and 5 after node 5 finishes, and then again for the resulting node from above and nodes 1, 6, and 7. This second merge occurs after node 7 is processed. The northwest corner is now all one node containing particle B. The

state of the lookup table is now as shown in Figure 11c. When the traversal reaches node 8, both particles C and D get added to node 8. Since particle E has already been moved, and is still within node 8, node 8 must be split into four children. Next, particle A adds into node 10. Figure 11d shows the lists after node 11 has been processed, but before the parent of 8, 9, 10, and 11 has called UPTRAVERSAL. Figure 11e shows the lists after this call has been made. Since particle F is moved during the forward pass, and particle J is not moved until the backwards pass, the fact that these two particles cross is insignificant.

Node 12 has two particles in it that cross; however, neither one leaves the current node. The particles need not be stored in the table for this case. This illustrates the savings from allowing two particles in a node. After processing, particle F resides in node 13. The lists are all empty. The tree has now been updated as shown in Figure 12. Note that particles I and J have not moved yet.

The backward pass is ready to begin. Particles moved during the forward pass will be ignored by this pass. This is the purpose of the boolean MOVED field in each object's record. Note that whenever this pass encounters a particle that has already moved, it resets the particle's MOVED field to FALSE before continuing on. This pass begins with the root node as did the forward pass. For this traversal, the children of the current node will be traversed in the reverse order, that is, in descending order by Morton code. Particles I and J will be moved during this traversal. Particle I will cause node 12 to split. Particle J just moves into node 11, causing no change to the structure of the quadtree. At the end of this traversal, all the particles will have moved, the tree will need no splitting or merging, and the lookup table will be empty. Everything will be ready for the next iteration to begin. The final quadtree is illustrated in Figure 13.

3.6. Analysis of the Algorithm

The fast update algorithm is difficult to analyze. The processing that goes on for each node of the tree is deceiving. The following section will analyze the update algorithm, and compare the analysis with that of the naive algorithm which always rebuilds the tree. Note that a worst case analysis of the update algorithm would show it to be far worse than it ever would be in practice. Furthermore, for typical uses (a particle simulation of a gravitational system), the node processing step will be constant.

First, a quick analysis of the naive algorithm is in order. Its initial cost is the tree construction: $O(n \log n)$ for n particles. This result is derived from the fact that an octtree of depth d can store at most 2^{3d} particles. Letting $n = 2^{3d}$, we see that $3d = \log n$. Since a tree insert is an $O(d)$ operation, n inserts will require $O(n \cdot d)$ time which is equivalent to $O(n \log n)$ time.

Next, after every iteration, a new tree will be constructed when all of the particles will have moved. Since n particles must be reinserted into the new tree, the time complexity for this step is the same as the initial construction step, so the total time complexity for this naive algorithm is $O(n \log n)$. Rebuilding the tree after every iteration is a clear, easy method for updating the particle system. However, for large n, this construction can take a prohibitively long time as demonstrated in the experiments described below.

Our update algorithm performs two traversals of the entire particle tree. From [Shaf86, Nels86], a PR quadtree containing n particles requires $O(n)$ nodes, so this implies that the time operations performed at each node must be constant for the whole algorithm to be $O(n)$.

Recall that six actions take place for each node of the tree. The first one is to perform any downtraversals necessary on the particles in the appropriate lists. Already

there is cause for alarm. INSERT_SORT performs sorted insertions into these lists, and the time required could be as bad as $O(n)$. INSERT_SORT also merges sorted lists, another $O(n)$ operation. These problems will be dealt with later. For now, we shall ignore the fact that INSERT_SORT is not a constant time operating procedure, and examine the other five node processing steps. The second step of node processing is the postorder traversal of the tree. If a node is internal, all paths to its children must be followed. Therefore, all the nodes in the tree are each visited one time during a traversal of the tree. The third step updates the position of the particle in the node, if it exists. If the particle does not leave the current node, there is no problem; however, if the particle does, it is inserted onto a list in the correct position by INSERT_SORT. Again, we shall analyze this procedure later. The fourth step is to check the lists for incoming particles. Since the lists are sorted, only the top element on each list needs to be checked. If the top particle does not belong in the current node, the rest of the list can be overlooked. If the top particle does belong, then the next particle on the list must be checked. Potentially, the entire list could be traversed. Yet, each particle can be in the lists only once per time step, so the maximum number of adds that can be performed to the tree from the lists is $O(n)$ amortized over all the nodes of the tree. Clearly, this does not affect the overall complexity of the algorithm. The fifth step is the reverse of the downtraversal, the uptraversal. Particles are moved up to higher level lists in the table. This step is identical in complexity to the downtraversal step. Its analysis depends only on INSERT_SORT. The final step is to determine if merging is possible. At most eight nodes can be checked for a possible merge. Therefore, merging is a constant time operation. Merging can only happen once per node; in fact, merging may only occur after a SW node has just been processed. So, by ignoring the INSERT_SORT procedure, we have shown that only constant time operations are performed at each node

of the tree.

Intuitively, INSERT_SORT requires in the worst case $O(n)$ time, and since it is called by DOWNTRAVERSAL, UPTRAVERSAL, and MOVEMENT, it seems unlikely that the node processing can be performed in constant time. In fact, it seems that the worst case time for this algorithm is $O(n^2)$. Clearly, particles moving out of their source nodes can cross through an arbitrary number of nodes before reaching their destination (given a large enough time step). However, each particle can only be examined on the average once per level when traveling from its source node to its destination node by way of the common ancestor node. Since the maximum level of the whole tree is $O(\log n)$, our algorithm has a worst case time complexity of only $O(n \log n)$.

Consider a particle that starts in a level 0 node, but travels out of a large quadrant of the image and stops in a nearby pixel node (Figure 14). This particle would move around in the lists for a long time ($O(d)$ lists for a tree of depth d), before it finally reaches its destination node. The table of lists actually solves this problem by concatenating the movement of the particles' motions on the lists together as single operations. If a small enough time step is used (so particles could only move at most one pixel per cycle), the number of swaps that can occur when a particle is added to a list is bounded. Therefore, the lists can be kept in sorted order with a constant number of swaps. All additions of particles to lists (as with MOVEMENT) would occur at the ends of destination lists. Furthermore, whenever two lists are merged together (as in UPTRAVERSAL and DOWNTRAVERSAL), the source list would simply be added onto the end of the destination list by CONCAT_LIST (plus some constant time swapping by INSERT_SORT if necessary). Since both lists are already sorted, this is a constant time merge. Thus, for small velocities, INSERT_SORT only adds particles to the ends of lists. This implies that the work done finding every

particle's destination node can be done in big steps by CONCAT_LIST (whole list moves) at every node for the UPTRAVERSAL calls and for the DOWNTRAVERSAL calls instead of moving every particle individually. Very little swapping occurs when particles (or lists of particles) are added to other lists of particles. Therefore, by keeping the time step small, node processing can be done in constant time, and the whole algorithm has $O(n)$ time complexity.

A swap in a list occurs when two particles cross paths during one time step, as well as a node border. The nature of Morton order combined with the fact that the insertion sort begins at the most likely place for the particle, the end of the list, means that particles are almost always added to the ends of lists and very little sorting (swapping) is occurring. Our empirical data shows that as the velocity of the particles is allowed to increase, more and more swaps occur. These extra swaps correspond to sorting the first few particles from the source list to their correct position on the destination list before adding the bulk of the source list to the bottom of the destination list. If particles can move further per time step, there should be more crossings of particles and thus, more swaps. Our experimental data supports this claim. Data was gathered for maximum velocity = 1, 5, 10, 50, and 200 pixels per time step (Figures 15 and 16).

Although the number of swaps blows up rapidly for the higher velocities, the times for the simulation still follow a roughly linear trend, because swapping is such a cheap operation. For velocities up to 1 pixel per time step, the number of swaps that can occur per particle is bounded. At most one particle can swap up the list one position per list concatenation. In this case, the simulation times will be truly linear with respect to the number of particles (Figure 16). For other particle system applications that might have more frequent crossings (swapping), our algorithm would still be an improvement over the

naive one, but it becomes difficult to analyze rigorously. Our data shows that as velocities increase to unrealistic values like 200 pixels per time step, the cost of our node processing approaches $O(n \log n)$, and our algorithm's operating time approaches that of the naive update algorithm.

In reality, swaps rarely happen in a particle simulation of a gravitational system. If swaps do occur, then the time step of the simulation is too large, because particles do not cross paths during the same time step in actual gravity-based particle systems. Particles either collide or rotate around past each other. Swapping rarely occurs in some other types of particle systems as well. For example, flocks of birds do not weave paths among each other while flying. For correlated motion like fluid flow, crossing of particles (swapping) is more prevalent, but not dominating since all the particles are moving in the same general direction.

3.7. Experimental Results

We performed a series of experiments on a Macintosh II equipped with 10M of internal memory, a 68020 processor with a 16 megahertz clock, and running Apple Unix (A/UX). All of our programming was done in C. Our experiments can be divided into two sets: one using the naive update algorithm, and one using our fast update algorithm. The results of the experiments were compared against each other and Carrier and Greengard's results [Carr87] for his force calculation algorithm. Note that Carrier and Greengard implemented their algorithm on a VAX 8600 which is approximately four times faster than a Macintosh II.

For our experiments, we gathered times for six iterations of a particle simulation. We simulated random (non gravitational) motion of particles in a 1024 pixel square

area. Due to memory restrictions, we conducted our experiments in two-dimensional space. Whenever a particle hit a wall, it simply bounced off. Each particle started with a random position in the region, and a random velocity between 0.01 and MAXVELOCITY (in pixels). For each algorithm (naive and ours), we used five different maximum velocities (1, 5, 10, 50, 200 pixels per time step), and ten different numbers of particles between 100 and 51,200. Information gathered for each run includes the total time for all six iterations and the total number of swaps occurring (for the new algorithm). Figure 15 presents a table of the experimental results. For the naive algorithm, only one series of simulations is reported, because the naive algorithm is independent of the velocity of the particles. Figure 16 is a graph of some of the gathered data. These graphs also show the naive algorithm performance ($O(n \log n)$) and the performance of a linear algorithm. The intermediate curves depict our update algorithm for different maximum velocities (equivalently, time steps). The data for the naive algorithm exhibited $O(n \log n)$ growth for the total times, while our new algorithm exhibited nearly linear growth for the total times at reasonable velocities. As the maximum velocity for each particle approached 200, our algorithm became non-linear; however, it was still noticeably less costly than the naive algorithm.

Carrier and Greengard [Carr87] approximated the force interactions among up to 25,600 particles using their linear algorithm. Comparing this to our simulations, we find that if our update algorithm were to be used with their calculation algorithm, there would be a 10% to 15% increase in total simulation time. The naive algorithm would increase total simulation times by 50% or more. Presumably, Carrier and Greengard's slowest step (for a gravitational particle simulation) was their $O(n \log n)$ tree update. Using our update algorithm, this step becomes almost trivial when compared to the processing time required to compute the the particle interactions.

3.8. Conclusions

The fast update algorithm greatly reduces the running time necessary for particle simulations as compared to the naive algorithm. Even though it is not truly a linear time algorithm in the worst case, experimental data shows that for random motion of particles with velocities up to ten pixels per cycle, there was little divergence from a linear time complexity. The number of crossings (swaps) blows up for large time steps (velocities) and for high densities of particles; however, these are not typical simulation conditions. Our simulations of 100 to 51,200 particles in a 1024 by 1024 pixel plane showed only slight divergence from linearity in the times gathered when low maximum velocities (pixels per cycle) were used, and a linear time curve if the maximum velocity was kept to one pixel per cycle. As the maximum velocity increased into the hundreds of pixels per time step, our algorithm diverged from the linear algorithm and approached the naive one: $O(n \log n)$.

The algorithm presented here uses a storage structure, the table of lists, to efficiently move particles within an octtree. For particle simulations, a tree structure is needed to aggregate the centers of mass of clumps of particles. This idea could easily be incorporated into our fast update algorithm. As the backward pass takes place, centers of mass for each node could be calculated. So, this algorithm would combine well with Carrier and Greengard's linear force approximation algorithm [Carr87, Gree87]. Astrophysicists have long been plagued by the slow running times of simulations, even on super computers. Now, with the improvements in both phases of computer simulation algorithms, particle simulations can be run in a fraction of the time.

4. CONCLUSIONS AND FUTURE WORK

Each of the algorithms presented uses a table driven traversal to yield new time bounds. The conversion algorithm maintains a table of color values to help determine the colors of absent regions of an image while the fast update algorithm uses a table of doubly linked lists to temporarily store particles as they move through space. Both algorithms implement traversals of a quadtree (octtree) with constant or nearly constant time operations occurring at every node in the tree. The conversion algorithm is unique in this sense, because it does not actually traverse a pointer-based tree; rather, it simulates a pointer-based tree traversal. The tree supposedly traversed is, in fact, never created. The filling phase of the conversion algorithm has linear time complexity; however, it requires a sorted list of the border pixels as input. The conversion algorithm could use either linear or pointer-based trees, and is general in that any boundary representation that can produce a set of border pixels can be converted into a region quadtree representation of the enclosed area.

The fast update algorithm could also be implemented using pointer-based quadtrees or linear quadtrees; however, pointer-based quadtrees are more appropriate. Using linear quadtrees, internal nodes would have to be kept to store the centers of mass for the subtrees. Furthermore, particle simulations need to be fast; they are traditionally kept in core for efficient processing. Recall that pointer-based quadtrees typically perform better than linear quadtrees in core memory. The fast update algorithm would work for any simulation of a particle system: fluid flow, flock modeling, etc. Particle crossings may become a limiting factor when using this algorithm with very dense systems or with large time step simulations; however, it operates in nearly linear fashion for typical particle system simulation applications. Typical simulations use small time steps for increased accuracy.

Furthermore, most particle systems in nature tend to be sparse; dense systems are infrequent. A galaxy or solar system is almost all empty space; crossings rarely occur. The fast update algorithm would be applicable in this case. In gravitational particle systems, particles do not cross paths during a single time step. Again, the update algorithm would be useful. In fact, many applications in astrophysics are well-suited to be modeled by a particle system simulation using our update algorithm.

Modeling animal behavior could also make use of a particle system simulation [Reyn87] that uses our update algorithm. Animating flocks of birds requires a fast algorithm for realism. Although flocks of birds tend to be dense, they rarely cross paths during a single time step because this most likely means a mid-air collision. Therefore, our update algorithm would be useful for this kind of model as well.

Ideas for boundary to quadtree conversion algorithms have nearly been exhausted, whereas particle system simulations is a rapidly growing field. For particle simulations, our idea of updating the tree is completely new. Until now, researchers have concentrated on computing the particle interactions faster. This work has been so successful that now attention must turn to updating the tree rapidly. Future investigations could include updating dense subtrees more often than the entire tree [Appe85]. Clumps of particles is what causes the need for a small time step and thus many iterations. Perhaps, only denser areas of the particle tree need to have their interacting forces computed at every step and sparser areas can wait longer before force recalculation becomes necessary. The update algorithm lends itself well to a particle simulation, but perhaps there are other applications that use trees and could also benefit from updating an existing tree rather than creating a new one. Another area of investigation is to increase the number of particles allowed in a single node. Carrier and Greengard [Carr87, Gree87] put up to 30 particles in a node;

however, [Nels86] indicates that increases beyond 2 will not likely have any benefit. Parallel processing may lend itself nicely to a particle system simulation. When each particle can by governed by its own processor, simulation times will decrease dramatically.

The real work still to be done in this area involves implementing the fast update algorithm in actual simulations. Several possibilities include a model for a gravitational system using Carrier and Greengard's work [Carr87, Gree87]. Their linear interaction calculations combined with this approximately linear update could significantly decrease the running times for large particle simulations. More particles could be put into the system, or current size simulations could run longer for the same cost. Flock modeling using Reynolds' ideas [Reyn87] and this algorithm could produce larger, and perhaps, more realistic animal behavior. In fact, any particle system application (that uses tree storage structures) looking for an increase in speed could benefit from this nearly linear update algorithm.

5. REFERENCES

1. [Abel83] D.J. Abel and J.L. Smith, A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Computer Vision, Graphics, and Image Processing 24*, 1(October 1983), 1-13.

2. [Appe85] A. Appel, An efficient program for many-body simulation, *SIAM Journal of Science and Statistical Computing 6*, 1(January 1985), 85-103.

3. [Atki86] H.H. Atkinson, I. Gargantini, and T.R.S. Walsh, Filling by quadrants or octants, *Computer Vision, Graphics, and Image Processing 33*, 2(February 1986), 138-155.

4. [Barn86] J. Barnes and P. Hut, A hierarchical O(n log n) force-calculation algorithm, *Nature 324*, 6096(December 1986), 1-3.

5. [Carr87] J. Carrier, L. Greengard and V. Rokhlin, A fast adaptive multipole algorithm for particle simulations, Research Report YALEU/DCS/RR-496, January, 1987.

6. [Dyer80] C. Dyer, A. Rosenfield, and H. Samet, Region representation: Boundary codes from quadtrees, *Communications of the ACM 23*, 3(March 1980), 171-179.

7. [Fink74] R. Finkel and J. Bentley, Quadtrees: a data structure for retrieval on composite keys, *Acta Informatica 4*, 1(April 1974), 1-9.

8. [Free74] H. Freeman, Computer processing of line-drawing images, *ACM Computing Surveys 6*, (March 1974), 57-97.

9. [Garg82] I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

10. [Garg84] I. Gargantini and H.H. Atkinson, Linear quadtrees: a blocking technique for contour filling, *Pattern Recognition 17*, 3(May 1984), 285-293.

11. [Gree87] L. Greengard, and V. Rokhlin, A Fast Algorithm for Particle Simulations, *Journal of Computational Physics 73*, 2(December 1987), 325-348.

12. [Hock88] R. Hockney and J. Eastwood, *Computer Simulation using Particles*, McGraw-Hill, New York, 1988.

13. [Hunt79a] G. Hunter and K. Steiglitz, Operations on images using quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.

14. [Hunt79b] G. Hunter and K. Steiglitz, Linear transformation of pictures represented by quadtrees, *Computer Graphics and Image Processing 10*, (July 1979), 289-296.

15. [Klin79] A. Klinger and M.L. Rhodes, Organization and access of image data by areas, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 1(January 1979), 50-60.

16. [Mark85a] D. Mark and D. Abel, Linear quadtrees from vector representations of polygons, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 3(May 1985), 344-349.

17. [Mark85b] D. Mark and J. Lauzon, Linear quadtrees for geographic information systems, *Proc. Int. Symp. Spatial Data Handling*, Zurich, Switzerland, August 1984, 412-430.

18. [Mark85c] D. Mark and D. Abel, Linear quadtrees from vector representations: Polygon to quadtree conversion, CSIRONET Tech. Rep. No. 18, Canberra, Australia. Not published.

19. [Meag82] D. Meagher, Geometric modeling using octtree encoding, *Computer Graphics and Image Processing 19*, 2(June 1982), 129-147.

20. [Mill68] R. Miller and K. Prendergast, Stellar dynamics in a discrete phase space, *Astrophysical Journal 151*, (January 1968), 699.

21. [Mill70] R. Miller, K. Prendergast, and W. Quirk, Numerical experiments on spiral structure, *Astrophysical Journal 161*, (January 1970), 903-916.

22. [Mort66] G. Morton, A computer oriented geodetic database an a new technique in file sequencing, IBM Ltd., Ottawa, Canada, 1966. Not published.

23. [Nels86] R. Nelson and H. Samet, A population analysis of quadtrees with variable node size, TR-1672, Computer Science Department, University of Maryland, College Park, MD, June 1986.

24. [Oren82] J. Orenstein, Multidimensional tries used for associative searching, *Information Processing Letters 14*, (June 1982), 150-157.

25. [Peuq84] D. Peuquet, A conceptual framework and comparison of spatial data models, *Cartographica 21*, 4(Winter 1984), 66-113.

26. [Reyn87] C. Reynolds, Flocks, herds, and schools: A distributed behavior model, *Computer Graphics 21*, 4(July 1987) 25-34.

27. [Same80] H. Samet, Region representation: quadtrees from boundary codes, *Communications of the ACM 23*, 3(March 1980), 163-170.

28. [Same82] H. Samet, Neighbor finding techniques for images represented by quadtrees, *Computer Graphics and Image Processing 18*, (January 1982), 37-57.

29. [Same84a] H. Samet, A. Rosenfield, C. Shaffer, and R. Webber, Use of hierarchical data structures in geographic information systems, *Proc. Int. Symp. Spatial Data Handling*, Zurich, Switzerland, August 1984, 392-411.

30. [Same84b] H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

31. [Same85a] H. Samet and C. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 6(November 1985),717-720.

32. [Same85b] H. Samet and M. Tamminen, Computing geometric properties of images represented by linear quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 2(March 1985), 229-240.

33. [Same89a] H. Samet, *Design and Analysis of Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*, Addison-Wesley, Reading, MA, 1989.

34. [Same89b] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1989.

35. [Shaf86] C.A. Shaffer, *Application of Alternative Quadtree Representations*, Ph.D. dissertation, TR-1672, Computer Science Department, University of Maryland, College Park, MD, June 1986.

36. [Shaf87a] C.A. Shaffer, H. Samet, and R.C. Nelson, QUILT: A geographic information system based on quadtrees, Computer Science TR-1885, University of Maryland, College Park, MD, July 1987.

37. [Shaf87b] C.A. Shaffer and H. Samet, Optimal quadtree construction algorithms, *Computer Vision, Graphics, and Image Processing 37*, 3(March 1987), 402-419.

38. [Shaf88] C.A. Shaffer and H. Samet, Set operations for unaligned linear quadtrees, to appear in *Computer Vision, Graphics, and Image Processing*. Also, Department of Computer Science TR 88-31, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, September 1988.

39. [Webb84] R.E. Webber, *Analysis of Quadtree Algorithms*, Ph.D. dissertation, TR-1376, Computer Science Department, University of Maryland, College Park, MD, March 1984.

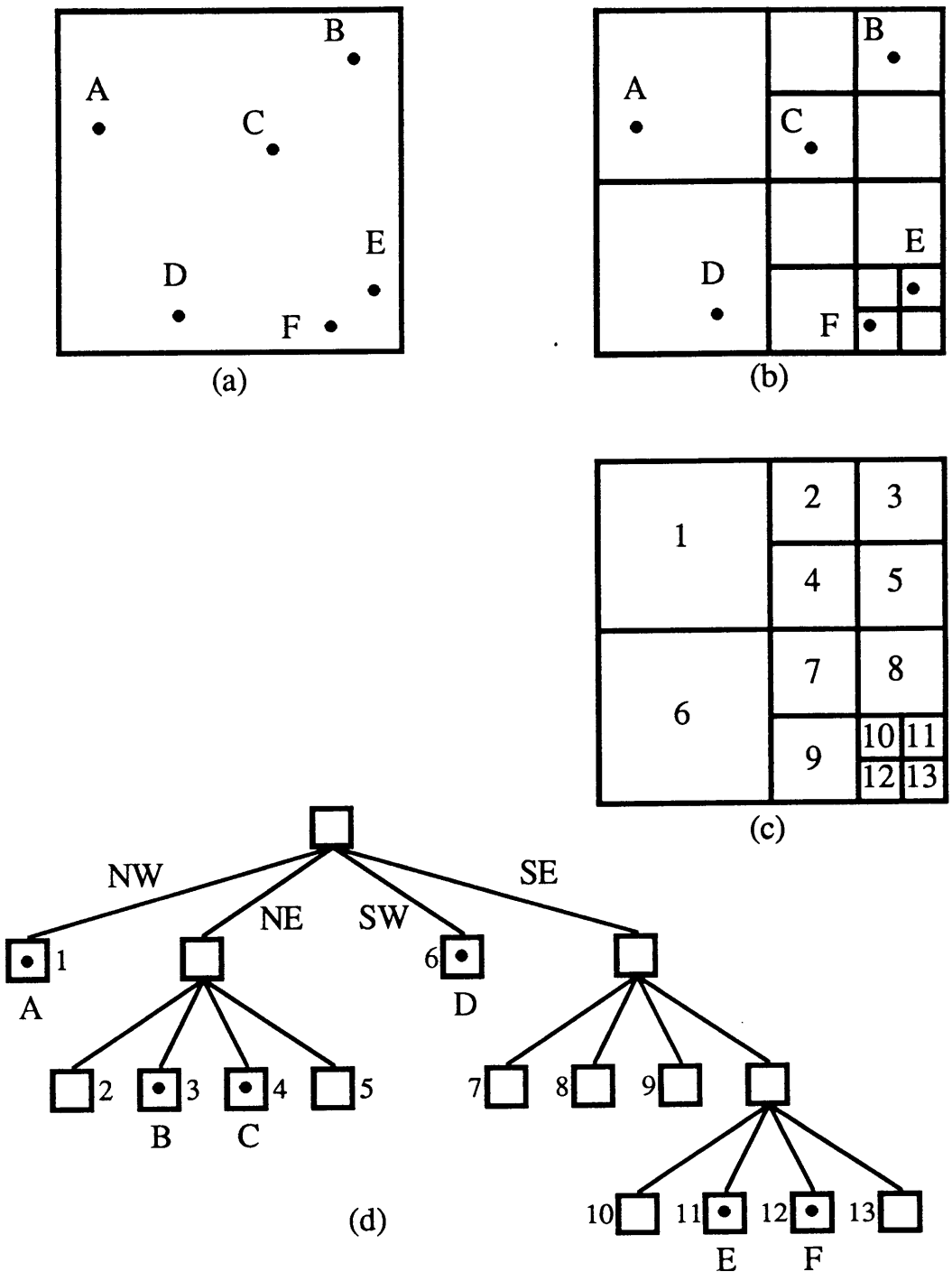


Figure 1. Typical Point Region quadtree. (a) Image space. (b) Particles in quadtree nodes. (c) Quadtree nodes labeled. (d) Pointer based quadtree structure.

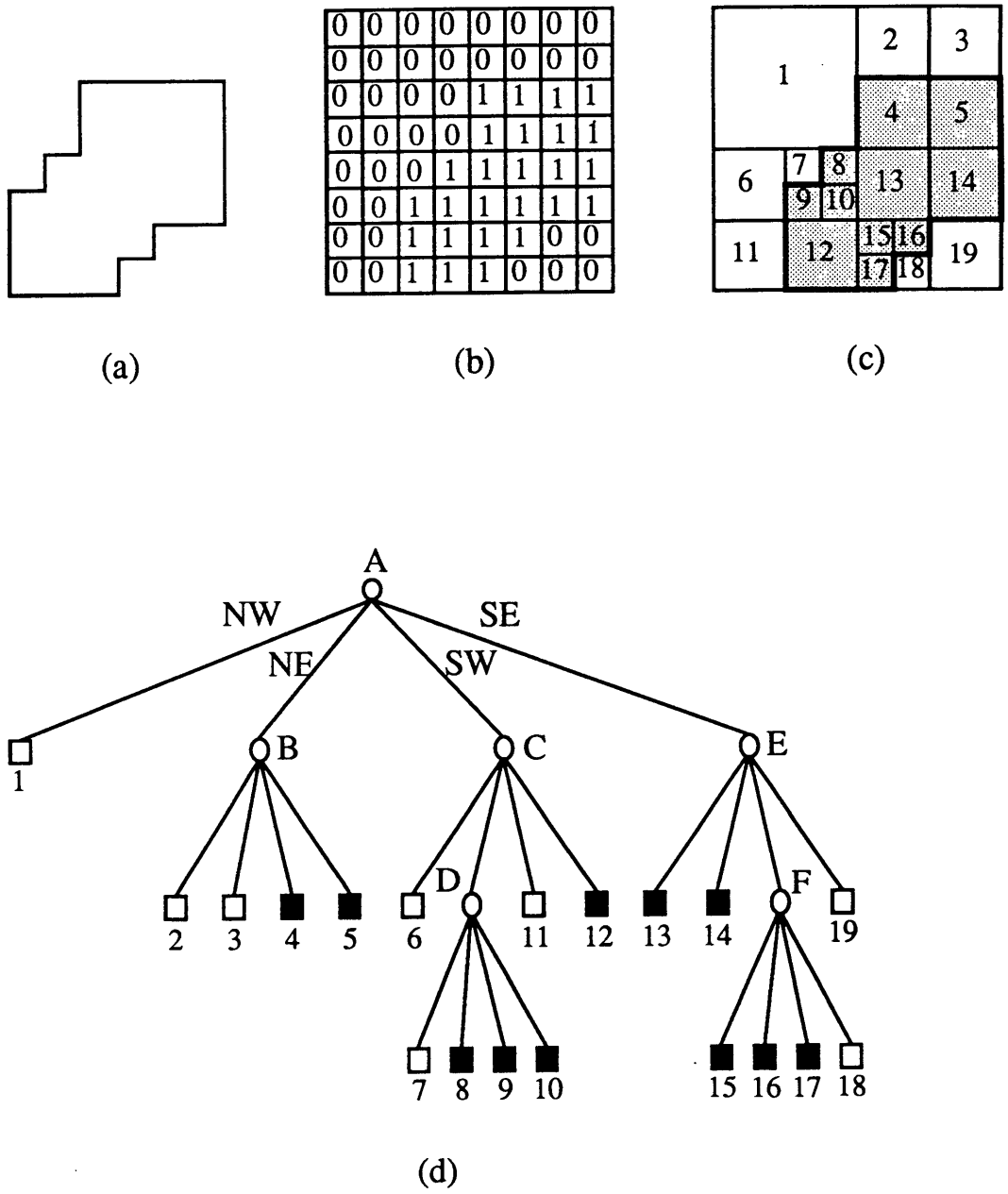


Figure 2. A region, its binary array, its maximal blocks, and the final quadtree. (a) Region. (b) Binary array. (c) Block decomposition of the region in (a). Blocks in the region are shaded. (d) Quadtree representation of the blocks in (c).

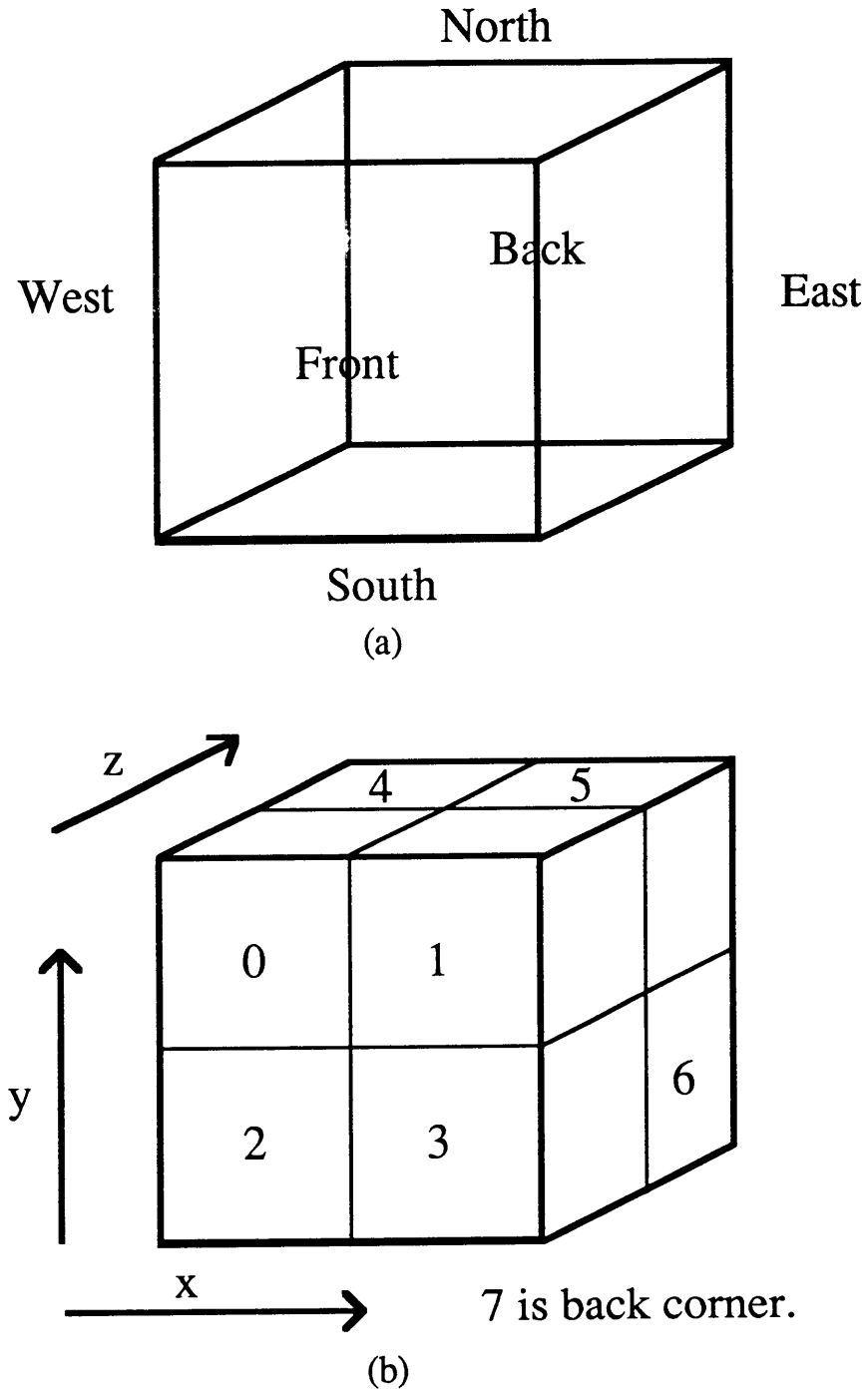
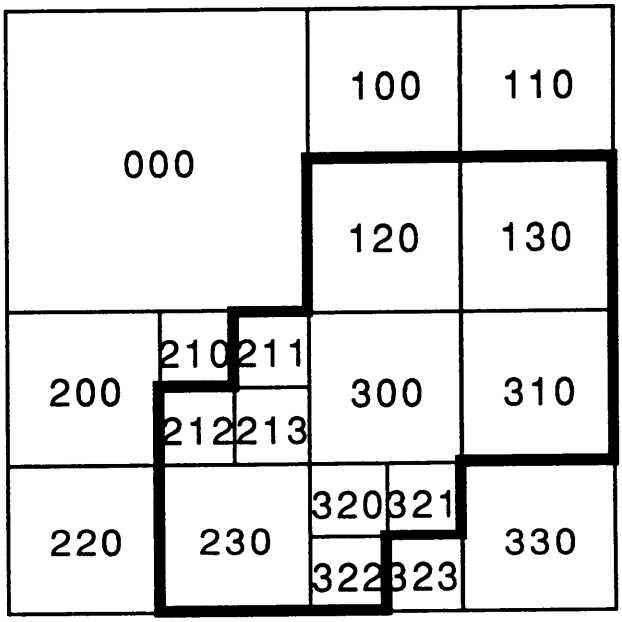


Figure 3. An octtree node. a) Faces of an octant. b) Octtree node divided into suboctants.

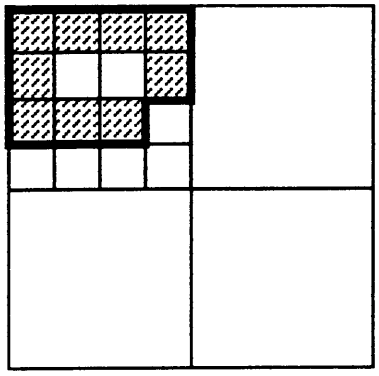
	000	001	010	011	100	101	110	111
000	000	001	010	011	100	101	110	111
001	002	003	012	013	102	103	112	113
010	020	021	030	031	120	121	130	131
011	022	023	032	033	122	123	132	133
100	200	201	210	211	300	301	310	311
101	202	203	212	213	302	303	312	313
110	220	221	230	231	320	321	330	331
111	222	223	232	233	322	323	332	333

(a)

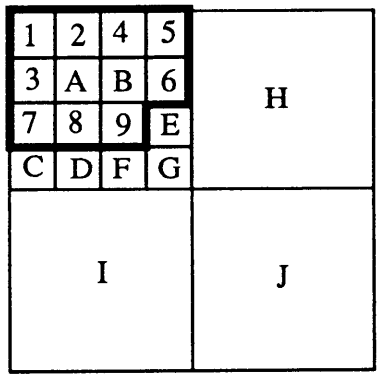


(b)

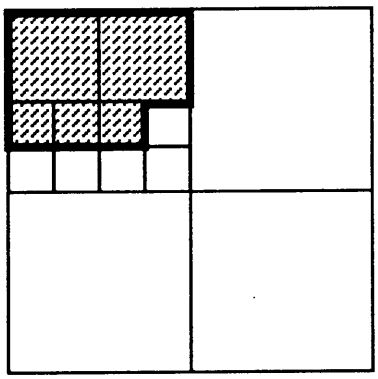
Figure 4. Morton codes for a quadtree. (a) The Morton code addressing scheme for labeling pixels. (b) The Morton code addresses for the blocks of Figure 2c.



(a)



(b)



(c)

Figure 5. Chaincode to Quadtree conversion process. (a) Initial list of border pixels and their boundary. (b) Boundary of region with border pixels labeled 1-9 and other nodes lettered. (c) Block decomposition of the final region quadtree.

1	NW	NE	SW	SE
0		B	B	
1				
2				

(a)

2	NW	NE	SW	SE
0		B	B	B
1		B		
2				

(b)

3	NW	NE	SW	SE
0		B	B	B
1		B	B	
2				

(c)

4	NW	NE	SW	SE
0		B	B	B
1		B	B	
2				

(d)

5	NW	NE	SW	SE
0		B	B	B
1		B	B	
2		W		

(e)

B	NW	NE	SW	SE
0		B	B	B
1		B	B	B
2		W		

(f)

Figure 6. Step by step example of color table update.
(a) the color table after it has been updated by pixel 1.
(b) after pixel 2. (c) after pixel 3. (d) after pixel 4.
(e) after pixel 5. (f) after node B.

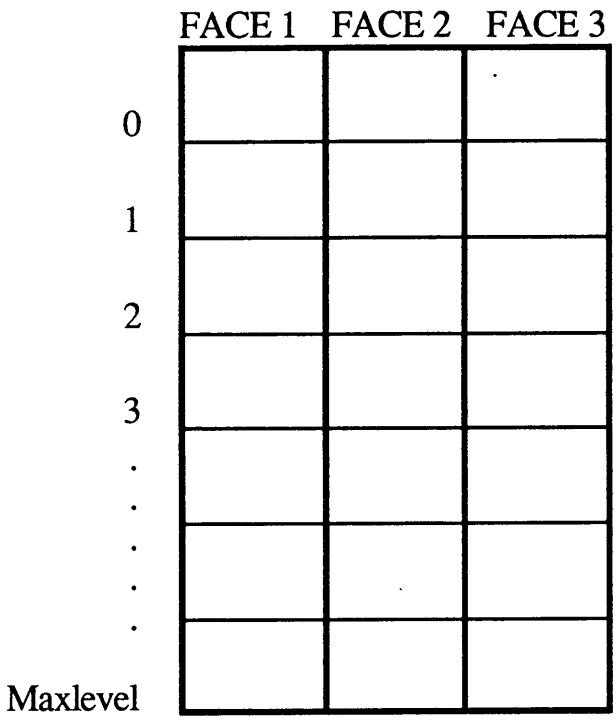


Figure 7. Lookup table for the particle update algorithm. Note that FACE 1 corresponds to the east, west faces. FACE 2 corresponds to the north, south faces. FACE 3 corresponds to the front, back faces.

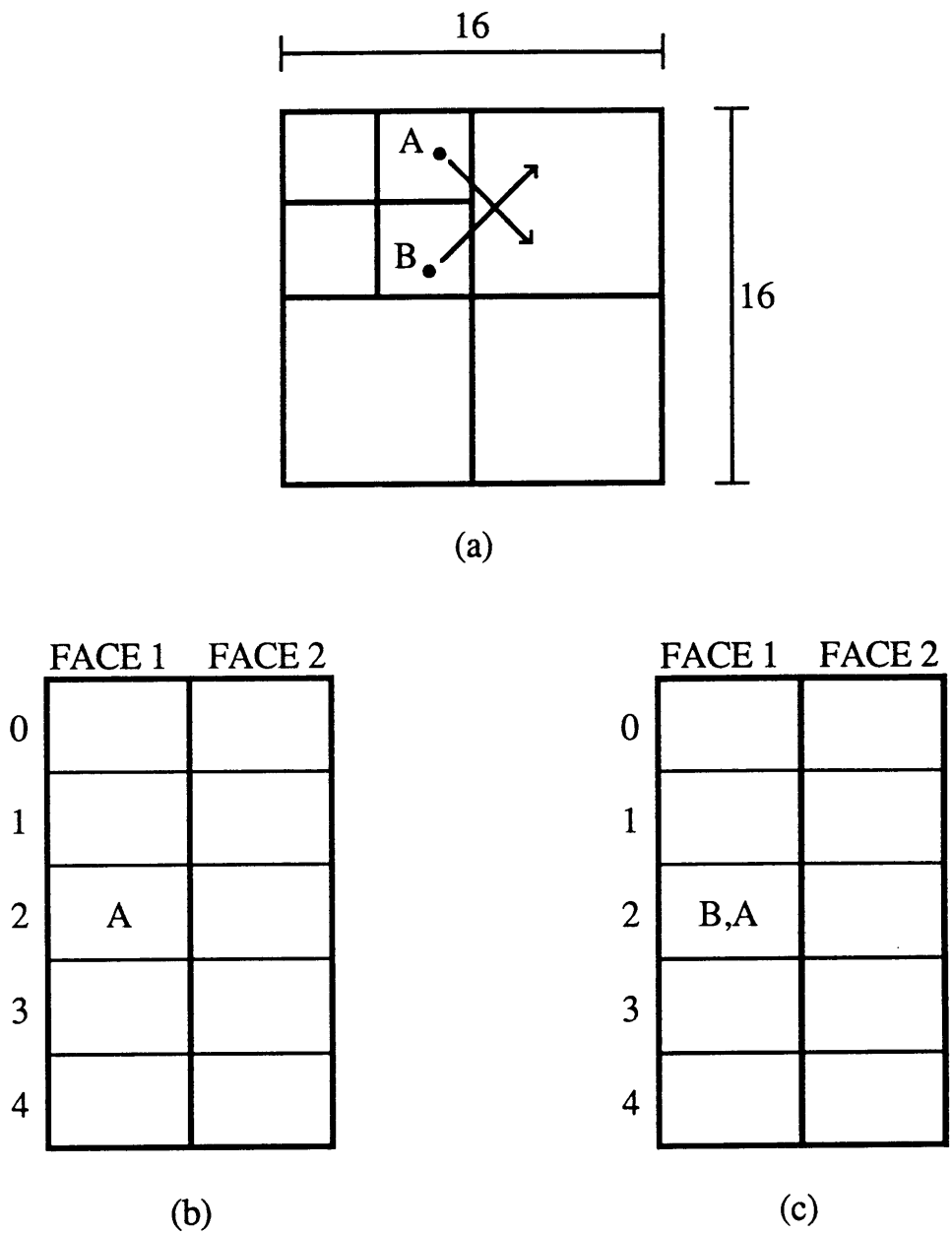


Figure 8. Visualization of a swap. (a) The motion of the particles in the quadtree. (b) Table after node containing A has been processed. (c) Table after node containing B has been processed.

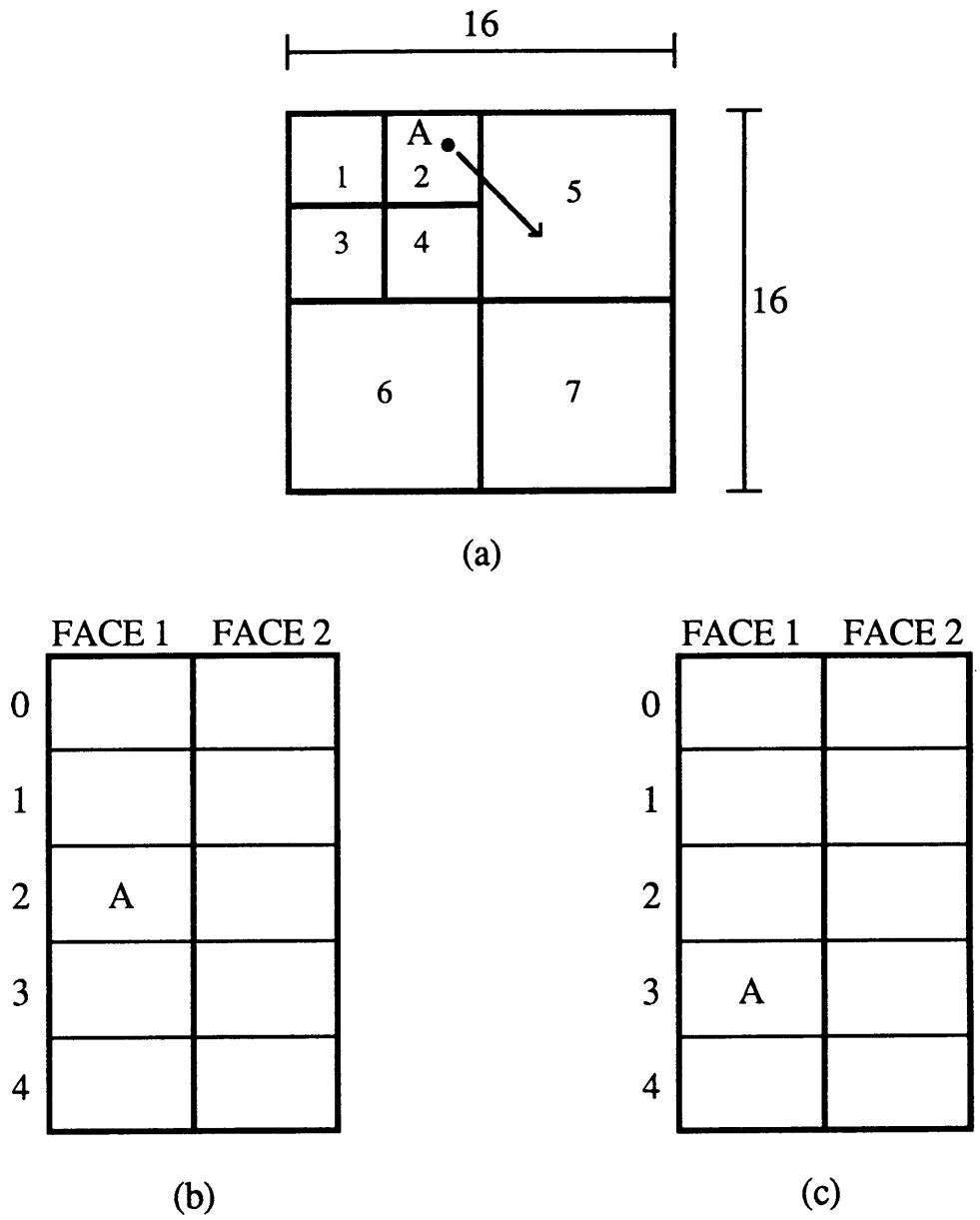
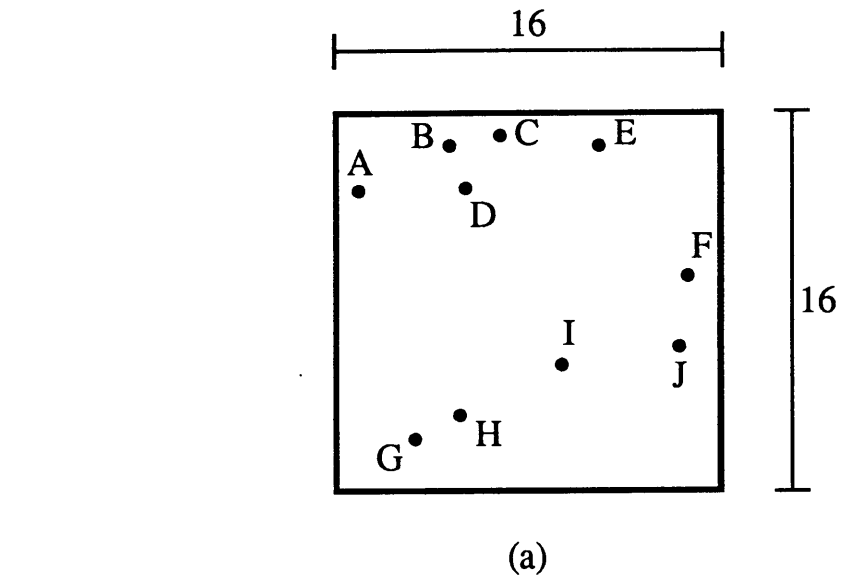
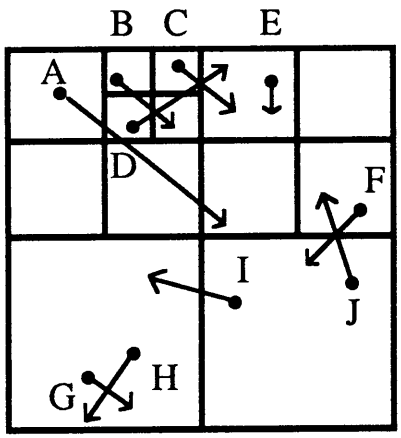


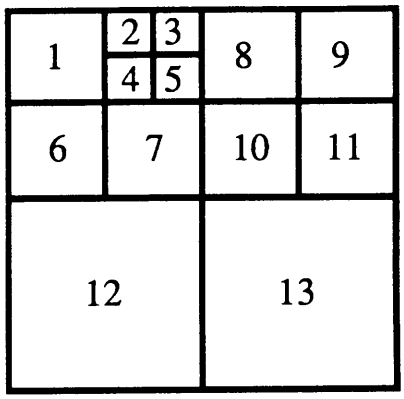
Figure 9. Visualization of an uptraversal. This happens when particle A moves out of the parent of its source node. (a) The motion of a particle in the quadtree. (b) Table after node 2 has been processed. (c) Table after parent of nodes 1,2,3 and 4 has been processed.



(a)



(b)

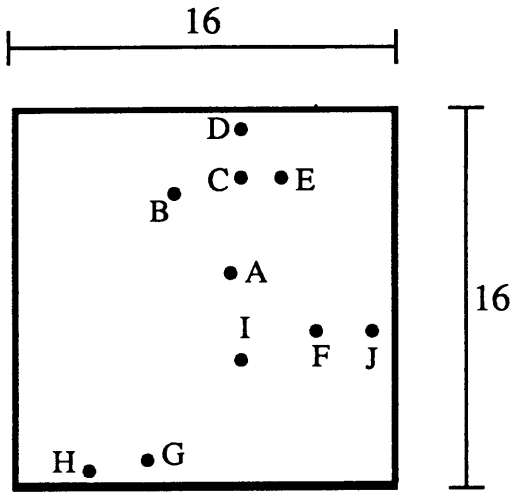


(c)

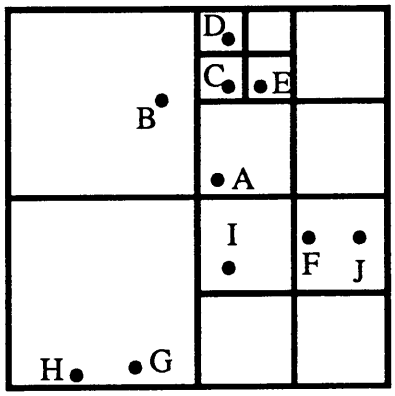
Figure 10. Initial configuration for the update example. (a) shows particles in the initial tree. (b) shows particle directions with vectors. (c) shows the labels for the quadtree nodes.

	AFTER NODE(S)	LEVEL	FACE 1	FACE 2
(a)	1	0		
		1		
		2	A	
		3		
(b)	4	0		
		1	D,C	B
		2	A	
		3		
(c)	Parent of 1-7	0		
		1		
		2		
		3	D,C,A	
(d)	11	0		
		1		
		2		F
		3		
(e)	Parent of 8,9,10 & 11	0		
		1		
		2		
		3		F

Figure 11. The table during various stages of the update example. a) Table after node 1 has been processed. b) After node 4 has been processed. c) After the parent of nodes 1 thru 7. d) After node 11. e) After parent of nodes 8,9,10, and 11.

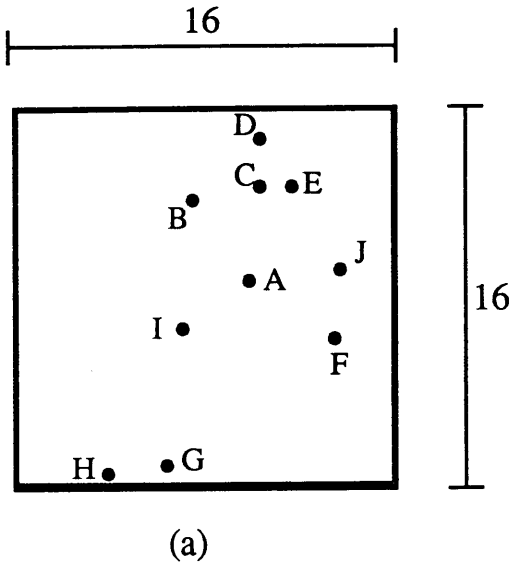


(a)

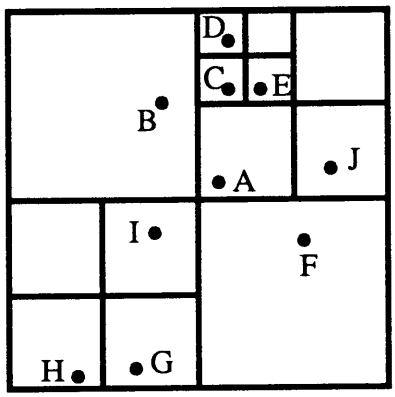


(b)

Figure 12. Example quadtree after the FORWARD pass of the update algorithm. (a) depicts the image space. (b) shows the final nodes of the quadtree.



(a)



(b)

Figure 13. Example quadtree after the update has completed.
(a) depicts the image space. (b) shows the final nodes of the quadtree.

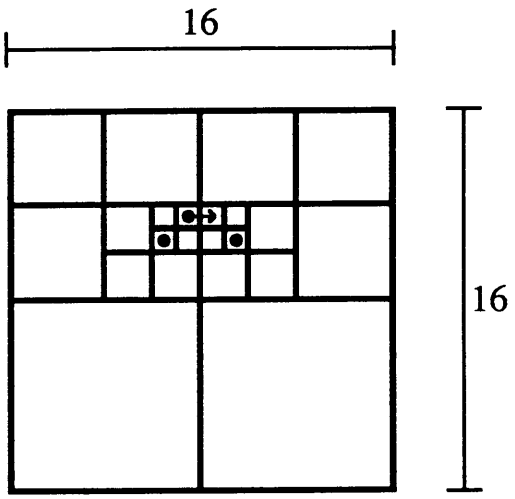


Figure 14. A particle that must move up through every level list and then back down through every list on its way to its destination.

# of particles	time for naive	time for linear	time (seconds)					number of swaps				
			v=1	v=5	v=10	v=50	v=200	v=1	v=5	v=10	v=50	v=200
100	1	1	1	1	0	1	1	0	0	0	15	242
200	2	2	1	2	1	2	2	0	1	3	101	836
400	4	4	2	3	3	4	4	4	11	22	605	5100
800	8	8	6	7	6	8	10	10	32	101	3050	2.39e4
1600	20	16	11	12	13	17	21	14	131	510	16851	1.14e5
3200	50	32	20	25	24	33	51	51	664	2608	88667	5.04e5
6400	140	64	46	54	62	79	129	290	3613	16216	4.69e5	2.29e6
12800	480	128	102	125	146	191	394	1475	19606	1.06e5	2.15e6	9.3e6
25600	1900	256	293	326	369	508	1338	7273	1.28e5	7.53e5	1.06e7	4.16e7
51200	7000*	512	611	775	974	1582	4213	4.1e4	9.70e5	4.44e6	4.58e7	1.90e8

Figure 15. Table of experimental results for a naive (rebuilding) algorithm, a linear (approximated) algorithm, and our algorithm using five different maximum velocities. Note that the naive simulation of 51200 particles was extrapolated due to memory constraints.

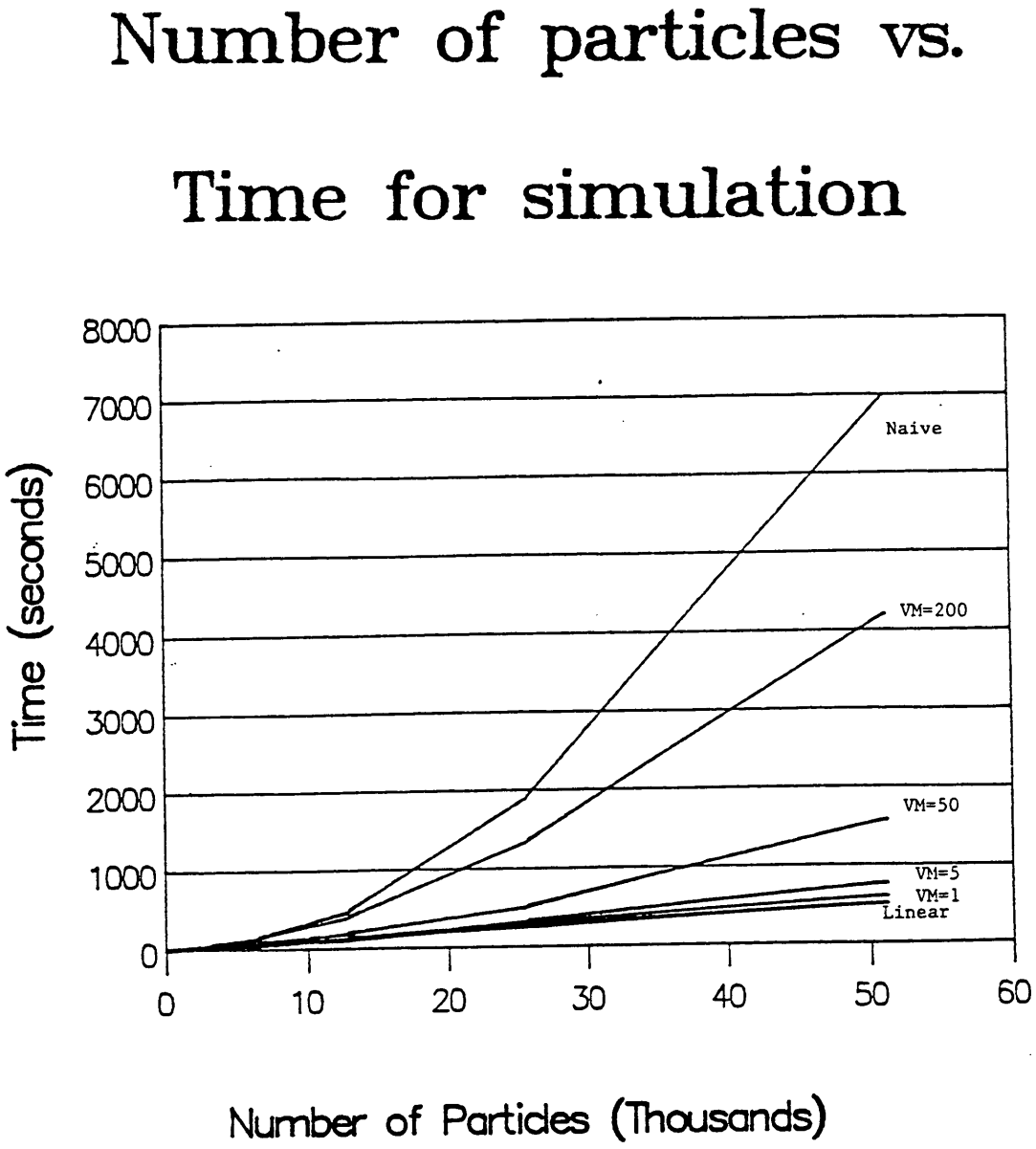


Figure 16. Plot of experimental results for a true linear update algorithm (approximated), and our update algorithm for maximum velocities of 1, 5, 50, and 200 pixels per time step.

The vita has been removed from the scanned document