ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/3649954

Integer dilation and contraction for quadtrees and octrees

Conference Paper · June 1995

DOI: 10.1109/PACRIM.1995.519560 · Source: IEEE Xplore

CITATIONS
17

READS
281

2 authors:

Leo J. Stocco
University of British Columbia - Vancouver
28 PUBLICATIONS **531** CITATIONS

SEE PROFILE

Guenther Schrack
University of British Columbia - Vancouver
55 PUBLICATIONS **443** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

High-level graphical programming languages View project

The first book in German on Computer Graphics; several chapters are now outdated. View project

All content following this page was uploaded by Guenther Schrack on 15 September 2014.

The user has requested enhancement of the downloaded file.

Integer Dilation and Contraction for Quadtrees and Octrees

Leo Stocco & Günther Schrack

Department of Electrical Engineering
The University of British Columbia
Vancouver, British Columbia, Canada V6T 1W5

Abstract - **Integer dilation and contraction are functions used in conjunction with quadtree and octree mapping systems. Dilation is the process of inserting a number of zeros before each bit in a word and contraction is the process of removing those zeros. Present methods of dilation and contraction involve lookup tables which consume considerable amounts of memory for mappings of large or high resolution display devices but are very fast under practical limits. A method is proposed which rivals the speed of the tabular methods but eliminates the tables, thereby eliminating the associated memory consumption. The proposed method is applicable to both dilation and contraction for both quadtrees and octrees.**

1. Introduction

A quadtree is a mapping system which divides a coordinate system into quadrants. Each quadrant is repeatedly subdivided into quadrants until the desired resolution of the coordinate system is achieved. Location codes are assigned to each of the elements in a hierarchical procedure similar to the segmentation procedure. They are ordered from zero to three as illustrated in Figure 1. The first segmentation determines the most significant quaternary digit and each subsequent segmentation determines the next significant quaternary digit. An example location code assignment is illustrated in Figure 1.

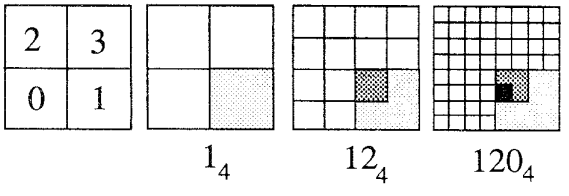

Figure 1: Quadtree Location Codes

A by-product of assigning location codes in the order prescribed by Figure 1 is that a location code can be calculated by interleaving the bits of its associated x-y coordinates. Once again consider the example location code in Figure 1:

$$(x, y) = (4, 2) = (100_2, 010_2)$$
$$interleave\,(x, y) = \{y_2, x_2, y_1, x_1, y_0, x_0\} = 120_4$$

Interleaving can be achieved by "OR"ing the quadtree dilation of "x" with the left shifted quadtree dilation of "y" where quadtree dilation is the process of inserting one zero before each bit in the word being dilated.

```
quad_location_code = quad_dilate(x) |
                     (quad_dilate(y) << 1);
```

Converting back to cartesian coordinates is achieved by the reverse procedure where the inverse operation of dilation is contraction.

```
x = quad_contract(quad_location_code);
y = quad_contract(quad_location_code >> 1);
```

The octree mapping system is the same except that it contains a third dimension. Just as quadtrees relate to a square area that is recursively divided into four parts, octrees relate to a cubic volume that is recursively divided into eight parts. Conversion from cartesian coordinates to an octree location code requires an octree dilation of each cartesian coordinate. Octree dilation inserts two zeros before each bit in the word being dilated.

```
oct_location_code =  oct_dilate(x) |
                     (oct_dilate(y) << 1) |
                     (oct_dilate(z) << 2);

x = oct_contract(oct_location_code);
y = oct_contract(oct_location_code >> 1);
z = oct_contract(oct_location_code >> 2);
```

Since quadtrees and octrees are used in graphics applications, efficient dilation and contraction functions are clearly a concern.

2. Present Techniques

The fastest known methods of dilation and contraction use linear look-up tables. For x-y coordinates of "n" bits, the size of the required table for one-part quadtree dilation is 2^n and for one-part quadtree contraction is

0-7803-2553-2/95/$4.00 © 1995 IEEE

qdil(2n - 1) which is on the order of 2^{2n-1}. Example look-up tables for n = 3 are shown in Figure 2 and Figure 3.

Integer	0	1	2	3	4	5	6	7
QDilate	0	1	4	5	16	17	20	21

Figure 2: Quadtree Dilation Look-up Table

Integer	0	1	2	3	4	5	6	7	8	9	10
QContract	0	1	✗	✗	2	3	✗	✗	✗	✗	✗

Integer	11	12	13	14	15	16	17	18	19	20	21
QContract	✗	✗	✗	✗	✗	4	5	✗	✗	6	7

Figure 3: Quadtree Contraction Look-up Table

Table sizes can, however, be reduced by performing the operations in more than one part and then concatenating the results. For example, a 16 bit integer can be dilated in two parts as follows.

```
dil_integer = dil_table[integer & 0xFF];
dil_integer |= (dil_table[integer >> 8]) << 16;
```

3. Proposed Techniques

The proposed dilation and contraction techniques do not use look-up tables. Instead, they use a loop which systematically spreads out or compresses the bits. The techniques are best described by a code segment and an example. It is assumed that a long integer contains 32 bits and is, therefore, capable of storing the dilation of a 16 bit integer.

```
#define INT_LEN 16
#define LOG2_INT_LEN 4

unsigned long demo( unsigned long *qdil,
                    unsigned long *qctc) {

static long dilate_masks[] = {0xFF00FF,
                              0xF0F0F0F,
                              0x33333333,
                              0x55555555};

static long contract_masks[] = {0x33333333,
                                0xF0F0F0F,
                                0xFF00FF,
                                0xFFFF};
int i;

  for (i = 0; i < LOG2_INT_LEN; i++) {

    *qdil = (*qdil |
            (*qdil << (INT_LEN >> (i + 1)))) &
            dilate_masks[i];

    *qctc = (*qctc |
            (*qctc >> (1 << i))) &
            contract_masks[i];
  }
}
```

After each iteration of the loop, the value of "*qdil" will be as follows where the final result is clearly the quadtree dilation of the original "*qdil".

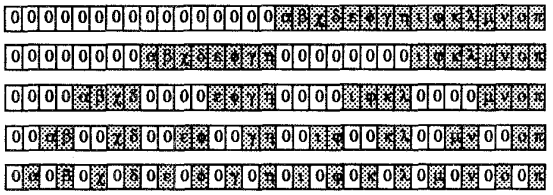

Figure 4: Example Integer Dilation

The 16 bit integer is split into two groups of eight which are then split into four groups of four, eight groups of two, and finally 16 "groups" of one which is the quadtree dilation. The algorithm is easily extended to octree dilation by doubling the amount of left shifting and adjusting the masks.

Quadtree contraction is the dual of quadtree dilation since each step of the contraction process undoes one step of the dilation process. The value of "*qctc" after each iteration is exactly the same as in Figure 4 with the order reversed. Quadtree contraction is also easily extended to octree contraction.

4. Analysis

Due to hardware or compiler constraints, the arrays used by the tabular methods could have size restrictions. When table sizes are inadequate for direct look-up, the operations must be performed in more than one part. For any fixed table size, the number of parts can be adjusted to accommodate any integer size. Fixing table sizes results in execution times that are proportional to the number of bits in the target integer. A relation derived from performance tests involving the tabular methods is as follows:

$$t = (14.6\alpha) / \ln(a) - 16$$

$t \equiv$ execution time in ticks
$a \equiv$ number of array elements
dilation $\Rightarrow \alpha = n$
contraction $\Rightarrow \alpha = 2n - 1$
$n \equiv$ number of bits in target integer

It should be noted that the number of array elements is not linearly proportional to memory consumption. For example, a quadtree dilation array with 16 elements needs only 8 bits per element since the largest integer it must store is 85. One with 32 elements, however, needs 16 bits per element since the largest integer it must store

is 341. Therefore a more representative indication of memory consumption is as follows:

$$b = \lceil \log 2(a)/4 \rceil \times a$$

$b \equiv$ memory consumption in bytes
$a \equiv$ number of array elements

The calculation methods use no tables and are symmetric for both dilation and contraction. A relation derived from performance tests involving the calculation methods is as follows:

$$t = 27.4ln(n) - 10$$

$t \equiv$ execution time in ticks
$n \equiv$ number of bits in target integer

Equating the two execution times yields the minimum memory requirement for the tabular methods to execute as fast as the calculation methods. Note that the ceiling function in the memory utilization algorithm accounts for the fact that memory is never allocated in fractions of bytes. For analytical purposes, the ceiling function can be omitted.

$$b = \frac{0.19\alpha}{ln(n) + 0.22} \times e^{(0.53\alpha/(ln(n)+0.22))}$$

$b \equiv$ memory consumption in bytes
$n \equiv$ number of bits in target integer
dilation $\Rightarrow \alpha = n$
contraction $\Rightarrow \alpha = 2n - 1$

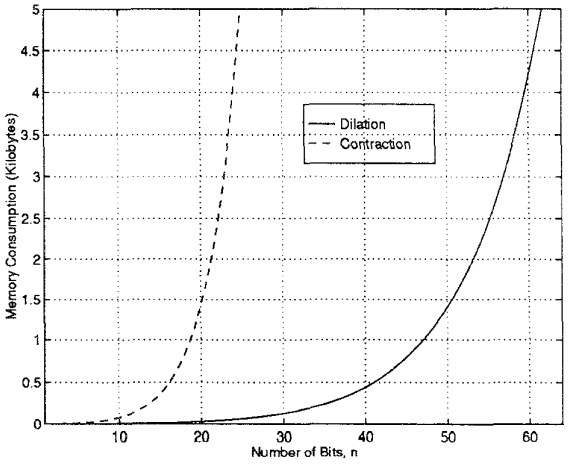

Figure 5: Memory Consumption of Array vs. Number of Bits

Since the calculation methods are of Order(ln(n)) and the tabular methods are of Order(n), the performance of the calculation methods surpass those of the tabular methods as the number of bits in the x-y coordinates increases. A direct comparison is provided in Figure 5 by fixing the table size at "a = 256" and plotting the performance of the methods for varying integer lengths. A 256 element table is chosen since it consumes a relatively small amount (512 bytes) of memory and fully utilizes the integer values recorded in the table. It is therefore a very efficient table size.

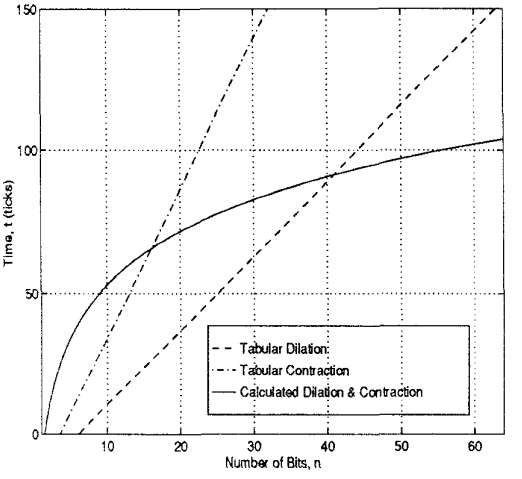

Figure 6: Execution Time vs. Number of Bits

5. Conclusion

The dilation and contraction techniques presented provide a sound alternative to the tabular method that is presently used. Due to their logarithmic order, the proposed algorithms can provide both a savings in memory utilization and a reduction in execution time of functions which encode and decode cartesian coordinates into location codes of large or high resolution quadtree or octree mappings.

6. References

[1] G. Schrack, "Finding Neighbors of Equal Size in Linear Quadtrees and Octrees in Constant Time", *CVGIP: Image Understanding*, **55** (3), May 1992, 221 - 230.

[2] I. Gargantini, "An effective way to represent quadtrees", *Commun. ACM* **25** (12), 1982, 905-910.