Pacific Graphics 2009
S. Lee, D. Lischinski, and Y. Yu
(Guest Editors)

Volume 28 (2009), Number 7

Linkless Octree Using Multi-Level Perfect Hashing

Myung Geol Choi[1] and Eunjung Ju[1] and Jung-Woo Chang[2] and Jehee Lee[1] and Young J. Kim[3]

[1]Seoul National University, Korea
[2]University of Hong Kong, Hong Kong
[3]Ewha Womans University, Korea

Abstract

The standard C/C++ implementation of a spatial partitioning data structure, such as octree and quadtree, is often inefficient in terms of storage requirements particularly when the memory overhead for maintaining parent-to-child pointers is significant with respect to the amount of actual data in each tree node. In this work, we present a novel data structure that implements uniform spatial partitioning without storing explicit parent-to-child pointer links. Our linkless tree encodes the storage locations of subdivided nodes using perfect hashing while retaining important properties of uniform spatial partitioning trees, such as coarse-to-fine hierarchical representation, efficient storage usage, and efficient random accessibility. We demonstrate the performance of our linkless trees using image compression and path planning examples.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.6]: Graphics data structures and data types—

1. Introduction

Uniform spatial partitioning data structures such as quadtrees and octrees are widespread in a variety of graphics applications including spatial indexing, image/volume encoding and compression, collision detection, visibility culling and path planning. These spatial partitioning structures often allow large spatial data sets to be maintained efficiently in terms of storage requirements and random accessibility, in particular, when strong spatial coherence exists in the data sets.

The standard C/C++ implementation of a spatial partitioning data structure subdivides each spatial cell into child cells recursively and maintains parent-to-child pointer links. Sometimes, the memory overhead for storing parent-to-child links is significant with respect to the amount of actual data in each cell. For example, a uniform spatial partitioning tree encoding a d-dimensional binary volume maintains a single bit (either zero or one) of data in each cell together with 2^d pointers. In a 32-bit addressing machine, the memory overhead for each non-leaf node is $32 * 2^d$ (bits), which is several orders of magnitude larger than the actual amount of data in the node.

A number of techniques have been studied for storing quadtrees and octrees memory efficiently. A linear quadtree converts a hierarchical structure into a one-dimensional array by traversing the tree in depth-first order [Gar82, OW83, Woo84]. The linear quadtree is memory efficient because it does not maintain parent-to-child pointers. The disadvantage of the linear depth-first traversal encoding is its inefficiency in random access to tree nodes. Accessing an arbitrary node necessitates sequential scanning of the array for tree traversal. Many of linear quadtree variants exhibit a trade-off between storage requirements and efficient random accessibility.

In this work, we present a novel data structure that implements uniform spatial partitioning without storing explicit parent-to-child pointers. Our linkless tree maintains non-leaf nodes at small extra storage by layering multiple hashing functions. Without storing explicit links, it retains several important properties of uniform spatial partitioning trees, such as coarse-to-fine hierarchical representation, efficient storage usage, and efficient random accessibility.

Our linkless tree is particularly useful for compactly representing high-dimensional binary bitmap volumes. Modeling free configuration space for path planning often generates such data sets. The free space map is a d-dimensional bi-

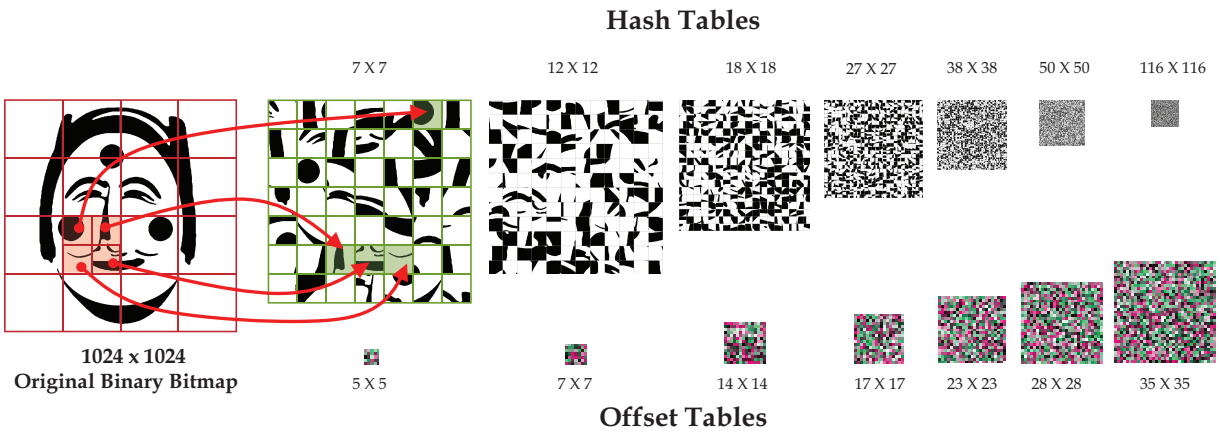


Figure 1: *The linkless quadtree of a binary image. Each entry of an offset table is a two-dimensional vector and each of its components is quantized to 8-bits. Vector values in the tables are color coded using red and green channels. Note that our algorithm does not generate the first two levels of the hierarchy because the compact packing of non-leaf cells at those levels is not smaller than the original image.*

nary bitmap volume, which represents whether each configuration cell is free or occupied. It is impractical to store an uncompressed bitmap volume because its size scales exponentially with its dimensionality. We will demonstrate three examples using three-, four-, and five-dimensional free space maps. The three-dimensional example models a planar rigid mover and static obstacles. The mover is allowed to translate and rotate on a plane. The use of an animated mover navigating through static obstacles adds an extra dimension (for parameterizing the pose of the mover) to yield four-dimensional configuration space. Modeling free configurations between two animated characters requires a five-dimensional binary bitmap, which can be compactly represented by using our linkless spatial partitioning tree.

2. Previous Work

Spatial partitioning is a standard technique in computer graphics. Though the standard pointer-based implementation is extremely popular in a variety of graphics applications, the variants of quadtrees and octrees have also been explored for reducing storage requirements, improving the performance of dynamic updates, and GPU implementation. Gargantini [Gar82] presented a linear quadtree that represents hierarchical tree data without pointers. The linear quadtree encodes only non-empty nodes with a quaternary integer whose digits indicate quadrant subdivision position and preserved the integer data in one-dimensional array. Oliver and Wiseman [OW83] and Woodwark [Woo84] compressed the linear quadtree further by providing sophisticated tree traversal code. Their major concern was deriving maximum benefit from compressing required storage and, therefore, the performance for accessing tree nodes and dynamically updating tree structures was compromised. Fabbrini [FM86] proposed an autumnal quadtree that stores the data of leaf nodes into the pointer fields in their parent nodes. The autumnal tree can be used only when the size of pointers is longer than data fields in each node. To avoid this restriction, Lefebvre [LH07] encoded each data value into 7 bits by using vector quantization method. This approach leads to lossy data compression.

The theoretical lower bound of the size of a binary tree encoding is $2n - o(n)$ bits, where n is the number of nodes. The theoretical lower bound can be achieved by level-order bit stream encoding that encodes leaf (zero) and non-leaf (one) nodes in a breath-first tree traversal order. Given such a succinct data structure, implementing basic operations such as finding a parent/child/sibling node is non-trivial. There have been significant research on compactly representing binary trees and augmenting auxiliary indexing structures to perform basic operations efficiently on succinct binary trees [Jac89, MR97]. Research on trees of higher-degree has arisen recently. Benoit et al. [BDM*05] encoded a tree of degree 2^d in $((\lceil d \rceil + 2)n + o(n) + O(\lg d))$ bits and implemented basic navigational operations in $O(1)$ time and random node access in $O(\lg n)$ time. On the other hand, several researchers explored efficient methods for constructing and dynamically updating quadtrees and octrees. Eppstein and his colleagues [EGS05] presented a skiptree data structure that allows for fast point insertion and deletion. Quadtrees and octrees have been implemented on GPUs by exploiting the general-purpose programming capability of modern GPUs [LSK*06, PF05, ZHWG08].

Spatial hashing techniques pack sparse spatial data into a compact table. Lefebvre and Hoppe [LH06] explored the use of perfect hashing in graphics applications and its implementation on GPUs. A hash function is *perfect* if it has no col-

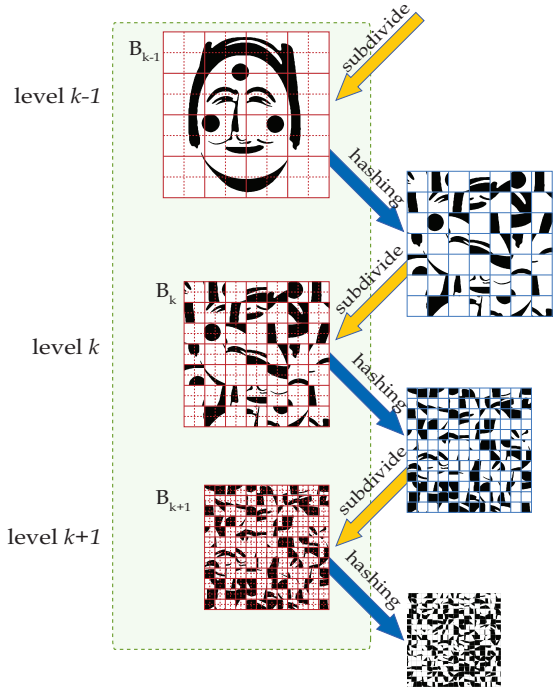


Figure 2: *Subdivision and hashing.*

lisions. Spatial hashing techniques have also been used to encode quadtrees/octrees exploiting the coherency in spatial data. Warren [WS93] presented a hashed octree that assigns unique keys to all (leaf and non-leaf) nodes and indexes them in a hash table. To improve search time, Castro [CLL*08] proposed a statistical model that selects the most frequenlty accessed level instead of the first level, as a starting point of the search. Bastos [BF08] similarly mapped octree nodes into a perfect hash table and accessed random nodes using octree level and point location indexes. The hash table store data associated with a sparse subset of the domain. If the indexing key does not correspond to a valid point in the domain, the hash function leads to arbitrary data in the table. Lefebvre and Hoppe [LH06] discussed several strategies for sparsity encoding, such as storing an extra binary bitmap over the domain to mark valid cells and augmenting position tags in the data fields. Whatever strategy is taken, sparsity encoding adds an extra burden for compact octree encoding. Our octree representation based on multi-level perfect hashing does not require extra storage for sparsity encoding because it retains the hierarchical structure of an octree. Invalid locations are automatically detected while tracing down the hierarchy.

3. Multi-Level Hashing

Our linkless octree employs a perfect spatial hashing technique [LH06] to maintain references to child nodes with small extra storage. Though our structure is called a linkless "octree", we actually refer to its d-dimensional generalization that performs uniform spatial partitioning for all axes. Our implementation allows 2D, 3D, 4D, and 5D volumes to be stored in an adaptive spatial partitioning structure. We will first explain the construction of a binary octree in Section 3.1. The generalization of the construction algorithm for dealing with longer data fields will be discussed later in Section 3.2.

3.1. Octree for Binary Data

The linkless octree consists of a pyramid of coarse-to-fine bitmaps. Each cell in the bitmap (except the bitmap at the finest level) has two bits. One bit encodes whether the cell is leaf node or not. A non-leaf cell includes both ones and zeros in its bitmap region. The non-leaf cell requires a subsequent subdivision to achieve higher resolution. The other bit marks the cell as either *all-zero* or *all-one* if the cell is leaf node. The finest level in the hierarchy has a one-bit binary bitmap because the leaf/non-leaf flag is not necessary any more.

Algorithm 1: Linkless octree construction

input : A binary bitmap volume V of size n^d.
output: Bitmap volumes B_k and offset tables Φ_k.

```
1  G ← SpatialGrids([0,n] × ··· × [0,n]) ;
2  b ← 1;   /* the size of spatial grids */
3  for k ← 0 to ⌈lg n⌉ do
4      b ← 2b ;
5      subG ← SubdivideGrids(G, b) ;
6      m ← CountNonLeafCells(subG, V) ;
7      if m = 0 then
8          B_k ← ConstructOneBitVolume(subG,V) ;
           /* generate the bitmap at the finest level and*/
           /* terminate the algorithm */
9          break ;
10     else if ⌈m^{1/d}⌉ < b then
           /* the bitmap is not generated if its size is not */
           /* smaller than the bitmap of the previous level */
11         B_k ← ConstructTwoBitVolume (subG,V) ;
12         Φ_k ← ConstructOffsetTable(B_k) ;
13         G ← CollectNonLeafCells(subG,h_k, Φ_k) ;
14         b ← ⌈m^{1/d}⌉ ;
```

The bitmap volume at level $(k+1)$ consists of non-leaf cells at its previous level k, which are compactly packed and then uniformly subdivided to produce finer grids (see Figure 2 and Algorithm 1). The hash function at each level maps a non-leaf parent cell to its child cells at the subsequent level. Perfect spatial hashing establishes conflict-free mapping between levels resorting on auxiliary offset tables. More precisely, let B_k be a d-dimensional bitmap at level k and its size

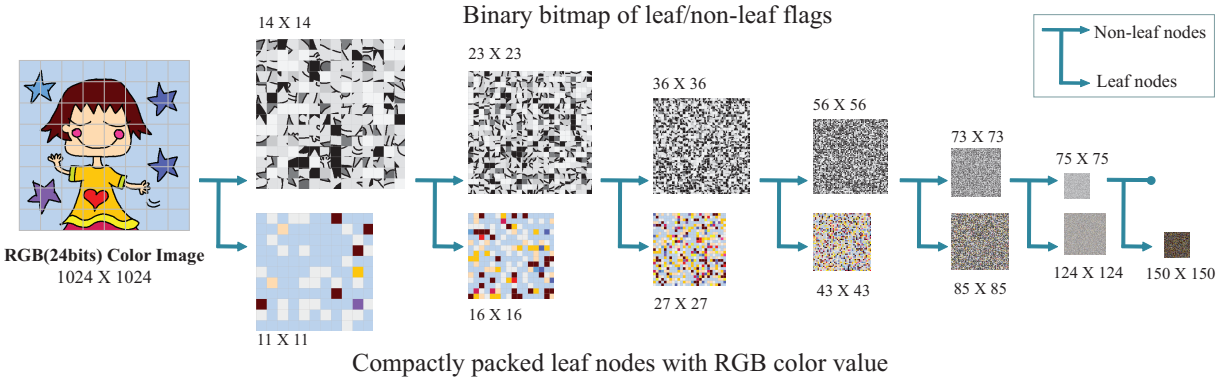


Figure 3: *The dual-hashing octree for a RGB color image. At each level, leaf nodes and non-leaf nodes are gathered separately and compactly packed using two spatial hashing functions. Only leaf nodes store data fields. Offset tables are omitted in the figure for clear presentation.*

is $(b_k)^d$ containing m non-leaf cells and the remaining leaf cells (line 6). The hash table of $\lceil m^{1/d} \rceil^d$ can accommodate m non-leaf cells. Let p be the spatial location in B_k and p' be its corresponding location in the hash table. The perfect hashing function at level k is

$$p' = h(p) = (p \mod b_{k+1}) + \Phi(p \mod r_{k+1}), \quad (1)$$

where b_k and r_k are the size of bitmap B_k and offset table Φ_k, respectively (line 11-13). The offset table is a d-dimensional array of offset vectors. We refer the reader to [LH06] for details on how to compute the offset table. In practice, the size of the offset table is insignificant comparing to the memory overhead of the pointer-based implementation. Then, bitmap B_{k+1} at the subsequent level is the uniform subdivision of the hash table (line 4-5). We repeat this process until no non-leaf cells remain in the bitmap.

3.2. Octree with Long Data Fields

The binary linkless octree can be generalized to deal with multi-bytes data by simply expanding the data field at each cell. However, the storage requirements can further be reduced if we store data only at leaf nodes. The data fields at non-leaf nodes are useful in some applications. For example, the non-leaf node of an image quadtree usually stores the average color value of its descendants nodes. Progressive transmission of an image can benefit from the average value, because progressively refining images can be viewed while the image is being transmitted. However, there are many other applications in which non-leaf data fields are useless and thus wasted. A number of geometric applications make use of octrees for spatial querying. Each octree cell maintains a set of assorted geometric primitives. Such an octree does not have meaningful values for non-leaf data fields.

Our dual-hashing octree removes the wasted data fields at non-leaf nodes to achieve better compression rates. Our dual-hashing octree makes use of two hashing functions at each level (see Figure 3). One hashing function is for locating non-leaf child nodes at the next level. Its functionality is the same as the hashing function for a binary octree. The only difference is that each cell in the hash tables has only one bit flag and does not have any data fields. The other hashing function is used for compactly packing leaf nodes with data fields. The dual-hashing octree performs better than the single-hashing octree if a larger amount of data is stored at each cell.

4. Experimental Results

The timing data provided in this section was measured on a 2.4GHz Intel Core2 Duo computer with 4Gbyte main memory and an nVidia GeForce 8800GTX GPU unless otherwise noted.

4.1. Free Configuration Space Modeling

The major advantages of our linkless octrees are efficient memory usage and random accessibility. To demonstrate the usefulness of the techniques, we modeled massive free configuration spaces using octrees. Animating and Planning character motions in a complex virtual environment require frequent interference-checking between characters and obstacles. Precomputation of free space maps and efficient random accessibility allow us to detect collisions very efficiently at runtime. Free space maps of 4D and 5D configuration spaces are too big to be accommodated in main memory. Therefore, compression is crucial. We constructed octree models of 3D/4D/5D free space maps approximately at a finite resolution.

The precomputation time for free space construction varies

significantly depending on its dimension, size and the complexity of geometry. It ranges from several minutes on a personal computer to several hours on a super computer. The largest 5-dimensional free space map was computed on a super computer with 484 nodes (5.6 Tflops performance). The computation time has nothing to do with octree construction. The collision detection between characters and obstacles was the bottleneck in computation. Collision checking was performed using V-COLLIDE, which is a collision detection library for large polygonal objects based on bounding volume hierarchies [HLC*97].

3D free space map. We consider the interference between a planar rigid mover and static obstacles (see Figure 4). The configuration of the mover can be described with three parameters (x, y, θ), where (x, y) is the position of the mover on the horizontal plane and θ is the rotation of the mover about the vertical axis with respect to the reference system. We built a free configuration space map having a $1024 \times 1024 \times 1024$ resolution of grids. The size of raw binary bitmap data is $1024^3/8 = 128$ (Mbyte), which can be compressed to 5.6 Mbyte using our linkless octree (see Figure 5 for details). The compression rate is about 95.6%. The top-down construction of the free space map requires the ability to check whether a cell is interference-free, partially-occupied, or completely-occupied without exhaustively examining all configurations in the range of the cell. This range query capability can save the tree construction time. Our system employs a range query method presented by Zhang et al. [ZKVM06], which can examine whether two convex objects overlap (or disjoint) for every configuration (translation and rotation) in a given spatial range. A concave object need to be decomposed into a collection of convex objects.

4D free space map. The goblin in Figure 5(top, left) is animated using 256 frames of motion data. We built a simple hand-crafted motion graph that allows transitioning between motion frames [LCR*02]. The relative configuration (x, y, θ, i) of the animated character with respect to a static obstacle is four-dimensional, where (x, y) is the relative translation, θ is the relative rotation, and i is the frame index of the motion graph. The garden in the figure includes 14 different kinds of polygon trees and flowers. A four-dimensional free space map was precomputed for the goblin and each individual object. We constructed 14 free space maps of $256 \times 256 \times 256 \times 256$ resolution. The size of each map ranges from 4 Mbytes to 30 Mbyte depending on the size and complexity of geometry. Since a raw bitmap volume requires $256^4/8 = 512$ (MByte) memory, the compression rate ranges from 93.9% to 99.2%.

5D free space map. The free space map of two animated characters has five-dimensional configuration space (x, y, θ, i, j). The characters' poses are described by i and j, respectively. We demonstrate two five-dimensional examples. The animal example in Figure 5(top, center) features a thousand characters of four different species (bird, frog, penguin, and pig). Each character has 32 poses for animation. We built a five-dimensional free space map of 128^5 resolution. Its size in memory is about 150 Mbytes. The goblin example in Figure 5(top, right) features a thousand goblins animated by using a motion graph with 512 poses. The skin deformation at each pose was precomputed and stored as polygon data. The goblin example used a five-dimensional free space map of $64 \times 64 \times 64$ spatial resolution and 512×512 animation resolution. The size of the map is about 320 Mbytes. Each goblin character consists of ten thousand polygons. Given the free space map, interference between a thousand animated goblins can be checked at interactive rate.

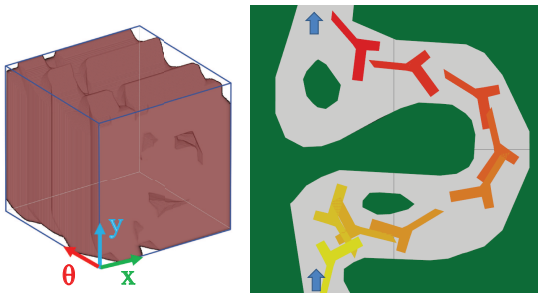
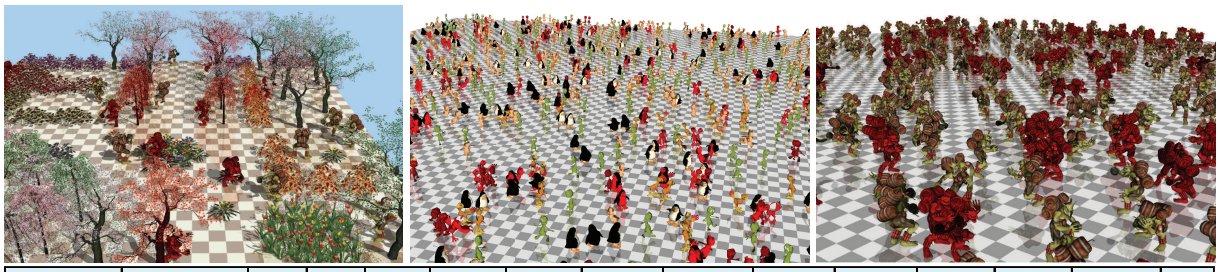


Figure 4: *Path planning of a planar rigid mover through static obstacles. The free configuration space can be modeled as a three-dimensional binary volume on the left.*

4.2. Performance Comparison

We used 2D images to compare our linkless quadtrees with existing techniques because conducting comparison tests with higher-dimensional data is too difficult and timeconsuming. The standard implementation of some existing techniques does not easily generalize to cope with higherdimensional data.

Binary Image. Though our octrees are not meant to compress 2D binary images, the comparison tests give a good sense how it performs in comparison with well-known lossless compression methods, such as pointer-based quadtree, ZIP, run-length encoding, and CCITT. CCITT Group3 and Group4 are industry standards for compressing bitonal image data and used by most facsimile machines. Group3 compression is a one dimensional algorithm that encodes image data scanline-by-scanline. Group4 compression encodes each scanline with reference to the previous scanlines to improve compression ratios. The comparison tests are conducted with two test images (see Figure 6). The face image has large all-black and all-white regions and the boundary between black and white regions is relatively clean. The text image, on the other hand, has a lot of details and thus do not compress well using octrees. For both images, the linkless octree is an order of magnitude smaller than the pointer-



Collision Map		Resolution		Level0	Level1	Level2	Level3	Level4	Level5	Level6	Level7	Level8	Level9	Total (Kb)	Total (Mb)
3D	Path Planning	1024x1024x1024	Bitmap	-	-	-	1,024	9,004	34,186	83,328	308,114	1,054,200	1,416,139	2,905,995	
			Offset	-	-	-	8,800	22,792	61,108	205,412	773,084	1,888,188	-	2,959,384	5.6
4D Garden	Cherry01	256x256x256x256	Bitmap	-	-	1,024	9,604	82,344	640,000	5,345,344	14,623,232	-	-	20,701,548	
			Offset	-	-	2,500	16,384	202,500	937,024	9,253,764	-	-	-	10,412,172	29.7
	Cherry02		Bitmap	-	64	324	2,500	16,384	114,244	777,924	1,847,042	-	-	2,758,482	
			Offset	-	1,024	324	2,500	16,384	114,244	937,024	-	-	-	1,071,500	3.7
	Cherry03		Bitmap	-	-	1,024	5,184	40,000	419,904	2,829,124	8,201,250	-	-	11,496,486	
			Offset	-	-	2,500	9,604	58,564	640,000	3,694,084	-	-	-	4,404,752	15.2
	Tulips01		Bitmap	-	-	1,024	9,604	82,944	521,284	4,194,304	12,500,000	-	-	17,309,160	
			Offset	-	-	1,024	9,604	153,664	1,119,364	6,718,464	-	-	-	8,002,120	24.1
	Tulips02		Bitmap	-	-	1,024	9,604	82,944	640,000	4,743,684	14,623,232	-	-	20,100,488	
			Offset	-	-	1,024	9,604	114,244	1,327,104	6,718,464	-	-	-	8,170,440	27.0
	Gazania01		Bitmap	-	-	1,024	5,184	58,564	419,904	3,240,000	8,954,912	-	-	12,679,588	
			Offset	-	-	2,500	5,184	82,944	521,284	4,194,304	-	-	-	4,806,216	16.7
	Gazania02		Bitmap	-	-	1,024	5,184	58,564	419,904	3,240,000	8,954,912	-	-	12,679,588	
			Offset	-	-	2,500	9,604	82,944	640,000	6,002,500	-	-	-	6,737,548	18.5
	Gazania03		Bitmap	-	-	1,024	5,184	58,564	419,904	3,240,000	9,759,362	-	-	13,484,038	
			Offset	-	-	2,500	9,604	114,244	937,024	4,743,684	-	-	-	5,807,056	18.4
	Gazania04		Bitmap	-	-	1,024	5,184	58,564	419,904	3,240,000	9,759,362	-	-	13,484,038	
			Offset	-	-	2,500	9,604	82,944	777,924	4,194,304	-	-	-	5,067,276	17.7
	Coleus01		Bitmap	-	-	1,024	9,604	58,564	521,284	3,694,084	10,616,832	-	-	14,901,392	
			Offset	-	-	1,024	26,244	114,244	1,327,104	6,718,464	-	-	-	8,187,080	22.0
	Coleus02		Bitmap	-	-	1,024	9,604	82,944	640,000	4,194,304	11,529,602	-	-	16,457,478	
			Offset	-	-	2,500	9,604	82,944	937,024	6,718,454	-	-	-	7,750,526	23.1
	Coleus03		Bitmap	-	-	1,024	9,604	58,564	521,284	3,694,084	10,616,832	-	-	14,901,392	
			Offset	-	-	1,024	16,384	82,944	777,924	4,743,684	-	-	-	5,621,960	19.6
	Coleus04		Bitmap	-	-	1,024	9,604	82,944	521,284	4,194,304	11,529,602	-	-	16,338,762	
			Offset	-	-	1,024	9,604	114,244	777,924	5,345,344	-	-	-	6,248,140	21.5
	Coleus05		Bitmap	-	-	1,024	9,604	58,564	521,284	3,694,084	10,616,832	-	-	14,901,392	
			Offset	-	-	1,024	9,604	82,944	777,924	4,194,304	-	-	-	5,065,800	19.0
5D	Animals	128x128x128x128x128	Bitmap	-	39,248	859,748	3,084,906	11,259,308	38,473,048	51,122,065	-	-	-	104,838,323	
			Offset	-	322,968	1,938,831	4,211,946	12,780,369	32,628,180	-	-	-	-	51,882,294	149.5
	Goblin	64x64x64x512x512	Bitmap	288	8,896	173,000	2,753,552	41,990,712	243,630,820	-	-	-	-	288,557,268	
			Offset	368	25,400	294,568	4,311,504	49,926,720	-	-	-	-	-	54,558,560	327.2

Figure 5: *Examples of free configuration space maps. (Up left) Goblins in a garden. A 4D map was precomputed for the goblin and each individual object in the garden. (Up middle) A thousand animal characters were animated using 256 frames of motion data. The interference between characters were checked in realtime using a 5D free space map. (Up right) A thousand animated goblins were animated using 512 frames of motion data. (Down) The size of free space maps.*

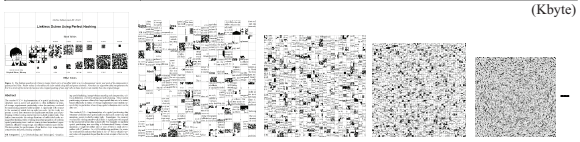
based octree. The linkless octree performs better than run-length encoding and ZIP for the face image, but not as well as CCITT Group3 and Group4. The text image is a particularly bad example for octree encoding because not many internal cells are pruned in the octree hierarchy. In our comparison tests, the linkless quadtree was not the best for compressing black-and-white images, but at least adequate for images with strong spatial coherency.

Color Image. The color image in figure 3 has a 1024×1024 resolution and each pixel has 24 bits for RGB color. The size of the raw uncompressed image is $1024^2/8 = 3072$(Kbyte). The size of the pointer-based quadtree is 439.6 Kbytes. Our single-hashing octree constructed by simply expanding the data field in each cell requires 203.2 Kbytes, which can further be compressed by using dual hashing functions, as explained in Section 3.2. Our dual-hashing octree requires 179.4 Kbytes, which achieves a compression rate of 11.7% with respect to the single-hashing octree. The compression rate is directly related to the length of the data field. A higher compression rate can be achieved for an octree with longer data fields.

Quadtree Comparison. We encoded the face image in Figure 6 in four quadtrees (pointer-based, sibling [HW91], autumnal [FM86] and our linkless tree) and compared the memory overhead for maintaining parent-to-child pointers, sibling pointers, and auxiliary hash tables (see Figure 7). The pointer-based tree has four pointers in every non-leaf node and each pointer uses 4 bytes. Therefore, the tree requires 16 bytes per node. The sibling tree and autumnal tree require 4 bytes and 1.125 bytes per node, respectively [LH07]. The size of auxiliary hash tables for our linkless tree is about one-fourth of the memory overhead of the autumnal tree, which is much more memory-efficient than the other two tree encodings. Succinct k-nary tree by Benoit et al. [BDM*05] requires $(4n + o(n) + C)$ bits for storing a quadtree, where $(o(n) + C)$ is the size of the auxiliary index-

Face	Resolution					
	32	64	128	256	512	1024
Linkless quadtree	0.222	0.471	0.959	2.176	4.349	9.259
Pointer-based quadtree	2.421	5.641	12.841	26.581	57.521	121.361
Zip	0.211	0.365	0.839	1.963	4.89	12.236
Run-length	0.4	0.7	1.5	3.4	7.8	16.9
CCITT (group3)	0.4	0.6	0.9	1.8	3.7	7.9
CCITT (group4)	0.3	0.4	0.6	1	1.8	3.4

(Kbyte)



Text	Resolution					
	32	64	128	256	512	1024
Linkless quadtree	0.245	0.870	4.207	11.709	35.937	91.316
Pointer-based quadtree	5.741	18.774	80.789	238.767	771.531	1910.700
Zip	0.224	0.505	1.377	4.389	15.842	43.002
Run-length	0.4	0.7	1.7	5.0	15.8	47.1
CCITT (group3)	0.4	0.7	1.7	4.5	15.1	32.9
CCITT (group4)	0.4	0.6	1.6	3.9	14.3	27.6

(Kbyte)

Figure 6: *2D image compression performance comparison.*

ing structure and C is a large constant. For the 1024×1024 image with 15,409 octree nodes, the succinct quadtree takes $7,705 + o(n)$ (bytes), which is 14.9% larger than our linkless octree.

Computation time. We compare our hash-based quadtree and the pointer-based quadtree for the 1024×1024 face image in Figure 6. The construction of the hash-based quadtree takes 0.735 seconds, which is ten times slower than the pointer-based tree construction (0.076 seconds). Evaluating a perfect hash function requires two modulo and one table lookup operations. Therefore, the random access to the hash-based quadtree is slower than the random access to the pointer-based quadtree, which requires only one pointer indirection. In our experiments, accessing a million random nodes in the hash-based quadtree and the pointer-based quadtree took 0.625 and 0.095 seconds, respectively. Though tree accessing using hash functions is slower than pointer indirection, it is much more efficient than other compressed linear quadtrees that requires $O(n)$ time for random node access.

5. Discussion

We have presented a pointerless octree that makes use of multi-level perfect spatial hashing. Our linkless implementation would allow octrees to be employed in a wider variety of applications.

Our linkless octree has several limitations. The perfect hashing functions we employed are near optimal in the sense that the storage requirement for storing offset tables is small, but may not be optimal. There exists a trade-off between the storage efficiency and the construction time. In our implementation, the tree construction was considered as a preprocessing phase and we were mainly concerned with reducing storage costs while allowing efficient access to data at runtime.

Image resolution (# of total nodes)	32 (299)	64 (743)	128 (1657)	256 (3585)	512 (7303)	1024 (15409)
Linkless quadtree	119	295	601	1508	1968	6724
Pointer-based quadtree	4784	11888	26512	57360	116848	246544
Sibling quadtree	1196	2972	6628	14240	29212	61636
Autumnal quadtree	336	836	1864	4033	8216	17335

(Byte)

Figure 7: *Memory overhead comparison. We measured the size of pointers and hash tables of four quadtrees. The data fields are not included in the size.*

Another limitation is the lack of local refineability. Inserting and deleting a point in a perfect hash table usually lead to rebuilding the entire hash table. Therefore, our linkless octree cannot allow for frequent local updates and thus may not be adequate for representing dynamically changing data. Developing a dynamically updateable perfect hashing function is an interesting direction for future research.

Acknowledgements

We sincerely appreciate the advice of Prof. Srinivasa Rao Satti. This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2007-511-D00332) and the grant from the strategic technology development program (Project No. 2008-F-033-02) of both the MKE (Ministry of Knowledge Econemy) and MCST (Ministry of Culture, Sports and Tourism) of Korea.

References

[BDM*05] BENOIT D., DEMAINE E. D., MUNRO J. I., RAMAN R., RAMAN V., RAO S. S.: Representing trees of higher degree. *Algorithmica 43*, 4 (2005), 275–292. 2, 6

[BF08] BASTOS T., FILHO W. C.: Gpu-accelerated adaptively sampled distance fields. In *Proceedings of IEEE International Conference on Hsape Modeling and Applications* (2008), pp. 171–178. 3

[CLL*08] CASTRO R., LEWINER T., LOPES H., TAVARES G., BORDIGNON A.: Statistical optimization of octree searches. *Computer Graphics Forum 27*, 6 (march 2008), 1557–1566. 3

[EGS05] EPPSTEIN D., GOODRICH M. T., SUN J. Z.: The skip quadtree: Asimple dynamic data structure for multidimensional data. In *Proceedings of the 21st Annual Symposium on Computational Geometry (SoCG'05)* (2005), pp. 296–305. 2

[FM86] FABBRINI F., MONTANI C.: Autumnal quadtrees. *The Computer JOURNAL 29*, 5 (1986), 472–474. 2, 6

[Gar82] GARGANTINI I.: An effective way to represent quadtrees. *Communications of the ACM 25*, 12 (1982), 905–910. 1, 2

[HLC*97] HUDSON T. C., LIN M. C., COHEN J., GOTTSCHALK S., MANOCHA D.: V-collide: accelerated collision detection for vrml. In *Proceedings of Symposium on Virtual Reality Modeling Language (VRML '97)* (1997). 5

[HW91] HUNTER A., WILLIS P. J.: Classification of quad-encoding techniques. *Computer Graphics Forum 10*, 2 (1991), 97–112. 6

[Jac89] JACOBSON G.: Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS '89)* (1989), pp. 549–554. 2

[LCR*02] LEE J., CHAI J., REITSMA P. S. A., HODGINS J. K., POLLARD N. S.: Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics (SIGGRAPH 2002) 21*, 3 (2002), 491–500. 5

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. *ACM Transactions on Graphics (SIGGRAPH 2006) 25*, 3 (2006), 579–588. 2, 3, 4

[LH07] LEFEBVRE S., HOPPE H.: Compressed random-access trees for spatially coherent data. In *Proceedings of the Eurographics Symposium on Rendering* (2007). 2, 6

[LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics 25*, 1 (2006), 60–99. 2

[MR97] MUNRO J. I., RAMAN V.: Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)* (1997), p. 118. 2

[OW83] OLIVER M. A., WISEMAN N. E.: Operations on quadtree encoded images. *The Computer Journal 26*, 1 (1983), 83–91. 1, 2

[PF05] PHARR M., FERNANDO R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Chapter 1. Octree Textures on the GPU)*. Pearson Education, 2005. 2

[Woo84] WOODWARK J.: Compressed quad trees. *The Computer Journal 27*, 3 (1984), 225–229. 1, 2

[WS93] WARREN M. S., SALMON J. K.: A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing (Supercomputing '93)* (1993), pp. 12–21. 3

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (SIGGRAPH Asia 2008) 27*, 5 (2008), 1–11. 2

[ZKVM06] ZHANG L., KIM Y. J., VARADHAN G., MANOCHA D.: Fast c-obstacle query computation for motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA 2006)* (2006), pp. 3035–3040. 5