A Domain Pool Classification Method for Better Fractal Volume Compression

Mihai Popescu, Mihai Pancu, and Razvan Tudor Tanasie
Department of Software Engineering
Faculty of Automatics, Computers and Electronics
University of Craiova, Romania
Email: {mpopescu, mpancu, razvan.tanasie}@software.ucv.ro

***Abstract*—In this paper we solve the problem of domain pool classification in fractal volume coding using all symmetries of a cube. Using the algorithms described in the article we are able to perform useful Domain-Range comparisons and achieve better compression rates while not loosing fidelity. In this paper we take advantage of the symmetric permutation group of cube and decrease the number of classes (from 10080 to 840). The same transformations are used in the compression and so we will have a better approximation of the final compressed volume. This paper represents a work in progress so no experimental results can be provided at this time.**

Keywords- domain pool, classification, volume compression, fractals.

I. INTRODUCTION

Fractal compression is a coding technique originally proposed by Barnsley [1] and it's based on the fact that data entropy is self-similar. For this, fractal compression is considered a special form of vector quantization method having the codebook vector self-contained rather than external.

Due to a huge compression time the method was considered impractical at first achieving almost the same rate-distortion curves as the DCT methods. The only attractive properties of the fractal coding that also attracted a lot of research was resolution independence (meaning that data can be decoded at any resolution) and fast decoder convergence.

These properties made fractal compression more suited for off-line applications rather than for real-time ones, like video compression, even if the first attempts [2] to extend the fractal image compression method to the 3D realm was to treat video signal as a volume.

II. STATE OF THE ART

The majority of fractal compression techniques are based on the method developed by Jacquin [3] and later by Fisher et al. [4] by which data is partitioned into blocks named *ranges* and *domains*. The bottleneck of the method resides in the search for the optimal pairing between a range and a domain. Even if the domain size is restricted to be twice the range size, the overlapping lattice of the domain can generate huge domain pools.

Moreover, to improve quality, a domain block is transformed using isometric symmetry operations such that the encoder will find nearly optimal domain-range pairing. If maximum speed is required the symmetry operations can be ignored but the overall quality will suffer dramatically [5].

The surest way is to use a brute force approach and to consider the entire domain pool but the time complexity will be $O(n^2)$ and it becomes impractical as the volume size increases.

Opposed to faster searching, less searching is a promising approach. If one can determine a-priori if a domain is not likely to be used in the final fractal code, eliminating it from the domain pool can improve performances while not loosing fidelity.

One way to reduce the domain pool is by considering only domains at even lattice locations. Another method is by searching in a restricted spatial partition defined by the parent quadrant/octant of the range block or in the near vicinity of it. Using these methods, the spatial information from the fractal code can be efficiently packed with a minimum amount of bits.

Local 2D spiral search from [6] can easily be extended to 3D using a Hilbert scan curve. Other methods can imply just the elimination of a specified fraction of the domains having small variance [7].

For images, Jacquin sorts the domains into three classes (shade, edge and mid-range blocks) and restricted the search only within the same class. On the other hand, Fisher [4] used 72 classes taking into account not only intensities but also the intensity variance across the domains.

The first complete research in fractal volume compression was elaborated by Cochran [8]. He proposed a new classification method by using Principal Component Analysis (PCA) where domains are classified by their variance. PCA is a well known technique for finding orthogonal basis vector that express the direction of progressively smaller variance in a given data set.

In our approach, the volumetric fractal coding algorithm [9] worked by segmenting the volume into domains using an octree and classified the domains using only one rotation. In this paper we take advantage of the complete symmetric permutation group of cube rotations and reflections in order to find the best candidates for Domain-Range comparisons. We are targeting to achieve a good rate-distortion curve disregarding the speed although experimental results cannot be provided at this time.

 ISBN: 978-1-61208-195-3

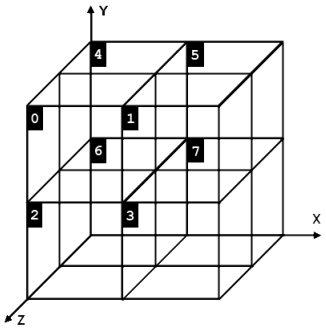


Figure 1. Cube

III. Cube Symmetries

The cube has octahedral symmetry like all the other octahedron solids (e.g., regular octahedron, truncated octahedron, truncated cube) but the cube is the only Platonic solid (among triangular prism, hexagonal prism, truncated octahedron) and also a hierarchical space-filling polyhedron. The octahedral symmetry group O_h has symmetry order 48 including transformations that combine a reflection with a rotation. It is isomorphic to $S_4 \times C_2$ and has 24 direction-preserving symmetries. The elements are:

1) 1 identity rotation that leaves the cube unchanged
2) 3 rotations (by $\pm\pi/2$ or π) around the centres of the 3 pairs of opposite faces
3) 1 rotation (by π) around the centres of the 6 pairs of opposite edges (that pass through the centre)
4) 2 rotations (by $\pm 2\pi/3$) around the 4 pairs of the opposite vertices (on diagonals)

To sum up, we have $1+3*3+1*6+2*4 = 24$ rotations.

The other 24 symmetries do not preserve directions because the transformation include a reflection and this implies changing the face normals along the symmetry plane. There are two types of symmetry planes for the cube. One is perpendicular to the coordinate system unit vectors $\vec{i}, \vec{j}, \vec{k}$ (one for each axis) and the other type is across diagonals (two for each pair of opposite vertices). So there are $3+2*3 = 9$ planes of symmetry for a cube.

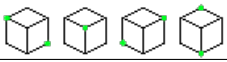
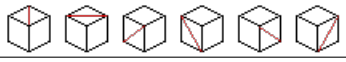
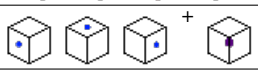
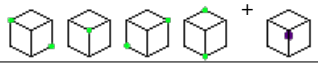
The other $24-9 = 15$ symmetries are turn-reflections and they combine a rotation and the antipodal reflection plus the antipodal reflection itself. All 48 cube symmetries are summarized in the Table I.

IV. Cube Symmetric Permutation Group

As we said in previous sections, cube's symmetric permutation group has 48 elements. For example, the right-hand rule rotations around the system axes (see Figure 1) can be encoded in permutations using the cycle notation as:

$$\begin{cases} \sigma_x = (0\ 2\ 6\ 4)(1\ 3\ 7\ 5) \\ \sigma_y = (0\ 1\ 5\ 4)(2\ 3\ 7\ 6) \\ \sigma_z = (0\ 2\ 3\ 1)(4\ 6\ 7\ 5) \end{cases} \quad (1)$$

Table I
Symmetries of the Cube

	identity
	$\pm\pi/2$ face rotation
	π face rotation
	π edge rotation
	$\pm 2\pi/3$ diagonal rotation
	axis plane reflection
	diagonal plane reflection
	$\pm\pi/2$ face rotation + antipodal reflection
	$\pm 2\pi/3$ edge rotation + antipodal reflection
	antipodal reflection

We know that σ_z can be expressed as a combination of σ_x and σ_y permutations because when rotating around the unit vector $\vec{k} = \vec{i} \times \vec{j}$, both $\vec{i}$ and $\vec{j}$ are rotated as well. We can generate the rotation permutation group using just the canonic generators σ_x and σ_y using the following algorithm.

The $GetPermutationNumber$ procedure just returns a unique number identifying the permutation (for example, the associated radix integer $r = \sum p[i] * 10^i$) where $GetPermutationName$ returns the generator name (e.g., e, x, y).

If we supply to *GeneratePermutationGroup* algorithm the $PermutationGenerators = \{\sigma_e, \sigma_x, \sigma_y\}$ it will generate the rotation symmetric permutation group, equivalent with the finite 3D rotation permutation group $SO(3)$. Note that $\sigma_e = (0)(1)(2)(3)(4)(5)(6)(7)$ is the identity permutation that leaves the cube unchanged. Its associated Cayley graph can be depicted in Figure 2. We can observe that the algorithm has found all 24 orientation-preserving permutations without providing σ_z as a generator because $z = xxyx$.

To find the other 24 permutations we just have to add antipodal reflection $\sigma_r = (0\ 7)(1\ 6)(2\ 5)(3\ 4)$ as a generator, $G = \{e, x, y, r\}$.

V. Classification of Permutations

We are making an isomorphism between the symmetric permutation of the 8 vertices of a cube and the arrangement of the density of its 8 partitioned octants. With this we can use the 48 symmetries at the octant level.

 ISBN: 978-1-61208-195-3

Algorithm 1 GeneratePermutationGroup

Require: Generators $G[?]$
Ensure: PermutationGroup $PG[48]$
  {Initialize permutation group map with generators}
  $Q \leftarrow \emptyset$
  $count \leftarrow Length(G)$
  for $i = 1 \rightarrow count$ **do**
    $n \leftarrow GetPermutationNumber(G[i])$
    $map[n] \leftarrow GetPermutationName(i)$
    $PG[i] \leftarrow G[i]$
    $Push(Q, G[i])$
  end for
  {Generate permutations in Breadth-First order}
  while $Length(Q) > 0$ **do**
    $p \leftarrow Front(Q)$
    for $i = 1 \rightarrow Length(G)$ **do**
      if $G[i] \neq \sigma_e$ **then**
        $q \leftarrow GeneratePermutation(G[i], i, p)$
        if $q \neq nil$ **then**
          $count \leftarrow count + 1$
          $PG[count] \leftarrow q$
          $Push(Q, q)$
        end if
      end if
    end for
  end while

Algorithm 2 GeneratePermutation

Require: Generator g, Generator Index i, Permutation p
Ensure: Permutation
  $n \leftarrow GetPermutationNumber(p)$
  $pit \leftarrow Find(map, n)$
  if $pit = nil$ **then**
    return nil
  end if
  $name \leftarrow Second(pit)$
  $q \leftarrow p \circ g$
  $m \leftarrow GetPermutationNumber(q)$
  $qit \leftarrow Find(map, m)$
  if $qit = nil$ **then**
    return nil
  end if
  $name \leftarrow Concat(name, GetPermutationName(i))$
  $map[m] \leftarrow name$
  return q

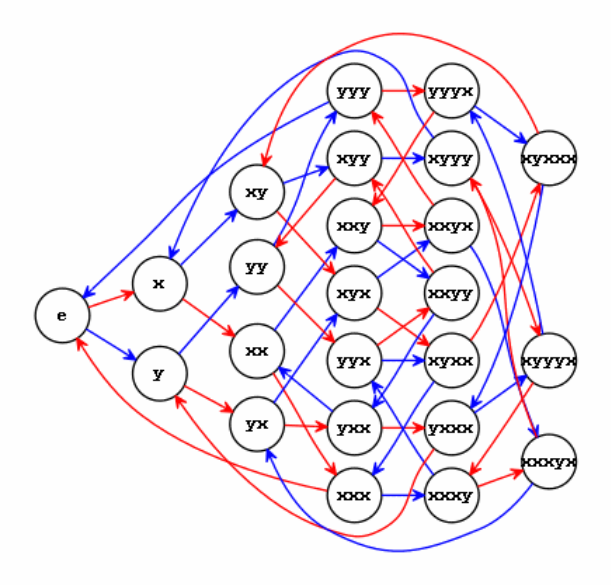


Figure 2. Cayley graph

There are $N = 8! = 40320$ possible configurations of the permutations of the order of density values for each octet $A_i, i = \overline{0..7}$. All 48 permutations form a conjugacy class so there are $N/48 = 840$ different classes.

For example the identity permutation σ_e is a permutation that corresponds to the first conjugacy class and to the ordering $A_0 \leq A_1 \leq A_2 \leq A_3 \leq A_4 \leq A_5 \leq A_6 \leq A_7$ being a 1-to-1 correspondence to the permutation canonic representation [10]: $\{0, 1, 2, 3, 4, 5, 6, 7\}$.

If we are rotating σ_e permutation along the x axis, we will get: $\sigma_x \circ \sigma_e = \{2, 3, 6, 7, 0, 1, 4, 5\}$ that corresponds to $A_2 \leq A_3 \leq A_6 \leq A_7 \leq A_0 \leq A_1 \leq A_4 \leq A_5$, which will be also in the same class.

If we do this for all 40320 permutations we will find the class for each permutation.

The class map is symmetric with itself, having the property that: $\forall i = \overline{1..40320}$, $C[i] = C[40320 - i - 1]$. This happens because the algorithm $GetNextPermutation$ (Knuth's L-Algorithm [11]) generates permutations in lexicographic order starting from $\{0, 1, 2, 3, 4, 5, 6, 7\}$ and with the antipodal reflection $\{7, 6, 5, 4, 3, 2, 1, 0\}$ the permutations will be in the same class.

The volume is partitioned using an octree generating for each level 8 octants. We must sort the octants by their average density to find the equivalent permutation but we are more interested for their initial indices than for the final sorted array. For this we are going to use a simple modified selection sort algorithm.

In Algorithm 4 we use the $GetPermutationIndex$ function (Knuth's P-Algorithm [12]) to find the index of the permutation and we use it with the pre-computed array class such that each octant will be classified.

 ISBN: 978-1-61208-195-3

Algorithm 3 GeneratePermutationClassMap

Require: PermutationGroup $PG[48]$
Ensure: PermutationClassMap $C[40320]$
  $k \leftarrow 1$
  $p \leftarrow \{0, 1, 2, 3, 4, 5, 6, 7\}$
  repeat
    $i_p \leftarrow GetPermutationIndex(p)$
    if $C[i_p] \neq nil$ **then**
      for $i = 1 \rightarrow 48$ **do**
        $q \leftarrow p \circ PG[i]$
        $i_q \leftarrow GetPermutationIndex(q)$
        if $C[i_q] \neq nil$ **then**
          $C[i_q] \leftarrow k$
        end if
      end for
      $k \leftarrow k + 1$
    end if
  until $GetNextPermutation(p) = nil$

Algorithm 4 FindPermutationClass

Require: Octants $O[8]$, PermutationClassMap $C[40320]$
Ensure: PermutationClass
  {Compute average density}
  for $i = 1 \rightarrow 8$ **do**
    $count \leftarrow VoxelCount(O[i])$
    $A[i] = \frac{1}{count} \sum_{j=1}^{count} VoxelDensity[j]$
  end for
  {Selection Sort}
  for $p = 1 \rightarrow 8$ **do**
    $min = p$
    for $i = p + 1 \rightarrow 8$ **do**
      if $A[i] < A[min]$ **then**
        $min \leftarrow i$
      end if
    end for
    $Swap(A, p, min)$
    $P[p] \leftarrow (min - 1)$
  end for
  return $C[GetPermutationIndex(P)]$

VI. Conclusion and Future Work

This paper uniquely presents how to use symmetric permutation groups to classify octree nodes into similar blocks. This classification is used to feed the Fractal Volume Coding algorithm such that the Domain-Range search will provide better distortion-curve results.

Future work will focus on speeding the domain pool search. We can extend the work of Kominek [5] to 3D by implementing a multi-dimensional r-tree indexing of the domain pool or by using other spatial data structures like the M-Tree [13]. Another interesting approach is to exploit SIMD architecture of the graphics commodity hardware available [14].

In this paper we complete our preliminary work done in Fractal Volume Coding [9] by finding a method for classifying the Domain Pool so that to have a good compression ratio. Having this done we can advance into the full implementation and have some useful experimental results. Unfortunately we do not have any experimental results to provide at this moment but a detailed description of the classification method will provide a better understanding of the method in matter.

References

[1] M.F. Barnsley, and L.P. Hurd. *Fractal Image Compression*, AK Peters, Ltd., Wellesley, Ma., 1992.

[2] M. S. Lazar and L. T. Bruton. *Fractal block coding of digital video*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 4, no. 3, pp. 297308, 1994.

[3] A. Jacquin. *A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding*, PhD thesis, Georgia Institute of Technology, 1989.

[4] Y. Fisher, E.W. Jacobs, and R.D. Boss, *Fractal image compression using iterated transforms*, Technical Report 1408, Naval Ocean Systems Center, San Diego, CA, 1991.

[5] J. Kominek, *Advances in fractal compression for multimedia applications*, Multimedia Systems, 5:255-270, Springer-Verlag, 1997.

[6] J. M. Beaumont, *Advances in block based fractal coding of still pictures.*, Proc IEEE Colloquium: The Application of Fractal Techniques in Image Processing, pp 3.13.6, 1990.

[7] D. Saupe, *Lean domain pools for fractal image compression*, Proceedings of PSIE Electronic Imaging'96, Science and Technology, Still Image Compression II, Volume 2669, San Jose, 1996.

[8] W. O. Cochran, J.C. Hart, and P.J. Flynn, "Fractal volume compression", IEEE Transactions on Visualization and Computer Graphics, Page(s):313 - 322, 1996.

[9] M. Popescu, M. Tudorache, and R. Tanasie, *Volume Content Indexing using a Fractal Coding Algorithm*, IEEE Computer Society, 2010.

[10] Donald E. Knuth, *The Art of Computer Programming*, Section 1.3.3, p.178-179, Volume 1, 3rd Ed, Addison-Wesley, 1997.

[11] D. E. Knuth, *The Art of Computer Programming*, Section 7.2.1.6, p.3, Volume 4, Fascicle 4a, Addison-Wesley, 2005.

[12] D. E. Knuth, *The Art of Computer Programming*, Section 3.3.2, p.65-66, Volume 2, 3rd Ed, Addison-Wesley, 1998.

[13] M. C. Mihaescu, D. D. Burdescu, "Using M Tree Data Structure as Unsupervised Classification Method", Informatica Journal, Ljubljana, 2011.

[14] U. Erra, *Toward Real Time Fractal Image Compression Using Graphics Hardware* Advances in Visual Computing, 2005 - Springer

 ISBN: 978-1-61208-195-3