World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:7, 2013

3D Oil Reservoir Visualisation Using Octree Compression Techniques Utilising Logical Grid Co-Ordinates

S. Mulholland and P. Cockshott

Abstract—Octree compression techniques have been used for several years for compressing large three dimensional data sets into homogeneous regions. This compression technique is ideally suited to datasets which have similar values in clusters. Oil engineers represent reservoirs as a three dimensional grid where hydrocarbons occur naturally in clusters. This research looks at the efficiency of storing these grids using octree compression techniques where grid cells are broken into active and inactive regions. Initial experiments yielded high compression ratios as only active leaf nodes and their ancestor, header nodes are stored as a bitstream to file on disk. Savings in computational time and memory were possible at decompression, as only active leaf nodes are sent to the graphics card eliminating the need of reconstructing the original matrix. This results in a more compact vertex table, which can be loaded into the graphics card quicker and generating shorter refresh delay times.

Keywords—3D visualisation, compressed vertex tables, octree compression techniques, oil reservoir grids.

I. Introduction

OIL reservoirs are large, spanning several miles horizontally but in contrast quite shallow. Seismic sampling generates data which can be computed to give a good representation of the sub-surface rock. In the case of oil field exploration, this has proved vital in developing accurate 3D geological models [1].

The past decades have seen the oil reservoir simulator as an extremely important tool for engineers for analysis of an oil reservoir's performance, its production levels and for indicating future events. The adoption of computer simulator analysis can help engineers to achieve the maximum efficiency obtainable from the reservoir [2]. Simulators allow engineers to study the anomalous behaviour of hydrocarbons and tracers through the reservoir [3].

Due to the vast volumes of these data sets, storing this data generates large file sizes. The greater the demand for accuracy, the greater the frequency in sampling, resulting in larger file sizes. The computer models generated by the simulators take the heterogeneous reservoir's information and based on various attributes fed into the simulator, accurate forecasts of oil stock and production levels can be calculated [4].

The ultimate target of the simulator is to provide oil engineers with a detailed depiction of the reservoir's physical and chemical make up modelled from the initial surveys so that accurate simulations of oil stock extraction can be run yielding effective and true evaluations and forecasts saving in time and eliminating pumping out unnecessary barrels of oil. [5].

Simulators use various techniques and therefore have varying degrees of stability and reliability which the underlying algorithms are ultimately responsible for [6]. The more accurate the model required, the smaller distance between samples. [7], suggest that the structure of the computer grid used for oil reservoir simulators dictates how precise the simulation will be owing to its interpretation of the rock formations and other attributes such as porosity and permeability. The physical dimensions and direction of the grid structure can diminish the accuracy in simulation equations. Feed-back from initial trials can lead to the re-design of both these characteristics of the grid model in an attempt to optimise simulation accuracy [8]. An actual reservoir test sample was supplied for this research by Sciencesoft Ltd, an oil and gas reservoir visualisation specialist company.[1]

II. Compression In Logical Space

A. Tree Structures

Tree structures have been well known for many years for their effective and efficient storing capabilities and their ability to accommodate fast searching of data [9]. Tree structures are hierarchical where, at its uppermost level, level-0, is the 'root node'. This node points to its child nodes where these child nodes can also have child nodes of their own and if so are referred to as 'header nodes'. If a child node does not point to any other node below it then it is referred to as a 'leaf node'.

Trees have proved to be an exceptionally effective and efficient method for storing data where the use of

S. Mulholland is a PhD student in the Department of Computer Science, The University of Glasgow, Scotland (phone: 0141-330-4256; e-mail: s.mulholland.1@research.gla.ac.uk).

Dr. P. Cockshott is a lecturer at The University of Glasgow, Scotland (e-mail: William.Cockshott@glasgow.ac.uk).

[1]http://www.sciencesoft.com/

quadtrees in 2D computer graphics has become standard practice [10]. Tree structures are used to break a multi-dimensional space into regions containing similar values using recursive programming techniques. Recursion is considered to be very elegant, generally having far fewer lines of code and thought to be more comprehensible [11].

Fig. 1 shows an example of a bitmap sub-divided into its homogeneous colour values and its resulting tree structure stored in memory. This illustrates how the image has been recursively, sub-divided into quadrants of equal colour values where leaf nodes may store the leaf node's RGB value as a payload.

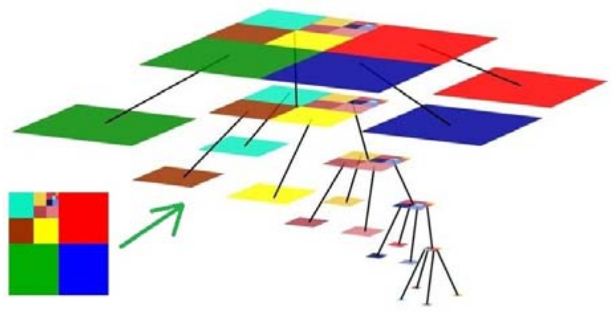

Fig. 1 Quadtree structure example

1) Octree Compression: 3D graphics can make use of octrees where regions in 3D are broken down into volume pixels called voxels [12]. With 3D visualisation, octrees can be used to represent a 3D matrix as octets of homogeneous values. This type of compression is ideally suited to datasets which have similar values in clusters. This is because these matrices exhibit larger homogeneous regions which generate shallower tree structures containing fewer leaf nodes. This not only results in smaller file sizes but as a result are quicker to traverse.

Oil engineers represent these reservoirs in a 3D grid where the volumes of hydrocarbons occur naturally in clusters. The reservoir can be first broken down into 2 distinct regions: those cells which have accessible hydrocarbons and those which do not. These are referred to as the active and inactive cells.

These engineers use a 3D integer array to represent the reservoir's active cell status in logical space where active cells are written as a '1' and inactive cells are written as a '0'. The reservoir sample supplied was a text file consisting of '1's and '0's. This represented the reservoir's active and inactive cells in 3 dimensional space in logical terms. The recursive function adopts sub-division at a power-of-2 so the original matrix is superimposed into the smallest power-of-2 3D matrix large enough to accommodate it; the file sample statistics are shown in Table I.

TABLE **I**
TEST SAMPLE STATISTICS

Oil Reservoir Sample	
Original matrix dimensions (cells)	196 x 129 x 105
New superimposed dimensions (cells)	256 x 256 x 256
Bits per cell	96
Total file size (MB)	10.13
Octree header nodes	96146
Bits per header node	32
Octree inactive leaf nodes	255621
Octree active leaf nodes	417402
Bits per leaf node	32
Total nodes stored in memory	769169
Octree saved to disk in binary format (MB)	2.93

This paper looks at compression ratios obtainable from adopting tree compression compared to current techniques and also details the savings in memory upon decompression of the bitstream. The active section of the reservoir can be replicated and visualised with no need to regenerate the original matrix, saving memory.

Octree data compression uses the same principles of compression but each header node can have up to 8 children and can be used to sub-divide 3D space. An example of an octree structure can be seen in Fig. 2 where the white cubes represent the active cells and shaded cubes the inactive cells. For demonstrative purposes only the tree structure of the top nearest cube of the matrix is illustrated.

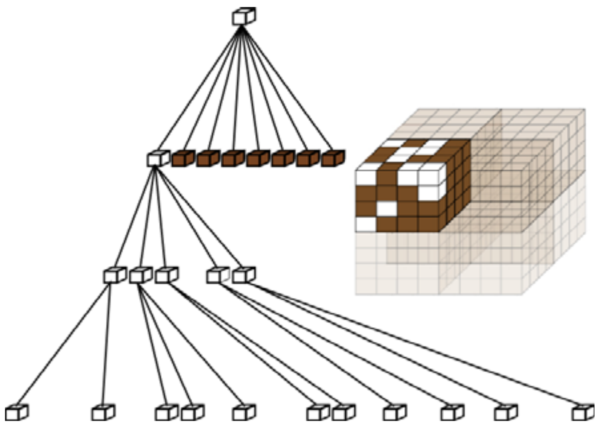

Fig. 2 Octree structure example

Following is the pseudo code used to compress the 3D array using octree compression techniques.

- Populate a 3D matrix with 1's and 0's.
- Superimpose the 3D matrix if necessary into a power-of-2 3D matrix.
- Create an octree using C# node objects linking to pointers by recursively calling the octree function passing in the starting position and thepower-of-2 size.
  - createOctree(x, y, z, cell length)
    - ∗ If cell length equals '1':

  - ∗ createLeafNode(payload).
- Else: recursively create 8 new child nodes, {child 1, child 2, ...child 8}.
- If all 8 child nodes are the same:
  - ∗ createLeafNode(payload).
- Else: createHeaderNode - pointers to, {child 1, child 2, ...child 8}.

Leaf nodes store a 32 bit word as a payload indicating whether it is active or inactive. This resulted in a file size of 2.93 MB in memory. A more efficient method of storing this payload would be to replace this by a single bit, '1' for active and a '0' for inactive and would further compress the file to 0.45 MB. In addition to this, the 32 bits required for storing header cell values was also inefficient if all that was required was an active status indicator for each of its to children. For these reasons a data structure was required which could store a precise number of bits for each node. Bitstreams have proved to be an deal data structure for performing such tasks and so was incorporated into the next stage of the compression algorithm.

B. *Compact Bitstreams*

Bitstreams allows data to be written without byte boundaries. This permits datafields which are not made up of whole bytes to be stored compactly. This data structure is therefore used as a compression technique for generating smaller file sizes where only the required bits are stored instead of unnecessary padding bits or zeros. Bits can be written to and read from the stream in whatever multiples are required. The bitstream is therefore a compressed version of the initial data [13].

The octree generated was a C# structure, where along with storing active leaf nodes and their ancestors, inactive leaf nodes and header nodes, who's descendants are all inactive leaf nodes were also stored. Inactive cells in the reservoir are of no interest to oil engineers as only those containing hydrocarbons are of importance. In order to further compress the tree before saving to file, these unnecessary nodes were culled. The resulting tree was then flattened out in a linear fashion and further compressed using a compact bitstream data structure and saved to file in binary format as described in Fig. 3.

1) Header Node Flag: An octree's header node can have up to 8 children, so a single unsigned byte flag is incorporated into the bitstream whereby each bit in the flag represents one of its individual child nodes. This is shown in Fig. 3.

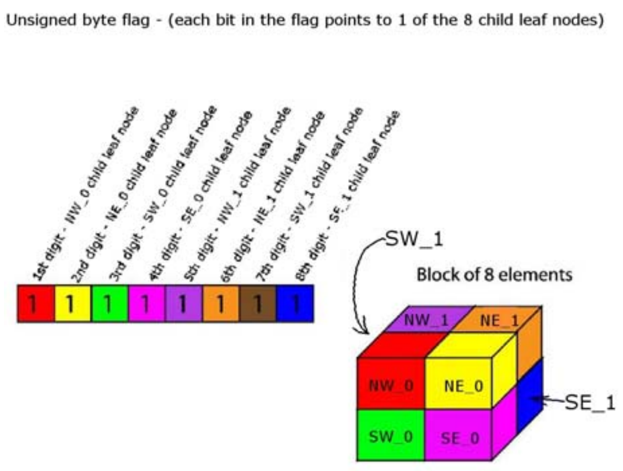
Unsigned byte flag - (each bit in the flag points to 1 of the 8 child leaf nodes)
1st digit - NW_0 child leaf node
2nd digit - NE_0 child leaf node
3rd digit - SW_0 child leaf node
4th digit - SE_0 child leaf node
5th digit - NW_1 child leaf node
6th digit - NE_1 child leaf node
7th digit - SW_1 child leaf node
8th digit - SE_1 child leaf node
1 1 1 1 1 1 1 1
SW_1
Block of 8 elements
NW_1
NE_1
NW_0
NE_0
SW_0
SE_0
SE_1

Fig. 3 Unsigned byte flag

On decompression the bitstream is read in the same traversal order as it is written. If a header node only contained 3 active cells, {NW_0, NW_1, SE_1}, then the unsigned byte flag would contain 3 '1' values and 5 '0's. This is illustrated in Fig. 4.

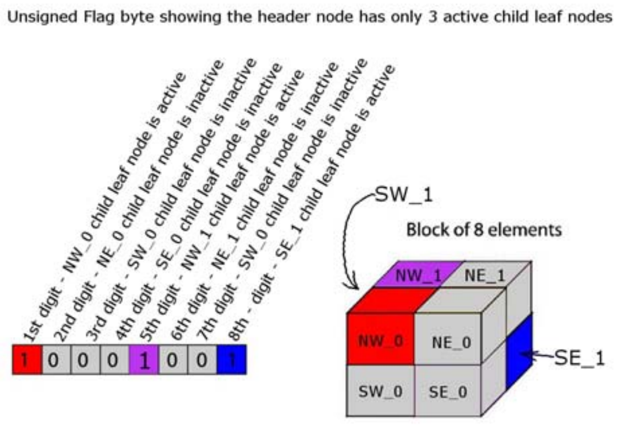
Unsigned Flag byte showing the header node has only 3 active child leaf nodes
1st digit - NW_0 child leaf node is active
2nd digit - NE_0 child leaf node is inactive
3rd digit - SW_0 child leaf node is inactive
4th digit - SE_0 child leaf node is inactive
5th digit - NW_1 child leaf node is active
6th digit - NE_1 child leaf node is inactive
7th digit - SW_0 child leaf node is inactive
8th digit - SE_1 child leaf node is active
1 0 0 0 1 0 0 1
SW_1
Block of 8 elements
NW_1
NE_1
NW_0
NE_0
SW_0
SE_0
SE_1

Fig. 4 Unsigned byte flag example

As a further method of compression any header node possessing only inactive children, indicated by an unsigned byte flag of 8 zeros, is not stored. This more compressed octree structure is subsequently flattened out as it is written to a bitstream. It contains only header nodes, their unsigned flag bytes, and active child leaf nodes. An illustration of a header node in the bitstream having 8 active child nodes is shown in Fig. 5 on the next page.

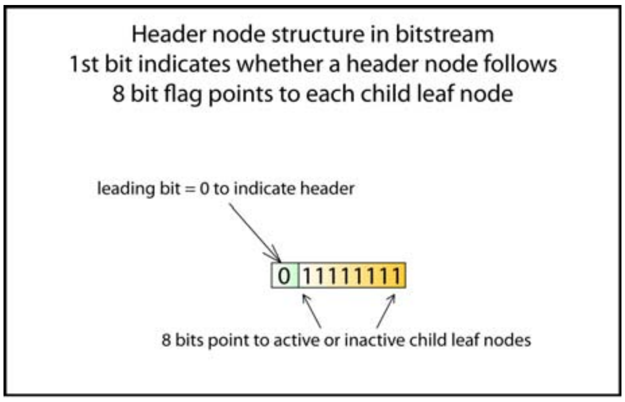
Header node structure in bitstream
1st bit indicates whether a header node follows
8 bit flag points to each child leaf node
leading bit = 0 to indicate header
0 11111111
8 bits point to active or inactive child leaf nodes

Fig. 5 Bitstream header example

A header section is added to the start of the bitstream which stores the original dimensions of the 3D matrix, power-of-2 dimensions and number of active leaf nodes. The bitstream is subsequently saved to file in binary format. Fig. 6 below shows a representation of the bitstream where the header flag indicates it has 5 active leaf node children to follow.

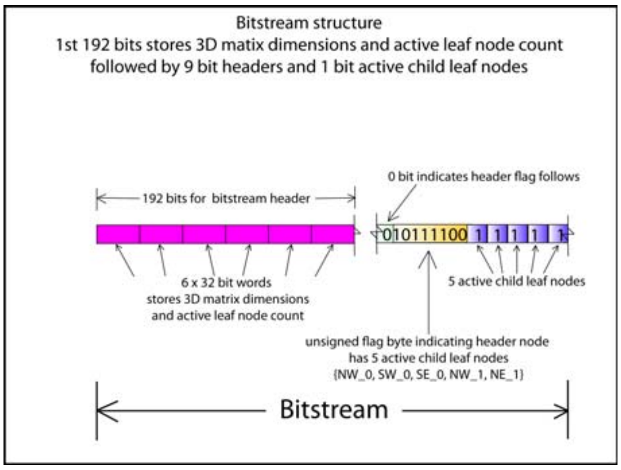
Bitstream structure
1st 192 bits stores 3D matix dimensions and active leaf node count
followed by 9 bit headers and 1 bit active child leaf nodes
192 bits for bitstream header
0 bit indicates header flag follows
010111100 1 1 1 1 1
6 x 32 bit words
stores 3D matrix dimensions
and active leaf node count
5 active child leaf nodes
unsigned flag byte indicating header node
has 5 active child leaf nodes
{NW_0, SW_0, SE_0, NW_1, NE_1}
Bitstream

Fig. 6 Bitstream snippet showing header node with 5 active child leaf nodes

Following is a pseudo code representation of the algorithm used to compress the octree data structure into the bitstream.

Traverse the octree in a depth-first-traversal fashion and write the flattened out octree to the bitstream starting at position 192 in the stream.

```
readOctree(node)
• If node is a leaf node:
    – If leaf node equals '0' it is an inactive leaf node:
        ∗ Ignore and do not write anything to bitstream.
    – Else: leaf node equals '1' and is active.
        ∗ Write a '1' to the stream.
        ∗ Increment active leaf node count.
• If node is a header node:
    – Create an 8 bit unsigned bit flag inserting a '1' for each active chilso as tod node as illustrated in Fig. 4 on the previous page.
        ∗ If all 8 bits in the header flag are zeros indicating all its child nodes are empty:
            · Do not write anything to bitstream.
• Else: Write header flag to bitstream.
```

The position of the bitstream is set to zero and the original dimensions of the 3D matrix, the power-of-2 size of the matrix and active leaf node count are written to the bitstream. The linear array is then saved to disk in binary format as a'dnsTree' file. Fig. 7 shows the octree generated from a cubic matrix of 4 X 4 X 4 cells. The cells are sub-divided into homogeneous regions and stored as an octree in memory. The white cubes are taken to be inactive cells along with any cubes which are hidden from view. The shaded cells are taken to be active cells and if the dimensions of the matrix are known, then the bitstream illustrated contains all the information required to replicate the active status of the original matrix.

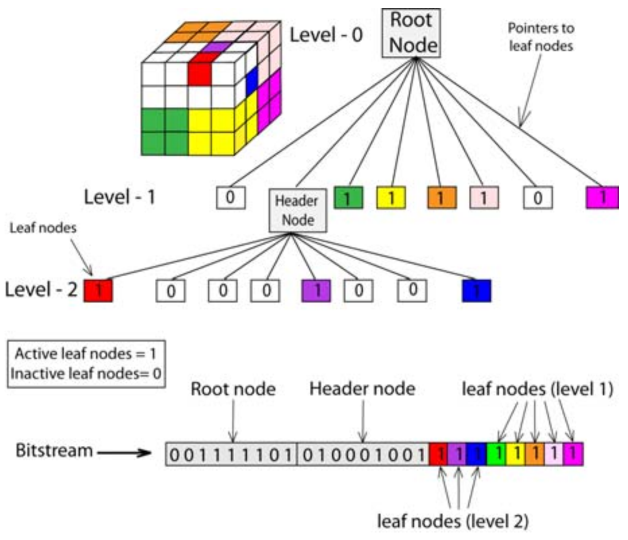

Fig. 7 Octree structure of a simple 64 cell matrix

As the original matrix stores 8, 32 bit single precision floating point values, high levels of compression can be achieved in representing this matrix in the bitstream format described. These savings are detailed in Table II.

TABLE II
64 CELL STATISTICS

Oil Reservoir Sample - (4 x 4 x 4 cells)	
Uncompressed 3D matrix file size	256 Bytes
Number of header nodes stored	2
Number of active leaf nodes stored	8
Compressed octree bitstream file size	26 bits
Compression Ratio percentage	1.27 %

III. Decompression

What has been described so far is a technique for compressing the logical structure of a simulated oil reservoir. The geometry of the original has not been saved on file. It is however possible to regenerate a logical version of the original structure in which all the cells are represented as cubes. In an actual geological model, these cells, whilst 8 sided may deviate from a cubic shape.

The compressed logical structure can be visualised on the screen using the graphics hardware. In order to visualise a 3D model on screen a 3D imaginary world is created in the computer's memory. Each object is created using triangles, each of which having their vertex positions stored as x, y and z co-ordinates. These are stored as floating point numbers in an array called a 'vertex table', so that each leaf node has 8 vertex points, equivalent to 24 floating point numbers. The vertex table can then be sent to a graphics package for 3D visualisation.

When reading back in the binary bitstream file there is no need to replicate the original 3D matrix as only the active regions of the matrix are required. A recursive depth-first-traversal method is adopted for reading the bitstream. Initially the first 192 bits are read from bitstream.

These first bits are made up of 6, 32 bit words. These words contain the dimensions of the original 3D matrix along with the power-of-2 size and active leaf node count of the octree. A linear, one-dimensional array is created and acts as the vertex table. This table stores the origin vertex position of each of the active leaf nodes along with its cell length as the number of cells it encapsulates in a single dimension. This is illustrated in Fig. 8.

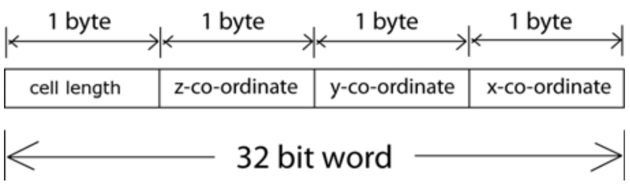

Fig. 8 Leaf node vertex structure

Following is the pseudo code used for decompressing the bitstream.

```
• Pass the starting co-ordinates, power-of-2 size, bitstream and vertex table to the recursive function.
• readFromStream(x, y, z, cell length, stream, vertexTable)
    – If first bit equal '1':
        ∗ populate the vertex table with the z, x, y co-ordinates and cell length)
    – Else: Read the following 8 bit header flag and for each bit in the bit flag which equals a '1'recursively read from the bitstream. For example, if the first bit in flag equals '1':
        ∗ recursively call the NW_0 child node.
        ∗ readFromStream (x, y, z, cell length / 2, stream, vertexTable).
        ∗ populate the next position of the vertex table with the cell length, z, x and y co-ordinate values.
• At the end of the stream, send the vertex table to the graphics card for visualization.
```

IV. Visualisation

A. *Compression of Information*

The surface of the leaf node can be depicted by drawing 2 triangles using the origin co-ordinates and the cell length value. This is accomplished by plotting each subsequent vertex perpendicular to one-another other, forming cuboids. Although these vertex positions are based on logical positions, the resulting visualisations are of an acceptable standard for the human eye. This is because these models can contain millions of cells, sometimes more than the pixel count on most computer screens. The graphics card renders the entire scene using the frame buffer and displays it on screen. The vertex table holds each of the active leaf node's origin x, y and z co-ordinate values and cell length as illustrated in Fig. 8.

The graphic card renders each frame typically, between 60 and 100 times a second and is referred to as the refresh rate [14]. If the refresh rate is set too high, a proportion of the frames will be identical and not updated as this allows the graphic card to catch up. Smoother animations sometimes require slower refresh rates and waiting functions written into the code [15].

V. Results

As the bitstream only contains the information of each active leaf node and parent header nodes, a high level of compression can be achieved. The sample, reservoir test file used for evaluations, has an aspect ratio of 196 x 129 x 105 cells producing a 3D matrix of 2654820 elements. It has an active percentage of 56.28% and its level of clustering was deemed typical by Sciencesoft Ltd. Table III details its octree header node and leaf node values.

TABLE III
TEST SAMPLE OCTREE STRUCTURE

Oil Reservoir Sample - (196 x 129 x 105 cells)		
Header nodes	Active leaf nodes	Total cells stored
96146	417402	513548

The compression ratio generated from this test sample is detailed in Table IV.

TABLE IV
TEST SAMPLE FILE SIZES AND COMPRESSION RATIO

Oil Reservoir Sample - (196 x 129 x 105 cells)	
Uncompressed 3D matrix file size	10.13 MB
Compressed octree bitstream file size	156.6 KB
Compression Ratio percentage	1.5%

A vertex table can be generated from the bitstream whereby a single 32 bit word is sufficient to represent an active leaf node's surface area in logical terms irrespective of the number of active cells it encapsulates. The sample had 1494128 individual active cells out of a total of 2654820. A vertex table generated from each of these individual cells would normally be stored as 3, x, y and z co-ordinate, single precision floating point numbers. The savings in vertex table size is detailed in Table V.

TABLE V
TEST SAMPLE'S VERTEX TABLE MEMORY SAVINGS

Oil Reservoir Sample - (196 x 129 x 105 cells)	
Uncompressed vertex table	136.79 MB
Compressed octree vertex table	1.59 MB
Compression Ratio percentage	1.16 %

In addition to these savings in memory, the graphic card does not have to perform as many calculations to replicate the model. The graphic card draws 12 triangles to form an active cuboid but as a single octree leaf node can represent multiple active 3D matrix cells there is a substantial saving in graphic card calculations. The results of this can be viewed below in Table VI.

TABLE VI
SAVINGS IN GRAPHIC CARD TRIANGLE CALCULATIONS

Oil Reservoir Sample - (196 x 129 x 105 cells)	
Triangles in uncompressed vertex table	17929536 ~ 1.7 10^7
Triangles in compressed vertex table	5008824 ~ 5.0×10^6

Every face drawn by the graphics card comprises of 2 triangle calculations.The savings in triangle calculations are illustrated below in Fig. 9 where a typical octree leaf node encapsulates 64 cells. As every cell within the leaf node is considered to be similar there is no need for the graphics card to draw the internal unexposed cell faces, instead only the 6 faces of the leaf node are drawn. There is therefore a substantial saving in required triangle calculations. This is illustrated in Fig.9 where 768 triangles are required to draw all the cells, whereas, using the octree compression algorithms, only 12 are required.

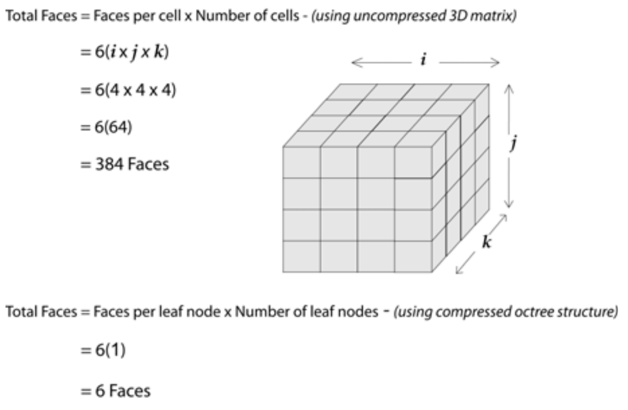
Total Faces = Faces per cell x Number of cells - (using uncompressed 3D matrix)
= 6(i x j x k)
= 6(4 x 4 x 4)
= 6(64)
= 384 Faces
i
j
k
Total Faces = Faces per leaf node x Number of leaf nodes - (using compressed octree structure)
= 6(1)
= 6 Faces

Fig. 9 Reduced face calculations of Leaf nodes

The savings made by only generating a vertex table based on the number of active leaf nodes not only occupies less memory but can be loaded into the graphics card quicker. In addition to this, this smaller vertex table allows the graphics card to refresh quicker resulting in less of a delay.

TABLE VII
LOADING AND GRAPHICS CARD TIME

Vertex Table Style	Loading Vertex Table Into Graphics Card (seconds)	Graphics Card Refresh Delay (milliseconds)
Uncompressed	2.49	119
Octree Compressed	0.89	14
Compression Ratio - (%)	35.7	11.8

VI. Conclusions

The greater the density of clustering of active cells within the oil reservoir the lower the entropy. This results in a shallower and less dense octree structure required for its storage. As the oil reservoir's octree representation only contains those active leaf nodes and their ancestors, oil reservoirs of low activity can be compressed to an even greater extent. This is partly because the entropy of the oil reservoir would still be low and the majority of the leaf nodes would be discarded inactive leaf nodes. The resulting octree generated would therefore be shallow and sparse resulting in a very compact binary file.

Oil reservoirs displaying high levels of entropy may not prove to be a suitable candidate for octree compression techniques. This is primarily due to the number of leaf nodes and levels required to represent it. The octree generated from such a reservoir would be deep and dense resulting in large file sizes. It is also worthwhile pointing out that reservoirs displaying high levels of entropy may also prove to be insufficiently clustered due to their lack of active cell continuity and therefore deemed unsuitable for oil extraction by engineers.

It can also be deduced that the more cells contained within the leaf node, the greater the saving in triangle calculations. Additionally, the more dense the clustering of active cells within a reservoir, the faster, its more compact vertex table can be loaded into the graphics card and ultimately visualised with shorter refresh delays.

References

[1] R. McCauley, *Marine seismic surveys: a study of environmental implications*. Australian Petroleum Production and Exploration Association, 2000.

[2] P. Samier, "Reservoir simulation in the oil industry," *APOS-EU*, vol. 1, p. 1, July 2011.

[3] D. G. Donato, E.-O. Obi, and J. M. Blunt, "Anomalous transport in heterogeneous media demonstrated by streamline-based simulation," *Geophysical Research Letters*, vol. 30, no. 12, pp. 1–4, 2003.

[4] C. Zhang, A. Bakshi, and V. K. Prasanna, "Data component based management of reservoir simulation models," in *Information Reuse and Integration, 2008. IRI 2008. IEEE International Conference on*, 2008, pp. 386–392.

[5] J. R. Fanchi, *Fundamentals of Reservoir Simulation*. Burlington: Gulf Professional Publishing, 2006a, pp. 162–186, doi: 10.1016/B978-075067933-6/50012-X.

[6] *Conceptual Reservoir Scales*. Burlington: Gulf Professional Publishing, 2006b, ch. 12, pp. 210–232, doi: 10.1016/B978-075067933-6/50014-3.

[7] J. E. Aarnes, V. Kippe, and K.-A. Lie, "Mixed multiscale finite elements and streamline methods for reservoir simulation of large geomodels," *Advances in Water Resources*, vol. 28, no. 3, pp. 257–271, 2005, doi: 10.1016/j.advwatres.2004.10.007.

[8] J. Yu and H. Sun, "Influence analysis of calculation error of reservoir numerical simulation by direction and size of grid," *Flow in Porous Media - from Phenomena to Engineering and Beyond*, vol. 1, pp. 152–156, 2009.

[9] J. Bonet and J. Peraire, "An alternating digital tree (adt) algorithm for 3d geometric searching and intersection problems," *International Journal for Numerical Methods in Engineering*, vol. 31, no. 1, pp. 1–17, 1991.

[10] M. Manouvrier, M. Rukoz, and G. Jomier, "Quadtree representations for storage and manipulation of clusters of images," *Image and Vision Computing*, vol. 20, no. 7, pp. 513–527, 2002.

[11] A. Filinski, "Recursion from iteration," *LISP and Symbolic Computation*, vol. 7, no. 1, pp. 11–37, 1994.

[12] G. Favalora, R. Dorval, D. Hall, M. Giovinco, and J. Napoli, "Volumetric three-dimensional display system with rasterization hardware," in *Proc SPIE*, vol. 4297, 2001, pp. 227–235.

[13] D. Salomon, *Data Compression*, 4th ed., W. Wheeler, Ed. London: Springer-Verlag, 2005.

[14] E. Angel and S. Dave, *Interactive Computer Graphics A Top-down Approach With Shader-based OpenGL*, 6th ed., M. Hirsch, Ed. Pearson, 2012.

[15] D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Addison-Wesley Professional, 2010, vol. 1.