# 3D Visualisation of Oil Reservoirs


Samantha Mulholland

Submitted in fulfilment of the requirements for the

Degree of Master of Philosophy

School of Computing Science

College of Science and Engineering

University of Glasgow

28th March 2017

## Abstract

*This research introduces a novel approach to storing compressed 3D grid information by applying octree compression techniques. This new data structure stores the octree in a pruned flattened fashion where only header and active leaf nodes are stored in a linear array. This generates high levels of lossless compression when applied to 3D geometry where clusters of homogeneous information exist. This data structure yields fast, log(n) look-up times and initial results show that when coupled with bespoke scanning methods searching times can surpass that of direct access. Hierarchical pyramid visualisations techniques are also presented using the information stored at each level in the tree structure. Integrating with this are face culling algorithms developed in this research, which eliminate hidden face and inner leaf node cells which eases the burden placed on the CPU and GPU. By integrating these pyramid scaling and face culling algorithms, grid models can be shown at various levels of resolution incorporating sub-regions, "regions of interest" displayed at full resolution. This further lightens the load on the GPU generating quicker loading times and higher refresh rates. This can potentially allow larger models to be visualised than would otherwise have been possible.*

*This research was sponsored by Sciencesoft an oil reservoir visualisation company and the algorithms developed in this research have been applied to compressing oil reservoir information. Oil companies require accurate 3D computer-generated models of oil reservoirs in order to make oil and gas extraction as cost effective as possible. Advances in computing power has meant that it is now possible to run multi-million cell oil reservoir grid models, increasing the level of accuracy and precision available to engineers. This thesis applies 3D octree compression techniques to these computer models and compares these with industry standard storage and cell searching algorithms as industry benchmarks. This thesis suggests that octree compression techniques may prove to be a more efficient data structure for storing and searching active cell information within oil reservoirs than existing procedures.*

# Contents

## Deliverables

A CD accompanies this thesis and contains the following folders:

- Excel – the spreadsheets with experiment results.
- Tree structures – a set of text documents detailing the tree structures of each of the 36 test grids supplied by Sciencesoft and the filled versions used for comparisons.
- Images – a series of screenshot images of the demo grid given in the chapters 6 and 7.
- Movie files – animations from the human liver MRI scan set used in chapter 9.

## Terms of Reference

This research was conducted by Samantha Mulholland while attending the School of Computing Science at The University of Glasgow from 2011 – 2017. The work in this thesis initially looks at oil reservoir engineering techniques used currently, including a literature review of current compression techniques. It details the research conducted, software developments and chosen methodologies adopted in order to prove the research hypotheses. Results and conclusions are given for the new compression, visualisation and scanning techniques followed by a discussion on how oil reservoir models can be displayed at different levels of detail before concluding with a future work chapter.

The research presented in this thesis has been commissioned by Sciencesoft Ltd, a 2D and 3D oil reservoir visualisation specialist company based in Govan, who supply software to over 140 companies in over 80 countries. Although the author had free-choice in the design of any data structures there were some contextual restrictions. The programming language C# had to be used to allow integration with existing applications, and the data structures had to fully integrate with an existing API.

**Author's Declaration**

"I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Signature _____

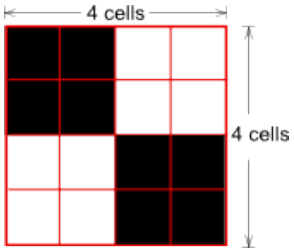
Printed name _____ "

## Definition of Terms

This section explains terminology used in the document.

- Oil Reservoir – This is a volume of rock containing oil and gas which is drilled to extract these hydrocarbons.
- Reservoir Model – This is a computer model used for simulation and visualisation purposes to help engineers better understand the various dynamics of an oil reservoir represented by an array of sub-divisions in 3D space (grid cells).
- Grid Cell – These are 3D volumes of rock in the reservoir model.
- Active cells – These are the cells in the grid which represent volumes of rock containing sufficient quantities of oil or gas to warrant inclusion in the simulator model. There natural cell index is stored in an array called the A2N.
- Inactive cells – These are empty cells in the 3D grid and are sometimes discarded from calculations and visualisations as they represent volumes of rock which do not contain hydrocarbons.
- Natural Grid – The simulation grids studied in this thesis are structured grids as $Nx \cdot Ny \cdot Nz$ cells formed in a raster-order ($x$ then $y$, then $z$).
- Natural Index Order – This is the numbering system used which defines each cell position within the natural grid model. The numbering starts at zero and increments whilst raster scanning.
- N2A – The N2A array is indexed by the natural cell number and returns the index of the corresponding cell in the A2N. Figure 1 on page x shows a 2-dimensional representation of an N2A array. It shows a grid of 4 x 4 cells. There are 8 inactive black cells and 8 active white cells. The natural cell indices are given for each cell in the right had grid. The cells are indexed using a raster scan order with $x$ varying fastest then $y$.

The N2A generated from this grid is {0 ,0, 1, 2, 0, 0, 3, 4, 5, 6, 0, 0, 7, 8, 0, 0} and is shown in Figure 2.




| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

Figure 1: Left: a 4 x 4 matrix of active and inactive cells and cell index positions – (black = active : white = active). Right: the natural grid position of the cells

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 2 |
| 0 | 0 | 3 | 4 |
| 5 | 6 | 0 | 0 |
| 7 | 8 | 0 | 0 |

Figure 2: N2A representation of the 4 x 4 cell grid

- A2N – This is the Active-to-Natural linear array. This array is used to reference active cells to their original positions in the Natural index order. This is often used to obtain various property values of an active cell. It is a more compressed data structure than the N2A as it only stores index values for active cells. Referring back to the Figures in 1, the A2N array stores the natural cell positions of the active cells: {2, 3, 6, 7, 8, 9, 12, 13}. The A2N is used to relate elements in the property array (pressure values, saturation levels, etc.) to grid cells in the natural grid order.
- The A2N array points from the array of objects which holds the required hydrocarbon characteristics of each of the active cells to the corresponding grid cell natural index.
- Direct Access – This is the term used to describe accessing elements from arrays by just directly calling its value: ( int value = myArray[x,y,z] )
- Power-of-2 – This is the term used to define squares or cubes which have sides equal to powers of 2 values.
- Binary Tree – These are trees which can have up to two child nodes and are ideal for storing and searching 1D systems where a linear stream of data is sub-divided in half recursively.

- Quadtree – Quadtrees can have up to four child nodes and are ideal for storing 2D information, such as images, in a compressed state. It does this by sub-dividing the 2D space into four sub-divisions (quadrants) recursively until all quadrants contain homogeneous values (leaf nodes). In this thesis quadtree compression was used to compress images representing slices through an oil reservoir.
- Octree – Octrees can have up to eight child nodes and are ideal for storing 3D information, such as 3D arrays, in a compressed state. It does this by sub-dividing the 3D space into eight sub-divisions (octants) recursively until all octants contain homogeneous values (leaf nodes). In this thesis octree compression was used to compress 3D oil reservoir grid active cell information.
- Root node – This is the top of the tree and is normally where all traversing methods start.
- Header nodes – These nodes have pointers to child nodes which may be header nodes themselves or leaf nodes.
- Leaf nodes – These are nodes which have no children. In the context of this thesis they represent homogeneous regions of cells sharing the same active status.
- Voxel – volumetric pixel element such as a cuboid.

**Thesis Statement**

Diagenetic processes present during an oil reservoir's formation causes hydrocarbon soaked rock to naturally form in clusters. Oil reservoir simulators sub-divide these grids into grid-blocks in a Cartesian grid where each grid-block represents a volume of rock within the oil field those containing hydrocarbons, active cells, and those which do not, inactive cells. Applying the data structures developed in this research and exploiting the phenomenon of naturally clustering characteristics present in oil reservoirs, we assert:

- The compressed tree structure will yield greater compression ratios than is obtainable by industrial techniques.
- By implementing the data structure described, loading and data accessing speed will surpass what is at presently considered state-of-the-art.
- For the purposes of this research state-of-the-art is considered to be what was previously achievable by Sciencesoft.
- Applying the bespoke scanning algorithms developed in this research to this domain yields quicker active cell lookup times than the state-of-the-art direct access methods.
- GPUs can only store a limited number of vertex positions, far less than those required to represent multi-million cell grids, but with the adoption of the hierarchical pyramid scaling methods presented in this research, larger grids can be visualised.
- By integrating tree node information with high resolution cell information, oil reservoirs can be displayed whilst loading detailed information only of "regions of interest".

## Hypotheses

The following two hypotheses were developed for this research:

1. Octree compression will prove to be a more efficient method for storing oil reservoir 3D active cell information.
2. Cell lookup times will prove to be quicker using recursive traversal methods with the octree representation than direct access methods.

In order to test the hypotheses made in this thesis the intermediate aims and objectives were:

1. Build test programs in C# to generate and test the data sets used in this thesis (octree and bitstreams) in a lossless fashion.
2. Perform detailed memory and performance experiments using these datasets and comparing and contrasting results with industry standard direct access benchmarks.
3. Develop a methodology for gauging grid entropy.
4. Generate a hierarchical pyramid visualisation technique where visualisations are generated using tree nodes from each level in the octree.
5. Develop a face culling algorithm which takes into account the various visualisation options (whether active or inactive cells are to be displayed) and is based on logical and geological information (grid co-ordinates and vertex positions).
6. Develop an algorithm which can represent geological faults within simulated grids.

# Chapter 1

# Background

Meeting world demand for energy has led to fossil fuels accounting for over 80% of the world's fuel market (Escobar *et al.* , 2009) and with today's new oil extraction methods has become an extremely complex business. Wealthier countries, with wealthier citizens demand more energy to meet their energy demands with China and the United States of America being the largest consumers.Improved standards of living and the rising human population means that meeting future energy undoubtedly increases the need for more cost effective practises to be developed. It is estimated that world energy demands will double over the next twenty years when fossil fuels will still supply over four fifths of global energy needs (Publishing & Agency, 2007; Shafiee & Topal, 2009). The current 2% growth in the world's population will only serve to increase human energy consumption demanding increased energy production levels (Azarpour *et al.* , 2013).

War zone countries such as Syria are estimated to have around two and a half billion barrels of oil but the wells are unkempt and production has almost ceased due to the collapse of the country's infrastructure and abandoned refineries (Leonard, 2013). Making these reservoirs profitable again requires oil companies to develop tailored cost effective oil extraction techniques.

There are two main types of oil extracted from reservoirs, conventional oil (liquid) mined using traditional methods which can easily pass through the rock pores by means of natural reservoir pressures and unconventional oil (heavy thicker oil) mined using less traditional methods due to its higher levels of viscosity which clogs up rock pores sometimes containing oil shale and shale rock (Tham, 1976). Sometimes the bending and faulting of the oil reservoir rock can force these traps up above ground or sea bed level (outcrops). Figure 1.1 depicts an illustrative cross-section through an oil field showing the tapering and bending lithology (Nikolaevskiy, 2005) of oil reservoir rock with various traps and outcrops. Once extracted the

hydrocarbons are either piped to on-shore refineries or shipped using large oil-tankers.

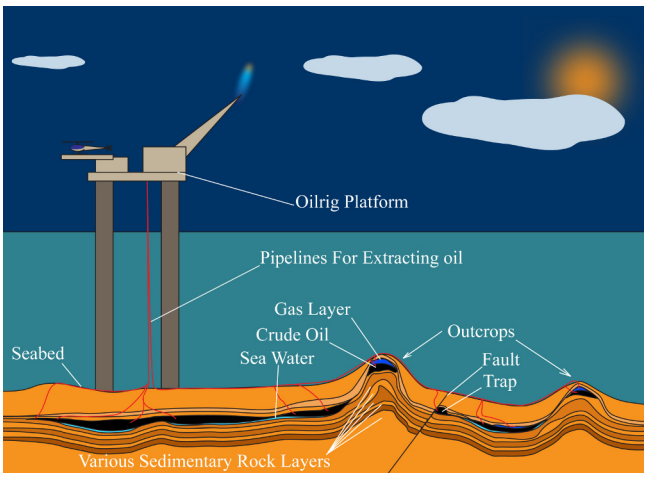



Figure 1.1: Schematic cross-section through an oil reservoir

## 1.1 Oil Recovery Today

Around three quarters of the Earth's oil reserves are found in the Middle East nations such as Iraq and Iran. Most of these nations are members of the Organisation of the Petroleum Exporting Countries (OPEC)[1], formed in 1960 who regulate the export price and production rates of oil in an attempt to make the market place less volatile. There are other energy sources such as renewables and many countries today foresee these alternative power sources playing a major role in meeting future energy needs (Peidong *et al.* , 2009) reducing the effects of global warming (Ku & Yoo, 2010). Although some are considered 'clean', (Parkes, 2012) generating zero carbon emissions such as with wind turbine technology they have limitations and disadvantages such as during unusual climatic conditions (Ringel, 2006). Others are sometimes viewed as unsafe especially after accidents such as in Japan 2011 at the Fukushima nuclear power plant (Zuiderveld & Viergever, 1992). Nuclear power has the drawback of generating extremely hazardous controversial radioactive by-products (Emmanuel & Baker, 2012), but these renewables still account for around a sixth of the world's energy needs. Even the safest of renewables such as those harnessing the power of the wind or the Sun can suffer from a lack of sunlight or unpredictable climatic conditions.

By-products from crude oil are used to form numerous products and plastics such as electrical casings, cable insulation, ink, soaps, shampoos, tyres, grocery bags and make-up and the world's increasing population increases their demand. One day these natural resources will be exhausted, so it is imperative to efficiently and effectively extract optimum levels of

[1] http://www.opec.org/opec_web/en/index.htm

oil from today's reservoirs. Advancements in oil recovery methods can sometimes make it profitable again to re-mine reservoirs once abandoned due to limitations of previously available oil extraction methods where they were not deemed profitable due to the complex and costly mining procedures available at the time.

## 1.2 Fossil Fuel

The term fossil fuel includes coal, natural gas and crude oil. The hydrocarbons which make up crude oil can be refined into various fuels and products such as petrol, diesel and bitumen. Around 50% of the world's energy usage is apportioned to the industrial sector and just over a quarter used for transport (aeroplanes, trains, cars etc). The commercial sector uses around a 7$^{th}$ of the world's total energy consumption whereas approximately 7% is generated for residential needs (Tanguay-Carel, 2013). Fossil fuels were formed millions of years ago and are made up of decomposed layering of organic matter, from foliage on land and micro-organic life forms over millions of years. Pressure from rock pushing down on these deposits compresses the trapped organic matter, slowly changing its form creating oil and gas.

Layering occurs as the rich decomposing organic matter gets sandwiched between impermeable layers of mud. These impervious layers overtime become dense, lacking pores and preventing the passage of fluids or gas, known as shale rock. The organic matter breaks down and with pressure and heat turning the prehistoric sludge into hydrocarbons. These hydrocarbons impregnate the pores of the porous rock layers (source rock) and this encapsulated, trapped oil soaked rock is the oil reservoir. Various forms of fossil fuel deposits are formed through a variety of different environmental factors and the formation of oil from original organic matter is known as diagenetics (McLimans, 1987; Roedder & Bodnar, 1980).

## 1.3 Oil Reservoir Formation

After sediments have been deposited, gravitational forces and the weight of overbearing material compress the trapped matter into 'pockets' and layers forming rock over millions of years, typically sandstone or limestone. The vast majority of oil fields were formed around 200–250 million years ago, spanning the Triassic, Jurassic and Cretaceous Eras, about a 5$^{th}$ around 65 million years ago (Cenozoic Era) and some as recent as 500 million years ago (Paleozoic Era). Forces act upon the rock where the greater the compression the denser the rock and the less porous and permeable the rock becomes (Ahmed, 2010). Oil reservoirs

can be hundreds of meters horizontally but usually far thinner in comparison, although still uncommonly reaching a thickness of over a 100 meters (Alsharhan & Whittle, 1995).

Greater Burgan situated in Kuwait is the world's largest sandstone oil reservoir. This land-based reservoir consists of a cluster of three large oil fields (Burgan, Magwa and Ahmadi) with an accumulated surface area of 1000 $km^2$ . Ghawar oil field in Saudi Arabia is one of the largest carbonate based oil reservoirs and spreads 174 by 16 miles horizontally and less than a $12^{th}$ of a mile in depth.

## 1.4 Oil Reservoir's Life-cycle

When a potentially financially viable site for oil recovery has been found, drilling will commence and oil extracted. During the reservoir's productive lifetime the extraction methods applied change to suit its unique natural characteristics. Monitoring helps maximise its potential allowing oil to be recovered in the most cost-effective manner. Different methods of extraction are applied as the oil field reaches maturity and can be thought of as the oil reservoir's life-cycle and can be split into three distinct stages:

- Primary Recovery – due to the natural pressures which exist within a reservoir, hydrocarbons can pass easily through the rock pores and recovered.
- Secondary Recovery – as oil is expelled during the primary recovery stage, reservoir pressure drops to a level where without human intervention oil extraction would not be possible.
- Tertiary Recovery – as the oil reservoirs mature and near the end of their life-cycle, various Enhanced Oil Recovery (EOR) methods are used to extract remaining reserves, such as polymer flooding and microbial injection.

When oil reservoirs are first drilled, natural pressures generated from heat and compression from material above, force the hydrocarbons trapped within the rock pores to the wellbore out to the surface. The primary stage typically only extracts around 10 – 15% of reservoir's oil.

The secondary stage (typically extracting 15 – 45% of the oil) sees the need to artificially increase the pressures, normally through injecting sea water at high pressures (typically with oceanic reservoirs due to its ready abundance). Pressure levels are carefully monitored avoiding rock fracturing within the reservoir (which can alter porosity and permeability levels) (Gerritsen & Durlofsky, 2005) and discontinuity in the flow of hydrocarbons – 'reservoir anisotropy' (Krogstad & Skare, 1995). The remaining oil is more expensive and time consuming to extract as it clings to the rock but the reservoir may still contain around 50% of its initial

reserves. The tertiary stage uses an array of EOR (Enhanced Oil Recovery) techniques and chemicals such as surfactants, foams and $CO_2$ injection processes to extract more oil .

## 1.5 EOR

Instead of selling less productive fields, large companies are now applying new EOR methods and techniques (Wood, 2013a). Around three quarters of the planet's oil fields are mature and "watered-out" where oil extraction was previously considered too costly and complicated (Donnez, 2007) – increased levels of investment in research is paramount in meeting the demands of the petrochemical industry such as simulating billion cell models (Wood, 2013).

There are numerous approaches taken to extract oil from mature oil reservoirs, most commonly injecting high pressure miscible gases (propane, methane, $CO_2$ and nitrogen) reducing the viscosity of the oil making it flow more easily(Sohrabi *et al.* , 2008). Thermal enhanced oil recovery methods (TEOR) introduces heat (typically as steam) lowering the oil's viscosity. Chemical injection (using polymers and surfactants) act as thickening agents and detergents – polymers thicken the water to a viscosity equalling that of the oil to force the oil through volumes of rock avoiding 'fingering' and surfactants thin the oil making it less sticky. Microbial enhanced oil recovery methods (MEOR), inject the reservoir with organisms naturally found in the reservoir, or the nutrients they feed upon to help thin the oil. New schools of thought see EOR, not left as an afterthought, or to latter stages in the reservoir's life-cycle, but instead considered earlier such as adding mineral deposits during the water injection phase reducing the cost and need of future EOR methods (Kokal & Al-Kaabi, 2010).

## 1.6 Simulation

Reservoir simulators are extremely important for analysing an oil reservoir's behaviour (Samier, 2011). Computer models are developed for studying the behaviour of reservoir hydrocarbons and for estimating recoverable reserves. They help engineers determine the most cost efficient means of managing their assets and enhance their conceptual understanding of the reservoir. By altering various reservoir properties, engineers can predict future production levels. Engineers are only ever concerned with simulating the reservoir they are working with and studying these simulations better equips them to optimise production. Accurate simulations provide reservoir engineers with a detailed depiction of the reservoir's physical, chemical and biological structure helping to reduce wastage (Fanchi, 2006a).

Data input into the reservoir simulator model consists of many different reservoir characteristic scenarios as the reservoir cannot be seen and its structure can only be at best, estimated. Results from these simulations are compared and contrasted against past production levels and events to develop a model which closely matches the actual reservoir structure. Simulation results can be extremely complex and time consuming to interpret and engineers require information as quick as possible. Oil reservoir visualisation software is used to interpret the simulated output results as they are often extremely quick to understand, sometimes just by looking a computer model, for example a sudden change in cell colour in the visualisation may indicate a fault but it could have taken days to find this by merely studying lists of values.

## 1.7 History Matching

The computer models generated from simulations take the reservoir data and based on various attributes fed into the simulator, accurate forecasts of reserves and production levels can be calculated (Zhang *et al.* , 2008). Simulators use various techniques and have varying degrees of stability and reliability based on their underlying algorithms (Fanchi, 2006b). History matching is a technique which compares simulated results with observed data and are performed as standard practice as reservoir models become increasingly intricate and time consuming to compute, multiplying the possibilities of inaccuracies (Schulze-Riegert *et al.* , 2004). Adjusting various attributes of the reservoir model such as the permeability can yield several possible alternate forecasts (Norgard, 2006; Northrop & Timmer, 1995) each viable options. Reservoir engineers accept that the most appropriate match is not a mirror image of the predicted data as different input parameters give different simulated results (Yang & Watson, 1988). They select what input parameters and forecasts are most accurate due to their familiarity with the reservoir. Once the input parameters yield production rates closely matching historical data the simulator is said to be matched (calibrated) and the model can be used to give future predictions (Janoski & Sung, 2001).

## 1.8 Reservoir Data

An oil reservoir's diagenetic history is a geological view of how the reservoir was formed. It includes the period of time it took to create, the sediment it was formed from, the forces which compressed it into rock and the climatic conditions which were present during its creation. It is of the utmost importance to geologists in the oil exploration field as they need to know its sub-rock structures and minerals which formed it, from initial deposits of sediment to today's

compressed rock (Fjaer *et al.* , 2008) – the dynamics and influences imposed on it through time affect how it presently behaves. Collating the data from various techniques, such as seismic testing and detailed logging analysis help engineers optimise their resources such as knowing, where best to drill, the rock's structure, expected gas pressures, saturation levels and possible production levels (Gutierrez *et al.* , 2008) and can include:

- Seismic Mapping – detailed contour maps, layering and composition of the seabed can be deduced using reflected sound waves generated from jets of compressed air fired at it (McCauley, 2000; Ramberg, 2008).
- Core Sampling – core samples of rock are drilled out of the reservoir and sent to laboratories for analysis. Different rock layers are clearly distinguishable in these tube-like sections and are matched to the seismic data indicating possible saturation levels, crystal composition, mineral and grain size (Stegemeier & Perry, 1992), although levels fluctuate throughout the reservoir's entirety where only 'pockets' of similar consistency exist (Porges, 2006).
- Logging – this uses probes lowered into the core sample cavities using different sampling techniques (gamma rays, thermometers and electrodes) to measure pressure, temperature, direction of flow and chemical composition (Mohaghegh *et al.* , 1996; Li & Jinliang, 2010; Elsharkawy, 2003).
- Outcrops – forces in rock layers can be bent where faulting causes sections of reservoir rock to be forced upwards and protrude above ground or sea-bed level.
- Fractures – These are cracks running through the rock in all directions allowing hydrocarbons to seep through more readily (Barenblatt *et al.* , 1960) they are more prevalent in carbonate reservoirs due to their tendency to fracture and move (closing, dilating and shearing) (Council, 1996).
- Porosity – The fraction of space or holes (pores) within the grain of rock referred to as its absolute porosity (Porges, 2006). Oil can only be extracted when these pores are all interconnected allowing the uninterrupted flow of hydrocarbons. Dividing the quantity of connected spaces within the rock by the overall bulk volume gives a truer level of the rock's porosity (effective porosity) and can be estimated in laboratory tests by saturating core samples at zero pressure with a fluid at a given density (Ahmed, 2010).
- Permeability – the ability for the rock to allow hydrocarbons to pass through it. The interconnecting network of pores, their size and the size of the grains and cavities are used to accurately evaluate how much oil can be extracted from the reservoir (Northrop & Timmer, 1995; Beach *et al.* , 1999; Meadows, 1997) estimated initially in laboratories by pushing oil through core samples. This can only give an average permeability (equivalent permeability) value as it does not take into account the actual varying irregular connectivity and porosity levels which exists throughout the oil reservoir's entirety and how the different rock layers interact and connect to one-another (Durlofsky, 1991).

## 1.9 Simulation Fundamentals

Computer performance has greatly improved with advances in computer technology leading to more precise 3D simulations detailing fault lines and at higher resolutions (Hay, 2003; Zuiderveld & Viergever, 1992) than previously possible. The structure of the computer grid used for reservoir simulation dictates how precise the simulation will be (Aarnes *et al.* , 2005; Yu & Sun, 2009).

Multiphase flow simulators calculate the flow of the oil, water and gas through pores and fractures (pore-scale modelling) (Blunt, 2001). Corner point simulation systems follow the contours of blocks and fault lines instead of using blocks of uniform dimensions can have irregular shapes with more sides and obtuse angled corners (such as pentagons or hexagons) often generating more precise models (Adamson *et al.* , 1996). Simulators perform numerous calculations using average property values within homogeneous regions defining the grid structure (Royer *et al.* , 1996). 3D simulation requires huge quantities of calculations and one method adopted reducing the lag between uploading and simulating the model is to use the power of parallel programming. Until around 2004 the trend was to develop more powerful processors reducing calculation speeds in a linear sequential fashion (Edwards *et al.* , 2011). Multi-core processor computers can perform smaller individual parts of a large task in a divide and conquer strategy where each processor is given an equivalent level of work to perform (Ma & Chen, 2004), in the case of oil reservoir grids this could be individual volumes of cells.

## 1.10 Simulation Software

Oil reservoir simulations are derived from complex mathematical equations and there are numerous industry standard oil reservoir simulation applications which run on 32/64-bit computers and on various platforms such as Windows and Linux. Engineers often require precise knowledge of individual elements or properties of the reservoir and although some simulators have been designed with a do-it-all approach and can cope with most of the common aspects of reservoir modelling, others are designed with very specific remits. Two-phase simulators simulate oil recovery by segregating pressure values from saturation levels (Chen *et al.* , 2004), but the addition of gas, heat or chemicals require a multi-phase simulator (compositional simulator). Sometimes the main elements being simulated are the chemical injection processes (Kumar & Okuno, 2014). Generally a three-phase (black oil) simulator splits the hydrocarbons into oil and gas keeping the water as a non-mixing element and temperature as a constant (Chen, 2007) as this is quicker to simulate. All simulators share a common purpose; they are tools for giving engineers the information they require as fast as possible allowing

them to make fast informed decisions, optimising production levels and costs. Some of the most commonly used simulator packages are as follows.

### 1.10.1 Schlumberger[2]

Schlumberger arguably have the largest petrochemical simulation and modelling software suite in the world covering all aspects of the geophysics and the petrochemical industry.

ECLIPSE is a black oil reservoir simulation suite and runs in the PETREL workflow environment. Some of the ECLIPSE packages include:

- E100 – this is a three-phase black oil simulator for Cartesian, corner point grids, flooding with temperature visualisations and faults.
- Compositional (E300) – this is a multi-phase simulator.
- Thermal – this models gas, temperatures and foam flooding.
- FrontSim – this generates flow models based on varying degrees of pressure present throughout the reservoir.

### 1.10.2 Halliburton Landmark

Halliburton Landmark have several simulator applications for analysing, modelling and simulating all aspects of the petrochemical industry. Nexus/VIP is their most commonly used and models reservoirs and wells .

### 1.10.3 CMG – Computer Modelling Group[3]

CMG offers a range of simulators:

- STARS – this is a thermal simulator, simulating heat from gas, chemical injection and foam flooding. It can simulate various grid types (Cartesian, Cylindrical and Corner Point).
- IMEX – this is a black oil simulator incorporating levels of local grid refinement and fracture detail.
- GEM – this is a chemical simulator used to model the injection process of gas and liquids in fractured black oil reservoirs, shale liquids and WAG (water-altering-gas).

[2]Schlumberger – http://www.software.slb.com/
[3]CMG – http://www.cmgl.ca/software

### 1.10.4 Streamsim Technologies[4]

Streamsim Technologies Inc have a range of software (studioSL) used to simulate large heterogeneous reservoirs and for history matching.

- 3DSL -this is a three-phase simulator used to simulate Cartesian, Cylindrical and Corner Point grids.

### 1.10.5 UTCHEM – University of Texas Chemical Compositional Simulator [5]

UTCHEM is a free 3D multiphase simulator and can model chemical and foam flooding on single and dual porosity black oil reservoirs.

### 1.10.6 Rock Flow Dynamics (tNavigator)[6]

Rock Flow Dynamics have modelling software compatible with most leading simulator packages (ECLIPSE E100 and E300, IMEX and CMG).

- tNavigator – tNavigator uses parallel computing on multi-core computers and computer clusters and has built-in standardised simulated simulations such as water flooding analysis.

## 1.11 Sciencesoft Ltd[7]

Sciencesoft offer a complete range of software applications for displaying and analysing simulation data compatible with all major simulators covering all aspects in the oil reservoir engineering field.

- S3GRAF – S3GRAF is a post-processing software package used to display 2D plots and graphs based on contours, barriers and cut planes.
- S3GRAF-3D – allows engineers to generate 3D views of simulated models in a fraction of a second.

[4]Streamsim Technologies – http://streamsim.com/
[5]CPGE – http://www.cpge.utexas.edu/?q=UTChem_GI
[6]RFD – http://rfdyn.com/about/
[7]http://www.sciencesoft.com/

- S3connect – S3connect links models of surface pipeline networks in GAP[8], a multiphase oil and gas Integrated Production Modelling (IPM) tool used to model oil reservoir well production levels to reservoir simulation models.
- S3control – S3control is a pre-processing package which incorporates a text editor and syntax editor capable of utilising and highlighting keywords incorporating the ability to display various plot types in a multi-window view and for uncertainty modelling.
- S3sector – S3sector allows engineers to isolate selected regions/sectors of a reservoir grid model for refined analysis and simulation at an increased level of detailed than would be possible if the whole model were simulated – reducing simulation times from weeks to minutes.

[8]IPM: http://www.petex.com/products/?ssi=5

# Chapter 2

# Research Topic Fundamentals

This thesis looks at the novel approach of applying octree compression techniques to 3D oil reservoir datasets by compressing their active cell information and storing them in memory in a flattened out linear format. The octree structure generated was '*pruned*' (Kidner & Smith, 1997), removing null pointers before being flattened out into an array structure reducing the original memory allocation while supporting fast lookup times. Preliminary experiments were conducted and results given when run within an actual oil reservoir software visualisation application before discussing the hierarchical storage and face culling algorithms developed this research.

Examples or the simpler 2D quadtree case are given applied to bitmap images before moving on to 3D octree compression techniques. In order to test the various algorithms and data structures presented in this thesis Sciencesoft supplied a set of actual oil reservoir grids for testing. Initial experiments incorporated these algorithms into one of their oil visualisation applications (S3GRAF-3D). The main objective in this thesis was to develop a technique to losslessly compress oil reservoir active cell information but resulting datasets should also accommodate fast gird scanning and individual cell lookups. As Sciencesoft use array data structures and searching generally involves scanning through 3D arrays ($x$, $y$ and then $z$, not ideally suiting tree's structures) subsequent experiments involved taking the fundamental requirements of S3GRAF-3D and removing all its overheads from external packages and classes and performing experiments more suited to tree structures. This gave an indication of how well this octree structure could perform if Sciencesoft adapted their application to suit tree structures. These experiments traversed the octree in breadth-first traversal using call-back methods demonstrating the potential performance gains.

Results from these experiments are used to prove the hypotheses made in this research:

1. Octree compression will prove to be a more efficient method for storing oil reservoir 3D active and inactive information.
2. Cell lookup times will prove to be quicker using recursive traversal methods with the octree representation than direct access methods.

Sciencesoft develop S3GRAF-3D using C# in Visual Studio and for this reason the Visual Studio environment and C# programming language was chosen, as any developed software would have to be fully integrable with their existing application.

The hierarchical pyramid visualisation techniques and associated face culling algorithms developed in this research are discussed illustrating how grids can be visualised using the information stored at each level of the tree. This was performed to address part of the thesis statement which states that a hierarchical vector scaling method would be developed which would allow large multi-million cell reservoir grids to be stored on the GPU and visualised more efficiently than is possible today. Further sets of experiments were conducted which detail the memory savings and time complexities of generating the different levels of the pyramid vertex tables comparing them to the high level detailed information required when sending individual cells, as used today. This thesis briefly introduces other areas where this research could be applied followed by overall conclusions detailing how the hypotheses made in this thesis have been addressed and the demands of the thesis statement met.

## 2.1 Tree Structure Suitability

Trees have proved to be an exceptionally effective and efficient method for storing data such as in the case of applying quadtrees compression techniques to 2D computer graphics (Manouvrier *et al.* , 2002). Trees can be used to break down data into areas of equal value, quadtrees have proven to be extremely good at storing images with its leaf nodes representing homogeneous areas of colour values. 3D graphics can make use of octree data compression techniques where 3D space is sub-divided into volume pixels called voxels (Favalora *et al.* , 2001). In the case of 3D oil reservoir visualisation octrees can be used to subdivide the reservoir into individual cube-like volumes of homogeneous values. These values could be oil pressure, gas pressure or volumes of stock. The following sub-sections of this thesis discusses the following topics:

1. Quadtrees.
2. Octrees.
3. Pyramid Techniques.
4. Entropy.

5. Open Toolkit – OpenTK.
6. Line-of-sight (LOS)

## 2.2 Trees

There are numerous data types used for compressing data; their application involves taking the initial data and removing redundancy so that it can be expressed with fewer bits (Mathews, 1996). Trees are have been used for many years and are well known for their effective and efficient storing capabilities and ability to accommodate fast searching of data (Bonet & Peraire, 1991). Trees are abstract data types which can be used to store data in a hierarchical structure providing fast searching and used for storing information in a more compressed state (Song *et al.* , 2010). They do this by sub-dividing a given domain into sub-sections of homogeneous values and representing each sub-section as a single leaf node (Song *et al.* , 2010). These nodes are linked together in memory forming a tree structure. The tree is a compressed representation of the original dataset since single nodes within the tree represent multiple elements in the initial uncompressed dataset.

Tree structures not only compress the original data but potentially allow fast memory access. This abstract data type is frequently used for storing linear sets of data such as names (Knuth, 1971) and for compressing 2D graphical images and 3D grids (Bonet & Peraire, 1991). As a data structure a tree is either a leaf node or a root node with one or more associated sub-trees. Typically the sub-trees are accessed via pointers stored in the root. From here all nodes contained within it, can be accessed using various tree traversal algorithms (Meagher, 1982). These include traversal methods such as post-order or pre-order where nodes are visited in a pre-determined sequence to one-another (Samet, 1984).

The root node at the top of the tree points to child nodes (header nodes or leaf nodes). Header nodes point to either, other header nodes or leaf nodes forming a hierarchical pyramid like structure (Watt & Brown, 2001) and (Rubin & Whitted, 1980). Leaf nodes differ from header nodes in that they do not point to any other node and have no children. Trees can become unbalanced meaning they contain more nodes and deeper on one side of the tree than the other, making them less efficient (Zou *et al.* , 2005). There are of course various methods and algorithms which can be adopted to balance trees either after they have been created or as they are being fed data such as the relaxed balancing method (Hanke *et al.* , 1997) or AVL tree algorithms (Ellis, 1980).

There are several variations of the tree compression algorithms and their use is designated by the candidate dataset topology. The characteristics of a given dataset greatly define how

well suited and efficient a given tree compression technique will be (Yu & Sun, 2009). Two basic and commonly used tree structures are quadtrees and octrees. Trees are programmed recursively and are considered to be extremely quick, generally having far fewer lines of code and more comprehensible (Filinski, 1994). Their 'recursive case' generates a duplicate instance of itself until it meets its stopping condition the 'base statement' (Wiedenbeck, 1989), a leaf node in the case of tree compression.

### 2.2.1 Quadtrees

This sub-section introduces 2D quadtree compression using bitmaps as 2D arrays. Bitmaps can be viewed as 2D arrays of pixels each having a colour value derived from the colour depth of the image (Bourke, 1993). One of the main problems with storing images can be file size (Miano, 1999). A file of dimensions 256 pixels x 300 pixels at a colour depth of 32 bit would result in an image of 300KB.

In order to store images more efficiently various techniques are used such as Huffman (Huffman *et al.* , 1952) coding which initially uses a tree structure to compress the colour information of an image. The smallest bit representation is apportioned to the most frequently occurring colour value in the image (Stabno & Wrembel, 2009). Another common bitmap compression technique is Run Length Encoding (Würtenberger *et al.* , 2003) where a palette stores each different colour value in the image and the number of sequential occurrences of each colour. The file can then store the palette and just a value of a number pixels pointing to the correct place in the palette so as to regenerate the image (Waggoner, 2010). He goes on to explain that RLE are suitable for images where there are large blocks. Both Hoffman encoding and RLE are lossless compression techniques but there are numerous others such as Lempel-Ziv-Welch (LZW) compression (Knieser *et al.* , 2003) which is similar to RLE but the colour palette does not need to be stored. There are of course lossy compression techniques where not all the information is stored, for example storing only average values taken from sections of an image where to the naked eye compression artifacts such jagged edges may not be visible with slight compression but higher compression rates could see high levels of 'pixelation' and 'blockiness'. Typical lossy image compression techniques include JPEG, PNG and GIF (Rabbani & Jones, 1991). These compression techniques do away with information which is assumed will not be missed or noticed by the human eye (Clunie, 2000).

Depending on the application being used smaller files with a degree of image noise or very minimal levels 'blockiness' are sometimes acceptable due to the ever growing competition for bandwidth (Pandur & Thiruvallur, 2009). Graphic applications just open images and are not aware of what changes it may already have gone through. This can cause them to contain

artifacts although if a 'quantizer table' was used, some computer software programs are able to reverse some of these unwanted attributes to a degree (Zhigang & de Queiroz, 2003).

Quadtrees are similar to binary trees but can have four child nodes and used used extensively in 2D computer graphics (Manouvrier *et al.* , 2002). When applied to images, quadtrees sub-divide the scene into homogeneous regions where each leaf node contains pixels of a similar value and the resulting tree structure represents a lossless compressed representation of the image (Yin *et al.* , 2011). Sometimes these homogeneous regions are referred to as the domain of the tree (Sundar *et al.* , 2008). There is no need to store all the pixel values from the image as each leaf node can represent a common pixel value within a 2D area in the image. Images containing large areas of homogeneous colour regions generate smaller trees as these images generate less leaf nodes, each representing larger 2D areas.

Sometimes a bottom up tree algorithm is used where the 2D array is first recursively sub-divided down to its lowest level, single cells as it can prove to be more efficient than a top-down approach as the grid is initially sub-divided to the stopping case for each quadrant (one cell) it is extremely fast and sometimes lending itself better to parallel processing than top-down traversal techniques (Sundar *et al.* , 2008). The recursive function sub-divides the image down to single cells creating nodes in memory, each pointing down to the four single cell leaf nodes at the bottom of its branch in the tree. If a node's four leaf nodes are equal, based on the homogeneous criteria a single leaf node is created in place of the previous node which represents the area which the previous four leaf nodes previously did. This leaf node will be evaluated with its three siblings and if they all match then, again a single leaf node will be created and their parent node represents the area which they collectively represent in the grid. This process recursively continues up the tree to the root node. If at any stage in the recursive process the four quadrant leaf nodes are not equal then those four leaf nodes are kept in the tree as leaf nodes and this will be the lowest point in that particular branch of the tree and a header node would be created which points to these child leaf nodes. Those leaf nodes which represent cells outside the grid dimensions are not stored, these are only created by the power-of-two size being larger than the original grid size. The compression technique used in this research used the principle of sub-division based on power-of-two values as this was more efficient than dividing at various aspect ratio sizes as it is faster due to a reduction in node length calculations to be carried and stored in the tree during compression and decompression. As the original root dimensions and aspect ratio of the grid and root node are known, with each traversal down the tree and sub-division the edge sizes which the tree nodes represent can be calculated by simply sub-dividing by two.

Figure 2.1 shows a simple image with sides lengths equal to a power-of-two value so no cells reside outside the grid dimensions (no padding cells). It shows how it is sub-divided into

the compressed quadtree structure, from the root at the top of the tree which holds all the heterogeneous colour values, its pointers to child header nodes down to homogeneous leaf nodes at the bottom of the branches.

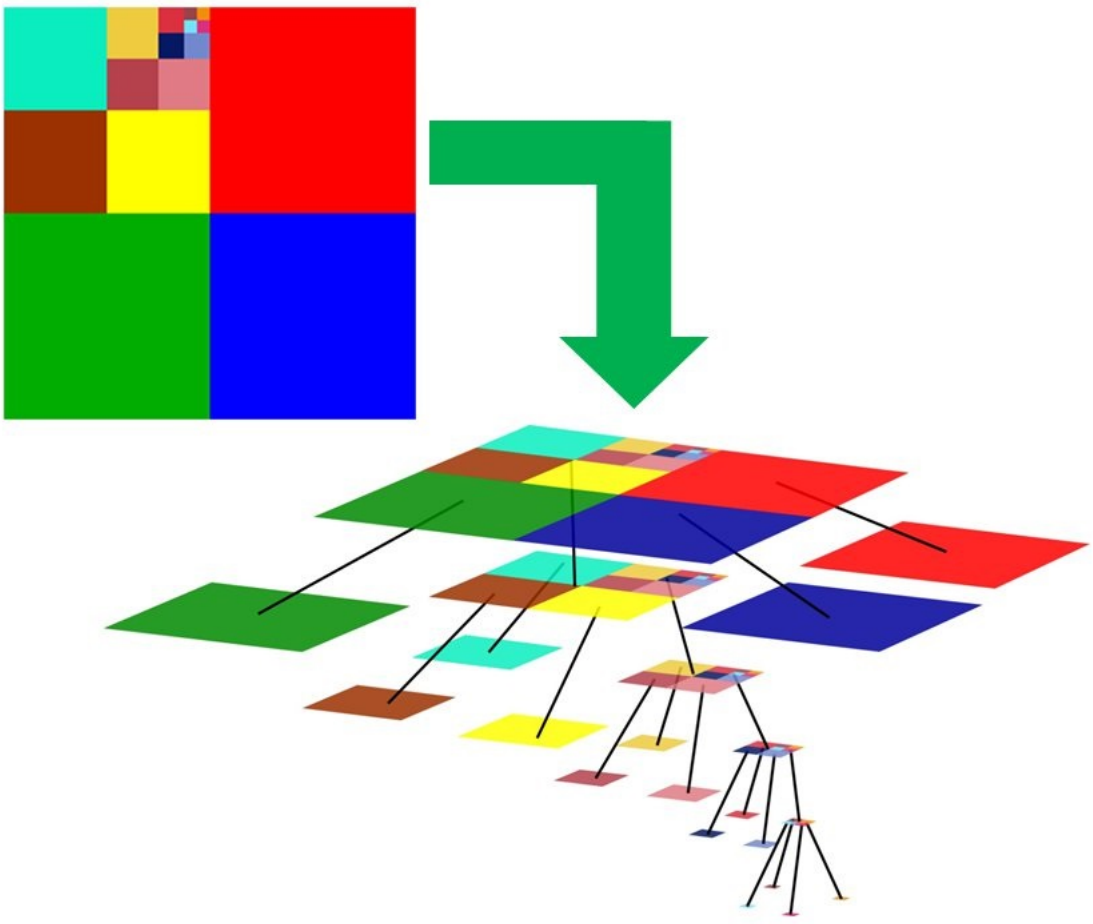


Figure 2.1: A simple quadtree structure visualisation

Figure2.2 shows the hierarchical tree structure created in memory when compressing 2.1 using quadtree compression techniques.

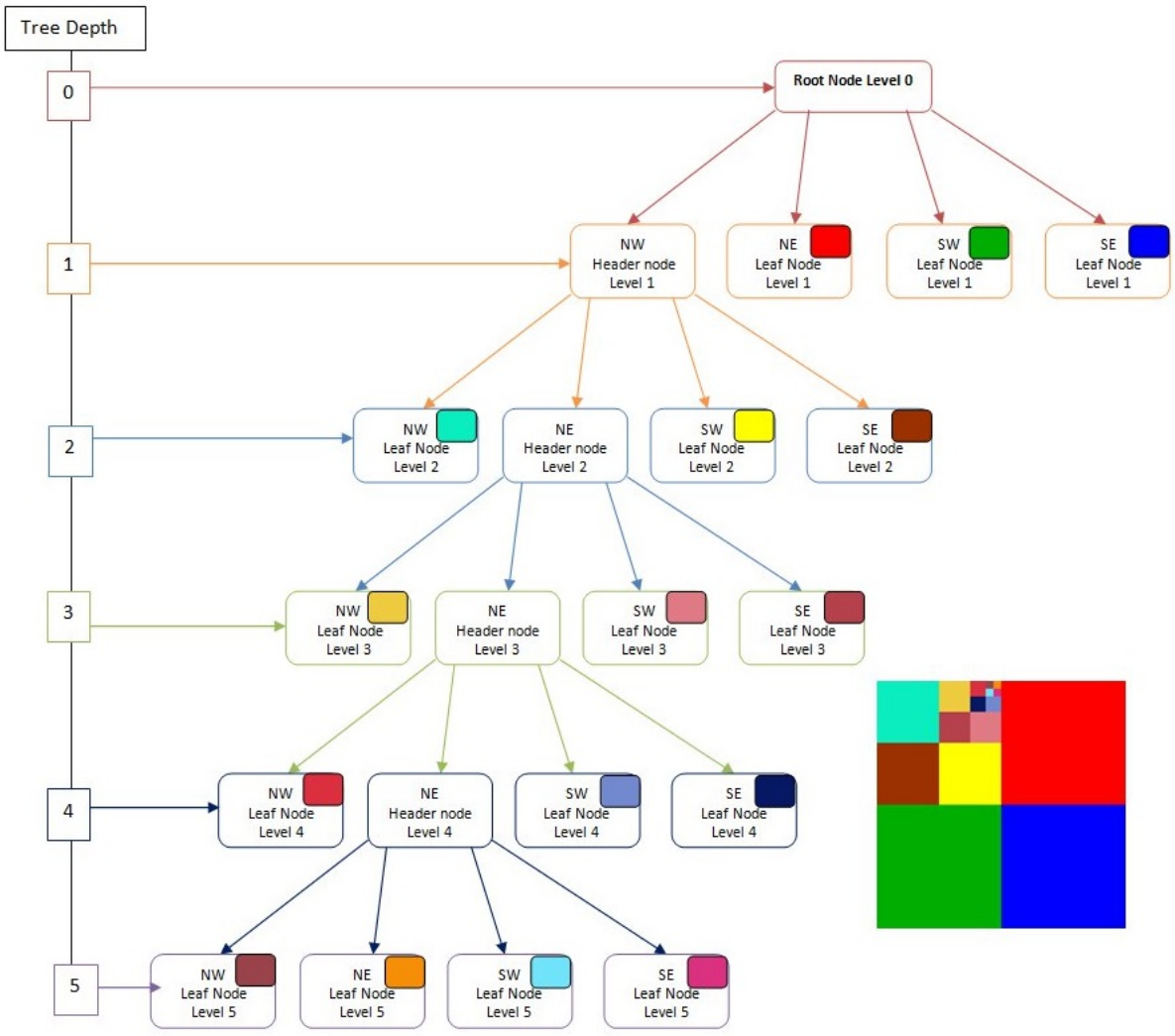



Figure 2.2: Analysis of the simple quadtree structure depicting the pointers, headers, leaf cells and tree depth

Figure 2.3 depicts two images with equal aspect ratios of 256 x 256 pixels, (a) has single blocks of colours and (b) has gradients of colours. These were compressed using bottom-up quadtree compression technique (Hunter & Steiglitz, 1979) and Table. 2.1 details their resulting node counts.

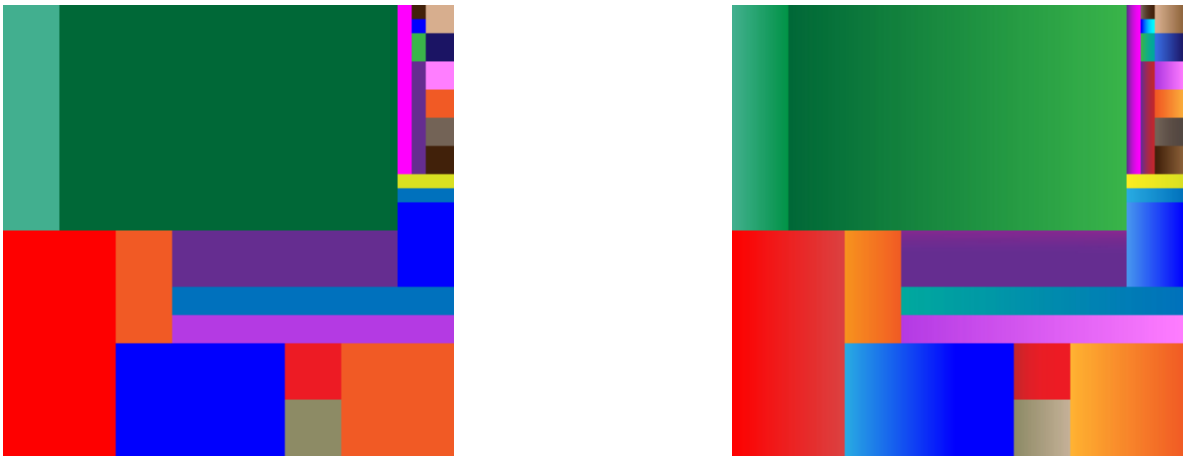


Figure 2.3: (a) flat single coloured regions (b) gradient coloured regions

| Image | Number of header nodes | Number of leaf nodes | Total number of nodes |
|---|---|---|---|
| a | 30 | 91 | 121 |
| b | 17190 | 51571 | 68761 |

Table 2.1: Tree node structure of Figure 2.3

Looking at the results from Table 2.1 it can be seen that as image (*a*) has large areas of flat homogeneous colour values, less leaf nodes are created generating a shallower, more compressed tree as its leaf nodes represent larger areas resulting in fewer header nodes and tree levels. Image (*a*) generated a tree structure containing six levels (zero to five) and (*b*), nine levels (zero to eight) as detailed in Table 10.1 in Appendix A of this thesis.

Looking at image (*a*) and Table 2.1, it can be seen that instead of representing each of the 65536 pixels with a 32-bit *argb* colour value, it can be represented using 91 leaf nodes and 30 header nodes. Assuming that there are no class overheads and a header node contains 4 pointers all of which are stored as a 32-bit values, image (*a*) could be compressed to 964 bytes. The original image would have been 256 KB, yielding an impressive compression ratio of 0.015. As image (*b*) has a larger array of colour values resulting in greater quantities of nodes, no compression was achieved as the tree structure would require 537 KB for storage almost twice the original number of bytes. It is therefore clear that quadtree compression techniques are best suited to 2D arrays where large areas of homogeneous regions exist and smaller tree structures on average yield quicker traversal times.

#### 2.2.1.1 Tree Time Complexity

The compression time of the bottom-up quadtree is linear as the algorithm recursively sub-divides the bitmaps down to single pixels. Suppose we have some sort of pyramid or tree in which each successive layer has half as many elements in it as the layer below. The total number of elements will be $N + N.2 + N.4 + N.8...etc$. This is a series of the form $1 + 1.2 + 1.4...etc$ and we know that that series is limited by 2, so the number of elements in our pyramid/tree is limited by $2N$ and thus grows as $O(N)$. Since trees which shrink by factors of 4 or 8 at each level shrink even more rapidly, they too will tend to a similar finite limit $1.5N$ for quadtrees and $1.25N$ for octrees. The number of components is also thus of $O(N)$ for these trees. Any algorithm, such as the compression stage which operates on each element, will also be of $O(N)$ where $N$ is the number of pixels in the image. Searching the quadtree is $O(log(n))$ as each individual lookup only has to traverse a single quadrant child node branch in the tree to reach a leaf node where $n$ is the number of nodes in the tree.

### 2.2.2 Octrees

In the 3D case such as with 3D graphics the scene can be sub-divided into volumes of homogeneous values and is a 3D extension to 2D quadtrees called octrees. The first node in the octree is a root node and represents all the cells in the grid. If these cells are not identical to one-another then eight header nodes are created. In the case of oil reservoir active cell information this would mean that not all the cells the node represents share the same active status. Octree header nodes can have a maximum of eight child leaf nodes where each octant leaf node represents a volume in 3D logical space (Samet, 1980, 1984, 1990; Samet & Kochut, 2002).

Leaf nodes do not have any descendant child nodes and their parent nodes are header nodes which point to them. It is not unreasonable to expect to see higher levels of compression ratios when compressing 3D volumes possessing similar characteristics as shown earlier in image (*a*). This is because given a leaf node 4 cells x 4 cells in the 2D case, the area represented by the leaf node is compressed to 1/16$^{th}$ of its uncompressed state but given a leaf node representing a 3D volume with an identical node length the compression would rise to 1/64$^{th}$ as this leaf node would represent 64 cells. Similarly to the characteristics of image (*a*) large clusters of similar values in a 3D volume generate shallower trees containing fewer header and leaf nodes than one with randomly scattered values throughout its entirety. An alternative fourth dimensional level of compression could be time in video or animation (Ahuja & Nash, 1984).

Sometimes trees are described as being balanced and indicates that the tree somewhat symmetrical where there are equal, active and inactive leaf nodes branching from each of the header nodes at each level of the tree. In this case for every header node only half of the pointers to the lower sub-trees are required as an equal number of cells are split between the active and inactive nodes.

The compression time of the bottom-up octree is linear as the algorithm recursively subdivides the 3D grid down to single cells, $O(N)$, as defined earlier in sub-section 2.2.1.1 on the previous page, where $N$ is the total number of cells in the grid. Searching the octree is $O(log(n))$ as each individual lookup only has to traverse a single octant child node branch in the tree to reach a leaf node.

In this research the algorithm used to compress the oil reservoir's active cell status is based on powers-of-two values and to eliminate the possibility of creating fractions of a cell, the grid is first encapsulated into a 3D space where all *x*, *y* and *z* axes are equal and at a power-of-two. This means that a stopping case of a single cell can be generated where its dimensions are 1 x 1 x 1 in all directions. In order to do this all the grid's dimensions are evaluated to see if they are equal and already at a power-of-two size. If they are not then it is superimposed into the smallest power-of-two dimension which is large enough to house it.

Figure 2.4 shows a 4 x 2 x 3 matrix, it illustrates how it would be encapsulated into a 4 x 4 x 4 matrix to allow for power-of-two sub-division where the black cubes are the inactive (0) valued cells. This grid has 24 cells; 13 active and 11 inactive. The grid's *x*, *y* and *z* axes and origin position placement (0,0,0) was as illustrated as adopted by many simulator software applications.

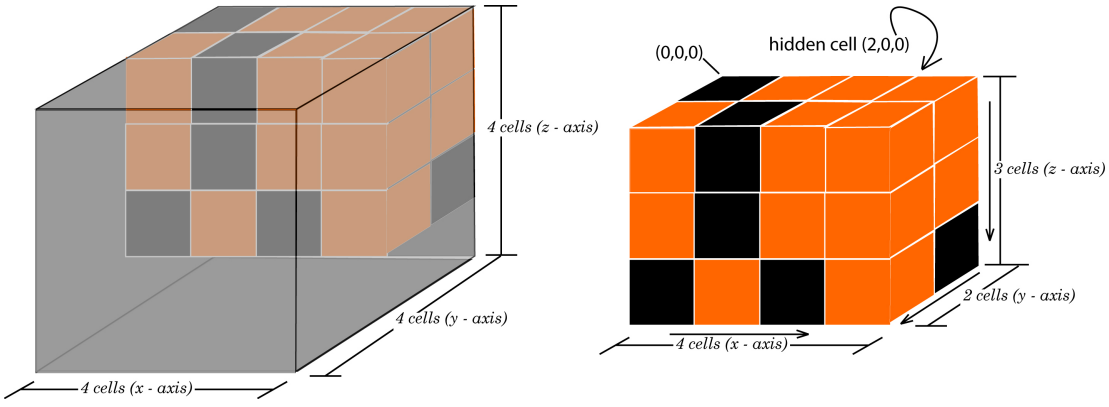



Figure 2.4: Illustration of encapsulating a 3D grid into the next power-of-two size

Figure 2.5 shows an illustration of the hierarchical octree in memory produced from Figure 2.4.

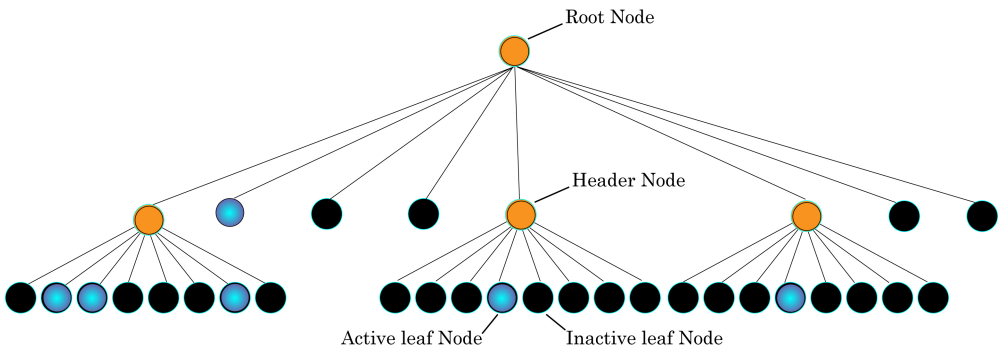



Figure 2.5: Octree representation of the 4 x 2 x 3 grid model

There are 33 nodes in Figure 2.4, most of them are inactive leaf nodes and can be removed from the tree, creating a smaller memory overhead eliminating redundancy (Kidner & Smith, 1997), a process known as pruning (Knoll, 2006). This pruned or hyperoctree (Yau & Srihari, 1983) is shown in Figure 2.6, creating a more compressed octree, only storing 10 nodes.

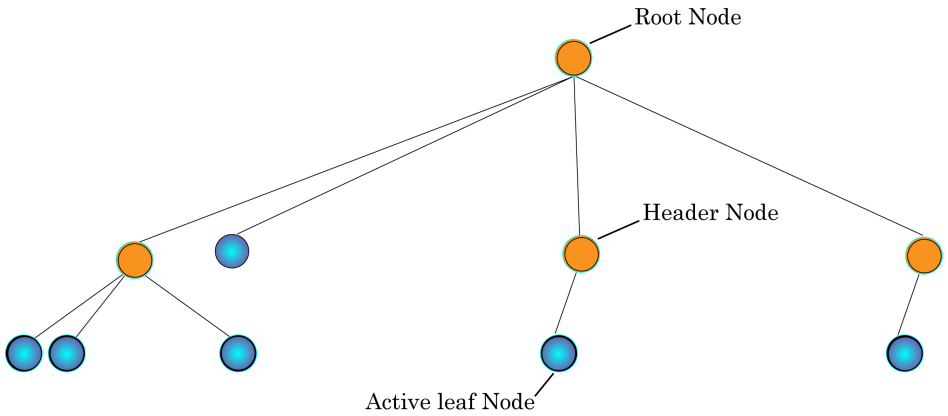



Figure 2.6: Pruned 4 x 2 x 3 grid model octree

During octree compression there exists, with each encountered header node, eight directions which the recursive function can take, one in the direction of each possible child node. Figure 2.7 shows the naming convention used for labelling the octree's header node child node directions, each representing an octet of its parent's volume.

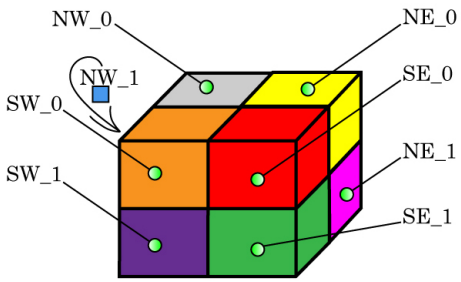



Figure 2.7: Octree header node octants

It is possible to flatten out the octree in the from of a linear array of integers removing redundant inactive nodes, *'pruning'* the tree (Ayala *et al.* , 1985; Dyer *et al.* , 1980). Pruning the tree can be problematic as on de-compression the algorithm must be able to calculate each omitted inactive child node would have resided in the tree otherwise the recursive function would try to read beyond the length of the array, searching for child nodes that no longer exist. Pruning the tree also means that grids with larger and more frequently occurring inactive nodes yield greater levels of compression. The solution to this problem was to adapt the header node to act as a child node indicator, the *header flag*.

### 2.2.3 Octree Header Flag

The octree *header flag* uses eight bits to indicate the active status of its child nodes, starting from the left most bit (bit seven) to the right most bit (bit zero) these bits represent the child node octants {*NW_0*, *NE_0*, *SW_0*, *SE_0*, *NW_1*, *NE_1*, *SW_1*, *SE_1*} as illustrated in Figure 2.8; a '1' is written to the *header flag* for child nodes containing active leaf nodes and a '0' is written for those containing only inactive child nodes.

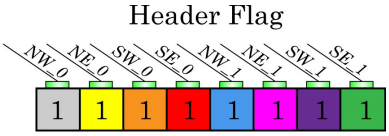



Figure 2.8: Octree header node octants

During decompression this *header flag* first checks the header flag prior to searching for child nodes as each bit in the *flag* indicates the active status of one of its child nodes. This is used as a direction indicator allowing the recursive function to traverse in a particular desired direction using the *flag's* 'on' and 'off' bits. An array can now be created to store the flattened-out octree (*octArray*), where each leader/leaf mode element is stored as a 32-bit integer and head flag as a byte. The pseudo code used in this thesis to compress 3D grids using octree compression can be found in Appendix 10.6 on page 136 of this thesis, 10.3 on page 135.

## 2.3 Pyramid Compression Techniques

Pyramid compression techniques are often applied to images (Adelson *et al.* , 1984) such as The Laplacian Pyramid (Burt & Adelson, 1983; Ghavamnia & Yang, 1995). This method of compression could also be applied to a 3D matrix of numerical values. With 2D images

the pyramid is split into several layers, each layer being of a fraction of the original image's resolution (Cockshott *et al.* , 2003). The algorithm works by taking the original image and passing a filter over it, such as a Gaussian blur where pixels within a given distribution of pixels such as square or rectangle are given a weighted average of the distribution of cells, the sampled pixel receives the highest average and those around it receive an average based on their proximity to the sampled pixel (Waltz & Miller, 1998). This resulting in a 'smoothing' (Lin *et al.* , 1996) effect where sharp colour contours are replaced with smother gradients of colour.

Some methods scale down images to a quarter of their original size and then scale them back up again using various interpolation methods which generates 'lowpassed filterred' images (Cockshott *et al.* , 2003) and adopt nearest neighbour, bi-cubic or bi-linear algorithms. The resulting image has all of the original image's energy and peaks removed. Subtracting this filtered image from the original image leaves what they refer to as the 'highpassed filtered' image. The lowpassed filtered image is then scaled down to a quarter of its size keeping its aspect ratio and the whole process is repeated again. With each iteration of this algorithm, the final highpassed image and all of the lowpassed images are stored. This results in a pyramid like structure as shown in Figure 2.9 where a Gaussian blur has been applied to the image at each level of the pyramid to generate the lowpassed filtered image. The pyramid has three levels of compression and Figure 2.9 illustrates the algorithm for compressing the image and decompressing the image is a reversal of this procedure using the stored lowpassed filtered images.

The time complexity of the algorithm depends on the distribution of pixel values based on the possible number of values. Burt & Adelson (1983) looked at compressing grey scale images sampling the frequency of pixels at each level in the pyramid, higher pyramid levels produce fewer frequencies than levels below allowing it to be represented in fewer bits so that the pyramid could be expressed as a level of entropy (bits per pixel) based on the sum of each pyramid level and is given below where $f$ is the frequency of the pixel value $i$. Although they go on to say that this can be reduced using various quantization techniques such as storing averages of threshold pixel values in 'bins' and although generating levels of error they were not considered perceivable by the human eye. The time complexity of this 2D compression is $O(N)$ where $N$ is the total number of pixels in the original image.

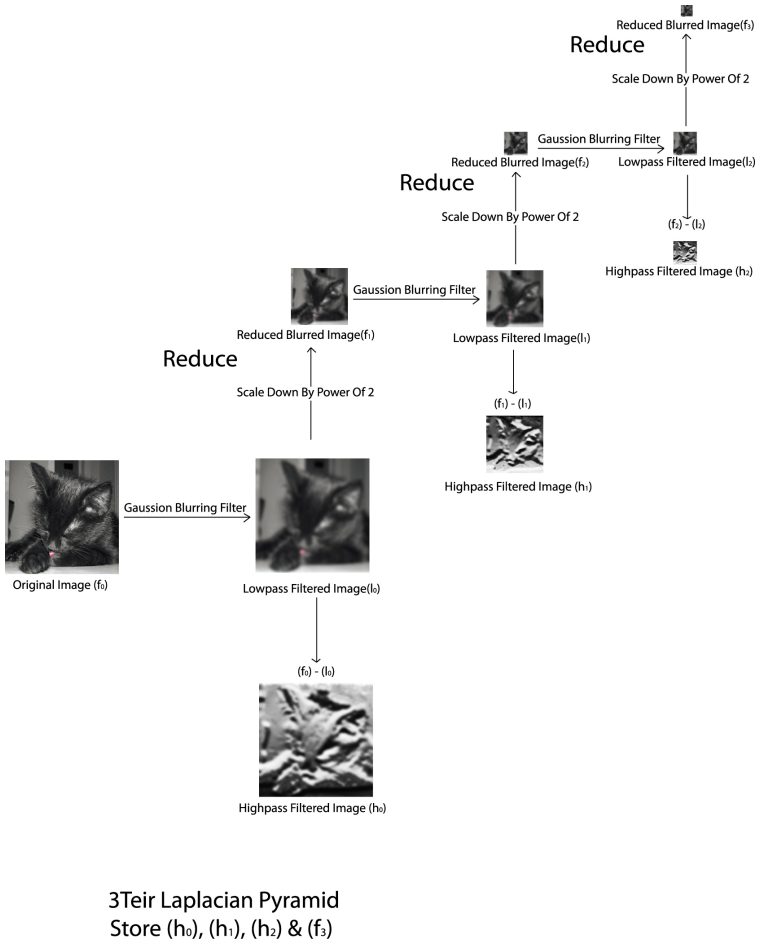



Figure 2.9: 3-tier Laplacian Pyramid compression algorithm

## 2.4 2D And 3D Pyramid Structures

Figure 2.10 on the following page shows a lossy compression technique applied to a 2D grid where the lowest level in the pyramid shows the highest resolution and each individual cell's value is displayed. Subsequent levels up the pyramid show their parent node display average values equal to the sum of their child nodes. With each level up the pyramid a coarser, more lossy and lower resolution of the 2D grid is displayed.

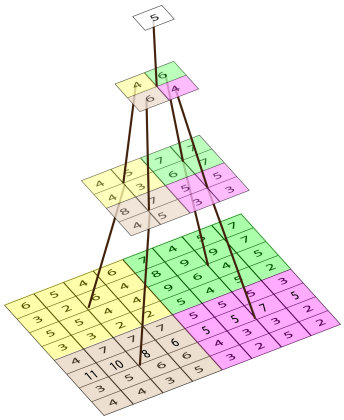


Figure 2.10: 2D averaging pyramid algorithm

The same principal could be used in 3D where the average of each block of eight elements in a 3D array could form a single element in the next level up the pyramid. If colours were only required the octants at each level could store averages of its child node's (Gervautz & Purgathofer, 1990). In the case of oil reservoir grids these 3D grid blocks could store property values such as porosity, saturation, wettability, etc.

## 2.5 Entropy

Entropy is a measure of randomness. Measuring the randomness of active cell distribution throughout a reservoir could prove useful as compression ratios and cell lookup times could be closely related to grid entropy. The active cell information of a grid cell can only have one of two states, active or inactive and by performing a scan through the grid a chain of ones and zeros can be generated. By looking at short sequences of states traversing through this chain a measure of randomness based on the number of state changes and lengths of sequences sampled can be deduced (Anderson & Goodman, 1957). These sequences of ones and zeros can be thought of as Morse code's dots and dashes. Looking at the probability of one or more states following another as a sequence of dots, dashes, ones, zeros, more frequently occurring sequences can be represented by fewer bits than those that do not, thus a level of redundancy can be removed (Shannon, 1951; Lelewer & Hirschberg, 1987). These chains of state changes are sometimes referred to Markovian Process Models (Cover *et al.* , 1994) and Markov Chains (Blackwell, 1957; Facelli & Pickett, 1990) and the correlation of entropy and Markov Chains with relation to oil reservoir grids are discussed in the following sub-sections.

In oil reservoir visualisation the reservoir is represented by a 3D grid of cells ($N_x \cdot N_y \cdot N_z$) for example an oil reservoir 200 meters x 200 meters x 50 meters may sub-divided into 200 x

200 x 50 cells where each in the computer model represents 1 metre$^3$ volume of rock in the reservoir although sub-divisions could be based on much smaller or larger values. With oil reservoirs the active status of cells sometimes do not differ as much in the horizontal plane as they do in the vertical plane. This means that although the oil reservoir may be thin in comparison to its width or breadth the computer model used to represent it may have far more cells in the z-axis represented as very thin rectangular cuboids. The geological grid cells represent volumes of rock using eight three dimensional $x$, $y$ and $z$ co-ordinates in 3D space and do not necessarily follow any uniform shape. However, the logical grid does, a value can be given to each cell depending on its active status, '1' for active, when a cell contains oil and '0' (inactive) for a cell which does not.

The formation of active and inactive cells in relation to one-another can be used as a guide to the stochastic nature of the reservoir. Oil reservoirs tend to have clustering characteristics where quantities of cells sharing a similar active status reside close to one-another. As octrees are ideally suited to compressing datasets containing large homogeneous regions, a reservoir containing large volumetric clusters of either active or inactive cells could be represented in a highly compressed state; the greater the size of clustering, the fewer the number of leaf nodes and the shallower the tree. It was proposed that it was therefore beneficial to have some level of gauging or measurement of clustering present within the reservoir grid model as this could give some insight into the effectiveness of applying octree compression.

### 2.5.1 Shannon's Mathematical Theory of Communication

With reference to Shannon's Mathematical Theory of Communication where the stochastic nature of three languages, English, German and Chinese (Shannon *et al.* , 1949) were evaluated. They looked at the probability of a particular letter following a given letter or a set of preceding letters, for example in the case of the English language, and given the letter 'Q' there would be a high probability that the next letter would be a 'U' based on 1$^{st}$ order approximation. This is due to the statistical nature of the English language. If the two previous letters were known then this would be referred to 2$^{nd}$ order approximation. He suggests that the higher the level of 'order of approximation', the greater the accuracy of predicting the correct next letter and explains that this stochastic nature of a given 'state' following a predetermined 'state' is known as 'Markov processes'. Shannon *et al.* (1949) proved this theory by analysing vast volumes of text and deduced that the probability rate at which the letter '*e*' would occur was 0.12, and for the letter '*w*' was about 0.02. As characters in ASCII code are sent using 8 bits, one could use a encoding such as Huffman encoding where the characters which occurred most frequently could be stored or transmitted using the least number of bits and characters which rarely occurred, using higher number of bits, as these larger strings of

bits would be sent less often, greatly reducing the number of bits required to send or store, so that that this message divided by the number of bits used to send it would give the optimum bit rate per character of the message. The equivalent to Markov's 1st order of conditional entropy where the frequency of each state, in this case a character in the English language, is not dependant on any previous characters, (Phamdo, 2004) but lower bit rates could be generated taking into account previous letters (2nd, 3rd, 4th order ) but at the time of their studies, this was computationally impossible to calculate but estimated it to be as low as 2.3 bits per character.

These 'Markov chains (Rrnyr, 1961) can be used to define the entropy of the system, removing the redundancy by increasing the number of previously known states (expressed as active status of neighbouring cells) of a *target* character, allows the system to be represented or transferred using fewer bits, in a compressed state. This research looks at evaluating the level of entropy (Basharin, 1959) of oil reservoir grids using four levels of previously known cell states, referred to as 4th order of conditional entropy and expressed in this case in bits per cell as this would indicate the level of compression obtainable using this level of previously known state conditions.

### 2.5.2 Markov's Conditional Entropy

Oil reservoir grid cells have two states, active and inactive. If a grid with a high percentage of active cells exists, then given a cell and its active status, it would be reasonable to assume that the probability of the next cell (target cell) being active is higher than in grids where the majority of the cells are inactive. Figure 2.11 shows an example of cell states and what probabilities of state changes from one state to another may be given either a very active or very inactive matrix.

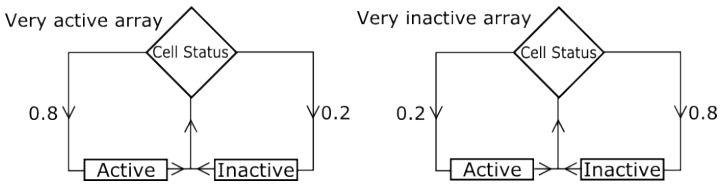
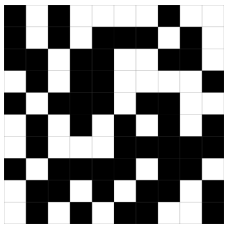



Figure 2.11: Typical probabilities of cell states for very active or very inactive grids

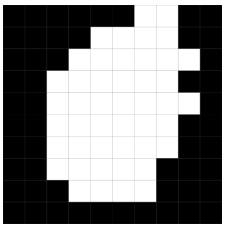
If a grid possessed the same number of active and inactive cells (equal number of both states) where these states changed in an alternating fashion similar to a chessboard then this would give a 1st order of conditional entropy of 1, known as maximum entropy. This is illustrated in the following expression where $P_i$ is the probability of the active state of the next cell in the chain of cells visited and $m$ is the number of state changes and $H$ is the measure of entropy.

$$H = \sum_{j=1}^{m} P_{i \log_2} P_i \; bits/cell$$

Given two grids 10 cells by 10 cells containing equal numbers of active cells it would not necessarily follow that they would have equal levels of entropy. One could have randomly distributed active cells and the other clustered regions of active cells. Randomly active grids could have maximum levels of entropy around 1, but grids displaying clustering characteristics would have lower levels of entropy being less stochastic as illustrated in Figure 2.12.



Grid (A) random, higher entropy



Grid (B) less random, lower entropy

Figure 2.12: 10 x 10 grids with 46% active cells (white squares) each with a different clustering of active white cells and levels of entropy

Table 2.2 gives the average active cell state probabilities of grids (A) and (B).

| States | Number of state occurrences | Probability ($P_i$) |
|---|---|---|
| White (active) | 46 | 46/100 = 0.46 |
| Black (inactive) | 54 | 54/100 = 0.54 |
| | Total state changes 100 | Sum of probabilities =1.0 |

Table 2.2: First order of conditional entropy example

First order of conditional entropy of the grid takes each cell state independently from all other cell states and so the grid would be represented as one bit per cell, in a chessboard this would be 64-bits (a one or a zero per cell). Second order of conditional entropy looks at the relationship between sets of states where an assumption could be made of a cell's active status given the active status of the previous cell. This would therefore remove redundancy where more cells could be represented by fewer bits. The chessboard example has maximum state changes but follows a pattern; white follows black and black follows white. For this reason using second order of conditional entropy the chessboard would have an entropy of zero and the grid could be represented by one bit, the entire grid could be predicted by only knowing the first cell's active status. Where first order of entropy has two states, second order of entropy has four. Table 2.3 shows a list of probabilities of states (1 for an active cell and 0 for an inactive cell).

| States | Previous cell (*i*) | Target cell (*j*) |
|---|---|---|
| A *(inactive then inactive)* | 0 | 0 |
| B *(inactive then active)* | 1 | 1 |
| C *(active then inactive)* | 1 | 0 |
| D *(active then active)* | 1 | 1 |

Table 2.3: Second order of conditional entropy states

Table 2.5 shows the state changes of the the two grids shown previously in Figure (A) 2.12 where the state changes in the chain generated when following a scan path through the grid as illustrated in Figure 2.4.

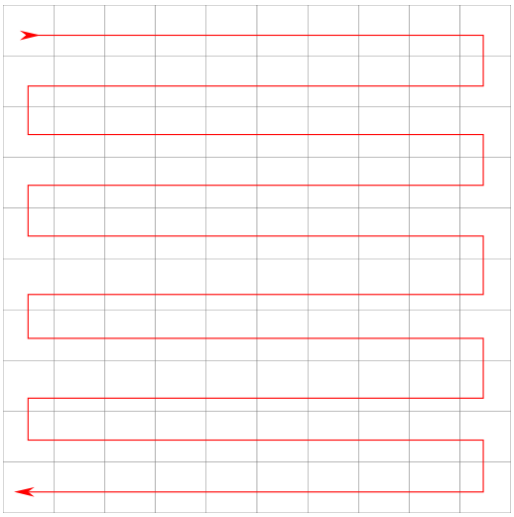


Table 2.4: Grid scan order

Both grids contains one hundred cells and second order of conditional entropy is defined by sets of two cells so that 50 state changes exist within the path. If the first cell encountered (*i*) was known to be zero then considering that this covers 54% of the initial previous states the next, target state will have a probability based on the initial state the 'weighting', if the $P_i$was 0 then the weighting would be $\frac{1}{P_i} = 0.54$. The probability of the next state given its previous state, $P_{j|i}$ is calculated by multiplying the probability of the next state change by its weighting. As the last state would always be a one or a zero, entropy, $H$ is calculated by taking the sum of all the $P_i$ and $P_{j|i}$ values multiplied by $\log_2$ of their respective $P_{j|i}$ values. The second order of condition entropy is illustrated in the following expression, defined by (Shannon *et al.* , 1949).

$$H = \sum_{j=1}^{m} P_i \sum_{j=1}^{m} P_{j|i \log_2} P_{j|i} \; bits/cell$$

The following tables show the different second order of condition entropy values of images (A) and (B) where it can be seen that the clustered grid (B) has lower entropy levels. Entropy (H) is a positive number but as the entropy calculations generate a negative number resulting values are multiplied by -1.

| **State** | **Number of states occurrences** | **Probability** $P_i$ | **Weighting 1/ probability** | **Probability of predicting the next cell given its previous cell** $P_{j\|i} \cdot weighting$ | $P_{j\|i}$ |
|---|---|---|---|---|---|
| **00** | 9 | Starting with a zero | 1 / 0.6 = | $\frac{9}{50} \cdot 1.667$ | 0.3 |
| **01** | 21 | 30/50 = 0.6 | 1.667 | $\frac{21}{50} \cdot 1.667$ | 0.7 |
| **10** | 15 | Starting with a one | 1 / 0.4 = | $\frac{15}{50} \cdot 2.5$ | 0.75 |
| **11** | 5 | 20/50 = 0.4 | 2. 5 | $\frac{5}{50} \cdot 2.5$ | 0.25 |

| **State** | **Image (A) entropy** $H$ |
|---|---|
| **00** | $0.6 \cdot 0.03 \cdot log_2(0.03) = 0.313$ |
| **01** | $0.6 \cdot 0.7 \cdot log_2(0.7) = 0.216$ |
| **10** | $0.4 \cdot 0.75 \cdot log_2(0.75) = 0.125$ |
| **11** | $0.4 \cdot 0.25 \cdot log_2(0.25) = 0.2$ |
| | $H = 0.854$ bits / cell |

Table 2.5: Second order of conditional entropy example:(Image (A)

| **State** | **Number of states occurrences** | **Probability** $P_i$ | **Weighting 1/ probability** | **Probability of predicting the next cell given its previous cell** $P_{j\|i} \cdot weighting$ | $P_{j\|i}$ |
|---|---|---|---|---|---|
| **00** | 24 | Starting with a zero | 1 / 0.52 = | $\frac{24}{50} \cdot 1.923$ | 0.923 |
| **01** | 2 | 26/50 = 0.52 | 1.923 | $\frac{2}{50} \cdot 1.923$ | 0.077 |
| **10** | 4 | Starting with a one | 1 / 0.48 = | $\frac{4}{50} \cdot 2.083$ | 0.167 |
| **11** | 20 | 24/50 = 0.48 | 2. 083 | $\frac{20}{50} \cdot 2.083$ | 0.833 |

| **State** | **Image (B) entropy** $H$ |
|---|---|
| **00** | $0.52 \cdot 0.923 \cdot log_2(0.923) = 0.923$ |
| **01** | $0.52 \cdot 0.077 \cdot log_2(0.077) = 0.077$ |
| **10** | $0.48 \cdot 0.167 \cdot log_2(0.167) = 0.167$ |
| **11** | $0.48 \cdot 0.833 \cdot log_2(0.833) = 0.833$ |
| | $H = 0.517$ bits / cell |

Table 2.6: Second order of conditional entropy example:(clustered image (B)

In order to achieve a more accurate level of the stochastic nature of the grid, third or fourth order of entropy could be calculated in a similar fashion. The higher the order of conditional entropy the more accurate the stochastic reading and the level of entropy will fall. Scanning through the two grids, Image (A) and (B) shown in Figure 2.12 on page 29, both grids had a first order of conditional entropy equal to almost 1, maximum entropy. However using fourth

order of conditional entropy they now give readings of 0.663 for the randomly grid (A) and 0.441 for the clustered active (B). Generating a value which can be used to represent the stochastic nature of an oil reservoir grid can be used to gauge the grid's clustering which in turn gives an insight into achievable compression ratios using octree compression techniques.

This could also be used to give an indication of the suitability of other grid structures such as medical images. As the order of conditional entropy applied to a path of states rises the entropy falls but the degree at which it diminishes lessens each time. This means that although a fifth or sixth order of conditional entropy may produce a lower entropy value the difference to that generated from fourth order calculations is very minimal and does not merit the extra computation effort required to calculate it. In this research fourth order of conditional entropy was deemed adequate to define oil reservoir grid entropy. Plotting the entropy based on increases in the conditional order would generate a curved slope eventually flattening out out as shown in Figure 2.13 as the number of different states decreases until only one state exists representing the entire grid, resulting in zero entropy.

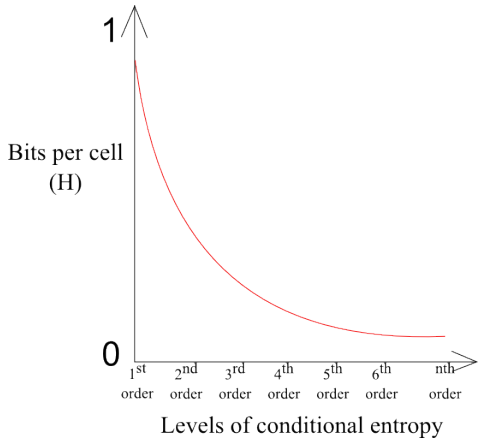



Figure 2.13: Levels of conditional entropy plotted against entropy value expressed in bits per cell (H)

As previously discussed there is a relationship between the measure of clustering of similar active status cells in a grid and resulting tree structure sizes and how this could be looked at a level of entropy. The previous example showed how a path of active and inactive cells can be generated by scanning through the grid in a scanning fashion and was given due to its simplicity. In the case of (Shannon *et al.* , 1949) strings of letters (domain) used to evaluate the entropy was sequences of letters within strings of words and these words formed sentences, because of the nature of written language, certain characters and words reside close to one-another both in the domain and in the string sequence as sentences are linear domains; grids are not linear but instead multi-dimensional – cells close to one-another in the path do not necessarily reside close to one-another in the grid. In order to accurately evaluate the entropy

of a grid, the path of bits passed to the entropy algorithm should visit cells in such an manner that cells close to one-another in the path are in close proximity to one-another in the grid, similar to a sentence of words.

There are of course other methods of scanning the grid to generate paths and in this research it was decided that a Hilbert curve design should be applied as a space saving curve (Sagan, 1994) where the curve passes through each position in a grid once. Unlike a scan line approach where each cell is visited in a raster formation, Hilbert curves visit cells in close proximity to one-another, similar to the active cell clustering effect present in reservoir grids. It was decided that the method which was best suited to the task was the 2D Hilbert Curve algorithm as this generated a path which visited cells in groups, similar to groups of clustered cells present in the layers of oil reservoirs. This was then extended to the 3D case where 3D grid were traversed in slices at a time starting with the top layer. The starting cell of each lower layer was the cell directly below the cell from the above layer. Figure 2.14 shows a 2D Hilbert Curve path through a 2D grid (64 cells x 64 cells).

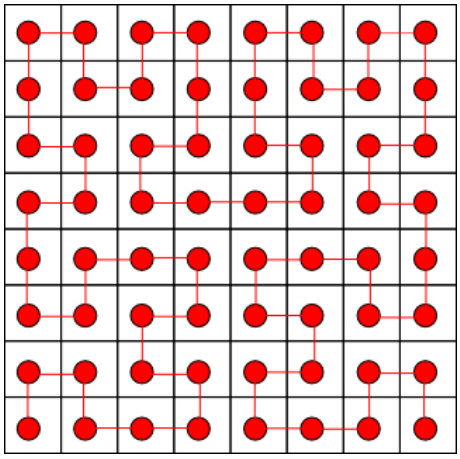


Figure 2.14: Hilbert curve path through a 2D 8 cell x 8 cell grid

This method of measuring oil reservoir grid entropy was adopted in this research and results are discussed in Chapter 5.

## 2.6 3D Visualisation Software

There are several 3D simulation software packages used today but as this research is developed in C# in a Microsoft Visual Studio environment OpenTK was chosen because it is designed to work in conjunction with C#. It is sometimes referred to as a ‘wrap-around’ for OpenGL as it uses OpenGL libraries but communicates with these libraries using OpenTK such as with

styles of materials, shaders and lighting effects. The appropriate drivers are loaded into the graphics card so as to allow OpenTK to communicate with it. In addition to this the simulation window allows for various styles of orthographical and perspective viewing angles. Keyboard inputs are also used to control objects, cameras or lighting in scenes.

Firstly a viewing screen size and aspect ratio is determined and a camera point in a 3D co-ordinate system is also defined. An imaginary 'clipping plane' is set up having an equal aspect ratio to that of the viewing window. The camera and near clipping plane is positioned at a desirable distance from one-another and the depth-of-view, viewing angle and far clipping plane attributes are also set. The camera point in the 3D world sets the position of the viewer's eye and the angle of view, unlike the human eye forms the angle at which the camera sees. It must also be noted that realistic perspective views can be created by carefully choosing specific camera types, viewing angles and clipping plane attributes. The camera sees everything within the viewing volume between the near and far clipping plane and everything outside of that is hidden and clipped as shown in Figure 2.15.

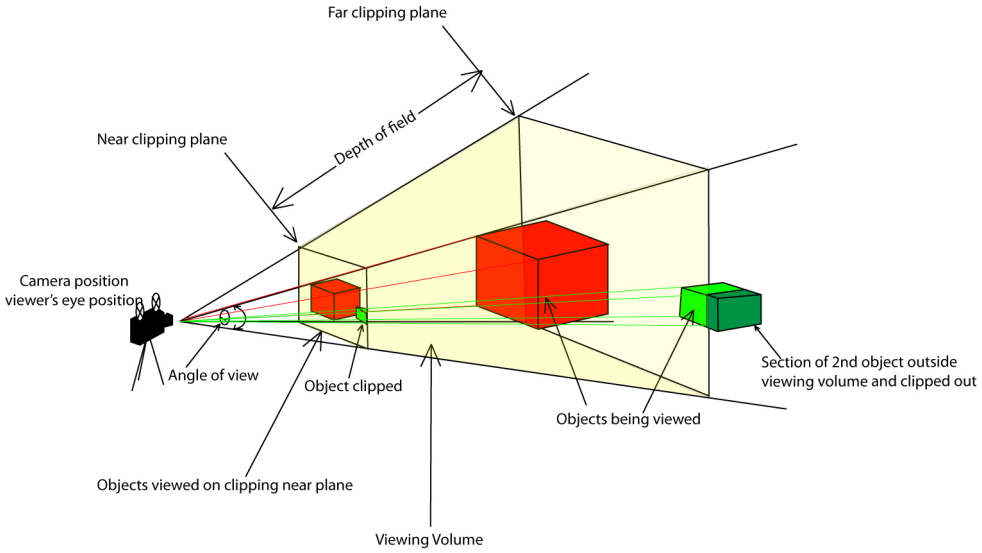



Figure 2.15: Basic set-up of cameras, planes and views in OpenTK

Vertex positions are fed into the 3D world and are joined together to form lines and polygons. These polygons can then be shaded to desired colours and have appropriate textures with lighting characteristics so as to give a realistic 3D appearance of computer simulated objects on a 2D plane. Everything is scaled down to fit into the 'unit volume' normally 0.0 – 1.0 in $x$, $y$ and $z$ directions where the camera or cameras can be situated in or around it. An illustration of this is shown in Figure 2.16.

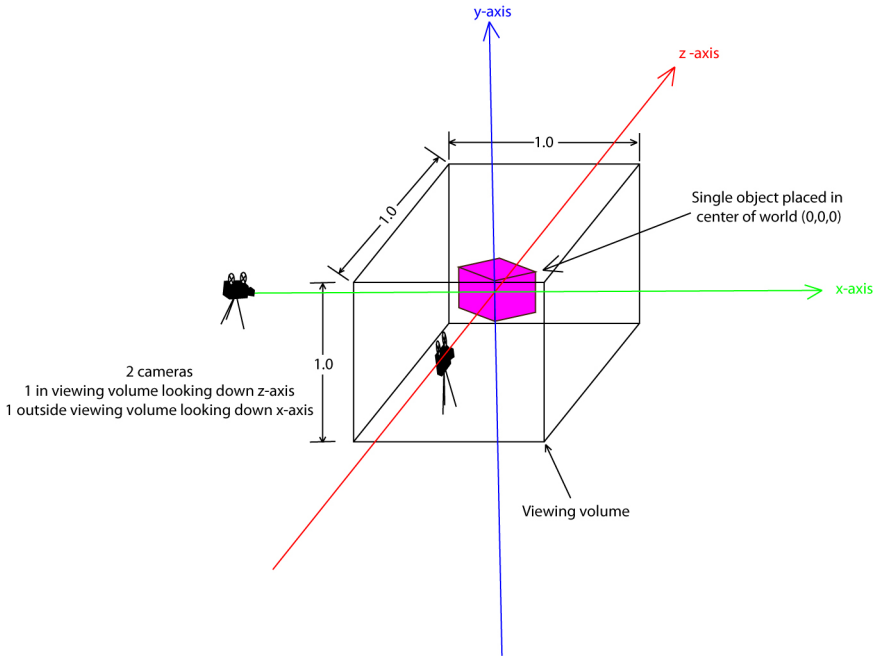



Figure 2.16: Unit volume in OpenTK showing two cameras looking along both alternate axes

### 2.6.1 Lines and Points

Vertex points which are single points in a 3D coordinate system are defined and these can be joined up to form a lines and shapes as follows.

- Points – each point represents 1 pixel.
- Line segments – these are formed by joining points together.
- Line strips – joining a series of line segments together to form curves.
- Line loops – joining start and end vertex points together to form closed circular shapes and polygons.

An illustration of these are shown in Figure 2.17.

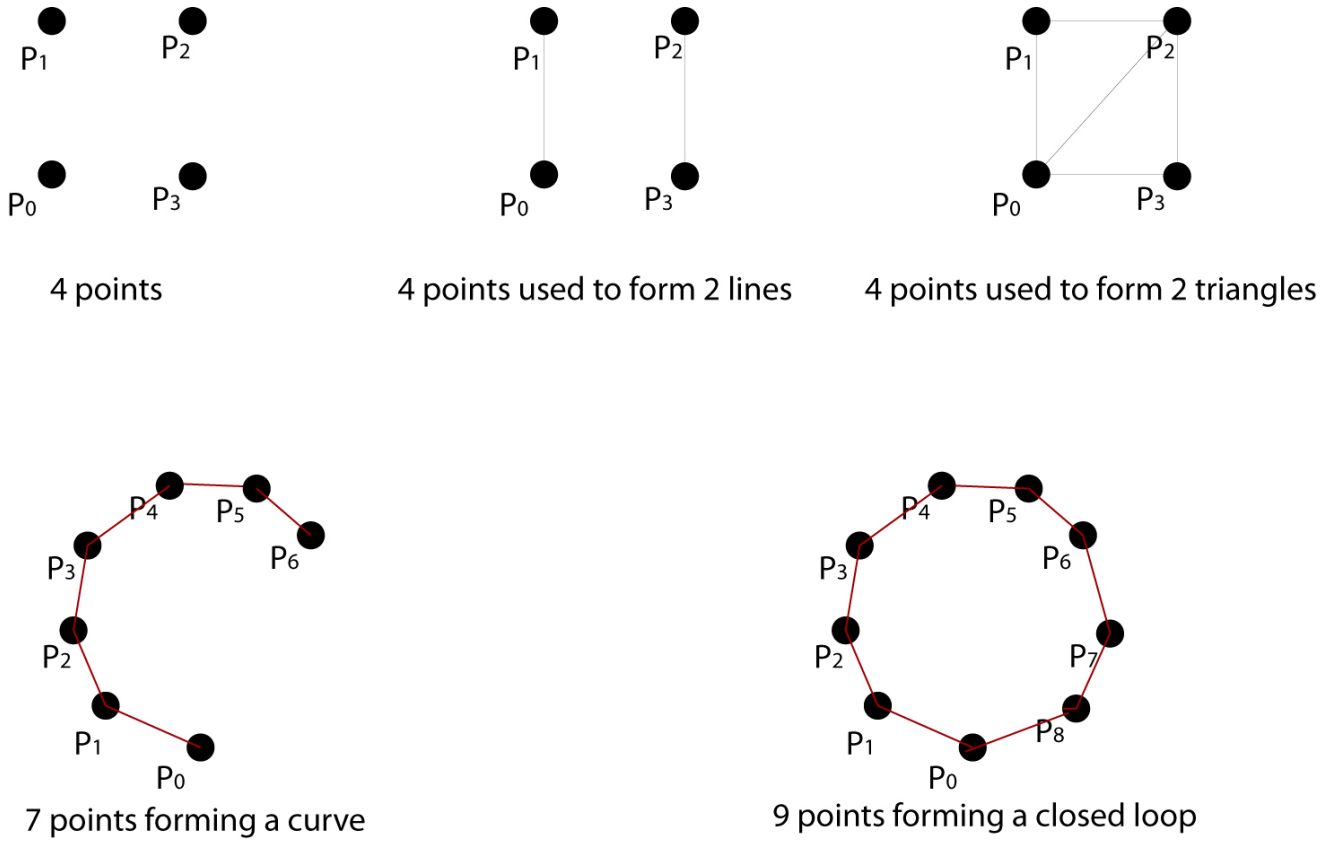



Figure 2.17: Example of points, lines and loops formed by vertices in OpenTK

### 2.6.2 Triangles and Polygons

Polygons are formed using triangles and there are two ways in which these triangles are generally formed and are as follows:

1. Triangle strips – the first 3 points define the first triangle. The next triangle is defined by joining up the first and third point with the next new point. Every subsequent triangle is defined by joining up the last two drawn points with the next new point. An example of a triangle strip comprising of four triangles is illustrated in Figure 2.18 where the first triangle is defined by $\{P_0, P_1, P_2\}$ the next by $\{P_0, P_2, P_3\}$ the next by $\{P_2, P_3, P_4\}$ and the last by $\{P_3, P_4, P_5\}$.

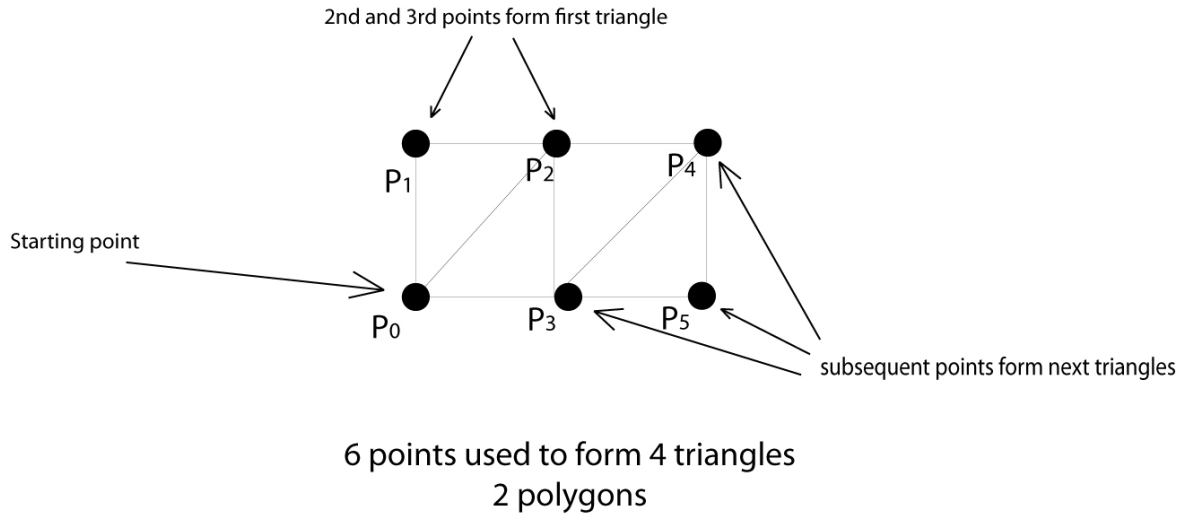



Figure 2.18: Triangle strip example containing four triangles formed by six vertices in OpenTK

2. Triangle fans – the first point defines an origin point at the centre of the fan and the next two points define the first triangle. Every subsequent triangle is defined by joining up the centre origin point, the last point drawn and the next new point. An example of a triangle fan comprising of four triangles is as illustrated in Figure 2.19 where the first triangle is defined by $\{P_0, P_1, P_2\}$ the next by $\{P_0, P_2, P_3\}$ the next by $\{P_0, P_3, P_4\}$ and the last by $\{P_0, P_4, P_5\}$.

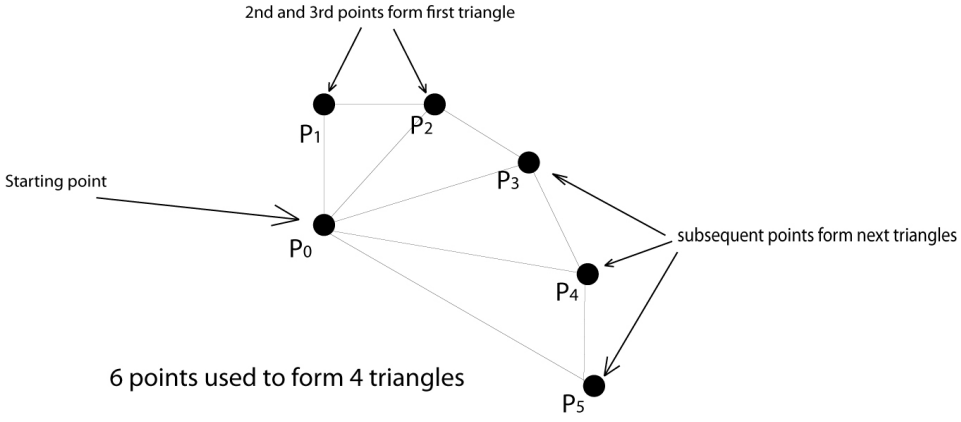



Figure 2.19: Triangle fan example containing four triangles formed by six vertices in OpenTK

Oil reservoir cells are made up of cuboidal shapes, they have eight vertex positions but their sides are not necessarily parallel to one-another. Polygons are represented using two triangles; a grid cell has six faces, twelve triangles. Each vertex is defined in 3D space using three $i, j$ and $k$ co-ordinates, in this research single floating point numbers are used as this is what is used at Sciencesoft. It is possible to draw each of the faces using individual triangles with separate

vertex points or with linked vertex points using triangle fans or strips. It is equally possible to draw the cuboid using two triangle fans. There are advantages and disadvantages to both. It is far simpler to define primitive shapes such as cuboids using triangle fans and strips but more complex when applied to more obscure shapes such as multi-curved bodies. Having lists of single polygon faces could also prove more advantageous during face culling or intersection routines where hidden faces are not rendered or when one body intersects with another and their intersecting faces are omitted. This is because their vertex points can sometimes be calculated easier with less complexity from a list of vertices where their removal does not sub-divide long fans or strips requiring the start and stop points of these fans and strips to be redefined. If drawing a single face at a time and using triangles then a face would contain 6 vertex points meaning that the resulting cuboid would have 36 vertex points defined by 108 single precision $i$, $j$ and $k$ values. Each face could also be drawn using a triangle strip using 4 points resulting in cuboid represented by 24 $i$, $j$, $k$ values, similar to using a triangle strip or less using a series of triangle strips. A more memory efficient way would be to draw the cuboid using two triangle fans, requiring only 14 vertices, illustrated in Figure 2.20.

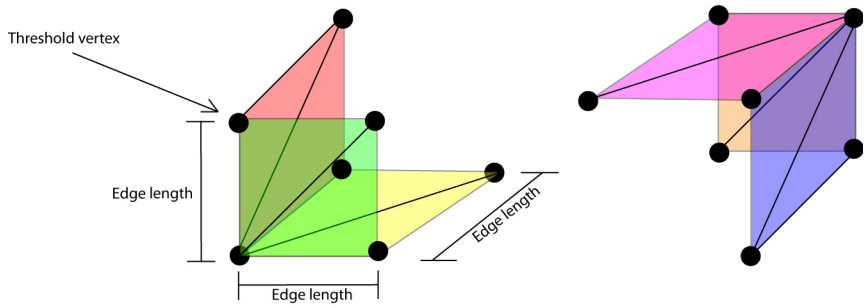



Figure 2.20: Triangle fan concept for drawing all six faces of a node

This approach to drawing faces would only prove efficient if all cell faces were to be drawn but generally the individual 4 vertices which define a cell face are sent as subsequent hidden face culling limits the number of polygons of a triangle fan to be drawn. This is because many cell faces are hidden from view by other cells so that some cells may only require two of its six faces to be drawn on either side of the cell which could not be linked up using triangle fans and strips.

### 2.6.3 Frame Buffer

The graphics card renders the entire scene using the frame buffer and displays it on screen typically between 60 and 100 times per second, referred to as the refresh rate (Angel & Dave, 2012). Fast refresh rates generate smooth animated simulations but this has to be lowered sometimes due to the lag in rendering large scenes from the frame buffer having to compute large quantities of calculations. In order to help compensate for this, double buffering is utilised as opposed to the default single buffering and that this is accomplished with the introduction of *front* and *back* buffers. The front buffer image is not removed from the display screen until the back buffer image has been drawn by the graphics card and vice versa, eliminating some *jerkiness*.

Screenshots can be captured from the display at runtime as a .PNG or bitmap file. Animations can be produced by capturing screenshots depending on how complicated a scene is. If the refresh rate is set higher than that which the graphics card can draw then a proportion of the frames will be identical and not updated allowing the graphics card to catch up. To get a smoother animation, sometimes it is best to reduce the refresh rate and to insert wait functions into the code (Shreiner, 2010).

Each vertex position within the 3D visualisation is defined by its 3D co-ordinates within a 3D space. The file sent to the graphic card for simulation is in the form of a list of vertex positions, normally in the form of single floating point numbers, one for each axis (x, y and z). Following is a detailed look at how this numerical value is stored in memory.

## 2.7 Vertex and Polygon Culling

This thesis deals with compressing oil reservoir grids using octree compression techniques and proposes a method of generating low resolution visual models using the information stored at each level in the tree in a hierarchical pyramid (see chapters6 and 7) culling hidden geometry and only rendering the information required by the viewer. Sending fewer vertices to the GPU results in faster refresh rates and would allow larger grid models to be loaded. For this reason, the following section of this thesis discusses other methods of vertex and polygon culling.

One could look at the surface of an oil reservoir simulated grid as a terrain similar to that used in computer games (Seixas *et al.* , 1999). In computer games, as well as in geographical information systems (GIS) the terrain model can be defined using a digital elevation model (DEM) which is a 2D array of height values for each voxel in the scene (Pouderoux *et al.* ,

2004). In computer games line-of-sight (*LOS*) refers to the direct visual path between objects in the scene, whether objects cannot directly see one-another due to obstacles between them (Lee *et al.* , 2013) used extensively in first person shooter games (*FPS*) (Vicencio-Moreira *et al.* , 2014; Adams, 2014; Tremblay & Verbrugge, 2013). If an object *B* is known to be hidden from view, from the users camera position *A*, then object *B* does not require rendering and can be *culled* (Zaugg & Egbert, 2001) from the scene so that only vertex information required for the scene is sent to the GPU.

There are several methods which are applied to reduce the vertex information sent to the GPU, some apply a shortened viewing volume, reducing the boundaries of the scene *clip planes (Fowler* et al. *, 2000)* and thus reducing the initial vertex information in the scene before any polygon culling or voxel re-alignment algorithms. It is often the case that scene backgrounds are merely 2D planes textured with bitmaps as they portray sufficient information about the game environment and many front facing models in a scene have rear polygons removed, reducing the number of polygons in the scene (Steed, 2010). Sometimes objects in the scene, such as foliage is put as a texture on a 2D plane with a transparent background giving the illusion of a 3D object in the distance (Egger *et al.* , 2002; Ahearn, 2014).

One such method of *polygon decimation* uses low level detailed models of the terrain generated by displaying clusters of vertex information, considered outwith the viewer's main region of interest, in approximated average values, culling unnecessary polygons from the scene (Luebke & Erikson, 1997). The vertex information is compressed using octree compression and stored in a triangle list where regions of vertices occupying scene space less than a specific threshold are shown at lower levels of resolution using the vertex information higher up in the triangle list.

Another method of polygon culling looks at sub-dividing the terrain as a 2D space into regions using quadtree compression methods where the leaf nodes represent partitioned areas of the scene (Cline & Egbert, 2001). Only those polygons within node regions close to the viewing volume are considered for rendering. Some gaming applications, such as Ogre[1], extend this, by partitioning the scene using octree compression.

The use of progressive meshes have been used in computer graphics for many years where a scene, or objects in the scene, are compressed and displayed storing the number of vertices and polygons used to define the object at varied levels of detail, from a coarse low level of resolution model to full resolution containing no loss of information (Hoppe, 1998). In its simplest term the list of meshes could store a sphere using thousands of vertex positions and triangles to show the model when viewed close to the viewing camera in the scene, but in a

[1]more information can be found at: http://www.ogre3d.org/tikiwiki/SceneManagersFAQ#Octree_Scene_Manage

more box-like fashion, even as a single pixel when far from the viewing camera (Kaick *et al.* , 2014).

Triangular irregular and semi-irregular networks (*TIN*) store a hierarchical stacked arrays of triangle vertex sets, where each set represents the terrain at a more precise level of detail down to the full resolution model (surface model) (De Floriani & Puppo, 1995). The terrain can be rendered using a single triangle vertex set or using a variety of them (multi-resolution) so that the surface of the terrain could be displayed with low levels of detail and regions of interest at higher levels of detail. Semi-irregular do not often generate as detailed a model at lower levels of resolution but yield quicker refresh rates (Cline & Egbert, 2001). Sometimes large data sets, such as terrain surface information is stored as a *datastream*, parsing this data stream can allow one to extract all or part of the terrain information allowing the model to be displayed at varied levels of detail (Skala & Kolingerova, 2011).

The *LOS* methodology described is similar to the periphery node inner cell culling as illustrated in Figure 7.1, chapter 7, where inner leaf node cells are culled as they are known to be hidden from view by their outer leaf node periphery cells. Other face culling algorithms defined in this thesis do not use LOS algorithms for grid cell polygon face culling but instead cull those faces which have a similar active status, are on the same axis plane and share vertex positions.

In oil reservoir engineering, grid models are normally rendered in their entirety with no lessening of the viewing volume so that no far clipping plane culling is applied. These grids are generally rendered with no perspective of distance from the viewer, allowing oil reservoir engineers to always see the grid boundaries.

The regions of interest algorithms detailed in section 7.3 on page 110 is similar to using progressive meshes and *TIN* methods (displaying the full field model at low resolution but regions of interest at high resolution) but instead of the hierarchical tree being formed by clustering vertex positions it instead uses the active cell information of the oil reservoir grid block cells.

# Chapter 3

# Sciencesoft Data Structures

This thesis is solely concerned with structured simulated oil reservoir grids which are 3D representations of the reservoir split into a collection of $N_x$, $N_y$, $N_z\, cells$, each representing a volume of 3-dimensional reservoir rock ($N_x \cdot N_y \cdot N_z\, cells$). Each cell is defined by eight vertices giving a distorted cube and, if active (containing oil), will be included in computer simulations, or if inactive (not containing oil) will be excluded from simulations. Each of these cells have related properties:

- Rock properties – porosity and permeability.
- Solution properties – saturation, pressure, permeability.

This thesis, however, is only concerned with the following six arrays, which store information about a cell's active status and physical geometry:

1. The ACTNUM array – this is an array of all the cells in a grid and is passed out from the simulator. It consists of a list of ones and zeros revealing each cell's active status (1 = active; 0 = inactive).
2. The zero-based Natural-to-Active array (N2A) – this is the list of all cells similar to the ACTNUM array but inactive cells are stored as a minus one and active cells are stored as a positive integer equal to its position value in the list of active cells and is zero based.
3. The Active-to-Natural array (A2N) – this is the list of active cells, each value is an active cell's natural grid position and is a one-based array. This is a compressed representation of the N2A as it only stores active cell indices.
4. Cell Geometry Arrays – this is a list of the *x*, *y* and *z* co-ordinate vertex positions for each cell in the grid. It is a list of unique vertices, where no vertex position is stored more than once. Each cell has eight vertices each defined by three floating point numbers.
5. Vertex pointers – this is used to access the vertex table. Normally the vertex table holds

only unique vertex positions therefore an array of pointers is required which points to these shared vertex position values in the vertex table.

6. Properties arrays – these are arrays of objects which hold all the characteristic values such as oil pressure, saturation values, associated with each active cell.

When displaying reservoir grid models generally cells are visited in a raster-order using the N2A, which is used to index the various cell properties. This could be substituted with the A2N, saving memory and traversal time. Unfortunately the N2A is always required, for example, searching for a cell's natural position. This takes too long using the A2N as this would require performing a linear search through the ACTNUM in order to get to the correct cell position and as many visualisation techniques such as face culling requires visiting neighbouring cells this would increase visualisation times to an unacceptable level.

## 3.1 ACTNUM array

Reservoir grids are sub-divided into cells in logical space. The vertex positions of these cells are referenced with positions in 3D space normally as floating point values ($x$, $y$ and $z$ co-ordinates). Large datasets can contain hundreds of millions of cells each representing a cubic volume of rock around several cubic meters (Uleberg & Kleppe, 1996), typically $(50ft \cdot 50ft \cdot 10ft)$. Some cells contain fluids and others are empty and can be omitted from the simulation. Oil reservoir grids comprise of active and inactive regions and represented as a binary array of ones or zeros; one representing an active cell containing hydrocarbons and zero an inactive empty cell containing no accessible hydrocarbons (Spigler & Maayan, 1985).

## 3.2 N2A array - (Natural-to-Active)

Visualisation of the grid geometry data requires more than just the active status of grid cells since the grid data is generally stored only for the active cells and it is often necessary to convert from a *natural* cell index to the *active* cell index, or, in particular, from the position in ($i$, $j$, $k$) logical space to *active* cell index so a list is created equal to the number of cells $(N_x \cdot N_y \cdot N_z)$ which stores inactive cells as a minus one and active cells as positive integers, each represented as a value equal to its position in the list of active cells. The length of this array is the total number of cells in the grid and each active value is used to reference the indirectory (an array of pointers) which points to the single floating point x, y and z vertex positions of cell vertices in the vertex table. The list is populated in a rasterscan fashion (x-axis

then y-axis then z-axis).

With very large grids (multi-million cell grids) this N2A array takes up a lot of computer memory, for example a grid $154 \cdot 85 \cdot 1975\, cells$ would occupy almost 99 MB of RAM. This thesis looks at how this can be dramatically reduced by replacing this array with a compressed octree representation of its active cells.

## 3.3 A2N array - (Active-to-Natural)

This array is called the A2N array and is a linear array equal in length to the number of active cells, as it only stores the natural cell position index (iActive value) of each active cell. Since property values are only stored for active cells it is often necessary to lookup a cell's natural cell index to get its associated properties. Using these arrays it is possible to reference back and forth between the natural cell indices, ($i$, $j$, $k$) indices and the active cell indices. so that given a cell's N2A value its natural index can be deduced by referencing the A2N and vice versa. Figure 3.1 shows a simplified 2D representation of this linkage between the ACTNUM, N2A, A2N and properties arrays using a $5 \cdot 6\, cell$ grid.

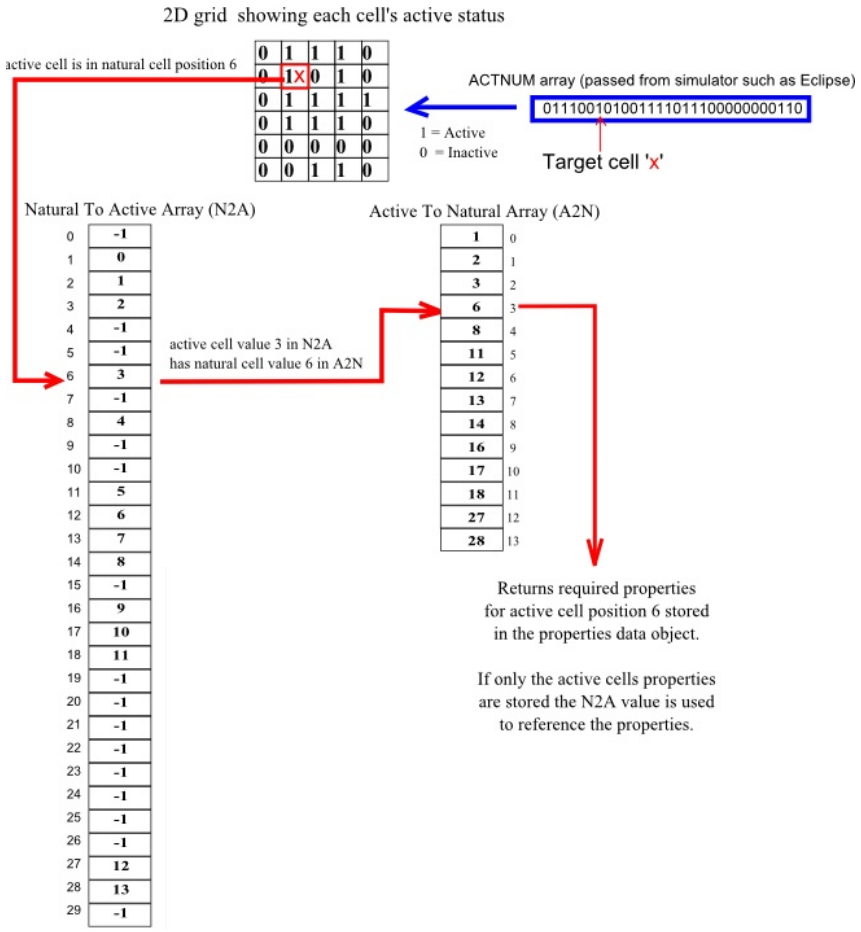



Figure 3.1: Linkage between Sciencesoft's ACTNUM, N2A, A2N arrays and property arrays

## 3.4 Vertex Tables

The vertex table is the list of $(x, y, z)$ co-ordinates of each vertex of each cell and is loaded into memory at runtime and using either the N2A or the natural cell index each cell's corresponding vertex co-ordinates can be located. If a vertex table containing vertices for all the cells are loaded (active and inactive) then the cell's associated vertex positions can be located using the cell's natural cell index to reference the vertex table. Natural cell co-ordinates are referenced using (x, y, z) and vertices, (i, j, k). Figure 3.2 on the following page shows a simple 3D grid where an example of a cell's natural index (iCell value) is deduced using its $i, j$ and $k$ co-ordinates. Vertex co-ordinates of cells in oil reservoir grids alter most in depth then width then height due to their thin layering characteristics. For this reason the vertex table is populated in k, j then i order.

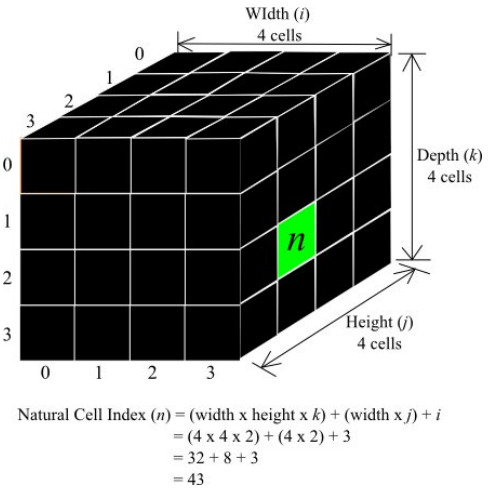



Figure 3.2: 3D grid illustrating how the natural cell position (iCell value) of a cell is calculated using its ($i$, $j$, $k$) co-ordinates

The vertices are stored in a 1-D list, with the vertices for each all grouped together. The cells are ordered with the $i$-index varying fastest, the $j$-index, then the $k$-index (the vertices are stored by layer). 8 vertices of each cell were ordered as shown in Figure 3.3. The natural cell index value is used to reference the first $z$ vertex position value of its first vertex (vertex 0).

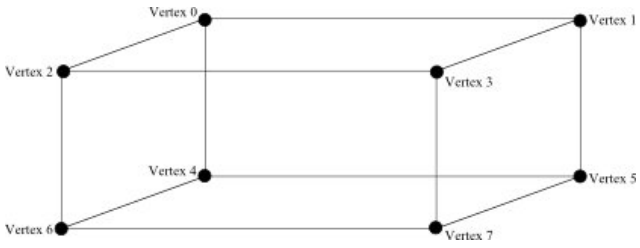



Figure 3.3: Grid cell vertex position numbering

Often only the active cell vertex positions are stored due to the memory overhead of storing millions of inactive cells (compressed vertex table). When a vertex table containing all the vertices of all the cells is loaded, inactive cells can be drawn. If a vertex table containing only active cells is loaded then the cell's active cell index (iActive value found in the N2A array) is used to locate its vertices. In all cases the N2A is equal in length to the total number of cells.

There are two main ways in which vertex co-ordinates are defined for cells within a reservoir grid model.

- Block-centred grids – block-centred grids are defined in terms of cell sizes ($D_x$, $D_y$, $D_z$) – for each cell along with the top depth of the top layer or sometimes every layer of cells. The grid compresses a collection of the $N_x$, $N_Y$, $N_z$ cuboids with varying sizes defined by the ($D_x$, $D_y$, $D_z$) arrays.

- Corner-point grids – corner-point grids are more general as all eight vertices are specified allowing the grid cells to have uniform non-cuboidal shapes.

The data supplied such as vertex positions and property values were hidden from the researcher due to the architecture of the software. Their vertex geometry were defined using the seismic data fed into the simulator model. Experiments conducted outside Sciencesoft's software applied vertex positions to cells based on corner point geometry where a cell's logical x, y and z co-ordinates determined each cell's vertex 0 position. The remainder of the vertex positions of the cell were determined by means of simple offsets in x , y and z directions forming uniform cubic shapes. In occasions a staggering offset was applied replicating faulting characteristics sometimes found in oil reservoir grids. Simple modification of the offsets could result in more elongated cuboids more closely mimicking cell shapes found in actual reservoir grids.

The vertex table only contains unique values (unique vertex table) storing all cell vertex information or only for active cells (compressed vertex table). This can save up to 80% of memory especially in grids where cells adjoin one-another and no faults or fractures exist since the grid represents a partitioning of real space, usually adjacent cells touch one-another and have perfectly matching faces in alignment so that adjacent cells share vertices on adjoining faces. With *perfect convex grids* a single cell may have 26 neighbouring cells which it touches so that each of its vertices can usually be shared by eight adjoining cells. In the case of very large grids, boundary cells (where the vertices are shared by less cells) are unimportant, there is, on average, one unique vertex per cell resulting in a significant saving in memory when only storing unique vertex positions. Indexing the vertex table therefore requires a level of indirection using a pointer to a shared vertex. Tables in the form of linear arrays store the these pointers (indirectories). A primary level of indirection is required to point to the vertex table if all cell vertices are loaded using a cell's iCell value, but a compressed vertex table (where only active cell vertices are loaded) requires a second level of indirection. In this case a second table of pointers is stored which points to the first level of indirection pointers (referenced using a cell's iActive value – active cell number in a list of active cells, the A2N value). These indirectories point to the first position in the vertex table where a cell's first vertex (vertex_0) can be found. By using a compressed vertex table which only stores the unique vertex positions of active cells a substantial saving in memory storage can be made.

Vertex tables were supplied by Sciencesoft during experiments conducted at their premises, discussed in Chapter 5, but dummy vertex positions can be generated if required and indeed this was performed in order to show the pyramid visualisations given in Chapter 6. In order to generate a unique vertex table from a 3D grid, each cell's positioning within the grid was first established by checking its co-ordinates with the grid's dimensions in a rasterscan order.

This establishes which of the cell's vertices had to be checked with neighbouring cell vertices, evaluated from *vertex_0* to *vertex_7* (see Figure 3.3). Given a boundary cell, only the inner vertices required checking, and only those which have not already been checked from the previous row or column in the rasterscan traversal but inner cells only require their eighth vertex *(vertex_7)* to be evaluated with its neighbouring cells.

## 3.5 Indirectories

Up to eight vertices may share a single vertex position and the indirectory stores pointers to these values and as each unique vertex was established it was written to a unique vertex table and an indirectory was populated with pointers to each them. This indirectory contains eight times the total number of cells and each index in the indirectory corresponds to a single cell vertex, so that given a cell's *i*, *j*, *k* position or *iCell* value a cross-match can be made to the indirectory which stores pointers for all of its eight vertices in the unique vertex table.

When creating a compressed vertex table only active neighbouring cells vertices are evaluated. The primary indirectory contains eight pointers, one for each active cell vertex position and a second indirectory is used to store pointers to this indirectory and contains the *iActive* value of each active cell.

## 3.6 Summary

In this chapter the state-of-the-art methodologies for storing grid geometry data adopted by Sciencesoft, have been set out. Arrays such as the ACTNUM array are standard, written out by the simulation software. The N2A and A2N arrays are typical industry standard techniques used to store grid cell active status information and mapping to cell property values. However, there are many other fields where multi-dimensional datasets containing millions of cells are used to store attributes, such as with earthquake simulations (Ma *et al.* , 2003). Many systems are modelled using 3D grids containing cells which have corresponding properties normally accessed using indices into auxiliary arrays, for example, crop cultivation or weather simulations where a properties arrays holds all the characteristics for each cell. In the case of weather simulators this could be cloud formation, temperature, wind speed and direction; with crop simulators, soil nitrate levels (Jones *et al.* , 2003).

Some of these properties can have mathematically computed formulae applied to them in order to simulate weather forecasts or for determining how well buildings in a certain area

would cope with a natural disaster such a hurricane (Friedman, 1972). This means that the grid can be loaded independently of its properties and these various characteristics can be loaded onto the grid for visualisation at runtime.

With oil reservoirs the property array holds values such as porosity, permeability, pressure, oil, water, gas or saturation and can be displayed on the grid as required. In the case of the octree example the property index for each cell in an active leaf node is stored as an array of pointers. Each pointer value in this array is a cell's active cell index'. This is the position of an active cell in an ordered list of all active cells. The position of the cell in the list of all cells, active and inactive is the cell's natural cell index.

Reservoir grid geometry and property data is stored in arrays which yield extremely fast direct lookup access times and are well suited for multi-dimensional scanning techniques. This research looks at compressing the N2A to remove the large memory overhead required by this array integers, one for each cell and presents the novel approach of substituting the large N2A arrays containing all cells with a compressed octree representation. It was proven through the initial experiments (detailed in Chapter 5) that octree compression is well suited for compressing reservoir cell data due to oil reservoir active cell clustering characteristics.

Figure 3.4 on the following page illustrates the previously given 2D example of a small 5 x 5 grid. It shows the grid and linkage between its ACTNUM, N2A, A2N, indirectory and vertex table. Inactive cells are stored as '-1' in the N2A.

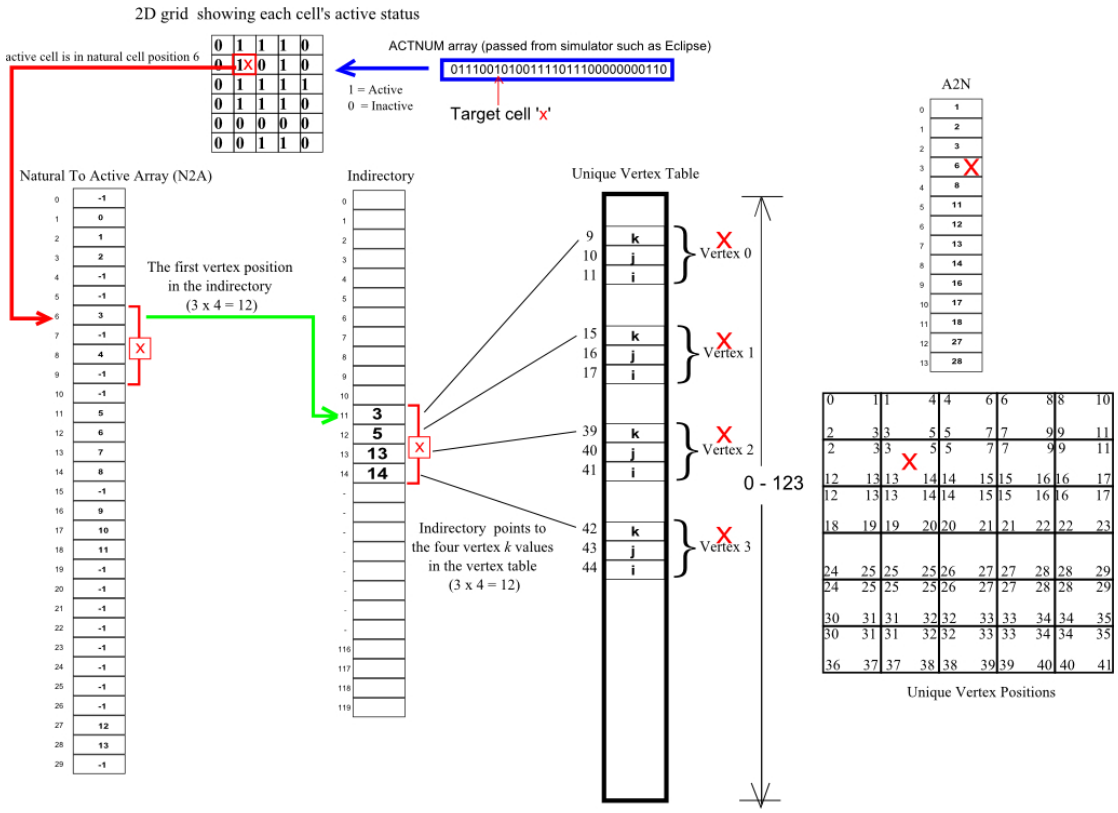



Figure 3.4: Sciencesoft's current ACTNUM, N2A, A2N and indirectory arrays and unique vertex table linkage

# Chapter 4

# Problems and Solutions

The following sections of this chapter outline the initial problems and programming language constraints of applying the algorithms developed so far to oil reservoir simulated models. These were partly due to the programming language adopted, C#. This programming language was chosen as it was intended that the algorithms developed in this thesis would be tested in Sciencesoft's existing software which was written in C#.

## 4.1 Test Grids

In order to test the algorithms, Sciencesoft provided a set of 36 oil reservoir grid models in the form of N2A arrays discussed in the previous chapter. Table 4.1 details their dimensions and active cell values listed by grid size as number of cells. These grids were used in the experiments detailed in this Chapter 5 of this thesis.

| Model | Grid dimensions | Total number of cells | Total active cells | Total inactive cells | Active Percentage (%) |
|---|---|---|---|---|---|
| 1 | 32x48x10 | 15360 | 8510 | 6850 | 55.40 |
| 2 | 43x67x20 | 57620 | 33342 | 24278 | 57.86 |
| 3 | 20x20x184 | 73600 | 72128 | 1472 | 98 |
| 4 | 26x68x44 | 77792 | 30835 | 46957 | 39.63 |
| 5 | 97x66x13 | 83226 | 13787 | 69439 | 16.57 |
| 6 | 30x30x100 | 90000 | 86444 | 3556 | 96.05 |
| 7 | 107x157x8 | 134392 | 62260 | 72132 | 46.33 |
| 8 | 56x99x25 | 138600 | 28705 | 109895 | 20.71 |
| 9 | 96x83x18 | 143424 | 79115 | 64309 | 55.17 |
| 10 | 241x125x5 | 150625 | 13709 | 136916 | 9.10 |
| 11 | 78x139x14 | 151788 | 132972 | 18816 | 87.60 |
| 12 | 40x74x60 | 177600 | 17442 | 160158 | 9.82 |
| 13 | 102x24x89 | 217872 | 52693 | 165179 | 24.19 |
| 14 | 104x40x59 | 245440 | 194051 | 51389 | 79.06 |
| 15 | 37x110x72 | 293040 | 206264 | 86776 | 70.39 |
| 16 | 55x55x103 | 311575 | 115932 | 195643 | 37.21 |
| 17 | 74x157x28 | 325304 | 77945 | 247359 | 23.96 |
| 18 | 135x102x24 | 330480 | 60286 | 270194 | 18.24 |
| 19 | 73x139x34 | 344998 | 191471 | 153527 | 55.50 |
| 20 | 114x144x33 | 541728 | 167367 | 374361 | 30.90 |
| 21 | 85x123x60 | 627300 | 13520 | 613780 | 2.16 |
| 22 | 60x147x76 | 670320 | 129993 | 540327 | 19.39 |
| 23 | 108x144x51 | 793152 | 506596 | 286556 | 63.87 |
| 24 | 211x64x60 | 810240 | 290784 | 519456 | 35.89 |
| 25 | 149x66x100 | 983400 | 88226 | 895174 | 8.97 |
| 26 | 180x91x75 | 1228500 | 147518 | 1080982 | 12.01 |
| 27 | 256x256x20 | 1310720 | 1310720 | 0 | 100 |
| 28 | 89x131x115 | 1340785 | 62237 | 1278548 | 4.64 |
| 29 | 50x114x326 | 1858200 | 342415 | 1515785 | 18.43 |
| 30 | 105x133x137 | 1913205 | 695681 | 1217524 | 36.36 |
| 31 | 118x109x169 | 2173678 | 1109411 | 1064267 | 51.038 |
| 32 | 196x129x105 | 2654820 | 1494130 | 1160690 | 56.28 |
| 33 | 80x160x211 | 2700800 | 491248 | 2209552 | 18.19 |
| 34 | 95x155x250 | 3681250 | 182004 | 3499246 | 4.94 |
| 35 | 300x300x10 | 9000000 | 9000000 | 0 | 100 |
| 36 | 154x85x1975 | 25852750 | 771509 | 25081241 | 2.98 |

Table 4.1: Grid structures of the 36 oil reservoir test grids supplied by Sciencesoft

## 4.2 Hierarchical Octree Memory Overhead

In order to fully test the memory overhead of the hierarchical octree structure in memory it was first integrated into a prototype of Sciencesoft's S3GRAF-3D software replacing their N2A

array. As C# was chosen as the programming for integration with Sciencesoft's software, the hierarchical octree created using C# is that of an abstract class. The main base class, points to the root of the tree (root node). The header and leaf nodes (concrete classes) hold values such as pointers or payloads.

Even a *pruned* octree still has a large memory overhead. A large proportion of the storage required for the tree comes from header node pointers (64-bit) as due to the scale of these grids, 64-bit processor computers are normally required. Previous research into compressing reservoir grids has been conducted using C# using a .Net environment, and pointers are 8 bytes long and as an object reference is 16 bytes (Hejlsberg *et al.* , 2006). This means that a class occupies 24 bytes, 8 bytes for a pointer to the class and 16 bytes for an object reference. Header nodes occupy 80 bytes in total and leaf nodes 24 bytes. This was verified when memory usage was measured using Redgate's ANTS Memory Profiler[1] within a .Net environment (Visual Studio 10) at Sciencesoft.

Inactive nodes are *pruned* reducing the memory overhead, header nodes can have up to 8 pointers, when an inactive leaf node has been *pruned* their parent header node is left with a null pointer which still occupies 8 bytes in memory using a 64 bit operating system. In extremely large reservoir grids where pockets of hydrocarbons are distributed sporadically, increased numbers of small leaf nodes, generating many more pointers would occupy more space in memory. This increased memory allocation would greatly impact on resulting compression ratios in C# but may not be present in other programming languages, such as in a functional programming language for example '*C*'.

### 4.2.1 Summary

Octrees are renowned for their fast searching abilities and the initial results proved this to be true, they are easily traversed, but programming language constraints required the inclusion of pointers, coupled with C# class storage demands which generated large unforeseen hidden memory overheads. Although permitting fast cell lookups when searching the tree in tree-order, non-contiguous searches such as those adopted in reservoir visualisation software applications indicated that when compared to direct access, this data structure may prove to perform up to 6 times slower as each search would be instigated from the root node.

What was required was a data structure which could adapt to the demands of 3D oil reservoir visualisation, yielding lookup times matching those used in industry while remaining in memory in a highly compressed state. Sciencesoft expected that any redesign of their data

[1]Redgate's ANTS Memory Profiler can be found at http:/www.red-gate.com

structure would have a detrimental impact on performance times and were willing to accept a fifteen to twenty percent impact on performance, as this could be offset by the compression achieved. A solutions to these problems was developed which eliminated the class overheads of C# and dramatically reduced the size of header node pointers. Inactive child null pointers were no longer required, whilst still retaining the high level of lossless compression generated using octree compression techniques where just as importantly the resulting data structure still matched, and sometimes out-performed state-of-the-art lookup times. This data structure and algorithms used to generate it and its searching and visualisation techniques are discussed in the following sections.

## 4.3 Solution

The solution was to flatten and *prun*e the hierarchical octree into a linear data structure. This was stored sequentially in memory as a 1D array containing pointers to active leaf nodes and header nodes with a *flag* to indicate the active status of its children. This flattened octree could be traversed in a similar fashion to the hierarchical octree without having to store added skipping information during individual cell lookups in an order other than the order it was written. The null pointers and class overheads associated with the C# hierarchical octree structure were removed and remaining pointers were now stored as 32-bit integers even in a 64-bit operating system. The following sub-sections of this chapter define this novel data structure and details all algorithms and techniques adopted to test its performance using Sciencesoft's S3GRAF-3D software and prototype applications.

## 4.4 Property Array

In many systems, 3D grid cells have corresponding properties which are normally accessed using indices into auxiliary arrays where a properties array holds characteristics cells. These properties are derived from complex mathematically computed formulae and can be loaded onto the grid and rendered at runtime as required. These arrays of properties hold values such as porosity, permeability, oil or gas pressure. The octree example has the property index for each cell stored as a pointer in each active leaf node, a cell's active cell index (iActive value) – the position of an active cell in an ordered list of all active cells. The cell position of a cell within the list of all cells is referred to as a cell's natural cell index (iCell value). These are illustrated in Figure 4.1 on the next page which illustrates a linear array of 10 cells. The shaded cells are inactive cells where each has a natural cell index in base 0 from 0-9 and an

active cell index in base 1.

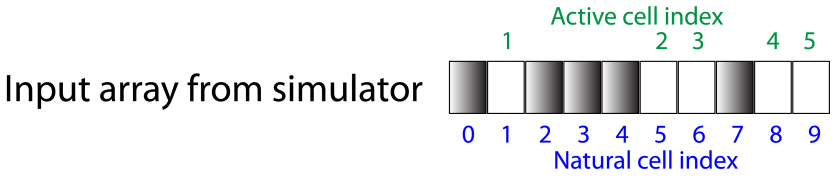



Figure 4.1: Active and natural cell indices of a 10 cell input array containing 5 active cells

## 4.5 Array of Structs (structArray)

Reducing the class overheads associated with the octree node classes comprised of:

1. A linear array of structs equal in length to the number of nodes in the *pruned* octree. The structs are data structures used to store a collection of variables, unlike a class it can be stored on the '*Stack'* and so does not have the same memory overhead as a class, using C# (object orientated languages). The structs are stored in an array, where each node is addressed and stored, sequentially in memory – yielding better cell lookup performance times. Adding objects such as an list of integers or a method to the stack can be thought of as building up a tower of containers where each container holds one of these integer containers or the method with all its member containers stacked in the order they were added (Sharp, 2010) and when the method is finished all its containers and member's containers are no longer required and removed from the stack (Schildt, 2008). Values from the stack can be removed and edited but only from the top in a last-in, first-out fashion (LIFO) (Solis, 2010) and explains how accessing containers elsewhere in the stack is performed using '*pushing*', '*popping*' methods. C# stores objects on the '*Heap*' which unlike the stack stores all its members in a non-sequential fashion where members added are put into empty containers and can be added and deleted in any order (Solis, 2010). The stack stores value types such as primitives (integers, strings, etc.) or user defined types such as structs and the heap stores value types such as classes and arrays where a container holds the object's data and container holds a reference used to access it which can be stored on the stack or the heap. In C# the stack has an internal array for holding its members and like other arrays is limited by the number of elements it can store limited by integers slightly under $2^{31}$ and both the stack and the heap are restricted by the computer's architecture used governed by available RAM.
2. This array of structs (*structArray*) stores two variables, an eight bit *header flag* and a four byte pointer. The terms *flag* and *pointer* with reference to the structArray are explained in more detail in subsequent sub-sections of this chapter.

3. Header nodes no longer contain null pointers to inactive leaf nodes. Their omission reduces the memory overhead required using hierarchical octree structures.
4. The struct's fields are aligned in memory automatically by the compiler in 4 byte blocks. This means that a byte variable will be stored along with 3 bytes of padding within a 4 byte container. To save on memory the fields of a struct can be packed. This misaligns the bytes and omits the need for padding bits. This misalignment sometimes impacts on performance due to the compiler having to remember variable lengths in order to select the correct number of bytes at the correct memory address but as memory compression was more important in this research, than performance, the structs were packed, as illustrated in Figure 4.2.

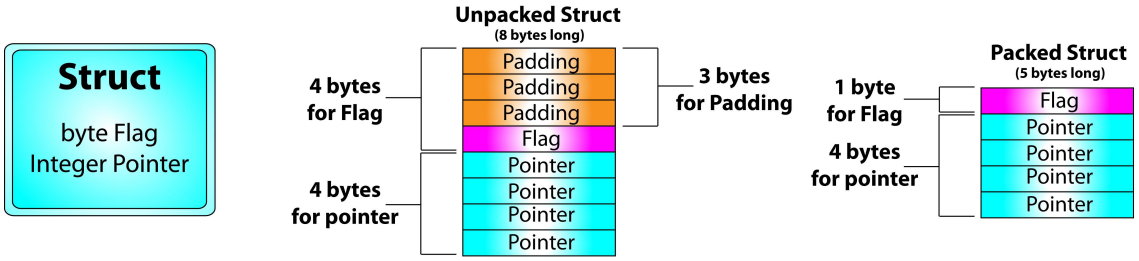



Figure 4.2: Illustration showing padding and packing of structs in memory

4. Stored pointers indexed the positions of child nodes in the linear array. They are stored as integer types, defined in memory as four bytes, even with a 64 bit computer, halving the header node pointer overhead required for the hierarchical octree structure in C#.
5. It is a common industry practice to have an array which stores active cell, *iActive* values, used to index the properties array for retrieving the various property values, in an array format, similar to Sciencesoft's A2N array. Another level of compression was therefore achievable using leaf node knowledge where a linear array was created which only stored the one index of each cell, in a row of cells within each leaf node. All *iActive* values could be deduced using this method as these values were stored in a known sequence, indexed sequentially in a rasterscan order. This compressed array (*compIndArray*) and its significance is discussed in more detail later in this chapter and an illustration is given in Figure 4.3, which shows how the *compIndArray* stored only those, leftmost values within active leaf nodes, and an illustration of how it is used to index the properties array is illustrated in Figure 4.4.
6. The class overheads associated with every node in the hierarchical octree were also removed as the only two reference types stored in memory were the structArray and compIndArray amounting to a combined memory overhead of only 48 bytes per grid, 16 bytes to store the object and 8 bytes to point to them in memory.

The octree is stored as an array of structs (*structArray*) where each struct holds a *flag* byte and a *pointer*. Non-zero flags indicate the active status of the header node's child nodes and

the pointer points to the position in the array of its first active child node. Zero flags indicate leaf nodes and the pointer points to the an array which stores only the active cell indices (compIndArray) and is used to index the properties array.

## 4.6 Tree Construction

The linear input from the simulator, containing active cell information (*input* array) ones and zeros, ones are active, zeros are inactive. The industry standard practice is to take the *input* array and generate a new array which stores inactive cells as '0's and as everything else is taken to be active, is given an *iActive* value equal to its active position within the list of active cells as illustrated in Figure 4.1.

The octree compression algorithm sub-divides the grid down to its lowest level of single cells using recursion. The nodes in this data structure are no longer C# classes but instead structs containing a flag byte and a integer pointer. Starting at the lowest level of the tree, each set of eight leaf nodes are evaluated with one-another to see if they share the same active statues, if they do then these nodes can be discarded and replaced by a node which represents the volume which the eight discarded nodes did. The recursive function continues up the tree and evaluates the next set of eight nodes and performs the same node matching evaluations. This process is continued up the tree to the root node and when a set of nodes evaluated do not all share the same active status, a header node is created and the eight child leaf nodes are stored as either active or inactive leaf nodes. The path the nodes are visited and written to the list of structs is in breadth-order where nodes are visited and written to the list starting from a header node's first child to its eighth. The *structArray* size cannot be pre-determined by looking at the *input* array, and as the programming language used was C#, a list type data structure was used to initially store the nodes. Once the whole tree had been created this list was simply converted to a standard array data type containing the structs (a four byte *pointer* and a one byte *flag* in each). The array generated from the *input* array contained cell, *iActive* values, but in this research a compressed representation of this array was developed (*compIndArray*). Again this array size could not be predetermined, so was initially generated as a dynamic list and subsequently converted to an integer array when the *structArray* was generated. The structArray's *flag* was used to indicate the active status of child nodes and leaf nodes, its *pointers* pointed to each of the header node's active child node positions within the *structArray* and in the active leaf nodes, to the *compIndArray*, the first position, of the first cell, of the leaf node. The pseudo code for creating the *structArray* and *compIndArray* can be found in Appendix section 10.9.

## 4.7 Octree, Lists to Array Structures

As the size of the *structArray* cannot be predetermined a dynamically growing array type structure was used to suit C# this was a list of structs. A static array could have been used but would have to have been initialised much larger than expected to be sure to be large enough to hold all nodes, a far less inefficient approach. Lists can be used to store any data type and expanded dynamically at runtime by the number of elements added to it so that it is only as long as the number elements it is required to store (Sharp, 2010). The problem they pose is that they can be extremely slow to search in comparison to arrays, although similar to a static array, the lookup time for a list can be up to O(n) whereas once the data is placed within an C# array structure can yield lookup times up to O(1). It is for these reasons that both lists were converted to a standard static array structure.

## 4.8 Octant Naming Conventions

The following octant child node, naming convention was used (Dyer *et al.* , 1980) and (Ayala *et al.* , 1985): {NW_0, NE_0, SW_0, SE_0, NW_1, NE_1, SW_1, SE_1}. Child nodes were visited and written to the *structArray* in this order (breadth-first-order) as all active child nodes were written or visited from its first child to its last. The *structArray* can then be traversed by searching with $x$, $y$ and $z$ co-ordinates or by natural position indices (*iCell* values) to return a cell's active status or active cell index (*iActive* values). The following sub-sections in this chapter detail how the *compIndArray* is constructed and indexed using the the *structArray's* header and leaf node flags and pointers.

The compression algorithm presented in this research is lossless as every cell can be recreated no matter why it resides in the tree. When a cell is searched for the recursive algorithm traverses to the leaf node which represents the 3D volume in space where that cell resides. As the recursive algorithm works by power-of-two sub-divisions, its 3D position and node edge length (number of cells in x, y or z-axis) can be calculated. Knowing these values mean that all cells within that leaf node and their positions can be deduced so that the original grid can be reconstructed exactly as it was before decompression with no degree of error.

## 4.9 The structArray Header Flag

The structArray's *header flag* was an 8-bit byte, one bit to indicate the active status of each of its child nodes as shown in Figure 2.8. This illustration shows how a header node's eight child leaf nodes are referenced {NW_0 .....SE_1}. An active child node was represented by a 'on' or '1' otherwise as 'off' or '0', e.g. given a header node with one active child node, SW_1, its *header flag* would equal 2 and be represented in binary as {00000010}.

As no headers are stored which do not have any active children this eliminates the possibility of a header flag having a zero value. This fact is exploited and so a *header flag* of zero was used to indicate the presence of an active leaf node and its *pointer* value points to the position in the *compIndArray* where the *iActive* value of the cell at the leaf node's origin position is stored. Non-zero *header flags* indicate header nodes and their *pointer* values, the position of their first active child node in the *structArray*. This allowed header node *flags* to be used for traversing the *structArray* in a hierarchical tree-like manner as each bit in the *flag* could be was used as a direction indicator.

## 4.10 The Compressed Indirectory (compIndArray)

A pre-requisite of the system was to be able to return cell active cell indices, the indirectory developed in the thesis (*compIndArray*) stored cell (*iActive*) values, used by reservoir visualisation software to reference cell properties such as porosity, permeability, etc and applied to the grid model at runtime. This array was stored in a compressed state as not all the values were required to be stored, only those values on the leftmost side of leaf nodes, as illustrated in Figure 4.3 on the following page. It details how the remaining node's *iActive* cell values can be deduced because the *iAcitve* numbering was generated by scanning through the grid.

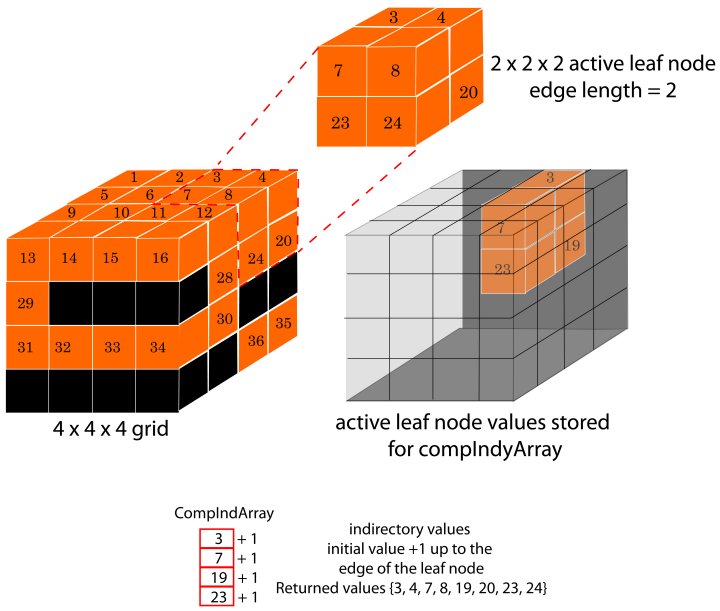



Figure 4.3: Example showing the required inner cell values of a 4 x 4 x 4 cell leaf node

## 4.11 Traversing and searching the structArray

Traversing the *structArray* to a particular grid cell (*target* cell) is performed using its logical 3D ($x$, $y$, $z$) co-ordinates or its *iCell* value (a cell's natural linear cell index within the list of cells) and represented the two main *target* cell searching styles performed during reservoir grid visualisations, generally 3D scanning using ($x$, $y$, $z$) co-ordinates in a treble-for-loop and 1D scanning using a single *iCell* value with a single-for-loop.

The binary representation of *target* cell 3D co-ordinates (*targetValues*) were used to navigate down through the tree, starting with its most significant leading bits of its ($x$, $y$, $z$) co-ordinates. As the *target* cell is searched the tree is descended, with each descent, the bits being evaluated were the bits at one position to the right of the previous bits that were previously evaluated, so that eventually the least significant digits would be evaluated. As the eight *header flag* bits, each represent one of the eight child node directions in the *structArray* and their active status one can traverse the tree to a *target* cell starting from the end of the *structArray* (root node). The structs were added to the original list as header and active leaf nodes were generated so that the root node is the last node to be written as a bottom-up algorithm was used. This could easily be reversed and would mean traversals would instead start at the beginning of the *structArray* travelling downwards.

### 4.11.1 Header Flag activeFlagBits

When searching the *structArray*, the *header flag* is first evaluated to ascertain whether the struct being evaluated is a header node or leaf node, a flag with a zero value indicates an active leaf node and anything else is a pointer to its first written active child node. During the traversal the number of its '*on'* bits are counted as this indicated how many struct positions to skip in the *structArray* to find each of its active child nodes. This was because inactive nodes were *pruned* and not stored, so this, *activeFlagBits,* was used to skip the correct number of positions in the *structArray*.

When searching for the *target* cell not all the bits used to represent its 3D co-ordinates have to be evaluated as this would take the traversal down to an individual cell, but once that *target* cell is found to reside within an active leaf node its *iActive* value can be deduced from referencing the compIndArray. In many instances during reservoir visualisation, only a cell's active status is required so inactive could be returned as soon as a header flag indicator bit indicates a zero for that branch and *active* as soon zero header flag is encountered. A header flag value of zero (all 'off' bits) indicates that this struct represents an active leaf node, its *pointer,* points to the leaf node's first *iActive* cell value in the *compIndArray* and using the node's edge length, the remaining *iActive* values can be deduced. Instead of counting the number of active '*on'* bits for the activeBitCount value, a simple lookup table could be generated which just passes in the *header flag* value and the activeBitCount (number of header's active child nodes) returned.

## 4.12 Data Structure Overview

Figure 4.4 on the next page illustrates how the *input* array, *structArray* and *compIndArray* interact with one-another and how it references the properties array. It illustrates how the grid is populated using the input array showing the resulting structArray with *header flags* and *pointers*. It also illustrates how active leaf node *pointers* are used to index the *compIndArray* which is used to index the *properties* array, which in a real system would be a large object of various cell characteristics which could be applied to the grid at runtime.

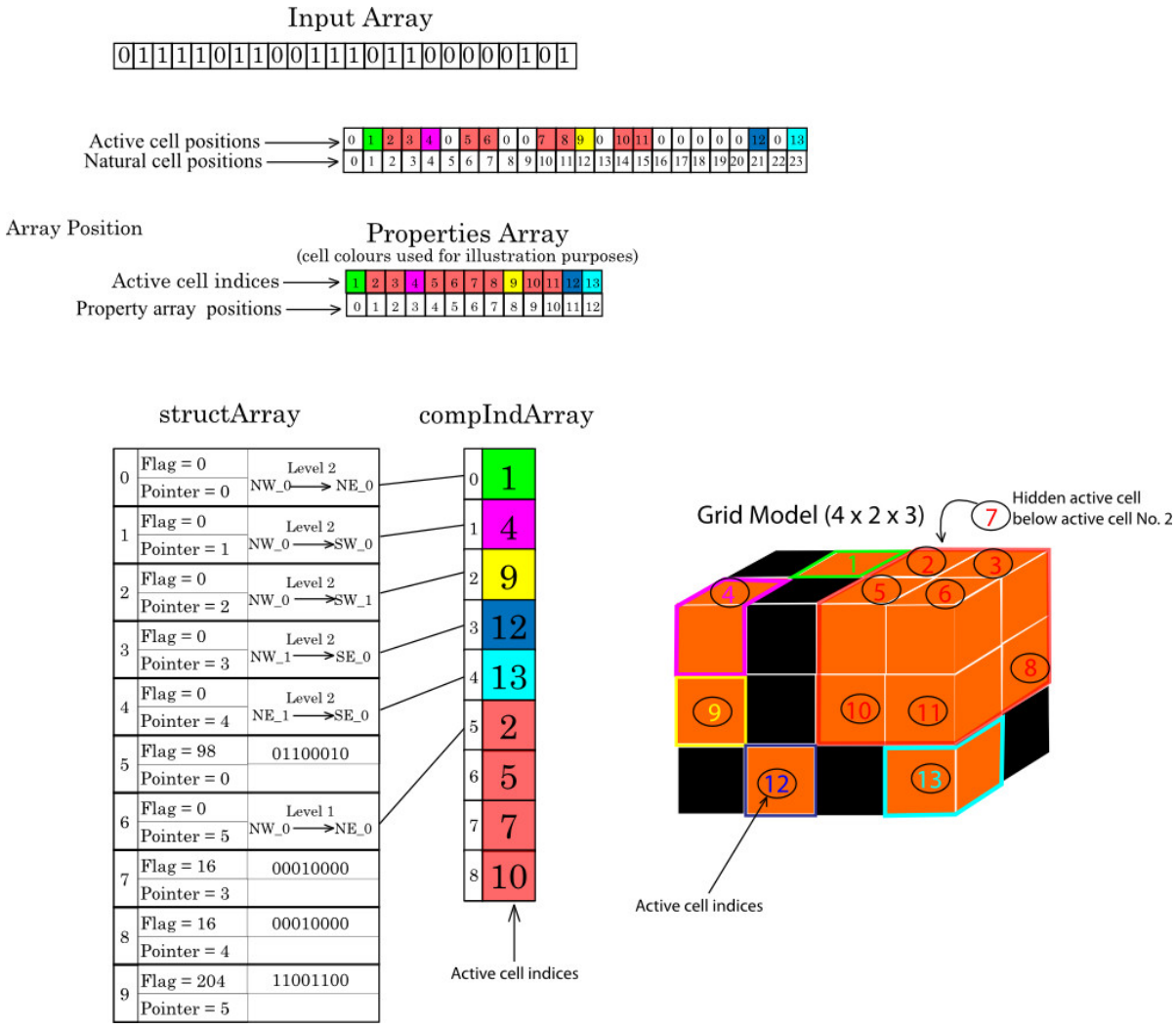



Figure 4.4: Illustration showing a 4 x 2 x 3 grid, its original input array, the structArray and compIndArray

## 4.13 3D Bitwise Searching Algorithm

In order to save memory, the variables passed into the recursive function can be referenced in C# using the 'ref' keyword so that new instances of the value types are not constantly being created and destroyed during the recursive traversal process. The pseudo code (Appendix section 10.2 )shows the algorithm for just returning the active status of a cell but, the cell's active cell index can be returned as detailed earlier in this thesis where its *iAcitive* value can be found by indexing the *compIndArray* using the leaf node's *pointer*. The time complexity of this algorithm is $N(K_1 + k_2(\log(n))$. $N$ is the number of cells in the grid, $n$ is the number of nodes in the the octree, $K_1$is constant overhead associated with the system such as when loading the data into memory and $k_2$is the constant overhead associated with performing each level of recursion in the traversal function.

## 4.14 Cell Searching

Looking at this pseudo code, the *dirFlag* starts at position 128, {10000000}, the NW_0 child node's direction bit. Sets of 3D co-ordinate bits are evaluated using masking (a bitwise '*AND*' method). If all the bits being evaluated from the searching co-ordinates equal zero then *dirFlag* the this would indicate an inactive leaf node, if for example the target cell's co-ordinates were {x = 0, y = 1, z = 1} then this would shift the *dirFlag* two bit positions to the right and *dirFlag* now equals {00100000} indicating that the SW_0 child node is to be traversed.

Table 4.2 below represents what the directions of the *structArray* representation would be traversed given the various values of *dirFlag* values based on the vertex ordering convention adopted.

| Octant navigation | NW_0 | NE_0 | SW_0 | SE_0 | NW_1 | NE_1 | SW_1 | SE_1 |
|---|---|---|---|---|---|---|---|---|
| X value | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Y value | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| Z value | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Binary Representation | 10000000 | 01000000 | 00100000 | 00010000 | 00001000 | 00000100 | 00000010 | 00000001 |
| Binary value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

Table 4.2: Bitwise masking of binary representation of input search co-ordinates shown individual direction values

Figure 4.5 illustrates how searching for a target cell with co-ordinates (7, 14, 3) in a $16 \cdot 16 \cdot 16\, cell\ grid$ would be performed if only the cell's active status was required and shows how traversing down to a single cell is not required as the cell resides in a larger leaf node at tree so that active could be returned at the fourth level in the search, tree level three. Each direction searched specifies the bit in the bit flag $dir\,flag$ as shown in the previous table.

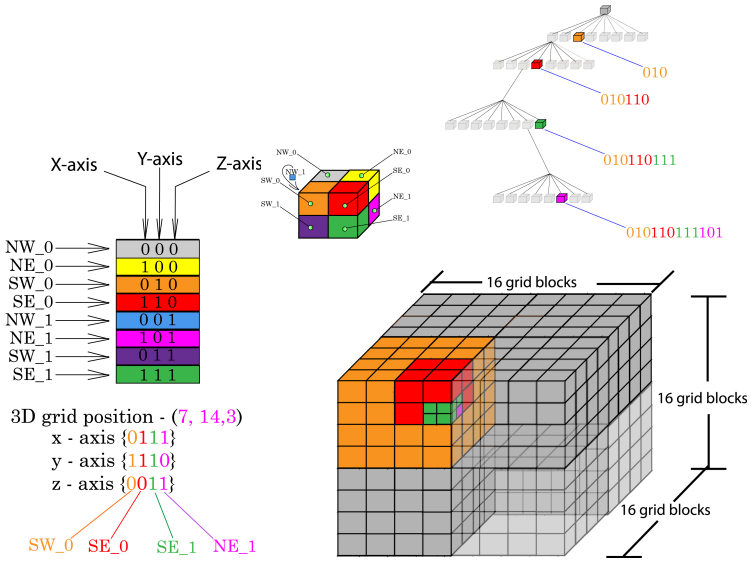



Figure 4.5: Bitwise techniques showing a 16 x 16 x 16 grid where the co-ordinates (7, 14, 3) are searched for

## 4.15 structArray Enumeration

Instead of using a normal looping function (such as a for loop) often programmers will use an enumerator in the form of a '*foreach*' loop for its simplicity and simplified code. It can be quicker than standard rasterscan searches due the elimination of boundary checks and stopping conditions.

An enumerator was designed where the calling function first visits all nodes in the *structArray* and returns each leaf node value in the form of a struct '*leafStructs*'. This struct held all the values necessary to recreate the 3D volume of cells and index property values:

- The natural cell value (*iCell*) – as a 4 byte integer.
- The *x*, *y* and *z* co-ordinates of the cell, each as 4 byte integers.
- The active cell (*iActive*) value of the cell, or -1 for an inactive node.

The enumerator traverses the *structArray* visiting each of the nodes in breadth-first-order and and passes back *leafStructs* containing all leaf node information required for reservoir runtime visualisations, and consisted of five integer values and one byte value as follows:

- Origin position (*x*, *y* and *z* co-ordinates) each represented as a 4 byte word.
- Node length – the number of cells it represents in each of its *x*, *y* and *z* axis directions, as a 4 byte word.

- *iActive* value – inactive cells as -1, active cells as the active cell position in the list of cells used to point to the *compIndArray*, as a 4 byte word.
- *Header flag* – represented as a single byte, the bits representing the active child node status of header nodes. When zero *header flag* values are encountered the enumerator *pops* the *leafStruct* of the stack,sending it to the second part of the enumerator.

The pseudo code used in the first part of the enumeration is given in Appendix section 10.4. Using an enumerator has the advantage that the stack at worst only has to store (*push* on to the stack) the maximum number of levels in the tree times the number of elements required for each recursive frame, as each value is removed (*popped*) of the stack, it is emptying so that the stack never gets so large that it the enumeration process becomes slow or runs out of memory.

The second part of the enumerator takes this *LeafStruct* and returns all individual cell values required to the calling function again as a struct (*cellStruct*). When only a cell's active status is required, only true and false is required, where false indicates an inactive cell (all cells within a leafStruct having an iActive value of -1).

When the user calls the *foreach* enumerator the returned *cellStructs* contain each cell's $x$, $y$, $z$ co-ordinates, natural cell index value (*iCell*) and the active cell index (active cells = iActive; inactive cells = 0).

Although when integrated with Sciencesoft's software the *cellStructs* holding individual cell values were returned to the calling method it would also be feasible to pass back the *leafStruct* instead. As the calling function knows a cell's position within a volume of neighbouring cells, each possessing a similar active status then the calling function may be able to optimise the routine by eliminating the need for certain inner cell calculations, such as with face culling evaluations.

## 4.16 Basic Recursive structArray Traversal Algorithm

In many cases the *structArray* was searched using a callback method where all elements with the structArray were searched in breath-first-order (as it was written), sometimes active/inactive cell information is all that was required, other times some or all cells active cell indices are required ($x$, $y$, $z$ co-ordinates *iCell*, *iActive*, node lengths). There were several variations of this algorithm used in this research, each designed to meet a particular reservoir visualisation demand, but they all derive from this basic structArray traversal algorithm, where each variation was adapted to suit a particular task and almost identical to it. The

pseudo code used to traverse through the *structArray* to perform such a particular task where a method was applied to all active cells (activeLeafNodeFunction) can be found in Appendix section 10.6.

The time complexity of this algorithm is $log(n)$ (where $n$ is the number of nodes in the *structArray*) as each individual cell search only has to traverse $log_8$ of the total number of nodes in the tree to reach the leaf node where the cell resides. The searching algorithm is recursive and each of these frames hold the cell's defining variables for example the node's staring co-ordinate positions, node edge length and a reference to the *structArray*. The maximum number of recursive frames in memory at one time is at most only ever equal to the number of levels in the tree so a grid with dimensions of 256 cells$^3$ would at worst only ever generate 9 recursive frames in memory at the one time. As each frame meets its stopping condition the frame is deleted freeing up its memory allocation. This algorithm forms the basis for many of the searching functions in this research and the results from experiments are given in the following chapter where the various searching techniques proposed in this thesis are evaluated using the 36 real oil reservoir grids as a test set, supplied by Sciencesoft.

# Chapter 5

# Memory And Performance Analysis Experiments

The following sub-sections in this chapter detail the methodologies used to test the performance and memory gains achieved using this novel approach of applying octree compression techniques to oil reservoir active cell information. Although its data structures could be used to compress other grid types, this research was targeted at compressing the active cell information of 3D oil reservoir grids and so these grid types were used for testing and analysis. In order to achieve meaningful and realistic results, the grids chosen for the experiments were a variety of actual real grids differing in cell numbers, aspect ratio and active cell percentage. The experiments were conducted in order to prove the hypotheses.

The test grids were evaluated in four ways:

- Compression time – the time taken to compress the grids into the octree data structures (*structArray* and *compIndArray*), the compressed octree structure.
- Memory – the memory of the original data structure (uncompressed *input* array) compared to the compressed octree structure (*structArray* and *compIndArray*.
- Entropy – the entropy of the grids were tested as detailed in section 2.5 on page 26 where a path was generated from scanning through each slice of the grids using a 2D Hilbert Curve algorithm forming a contiguous path through the 3D grid. Markov fourth order of conditional entropy was then applied to this path containing active cell information (ones and zeros).
- Performance – cell lookup times comparing the times generated from scanning through the original uncompressed input array and the new compressed *structArray*. These two experiments can be categorised in two flavours:
  - Real-life – these were initial experiments conducted at Sciencesoft's offices where the

octree structure was substituted for their current data structure where searching was performed in traditional array scanning for-loops. These experiments used a sample set of test grids made available to the author during this research. The experiments were conducted on a standalone version of their S3GRAF-3D software application with associated file types such as vertex files and property arrays. These grids were chosen as they represented the larger 3D oil reservoir grids with varying active cell percentages and dimensions and their associated file types were available and compatible with the standalone software package. The grids chosen were grids 24, 27, 30, 32 and 36 supplied by Sciencesoft detailed in 4.1.

– Controlled – these experiments were conducted using a prototype application which made the same input and output demands as Sciencesoft's software, but stripped out all of the hidden overheads of S3GRAF-3D. These experiments were designed to evaluate how efficiently the *structArray* could perform if Sciencesoft redesigned their software around octree storage as opposed to 3D arrays. The grids used for these experiments were the 36 oil reservoir grids supplied by Sciencesoft presented in section 4.1 of this thesis.

## 5.1 Test Grid Compression Times

The time to compress the grids into the octree structure were taken as it was suspected that there would be a close correlation between compression time and grid size. Figure 5.1 shows that this is indeed the case showing the time to compress each of the 36 test grids plotted against total cells. It can be seen that the time to compress the grids rises linearly in proportion to the number of cells contained in the grid and has a time complexity of $O(N)$ where $N$ equals the number of cells in the grid.

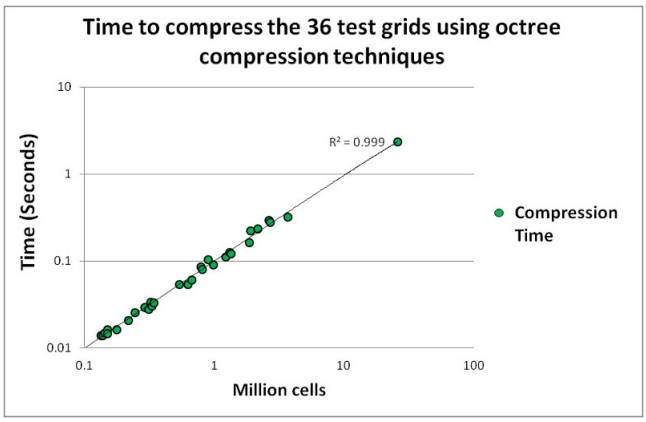



Figure 5.1: Plots showing the time to compress each of the test grids using octree compression techniques

## 5.2 Test Grid Memory Evaluations

Table 5.1 details the compression ratio achieved when compressing each of the 36 test grids using octree compression techniques.

| Grid Model | Input array (MB) | Octree (MB) | Compression Ratio |
|---|---|---|---|
| 1 | 0.059 | 0.019 | 3.01 |
| 2 | 0.22 | 0.09 | 2.58 |
| 3 | 0.28 | 0.05 | 5.54 |
| 4 | 0.30 | 0.07 | 4.13 |
| 5 | 0.32 | 0.04 | 7.51 |
| 6 | 0.34 | 0.12 | 2.89 |
| 7 | 0.51 | 0.21 | 2.46 |
| 8 | 0.53 | 0.15 | 3.45 |
| 9 | 0.55 | 0.21 | 2.67 |
| 10 | 0.57 | 0.09 | 6.69 |
| 11 | 0.58 | 0.16 | 3.68 |
| 12 | 0.68 | 0.14 | 4.82 |
| 13 | 0.83 | 0.24 | 3.40 |
| 14 | 0.94 | 0.25 | 3.68 |
| 15 | 1.12 | 1.00 | 1.12 |
| 16 | 1.19 | 0.12 | 9.53 |
| 17 | 1.24 | 0.37 | 3.39 |
| 18 | 1.26 | 0.21 | 6.13 |
| 19 | 1.32 | 0.31 | 4.27 |
| 20 | 2.07 | 0.60 | 3.46 |
| 21 | 2.39 | 0.09 | 27.32 |
| 22 | 2.56 | 0.40 | 6.47 |
| 23 | 3.03 | 1.65 | 1.84 |
| 24 | 3.09 | 0.89 | 3.47 |
| 25 | 3.75 | 0.13 | 28.84 |
| 26 | 4.69 | 0.38 | 12.47 |
| 27 | 5.00 | 0.53 | 9.48 |
| 28 | 5.11 | 0.42 | 12.31 |
| 29 | 7.09 | 0.38 | 18.43 |
| 30 | 7.30 | 4.77 | 1.53 |
| 31 | 8.29 | 3.91 | 2.12 |
| 32 | 10.13 | 5.21 | 1.94 |
| 33 | 10.30 | 3.22 | 3.20 |
| 34 | 14.04 | 0.21 | 65.41 |
| 35 | 3.43 | 0.85 | 4.05 |
| 36 | 98.62 | 6.34 | 15.55 |

Table 5.1: Test grid dimensions, cell count, uncompressed input array size, compressed octree size and compression ratios achieved (rounded to 2 decimal places) listed in order of number of cells per grid

## 5.3 Test Grid Entropy

The stochastic nature of active cells with regard to the placement of active cells within each test grid was evaluated in order to get a measure of active cell clustering, as it was suspected that there would be a link between grid entropy and achievable compression ratios. This level of clustering was generated from performing fourth order of conditional entropy calculations (detailed in section 2.5) applied to a linear path of cells generated from traversing through the grid, passing through each cell once and adding the active status of each cell visited to the path. The order of active and inactive cells which made up each path was formed by traversing through each slice in the grid in a 2D Hilbert Curve fashion, where each path from each slice was appended to the end of the path generated from the slice above. With each descent down through the slices of the grid, the path started from the same $x$ and $y$ co-ordinates so as to keep neighbouring cells in the grid close to one-another in the path.

If the active cell placement was of a purely random nature then grids displaying around the 50% active range would generate trees where most leaf nodes contained a single cell, resulting in a very inefficient octree structure. As oil reservoir grids do not have their active cells placed in a random manner but instead reside in clusters with a *'peppered'* appearance, it is not unreasonable to suggest that the further a grid's active status is from around 50% the lower its entropy may become and the greater the level of compression achieved. This is because the octree generated would be shallower containing fewer leaf nodes each generally representing larger volumes of cells.

## 5.4 Initial Real-life Experiments

In order to test the hypotheses made in this thesis the new octree structure (*structArray*) was substituted for Sciencesoft's N2A array using their 3D oil reservoir visualisation software, S3GRAF-3D. This state-of-the-art software package takes the input array of ones and zeros (ACTNUM) and generates an integer array (Natural to active array, N2A) so that each active cell (indicated by a one in the ACTNUM input file) is represented by a positive integer equal to the position of the active cell in a list of active cells as an active cell index (iActive value) and the zero, inactive cells are stored as a minus one. This array is used for searching all cells within a grid in raster formation and for indexing individual cells and for the purposes of this thesis, this direct array lookup is referred to as *direct access*.

In reservoir visualisation, scanning through grid generally involves searching using either a single-for-loop or a treble-for-loop to rasterscan through the N2A array. When a single-for-

loop is used the natural cell position value of the cell is known as this is the iteration counter in the loop, but when a treble-for-loop is used using the grid's *x*, *y* and *z* dimensions, natural cell indices have to be calculated using the formula given in Figure 3.2 and generally required to index the N2A array, such as for nearest neighbour evaluations. Most of the methods also require a cell's *iActive* value (the value equal to the cell's active cell position in a list of active cells or a minus one if the cell is inactive). This value is used for indexing cell properties such as pressure and permeability values. Due to the various requirements of the system generally a cell's *x*, *y* and *z* co-ordinates are also required and so have to be calculated when a linear single-for-loop is used for visiting all cells. When individual cells are searched the N2A lookup times are fast due to direct access array indexing times. When only active cell information has been loaded, as is sometimes the case due to large grids and computer memory restrictions, the natural cell position of a cell within the list of loaded cells is required, such as performing a single-for-loop search through all loaded cells, and the *iActive* value is used to index cells as the N2A only contains active cells.

When the octree structure was substituted for the N2A, lookup times were compared to those using the N2A. The *structArray* was traversed using the foreach enumerator and when the an active cell's iActive value was required, this was sourced by indexing the *compIndArray,* which held active cell, *iActive* values within each leaf node in a compressed state, as illustrated in Figure 4.3.

### 5.4.1 Real-life Performance Experiments

S3GRAF-3D uses several methods which traverse through grids evaluating whether cells are active or inactive, performing calculations based on their active status. Similarly, individual cell searches are made such as during nearest neighbour evaluations, which performs various evaluations on a *target* cell's logical space adjoining cell faces, such as determining if a *target* cell's face is hidden by a neighbouring cell face. The octree was substituted for the N2A rasterscan and linearscan searches within the S3GRAF-3D software method calls, using the foreach enumerator which referenced each struct in the *structArray* as detailed in section 4.15. Individual cell searches were performed using an iterator adapted for the *structArray* which traversed the octree from the root node to the leaf node containing the *target* cell searched for. If the *target* cell resides within an inactive leaf node then minus one was returned otherwise the active cell index (*iActive* value) was calculated by indexing the *compIndArray*, as illustrated in Figure 4.3.

Following are a list of the main methods within S3GRAF-3D where the *structArray's* enumeration was substituted for Sciencesoft's N2A direct access lookup methods briefly detailing

their purpose. Some of the methods are performed when the model is first loaded and others are user defined such as clipping, planer and isosurfacing methods.

- VertexMemory- this function fills an array with information on all visible faces such as the grid model loaded, natural cell number, face number and positions within a list of visible faces. This array is later used to fetch the vertices required to draw each of the visible faces.
- Clipping – where the grid is clipped in one or all of its axis so only a portion of it remains.
  - ClipVolumeFlags – Builds and array of flags to indicate which cells are in the clipped volume.
  - VisibleFaceFlags – calculates the face visibility of cell faces performed as the lists change, e.g. after clipping exposing previously hidden faces.
- Cut Planes – Triangle calculations which checks if the cut plane intersects a cell and on which side. The cut plane is a 2D plane which is used to dissect the grid in any direction. This generates visualisations showing the grid model before the slice plane, after the slice plane or only the slice plane. It then recalculates all of the polygons which have been intersected by the cutting plane.
- Isosurfacing – average property values taken from neighbouring cells are used to generate a triangular 3D contouring visualisation effect.
  - Isosurfaces – each cell within the grid is given a property value and contours are drawn by applying average property values to neighbouring cells. These property values are then drawn for each of the vertices when more than one cell shares the same vertex position their average cell property value is applied.
  - IsosurfaceVertices – generates lists of vertices making up triangular contours.

The octree substituted single/treble-for-loops using the foreach enumerator where individual cell information was required, such as their, iCell, iActive, ($x$, $y$, $z$) co-ordinates and active status values. Booleans are extensively used to determine whether actives or inactive cells are to be drawn as well as flagging whether a compressed vertex table had been loaded (active cell vertices only). A cell's natural cell index is first calculated from its $i^{\text{th}}$, linear looping position in a single-for-loop or its $x$, $y$ and $z$ co-ordinates in a treble-for-loop. A cell's iActive value is used when only the active cell vertices have been loaded (activesOnly). Most of the methods required all variables and Table 5.2 details a list of these methods where structArray's *foreach* enumerator was substituted for the N2A and the cell values required.

| Method | Required values | Substituted loop type |
|---|---|---|
| ClipVolumeFlags | x, y, z, iCell, iActive | 3D |
| ClipVolume | x, y, z | 3D |
| VisibleFaceFlags | x, y, z, iCell, iActive | 3D |
| VertexMemory | x, y, z, iCell, iActive | 1D |
| CutPlane | x, y, z, iCell, iActive | 3D |
| Isosurfaces | x, y, z, iCell, iActive | 3D |

Table 5.2: List of required cell variables during foreach loops

Each of the five sampled test grids were evaluated using Sciencesoft's N2A and the new octree methods and their performance times were compared and contrasted with one-another. The times generated were based on the average of five runs except for the isosurfacing methods as these took a substantially longer time to complete where the difference in time for each run differed an insignificant amount. The specification of the computer used for the experiments at Sciencesoft's premises is detailed in Table 5.3.

| | |
|---|---|
| Operating System | Windows Vista Business 2007 |
| System Type | 64-bit |
| RAM | 8.00 GB |
| Processor | Intel(R) Core(TM)2 Duo CPU E8400 |
| Processor Speed | 3.00 GHz |
| Cache | L2 6 MB |
| GPU | ATI Radeon HD 4650 |

Table 5.3: Computer specifications used for the initial experiments

When data is compressed it takes an additional time to search within a compressed structure compared to directly fetching values from an uncompressed structure such as an array (direct access). Sciencesoft's searching methods are build around direct access methods using arrays as these data types are well suited to storing Cartesian grid information. It was envisaged that there would be a performance hit apportioned to *structArray* lookup times due to time associated with accessing data from within this highly compressed structure. It was agreed that a performance hit of around 20% would be acceptable as it this would be offset by the impressive memory savings achieved using this novel octree structure. Following is a discussion of the performance times yielded from these experiments where the *structArray* was substituted for Sciencesoft's scanning methods and expressed as runtime ratios where values greater than one indicate faster average cell lookup times than direct access.

### 5.4.2 Real-life Experiment Results

The results from these initial experiments showed that the octree structure could deliver fast average cell lookup times and in some occasions surpass Sciencesoft's direct access methods such as during the ClipVolumeFlags routine. In this routine each cell is evaluated to see whether it resides in the clipped volume of the grid to be rendered before its faces are evaluated and the visible faces list updated. Table 5.4 details the runtime ratios of the ClipVolumeFlags during the clipping operations where it can be seen that the *structArray,* at worst, matched direct access times, performing especially well with the largest grid. This table also shows that direct access often outperformed the *structArray* such as during Isosurfacing routines.

| *Grid Model* | *ClipVolumeFlags* | *VertexMemory* | *ClipVolume* | *Cut Plane* | *Total Isosurfacing* |
|---|---|---|---|---|---|
| *24* | *0.92* | *0.99* | *0.57* | *0.87* | *0.93* |
| *27* | *1.02* | *1.23* | *0.60* | *0.90* | *0.92* |
| *30* | *1.10* | *0.73* | *0.62* | *0.90* | *0.86* |
| *32* | *1.03* | *0.87* | *0.59* | *0.87* | *0.90* |
| *36* | *0.33* | *2.03* | *0.61* | *0.90* | *Not Loaded* |

Table 5.4: Initial experiment results given as runtime ratios (values greater than one indicate faster individual cell lookup times than direct access method)

The results from the VertexMemory method show that the octree structure matched and outperformed direct access methods with three of the grids, the greatest saving with the largest grid, yielding a runtime ratio of almost two, a saving around half a second per grid.

The ClipVolume method yielded poor performance results as only the ($x$, $y$, $z$) co-ordinates were required in the method, the *structArray*'s enumerator had to generate this along with the *iCell* and *iActive* values, but in the case of a treble-for-loop, using the N2A no calculations were required as these values were the x, y and z loop boundary values. Also this method has little work to perform when a grid is first loaded and not clipped, with the N2A method every cell value within its loop boundaries are known to be in the clip volume but the *structArray* has to check each cell is within these boundaries first before passing the variables.

Results from the cut plane routine show that the *structArray* yielded a small performance hit, within a thirteen percent tolerance and the large grid, only around a ten percent.

During isosurfacing triangle calculations a *target* cell's nearest neighbouring cells are searched to see if they share the same active status as the *target* cell and if so, their vertices are also evaluated to establish if they match (share the same 3D co-ordinates). Cells matching in active status only, indicates they are joined in logical space, only when faces share the same vertex positions can it be established that they are joined in geological space and their average

property values can be input to the isosurfacing calculations. With direct access methods this can be performed extremely efficiently as these neighbouring cells are accessed directly by merely referencing an array but with a octree structure there was an additional overhead as each neighbouring cell instigated a new search through the *structArray* from the root node; grids of low entropy contain many small leaf nodes generating longer searches due to trees having maximum levels.

The performance hit using this method is especially prevalent, as each *target* cell can have up to twenty six neighbouring cells. Despite these overheads, the *structArray* performed well against the direct access methods, shown in the previous table with a performance hit well within the 20% acceptable limit. Only four of the grids were used in the isosurface performance tests as the large grid (grid 32) could not be loaded with the inactive cell vertex positions required for isosurfacing calculations, due to the memory constraints of the programming language as the size of the arrays required to store all the grid's vertex information surpassed that permitted by Visual Studio 10 using C# at the time this research was conducted.

### 5.4.3 Initial Experiment Conclusions

Looking at the initial experiment results it can be seen that adopting octree compression techniques generated a data structure which impacts on performance although within the previously defined acceptable tolerance limits stipulated by Sciencesoft. The octree structure can sometimes outperform state-of-the-art direct access methods but was frequently outperformed by direct access methods when embedded into Sciencesoft's S3GRAF-3D software, not because of a poorly designed data structure, but the way in which it was implemented. Sciencesoft store their data in Cartesian grid array formats, well suited to traditional single and treble-for-loop scanning methods. It was envisaged that if they implemented octree compression techniques into their software, they would have to adapt their searching methods to suit the data structure being searched, breadth-first-tree-order traversal methods and it was perceived that this data structure would outperform the state-of-the-art direct access lookup times which they presently achieve. In order to test this theory a further series of performance experiments were conducted which adopted this philosophy and its methodologies and results and analysis are detailed in the following sub-section of this thesis.

## 5.5 Controlled Octree Experiments

The controlled octree experiments comprised of performing lookup time experiments in an identical fashion to the the real-life experiments but without using Sciencesoft's S3GRAF-3D software. This was done for two reasons:

- To eliminate any overheads brought about by method calls and classes within Sciencesoft's S3GRAF-3D software, especially those hidden from the researcher due to the architecture of the software and who had no control over their design or implementation.
- To establish whether re-designing the searching methods around the octree structure would yield better performance results, proving that the octree compression techniques developed in this thesis could outperform direct access lookup times presently achieved by Sciencesoft, proving the second hypothesis made in this thesis.
  - Cell lookup times will prove to be quicker using recursive traversal methods with the octree representation than direct access methods achievable at Sciencesoft today.

### 5.5.1 Controlled Octree Performance Experiments

The experiments conducted using the prototype application used the same inputs, generated the same outputs and implemented similar method calls, as used within Sciencesoft's S3GRAF-3D software, where the octree structure (structArray) was substituted for the Natural-to-active array (N2A).

In order to test the new octree representation and test the second hypotheses, experiments were designed which met the demands of 3D oil reservoir visualisation by mimicking the demands of S3GRAF-3D where each cell returned the same variables required in their method calls ($iCell, iActive, x, y$ and $z$ values) and were sub-divided into three categories:

1. Rasterscan and Foreach – (iterator and enumerator).
2. Callback – passed a method to the recursive loop.
3. Periphery – periphery cells only using a callback design.

In order to stop the compiler optimising searches, and to mimic actual work performed by a system during runtime, such as nearest neighbour evaluations, a unit of work (*workload*) was performed on each active cell, emulating real-life reservoir visualisation scenarios, when inactive cells are ignored and calculations only performed on active cells.

- Scanning – these experiments were conducted using traditional treble-for-loops using a simple iterator (octree[*i*, *j*, *k*]). Foreach searches used in the experiments were straight-

forward when using direct access methods, foreach(int *i* in array (N2A)). When using the octree the foreach enumerator returned a struct which held all cell attributes as detailed in section 4.15 of this thesis. The iterator experiments were expected to yield poor results as this style of searching orientated around Cartesian style searching and not in any sort of tree-order. The enumerator (foreach) searches were expected to yield improved results due to cells being visited in tree-order.

- Callback- these experiments were conducted using a passed in delegate function – here a function was passed into a recursive breadth-first searching function which traversed the tree in tree-order applying the *workload* function to all cells within active leaf nodes and only returning to the calling method once all cells have been searched and the methods complete. This method of searching was expected to yield better results as the searching method was designed around the data structure (octree).
- Periphery – these experiments were conducted using the same callback delegate function as before, but calculations were only performed on the periphery cells on leaf nodes. These experiments were conducted because the vast majority of the 3D oil reservoir visualisation calculations establish which cell faces are to be rendered based on neighbouring cell evaluations. Octree leaf nodes naturally hide their inner nodes, meaning, a leaf node's inner cells could be excluded from nearest neighbour calculations. These experiments were expected to yield the best results especially those possessing larger leaf nodes.

The *workload* function used in the experiments passed cell attributes and used trigonometry calculations as this level of calculation was deemed suitably computably demanding by Sciencesoft and was as follows:

- Define a global variable, *globalvariable* = 0
- Define a method *myFunctionMethod* ( *x*, *y*, *z* axis co-ordinates, *iCell* , *iActive*)
- *globalvariable* = $x \cdot y \cdot z \cdot iCell \cdot iActive$
- for each of these variables 'k'
  - *globalvariable* = *globalvariable* + tan(globalvariable, k)
- *globalvariable* = *globalvariable* + tan(globalvariable*, globalvariable*)
- return *globalvariable*.

#### 5.5.1.1 Controlled Octree Experiment Applied Workload Scenarios

The main output cell attributes required from each cell during all the experiments were:

- The natural cell index (*iCell* value)– as a 4 byte integer.
- The *x*, *y* and *z* co-ordinates of the cell each as 4 byte integers.
- The active cell index of the cell (*iActive* value).

The experiments developed to test the performance of the system was designed to calculate all cell attributes and perform the *workload* on each cell emulating a unit of work which would be performed by software such as nearest neighbour evaluations, when cells have their six faces evaluated with their adjoining neighbouring cell faces (in logical space) checking whether it is hidden from view by these neighbouring cells and vertex position evaluations (as vertex information was not required in these experiments). In oil reservoir visualisation, sometimes the cells sent to the GPU for rendering are inactive, other times active, but on rare occasions, both so when active cells are being rendered, inactive cells do not require evaluating. Sometimes only active cells can be rendered as the grid is too large to load vertex tables containing the cell information required to render all cells and the controlled octree experiments conducted were designed to match these scenarios, where the *workload* was only applied to specific cell types. In each of the experiments every cell searched had its *iCell* value checked that it returned the correct *iActive* value using the N2A array to verify correct outputs,as a unit test, using C# within Microsoft's Visual Studio 10 environment. The following four sets of experiment scenarios were applied

1. Iterator scan – apply the *workload* to all cells, using a treble-for-loop comparing direct access and octree lookup times.
2. Foreach enumerator scan – apply the *workload* to active cells only using direct access (treble-for-loop) and the octree (foreach enumerator) when all cells were loaded.
3. Delegate (passing a delegate function as a callback method) – apply the *workload* to active cells only comparing direct access using a (treble-for-loop).
4. Periphery(passing a delegate function as callback method) – apply the *workload* only on the periphery cells of active leaf nodes when all cells were loaded, comparing direct access using a (treble-for-loop).

### 5.5.2 Results

Following are the results from the lab performance experiments which compare and contrast the average lookup times per cell using octree and direct access methods. Table 5.5) shows the runtime ratios where direct access times (uncompressed, existing data structure) was divided by the octree's access time (compressed data structure) resulting in performance time comparisons expressed as runtime ratios. In each of the results, the higher the value the better the octree structure performed, so that a value of '1' matched direct access and a value greater than one, for example, '1.1' would signify a 10% improvement in performance. All of the results were taken from averages of fifty iterations (each grid was traversed fifty times and the average cell lookup time taken for the comparisons).

#### 5.5.2.1 Iterator Results

The first column in Table 5.5 on page 84 (Experiment 1) shows the runtime ratios generated in the iterator experiments using a treble-for-loop, when the uncompressed direct access method was compared to traversing the octree's *structArray* and the *workload* was applied to all active cells.

Looking at these results it is clear that there is an overhead in searching through the octree structure in a scanning fashion more suited to traditional Cartesian array searching methods using direct access. The octree performed poorly in comparison to direct access yielding on average runtime ratios of around 6 times slower. Grids with high active cell percentages, such as grid 35 which was 100% active have shallower trees than some smaller grids and so each search performed had less levels to traverse, yielding faster lookup times.

Traversing the octree structure and returning a cell's active status (IsActive value) using an iterator method where each individual cell searched starts from the root node has a time complexity of $O(N + \log_n)$. $N$ is the number of cells in the grid, $n$ is the number of nodes in the the octree. In the worst case where every node represented a single cell ($N = n$) then the time complexity would be $O(N)$. Figure 5.2 shows the average cell lookup times generated from traversing each of the 36 test grids in a random manner. The average time for each grid was generated from ten complete searches through each of the grids.

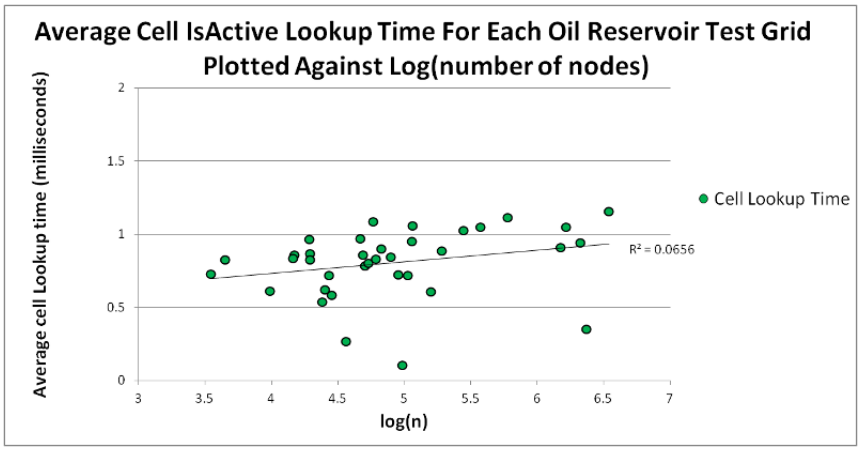



Figure 5.2: Average (IsActive) cell lookup time per grid plotted against the log of the number of nodes in their tree.

The time to return a cell's active cell index (iActive value) is almost identical and only has a slight overhead compared to just requiring its active status as a simple calculation is performed to deduce the cell's active cell index.

Results from the previous plot illustrates that there is very little difference in lookup time

irrespective of grid size, suggesting that extremely large grids will prove to be just as fast. Larger nodes, such as those generated from grids with higher levels of activity are quickest as these nodes reside higher up the tree (indicated by the lower plots in the chart) and locating a cell's active cell index within a leaf node only requires a simple arithmetical calculation and a direct lookup to the compIndArray.

The next experiment looked at scanning the grids using an enumerator as used in the real-life experiments.

The second column in Table 5.5 on page 84 (Experiment 2) shows the runtime ratios generated when the *workload* was applied to all active cells and compared to the direct access times, values greater than '1' show faster average lookup times per cell than direct access. The foreach enumerator performed better than using cell searches based on direct access methods (treble-for-loops). This is because the octree was traversed in a manner more fitting the octree searched in the order nodes were written to the tree. This meant that with each leaf node encountered (represented as a *leafStruct* within the enumerator) each *cellstruct* containing all the *leafStruct*'s cell's attributes were passed back before another leaf node search was performed so that searches no longer started from the root node of the tree but instead from the last node in the recursive function. These results showed an increased level of performance yielding an average cell lookup time of around three times that of direct access and with some grids returning average cell lookup times within 25% of direct access. Again it can be seen that grids such as Grid 35 (being 100% active) yield faster average cell lookup times than smaller grids.

Figure 5.3 shows the average cell lookup times per grid generated from the foreach enumerator method plotted against the log of the number of nodes in the grids (depth of the tree).

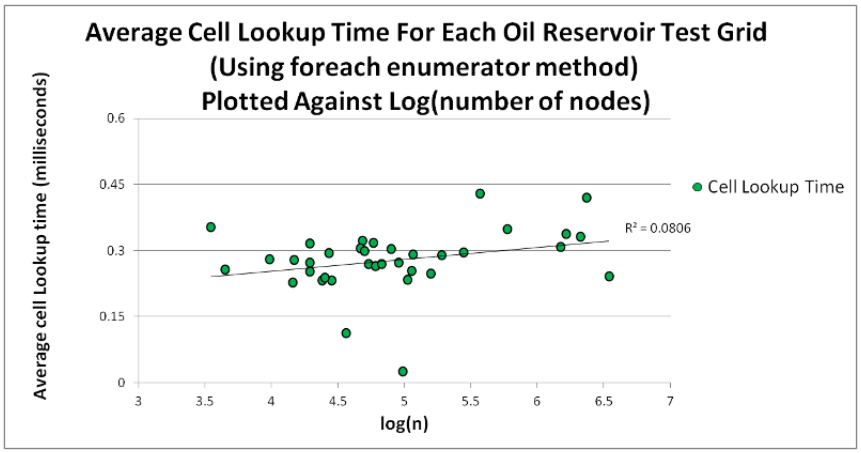



Figure 5.3: Average cell lookup time per grid using the foreach enumerator method plotted against the log of the number of nodes in their tree (depth of the tree).

#### 5.5.2.2 Callback Experiment Results

The results from the previous two experiments were designed to test the performance of searching a tree structure using traditional direct access rasterscan searching techniques typically used at Sciencesoft and tree-order techniques using the octree structure. Looking at the results in Table 5.5 on page 84 prove, that it is far more advantageous in terms of lookup performance times to search the octree in tree-order. The octree was flattened and written as an array of structs to the *structArray* in breadth-first-order in a similar fashion to the octree's foreach enumerator. This means that when a header node was written to the *structArray* its active child leaf nodes were immediately, sequentially written to the array after it in a unified order: {NW_0, NE_0, SW_0, SE_0, NW_1, NE_1, SW_1, SE_1}.

In accordance with the previous experiments, the *workload* was applied to cells based on their active status, but the *workload* passed as a delegate function into the recursive octree searching method in a callback manner. The reason a delegate function was used was to gauge the octree's optimum performance levels as it was perceived that an improvement in performance would be noticeable. This was because it was envisaged that the octree would perform quicker in actual 3D reservoir visualisation software if all cell evaluations and calculations were handled within the *structArray*'s recursive searching functions instead of passing back individual cells and their various attribute values to a calling function. For this reason the workload was passed into the traversal method as a delegate function and applied to leaf node's inner cells as opposed to passing parameters out of the leaf node as *cellStructs* to the calling function, eliminating any of the octree's foreach enumerator overheads. In order to eliminate any discrepancy in experiment equality the direct access method was modified so as to accept the same delegate function. In these experiments the workload was applied to all active cells such as in a typical oil reservoir visualisation software function.

As with the previous experiments all cell attribute values were calculated and their *iActive* values checked against the original input array, emulating a C# unit test but on all cells. Following are the results of these experiments and in an identical fashion to the previous experiments, values greater than one indicate faster average cell lookup times than direct access and were calculated from an average of fifty complete grid traversals.

The third column in Table 5.5 on page 84 (Experiment 3) shows the performance runtime ratios comparing direct access times with the octree data structure when passing a delegate *workload* function as a callback function where the *workload* was applied to all active cells. The results show that using callback methods, the octree data structure now yields an average of 91% of direct access and around a third matched or surpassed direct access methods. Again it can be seen that grids with high levels of active cells yield faster average cell lookup times

than many smaller grids due to their shallower tree depth. This is also true for grids with low active cell percentages where the resulting octree generated has very few nodes in comparison to similar sized grids with higher levels of activity, quicker average cell lookup times are returned due to these shorter structArray traversals, such as with grid 21 which traversed the octree over twice as fast as direct access methods.

Figure 5.4 shows the average cell lookup times using the callback method per grid plotted against the log of the number of nodes in the grids (depth of tree).

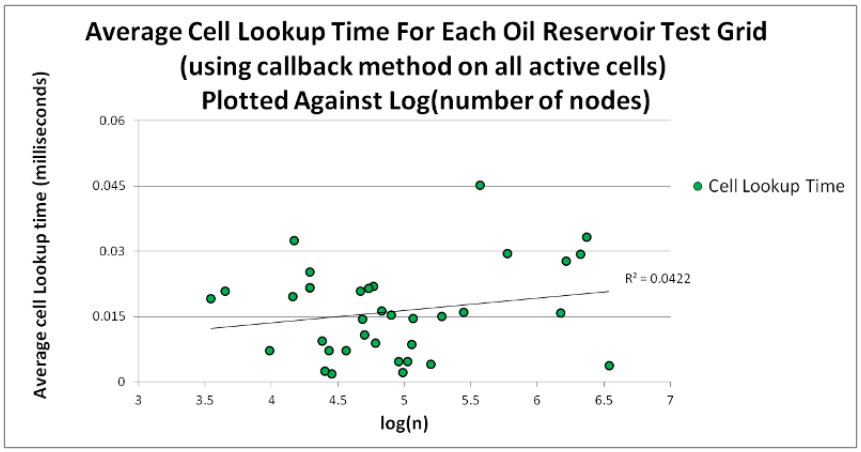



Figure 5.4: Average cell lookup time per grid using the callback method plotted against the log of the number of nodes in their tree.

As leaf nodes are constructed by storing clusters of cells possessing similar active status their inner cells are therefore similar to those lying on their peripheries where these periphery cells would hide inner cells obscuring them from view. With this in mind, inner cells could be omitted from many reservoir visualisation calculations, such as, face culling evaluations as cells on a leaf node's periphery obscure inner cells. Applying this node knowledge, and enhanced level of performance could be appended to the octree's already impressive performance gains. To test this theory the following experiment, equal to Experiments 3 was conducted but this time the delegate *workload* function was only applied to leaf node periphery cells.

The fourth column in Table 5.5 on page 84 (Experiment 4) shows the runtime ratio of comparing the active cell lookup times using direct access and the delegate *workload* function applied to all active cells and the octree, where the delegate *workload* function was only applied to active leaf node periphery cells. These results show an enhanced level of performance gain compared to Experiment 3 with the octree periphery experiments yielding on average almost 40% quicker average cell lookup times, faster than direct access with over 86% of the grids. Figure 5.5 shows the average cell lookup time per grid using the callback method where the workload was only applied to periphery node cells plotted against the depth of tree.

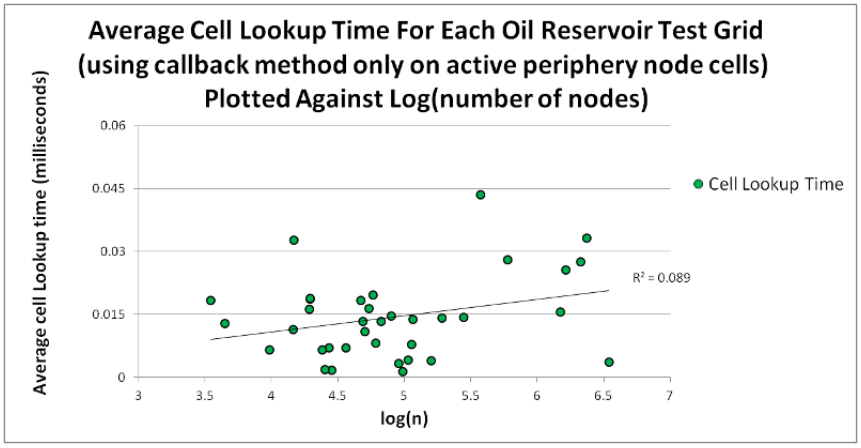



Figure 5.5: Average cell lookup time per grid using the callback method where the workload was only applied to periphery node cells plotted against the depth of tree.

| Runtime Ratios of 4 Experiment flavours (Average cell lookup using direct access / Average cell lookup time using the octree) | | | | |
|---|---|---|---|---|
| Grid Model | Experiment 1 IsActive | Experiment 2 Foreach | Experiment 3 Callback | Experiment 4 Callback using Periphery |
| 1 | 0.28 | 0.43 | 1.18 | 1.44 |
| 2 | 0.22 | 0.4 | 0.96 | 1.15 |
| 3 | 0.34 | 0.73 | 0.94 | 1.95 |
| 4 | 0.16 | 0.29 | 0.91 | 0.94 |
| 5 | 0.09 | 0.14 | 1.01 | 1.23 |
| 6 | 0.34 | 0.68 | 0.97 | 1.43 |
| 7 | 0.17 | 0.31 | 0.97 | 1.11 |
| 8 | 0.1 | 0.17 | 1 | 1.05 |
| 9 | 0.21 | 0.39 | 0.99 | 1.05 |
| 10 | 0.06 | 0.09 | 1.16 | 1.19 |
| 11 | 0.3 | 0.64 | 0.99 | 1.4 |
| 12 | 0.06 | 0.09 | 1.12 | 1.1 |
| 13 | 0.11 | 0.18 | 0.99 | 1.05 |
| 14 | 0.3 | 0.58 | 0.95 | 1.5 |
| 15 | 0.24 | 0.41 | 0.89 | 0.95 |
| 16 | 0.21 | 0.32 | 1.08 | 2.25 |
| 17 | 0.11 | 0.19 | 1 | 1.05 |
| 18 | 0.09 | 0.16 | 1.1 | 1.33 |
| 19 | 0.21 | 0.4 | 1 | 1.37 |
| 20 | 0.13 | 0.24 | 1 | 1.08 |
| 21 | 0.03 | 0.01 | 2.34 | 2.38 |
| 22 | 0.09 | 0.17 | 1.06 | 1.21 |
| 23 | 0.19 | 0.37 | 0.84 | 0.97 |
| 24 | 0.14 | 0.27 | 0.97 | 1.08 |
| 25 | 0.06 | 0.1 | 1.35 | 1.88 |
| 26 | 0.07 | 0.12 | 1.23 | 1.53 |
| 27 | 0.36 | 0.76 | 0.93 | 2.23 |
| 28 | 0.04 | 0.06 | 1.54 | 1.58 |
| 29 | 0.09 | 0.15 | 1.13 | 2.11 |
| 30 | 0.42 | 0.24 | 0.9 | 0.93 |
| 31 | 0.18 | 0.35 | 0.92 | 1.13 |
| 32 | 0.2 | 0.39 | 0.93 | 1.07 |
| 33 | 0.08 | 0.14 | 0.96 | 0.99 |
| 34 | 0.05 | 0.07 | 1.72 | 2.38 |
| 35 | 0.3 | 0.74 | 0.96 | 1.45 |
| 36 | 0.02 | 0.05 | 1.65 | 1.76 |
| Averages | 0.17 | 0.3 | 1.1 | 1.4 |

Table 5.5: Runtime ratios yielded from dividing average cell lookup times per grid using direct access by the average cell lookup times using the octree structure. Four flavours – Iterator, enumerator, callback and callback using periphery node optimisation. (values rounded to 2 decimal places) listed in order of number of cells per grid where the average compression ratio of each flavour is given. Values greater than one show quicker average cell lookup times than direct access.

## 5.6 Complexity Analysis

The R values seen when plotting average cell lookup times against log of the number of nodes in the tree (Figures (5.2, 5.3, 5.4 and 5.5)) are too small to either prove nor disprove that the lookup is $O(N \log_n)$. In the worst case where every node represented a single cell this would be $O(N)$ but oil reservoir grids do not have their active cells distributed in such a fashion. The problem with using the real grids for evaluating average cell lookup time plotted against $O(\log_n)$ of the tree size is that the tree size does not reflect the grid size as large grids can have small trees; a grid with few, but very scattered small active cells would perform many dummy workload functions but could have large numbers of nodes. To get a true representation of the time complexity the grid must be of a controlled type, taking away any boundary inactive cell calculations and controlling the proportionality between grid and structArray size. This was performed by generating 8 grids, 256 cells$^3$and scanning each of the grids, returning cell IActive values. The first Grid (A) is the top level and 100% active and only contains 1 node, the remaining grids (B – I) are 50% active where every level is split with identical sized active and inactive nodes, balanced tree. The bottom level only contains nodes representing single cells, as illustrated in Table 5.6. The times given are the average cell lookup times per cell and are an average of 50 complete grid scans.

| Grid size 256 cell$^3$ | active (%) | number of nodes (active leaf nodes) | node edge length (number of cells) | average cell lookup time (milliseconds) | log(n) Tree depth |
|---|---|---|---|---|---|
| A | 100 | 1 (1 active) | 256 | 0.62 | 0 |
| B | 50 | 8 (4 active) | 128 | 0.76 | 1 |
| C | 50 | 64 (32 active) | 64 | 0.84 | 2 |
| D | 50 | 512 (256 active) | 32 | 0.95 | 3 |
| E | 50 | 4096 (2048 active) | 16 | 1.06 | 4 |
| F | 50 | 32768 (16384 active) | 8 | 1.16 | 5 |
| G | 50 | 262144 (131072 active) | 4 | 1.27 | 6 |
| H | 50 | 2097152 (1048576 active) | 2 | 1.38 | 7 |
| I | 50 | 16777216 (8388608 active) | 1 | 1.55 | 8 |

Table 5.6: Table detailing the node structure and active status per controlled 256 cell$^3$ grids with average cell lookup times per grid yielded from scanning through the grid and returning each cell's IActive value

Figure 5.6 shows the average cell lookup times using the IActive method per grid plotted against the log of the number of nodes in the octree using controlled test grids (A – I) where the R value gives proof of $O(N \log_n)$ time complexity .

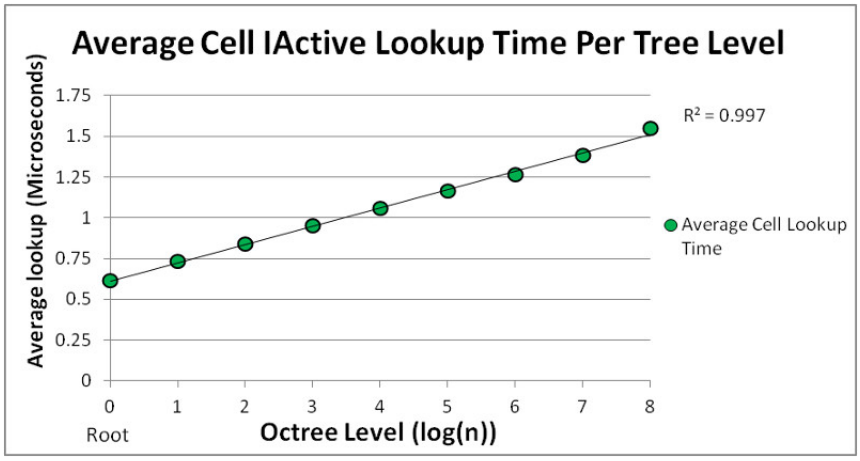



Figure 5.6: Average cell lookup time per controlled test grid (A – I) plotted against log(n) (n = the number of nodes in the tree) where the R value suggests $O(N \log_n)$ time complexity.

## 5.7 Conclusions

The system presented adopted the philosophy used in a particular system; 3D oil reservoir visualisation. This system has grids containing active and inactive cell information where pointers are used as a level of indirection, in this case the *compIndArray* as a method of indexing cell properties in order to apply cell characteristics, such as, pressures, permeability and porosity values. Traditionally this indexing is performed in a rasterscan fashion, using direct array access cell lookups, performed by directly indexing the linear input array passed from the simulator model. The compIndArray was required as the cell index order used was formed by rasterscanning through a 3D grid. If, however the octree system proposed in this thesis was substituted for traditional indexing methods the order in which cells would be referenced would now be in tree-order. As the visiting order of cells would now be in tree order, the level of indirection, required by the *compIndArray* could be omitted, resulting in a further saving of around 50% of the memory required to compress the reservoir grids using these octree compression techniques.

Using the 36 test grids supplied by Sciencesoft it can be seen that the time taken to compress the grids is directly proportional to the number of cells in the grid, as illustrated in Figure 5.1. Initial results proved that this novel method of flattening a *pruned* octree into an array of structs (*structArray*) shows greater compression ratios with grids possessing lower levels of entropy. This is because grids possessing higher levels of entropy have cells with similar active status positioned in a more stochastic fashion, where active and inactive cell placement occurs more randomly, resulting in an less dense, deeper octree, possessing leaf nodes, each encapsulating fewer cells representing smaller volumes in 3D space. Visiting all cells

in such a tree would result in far more traversals and would create larger *structArray*'s and *compIndArray* arrays, increasing the overall memory overhead allocation of the octree.

The *pruned* octree successfully compresses the oil reservoir data set in a lossless fashion where the removal of inactive nodes frees up more computer memory (Lelewer & Hirschberg, 1987). This could therefore allow larger grid models to be loaded into memory than is currently possible.

It was envisaged by Sciencesoft that a performance hit would be present, if they substituted their N2A array with this proposed octree compression technique, into their 3D visualisation software package (S3GRAF-3D). When the octree structure developed for this research (*structArray*) was integrated into their system and performance experiments conducted this proved to indeed be the case, as documented in the real-life experiments section of this thesis. It was proposed by Sciencesoft that a small performance hit of no more than 20% would be acceptable as this would not impose too much on performance as long as the main requirement of the research was to reduce the memory overhead of their current system using their N2A array. Table 5.1 illustrates the substantial memory savings achievable, especially with the larger grids. A prerequisite of this thesis for permitting visualisations of multi-million cell grids – proving its first hypothesis:

Hypothesis 1 – *Octree compression will prove to be a more efficient method for storing oil reservoir 3D active and inactive information.*

The performance hit was mainly down to the implementation of the searching algorithms adopted, Sciencesoft use direct access methods designed around array structures and so the real-life experiments showed that in order to out-perform their method calls, their software would have to be built around the octree data structure adopting searching methods such as callback functions. This was demonstrated with the controlled octree experiment results when the geological data and function methodologies were striped from Sciencesoft's S3GRAF-3D application and merged with a prototype application. In these experiments a surrogate *workload* function (introduced as a unit of computational work performed during reservoir visualisation) was applied to cells based on their active status where identical input and output values, were sought from each cell lookup as normally required at runtime, during reservoir visualisation.

Experiments 1 (columns 1 in Table 5.5) shows how the octree is ill-suited to searching using direct access methods such as treble-for-loops because each individual search instigates from the root node. In order to match or surpass state-of-the-art lookup times, searching methods have to be built around the octree structure. The octree's foreach enumerator (Experiments 2, column 2 in Table 5.5) showed a level of improvement through the elimination of boundary

checking and also that each cell searched no longer instigated a search from the root of the tree.

Further performance improvements were achieved (Experiment 3, (column 3 in Table 5.5) by adopting callback methodology. This is where the workload function was passed as a delegate function to the octree's recursive function. This emulated what performance to expect during complete traversals of the octree within Sciencesoft's software if they designed their software around the octree structure, searching in the order the tree was written (tree-order).

As leaf nodes contain volumes of cells, with their inner cells known to be obscured from view coupled with the fact that many oil reservoir calculations are not required for hidden cells, a further level of optimisation with regard to performance gain could be achieved by utilising this leaf node knowledge. This was illustrated in Experiment 4 where the workload was only applied to the periphery cells of active leaf node (column 4 in Table 5.5). The results from these experiments proved the second hypothesis, but only when the active cell status of the grid was at a low percentage.

Hypothesis 2 – *Cell lookup times will prove to be quicker using recursive traversal methods with the octree representation than that of direct access methods*.

# Chapter 6

# Hierarchical Pyramid Visualisations

The *Hierarchical Tree* and *Leaf Pyramid* techniques (discussed in the following two chapters) along with the region of interest visualisations (section 7.3) allow oil reservoir grids to be displayed at various levels of resolution, permitting less information to be sent to the graphics card when required, thus allowing larger grids to be displayed and meeting the thesis statement:

*With the adoption of the hierarchical pyramid scaling methods presented in this research, larger grids can be visualised than is currently possible.*

3D oil reservoir models are sub-divided into cells. These cells are not cubed in shape but do have 6 faces constructed using eight vertex positions defined in 3D space using $x$, $y$ and $z$-axis co-ordinates and expressed as single precision floating point numbers. The number of cells an oil reservoir model (grid) is sub-divided into defines its resolution displaying the grid to a degree of accuracy; as the resolution increases the accuracy of the grid model and each individual cell contained within it increases. There are of course limits to what a computer's CPU and GPU can store in memory defining an finite number of sub-divisions which a grid can be subjected to, thus limiting the number of cells used to represent it. This research was conducted using C# in Visual Studio 2010 version .Net 4.5 where arrays are limited to a maximum size of 2 GB within 64 bit applications [1].

As cell indices are declared as four byte words the maximum number of cells used to represent grids is just over half a billion. Such large grids generate extremely large vertex table files even when only storing active cell vertices and unique vertex positions. A grid possessing 2000 cells in a single axis may be loaded by a user who may only have a monitor containing 1024 pixels in that axis; a single pixel on the screen would represent more than a single grid

[1] http://msdn.microsoft.com/en-us/library/hh285054(v=vs.110).aspx

cell. It is therefore extremely inefficient and pointless to send all vertex position values for all cells to the graphic card to render when the viewer cannot distinguish between individual cells. The viewer can zoom into a section of the model if required at higher resolutions for exact evaluations, but if only an overview of the model is required then a scaled down version would suffice so that a cell could be displayed as an average of its neighbouring cells and would look no different to the viewer. For this reason a hierarchical pyramid scaling technique was developed where an oil reservoir grid model could be visualised at various depth using the information stored at each level in the octree structure down to its lowest leaf node level (referred to as the *Tree Pyramid*) and then down to individual cells (referred to as the *Leaf Pyramid*). The need for resolution scaling has become more important as advancements in technology has seen the latest versions of Visual Studio permitting arrays or around 8 GB to be created containing 4 byte words (Griffiths, 2012).

The hierarchical visualisation developed displays reservoir grid models at varying levels of detail from the root node to individual cells where single cells are displayed with no loss of information. Visualisations generated are based on geological co-ordinates referencing grid vertex tables directly or use a level of indirection using pointers in an indirectory when compressed vertex tables are loaded. This pyramid scaling can be viewed down to the bottom of the tree at leaf nodes level before a further level of resolution scaling is applied which sub-divides each leaf node into their individual cells. Leaf nodes contain cells at power-of-two quantities (1, 8, 64, 512... etc) each sub-division within the *Leaf Pyramid* shows eight times the level of detail than its parent level. In a bid to hasten vertex table traversal and graphic card buffering times and save memory a further level of optimisation is applied where only faces that are visible are rendered (face culling).

The term '*logical space*' refers to the cell positions within the 3D Cartesian grid of cells (3D array positions) irrespective of their vertex co-ordinates. The term, '*geological space*' refers to cell positions taking into account cell vertex positions. Cells may be next to each other in logical space such as having 3D array positions array[0,0,0] and array[1,0,0] but may not share geological space having non-sharing vertex positions, perhaps several meters apart in the real-world.

The algorithms developed perform face culling by means of nearest neighbour evaluations. Each cell face is evaluated against adjoining cell faces to determine if it is hidden in logical space before performing vertex position matching evaluations checking that they are also hidden in geological space (adjoining cell faces butt perfectly to one-another sharing vertex positions). Further optimisation only evaluates those faces on the periphery of a leaf nodes as leaf nodes by their very nature are blocks of similar cells (active or inactive) so that all internal cells are known to be hidden by outer leaf node skin cells.

## 6.1 Visualisation Options

There were three main visualisation options developed in this thesis to suit individual vertex table styles loaded at runtime. Sciencesoft Ltd only ever load vertex tables which contain unique values (compressed vertex tables) for this reason all code developed in this thesis follows this stipulation where a level of indirection is used to store pointers which point to this table. The three main visualisation options are:

1. Showing only active cells when all cell vertices are loaded (*inactivesLoaded* – all unique vertex position values loaded).
2. Showing only inactive cells when all cell vertices are loaded (*inactivesLoaded* – all unique vertex position values loaded).
3. Showing only active cells only when only the active cells vertices are loaded (referred to as a pruned vertex table – *activesOnly*).

For simplicity reasons the first option is discussed more fully than the others as the second option is just an inverse of the first and the third only requires the use of an additional indirectory array of pointers.

## 6.2 Hierarchical Tree Pyramid – 2D

The greater the number of cells used to represent a grid model the greater the accuracy but there are only so many pixels on a computer monitor screen. With advancements in computer technology processors are able to compute multi-million cell grid models (up to 400 million cells) but viewing these models poses the problem of how to visualise them. There is no point rendering all grid cell faces if the viewer cannot distinguish individual cell boundaries. A standard computer monitor used in reservoir visualisation (supplied by Sciencesoft Ltd for this research) had a screen resolution of 1680 x 1050 pixels, just under two million pixels but a such a large grid could have more cell faces to display on the screen than there are pixels.

A common animation technique known as tweening morphs shapes from one state to another over time using a timeline. If a shape is tweened from its starting shape to its end shape and screenshots taken at regular time intervals, a series of images could be seen where the shape, slowly through time appears to more accurately represent the end shape than the start shape. Figure 6.1 depicts such a scenario using an image representing a slice through a grid in the z-axis where the white area represents active cells the black inactive cells.

Figure 6.1: Oil reservoir grid model slice increasing in resolution in time step intervals from left to right

Although this is a 2D example and the resolution is increased by advancing in time, the same concept can be applied to a 3D grid model using an octree structure, instead of thinking of the model increasing in detail linearly by time (in time step intervals) instead increases in detail in all three axes with each tree level descent.

At each level in the octree there are header nodes and leaf nodes starting from the root node at the top of the tree. As the octree is written to memory in breadth-first-order each node's geological 3D co-ordinates are referenced by two different means depending on the vertex information loaded; using a cell's natural grid position when all vertex information is loaded (*InactivesOnly*) or its indirectory – *vertexInd* when only active cell vertex information is loaded (*activesOnly*). This means that if a non-pruned vertex table is loaded (a vertex table which contains active and inactive cell vertices, *InactivesLoaded*) and active cells are to be rendered, then at the root node level in the tree an octet can be drawn which represents the eight extremity grid vertex positions (the bounding box containing the oil). One level down the same can be drawn for each header node in a similar fashion. When descending the octree level by level, each header node is drawn in this way as are any leaf nodes encountered during the descent. This process is continued to the bottom level of the tree where all that remains are leaf nodes.

## 6.3 Hierarchical Tree Pyramid – 3D

Sciencesoft only use compressed vertex tables (unique values only) therefore all algorithms developed in this thesis perform visualisations using compressed vertex tables. As the recursive algorithm used for visualising the grid model depicts the grid using header and leaf nodes at various tree levels each visualisation for each level requires another pass of the *structArray* (flattened octree). As the original grid dimensions are known along with the power-of-two size (input file header) permits the computation of node sizes and logical grid co-ordinates. The maximum depth of trees are equal to the number of bits required to represent their power-of-two value but may be several levels less than this as grids containing large clusters of homogeneous regions generally generate shallower trees.

The GPU renders the vertex positions passed to it but the *Hierarchical Tree Pyramid* visualisa-

tion algorithm does not need to send it all vertex information, only what relates to the visible cell faces required at each level in the pyramid. An array of arrays is generated (*treeVertexTableArraySizes*[number of levels]) where each index in the *level* array stores an array (*treeVertexTable*[number of vertex positions]) which can hold all the natural positions of all required vertices for each level in the pyramid. The tree structure defines the number of nodes and their type at each level of the tree. These values can be used to determine the number of pyramid levels and the length of each *treeVertexTable* by evaluating the number of header nodes, active and inactive leaf nodes at each level in the octree.

If all vertices are loaded (*inactivesLoaded*) then header and leaf nodes can be used to determine vertex table sizes. Pyramid visualisation starts from the root node using node corner points to calculate the natural cell positions used to reference their geological co-ordinates within vertex tables. When a pruned vertex table is used (*activesOnly*) only active leaf nodes can be used, the first encountered determining the pyramid's starting level. This is because it is not possible to guess header node geological co-ordinates as they do not reside in predeterminable positions as they are not stored in the vertex table so only the active leaf nodes can have their co-ordinates referenced from the vertex table.

Low resolution *Tree Pyramid* visualisations are drawn using very few vertices compared to the higher resolutions of the *Leaf Pyramid* visualisations and so no face culling was performed as the memory and computation complexity posed on the GPU was not over-burdening with these smaller vertex tables. At the leaf node level, at the base of the tree there are sufficient numbers of leaf node faces to justify spending time performing face culling evaluations, especially when larger multi-million cell grids are loaded.

At each level of hierarchical pyramid visualisation algorithm the original grid dimensions, compressed vertex table, tree level, *activesFlag*, vertex table's indirectory (*vertexInd*), grid starting co-ordinates (0,0,0), power-of-two size and pass level were passed to the recursive function, *GenerateTreePyramidVisualisation*() and is discussed in detain in the following sub-section using visualisation (showing active cells when all cell information is loaded (*inactivesLoaded*).

## 6.4 Hierarchical Tree Pyramid Visualisations

Using the *demo grid* (see Appendix B) was generated in order to depict the various images generated when applying the hierarchical pyramid algorithms detailed in this chapter. This was performed by populating a 3D array at the same aspect ratio as the 2D image used in the

2D slice and having a depth of 164 cells generating an efficiently large grid for demonstration purposes. Each level in the 3D array was populated in an identical fashion generating a 3D grid model generated by stacking the 2D slices. Eight vertex positions were given to each cell so each represented a 3D volume in 3D space. The vertex table loaded was compressed so that only unique cell vertex positions were stored. Using this hierarchical scaling, it is possible to generate eight levels of resolution, from the root node to the bottom leaf nodes. Screenshots of these pyramid level visualisations are shown in sub-section 6.4.1 where node boundary lines are used to help distinguish them. In a real life example the colours used for each cell would be taken from a properties object storing the various property characteristics of the grid (oil, gas, pressure, etc) but as this is only a demonstration of the hierarchical pyramid scaling, no such property array exists and the colours given to nodes were those based on tree level and node size (see Table 6.1).

The visualisations show that the header nodes are used solely to generate the visualisations from tree level 0 to tree level 3. Header nodes have not been given geological vertex positions within the vertex table but their origin position and node edge length can be used to calculate the eight vertex positions required to encompass all their child nodes. As the octree is written in a known order (breadth-first-order) the volume of the grid which it represents can be calculated. A header node with origin co-ordinates (2, 2, 4) and edge length 4 in a grid 6 x 6 x 6 cells will have the following eight corner logical cell positions using a zero based indexing system, {(2, 2, 4), (5, 2, 4), (2, 5, 4), (5, 5, 4), (2, 2, 8), (5, 2, 8), (2, 5, 8), (5, 5, 8)} and used to retrieve cell vertex values. Grid co-ordinates are first checked to see if they lie within grid boundaries – illustrated in Figure 6.2. In the example previously given the header node's vertices which make up the $z$-axis corner cells lie outside the grid's dimensions and have to be constrained to its boundaries. Boundary cells {(2, 2, 5), (5, 2, 5) (2, 5, 5) (5, 5, 5)} are then substituted for header node corner cells as they lie on the grid's z-axis boundary and their vertices (4, 5, 6 and 7) are fetched by referencing the *vertexInd* which points to the vertex table (see Figure 3.3 for cell vertex position ordering).

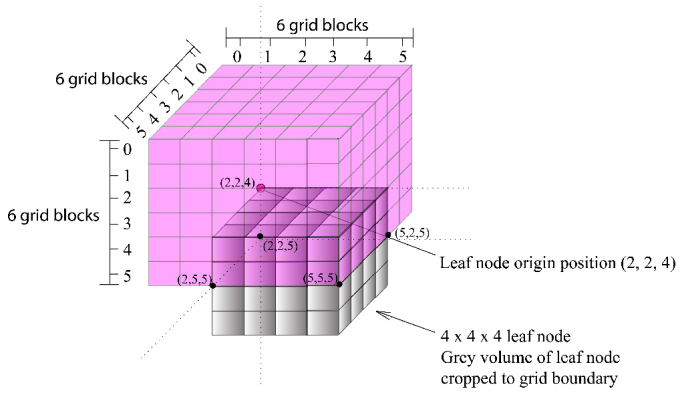



Figure 6.2: Grid boundary cropping of header node

This can also be seen in the pyramid screenshot visualisation, looking at Figure 6.4 it can be seen how the nodes in the $z$-axis of the grid are cropped to grid boundaries, resulting in a narrower row of blocks. This procedure is continued traversing down the tree and leaf nodes are drawn as encountered. The colour scheme adopted applied colours dependant on the edge length of a node and tree level. A grid with a power-of-two size of 256 will have a root node edge length at 256 but 128 one level down; as the colours of nodes change to suit node length this indicates their tree level positions and helps to portray a better visual understanding of the *Hierarchical Tree Pyramid* level structure. Table 6.1 defines the colour scheme used in these visualisations.

| Tree Level | Colour | RGB Value | Node Edge Length | Number of Cells |
|---|---|---|---|---|
| 0 | Blue | 0000FF | 1 | 1 |
| 1 | Orange | D4BB8C | 2 | 8 |
| 2 | Plum | ED1E79 | 4 | 64 |
| 3 | Purple | B43AE3 | 8 | 512 |
| 4 | Yellow | FFFF00 | 16 | 4096 |
| 5 | Green | 00FF00 | 32 | 32768 |
| 6 | Beige | D4BB8C | 64 | 262144 |
| 7 | Brown | 7B3300 | 128 | 2097152 |
| 8 | Turquoise | 28FFCF | 256 | 16777216 |
| 9 | Grey | BAB9BA | 512 | 134217728 |
| 10 | Salmon | FFB3B3 | 1024 | 1073741824 |
| 10 | Skyblue | 8BFFF9 | 2048 | 8589934592 |
| 11 | Lilac | EDA8FF | 4096 | 68719476736 |

Table 6.1: *Hierarchical Tree Pyramid* colour scheme

### 6.4.1 Visualisations

The following set of illustrations show 3D visualisations of the various levels in the *Hierarchical Tree Pyramid* using the *demo grid* where nine colours, one for each level in the pyramid were used. Looking at Figure 6.7 on page 97 it can be seen that there are two sets of colours (green and yellow). As each level has a particular colour applied to it and the power-of-two size used in this model was *256*, tree level 3 has a node length of *32* and tree level 4, *16*. With each traversal down the pyramid, header nodes are sub-divided into leaf nodes, but, Level 4 shows colours from the level above it meaning that in these nodes were active leaf nodes in level 3. This effect repeats itself down to the lowest level in the pyramid (tree level 8) where only leaf nodes remain. In the following illustrations the images on the left looks down the $z$-axis at the model and the image on the right shows a more tilted oblique view giving a truer 3D perspective view of the grid model and are based on visualisation option 1.

Figure 6.3: Tree level 0 (root node/bounding box) – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.4: Tree level 1 – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.5: Tree level 2 – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.6: Tree level 3 – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.7: Tree level 4 – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.8: Tree level 5 – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.9: Tree level 6 – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.10: Tree level 7 – Left: z-axis view; Right: tilted view – (Active cell nodes)

Figure 6.11: Tree level 8 – Left: z-axis view; Right: tilted view – (Active cell nodes)

## 6.5 Hierarchical Tree Pyramid Visualisation Algorithms

The recursive function used to generate the *Tree Pyramid* is a modified version of the basic recursive *structArray* traversal algorithm (see section 4.16) where all nodes in the flattened octree (*structArray*) are visited. At each level in the tree, header nodes are rendered using the geological co-ordinates which encompasses the volume of all its child leaf nodes. Leaf nodes are drawn in a similar manner until the pyramid reaches its base level, where no more header nodes remain, leaving only leaf nodes. The vertex table stores the vertices in $z$, then y then $x$ axis order, a common practice in reservoir engineering as the $z$-axis varies the most, then the $y$-axis then the $x$-axis. An array was used as its required size could be pre-calculated by performing a first pass of the *structArray*. At each level in the tree pyramid this array stored all required vertices of that level and passed this to the GPU which populated a vertex buffer object containing all the required vertex positions. All header nodes are cropped to the grid dimensions when required as shown in Figure 6.12 on the following page.

Constructing the tree pyramid requires traversing down to the required level in the tree and fetching the node's six corner vertex positions from the vertex array. This has a time complexity of $O(dn \log_n)$ where $dn$ is the number of nodes at the required depth in the tree and $n$ is the number of nodes in the octree.

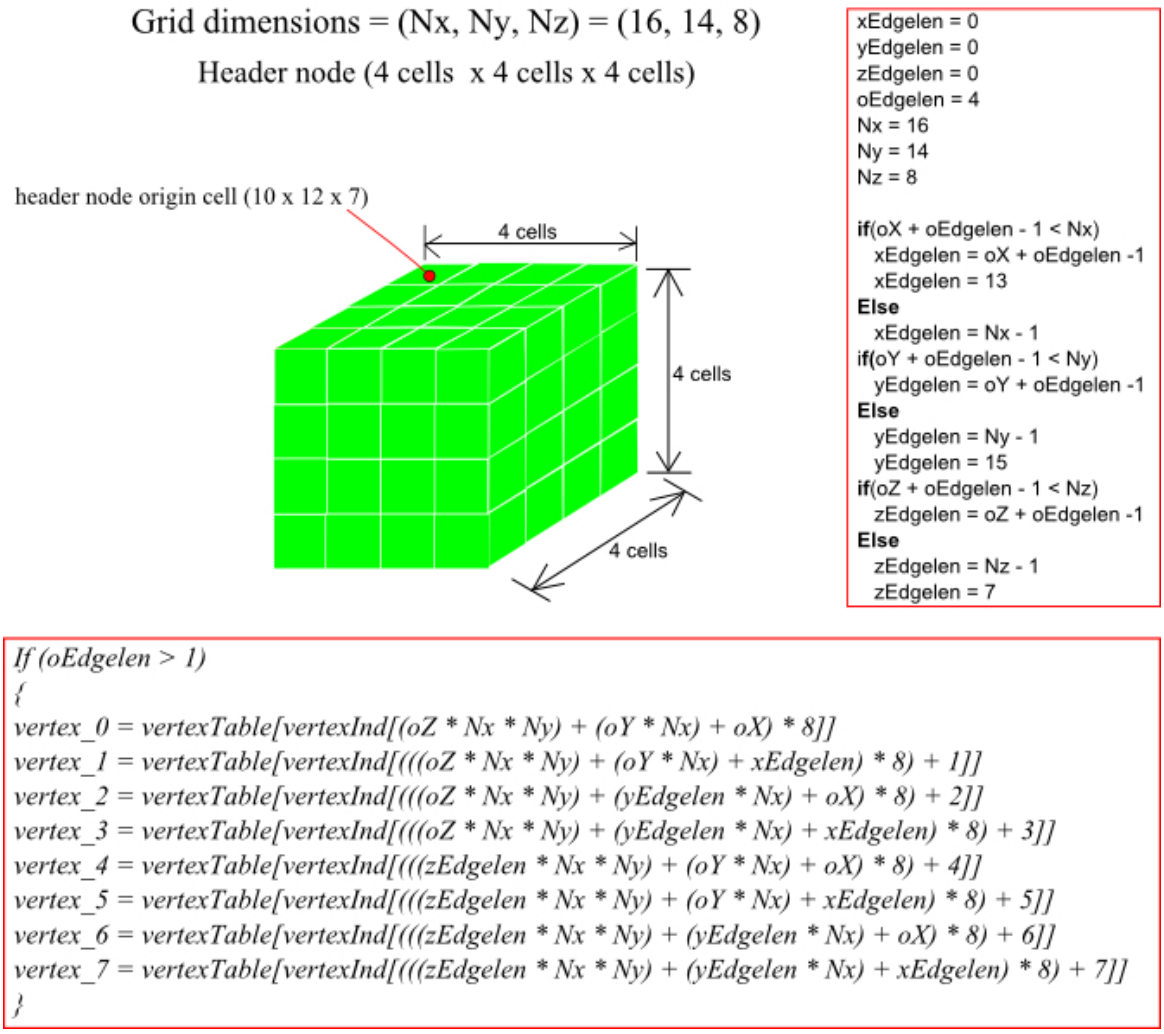



Figure 6.12: Header node vertex positions cropped to grid dimensions

The example given is for when all vertices are loaded and the natural cell index is obtained using logical (*x, y, z*) co-ordinates but if only the active cell vertex values are loaded the active index (*iActive* value) of the cell is used to reference the vertex table's indirectory (*vertexInd*).

The *vertexInd* is used to reference the vertex table and these values are entered into the *vertexArray* which is similar to a cropped vertex table as it only contains the required vertices for that level as illustrated in Figure 6.13. The GPU fills its vertex buffer object by referencing the vertex table using the *vertexArray*.

Fetching each of the node vertex positions requires six direct access lookups to the vertexInd used to populate the vertex array sent to the GPU and has a time complexity of $O(1)$ where $n$

is the number of nodes at each level in the *Tree Pyramid*.

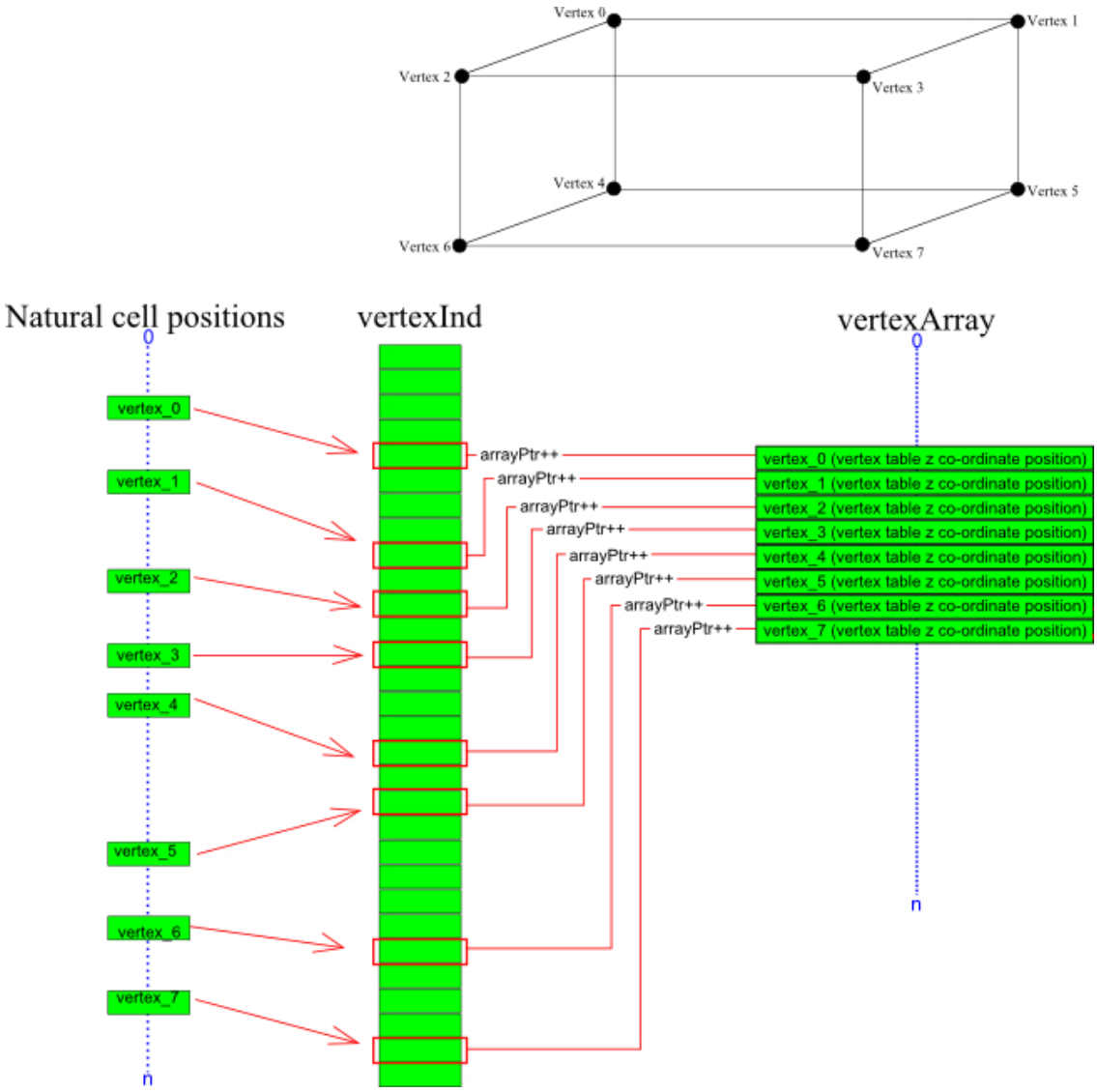



Figure 6.13: Node vertex values added to vertexArray

The algorithms for generating the *Tree Pyramid* can be found in the Appendix section of this thesis, Algorithm 10.9, where the algorithm to populate the *vertexArray* is also given (Algorithm 10.10).

## 6.6 Conclusions

The $TreePyramid$ generates 3D visualisations based on nodes, not individual cells. A leaf node may contain thousands of cells and this low detailed scaling of the tree level generates a node volumes which with its eight corner vertex positions taken from the node's corner cells

creating straight line boundaries. This is not always how the cells exist in real life as they can deviate from straight line projections having different individual axis lengths. In order to generate 3D visualisations which depict the true shape of cells, leaf nodes have to be sub-divided down to their individual cells in a hierarchical leaf scaling pyramid fashion discussed in the following subsection.

## 6.7 Hierarchical Leaf Pyramid Visualisation

The lowest level in the tree only stores leaf nodes which represent single cells to several thousand, tree level 8 using the *demo grid*. Looking at Figure 6.11 it can be seen that there are large nodes in the centre of the grid but smaller ones containing far fewer cells on its boundary. The *Leaf Pyramid* generates a hierarchical visualisation where each of its levels show increased detail by sub-dividing leaf nodes into eight octant nodes. Fractions of a cell cannot be displayed so single cell leaf nodes are displayed as as single, already rendered at their highest resolution. This hierarchical scaling is continued until the largest of its leaf nodes are displayed as individual single cells.

The number of pointers used to populate the vertexArray at each level in the *Hierarchical Leaf Pyramid* cannot be pre-determined by a single pass of the octree as in the *Hierarchical Tree Pyramid*. Using C# the size of the array required to hold these vertices could only be determined by performing two passes of the octree, the first to determine the size of the array required and the second to populate it, but a dynamic C# list object of pointers (*vertexList*) could be used and passed to the GPU as it is only scanned once.

This algorithm is applied to each level of the tree defined by the largest leaf node so that each leaf node is sub-divided into single cells but it would be inefficient to draw all these cells as many of them are hidden from view by other by cells outside their encapsulating leaf node and those cells within the leaf node volume. There was therefore a need for face culling to be applied so that the GPU only renders visible cell faces. As cells within a leaf node are homogeneous in nature an initial stage of cell culling involved ignoring the internal clandestine leaf node inner cells and only evaluating those on its periphery ($peripheryfaceculling$).

Figure 6.14 illustrates how a leaf node containing 64 cells is sub-divided into its octant nodes to single cells (3, 3, 0). The vertex positions and *vertexInd* values are given for a single cell and its encapsulating parent leaf node and octant node.

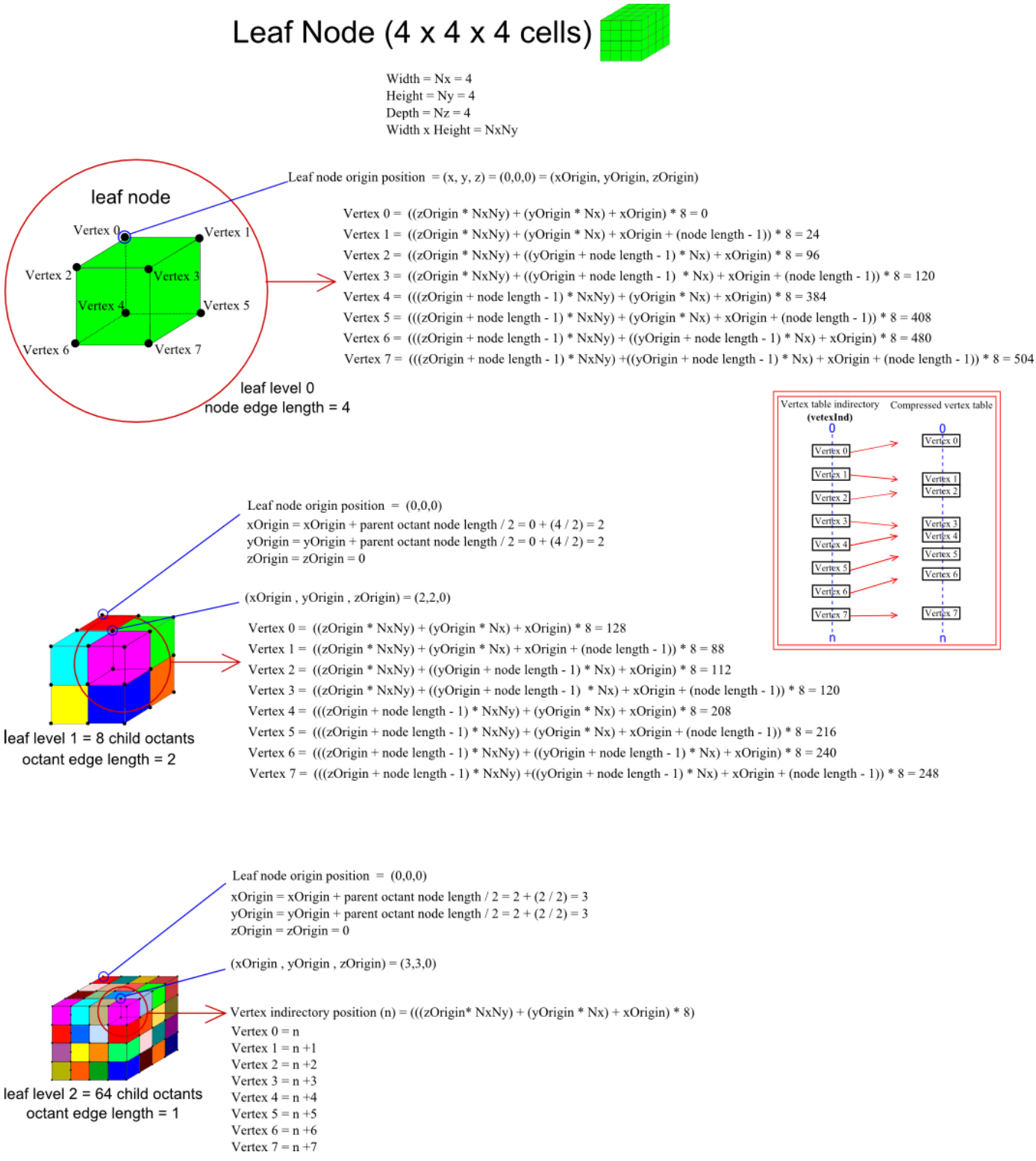



Figure 6.14: *Hierarchical Leaf Pyramid* example using a 4 x 4 x 4 cell grid

Looking at the following visualisations using the *demo grid* it can be seen how larger nodes are gradually broken down into individual cells where the same colour scheme as used in the *Tree Pyramid* visualisations was used. With each sub-division of the leaf node their octant

edge lengths is halved. With each descent in the *Leaf Pyramid*, node edge lengths are sub-divided by two meaning that this volume is now represented by the colour of leaf nodes on level below it until all nodes are represented as single cells (edge length equal to one and colour value 0000FF – blue). The *demo grid* generated a Leaf Pyramid scaling containing six levels. This is because the first active leaf nodes existed in tree level 3 having node edge lengths of 32 cells, a further five levels of sub-division are required to bring these nodes down to single cells. These *Leaf Pyramid* visualisations are shown in the following screenshot where the images on the left show the view looking down the $z$-axis and those on the right, a tilted view looking along the $y$-axis and depict the grid's active cells.

Figure 6.15: Leaf level 0 – Left: z-axis view; Right: tilted y-axis view – (Active cell octant nodes)

Figure 6.16: Leaf level 1 – Left: z-axis view; Right: tilted y-axis view – (Active cell octant nodes)

Figure 6.17: Leaf level 2 – Left: z-axis view; Right: tilted y-axis view – (Active cell octant nodes)

Figure 6.18: Leaf level 3 – Left: z-axis view; Right: tilted y-axis view – (Active cell octant nodes)

Figure 6.19: Leaf level 4 – Left: z-axis view; Right: tilted y-axis view – (Active cell octant nodes)

Figure 6.20: Leaf level 5 – Left: z-axis view; Right: tilted y-axis view – (Active cells)

The same leaf scaling can be applied to show inactive nodes and uses the inactive node information instead of the active nodes. When only the active vertices are loaded – pruned vertex table, a second level of indirection is required as a cell's natural position within a list containing only active cells (*compVertexInd*) cannot be deduced by means of a simple calculation as their positioning within this array is not based on any pre-determined manner.

The *iActive* value stored in the compressed indirectory (*compInd*) which is used to reference the property object (which is the pointer value of a leaf node in the *structArray*) returns the active index of a cell. This active index is the linear position of an active cell in a list of active cells and is used to reference the *compVertexInd* which points to the compressed vertex table. A 2D example of this referencing is given in Figure 3.4.

# Chapter 7

# Face Culling

The *Tree Pyramid* visualisations depict the grid model at low levels of detail requiring far fewer vertices than those produced at the bottom of the tree and in the leaf level pyramid levels. *Tree Pyramid* levels can be sent to the GPU with no need for hidden face culling as they are displayed at such a low resolution that the graphics card can easily store and buffer their vertex positions and still generate fast loading times and refresh rates. At the lowest level of the tree, tree level 8 (using the *demo grid* discussed in the previous chapter, Appendix B) there are sufficient numbers of cells present to merit performing performance enhancing compression algorithms. Leaf level 0 of each visualisation option is equivalent to the highest tree level in each grid model where only leaf nodes reside in the tree. Therefore during pyramid visualisation only one of these levels would be shown depending on the complexity of the model.

The *Hierarchical Tree* and *Leaf Pyramid* visualisation levels allow grids to be displayed at lower levels of detail as they contain far more cells than can be presently accurately visualised. Hidden face culling removes cell information not required to be sent to the GPU, allowing larger grids to be loaded generates quicker refresh rates, meeting the thesis statement:

*With the adoption of the hierarchical pyramid scaling methods presented in this research, larger grids can be visualised than is currently possible.*

It is inefficient sending all vertex information to the GPU to render when many of them are visually obstructed from view by cells in the foreground. One method of removing levels of redundancy is to remove hidden faces. As each leaf node is a set of cells of the same type (active or inactive) all cells inside its periphery cells are known to be hidden. This means that given a node 4 cells x 4 cells x 4 cells (64 cells ) its inner 8 cells can be omitted from any visualisation in the *Leaf Pyramid* visualisations. This leaves the leaf node's periphery cells

(56 in total) for face culling evaluation – these cell's adjoining faces and inner faces will also be hidden so that only external faces require evaluating (potentially only the skin of the node requires rendering). Leaf node level 0 requires six faces to be rendered but at leaf level 1, sub-division has created 8 octant nodes each with 3 exposed periphery faces. At leaf level 2 (individual single cells) requires only 96 periphery cell faces of its 384 total cell faces to be drawn as illustrated in Figure 7.1.

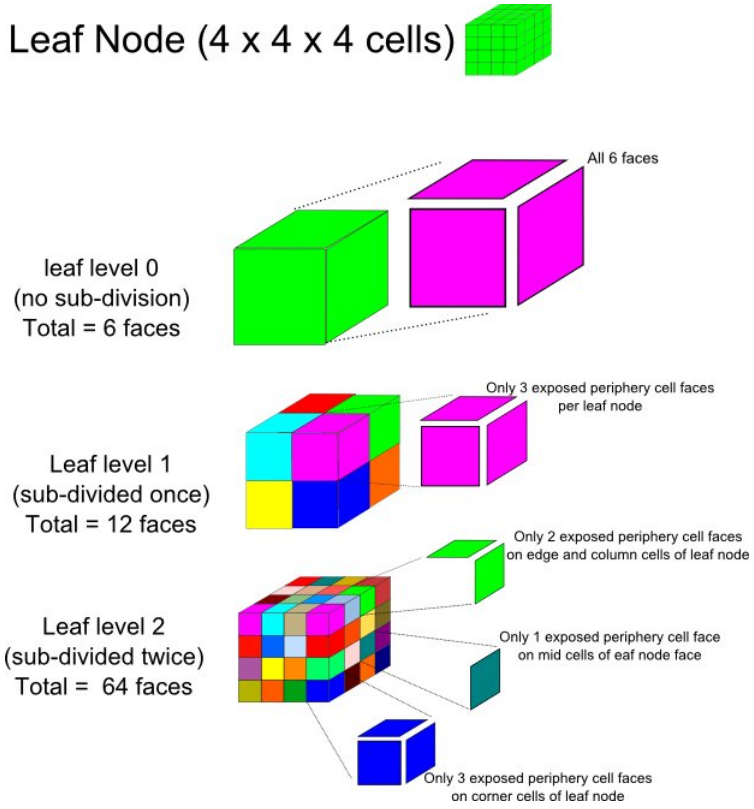



Figure 7.1: Leaf node periphery cell inner cell face culling

As the starting power-of-two size of the grid is known, each octant node's edge length can be calculated with each descent down the tree, to single cells. Looking at Figure 7.1, the starting node power-of-two size is 4 as this is the smallest power-of-two size which can hold the grid's largest edge size. One level down the tree, equals 2 and at its lowest leaf node level (*Leaf Pyramid* base level) equals 1. The periphery face culling algorithm uses node edge length information to determine which node faces are potentially visible or hidden at a given depth in the tree by first evaluating which of its cells are on its periphery. This is performed at each level in the *Leaf Pyramid*. Subsequent sub-divisions with each descent down the tree generates 8 times the number of child octant nodes, each containing the same number cells as its parent octant node.

The face culling algorithm first examines the volume of cells which the leaf node represents. At leaf node 0 the 6 faces are potentially visible. pointers to their vertices making up the face

are added to the *vertexList* and sent to the GPU for rendering. At subsequent levels in the *Hierarchical Tree Pyramid* octants are examined to establish their position with regard to the original leaf node as this determines whether it is in a set of inner hidden cells. If it lies on the leaf node's child octant's periphery it can potentially be added to the *vertexList*. Cell faces are referenced as illustrated in Figure 7.2, used frequently in 3D oil reservoir visualisation.

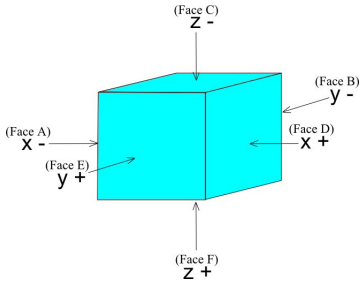



Figure 7.2: Face direction naming convention

Although face culling can dramatically reduce the number of faces to be drawn compared to drawing all cell faces it only takes into account cell and octant node faces based on a relationship with its parent leaf node, further culling is required which takes into account neighbouring nodes and cells by evaluating opposite cell and node faces in logical $x$, $y$ and $z$-axis directions (nearest neighbouring face culling evaluations).

## 7.1 Nearest Neighbour Face Culling Evaluations

In order to establish the least number of visible faces at each given level of the *Leaf Pyramid*, a further level of face culling has to be performed after the periphery face culling evaluation. This eliminates drawing faces which, although may be on the periphery of a leaf node, are still hidden behind other leaf nodes or cells. There are a general set of defining rules, i.e. when two opposing faces had the same active status then they were said to match and hidden by one-another, if they had a different active status to one-another then the face would be rendered. When faults are present in the grid sometimes two faces are slipped out of synch with one-another, even if both cells share the same active status they are rendered. Omitting these evaluations would result in more information being sent to the GPU than required as adjoining node faces sharing similar active status would be unnecessarily sent to the GPU for rendering.

The nearest neighbour algorithm works in two stages. Firstly the opposing faces are checked to see if they are hidden in a logical sense and secondly, if they are potentially hidden then their vertex positions are evaluated with each other to check that they share the same vertex

positions before being added to the vertexList and sent to the GPU for rendering, see Appendix 10.7.

Starting at the top of the $LeafPyramid$ where only leaf nodes exist, each of these leaf nodes are sub-divided into 8 octant nodes. The edge length of the leaf nodes are known and as the recursive algorithm works in power-of-2 values the octant nodes will have an edge length half of its parent node. Using the parent node's edge length and 8 ($x$, $y$, $z$) logical vertex co-ordinate positions, the child octant node's ($x$, $y$ and $z$) logical vertex co-ordinates can be deduced. Each of the octant node faces are then evaluated to see if they lie on the parent node's periphery. If they do not then they are omitted from further face culling evaluations are they are known to be hidden, obscured by the periphery cells of its encapsulating parent node.

The following discussion outlines the methodology of face culling when displaying active cells but if inactive cells were to be displayed instead, then inactive cells would be matched to other inactive cells and if butted against an active cell then the face would not be fully covered and would be rendered.

Octant node faces which lie on the parent node's periphery are then checked to see if they are hidden from view by other periphery node cells in logical terms. This means that if given an octant node containing 64 cells and its $x - axis$ face was being evaluated, all of the 16 faces on this $x - axis$ face would have to be checked to see if they were hidden by other node faces. If any of these faces are not covered, e.g. one of its face cells butts against an inactive node/cell then the full face will be sent to the GPU to be rendered. If all cells in the octant node's face are hidden in logical terms then the 4 corner vertex positions which define this face have to be evaluated with the adjoining node's vertices to check that the faces perfectly butt up against one-another and so are matched in geological space also. If they do then the face is culled as it would be hidden from view.

Frequently the cells opposite the node face being evaluated does not fit into a single node and so requires additional searches down different branches in the octree. It would be inefficient to perform individual cell searches from the root node as the required cells reside no higher than this level in the pyramid from the exact point in the tree where the face ceased to fit into a single leaf node. This would mean that these traversals would instigate from the parent node's sibling nodes. A small array of boolean values which represents each of the opposite faces is passed to each sibling node to fetch the active status of any of the opposite faces required. When opposite faces values are required from a single celled leafnode the array used is similar to cage which can store all 26 nearest neighbour cells values to the target cell, see Appendix 10.17. If all faces are hidden then the array is returned as all being true but if an inactive cell is found then the boolean would be flagged as false and the face rendered.

## 7.2 Fault Analysis

Sometimes due to faulting present in the oil reservoir's geometry, slipping occurs where the natural layering of the reservoir steps up or down resulting in sudden changes in vertex point alignment. A fault in the geometry of a reservoir manifests itself as a step in geological co-ordinates where the grids (*i*, *j*, *k*) vertex values suddenly jump out of alignment at a given point in the grid. The grid could be drawn ignoring any fault planes present but the pyramid visualisation would show unusual artifacts in the shape of sloped surfaced nodes where node faces span across fault planes.

The left-hand image in Figure 7.3 illustrates these artifacts where a simple grid is sub-divided from the leaf node to octant nodes containing individual cells. It shows how a node (highlighted in red) crosses the fault plane and has faces at a gradient as the *FaceD (vertex_3*, *vertex_1*, *vertex_5*, *vertex_7*) vertex positions of the node are used to join up the node from the vertices on *FaceA* (*vertex_2*, *vertex_0*, *vertex_4*, *vertex_6*). It is only when the octant node's sub-division lands exactly on the fault plane or when the tree is at its base level is the fault drawn correctly in geological terms. To compensate for this fault analysis is added into the face culling algorithm. If a fault exists in the reservoir model then any node which is dissected by it has its faces drawn on the fault plane eliminating the sloping faced artifacts as illustrated in the right-hand image of Figure 7.3. Looking at the this illustration it can seen that at leaf *level 0* the leaf node is split into two (highlighted in red). Leaf *level 1* again sees the left-hand octant nodes (*NW_0*, *SW_0*, *NW_1*, *SW_1*) being split into two and again in *level 2* and *level 3*.

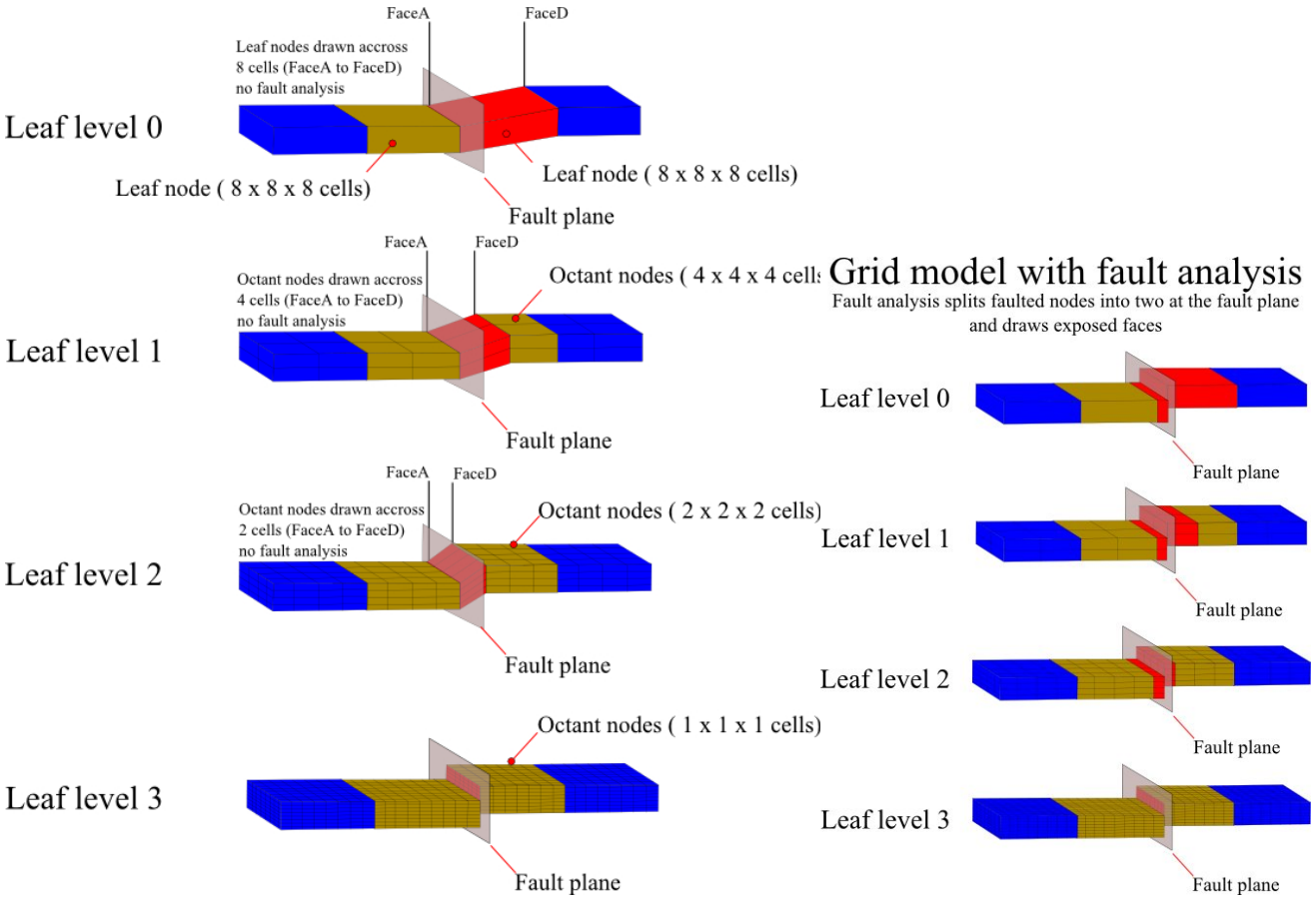



Figure 7.3: Left-hand image shows a section of a grid showing the sloping artifacts as they span across a fault. The right-hand image shows the same region with fault analysis applied with node faces drawn at fault plane

## 7.3 Regions of Interest

The octree's hierarchical pyramid scaling allows reservoir engineers to view oil reservoir grids at increasingly finer levels of detail as the pyramid is descended. If an engineer required to view a *region of interest* from the grid for highly detailed evaluations only that volume of cells is required at full resolution and should be isolated from the remainder of the grid. This *region of interest* can be rendered at the highest level of detail (bottom of the *Leaf Pyramid*) for detailed analysis. Figure 7.4 depicts a *region of interest* with starting origin co-ordinates (40, 0, 0) and a volume of 840,000 cells (100 cells wide x 60 cells high x 140 cells deep) selected from the *demo grid* which contained over eight million cells. The illustration also shows how the *region of interest* has also been dissected by a fault.

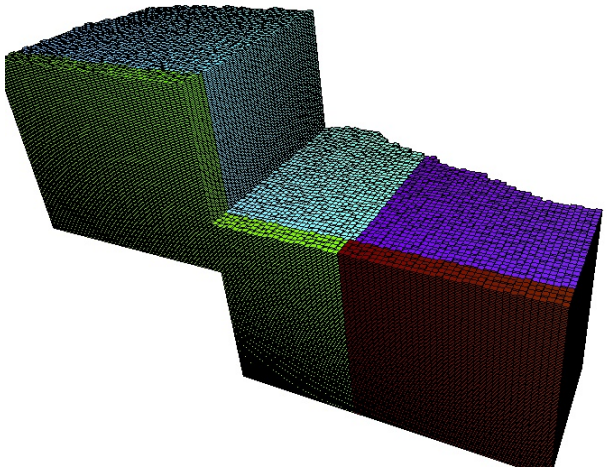


Figure 7.4: Initial test grid showing the region of interest only

can also be useful to reservoir engineers to see the placement of the detailed *region of interest* within the larger grid model, but this surrounding volume should ideally be rendered at a lower resolution, as it is of far less importance and out of the engineers focus. The larger model view can be rendered at the top of the *Leaf Pyramid*, but the *region of interest*, rendered at cell level (bottom of the leaf pyramid – highest level of detail). Alternatively the larger model view could be rendered further up the pyramid showing less detail with the selected region still displayed at the cell level or higher. Figure 7.5 illustrates the same region of interest within the larger model where two images of the same region of interest are given within the larger model where the image on the left has its leaf node boundary lines omitted from the low level rendering in order to help highlight the selected region.

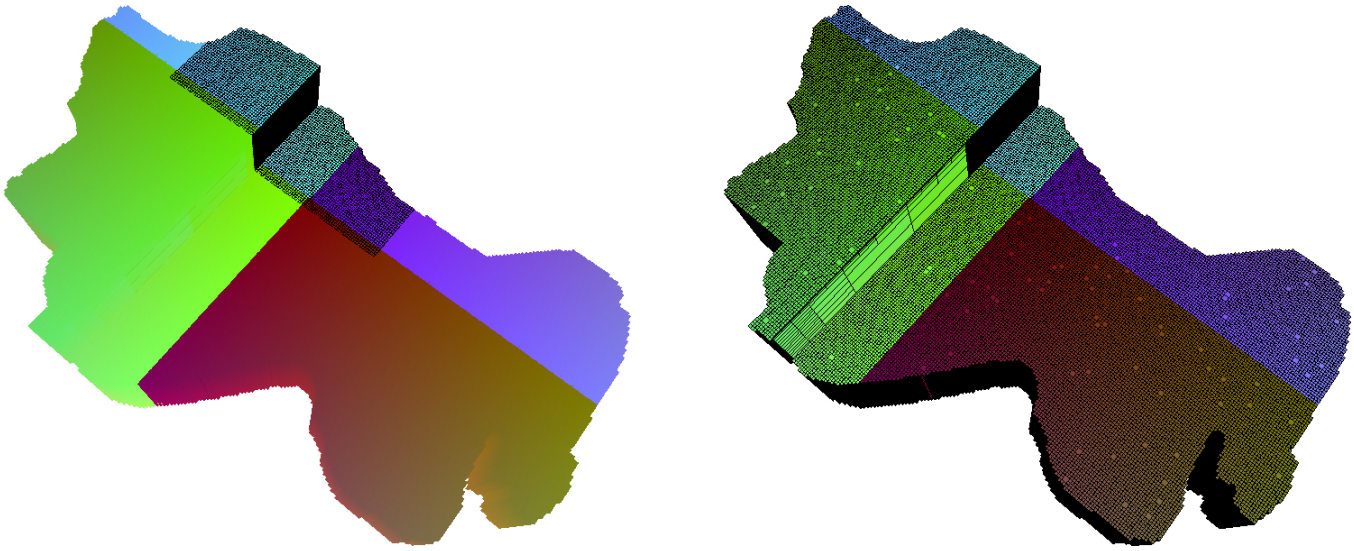


Figure 7.5: Initial test grid showing the region of interest within the remainder of the grid shown at leaf node level (Image on left – has leaf node boundary lines omitted, image on right has leaf node boundary lines added).

On many occasions the larger model view does not require drawing but the engineer may still find it useful to see the *region of interest's* placement within the surrounding grid, again not requiring to see this at a high resolution but also not requiring to see cell faces. A useful method of permitting this is to only show the wire-frame of the surrounding grid, as this has less of an overhead on the GPU and allows engineers to see all faces of the *region of interest* (Figure 7.6).

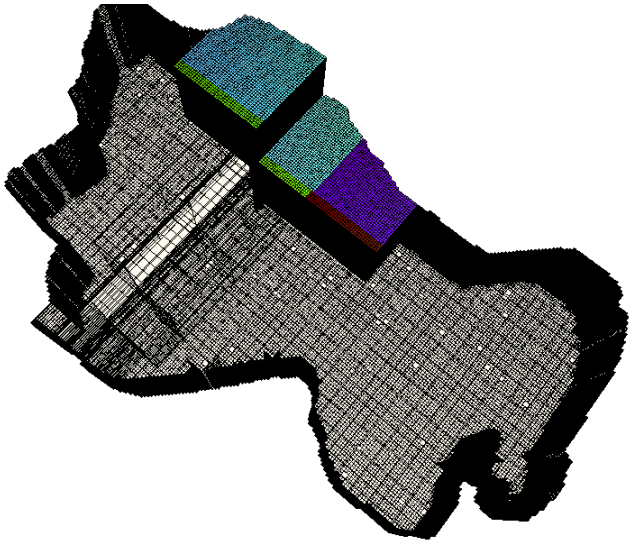


Figure 7.6: Initial test grid showing the region of interest within a wire-frame of the grid model.

## 7.4 Results and Conclusions

The *Hierarchical Tree Pyramid* displays the grids at their lowest possible detail (a single node taken from the root node) down to leaf nodes. The *Hierarchical Leaf Pyramid* displays the grids from the bottom of the *Tree Pyramid* (leaf nodes) down to individual cells where the leaf nodes are sub-divided by two at each descent down the pyramid. Table 7.1 details the number of faces sent to the GPU as size ratios based on each of the initial test grid's pyramid levels where each value given was the number of faces which would be drawn if no face culling was performed divided by the number of faces sent after face culling was applied. The values are based on the results from the *demo grid* when all cell information was loaded and active cells were drawn.

| Tree Pyramid Level | *Demo Grid* – Size Ratio (Number of active faces per grid / Number of faces per pyramid level) |
|---|---|
| **0** | 45309264 |
| **1** | 943943 |
| **2** | 209765.11 |
| **3** | 34015.96 |
| **4** | 7965.76 |
| **5** | 2058.76 |
| **6** | 507.12 |
| **7** | 166.65 |
| **8** | 73.87 |
| **Leaf Pyramid Level** | |
| **0** | 399.34 |
| **1** | 288.29 |
| **2** | 238.79 |
| **3** | 232.07 |
| **4** | 224.17 |
| **5** | 215.55 |

Table 7.1: Size ratio of faces sent to the GPU of the initial test grid (240 x 204 x 164 cells) based when showing active cells at each level in the tree and *Leaf Pyramids*

Looking at this table it can be seen that the size ratio drops at *Tree Pyramid* level 7 and 8 and rises again at *Leaf Pyramid, level 0*. As previously explained in this chapter the lowest level in the *Tree Pyramid* would not be displayed but instead replaced by the *Leaf Pyramid*, *level 0* as this is the same visualisation except leaf *level 0* has face culling applied but as it is split into octant nodes there are potentially 8 times as many node faces. It was thought that not applying face culling until the bottom level of the *Tree Pyramid* would prove to be efficient strategy as the number of nodes to be drawn at these levels would be so small in comparison to the total active cells, but looking at the results from the initial test grid it can be seen that it would prove more efficient in saving GPU memory by performing face culling at the second bottom level of the *Tree Pyramid* as well. This saving would depend greatly on the characteristics of the grid evaluated; grids with higher levels of entropy and smaller leaf nodes would have less faces culled due to a greater number of inactive cell faces being present amongst active ones.

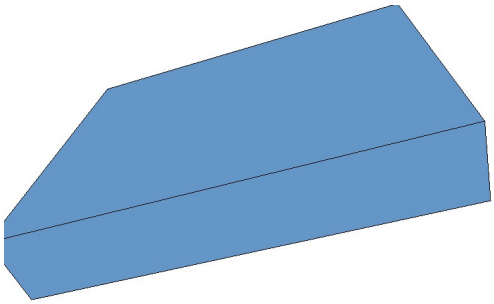
To test if the same size ratio trend can be seen in a real-life scenario the large grid, *Grid 32* used earlier in chapter 5 for the memory and performance experiments was used because of its size and that it possessed one of the highest levels of entropy out of the 36 test grids. Table 7.2 details the number of faces sent to the GPU as size ratios based on each of its pyramid levels where each value given was the number of faces which would be drawn if no face culling was performed divided by the number of faces actually sent after face culling was applied. It can be seen by looking at these results that the size ratio of the number of faces sent to the GPU drops in the second bottom level of the *Tree Pyramid* and rises again in *Leaf Pyramid*, *level*

*0* similar to the trend displayed with the initial test grid. This would suggest that face culling should be applied higher up in the pyramid, at the second bottom level of the *Tree Pyramid* as opposed to the bottom. This grid has less L*eaf Pyramid* levels compared to the initial test grid due to it having its largest leaf node on power-of-two size smaller than the initial test grid's.

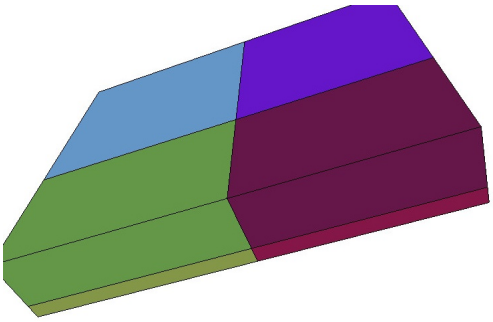
| Tree Pyramid Level | Grid 32 – Size Ratio (Number of active faces per grid / Number of faces per pyramid level) |
|---|---|
| **0** | 8964780 |
| **1** | 498043.33 |
| **2** | 93383.13 |
| **3** | 15563.85 |
| **4** | 2603.014 |
| **5** | 361.25 |
| **6** | 56.50 |
| **7** | 10.85 |
| **8** | 3.58 |
| **Leaf Pyramid Level** | |
| **0** | 15.08 |
| **1** | 14.32 |
| **2** | 14.20 |
| **3** | 14.19 |
| **4** | 14.19 |

Table 7.2: Size ratio of faces sent to the GPU test grid D (196 x 129 x 105 cells) based when showing active cells at each level in the tree and *Leaf Pyramids*

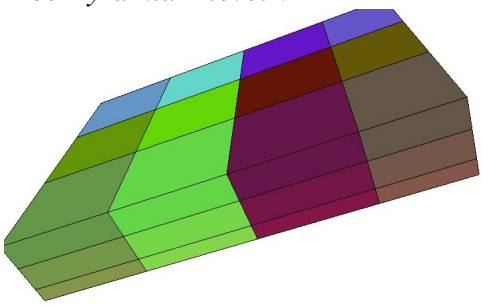
Looking at the size ratios of *Grid 32* it can also be seen that the last two values are identical. This is because although the between leaf level 3 and 4 octant nodes were being sub-divided from eight cell nodes to single cell nodes the extra level of sub-division did not generate any extra visible faces meaning that one level up from the bottom of the *Leaf Pyramid* was at the highest level of detail possible for viewing. The following set of illustrations show the initial test grid through the various levels of the pyramid showing the active cells. It can be seen that there level of detail is very distinguishable between the top of the *Tree Pyramid* and level three but lower levels look almost identical. The node and cell boundary lines have been inserted to help illustrate the sub-divisions applied at each level of the pyramids. If a less detailed analysis of a 3D oil reservoir grid is only required of particular volume of cells using *Tree Pyramid* level 4 may suffice as it clearly depicts all the characteristics which defines the grid. This allows engineers to view grid models at with quicker refresh rates using less memory and GPU power than previously whilst permitting the use of less powerful GPUs. This grid contains just over eight million cells but greater savings will be achieved with larger grid models containing hundreds of millions of cells.



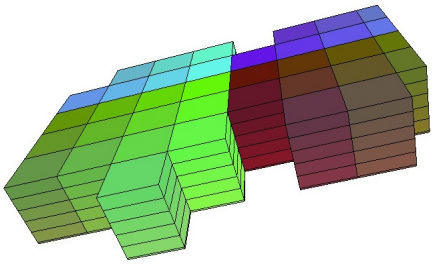
*Tree Pyramid – level 0*



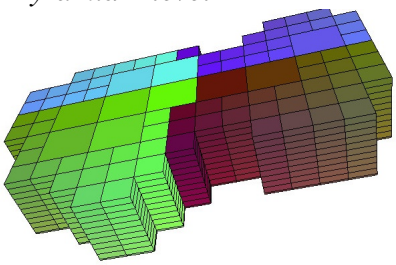
*Tree Pyramid – level 1*



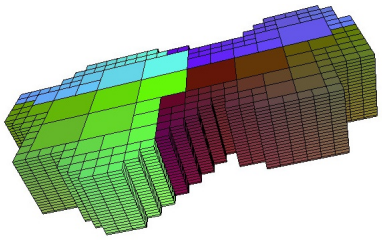
*Tree Pyramid – level 2*



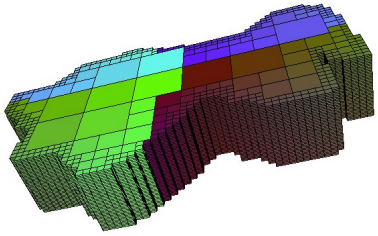
*Tree Pyramid – level 3*



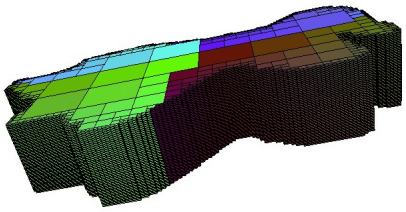
*Tree Pyramid – level 4*



*Tree Pyramid – level 5*

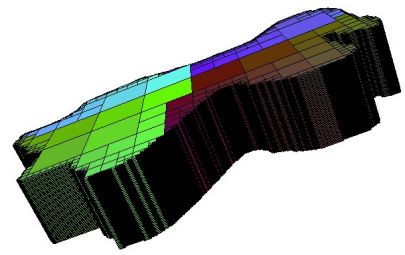


*Tree Pyramid – level* 6

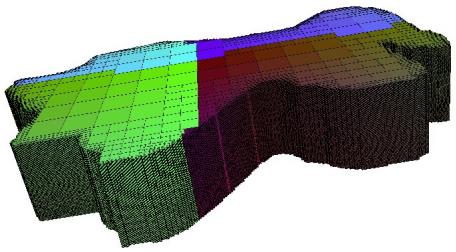


*Tree Pyramid – level 7*

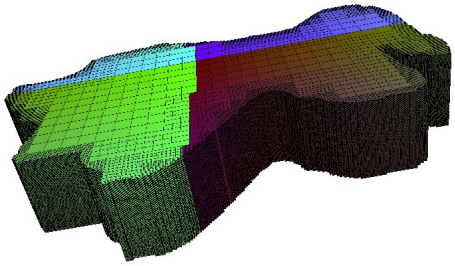
Figure 7.7: Hierarchical *Tree Pyramid* visualisations of *demo grid* (level 0 to 7)



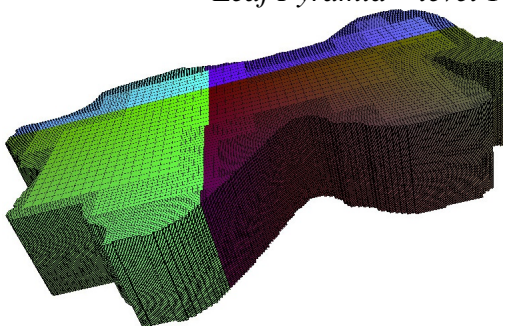
*Leaf Pyramid – level 0*



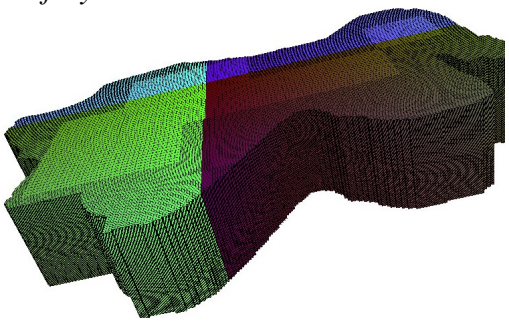
*Leaf Pyramid – level 1*



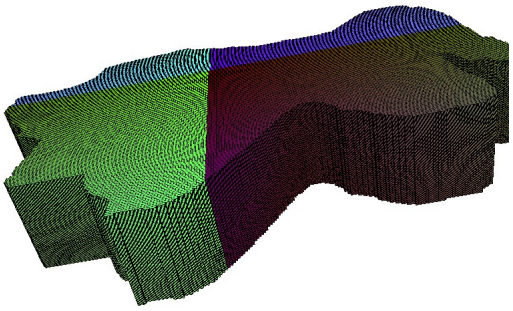
*Leaf Pyramid – level* 2



*Leaf Pyramid – level 3*



*Leaf Pyramid – level* 4



*Leaf Pyramid – level* 5

Figure 7.8: Hierarchical *Leaf Pyramid* visualisations of *demo grid* (level 0 to 5)

When the face culling algorithm is applied to a grid, only the exposed faces are sent to the GPU so that large homogeneous regions will only have its exposed skin drawn. Given a 100% active (convex grid) grid where actives are being drawn this would create a completely hollow grid model where only its skin is drawn. The following illustrations in Figure 7.9 shows views from inside the *demo grid* after applying face culling and illustrates how all its inner hidden faces have been culled. Grids with larger leaf nodes will create larger cavernous regions during visualisations greatly reducing the number of faces required to be drawn leading to quicker refresh rates as well as permitting the use of smaller GPUs.

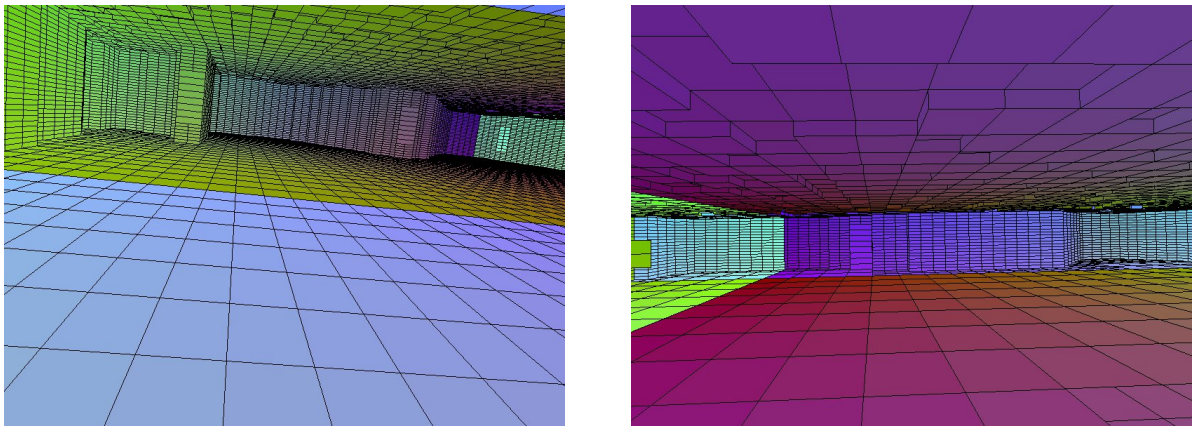


Figure 7.9: Inside views of the face culled test grid where only the outer skin has been drawn

Although the face culling shows its efficiency with regard to GPU memory savings by being able to send fewer cell faces to the GPU through the face culling and octree compression techniques developed in this thesis, an analysis had to be performed to gauge its impact on runtime performance. To test this the time taken to perform the face culling evaluations for a sample set of test grids taken from the 36 grids supplied by Sciencesoft. Five were chosen as they represented the range of grids as it included the small and the largest grids and those of lower and higher entropy levels and with varying percentages of active cells. The sample set of grids included test grids {*Grid 24, Grid 27, Grid 30, Grid 32, Grid 36*}, detailed in chapter 5 on page 67). Figure 7.10 illustrates the results yielded from these experiments and it can be seen that as expected the time rises per number of nodes being evaluated with a slight exception with of *Grid 30* and *Grid 32* but this because *Grid 30* has small nodes indicated by the fact that its largest only contained sixty four cells as it only had three *Hierarchical Leaf Pyramid* levels meaning that on leaf level 2, *Grid 30* was being evaluated using the *Cage* searching for single cell faces around the target whereas *Grid 32* at this level was still performing many *faceArray* searches where the faces had sixteen cells to search and so was performing more cell look-ups (see Appendix 10.17).

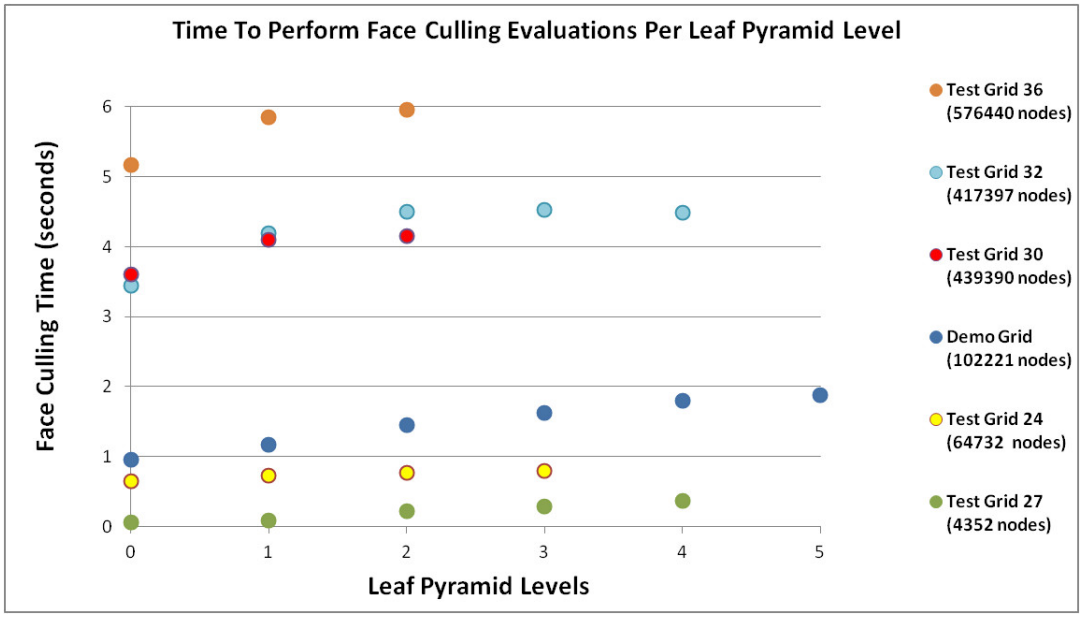



Figure 7.10: Time taken to perform face culling at each level in the *Hierarchical Leaf Pyramid* levels for the five test grids and initial visualisation test grid.

The hierarchical pyramid scaling algorithms allow reservoir grid models to be rendered at varying levels of detail, suiting the individual visualisation demands required for different runtime scenarios. This *region of interest* visualisation methods developed in this research allows engineers the flexibility of viewing highest level of detailed volumes of the grid while still displaying the full grid at far lower levels of detail. By applying these techniques grid models can be visualised at various levels of detail eliminating the need for the GPU to inefficiently render the grid at full resolution when it is out of the focus of the engineer. The storage space saved on the GPU will allow for larger grids to be loaded and visualised meeting the thesis statement made in this thesis:

*With the adoption of hierarchical pyramid scaling methods larger grids can be visualised than is possible today.*

# Chapter 8

# Conclusions

Results are used to prove the hypotheses made in this research:

1. *Octree compression will prove to be a more efficient method for storing oil reservoir 3D active cell information.*
2. *Cell lookup times will prove to be quicker using recursive traversal methods with the octree representation than direct access methods.*

This thesis meets the first hypothesis made in this thesis by successfully being able to compress the 3D oil reservoir active cell information using the lossless octree compression techniques developed in this research. The second hypothesis has been met by developing algorithms which depict the grid at various levels of resolution in a pyramid fashion generated from the taking information stored at each level in the octree structure. This chapter discusses the conclusion derived from the experiments conducted on real oil reservoir grids, chapter 5 and hierarchical pyramid evaluations, chapter 6 and 7.

Throughout this thesis the various characteristics which make up oil reservoirs, and how cell information is stored, has been discussed, such as how they were formed and how hydrocarbons are dispersed throughout their entirety, see chapter 1. Oil reservoir visualisation software demands have also been discussed, such as how these grids are represented as a collections of cells in logical and geological space (chapter 3). This thesis discusses how cell boundaries are defined from seismic data, and how grid cells are given 3D geological vertex positions. Detailed explanations of active and inactive cells have been given, and how reservoir engineers and visualisation specialists such as Sciencesoft use this active cell information for visualising grids detailing the various state-of-the-art techniques and data sets used in industry, see chapter 3.

The need for accuracy and advances in computer technology has led grid models being represented at higher levels of detail in increasing numbers of cells. These grids can represent volumes of rock several miles in both directions horizontally, but sometimes only several hundred feet vertically. Searching and visualising these large grids has generated a bottle-neck effect, although grids can be sub-divided, generating many more cells which can easily be stored on disk than before, loading and processing this level of information requires careful consideration as many more cells generate a proportional rise in calculations which have to be performed, such as nearest neighbour face culling evaluations, where the increased numbers of cells could significantly increase grid traversal times. GPUs are only capable of storing a finite number of vertices in their buffer memories before impacting refresh rates. The time taken searching for information within these larger grids also increases due to the increased number of cells which have to be searched through. This research looked at how this trend can be reversed by applying algorithms which took advantage of the natural clustering characteristics of grid cells using their active cell information (chapter 4).

The novel approach of storing this active cell information using octree compression techniques was introduced which was able to successfully integrate with actual software in production today, where not only was the grid information stored at a fraction of its original size, but cell lookup times outperformed state-of-the-art direct access methods, see sub-section 5.5.1.

As the number of cells used to represent these oil reservoirs has increased to several million and is set to increase to hundreds of millions, the ability to individually display each cell on a computer monitor screen has become impossible as there are more cells than pixels – each pixel would represent several such cells. There is no point performing individual cell calculations which are then sent to the GPU for visualisation when the human eye cannot perceive individual cells, improving this inefficiency required removing levels of redundancy. The information sent to the GPU should only be at a sufficiently high level of detail to portray the relevant information to the reservoir engineer, removing the need for over-burdening the CPU and GPU with information not required. In order to address this problem, hierarchical pyramid visualisation scaling algorithms were developed (chapters 6 and 7) derived from nodes at each level in the octree so as to generate a series of grid visualisations containing various levels of refinement.

The pyramid was split into two stages, the first being the *Tree Pyramid* (subsection 6.4) which represents the grid as a 3D model starting from the octree root node down to its leaf nodes and secondly, the *Leaf pyramid* (subsection 6.7) which depicts the 3D visualisations derived from the leaf nodes, down to the individual cells, thus bestowing the reservoir engineer with a range of models, each displaying an increased level of detail as the pyramid is descended.

This following sub-sections of this chapter detail conclusions derived from applying these algorithms for storing, searching and visualising reservoir data given in this research and is sectioned as follows:

- Suitability – analyses the suitability of using octree compression techniques looking at how this compression technique performed with actual grids, detailing their individual characteristics which defined resulting tree structures (see table 5.1).
- Memory – details the memory savings achieved using real-life grid models where the test grids supplied by Sciencesoft were successfully compressed to a fraction of their original size (see Table 5.1).
- Performance – discusses how the algorithms developed in this research outperformed the state-of-the-art methods used today (table 5.5).
- Visualisation – reflects on the pyramid visualisation techniques given in this research and details how this can be integrated into reservoir visualisation data used today, enabling engineers to study grids at low level resolutions when required while permitting high levels of refinement at specific regions of interest (see chapter 6).
- Integration – compares and contrasts octree compression techniques, with direct access array storage methods discussing how this research has given insight into how today's methods of storing and traversing reservoir grids could be adapted to suit tree structures (see subsection 5.7).

## 8.1 Suitability

Like any algorithm or compression technique its efficiency is generally dependant on its suitability with the data type and structure it has to compress. This is also true with octree compression techniques, where its strengths lie in being able to represent 3D volumes as single nodes irrespective of the size of the volume which each represent. It does this by sub-dividing 3D space into regions of similar types; grids containing large clusters of similar type cells create nodes which represent larger volumes, which in turn generates shallower trees, achieving higher levels of compression. Oil reservoirs are stored as active and inactive cells, these cells naturally occur in clusters and are ideally suited for octree compression, where each node represents a 3D volume of oil reservoir rock.

Further compression was achieved when the octree was *pruned* (inactive nodes removed, see Figure 2.6). Grids possessing very low or very high active cell percentages can sometimes generate better compression ratios than similar sized grids because of their larger clusters of homogeneous volumes as this results in a more compressed octree structure containing less

nodes, many of which representing larger volumes of cells, such as test grids 36, see Table 5.5. It is of course not only the active percentage of a grid, which governs the compression ratios achievable, but how these clusters of active cells are displaced throughout its entirety. This active and inactive cell scattering can be measured by following a path through the grid, where cells situated close on the path also reside within close proximity to one-another in the grid, the alternating states of cell active information was used to define grid entropy – a measurement of grid clustering (see section 2.5). As oil reservoir grids tend to store active cells in clusters, those possessing low and high percentages of active cells, tend to have lower levels of entropy due to them being less stochastic in nature.

## 8.2 Memory

*Convex grids* have their inner cell faces perfectly matched to one-another, and this is generally the case with most oil reservoir grid models. This has a significant impact on the octree compression style applied, when the model is is first loaded only the exposed outer cell faces are rendered irrespective of the active status of its inner cells, as these are naturally hidden from view by outer cells. An initial first pass of a grid model is all that is required to populate a list of active faces which can be saved to disk and loaded, the next time the grid loaded. Only when a grid has been clipped or cut are cell faces exposed and have to be re-evaluated and the list of active cell faces, updated for any subsequent run. This is a practice generally adopted in the 3D oil reservoir field and the same concept could be applied using octree compression techniques so that . If on first load all that is required to draw is the outer skin of the model then the grid could be compressed where it is presumed to be completely active, resulting in a far smaller tree structure and file. Only when the grid is cut or clipped in some way would inner cells require evaluation.

The octree data structure developed in this research overcame the memory constraints of using a hierarchical C# octree class structure discussed in section 4.2 of this thesis as this proved to be inefficient in memory due to the overheads of the programming language adopted C#, a prerequisite of this research, as any developed code had to integrate seamlessly with Sciencesoft's software. These were the memory overheads derived from the classes; header nodes had pointers to child nodes resulting in a 64-byte overheads and node classes of 24-bytes and could be significantly diminished using a functional programming language instead of an object orientated one.

The array of structs (*structArray*) designed to meet all the requirements of the system addressed all the problems of the applying the hierarchical and flattened-out octree data struc-

tures by removing the need for storing null pointers to inactive nodes, and as the structs stored pointers as integers this effectively halved the size of pointers to active child nodes, even in a 64bit computer (see section 4.3) and could still be thought of as a flattened-out hierarchical octree. Memory was also saved through the elimination of class overheads as nodes were contained within a linear array structure, requiring only one reference overhead of 16 bytes and the *structArray* proved to outperform industry standard state-of-the-art lookup times.

The test grids used to test the algorithms in this thesis were supplied by Sciencesoft and the compression achieved using octree compression, see Table 5.1. Using the octree compression techniques developed in this research, all 36 test grids supplied by Sciencesoft were successfully compressed to a fraction of their original file size in a lossless fashion. Even the largest grid possessing over 25 million cells yielded a compression ration of over 15.

The time to compress these test grids grows linearly to the number of cells in the grid, as illustrated in Figure 5.1.

Octree compression has therefore proved to be an extremely efficient means of compressing oil reservoir active cell information or indeed any other 3D grid possessing high levels of similar values in clusters. The results of the compression achieved from applying octree compression to the actual grid models supplied by Sciencesoft prove the first hypothesis made in this thesis:

Hypothesis 1 – *Octree compression will prove to be a more efficient method for storing oil reservoir 3D active and inactive information than is currently used today*.

## 8.3 Performance

Oil reservoir engineers constantly monitor oil reservoir cell information for establishing which production techniques should be applied adopting a variety of visualisation techniques to quickly estimate, check and display cell, wellbore information and flow rates. Cell lookup times are extremely important as these calculations can be extremely complex and with grid models sometimes containing millions of cells, delays in visualisation and inaccuracies in data would hold up production – it is of the utmost importance that the software is reliable yielding high levels of accuracy whilst displaying results as fast as possible. Sciencesoft's software is used extensively around the world in over 80 countries by reservoir engineers, and so it was imperative that any software developed could match their industry standards (direct access). Sciencesoft envisaged that there may be a small, fifteen to twenty percent overhead

performance hit if they substituted their array structures with the octree structure developed in this research, but this would be offset by the impressive memory savings.

In order to gauge the performance of the octree structure a series of experiments were conducted in two distinct flavours, as discussed in see chapter 5 .

- Real-life experiments – where the *structArray* was substituted for Sciencesoft's present array structure within their S3GRAF-3D oil reservoir visualisation package (section 5.4).
- Controlled octree experiments – where a prototype application specifically developed to test the performance of the octree structure with all the overheads of Sciencesoft's classes removed. These experiments were designed to to gauge the *structArray's* optimum performance, as the searching techniques applied were designed around the octree structure, as opposed to array structures (section 5.5).

On many occasions Sciencesoft's scan through there arrays performing individual cell lookups, using treble-for-loops, where all the cells are visited in the $x$, then the $y$ and then the $z$ axis and single-for-loops, where the array is searched in a sequential manner from its first to last element. These loops were substituted with the octree data structure (*structArray*) and indexed using a bespoke enumerator specifically developed in this research (section 4.15) to return the necessary cell values required at runtime, such as a cell's active cell index, or natural cell position.

The results from the *real-life* experiments showed that on many occasions the octree performed within the 20% threshold such as with the *cut plane* method calls as well as sometimes out-performing direct access methods, see Table 5.4. Certain methods required individual cell lookups such as for face culling purposes and isosurfacing calculations. As there was no provision developed for optimising individual cell searches at the time of the *real-life* experiments, individual cell lookups were initiated from the root node, so methods such as for volume clipping yielded poor performance times. The development of an algorithm which could remember where it was in the tree, searching for multiple cells in surrounding nodes from the last searched position in the tree was not developed until later during when the hierarchical pyramid visualisation algorithms were being developed, see chapter 7.

There were many other classes and method calls which were not accessible to the researcher, made from within S3GRAF-3D during the experiments due to the architecture of the software. This meant there existed a lack of control in optimising the octree structure, so to gain a true and accurate comparison of *structArray* performance a prototype program was developed which took all the fundamental requirements from each of S3GRAF-3D's method calls but did not have any of its class overheads or hidden single cell lookups. It was hoped that by removing these redundancies a true analogy of the octree performance could be ascertained.

This was achieved by performing direct access lookups in a scanning fashion and comparing these times to that of the structArray's lookup times (*Controlled octree experiments*, see section 5.5.1).

Various experiments were conducted which involved searching the grids using various searching techniques (iterator, enumeration and callback) where a dummy *workload* was applied to active cells as it is normally always the case that calculations are only performed on active cells as these contain hydrocarbons. The dummy *workload* was a unit of work which stopped the compiler optimising searching and emulated computational work carried out in a real-life environment, such as face culling evaluations.

The iterator experiments where cells were searched in a scanning fashion using treble-for-loops showed that, as expected the octree performed very poorly in comparison to direct access methods. This was because arrays are well suited for searching in rasterscan formations unlike the an octree structure which seen each new search start from its root node, (column 1 in Table 5.5). A bespoke enumerator was developed which would visit each node in turn within the *structArray* (see section 4.15) which could return all required values as a struct using a foreach loops. As expected the results from these experiments showed improvement (column 2 in Table 5.5) due to the elimination of boundary checking, but still did not match the fast lookup times of direct access.

In order to evaluate how well the *structArray* structure could perform within Sciencesoft's software if they were to re-design their application around tree storage instead of 3D arrays a further set of experiments were conducted. As the *structArray* was written recursively in breadth-first-order, it seemed very plausible to expect improved results if the cell lookup method was re-designed to suit the data structure visited order as it was written. This level of optimisation used a callback method where the octree was recursively traversed in the same manner as it was written and the *workload* applied performed to all active cells. As expected these results proved that by adapting the traversal method to suit the data structure the *structArray* could match or out-performed direct access methods more with more than half of the test grids with an average runtime ratio of 1.1 (column 3 in Table 5.5). This means that on average the callback method proved to be 10% quicker using the structArray than direct access.

As a further level of optimisation a further set of experiments were conducted. Leaf nodes, by their very nature, encapsulate clusters of cells sharing the same active status, within the context of this thesis. With this in mind, inner cells are naturally hidden from view by outer cells, on the leaf node's periphery. This means that there is no point performing calculations such as face culling evaluations on leaf node inner cells if they are known to be obscured

from view by those cells on the node's periphery, so at runtime calculations and rendering of inner nodes could be ignored, except where clipped or cut. A further set of experiments were conducted utilising this optimisation in order to establish whether greater performance gains could be achieved when only applying the *workload* to leaf node periphery cells. The results proved this to indeed be the case, see column 4 in Table 5.5. Looking at these results it can be seen that the *structArray* outperformed direct access with almost all the grids yielding an average runtime ratio of 1.4, an average of 40% quicker than direct access. Grids possessing larger clusters of active cells would generate larger leaf nodes and in turn would yield faster runtime ratio when applying this leaf node periphery optimisation as the leaf nodes on average would contain larger quantities of inner cells omitted from the workload calculations.

These results showed that octree compression has therefore proved to be an extremely efficient means of not only compressing oil reservoir active cell information, but can also outperform state-of-the-art direct access methods as detailed using the 36 test grids supplied by Sciencesoft – proving the second hypothesis made in this thesis:

Hypothesis 2 – *Cell lookup times will prove to be quicker using recursive traversal methods with the octree representation than direct access methods.*

## 8.4 Hierarchical Pyramid Visualisation

Part of this research was dedicated to developing a hierarchical pyramid visualisation algorithm which could exploit the hierarchical tree structure of the octree as stated in the thesis statement:

*With the adoption of the hierarchical pyramid scaling methods presented in this research, larger grids can be visualised than is currently possible.*

This was successfully developed where visualisations were generated using the tree nodes in the octree, see chapter 6. Two pyramid scaling techniques were developed:

- *Tree pyramid* – visualisations from the root node to the leaf nodes, section 6.4.
- *Leaf pyramid* – visualisations from the leaf node to individual cells, section 6.7.

These pyramid algorithms were developed addressing the three standard visualisation styles as defined in section 6.1. Various GPU optimisation techniques were integrated into the pyramid algorithms such as face culling (see chapter 7) algorithms which culled hidden matching node and cell faces. The nearest neighbour algorithms developed would also address the problems of slow lookup times which occurred in the *real-life* experiments, where individual nearest

neighbour cell lookups were instigated from the root node instead of sibling nodes and so would greatly reduce search times.

Grids often contain far more cell faces than is possible to visualise accurately, it is not possible to render all cell information of multi-million cell grids on a standard monitor, nor could the human eye distinguish between them. It is therefore very inefficient to send all cell information to the GPU for rendering when it cannot possibly draw them correctly as several cells would share a single pixel. Reservoir engineers only require to view the model with enough detail which portrays the model at a satisfactory level, suiting their task. The hierarchical pyramid visualisation scaling developed in this research displays grid models at each level in the octree, down to the leaf nodes using the *Tree Pyramid,* then sub-divides these leaf nodes into their octant nodes until only individual cells are rendered (*Leaf Pyramid).* This allows engineers to view the model at their required level of detail and not overburdening the GPU with vertices which can not be accurately drawn or distinguishable to the human eye.

Face culling was performed at each level in the pyramid as the number of faces evaluated grew in accordance to diminishing node sizes. The times to perform the various face culling evaluations at each descent of the *Leaf Pyramid* was performed using the initial five test grids supplied by Sciencesoft and the results can be seen in Figure 7.10. Looking at these results, it can be seen that even with the largest grids, containing over 25 million cells (*Grid 36*) face culling evaluations still only took a few seconds to perform. In this research the face culling was only applied to the *Leaf Pyramid* as the *Tree Pyramid* contained too few node faces, to merit the computational overhead of performing these evaluations and the GPU could easily cope with the scaled down number of vertex positions, but could easily be applied to the *Leaf Pyramid* if required, such as with grids containing hundreds of millions of cells.

It is common practice when loading a grid into memory to also load a list of active faces, a single pass of the grid can be used to generate a small file containing flags, these indicate the visible status of cell faces, saved on disk and re-loaded the next time the grid is loaded. Only when the grid is manipulated to some extent, such as when the grid is *clipped,* do cell faces require re-evaluation. This practice could also be adopted using this hierarchical pyramid scaling where a small file using single bits could be used to indicate the visible status of each node and cell face at each pyramid level. Higher up levels would require a smaller list than the levels below due to larger leaf nodes containing more hidden cells and faces.

Many grids may have slow refresh rates resulting in jerky visual rotations of the grid model due to large cell numbers, such as with multi-million cell models. The low resolution models generated from the *tree pyramid* could be substituted for the full field model during grid rotations, eliminating these jerky visual effects as the low level models contain far fewer

vertices, requiring fewer rotation calculations, yielding quicker refresh rates.

On many occasions oil reservoir engineers are only concerned with small regions of the full field model (*region of interest*) perhaps only a small percentage of the total cells. Using the hierarchical pyramid visualisation styles developed in this thesis a full field model could be displayed at a low level of detail using the tree pyramid and the region of interest at the highest level of detail, using the leaf pyramid. For this reason a regional view algorithm (see section 7.3) was developed similar to Sciencesoft's S3sector software, this state-of-the-art application allows engineers to select a volume of the grid model for close analysis and refinement, where the remainder of the grid can be omitted allowing fast detailed evaluations and visualisations.

# Chapter 9

# Future Work

## 9.1 Rotation Refresh Rates

The *regions of interest* described in section 7.3 of this thesis, described how a selected volume cells can be displayed at the highest level of detail, but the surrounding grid drawn at lower levels of detail taken from higher levels in the pyramid. This would allow selected *regions of interest* to be further refined as the GPU could accommodate the overhead of increased vertices due to the reduced number of vertices used to define the smaller selected region; giving reservoir engineers greater accuracy and detail, where and when required.

The low level of detail models found at the bottom of the octree have much faster refresh rates than those containing models rendered at the highest level of detail due to possessing fewer vertices. At present reservoir engineers view models in their entirety, at high levels of detail, but the refresh rates can be slow, resulting in staggered visualisations as the GPUs buffer memory struggles to cope with having to re-drawn such vast numbers of vertices as each rotation requires the GPU to apply a transform function to every vertex. In order to alleviate the burden put on the buffer during grid rotations, the rendering could be swapped for a low-level detailed model only during rotations. The level selected would be such that there should be no dramatic perceivable difference as the model is rotated, except that it would be quicker. This would not instigate refilling vertex buffer as the low-level pyramid model has far fewer vertex positions and so both tables could reside on the buffer simultaneously.

## 9.2 Medical Imagery

Clinicians use a variety of digital image formats such as X-ray, Ultrasound, Computed tomography (CT) scans and Magnetic Resonance Imaging (MRI) (Anon, 2007; Robin, 2011) of human and animal anatomy to assist them in diagnosing medical conditions and for planning surgical procedures Hedgcock *et al.* (1993); X. *et al.* (2015). Batches of digital 2D slices (sometimes several hundred) generated from CT and MRI scans, normally 256 – 512 pixels long in both dimensions, can be stacked together to form 3D digital models (Zuiderveld & Viergever, 1992). Sometimes a process of *segmentation* is applied to these 3D models where selecting different threshold parameters, segregates the image into *regions of interest* (Zhao *et al.* , 2009). This can help surgeons to distinguish between various aspects of the anatomy being studied by isolating those sections of the model defined by the threshold parameters applied (Franek *et al.* , 2011). The now filtered model is easier to assess due to it being less cluttered with redundant information removed, thus helping clinicians to better perform detailed analysis of the data, such as when measuring the growth of a lesion within the human body (Y. *et al.* , 2013). Sometimes clinicians require to see this *region of interest* highlighted within the original or collection of filtered models, a process known as *fusion* Li & Yang (2008), where the suspected tumour lies in relation to the patient's body can often determine what is the best or safest medical procedure to adopt (Franek *et al.* , 2011).

The basic concept of showing a large dataset at low resolution with a *region of interest* at high resolution can be applied to other fields of technology. The following example details how this technique could be applied to medical imagery and gives images obtained from Magnetic Resonance Imaging (MRI) scans supplied by Aleksandra Radjenovic, a senior lecturer in MRI at the Institute of Cardiovascular & Medical Sciences department at the University of Glasgow. The MRI image slices depict a lesion through a human liver. There were 72 slices in the set and each slice was 414 pixels wide by 242 pixels high. Figure 9.1 shows slice 47 where the liver region has been highlighted by the larger red line and the lesion with a smaller one.

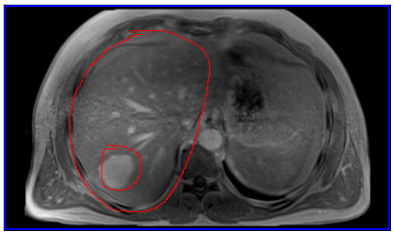


Figure 9.1: MRI image slice 47 depicting a lesion through a human liver

A 3D model can be generated by stacking these images in a similar manner to stacked oil

reservoir grid slices. Compressing the 3D image stack into homogeneous grey-scale values using octree compression techniques would allow for lossless storage with a lower memory overhead.

Figure 9.2shows a screenshot of the 3D stack model displayed at the top of the *Leaf Pyramid* (leaf nodes with face culling) where leaf nodes were solely based on 'on' and 'off' bits similar to Sciencesoft's ACTNUM array where black was 'off' and everything else, 'on'.

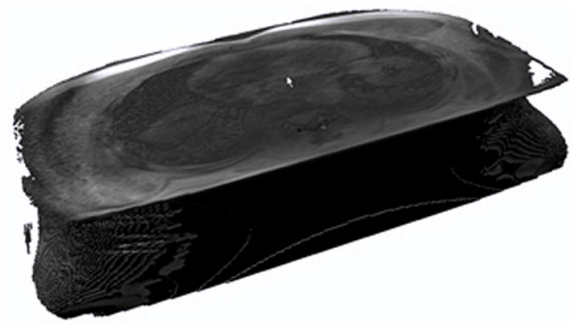


Figure 9.2: 3D model of the MRI dataset compressed using octree compression techniques

Figure 9.3 illustrates how only the skin of the model requires rendering as hidden cell faces are culled, leaving a hollow structure.

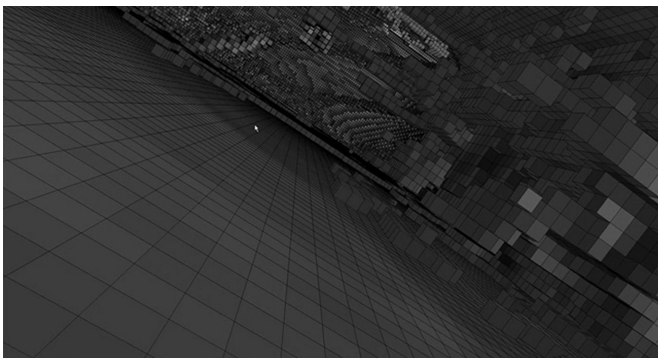


Figure 9.3: Inner view of the 3D medical model

The image dataset contains a lesion in a human liver, as illustrated in Figure 9.1. With medical imagery accuracy is of the utmost importance and therefore it is vital that resulting models should contain no loss of information as this would impact on any measuring or growth analysis. Applying the same *region of interest* concept as those detailed in section 7.3, the suspected lesion can be displayed at the highest level of resolution, shown in the left-hand image in Figure 9.4 with no loss of information, but displayed within the low resolution model of complete dataset in a *segmentation* and *fusion* fashion. This is illustrated in the right-hand

image in Figure 9.4 where the lesion is displayed within a wire-frame leaf node representation of the dataset and given a different colour for clearer distinction.

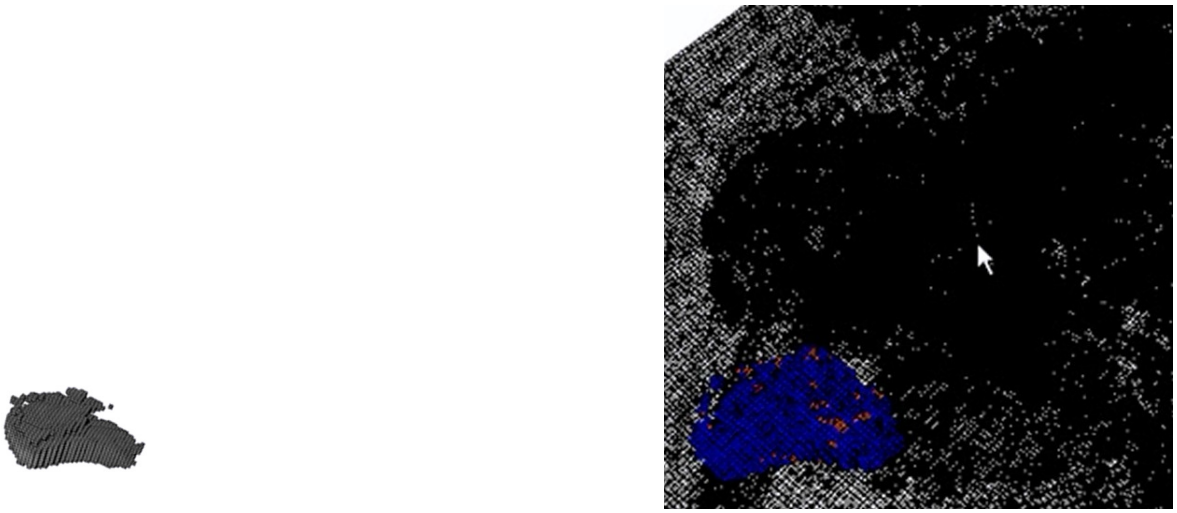


Figure 9.4: Left-hand image depicts the lesion in the human liver, the right-hand image depicts the lesion within a 3D wire-frame of the complete image stack and coloured for better distinction

# Chapter 10

# Appendix

## 10.1 Appendix A

The following table in this sub-section shows the resulting tree structures generated from applying quadtree compression techniques to the two images used in section 2.2.1 of this thesis.

| Tree Level | Node Type | Image (a) | Image (b) | Leaf node dimensions |
|---|---|---|---|---|
| Level 0 | Number of header nodes (root) | 1 | 1 | 256 x 256 |
| | Number of leaf nodes | 0 | 0 | |
| Level 1 | Number of header nodes | 4 | 4 | 128 x 128 |
| | Number of leaf nodes | 0 | 0 | |
| Level 2 | Number of header nodes | 8 | 16 | 64 x 64 |
| | Number of leaf nodes | 8 | 0 | |
| Level 3 | Number of header nodes | 9 | 62 | 32 x 32 |
| | Number of leaf nodes | 23 | 2 | |
| Level 4 | Number of header nodes | 8 | 240 | 16 x 16 |
| | Number of leaf nodes | 28 | 8 | |
| Level 5 | Number of header nodes | 0 | 956 | 8 x 8 |
| | Number of leaf nodes | 32 | 4 | |
| Level 6 | Number of header nodes | - | 3800 | 4 x 4 |
| | Number of leaf nodes | - | 24 | |
| Level 7 | Number of header nodes | - | 12111 | 2 x 2 |
| | Number of leaf nodes | - | 3089 | |
| Level 8 | Number of header nodes | - | 0 | 1 x 1 |
| | Number of leaf nodes | - | 48444 | |

Table 10.1: Tree structures of Figure 2.3

## 10.2 Appendix B

Figure 10.1 illustrates the demo grid with elongated cells, surfaces roughened and a colour scheme applied as a set of mock property values which vary throughout its entirety.

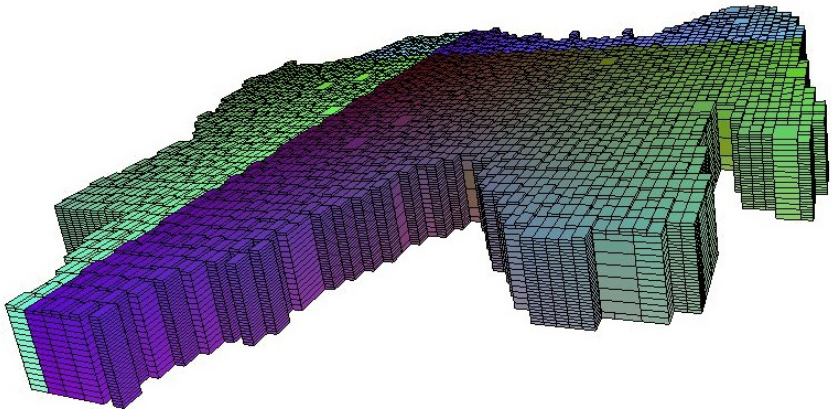


Figure 10.1: A screenshot of the demo grid with overlaid mock property values

Figure 10.2 shows a screenshot of more flattened variation of the demo grid dissected by a fault.

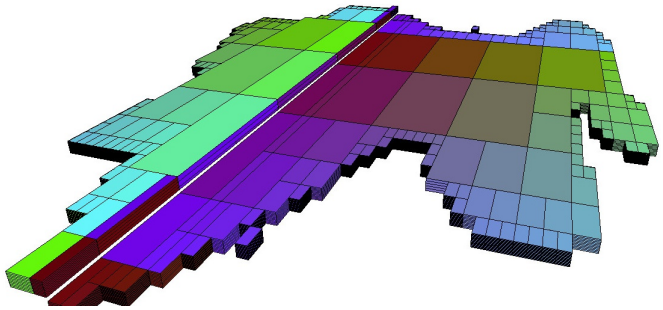


Figure 10.2: Screenshot at leaf level 0 of a faulted version of the initial test grid

## 10.3 Appendix C

The following sub-section of this thesis details the various algorithms used in this thesis.

**Algorithm 10.1** Tree Construction Algorithm (discussed in section 4.6)

```
Define a variable edge = power-of-2 size
Create a C# dynamic array (list<>) "nodeStructs" structs:
Create a linear list of integers:  to hold active cell indices
Create a pointer to the start of the list, "listPtr"
CreateOctreeStructs(0, 0, 0, grid width, grid breadth, grid depth, edge)
If edge = 1 and input array value does not = 0
  Add an active leaf node to the end of the list
  flag = 0 and pointer = 1
If edge = 1 and input array value < 1
  If at least one node cell is within original grid dimensions
    flag = 0 and pointer = 0
  Else Create 8 new nodes
    Calculate node volume using node origin position and edge / 2
  If all 8 nodes are the same and (flag = 0; pointer = 1)
    Create an active node
    flag = 0 and Pointer = origin position cell's iActive value
  ElseIf all 8 childStructs are not the same
    Create a header node
    Pointer = position of last (SW_1) child node
    Set its flag value to indicate the active status its 8 child nodes
    Count the number of on bits in flag and add to listPtr
    Pointer = listPtr
Convert the list to an array of structs (structArray)
Convert the compressed iActive list to an array (compIndArray)
```

**Algorithm 10.2** 3D Bitwise Searching Algorithm (discussed in section 4.13)

```
Define a variable Ptr, points to the root node
Define a variable edge = power-of-2 size
Define a variable mask = to the number of bits to store the power-of-2 size
Define a variables to the original grid dimensions:  X, Y and Z
Define starting root node origin position variables:  xO = 0, yO = 0, zO = 0
Define variables to store the target cell's three co-ordinates, xS, yS, zS
SearchOctreeStructs(structArray, Ptr, edge, mask, xO, yO, zO, X, Y, Z, xS, yS, zS)
  If (structArray[Ptr].Flag = 0)
    Return Active Status:  true
  Else create a direction byte flag variable:  dirFlag = 128
    If (x-axis flag is 'on') then shift dirFlag to the right 1 position.
    If (y-axis flag is 'on') then shift dirFlag to the right 2 positions.
    If (z-axis flag is 'on') then shift dirFlag to the right 4 positions.
    divide mask by 2
    divide edge by 2
    If (structArray[Ptr].Flag & dirFlag does not = 0)
      For Each active child node
        activeFlagBits = number of flag 'on' bits before the dirFlag bit
        Ptr= structArray[Ptr].pointer + activeFlagBits
        SearchOctreeStructs(structArray, Ptr, edge, mask, xO, yO, zO, X, Y, Z, xS, yS, zS)
    Else Return Active Status:  False
```

**Algorithm 10.3** Generate Octree Algorithm (discussed in section 2.2.3)

```
Define a variable edge = power-of-2 size
CreateOctree(x =0, y=0, z=0, edge)
  If (edge = 1)
    If (all nodes are identical
      If (payload = 1) then create active leaf node
      If (payload = 0 and (x, y, z) are within grid dimensions) then create inactive leaf node
    Else create 8 header nodes to point to child nodes
  Else
    divide edge by 2.
    Create 8 new child nodes (CreateOctree(x, y, z, edge)
```

**Algorithm 10.4** Enumerator Algorithm Part I (discussed in Chapter 4)

```
Pass in the structArray and grid dimensions into the enumerator class
Define a stack to hold the structArray's nodeStructs
Define an 'indexer' to the first position in the stack
PUSH the first nodeStruct at the index position onto the stack
Move the indexer to the next position in the collection
While another nodeStructs exists, point the indexer to it
  POP the nodeStruct off the stack
    If (nodeStruct represents a header node)
      Set ActiveBitCounter to the number of 'on' bits
      Set the x, y and z co-ordinate positions from the leaf node
      Set the length of the leaf node edge.
      Create nodeStructs for each child leaf node
      If: (nodeStruct = inactive leaf node and within grid boundaries)
        Create a leafStruct containing all leaf node values(iActive = -1)
        PUSH: leafStruct onto the stack
      Else:
        Create a leafStruct containing all active leaf node values
        PUSH: leafStruct onto the stack
  Peek: if nodeStruct is a leafStruct
    Send: leafStruct to Enumerator part II
```

**Algorithm 10.5** Enumerator Algorithm Part II (discussed in Chapter 4)

```
If the leafStruct is a header value break
Else create a cellStruct for this leafStruct
  cellStruct (x, y, z) positions = cell (x, y, z) co-ordinates
  cellStruct iCell value = position of cell in list
  If: leafStruct pointer is -1 then the cell is an inactive cell
    Set active value to zero and return cellStruct
  Else: leafStruct is an active cell
    cellStruct iActive value is fetched from compIndArray
    Return cellStruct
```

**Algorithm 10.6** Basic Recursive structArray Traversal Algorithm (see in section 4.16)

```
Define grid dimension (N_X, N_Y, N_Z) and edge (power-of-two size)
Define origin positions (oX=0, oY=0, oZ=0)and Ptr (points to root)
Define variable mask = the number of bits required to store edge
SearchOctree(structArray, Ptr, edge, mask, oX, oY, oZ)
  If (structArray[Ptr].Flag = 0) //Active leaf node
    Leaf node origin position = (oX, oY, oZ)
    activeLeafNodeFunction(oX, oY, oZ, iCell, iActive, edge)
  Else: create a direction byte flag variable:  dirFlag = 128
    If: (x-axis bit does not = 0) then shift dirFlag to the right 1 position
    If: (y-axis bit does not = 0) then shift dirFlag to the right 2 positions
    If: (z-axis bit does not = 0) then shift dirFlag to the right 4 positions
    divide mask by 2.
    divide= edge by 2
    If: (structArray[Ptr].Flag & dirFlag does not = 0)
      For each: ('on' bit in dirFlag)
        activeBitCounter = next 'on' bit in dirflag
        Ptr = structArray[Ptr].pointer + activeBitCounter - 1
        Create new child node:  SearchOctree(structArray, Ptr, edge, mask, newX, newY, newZ
    Else:  Return False.
```

**Algorithm 10.7** CullFaces Algorithm (discussed in section 7.1)

```
Passed in variable oEdgelen = leaf node edge length
Passed in variables (oX = 0, oY= 0, oZ= 0) are the origin co-ordinate positions of the leaf
node
Passed in variable edgeTarget is the octant node edge length derived from sub-dividing the
leaf node
Passed in variable vertexList is the list of vertices to be sent to the GPU
Passed in variables (iOrigin = 0, jOrigin = 0, kOrigin = 0) hold the origin positions of
octants
Define three variable Nx, Ny, Nz = grid dimensions
Define a variable called pointer, points to the end of the structArray (flattened octree -
array of structs)
CullFaces(oEdgelen, oX, oY, oZ, iOrigin, jOrigin, kOrigin, stop, vertexList)
  If (node is a single cell)
    Create new 2D array called Cage[27, 2] (26 nearest neighbour cells to target cell)
    call UpdateKnownCageValuesFromBoundaryChecks(See Algorithm10.16)
    Call PopulateCageWithinLeafNode (See Algorithm 10.18)
    Define a boolean called drawFace
    Foreach (i face cell)
      If (Cage[i, 1] < 0) (at least one cell is not hidden and so the face is rendered)
        populate the vertex array with the 4 face vertices to be rendered
      Else
        drawFace = MatchedVertices(Cage, i) (See Algorithm 10.20)
        If (drawFace = true)
          populate the vertex array with the 4 face vertices to be rendered
  Else
    Define a variable called octantPos
    octantPos = GenerateOctantPos (See Algorithm 10.11)
    Switch (octantPos)
      edgeTarget = octant edge length.
      Call PopulateCageOutsideLeafNode (See Algorithm 10.19)
      If (a fault exists)
        Call FaultAnalysis (See Algorithm 10.21)
      Foreach (potentially visible node face i, other than those dissected by faults)
          Define an array (faceArray) equal to the number of cells on the node face
          If (any faceArray[i]= 0) (inactive cell found)
            populate the vertex array with the 4 face vertices to be rendered
          Else
            drawFace = MatchedVertices(Cage, i) (See Algorithm 10.20)
            If (drawFace = true)
              populate the vertex array with the 4 face vertices to be rendered
```

**Algorithm 10.8** Grid Boundary Position Algorithm (discussed in section 7.1)

```
Passed in variables (iOrigin, jOrigin, kOrigin) are the logical i, j and k co-ordinates of the cell or node's origin position
Define variables (Nx, Ny, Nz) = (grid width, grid height, grid depth)
Define variable gridPos = 0
GridBoundaryCheck(iOrigin, jOrigin, kOrigin, Nx, Ny, Nz)
    If (xOrigin = zero) then gridPos = 1
    If (yOrigin = zero) then gridPos = gridPos + 2
    If (zOrigin = zero) then gridPos = gridPos + 4
    If (xOrigin = Nx -1) then gridPos = gridPos + 8
    If (yOrigin = Ny -1) then gridPos = gridPos + 16
    If (zOrigin = Nz -1) then gridPos = gridPos + 32
    Return gridPos
```

**Algorithm 10.9** Tree Pyramid Algorithm (discussed in section 6.5)

```
Define node origin position variables xO = 0, yO = 0, zO = 0
Define variable oEdgelen = power-of-two size = leaf node edge length
Define variable pointer, points to root node
Define variable stop = maximum edge length of nodes in the pyramid at each level
Define variable arrayPtr, points to the next free position in vertexArray
Define variable vertexPtr, points to the natural vertex_0 position of node, Figure 6.14
GenerateTreePyramid(structArray, pointer, xO, yO, zO, oEdgelen, stop, vertexArray)
  If (pointer < 0) then flag = -1 (inactive leafnode)
  Else
    flag = tree[pointer].pointer
    If (flag < 1) (active leaf node)
      vertexPtr = ((zO * Nx * Ny) + (yO * _Nx) + xO) * 8
      call PopulateVertexArray(see Alg 10.10)
    Else
      If (structArray[pointer].flag does not = 0) (header node has active child nodes)
        Foreach active header node
          oEdgelen = oEdgelen / 2
          If (oEdgelen = stop)
            call PopulateVertexArray(see Alg 10.10)
          Else
            call GenerateTreePyramid
```

**Algorithm 10.10** PopulateVertexArray Algorithm (discussed in section 6.5)

```
PopulateVertexArray(oX, oY, oZ, oEdgelen, vertexPtr, vertexArray, arrayPtr)
  If (oEdgelen = 1):  (a single cell node)
    Add the 8 vertex positions of the node to the vertex array (vertexArray)
  Else :  (a multi-cell node)
    Define variables (xEdgelen, yEdgelen, zEdgelen) holds cropped edge length of a node
     Increment the node starting (x, y, z) axis values with the node length -1
     Crop any node (x, y, z) origin position to grid boundary if > grid size
    Else
       Add the 8 vertex positions of the node to the vertex array (vertexArray)
```

**Algorithm 10.11** GenerateOctantPos Algorithm (discussed in section 10.3)

```
Define octantPos
octantPos = GenerateOctantPos(oEdgelen, oX, oY, oZ, iOrigin, jOrigin, kOrigin, edgeTarget).
  If oEdgelen does not = edgeTarget then octantPos = 0
    If (iOrigin = oX) then octantPos = 1
    If (jOrigin = oY) then octantPos = octantPos + 2
    If (kOrigin = oZ) then octantPos = octantPos + 4
    If ((iOrigin = oX + oEdgelen - 1) or (iOrigin + edgeTarget = oX + oEdgelen)) then add 8 to
octantPos
    If ((jOrigin == oY + oEdgelen - 1) or (jOrigin + edgeTarget = oY + oEdgelen)) then add 16
to octantPos
    If((kOrigin == oZ + oEdgelen - 1) or (kOrigin + edgeTarget = oZ + oEdgelen)) then add 32
to octantPos
    return octantPos
  Else
    return -1
```

**Algorithm 10.12** GenerateOctants Algorithm (discussed in section 10.3 for active cell faces)

```
Leaf level = 0 = pass.
Define a variable called oEdgelen = power-of-two size = leaf node edge length
Define three variables equal to the origin position of a leaf node (oX, oY, oZ)
Define variable stop = oEdgelen / (one shifted pass number of bits to the left)// {1, 2 ,4,
etc}
Define a variable called pointer which points to the last position in the structArray.
Define variables to hold octant origin positions (iOrigin, jOrigin and kOrigin)
GenerateOctants(oX, oY, oZ, oEdgelen, stop, pass, vertexList)
  If (structArray[structPtr].Flag = 0) = active leaf node
    for each pass
        If (oEdgelen = 1)
          Call CullFaces( see algorithm 10.7)
        Else
          If (oEdgelen = stop)
            Call CullFaces( see algorithm 10.7)
          Else
            stop = oEdgelen / 2
            call SubdivideLeafNode(see Algorithm 10.13 for active cell faces
    Else
      Foreach active header node
        GenerateOctants(oX, oY, oZ, oEdgelen, stop, pass, vertexList)
```

**Algorithm 10.13** SubdivideLeafNode Algorithm (Called from Algorithm 10.12)

```
Passed in variable oEdgelen = leaf node edge length.
Passed in variables (oX, oY, oZ) are the origin co-ordinate positions of the leaf node
Passed in variable stop is required octant node edge length from sub-dividing nodes into 8
Passed in variable vertexList is the list of vertices to be sent to the GPU
Define three variables which hold the origin positions of octants (iOrigin = oX, jOrigin =
oY, kOrigin = oZ)
Passed in variable edgeTarget = oEdgelen
SubdivideLeafNode(oEdgelen, oX, oY, oX, Origin, jOrigin, kOrigin, edgeTarget, stop,
vertexList)
  If (edge = stop)
    Call CullFaces(see Algorithm 10.7)
  Else
    edgeTarget = edge / 2
    For each of active octant nodes
      Call SubdivideLeafNode with new octant origin positions and edgeTarget value
```

**Algorithm 10.14** EvaluateFace Algorithm (discussed in Chapter 7)

```
Define variable oEdgelen = power-of-two size = leaf node edge length
Define variables equal to leaf node origin positions (oX = 0, oY = 0, oZ = 0)
Define variable Face = 0, used to determine the face to be evaluated {0 - 5}
Create a 1D array called faceArray equals the number of cells on the nodes face
Define a variable called pointer which points to the last position in the structArray
Define a variable flag called face = false
Define variables which holds evaluated face origin positions (iSearch, jSearch, kSearch
=(0,0,0)
face = EvaluateFace(structArray, pointer, 0 , 0, 0, iSearch, jSearch, kSearch, oEdgelen,
edgeTarget, faceArray, Face)
  Switch (Face)
    0 =  Call EvaluateFaceA(Called from Algorithm 10.7)
    1 =  Call EvaluateFaceB(Called from Algorithm 10.7)
    2 =  Call EvaluateFaceC(Called from Algorithm 10.7)
    3 =  Call EvaluateFaceD(Called from Algorithm 10.7)
    4 =  Call EvaluateFaceE(Called from Algorithm 10.7)
    5 =  Call EvaluateFaceF(Called from Algorithm 10.7)
  EvaluateFaceAInSingleLeafNode(See Algorithm 10.15)
    Traverse tree defined by (iSearch, jSearch, kSearch) using the basic recursive function
    If (all face cells land within a single active leaf node)
      return false// all face cells are hidden in logical space
    Else
      edgeTarget = octant node edge length.
      Define variable x = face side length in first axis direction
      Define variable y = face side length in second axis direction
      Define variable facePosCounter = 0
      Define variable flag called face = false
      For (all cells in face)
        If (faceArray[facePosCounter] does not = 0)
          flag = EvaluateFaceAoutsideSingleLeafNode(See Algorithm 10.15)
        If (faceArray[facePosCounter] does not = 0)
          return true.  (at least one cell is not hidden and face should be drawn)
```

**Algorithm 10.15** EvaluateFaceAoutsideSingleLeafNode Algorithm (Called from Algorithm 10.14 showing active node face cells)

```
oEdgelen = leaf node edge length
Origin position of leaf node = (oX, oY, oZ)
faceArray equals the number of cells on the nodes face
pointer which points to the last position in the structArray
facePosCounter = position in double-for-loop
flag = false
(iTarget, jTarget, kTarget) = cells opposite octant face to be evaluated
EvaluateFaceAoutsideSingleLeafNode(structArray, pointer, oX, oY, oZ, iSearch, jSearch,
kSearch, oEdgelen, edgeTarget, iTarget, jTarget, kTarget, facePosCounter)
  Traverse tree in direction to (iTarget, jTarget, kTarget) opposite face cells using the
basic recursive function (see Algorithm 10.6)
    If (any target cell lies in an inactive leaf node)
      return true (at least one cell is not hidden and the face should be rendered)
    Else
      If (node face = a single cell)
        faceArray[facePosCounter] = 1
      Else
        For (all face cells not found yet)
          faceArrayPos = (z * edgeTarget) + facePosCounter + x
          If (faceArray[faceArrayPos,0] does not = 0 (still to search for cell)
              faceArray[faceArrayPos, 0] = 1 (cell has been searched)
```

**Algorithm 10.16** UpdateKnownCageValuesFromBoundaryChecks Algorithm (called from Algorithm 10.7)

```
Passed in variable iOrigin, jOrigin, kOrigin = the leaf nodes logical origin position
Passed in variables Nx, Ny, Nz = the grid dimensions
UpdateKnownCageValuesFromBoundaryChecks(Nx, Ny, Nz, iOrigin, jOrigin, kOrigin, Cage)
  Define a variable called gridPos = 0.
  gridPos = call GridBoundaryCheck(See Algoritm 10.8)
  switch: (gridPos)
    Foreach: (i cell to find in Cage)
      If: (i = exposed face)
        Cage[i, 0] = 1
        Cage[n, 1] = -1
```

**Algorithm 10.17** PopulateCage Algorithm (called from Algorithm 10.7)

```
Variables (iSearch, jSearch, kSearch) = the origin position of target cell
Variables (xO=0, yO=0, zO=0)
Variable pointer is a pointer to the end of the flattened out octree (structArray)
Passed in variable edge is the power-of-two size of the grid
PopulateCage(structArray, pointer, xO, yO, zO, iSearch, jSearch, kSearch, edge, Cage)
  If (node = leaf node)
    Call PopulateCageWithinLeafNode(See Algorithm 10.18)
  Else
    If (Node is node has active child nodes)
      For each (active child node)
        Call PopulateCageOutsideLeafNode (See Algorithm 10.19)
```

**Algorithm 10.18** PopulateCageWithinLeafNode Algorithm (Called from Algorithm 10.17)

```
PopulateCageWithinLeafNode(structArray, pointer, xO, yO, zO, iOrigin, jOrigin, kOrigin, edge,
Cage)
  Define a variable n = leafnode pointer
  For: ( All required cage positions (max = 27))
     Cage[cagePos, 0] = 1 (indicates found)
     Cage[cagePos, 1] = active cell value
```

**Algorithm 10.19** PopulateCageOutsideLeafNode Algorithm (Called from Algorithm 10.17)

```
PopulateCageOutsideLeafNode(structArray, pointer, xO, yO, zO, iSearch, jSearch, kSearch,
edge, Cage, iTarget, jTarget, kTarget, cagePos)
  If (pointer < 0)
    activeValue = -1 (inactive cell)
  Else
    activeValue = active cell value
  If (edge = 1) (cell found)
    Cage[cagePos,0] = 1
    Cage[cagePos,1] = activeValue
  For each (cell still required to be found)
    call PopulateCageOutsideLeafNode (searches start from header node in tree)
```

**Algorithm 10.20** MatchedVertices (Called from Algorithm 10.7)

```
MatchedVertices(Cage, cagePos)
  Create an array to hold the 8 vertex positions, 4 for each cell face being evaluated
  Compare each of the opposite face cell vertices in turn
    If (any vertex pair does not match) then return false and the face will be rendered
    If (all vertex pairs match) then return true, the faces are hidden and perfectly match
```

**Algorithm 10.21** Fault Analysis Algorithm (discussed in section 7.2)

```
  If (a fault exists)
    If (the fault plane runs through the x-axis and z-axis):  (vertical plane)
    Else If (the fault plane runs through the y-axis and z-axis):  (vertical plane)
    Else (the fault plane runs through the x-axis and y-axis):  (horizontal plane)
      For each (node dissected by the fault plane)
        Split the node into 2 at the point of the dissection and render dissected faces
```

# Bibliography


Aarnes, Jorg E., Kippe, Vegard, & Lie, Knut-Andreas. 2005. Mixed multiscale finite elements and streamline methods for reservoir simulation of large geomodels. *Advances in Water Resources*, **28**(3), 257 – 271.

Adams, Ernest. 2014. *Fundamentals of Shooter Game Design*. Pearson Education.

Adamson, G., Crick, M., Gane, B., Gurpinar, O., Hardiman, J., & Ponting, D. 1996. Simulation throughout the life of a reservoir. *Oilfield Review*, **8**(2), 16 – 27.

Adelson, E.H., Anderson, C.H., Bergen, J.R., Burt, P.J., & Ogden, J.M. 1984. Pyramid methods in image processing. *RCA engineer*, **29**(6), 33 – 41.

Ahearn, Luke. 2014. *3D game textures: create professional game art using photoshop*. CRC Press.

Ahmed, Tarek. 2010. Fundamentals of Rock Properties. *Chap. 4, pages 189 – 287 of: Reservoir Engineering Handbook*, fourth edition edn. Boston: Gulf Professional Publishing.

Ahuja, Narendra, & Nash, Charles. 1984. Octree representations of moving objects. *Computer Vision, Graphics, and Image Processing*, **26**(2), 207 – 216.

Alsharhan, A.S., & Whittle, G.L. 1995. Sedimentary-diagenetic interpretation and reservoir characteristics of the Middle Jurassic (Araej Formation) in the southern Arabian Gulf. *Marine and petroleum geology*, **12**, 615 – 628.

Anderson, Theodore W, & Goodman, Leo A. 1957. Statistical inference about Markov chains. *The Annals of Mathematical Statistics*, **28**, 89 – 110.

Angel, Edward, & Dave, Shreiner. 2012. *Interactive Computer Graphics A Top-down Approach With Shader-based OpenGL*. 6th edn. Pearson.

Anon. 2007. CT Scans. *Pages 505–505 of:* Schmidt, Robert F., & Willis, William D. (eds), *Encyclopedia of Pain*. Springer Berlin Heidelberg.

Ayala, Dolors, Brunet, Pere, Juan, R, & Navazo, Isabel. 1985. Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics (TOG)*, **4**(1), 41 – 59.

Azarpour, Abbas, Suhaimi, Suardi, Zahedi, Gholamreza, & Bahadori, Alireza. 2013. A Review on the Drawbacks of Renewable Energy as a Promising Energy Source of the Future. *Arabian Journal for Science and Engineering*, **38**(2), 317 – 328.

Barenblatt, G. I., Zheltov, Iu P., & Kochina, I. N. 1960. Basic concepts in the theory of seepage of homogeneous liquids in fissured rocks [strata]. *Journal of Applied Mathematics and Mechanics*, **24**(5), 1286 – 1303.

Basharin, GP. 1959. On a statistical estimate for the entropy of a sequence of independent random variables. *Theory of Probability & Its Applications*, **4**(3), 333 – 336.

Beach, Alastair, Welborn, Alastair I., Brockbank, Paul J., & E., McCallum Jean. 1999. Reservoir damage around faults; outcrops examples from the Suez Rift. *Petroleum Geoscience*, **5**(2), 109 – 106.

Blackwell, David. 1957. The entropy of functions of finite-state Markov chains. *Pages 13 – 20 of: Trans. First Prague Conf. Information Thoery, Statistical Decision Functions, Random Processes*.

Blunt, Martin J. 2001. Flow in porous media – pore-network models and multiphase flow. *Current Opinion in Colloid & Interface Science*, **6**(3), 197 – 207.

Bonet, Javier, & Peraire, Jaime. 1991. An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems. *International Journal for Numerical Methods in Engineering*, **31**(1), 1 – 17.

Bourke, P.1. 1993. A Beginners Guide to Bitmaps. *Geometric data formats. Centre for Astrophysics and Supercomputing, University Swinburne*, **1**, Total pages.

Burt, P., & Adelson, E. 1983. The Laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, **31**(4), 532 – 540.

Chen, Zhangxin. 2007. *The Black Oil Model and Numerical Solution*. SIAM. Chap. 6, pages 103 – 129.

Chen, Zhangxin, Huan, Guanren, & Li, Baoyan. 2004. An Improved IMPES Method for Two-Phase Flow in Porous Media. *Transport in Porous Media*, **54**(3), 361 – 376.

Cline, D., & Egbert, P.K. 2001. Terrain decimation through Quadtree Morphing. *Visualization and Computer Graphics, IEEE Transactions on*, **7**(1), 62–69.

Clunie, D.A. 2000. Lossless compression of grayscale medical images-effectiveness of traditional and state of the art approaches. *Pages 74 – 84 of: SPIE Medical Imaging*, vol. 3980. Citeseer.

Cockshott, W. Paul, Tao, Yegang, Ao, Gang, Balch, Peter, Briones, Ana M., & Daly, Craig. 2003. Confocal microscopic image sequence compression using vector quantization and three-dimensional pyramid. *Scanning*, **25**(5), 247 – 256.

Council, National Research. 1996. *Rock fractures and fluid flow: contemporary understanding and applications*. Natl Academy Pr. (US).

Cover, Thomas M, Thomas, Joy A, & Kieffer, John. 1994. Elements of information theory. *SIAM Review*, **36**(3), 509 – 510.

De Floriani, Leila, & Puppo, Enrico. 1995. Hierarchical Triangulation for Multiresolution Surface Description. *ACM Trans. Graph.*, **14**(4), 363–411.

Donnez, Pierre. 2007. *Essentials of reservoir engineering*. Technip.

Durlofsky, L.J. 1991. Numerical calculation of equivalent grid block permeability tensors for heterogeneous porous media. *Water Resour. Res*, **27**(5), 699 – 708.

Dyer, Charles R., Rosenfeld, Azriel, & Samet, Hanan. 1980. Region representation: boundary codes from quadtrees. *Commun. ACM*, **23**(3), 171 – 179.

Edwards, A. David., Hunasekera, Dayal., Morris, Jonathan., Shaw, Gareth., Shaw, Kevin., Walsh, Dominic., Fjerstad, Paul., Kikani, Jitendra., Franco, Jessica., Hoang, Viet., & Quettier, Lisette. 2011. Reservoir Simulation: Keeping Pace with Oilfield complexity. *Oil Field Review*, **23**, 4 – 15.

Egger, Karin, Geier, Bettina, & Muhar, Andreas. 2002. 3D-visualization-systems for landscape planning: concepts and integration into the workflow of planning practice. *Trends in GIS and Virtualization in Environmental Planning and Design. Proc. at Anhalt University of Applied Sciences.–Wichmann, Heidelberg*, 154–161.

Ellis, C. S. 1980. Concurrent Search and Insertion in AVL Trees. *Computers, IEEE Transactions on*, **C – 29**(9), 811 – 817.

Elsharkawy, Adel M. 2003. An empirical model for estimating the saturation pressures of crude oils. *Journal of Petroleum Science and Engineering*, **38**(1 – 2), 57 – 77.

Emmanuel, M Rohinton, & Baker, Keith. 2012. *Carbon management in the built environment*. Routledge.

Escobar, Jose C., Lora, Electo S., Venturini, Osvaldo J., Yanez, Edgar E., Castillo, Edgar F., & Almazan, Oscar. 2009. Biofuels: Environment, technology and food security. *Renewable and Sustainable Energy Reviews*, **13**(6 – 7), 1275 – 1287.

Facelli, JosÃ©M., & Pickett, Steward T.A. 1990. Markovian chains and the role of history in succession. *Trends in Ecology & Evolution*, **5**(1), 27 – 30.

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

Fanchi, John R. 2006b. *Conceptual Reservoir Scales*. Burlington: Gulf Professional Publishing. doi: 10.1016/B978-075067933-6/50014-3. Chap. 12, pages 210 – 232.

Favalora, G., Dorval, R.K., Hall, D.M., Giovinco, M., & Napoli, J. 2001. Volumetric three-dimensional display system with rasterization hardware. *Pages 227–235 of: Proc SPIE*, vol. 4297.

Filinski, Andrzej. 1994. Recursion from iteration. *LISP and Symbolic Computation*, **7**(1), 11 – 37.

Fjaer, E., Holt, R. M., Horsrud, P., Raaen, A. M., & Risnes, R. 2008. *Chapter 3 Geological aspects of petroleum related rock mechanics*. Vol. 53. Elsevier. Chap. 3, pages 103 – 133.

Fowler, M.C., Haskell, K.S., Horton, R.S., Kwok, T.Y.K., Narayanaswami, C., Schneider, B.O., Van Horn, M., & van Welzen, J.L. 2000 (Apr. 18). *Method and apparatus for deferred clipping of polygons*. US Patent 6,052,129.

Franek, Lucas, Abdala, DanielDuarte, Vega-Pons, Sandro, & Jiang, Xiaoyi. 2011. Image Segmentation Fusion Using General Ensemble Clustering Methods. *Pages 373–384 of:* Kimmel, Ron, Klette, Reinhard, & Sugimoto, Akihiro (eds), *Computer Vision ACCV 2010*. Lecture Notes in Computer Science, vol. 6495. Springer Berlin Heidelberg.

Friedman, DG. 1972. Insurance and the natural hazards. *The ASTIN Bulletin: International Journal for Actuarial Studies in Non-Life Insurance and Risk Theory*, **7**(1), 4 – 58.

Gerritsen, M. G., & Durlofsky, L. J. 2005. *Modeling fluid flow in oil reservoirs*. Annual Review of Fluid Mechanics, vol. 37. Palo Alto: Annual Reviews. Pages 211 – 238.

Gervautz, Michael, & Purgathofer, Werner. 1990. A simple method for color quantization: octree quantization. *Chap. A simple method for color quantization: octree quantization, pages 287 – 293 of:* Glassner, Andrew S. (ed), *Graphics gems*. San Diego, CA, USA: Academic Press Professional, Inc.

Ghavamnia, M.H., & Yang, X.D. 1995. Direct rendering of laplacian pyramid compressed volume data. *Page 192 of: Proceedings of the 6th conference on Visualization'95*. IEEE Computer Society.

Griffiths, Ian. 2012. *Programming C 5.0: Building Windows 8, Web, and Desktop Applications for the .NET 4.5 Framework*. O'Reilly Media, Inc.

Gutierrez, F., Bouleau, C., Howell, A., & Gehin, H. 2008. An Integrated, Innovative Solution To Optimize Hydrocarbon Production Through the Use of a Workflow Oriented Approach. *Page 10 of: SPE Gulf Coast Section Digital Energy Conference and Exhibition*. Society of Petroleum Engineers.

Hanke, S., Ottmann, Th, Soisalon-Soininen, E., Bongiovanni, Giancarlo, Bovet, Daniel, & Di Battista, Giuseppe. 1997. *Relaxed balanced red-black trees Algorithms and Complexity*. Lecture Notes in Computer Science, vol. 1203. Springer Berlin / Heidelberg. Pages 193 – 204.

Hay, RJ. 2003. Visualisation and presentation of three dimensional geoscience information. *In: Proceedings of 21st International Cartographic Conference*.

Hedgcock, M.W. Jr., Karshat, W., Levitt, T.S., & Vosky, Dmitry. 1993. Large Scale Feature Searches of Collections of Medical Imagery. *Pages 759 –759 of:* Lemke, HeinzU., Inamura, Kiyonari, Jaffe, C.Carl, & Felix, Roland (eds), *Computer Assisted Radiology*. Springer Berlin Heidelberg.

Hejlsberg, Anders, Wiltamuth, Scott, & Golde, Peter. 2006. *The C# programming language*. Addison-Wesley Professional.

Hoppe, Hugues. 1998. Efficient implementation of progressive meshes. *Computers & Graphics*, **22**(1), 27 – 36.

Huffman, David A, *et al.* . 1952. A method for the construction of minimum redundancy codes. *proc. IRE*, **40**(9), 1098 – 1101.

Hunter, Gregory M., & Steiglitz, K. 1979. Operations on Images Using Quad Trees. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **1**(2), 145 – 153.

Janoski, Guadalupe I, & Sung, Andrew H. 2001. Alternate methods in reservoir simulation. *Pages 253 – 262 of: Computational Science-ICCS 2001*. Springer.

Jones, James W, Hoogenboom, G, Porter, CH, Boote, KJ, Batchelor, WD, Hunt, LA, Wilkens, PW, Singh, U, Gijsman, AJ, & Ritchie, JT. 2003. The DSSAT cropping system model. *European journal of agronomy*, **18**(3), 235 – 265.

Kaick, Oliver Van, Fish, Noa, Kleiman, Yanir, Asafi, Shmuel, & Cohen-Or, Daniel. 2014. Shape Segmentation by Approximate Convexity Analysis. *ACM Transactions on Graphics (TOG)*, **34**(1), 4.

Kidner, DAVID B, & Smith, Derek H. 1997. Storage-efficient techniques for representing digital terrain models. *Innovations in GIS*, **4**, 25 – 41.

Knieser, Michael J, Wolff, Francis G, Papachristou, Chris A, Weyer, Daniel J, & McIntyre, David R. 2003. A technique for high ratio LZW compression. *Page 10116 of: Proceedings of the conference on Design, Automation and Test in Europe*, vol. 1. IEEE Computer Society.

Knoll, Aaron. 2006. A Survey of Octree Volume Rendering Methods. *Scientific Computing and Imaging Institute, University of Utah*, **1**, 9.

Knuth, D. E. 1971. Optimum binary search trees. *Acta Informatica*, **1**(1), 14 – 25.

Kokal, Sunil, & Al-Kaabi, ABDULAZIZ. 2010. Enhanced oil recovery: challenges & opportunities. *World Petroleum Council: Official Publication*, **78**, 64 – 69.

Krogstad, P.A., & Skare, P.E. 1995. Influence of a strong adverse pressure gradient on the turbulent structure in a boundary layer. *Physics of Fluids*, **7**(8), 2014 – 2024.

Ku, Se-Ju, & Yoo, Seung-Hoon. 2010. Willingness to pay for renewable energy investment in Korea: A choice experiment study. *Renewable and Sustainable Energy Reviews*, **14**(8), 2196 – 2201.

Kumar, Ashutosh, & Okuno, Ryosuke. 2014. Reservoir Oil Characterization for Compositional Simulation of Solvent Injection Processes. *Industrial & Engineering Chemistry Research*, **53**(1), 440 – 455.

Lee, Ho, Kozlowski, Eric, Lenker, Scott, & Jamin, Sugih. 2013. Multiplayer Game Cheating Prevention with pipelined lockstep Protocol. *Entertainment Computing: Technologies and Application*, **112**, 31.

Lelewer, Debra A., & Hirschberg, Daniel S. 1987. Data Compression. *ACM Computing Surveys*, **19**, 261 – 296.

Leonard, Nicole J. 2013 (June). *Does oil equal power in the Syrian conflict?* News letter.

Li, Ming, & Jinliang, Zhang. 2010. Application of neural network technique for logging fluid identification in low resistance reservoir. *Pages 163 – 166 of: Natural Computation (ICNC), 2010 Sixth International Conference on*, vol. 1.

Li, Shutao, & Yang, Bin. 2008. Multifocus image fusion using region segmentation and spatial frequency. *Image and Vision Computing*, **26**(7), 971 – 979.

Lin, Hsin-Chih, Wang, Ling-Ling, & Yang, Shi-Nine. 1996. Automatic determination of the spread parameter in Gaussian smoothing. *Pattern Recognition Letters*, **17**(12), 1247–1252.

Luebke, David, & Erikson, Carl. 1997. View-dependent Simplification of Arbitrary Polygonal Environments. *Pages 199–208 of: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Ma, Kwan-Liu, Stompel, Aleksander, Bielak, Jacobo, Ghattas, Omar, & Kim, Eui Joong. 2003. Visualizing Very Large-Scale Earthquake Simulations. *Pages 48 – 48 of: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. SC '03. New York, NY, USA: ACM.

Ma, Yuanle, & Chen, Zhangxin. 2004. Parallel computation for reservoir thermal simulation of multicomponent and multiphase fluid flow. *Journal of Computational Physics*, **201**(1), 224 – 237.

Manouvrier, Maude, Rukoz, Marta, & Jomier, GeneviÃšve. 2002. Quadtree representations for storage and manipulation of clusters of images. *Image and Vision Computing*, **20**(7), 513 – 527.

Mathews, G Jason. 1996. Evaluating data-compression algorithms. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, **21**(1), 50 – 53.

McCauley, R.D. 2000. Marine seismic surveys: a study of environmental implications. *Australasian Petroleum Production and Exploration Association*, **40**, 692 – 708.

McLimans, Roger K. 1987. The application of fluid inclusions to migration of oil and diagenesis in petroleum reservoirs. *Applied Geochemistry*, **2**(5 – 6), 585 – 603. doi: 10.1016/0883-2927(87)90011-4.

Meadows, Neil S. 1997. Characteristics of fault zones in sandstones form NW England: application to fault transmissibility. *In: Petroleum geology of the Irish Sea and adjacent areas*.

Meagher, D. 1982. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, **19**(2), 129 – 147.

Miano, J. 1999. *Compressed image file formats: Jpeg, png, gif, xbm, bmp*. Addison-Wesley.

Mohaghegh, Shahab, Arefi, Reza, Ameri, Sam, Aminiand, Khashayar, & Nutter, Roy. 1996. Petroleum reservoir characterization with the aid of artificial neural networks. *Journal of Petroleum Science and Engineering*, **16**(4), 263 – 274.

Nikolaevskiy, V. N. 2005. Theory of plastic sand flow with fluid pressure effect. *Journal of Engineering Mechanics-Asce*, **131**(9), 986 – 996.

Norgard, Jens-Peter. 2006. Revolutionising History Matching and Uncertainty Assessment. *Reservoir Management*, **1**, 34 – 35.

Northrop, P.S., & Timmer, R.S. 1995. *Method for producing low permeability reservoirs using steam*. US Patent 5,415,231.

Pandur, T., & Thiruvallur, TN. 2009. Images and its Compression Techniques Review. *Information Processing and Management*, **21**(3), 19.

Parkes, Graham. 2012. Nuclear power after Fukushima 2011: Buddhist and promethean perspectives. *Buddhist-Christian Studies*, **32**(1), 89–108.

Peidong, Zhang, Yanli, Yang, jin, Shi, Yonghong, Zheng, Lisheng, Wang, & Xinrong, Li. 2009. Opportunities and challenges for renewable energy policy in China. *Renewable and Sustainable Energy Reviews*, **13**(2), 439 – 449.

Phamdo, Nam. 2004 (4). *Theory of Data Compression*. http://www.data-compression.com/resources.shtml.

Porges, F. 2006. *Fundamentals of Rock Properties*. Burlington: Gulf Professional Publishing. doi: 10.1016/B978-075067972-5/50007-7. Pages 189 – 287.

Pouderoux, Joachim, Gonzato, Jean-Christophe, Tobor, Ireneusz, & Guitton, Pascal. 2004. Adaptive Hierarchical RBF Interpolation for Creating Smooth Digital Elevation Models. *Pages 232–240 of: Proceedings of the 12th Annual ACM International Workshop on Geographic Information Systems*. GIS '04. New York, NY, USA: ACM.

Publishing, OECD., & Agency, International Energy. 2007. *World energy outlook 2007: China and India insights*. Organisation for Economic Co-operation and Development.

Rabbani, M., & Jones, P.W. 1991. *Digital image compression techniques*. Vol. 7. Bellingham, Washington: SPIE-International Society for Optical Engineering.

Ramberg, Ivar B. 2008. *The Making of a Land: The Geology of Norway*. Geological Society.

Ringel, Marc. 2006. Fostering the use of renewable energies in the European Union: the race between feed-in tariffs and green certificates. *Renewable Energy*, **31**(1), 1 – 17.

Robin, Sekerak. 2011. CT Scan. *Pages 660–661 of:* Kreutzer, JeffreyS., DeLuca, John, & Caplan, Bruce (eds), *Encyclopedia of Clinical Neuropsychology*. Springer New York.

Roedder, E., & Bodnar, RJ. 1980. Geologic pressure determinations from fluid inclusion studies. *Annual review of earth and planetary sciences*, **8**, 263.

Royer, P., Auriault, J. L., & Boutin, C. 1996. Macroscopic modeling of double-porosity reservoirs. *Journal of Petroleum Science and Engineering*, **16**(4), 187 – 202.

Rrnyr, Alfred. 1961. On measures of entropy and information. *Pages 547 – 561 of: Fourth Berkeley Symposium on Mathematical Statistics and Probability*.

Rubin, Steven M., & Whitted, Turner. 1980. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *Pages 110 – 116 of: Computer Graphics*.

Sagan, Hans. 1994. *Space-filling curves*. Vol. 18. Springer-Verlag New York.

Samet, H. 1990. Hierarchical spatial data structures. *Pages 193–212 of: Proceedings of the first symposium on Design and implementation of large spatial databases*. SSD '90. New York, NY, USA: Springer-Verlag New York, Inc.

Samet, Hanan. 1980. Region representation: quadtrees from boundary codes. *Commun. ACM*, **23**(3), 163 – 170.

Samet, Hanan. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*, **16**(2), 187 – 260.

Samet, Hanan, & Kochut, Andrzej. 2002. Octree approximation and compression methods. *Pages 460 – 469 of: Proc. of the 1st Intl. Symp. on 3D Data Processing Visualization and Transmission*.

Samier, Pierre. 2011 (27/07/2011). *Reservoir Simulation In The Oil Industry*.

Schildt, Herbert. 2008. *C# 3.0: A Beginner's Guide*. 2 edn. McGraw-Hill Osborne Media.

Schulze-Riegert, R., Diab, A., & Haase, O. 2004. Streamline-Based History Matching With Application of Global Optimisation Techniques. *In: DGMK Spring Conference held in Celle, Germany., 29-30 April 2004.* Citeseer.

Seixas, RdB, Mediano, M, & Gattass, Marcelo. 1999. Efficient line-of-sight algorithms for real terrain data. *III Simpósio de Pesquisa Operacional e IV Simpósio de Logística da Marinha–SPOLM 1999*.

Shafiee, Shahriar, & Topal, Erkan. 2009. When will fossil fuel reserves be diminished? *Energy Policy*, **37**(1), 181 – 189.

Shannon, C.E., Weaver, W., Blahut, R.E., & Hajek, B. 1949. *The mathematical theory of communication*. Vol. 117. University of Illinois press Urbana.

Shannon, Claude E. 1951. Prediction and entropy of printed English. *Bell system technical journal*, **30**(1), 50 – 64.

Sharp, J. 2010. *Microsoft Visual C# 2010 Step by Step*. Step by Step Developer Series. Microsoft Press.

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

Skala, J., & Kolingerova, I. 2011. Dynamic hierarchical triangulation of a clustered data stream. *Computers & Geosciences*, **37**(8), 1092 – 1101.

Sohrabi, Mehran, Danesh, Ali, Tehrani, DabirH., & Jamiolahmady, Mahmoud. 2008. Microscopic Mechanisms of Oil Recovery By Near-Miscible Gas Injection. *Transport in Porous Media*, **72**(3), 351 – 367.

Solis, Daniel. 2010. *Illustrated C# 2010*. 1st edn. Berkely, CA, USA: Apress.

Song, Hyunjoo, Kim, Bohyoung, Lee, Bongshin, & Seo, Jinwook. 2010. A comparative evaluation on tree visualization methods for hierarchical structures with large fan-outs. *Pages 223 – 232 of: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. New York, NY, USA: ACM.

Spigler, Israel, & Maayan, Rafi. 1985. Storage and retrieval considerations of binary data bases. *Information Processing & Management*, **21**(3), 233 – 254.

Stabno, Michal, & Wrembel, Robert. 2009. RLH: Bitmap compression technique based on run-length and Huffman encoding. *Information Systems*, **34**(4 – 5), 400 – 414.

Steed, Paul. 2010. *Modeling a Character in 3ds Max*. Wordware Publishing, Inc.

Stegemeier, G.L., & Perry, G.E. 1992. *Method utilizing spot tracer injection and production induced transport for measurement of residual oil saturation*.

Sundar, Hari, Sampath, Rahul S., & Biros, George. 2008. Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel. *SIAM J. Sci. Comput.*, **30**(5), 2675–2708.

Tanguay-Carel, Matthieu. 2013 (May). *Our Energy Use In Numbers*. Webcast.

Tham, Min Jack. 1976 (October – 26). *Serially burning and pyrolyzing to produce shale oil from a subterranean oil shale*. US Patent 3,987,851.

Tremblay, Jonathan, & Verbrugge, Clark. 2013. Adaptive companions in FPS games. *FDG*, **13**, 229–236.

Uleberg, K., & Kleppe, J. 1996. Dual porosity, dual permeability formulation for fractured reservoir simulation. *In: Norwegian University of Science and Technology, Trondheim RUTH Seminar, Stavanger*. Norwegian University of Science and Technology.

Vicencio-Moreira, Rodrigo, Mandryk, Regan L., Gutwin, Carl, & Bateman, Scott. 2014. The Effectiveness (or Lack Thereof) of Aim-assist Techniques in First-person Shooter Games. *Pages 937–946 of: Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*. CHI '14. New York, NY, USA: ACM.

Waggoner, Ben. 2010. *Fundamentals of Compression*. Boston: Focal Press. Pages 35 – 60.

Waltz, Frederick M, & Miller, John WV. 1998. Efficient algorithm for gaussian blur using finite-state machines. *Pages 334–341 of: Photonics East (ISAM, VVDC, IEMB)*. International Society for Optics and Photonics.

Watt, David. A., & Brown, Deryck. F. 2001. *Java Collections: An Introduction to Abstract Data Types, Data Structures and Algorithms*. John Wiley & Sons.

Wiedenbeck, Susan. 1989. Learning iteration and recursion from examples. *International Journal of Man-Machine Studies*, **30**(1), 1 – 22.

Wood, Dr Lindsay. 2013 (August). *Sciencesoft Newsletter*.

Wood, Dr Lindsay. 2013a (April). *Advanced method cuts time and costs for EOR modeling p230-231*. Online magazine.

Würtenberger, Armin, Tautermann, Christofer S, & Hellebrand, Sybille. 2003. A hybrid Coding Strategy for Optimized Test Data Compression. *2003 IEEE International Test Conference (ITC)*, **1**(9), 451 – 459.

X., Zheng, and. Yang S., Kim T. M., & Kim, Y. 2015. CARS 2015 Computer Assisted Radiology and Surgery Proceedings of the 29th International Congress and Exhibition Barcelona, Spain, June 24 27th, 2015. *International Journal of Computer Assisted Radiology and Surgery*, **10**(1), 1–312.

Y., Tan, L.H., Schwartz, & B., Zhao. 2013. Segmentation of lung lesions on CT scans using watershed, active contours, and Markov random field. *In: Medical Physics*, vol. 40.4. PMC.

Yang, P.H., & Watson, A.T. 1988. Automatic history matching with variable-metric methods. *SPE reservoir engineering*, **3**(3), 995 – 1001.

Yau, Mann-May, & Srihari, Sargur N. 1983. A hierarchical data structure for multidimensional digital images. *Commun. ACM*, **26**(7), 504 – 515.

Yin, Xiang, Dafantsch, Ivo, & Gediga, Gafanther. 2011. Quadtree Representation and Compression of Spatial Data. *Pages 207–239 of:* Peters, James F., Skowron, Andrzej, Chan, Chien-Chung, Grzymala-Busse, Jerzy W., & Ziarko, Wojciech P. (eds), *Transactions on Rough Sets XIII*, vol. 6499. Springer Berlin Heidelberg.

Yu, Jinbiao, & Sun, Hongxia. 2009. 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*, **1**, 152 – 156.

Zaugg, Brian, & Egbert, ParrisK. 2001. Voxel Column Culling: Occlusion Culling for Large Terrain Models. *Pages 85–93 of:* Ebert, DavidS., Favre, JeanM., & Peikert, Ronald (eds), *Data Visualization 2001*. Eurographics. Springer Vienna.

Zhang, Cong, Bakshi, Amol, & Prasanna, Viktor K. 2008. Data component based management of reservoir simulation models. *Pages 386 – 392 of: Information Reuse and Integration, 2008. IRI 2008. IEEE International Conference on*.

Zhao, B., James, L. P., Moskowitz, C. S., Guo, P., Ginsberg, M. S., Lefkowitz, R. A., & Schwartz, L. H. 2009. Evaluating Variability in Tumor Measurements from Same-day Repeat CT Scans of Patients with Non-small Cell Lung Cancer. *Pages 263 – 274 of: Radiology*, vol. 10.

Zhigang, Fan, & de Queiroz, R. L. 2003. Identification of bitmap compression history: JPEG detection and quantizer estimation. *Image Processing, IEEE Transactions on*, **12**(2), 230 – 235.

Zou, Xukai, Ramamurthy, Byrav, & Magliveras, Spyros. 2005. *Tree Based Key Management Schemes Secure Group Communications over Data Networks*. Springer New York. Pages 49 – 89.

Zuiderveld, K.J., & Viergever, M.A. 1992. Visualization of Volumetric Medical Image Data. *Pages 363–385 of:* Dewilde, Patrick, & Vandewalle, Joos (eds), *Computer Systems and Software Engineering*. Springer US.