UNIVERSITY OF CALGARY

Tackling the Occlusion Problem in 3D Grids

by

Zamir Martins Filho

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER, 2013

Abstract

Occlusion is one of the major challenges in 3D grids. In this thesis we present different approaches to tackle the problem of occlusion in 3D grids. In order to visualize occluded objects some techniques remove a portion of the remaining grid (the occluding part) while others keep all the data exposing the occluded objects by splitting and shifting the remaining grid. In this work we propose the use of the *Cutaway Views* technique to improve the inspection and analysis of grid cells in corner-point grids, and we also present a system for creating interactive *Exploded View Diagrams* in generalized 3D grids.

The Cutaway View technique was used to improve the inspection and analysis of grid cells in corner-point grids. This approach places objects (or parts) of interest in focus by removing occluders. One important point is the notion of keeping the focused object in context within the whole scene. This is particularly challenging in cases where cells are tracked not only by spatial (geometrical and topological) information but also by their containing values, shifting the paradigm of traditional illustrative techniques. Here, we propose a first investigation on how to adapt the Cutaway approach to track grid cells either by spatial location or by value.

In regards to the Exploded View technique, the primary difference between our approach and existing research is that our technique neither requires geometrical information of the whole model nor any information regarding the relationship among model parts; instead our implementation depends on which grid cells are marked as primary objects, and which view angle to use. To achieve this, we introduce the *Explosion Tree*, a data structure closely related to a BSP tree, which is based on the relationship between grid cells and the gaze. We discuss the application of this technique to both synthetic and real data. In this thesis, regular grids are synthetically generated while non-regular grids are composed of real data. More specifically, the type of non-regular grid presented in this work is known as a *Corner-Point* grid which has been widely used for flow simulation and geological modeling.

Publications

Some of the materials, ideas and figures in this thesis has previously appeared in the following:

- Zamir Martins, Emilio Vital Brazil, Mario Costa Sousa, Felipe de Carvalho, Ricardo Marroquim. Cutaway Applied to Corner Point Models. In Workshop on Industry Applications (WGARI) in SIBGRAPI 2012 (XXV Conference on Graphics, Patterns and Images), August 2012, Ouro Preto, Brazil.

The following work was recently submitted, and is currently under review.

- Zamir Martins, Emilio Vital Brazil, Mario Costa Sousa. Exploded View Diagrams of 3D Grids. Eurographics 2014, April 2014, France.

Acknowledgements

I would like to thank my supervisor Dr. Mario Costa Sousa, for giving me the opportunity to study in Canada and for the support through these years. Many thanks to Emilio Vital Brazil, who gave me the opportunity to work as close as possible to computer graphics, who helped me in all my researches, to publish them and for all the unofficial lectures. My Master would be a whole different experience without you. I would also like to thank Nicole Sultanum who made my transition from Brazil to Canada so smooth and easy. Many big thanks to Sowmya Somanath, for suggestions, all endless crucial reviews and for being a good friend. In addition, I need to thank Ahmed, Charles, Elisa, Juliana and Ronan for keeping the lab atmosphere nice and enjoyable. I wish to thank my father, Zamir, and my sister, Caroline, for all their support and encouragement to follow my heart wherever it takes me. Lastly, a special thank to my wife, Tatiene, for all the love and encouragement you constantly provide. I also appreciate the patience you have had during tight submission deadlines, exams and writing of the Thesis.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
U of C	University of Calgary
BSP	Binary space partitioning
2D	Two-dimensional
3D	Three-dimensional
CAD	Computer aided design
Δ_k	The scalar difference between k_{i+1} and k_i (given any axis k)
NPR	Non-photorealistic rendering
HLSL	High Level Shader Language
API	Application programming interface

Chapter 1

Introduction

This thesis presents different approaches to tackle the problem of occlusion in 3D grids. In this work we propose the use of the *Cutaway Views* technique to improve the inspection and analysis of grid cells in corner-point grids. In addition, we present an approach for creating interactive *Exploded View Diagrams* in generalized 3D grids and an early approach based on a mass-spring system.

1.1 Motivation

1.1.1 Occlusion Problem in 3D Grids

One of the challenges in 3D space is the problem caused by occlusion. By simple definition, what we mean by occlusion problem is that objects of interest lie behind opaque objects causing difficulties in gaining further insights (see Figure 1.2). There are many approaches to solve this problem inspired by illustration techniques [23] which have been created to enhance the understanding of complex scenes by, for example, exposing internal structures and highlighting important features (i.e., focus and context, distortion view) – Figure 1.1. The generation of ever-larger 3D datasets in various applications domains, such as architecture, engineering and manufacturing, has created a need for effective visual communication techniques that allow the user to intuitively and interactively explore and comprehend the data. In order to visualize occluded objects, there are some techniques that remove a portion of the data (i.e., the occluding data) while others, try to achieve the same effect by applying deformation, transparency or splitting the data. The best visualization approach to be used depends on the applied domain and on the tasks.

The 3D objects under focus in this thesis are 3D grids – they are typically used whether

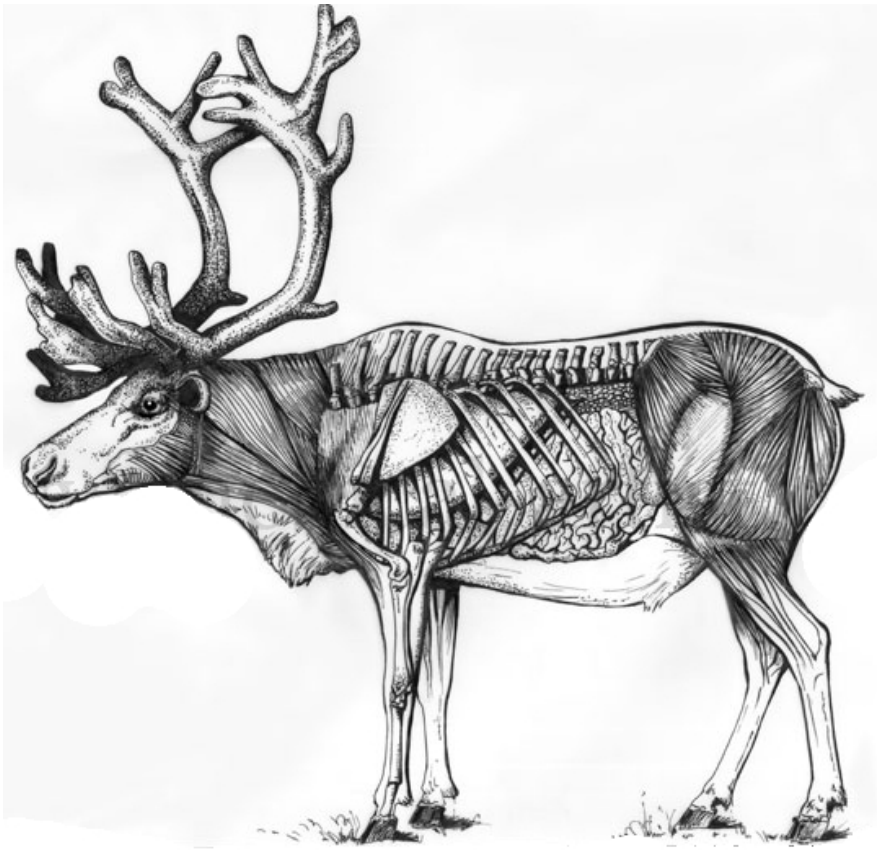

Figure 1.1: Internal structures are exposed by traditional illustration techniques.

for CAD or for physical simulation such as finite element analysis or computational fluid dynamics. In general, there are two categories of grids: regular and irregular. In a regular grid, all cells have the same dimensions (i.e., height, width and depth), therefore indices and coordinates can be obtained in a straight forward manner (see Figure 1.3 a). Among the non-regular grid types, corner-point grids have been widely used in flow simulation and geological modeling. In comparison to regular grids, corner-point grids reduce the numerical error by adjusting themselves closer to the actual geometry [40](see Figure 1.3 b). Although regular grids have been used for flow simulation until nineties, later on, corner-point grid became a standard in the industry. For example, oil and gas companies assume that software related to flow simulation must support the corner-point geometry. Therefore, it demanded a new set of tool to attend to the new industrial requirements and the domain expert needs [13].

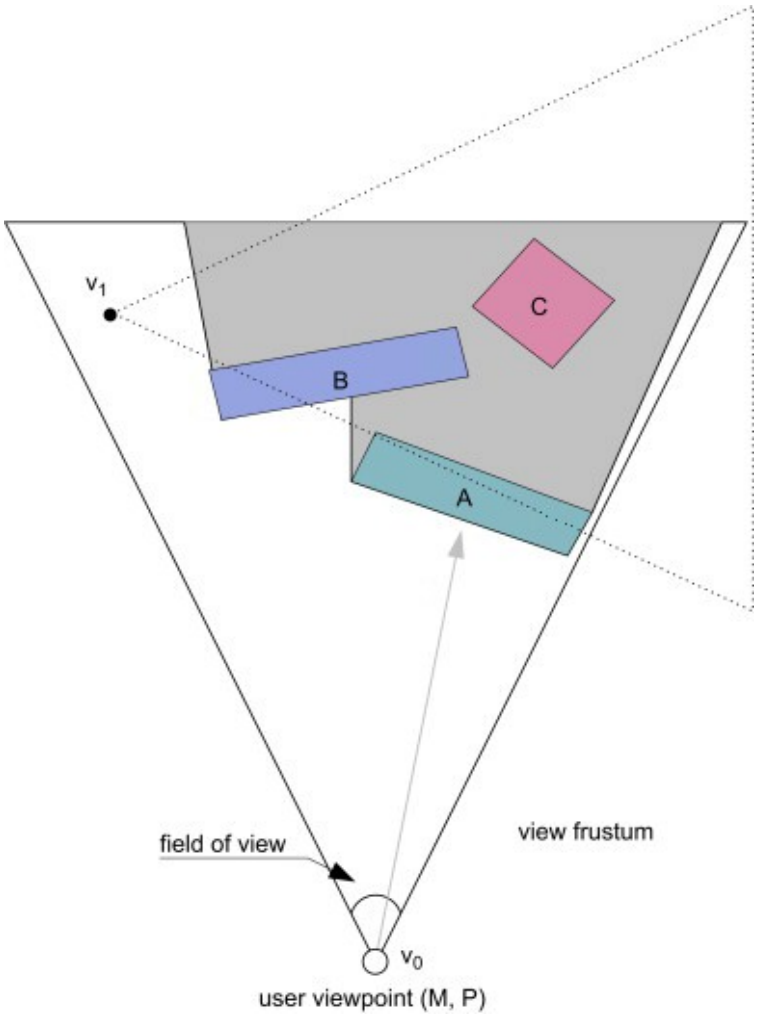

Figure 1.2: Occlusion problem: from v_0, object A occludes object C and partially occludes B. From v_1, object B partially occludes objects A and C.

1.1.2 Illustration-inspired Techniques

Complex 3D objects composed of many distinct parts and structures arise in a number of domains, including medicine, architecture and industrial manufacturing. Illustrations are often essential for conveying the spatial relationships between the constituent parts that make up these objects. For example, medical reference books are filled with anatomical illustrations that help doctors and medical students understand how the human body is organized. Similarly, repair manuals and technical documentation for industrially manufactured machinery (e.g., airplanes, cars) include many illustrations that show how the various parts in complex

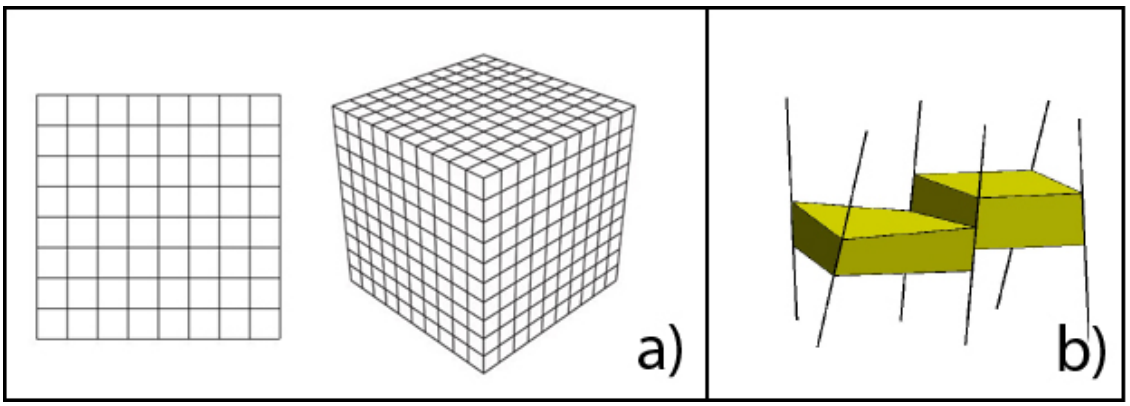

Figure 1.3: Grids: (a) regular 2D and 3D grids; (b) non regular 3D grid (corner-point grid).

assemblies fit together. In all of these cases, well designed illustrations reveal not only the shape and appearance of important parts, but also the position and orientation of these parts in the context of the surrounding structures.

Figure 1.4: Cutaway View: (left) the whole object; (right) object of interest is exposed by removing some part of the object.

Cutaway views [12] can be defined as a technique which removes occluding objects (of secondary importance) to expose the object of interest (of primary importance)(see Figure 1.4). When illustrators need to visualize important parts of the 3D objects occluded by other objects, the most commonly implemented concept is the use of Cutaway technique [12]. Cutaway drawings in technical illustrations allow the user to view the interior of a solid opaque object. In these illustrations, entities lying inside or going through an opaque object are of more interest than their surroundings. Instead of letting the inner object shine

through the girdling surface, parts of the exterior object are removed. This produces a visual appearance as if someone had cutout a piece of the object or sliced it into parts. Cutaway illustrations avoid ambiguities with respect to spatial ordering, provide a sharp contrast between foreground and background objects, and facilitate a good understanding of spatial ordering. This is possible because cognitively, the mental model in the mind of the user "fill in the gaps" of the removed material and can estimate the spatial relationship. Another reason for the popularity of Cutaway illustrations is perhaps the fact that they are the most straight forward approach to view occluded objects.

Figure 1.5: Exploded View: (left) the whole object; (right) object of interest is exposed by splitting the object.

However, in the scenarios where assembling the missing spatial parts becomes difficult (i.e., when the size and complexity of the 3D model increases), then Cutaway can prove to be not so suitable and efficient for communicating the necessary information. To overcome this drawback, Exploded View is a technique used by illustrators in which parts are separated (i.e., exploded) away from each other to reveal parts of interest while nothing is removed. It is another way of conveying the internal structure of 3D objects. Well-designed Exploded views not only expose internal parts, they also convey the global structure of the depicted object and the local spatial relationships between parts. Furthermore, unlike other illustra-

tion techniques that reveal internal parts *in situ* by removing or de-emphasizing occluding geometry, such as Cutaways and transparency, Exploded views show the details of individual parts. From a computer science perspective, Exploded view diagrams [31] can be defined as a technique which displaced and even split secondary objects while primary objects are exposed (see Figure 1.5).

1.2 Goals

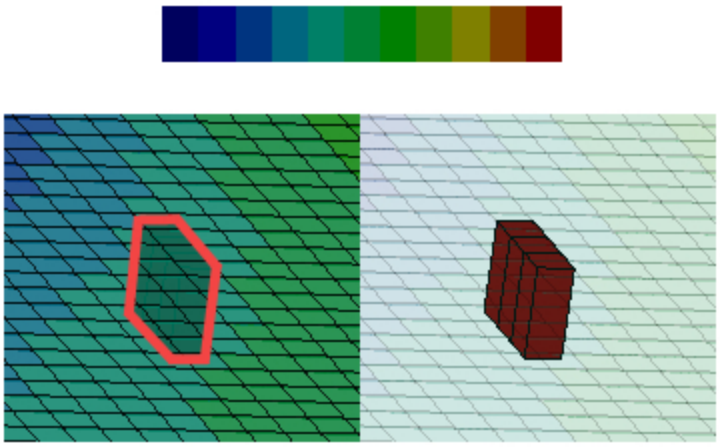

Figure 1.6: Some techniques advocated the use of transparency, however it makes difficult to correlate grid cell with the legend: (top) the legend and its actual colors; (bottom-left) occluded grid cells colors are unclear while it is possible to correlate the remaining cells colors with the legend; (bottom-right) in order to make occluded grid cells colors easy to be correlated, the remaining cells colors become unclear.

In the context of 3D grids, the fundamental goal of this thesis is to explore visualization techniques to address the problem of occlusion. Our focus is not on proposing different rendering styles, instead we focus on proposing variations of existing techniques and how they fit into our data model. We believe that when dealing with data that colors are mapping meaningful values, techniques that modify the color should be carefully considered. Otherwise, the relation between the values and the color might become loose (see Figure 1.6). One of the major components of our research goals was to apply the two techniques – Cutaway View and Exploded View Diagrams, to regular and more importantly non-regular 3D grids

with no part hierarchies in an interactive manner. Although scalability is not the main goal of this thesis, it is taken into consideration in our approaches.

1.3 Methodology

All our approaches depend on a way of selecting primary objects and also a way to generate and load 3D grids (whether they are regular or not). A shared library was develop to generate regular grids and also to load those generated models in addition to the Corner-point grids. Primary objects are composed by using a concept of filter by selecting grid cells that fall into a range of value. In our first approach we used the Cutaway technique (Chapter 3) which at the same time that it achieves our goal it has a straight forward implementation. We have made a simplification to have more balance between visual effect and performance, that is, we removed more data than necessary by applying the technique in a discrete form (grid cells are not partial removed, they are either drawn or not) to gain more performance as it decreases the amount of calculation (instead of compare each of the 8 vertices we just compare their centroid). This approach gave us some results for the case that losing some amount of data is allowed.

From a different perspective, our second approach expose objects of interest without losing any data. we created an implementation to validate the use of Exploded View technique with grids and in our first implementation we used a mass-spring system (Chapter 4) to support the deformations and also to create animation. The outcome of this implementation led us to choose a different direction and as a consequence we have applied the Exploded View (Chapter 6) by using projected coordinates and also creating a data structure (Chapter 5) based on BSP-tree to perform the expected effect by using limited amount of information.

1.4 Contributions

In order to propose solution to the problem of occlusion in 3D grids, we propose some approaches based on illustration techniques. Apart from the main problem we also take into consideration factors like scalability and performance. Each of our techniques belong to one of the following categories: (1) techniques that expose part of interest by removing data; (2) techniques that objects of interest are exposed without removing any data.

The contributions of this thesis are:

- A variation of the Cutaway Views technique applied to corner-point grids. We applied the Cutaway technique to allow the inspection of occluded grid cells in corner-point grids, balancing performance and visual effect in a CPU based algorithm.

- A variation of the Exploded View Diagrams technique applied to 3D grids. This approach was implemented assuming that: (1) 3D grids do not have geometrical information regarding parts; (2) the solution takes the problem of scalability into consideration; and (3) without relying on pre-processing stage;

- A data structure – Explosion Tree, to support Exploded Diagram with limited amount of information; This data structure is the core of the approach that we developed, it receives limited amount of information as input and offer the final position for every vertices as an output.

1.5 Overview

This thesis comprises seven chapters. Chapter 2 provides a general background about 3D grids, specifically about Corner-Point grids. The background chapter also discusses about the concept of Cutaway-view and Exploded Diagrams techniques, and it discusses how the

selection of regions of interest are applied to 3D grids. In Chapter 3, we present an implementation to apply the Cutaway-view technique to corner-point grids. In Chapter 4, we present a first implementation of the Exploded views technique by using a mass-spring system [24]. Following up on the limitations of this approach, we developed a data structure (Chapter 5) which is used in a second implementation (Chapter 6). Finally, Chapter 7 summarizes the contributions and discusses possible directions for further research.

Chapter 2

Background

The origin of illustration can be found in the Paleolithic period somewhere between 30,000 and 10,000 B.C. Illustration has always been an important visual communication medium among humans. In this period, cave paintings displayed mostly large wild animals, such as horses, bison, aurochs, deer, and tracing of human hands. It is rare to find drawings of humans and they are usually schematic rather than the more naturalistic animal subjects. Charcoal, manganese oxide, hematite, red and yellow ochre were the main components used for painting. The silhouette of the animals were sometimes incised in the rock first [4] (Figure 2.1(a)).

Figure 2.1: Paintings: a) prehistoric ; b) ancient Egypt (hieroglyphic); c) ancient Greece.

Among the first professional artists, the painters of ancient Egypt (3200 B.C to 30 B.C.) were one of the most well-known. They used their visual language, hieroglyphs, in religious

practices, scientific data, political propaganda and also as part of their daily life. The line is the most important element in Egyptian paintings. All paintings are bordered by black lines. The fact that Egyptians did not use perspective is generally accepted. Instead, in their early profile drawings they used hierarchical perspective where through overlapping, they tried to give the idea of depth. For instance, to represent workers involved in the seeding of the fields, they created a scene with sets of people overlapping each other. The main idea was to give the impression that there were several people working next to each other. The base of these paintings were sarcophagus wood or tomb wall [4](Figure 2.1(b)).

Not different from the hieroglyphs of the ancient Egyptians, the architectural drawings of the early Greeks (1100 B.C. to 100 B.C.) also lacked perspective. The ancient Greek architects were also influence by this kind of art and designed their buildings to visually counterbalance the intuitive understanding of perspective of the viewer. For instance, the Parthenon of Athens, which was situated at the top of the Acropolis compound was build based on this principle. As it could be just approached from one access point, the construction technique gave the Parthenon an appearance through which it approximated the flat or axonometric views that the Greeks were used to seeing in their art. This effect was achieved by building the rear of the structure bigger and wider than the front, and the side columns increase in mass from front to rear [4] (Figure 2.1(c)).

In the year 1000 A.D. the principle of perspective was defined by the Arabian mathematician and philosopher Alhazen [39]. In his work, he explained how light is conically projected into the eye. Three hundred years later, an improvement in perspective was developed during the Renaissance and a method for projecting a scene onto an image was developed. In this period, through the work of artists such as Leonardo da Vinci (1452-1519), the beginning of descriptive technical illustrations took place. The personality of da Vinci combined artistic abilities with a scientific mind. This enabled a merging of invention with visual art. Another major achievement of this period was the creation of spatial illusions. Therefore,

the evolution of what is called illusionistic perspective was taking place (Figure 2.2).

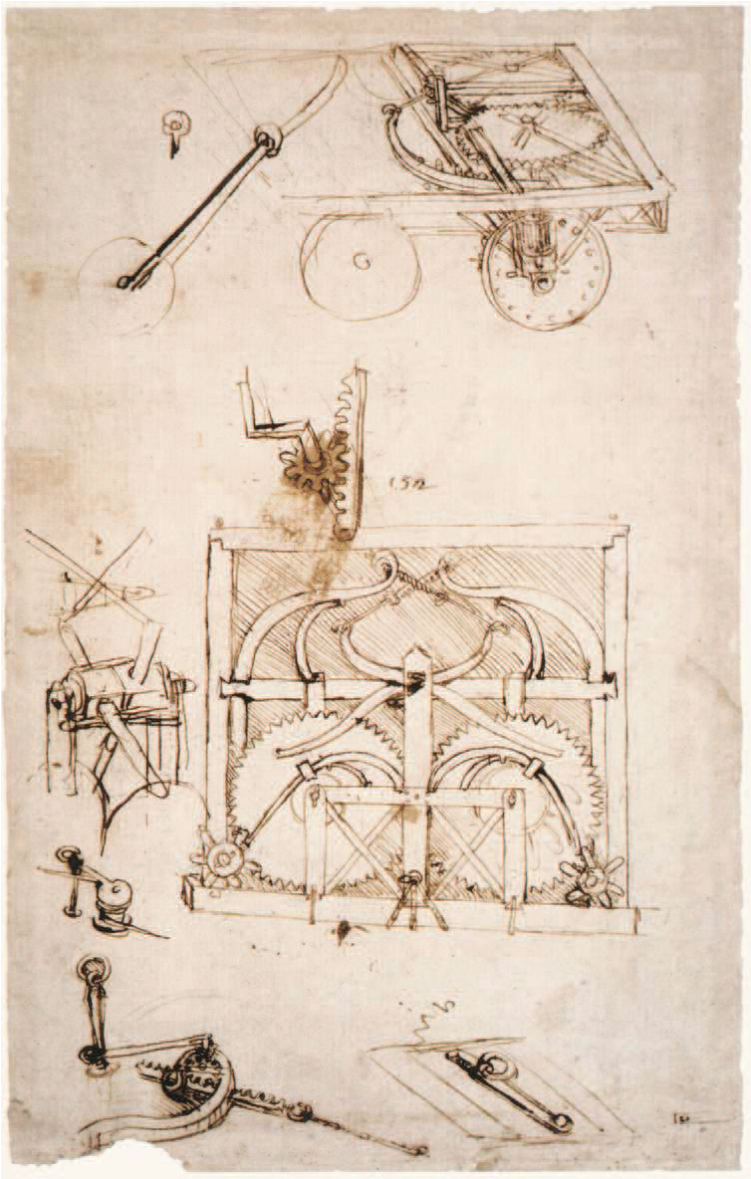

Figure 2.2: Technical illustration of a vehicle created by Leonardo da Vinci

The industrial revolution was directly influenced by the flourishing of technical illustrations. Conventions and standards in technical illustrations that were universally understood were created as a need for mass production and outsourcing. Both artistic and technical illustrators had a set of methodology available for illustrating objects and environments more realistically around the 50s. The design of illustrative techniques has often the goal of allowing a person with no technical understanding clearly understands the subject being

depicted [23]. By using various line widths to emphasize mass, proximity, and scale, help to make simple line drawing more understandable to the lay person. Other basic techniques like cross hatching and stippling, add a greater depth and dimension to the subject matter. Around 60s, technical illustration was further advanced during the photorealistic art direction. Photographic slide projection onto canvases were often used by photorealists. With an enphasis on details, this style is very accurate and often simulates glossy effects such as reflections in specular surfaces (Figure 2.3).

Figure 2.3: An example of the photorealistic art: Ralph's Diner oil on canvas.

Using different media tools and techniques (i.e. pencil, ink, watercolor, photorealism), the technical illustrator could now convey highly complex technical information to a large number of people with different levels of expertise, thus matching different communication goals. To further increase the expressivity of illustrations, various techniques have been established. Not different from handmade illustrations, digital illustrative techniques have been addressing different challenges in order to reproduce natural illustration in a digital form.

The problem of automatically reproducing various illustration rendering styles have been

investigated by many researchers in the field of Non-Photorealistic Rendering (NPR), a field of Computer Graphics established around the early 90s [20, 45]. The aim of this style is to create attractive and informative images that effectively convey the texture, material properties and shape of the depicted objects. The problem of generating effective line rendering have been focused by many NPR researchers. The work of Saito and Takahashi [43] is an example of a seminal contribution in this area. They present a method for rendering 3D objects by enhancing meaningful linear features, such as sharp creases and silhouettes. Dooley and Cohen [14] provides to users an intuitive interface for customizing how different types of linear features are rendered. As they are so effective at conveying form, silhouettes in particular have received a wealth of attention in the NPR community. Numerous methods for efficiently computing and rendering silhouettes have been proposed by Hertzmann [22], Markosian [37] and many others researchers.

New linear features that help convey shape were introduced by more recent work. The use of "suggestive contours" was proposed by DeCarlo et al. [10], defined as surface contours that turn into silhouettes in nearby views by Koenderink [29]. "Apparent ridges" is introduced by Judd et al. [27], which indicate regions of maximal view-dependent curvature. Similarly to line drawing, other illustrative rendering styles have been investigated by researches. Interrante and collaborators [18] [26] [25] propose line-based textures that help communicate surface shape by following principle curvature directions. Winkenbach and Salesin [50] [51] introduced techniques for generating pen-and-ink style renderings and then these techniques were extended to support real-time hatching by Praun et al. [41]. Other rendering styles that have also been considered include stippling [41], graphite [44], charcoal [36] and watercolour [9].

Illustrative rendering techniques include the use of non-physical or exaggerated lighting and shading models that emphasize the shape, texture and reflectance properties of the depicted object. Gooch et al. [19] incorporate lighting principles from technical illustra-

tion to produce a shading model that conveys shape through warm-cool colour variations. Hamel [21] introduces a component-based model that supports standard lighting configurations such as back lighting, as well as novel techniques like curvature lighting, which indicates surface curvature via variations in luminance. More recent work includes Luft et al.'s [35] technique for emphasizing depth discontinuities with false shadows, and Rusinkiewicz et al.'s [42] method for generating "exaggerated" shading effects that emphasize surface detail. Researchers have also considered the use of image-based techniques for creating illustrative lighting effects. Akers et al. [3] and Fattal et al. [16] propose methods for combining photographs of an object under different illumination to create non-physical lighting conditions that enhance shape cues and surface detail. Advances in illustrative rendering have made it possible to create high-quality digital images that emphasize important features such as shape, texture, and material properties. However, these rendering techniques alone do not address the problem of occlusions in complex objects.

In this chapter we give a brief background to the main topics of this thesis. Since the focus of our thesis is to propose solutions for the occlusion problem in 3D grid, the chapter is essentially divided into two main parts – 3D grids and visualization techniques. Section 2.1 introduces 3D grids – their types, their problems and their advantages. Section 2.2 explain the concept of selecting cells which is shared by all approaches. Section 2.3 moves ahead introducing state-of-art research in the area of Cutaway view technique. Section 2.4 provides a brief background to Exploded view diagrams.

2.1 Data Model

The understanding about 3D grids can be introduced by the understanding of the grid generation process which is, of course, only a means to an end: a necessary tool in the computational simulation of physical field phenomena and processes. Grid generation is, from a technology standpoint, still something of an art, as well as a science. Mathematics provides

the essential foundation for moving the grid generation process from a user-intensive craft to an automated system. Since there are no inherent laws (equations) of grid generation to be discovered, grid generation systems rely in both art and science in the design of grids. The grid generation process is not unique; rather it must be designed. There are, however, criteria of optimization that can serve to guide this design. The grid generation process has matured now to the point where the use of developed codes freeware and commercial is generally to be recommended over the construction of grid generation codes by end users doing computational field simulation. Some understanding of the process of grid generation and its underlying principles, mathematics, and technology is important, however, for informed and effective use of these developed systems. And there are always extensions and enhancements to be made to meet new occasions [48].

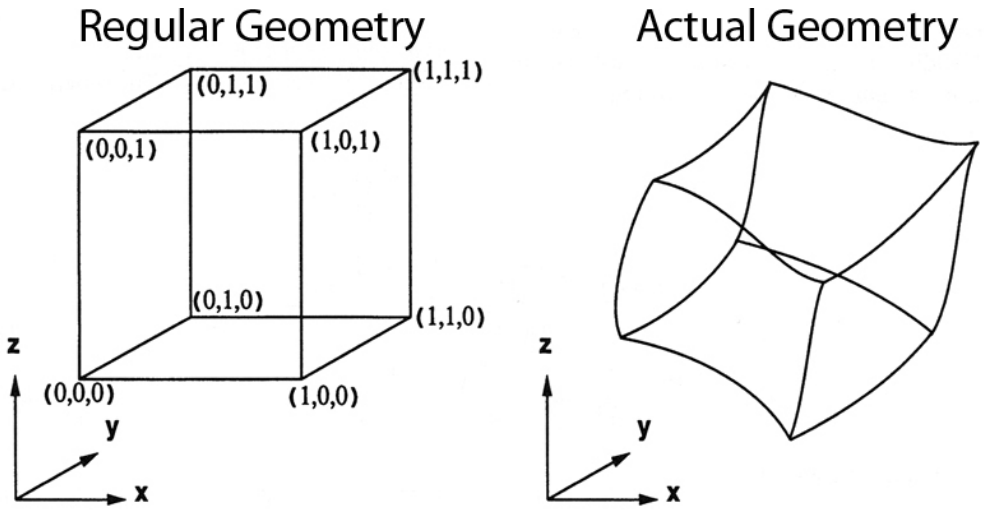

Figure 2.4: Regular geometry vs. Actual geometry.

In order to create a computational model of the actual reservoir some different representations have been used. A rectangular Cartesian coordinate system was widely used as it has the advantage of familiarity; since the equations describing physical phenomena were traditionally described in Cartesian coordinates, for the reservoir engineer it was straightforward to construct rectangular grid systems as they appeared natural and intuitive. By using regular grids the position of grid cells and their indices can be predicted just by the

fact that all of them share the same geometry, Figure 2.5 shows how it is possible to retrieve the final vertex position by having its indices (i.e., i and j) and the variation in each axis (Δ_x and Δ_y) which is shared among of vertices of this grid. Unfortunately, regular grids do not permit a good representation of the reservoirs geological features, since they impose a rectangular structure with possibly no physical relation to the reservoir being studied (see Figure 2.4), and, therefore, may produce erroneous results [49].

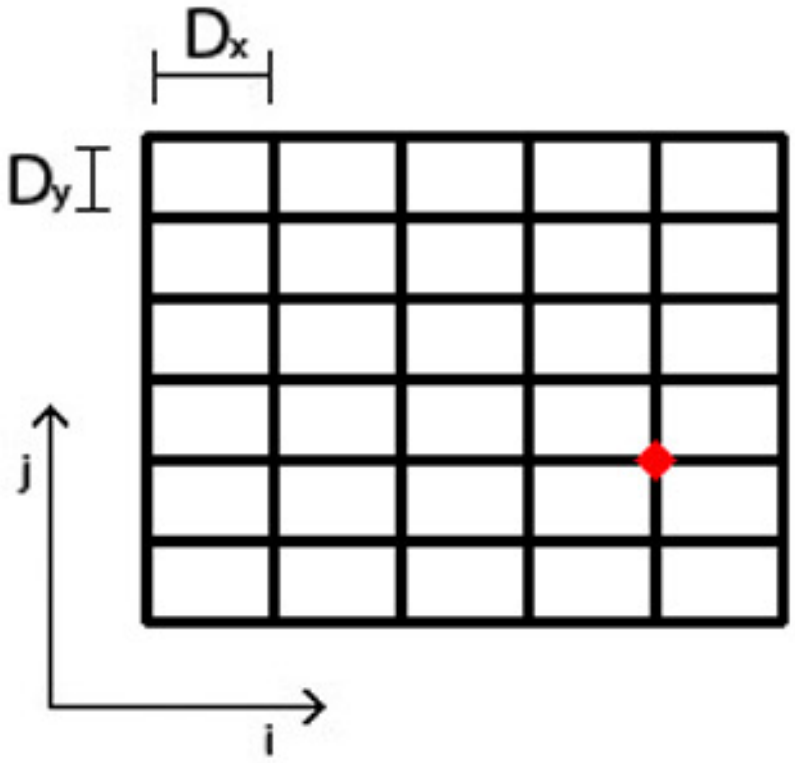

Figure 2.5: Regular grid example: the grid vertex indexed by i equals four and j equals two has coordinates values x equals $4 \times D_x$ and y equals $2 \times D_y$.

In order to reduce simulation errors, other approaches than using regular grid (based on a rectangular coordinate system) are used. Different flexible grids are derived from a non-rectangular coordinate system as they better adapt to irregular geometries. Therefore, geological features like reservoir boundaries, faults, horizontal wells and others have a better

approximation to their actual geometry and as a result a better simulation can be achieved. Furthermore, flexible grids make possible the creation of some applications like well modeling. Among non-rectangular grids (also known as distorted or flexible grids) there are triangular grids, Voronoi grids and Corner-point grid. The latter has become specially popular because it has a straightforward implementation in standard reservoir simulators (even in complex full field studies), while achieving a performance gain due to its regular matrix structure.

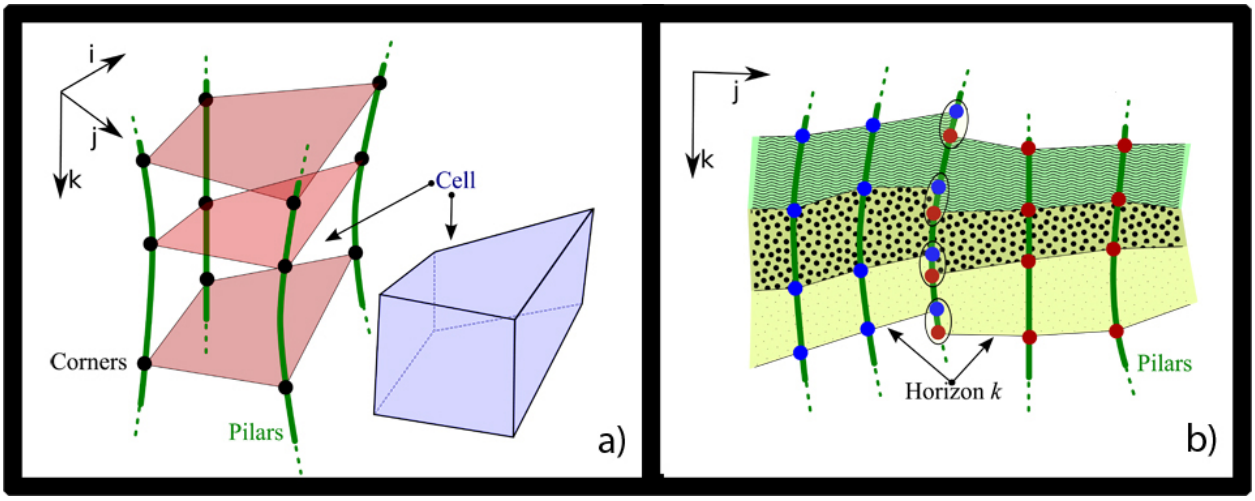

Figure 2.6: Corner-Point Features: (a) the three main elements of a corner-point geometry: corners, pillars, and cells. (b) typical corner point grid discontinuity. Circulated corners emphasize the irregular geometry.

Corner point geometry is composed of three main elements: corners, pillars, and cells (Figure 2.6 a). Cells are defined by eight vertices (corners) and are the smallest volumetric element in the grid. Each corner point cell contains 3D information (x, y, z) and is identified by integer coordinates (i, j, k), where i and j span each layer, and the k coordinate runs along the pillars. During the modeling stage pillars are represented by 3D curves (usually splines), however, this information is lost upon model completion. Since we focus on applications at later stages, for our purposes, the integer coordinates (i, j, k) has only a structural (topological) meaning. The 3D coordinates of the corners have neither to be regularly spaced (Figure 2.6 b) nor spatially continuous [13].

The following pseudo code shows the difference between regular and corner-point grid

traversal:

```
\\Regular Grid Example
Input-------------------------------------------------------------
origin \\x,y and z coordinates representing the point of origin.
size \\x,y and z coordinates representing the total size.
step \\x,y and z coordinates representing Dx, Dy and Dz.
Begin-------------------------------------------------------------
From z = origin.z to size.z
| From y = origin.y to size.y
| | From x = origin.x to size.x
| | | StoreVertex(x,y,z) //The vertex position can be obtained
| | | x += step.x        //by relying on the regularity.
| | y += step.y
| z += step.z
End---------------------------------------------------------------

\\Corner-point Grid Example
Input-------------------------------------------------------------
size \\ number of grid cells in i,j and k direction.
cpg \\ the grid itself
Begin-------------------------------------------------------------
From k = 0 to size.k - 1
| From j = 0 to size.j - 1
| | From i = 0 to size.i - 1
| | | cell = cpg.cell[i,j,k]
| | | if (cell.isValid) //Some cells might be flagged as invalid
```

```
| | | |
| | | | From n = 0 to 7
| | | | | x = cell.corner[n].x
| | | | | y = cell.corner[n].y
| | | | | z = cell.corner[n].z
| | | | | StoreVertex(x,y,z) //For valid cells the position needs
| | | | | n += 1             //to be retrieved for each corner
| | | i += 1
| | j += 1
| k += 1
End-------------------------------------------------------------
```

Note that different from the regular grid, in a Corner-point grid, cells can be invalid (e.g., fault, pinchout), vertices can be duplicated and it does not provide a way to get vertices in any primitive fashion. The validity of each cell is an attribute stored in the grid and it can be either globally accessible or through each cell.

2.2 Selecting Grid Cells of Interest

Both Exploded Diagram and Cutaway techniques have been mainly applied to CAD models. In these models, objects are well defined and also their parts, therefore, to classify one (or more objects) as object of interest while considering the remaining as context is an intuitive process. In the case of 3D grids, it is composed of grid cells and some rule must be applied to classify grid cells either as object of interest or as context (objects of secondary importance).

The selection of grid cells of interest has the goal of defining which cells are in focus, and it can be done in several ways. Depending on the data type and even the meaning of the data some ways would be more suitable than others. One example of selection can be done by index values, for instance, in a 3D grid with 10, 20, 30 cells in the i, j and k respective

directions, one might want to select the cell with i, j, k values 5, 10, 15. Another example of selection can be done by coordinate values either by an exact values or a value range. For those grids which cells have not only geometrical information but also values associated to them, selecting using a range of values is often used. Next, we are going to present this latter selection in the context of Oil and Gas domain.

2.2.1 Applying the Selection in the Context of Oil and Gas

The main goal of the oil and gas industry is the search for hydrocarbon deposits beneath the earth's surface. Among its many sectors there is the upstream oil sector, also known as the exploration and production sector. Due to its complexity, it requires professionals from many different domains (e.g., petroleum engineering, reservoir engineering, geology, geophysics) to collect, analyze and interpret the available data. In order to have an effective reservoir characterization the following are required: a detailed geological description (from static data); the analysis of the dynamic behavior of the reservoir (from a measurement known as well testing); and thorough geological modeling and numerical simulation to integrate these two aspects. A 3D geological model improves understanding of the reservoir providing a better presentation and visualization of its architecture and fluid behavior. The main purpose of the geological model is to evolve an actual field development and future reservoir management in order to ensure the maximum hydrocarbon recovery [46].

From the oil and gas perspective, grid cells apart from their geometry, can have one or more properties (scalar attributes) such as pressure, porosity and permeability. In order to focus on a subset of grid cells, a selection mechanism, for instance, based on a specific range of property's values could be provided. By selecting different ranges, one could learn more about the property distribution while having a better understanding of the whole geological model. For instance, in most cases, grid cells with high values of pressure are a good indication of gas, oil or water, therefore, based in this knowledge one might want to focus on grid cells which have high values.

In the context of cutaway-view and corner-point grid, a way to select grid cells was built around the concept of filters. In this context, a filter is a selection performed (through two sliders) in the well testing window, which is a dialog with a 2D embedded plot showing time values in the x-axis and property values (e.g., pressure, porosity) in the y-axis. While the blue curve represent the actual time vs. property, the red curve represent a derivative values of the current property. The grid blocks with property values falling inside the selected range will be visible, while others will be used for context. The well testing window has three states, both filters not activated (see Figure 2.7 (a)), first filter activated (see Figure 2.7(b)) and both filters activated (see Figure 2.7(c)).

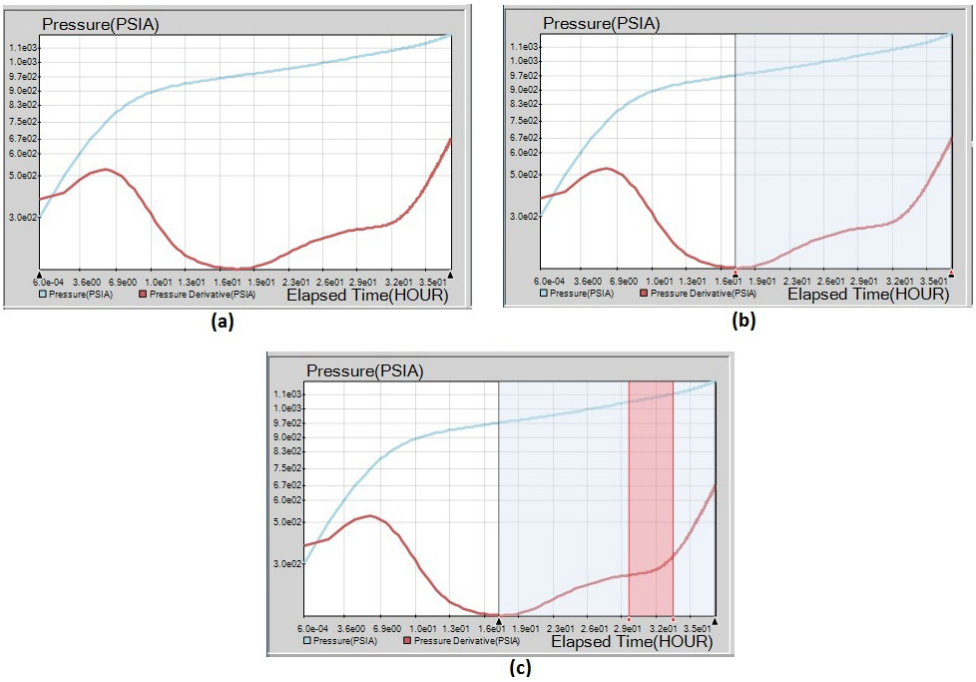

Figure 2.7: Well testing window shows time (x-axis) vs. pressure values (y-axis). While the blue curve represent the actual values, the red curve represent the derivative of the current property. With two set of sliders, the current window can have the following states : (a) both filters deactivated, (b) just one filter activated, (c) both filters activated.

In order to create the focus+context effects for the first filter, only the selected blocks are drawn while other are not. Therefore, to avoid losing the context the layers of the geological model are drawn with some degree of transparency (see Figure 2.8). The second

selection's context is primarily done by applying transparency into the unselected blocks while the selected ones are drawn with full opacity (see Figure 2.8). However, depending on the relation between the positions of the selected and unselected blocks the cells the regions in focus becomes unclear and to solve we applied both Cutaway and Exploded Diagram techniques in Chapters 3 and 6 respectively.

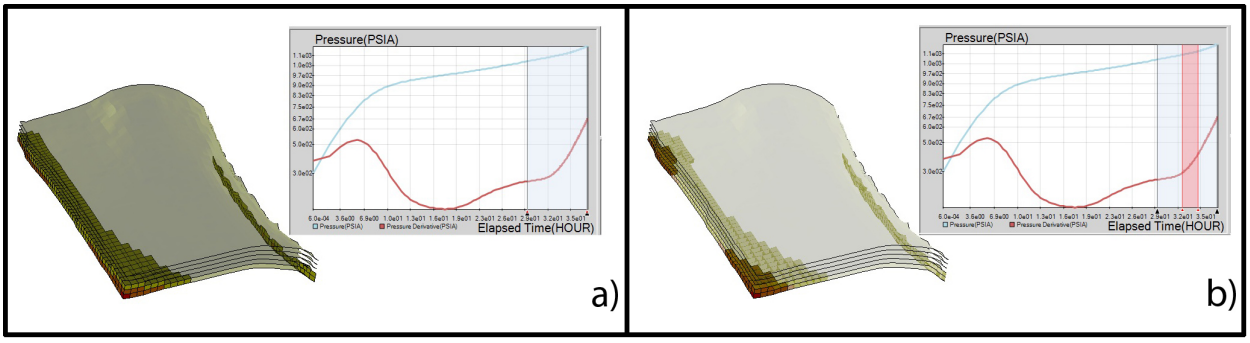

Figure 2.8: (a) after the first selection using the sliders the first filter is applied. Only selected blocks are visible. (b) After the second selection using the sliders the second filter is applied. Unselected blocks have some transparency and the selected ones have full opacity.

2.3 Cut-Away Views

Illustration techniques have been created to enhance the understanding of complex scenes by, for example, exposing internal structures and highlighting important features. The generation of ever-larger 3D datasets on various applications domains, such as architecture, engineering and manufacturing, has created a need for effective visual communication techniques that allow the user to intuitively and interactively explore and comprehend the data. Cutaway is one of these techniques that visually aid the inspection of 3D models, and is specially useful when the model in question has a volumetric character, contains multiple layers, or has interconnecting pieces. Although there are classes and rules brought from the traditional illustration a simply and efficient cutaway approach is to just remove occluding objects (of secondary importance) to expose the objects of interest which have primary importance (see Figure 2.10). However, in this way, the user is forced to mentally "fill the

gaps" of the removed material and estimate its spatial relation. A better approach that increases comprehensibility, is to selectively discard portions of these overlapping objects while still rendering some of their outlines in order to, at the same time, minimize occlusion and provides effective visual depth and spatial cues of the interior. This approach produces a visual appearance as if someone had carved a piece of the object.

Diepstraten et al. [12] define cutaway for technical illustration as the main motivation, and also justifies its categorization into NPR (i.e. non-photorealistic rendering) and discuss different methods to implement this technique. They provide some rules for the described methods in order to help the reader to choose one of them for a specific purpose. When manually selecting and tracing cuts, an artist can carefully reason about the visibility issue; however, an interactive system faces the challenge to somehow achieve this goal respecting performance constraints. One obvious approach is to try to simulate the craft of the illustrator as best as possible, where the user can manually delimit cuts by visually tracing paths over the model. The top shell is then removed exposing the underlying parts. There are two sub-classes more well-known among illustrator, they are known as cutout and breakaway (see Figure 2.9).

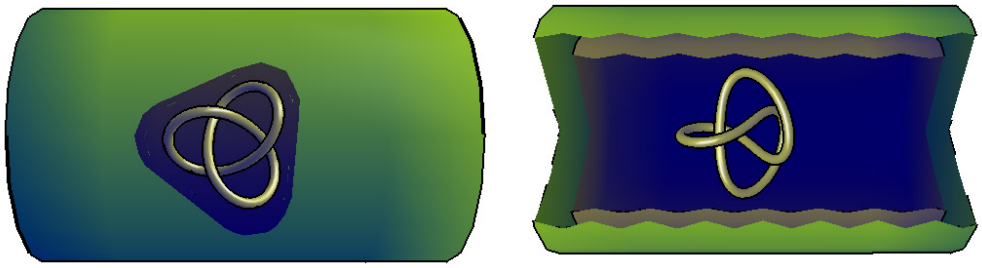

Figure 2.9: Cutaway sub-classes: (left) cutout; (right) breakaway.

Regarding the cutout, some specifications must be followed in order to achieve its original idea:

(1) Inside and outside objects have to be distinguished from each other. Note that this requirement not only covers scenes with a single outside object, but has to

allow for scenarios with several disjoint outside objects and even nested layers of outside objects. Another issue is the shape of the cutout geometry.

(2) The cutout geometry is represented by the intersection of (a few) half spaces. By construction, this cutout geometry is convex.

(3) The cutout is located at or around the main axis of the outside object. This rule determines the position and orientation of the above cutout geometry

(4) An optional jittering mechanism is useful to allow for rough cutouts.

(5) A possibility to make the wall visible is needed. This requirement is important in the context of boundary representations of scene objects, which does not explicity represent the solid interior of walls.

The other Cutaway sub-class in the form of a breakaway is based on a slightly different set of rules. The first requirement (1) for a distinction between inside and outside objects is the same as in cutout drawings. However, the shape and position of the breakaway geometry is not based on rules (2) and (3), but on: (6) The breakaway should be realized by a single hole in the outside object. If several small openings were cut into the outside surface, a rather disturbing and complex visual appearance would be generated. Nevertheless: (7) All interior objects should be visible from any given viewing angle. The above rule for making the walls visible (5) can be applied to breakaway illustrations as well. Jittering breakaway illustrations are seldom and therefore (4) is not a hard requirement for these illustrations.

Li et al. [30] present a system for authoring and viewing interactive cutaway illustrations of complex 3D models using conventions of traditional scientific and technical illustration. their goal is not only to apply the Cutaway technique but also to support interactive exploration. Most of their data are CAD and anatomical models. In their approach, an author instruments a 3D model with auxiliary parameters, which they call rigging, that define how

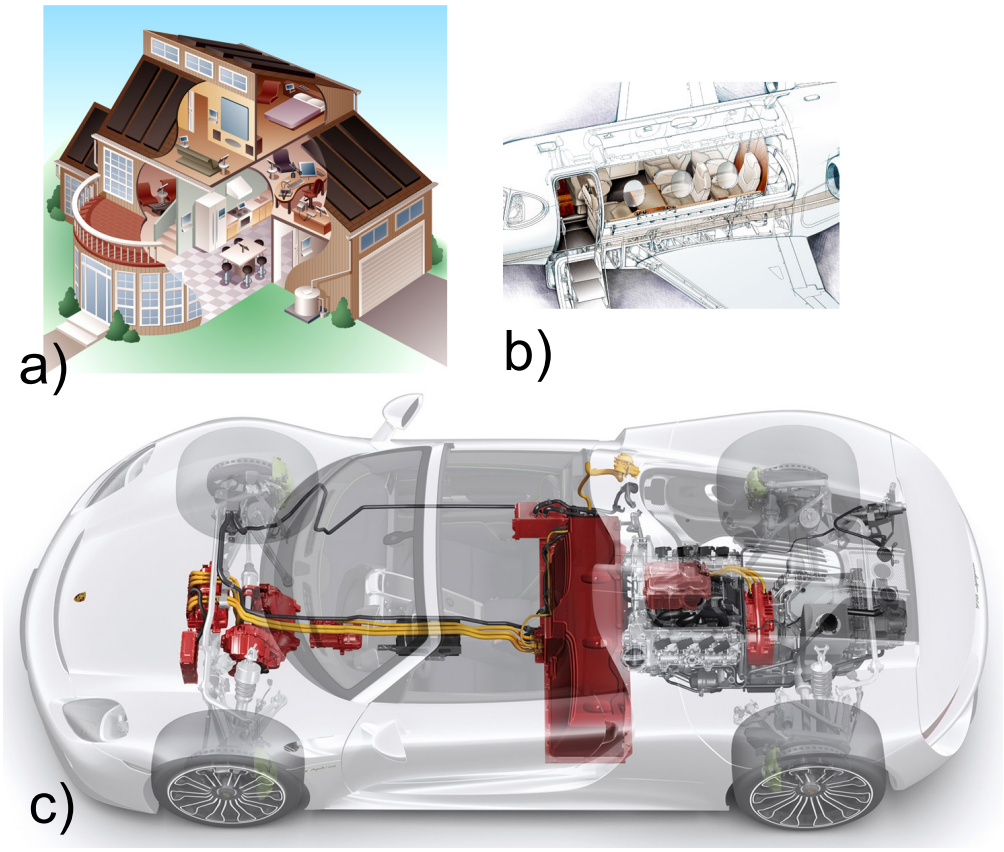

Figure 2.10: Cutaway View technique: occluding objects are removed to expose the object of interest: (a) house interior; (b) aircraft seats; (c) car parts.

cutaways of that structure are formed. They also provide an authoring interface that automates most of the rigging process. Also they provide a viewing interface that allows viewers to explore rigged models using high-level interactions. In particular, the viewer can just select a set of target structures, and the system will automatically generate a cutaway illustration that exposes those parts.

Others approaches involves the paradigm of primary and secondary objects, where the former are regions or objects of interest that one would like to keep visible throughout an investigation session. In other words, the goal is to automatically clip secondary objects to maintain the primaries in focus in a view dependent manner. Burns and Finkelstein [7] describe a real-time algorithm to adaptively generate cutaway illustrations by only selecting what are the primary and secondary objects, as well as an aperture angle defining the size of

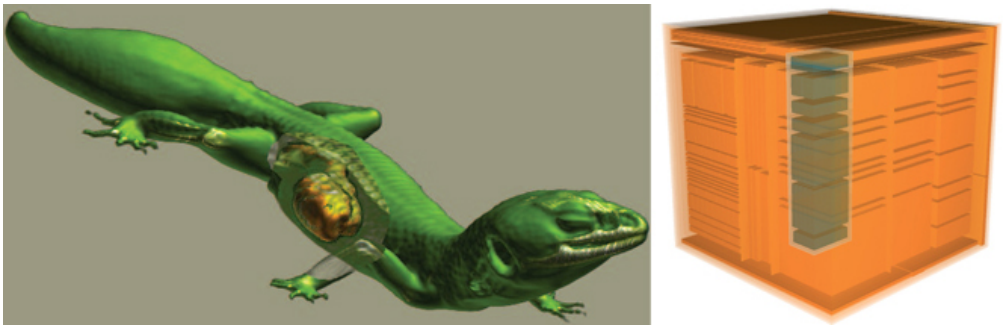

Figure 2.11: Techniques that remove occluding data : Cutaway View (left) and Ghost View (right).

clipped regions around the primaries. Different from previous approaces, they add techniques like perspective compensation and ghost line to the general cutaway effect.

Lidal et al. [33] apply Cutaway in a 3D geological model. They work with models which have mainly stratigrafic information. In other words, this data is basically a set of layers and their properties (i.e. rock properties). One of the main limitation is the fact that they assume that the layers are strictly horizontal and the layer's properties are static (i.e. they do not change over time). Another fact that would provide limitation when working with geological models is to assume that objects of interest are always near each other, therefore, they place them within a single cut section.

2.4 Exploded Views

In terms of conventions, when creating exploded views, illustrators carefully choose the directions in which parts should be separated (explosion directions) and how far parts should be offset from each other based on the following factors: (1) Blocking constraints: Parts are exploded away from each other in unblocked directions. The resulting arrangement of parts helps the viewer understand local blocking relationships and the relative positions of parts;(2) Visibility: The offsets between parts are chosen such that all the parts of interest are visible; (3) Compactness: Exploded views often minimize the distance parts are moved from their original positions to make it easier for the viewer to mentally reconstruct the

model; (4) Canonical explosion directions: Many objects have a canonical coordinate frame that may be defined by a number of factors, including symmetry, real-world orientation, and domain specific conventions. In most exploded views, parts are exploded only along these canonical axes. Restricting the number of explosion directions makes it easier for the viewer to interpret how each part in the exploded view has moved from its original position; (5) Part hierarchy: In many complex models, individual parts are grouped into sub-assemblies (i.e., collections of parts). To emphasize how parts are grouped, illustrators often separate higher-level sub-assemblies from each other before exploding them independently.

One of the challenges of 3D grids, and more specifically irregular grids is that their visualizations are often faced with problems caused by occlusion in 3D space. In order to visualize occluded objects, techniques like ghost view [34] and cut-away [12] have been applied, however, both of them remove some portion of the data, also known as the occluding data (see Figure 2.11). On the other hand, there are techniques that do not remove data and those might be more suitable, in cases where removing occluding data is strictly correlated to losing information. One such technique is exploded view diagrams [31].

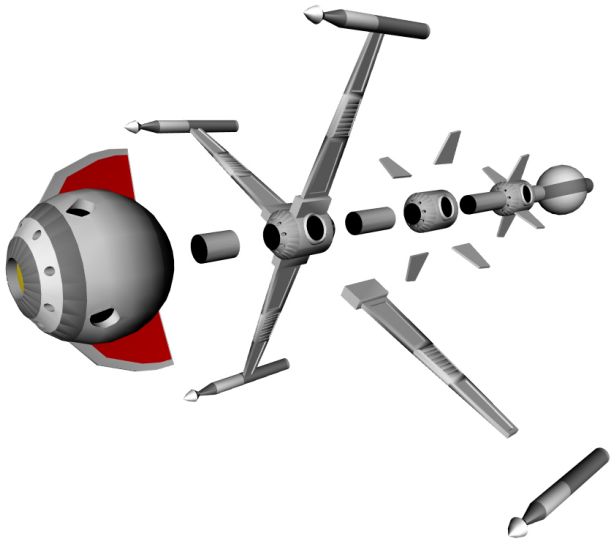

Figure 2.12: Exploded Views: secondary objects are shifted and maybe split until primary objects become exposed.

In this technique, objects of interest are considered primary, while the remaining data

is taken as secondary. The final result is that the secondary objects are shifted and maybe split until primary objects become exposed (see Figure 2.12). Li et al. [31] apply exploded diagrams in CAD models which all object parts and their hierarchy are well-known. In addition to the model their system requires the direction and displacement magnitude are calculated to achieve the final effect. Also it is important to note that in the previous applications parts were static, meaning that their geometry never changed, hence the relationship between parts was also static. Their system works in a interactive time and are based on a data structure known as *Explosion Graph*. This data structure is introduced by Agrawala et al. [1] and it has the goal of splitting the model's parts without causing any overlapping. It not only take information about the object of interest but its computation is based on the whole model. The interactive time of the system is aided by precomputation of the possible camera positions and the data structure state in each position.

Apart from 3D exploded diagrams, Li et al. [32] propose a 2D exploded diagrams using images as input. Most of their data comes from illustration manuals. There is a semi-automatic tool for layering the model's parts and also to specify how the parts of an object expand, collapse, and occlude one another. The sytem is based on a 2.5D diagram representation and is useful for 2D data.

Bruckner et al. [6] applied physical equations to add extra effect to the exploded view and in most cases he splits the secondary object into fixed number of pieces. His approach is simpler as the model is always split into four pieces, therefore, it reduces the complexity among the model's parts.

Some other work related to exploded view diagrams are: Karpenko et al. [28] explores exploded view diagrams applied to parametric surfaces and Tatzgern et al. [47] extends in the sense of choosing which parts to explode or not. Elmqvist [15] presents a prototype which is suitable when unrelated 3D objects shares the same scene and user can use a 3D pointer to explore the space. McGuffin et al. [38] use the concept of semantic layers to create the

concept of parts in volumetric data, still any of them consider that the relationship between primary and secondary objects can be changed.

Chapter 3

Cut-Away Views Technique Applied to Corner-Point Grids

In this Chapter, we present the Cutaway view technique applied to corner-point grids. In the scope of this thesis we are going to consider occluded object as one or more groups of grid cells. In this Chapter, cell selection is going to be performed based on the cells' values, more information on how selection is performed is provided in Section 2.2. The goal of our visualization is to expose objects of interest by removing some amount of the data which in this case are the remaining grid cells located between the object of interest and the view position.

Our approach was built around the concept of filters. The grid cells with property values falling inside the selected range will be visible, while others become the context. The main goal is to not draw the unselected blocks which are between the camera position and any selected block. According to Diepstraten et al. [12] when employing cutaway (breakaway technique) the following conditions must be satisfied:

1. Interior objects have to be distinguished from exterior ones;
2. From any given viewing angle all interior objects should be visible;
3. And a single hole in the exterior object should realize the breakaway.

It is important to mention that in the corner point grid, all objects are grid cells, and according to their properties they are classified as an interior or exterior object (analogous to the primary and secondary terminology). However, since this classification does not depend on the geometrical and topological structure, but only on the cells properties, interior objects

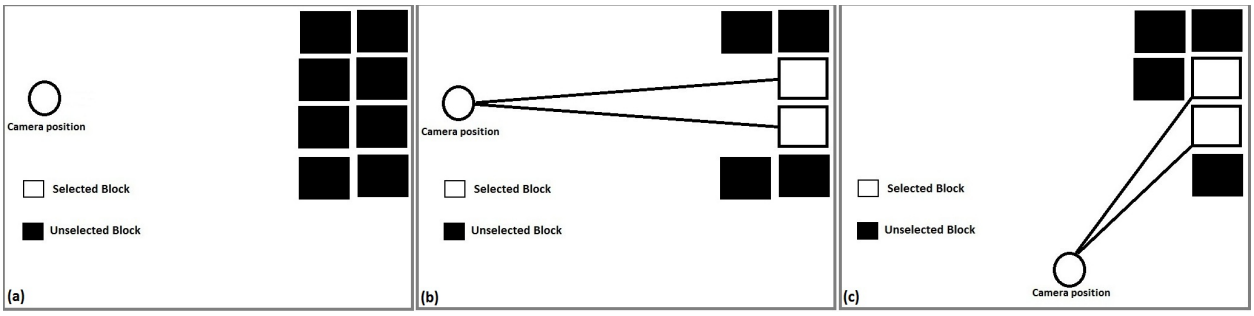

Figure 3.1: Illustration of our cutaway algorithm. Scenarios: (a) When there is no selected cells, all will be drawn, (b) removal of occluding cells based on the ray intersection results and (c) dynamic update of visible blocks according to camera position.

are not necessarily close to each other; hence, the third condition (3) is not always being satisfied (different from related work like Lidal et al. [33]). Note that, it is possible to have interior cells located at opposite extremes of the grid, thus, by creating a single hole, a large amount of grid cells (exterior objects) would be culled. Which means in the worst case that the whole idea of preserving context would be lost. In the current approach the grid cells are discretely classified, i.e., they are either entirely rendered or culled. From the domain point of view, a classification based on the grid properties is the most meaningful, so after defining the desired range of property values which should be visualized, all grid cells will be marked as selected (interior object) or unselected (exterior object).

A Different approach is done by Lidal et al. [33] where they assume that there is always a single cut in the entire scene by advocating the use of a cutting box as a guide to the cut. Also he applies shading models to improve the perception of depth while in our case, as we are dealing only with the CPU, it would reduce the scalability and performance. Diepstraten et al. [12] focus on following both sub-classes (i.e., breakaway and cutout) and propose the use of shaders, and it differs from our approach which is based on the CPU and we abdicate one of the rules in order to reduce the amount of removed data. Li et al. [30] also advocate the use of shaders and they focus on creating cutting volumes that their shape depends on the model geometry. They also follow the constraints of having a single hole to define a cut.

3.1 Approach

Cutaway is usually achieved with two or more distinct objects, in a way that one object is surrounded or at least occluded (positioned between the view and primary object) by others. In this perspective, it becomes clear which object is to be focused and which one will have eventually some of its parts removed. In our work, all objects are composed by grid cells and knowing that, as mentioned in Section 2.2 the relationship between objects of primary and secondary importance can be changed. Also different from previous research [33, 30, 7], we have cases where there are many primary objects and their numbers might also change. Therefore, some dynamics in the implementation must be provided. Some implementation strategies of this technique work defining the pixel as unit, so roughly the decision on whether to draw or not is performed in the level of pixel, mostly done by advocating the use of pixel shader. Usually a volume is used as a reference to the cut and it is sent through all rendering pipeline like other vertices and then in the pixel shader pixels can be culled or not based on the final projection of this cutting volume. In our work, we apply all the calculation in the CPU (single thread algorithm), and in order to keep balance between performance and workload we consider the grid cell as unit. In other words, instead of working in a pixel level or even in a vertex level [12, 30], we make a single decision for the whole grid cell by using its centroid. As we adopted the approach of using a CPU-based algorithm and in order to provide a real-time animation each grid cell is only evaluated by its centroid, therefore, instead of eight comparisons per cell only one is performed. The following pseudo code shows how a decision is made:

```
\\Cutaway-View
Input-------------------------------------------------------------
Selected \\ List of selected cells.
Unselected \\ List of unselected cells.
Camera \\x,y and z coordinates representing the camera position.
```

```
Aperture \\ scalar value representing the aperture of the cut.
Begin-------------------------------------------------------------
Foreach Cell s into Selected
| sCentroid = s.Centroid
| Foreach Cell u into Unselected
| | uCentroid = u.Centroid
| | distance = ||CrossProduct[ (uCentroid - sCentroid),(uCentroid - Camera) ]||
| | distance /= ||sCentroid - Camera||
| | if (distance < Aperture)
| | | DoNotDraw(u)
| |
|
End---------------------------------------------------------------
```

In order to define which unselected cells should be culled the following scheme is proposed. From the view point a ray is traced for each selected cell (targeting its centroid), and the distance between each unselected cell and the ray is computed. If the distance is greater than a user-defined threshold the cell is drawn; otherwise, the unselected cell is culled (see Figure 3.1). Our algorithm starts with the complete corner point model where each cell is colored according to its property value (Figure 3.2(a)). The user defines a range of the property to be tracked during inspection, and the cells that fall within this range are considered as interior objects (Figure 3.2(b)), while the remaining ones are considered as exterior objects. Figure 3.2(c) illustrates this approach, when interior objects are not located on the boundaries, in most cases it becomes impossible to visualize them just by applying transparency. On the other hand, Figure 3.2(d) illustrates how the cutaway technique visually provides a clear path to the interior objects, and at the same time preserves context by drawing most of the exterior ones. In order to validate our proposal, we applied the discussed techniques to

different reservoir models, as illustrated in Figure 3.4, which differ in shape and number of cells. By definition the the size of the opening, the aperture, should large enough to see all primary objects, in this way the amount of removed data is minimized. However, depending on the kind of data or even on the purpose of the data analyst larger apertures might be used. Figure 3.3 shows different aperture settings for the same set of objects and view angle. On the other hand, Figure 3.5 show different view angles for the same set of objects and aperture.

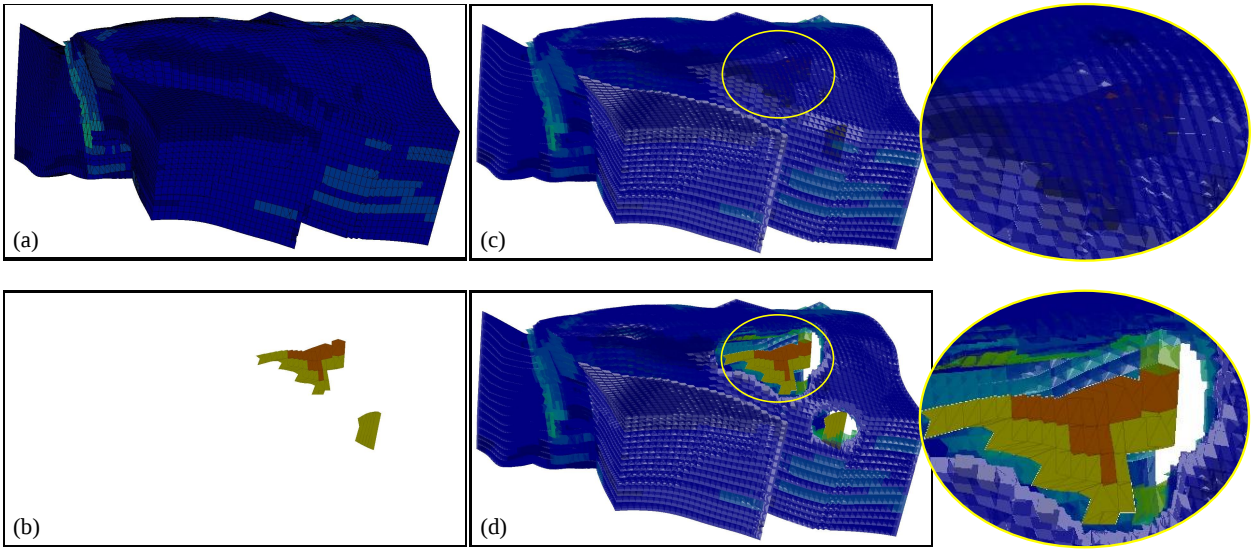

Figure 3.2: Opnine corner point model: 76000 cells; (a) complete corner point model, (b) selected cells, (c) without applying cutaway, (d) applying cutaway.

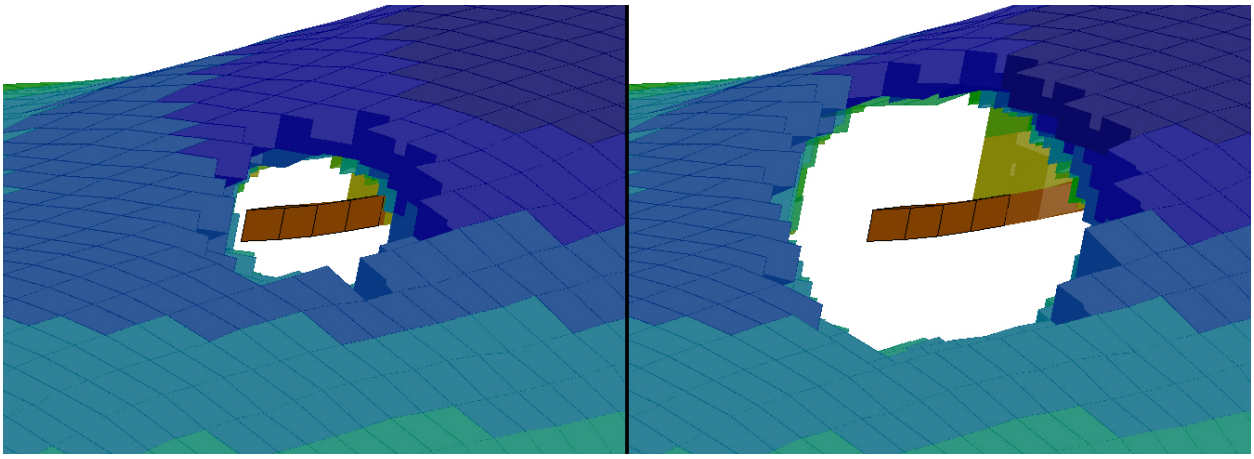

Figure 3.3: Aperture Settings: (left) minimizing the lost of data; (right) larger aperture.

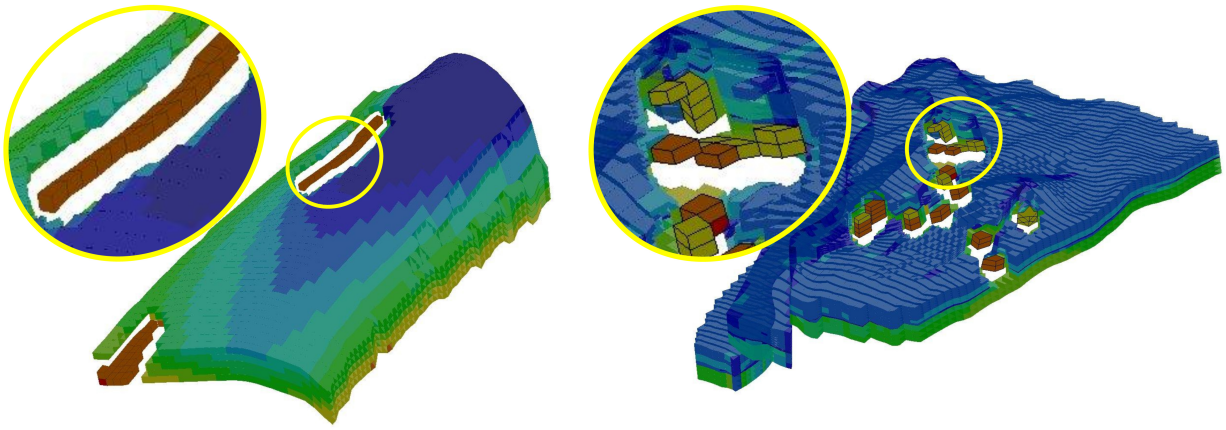

Figure 3.4: Two corner point visualizations using cutaway. (left) Zmap, 7500 cells; (right) Emerald, 72000 Cells.

Although the current implementation has achieved satisfactory visual results for different corner point models, there are still some limitations.All comparisons are based on the grid-cell's centroid, each unselected cells is tested against all selected ones and only those which do not occlude any selected grid cells are drawn. The algorithm's complexity is m x n where m is the number of selected cells and n is the number of unselected ones. In our experiments, we observed that the cutaway implementation works well when the total (selected and unselected) number of cells is up to eight thousand, we implemented On a Windows 7 PC with Intel Xeon E5620 Processor and 12 GB RAM. For larger amounts, the increase in the number of comparisons compromise the visual effect. Furthermore, as a consequence of working with a discrete classification of the grid cells (either entirely drawn or culled), there is no way to refine and/or explore the cutting within the cell; in other words, there is no way to work with grid cells as a continuous object.

3.2 Conclusions

We advocate the use of the cutaway technique and we expect that it might visually improves the inspection of geological data in the oil and gas domain, as well as other application areas that share similar geometrical structures. Our objective was to explore technical aspects of

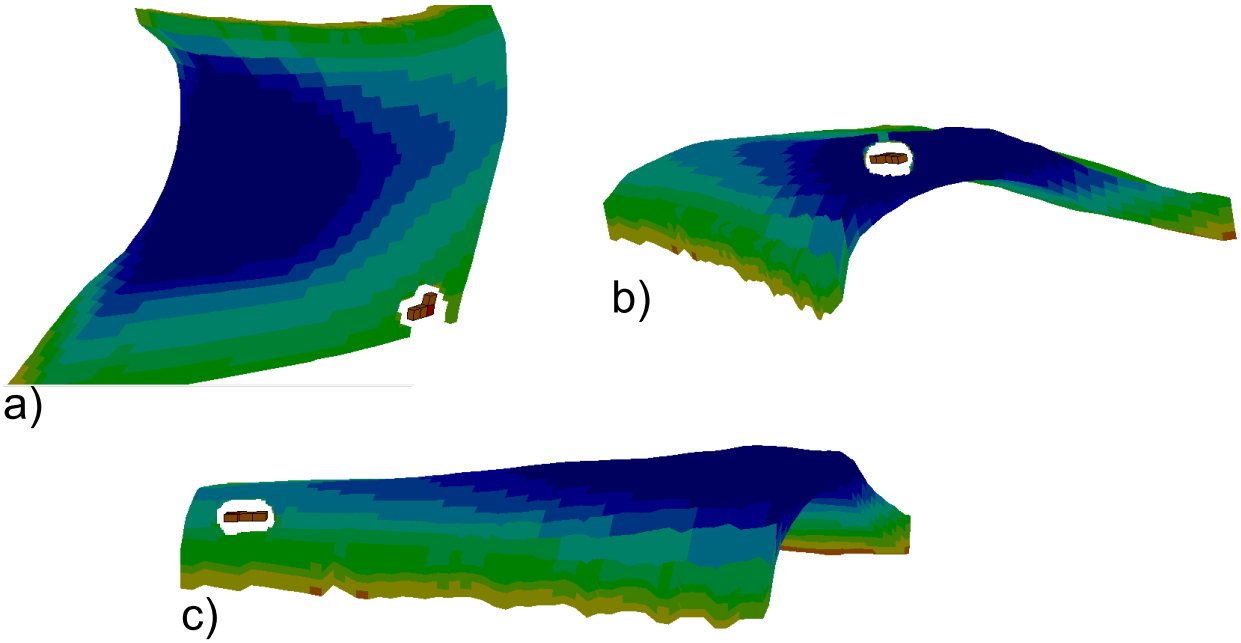

Figure 3.5: Different gazes for the same arrangement of primary and secondary objects.

this technique allowing then later to be extended to the aspect of the visualization aspects. We have described how to select regions of interest in a corner point model (simulated data), using the well pressure information (measured data). To visually correlate these representation we employ technical illustration methods to investigate the corner point model. More specifically, we show that cutaway is an efficient way to emphasize the desired cells without loosing context. Even though the described implementation provided satisfactory results in regards to what was anticipated, we believe that there is an open avenue for research in this direction.

Chapter 4

Exploded Views Based on a Mass-Spring System

In this chapter we present our fist approach for the application of Exploded view for 3D grids. In the concept of 3D grids, each vertex is a mass while each connection between vertices are springs. By setting a mass-spring system within the grid, the system always tries to keep its original state and deformations and animations can be obtained by the system without additional processing. As a consequence, by changing the position of some vertices, all other vertices are adjusted by the system itself. Different from the cutaway (Chapter 3), the main goal is to expose occluded cells without the removal of any other cells. In this approach we achieved this by designing three different styles – one, two and four corner deformation. This work is limited to support only regular grids with limited amount of cells.

As discussed in Section 4.4, after we generated results we could evaluate its drawbacks and then we decided to put effort toward other approach instead of extending the algorithm to corner-point and keep exploring more in this direction. For instance, changing the amount of cells has a huge impact in the system behavior and performance as well. By keeping the numerical equations with the same behavior we could focus on the whole picture and evaluate the technique itself.

The remaining of this Chapter is organized as follows: Section 4.1 describes how the mass-spring was applied to achieve our goal. In section 4.2 UI components and user interaction are detailed, the design and implementation are described, followed by results which are found in section 4.3, conclusion in section 4.4 and further implementation.

4.1 Approach

A mass-spring system is implemented over the grid. To achieve this, the first step is to store the initial position of all vertices, so whenever there is a change in any vertex position, with the current position and initial one the resulting force can be calculated for each vertex, like the following:

$$F_{result_i} = F_{current_i} - F_{damp_i}, \text{ where}$$
$$F_{current_i} = \sum_j K\left(L_{i,j} - l_{i,j}\frac{L_{i,j}}{|L_{i,j}|}\right) \text{ and}$$
$$F_{damp_i} = -c_{damp}v_i.$$

$F_{current_i}$ will be evaluated for all neighbors j of the vertex i, K is a constant stiffness value, $L_{i,j}$ is a vector between vertex i and the neighbor j in their current position while $l_{i,j}$ is the length of the vector between vertex i and the neighbor j in their initial position. F_{damp_i} is calculated by multiplying c which is a constant viscous damp value times v_i which is the velocity of the vertex i in the current time. After having F_{result} of all vertices their new positions can be obtained using Euler method with the following equations:

$$a_i = \frac{F_{result_i}}{m}, \tag{4.1}$$

$$v_i = v_i + a_i\Delta t, \tag{4.2}$$

$$p_i = p_i + v_i\Delta t. \tag{4.3}$$

m is a constant value representing the mass, Δt is a constant time value, p_i is the position of the vertex i and a_i is the acceleration of the vertex i. More details about equations used in the mass-spring system can be found at [24]. The following pseudo code shows how the deformation is calculated through the time-steps:

```
\\Mass-Spring deformation
Input-------------------------------------------------------------
original \\List of the original position of all vertices.
current \\List of the current position of all vertices.
fixed \\List of the indices of fixed vertices.
force \\List of the resultant force of all vertices.
velocity \\List of the velocity value of all vertices.
total \\number of vertices.
size \\ number of grid cells.
cDamp \\ constant viscous damp.
dt \\ constant time-step value.
m \\ constant mass value.
k \\ constant stiffness.
grid \\ the grid itself.
Begin-------------------------------------------------------------
//This step calculates the resultant force for all vertices
//based on their neighbors
From i = 0 to size - 1
| cell = grid.cell[i]
| From j = 0 to 7
| | index = cell.vertex[j]
| | if index is not into fixed
| | | From k = 0 to 7
| | | | index2 = cell.vertex[k]
| | | | if index is different from index2
| | | | | vCurrent = current[index2] - current[index]
```

```
| | | | | vOriginal = original[index2] - original[index]
| | | | | force[index] += k . (vCurrent - ||vOriginal|| . vCurrent / ||vCurrent||)
| | | | | force[index] -= cDamp . velocity[index]
| | | | k += 1
| | |
| | j += 1
| i + 1
//This step calculates the position for all vertices
//into this current time-step
From i = 0 to total - 1
| if i is not into fixed
| | acceleration = force[i] / m
| | velocity[i] += acceleration . dt
| | current[i] += velocity[i] . dt
| i += 1
End-------------------------------------------------------
```

Having the mass-spring system defined, the next step is to define the layer to be cut, for the clusters-based deformation the highest row value i is chosen. The vertices of the cubes which are shared with those from the next row are duplicated in order to remove the dependency of the cubes in the rendering (i.e., there is no more shared vertex between the cubes) and the deformation aspects (i.e., the mass-spring model do not consider them as neighbors anymore). For the same reason, the cluster's vertices will be duplicated and after this step, the grid is ready to be deformed.

For the cluster-based methods, the deformation is done by re-positioning some of the grid corners in order to create the opening, fixing them and running the mass-spring algorithm for

the non-fixed vertices. The number of corners to be changed will depend on the deformation type applied. For the one-corner method, the closest corner to the cluster has its position changed while the other are kept unchanged. The two-corners method changes the two closest corner and the four-corners method changes the four closest ones. Only the corners located above the cutting layer are taken into consideration. Those methods require the user to move the camera in order to find the best view of the selected cells. The amount of change will depend on the height of the cluster.

After the first implementation, the four-corners method was updated in order to be a view-based method and now the cut layer and even the axis depend on the relation between the cluster and view position. After selecting the cluster's cells, it centroid is calculated and a ray is traced between this centroid and the camera position. The grid boundary is composed of six planes, among those which intersect the ray, the closest to the camera position is chosen. The intersection point between the chosen plane and the ray is used to find the two closest cubes and as they are adjacent, they share two of the three axes (i.e., i, j, k) and then the different axis is defined to be cut. This solution has limitations that is going to be discussed in the next sections.

4.2 Implementation

The UI (see figure 4.1) is composed of two child windows. The left one contains all UI controls, this window itself can be decomposed into four control groups defined by separators. The first group is the grid settings one, the user can type the amount of cubes in the directions i,j, k and after pressing the "Generate" button the new grid will be create and displayed in the right child window. The second control group contains a list of all grid cells and it is the place where the user select the cells which will form the clusters. The third group contains the deformation types (i.e., one-corner , two-corners and four-corners deformation) where the user must select one option. The last group is formed by a single button (i.e.,

"Toggle Effect" button) which enable and disable the deformation effect. Regarding the right window, it shows the grid in the current state allowing the user to move around the grid, zooming in/out as well.

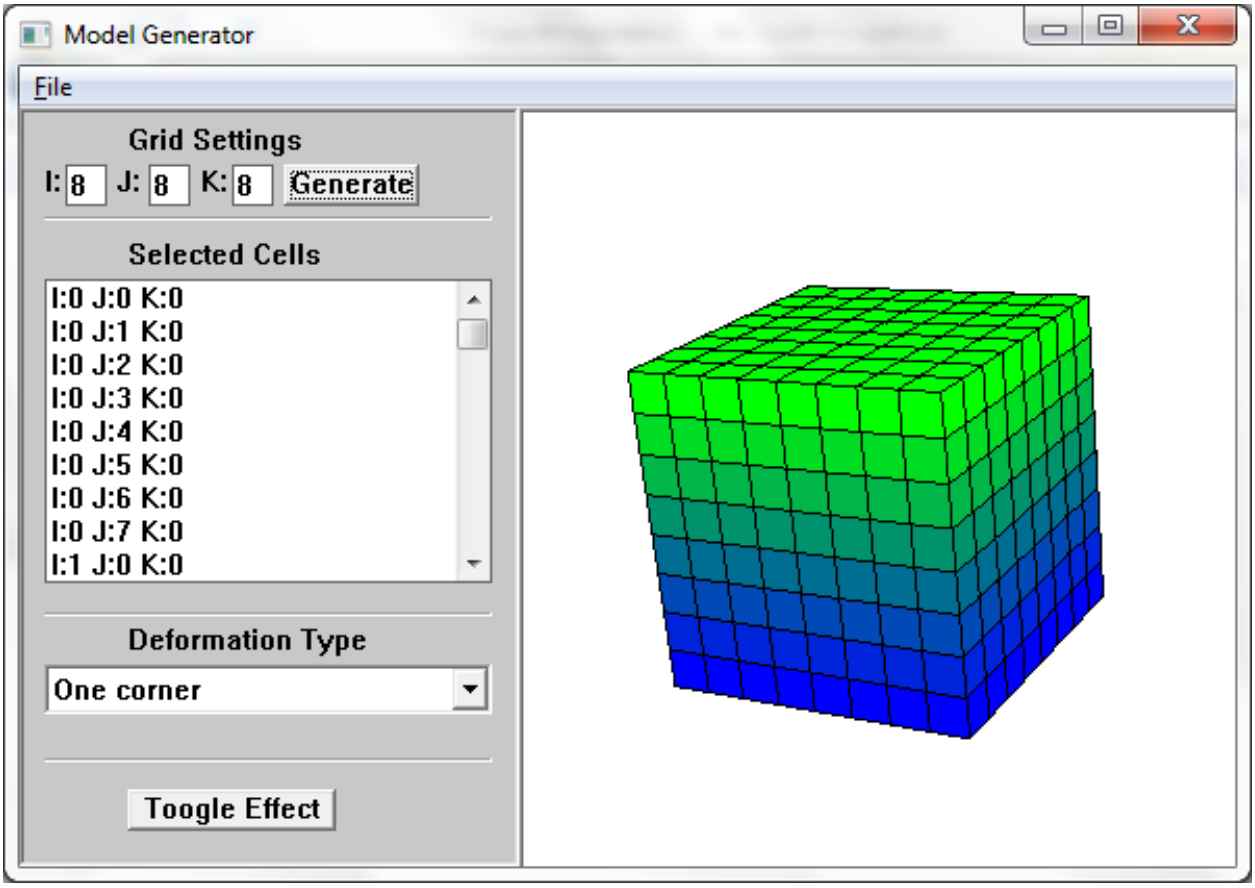

Figure 4.1: User Interface.

The general developed architecture could be split into two library projects and one main executable project. The first component is the WinApi library which encapsulate the main components of the Win32 API implementing re-sizing handling for more than one child windows, a manage to create and control UI components (e.g., button, listbox, combobox, label, separator) and the feature of delegating through the encapsulated message pool. The second implemented library was the DirectXCore which encapsulate all DirectX 11 features used in this work. It is divided into two namespaces: (1) DirectXCore which contains implementation of internal DirectX structures (e.g., vertex buffer, index buffer, devices, viewport) and

(2) DataCore which contains the implementation related to the specific application data to be rendered. Those libraries are included into the main project which intermediate all the data/message exchange among the namespaces. It contains the main logic of this specific application.

The code has been written in C++ targeting Windows 7 operational system. Two main libraries were used, Win32 API allows communication between the application and the O.S., by communication it means, accessing message pool for input/output handling, full control over window's resources and native multithread features (although not included in this implementation they are worth to be mentioned). DirectX 11 provides access to the video card in a non hardware dependent layer, it also includes HLSL (i.e., High Level Shader Language) which allows shader's code to be written in a C-language style. For this work, the shader model 4.0 was used for pixel, vertex and geometry shaders.

To avoid redundancy, vertices are shared among the mass-spring grid cells, in other words, a grid cell contains pointers to eight vertices and at least one of them is shared with at least one neighbor. This can be done by using the vertex buffer with unique vertices and an index buffer to indicate which vertices are used in each primitive. It introduces a problem of coloring cubes differently while they sharing some vertices, and as a solution for this, the coloring is being done at the geometry shader level (i.e., after the vertex shader and before the pixel shader), the idea is that for each primitive which arrives on the shader, by knowing which cube it belongs the color is then assigned to its vertices. The color information is sent to the shader once in a 3D texture format.

A trackball camera was implemented to allow the user to move around the grid, its values are used as an input to the view matrix. Zooming in/out operations do not change the camera position while it just change the field of view angle in the projection matrix (i.e., a perspective projection is used).

4.3 Results and Discussions

In this section we present the results based on this work. Although there are more than one way of applying the effect, a common information among them is that the red cells are the occluded cells (i.e., cells which the user wants to expose), cells colored from green to blue are occluders which are the context and all figures in this section go from an initial state *a*) to a final state *c*) going through a transition state *b*) where details about this states are found in each figure.

Figure 4.2 to 4.7 show the result of cluster-based methods, in other words, cuts which don't take the view position into consideration. Those figures are result of the three deformation types: (1) one-corner deformation (see figure 4.2 and 4.3), (2) two-corners deformation (see figure 4.4 and 4.5) and (3) four-corners deformation (see figure 4.6 and figure 4.7). Figure 4.8 to 4.11 show the result of view-based method, exploring different cutting axis according to the view position and details about which plane was used are found at each figure.

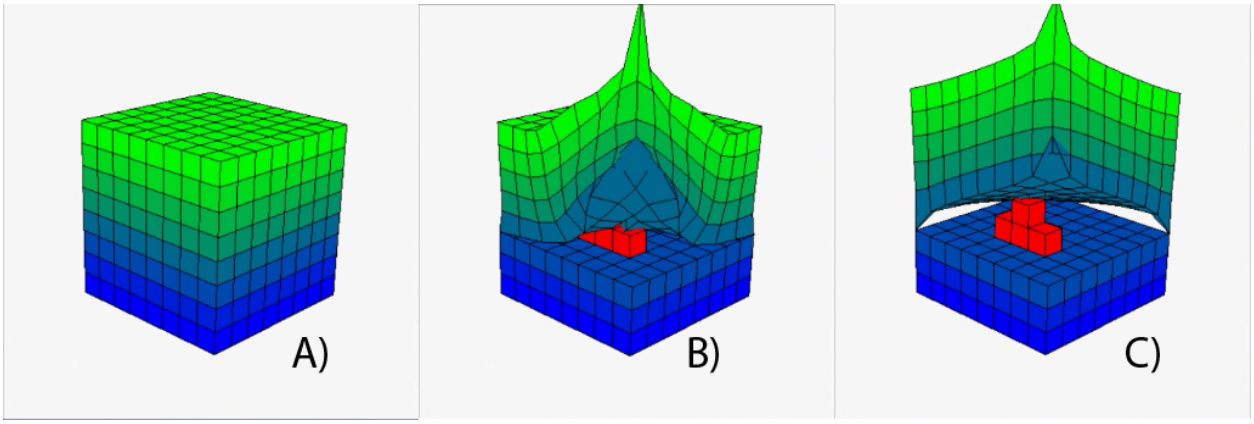

Figure 4.2: one-corner deformation: A) initial grid, B) transition snapshot and C) deformed model.

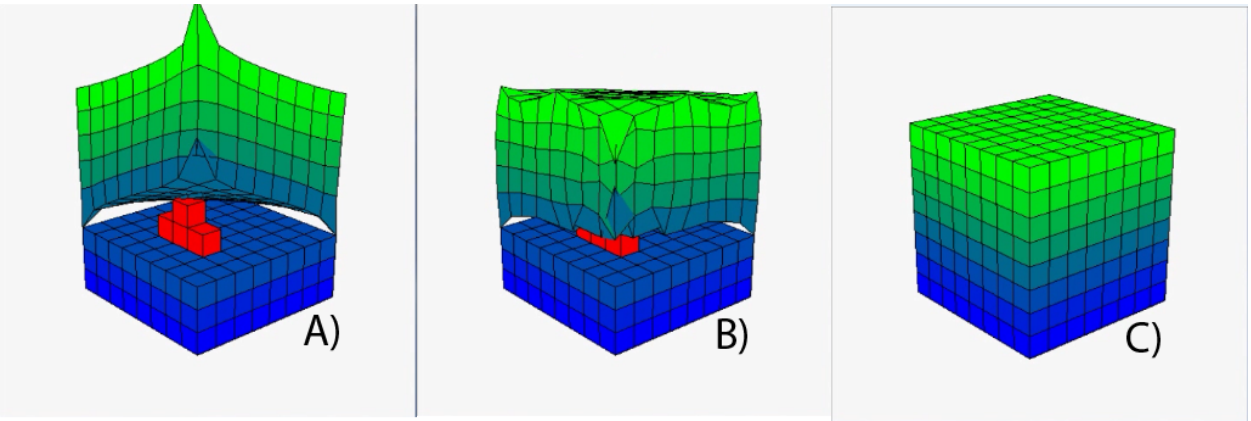

Figure 4.3: one-corner deformation: A) deformed model, B) transition snapshot and C) restored grid.

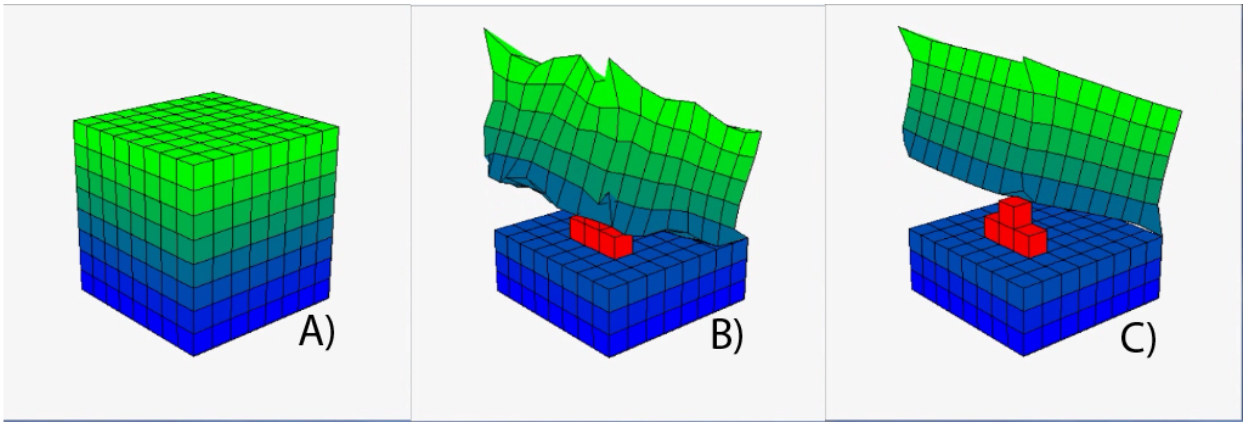

Figure 4.4: two-corners deformation: A) initial grid, B) transition snapshot and C) deformed model.

4.4 Conclusions and Discussions

Among the two types of method, the one that just take into consideration the cluster position (i.e., cluster-based) works well with all deformation types (one-corner, two-corners and four corners) but it has the disadvantage of requiring the user to move the camera in order to see the exposed cells. The second type is the one which select the cut axis based on the view position (i.e., view-based) and therefore just works with the four-corner deformation type. It has limitations as it would fail (in terms of exposing occluded cells) in more generic cases. A more robust approach could be done by defining the opening through a cutting plane not necessarily aligned with any axis (i.e., i, j, k).

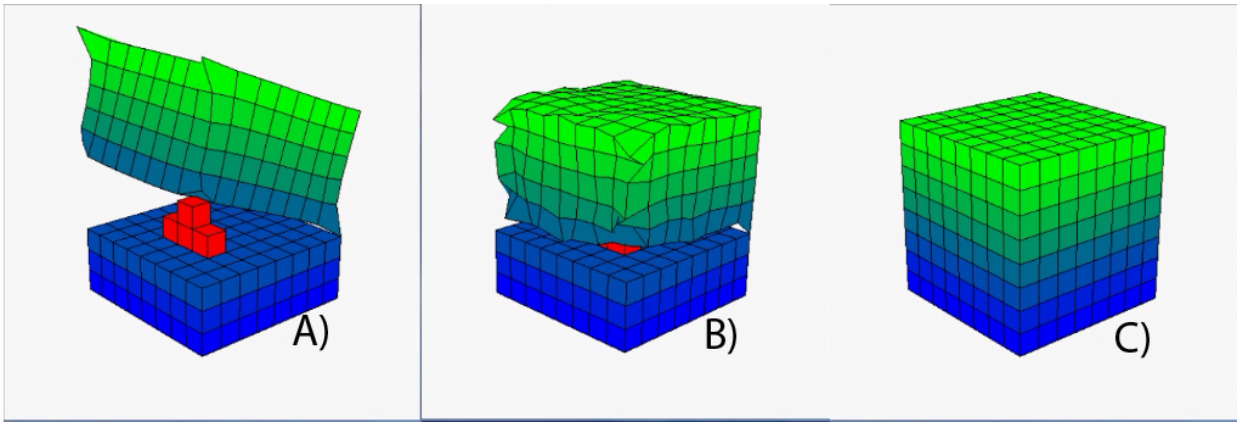

Figure 4.5: two-corners deformation: A) deformed model, B) transition snapshot and C) restored grid.

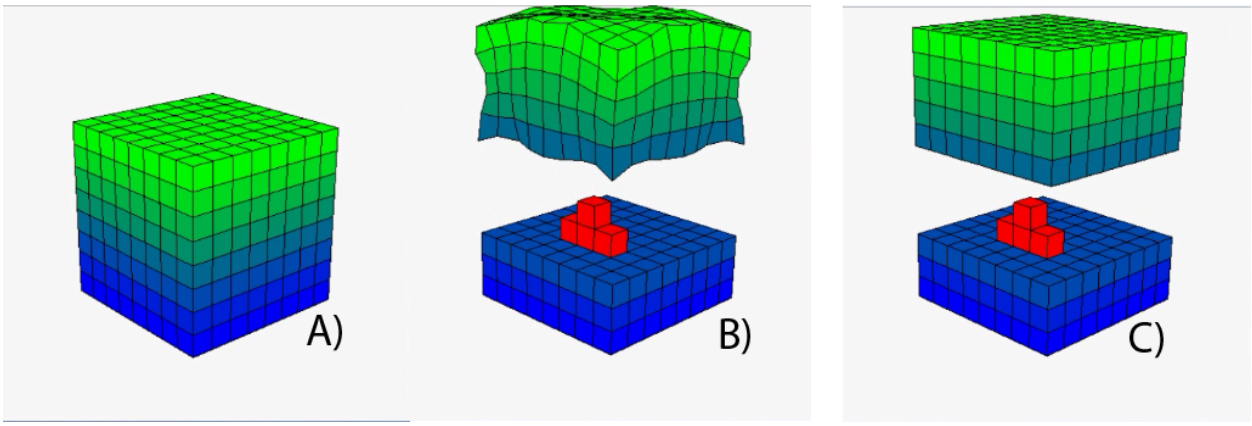

Figure 4.6: four-corners deformation (cluster dependent): A) initial grid, B) transition snapshot and C) deformed model.

The behavior of the mass-spring system changes according to the grid's dimension and their relation, therefore depending on the amount of cells the parameter's values (i.e., t, m, k, C_{damp}) need to be reset. When it is not done, the mass-spring system can become very slow (due to the increasing amount of computation) and/or unstable.

As the corner-point grid can have millions of cells, other ways of achieve the final goal must also be investigated, more than one solution must be taken in consideration mostly when dealing with multiple clusters.

As a same set of mass-spring's parameters might not be the best fit for all cases, an improvement could be done by creating an heuristic in order to set those parameters dynam-

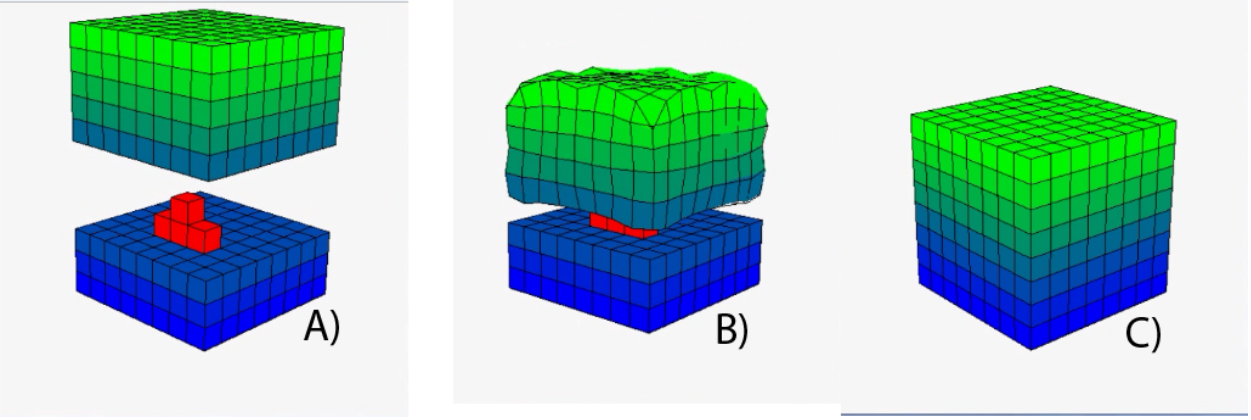

Figure 4.7: four-corners deformation (cluster dependent): A) deformed model, B) transition snapshot and C) restored grid.

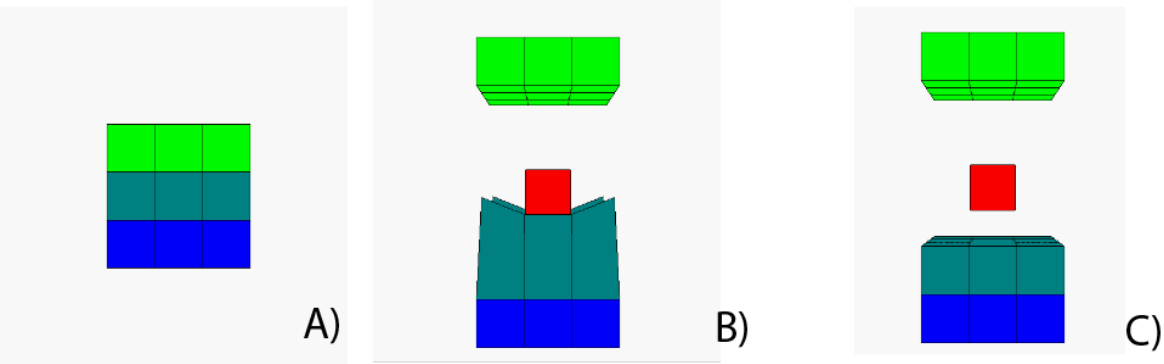

Figure 4.8: four-corners deformation (view dependent): A) initial grid, B) transition snapshot and C) deformed model. Note that the cut was defined at $i = 0$ (i.e., first row).

ically. Also, the current solution supports single cluster and clusters which share at least one index (e.g., i, j, k), but to support more general scenarios the algorithm must also be able to support multiple clusters. Those are the main issues to be addressed in the future and therefore, a final evaluation could be made in order to decide whether it should be integrated in the main tool or not.

The goal of exposing occluded cells while not removing any other cells was achieved. The main components are implemented, and the algorithm allows the execution of many tests in order to stress and evaluate the exploded diagram effect in 3D grids, to identify the performance issue when applying numerical methods to a larger amount of grid cells, and

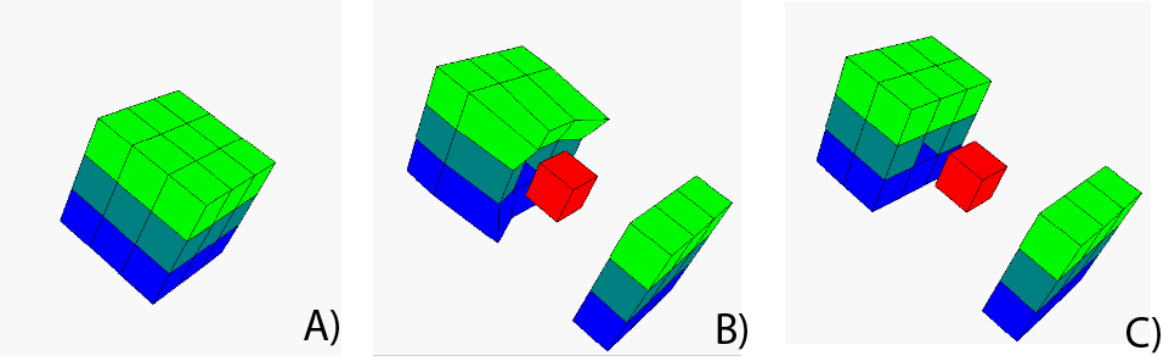

Figure 4.9: four-corners deformation (view dependent): A) initial grid, B) transition snapshot and C) deformed model. Note that the cut was defined at $j = 1$ (i.e., second column).

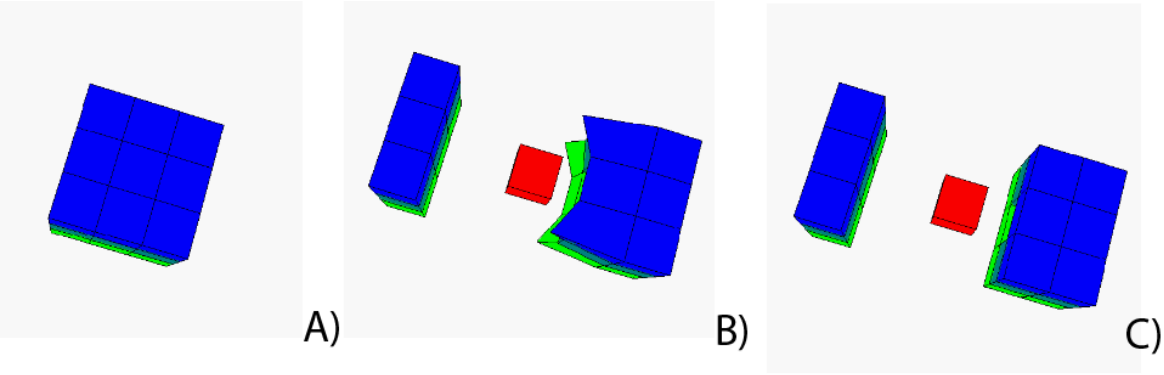

Figure 4.10: four-corners deformation (view dependent): A) initial grid, B) transition snapshot and C) deformed model. Note that the cut was defined at $j = 0$ (i.e., first column).

to provide a better understanding of the problems and other possible ways to tackle them. In the next chapters 5 6, we provide another approach to implement the Exploded View Diagrams technique in both regular and corner-point grids.

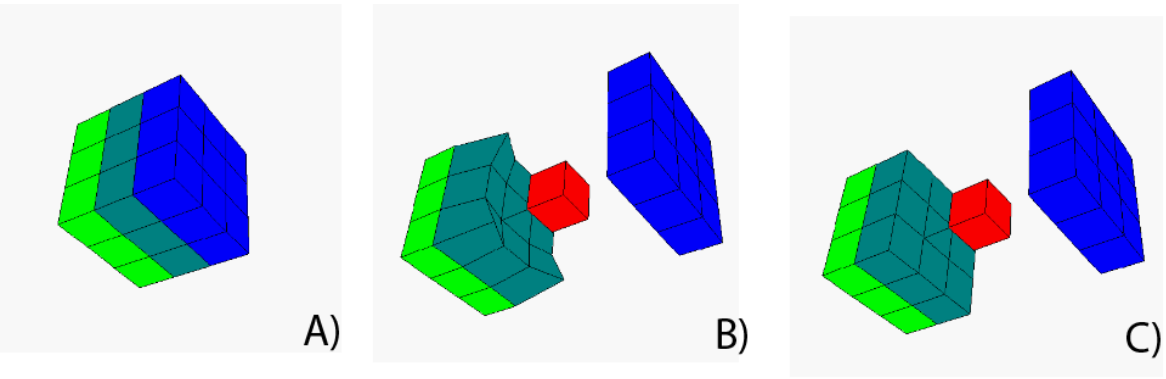

Figure 4.11: four-corners deformation (view dependent): A) initial grid, B) transition snapshot and C) deformed model. Note that the cut was defined at $i = 1$ (i.e., second row).

Chapter 5

Explosion Tree

In this chapter, we propose, Explosion Tree, a data structure which receives primary-object information and view angle as input and returns a final position to all vertices (including those that belong to the secondary object) as an output. The goal is to create a final effect similar to technical illustration based on the traditional exploded views technique [11], having limited amount of information in a dynamic environment. Although most of previous work [47, 31, 1] are based on a data structure [1] which requires information related to all parts, how they are connected to each other and reasonable preprocessing stage, in this work we try to minimize the amount of calculation as we cannot rely on preprocessing. This chapter is organized as follows: Section 5.1 briefly present the BSP-tree which is a data structure very related to ours. Section 5.2 introduces the Explosion Tree in terms of design and implementation. And Section 5.3 discusses about the performance and briefly concludes this Chapter.

5.1 Binary Space Partitioning

Binary space partitioning (BSP) is a method for recursively subdividing a space into convex sets by hyperplanes. This subdivision gives rise to a representation of objects within the space by means of a tree data structure known as a BSP tree. Binary space partitioning was developed in the context of 3D computer graphics [17], where the structure of a BSP tree allows spatial information about the objects in a scene that is useful in rendering, such as their ordering from front-to-back with respect to a viewer at a given location, to be accessed rapidly. Other applications include performing geometrical operations with shapes (constructive solid geometry) in CAD, collision detection in robotics and 3-D video games,

ray tracing and other computer applications that involve handling of complex spatial scenes.

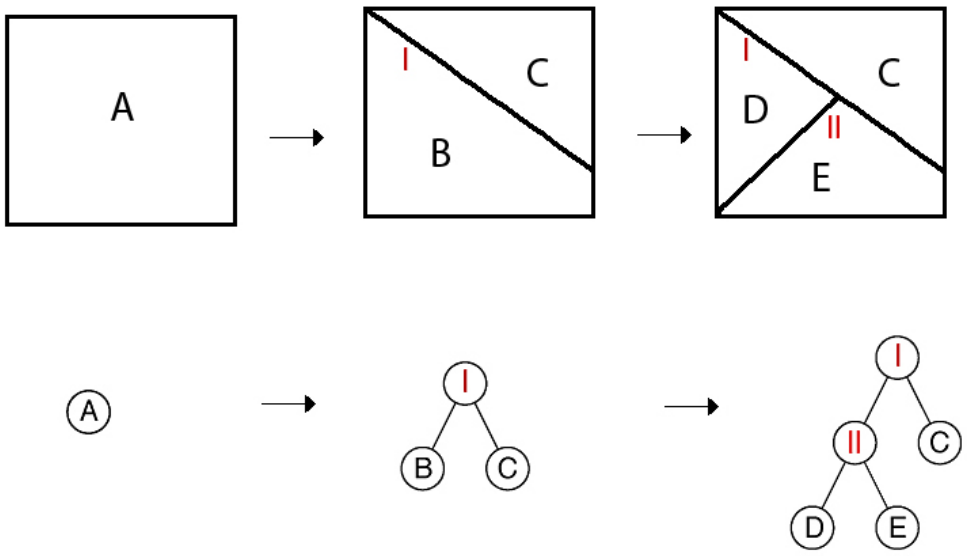

Figure 5.1: Construction of a BSP tree.

The construction of a BSP tree can be explained as follows: Consider a set of polygons in three dimensional space, and the goal is to build a BSP tree which contains all the polygons in between the set of planes. First select a partition plane and then partition the set of polygons with the plane. Recursively apply the first two steps on the new partitions of the set. In three dimensional space, planes are used to partition the space, while in two dimensional space, lines are used to obtain the sub-spaces. There is no universal criteria for choosing the hyperplane for partition but rather the criteria varies from application to application. A test is performed for the partitioning of polygons, i.e., each point in the polygon is tested against the plane such that if all the points lie on the same side of the plane then the polygon is not partitioned.

Figure 5.1 shows an example of how a BSP tree is constructed for a two dimensional object. For simplification only we are going to use a 2D object, therefore whether we consider the BSP tree composed by planes in 3D, in 2D it will be composed by lines. Consider the rectangular object aligned with the XY axis and it can be considered as having a single

region (i.e., A). For the first split, we create the first plane (i.e., I) which is the root of the tree, as a consequence instead of a single region, there will be two regions (i.e., B and C) and they are going to be considered as children of the root node corresponding to the two subspaces. In the same way that the first plane, I, was created based on the region A, the second split is going to generate a plane II that is based on the region B. Two new regions are going to be created as a result of subdivision of the region B, they are C and D. The process of splitting the polygon, a rectangular one in this case, is applied recursively until a stopping criterion is reached. Regarding the stopping criteria, some implementations use a minimum amount of objects per region while others prefer to define a maximum number of regions.

Building an optimal BSP tree is an NP Complete problem. The problem of building optimal trees can also be looked upon as the problem of splitting a tree versus balancing a tree. In some respects the two requirements for building optimal trees are opposed to one another. In most cases the intended use of the tree determines how these two factors should be balanced against one another. The complexity of the BSP tree also depends upon the application. Thus for hidden surface removal and ray tracing acceleration the complexity for the worst case is $O(n^2)$, and $O(n)$ for the average case, where n is the number of polygons [2].

5.2 Approach

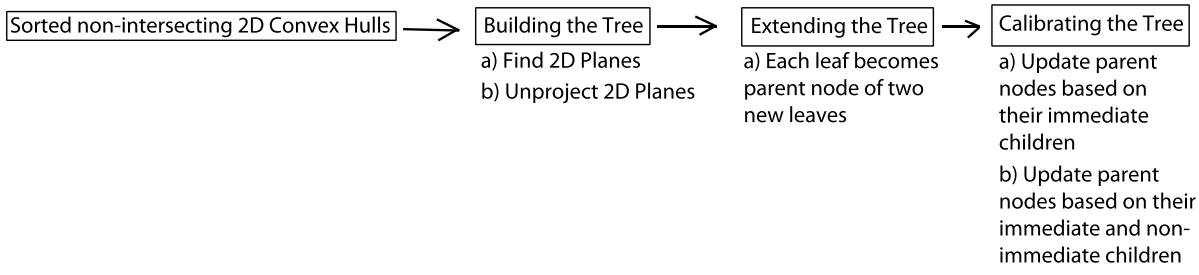

Figure 5.2: Tree overview: the main steps required to complete the tree and their respective sub-steps.

We molded the Explosion Tree to work with limited amount of information regarding the grid geometry and in dynamic scenarios. For some applications to calculate the whole geometric information is expensive or even impossible. Using this data structure we avoid these calculations, thus we do not compromise the interactive effects. Similarly, dynamic scenarios, where primary and secondary objects are not the same all the time, we cannot pre-process the data. In addition to this, each of the following steps need to be done every time that an object or the view angle change (see Figure 5.2). Therefore, it requires a light data structure to achieve an interactive application. To build the tree we assume that there is a collection of disjoint convex polygons, which are derived from primary objects (see Chapter 6). The main purpose of the tree is to guarantee that we can move some parts of the model without any overlapping. We have three major steps to build the tree: (1) create a binary partitioning space tree based on 2D information; (2) extend the tree to 3D object space; and (3) calculate offsets for all leaves.

The first step to build the tree is to find a hyperplane which does not intersect any polygon while subdivide the space into two regions. Given a point x_0 and a normal n, we define the hyperplane as:

$$\left\{x \in \mathbb{R}^d \middle| \langle x - x_0, n\rangle = 0\right\},$$

where $\langle \cdot, \cdot \rangle$ means a dot product between two vectors. This first hyperplane is the tree root. Afterwards, more planes are added to the tree until there is just one convex object per region, see Figure 5.3. Therefore, each plane is associated to two regions, one in the same direction as its normal (right side) and the other opposed to its normal (left side). Each tree node has one plane and each plane two children, knowing that each node knows which convex objects are associated to each of its regions. The only exception for the rule of one convex object per region is when no plane is found between two convex objects, in this case they share the same region. This algorithm is pretty similar to those used for constructing a BSP-tree.

Although there is always a plane that separates two convex objects, it is not true for

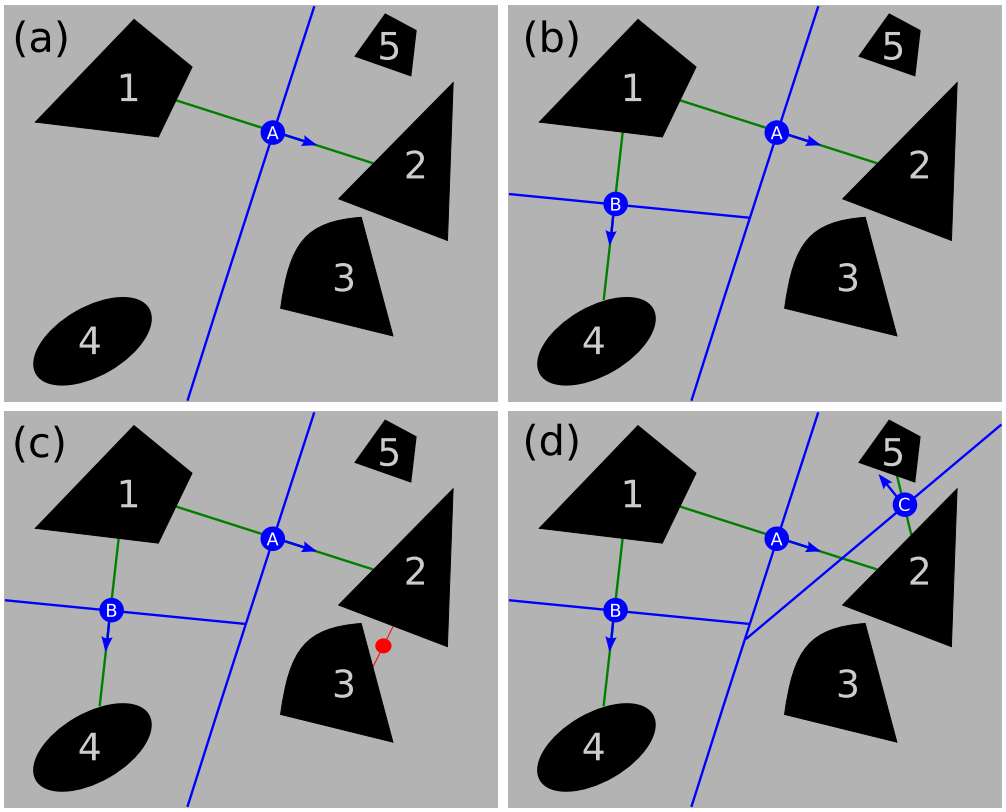

Figure 5.3: Build sequence (a) the first plane, A, is defined between objects 1 and 2; (b) on the left side of plane A, a plane B is defined between 1 and 4; (c) on the right side of A no plane is found between 2 and 3; (d) still on the right side of A plane C is defined between 2 and 5.

any set of convex objects. Also, calculating a separating plane between two objects which does not intersect any other objects of this region can be very expensive. We create an algorithm that compromises precision (as it does not test all possible planes) in order to prioritize performance (Figure 5.3 (c)). Given a pair of objects we use the line segment ℓ that connect their centroids to pick a point x_0 and one direction in order to define a plane that does not intersect any convex objects. The point x_0 is the middle point of the line segment $\ell - \{\Omega_1 \cup \Omega_2\}$ (Figure 5.4), where (Ω_1, Ω_2) is the pair of objects. To find a direction we rotate the normal by a constant angle (30 degrees) until we find the separating plane (Figure 5.5). For instance, in Figure 5.3 (d) the normal of the plane C was rotated to find the separating plane. In the end of this step, there is a transition from a set of 2D convex objects to a tree of 2D planes defined based on how the convex object list is sorted. The

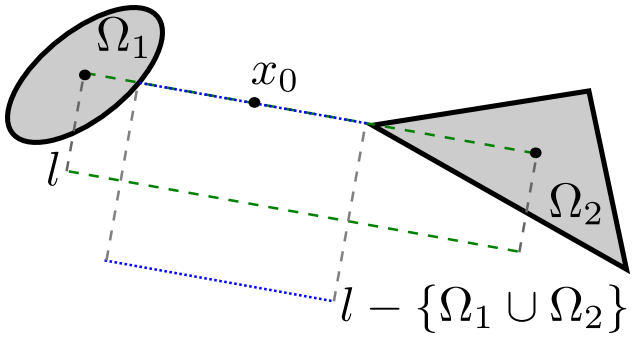

Figure 5.4: Point x_0 is defined as the point in the middle of the line segment $\ell - \{\Omega_1 \cup \Omega_2\}$.

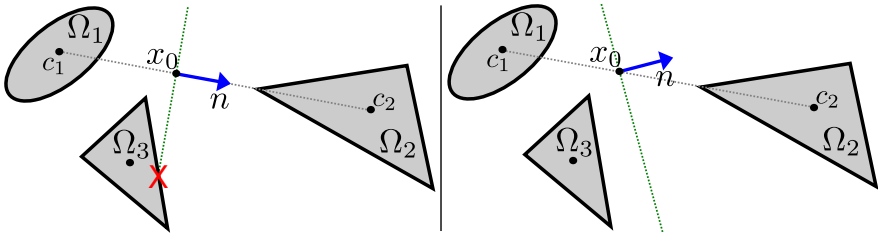

Figure 5.5: Plane definition:(left) the first normal n is defined as $c_2 - c_1$, however when it intersects polygons belonging to the same region it is rotated by thirty degrees and a new test is performed; (right) this normal does not intersect any polygon therefore, there is a valid plane between convex polygon Ω_1 and Ω_2.

data structure now hold the information of which convex objects belongs to each region and also the sequence of nodes to reach it from the root downward. Note that in order to keep the final effect as close as possible to the original Exploded View technique, we look for a separating plane only in the point x_0, in case there is not valid plane in this point we consider both objects are considered as a single one (sharing the same region) as shown in Figure 5.3c.

Next, the first operation to be performed is to unproject (orthographic) all planes from screen coordinate system to 3D object space, so where there were 2D planes now there are 3D ones. Although both the normal n and the point x_0 are unprojected to the depth of 0.5, as we are dealing with orthographic projection any depth would work. At this point, the tree will be extended, meaning that each leaf node originates two children. The difference between the extended nodes and the others is the fact that in order to define a plane they inherit the respective normal from their immediate parent while the point comes from the

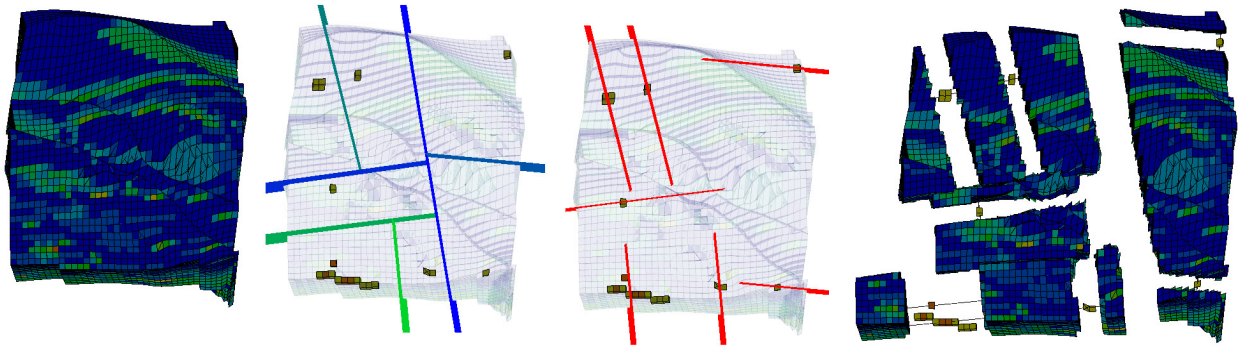

Figure 5.6: Petrel Original. Separating planes (from left to right): initial grid; clusters of grid cells within a subdivided space (first level of planes); planes within each cluster, defining how they will be exposed (second level of planes); after the effect has been applied.

cluster's centroid (in case of more than one cluster per region the centroid is the resultant of centroids within this region). These extended nodes can be seen at the third image of Figure 5.6. Figure 5.7 shows the relationship among the tree nodes. In this step planes are embedded into 3D space and each leaf node originates two extended nodes.

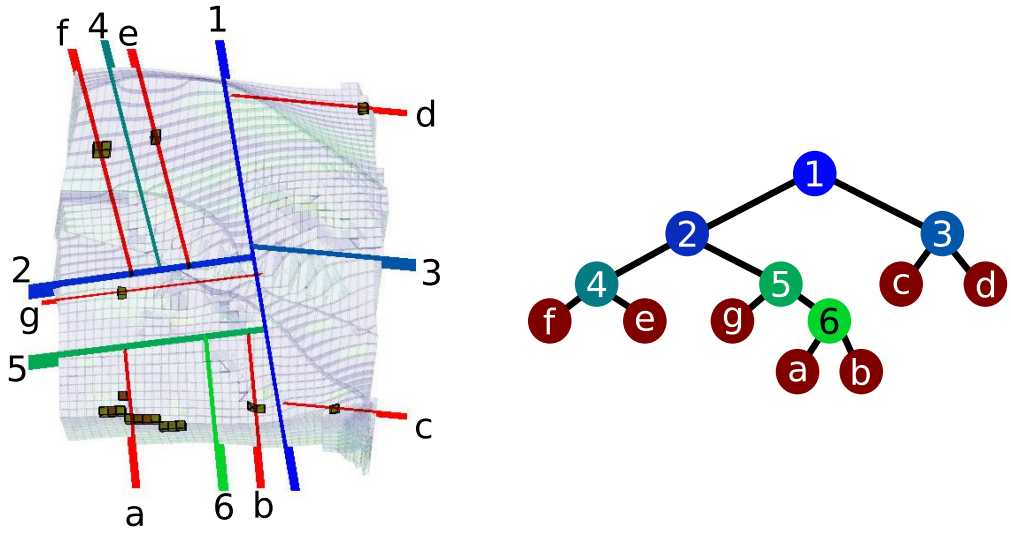

Figure 5.7: The tree structure: (left) original planes are drawn in colors varying from dark blue to light green while extended planes are drawn in red; (right) all nodes (including the extended ones) are arranged in a tree structure, with the same color of their respective planes.

Having all planes defined, although the direction of each grid cells is defined by the planes, one missing information is how much each plane is influencing each vertex position. to keep

track of this information each plane has two offset values (one to each side). While the planes were calculated in a top-down approach, offsets are calculated from the bottom up to the root. First, the offset needed to expose the primary objects in each leaf node (extended node) is calculated, and as leaf nodes they do not require information of any other node apart from themselves. As we cannot assume that primary objects have a symmetric relationship with their immediate planes, two offset values are required, one makes the influence in the normal direction and the other in the opposite direction. Having all leaf nodes updated, all parent nodes have their offsets calibrated based on their children where the final goal is to expose all primary objects without creating any overlay between primary and secondary objects. All nodes have their offset calibrated based on the following steps:

- Through a post-order algorithm, each node (non-extended) takes in consideration its children by calculating how much offset those children will apply in its direction. For instance, if a child C of P has an offset α_C in its parent direction (i.e., the dot product between the two normal vectors is less than zero), the parent node's offset need to compensate with an offset of at least $\alpha_C \langle n_P, n_C \rangle$ in a opposite direction, where n_P and n_C are the normal of the planes P and C respectively. This is done for all non-extended nodes up to the root node.

- Still based on a post-order traversal, but instead of updating the parent nodes just based on their immediate children, it gets all nodes that belong to the paths that of the current node down to all its related leaf nodes. For each path it sums all the components which turns into its direction and compare with the current offset value. The current offset is updated only if the total sum is greater than its current value.

Finally, the data structure has all required information and can give as an output the final position for any vertex of the grid, secondary or primary objects. It gives the final

position into two levels. First, we use the non-extended tree to calculate the final position for the primary objects and the partial position for secondary ones. The final position of the secondary objects is then calculated adding the offsets of the extended nodes. As a result of these steps, the exploded view diagrams effect can be achieved (see Figure 5.6).

5.3 Conclusion

In order to evaluate the tree's performance, some measurements were taken and we concluded that the bottleneck to build the tree is the total amount of comparisons to find all its separating planes. In a higher level, this amount of comparisons depends mainly on the gaze and the number of primary objects. For our typical case (i.e., up to 20 primary objects), the time to build the tree is insignificant (less than 2 ms), so to evaluate this bottleneck we created artificial examples with larger numbers of primary objects. In the worst case in terms of number of comparisons, Emerald original, it took 10 ms to build a tree with 270 comparisons for a grid with 33655 cells and 148 primary objects. Further work could be done on the calibration algorithm in order to minimize the distance between planes, as in cases where there are too many convex objects our solution is not optimum.

Chapter 6

Exploded Views Technique Applied to 3D Grids

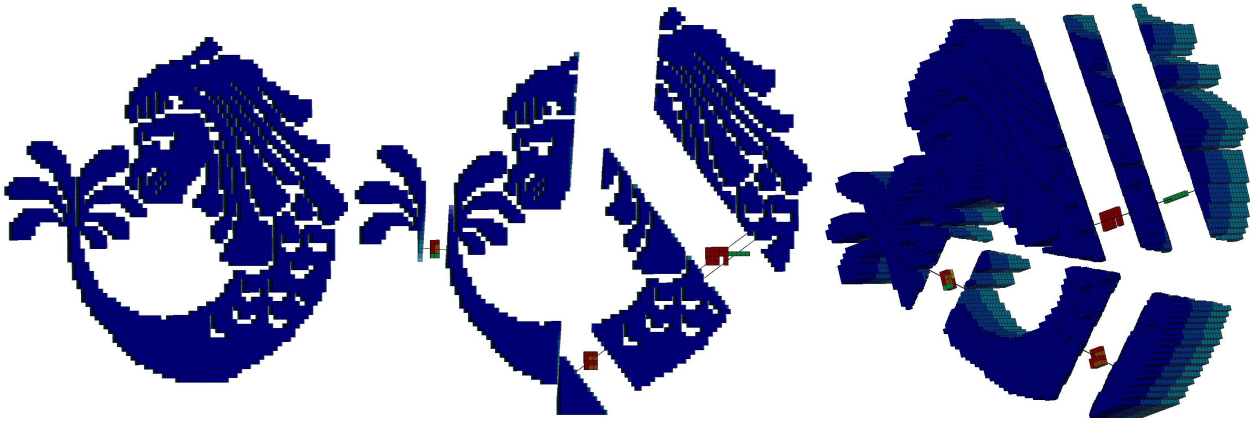

Figure 6.1: Exploded view diagrams applied to 3D grid (from left to right): the whole grid; the final result with primary objects exposed; another final result from a different view angle.

In this chapter, we propose a solution for the problem of visualizing occluded cells in any 3D grid by applying the exploded diagrams technique different from what is done in Chapter 4 (see Figure 6.1). In a 3D grid, a primary object is a group of contiguous cells which shares at least one vertex. The heuristic to choose cells can be based on cells' values or by their location, more information about selection of cells can be seen at Section 2.2 (Chapter 2). One grid can have more than one primary object and remaining context is considered secondary (see Figure 6.2). Since the selection of primary object and thus the corresponding secondary object is changeable, and assuming that the geometry of the secondary object is unknown, none of the previous approaches for computing explosion diagrams are applicable. Thus the novelty of our approach is to apply Exploded Diagram in 3D grids with no information regarding secondary objects neither part hierarchies.

In Chapter 5, we have introduced Explosion Tree, a data structure which receives primary-object information and view angle as input and returns a final position to all vertices (including those that belong to the secondary object) as an output. The goal is to create a final

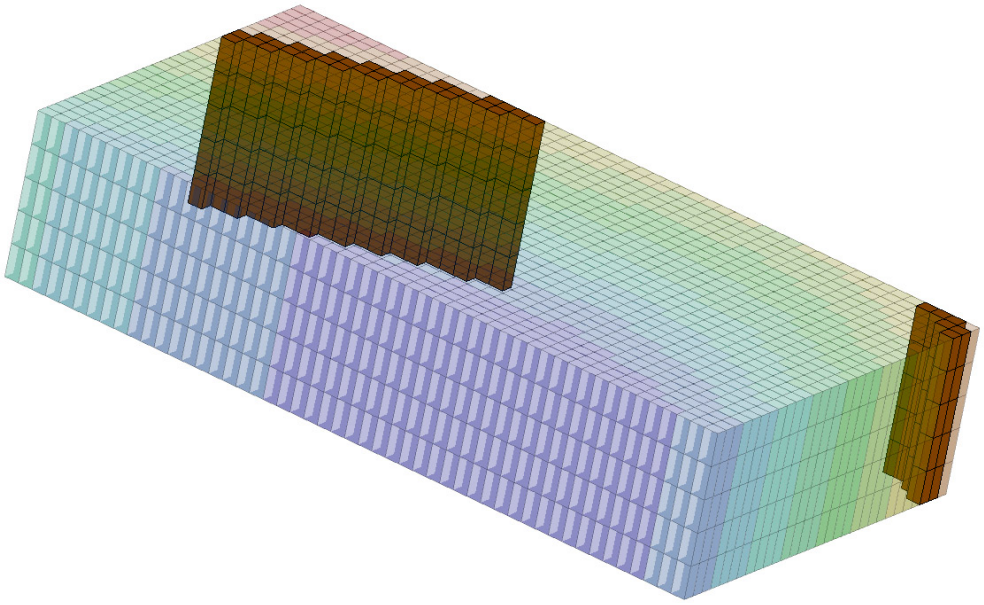

Figure 6.2: Object types: each primary object is composed by one or more contiguous grid cells which share at least one vertex (with full opacity) while Secondary objects are all remaining grid cells (with low opacity). In this figure there are two primary objects and one secondary object.

effect similar to technical illustration. We are inspired by existing traditional exploded views technique [11], however we propose ths application of it for datasets having limited amount of information in a dynamic environment. By dynamic environment we mean that the user can freely change the area of interest, unlike previous work. As previous approaches require the whole geometry information in addition to information regarding parts hierarchy, we want to achieve the same effect by using only our primary objects (objects in focus). Agrawala et al. [1] introduces a data structure which is used by most work, however this data structure assumes that all parts are statics, it takes the whole geometry into consideration. On the other hand in our approach, primary and secondary objects are all grid cells, we assume that the relationship between them can be changed dynamically (e.g., selection based on property distribution, geometric location), in this case we need to provide a flexible way of performing the effect whether parts remain the same or not. Li et al. [32] applies the exploded views technique in 2.5D, with static parts and a simpler approach (e.g., there is no occlusion) than

it would be required to 3D space. Brukner et al. [6] propose a physically based approach which assumes that there is just on primary object and the remaining model will also be split from two to four parts (i.e., divided by orthogonal planes). In our approach the amount of primary object can change all the time and most of time is larger than one, therefore, we cannot pre-define the amount of planes to be used. Li et al. [31] provide an approach based on their previous data structure which requires pre-processing to store the different view angles as it reduces the workload between the data structure and the wrapper. As our data structure only requires information about the primary objects, the amount of calculation is reduced, which allow recalculating without preprocessing stages. Also preprocessing stages assume that the relationship between primary and secondary objects does not change. To achieve this goals, we implemented a data structure which provides final position to all vertices based on the primary object information, and a implementation that wraps up this data structure providing an interface to the final user.

6.1 Approach

In order to validate and explore the *Explosion Tree*, an implementation was developed to visualize the results and present the interactions supported by our visualization. An overview of our approach is shown in Figure 6.3. The user can load 3D grids (regular and non-regular), rotate, zoom in, zoom out and pan the actual object. All actions related to navigation are done using a mouse. The exploded view effect is achieved by pressing the respective button and in case the effect is already applied it will restore the position of all vertices and reapply the effect. In cases where either the grid cells selection or the view angle change, those changes are going to be used when the effect is reapplied. Grid cells can be selected (to compose primary objects) based on different constraints (more information in section 2.2). In this work we present a few such selection criteria. However, it is important to note that there are several other criteria depending on domain knowledge and the requirements of the

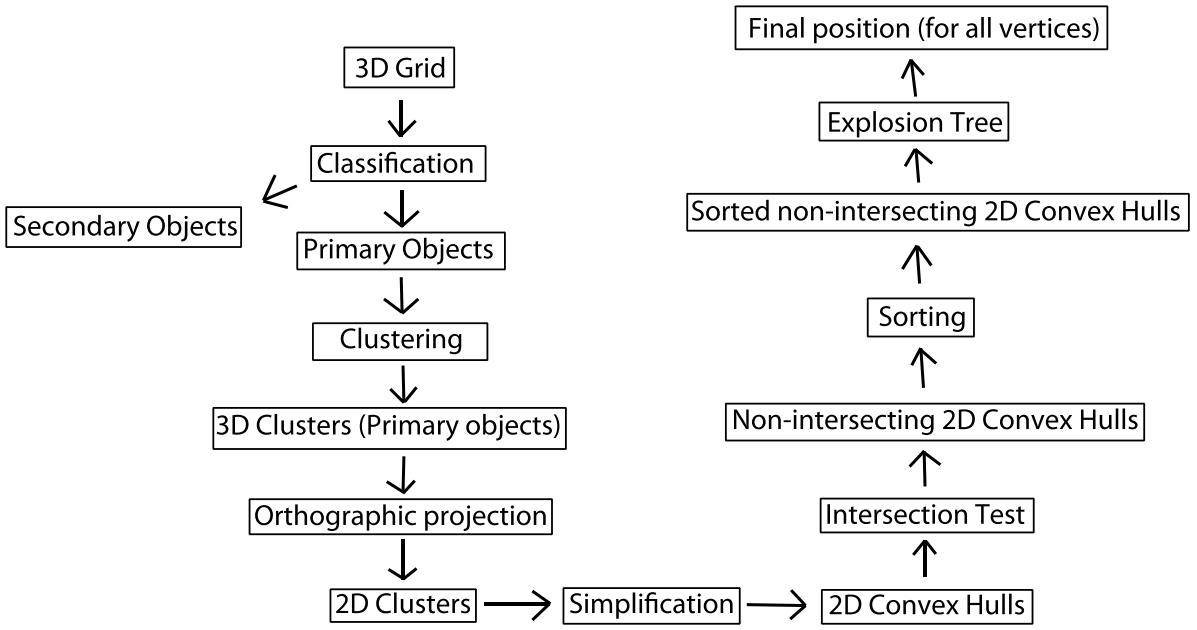

Figure 6.3: Overview: it takes several steps to obtain the final position for all vertices from the 3D grid.

user. One possible criteria is to have a value in each cell (in addition to its geometry), then selecting a range of values causes of all the cells that fall in that range to become primary objects while others secondary ones. Another criteria can be performed by specifying i, j and k index of a set of grid cells which are going to be defined as primary-objects. Either way, after the selection is performed all grid cells are going to fall under one of the two categories already mentioned. It is important to note that the only grid cells that are going to be taken into consideration for further processing are those classified to compose primary objects.

After having all grid cells classified, a 3D clustering algorithm is applied only to those cells set as primary object's members. The goal of the algorithm is to define which cells compose each primary object. More specifically, it runs through all grid cells and cells which share at least one vertex are defined into the same primary object. To convert primary objects into a set of convex polygons, each primary object is projected (orthographic projection)

to 2D screen coordinates, the convex hull is calculated (using Jarvis March) and stored into a buffer. For each buffer a pixel-based convex hull is created and then, an intersection verification is performed between the convex hulls. In cases where there is an intersection between two or more convex hulls a new convex hull is calculated based on all intersecting polygons and from now on it will be used instead of these intersecting convex hulls (see Figure 6.4). The meaning of merging intersecting convex hulls is to reduce the total number of objects passed to our data structure and to isolate the processing of each view angle from the general processing. Therefore, primary objects remain unchanged so if the view angle changes to a position where there is no intersection between convex hulls it is still possible without having to process everything from scratch.

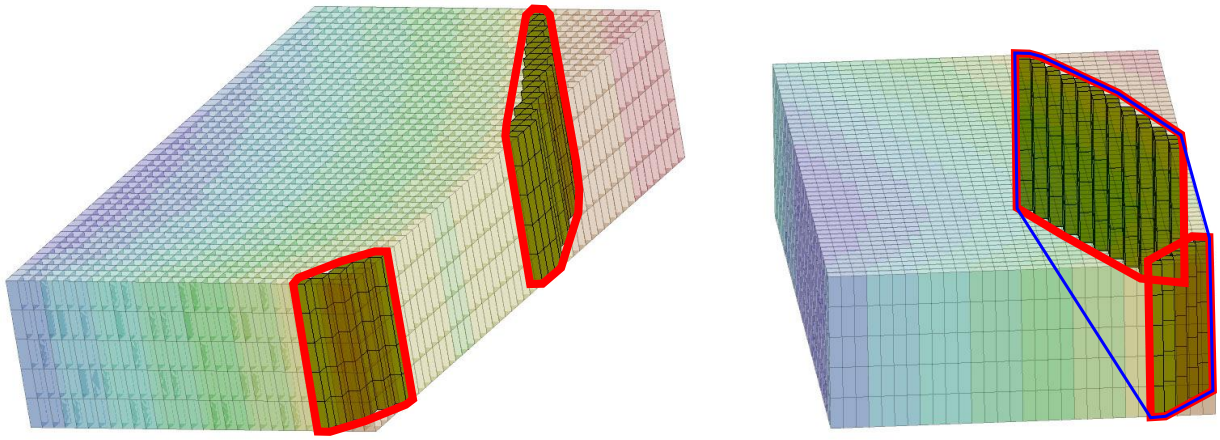

Figure 6.4: Convex hulls (originated from primary objects): (left) two non intersecting convex hulls, red lines; (right) the same primary objects projected based on a different gaze originate two intersecting convex hulls (red lines) which are going to be considered as one (blue line).

The last step before sending data as input to the *Explosion Tree* is to sort the convex hulls. The previous section describes how a set of planes is created based on the relation between convex hulls, therefore, the order of comparison implies in the arrangement of planes that happens inside the data structure, we depict in Figure 6.5 two different orders for the same example that can be found in Figure 5.3. To explore some of the possible variations,

our approach implements two different sorting algorithms. One is the lexicographic sorting in (x,y), i.e., first it takes into consideration the x-axis and then the y-axis (if a draw on the x-axis occurs). Additionally to this method, a binary sorting is also implemented, it is a step performed after the lexicographic sorting which creates a new convex hull sequence by re-sampling the elements based on a binary search algorithm. The reason for having two methods is to explore different results for the same set of primary objects and the same gaze. The following pseudo code shows both algorithms:

```
\\Lexicographic Sorting
Input-------------------------------------------------------------
pObjects \\List of convex polygons
total \\ Number of convex polygons
Begin-------------------------------------------------------------
updated = true
while(updated)
| updated = false
| From i = 1 to total - 1
| | if (pObjects[i].x > pObjects[j].x)
| | |Swap(pObject[i],pObject[j])
| | |updated = true
| | else if (pObjects[i].x = pObjects[j].x) && (pObjects[i].y > pObjects[j].y)
| | |Swap(pObject[i],pObject[j])
| | |updated = true
| | i += 1
|
End---------------------------------------------------------------
```

```
\\Binary Sorting (requires the lexicographic sorting)
Input-------------------------------------------------------------
pObjects \\List of convex polygons
output \\Output list of convex polygons (in the next order)
total \\ Number of convex polygons
Begin-------------------------------------------------------------
 RecursiveBinarySorting(pObjects,0,total-1, output)
End---------------------------------------------------------------

\\Recursive Binary Sorting
Input-------------------------------------------------------------
pObjects \\List of convex polygons
output \\Output list of convex polygons (in the next order)
first \\ index of the first element
last \\ index of the last element
Begin-------------------------------------------------------------
if (first <= last)
|index = (first + last) / 2
|output.Add(pObjects[index])
|RecursiveBinarySorting(pObjects,first,index-1, output)
|RecursiveBinarySorting(pObjects,index+1,last, output)
End---------------------------------------------------------------
```

So far, as secondary objects are not taken into consideration, performance is influenced by the number of primary objects and the total amount of grid cells among them. The sorted convex hulls set is going to be the input data to our data structure and after the tree processing, the final destination of each vertex can be taken from it, where final destination

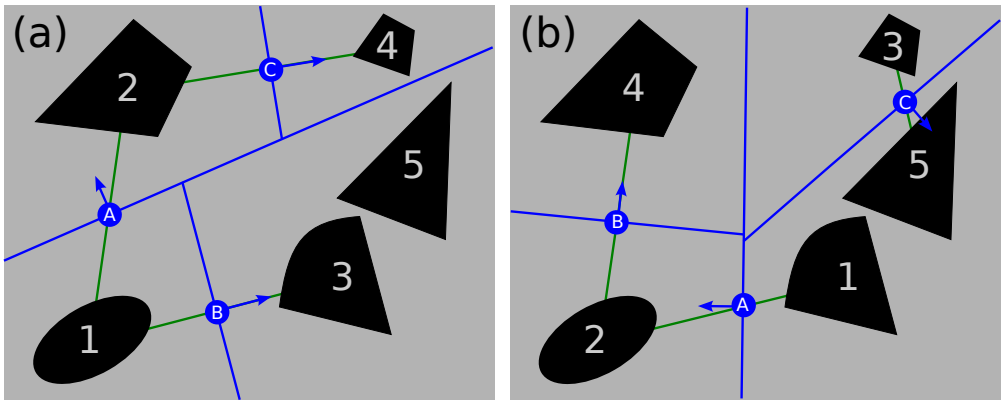

Figure 6.5: Two different trees are created based on the same convex hull set in different orders, using the lexicographic sorting (left) and the binary sorting (right).

means a final position where all primary objects are exposed without causing any overlapping between either primary or secondary objects. To improve the visual correlation between primary and secondary objects a line which connects the original primary object's centroid position in each half of the respective secondary object is drawn for all primary objects (Figure 6.6). Additionally, in order to provide a smooth transition our implementation animate the final effect by a linear interpolation between initial and final positions in ten steps.

6.2 Results and Discussions

We have used our implementation to successfully generate exploded views for both regular and non-regular 3D grids. Besides primary and secondary objects, the final result also depends on the view direction and sorting algorithm. We compute the time to build the tree and perform the effect to different models (see Table 6.1) and primary/secondary object arrangements. Test were performed with an Intel I5 CPU (2.53GHz) PC with 4GB RAM memory.

Changing the gaze might also change the number and order of generated convex hulls, thus it affects the final result. Figure 6.1 shows the results of two different view angles for the same grid, note that, while the leftmost result has a set of planes defined by three convex

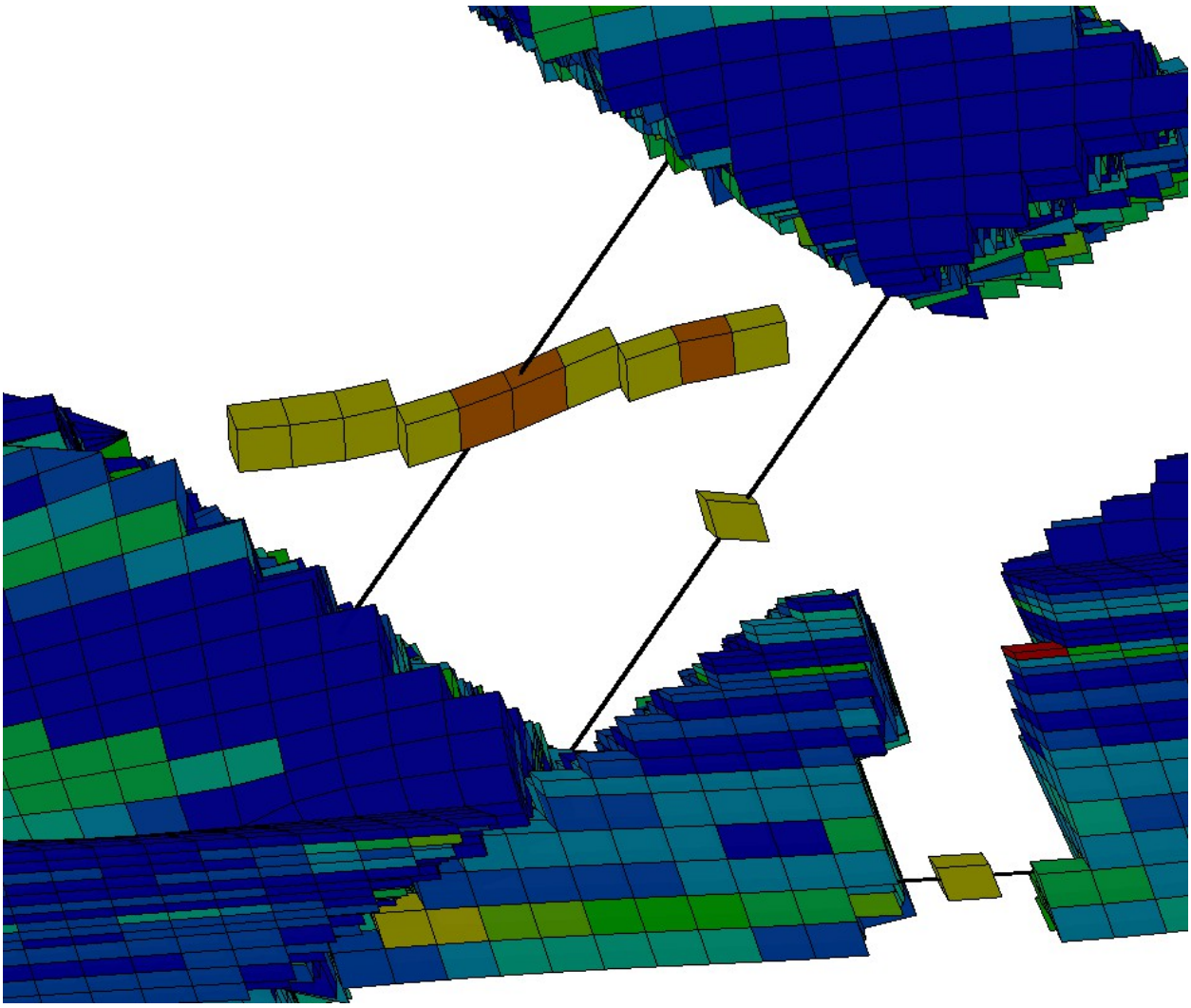

Figure 6.6: Detail of the lines used to improve the visual correlation between primary and secondary objects.

hulls (there is an intersection between two primary objects), the other result (on the right) defines a different set of planes as now there are four convex hulls, instead of three. The exploded views effect is also affected by the sorting algorithm, for instance in Figure 6.7 we apply two sorting algorithms for the same gaze. Despite generating great visual differences, there is not a best algorithm for all cases.

Apart from the time to build the tree we have also measured the total time to achieve the effects. This time can be split into two parts, while the first one happens before building the tree, the second one happens after it. Our tree requires convex objects as input and therefore,

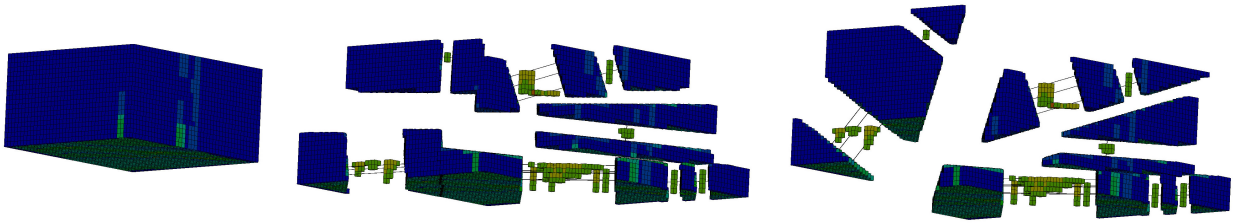

Figure 6.7: Petrel Regular. Different order within the convex objects list (from left to right): original grid; exploded grid using axis aligned sorting; exploded grid using binary sorting.

Name	Type	# Grid Cells	Figures
Petrel Original	corner-point	28507	5.6
Petrel Regular	regular	32760	6.10 6.7
Zmap Original	corner-point	7500	6.8
Emerald Original	corner-point	33655	6.9

Table 6.1: Both real and synthetic models followed by their type, amount of grid cells and respective figures.

the time to convert grid cells into convex objects might be taken into consideration. This is the first part and it is mainly influenced by the total amount of vertices of all 2D convex hulls. In this way, the screen resolution does not affect the performance as we are not dealing with pixels. After the tree is built, our implementation is ready to perform the exploded views effect to all vertices, therefore, the total number of vertices (including secondary objects) is the major factor that affects the performance in this second part. For instance, when applying the effect to a grid with 269240 vertices (Emerald original), the time taken during the conversion (from primary objects to convex hulls) and to update all vertices are 370 and 90 milliseconds respectively. To finalize our workflow we use a ten-step interpolation between initial and final vertex positions, in our worst animation case, Petrel original (228056 vertices), a 37 fps animation was achieved (Figure 6.8).

The exploded diagrams effect was supported in an interactive time for all grids with no distinction between regular (Figures 6.1, 6.7 and 6.10) and non-regular (Figures 5.6, 6.9, 6.8) grids. In our approach, we define separating planes to subdivide the 3D space,

however when there is a large amount of primary objects it creates more empty space than it would be required to just exposed the primary objects (Figure 6.9). A possible solution to overcome this limitation is the use of flexible surfaces to subdivide the space in a more optimized way. A functionality which might be useful to investigate the data is to apply the exploded diagrams from one gaze and varying the view angle to explore the model from different perspective (Figure 6.10).

6.3 Conclusions

We have created an approach that implements the exploded view diagrams similarly to traditional exploded views techniques. In order to apply this effect to 3D grid we introduced a novel data structure, *Explosion Tree*, which offers the correct final vertex position to all vertices and requires limited information regarding the grid. We have applied our approach to both synthetic and real data, in both cases we have achieved the effect in an interactive time. Further research could be done toward extending the implementation to support other types of grids (e.g., non-hexahedron), enhancing the animation between different view angles and calculating the optimum camera position minimizing the amount of intersections among primary objects.

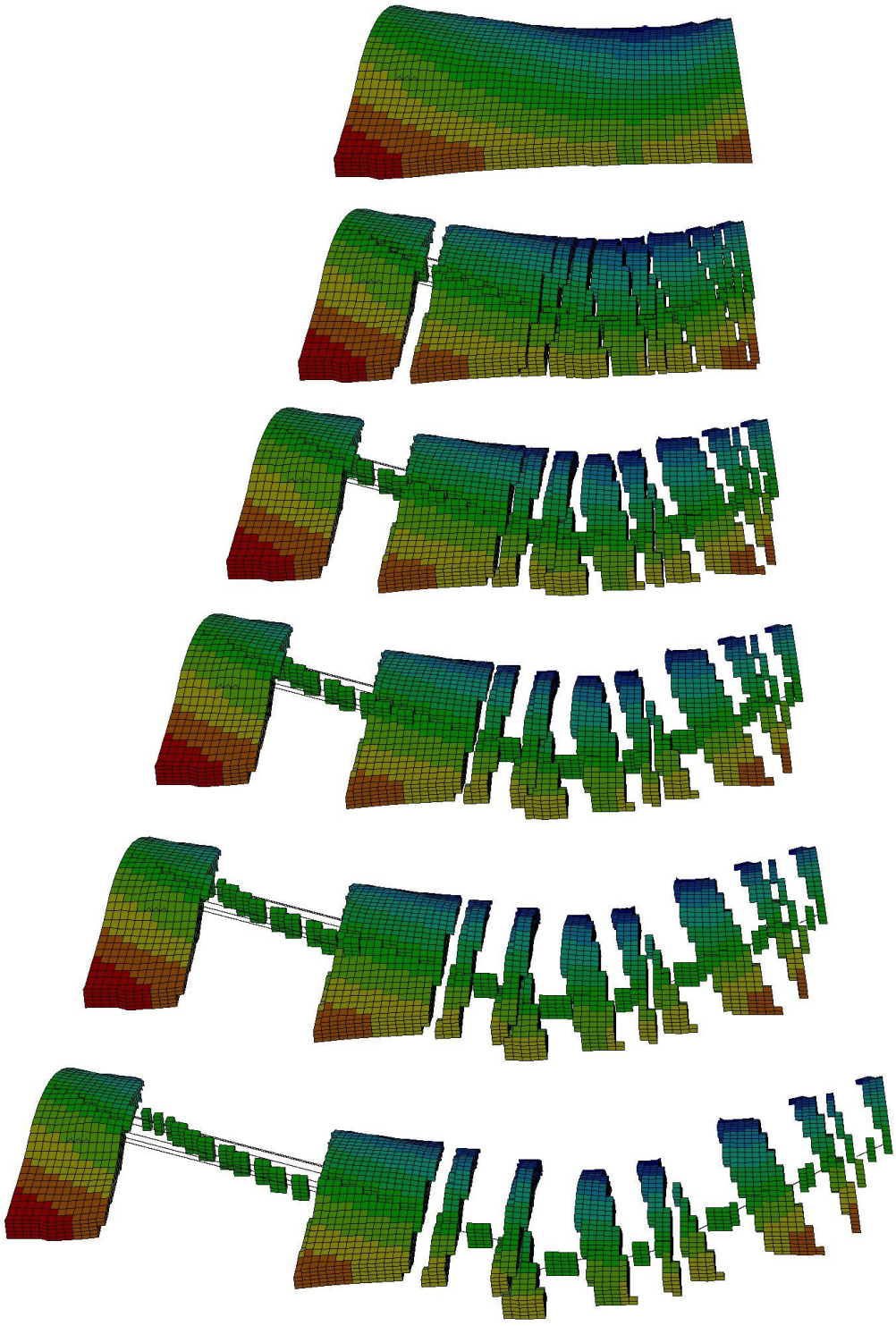

Figure 6.8: Zmap original, six animation steps.

Figure 6.9: Emerald original. Corner-point Grid: (left) original grid; (right) grid after the effect.

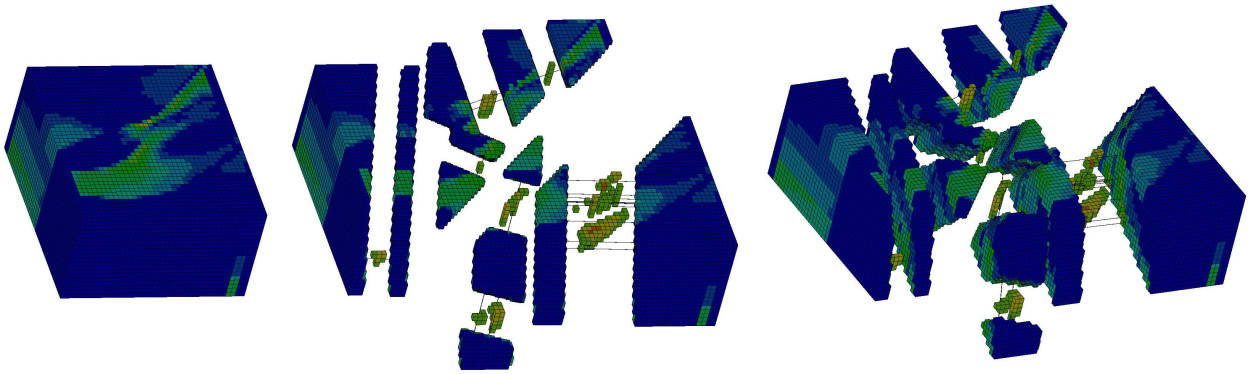

Figure 6.10: Petrel regular. Regular Grid: (left to right) original grid; grid after the effect; the same effect shown from a different perspective.

Chapter 7

Summary and Future Work

In this thesis we presented our research efforts to tackle the problem of occlusion in regular and non-regular 3D grids. Our implementations were based on the Cutaway and Exploded diagrams techniques. Our focus was not on proposing different rendering styles, instead we focused on proposing variations of existing techniques and how they fit into our data model. Regarding the Cutaway view technique, we show that it is an efficient way of emphasize the desired grid cells (focus) without losing the whole context for corner-point grids. The Exploded view technique originated two implementations, one based on mass-spring system and other based on BSP-tree that has been also applied not only for corner points but also for regular grids and we were able to achieve this effect in an interactive time. To make the latter technique works for general grids we have introduced a novel data structure, Explosion Tree, which offers the correct final vertex position to all vertexes and requires limited information regarding the grid.

Revisiting our thesis contributions, we presented a variation of the Cutaway Views technique applied to corner-point grids. We applied the Cutaway technique to allow the inspection of occluded grid cells in corner-point grids, balancing performance and visual effect in a CPU based algorithm. In Chapter 3 we present the design and implementation of our approach, results and our conclusions. In Chapter 6 we present the design and implementation of an approach which supports Exploded Diagram, we also discuss results and our conclusions. We proposed a variation of the Exploded View Diagrams technique applied to 3D grids. The system was implemented assuming that: (1) 3D grids do not have geometrical information regarding parts; (2) the solution takes the problem of scalability into consideration; and (3) without relying on pre-processing stage. In Chapter 5 we discuss the design and

implementation of a data structure based on BSP-tree which support the Exploded Diagram technique acting as the core of one of our implementations. The data structure, Explosion Tree, supports Exploded Diagram with limited amount of information, and it is the core of the system that we developed. It receives limited amount of information as input and offer the final position for every vertices as an output.

7.1 Future Work

7.2 Research Directions

To expand this work and to propose new research direction to this research, some points could be addressed. Although our implementations are based on orthographic projections, different projections could be explored with the aim to provide more insights to the data analyst [5]. For instance, deformation would be achieve not by changing the data but through different projections. By investigating the problem of occlusion in regular and non-regular grids we need to mention that all of them are mainly hexahedron, therefore expanding this research to grids with hexagonal faces, tetrahedron grids and other types would lead our research to a better understanding of the occlusion in more generic 3D objects. In order to enhance interactivity and aesthetics, more research could be done toward better ways of animating the scenes between the effects. This might help the user to keep track of the changes. Regarding the Cutaway technique, the sense of depth is sometimes unclear, therefore, applying shading and other visual aids might also be helpful to a general understanding of the scene. Some case where objects intersect each other could be solved by changing the camera position, and in a more general point of view, the best camera position could be calculated based on the primary object information. Some research toward camera navigation [8] could benefit both technique by balancing the workload between the technique and the camera position. More ways of performing grid cell selection could be more explored in all implementations.

More intuitive and smart ways of performing selection like sketching over the grid cells or retrieving information from a knowledge base using machine learning could make a significant improvement over the way it is being done now. Apart from applying a single technique in its own implementation, a hybrid solution might offer a good advantage when facing too many different types of data. Explore ways of integrating those two techniques in a single implementation and perhaps to support prediction based on the relationship between primary and secondary objects, view angle and data size could be an important research contribution. Lastly, gather feedback from data analysts of various domains would offer a new set of improvements to both approaches.

7.3 Implementation Issues

Although both techniques have provided promising results, there are many aspects to be considered as future work. Cutaway has scalability as a main limitation, for this, we propose the use of shader programming. The depth perception is not accurate in the current implementation and it could also be improve by employing shading techniques. Also, in our implementation we are considering the grid cell as unity, therefore a grid cell can be drawn or not, and as an improvement each grid cell vertex should be an unity, in this way less data would be lost. The Exploded Diagram does not share the same limitations, however it has its own set. Space division could be improved by using different surface representations to subdivide the space in a more optimum way. A better calibration method can be employed in order to reduce the extra space between planes still maintaining that no overlap would occur, and improvement in the data structure could be done in order to support more general geometries, e.g., hexagonal and tetrahedrons.

Bibliography

[1] Maneesh Agrawala, Doantam Phan, Julie Heiser, John Haymaker, Jeff Klingner, Pat Hanrahan, and Barbara Tversky. Designing effective step-by-step assembly instructions. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 828–837, New York, NY, USA, 2003. ACM.

[2] M. A. Ahmad. The bsp tree. Technical report, Minneapolis, MN, 2006.

[3] David Akers, Frank Losasso, Jeff Klingner, Maneesh Agrawala, John Rick, and Pat Hanrahan. Conveying shape and features with image-based relighting. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 46–, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Norbert Aujoulat. *Lascaux: Movement, Space and Time*. Harry N. Abrams, 2005.

[5] John Brosz, Sheelagh Carpendale, Faramarz Samavati, Hao Wang, and Alan Dunning. Art and nonlinear projection. In *Bridges 2009: Mathematical Connections in Art, Music andScience*, 2009.

[6] Stefan Bruckner and M. Eduard Groller. Exploded views for volume data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1077–1084, September 2006.

[7] Michael Burns and Adam Finkelstein. Adaptive cutaways for comprehensible rendering of polygonal scenes. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–7, New York, NY, USA, 2008. ACM.

[8] Marc Christie and Patrick Olivier. Camera control in computer graphics: models, techniques and applications. In *ACM SIGGRAPH ASIA 2009 Courses*, SIGGRAPH ASIA '09, pages 3:1–3:197, New York, NY, USA, 2009. ACM.

[9] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 421–430, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[10] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. *ACM Trans. Graph.*, 22(3):848–855, July 2003.

[11] J. A. Dennisson and C. D. Johnson. *Technical Illustration: Techniques and Applications.* Goodheart-Wilcox, 2003.

[12] J. Diepstraten, D. Weiskopf, and T. Ertl. Interactive cutaway illustrations. In *Computer Graphics Forum*, pages 523–532, 2003.

[13] Y. Ding and P. Lemonnier. Use of corner point geometry in reservoir simulation. *Society of Petroleum Engineers*, 29933-MS, 1995. `http://dx.doi.org/10.2118/29933-MS`.

[14] Debra Dooley and Michael F. Cohen. Automatic illustration of 3d geometric models: lines. *SIGGRAPH Comput. Graph.*, 24(2):77–82, February 1990.

[15] Niklas Elmqvist. Balloonprobe: Reducing occlusion in 3d using interactive space distortion. In *In Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 134–137, 2005.

[16] Raanan Fattal, Maneesh Agrawala, and Szymon Rusinkiewicz. Multiscale shape and detail enhancement from multi-light image collections. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[17] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics*, pages 124–133, 1980.

[18] Ahna Girshick, Victoria Interrante, Steven Haker, and Todd Lemoine. Line direction matters: an argument for the use of principal directions in 3d line drawings. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, pages 43–52, New York, NY, USA, 2000. ACM.

[19] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 447–452, New York, NY, USA, 1998. ACM.

[20] Bruce Gooch and Amy Gooch. *Non-Photorealistic Rendering*. A. K. Peters/CRC Press, 2001.

[21] J. Hamel. A new lighting model for computer generated line drawings. In *PhD Thesis*. University of Magdeburg, 2000.

[22] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 517–526, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[23] E.R.S. Hodges. *The Guild Handbook of Scientific Illustration*. Van Nostrand Reinhold, 1988.

[24] Xavier Provot Institut and Xavier Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *In Graphics Interface*, pages 147–154, 1996.

[25] Victoria Interrante. Illustrating surface shape in volume data via principal direction-driven 3d line integral convolution. Technical report, 1997.

[26] Victoria Interrante, Henry Fuchs, and Stephen Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *Proceedings of the 6th conference on Visualization*

'95, VIS '95, pages 52–, Washington, DC, USA, 1995. IEEE Computer Society.

[27] Tilke Judd, Frédo Durand, and Edward H. Adelson. Apparent ridges for line drawing. *ACM Trans. Graph.*, 26(3):19, 2007.

[28] Olga Karpenko, Wilmot Li, Niloy Mitra, and Maneesh Agrawala. Exploded view diagrams of mathematical surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1311–1318, November 2010.

[29] J. J. Koenderink. What does the occluding contour tell us about solid shape? *Perception*, 1984.

[30] W. Li, L. Ritter, M. Agrawala, B. Curless, and D. Salesin. Interactive cutaway illustrations of complex 3d models. *ACM Trans. Graph.*, 26(3):31, 2007.

[31] Wilmot Li, Maneesh Agrawala, Brian Curless, and David Salesin. Automated generation of interactive 3d exploded view diagrams. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 101:1–101:7, New York, NY, USA, 2008. ACM.

[32] Wilmot Li, Maneesh Agrawala, and David Salesin. Interactive image-based exploded view diagrams. In *Proceedings of Graphics Interface 2004*, GI '04, pages 203–212, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[33] E. Lidal, H. Hauser, and I. Viola. Design principles for cutaway visualization of geological models. In *Proc. Spring Conference on Computer Graphics (SCCG 2012)*, pages 53–60, May 2012.

[34] Martin Luboschik and Heidrun Schumann. Discovering the covered: Ghost views in information visualization. In *Proc. Int. Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2008.

[35] Thomas Luft, Carsten Colditz, and Oliver Deussen. Image enhancement by unsharp masking the depth buffer. *ACM Trans. Graph.*, 25(3):1206–1213, July 2006.

[36] Aditi Majumder and M Gopi. Real-time charcoal rendering using contrast enhancement operators. In *NPAR*, 2002.

[37] Lee Markosian, Michael A. Kowalski, Daniel Goldstein, Samuel J. Trychin, John F. Hughes, and Lubomir D. Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 415–420, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[38] Michael J. McGuffin, Liviu Tancau, and Ravin Balakrishnan. Using deformations for browsing volumetric data. In *Proceedings of IEEE Visualization (VIS) 2003*, pages 401–408, October 2003.

[39] S. Omar. *Ibn al-Haytham's Optics: A Study of the Origins of Experimental Science*. Bibliotheca Islamica, 1977.

[40] D. K. Ponting. Corner point geometry in reservoir simulation. In *The Mathematics of Oil Recovery*. EAGE, July 1989.

[41] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 581–, New York, NY, USA, 2001. ACM.

[42] Szymon Rusinkiewicz, Michael Burns, and Doug DeCarlo. Exaggerated shading for depicting shape and detail. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 1199–1205, New York, NY, USA, 2006. ACM.

[43] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *SIGGRAPH Comput. Graph.*, pages 197–206, 1990.

[44] Mario Costa Sousa and John W. Buchanan. Computer-generated graphite pencil rendering of 3d polygonal models. *Computer Graphics Forum*, 18:195–208, 1999.

[45] Thomas Strothotte and Stefan Schlechtweg. *Non-photorealistic Computer Graphics: Modeling, Rendering and Animation*. Morgan Kaufmann, 2002.

[46] I. N. Suta and S. O. Osisanya. Geological modelling provides best horizontal well positioning in geologically complex reservoirs. *Society of Petroleum Engineers*, 2004-098, 2004. http://dx.doi.org/10.2118/2004-098.

[47] Markus Tatzgern, Denis Kalkofen, and Dieter Schmalstieg. Compact explosion diagrams. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '10, pages 17–26, New York, NY, USA, 2010. ACM.

[48] Joe F. Thompson, Bharat K. Soni, and Nigel P. Weatherill. *Handbook of Grid Generation*. CRC Press, 1999.

[49] W. A. Wadsley. Modelling reservoir geometry with non-rectangular coordinate grids. *Society of Petroleum Engineers*, 9369-MS, 1980. http://dx.doi.org/10.2118/9369-MS.

[50] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 91–100, New York, NY, USA, 1994. ACM.

[51] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 469–476, New York, NY, USA, 1996. ACM.

Appendix A

Appendix A. Explosion Tree Source Code

A.1 Explosion Tree Declaration

```
#ifndef PLANETREE
#define PLANETREE

#include "PlaneTreeNode.h"
#include <D3DX11.h>

namespace DataCore
{
  class PlaneTree
  {
    protected:
      //Fields
      PlaneTreeNode* root;

      //Protected Methods
      void DeleteRecusively(PlaneTreeNode* node);
      PlaneTreeNode* NonVisitedRecursivelyPreOrder(PlaneTreeNode* node);
      PlaneTreeNode* NonVisitedRecursivelyPostOrder(PlaneTreeNode* node);
      void ResetRecusively(PlaneTreeNode* node);
      void UpdateAllOffsetRecursivelyPostOrder(PlaneTreeNode* node);
      void CalibrateAllOffsetRecursivelyPostOrder(PlaneTreeNode* node);

    public:
      ///Public Methods
      PlaneTree();
```

```
      ~PlaneTree();
      void DeleteAll();
      void AddRoot(PlaneTreeNode* node);
      PlaneTreeNode* NonVisitedNodePreOrder();
      PlaneTreeNode* NonVisitedNodePostOrder();
      void ResetVisitedStatus();
      DataCore::Vector3D GetOffSetVector(DataCore::Vector3D& point, bool
         considerAll = true);
      void UpdateAllOffset();
      void CalibrateAllOffset();
          PlaneTreeNode* GetRoot();
      void GetPath(DataCore::Vector3D& point, std::vector<PlaneTreeNode*>&
          parentNodes);
  };
}
#endif
```

A.2 Explosion Tree Definition

```
#include "PlaneTree.h"

DataCore::PlaneTree::PlaneTree()
{
  this->root = NULL;
}

DataCore::PlaneTree::~PlaneTree()
{
  if (this->root)
  {
    this->DeleteRecusively(this->root);
    this->root = NULL;
```

```
  }
}

void DataCore::PlaneTree::DeleteAll()
{
  this->DeleteRecusively(this->root);
  this->root = NULL;
}

void DataCore::PlaneTree::DeleteRecusively(PlaneTreeNode* node)
{
  if (node)
  {
    if (node->LeftChild)
      DeleteRecusively(node->LeftChild);
    if (node->RightChild)
      DeleteRecusively(node->RightChild);
    delete node;
  }
}

void DataCore::PlaneTree::AddRoot(PlaneTreeNode* node)
{
  this->root = node;
}

DataCore::PlaneTreeNode* DataCore::PlaneTree::NonVisitedNodePreOrder()
{
  if (this->root)
    return this->NonVisitedRecursivelyPreOrder(this->root);
  return NULL;
}
```

```
DataCore::PlaneTreeNode* DataCore::PlaneTree::NonVisitedNodePostOrder()
{
  if (this->root)
    return this->NonVisitedRecursivelyPostOrder(this->root);
  return NULL;
}

DataCore::PlaneTreeNode* DataCore::PlaneTree::
   NonVisitedRecursivelyPostOrder(PlaneTreeNode* node)
{
  if (node)
  {
    DataCore::PlaneTreeNode* t = NULL;

    if (node->LeftChild)
    {
      t = this->NonVisitedRecursivelyPostOrder(node->LeftChild);
      if (t)
        return t;
    }
    if (node->RightChild)
    {
      t = this->NonVisitedRecursivelyPostOrder(node->RightChild);
      if (t)
        return t;
    }

    if (!node->IsVisited())
      return node;

    return t;
```

```
  }

  return NULL;
}

DataCore::PlaneTreeNode* DataCore::PlaneTree::
   NonVisitedRecursivelyPreOrder(PlaneTreeNode* node)
{
  if (node)
  {
    DataCore::PlaneTreeNode* t = NULL;
    if (!node->IsVisited())
      return node;
    if (node->LeftChild)
    {
      t = this->NonVisitedRecursivelyPreOrder(node->LeftChild);
      if (t)
        return t;
    }
    if (node->RightChild)
    {
      t = this->NonVisitedRecursivelyPreOrder(node->RightChild);
      if (t)
        return t;
    }
    return t;
  }
  return NULL;
}

void DataCore::PlaneTree::ResetVisitedStatus()
{
```

```
  this->ResetRecusively(this->root);
}

void DataCore::PlaneTree::ResetRecusively(PlaneTreeNode* node)
{
  if (node)
  {
    node->SetVisited(false);
    if (node->LeftChild)
      ResetRecusively(node->LeftChild);
    if (node->RightChild)
      ResetRecusively(node->RightChild);
  }
}

DataCore::Vector3D DataCore::PlaneTree::GetOffSetVector(DataCore::Vector3D
   & point, bool considerAll)
{
  DataCore::Vector3D offsetVector(0.0f,0.0f,0.0f);
  PlaneTreeNode* node = this->root;
  do
  {
    if (node)
    {
      if ((!considerAll) && (node->extendedNode))
        break;
      DataCore::Vector3D currentPoint = node->cutplane3D.GetPoint();
      DataCore::Vector3D currentNormal = node->cutplane3D.GetNormal();
      DataCore::Vector3D tmp(point.GetX() - currentPoint.GetX(),point.GetY
          () - currentPoint.GetY(),point.GetZ() - currentPoint.GetZ());
      tmp.Normalize();
      float value = currentNormal.Dot(tmp);
```

```
      tmp = currentNormal;
      tmp *= (value < 0)?node->cutplane3D.GetNegativeOffset():node->
          cutplane3D.GetPositiveOffset();
      if (value <= 0.0f)
      {
        offsetVector -= tmp;
        if (node->extendedNode)
        {
          offsetVector = tmp;
          offsetVector *= -1.0f;
        }
        node = node->LeftChild;
      }
      else
      {
        offsetVector += tmp;
        if (node->extendedNode)
          offsetVector = tmp;
        node = node->RightChild;
      }
    }
  }while(node);

  //offsetVector.Normalize();
  return offsetVector;
}

void DataCore::PlaneTree::UpdateAllOffset()
{
  this->UpdateAllOffsetRecursivelyPostOrder(this->root);
}
```

```
DataCore::PlaneTreeNode* DataCore::PlaneTree::GetRoot()
{
  return this->root;
}

void DataCore::PlaneTree::UpdateAllOffsetRecursivelyPostOrder(
   PlaneTreeNode* node)
{
  if (node)
  {
    if (node->LeftChild)
    {
      this->UpdateAllOffsetRecursivelyPostOrder(node->LeftChild);
    }
    if (node->RightChild)
    {
      this->UpdateAllOffsetRecursivelyPostOrder(node->RightChild);
    }
    DataCore::Vector3D currPoint= node->cutplane3D.GetPoint();
    DataCore::Vector3D currNormal= node->cutplane3D.GetNormal();
    DataCore::Vector3D inverseCurrNormal(currNormal);
    inverseCurrNormal *= -1.0f;
    std::vector<PlaneTreeNode*> parentNodes;
    this->GetPath(currPoint, parentNodes);
    for(unsigned int i=0; i<parentNodes.size(); i++)
    {
      DataCore::Vector3D tmpVector(currPoint);
      tmpVector -= parentNodes[i]->cutplane3D.GetPoint();
      tmpVector.Normalize();
      float dotProduct = tmpVector.Dot(parentNodes[i]->cutplane3D.
          GetNormal());
```

```
      DataCore::Vector3D parentNormal(parentNodes[i]->cutplane3D.GetNormal
          ());
      if (dotProduct > 0.0f)
      {
        //Right
        float rResult = parentNormal.Dot(currNormal);
        float lResult = parentNormal.Dot(inverseCurrNormal);
        float lvalue = node->cutplane3D.GetNegativeOffset() * lResult;
        float rvalue = node->cutplane3D.GetPositiveOffset() * rResult;
        float result = (lvalue < rvalue)?-lvalue : -rvalue;
        float currentp = parentNodes[i]->cutplane3D.GetPositiveOffset();
        if (currentp < result)
          parentNodes[i]->cutplane3D.SetPositiveOffset(result);
      }
      else
      {
        //Left
        parentNormal *= -1.0f;
        float rResult = parentNormal.Dot(currNormal);
        float lResult = parentNormal.Dot(inverseCurrNormal);
        float lvalue = node->cutplane3D.GetNegativeOffset() * lResult;
        float rvalue = node->cutplane3D.GetPositiveOffset() * rResult;
        float result = (lvalue < rvalue)?-lvalue : -rvalue;
        float currentn = parentNodes[i]->cutplane3D.GetNegativeOffset();
        if (currentn < result)
          parentNodes[i]->cutplane3D.SetNegativeOffset(result);
      }
    }
  }
}

void DataCore::PlaneTree::CalibrateAllOffset()
```

```
{
  this->CalibrateAllOffsetRecursivelyPostOrder(this->root);
}

void DataCore::PlaneTree::CalibrateAllOffsetRecursivelyPostOrder(
   PlaneTreeNode* node)
{
  if (node)
  {
    if (node->LeftChild)
      this->CalibrateAllOffsetRecursivelyPostOrder(node->LeftChild);
    if (node->RightChild)
      this->CalibrateAllOffsetRecursivelyPostOrder(node->RightChild);
    std::vector<PlaneTreeNode*> parentNodes;
    this->GetPath(node->cutplane3D.GetPoint(), parentNodes);
    parentNodes.push_back(node);
    for(unsigned int i=0; i< parentNodes.size(); i++)
    {
      //accumulated values to be compared
      float rightAlpha = 0.0f;
      float leftAlpha = 0.0f;
      //current plane
      Vector3D n0(parentNodes[i]->cutplane3D.GetNormal());
      Vector3D invn0(n0);
      invn0 *= -1.0f;
      Vector3D p0(parentNodes[i]->cutplane3D.GetPoint());
      for(unsigned int j=i+1; j< parentNodes.size(); j++)
      {
        //plane to be tested
        Vector3D n1(parentNodes[j]->cutplane3D.GetNormal());
        Vector3D invn1(n1);
        invn1 *= -1.0f;
```

```
  Vector3D p1(parentNodes[j]->cutplane3D.GetPoint());
  //which side of the current plane this testing plane is?
  Vector3D tmp(p1);
  tmp -= p0;
  tmp.Normalize();
  float dotProduct = tmp.Dot(n0);
  if (dotProduct > 0.0f)
  {
    //Right
    float n0n1 = n0.Dot(n1);
    float n0invn1 = n0.Dot(invn1);
    float rvalue = parentNodes[j]->cutplane3D.GetPositiveOffset() *
       n0n1;
    float lvalue = parentNodes[j]->cutplane3D.GetNegativeOffset() *
       n0invn1;
    float result = (lvalue < rvalue)?-lvalue : -rvalue;
    rightAlpha += result;
  }
  else
  {
    //Left
    float invn0n1 = invn0.Dot(n1);
    float invn0invn1 = invn0.Dot(invn1);
    float rvalue = parentNodes[j]->cutplane3D.GetPositiveOffset() *
       invn0n1;
    float lvalue = parentNodes[j]->cutplane3D.GetNegativeOffset() *
       invn0invn1;
    float result = (lvalue < rvalue)?-lvalue : -rvalue;
    leftAlpha += result;
  }
}
//Check is the value is greater than the current one
```

```
      float currentp = parentNodes[i]->cutplane3D.GetPositiveOffset();
      if (currentp < rightAlpha)
        parentNodes[i]->cutplane3D.SetPositiveOffset(rightAlpha);
      float currentn = parentNodes[i]->cutplane3D.GetNegativeOffset();
      if (currentn < leftAlpha)
          parentNodes[i]->cutplane3D.SetNegativeOffset(leftAlpha);
    }

  }
}

void DataCore::PlaneTree::GetPath(DataCore::Vector3D& point, std::vector<
   PlaneTreeNode*>& parentNodes)
{
  parentNodes.clear();
  PlaneTreeNode* node = this->root;
  DataCore::Vector3D planePoint, planeNormal;
  while( node && (node->cutplane3D.GetPoint().GetX() != point.GetX()
      || node->cutplane3D.GetPoint().GetY() != point.GetY()
      || node->cutplane3D.GetPoint().GetZ() != point.GetZ()) )
  {
    parentNodes.push_back(node);
    planePoint = node->cutplane3D.GetPoint();
    planeNormal = node->cutplane3D.GetNormal();
    DataCore::Vector3D tmpVector(point);
    tmpVector -= planePoint;
    tmpVector.Normalize();
    float dotProduct = tmpVector.Dot(planeNormal);
    node = (dotProduct > 0.0f )? node->RightChild : node->LeftChild;
  }
}
```