View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório da Universidade Nova de Lisboa

**Ricardo Boto Semblano da Silva**

BSC in Computer Science

# Virtual Reality Integration on Tomo-GPU System

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
**Computer Science and Informatics Engineering**

Adviser: Doutor Pedro Abílio Duarte de Medeiros,
Associate professor, NOVA University of Lisbon

Examination Committee

Chairperson: Doutor Pedro Manuel Corrêa Calvente Barahona
Raporteurs: Doutor Adriano Martins Lopes
Doutor Pedro Abílio Duarte de Medeiros

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**September, 2019**

**Virtual Reality Integration on Tomo-GPU System**

Copyright © Ricardo Boto Semblano da Silva, Faculty of Sciences and Technology, NOVA University Lisbon.
The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

*For self learning academically and personally*
*Dedicated to nobody.*

# Acknowledgements

First and foremost, I must show my gratitude to my adviser, Pedro Medeiros, where without him this work would be impossible, and he was able to salvage me through a difficult situation.

To my institution, Faculdade de Ciências e Tecnologia, that provided all the conditions necessary to perform this work.

To professor Maria Cecília Gomes, by having no responsibility whatever but always available to help and provided one of the best discussions on a coffee break.

To every single colleague where some were already friends, and the others became friends, beeing too many to mention. However, Miguel Cordeiro, João Simões, João Oliveira and Alexandre Ferreira are some of the people that made every workday much more enjoyable.

And finally to my parents that even in spite of many things, they provided everything required to finish this work.

# Abstract

With the ever-greater creation of data, new ways to extract information from it in faster ways is a subject of great interest to the scientific community in general and any entity that may benefit with the interpretation of data. Virtual reality, although not a recent discovery only now is becoming broadly available and driving new state of the art designs and implementations. Nonetheless, already existing results, provide positive feedback of virtual reality on some cases of data visualization.

One of the scientific areas that may benefit from virtual reality technology visualization is the scientific field of material sciences. A current project of FCT is the Tomo-GPU system that was developed to aid the material scientists in processing and visualizing their data. This work focuses on the integration of a virtual reality visualization on the Tomo-GPU system to aid material scientist in interpreting their data more efficiently.

**Keywords:** Virtual reality, data visualization, 3D visualization, Tomo-GPU.

# Resumo

Com a geração de dados aumentando constantemente, novas maneiras de extrair informação de forma rápida e eficaz é do interesse tanto para a comunidade científica como para qualquer entidade que beneficie de interpretação de dados. As interfaces de realidade virtual, apesar de não ser uma descoberta recente, só agora é que estas estão a ficar facilmente disponíveis ao público e assim fomentando novos estudos sobre a mesma. No entanto, estudos já efetuados, comprovam a utilidade da nova interface em alguns casos de visualização de dados.

Uma das áreas científicas que poderão beneficiar com realidade virtual é a área de ciências dos materiais. Atualmente, existe um sistema, chamado Tomo-GPU, com o objetivo de processar e visualizar os dados destes especialistas. Este trabalho foca-se no desenvolvimento e integração de visualização de dados com realidade virtual no sistema Tomo-GPU para que seja possível interpretar os seus dados de maneira mais eficiente.

**Palavras-chave:** Realidade Virtual, visualização interativa de dados, visualização 3D, Tomo-GPU

# Contents

**List of Figures** **xv**

**1 Introduction** **1**

1.1 Context and Motivation . . . . . . . . . . 1
1.2 Objective . . . . . . . . . . 2
1.3 Problem . . . . . . . . . . 2
1.4 Approach and document structure . . . . . . . . . . 3

**2 Related Work** **5**

2.1 Information Visualization Systems . . . . . . . . . . 5
2.1.1 Interactive Data Visualization Framework . . . . . . . . . . 6
2.1.2 Interactive Data Visualization System Validation Framework . . . 7
2.1.3 Visual Encoding Taxonomy . . . . . . . . . . 9
2.2 Virtual Reality . . . . . . . . . . 11
2.2.1 Immersive VR Foundations . . . . . . . . . . 11
2.2.2 3D Interfaces Considerations . . . . . . . . . . 14
2.3 *Vis* Systems with VR . . . . . . . . . . 15
2.4 Problem Solving Environments(PSE) . . . . . . . . . . 15
2.5 *SCIRun* . . . . . . . . . . 16
2.5.1 Modular Architecture . . . . . . . . . . 17
2.5.2 Data Types, Ports and Connections . . . . . . . . . . 17
2.5.3 Dataflow network . . . . . . . . . . 17
2.5.4 Module Development . . . . . . . . . . 17
2.5.5 Limitations . . . . . . . . . . 18
2.6 TOMO-GPU System . . . . . . . . . . 18
2.7 Unreal Engine 4 Framework . . . . . . . . . . 19
2.7.1 Architecture Basics . . . . . . . . . . 20
2.7.2 Development . . . . . . . . . . 22
2.7.3 Blueprints . . . . . . . . . . 22
2.7.4 Attachment System and World Composition . . . . . . . . . . 23
2.7.5 Execution Flow . . . . . . . . . . 24
2.7.6 VR Support . . . . . . . . . . 24

2.8 Prototype Framework Characterization . . . . . 25
2.8.1 Geometric Data Abstraction(What) . . . . . 25
2.8.2 Material Engineer Tasks(Why) . . . . . 27
2.8.3 Visualization Algorithms of Geometric Data(How) . . . . . 27
2.9 Summary . . . . . 30

**3 Organization Solution** **33**
3.1 Tomo-GPU VR Module . . . . . 34
3.2 SCIRunVR Prototype . . . . . 34

**4 Implementation** **37**
4.1 Tomo-GPU VR module . . . . . 37
4.1.1 Module Implementation . . . . . 37
4.2 SCIRunVR UE4 . . . . . 38
4.2.1 System Architecture . . . . . 39
4.2.2 DataStructs . . . . . 42
4.2.3 Readers . . . . . 42
4.2.4 StaticLibs . . . . . 43
4.2.5 Tasks . . . . . 44
4.2.6 Model . . . . . 47
4.2.7 Blueprint Control Management . . . . . 50
4.2.8 World Hierarchy Tree . . . . . 51

**5 Evaluation** **55**
5.1 Test Environment . . . . . 55
5.2 Tasks Execution . . . . . 56
5.2.1 Load . . . . . 56
5.2.2 Calculate Edges . . . . . 57
5.2.3 Surface Extraction . . . . . 57
5.3 Draw Execution . . . . . 58
5.4 Liveness Test . . . . . 59

**6 Conclusion** **61**
6.1 Future Work . . . . . 61

**Bibliography** **63**

# List of Figures


2.1 Elements of interaction in visualization. Adapted from [15] . . . . . . . . . . . 6
2.2 Elements of interaction in visualization. Adapted from [11] . . . . . . . . . . . 7
2.3 Construction and validation framework. Adapted from [11] . . . . . . . . . . 8
2.4 Possible marks. Retrived from [11] . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.5 Visual channels for data encoding. Adapted from [11] . . . . . . . . . . . . . . 9
2.6 Ames room optical illusion. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.7 Examples of 3D disadvantages. . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.8 The interface of the SCIRun system framework. On the left module list, on the right data flow network on center top a visualization widget and on center down is one module interface. . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.9 Standard execution of Tomo-GPU. Retrieved from [25] . . . . . . . . . . . . . 20
2.10 Unreal Engine 4 editor interface. . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.11 UE4 blueprint system. Retrieved from [1] . . . . . . . . . . . . . . . . . . . . . 23
2.12 World with four levels. Adapted from [13] . . . . . . . . . . . . . . . . . . . . 23
2.13 Geometry topology abstraction. Retrieved from [18] . . . . . . . . . . . . . . . 26
2.14 Geometry dataset structures. Retrieved from [18] . . . . . . . . . . . . . . . . . 26
2.15 Simple cubes algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.16 Contouring algorithm cases. Retrieved from [18] . . . . . . . . . . . . . . . . . 29
2.17 Marching cubes algorithm base cases. Retrieved from [18] . . . . . . . . . . . 30
2.18 Results of Marching cubes algorithm to the left and dual marching cubes to the right. Retrieved from [17] . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.1 Component diagram for with overall functionality to integrate VR on Tomo-GPU system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.1 VR module with its interface inside *SCIRun* framework. . . . . . . . . . . . . . 38
4.2 SCIRunVR c++ class diagram. . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3 SCIRunVR Blueprints class diagram for control functionalities. . . . . . . . . . 41
4.4 Voxel direction standard use in simple cube algorithm implementation. . . . . 46
4.5 Flux diagram of *AModelManager*. . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.6 Resulting UI to currently manipulate SCIRunVR prototype. . . . . . . . . . . 50
4.7 Room plant. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.8 Level n-ary tree structure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.9 Prototype running NRRD file with dual marching cubes surface algorithm. . 53
4.10 Prototype running NRRD file with simple cubes surface algorithm. . . . . . . 53
4.11 Prototype running Bruno file type with dual marching cubes surface algorithm. 53

5.1 Execution times of loading tasks with different data size of both supported files. 57
5.2 Execution time of edge calculation with different data size of both supported files. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.3 Surface extraction time execution. . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.4 Execution times of mesh drawing by the number o displayed vertexes for both supported mesh techniques. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.5 FPS average displaying with both types of mesh depending on vertex count. 59

CHAPTER 1

# Introduction

## 1.1 Context and Motivation

The Tomo-GPU project was funded by FCT/MCTES (PTDC/EIA-EIA/102579/2008 - Ambiente de Resolução de Problemas para Caracterização Estrutural de Materiais por Tomografia) to assist Material Science Engineers specialists in the development of new composite materials, and it's fabrication methods[14].

A composite material is formed from at least two different materials. One used as a base/continuous material, while the others are, often, discrete/clustered and used as reinforcements. The properties manifested by the composite depends not only on the base materials applied but also, on the relative amounts, formations, distribution, orientation and many other chemical and physical conditions[2]. The new characteristics obtained can prove useful in some functionality and then applied in the most varied of industry fields from the creation of surgical tools to the construction of buildings.

Thus to aid material engineers, a system with the following characteristics was built[2]:

- Sophisticated 3D data visualization.
- Allow computational steering by material experts that usually is not a computer expert.
- Easy integration of new features, where those can be already existing programs or special purpose functionalities.

These requirements are currently achieved by using a problem solving environment tool(PSE) called *SCIRun*. *SCIRun* is a graphical tool that consists of a menu that lists all the available modules that can have input and/or output ports. Besides, it allows the user to assemble those modules by connecting its ports. This ease of use allows for the

non-computer expert to apply powerful functionalities on its data either for processing or for visualization. The Tomo-GPU system consists of several modules based on *SCIRun* framework, that satisfies material engineers tasks.

Although VR isn't a brand new technology, only recently became broadly available and consequently stimulating research and development with such technology. However, some research has already been done, and current literature[4] exposes a vast potential in VR with data that possess an intrinsic three-dimensional aspect. A part of this potential is because of natural interaction that VR provides, however such interaction comes great computations costs.

## 1.2 Objective

This work has the objective of developing a virtual reality visualization and allow integration on the Tomo-GPU system currently used by material engineers to handle their data. Such integration will harness the modular capabilities of *SCIRun*, however as discussed in the previous point, VR as a new interface technology it as some peculiar and demanding characteristics. Such characteristics need not only a real-time render engine tool but also a VR controller support, to manage the development in a feasible amount of time.

Two independent goals may be extracted from the previous broader objective. Those are as followed:

- The construction of a prototype system that allows the visualization of the three-dimensional data processed and stored by the Tomo-GPU system in a VR environment.
- Full integration with Tomo-GPU system, allowing its use by the non-computer expert.

## 1.3 Problem

The main concern of this work is how to integrate the new interface technology, that is accessible for development, to visualize the data handled by the Tomo-GPU system and make it available on its workflow. To address such a wide problem, one must break it apart in several underlying subjects. Those are:

**Characterization Visualization Systems:** The integration pretended in this work is primarily a visualization system. For that reason, a concise framework allowing to correctly address such type of systems and also how to properly validate them.

***SCIRun* architecture:** The understanding of Tomo-GPU system, built over *SCIRun* framework, is fundamental to add a new module to the project. However, a current limitation of the system used is that it does not provide easy integration with any

virtual reality equipment. This limitation forced the development of a prototype system apart from the *SCIRun* framework. Nonetheless bridging between systems is fundamental.

**Characterization of data:** How data is stored, and its meaning is fundamental to allow understanding of how to handle it for visualization.

**Characterization of surfaces:** Different ways to define surfaces exist and literature is extensive on this subject. For this reason only a small portion, that is directly useful for the development will be considered in this work.

**VR usefulness in data visualization:** Data visualization is another subject with extensive literature and considered a scientific area on itself. However, basic knowledge of it and also about the foundations behind VR technology is required. Such knowledge will help grasp VR potential in the data visualization.

## 1.4 Approach and document structure

In this document will start by addressing, in related work chapter, the relevant subjects derived from the central issue of this work established in the previous point. These subjects start with a more theoretical approach to establish the required knowledge and vocabulary to discuss a possible solution. And they end with the explanation of the used technologies and a description of the intended system to implement.

The work will progress to an explanation of the solution organization achieved with all gathered knowledge. After, it will be explained the implemented solution and is followed by the results.

# Chapter 2

# Related Work

This chapter has the purpose of gaining background knowledge necessary for the prototype system implementation 1.2. It will start with a more theoretical approach that will decompose and characterize the two main concepts. The first is information visualization systems (*InfoVis* systems) and the second is virtual reality. Such characterizations will provide information over the consequences of integrating VR in *InfoVis* systems and allow to formulate a conclusion, along with some other research conclusions on the subject, in the following section.

After the theoretical foundations, it will be addressed the technologies used by this work, starting with *SCIRun* framework technology and evolve to the Tomo-GPU system to conceptualize the development for VR integration. Next, it will be considered the unreal engine 4 framework that will be required to develop the prototype in a conceivable period of time.

The last section will gather information about this chapter and apply a framework to characterize the prototype succinctly.

## 2.1 Information Visualization Systems

Information Visualization systems or *InfoVis* systems resort to visualization, by transforming the symbolic into the geometric, and allowing for the user to observe their data. Data on itself holds little value. It is just when analyzed and used in decision making that its value manifests.

The use of the visual channel is due to be well characterized and most suited among the other sensory channels. As visual beings, the majority of information used is obtained through sight, and *InfoVis* systems explore this information retrieval "machinery"to establish communication between system and user [11]. Visualization systems possess one

other fundamental component, although, with less focus on research [27], it is almost mandatory in current *InfoVis* systems. This component is interaction, and it allows the user to handle complexity better. By establishing communication between user and system, the user can change the visualization on demand allowing for different perspectives from data quickly and efficiently. The dialogue established is always mediated by a computer system with its user interface. This concept is depicted in image 2.1, and this concept is at the core of current days *InfoVis* system. However, possible combinations of evolving factors in this type of systems allows for an infinity of possibilities. The computer field of interactive data visualization studies these possibilities and literature is vast on the subject.

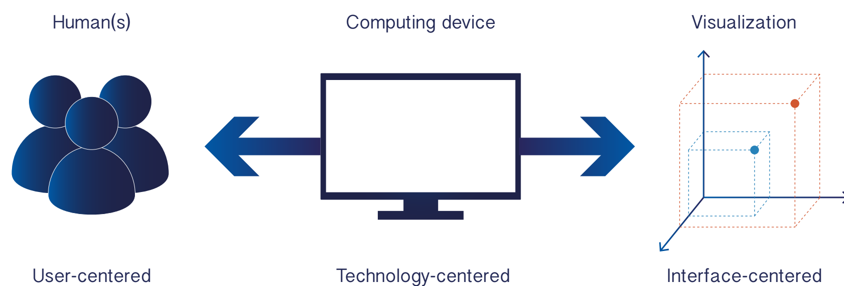



Figure 2.1: Elements of interaction in visualization. Adapted from [15]

### 2.1.1 Interactive Data Visualization Framework

In this subsection, one possible framework, retrieved from [11], is analyzed. This framework acknowledges three primary components; those are:

**Data(What):** The data component focuses on the recognition of patterns in data. Each piece of data as two main characteristics. First its semantic value (ex. is a name or city?) that must always be taken into consideration by the programmer. The second is the type of data that focus on the mathematical interpretation(ex. is an attribute or link?). Differently structured aggregations of different types of data generate a data set where several well-known patterns exist like tables, graphs, fields and many more. The recognition of these patterns in data is of paramount importance in the design and implementation of a data visualization system.

**Task(Why):** The task component focuses on the recognition of different objectives from the user. As a component that is extremely influenced by the domain area that visualization is applied, it is essential to distantiate from domain-specific objective into the most generic term. This generic term is more concise of what the user intends from the visualization. This framework recognizes two different aspects of a task the action and the target. Actions can be to discover, present, summarize and

many more. Where targets can be trends, outliers, attribute, topology, shape and others.

**Visual/Interaction idiom(How):** This component focuses on structuring the vast possibilities in visualization and interaction. It devises into four different groups. The first, visual encoding that consists of characterizing the elements of one image and the mapping of the data to such elements. As a fundamental concept in this thesis, it will be further addressed in 2.1.3. The second is manipulation that classifies how the visual elements can be interacted either by change, selection or navigation. The third is facet, that consists in the assembly of more than one image, useful for comparisons. The last is reduce that describes how data can be filtered, aggregated or embedded for drill-down operations.

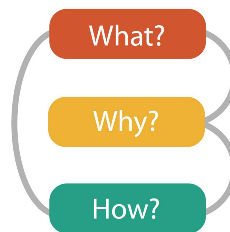



Figure 2.2: Elements of interaction in visualization. Adapted from [11]

As depicted in 2.2, these components are interdependent and are full of trade-offs. Knowledge gathered over some component should be taken into consideration and start a refinement loop and evaluate the consequences over the other components.

### 2.1.2 Interactive Data Visualization System Validation Framework

The framework, addressed in the previous point, allow us how to describe and categorize some data visualization technique. However, it does not specify a visualization system on its whole. The new framework consists of four nested levels as is meant to describe the entire *InfoVis* system and provide a consistent framework for validation of such systems [10]. Nonetheless, the previously discussed components make part of this framework and constitute two of the four layers. The overall structure is depicted on 2.3 and possesses a nested hierarchy where information in one layer should be passed into the other layers yet not strictly in a linear way. The layers are as followed:

**Domain problem characterization:** Evaluates the problems and data of the specific domain, where the challenge lies in connecting the concepts between users and designers.

**Data and task abstraction:** This layer unifies the data and task abstraction, specified in the previous item, into one single validation step because both consist in the abstraction of the domain-specific vocabulary into a more generic description. However, one focuses on data, while the other focuses on domain problems.

**Visual/Interaction technique:** The third layer focuses on the design of the visual and interaction idioms and attempts to maximize user perception of data and minimize problem-solving times, where the definition of both occurs in the previous layer.

**Algorithmic:** This last layer focuses on the construction of the algorithms that implement the previous layer designs automatically.

With each layer possessing different threats, they also require different forms of validation. Some threats and validation methods are listed on image 2.3, and each work should only approach one or a subset of layers.

One other distinction must be clarified; this is between upstream and downstream parts that correspond to the top half and the bottom half of validation methods respectively. The upstream validation methods can only provide us with partial validation over its layer and should be used for design refinement before implementation. The downstream counterpart provides the validation required for some system but it needs to be fully implemented, as the nested structure suggests.

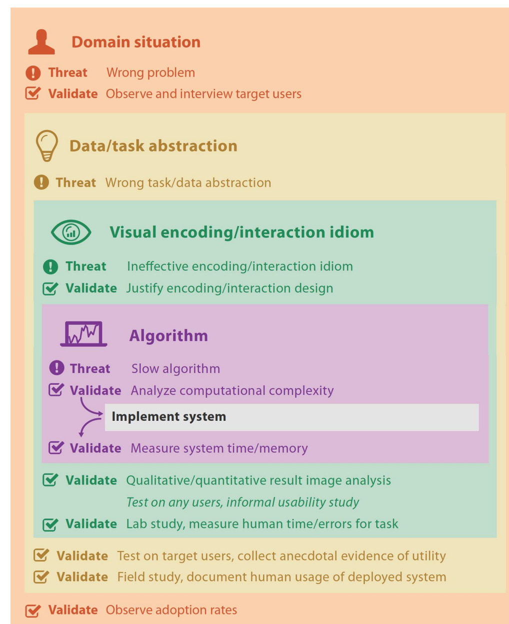



Figure 2.3: Construction and validation framework. Adapted from [11]

#### 2.1.2.1 System Construction Approaches

The four-level design framework possesses two commonly used approaches to start the construction and aid validation of a visualization system. Those approaches are problem-driven and technique-driven. The problem-driven method consists of a top-down iteration of the framework where the visualization designer starts with a real-world domain problem and analyses the domain users with a field study. Generally, with this approach, the challenge lies in a good abstraction from the domain situation where it is usually possible to match to an existing visual encoding idiom.

With a technique-driven approach, that is an opposite approach to problem-driven, where the iteration made over the framework starts at one of the two bottom levels. If the starting point is on the third level, it means that the designer possesses some new visual encoding or interaction idea. If the starting point is on the bottom layer, it means there exists some new idea in increasing the algorithmic performance of some idiom.

### 2.1.3 Visual Encoding Taxonomy

To fully understand VR potential one must first understand the primitives used by human visual channel. For this, it is addressed one possible taxonomy, from [11], and formalize visual encoding that makes part of the "How"component on *Vis* systems framework.

The taxonomy makes a first high-level break down into marks and visual channels.

**Marks:** There are four possible classes of marks classified according to the number of spatial dimensions that are required to draw them. Those are points (0D), lines (1D), areas (2D) and volumes (3D). There are two different applications of these marks. One is to represent the elements in data the other is to represent connections in data. However, the latest requires marks at least 1D or higher. This notion is best summarized with image 2.4.

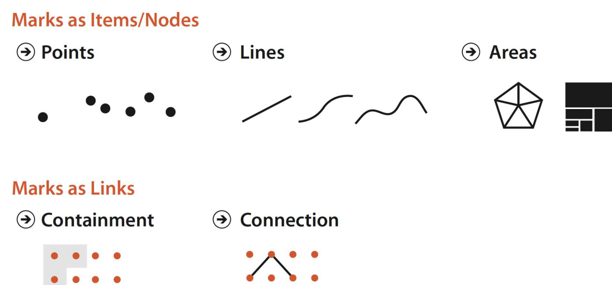



Figure 2.4: Possible marks. Retrived from [11]

**Visual channels:** Several visual channels control the appearance of marks, represented in 2.5, where for each visual channel is possible to encode/map some attribute about that mark. However, as the image suggests, it possesses two primary groups of visual channels, the spatial and non-spatial channels where the first is of special interest in this work.

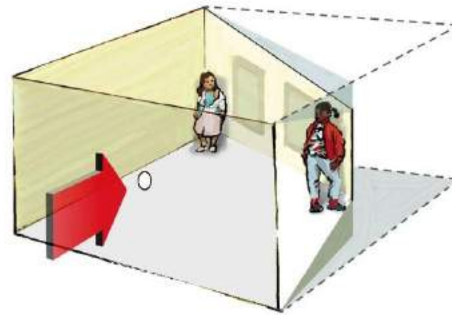
Figure 2.5: Visual channels for data encoding. Adapted from [11]

#### 2.1.3.1 Spatial Channels

The spatial channels are primitives of particular interest for visual encoding possessing a category of its own. Such importance is given because only with proper positioning

of marks is possible to start creating informative images of data. This notion is easily understood by conceptualizing two possibilities. First, several overlapped marks and second several marks spread out on screen. The information provided by the overlapped marks is significantly less than the spread marks, where even readability of other non-spatial channels reduces drastically.

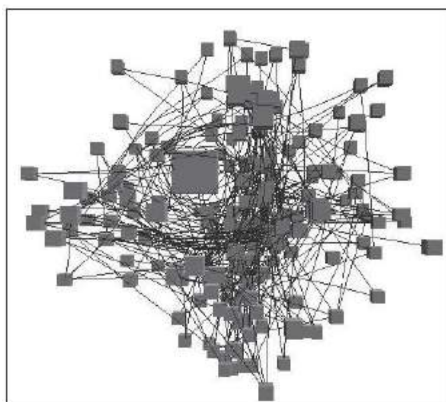
a ) Front view. Retrived from [26].



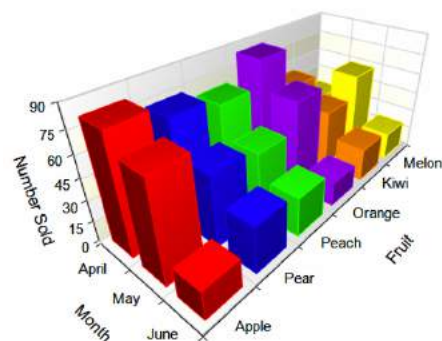
b ) Perspective view. Retrived from [24].

Figure 2.6: Ames room optical illusion.

To continue explanation of spatial channels, one last concept of human capabilities is required. The concept is that humans formulate a mental construction of third dimension (depth) in their brains. The construction is necessary because the image created in our cornea is just a two-dimensional image that is a point of view projection of our three-dimensional world. The brain uses several depth cues, to aid 3D construction, like stereoscopic vision, the disposition of shadows, occlusion and parallax movement. Such a process may sometimes lead to miss interpretations of reality. For example, the optic illusion of the room experience, pictured in figure 2.6a, gives the wrong perception of the room and the people inside where its true form is more perceptible in figure 2.6b.



a ) Data occlusion. Retrieved from [11]




b ) Perspective distortion. Retrieved from [5]

Figure 2.7: Examples of 3D disadvantages.

For *Vis* systems, the use of 3D can lead to wrong interpretations of data by the user, which is a non-desirable effect. Some examples of 3D characteristics that may affect data visualization negatively are occlusion and perspective distortion. The first is exemplified in figure 2.7a can be a major concern by hiding crucial information to the viewer. The second causes distant objects to seem smaller, illustrated in figure 2.7b, where comparison of some bars sizes can lead to doubts.

All the problems associated with 3D makes 2D more efficient on the majority of data visualizations. The "power of the plane"is well acknowledged by data visualization literature, with no distortions then providing a clear interpretation of data. 1D possesses the same good characteristics of 2D, although with one less possible channel, but it should not be overlooked if data is simple enough.

Nonetheless, all these disadvantages of 3D visualization are out weighted when the task is to understand the geometry of an object or scene, having even a greater performance if using interaction, as it is possible to examine in the study [23].

## 2.2 Virtual Reality

Broadly considered the next generation of man-machine interfaces, virtual and augmented reality, brings endless new possibilities for human in the loop systems.

Virtual reality will also be referred by its initials VR several times and also with the meaning of immersive VR and not non-immersive VR. Non-immersive VR is merely the display of 3D pictures in a standard monitor, while immersive VR concept was coined in the ’90s by Jaron Lanier and is explained in the next point.

### 2.2.1 Immersive VR Foundations

Jaron Lanier made the holodeck analogy as it was a mainstream concept to the public due to the Star Trek series. This analogy tries to explain the immersive effect in is ultimate form where, in the holodeck, was possible to create entirely new worlds and interact with it and its inhabitants as similar as in real life. Decomposition of this analogy is explained in [16], which defines the perfect immersive experience and derives from it, eight different themes that are at the core of this effect. However, only those that capture the prerequisites of the immersive effect will be addressed, and not those that reflect the possibilities that came alongside being immersed.

These four primary requisites are sensor diversity, spatiality of display, transparency of medium and natural language. Although in nowadays VR interface models don’t fully implement these prerequisites, they already provide a significant immersive feeling with partial implementation. They capture the essence of how we perceive and interact with the world and became the technical challenges to create VR.

#### 2.2.1.1 Sensor Diversity

All the sensory systems the human possesses, contribute in some way to the overall internal construction of our reality/environment. The understanding of how humans perceive through the senses is fundamental to fool them into creating the pretended world.

With the vision as the main human sensor, the ability to fully recreate this sense is fundamental, where a majority of prerequisites for VR involve tampering with vision directly or indirectly.

Sound is also used, in nowadays VR systems, to contribute to the immersive effect. With algorithms that spatialize sound, giving a sense of directionality, it is possible to recreate the way humans perceive sound in the real world.

The exclusive use of vision and sound on VR systems is due to their recording and reproduction technologies are well developed and cheap.

The gustatory and olfactory senses haven't been scrutinized by the developers of this area, mainly due to the affirmation that those senses would not add much more detail to the environment construction. Although there are several systems where the sense of smell can be recreated, those are still very rudimentary.

The major gap in today technologies that try to recreate how our senses perceive the world are those that try to recreate haptic sensations. Haptic senses convey a lot of information to the observer, and the introduction of such senses in VR systems may cause a new leap in the immersive effect experienced by the user. However, such technologies are still in a very primitive state.

#### 2.2.1.2 Transparency of Medium

Another prerequisite of immersiveness is the notion of not perceiving the medium that is reproducing the information that our senses are gathering.

Since the beginning of the computer age, the trend is to separate machine from man thought user-friendlier interfaces. However, as much user-friendly the applications might be, there is usually a clear separation between man and computer. With the display as a frame description of the computer world, within the real world.

Immersive VR is the technology that tries to eliminate this visibility of the computer, leaving only the user inside the generated environment. This transparency of medium is achieved by making the display occupy the user entire field of view so that the user doesn't see the screen itself.

Just by fully altering the vision, it is possible to entirely change the awareness state of the observer and fool it by not perceiving the medium of communication separating men and machine with the "perfect display".

#### 2.2.1.3 Spatiality of Display

So that is possible to enter a world, as three-dimensional beings, this world must be a 3D too allowing to perceive space. The act of presence, of being in a world, can be divided into three different components. Those are the sense of depth; the sense of surrounded; and the possession of a roving point of view.

The invention that best allows the user to perceive depth is the stereoscope that was created in 1833 by Charles Wheatstone. The invention uses two drawings, one for each eye, in such way, that fools the brain in unifying the images. These images differ slightly between them as it happens when visualizing the real world due to the separation of human eyes.

Combination of different technologies achieves the sense of being surrounded. By joining the concept of "perfect display" with the stereoscope that contains 3D images, it is possible to achieve the sensation of surrounded, of insertion in a different world.

The possession of a roving point of view is self-explanatory and allows to change point the point of view of the world. It is a fundamental mechanism that allows, for example, to trigger the parallax movement that is one depth cue used by the human brain. However, such movements in a non-immersive system are encoded, usually through keyboard use, and something else is required, that something is to navigate in a natural way.

#### 2.2.1.4 Natural Language

As roving beings, we already possess ways to interact with the physical world, such as movements of limbs, head or even facial expressions. Those are a set of tools that humans naturally have and used since their existence to explore the real world. This type of non-symbolic language is crucial for the immersive effect as we are using at all times navigating through the real world. This fundament is also mentioned as one of the most exciting possibilities of VR by Jaron Lanier quoting:

> *"There's also the ability of communicating without codes. This is a subtle distinction, but one that is very, very profound. ... I'm talking about people using their hands and their mouth, whatever, to create virtual tools to change the content of a virtual world very quickly and in an improvisational way."[8]*

Lanier makes it clear that this natural language doesn't have the purpose of substituting symbolic language but to complement it. This is because symbolic language is as part of the human communication systems as the non-symbolic language in current days.

Nonetheless, the introduction of natural language within the man-machine interface will facilitate some simple actions, like direct manipulation.

This objective is by no means a simple technological task, and numerous prototypes have been assembled that track parts of the user body, mainly head and hands. With the continuous miniaturization of sensors and with the increase in computational power to

track more sensors at the time, is starting to provide enough precision to allow natural communication.

### 2.2.2 3D Interfaces Considerations

With a 3D environment as one of the foundation requirements of VR, the interface mechanism of such systems must also be 3D. However, the problems adjacent to 3D are applied and user studies reveal that complex user actions, disorienting navigation and annoying occlusions can affect overall performance negatively [20]. For these reasons, a hybrid concept becomes a more modest approach in interface design. This approach consists of using 2D interfaces in a 3D environment that can be windows that leave shadows or icons that match real-world objects.

To guide the design of effective 3D interfaces in book [20] is presentment a list of basic and advanced features that assist the effectiveness of 3D interfaces:

**Basic:**

1. Use occlusion, shadows, perspective/ and other 3D techniques carefully.
2. Minimize the number of navigation steps for users to accomplish their tasks.
3. Keep text readable (better rendering, good contrast with background, and no more than 30-degree tilt).
4. Avoid unnecessary visual clutter, distraction/ contrast shifts, and reflections.
5. Simplify user movement (keep movements planar/ avoid surprises like going through walls).
6. Prevent errors (that is/ surgical tools that cut only where needed and chemistry kits that produce only realistic molecules and safe compounds).
7. Simplify object movement (facilitate docking/ follow predictable paths, limit rotation).
8. Organize groups of items in aligned structures to allow rapid visual search.
9. Enable users to construct visual groups to support spatial recall (placing items in corners or tinted areas).

**Advanced:**

1. Provide overviews so users can see the big picture (plan view display, aggregated views)
2. Allow teleportation (rapid context shifts by selecting destination in an overview).
3. Offer x-ray vision so users can see into or beyond objects.
4. Provide history keeping (recording, undoing, replaying, editing).
5. Permit rich user actions on objects (save, copy, annotate, share, send).

6. Enable remote collaboration (synchronous, asynchronous).
7. Give users control over explanatory text (pop-up, floating, or ex-centric labels and screen tips) and let users select for details on demand.
8. Offer tools to select, mark, and measure.
9. Implement dynamic queries to rapidly filter out unneeded items.
10. Support semantic zooming and movement (simple action brings object front and center and reveals more details).
11. Enable landmarks to show themselves even at a distance.
12. Allow multiple coordinated views (users can be in more than one place at a time, users can see data in more than one arrangement at a time).
13. Develop novel 3D icons to represent concepts that are more recognizable and memorable.

## 2.3 *Vis* Systems with VR

As discussed in point 2.1.3.1, the use of 3D in a visualization system has many disadvantages. However, VR interfaces compared to the keyboard, mouse and monitor interface, introduce more elements that are beneficial for the user perception of a 3D image and may mitigate several of the disadvantages in *Vis* VR. Several researches already have positive feedback of VR technologies on *Vis* systems. One particular paper,[4], that gathers several novel VR studies

one of its conclusions fallows quoted:

> *"VR has been shown to lead to better discovery in domains that whose primary dimensions are spatial."*

Some of the studies where VR revealed useful are palaeontology, brain tumour, shape perception, underground cave analysis, only naming few, where the common attribute is the spatial data attribute. Also, some studies support VR usefulness for collaborative tasks and remote experiences.

## 2.4 Problem Solving Environments(PSE)

PSE is a type of system that grants all sorts of computational facilities required to solve some target audience problems [6]. The system comprises of solution methods, either automatic or semi-automatic and ways of easily integrate new solution methods. The system also uses a language that is easily understandable by the target audience and provides easy access to computer resources without specialized knowledge of them.

Current days PSE usually possess a graphical interface sometimes used to visualize data and are of great importance in many research fields. This broad search is leading

to more generalized PSE environments that allow answering the needs of many science fields.

## 2.5 *SCIRun*

*SCIRun* is a specific PSE framework that allows assembly, debugging and steering of large scale scientific computations[12]. It is an open-source licensing software primarily funded by the SCI Institute's NIH/NIGMS CIBC Center. It provides a high-level control over parameters, to the non-specialist, efficiently and intuitively using a graphical interface and scientific visualizations.

The graphical interface provides a menu with a list of modules, depicted in image 2.8 on the left, where each contains its own set of input and output ports. The user may select them by dragging them on to the grid area an connect the module ports with the mouse and easily apply complex computations. The scientific visualizations allow the user to understand its data, and with interactive parameter control, understand the consequences of input changes.

The framework possesses some features that allow it to fulfil the requirements of PSE systems and more. In the following points its discussed such features.

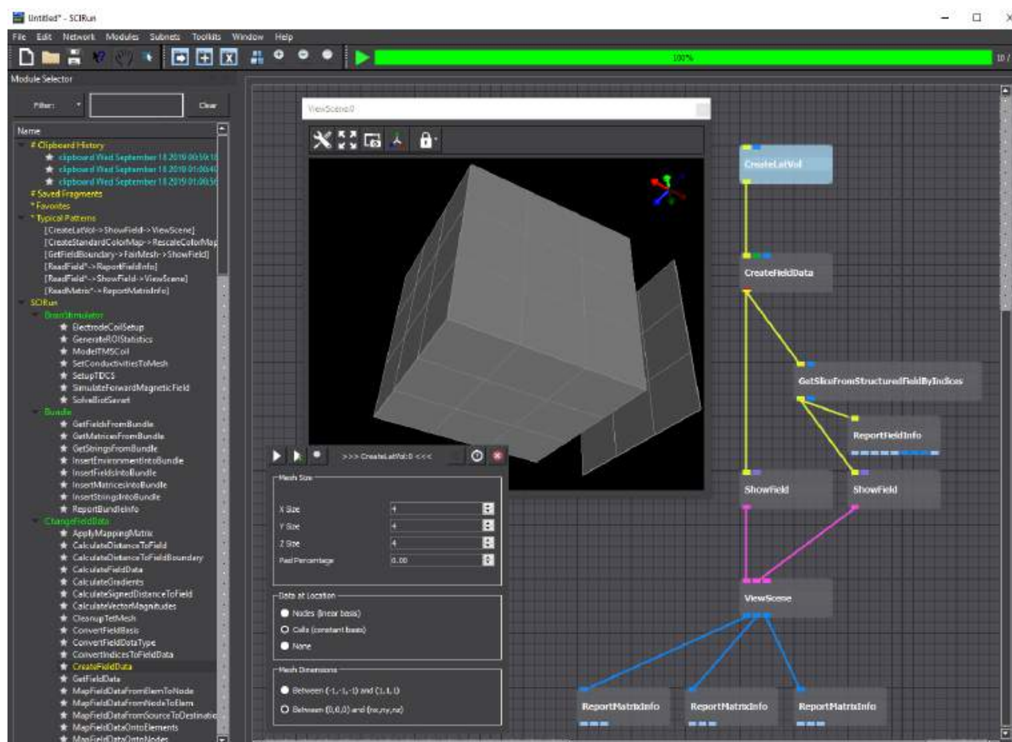



Figure 2.8: The interface of the SCIRun system framework. On the left module list, on the right data flow network on center top a visualization widget and on center down is one module interface.

### 2.5.1 Modular Architecture

*SCIRun* modular architecture is the heart of this PSE system and is crucial to understand such methodology for both users and programmers.

Each module represents an algorithm or function that can perform the more diverse of tasks. Each one, depending on its functionality, may have input and/or output ports or even be managed dynamically. The four major groups of performed tasks by this system are data reading, geometric modelling, numerical analysis and scientific visualizations. By combining such tasks, through the graphical interface, the user can perform diverse and complex computations. Additionally, each module may possess its how interface for parameter changing, image 2.8 center down, allowing for even better control of the system and possibilities of execution.

### 2.5.2 Data Types, Ports and Connections

The framework possesses a variety of data types that represent a good portion of scientific data [21]. Besides the more common data types, like *Strings*, and primary types, like *integers*, some of the available complex data types are as follow: (1)Mesh, (2)Surface, (3)Matrix, (4)ScalarField, (5)VectorField, (6)Geometry (cone, cylinder, point, ...).

For each of the existing data types, there is a correspondent *portag* that can be used to assign some data structure into some module input or output port and are colour-coded to aid visualization. An output port can be connected to an input port by the user and allow passage of data from one module to another.

### 2.5.3 Dataflow network

The features addressed on the previous point allow to create a flow network, depicted in 2.8 the right. The network fallows a *data-driven* policy. Meaning that modules that possess no input ports (no dependencies) are executed, and modules with dependencies await for the availability of their data dependencies to start execution.

For each module, a new execution thread is created, and a scheduler manages their creation along with the management of any interface changes that require re-computation of modules.

The user must have caution in the creation of dataflow graphs because it may create cyclic dependencies leading to deadlocks on the execution.

### 2.5.4 Module Development

The development over this tool uses C++ language, and it requires Qt, Git and Cmake modules for source code compilation. The system is designed to compile and run on the three major operating systems(Windows, MacOs, Linux).

A module as four different components although only the first two are required and they are as follows:

**Module configure:** Consists of only one file(*modulename.module*) and is the root file that allows registration with the system. The following components must be discriminated in this file, however, only the next one is mandatory. Also some basic module information must be written and the relative paths for files used.

**Module Source:** This element is comprised of two files (*modulename.h* and *modulename.cc*). The main function of it is to declare the module ports. Nonetheless, if implementing simple functionalities, the required code can all be written here.

**Graphic Interface(Dialog):** It contains three different files. The UI file(*modulenameDialog.ui*) created on Qt tool, where at its core is just a *xml* file declaring buttons or text-boxes, along with their id names and several other attributes, on its tag nested hierarchy. There are other two files (*modulenameDialog.h* and *modulenameDialog.cc*) that manage the mapping of UI variables to other variables declared either on Module Source and/or Algorithmic component files.

**Algorithmic:** Is composed of two files (*modulenameAlgo.h* and *modulenameAlgo.cc*). Usually, they are meant to contain all the computation of the module and where the previous two components are meant for gathering the information needed to process such computations.

A more detailed explanation over module implementation exist in *SCIRun* documentation [19] along with example code.

### 2.5.5 Limitations

The major limitation of this framework, in context with the work, is the unavailability of an easy way to integrate any virtual reality system. Its interface and visualization widgets target the standard computer interface comprised of a display, mouse and keyboard, as the vast majority of human-in-the-loop computer systems does. The significant divergence between both interfaces in their ways operating and with the much greater complexity of implementing VR interfaces, forces this work to search a new framework that may lead to a more feasible development of a system that uses VR.

## 2.6 TOMO-GPU System

The project Tomo-GPU was founded by FCT/MCTES, and it developed a system with several modules using *SCIRun* framework. The objective is to assist material engineers in the analysis of tomographic images to aid the research of new composite materials.

Tomography is the scanning of some matter through the use of X-rays. The raw data generated from it corresponds to a 3D-matrix that represents a regular 3D grid of space, although in memory behaves as a one-directional vector data. In each position is stored one single value, more specifically a single byte int ranging from 0 to 255.

The implemented models provide several computations over the described data and are as follows:

**Segmentation:** This module possesses two different ways of process, those are segmentation and bi-segmentation and are selectable by its graphic interface. Both of them have the objective of deriving only three colours from the original dataset, white with the value of 255, black with the value 0, and grey with value 127. However, segmentation possesses just one input and splits based on that one value. Bi-segmentation haves two input values splinting the black and white areas with a broader range of grey values.

**Hysteresis:** The module functionality consists in transforming all grey cells into black or white through analysis of its neighbourhood, leading to a fully divided space of black and white.

**Image Labeling:** Consists of identifying the reinforcements/objects within the data where the objects consist of clusters of black cells. With such indentification a new storing structure is created. Instead of storing each cell with a value, only cells that belong to objects are stored, creating a more compact file.

**Image Cleaning:** It requires the objects to be discriminated, and its process is to remove the reinforcements that possess less than some inputted amount of cells.

**VisAttributes:** Its a module for visualizing the attributes obtained at the image labelling module.

The normal use of the implemented modules is sequential one, with the order presented on the explanation list. Some other native modules of SCIRun must be used for visualization and data reading purposes. In image 2.9, it is demonstrated such standard use of TOMO-GPU system.

## 2.7 Unreal Engine 4 Framework

The unavailability of VR support within the *SCIRun* framework, see 2.5.5, lead to the consideration of another framework that could handle VR heavy demands. The first needed requirement was to allow real-time renderization of an environment to "insert"the user along with the data to be visualized. The second requirement is an easy integration with the available virtual reality equipment on campus.

Two frameworks fulfil such demands; those were Unity engine and Unreal Engine 4 (UE4). At initial stages, the unreal engine was chosen because of previous knowledge of the tool. However, an available study [22] showed some quantitative analyses by bench-marking the two systems on a simple 3D Pac-man game and formulated some conclusions. The conclusions that UE4 performed better were: First, with the increasing

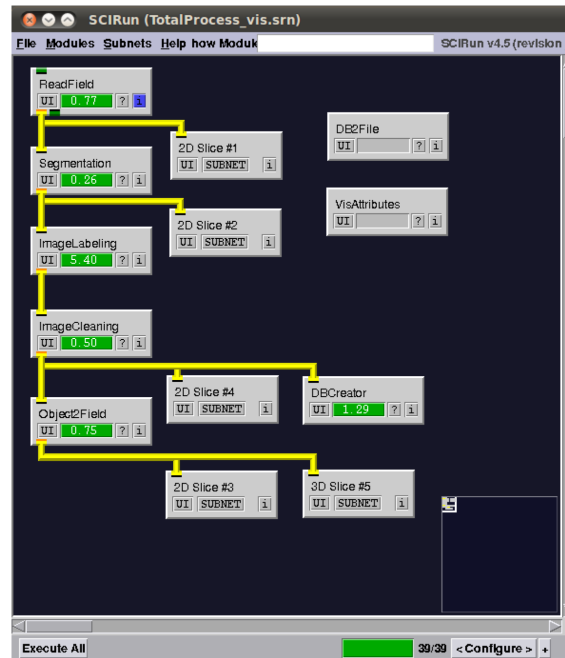



Figure 2.9: Standard execution of Tomo-GPU. Retrieved from [25]

vertex count on the map, it performed better in terms of frame rate. Second, the average frame time with VR system was lower. Third, by lowering the render setting the relative speedup was greater. Some negative conclusions were made about UE4. The only one relevant is: with the increase of projection size for an increased resolution, made frame rate time rise more.

To be able to develop with Unreal Engine, some concepts and structures used are essential to understand and are discusses throughout this section.

### 2.7.1 Architecture Basics

The unreal engine framework provides a wide variety of classes that allow the implementation of the most varied of functionalities from lighting to mesh insertions. However, four classes are the building blocks of the architecture:

**UObject:** The base block of the unreal engine where the vast majority of classes have it at their root. It registers the class on the system and provides it with some core services: Reflection properties and methods, serialization of properties, garbage collection, finding UObjects by name, Configurable values for properties, networking support and c++ blueprint communication.

**AActor:** Is a UObject that can be placed. It possesses position, rotation and scale and are the building blocks of world construction, not only, and where many classes derive from it. Some examples are AStaticMeshActor, APointLight and ACameraActor.

**UActorComponent:** Is a class meant for implementing shared functionalities across actors. Actors are usually assigned with higher-end tasks, while components are, typically, for lower-end tasks.

**UStruct:** Classic c++ structs, for data container construction, that benefit from services provided by UObject class.

Some additional relevant classes are:

**APawns:** Are basic classes for implementation of moving objects either controlled by IA or player controller.

**ACharacter:** Extends the pawn class and is specially design for user use possessing some additional functionalities.

**UProceduralMeshComponent:** A component that allows creation and animation of meshes on runtime with explicit declaration of vertexes and triangles.

**FNonAbandonableTask:** A class that allows to easy implementation of async threads that must terminate.

**UInstancedStaticMeshComponent:** Its a component with the functionality of replicate one previously associated static mesh in an efficient way.

**UMaterial:** Object that possesses reference to a single material. Materials are complex structures with lots of properties. Those properties may be parameterized to allow the dynamic creation of materials by affecting such parameters with code.

**UMaterialInstanceDynamic:** Its a specific instance of a UMaterial.

One specific class that deserves special consideration is called AGameMode. Such class is where is possible to specify many options, mainly game oriented, like if the game can be paused or not. Another main feature is the mapping of some classes to core variables of the engine. Those variables are game session(AGameSession), game state (AGameState), player controller (APlayerController), HUD(HUD), default pawn (APawn), spectator (APawn) and spectator controller (AController).

UE4 uses a custom prefix notation. This notation is to identify the inheritance of classes where the used prefixes are: T for template classes, U object classes, A for actor classes, I for abstract interfaces, E for enums and F for other types although some subsystems posses their custom notation.

As mentioned before, UE4 has a reflection system that drives many useful systems. A reflective system, in computational terms, is a system that can monitor and even change its elements. A special annotation language exists, called property system, to access and manipulate some options of the reflection system where UE4 class wizard automatically generates many of them. Some of the most elemental are: UCLASS(), USTRUCT(),

GENERATED_BODY(), UPROPERTY() and UFUNCTION(). Every markup has input parameters that allow changing behaviour, for example, the change policy on editor or if it is visible on blueprints system.

### 2.7.2 Development

There are two major ways of developing a system with UE4, and one must be selected on project creation. Those are editor only or C++ with editor.

The editor is a central part of unreal engine development with its default main interface depicted on 2.10. It is basically a 3D environment builder with a vast amount of functionalities, and it consists of 5 different panels. The world viewport in the center where the environment can be inspected and modified. The word outliner on the top right that hierarchy lists all world instanced objects. The detail panel on the bottom right that lists the details of a selected object. The content browser on the bottom to manage all assets and objects created. Finally, the modes panel on the left that contains the most varied of tools to add and manage the classes that UE4 provides, on some world.

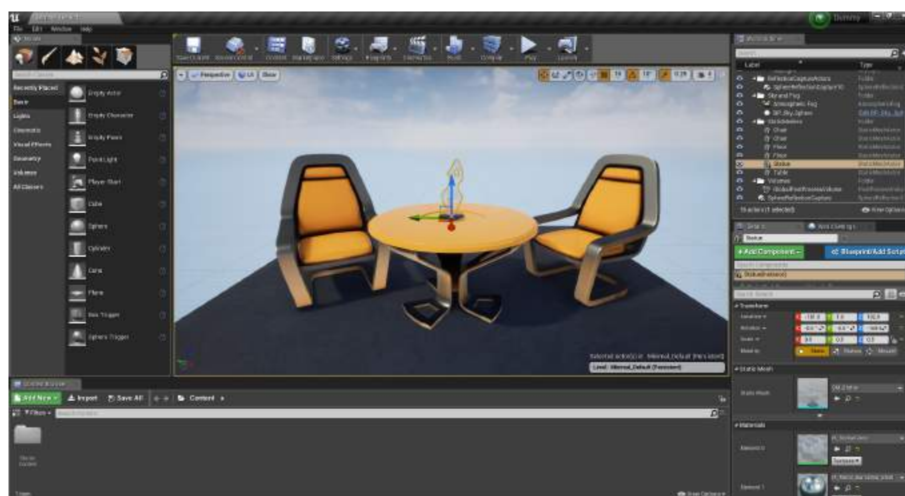


Figure 2.10: Unreal Engine 4 editor interface.

The major bifurcation of development is on logic implementation, where on the editor can be done using the blueprint system, discussed in the next point, or externally with Microsoft Visual Studio(VS) editor using c++ code. To use VS, one must specify on project creation to generate, also, the VS files and it possesses many tools to bridge functionalities with UE4 editor.

### 2.7.3 Blueprints

Blueprint is a system that can be used in on unreal engine editor, and an example may be seen in 2.11. They essentially are tools to aid the non-programmer in introducing logic on the system. They function through modules with the drag, drop and connect scheme, very similar to *SCIRun* dataflow network 2.5.3. The use of such system is almost

as programming where module functionalities vary from the basic *If* or *For* statements, to complex functionalities of mesh loading, texture mappings and so forth.

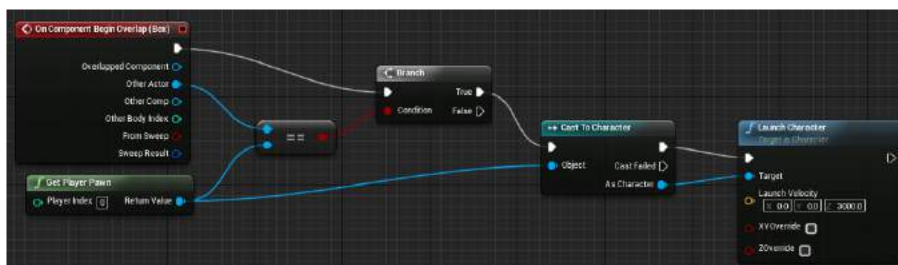



Figure 2.11: UE4 blueprint system. Retrieved from [1]

The white line represents flux of execution, where other colour-coded lines represent data connection with colour representing its type. Each module possesses two types of ports, for flux lines and for data. At the left of the module, is their input ports, with one flux and n data ports, at its right is its output ports, with at least one flux and n data ports.

Such high-end system has its costs and processing large quantities of data with blueprints is inefficient. However, general best practices used them for event triggering management or bridging information between actors, for example.

### 2.7.4 Attachment System and World Composition

A typical structure found in real-time renderization engines is the object hierarchy n-ary tree. This system allows the implementation of many functionalities. A primary functionality is the implementation of relative positioning being very useful in the creation of environments where to find some object absolute position, one must iterate the transformations of the object parents until the root is reached. On Unreal Engine isn't mandatory to use such relative positioning and such n-ary tree may be used as a list. However UE4 leverages on such structure in many different ways but the main purpose is for the reflective system to keep track of all spawned entities in the world.

To use such system the engine denominates attachment the act of creating a new relationship in such three. Parent and child objects must be given, along with some other information like relative(or absolute) translation, rotation scaling and some other characteristics depending on the nature of the object attached.

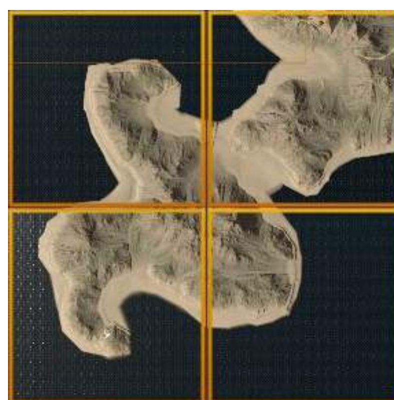


Figure 2.12: World with four levels. Adapted from [13]

There are two important concepts to build an environment, those are level/map and world. A level is the root component of one hierarchy three, and the world is a grid of several levels. This system potential lies in the creation of large environments

allowing partitioning the amount of objects that need loading depending on user positioning on the map. This concept is better explained with the image 2.12 where each yellow square is a level or map, and all of the squares are the world. Nonetheless, for simple environments, only one level may be used.

One last concept is that for each world, one AGameMode class must be associated and as explained in 2.7.1, such class possesses several references of some basic classes that are required for a world to be "playable".

### 2.7.5 Execution Flow

In this point is made the distinction of two separate flows, the launch flow and the active cyclic flow. The launch flow as also two major branches, the launch branch from the editor and the launch from the standalone version. There are some differences due to some pre-loaded elements that are required on the editor. Nonetheless, whatever may be the launch flow the GameMode class is of central interest here. It contains, usually, the first logic introduced by the developer. Also, important notice is that the launch flow possesses many stages and such stages must be respected on the system development to avoid errors.

The initializations made system launch provides the the elements to start actors and components life cycle. These objects life cycle may start on world creation, or if spawned, nonetheless these stages are almost the same and can be accessed with some built-in methods. Some of them are *PostActorCreated()*, *BeginPlay()* and *BeginDestroy()*. The mention methods are event driven and represent, usually, a small part of the object life cycle.

The cyclic flow of the unreal engine is at the core of this framework and is the functionality that, literally makes "time"move forward in the system. Internally it also works as an event driven method but is called cyclically to update the associated object, representing most of the object life cycle. Actors and Components can leverage from it with the use of the built-in method *Tick()* that it is called at every frame. Inside the engine it is what triggers all rendering functionalities after all inputs, IA's, physics and many other systems have made their changes. Such a demanding process should lead to carefully consideration, where significant blocks of computation can reduce the system liveness.

### 2.7.6 VR Support

The unreal engine provides full support for two VR hardware systems. Those are OCulus Rift and HTC vive head set, where the latest is the available equipment for test and development.

The management of the SDK of each possible VR set is throght the plugin system. There are a vast number of native plugins on UE4. Usually they possess functionalities that are very specific and ain't required on a majority of projects. Because of it they are considered plugins and may be added to a project on demand.

## 2.8 Prototype Framework Characterization

This section has the purpose of assembling some information gathered through the chapter and applies the framework assessed in the specific point 2.1.1, to be able to describe the prototype. Such description will allow to categorize it and examine the current state of the art for such categories.

First, the system is for integration with TOMO-GPU project, that was design to help material engineers on the development of new composite materials. The data handled by the system corresponds to a 3D grid that is a representation of a regular grid of space, with some additional meta-data extracted. Nonetheless, the geometric element of data is the key component of this work because it is the one that would benefit the most with VR visualization. Has mentioned in point 2.3, VR leads to a better understanding of data that its main characteristic is spacial. However, current TOMO-GPU implementation over the *SCIRun* framework didn't possess support for VR interface leading the development to use Unreal Engine 4 framework.

With some knowledge over data, task, and development tools involved for visualization it is possible to initialize a prototype description by accessing the three major components of the *Vis* framework, with some additional help of current state of the art literature.

### 2.8.1 Geometric Data Abstraction(What)

With the focus on geometric data, having a abstract characterization of it will prove useful to describe design stages of the prototype. One possible characterization is presented on [18], and classifies geometric data sets as a structure with its associated attributes where this structure has a topology and a geometry component.

The topology consists of the connections established between vertexes if any. Possible topologies are depicted in image 2.13 along with exemplary numbered ordination that can be used to conventionalize vertex order in memory. A particular characteristic of this abstraction is its invariance under traditional geometric transformations(scale, rotation and translation).

The other major element, geometry, refers to a particular instance of a topology where positioning in space is specified.

The two elements describe a cell in its whole and are usually associated with some attribute data. Some common manifestations of such data are scalars, vectors, tensors, normal's and UV's.

To form a data set, cells must possess some type of organization. Usual structures are figured in 2.14 and are grouped into two primary categories, structured and unstructured. Structured means that there is a single mathematical formula to describe the relationship of cells allowing significant memory savings. Unstructured must be explicitly represented having higher memory costs.

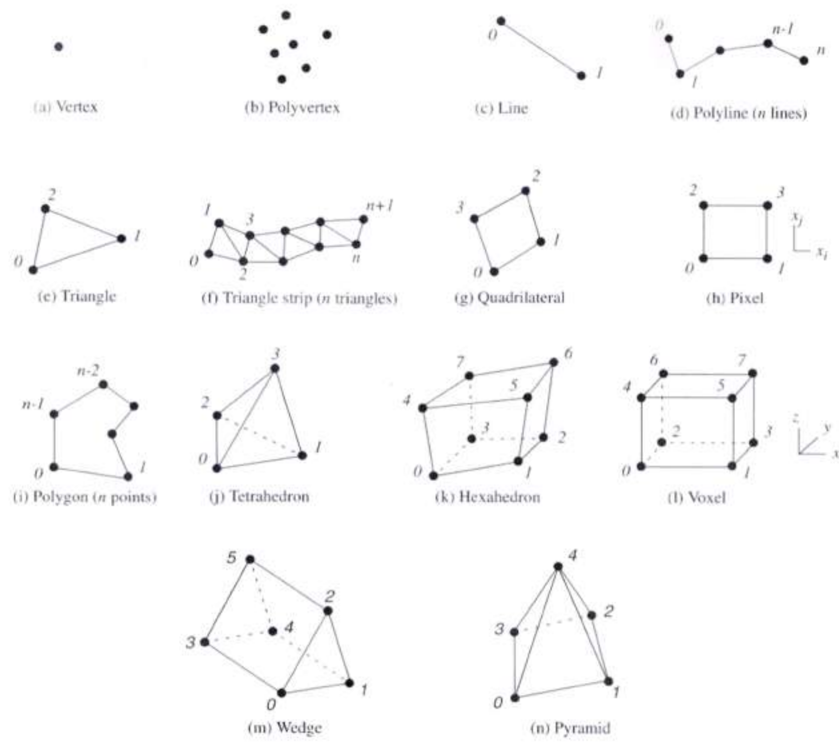



Figure 2.13: Geometry topology abstraction. Retrieved from [18]

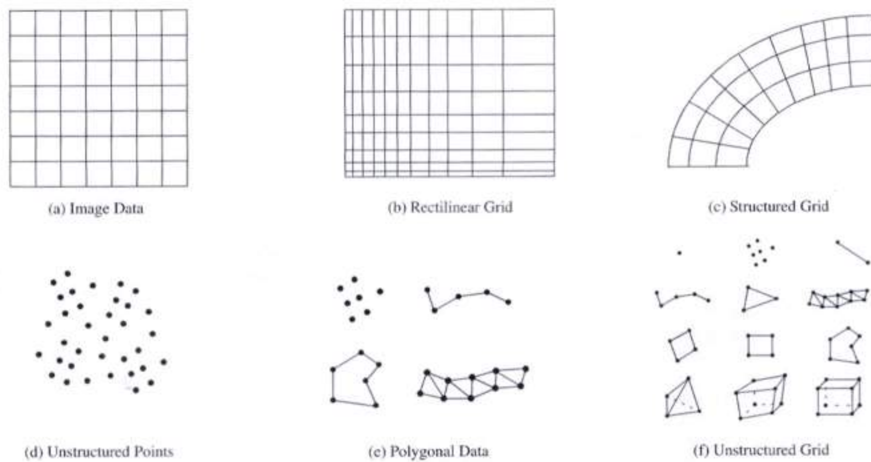



Figure 2.14: Geometry dataset structures. Retrieved from [18]

Two different topologies structured into two types of data sets heavily influence this work. The first combination is the voxel topology structured into image data. Image data organization refers not only to the easily associated 2D images, but also to 1D lines and 3D voxels. Such a combination provides the first classification of data used by the TOMO-GPU system 2.6. The tomography scan provides a regular grid of space(3D voxels) and is associated with one value(scalar).

The second combination is triangle strips with polygon data organization. Polygon organization is an unstructured one and falls in the range of UE4 engine structure. Such engines designed for the game industry have the objective of creating realistic environments where usually no mathematical formula can express all of the cells positioning. Triangle strip is a type of topology assembled with 2D triangles where those may share vertexes and edges. Such triangles don't need to be coplanar and can then be used to describe complex 3D surfaces. It's the primary method used in real-time rendering engines to describe such surfaces because it is more efficient for real-time rendering compared to other complex surface descriptors like quad strip or poly strip.

### 2.8.2 Material Engineer Tasks(Why)

The objective of material scientist engineer is the construction of new composite materials and techniques for such development. Composite proprieties depend not only on the elemental materials used but also, on the spatial characteristics and other chemical and physical phenomenons.

This work focuses mainly on the interpretation of the spatial characteristics. Some work on TOMO-GPU project [3] already provides many geometric meta-data. Those are centroid, oriented and axis-aligned bounding box, volume, surface area and principal component analysis. Such information provides only a resume of spatial data not allowing to perform some of the tasks in the geometrical realm, for example, shape recognition. Nontheless, SCIRun provides visualization modules for this type of data but is not suitable to explore the VR potential in 3D visualizations.

### 2.8.3 Visualization Algorithms of Geometric Data(How)

In this point, a classification algorithms, retrived from [18], used to transform geometric data will be presented, to establish the useful the ones for this work.

The classification possesses two primary components: structure and type. The algorithm structure is the definition of changes made in topology and geometry. The algorithm type, is the type of data operated on, and also the type of data generated, if any.

Structural classes are: (1)Geometric transformations, that change only the geometric part of data; (2)Topological transformations, that change only the topology part of data; (3)Attribute transformations: that convert attribute from one form to another; (4) Combined transformations, different combination of the previous three types.

Type classes are: (1) Scalar algorithms, operate on scalar data; (2) Vector algorithms, operate on vector data; (3)Tensor algorithms, operate on tensor fields; (4) Modelling algorithms, that may generate topologies, geometries or attribute data that operate over more than one previous class and is meant as a catch-all class.

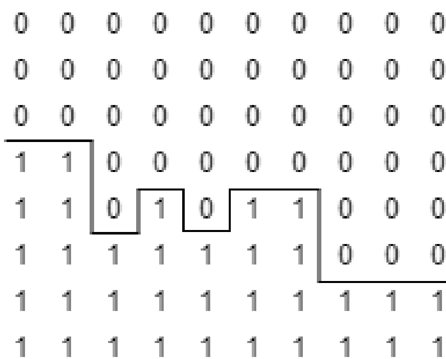
With the knowledge over data, gathered on 2.8.1, and this algorithm classification is then possible to conclude the class of algorithms needed.

The algorithm structure needs to operate over scalar data and possess topological transformation from a regular grid(voxels) to an isosurface represented by a triangle strip, as required by the Unreal Engine framework. Such type of algorithms are well known in the computer field of graphics and visualization and are usually known as contouring or modeling algorithms that linearly interpolate the surface cell by cell of the regular grid using some technique.

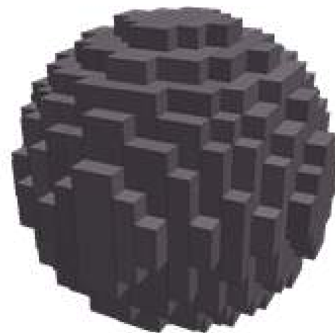
With it many possible algorithms, in the following points it will only be addressed those that are implemented on the final prototype and those that heavily influence this area of knowledge.

#### 2.8.3.1 Simple Cubes

The concept behind the simple cubes algorithm is to process each voxel by looking at its six face neighbours and draw a square face between filled and unfilled voxels. A slice side view of the algorithm behaviour is depicted on 2.15a and a possible resulting mesh of a 3D sphere mapped on a regular grid on 2.15b. The process creates a squared strip, but each square may be subdivided into two triangles to formulate a triangle strip required.




a ) Slice view.



b ) 3D view of mesh.

Figure 2.15: Simple cubes algorithm.

The advantages of this algorithm are the easy of implementation, the nonexistent slivers(odd shapes formed on the mesh) and is locally independence, meaning that the process in each voxel does not account for the process of any other voxel. However, the produced mesh is rough, and features of its true form may be lost.

#### 2.8.3.2 Marching Cubes

To understand the marching cubes algorithm first, we will address its simpler 2D version on a pixel topology denominated marching squares or contouring. The algorithm requires the scalar values to be attributed to each vertex. The threshold of the scalar value must be selected, deciding if a vertex belongs inside or outside of the contour. The binary classification of each vertex results in a $2^4 = 16$ possible combinations in a pixel topology, depicted on 2.16.

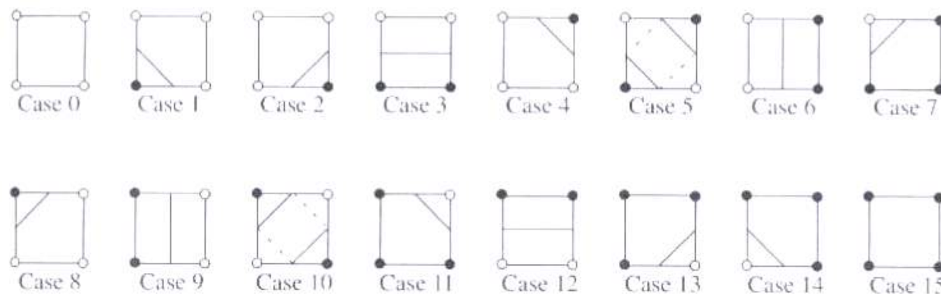



Figure 2.16: Contouring algorithm cases. Retrieved from [18]

The marching cubes algorithm applies the same principle as the marching squares. However, the possible neighbourhood combination is much larger. By attributing the scalar value to a voxel vertex and classify each as inside or outside, it totals on $2^8 = 256$ possible cases of a neighbourhood. There are many mirrored cases on these combinations, and all combinations may be resumed to 15 different topological situations depicted on image 2.17.

Compared with the previous algorithm, it has a much harder implementation and it may generate slivers on the mesh. However, it smoothens the object mesh and approximates to it's true form and is still a local independent process for each voxel.

Many algorithms derived from this one, the majority reflects one great problem that is the lack of expressivity [17]. This deficiency leads to the loss of thin features of an object meaning, if two vertexes of the real object lye inside of a voxel that information is lost. This problem can be counter by making a finer grain grid. However, that approach can quickly generate a large number of polygons.

#### 2.8.3.3 Dual Marching Cubes

The dual marching cubes algorithm was developed to resolve the major problem that exists in the family of algorithms derived from the marching cubes algorithm discussed in the previous point.

The algorithm uses an octree structure, that is a typical structure used to represent grided volume where the eight children of each node are the respective octants of the parent volume and space is recursively subdivided to create a finer grain grid. The dual marching cubes adaptively samples this structure and allowing a significant reduction

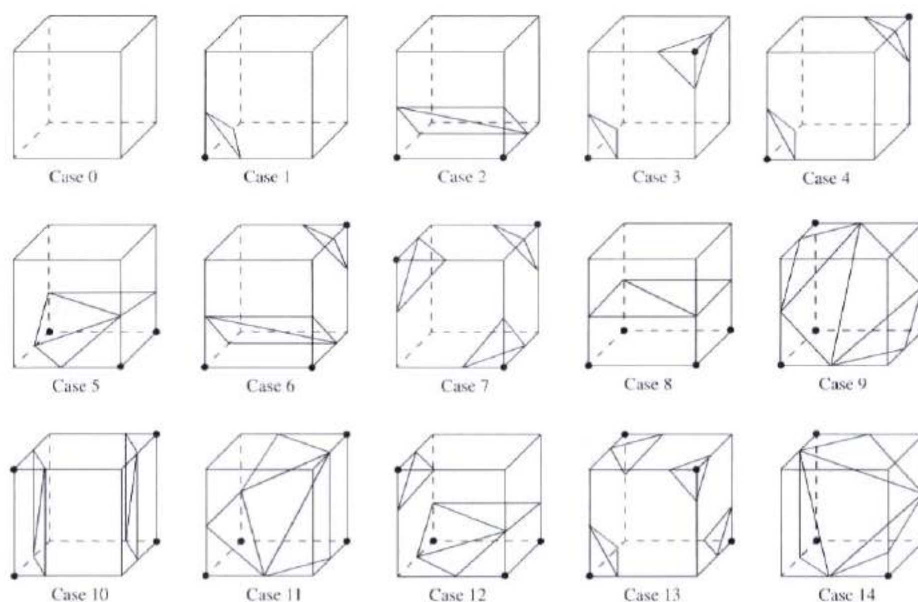



Figure 2.17: Marching cubes algorithm base cases. Retrieved from [18]

in generated polygons. A more detailed explanation can be explored in [17] however in image 2.18 is depicted the result of dual marching cubes against marching cubes over a room with thin walls and it is possible to perceive the greater efficiency in reducing polygons generated.

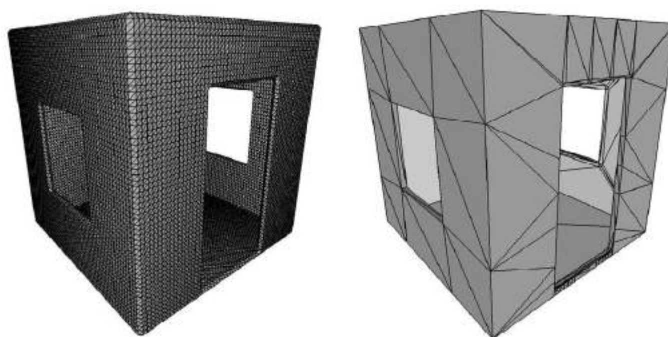


Figure 2.18: Results of Marching cubes algorithm to the left and dual marching cubes to the right. Retrieved from [17]

## 2.9 Summary

In this chapter, it was gathered the knowledge required for the development of a visualization tool for composite material research.

At first we looked into a possible framework for designing *Vis* tools. The overall framework is described in 2.3 and reveals the different layers of abstraction to take into consideration. Furthermore, such layers are divided into two different parts, the upstream and downstream parts where only the downstream part can completely validate a system.

Nonetheless, the upstream part must be taken into consideration for a good design choices and providing a map of content thought this chapter.

Based on literature an upstream validation on the domain layer is presented in subsection 2.3 and where the remaining layers are characterized on point 2.8. However before is possible to fully characterize the framework for the current problem first is necessary to explore the current used technologies and those are addressed in points 2.4 and 2.5. In this last point, the lack of support in *SCIRun* framework for VR interfaces was discussed. For that reason, it was explored another framework, in point 2.7 that allows easy integration with VR, however bridging between systems is still fundamental. With all upstream validation completed is now necessary to proceed with design and implementation to reach the downstream validation.

With this work only changing the technological medium it only affects the idiom layer more specifically the interaction idiom, and where the remaining previous layers and visual encoding were retrieved from literature. However, by changing interaction idiom layer, has the framework structure suggests, it affects it inner most layer named algorithm. Nonetheless this work will only proceed with validation on the first layer after implementation, meaning it will only test the viability of the prototype on an algorithmic level.

# Chapter 3

# Organization Solution

The introduction of a new framework that supports VR, lead the development of a system that comprises two complete different processes that exchange information between them. The first one is the already existing Tomo-GPU system and the second is the UE4 VR prototype denominated SCIRunVR. They are depicted on the component diagram 3.1 that provides an overview of the interaction and functionality of such components.

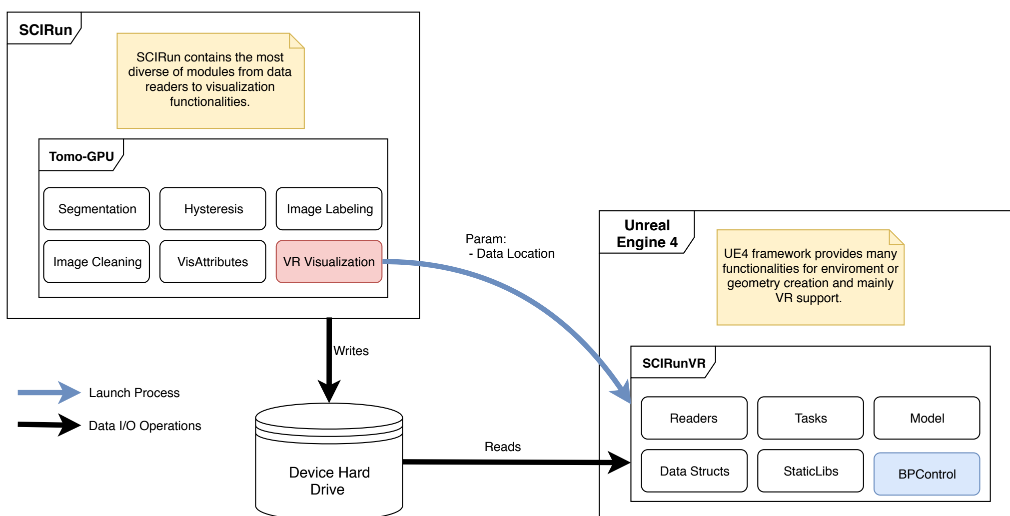



Figure 3.1: Component diagram for with overall functionality to integrate VR on Tomo-GPU system.

The intended use case with this system assembly is to allow the material specialist to use the Tomo-GPU system normally, and when VR visualization is required, the user may add the VR module. Colour-coded in red, the module is depicted on diagram 3.1. The use of such a module will initialize the implemented VR prototype with Unreal Engine technology.

Communication between components is made through the secondary memory of the computer. The reason for such approach is its ease of implementation. The used files can reach considerable dimensions, and if several files are to be transmitted, it can become quite useful the use of the secondary memory.

On this chapter, it is further analyzed each component separately and existing constrictions of the Tomo-GPU current implementation are considered.

## 3.1 Tomo-GPU VR Module

The module functionality is straightforward. It has to obtain, some way, the data location on disc and provide it, on launch, to SCIRumVR system. On point 2.5.4, two distinc ways of providing data to a module were considered: those are through ports or through module graphic interface, and both must be explored for implementation.

In the following is considered the structure of Tomo-GPU and its data files. There are two main file types used. First one is the *Bruno* file type that was specially developed for the Tomo-GPU system. Such files are organized by objects where each object contains the index of all its constituent voxels. The second is *NRRD* file types that are the usual input file on Tomo system. These files don't possess the distinction between filled voxels or empty voxels. They contain scalar values in each voxel, leading to significantly larger files then *Bruno* files. Nonetheless, as a standard type of file being able to use them on SCIRunVR system can prove beneficial, for example, in comparing the initial geometry inputted on a Tomo process with its final result.

## 3.2 SCIRunVR Prototype

As a ground-up application, it required many design decisions that need clarification. On this work, it is only considered a small portion of functionalities that may help material engineers on their research, compared to huge potential and possibilities that VR systems may offer. For that reason, all design decisions follow the primary rule of modularity. By providing reusability of its parts and ease of use for future developers of SCIRunVR it may lead to further research of this system potential.

Although modularity is a primary concern, efficiency both in terms of execution time and memory can't be forgotten. However, the most important performance aspect is in maintaining the system liveliness. On point 2.7.5 the cyclic flow of unreal engine is explained including how it triggers all sorts of tasks that update the environment and render the new image to be displayed. Such cyclic flow gives the system its liveliness attribute and, to maintain it, the main concern is not time or memory efficiency, but rather the good management of resources. Such management consists in a parallel approach to handle chunks of code that would halt the cyclic thread.

On point 2.7.2, two ways of developing with unreal engine were discussed. Only through the editor, where logic was added through the blueprint system, or with the

editor and visual studio to add new logic through c++ code. The chosen method is the c++ version due to its increased time performance compared to the blueprint system. However, best practices use blueprints for event dispatch and triggering and should be considered to bridge VR inputs to the intended functionalities.

Subsection 2.8 gives a complete system description, including a first higher component breakdown into the following components:

**Tasks:** Are computational chunks of code that may compromise system liveliness and must be managed in asynchronous way.

**Readers:** To target the intended files to complete the communication between the TomoGPU and SCIRunVR. Readers should be considered a particular type of task because they may also compromise system liveliness. However such particular functionality requires a distinct component of its own.

**DataStructs:** Should be devised to map appropriately to the two distinct data structures established on 2.8.1, along with any additional data required for its correct management.

**StaticLibs:** To contain simple computations, that don't compromise system liveliness, and that are required across several components.

**Model:** The central component with the main purpose of displaying the geometric information of a single file by spawning a representative mesh into the word system. However, to spawn such mesh is necessary to proceed with several computations, either asynchronously or synchronous, and consequently it must manage them accordingly.

**Blueprint Control Management:** Control logic for movement and trigger of functionalities on the model component that allows interaction.

# Chapter 4

# Implementation

In the previous chapter, the two principal components to implement, the Tomo-GPU VR module and SCIRunVR prototype were identified. The objectives/functionalities for each one to achive the primary goal of this work were also identified, including the communication between them. In this chapter, it will proceed with a thorough explanation of the implementation of each component separately and how such objectives/functionalities were achieved.

It will start with the Tomo module. It possesses a simple functionality and trivial implementation. Nonetheless, it is of significant importance providing the foundational information necessary for SCIRunVR that will be explained right after.

## 4.1 Tomo-GPU VR module

The objective of Tomo-GPU VR module developed is to initiate SCIRunVR and pass a path to it with the location of data on the disc. It must be a full path to a specific file or directory, where the latest allows the transmission of several files. On image 4.1 it is possible to visualize the module interface, its parameter interface and the position in all modules list.

### 4.1.1 Module Implementation

On point 2.5.4, the *SCIRun* module implementation was explained and four different components exist but only two are mandatory, where each contains its own files and functionalities. This module implementation makes use of three components: configure, source and interface.

**Configure:** A simple file where it is declared basic information like module name and

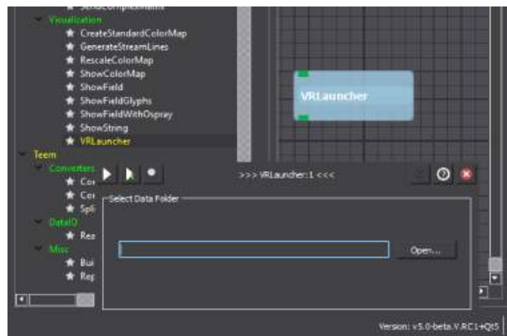



Figure 4.1: VR module with its interface inside *SCIRun* framework.

development status. It also provides the declaration of components used and their relative path(source and interface) and, components not used(Algorithmic).

**Source:** With the overall module possessing such a simple functionality, all needed logic is coded on this component.

One of the main characteristics of this component is the port declaration. In this case, there are only two ports. One input and one output, where both consist of a simple *String* type. The input port must be a valid path to be transmitted to SCIRunVR system. The output port echos the used directory that may be the module input or interface parameter. At the moment, since SCIRunVR build is only available for windows, the implemented code is only for launching processes on windows too.

**Interface(Dialog):** The interface component was added to give the user more flexibility on how he can use the system. The interface possesses a simple folder selection box, where a path can be written or found on system file explorer. Such functionality means that the module does not need to be connected to an existing DataFlow and still can execute and pass information to SCIRunVR. However, if the input is connected, it will override any interface input.

## 4.2 SCIRunVR UE4

This section consists of a thorough explanation of SCIRunVR system. As a ground-up application, it required the development of many functionalities that will allow the visualization of the geometric data on a VR environment.

On the previous chapter six different components that portrait six different high-resolution functionalities were identified; these components will be the building blocks for explaining this system. The first point of explanation will be an overview look of SCIRunVR architecture and identify all classes of each component and understand component interaction and available functionalities. After, it will be explained each class implementation by their respective component. Having all classes defined its then possible to proceed with the world construction explanation that will allow for the user to enter on a visual space and visualize the intended data.

### 4.2.1 System Architecture

With a system containing lots of parts and to ensure a full understanding of it first, one will take an overview look through two distinct class diagrams where one pictures the classes developed on c++, and the other, the ones developed on the blueprint system.

The c++ class diagram is depicted on image 4.2. Within, it is possible to identify five of the six high-level functionalities that were identified in the previous chapter, where they also provide organization to the folder structure of the c++ project. In each of them, we can visualize the existing classes that provide specific functionalities to the system and are colour-coded according to which class they extend.

One crucial functionality about this architecture can be extracted by observing the Mesh folder area in the c++ class diagram 4.2. The two classes allow two possible approaches for mesh spawning/construction. The *ATrigStripMesh* is a class that handles the direct input of vertex position and its trig-strip topology. The *ACubesMesh* class, as the name implies, uses a cube mesh created and triangulated previously, and replicates it in each voxel position.

Another crucial characteristic to retrieve is that the Model component, more specifically, the *AModelManager* is a central piece of the system that uses all other components directly or indirectly and manages all processes accordingly. One last characteristic is about the methods that are colour-coded in blue that means that they are exposed to the blueprint system using the UE4 special annotation system more specifically the annotation UFUNCTION().

The class diagram of the blueprint system targets the last of the six high-level functionalities, the Blueprint Control Management. Depicted on 4.3 and where classes are also colour-coded by their respective class extension.

The main conclusion from the blueprint diagram is the distinction between the classes that represent the user control, with the VR gear, from the ones that represent the user interface. The user control classes are the *MotionControllerPawn* and the *BP_MotionController*. The remaining ones are the interface classes where classes that extend *SWidget* consist in the construction of a widget interface that contains all the possible parameters for the SCIRunVR system and, the *UIDesk* class that contains a model of a table and where the widget is attached to it. Such desk actor is then added to the world system allowing the user to see it and interact with it.

With the complete overview of the six high-level functionalities required, the explanation of their implementation will begin by the peripheral classes of the c++ diagram, followed by the model and then the blueprint interface control. After it, it will be explained all the attachments established on world system along with the results of such world construction.

Extends: UObject AActor FNonAbandonableTask

Blue methods: Are exposed to the Blueprint system

Figure 4.2: SCIRunVR c++ class diagram.

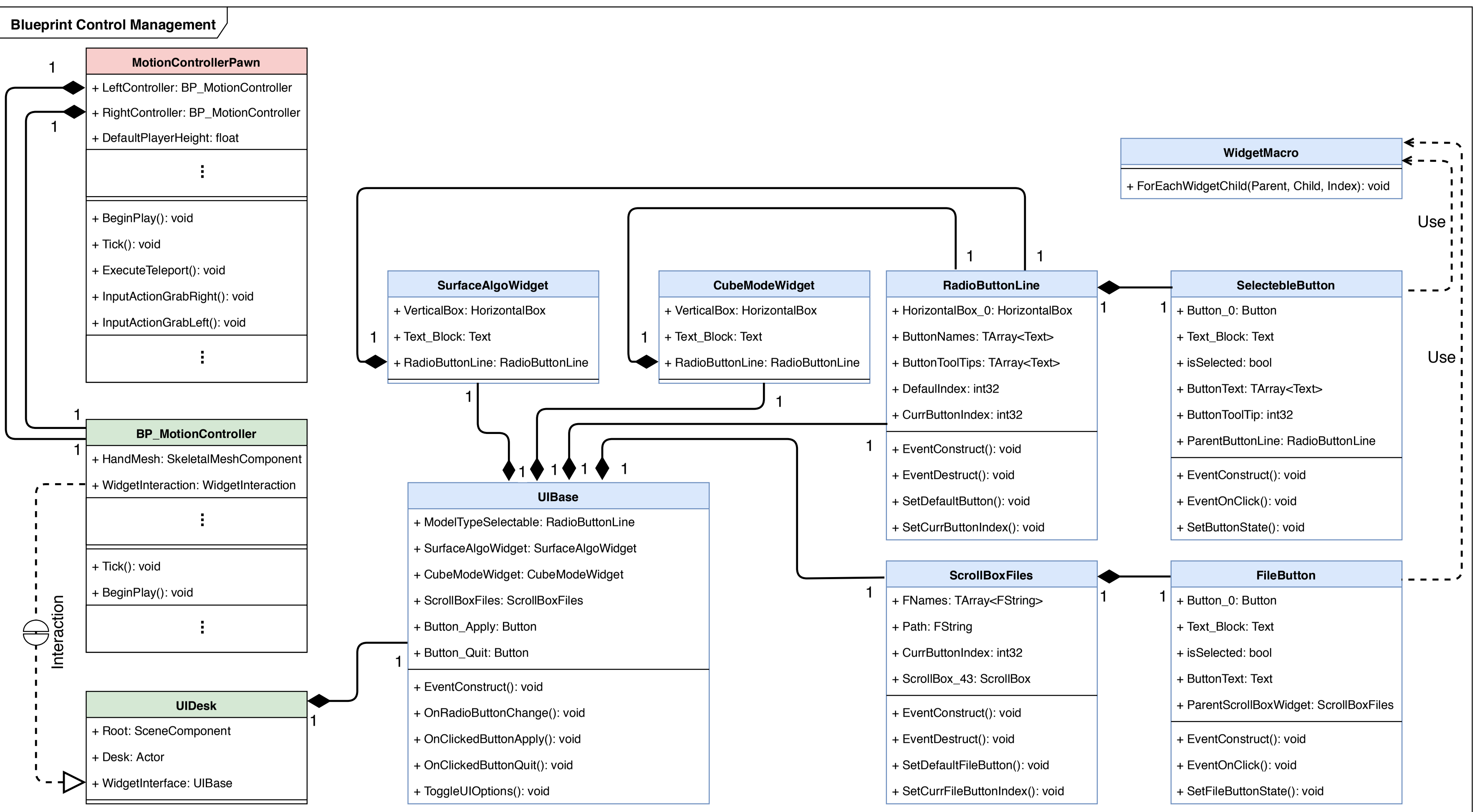



Figure 4.3: SCIRunVR Blueprints class diagram for control functionalities.

### 4.2.2 DataStructs

DataStructs component contains three classes to divide data structures according to its purpose. Two of them are of particular relevance, those are *VoxelData* and *PolygonData*. They are design to contain the two types of data identified on 2.8.1 and the root structs are named *FVoxelGraphData* and *FPolyObjects* respectively. The main concepts to retrieve are that both structs are organized by objects and also, on the voxel struct, only "filled"voxels are stored for memory efficiency and are organized by their voxel index. All structs implemented use the macro *UStruct* to leverage from all unreal engine features. The third class, *NRRDAux*, is a simple auxiliary structure to assist header reading of NRRD files.

### 4.2.3 Readers

Readers are considered a specif type task because they may compromise system liveness. For this reason, readers must extend *FNonAbandonableTask* class to run asynchronously from the cyclic thread. The two implemented readers target the files used on Tomo, identified on point 3.1. Those are the *FBrunoFileReader* and the *FNRRDFileReader* for Bruno and NRRD file types respectively. Independent of the file type, both readers produce one *FVoxelGraphData* struct and should be a rule to keep for future readers of voxel data.

In the listing 4.1, it is described a pseudo-code implementation of *FBrunoFileReader*. The file already possesses object distinction and at is head, contains information about data size and the number of existing objects. It is followed by a sequential loop that reads the object header containing an object ID and the corresponding number of voxels and is then followed by a parallel loop that reads the voxels of each object. Such implementation only allows parallelization of voxel reading because the object position is only known after reading the previous object.

Listing 4.1: *FBrunoFileReader*

```
1  //Global scope variables
2  FString filePath;
3  FVoxelGraphData* data;
4  bool * isWorking;
5
6  procedure DoWork()
7    ReadBrunoFile();
8    *isWorking ← false;
9
10 procedure ReadBrunoFile()
11   fh ← openFile(filePath);
12   if(fh = NULL)then error;
13   else
14     //Read file header
15     data.size ← fh.Read()
16     data.nrObjects ← fh.Read()
```

```
17    foreach i from 0 to data.nrObjects
18      data.object[i] ← fh.Read()  //Read object header
19      uint8[] tmpBytes ← fh.Read()
20      parallel_for k from 0 to data.object[i].size
21        data.object[i].vox[k] ← tmpBytes[k];
```

The *FNRRDFileReader* implementation is described in pseudo-code in the listing 4.2 and is quite similar to the previous reader. However, *NRRD* files don't possess object distinction and neither fill or unfilled voxels. For that reason, an extra value called *threshold* must be passed down and will allow to segment between filled and unfilled voxels. All filled voxels are assumed to belong to only one single object, and its index (not is value) is stored.

Listing 4.2: *FNRRDFileReader*

```
1   //Global scope variables
2   FString filePath;
3   FVoxelGraphData* data;
4   bool * isWorking;
5   int threshold;
6
7   procedure DoWork()
8     ReadNRRDFile();
9     *isWorking ← false;
10
11  procedure ReadNRRDFile()
12    fh ← openFile(filePath);
13    if(fh = NULL)then error;
14    else
15      //Read file header
16      FNRRDHeader tmp ← fh.Read();
17      data.size ← tmp.size;
18      data.nrObjects ← 1;
19      uint8[] tmpBytes ← fh.Read()  //read all voxels
20      parallel_for k from 0 to data.object[i].vox.size
21        if(tmpBytes[k]<← threshold)then //segment fill and unfill voxels
22          data.object[0].vox[k] ← k;
```

One last clarification about the high-level functionality readers is about the class *UReaderDeclaration*. It is a simple class that extends the lightweight *UObject* class because it isn't a reader but is where existing readers must be declared. Such class will then be used to filter the allowed files whenever required.

### 4.2.4 StaticLibs

This component is designed for implementing simple computations that may be required across the entire system and by themselves, they don't compromise system liveness. Every implemented class here should extend the UObject class, as it's the lightweight type of

class, allowing for faster execution times and giving all benefits of registration on the system.

There are only two implemented classes with simple functionalities. The *UConverter_VoxIndexCartesian*, provides a simple converter functionality from linear indexing to 3D Cartesian coordinates and vice versa.

The *UComLineInterperter* is an auxiliary class that interprets the file path given by the module created on the Tomo-GPU project. If the file path consists of a directory, then all the directory is searched for supported files with the auxiliary registration class *UReaderDeclaration*. In case of a file, the immediate up directory is searched with the same conditions. This class possesses one method exposed to the blueprint system to be called when needed by the user interface.

### 4.2.5 Tasks

With tasks providing functionalities with the most complexity, they also require to extend *FNonAbandonableTask* to run asynchronously from the cyclic thread. The more complex classes on this component are the ones focused on the implementation of algorithms addressed in 2.8.3, that focus on the computation of meshes from a voxel mesh structure to a triangle strip mesh structure. However, it isn't the only case, and an individual description follows.

#### 4.2.5.1 FCalculateEdgeVoxels

This task was specially created to use when the mesh being applied is *ACubesMesh*. This is due to the extremely high vertex count that is possible to reach with such an approach. By spawning a cube in every voxel position, including the interior of an object it leads to unnecessary use of resources that may hinder the liveness of the system. For that reason was created this task that removes the voxels that are surrounded by others. Meaning that they reside inside and then cannot be seen.

Within listing 4.3 it is a pseudo-code implementation of the central behaviour of the algorithm. The removal is achieved by changing the flag *isActive* to false in the array of voxels and causes them to be ignored when drawing the cubes. The task loops every voxel, in parallel, of every object, sequentially, and executes the needed verifications. First is verified if the voxel is on the border if so it will cause it to be visible from at least one direction. Second is verified if at least one neighbour doesn't exist. However, this verification is not a trivial one because of the way data is stored. By only storing the index of the filled voxels and not possessing the matrix-like organization, it is not possible to directly access the neighbouring voxel. However, because voxels are ordered in the voxel data struct, it is possible to perform a binary search index to speed up considerably the search. With some arithmetic is possible to obtain the index of all its neighbours and then search for them and with the first missed searched causes the voxel to be activated. When all adjacent voxels are found then causes the voxel to be deactivated.

Listing 4.3: *FCalculateEdgeVoxels*

```
//Global scope variables
FVoxelGraphData* data;
bool * isWorking;

procedure DoWork()
  CalcEdgeVox();
  *isWorking ← false;

procedure CalcEdgeVox()

  foreach i from 0 to data.nrObjects
    parallel_for k from 0 to data.object[i].vox.size
      if(CheckIfOnBorder(data.object[i].vox[k])) then
        data.object[i].vox[k].isActive ← true;
      else_if(IsNeighborhoodNotFull(data.object[i].vox[k]))
        data.object[i].vox[k].isActive ← true;
      else
        data.object[i].vox[k].isActive ← false;
```

#### 4.2.5.2 FSimpleCubesAlgo

It is one of the tasks that focus on mesh computation. It implements the simple cubes algorithm, where general behaviour is explained in 2.8.3. However, implementation is more complex and requires some attention to detail.

A first concept to retain in mesh computation tasks is that all of them should populate a *FPolyObjects* structure, where the mesh spawner can then interpret such structure. The implemented version accelerates the process with parallelization by object, and its pseudo-code implementation is on 4.4. The reason behind parallelization by object and not by voxel is because such an approach would require an intricate locking mechanism that could handle appropriate access to three separate data structures when adding some new plane. Those are the vertex, triangular topology and normal arrays and their correct indexing is essential to produce the desired surface. For this reason, it was selected parallelization that avoids such intricate solution.

Listing 4.4: *FSimpleCubesAlgo*

```
//Global scope variables
FPolyObjects* meshData;
FVoxelGraphData* data;
bool * isWorking;

procedure DoWork()
  SimpleCubesAlgo();
  *isWorking ← false;

procedure SimpleCubesAlgo()
```

```
11  parallel_for i from 0 to data.nrObjects
12    foreach k from 0 to data.object[i].vox.size
13      if(CheckFaceUp(data.object[i].vox[k]))
14        AddFace(data.object[i].vox[k], UP)
15      if(CheckFaceDown(data.object[i].vox[k]))
16        AddFace(data.object[i].vox[k], DOWN)
17      if(CheckFaceLeft(data.object[i].vox[k]))
18        AddFace(data.object[i].vox[k], LEFT)
19      if(CheckFaceRight(data.object[i].vox[k]))
20        AddFace(data.object[i].vox[k], RIGHT)
21      if(CheckFaceFront(data.object[i].vox[k]))
22        AddFace(data.object[i].vox[k], FRONT)
23      if(CheckFaceBack(data.object[i].vox[k]))
24        AddFace(data.object[i].vox[k], BACK)
```

The central behaviour is to loop each filled voxel, as explain, and verify each of its six face neighbours independently. For each nonexistent neighbour, add a triangulated square plane between them. In picture 4.4 is demonstrated the standardization of face direction with the Cartesian axis along with an example of vertex ordering for the triangle topology. Such order must be clockwise in the intended direction, that is outward.

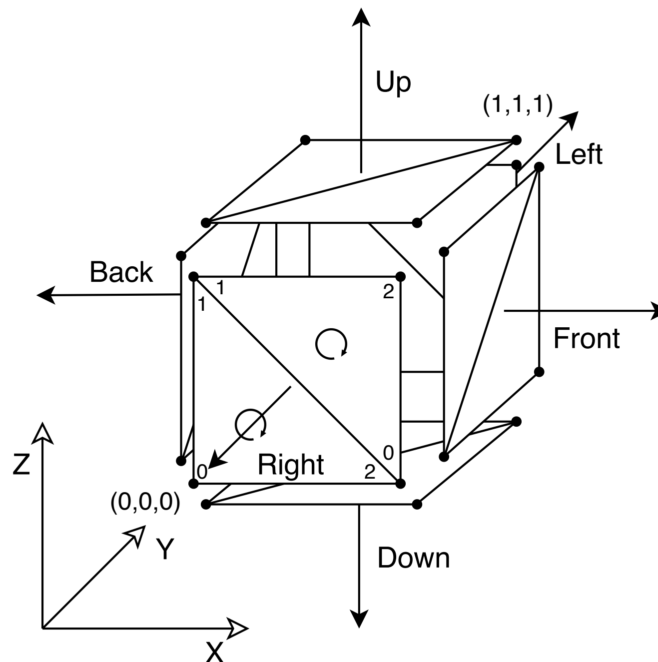



Figure 4.4: Voxel direction standard use in simple cube algorithm implementation.

When some direction is checked, it tests first if it is a border case and if not searches for that particular neighbour. However, it possesses the same limitation as *FCalculateEdgeVoxels* task where to find its neighbour is necessary to perform a binary search due to the nature of our data.

#### 4.2.5.3 FDualMarchingCubesAlgo

To implement the Dual Marching cubes algorithm, an open-source code was used, retrieved from [7]. However, to correctly set up third-party libraries it is required to recompile the code with the followings restrictions:(1)a 64-bit platform, and (2) Multi-thread environment, for them to be smoothly linked. For a more detailed explanation see Unreal Engine 4 official wiki [9].

With the use of the third-party code, the *FDualMarchingCubesAlgo* focuses in converting the used structure to store voxels into the structure used by the Dual Marching cubes algorithm, and to, convert its output into the *FPolyObjects* struct that can be interpreted by SCIRunVR. The algorithm is only given the information of each object and is then possible to parallelize implementation by object.

### 4.2.6 Model

The model component is a central system, where all management of tasks and functionalities occur. The core class is *AModelManager* and it manages a single model that is the data of a single file. On future implementation if one wishes to open more files at the same time, it is just necessary to create more instances of *AModelManager* in the world. To easily spawn the class it extends the AActor class that is designed to easily spawn in the world with some displacement, rotation or scaling.

There are three primary services that the model is in charge of those are data handling, mesh spawning and the management of asynchronous tasks. The first service is trivial where the representation of the original state of data is kept for future operation over it, like selection of another type of mesh. The second service and third, however, requires a more thorough explanation.

#### 4.2.6.1 Mesh Spawning

Mesh spawning is the process of registering the mesh in the world, allowing for it be visible to the Unreal reflection system and consequently can be used for renderization pipeline if itself is on the user field of view. Such functionality must be executed on the system cyclic thread and never asynchronously because it may lead to conflicts with the engine process. For that reason, the mesh spawning is integrated into this component and easily accessed when required.

With data organized by object, an intermediate class, named *AModelObject*, was created. This class was developed to contain all the information about a single object of the model. Currently, the only information that includes is the mesh information, and the material to use. Future developments may use it to store additional details of the object, like centroid, volume, surface area and more.

The mesh currently used by *AModelObject* can be of two types, *ATrigStripMesh* or *ACubesMesh* and it possess the according methods to implement each type with the

its respective data. The *ACubesMesh* is a simpler methodology where it just spawns a static mesh, that is a simple cube, in each voxel position. To do that it uses the *UInstancedStaticMeshComponent* that is a class native to unreal specially created to spawn several instances of the same static mesh. Such a method is extremely inefficient due to the majority of vertexes created not being useful for the visualization because they lay inside the object. For that reason *FCalculateEdgeVoxels* should be used for better performances. Nonetheless, either way, is still a viable way of displaying the data.

The *ATrigStripMesh* is a more efficient method where the direct use of vertexes with their triangular strip topology define just the surface of the object. What enables such approach is the use of the *UProceduralMeshComponent* that is a native class of UE4. The *ATrigStripMesh* contains such class and maps the vertexes on *FPolyObjects* structure into *UProceduralMeshComponent*. Currently, there are two implemented surface extractors of voxel data explained in 4.2.5, that extract vertex information into *FPolyObjects* struct and such policy should be maintained for future implementation of surface extractors.

#### 4.2.6.2 System Flux Control

To correctly manage the model, there are two relevant groups of variables, the settings and task tracking variables. The settings consist of three different enums that are declared in *AModelManager*. Those are:

***EMeshType*:** That discerns between the two possible ways of mesh construction. Those are the cube replication(MT_Cubes) and the direct vertex input with its trig-strip topology(MT_TrigStrip).

***ECubesMode*:** Within the mesh type cube replication, there are two possible options. Represent all voxels(CM_All) or to calculate the edge voxels of each object by cleaning the interior ones(CM_ClearIn).

***EMeshAlgorithms*:** Within the mesh type trig-strip, it is necessary to use some algorithm to extract the surface mesh. There are two implemented surface extractors the simple cubes algorithm(MA_SimpleCubes), and the dual marching cubes algorithm(MA_DMarchCub).

For each of the previous enums, it was created one global scope variable of the respective enum type to register the settings to be used. This variables, plus the *newFilePath*, can be modified by the exposed methods to the blueprint system in *AModelManager*, except the *Compute()* method.

The second group of variables, the task tracking, consist of just two variables, an enum named *nextTask*(NT) and a bool named *isWorking*(IW). The enum possesses seven different entrances to track every single state of the *AModelManager*, those are (1)Idle, (2)LoadFile, (3)LoadFinalize, (4)CalcEdge, (5)CalcMesh, (6)DrawCubes and (7)DrawTrigStrip.

This enum is completely handled on *AModelManager* and is set to the proper state when the method *Compute()* is called from the blueprint system. The *isWorking* flag is set to true by *AModelManager* when lunching an async thread but the responsibility of switching to false lays down with the async task.

The explained variables will allow controlling execution and such control is implemented on the *Tick()* function of the *AModelManager* class that is called at every cycle of the cyclic thread of unreal engine and, usually, representing most of an object life cycle 2.7.5. The objective of this structure is to keep the cyclic thread of unreal engine the minimum time possible processing the *AModelManager* functionalities by lunching threads to perform bulks of computation and allow the engine to resume its cyclic thread so that the entire system can keep its liveness. However, at each *"Tick()"* it is necessary to verify if the thread launched has finished, if so, begin the next task. On image 4.5, is depicted the explained behaviour with all existing states of execution and the flag changing behaviour.

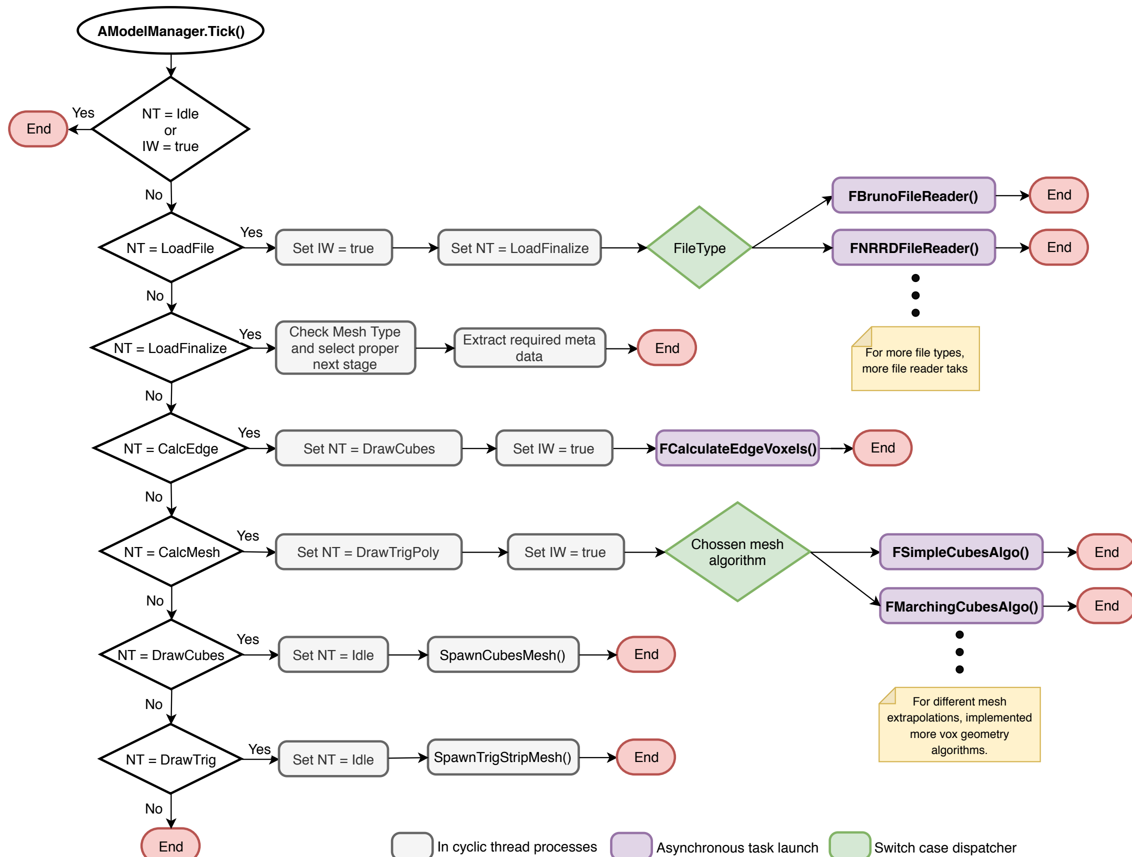


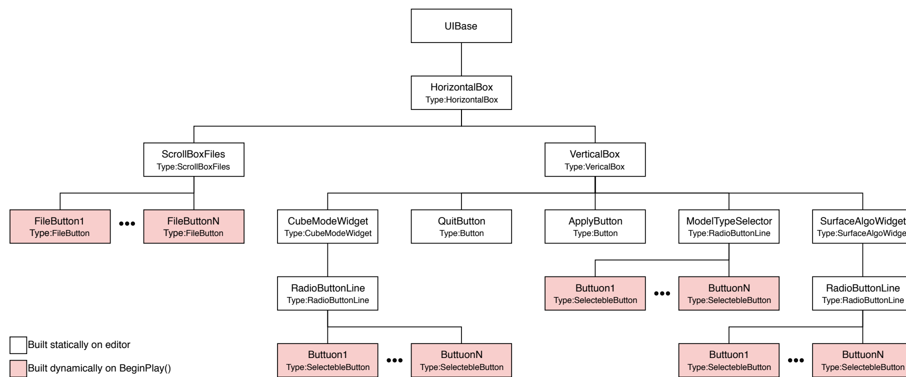

Figure 4.5: Flux diagram of *AModelManager*.

Despite having a system to allow parallelization, three functionalities run on the cyclic thread. Two of them are mandatory and are the ones mentioned in the previous point. The draw cubes and draw trig-strip depicted on the last two cases on 4.5 that may compromise system liveness. The third one is the LoadFinalize, that is a simple process containing some metadata extractions and will not compromise the system liveness.

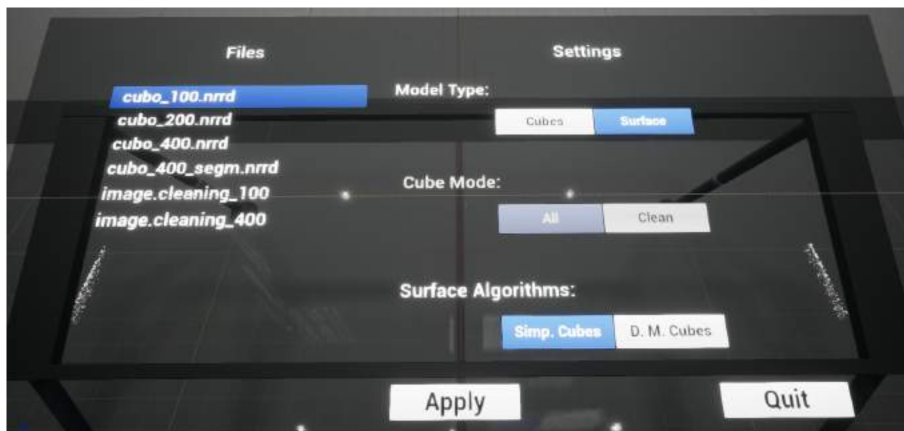
### 4.2.7 Blueprint Control Management

With the blueprint system consisting of a parallel programming environment, it was developed a separate class diagram, depicted on 4.3, to distinguish what is implemented where. This subsection high-level functionality is the only one implemented on the blueprint system and in 4.2.1 is provided with a further sub-division of such functionality comprising of the user-related features and the user interface(UI).

The user features consist of classes that represent the user head and hands. The *MotionControllerPawn* and *BP_MotionController* are the classes that respectively map the user body parts. The former extends the engine native pawn class that primarily allows easy implementation of movement with some input and is comprised of a camera and a hitbox to collide with the environment. The latest extends the actor class consisting of hand mesh and where two instances of it are created to map the right and the left hand of the user.




a ) Overall UI layout and respective hierarchy




b ) Instantiated UI over desk mesh, with dynamic elements created

Figure 4.6: Resulting UI to currently manipulate SCIRunVR prototype.

The used classes were retrieved from the Unreal Engine VR base projects. However, they were slightly changed, and the main modification is to allow interaction with widget objects. To achieved such functionality, it was added a *WidgetInteraction* object as a child

of the hand object. Such an object is a pointer that allows the trigger of click events on widget class objects.

The user interface consists of several widget classes that have their root on the widget class *UIBase*. The *UIDesk* is simply a textured mesh table with a *UIBase* object on top of it, that is then added into the world system.

Due to the lack of radio button like logic in the unreal framework and by being a very suitable way to select the options available, it was implemented such logic on *RadioButtonLine* and *ScrollBoxFiles* classes using the native selectable button of UE4. Both of them generate buttons dynamically on creation according to the number of entries on their names list variable. However, the former possesses such list statically built on UE4 editor and the latest searches for the files using the auxiliary static lib *UComLineInterperter*. Furthermore the *ScrollBoxFiles* haves at its root a vertical scroll panel while the *RadioButtonLine* haves a simple horizontal box.

The *SurfaceAlgoWidget* and the *CubeModeWidget* were created to wrap one radio button line with their respective title to easily grey out the entire sub-widget.

The overall widget hierarchy is visible on image 4.6a, and on image 4.6b is depicted the instantiated *UIBase* widget with all the settings mapped according to the available settings of *AModelManager*. The instantiated widget is also possible to see the grey out sub-menu cube mode, due to the mode type having the surface mode selected. Finally, it is possible to visualize the two statically add buttons with name quit and apply. The quit one exists the SCIRunVR system, and the apply button triggers the execution of the *Compute()* function in *AModelManager* but first it sets the required settings variables that are currently selected on the UI.

### 4.2.8 World Hierarchy Tree

The constructed world for the user to step in was created to possess at is center the developed *AModelManager* that will spawn its mesh dynamically and confine such space with a room that would not be distractive. In image 4.7, a plant view of the room is displayed where the red dot represents the world and *AModelActor* center position. However, the data point of origin is mapped with a blue dot that is located on the bottom left corner of the display zone. Such point will be the root for all *AModelObject* instances.

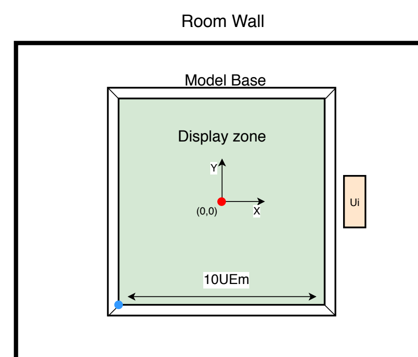



Figure 4.7: Room plant.

As a simple delimited environment, the world only possesses one level tree. The implemented tree structure is depicted on image 4.8. The majority of the tree construction was elaborated on the editor. The great majority of objects is attached to the room that is a blueprint class named BP_DemoRoom, that is one of the many UE4 asset resources. It

extends the actor class, and it spawns a variety of meshes on construction to build a room.

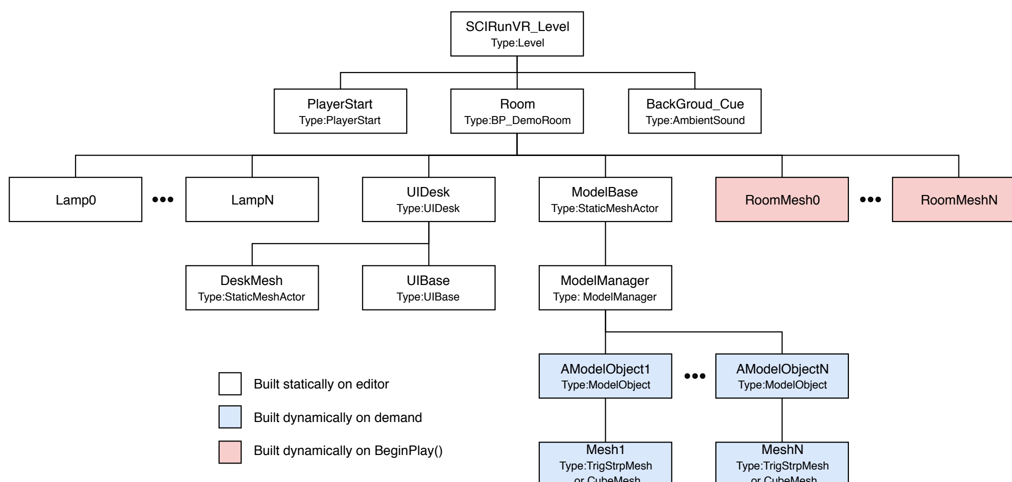



Figure 4.8: Level n-ary tree structure.

To the room, three major groups are attached. The lamps, that consist of two different lamp meshes where each instantiated four times and provide illumination to the room. The *UIDesk*, that is the object that instantiates into the level the UI. And finally the Model group, that first has a trapezoid like mesh that delineates the area where data will be placed, and it is attached to the central object *AModelManager* that possesses all the implemented logic.

The final aspect of world construction is the association with the created blueprint class *SCIRunGameMode* that extends the AGameMode class. The only alteration to such class was to map the desired pawn to the *MotionControllerPawn* that will be spawned attached to the level root and will be "possessed"by the user to control it.

With this world construction, the achieved results can be seen in the next images. First image 4.9, pictures an NRRD file, distinguishable by the use of only one colour due to being only one object, using the surface algorithm dual marching cubes. Second image 4.10 using the same NRRD file but using the simple cubes algorithm. The third and last image 4.11, pictures an exemplary Bruno file type that textures different objects with different colours.

Figure 4.9: Prototype running NRRD file with dual marching cubes surface algorithm.

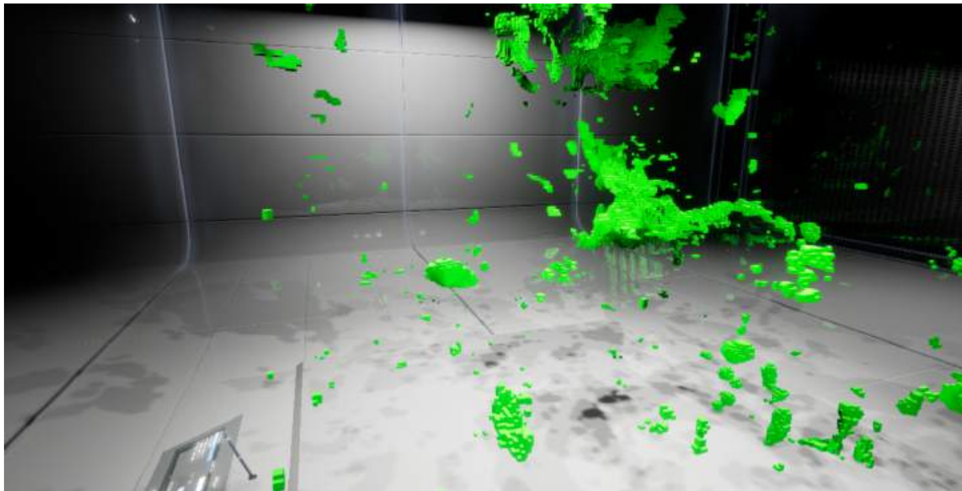


Figure 4.10: Prototype running NRRD file with simple cubes surface algorithm.

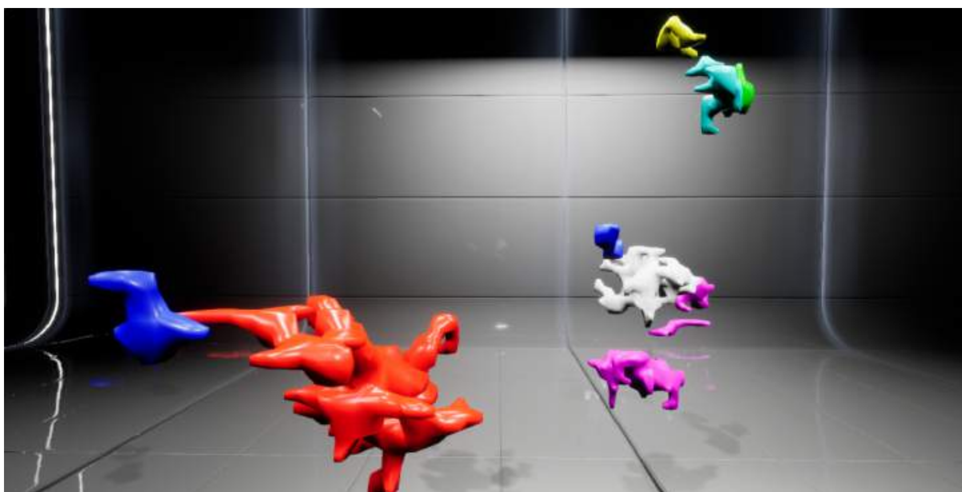


Figure 4.11: Prototype running Bruno file type with dual marching cubes surface algorithm.

# Chapter 5

# Evaluation

The performance evaluation will be branched into three major categories. The first one is dedicated to time execution analyses of all the involved tasks to access the tasks scalability and general waiting time for data to be computed. The second is for discussion of the drawing time results. With the drawing functionality not falling into the classification of task on this project because of not being an asynchronous computation, it requires some special consideration.

The third and final category will provide the validation necessary to the system. This is done by verifying the system liveliness characteristic, by measuring frame rate per second (FPS). By measuring it will allow knowing the viability of the solution and, if viable, what is the maximum amount of data possible. To access such viability, one first rule must be established that is the minimum of accepted FPS that will grant the quality of "alive". Such bottom line will be established on the 30 FPS or the equivalent 30ms between frames. Such is the number that is required for the image generated not appear delayed for the user.

## 5.1 Test Environment

To describe the full test environment first and foremost it is necessary to describe the hardware involved. The tests were executed on a desktop with the following characteristics:

**Motherboard:** MPG Z390 Gaming Plus (MS-7B51)

**CPU:** Intel Core(TM) i5-9600 CPU 3.7GHz, 6 cores, 6 logical cores.

**Graphic card:** NVIDIA GeForce RTX 2070

**StaticLibs:** 16Gb, 2666MHz

To complete hardware specs, the used VR gear its the HTC Vive. It consists of a head-mounted stereoscopic display and two hand controllers. Such gear possesses a crucial technicality that must be addressed. Such technicality is the FPS locking mechanism that it contains that, depending on the system liveness, it will lock the system FPS to 90, 45 or 30. Only if less then 30 will decrease frame rate gradually until the 10 FPS are reached. When 10 FPS are reached it blocks VR renderization.

Another description required is what data was tested. The files tested comprise of three distinct samples of 100, 200 and 400 voxel grid size. Such samples are initially stored in three NRRD files and were transformed into Bruno files on the TOMO-GPU project with a segmentation value of 90. For that reason, the SCIRunVR will use an equal value on segmentation on the NRRD files. This will allow, for each respective pair of files of the same size, to posses similar geometry, however, is not completely equal due to cleaning functionality on the Tomo-GPU system.

To finalize environment test characterization, one last description of how it was made is required. First, all graphic settings were placed on the level tree. Second, the execution of tests was realized in the editor environment and not on the stand-alone version. This means that all executions possess some overhead due to the spending of computation resources on the editor. Nonetheless, it is possible to consider such characteristic the worst-case scenario and consequently derive the maximum size of data for the worst-case scenario.

## 5.2 Tasks Execution

Such measures must be undertaken with different file sizes, for obvious reasons, and with different file types where the various parallelization approaches will affect each type of file differently. Such observation doesn't possess the focus of validation but instead to check waiting times for data to be processed and judge if it was correct to employ asynchronous thread processing.

All the measurements of time in this section are established by averaging the time spent over three executions where it should be enough due not being expected much variance.

### 5.2.1 Load

It is possible to depict load time behaviour on the graphic in 5.1. With time mapped on the logarithmic scale, it is possible to verify that that time scales exponentially. However, Bruno file types scale better than NRRD files. With very similar palatalization politics, the cause of such difference lies in the file structure where Bruno file types, that store just the filled voxels, are much smaller, allowing for greater scalability. With the graphic is also possible to conclude that it was a good design decision to establish the loading of files

as an asynchronous task by observing that such files will easily pass the 30ms execution time.

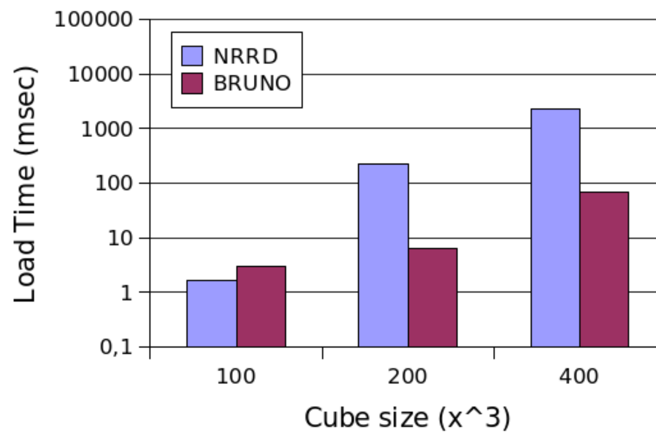



Figure 5.1: Execution times of loading tasks with different data size of both supported files.

### 5.2.2 Calculate Edges

The edge calculation costs depicted on image 5.2 is also possible to retrieve the exponential costs with the file size and where Bruno file type execution times scale better then NRRD files. It is also possible to verify the good design decision of parallelizing such task where it may easily break liveness of the system when passing the 30ms execution time.

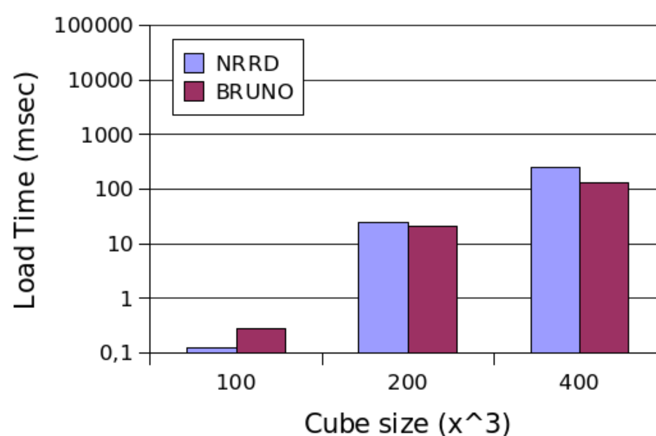



Figure 5.2: Execution time of edge calculation with different data size of both supported files.

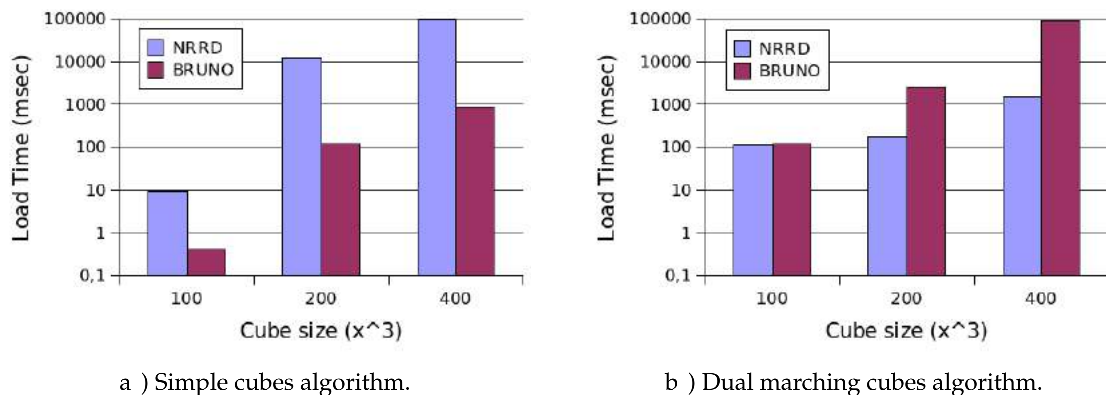
### 5.2.3 Surface Extraction

In this subsection is aggregated the two surface extractors implemented. Those are the simple cubes algorithm depicted on image 5.3a and the dual marching cubes depicted on 5.3.

Within each one, it is possible to derive their exponential behaviour, keeping into consideration that time is on a logarithmic scale. As the most time-consuming tasks, the async implementation was of paramount importance avoiding large time gaps where the image would "freeze".

Generally speaking, the simple cubes algorithm possesses higher execution times in spite of being a simpler method. This is due to the lack of a matrix-like data struct where it would be possible to directly access some neighbour, instead of searching for it. Such an approach would lead to much better efficiency and possibly allow to outperform the dual marching cubes algorithm. Nonetheless, execution times on Bruno files are quite acceptable due to two aspects. First, the parallelization approach over each object and second because by dividing the model into objects reduces the search space for the neighbour, reducing the time it takes to find it.

The dual marching cubes have an extremely well performance over NRRD files because they contain just one object allowing for all information to be passed just once and computed once leading to such results.




a ) Simple cubes algorithm.

b ) Dual marching cubes algorithm.

Figure 5.3: Surface extraction time execution.

## 5.3 Draw Execution

The mesh drawing functionality is one that could not be implemented asynchronously because it could lead to conflicts with the engine itself. As is possible to determine in 5.4, some of the executions exceed the 30ms line hindering the system liveness.

The two axis on the image is mapped on a logarithmic scale, and the lines represent a linear regression of the execution times of each mesh. With them is easily analysed that trig-strip topology drawing is more costly for the same amount of vertexes. Nonetheless, it is necessary to remember that to draw the same data with both approaches, results in considerably more vertexes on the cubes approach. This means that the same model is not mapped to the same vertex count, consequently not aligned vertically.

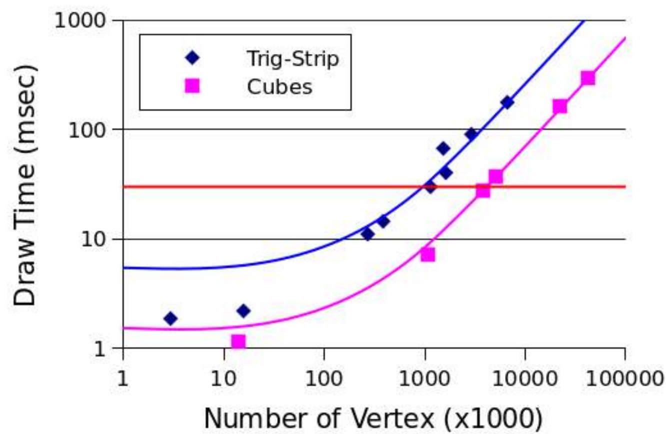



Figure 5.4: Execution times of mesh drawing by the number o displayed vertexes for both supported mesh techniques.

## 5.4 Liveness Test

The liveness test will is the one that will provide algorithmic validation over the system. In real-time renderization engines, the usual measure of liveness is the frame rate per second or FPS, that counts the number of images produced by the cyclic thread of the engine. As defined initially in this chapter, the bottom line for FPS count will be established on 30FPS however, was not established how it would be retrieved. Such a method consists of averaging the FPS count over 30 seconds after all computations have finished.

The FPS rate of the system depends on the number of vertexes that are accounted and the type of meshed used, when not dealing whit any other logic implemented by the programmer. For this reason, the mapping of data will be done by the number of vertexes for each of the mesh types independent of what algorithm or steps were used to archive the vertex count.

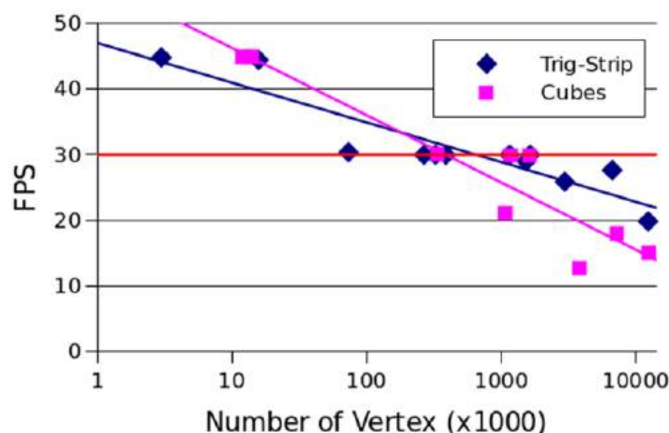



Figure 5.5: FPS average displaying with both types of mesh depending on vertex count.

The results are depicted in 5.5 where it is possible to discern the FPS lock that the VR gear makes on the 45 and 30 FPS and after it decreases steadily. Other important information is that it wasn't possible to test all the Cubes mesh possibilities due to frame

rate dropping bellow 10FPS on the 200 and 400 file sizes where the interior wasn't removed.

The remaining conclusions are: that this data fits best a logistic regression, that is represented by a line due to the logarithmic scaling on the x axis. And that the trig-strip topology is much better to maintain the FPS count high, allowing a vertex count near the two million vertexes.

# Chapter 6

# Conclusion

The two primary objectives of this work being the construction of a prototype system that handles Tomo-GPU data and full integration in Tomo-GPU system were completed. Following the validation structure accessed on *ssec:VisValFram*, the system validation was only on the algorithmic level. However, all upstream validations were discussed in this work and should influence positively in further down validations.

With results showing the algorithmic viability of SCIRunVR system it has all the potential to aid material scientist in their studies of composite materials in the particular case of studying the effects of object geometry on the composites developed. Furthermore, with implementation of more functionalities, like individual objects possessing further drill down information, it could lead to even more benefits on knowledge extracted then just geometry understanding.

## 6.1 Future Work

With such novice system using a powerful new interface tool the possibilities are huge. It is possible to divide future work in to distinct sections those are the implementation of new features or the further down validation levels on the framework in 2.3.

The implementation of new features varies greatly, from the insertion new surface extraction algorithms, to the supported of more types of files and their respective geometry. The further down validations comprise mainly of user test cases. There are three possible user studies those are: (1)Visual encoding idiom layer study that targets the performance in shape perception with SCIRunVR against the normal used environment on SCIRun.(2) The interaction idiom study that targets performance on both user control and user interface with both systems against each other again. (3)Final field study that targets performance with real users.

# Bibliography

[1] *Blueprints Quick Start Guide | Unreal Engine Documentation*. URL: https://docs.unrealengine.com/en-US/Engine/Blueprints/QuickStart/index.html (visited on 09/18/2019).

[2] T. Cadavez, S. C. Ferreira, P. Medeiros, P. J. Quaresma, L. A. Rocha, A. Velhinho, and G. Vignoles. "A graphical tool for the tomographic characterization of microstructural features on metal matrix composites." In: *International Journal of Tomography and Statistics* 14.S10 (2010), pp. 3–15. ISSN: 09729976. URL: https://www.researchgate.net/publication/282058411.

[3] H. Delgado. "Characterization and Surface Reconstruction of Objects in Tomographic Images of Composite Materials." In: (2013). URL: https://run.unl.pt/handle/10362/13141.

[4] C. Donalek, S. G. Djorgovski, A. Cioc, A. Wang, J. Zhang, E. Lawler, S. Yeh, A. Mahabal, M. Graham, A. Drake, S. Davidoff, J. S. Norris, and G. Longo. "Immersive and Collaborative Data Visualization Using Virtual Reality Platforms." In: *IEEE International Conference on Big Data* (2014), pp. 609–614. ISSN: 9781479956661. DOI: 10.1109/BigData.2014.7004282. arXiv: 1410.7670.

[5] *Examples of Good and Bad Visualization – kaijiezhou*. URL: https://kaijiezhou.wordpress.com/2016/01/27/examples-of-good-and-bad-visualization/ (visited on 02/18/2018).

[6] E. Gallopoulos, E. Houstis, and J. R. Rice. "Computer as Thinker/Doer: Problem-Solving Environments for Computational Science." In: *IEEE Computational Science and Engineering* 1.2 (1994), pp. 11–23. ISSN: 1070-9924. DOI: 10.1109/99.326669. URL: http://ieeexplore.ieee.org/document/326669/.

[7] *GitHub - dominikwodniok/dualmc: Simple C++ implementation of the (manifold) dual marching cubes algorithm from Gregory M. Nielson*. URL: https://github.com/dominikwodniok/dualmc (visited on 12/04/2019).

[8] J. Lanier, F. Biocca, and N. Carolina. "An Insider ' s View of the Future of Virtual Reality." In: 42.4 (1992).

[9] *Linking Static Libraries Using The Build System - Epic Wiki*. URL: https://wiki.unrealengine.com/Linking{_}Static{_}Libraries{_}Using{_}The{_}Build{_}System (visited on 12/04/2019).

[10] T. Munzner. "A nested model for visualization design and validation." In: *IEEE Transactions on Visualization and Computer Graphics*. Vol. 15. 6. 2009, pp. 921–928. DOI: 10.1109/TVCG.2009.111. URL: http://ieeexplore.ieee.org/document/5290695/.

[11] T. Munzner. *Visualization Analysis and Design*. 2014, p. 428. ISBN: 9781466508934. URL: https://books.google.de/books?id=dznSBQAAQBAJ.

[12] S. Parker, M. Miller, C. Hansen, and C. Johnson. "An integrated problem solving environment: the SCIRun computational steering system." In: *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*. Vol. 7. IEEE Comput. Soc, pp. 147–156. ISBN: 0-8186-8255-8. DOI: 10.1109/HICSS.1998.649208. URL: http://ieeexplore.ieee.org/document/649208/.

[13] *Particle System User Guide | Unreal Engine Documentation*. URL: https://docs.unrealengine.com/en-US/Engine/LevelStreaming/WorldBrowser/index.htmlhttps://docs.unrealengine.com/en-US/Engine/Rendering/ParticleSystems/UserGuide/index.html (visited on 09/18/2019).

[14] P. J. Quaresma and C.-f. Unl. "Tomo-GPU : Um Ambiente de Resolução de Problemas Destinado à Análise de Dados Tomográficos Relativos à Caracterização Estrutural de Materiais." In: *Revista de Ciências da Computação* (2010), pp. 39–54. ISSN: 1646-6330. URL: https://repositorioaberto.uab.pt/handle/10400.2/1891.

[15] R. E. Roth. "Cartographic Interaction Primitives: Framework and Synthesis." In: *The Cartographic Journal* 49.4 (2012), pp. 376–395. ISSN: 0008-7041. DOI: 10.1179/1743277412Y.0000000019. URL: http://www.tandfonline.com/doi/full/10.1179/1743277412Y.0000000019.

[16] M.-l. Ryan. *Narrative as Virtual Reality*. 2001, pp. 1–5. ISBN: 0-8018-6487-9. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3.

[17] S. Schaefer and J. Warren. "Dual marching cubes: Primal contouring of dual grids." In: *Proceedings - Pacific Conference on Computer Graphics and Applications*. 2004, pp. 70–76. ISBN: 0769522343. DOI: 10.1109/PCCGA.2004.1348336.

[18] W. J. Schroeder and K. M. Martin. "The visualization toolkit." In: *Visualization Handbook*. Prentice-Hall, Inc., 2005, pp. 593–614. ISBN: 9780123875822. DOI: 10.1016/B978-012387582-2/50032-0. URL: https://dl.acm.org/citation.cfm?id=272980.

[19] *SCIRun 5 : Build*. URL: http://sciinstitute.github.io/scirun.pages/build.html (visited on 09/18/2019).

[20] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010.

[21] Sigarch., IEEE Computer Society. Technical Committee on Supercomputing Applications., and IEEE Computer Society. Technical Committee on Computer Architecture. *Proceedings of the 1995 ACM/IEEE Supercomputing Conference ; Supercomputing '95 : San Diego, California, USA December 3 through 8, 1995.* Association for Computing Machinery, 1995. ISBN: 0897918169. URL: https://ieeexplore.ieee.org/abstract/document/1383188https://ieeexplore.ieee.org/document/1383164.

[22] A. Šmíd. "Comparison of Unity and Unreal Engine." Doctoral dissertation. 2017, p. 69. URL: https://core.ac.uk/download/pdf/84832291.pdfhttp://dcgi.felk.cvut.cz/projects/pacman-benchmark/thesis-compressed.pdf.

[23] M. St. John, M. B. Cowen, H. S. Smallman, and H. M. Oonk. "The Use of 2D and 3D Displays for Shape-Understanding versus Relative-Position Tasks." In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 43.1 (2001), pp. 79–98. ISSN: 0018-7208. DOI: 10.1518/001872001775992534. URL: http://journals.sagepub.com/doi/10.1518/001872001775992534.

[24] *The Ames Room: Engineering a Human Giant or an Optical Illusion? - CADENAS PARTsolutionsCADENAS PARTsolutions*. URL: http://partsolutions.com/the-ames-room-engineering-a-human-giant-or-an-optical-illusion/ (visited on 02/18/2018).

[25] *Tomo-GPU*. URL: http://asc.di.fct.unl.pt/{~}pm/Tomo-GPU/ (visited on 09/18/2019).

[26] *Vintage Ames Room Illusion | An Optical Illusion*. URL: http://www.anopticalillusion.com/2012/07/vintage-ames-room-illusion/ (visited on 02/18/2018).

[27] J. S. Yi, Y. ah Kang, J. T. Stasko, and J. A. Jacko. "Toward a deeper understanding of the role of interaction in information visualization." In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (2007), pp. 1224–1231. ISSN: 10772626. DOI: 10.1109/TVCG.2007.70515. URL: http://ieeexplore.ieee.org/document/4376144/.