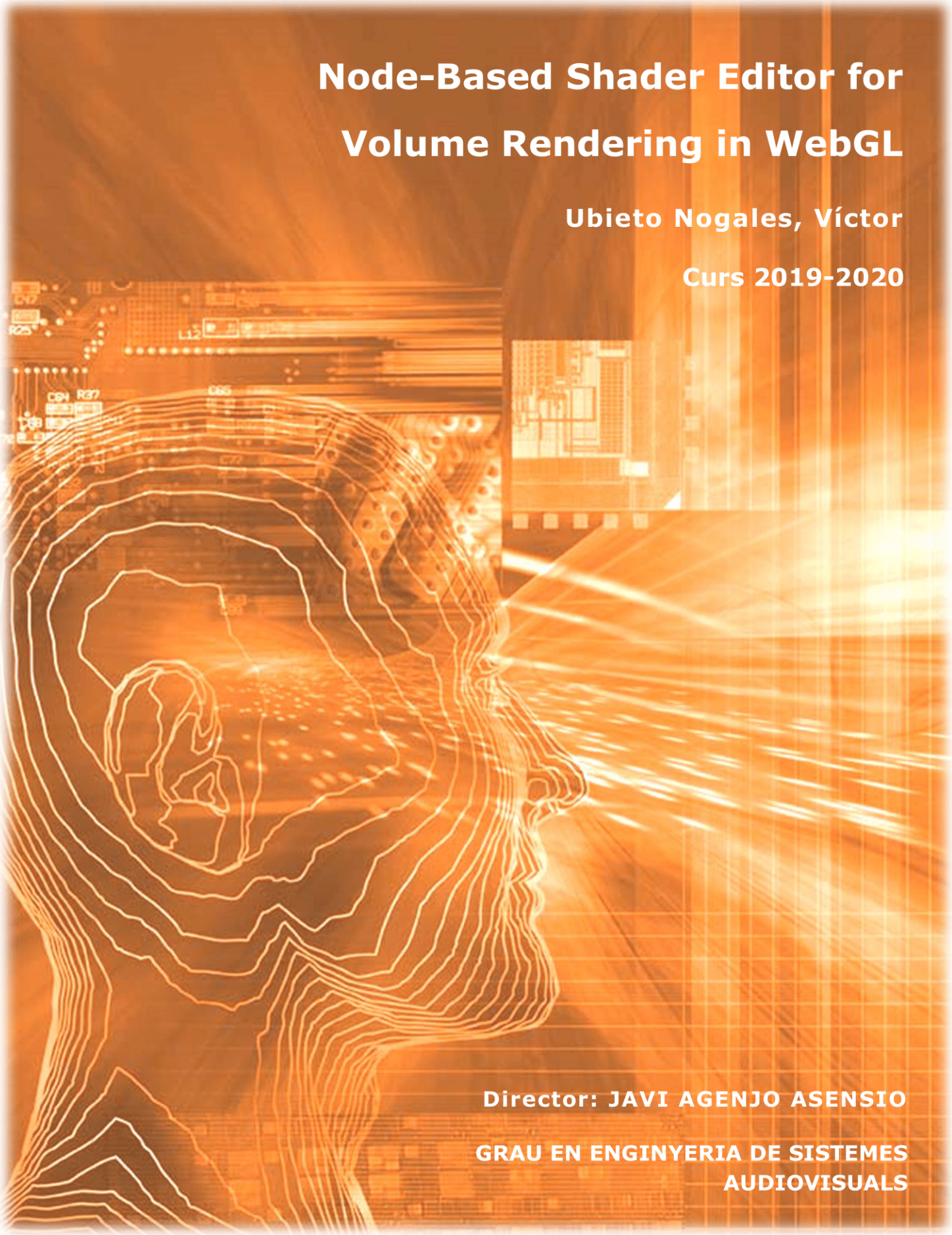
Node-Based Shader Editor for Volume Rendering in WebGL

Ubieto Nogales, Víctor

Curs 2019-2020



Director: JAVI AGENJO ASENSIO

GRAU EN ENGINYERIA DE SISTEMES AUDIOVISUALS

upf. Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Node-Based Shader Editor for Volume Rendering in WebGL

Víctor Ubieto Nogales

TREBALL FI DE GRAU

GRAU EN ENGINYERIA DE SISTEMES AUDIOVISUALS

ESCOLA SUPERIOR POLITÈCNICA UPF

JUNY 2020

DIRECTOR DEL TREBALL

JAVI AGENJO ASENSIO

INTERACTIVE TECHNOLOGIES GROUP (UPF-GTI)

To my family,

whose support and love makes me keep going forward.

Acknowledgements

I would like to express my deep gratitude to my mentor Javi Agenjo for this overall guidance throughout the project, spending a long time helping me when I needed it. And also to my supervisor Josep Blat for being always so proactive at advising me.

Thanks to Miquel Floriach and all the member of the GTI, who always found time to answer my questions.

And finally, thanks to my family who have been supporting me in my actions and my decisions academically and personally.

Abstract

Visual editors have made a name for themselves in almost every work field, and more specifically in Computer Graphics, where they are very popular due to the facilities they provide and to the high accessibility that gives to the user to manipulate programming algorithms without having the necessary knowledge to carry out their implementation.

Among all of them are also the algorithms belonging to the term *Volume Rendering*, which compute the visualization for volumetric materials or data. Its algorithms evaluate the received color of the objects taking into account their inside, which is used in medicine for visualizing the internal parts of the body (that would not be visible with normal rendering techniques), and for rendering special elements like clouds, or smoke (which are not constrained by a concrete physical shape).

In this report, the project consisting in the development of a web shader editor for volumetric will be explained. The editor addresses the already mentioned application cases, which are also discussed.

Keywords: Volume Rendering, Dicom, Visual Shader Editor.

Resum

Els editors visuals han trobat un lloc en una gran quantitat d'àmbits, especialment en els Gràfics per Ordinador, on tenen una gran popularitat gràcies a la facilitat d'ús i a l'alta accessibilitat que dóna als usuaris per manipular algorismes de programació sense tenir els coneixements necessaris per realitzar la seva implementació.

Entre tots ells, també s'inclouen els algorismes que pertanyen al terme de *Volume Rendering*, els quals permeten la visualització de materials i conjunts de data tridimensionals. Aquest algoritme avalua el color de l'objecte tenint en compte la seva estructura interior, el qual és utilitzat a la medicina per visualitzar parts internes del cos humà com els óssos o els òrgans (els quals no seríem capaços de veure utilitzant tècniques de visualització con-

vencionals), i també per representar elements sense cap forma concreta com els núvols o el fum.

En aquest informe, s'explica el desenvolupament d'un editor de *shaders* web per volumetria. L'editor aborda els casos d'ús ja esmentats, que també seran discutits en aquest escrit.

Preface or Prologue

The reasons behind the realization of this project come from my interest and curiosity for the world of Computer Graphics. It was my reason for choosing this degree and the subjects that were related to this topic were my motivation to keep working each term. For this reason, I was completely sure that my final project had to be related to this field.

Luckily, the UPF has several research groups, and one of them is specialized in this subject. I got carried away for my curiosity and accepted a proposal idea of working on a web editor, which also led me to investigate more about the subject and introduced me to some applications like Blender.

I knew it was going to be difficult since I had no previous experience with most of the topics involved in the project, but the good atmosphere in the working group made me want to try to undertake it successfully.

Index or Summary

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Context

Most of the time surfaces are enough for representing objects in Computer Graphics (CG). This happens because the inside of an object is not normally seen, and considering the internal information would add extra difficulties to the way we compute the final color of the object. But there are some cases that require the visualization of the interior of the volumes, and therefore a specific algorithm appropriate to achieve it is needed. The subject dealing with this visualisation of volumes in CG, is known as *Volume Rendering*. Let us remark that rendering is the CG process of computing an image from a model.

Some specific examples of applications where this technology is required are intangible and non-opaque elements like gas or steam, and also the medical visualizations of a human body, rendering information stored in datasets obtained in most cases through computer tomographies (CTs) scans. As a consequence, there exist quite a number of programs and engines created specifically for people who require this type of algorithms to solve their respective application cases. For example, the Philips app "IntelliSpace Portal", which helps doctors to provide medical diagnoses, is used in many hospitals like "Hospital Universitario de Tarragona Juan XXIII" and "Hospital del Mar de Barcelona".

1.2 Problem Definition

We can identify broadly two types of agents from the scenario presented in the previous context, namely, the developers (the people who create the programs) and the users (the people who use the computer applications).

The proper functioning of the workflow is achieved thanks to accumulate the contribution of the individual work of each part. Nevertheless, all their effort would not create the desired results if they were not working towards a common goal, and this can only be obtained if both groups interact with each other. As can be guessed, the principal difference between the two parties is its respective specific training and the knowledge they possess. On one side, the creators/developers know the programming languages and therefore are able to program the computer applications. And on the other side, the clients/users have a deep knowledge of the application use cases field and consequently are able to make use of the tools created by the programmers and give useful results. For that reason, the gap of knowledge between the developers and the users must be a primary aspect to focus on and find solutions for an efficient and productive workflow.

The best-known interaction method are forums. They are usually websites, which represent bridges where the community interacts with the programmers and other users reporting bugs, suggesting changes and new implementations and helping novel users to learn and solve problems more easily. This latter idea is not just speculation, but facts because "The best and most continuously rewarding developments in modern IT seem to draw on community-based efforts." [28]. It is also important to emphasize that the key characteristic that defines a good application is not how well it is programmed, but how good the results the users create are. This may sound tricky, but it actually shows how necessary it is that users have control over the applications, so they can adjust them to fit better for their use case.

However, the lack of knowledge in programming generates a problem to achieve this objective. Moreover, the requirements of a user are different from those of the other ones, so that there cannot be a standard program that satisfies all the necessities of everyone. As a summary, it could be said that the ideal solution for these problems would be a program

that was created by the developers to be modified by the users as they desire (taking into account that they may have no previous programming experience or knowledge).

Luckily, these programs exist and are called Visual Programming Languages (VPLs), which mainly consist of visual editors, where the user is able to manipulate graphic elements (as nodes) by using input devices such as the mouse or the keyboard. But, on the other hand, due to the presented reasons, they are a challenging application to program for the developers.

1.3 Project Objectives

By combining the information of the previous sections, that talked about the context of this project and the detected problem, we can identify the benefits of the VPLs when working with Volume Rendering. They can be also denominated as Visual Shader Editors (VSE), as they are known, when they specify its usage and application for shading (affecting mainly the GPU). Shading, in this context, is synonymous of rendering. Besides that, this project will be implemented for the web, because of the availability of concrete web-based libraries that will be used, and due to the accessibility that this approach gives to the users irrespective of operating systems and devices.

Thus, more specifically, the objective of this project is to create a web application that allows visualizations of volumetric data. The visualization will be defined by the resulting shader formed by the graph system, whose manipulation will be possible for the user in order to cover the different application cases discussed. One of them includes the capability of dealing with volume datasets (Dicoms). So, without writing a single code line, the user will create a functional shader that uses volume rendering techniques.

Chapter 2

STATE OF THE ART

In the second chapter, we will see the background for this project, consisting of the volumetric data and its use cases, the Visual Programming Languages (VPLs), and consequently the Visual Shader Editors (VSE), and finally the web applications. We will accompany the explanation with the references used and with models or cases that inspired the realization of the project and that were taken as examples.

2.1 Volumetric Data

In order to define correctly the volumetric data, I will divide the overview into three aspects: what is the data, how it is conceived, and the techniques to acquire it.

2.1.1 Data Characterization: Domain and Data Dimensionality

In the same way that visualization techniques can be organized depending on the characteristics of the data generated from the computations, scalar data can also be categorized according to the domain dimensionality. It is commonly ranged between 1 to 3 dimensions but it can go further, obviously increasing the difficulty of the rendering of the data, and therefore its computational cost.

Scalars, represent individual values but are usually samples from a continuous function, such as temperature along continuous time. In this example, the dependence of temper-

ature with respect to time, which is 1D, allows us to express scalar values as a function of an independent variable $y = f(x)$, or, in this case $T = f(t)$. When the dimension of the domain increases, we use the common functions of several variables, such as $y = f(x_0, x_1, ..., x_n)$, in the case that the domain has n dimensions. Scalar data are also called scalar fields because data are values defined for the physical space. In this project, we will specifically focus on the three-dimensional scalar fields, which we will notate as E_S^3 (Brodlie, 1992) [26]. There are several ways of visualizing E_S^3, the most important techniques are the isosurfaces and volume rendering.Unlike scalar fields, vector fields provide a vector for each point in the 3D space.

2.1.2 Discretization of Volume Datasets

As explained, volumetric data can be understood as a three-dimensional field, where for each spatial point we have a scalar value. Computations cannot deal with continuous representations, and instead of continuous spatial points, we use the so called *voxels*, which constitute the minimum processable unit of a three-dimensional array, and are the equivalent of the pixels in 2D. In volumetric contexts, voxels are also used to describe the resolution (as pixels in two dimensions).

Thus, instead of points represented by position or 3D coordinates in the space, the voxels will be usually represented by 3 integers which denote horizontal, vertical and depth position with respect to an origin. In more traditional computer graphics, only object surfaces are rendered and these surfaces are discretized by means of polygons. Rendering can be performed in most cases without considering all the interior space delimited by the surface. Instead, in the case of volume datasets, we need all this interior space to be regularly sampled in order to make use of it, and obviously this will imply an increase of complexity and also an increase of the computational cost of the rendering techniques used.

When working with E_S^3 we can consider that the data is available all over the space, or that it only exists in one specified region. We need to specify this difference, because when using closed datasets, a usual object with surface rendering as in more traditional CG, it

is easy to delimit the area that the object is occupying, and therefore it is easy to work with. On the other hand, in E_S^3 we need to consider all the values in a concrete region in order to compute the values of the visualization, the values in these delimited regions are called volume datasets.

2.1.3 Acquisition of Volume Datasets

There exist mainly two methods of acquiring a volume dataset, physically and virtually. In the physical approach, data are obtained from physically based processes (X-rays and others), usually by comparing the difference between the information sent and received after going through the object. Currently, the 3D information is mostly composed of 2D slices, which are called (computational) tomographies, in a way analogous to the physical ones. Nowadays, there exist two main methods for obtaining tomograms (images resulting from a tomography), Computational Tomography (CT) and Magnetic Resonance Imaging (MRI). There exist other techniques, like Ultrasounds are used to obtain ecographies, and PETs are based on emitting positrons. Additionally, there also exists variants of the mentioned techniques, like the micro-CT scanners.

On the other hand, the virtual methods are usually based on a series of mathematical calculations that define the behavior of the region. The calculations usually intend to simulate a physical behaviour. As a result, it gives a lot of freedom to the user to control the behavior and the internal characteristics of the volume, as an artificial, rather than a natural, behaviour is sometimes intended, with a creative intention. This is the basic explanation behind a vast diversity of algorithms such as Noise [17] and many others whose implementation will be seen later on in this document.

The principal computational difference between both approaches is that the first method assumes that the dataset is stored, therefore you have to handle how and where to load in order to use it. In the other method, the value at each point of the field must be defined directly in the computation process, creating an increase in the cost. If we go further on discussing the uses of the datasets, we can say that the method used to get the data defines the purpose behind its use. We can express these two cases with one word each, study and

create (this is more of a metaphorical approach, I do not want to give much importance to the terms used). Our intention can be the study and manipulation of the data read, whose more common use is within medicine. Or our intention can be the creation of personalized behaviors that can be difficult to replicate in real-world but not virtually, whose translation is the shading art.

2.1.4 Rendering Techniques

As introduced in subsection 2.1.1, there exist several techniques for volume datasets visualization and are divided into two main categories: the direct and the indirect visualization methods.

- Volumetric data is first converted into a set of polygonal isosurfaces, this process is called isosurface extraction. There are many algorithms for creating such a model (Lorensen, 1987; Wilhelms, 1992; O'Rourke, 1994; Chiang, 1997). Then, it is rendered with traditional surface rendering techniques, in consequence, it is easier to display. This is referred to as indirect volume rendering (IVR).
- Volumetric data is directly rendered without the intermediate conversion step. This is referred to as direct volume rendering (DVR). There are four most popular techniques in this area: Ray casting (Hall, 1991), Splatting (Westover, 1990; Mueller, 1999), Shear-warp (Drebin, 1988; Lacroute, 1994) and Texture slicing (also called 3D texture-mapping) (Cabral, 1994). However, in this project we will only explain two of them, ray casting and texture slicing, discussed next. Even though these methods were created a long time ago, the independent work of the researchers in this field have refined them, making them reach a high level of maturity.

Ray Casting

Ray casting is the most common technique in the community, this is due to the physical bases behind the rendering equation. It works by setting the camera in front of the volume and sending a ray for each pixel of the frame where the render will be stored throughout the volume. The rays travel along the volume step by step with a ray marching algorithm,

at each step, it is computed the accumulated color and opacity using a light transport model.

Finally, the iterative algorithm ends either when it reaches the end of cycles defined, when the opacity is equal 1 or more (that will mean that the continuation of the ray can not be seen), or when it leaves the volume.

Because the complete equation of light transport is very complex, simplified models are usually used. These models are obtained by simplifying or bypassing the types of interaction of the light (emission, absorption, and scattering), the level of fidelity of the model selected by the programmer will depend on the relation cost-quality required in the application. The most common among all of them is the emission-absorption model [13].

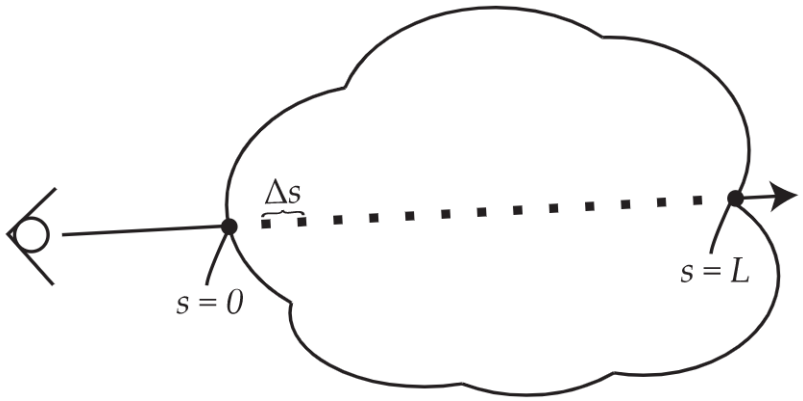


Figure 2.1: Ray Marching on a volume

Texture Slicing

Texture Slicing technique is divided into two parts. First, the volume data is replaced by a set of parallel planes composed using the same dataset and integrated along the direction of view. This resampling algorithm ideally is implemented using 3D texture maps, but it also works using 2D texture maps if 3D textures are not supported by the OpenGL implementation. The principal difference is that slices can be oriented perpendicular to the camera using 3D textures because they can take a spherical form. On the other hand, using 2D textures data-slice polygons can't always be perpendicular to the view direction, but they can only be oriented perpendicular to the major axis X, Y and Z. For that reason, this approximation creates worse results if the data slices get near the 45 degrees from the view direction.

Since this project will be implemented in WebGL 2.0, which supports 3D textures, there is no need to use this technique. But I wanted to mention it because it will be useful within the scope of the future work that we will see at the end of the report.

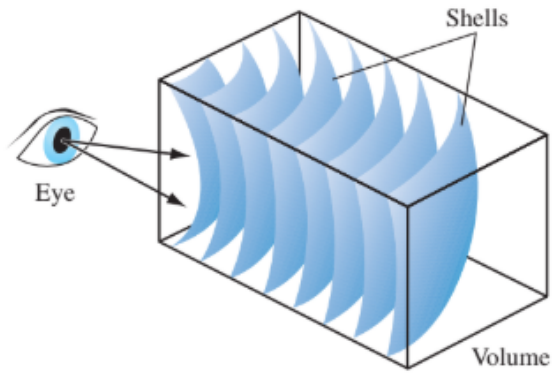


Figure 2.2: Texture Slicing technique

2.2 Use Cases

In the previous section, we mentioned two different approaches to obtain a volume dataset, the principal difference between them is the reason behind the data acquisition. Consequently, each independent case has its own workflow and behavior.

2.2.1 Medical Case

Medical volume data is acquired for different purposes, such as diagnosis, therapy planning, surgery planning, biomedical research, etc. Nevertheless, in specific situations, doctors demand a volumetric visualization of the injured/diseased part. There exist different specialties that benefit from this type of visualizations; depending on the part of the body to be studied, the data acquisition technique will change.

The working process taken varies at each Hospital. For this project, we use the information obtained from the experience of doctors from “Hospital Juan XXIII de Tarragona”, “Centro de la Imagen de Tarragona” and “Hospital del Mar de Barcelona”. Moreover, we had direct explanations and advice of Dr. Tomás Luis Sempere Durá, writer of the “Atlas Interactivo de Anatomía Radiológica” [11] among other important books. His book is

complemented with a mobile app that "allows to navy for all the different images acquired with different techniques that let the visualization of organs" [12].

The parts of the body are classified by the radiological density, bones, soft parts, and air, they are measured in Hounsfield units (HU). More precisely, the low-density parts are normally obtained using MRI (only if the CT is not good enough) and the rest is obtained by CT. One of the main problems of modern visualizations is the difficulty of discrimination of different parts of the body that have similar density values (most of the time is solved by cutting out the structure manually).

Tissue Type	**Hounsfield Value Interval**
Air	-1000
Lung tissue	-900 to -170
Fat tissue	-220 to -30
Water (H_2O)	0
Pancreas	10 to 40
Liver	20 to 60
Heart	20 to 50
Kidney	30 to 50
Bones	45 to 3000

Table 2.1: Hounsfield values for selected tissue types from [29]

Computational tomography consists of an acquisition of a group of X-ray images that will finally compose the final volume. A cross-section capture is taken at each angle when performing full rotation around the body, it measures the quantity of energy (radiation) received by the detector compared with the emitted. Then, the table where the scanned subject is located moves forward and the next image slice is taken. This can be seen in the Figure 2.3.

Magnetic resonance imaging is based on the excitation of the hydrogen protons using a strong magnetic field to a defined resonance frequency. This will cause an increase of energy on the protons that will be released as photons and detected by the scanner as an electromagnetic signal.

Micro-CT scans have the same principles of CT but on a small scale with greatly increased resolution. Samples can be imaged with pixel sizes as small as 100 nanometers and objects can be scanned as large as 200 millimeters in diameter. There exist two companies

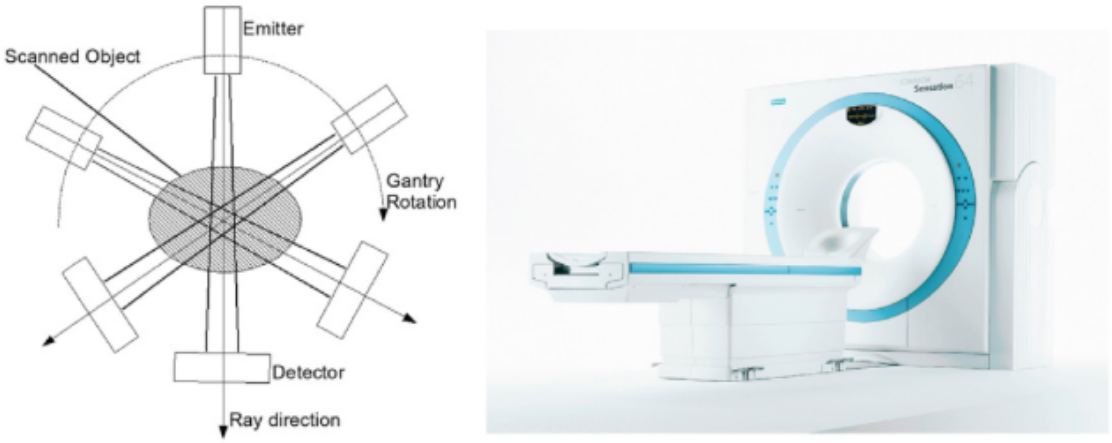


Figure 2.3: Left: Measures reconstructed to a 2D image. Right: Modern CT scanner [29]

that own this device in Barcelona (CORELAB Laboratory, a research group of Barcelona University and X-Ray Imatek, digital pixelated detectors start-up), unfortunately, none of them was available for a visit to their facilities.

The obtained results for all these techniques are very complex and provide a lot of data, therefore imaging technicians define an inter-slice distance in order to simplify the volume and create a more accessible and manageable file called Digital Imaging and Communication On Medicine (DICOM). Dicom files consist of a set of images that contain the density of a voxel in the space in one slice, thus, a Dicom loader is needed in the applications in order to be able to use them.

2.2.2 Modeling Case

The industry of films and video games also benefits of the usage of volume rendering, in this case, the person that requires the application is an artist. God rays, clouds, smoke, nebulae, and creation of new materials are topics where volume rendering is commonly used (there exist other approaches as the cloud mesh modeling, but we will not consider them since the result is not volumetric).

In these cases, the dataset is not loaded but created using mathematical operations. The complexity of the calculation is variable, it can be very basic by defining the density as constant throughout the whole material, it can be like a gradient in one direction (useful

in fog), it can have a noise distribution in the space (useful in clouds and smoke), and more possibilities (Voronoi or Musgrave). Since the density value does not come from any preloaded volume dataset, there are techniques that we are not able to use in this case like Texture Slicing, therefore the ray casting technique is the best approach in this situation.

2.3 Visual Programming Languages

Another big main topic of this project are the VPLs. In order to justify its importance in this project and understand clearly why they are very well-known, we will explain its popularity and detail the models from their creation until now.

2.3.1 Dataflow Language

As I already mentioned in section 1.3, one of our objectives is to build a graph system, therefore we need to understand the graph models with the purpose of constructing the correct model for the application. A graph has vertices or nodes that are connected by edges or links, graphs can be classified into directed or undirected graphs depending if they have an orientation or not. In our case, we will only consider the directed graphs because we are trying to circulate information in a concrete direction (output).

It is also necessary to specify how to define the behaviors of the system (including the organization of the code or even its execution model), this term is known in the programming field as programming paradigms and it is a way of classifying programming languages. In this situation, the programming paradigm that defines the most our VPL is dataflow programming which models the program as a directed graph.

This architecture was first proposed in the 1960s by Jack Dennis, and have been an important pillar of VPL. Since then, many VPLs started to follow his model, such as ConMan (Paul E. Haeberli, 1998) [18] whose operation of the application was defined as: “The user constructs and modifies applications by creating components that are interconnected on the screen. The window manager supports the creation and deletion of individual

components, while the user changes the interconnection by interacting". And these specifications fit perfectly on the objective proposed for this project.

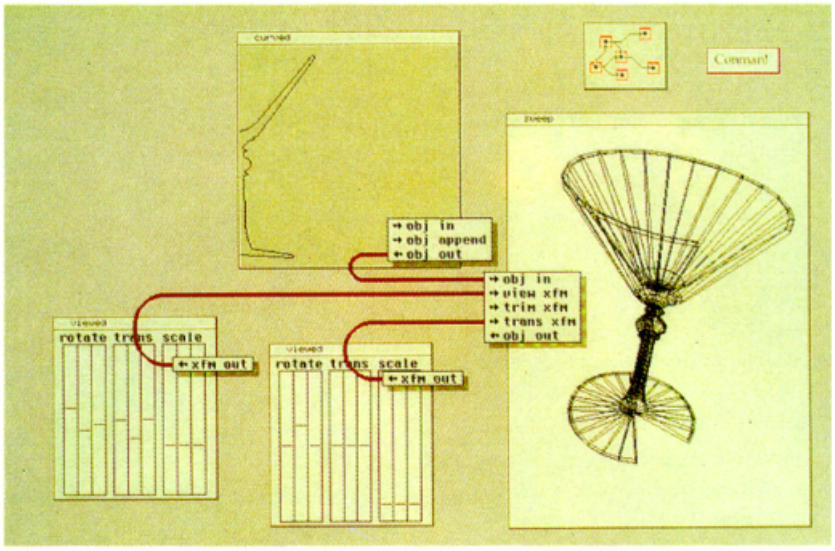


Figure 2.4: A simple ConMan application

2.3.2 Democracy of VPLs

In recent years the democratization of the Internet has provoked an increase of audiences that use it daily without programming knowledge, therefore most of the tools are created to be user-friendly. Proponents of VPL say that people tend to remember things in terms of pictures and so there is no need to keep on trying to "communicate" with the computers using text programming languages [8], it also skips the barriers of the natural languages (pictures are understood regardless of which language you speak). Furthermore, this would help to mitigate this steep learning curve problem and make modern computer capacities more accessible to a wider range of people.

For those reasons, VPL is already in use in multiple domains. Many surveys were made showing the distribution in different fields (Figure 2.5).

Some real cases in education are Kodu [24] or Scratch [25], furthermore there are studies reflecting a significant improvement in the understanding of concepts while keeping a high level of concentration and having fun (which is also an important factor when dealing with kids) [34].

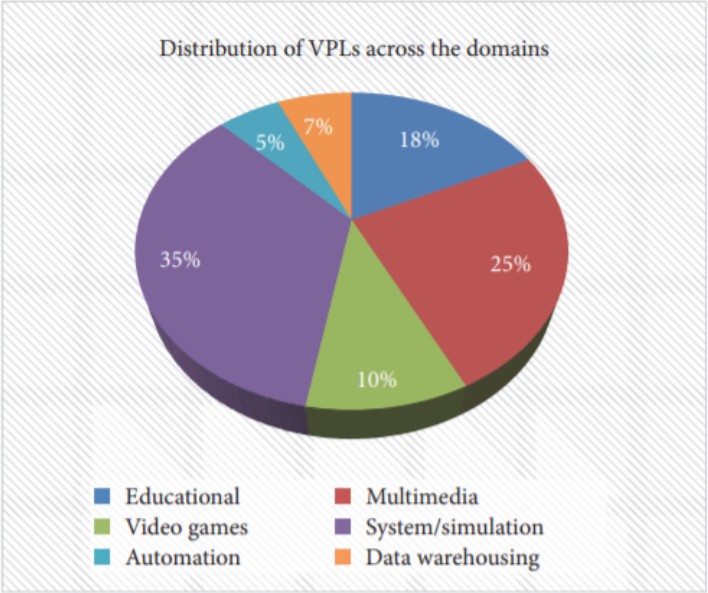


Figure 2.5: Percentage-wise distribution of VPLs [33]

2.3.3 Classification Models

The most common VPL models within all the implementations can be categorized into two types: blocks and graphs.

The most popular examples for the blocks model are Scratch (MIT Media Lab, 2003) which was originally created to teach young people about programming, and Blocky (Google, 2012), which has the advantage of being run in a web browser. Moreover, Blocky is also being implemented for mobile phones, which will give a huge improvement in accessibility.

The interface of Blocky is very simple and tidy, on the left, it has all the possible nodes gathered in different folders depending on the functionality. On the center, the user can manipulate the system, add, delete, and connect nodes. Finally, on the right, the code that is resulting from the blocks system is shown. Its interface also helps the novel users to understand what the nodes do without using a single text box, this can be seen, for example in the loop nodes, that wrap the code that will be inside them. Moreover, it uses different colors to represent a difference between the types of nodes, and the connections are easy to visualize, they appear as wholes for the inputs and lumps for the outputs.

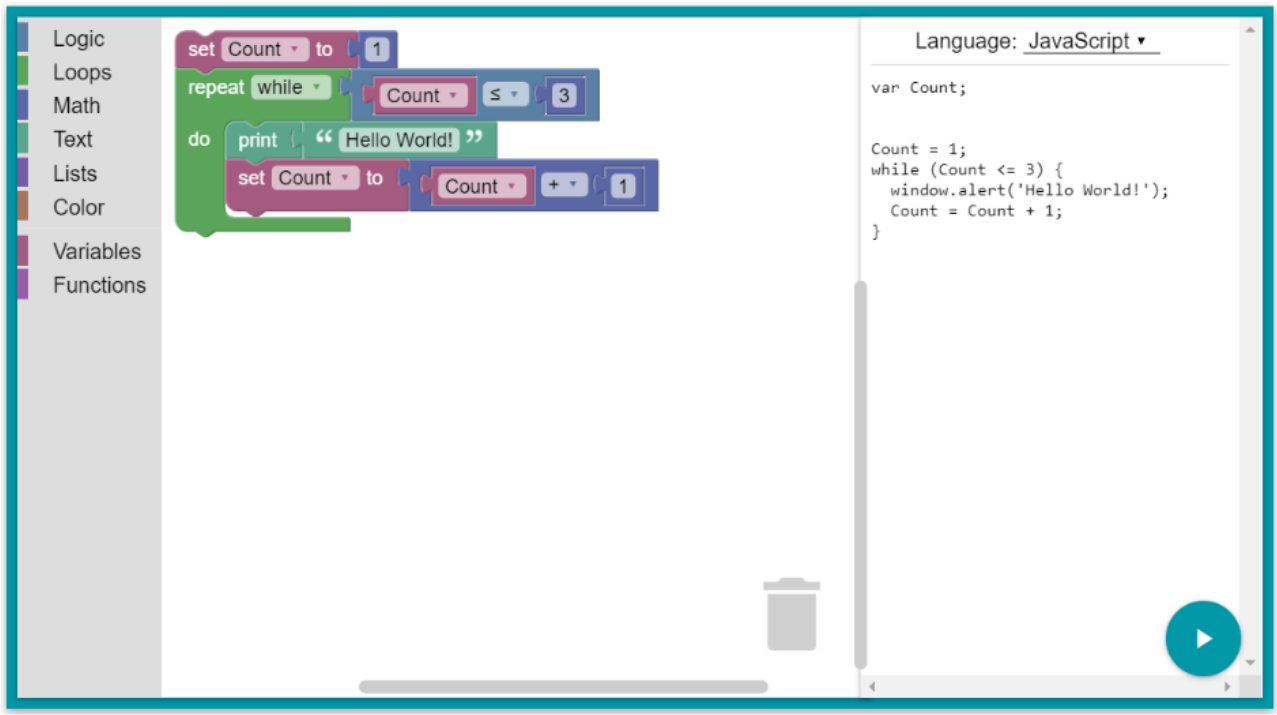


Figure 2.6: Blocky basic interface

On the other hand, graph models help to conceive better the direction of the dataflow thanks to the use of the linkers and the nodes. This type of architecture is more popular in the film and video game industry, specifying its objective field and consequently its programming. For that reason, graph systems are not usually conceived for educational purposes, but they expect users to create results that will have a plausible application in some of the previously mentioned industries.

2.4 Visual Shader Editors

As a derivation of the VPLs, the VSEs are very common these days. The advantages of visual programming are notorious in the shading field, for that reason, almost all of the editors of the game industry have one specifically created for their users.

The explanation of this section is divided into four subsections that will help to understand the basis of the VSEs and their relation with this project.

2.4.1 Shaders Programming

Central Processing Units (CPUs) would struggle if they had to handle the computation needed to calculate the final result of each pixel, furthermore, in 3D games, geometries and perspectives need to be calculated as well. For those reasons, we created Graphics Processing Units (GPUs), which architecture is built specifically for parallel processes, which means that it increases the number of processes you can perform at the same time, and consequently reduces the computational cost.

The term "shader" was first introduced by Pixar with the version 3.0 of their RenderMan Interface Specification (May 1988). From there on, a diversity of shader languages have appeared depending on the Application Programming Interface (API), all of them with the goal of providing a high-quality rendering. The official OpenGL Shading Language (GLSL), the official Direct3D High Level Shader Language (HLSL), and the Nvidia C for Graphics (CG) are the most known shading languages these days.

Thanks to the shading languages and the evolution of the GPU we can render complex surfaces and make them appear more realistic by using either physical algorithms that define the behavior of parameters such as light or approximations that recreates the appearance of materials or particles.

2.4.2 Rendering Pipeline

In order to visualize the function of the shaders within the whole scene, we need to refer to the graphics pipeline presented in the Fourth Edition Real-Time Graphics book (Figure 2.7). The pipeline contains four stages, which performs part of a larger task containing several substages.

As the name indicates, the Application stage is driven by the application and is implemented in the CPU. Nevertheless, there exist methods to delegate some computation to the GPU using the compute shader. At the end of this stage, the information is sent to the Geometry Processing, which we could consider the origin of the shader. The shader contains all the steps the GPUs do to render one object into the framebuffer.

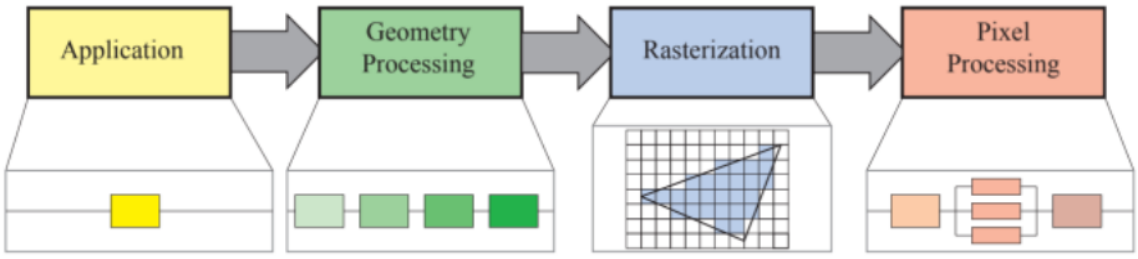


Figure 2.7: Basic construction of the Graphics Pipeline [3]

Originally shaders only performed pixel operations (operations are done for each pixel), but this is what we now call pixel shaders. Due to the introduction of other types of shaders, the term shader now describes a more complex process that covers a full complete pipeline (Figure 2.8) which includes all the new existing shaders.

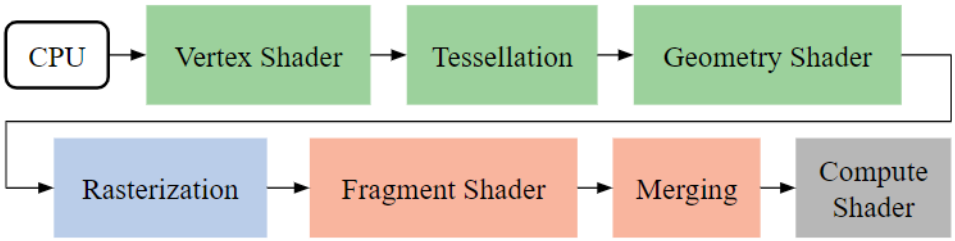


Figure 2.8: GPU Implementation of the Rendering Pipeline

Vertex shader is executed for each vertex of the render scene with the purpose of transforming the 3D space into frame space (2D). Since vertex shader can modify models and coordinates, it is useful when we want to alter the structure of the meshes.

Tessellation shaders are composed of two independent shader stages (hull shader and domain shader) which allow the subdivision of meshes at run-time. This allows creating a level-of-detail scale depending on the distance from the camera. Besides, domain shader computes the actual vertex position of any given point that resulted from the tessellation.

Geometry shader can generate vertices that will affect the graphics pipeline, this allows the possibility of programming meshes to modify automatically its complexity.

Pixel shader also known as fragment shader, computes the color and other attributes of each pixel. Its complexity can go further by having several inputs and outputs or altering the depth of the fragment. The knowledge of the screen coordinates being drawn enables a

quite amount of post-processing effects such as blur. For this project, we will be working specifically with the fragment shader because the objective is to edit the returning color for a pixel.

Compute shader is not fully related to graphics, but it takes advantage of the architecture of the GPU to compute additional code specially designed into graphical form (parallel programming).

Apart from these, there are different shaders that some people have been tried to standardize (mesh/primitive shader) [27], but none of them have turned out to be popular for the moment so it does not makes sense to explain possible future ideas.

2.4.3 Node-Based Shaders

Nodal architectures have received an ever-increasing endorsement in computer graphics in recent years, its functionality is the same as VPL but specific for shaders. A node-based shader is formed by three components: the nodes, the connections, and the graph itself.

Nodes contain pieces of code or even operation that will be used in the output node, this interim operations will affect in different locations of the final output shader, therefore by creating and connecting new nodes the shader will be modified. In order to group and define the whole structure, they are connected with each other depending on the decisions of the user. The connections are executed by linking and input with an output, always having in mind that the connector type must match between them.

The types of the connections depend on the type of data flowing through them. For example, a color could be defined as a vector3 (RGB) or a vector4 (RGBA), and a vector can not obviously input a node that is expecting a float. This design is considered as a requirement of the project.

Once the system is completed, it is then traversed recursively to generate an output. We observed an already discussed (2.2) that represents a directed acyclic graph (DAG), in which nodes represent operations and wires represent the geometry stream. With the finality of shifting the development paradigm from a product-based representation to a

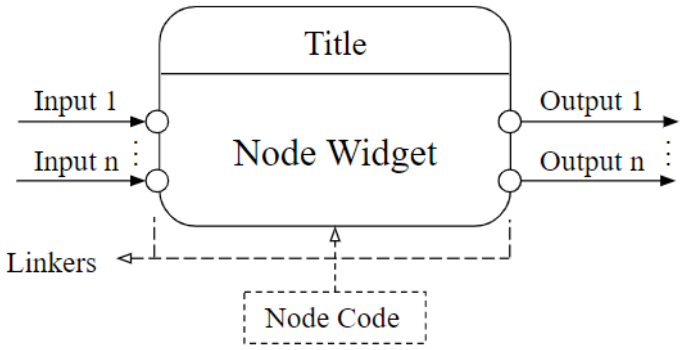


Figure 2.9: Node Structure

user-friendly visual-operation-based one. As the name says, DAG is acyclic, therefore its data flow goes from the first input nodes to the final output nodes, and they never return to a previous node. The graph is then traversed recursively to generate the output. Each node is responsible for querying its sources for the values needed to compute its output.

Visual programming language for interactive graphics will let users create shaders from scratch by manipulating nodes, interactively connecting them, and controlling the creation process workflow. This change will let users create rule bases from scratch without having to edit text files, which contributes to bridging the gap commented in this project.

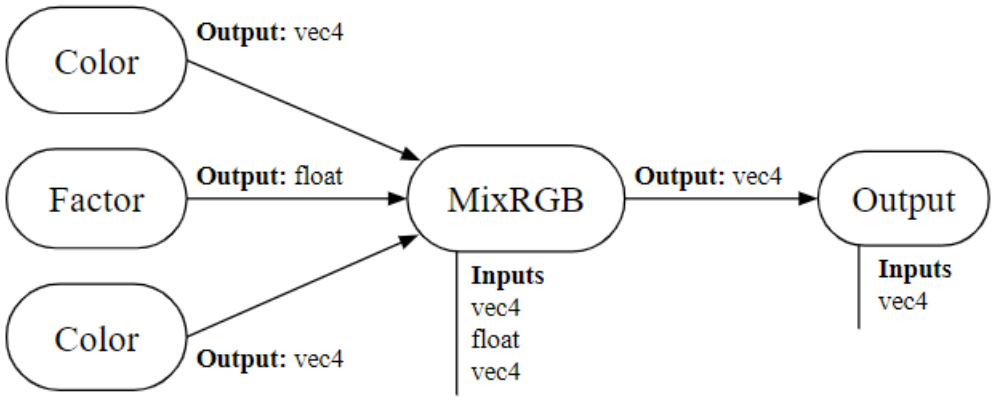


Figure 2.10: DAG Structure for interactive graphics

2.4.4 Visual Shader Editors References

In the last few years, along with the growth of the computer graphics industry, visual shader editors have been implemented in all applications like Unity, Unreal Engine, etc.

They are so common that users already expect to control the app using a node editor, for that reason, when thinking of creating a visualizer for objects, there must be always a visual editor for the user manipulation of the system. Therefore, because of the objective of this thesis, Blender has been taken as a motivation and as a reference.

Blender is a free software program fully dedicated to computer graphics. It is a platform that lets artists create different kinds of art, like modeling, rendering, or animating, among others. In the case of shading, Blender has a very complex node graph system that allows users to create all types of materials by modifying every existing property. As with Blocky, the code is available in their repository [5].

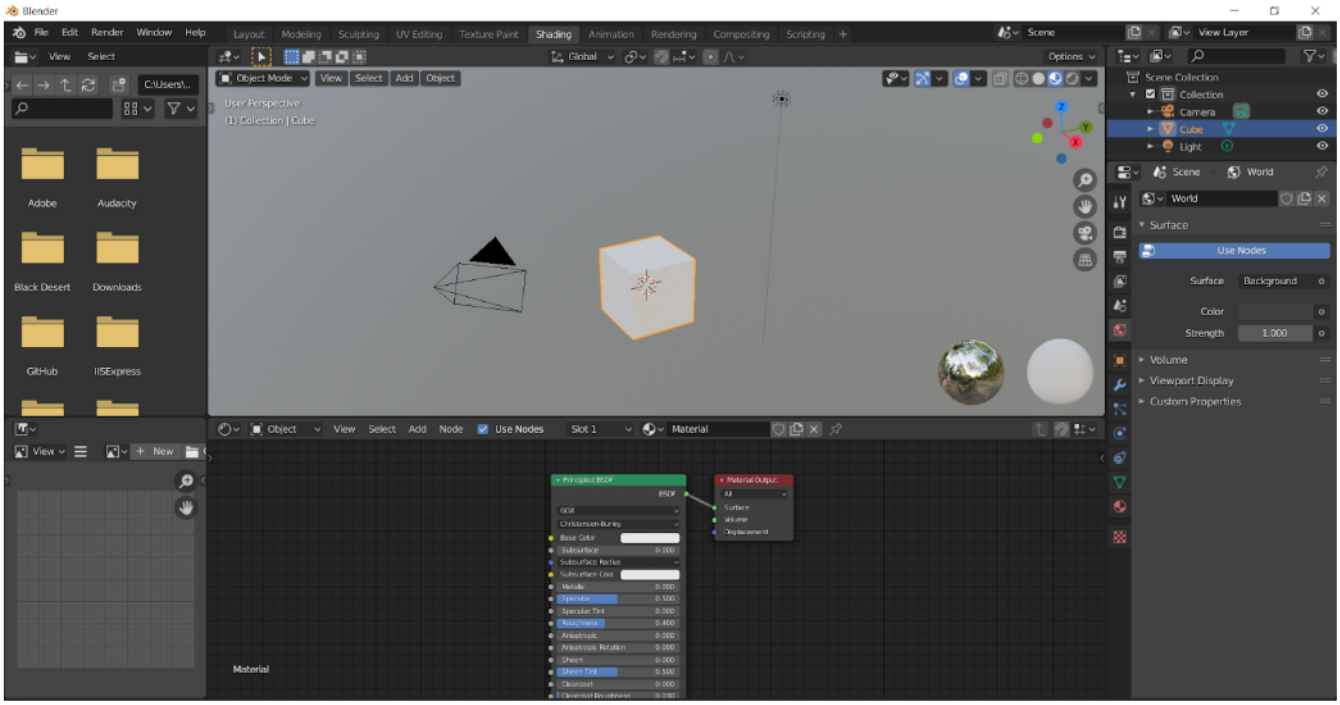


Figure 2.11: Blender Node Graph System

In Figure 2.11 we can see the interface for shading of Blender. Underneath the canvas view, there are two nodes connected, the one from the right is called Material Output is the final node that will end the workflow and create the final shader that will be used to render the material, which inputs are Surface, Volume and Displacement. That means that Blender supports the light interaction with the surface of the objects, the emission absorption and scatter effects for the volume, and the possibility of detailing the bumping of the object. This is a complete control of the material properties, for that reason Blender is one

of the top applications with one of the hugest and more active communities of computer graphics. Therefore, the Volume graph system of Blender will be a nice reference for the objective of this project.

Another important aspect to mention is that Blender does not give freedom to the user for selecting which shader the code is sending to. As explained, there exists several types of shader, and each of them plays an important role in the final result. This decision is due to the unnecessity of this, Blender defines very well the different functionalities of the applications by creating sections on the top. If the user is in the shading section, there is no reason of letting the user select the final shader since it is surely going to the fragment shader. Nevertheless, there is a way to modify the vertices and that is using the Displacement input of the Material Output node (actually, there's a difference in the color of the connector with the other two inputs), this input affects the displacement shader (domain shader).

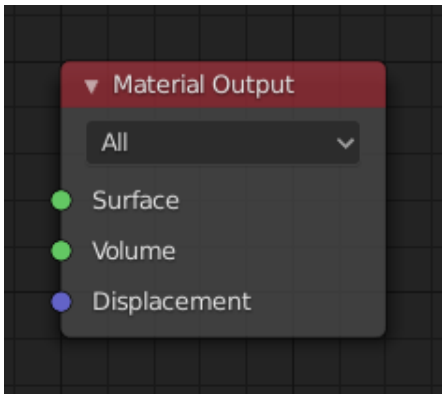


Figure 2.12: Material Output node

2.5 Web Applications

Web applications are currently very popular due to the advantages they have respect application softwares. Some of these benefits are the accessibility (compatible by almost all the computers, and downloads of files are not needed), the permanent update to the last version without any effort, and the reliability for some uses who are reluctant to download external programs to their computer.

On the other hand, it requires a programming knowledge for the developer, more precisely the programming languages HyperText Markup Language (HTML), Javascript (JS), and Cascading Style Sheets (CSS). These are the basic components of all web frameworks. To be more specific, HTML controls the structure for the web content, CSS describes the presentation of the structure and JS defines the behavior of the program. This website represents perfectly the relation between them [19].

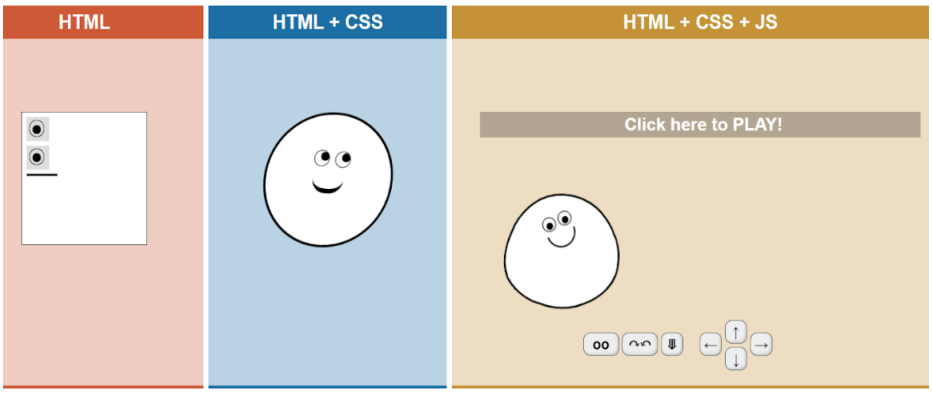


Figure 2.13: Visual relation between HTML, JS and CSS [19]

Thanks to the simplicity, many novel programmers develop their own applications from scratch giving very interesting results and helping them to enter the industry. This can be complemented with the GitHub disponibility, which gives visibility to users and reserve a unique website for testing the applications.

In many of them, it is required the 3D visualization of a created scene. For example in this case, the user developed a web app to visualize some examples of volumetric datasets [14]. Thanks to frameworks like that, we can notice the importance of Computer Graphics in web applications.

2.5.1 Computer Graphics on Web

The beginning of Computer Graphics on the web is in 2011, when Khronos Group developed WebGL. WebGL is an API based on OpenGL ES that allows the 3D rendering in HTML5 canvas elements. WebGL is a DOM API, which means that it is compatible with Javascript (and other DOM-compatible languages).

There exist several libraries that makes it easier for the user to manipulate WebGL, by creating useful classes and simplifying its syntax. In this project, we will use the library LiteGL.js [20].

Chapter 3

DESIGN

In order to achieve a fulfilling experience for the user, a web application is needed. To do so, it is essential to explain a design stage detailing very concrete aspects both visual (user interface, functionalities) and internal (framework, graph system).

3.1 Collection of Requirements

The requirements and consequently the tests will prove the strength of the system and the quality of the work done. These can be divided into "functionals" and "not functionals". Functional requirements describe the behavior of the system, on the other hand, non-functional requirements indicate the characteristics of the system.

3.1.1 Functional

- **Datasets Portability:** it must accept the data file .dicom if someone uses its own dataset.
- **Volume View:** there must be always a canvas showing the current result of the shader resulting for the system.
- **Node Creation:** user can create nodes and connect them with others.
- **Supporting Nodes:** nodes input must modify its output (node interaction).

- **User Case Adaptative:** the system can be modified as the desires of the user in order to fit nicely in different applications, and being functional in all of them.
- **Code View:** you have to be able to visualize the final code generated.

3.1.2 Non-Functional

- **Browser Independence:** it must work independently of the browser running.
- **Robustness:** it must not contain bugs, all error possibilities should be considered.
- **Intuitive:** the algorithm must be easy to use for the users without experience.
- **Quality:** the results must be qualitatively good in order to make it really useful and needed.
- **Graph Improvement:** it must show the advantages of using graphs over code programming.
- **OS Independence:** it must work independently of the OS using.

3.2 Visual Aspects

This section is dedicated to developing how the application was designed visually, the explanation is divided into three parts: the User Interface (UI), its functionality, and the visual aspect of the nodes.

3.2.1 User Interface

The UI is the link between users and your website, it must be designed carefully by anticipating the user preferences and behavior. The UI not only worries about aesthetics but also about the efficiency of the website by being intuitive for the users.

The most important feature of this project is the graph editor and its visualization, therefore it will have an equal impact on the interface. Moreover, the header is necessary when creating a website. It is said that the header is the first thing that people see when they

land on a website, and it also offers the possibility of showing different functionalities, options, and information using buttons or links.

The following image shows the distribution of the webpage, giving all the protagonism to the two main canvas and letting the header as a helpful multitool for the user.

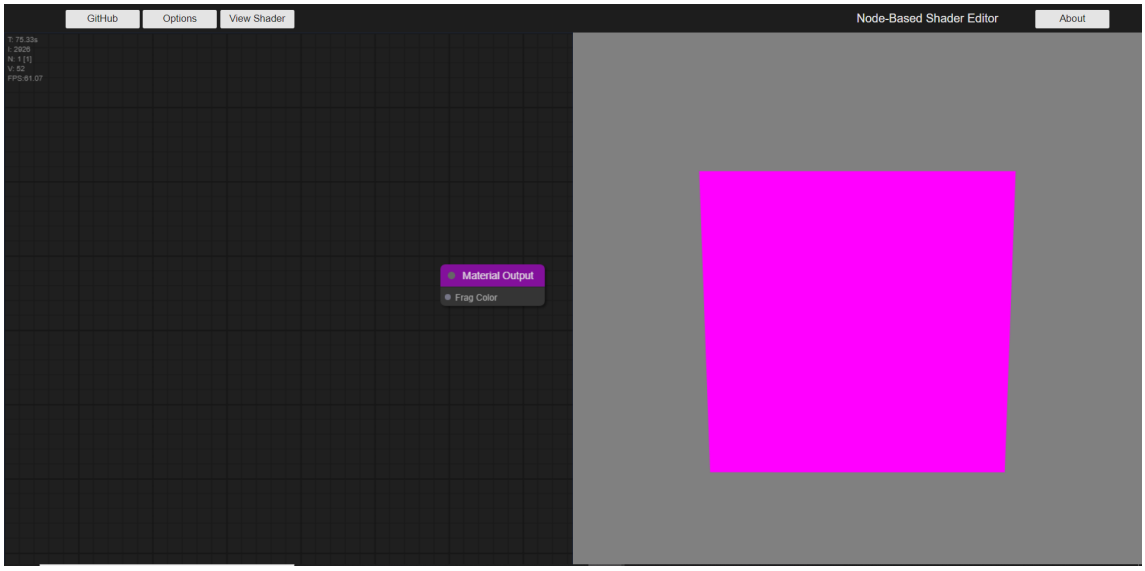


Figure 3.1: Basic interface of the website

3.2.2 Functionalities

From now on, all the responsibility lies in the response to the user petition by interacting via inputs (mouse).

The left canvas of Figure 3.1 is created from the LGraphCanvas class from Litegraph library (explained in the Implementation section), additionally, it also provides different functions that help the control of the user over the editor. The dragging of the left mouse button permits to move around a vast space where the nodes can be positioned, if the same action is done over a node, this is the one who will be moved. If pressing two times the left button, it will open a widget that recreates a search bar where the user can search for a specific existing node. On the other hand, if the right button is pressed the user can open the node selector (the nodes are stored in different sections) or can create a group, which is an area that joins all the nodes inside it and allows the global movement, making them move together.

The right canvas contains the visualization of the scene, as common in all computer graphics views the user is able to move the camera around the mesh using both left and right mouse buttons and control the perspective scrolling the wheel of the mouse.

Finally, a not very pronounced header is placed on top, which includes four buttons: Github, Options, View Shader, and About. The first button is a direct link to the source code stored in a Github repository. The Options button opens a panel where a bunch of different general properties of the scene can be modified. The View Shader button also opens a panel where the code created by the graph editor is shown, it includes both the vertex shader and the fragment shader and also an option to download it as a shader_atlas.gltf, which is the typical file format used to put all shaders into a single file, separated by some markers. The About button opens a general web page information panel, introducing the libraries used, and a little guide for the usage of the node if the user is lost.

3.2.3 Visual Nodes

Like in the editor canvas, the nodes are mainly controlled using the Litegraph library. Their visual appearance already shows the three parts division of the whole structure (Figure 3.2).

Figure 3.2: Basic node appearance

Properties: in order to create a correct customizable application, not only the graph can be edited, but each node internally. For that reason, the main properties that define nodes can be modified by the user visually using the widgets.

Outputs: the sockets from the right called outputs represent the flow of the inside coding of the nodes to the followings. Since the information direction is from front to back, outputs do not need to consider the receiving of information. Therefore the type and

value of the output are the ones that dictate the inputs how are they supposed to prepare for getting the information. The link of an output is represented by setting the output point color to green instead of grey.

Inputs: the inputs indicate that the node may require external factors to modify its function (additionally to the widget properties). As explained with outputs, the inputs must be consistent with the type of information they want to receive for each property. Regardless outputs can connect different inputs, inputs can only be connected to one single output. The link of an input is represented with the green color and the disconnection with grey. In the case the input is disconnected, it has a default value (except for the nodes that expects a shader input, in our case the Material Output node) which can be edited via widgets.

Links: the connections realized between nodes are represented by a line that appears when clicking over an output socket and dragging the mouse. This line joins with the desired input if both sockets share data type (color - color, ...).

Apart from that, the double left click on a node opens a panel where basic information of its function is shown, it also allows the modification of its properties and the delete of the own node.

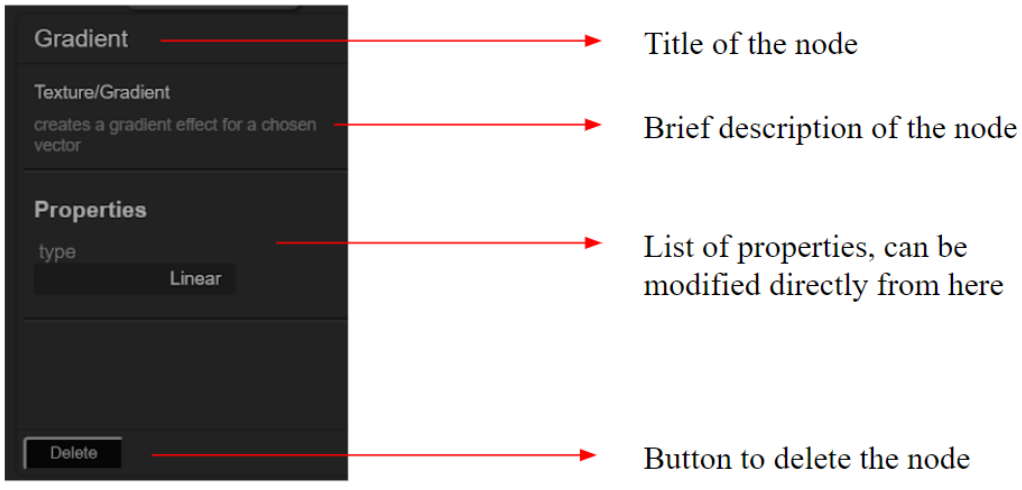


Figure 3.3: Structure of the support panel of the nodes

It is placed in the bottom-left corner of the window in order to not block the volume visualization.

3.3 Internal Aspects

Once explained how the application works visually, it is necessary to explain its internal operation. More precisely, the architecture and how does the global graph works.

3.3.1 Architecture

We can understand the behavior using a Model-View-Controller Architecture, where the View shows the content to the user and receive the inputs, the Controller manages the information passed from the View and gives the respective instruction to modify the data, and the Model is the Graphics Engine which contains the business logic of the visualization (Figure 3.4).

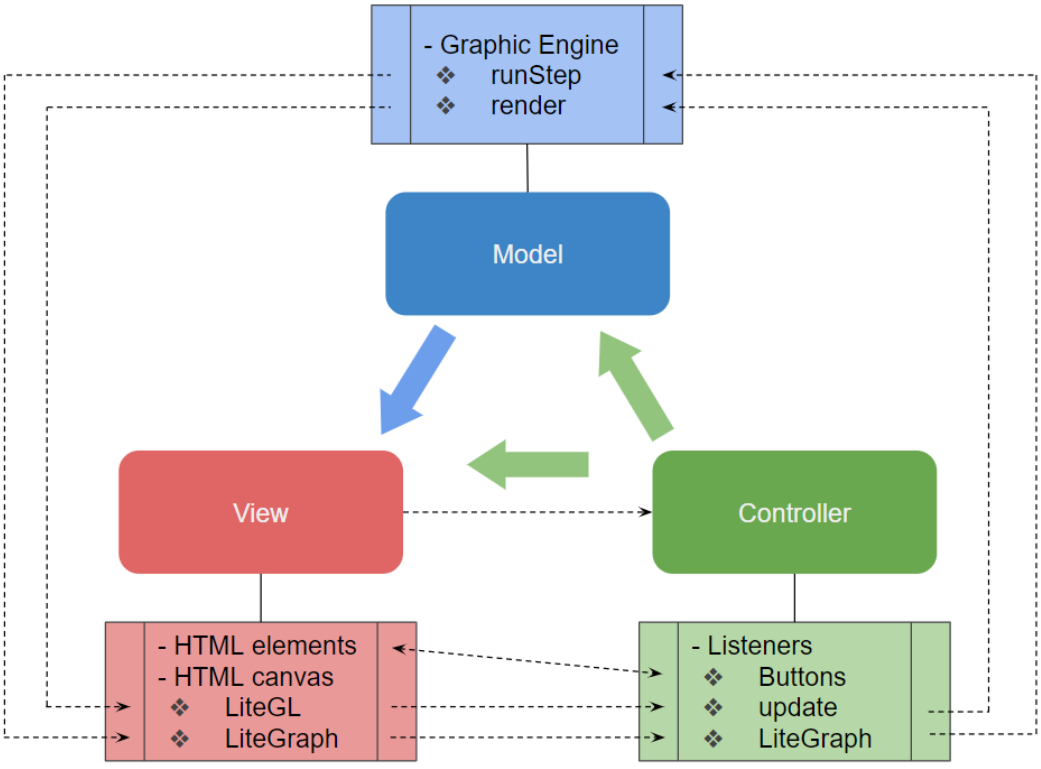


Figure 3.4: MVC pattern of the framework

But first, the function init() is called, this method has several functionalities in order to complete the initialization of the framework:

1. **Init the module:** initGraphCanvas() and initWebGLView() prepare their canvas with their respective functionalities.
2. **Init the listener:** initListeners() method assigns the callback at each event of the section of the Controller "Button" and Mouse (which controls the "update").
3. **Load:** loadShaderAtlas() and addNodes() load the shader templates and register the nodes implemented in the LiteGraph library.
4. **Init scenes:** once the modules are operative (done in the first step), the method graphTemplate() creates an initial scene with a specific graph, and the method createScene() enables the tools used in the graphic engine (camera, entity, and shader).
5. **Loop:** the last step is to call the main loop that will call itself recursively and start the Model-View-Controller scenario.

The file classification consists of the main.js, that calls the init() function and its corresponding functionalities, the volumeNodes.js, which contains the implementation of all the nodes, the sceneTools.js, that extracts as classes two main tools of the render, and the utils.js, which contains all the useful functions that are used in the framework. Apart from that, the required libraries are grouped together in the folder "external".

3.3.2 Shader Generation

The second part of the internal design is the shader generation, how the nodes affect, and how they are read in order to build a final shader.

We run the graph at each render iteration, and it executes all the main functions for all the existing nodes (in the editor). Each node sends to the following nodes the received input plus a contribution until it arrives at the Output node and creates the shader by filling a string template. It is also in charge of passing to the shader the required code and uniforms from the nodes in use. This process is represented visually in Figure 3.5.

The decision of making the Output node to fill the code that will be sent to the shader instead of delegating this function to each node was taken due to some errors that could happen if the addition of the codes of the nodes was applied at its own execution method.

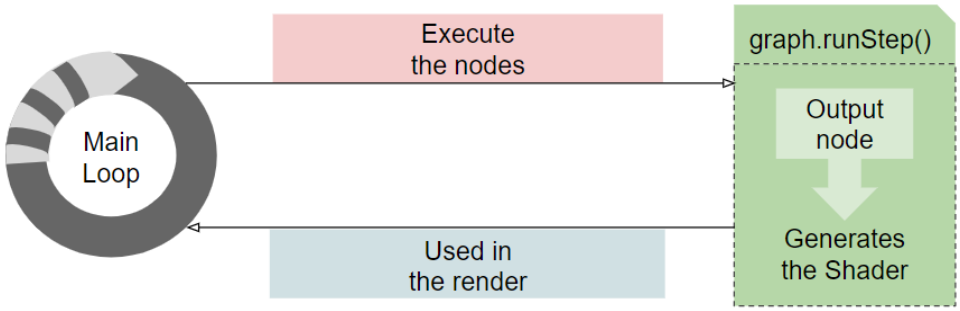


Figure 3.5: Visual Pipeline of the shader generation

In this case, if there were some nodes repeated in the graph, it would add two times the same code and create errors. All the decisions taken to control these situations are explained in section 4.3.

3.4 Nodes Design

Even though the design of the previous points is required in order to achieve a nice structured application, the most important part of this project is the decisions taken in the creation of the nodes.

The first thing to do is to think which nodes are going to be implemented and organize them by different typologies. In order to do that, a search was made for the needs of the users, which were classified into two application cases. On one hand, for the medical, their main tool is the usage of an editor for a transfer function (we will see this later). On the other hand, I could learn how the artists work by participating in the Blender community and interacting in the forums. A big amount of the implementations of volume rendering had basically the same set of nodes [6] [7], so with this information in mind, I decided on my final list.

As a method of visual discrimination, each group has its own representing color. This is an indicator of the node functionality and of its inputs and outputs. The descriptions are accompanied by an image of the expected result for each node and its contribution to the final view.

3.4.1 Input Nodes

This group is formed by the nodes Color, Number, Coordinates, and Transfer Function (TF). Its only function is to offer the user a value to use in the shader, therefore the node has no inputs and only one output. They are represented by the red color and their output value can be a vector4 (Color, TF), a float (Number), or a vector3 (Coordinates).

Expected results can be seen in the following images, for example, in Figure 3.6 we can see how the Value node is connected to two different nodes and changes the value of the input linked, it also affects the result visually.

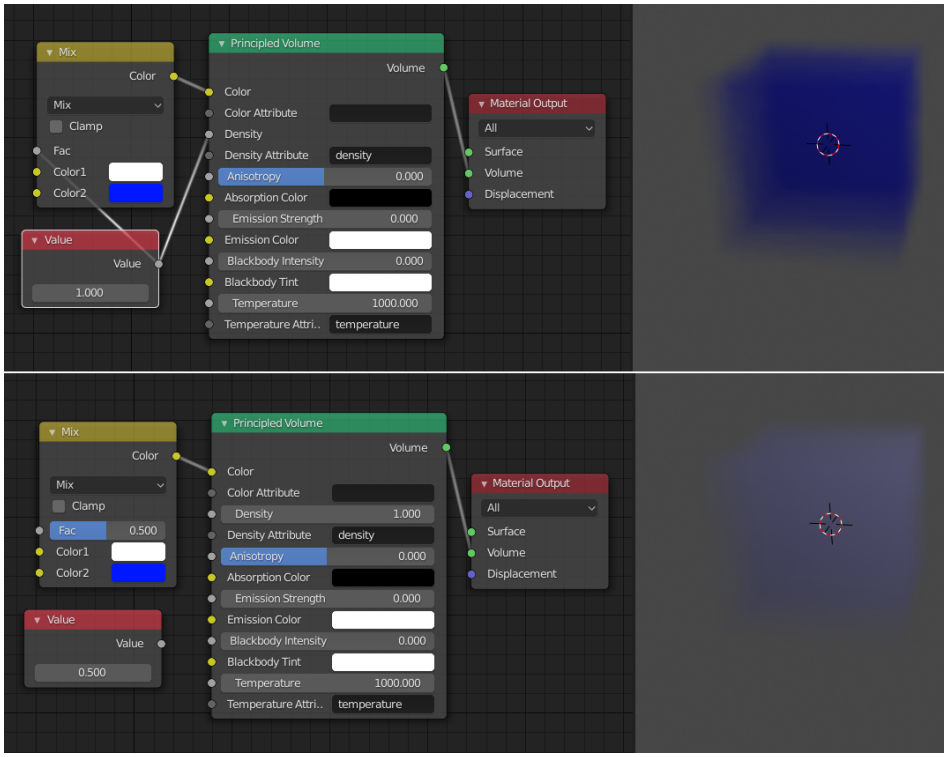


Figure 3.6: Value node in Blender

The Color node should have a color palette so the user can select the desired color and link it to a specific input value and control the color in use.

The Texture Coordinate node allows the selection of the coordinates in use, in the first example from 3.8 the gradient uses the coordinates from the ray (from 0 to 1 in parallel faces), but in the second image the result changes because the gradient is produced using the camera coordinates (from 0 to 1 in the camera view).

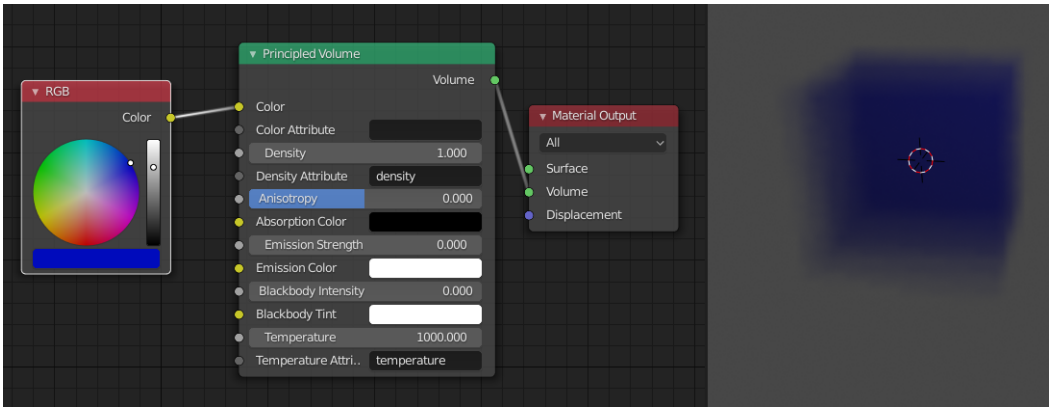


Figure 3.7: Color node in Blender

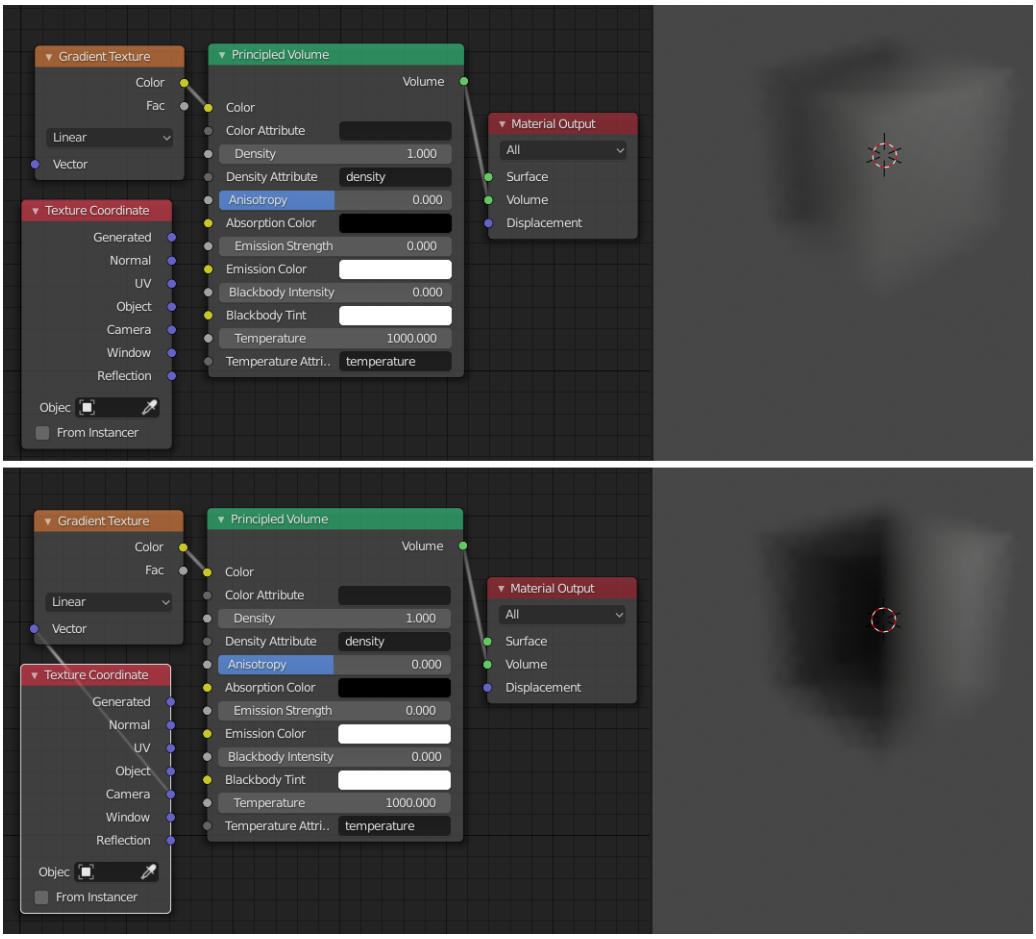


Figure 3.8: Texture Coordinate node in Blender

The transfer function should allow the modification of the color and opacity of the volume depending on the density value by using a widget. Similar to Figure 3.9, but also applied for RGB values.

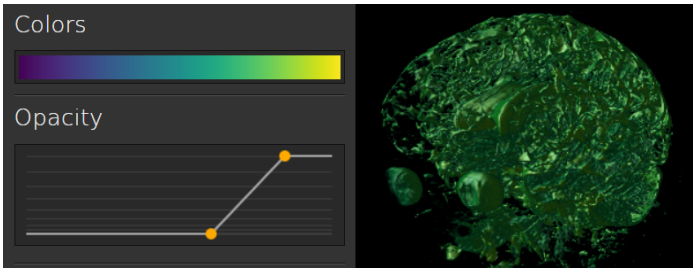


Figure 3.9: Transfer Function utility

3.4.2 Texture Nodes

This set of nodes generates a color value depending on the input coordinates. It is formed by Gradient and Noise nodes and they have the bar orange-colored, the group is called "Texture" because they can be visualized as an image (there even exists some approximations that use textures to compute the output color). The node Dicom is also included in this group since the dataset is stored and used in the shader as a 3D texture.

In the following pair of images, we can see how the nodes change visually the obtained result. The Gradient produces an effect of a color-changing from 0 to 1 (3.10), and the Noise produces a random color variation (3.11).

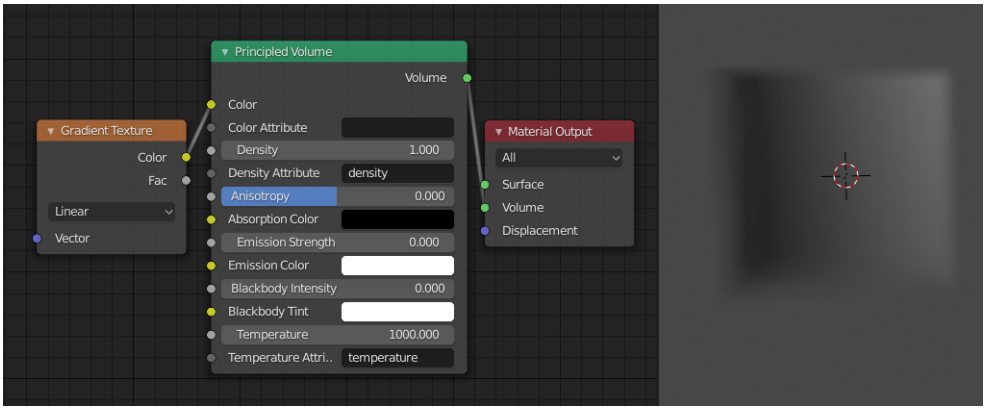


Figure 3.10: Gradient Texture node in Blender

3.4.3 Operator Nodes

Represented in blue, this section contains all the nodes that modify the value of the input using a specific operation. The complete list is formed by Math, MixRGB, ColorRamp, Translate, Rotate, Scale.

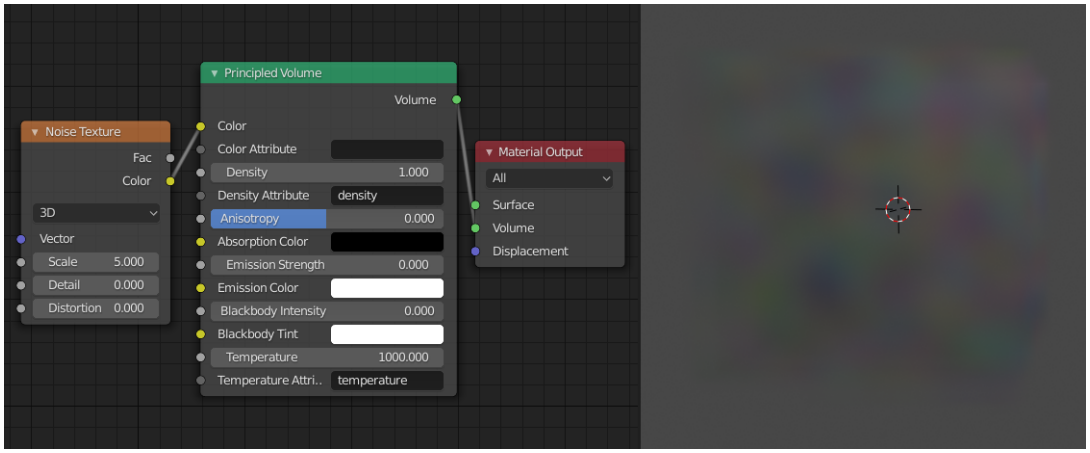


Figure 3.11: Noise Texture node in Blender

The expected results are shown in the following figures, where one or more nodes are connected to the Operator node, and the output is changed depending on the operation type. The Math node should apply a selected operation to two given inputs (Figure 3.12).

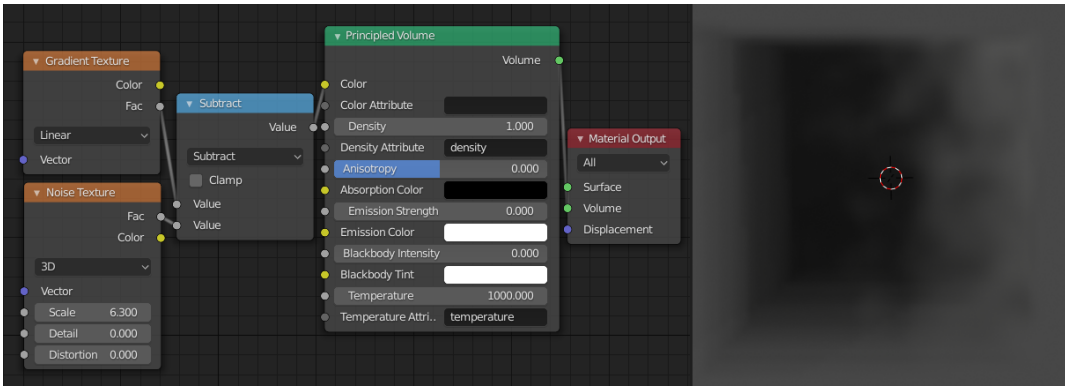
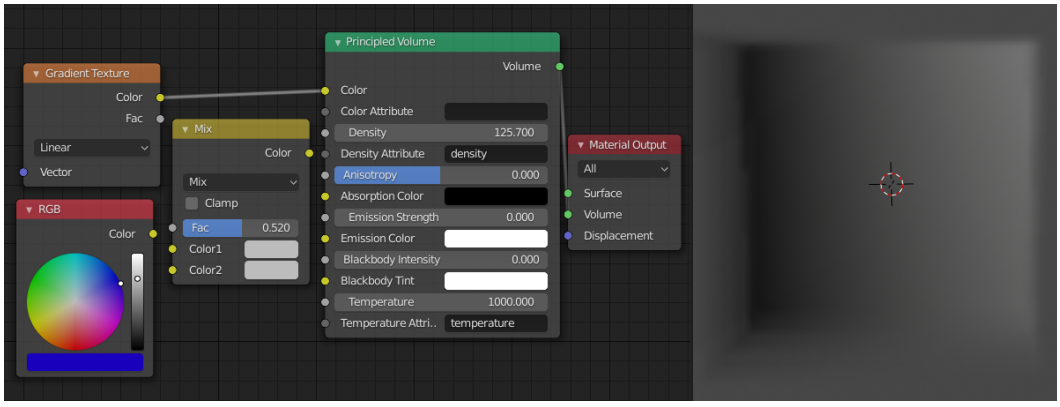


Figure 3.12: Math node in Blender

The Mix node should interpolate the colors given as inputs (Figure 3.13).



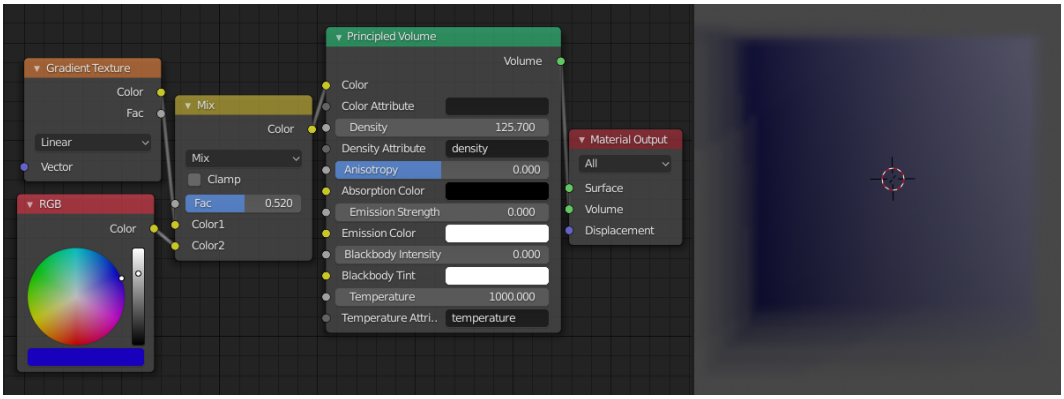


Figure 3.13: Mix node in Blender

The Color Ramp node is used to reduce the curve between the minimum and maximum value, and consequently, will create more contrast in the visualization. This can be seen in 3.14, in the first image, the change from blue to yellow is smoother than on the second image, and this helps to differentiate between extreme values.

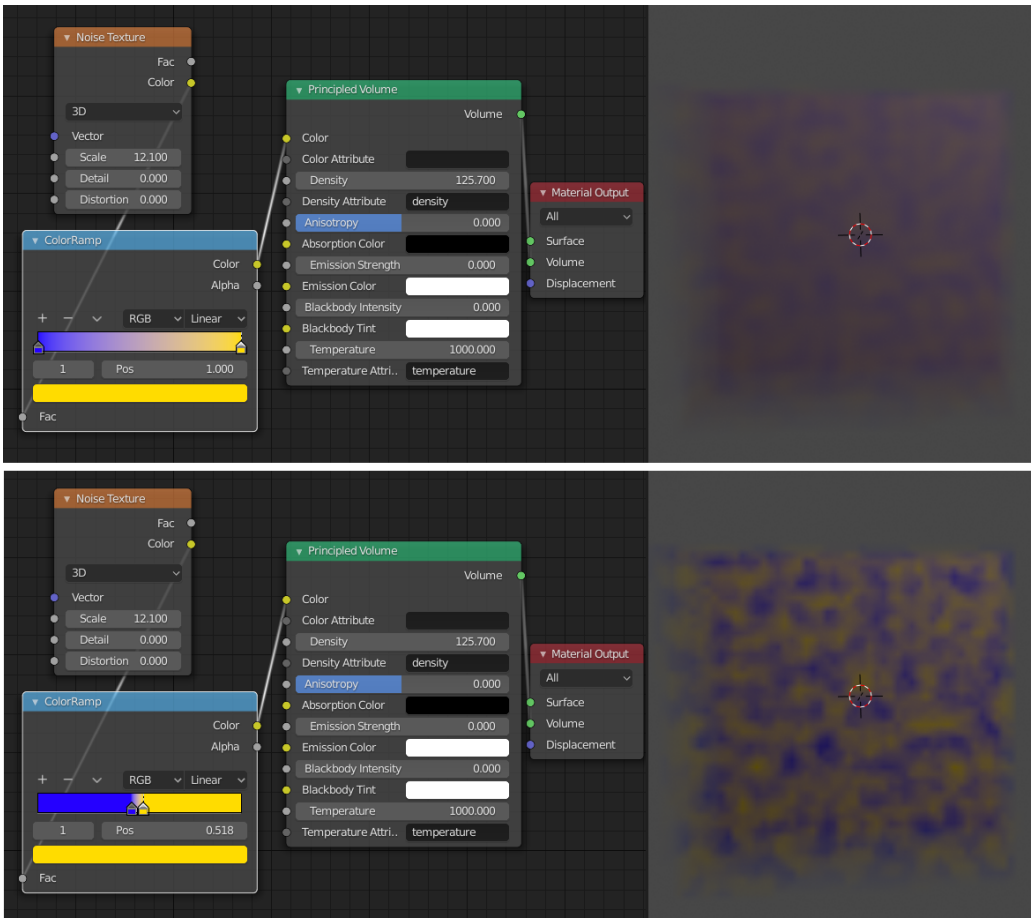


Figure 3.14: Ramp node in Blender

Finally, the Mapping node from Blender wraps the functionality of all the vector operations, in the image 3.15 a vector is selected to produce a gradient effect, and then the operations applied allows the user to control more freely the result.

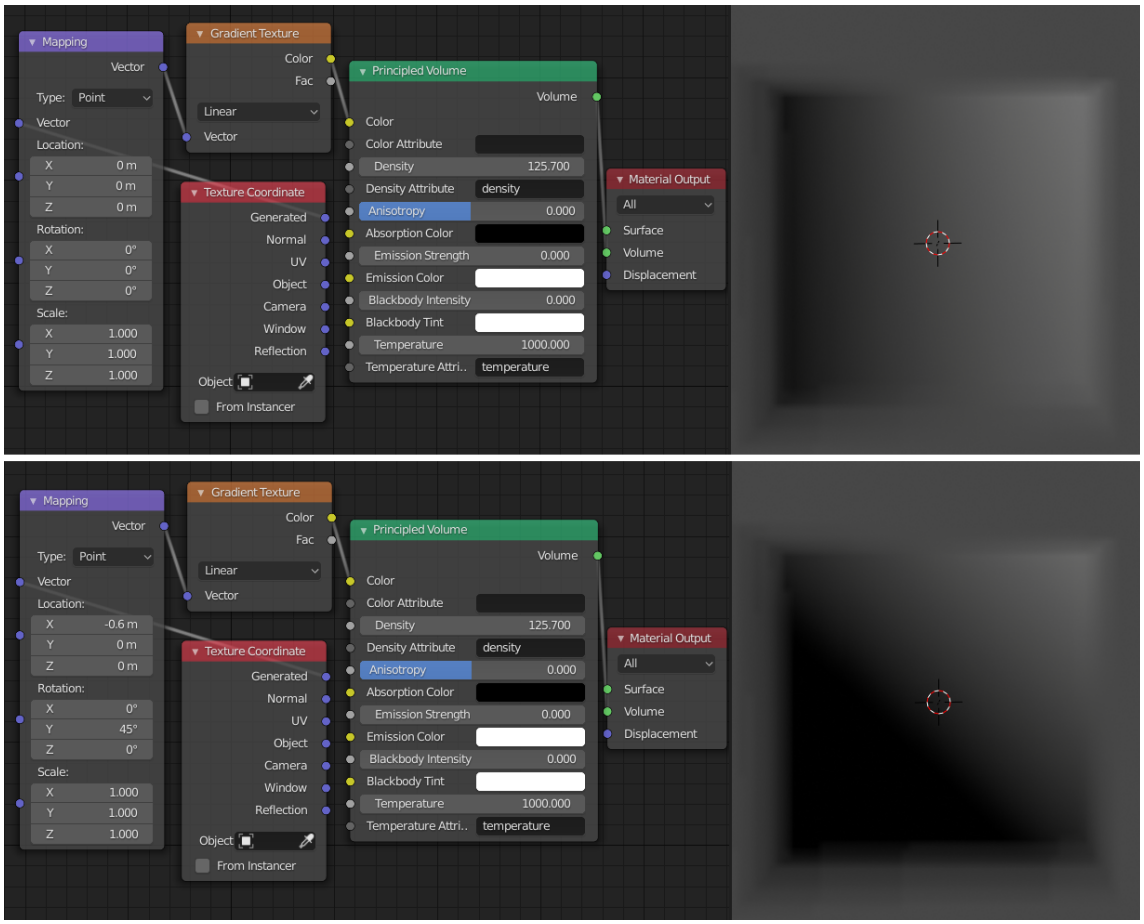


Figure 3.15: Mapping node in Blender

3.4.4 Shader Nodes

This group represents the nodes that complete a final render algorithm with the information receives from the previous nodes. However, in this project, we already presented the specialization on volume render, for this reason, the only node that exists in this folder is Volume. This group is shown in green color.

Its design is trivial, the simple connection to an empty node (which has a basic result, black surface render) changes completely the obtained result, due to the changes produced by the rendering algorithm applied (Figure 3.16).

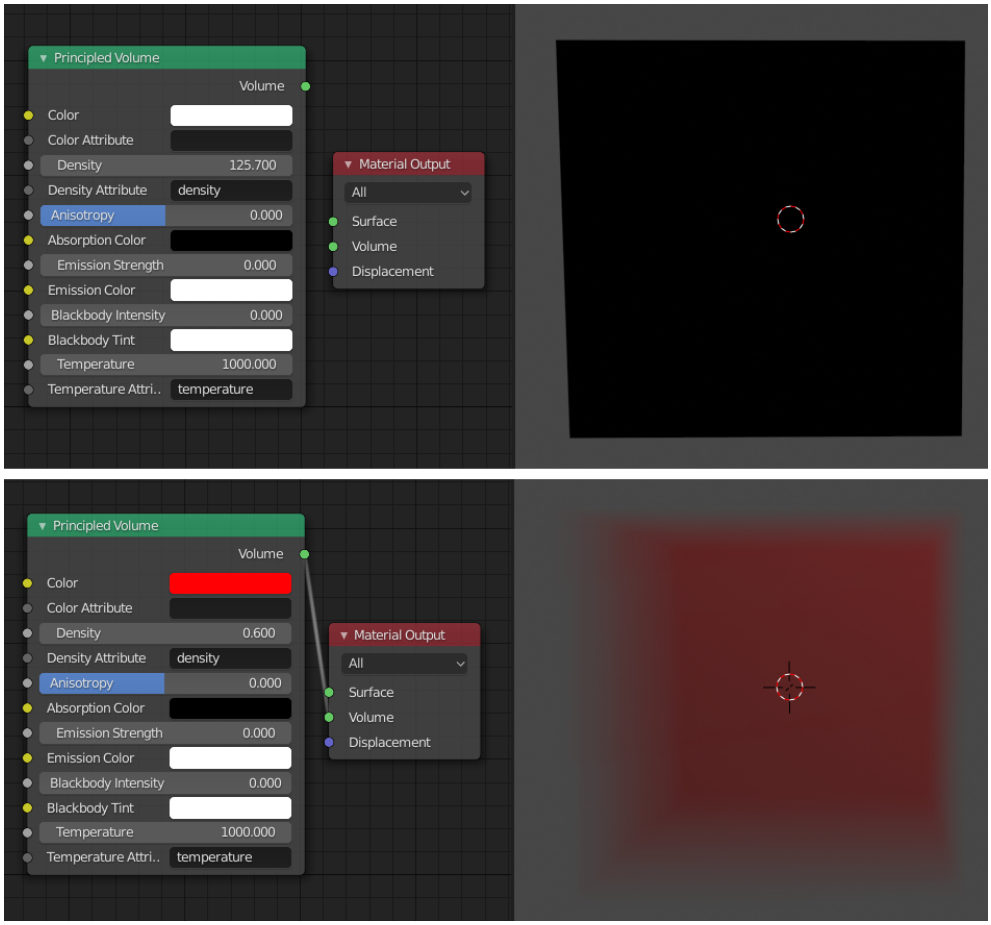


Figure 3.16: Principled Volume node in Blender

3.4.5 Output

In the same way, this division only has one single node (Fragment Color). This node will be in charge of building up and creating a formal shader and passing it to the main loop, so it will be used in the render. Therefore this is the only node that does not have any output, also, its input can only be linked with the outputs from the Shader class nodes. It can be purple-colored if it is in use or without color if it not in use (explained in 4.3).

Chapter 4

IMPLEMENTATION

As already mentioned, in order to fulfill the project we require a web application that will carry out the theory explained above. In this chapter, we will focus on the programming development of the Framework, the Nodes, and the System Control.

4.1 Framework

In order to explain the development of the framework, I decided to decompose the framework into three parts (Interface, Main Loop, and Logic of the Graph). I think that this classification will help to smooth the understanding of all the code.

4.1.1 External Libraries

First, it is necessary to present the external libraries used for the realization of this project, along with a comment about its functionality in the application.

W2ui.js (1.5 version)

It is a library under MIT License that allows the creation and control of the most common UI widgets (grid, sidebars, pop-ups, ...) [38]. One example is the use of external text as body text or the use of buttons inside the panel elements (we will see this implementation later on). It uses HTML5 and is supported in most of the modern browsers. The usage of this library was exclusively for the deployable panels of the header buttons.

JsColor.js

JsColor is a web color picker component very useful, easy to use, and good-looking [23]. It was considered that it was better for the user to select a color within a palette than write it in RGBA style.

LiteGL.js

It is a library created by Javi Agenjo (project director) in 2013, which is also a fork from LightGL.js by Evan Wallace but with quite major changes [20]. It simplifies the interaction with WebGL "without having to handle all the low-level calls but without losing any freedom". It is achieved by creating useful classes and functions. Some of its uses have been for the view canvas, for capturing the mouse events and above all the usage of the classes Buffer, Mesh, Texture, and Shader.

Litegraph.js

Also created by Javi Agenjo in 2013, this library is oriented in the graph creation. It also makes it very easy to create new nodes [21]. As supposed, its use and effectiveness were one of the most important reasons behind the realization of this project. It has been used for creating all the nodes that will be introduced later on and also is the class that manages the left canvas of the application (the editor).

Rendeer.js

Also created by Javi Agenjo in 2014, this library contains some classes with quite an amount of methods that help to control the render process (SceneNodes, Renderer, Scene, Camera, etc.) [1]. In the project, we will only use the Camera class.

Volume-base.js

Created by Miquel Clark (2019), it contains the class Volume which describes a 3D dataset. It is fully though for WebGL 2.0 since the previous version did not support tridimensional volumes [36]. It has functions to create corresponding textures for the volume in use and takes care of the format and type of the data.

Volume-loader.js

Created by Miquel Clark (2019) and complementing the previously mentioned, it allows the user to upload a 3D texture as a Volume class. It considers VL, Dicom, and Nifti files

[37]. Along with Volume-base, they were crucial when implementing the Dicom node.

Finally, some additional libraries were required due to their dependence on one of the already explained previously. **JQuery.js** (3.5.1 version) [22], dependence on W2ui, it facilitates the manipulation of HTML documents, event handling, and some other things. **GL-Matrix.js** [15], dependence on Litegl, it provides a high performance vector and matrix classes and functions. **Daikon.js** (2.0.1 version) [9], dependence on Volume-loader, it is a Dicom parser.

4.1.2 Interface

As explained in the design, the interface application is divided into 3 modules.

The header contains four buttons, three of them display a panel in front of the app created using W2ui. Since they were already presented in section 3.2.2, only mention that for the implementation of the download functionality of the "View Shader" panel, an external function created by Dan Allison [4] has been used.

For the left canvas, we first create a new class LGraph that will contain the full graph and store all the nodes. On the other side, we also need a LGraphCanvas object that renders the graph inside a canvas (which will be defined in the HTML), and it also supports the use of panels [2]. Later on, the graph object is used to run the execution loop, and in section 4.3 we explain different control measures taken to improve the performance of the application.

Lastly, the class GL is used to create a WebGL content on the other canvas defined in the HTML document. Thanks to that, it is possible to display in this canvas the render of our main event loop.

4.1.3 Main Loop

In the previous section 3.3.1, it is explained how the application initializes and starts the main loop (*onLoad()*). The main loop has four basic functions: call the execution of the graph nodes, update the time, update the scene, and render.

The *runStep()* method is in charge of calling the code from all the nodes that exist in the editor canvas at the execution time. The resulting of this process is the construction of the shader.

The next step uses the *getTime()* function to store the execution time (now), also it is stored the time in the previous iteration (last) and the delta time (dt). The dt is computed as the difference between now and last, this is done so that computers with different rendering time (time required to complete one iteration) can use this value to create movement at the same speed.

The update phase controls the position and direction of the camera in the scene depending on the user inputs, to do so we use the Mouse class (from sceneTools.js) and the Camera class (from Rendeer.js).

The mouse class stores some characteristics of the mouse like the button pressed, the position in the visualization canvas, the wheel state, and the direction of dragging. All these values are obtained from the method onmouse() (from Litegl.js) that classify the events (button, wheel, ...) into their respective listener.

The camera class contains the basic properties of a camera such as the matrices (model, view, projection, viewprojection) that indicates where it is placed and where it is aiming. By combining both classes we can control the movement of the camera depending on the input from the mouse. This movement can be orbit (by pressing any mouse button and dragging) and zoom (by using the wheel). The orbit allows the movement around the object by using the pitch and the yaw, they represent the horizontal and vertical inclination respectively. On the other side, the zoom varies the angle of vision of the camera (fov), this is the same operation as the cameras.

The final step is the render of the scene, that in this case, it is only one object. First, we set the flags for the render:

- **Depth Test:** It does not affect because we only have one entity on the scene.
- **Cull Face:** Enable, mode Back. For the moment we want to cull the back faces, later on, when the option of placing the camera inside the volume is considered, we

will cull the front faces.

– **Blend:** It depends if the final shader uses a volume rendering algorithm or not. If it does, the blend will be set to enable because we will want to mix the resulting color with the background.

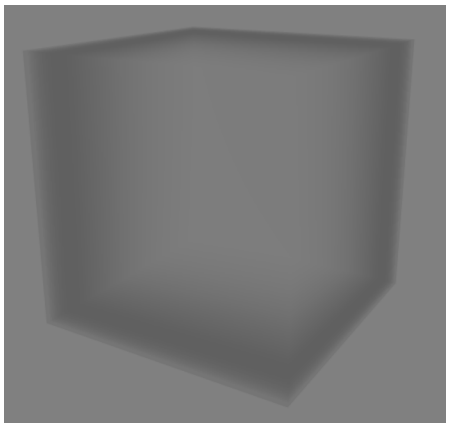


Figure 4.1: Result with Blend enabled.

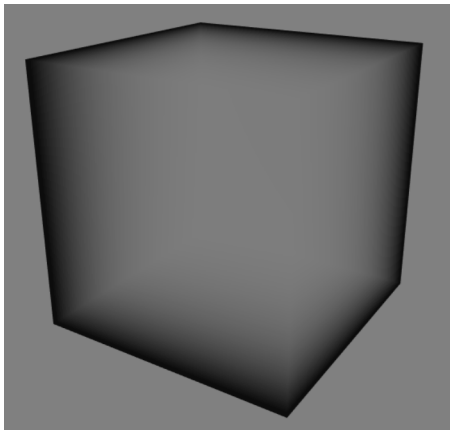


Figure 4.2: Result with Blend disabled.

Then, it is computed the local camera position, this transformation will facilitate the break condition in the shader (it will be easy to say when the ray has gone out of the object). To pass from world space to local space, we need to multiply the given coordinate with the inverse of the model of the local object. These operations are easy to implement thanks to the Gl-matrix library.

Finally, we render the entity using the *draw()* method and passing all the required uniforms to the shader.

4.1.4 Logic of the Graph

Once the graphic engine structure is explained, we can explain how the own logic of the graph object works.

The control of the graph is located on the main loop, at each iteration, the *onExecute()* method of all the existing nodes placed on the editor is called. Since it is important to know if there have been any changes in their properties, inputs, or outputs, we need to check the nodes every time. Regardless, I decided to implement a series of checks at

each node to test if it is necessary to execute it or not. All of these control measures are explained in section 4.3.

The loop executes the nodes in a specific order (concreted by the LiteGraph library), where the first nodes are the ones that do not have inputs. Once it arrives at the Output node, the flow will have gone through all the system and therefore the input values received would have been correctly created to be implemented in their location of the skeleton code.

Apart from that, each node contains both its required uniforms and functions as lines of code (in strings). The Output node select which are the nodes that affect the final pipeline and constructs a variable for the uniforms and the auxiliary functions that will also be included in the build of the final shader.

4.2 Nodes programming

The development of the nodes was done with two important considerations in mind. They must fit in a Volume Rendering pipeline, they must generate a final shader that can be exported and be useful for the user.

Nodes are Javascript objects with properties and functions that can be private or public for the others. Its main methods are the constructor which defines the inputs, outputs and properties of the node, and the *onExecute()* which is the method that defines the behavior of the node and is compiled at each loop iteration. It is also important to remark the *pixel_shader* and *uniforms* properties that characterize each node and contains the code and uniforms respectively that it brings to the shader. In order to add them to the Litegraph editor, we need to register them to the LiteGraph class using the *registerNodeType(name, base_class)* method.

We will now take a deeper look by going one by one at their contribution:

4.2.1 Input Nodes

Number

Number is a very basic node that provides a number. As said in the design chapter, default values are contemplated, but it may be useful to extract the functionality of selecting and controlling a value and using it in different aspects (as a density, as a factor for an interpolation, ...). Its only property is the value, which can be modified by typing on the panel or by interacting with the widget.

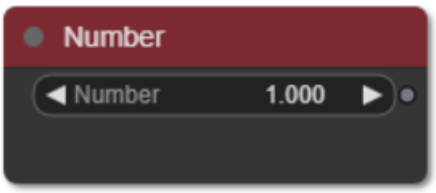


Figure 4.3: Node Number

When the output is linked, it passes to the following node a simple number. Since we want all the modifications to be seen in the shader we could think that it must send a string instead of a number, but I decided to keep this node pretty simple and give the other nodes the responsibility of converting the value into a string.

Color

Pretty much the same as the Number node, it provides a color value to the continuous nodes. The value property can be modified in the personal panel of properties by using the Jscolor library. The data sent through the node is an Array of 4 values (RGBA). However, the alpha of the color will not change since the transparency is determined by the density, not the color. The color in use is represented visually by coloring the node with it value.

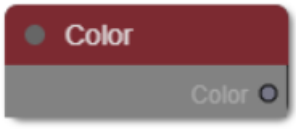


Figure 4.4: Node Color

Coordinates

As the name says, this node allows the user to select from a list of different coordinates. It

only returns a string of the selected coordinates which will be treated as a vector3. Taking Blender as example, the outputs coordinates considered are the followings:

- **Generated:** It corresponds to the ray position, therefore it will go from minus half of the size of the mesh to half of the size of the mesh (-1 to 1 if the cube size is 2).
- **Normal:** It corresponds to the normals of the mesh, 1 up, front and right, -1 its opposites.
- **UV:** It corresponds to the screen coordinates, they go from (0.0) on the top left corner to (1.1) on the bottom right corner.
- **Object** It represents the vertex position (we are working in local), therefore they go from -1 to 1.
- **Camera:** Position coordinates in camera space, therefore it will be 0 on left side and 1 on the right side.

Figure 4.5: Node Coordinates

Transfer Function

Just like the Dicom node affects directly the density of the volume, the Transfer Function node does it for the color. I took an existing node "Curves" form Litegraph library and did some changes so it fits correctly in the graph pipeline. It has a selector that gives the option to choose between the edition of all the channels together or separately. Then, the form of the curves is read and converted into an array that will be used to create the texture. In the shader, it uses the density value to find the respective color value within the texture, therefore the UV is a vector2 formed by the density clamped between 0 and 1, and 0.5 because the texture is flat (1 in the Y dimension) *"texture(u_tf, vec2(clamp(v, 0.0, 1.0),0.5))"*.

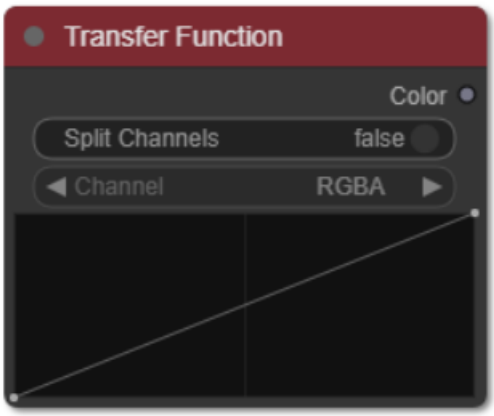


Figure 4.6: Node Transfer Function

4.2.2 Texture Nodes

Gradient

Gradient is a texture type node that modifies the input vector by passing through a gradient function, which makes it go from 0 to 1. Additionally, in order to make it more customizable and rich, it possesses a widget which enables the selection of the type of gradient (Figure 4.7):

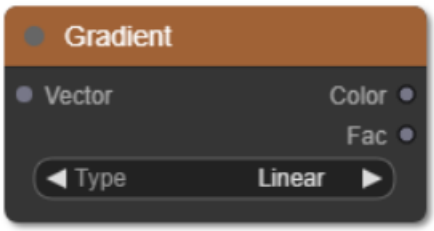


Figure 4.7: Node Gradient

- **Linear:** Simple clamp the vector between 0 and 1.
- **Quadratic:** Similar to the linear, but multiplying by itself (taking care that the value is not negative, because -1x-1 would make 1).
- **Diagonal:** To achieve the diagonal we need to sum two coordinates and then half its result, also clamp the result like in the other cases.
- **Spherical:** This is done by evaluating the points using the equation of the surface of a sphere $x^2 + y^2 + z^2 = 1$, the results lower than 1 will mean that the point is

inside. Therefore the results from $1 - (x^2 + y^2 + z^2)$ that are greater than 0, will take place inside the sphere.

Noise

This node performs a Value Noise at a given vector3. Regardless of considering and testing different 3D Perlin Noise approximation, all of them carried several computational costs that the application could not handle, and consequently, the frames per second (fps) went down to 5. For this reason, considering that we can call real-time interaction when having around 30 fps, I decided to move to a cheaper approximation presented by Inigo Quilez [30]. He has some demos on ShaderToy using this algorithm and it gives him very good results [32]. Basically the Value Noise assumes a lattice of random points, therefore by given coordinates, three random values are obtained in the field. Then, the result is the trilinear interpolation of these numbers.

The input of the node is the coordinates used to find random values and the outputs are a 1D noise (factor) and a 3D noise adding a 1.0 in the fourth component (color). For the properties, I decided to implement the scale and detail to the algorithm (as Blender does with its Noise Node). I also added a simple movement flag that translates the position of the coordinate to give the effect as it if the noise is moving (like clouds or smoke).

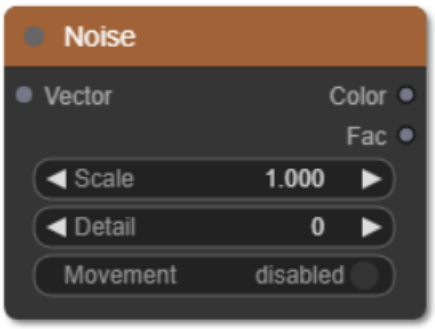


Figure 4.8: Node Noise

By understanding the meaning of each property and helping of the source code of Blender, it can be seen that scale represents “the magnificence of the vector3 evaluated” or also “the distance where the map is being shown”, for that reason when increasing the scale, the current map stretches and the final result is an expansion of the base octave.

Figure 4.9: Noise with scale = 1

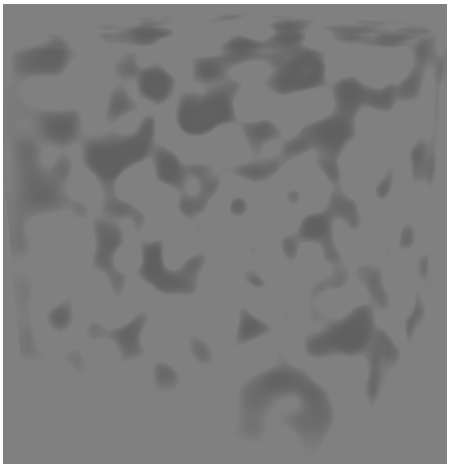


Figure 4.10: Noise with scale = 4

Octaves are related to precision, and applied to simple noise functions creates the effect of fractal noise. Octaves in noise can be understood as “how much you want to detail the continuity of the noise”, the bases behind this theoretical approach are the addition of different noise functions with varying frequencies and amplitudes (where the frequencies are the sampling period and amplitudes are the result range).

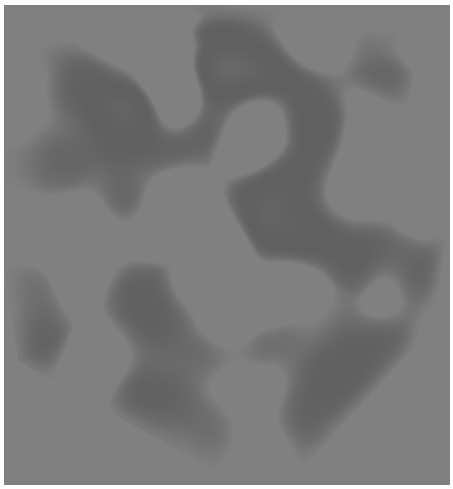


Figure 4.11: Noise with detail = 0

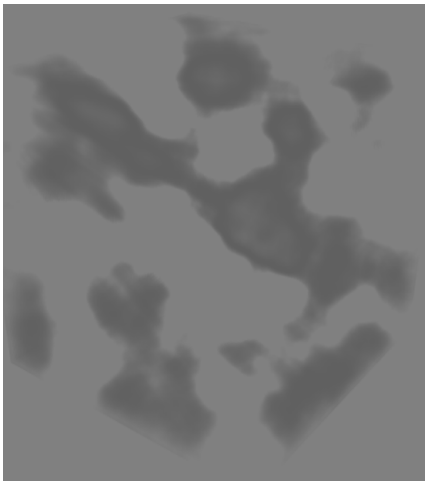


Figure 4.12: Noise with detail = 4

Finally, the distortion represents the “deformations in the appearance of the basic result of the noise function” (see Figure 4.13). As we can assume, it is achieved by repeating the same function using similar but different points. Since the cost of all the noise implementation was already high and the contribution of the distortion is not so important than scale or detail, I decided to not consider it in my noise equation.

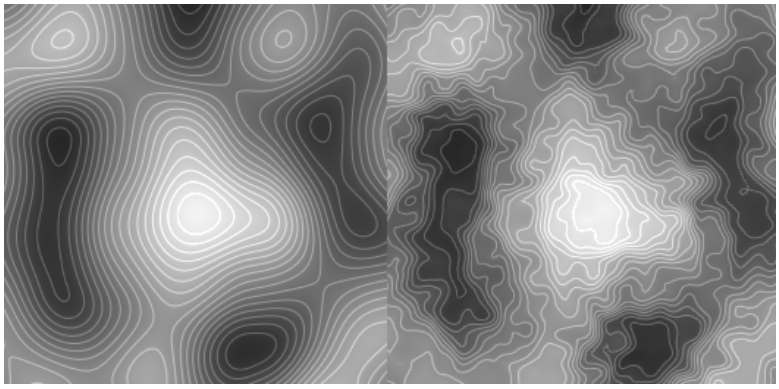


Figure 4.13: Effect of the distortion taken from [10]

Dicom

The inclusion of the 3D datasets on the pipeline is evidently made through nodes. Dicom is also an Input node type which has a property called volume, it will store the volume data once it is uploaded, it also stores the corresponding texture that will be sent to the shader.

In the support panel, there is a button along with a state of the node, by default is “Empty” and it will change to “Loaded!” when the loading process is done (Figures 4.14 and 4.15). The onChange() callback function from the button calls handleInput() which checks that there exist some files to read and call the next method loadDicomFiles(). As commented in previous chapters, this is done by using Volume-base and Volume-loader libraries, but in my case, I restricted to only load Dicom files. Once the files are loaded, we call the method createTexture() and store the result in the texture property of the node. This state can be perceived by the user because both the state and the color of the node have changed.

Since the contribution of the nodes may imply several code lines, I had to create a function for each node that wraps all their functionality. The method is sent to the shader, and the line added to the main loop consists of the call to this function. In the case of the Dicom node, the function is called getVoxel(vec3) and within it, we do a voxel interpolation to improve the result, and then we read the value from the texture [31].

Additionally to the complete Dicom visualization, a simple technique to improve the result when reducing the number of samples used (quality) is implemented. This is the jittering, which hides wood-grain artifacts by adding small offsets to the sampling posi-

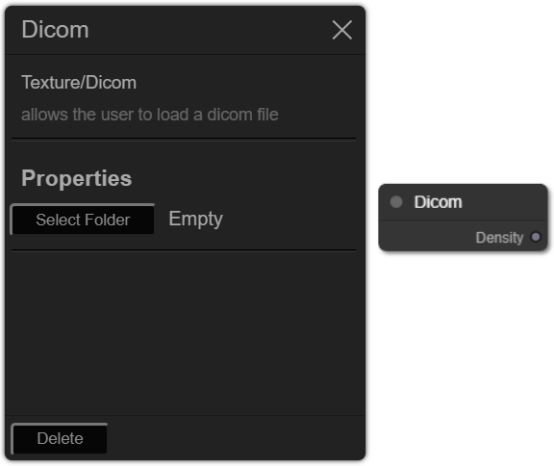


Figure 4.14: Node Dicom empty

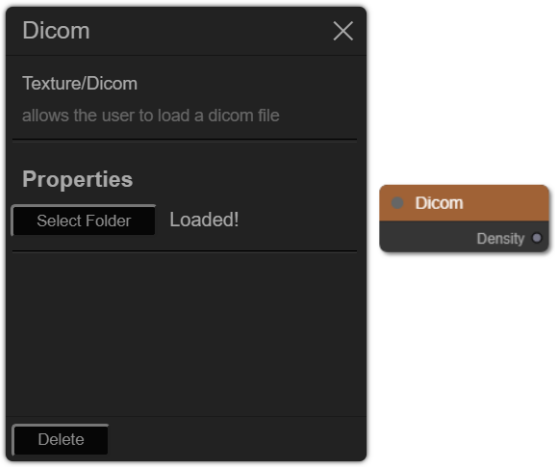


Figure 4.15: Node Dicom loaded

tions of rays in the viewing direction. To do so, I use a pseudo-random function in the shader [16]. As can be seen in the following figures there is a great improvement.

Figure 4.16: Quality = 30, no jittering

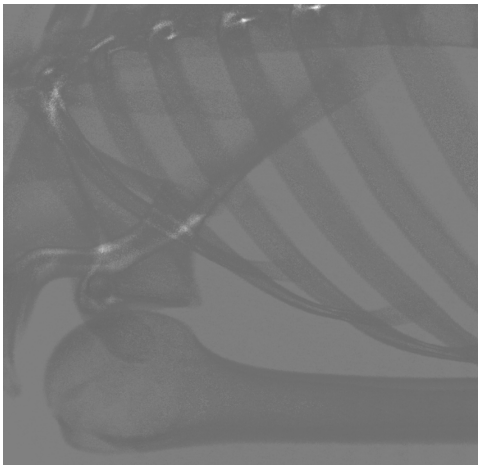


Figure 4.17: Quality = 30, with jittering

Furthermore, it is necessary to take into account that the loaded datasets may have different forms than a cube (our default entity). For that reason, when using a Dicom it is important to scale the model of the mesh with the dimension properties of the dataset.

4.2.3 Operator Nodes

Math

This node is taken from the Litegraph library, there exists a quite amount of existing nodes, but for this project, this is the only one I considered really useful. As the name

says, it allows the operation between input values, furthermore, it offers a group of several mathematical operations. Since this is not considered an Input node but an Operator node, it takes care of receiving a string value and create the code line of the operation desired and passed it through the output.

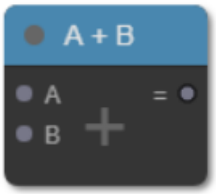


Figure 4.18: Node Math

MixRGB

MixRGB is an Operator class node that interpolates the colors it receives as input using a factor. It does not need any more explanation because the operation is straightforward the mix() function of OpenGL.

Figure 4.19: Node MixRGB

ColorRamp

This node discriminates the input between two frontier values. If the input is higher than 0.5, it gets clamped between the max and 1.0, otherwise, if the input is less than 0.5, it gets clamped between 0.0 and min (Figure 4.20). I considered the utility of this node for the cases when the user wants to “hide” middle values, in other words when it is needed to remark huge discrimination between values. A real case could be to reduce the data of the middle values in a noise volume which does not give very important information.

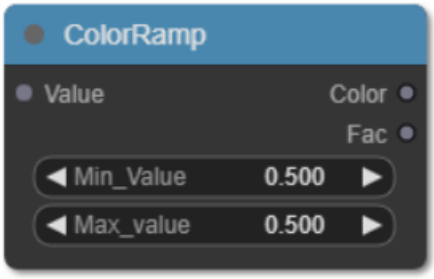


Figure 4.20: Node ColorRamp

Translate / Scale / Rotate

As the name indicates, these are nodes that perform translation/scaling/rotation respectively to the vector selected. The default values are 0 for Translate and Rotate and 1 for Scale and the Rotate properties are read as degrees. Their value limits also are considered, being from -360 to 360 for Rotate, 0 to 100 for Scale and -100 to 100 for Translate (they are limited to 100 since higher values will not make any difference, besides that it is improbable that someone needs to reach those values).

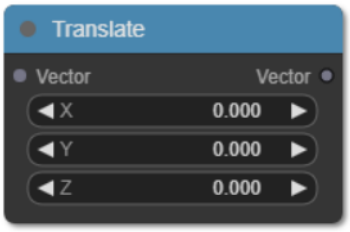


Figure 4.21: Node Translate

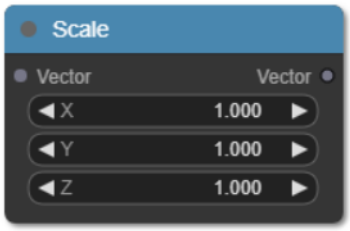


Figure 4.22: Node Scale

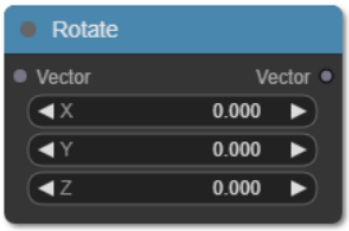


Figure 4.23: Node Rotate

Its functions are basic for the Translate (add to the vector) and the Scale (multiply to the vector), on the other hand, to apply rotation in the euclidean space we have to use the 3D rotation matrices.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\operatorname{sen}\theta \\ 0 & \operatorname{sen}\theta & \cos\theta \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \operatorname{sen}\theta \\ 0 & 1 & 0 \\ -\operatorname{sen}\theta & 0 & \cos\theta \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & -\operatorname{sen}\theta & 0 \\ \operatorname{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 4.24: 3D rotation matrices

4.2.4 Shader Nodes

Volume

This is the only node of the class Shader, it possesses the algorithm broken-down and prepared to receive different data that will change the final result. As the inputs indicate, I considered that the color and the density are the main properties that give the most personalizable result. Since the output is the algorithm code, it can only be linked by the output with the Output node.

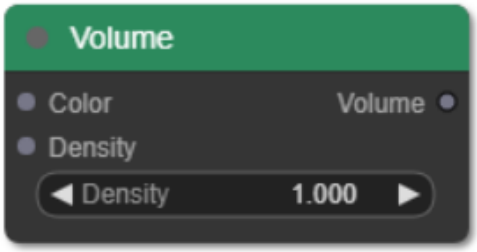


Figure 4.25: Node Volume

Even though we explained the importance of the light equation and the different models, there exist physical limitations that do not let us compute analytically an integral, therefore a numeric approximation is needed. We can consider the light integral as a sum of samples of the volume along the ray, and the exponential term that explains the attenuation of the light when returning to the eye, which is an exponential of a summation, can be approximated as a product of sequence [35].

These transformations can be seen in the following equations. On one hand, in Figure 4.26 we integrate along s (the volume dimension), $C(s)$ is the color emitted at each point, and $\mu(s)$ is the absorption at each point. On the other hand, in Figure 4.27 we use a summation along i, Δs is the distance between steps, and the exponential (attenuated) is transformed using the Taylor series.

$$C(r) = \int_0^L C(s)\mu(s)e^{-\int_0^s \mu(t)dt}ds$$

Figure 4.26: Light equation

$$C(r) = \sum_{i=0}^{N} C(i\Delta s)\alpha(i\Delta s)\prod_{j=0}^{i-1}(1-\alpha(j\Delta s))$$

Figure 4.27: Transformation of the light equation

With this result, we can already tell that we need to use a ray marching algorithm. The exit conditions will be when the alpha of the accumulated color is 1 (that means we cannot see any further) or when the ray exits the volume. So the pseudocode is the following:

Algorithm 1: Ray marching through the volume

```
Result: Final color value.
Initialize the ray position and direction;
for i ← 0 to 100000 do
    Get the density value in the ray position;
    Get the color value in the ray position;
    Apply the density to the alpha channel of the color;
    Apply the transparency to the rgb values of the color;
    Accumulate this sample color to a final color value;
    if alpha is equal or greater than 1 then
        Break the loop;
    end
    Compute the next position of the ray;
    if sample position is out of the bounds of the mesh then
        Break the loop;
    end
end
```

4.2.5 Output

Finally, the Output node (called Material Output) only takes one input that is the code that defines how the pixel is colored. It builds up the final shader using the basic structure defined and creates a Shader instance which will be the one used in the render. Inside this node also takes place some control tasks that we will explain in the next section.

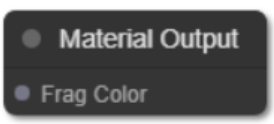


Figure 4.28: Node Output unused

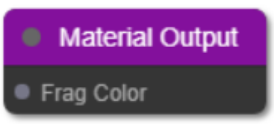


Figure 4.29: Node Output

4.3 System control

As we just introduced, the application has implemented different control tasks in order to improve the efficiency of the render and the whole system. In this section, we will enum the problems and the decisions taken to control them.

- Since the output creates the shader instance, if the user (for some reason) decides to create more than one output node, this process will be repeated for each additional node. As we can see this is nonsense because only one shader (the one connected to the whole graph) will be necessary to use. Therefore, at the beginning of the execution of the output node code, we check that no other outputs have been executed in this iteration. This can only be seen visually in the editor, where the node with color indicates that it is the one that will be used (Figure X).
- Another idea to reduce code lines to compile is to skip the nodes that are not linked to the final shader. To achieve this I created two new functions that check if a node is connected with another, one of them makes the check backward and the other forwards, hasConnection(), and isConnected() respectively. So the method hasConnection() is called at the start of the execution of each node that is not an output type.
- Due to the cost of the shader instance creation at each frame, I considered necessary to save the previous shader in a global variable and make a check at each loop that the new shader is not the same, if so there is no need to create a new instance.
- The output node builds the final shader using "pieces" of a basic template, but if the input is not linked, that means that there is no reason to construct a volume shader, but use a basic shader that colors the mesh uniformly using a property color.
- One of the Volume Render algorithm exit conditions takes into account the size of the mesh, for that reason is important to automatize the value. Even though it can sound logic, it is quite hard to see because if the volume size is set to 1 and centered to 0, that means that the mesh has 0.5 at each side, therefore the exit condition must check if the value is out of [-0.5, 0.5].

- It can seem trivial, but even if the Output node has no connections, the render still needs a shader to complete the main loop process. If this is not controlled, it will use always the last modification of the shader, which would not make any sense to visualize a volumetric material if the output does not receive any shader node. Therefore, there must be a default shader to be used when this happens. To be more precise, I tried to recreate what more applications and game do when a material has no texture, that is to set it pink-colored. (However, the color value of the mesh can be modified in the options menu).

Chapter 5

RESULTS & CONCLUSIONS

In this last chapter, we will see the results of this project by comparing the final product with a hypothetical user case pipeline. Then, we will check the fidelity to the requirements presented in the design. Followed by an extension of a highly important quality aspect, the performance, and then an introduction to the future work. Finally, I will give my final conclusions about this project.

5.1 Use Cases

It is interesting to test how it would be the experience of a user that requires of the usage of a shader editor for volume rendering. This will be done by comparing the needs with the available tools. As explained in section 2.2, we classified the cases in Medical and Artistic. I attached an anonymized dataset along with the code (5.5), so any user can try the medical case for themselves.

In both cases, the user would start with the same view, with the node Volume connected to a node Output (Figure 5.1). Normally, in some editors, it would only be the Output node by default, but since the only Shader node that can affect it is Volume, I considered to better be created by default to speed up the start.

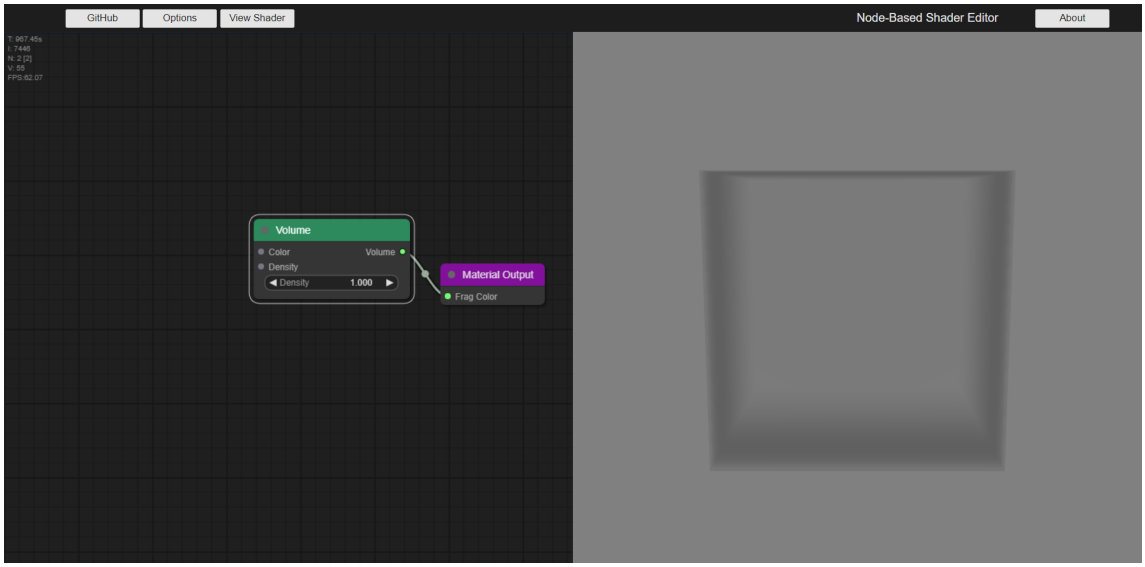


Figure 5.1: Default view for a new user

5.1.1 Medical Usage

In this case, the main task for the user is to load the dataset. So the Dicom node would be created, and the user would load an own dataset. The UI is perfectly controlled so there are no problems with respect to the possible connections of the output, and neither with the loading process.

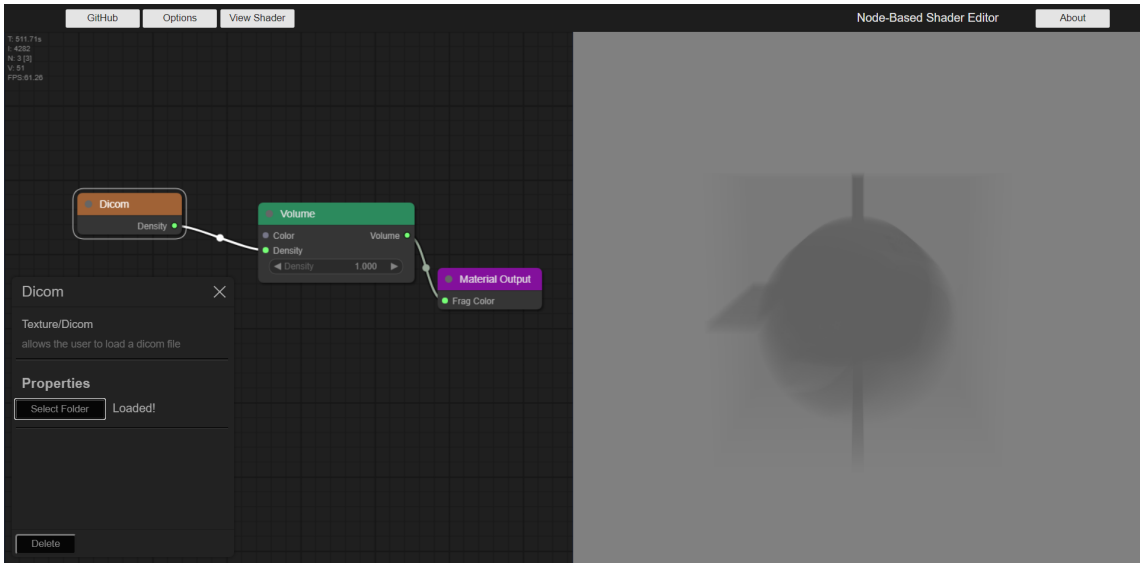


Figure 5.2: Example using a Dicom loaded

Now, the user would like to control the visibility of the voxels because the visualization is not showing exactly what the user is expecting, in order to do that the Transfer Function

node needs to be created. The widget of the node is easy to understand and use, so it is fast to get the result from the Figure 5.3.

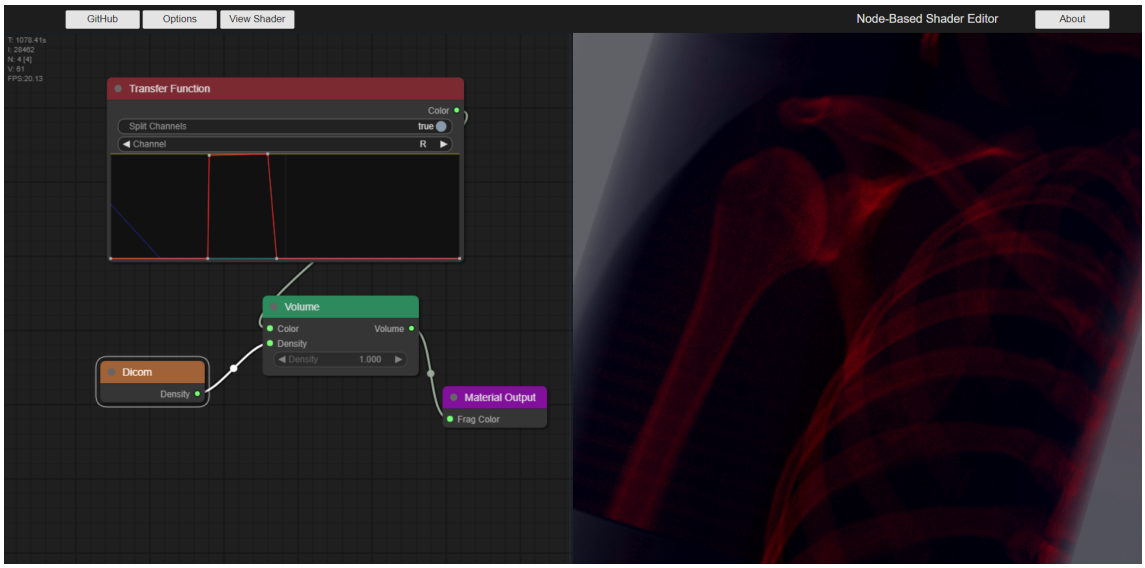


Figure 5.3: Example using a Dicom with a TF

Some other important possible features that this type of user would like to have available are the clipping of the volume, and templates of transfer functions to use (standard templates for specific cases of visualization), unfortunately, the project does not consider that yet. Nevertheless, a simple clipping effect can be modeled with a little ingenuity using correctly the available nodes.

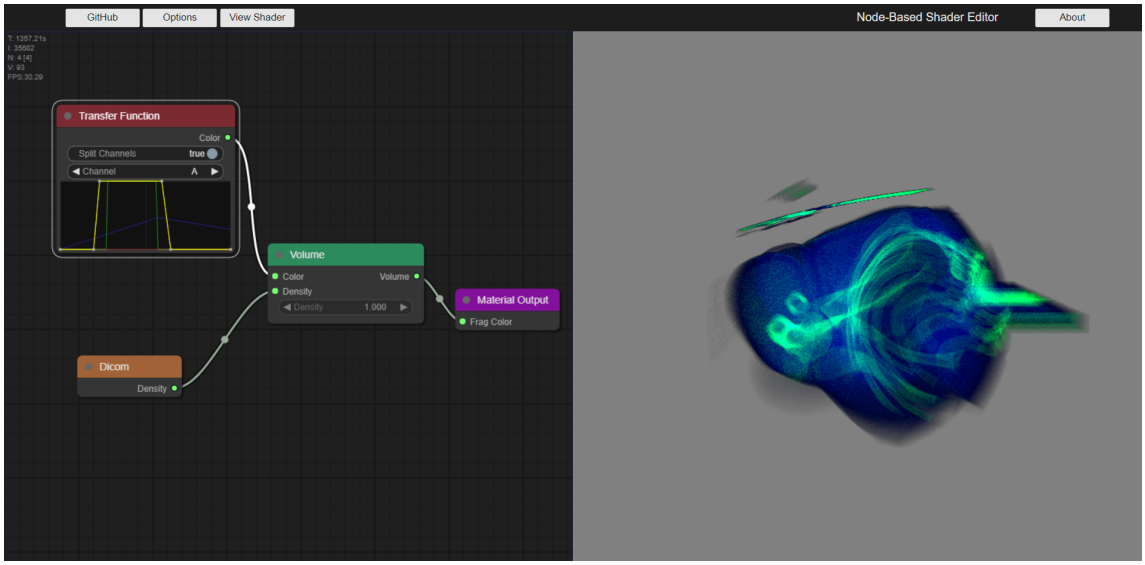


Figure 5.4: Example with a part that we need to hide

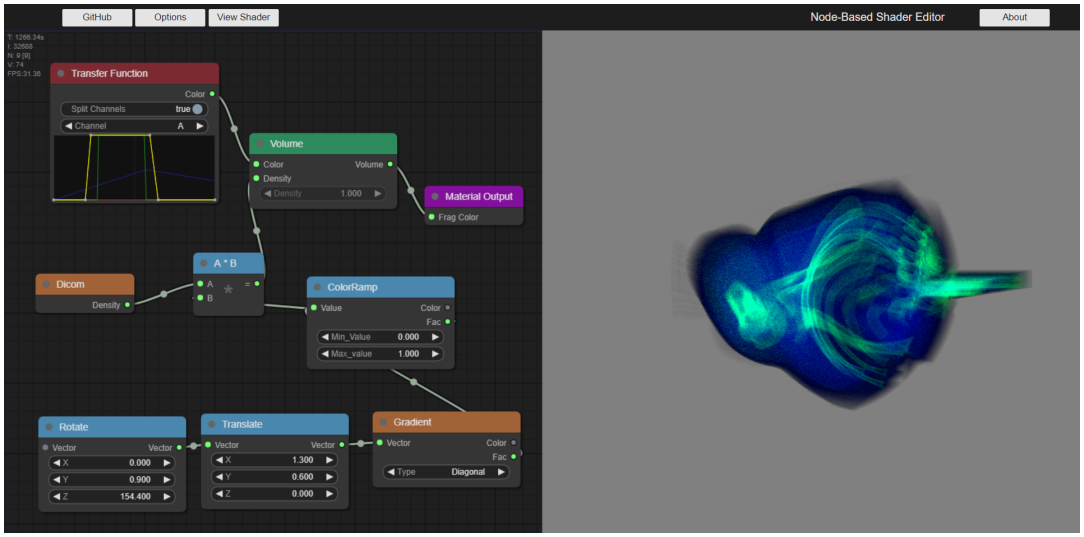


Figure 5.5: Example creating a clipping effect with nodes

This result is obtained using the Gradient node rotated and translated to the desired position, it also applies the ColorRamp node to enhance the discrimination of middle values.

5.1.2 Shading Usage

On the other side, during this report we evaluated that the applications of volume rendering are mostly to visualize clouds or smoke, for that reason, the Noise node is required in order to get this effect. Additionally, the usage of nodes of type Math like the vector transformations (Translate or Rotate) is useful to personalize the noise behavior.

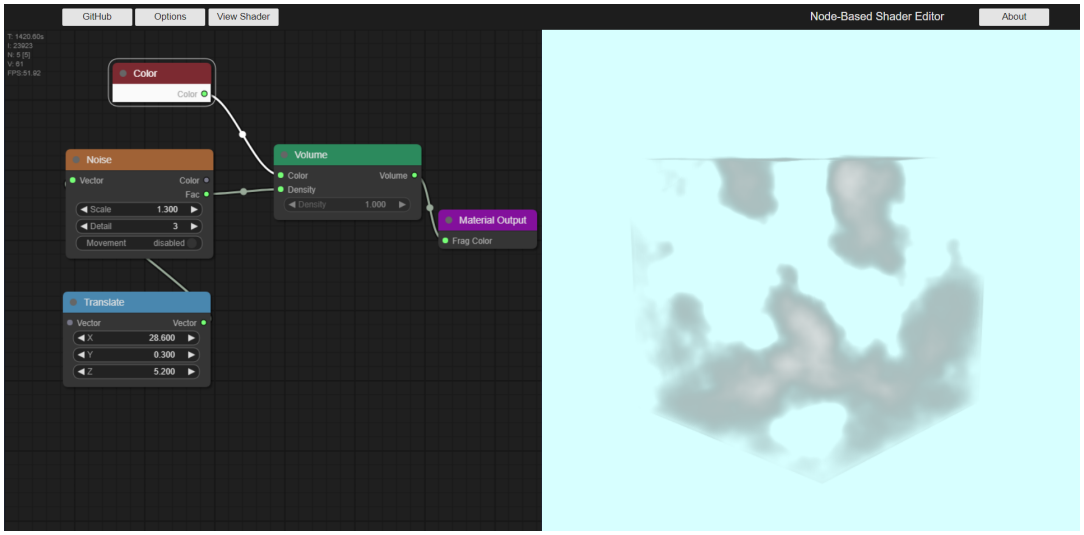


Figure 5.6: Example that simulates clouds

By changing the background color so it looks like sky and setting the volume color as white, the user can easily create a nice cloud effect (Figure 5.6). In these cases, normally the node Gradient is used because it can be mixed with the noise, and therefore specify at which zones do you want the effect (gradient goes from 0 to 1, so the areas near to 0 will disappear).

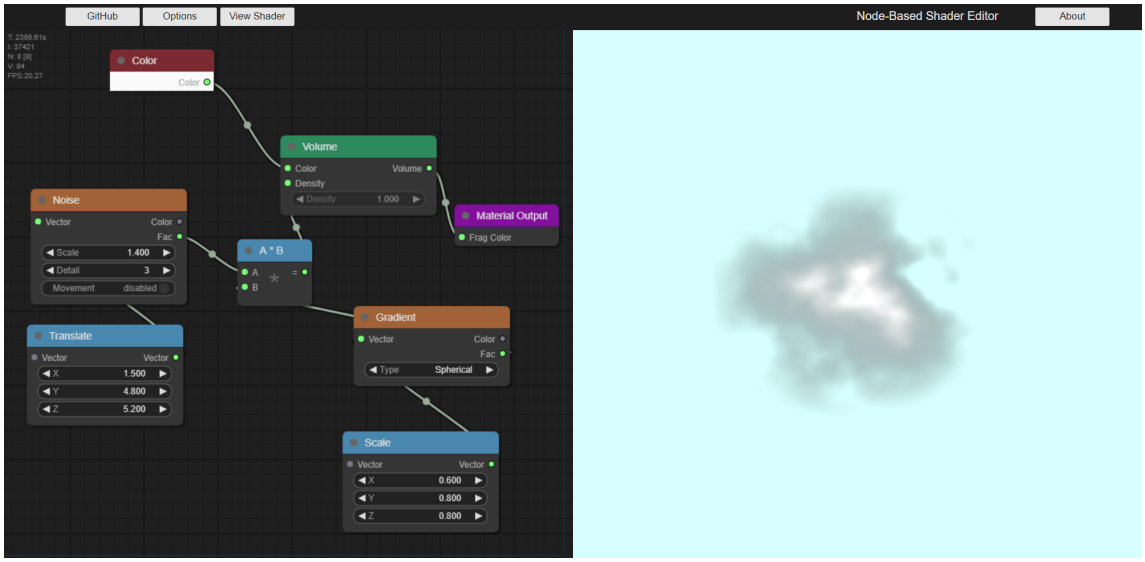


Figure 5.7: Example that simulates a single cloud

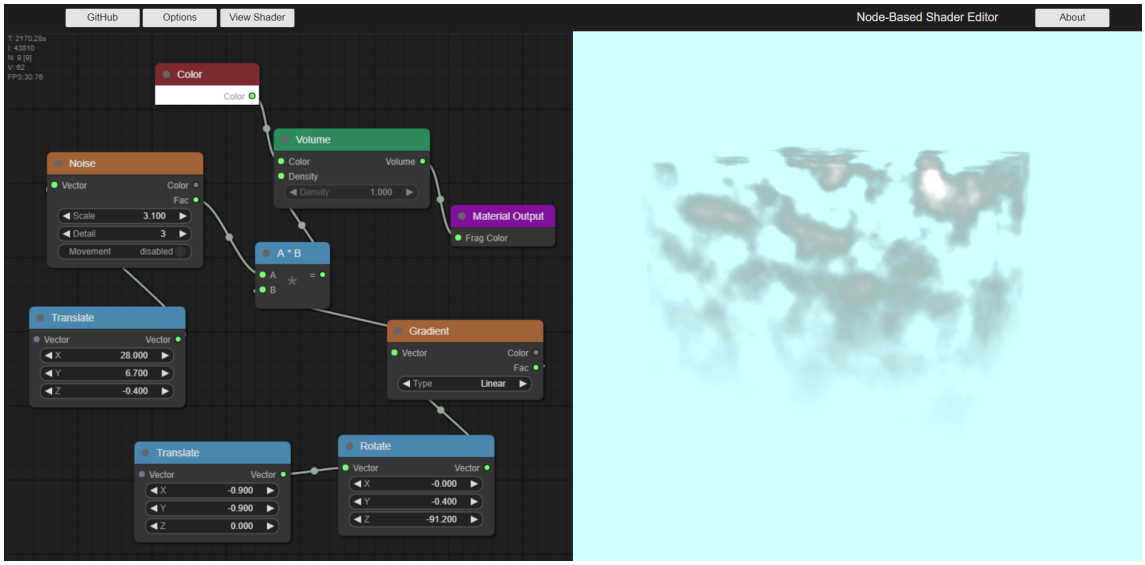


Figure 5.8: Example that simulates fog

Figures 5.7 and 5.8 represent two different possible effects that the user can get for this application case. The first one defines a lonely cloud, which does not collapse with the borders of the mesh, so it gives a more realistic result of a cloud. The second image

simulates fog inside the object, this makes the clouds denser while going up. The effects do not look bad and could fit correctly in a scenario with modeled terrain.

Moreover, other effects can be obtained by combining some of the nodes. For example, in the following figure, it is simulated the effect of a lava lamp. But I do not want to count it as case result because I did not find any information that backs up these results.

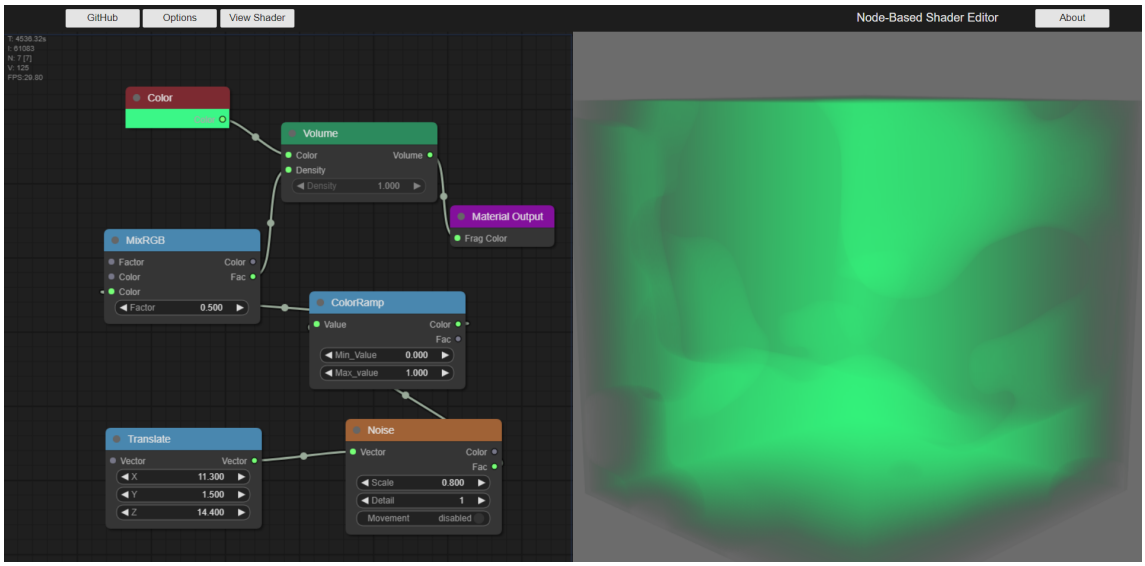


Figure 5.9: Example that simulates a lava lamp

However, I am aware that some algorithms increase the computational cost of the render. For example, the result images show a frame rate between 20-30 fps (this test was made using the integrate graphic card, discussed in 5.3.1), and if I would have got closer to the mesh it would have decreased more. Most of this cost comes from the noise algorithm, but even though I only explained the one I decided to use, I tried different approaches and all of them gave me worse performance results. Regardless, there must be or a better approximation or some improvement techniques that could have been implemented to reduce this cost. Furthermore, another interesting usage would be the volumetric light scattering, unfortunately, light is yet not supported in the framework.

5.2 Evaluation

To evaluate the application, I will take into account the requirements presented in 3.1. Therefore I have classified the evaluation in five main categories: *Compatibility*, *Functional*, *Accessibility*, *Quality*, *Robustness*.

- The Compatibility indicates where the application runs and where it can not. I checked to be working correctly in Chrome, Firefox, Opera; and it does not work in Safari and Internet Explorer. Related with the operating systems, it works correctly both in Mac and Windows.
- The block *Functional* includes all the initial requirements that I considered when designing the framework. As we can see, the application meets all the functionalities specified at the beginning (section 3.1.1).
- I decided to wrap in *Accessibility* both how intuitive it is for the users to manipulate the editor, and how easy it is to decompose the framework.

  For the first part, I asked some people to use the editor for some time and I found that they had some difficulties when connecting nodes with others, especially with the Math node, which does not specify the inputs and output types and currently does not support color operations, only values. That indicates that more compatibility between nodes has to be improved in the editor.

  The second part makes reference at the case that I or someone needs to reuse only one functionality of the application, this would be the case if I have to take the nodes to use them otherwhere. In that case, the framework is structured so the file main.js contains the initialization of the app and the main loop, sceneTools.js contains two classes used in the implementation, utils.js contains some functions that I required its usage at some point, and finally, volumeNodes.js contains the creation and register of all the nodes created in this project. So we can say that it is accessible.
- By *Quality* I mean how good are the results obtained compared with other editors, or compared with the initial ideas. In this case, I already presented some of my

opinions in the last chapter, and that is that the frame rate is lower than desire when using computational expensive techniques. The application has the option of changing the quality which reduces the cost, but still, I think that (in standards situations) it should not drop lower than 30 fps. The explanation of this category is more developed in 5.3.

It is hard to compare with other editors, since not most of them supports volume rendering. But if we use Blender as reference (with the Eevee engine), it always runs fast. I also think that Blender sacrifices part of the visual quality to give more power in real-time (obviously without taking into account the difference of effects that Blender has implemented), this can be seen in Figure 5.10.

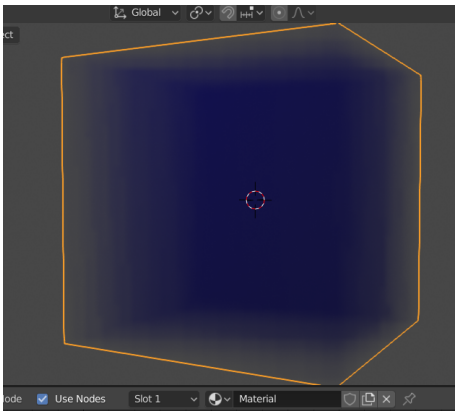


Figure 5.10: Default visualization in Blender

- Lastly, the *Robustness* indicates the quantity of bugs and errors the program can give and how much protection has towards that. This is a very challenging thing when working with a node editor because all the possible connections between nodes must be considered.

  In this project, I am aware that there exist some bugs that I could not fix in time. The three most important are, that I do not check how is the data uploaded in the Dicom node (I assume that the user will pass the right data, but that can be false), that a single input can not be linked to two different outputs (well, it can be, but it will not send anything), and when increasing the zoom too much. Regardless, I think I considered all the other possible bugs and prevented them.

5.3 Performance

This section summaries the measurement of the performance of the final application. Where an acceptable performance level is accomplished, and where the quality of the application is under the expected results. We will not mention the aspects presented in 4.3 to improve the performance, only the results obtained.

To do so, I will use the fps counter that provides the Litegraph canvas, which is computed by dividing 1 second by the time between renders (code used in Figure 5.11).

```
//fps counting
var now = LiteGraph.getTime();
this.render_time = (now - this.last_draw_time) * 0.001;
this.last_draw_time = now;

//more code here//

this.fps = this.render_time ? 1.0 / this.render_time : 0;
this.frame += 1;
```

Figure 5.11: Javascript code to calculate the fps

Since the objective of the project was to make a real-time application, we need to consider that real-time graphics systems must render each image in less than $1/30th$ of a second, which means that the frame rate must be at least 30 fps. Because it is fully related to the GPU of the computer, I must indicate that the graphic card used for this evaluation is the NVIDIA GeForce GTX 950M.

Even though the default graph when executing the web app uses expensive techniques like the Noise node, we manage to obtain a frame rate of 60 fps. Nevertheless, the addition of new Noise nodes to the pipeline reduces it considerably, to half of it (30 fps). The worst situation appears when the object of the scene occupies the whole frame, it requires the ray-marching algorithm for each pixel of the frame and, consequently, the reduction of the frame rate to 10-20 fps, which is less than the real-time experience needs.

To solve this problem, there is an option of changing the quality of the algorithm (the distance of the step). By reducing it to 10, we get always a minimum of 30 fps.

5.3.1 Using the Integrated Graphics Card

Because the application is run on the web, that means that the user may have one or more graphics cards (the integrated and the dedicated). For this reason, I will present the same points from the last section but using my laptop integrated graphic card (Intel(R) HD Graphics 530) which is considerably worse.

The default scenario that gave us 60 fps in 5.3, now it is reduced to 40 approximately. And consequently to 20 when using more than one Noise node (the most computational expensive node). Moreover, when we zoom in and the ray-marching is used for all the pixels in the frame the obtained frame rate is around 5 fps, and 15 fps if we reduce the quality to 10.

In conclusion, despite the performance obtained using the dedicated graphic card is not that bad (we manage to keep 30 fps sacrificing a bit of quality), we cannot say that the application meets the real-time objectives in all scenarios. Because as we have seen with the integrated graphic card, there will be cases where a good performance is not achieved and therefore we will not be giving a good experience to the user. This gains more importance, if we know that some browsers use the integrated graphics card by default, therefore even if the user has a better dedicated graphic card, it is not going to be used unless the user changes it manually.

5.4 Future Work

In this section, I will present a few implementations that would be interesting to add to the project since they would make it look better.

1. **Light Interaction:** By implementing this functionality, we would be able to create the volumetric scatter effect. This would also mean creating lights affecting the scene, specifically spotlights (because they are better to visualize this effect).
2. **Volume Rendering on Scene:** By now the framework only considers the volume render applied as a material for the entity. But it would be interesting to develop

a more complex scene, where each entity has its own material, and even the scene can have the volume rendering effect (like the volumetric scatter).

3. **Implementation in WebGLStudio:** It was one of the initial ideas for this project, but in the end, I decided to let it out of the scope of the thesis. This would also imply that not only the shader but other features like the Dicom usage would need to be remade (because of the change from WebGL 2 to WebGL 1).

4. **Increase the number of nodes:** The creation of new nodes would improve the editor and its possibilities of usage. For example, Blender has a higher number of nodes that allows the user to personalize even more its interaction. At this point, it would also include the ideas taken in 5.1.

As of the delivery of this report, all the new implementations, bugs fixed, or new functionalities will be documented in the repository of GitHub (5.5).

5.5 Conclusions

During the realization of this project, I could realize the complexity of creating a node graph editor. This is not because there are many difficult algorithms, but because of all the considerations that the developer has to take into account. The whole system must work as one, and in order to create a rich editor, you must have knowledge of a lot of different topics (for all the existing nodes).

More specifically, in my case, I spend a lot of time searching for features that are far away from my realistic scope. Both for trying to recreate top quality applications as Blender, and for investigating deep in the medicine field, where a node-based shader editor may not be a high-priority functionality.

Nevertheless, thanks to all of that, I could improve my vision when organizing projects and facing problems. And I also had the opportunity to learn from the field of Computer Graphics, which was one of my motivations.

Even though the final framework may not be a very professional editor, it does accomplish with several of the expectations I put on it. Moreover, it was my very first time programming in web languages and using new libraries for creating a graphic engine from zero.

Annexes

In this last section are attached all the complementary tool that supports the report and completes the project.

- Online Web Application: Web page of the Node-Based Shader Editor.
`https://victorubieto.github.io/graph_system/`

- Code Repository: Web page where can be found the source code.
`https://github.com/victorubieto/graph_system`

Bibliography

[1] Javi Agenjo. jagenjo/rendeer.js: Light-weight 3D Scene graph library with renderer in WebGL. https://github.com/jagenjo/rendeer.js. Last access: 2020-06-16.

[2] Javi Agenjo. litegraph.js/litegraph-editor.js at master jagenjo/litegraph.js. https://https://github.com/jagenjo/litegraph.js/blob/master/src/litegraph-editor.js#L118, 2014. Last access: 2020-06-16.

[3] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[4] Dan Allison. Download string as a text file. https://gist.github.com/danallison/3ec9d5314788b337b682, 2014. Last access: 2020-06-16.

[5] git.blender.org. https://git.blender.org/gitweb/. Last access: 2020-06-16.

[6] Transparent / Invisible Material - Support / Materials and Textures - Blender Artists Community. https://blenderartists.org/t/transparent-invisible-material/1224961/4. Last access: 2020-06-16.

[7] Volumetric lighting issues - Support / Lighting and Rendering - Blender Artists Community. https://blenderartists.org/t/volumetric-lighting-issues/1224047/6. Last access: 2020-06-16.

[8] Marat Boshernitsan, Michael Downes, and Michael S Downes. Visual Programming Languages: A Survey. Technical report, Computer Science Division (EECS), University of California, 2004.

[9] rii-mango/Daikon: A JavaScript DICOM reader. https://github.com/rii-mango/Daikon. Last access: 2020-06-16.

[10] Game Development Stack Exchange - unity - How to distort 2d perlin noise. https://gamedev.stackexchange.com/questions/162454/how-to-distort-2d-perlin-noise. Last access: 2020-06-16.

[11] Tomás Sempere Durá. *Atlas interactivo de anatomía radiológica anatomía a través de la radiología*. Skadi, Madrid, 2014.

[12] Tomás Sempere Durá. Atlas interactivo de anatomía radiológica en App Store. https://apps.apple.com/es/app/atlas-interactivo-de-anatom{í}a-radiol{ó}gica/id921475745, 2014. Last access: 2020-06-30.

[13] Klaus Engel. Real-Time Volume Graphics. Technical report, Taylor & Francis Inc, 2006.

[14] Marcus Fantham and Clemens F. Kaminski. A new online tool for visualization of volumetric data. https://github.com/fpBioImage/VRBioimage, feb 2017. Last access: 2020-06-16.

[15] toji/gl-matrix: Javascript Matrix and Vector library for High Performance WebGL apps. https://github.com/toji/gl-matrix. Last access: 2020-06-16.

[16] Patricio Gonzalez. The Book of Shaders: Random. https://thebookofshaders.com/10/, 2015.

[17] Patircio González. GLSL Noise Algorithms. https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83. Last access: 2020-06-30.

[18] Paul E. Haeberli. ConMan: A Visual Programming Language for Interactive Graphics. *Computer Graphics*, 22:103–111, 1988.

[19] HTML CSS JavaScript - The Client-Side Of The Web. https://html-css-js.com/. Last access: 2020-06-16.

[20] Javi Agenjo. jagenjo/litegl.js: Lightweight Javascript WebGL library. https://github.com/jagenjo/litegl.js, 2013. Last access: 2020-06-16.

[21] Javi Agenjo. jagenjo/litegraph.js: A graph node engine and editor written in Javascript. https://github.com/jagenjo/litegraph.js, 2013. Last access: 2020-06-16.

[22] jquery/jquery: jQuery JavaScript Library. https://github.com/jquery/jquery. Last access: 2020-06-16.

[23] EastDesire/jscolor: Web Color Picker that aims to be super easy both for developers and end users. https://github.com/EastDesire/jscolor. Last access: 2020-06-16.

[24] Matthew MacLaurin. The design of kodu: A tiny visual programming language for children on the Xbox 360. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 241–245, 2010.

[25] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):1–15, nov 2010.

[26] Tom McReynolds and David Blythe. CHAPTER 20 - Scientific Visualization. In Tom McReynolds and David BT Advanced Graphics Programming Using OpenGL Blythe, editors, *The Morgan Kaufmann Series in Computer Graphics*, pages 531–570. Morgan Kaufmann, San Francisco, 2005.

[27] Microsoft. Coming to DirectX 12 — Mesh Shaders and Amplification Shaders: Reinventing the Geometry Pipeline | DirectX Developer Blog. https://devblogs.microsoft.com/directx/

coming-to-directx-12-mesh-shaders-and-amplification-shaders-reinven
2019. Last access: 2020-06-16.

[28] Paul Nagy. Open source in imaging informatics. *Journal of Digital Imaging*, 20(SUPPL. 1):1–10, nov 2007.

[29] Bernhard Preim and Charl P. Botha. *Visual Computing for Medicine: Theory, Algorithms, and Applications: Second Edition*, volume 2. Elsevier Inc., 2014.

[30] Inigo Quilez. Fractals, computer graphics, mathematics, shaders, demoscene and more. https://iquilezles.org/www/articles/morenoise/morenoise.htm, 2008. Last access: 2020-06-16.

[31] Inigo Quilez. Fractals, computer graphics, mathematics, shaders, demoscene and more. https://iquilezles.org/www/articles/texture/texture.htm, 2009. Last access: 2020-06-16.

[32] Inigo Quilez. Analytic Normals 3D. https://www.shadertoy.com/view/XttSz2, 2016. Last access: 2020-06-16.

[33] Partha Pratim Ray. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017:1231430, 2017.

[34] José Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. Visual programming languages integrated across the curriculum in elementary school: A two year case study using "scratch" in five schools. *Computers and Education*, 97:129–141, jun 2016.

[35] Will Usher. Volume Rendering with WebGL. https://www.willusher.io/webgl/2019/01/13/volume-rendering-with-webgl, 2018. Last access: 2020-06-16.

[36] Volumetrics/volume-base.js at master · upf-gti/Volumetrics. https://github.com/upf-gti/Volumetrics/blob/master/src/volume-base.js. Last access: 2020-06-16.

[37] Volumetrics/volume-loader.js at master · upf-gti/Volumetrics. https://github.com/upf-gti/Volumetrics/blob/master/src/volume-loader.js. Last access: 2020-06-16.

[38] vitmalina/w2ui: JavaScript UI library for data-driven web applications. https://github.com/vitmalina/w2ui. Last access: 2020-06-16.