Node-Based Shader Editor for Volume Rendering in WebGL

Víctor Ubieto, 2020

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.

Contents

List of Figures

List of Tables

1 Introduction

As the title mentions, in this report we will talk about different topics. Node-based shader editors, which are visual editors that allows the user to modify how the scene is seen. Specifically applied in cases where Volume Rendering techniques are required, that allows the visualization of the inside of the volumes. And all of that will be applied in web, which will improve the accessibility of the application and will also ease its implementation with the use of some libraries.

1.1 Objective Identification

If we analyse the people who work with 3D data, we find two main cases. The medical case consists on the visualization of human datasets to help doctors detect diseases and therefore obtain correct diagnoses. On the other side we have the case of people who work on the film or videogames industry and need to create realistic scenes using amorphous phenomena such as cloud, fog, and fire. In both cases they are helped with the use of computer applications.

In both cases we can detect the necessity of manipulate 3D data which is achieved by using computer applications. And here is where we introduce the Visual Programming Languages (VSL), which ease the use of complex techniques to any user with the finality that him or her obtains a more satisfaction while using and consequently better results.

For this reason, the objective of this project is to create a web application which contains a graph editor for shaders that use volume rendering techniques.

2 State of the Art

In order to understand the whole report, we first need to go deeper in the two main topics of the project, Volumetric Data and Visual Shader Editors (VSE).

2.1 Volumetric Data

Volumetric data can be represented as a 3D scalar field, "3D" because it is defined by the X, Y, and Z axis (we could say 4D if we would consider time), and "scalar field" because it associates one value for each point of the continuous space, it is represented by the function $y = f(x_0, x_1, ..., x_n)$, where n is the domain.

Since we will be working in computer engines, the data is represented in voxels, which is the minimum processing unit of a 3D grid in Computer Graphics (CG). Just like pixels in 2D space, they represent the resolution. (Figure 1).

The inner values between voxels are obtained by using interpolation techniques, which significantly improves the outcome. 3D scalar fields are visualized by using Volume Rendering techniques.

But before explaining these techniques, let me explain how it is the 3D data obtained and stored in the mentioned application cases.

Human body datasets are found in Radiology, a medical discipline that uses Medical Imaging techniques to diagnose and treat diseases. Data is acquired via imaging techniques, such as Computer Tomography (CT) and Magnetic Resonance Imaging (MRI), or functional imaging, such as Positron Emission Tomography (PET). Additionally, there exists more methods and even specializations of other techniques such as fMRI or Micro-CT scanning.

In all cases, the data obtained is called raw data, which is then derived into specific format files depending on their application use, like Dicom (.dcm), Nifti (.nii), Minc (.mnc), and

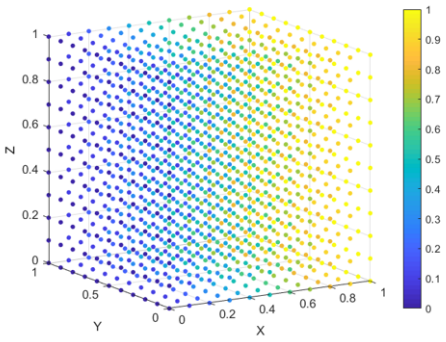


Figure 1: 3D sampled scalar field.

Analyze (.img, .hdr) [3]. As explained, the data is visualized for diagnostic purposes or for planning of treatments or surgeries. Each hospital has a contract with a company who provides the application they will use, such as IntelliSpace Portal (Philips application), and Syngo.via (Siemens application).

On the other side, in the modelling case, the data is not obtained as datasets but as a function that represents the behaviour of the scalar field. For that reason, the field is modified using mathematical functions such as noise. The applications that handle this case are the Shader Editors such as Blender.

Back to the volume rendering techniques, they can be classified in two types: Indirect Volume Rendering (IVR), and Direct Volume Rendering (DVR).

On one hand, the IVR makes an inner step convert the data into a set of polygonal isosurfaces, this process is called isosurface extraction.

On the other hand, DVR techniques can be divided into four groups: 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).

For this project, I compared them (Table 1) and decided that the Ray Casting technique was the optimal in this case since we were aiming for the best possible quality. Moreover, if needed, its disadvantages could be corrected by implementing acceleration techniques.

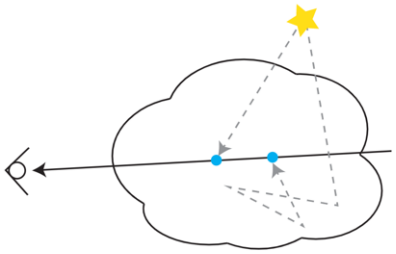
The Volume Ray Casting is an image-based volume rendering technique. The basic idea is to directly evaluate the volume-rendering integral along rays that are traversed from the camera. To do so computationally, we cast a single ray into the volume for each pixel of the image. The rays travel along the volume step by step with a ray marching algorithm. And at each step we compute the colour and opacity using a light transport model. We can visualize the process in the Figures 2 and 3.

2.2 Visual Shader Editors

In order to understand VSEs, we first need to understand VPLs. VPLs describes any system that lets the user specify a program using graphic elements instead of writing the code. The most popular VPLs are based in the Dataflow model, which is based a net of connections between elements called nodes via links. This structure where a set of entities are connected in pairs between them is known as graph (Figure 4).

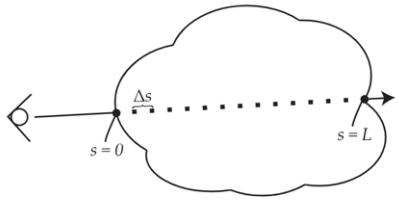
Methods	Advantages	Disadvantages
IVR	- Occlusion of hidden surfaces - Easier implementation in shader	- Not the entire data is visible - Mesh extraction is not interactive
Ray Casting	- Best image quality results - Acceleration techniques	- Requires hi-end graphics hardware - One of the slowest methods
Texture Slicing	- No need of expensive graphics hardware - Fast for moderately sized data sets	- Artifacts at some rotations - Limited by texture memory
Shear Warp	- High speed - Good optimizations (efficient)	- Less accurate sampling - Less image quality
Splatting	- High speed - Air voxels can be disregarded	- Lower quality (blurry) - Cuts away high frequencies

Table 1: Volume rendering comparison techniques.



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

Figure 2: Volume Ray Casting (step 1) [1].



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

Figure 3: Volume Ray Casting (step 2) [1].

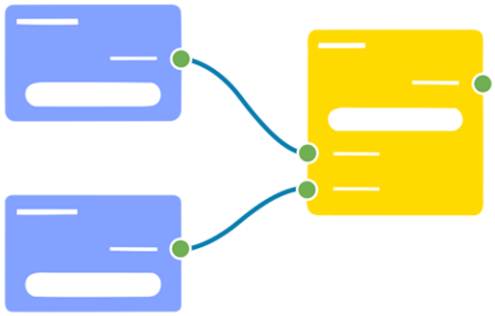


Figure 4: Graph structure.

The creation of the VSEs (also called Node-Based Shader Editors) comes from the relation between a specific type of graph called Direct Acyclic Graph (DAG) and a part of the Rendering Pipeline that takes part in the Graphics Processing Unit (GPU) (Figure 5).

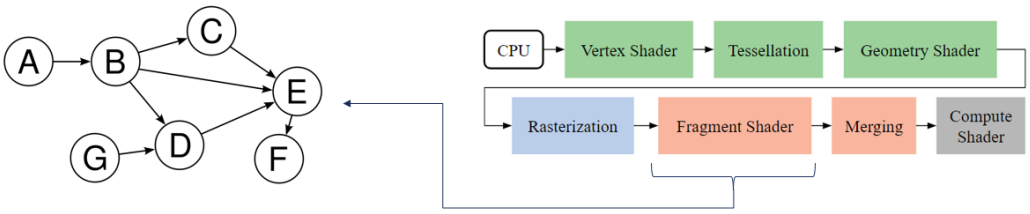


Figure 5: Relation of a Direct Acyclic Graph and the Fragment Shader.

Thanks to that, many computer programs were created offering the edition of shaders by using nodes, some examples are Blender [4] and Babylon [5].

3 Design

In order to correctly design my application, I studied the current programs in use for all the cases and collected the advantages and the disadvantages. By combining this and the first objective proposed, I created a list of requirements both functional and non-functional. The main aspects could be expressed in three points:

- Provide a web shader editor to model volumetric materials.
- Provide an online renderer for volume datasets.
- Provide a tool to create personalized shaders.

Additionally, since we already explained the State of the Art, we can see the considerations I took to implement the project:

- It will be programmed in Javascript, specific for web applications.
- I will use WebGL2, the new version of the API for rendering interactive 2D and 3D graphics within any compatible web browser.
- I will only consider the upload of Dicom files, since it is the most used and easy to obtain.
- I will use the Volume Ray Casting technique, as explained previously.
- The shader will be generated in forward direction, this is related in how the program will build the code for the GPU.

3.1 Framework

The User Interface (UI) is the link between users and your website, it must be designed carefully by anticipating the user preferences and behaviour. 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 attention to the two main canvas and letting the header as a helpful multi-tool for the user (Figure 6).

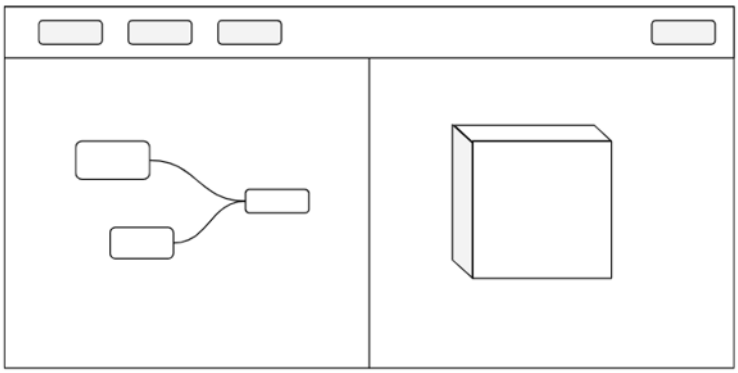


Figure 6: Design of the interface of the application.

We can understand the behaviour using a Model-View-Controller (MVC) 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 7).

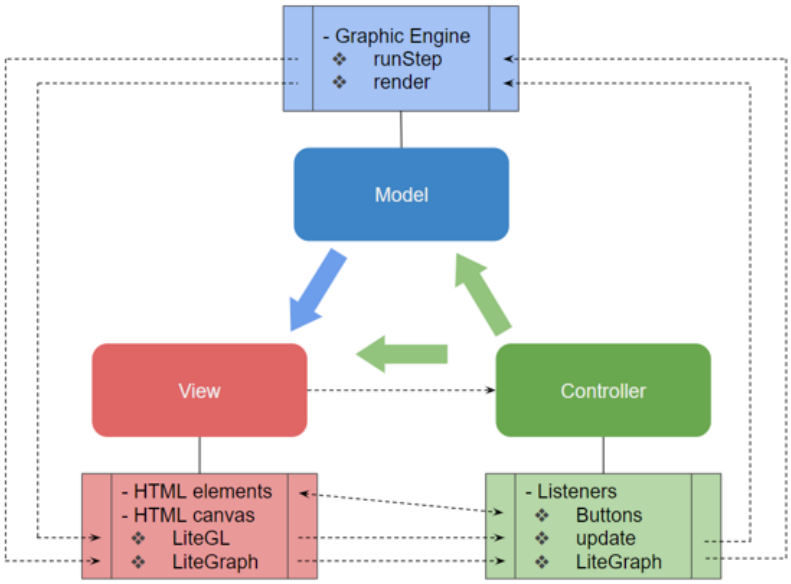


Figure 7: MVC pattern of the framework.

Finally, the shader generation will proceed in forward, which means that the nodes will inject code to their connections, this will repeat until the last node. The Output node creates the final code and uses it to create the shader that will be used in the graphic engine. This can be seen in Figure 8.

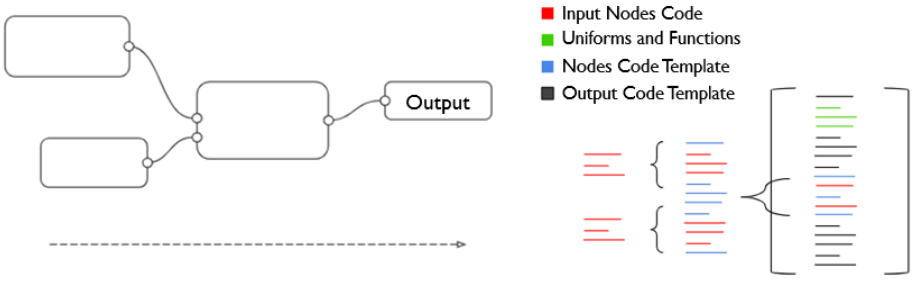


Figure 8: Forward shader generation.

3.2 Nodes

The basic structure of a node consists on different parts.

- The inputs passes information to the node, which will be used to create its own output.
- 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 links or 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, ...).
- The widget are a fast tool to modify the behaviour of the node. Each node may have its own widgets for different purposes.
- Each node will has its own uniforms, methods, and code-lines that will be placed in the final shader at different locations.

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

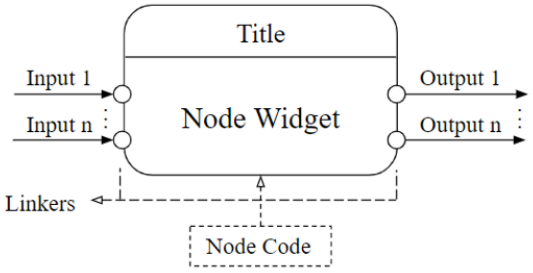


Figure 9: Design of the nodes.

In order to select the list of nodes that I was going to implement, I made a huge research of the needs for different types of users (the targets of the project). Moreover, I decided to classify the nodes into groups, each one having a unique characteristic and representative color.

Figure 10: Classification of the nodes implemented.

4 Implementation

One of the most important reasons to do this project, is the possibilities that some libraries gave. Being the most important LiteGL.js, which helps simplifying working with WebGL, and Litegraphs.js, which is a library in Javascript to create graphs in the browser. (The complete list of libraries can be found in my GitHub repository [6])

The framework basically consist in two parts: the initialization and the main loop. The first step prepares the application to work properly, and the second step is an infinite loop where the visualization is updated depending on the user actions.

The implementation of all the nodes was not simple, and would spend several pages to explain each one individually. For this reason, we will only see three nodes: Dicom, Transfer Function, and Output.

4.1 Nodes

The Dicom node is a simple box with no inputs and one output, the user must interact with the node by opening the support panel. By default, it will be empty, this will be represented with a sentence, the charging bar and the color of the node (grey if it is empty, orange if we uploaded a dicom).

More internally, the addition of this node to the graph implies different processes in order to work correctly:

- The mesh must be escalated like the dicom data specifies (the data is intended to be visualized in the same scale that it was stored).
- We consider 3 different types of data (depending on the number of bytes it was stored).
- We interpolate the value between the nearest voxels. This makes a nice improvement (Figures 11 and 12).
- We apply a basic technique to improve the quality of the result when working with low quality (very few steps in the Ray Casting iteration). This is called jittering, and basically makes the rays to not start at the same distance every time (Figures 13 and 14).

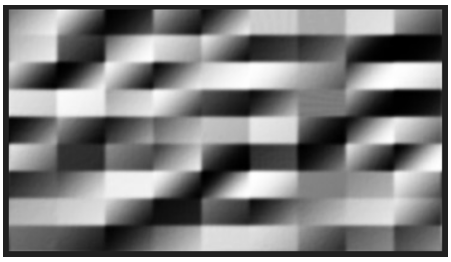


Figure 11: Texture interpolation not applied.

Figure 12: Texture interpolation applied [2].

Figure 13: No jittering applied.

Figure 14: Jittering applied.

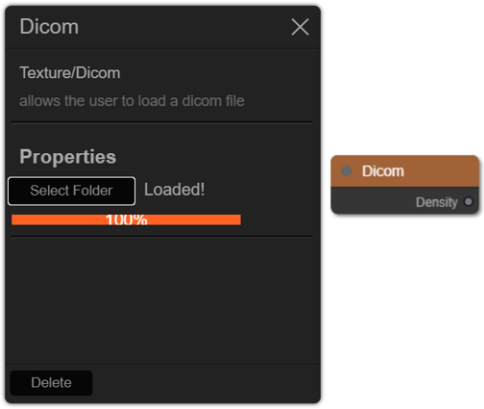


Figure 15: Dicom node.

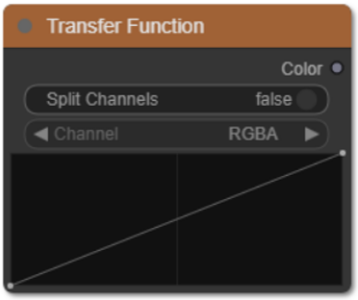


Figure 16: TF node.

The Transfer Function is a well-known tool when working with 3D datasets. It allows the control of the visibility of the data depending on its density, it also allows the user to differentiate the different parts of the dataset with a different color each. The node implemented

consist on a curve editor that creates a texture that will be applied to the density volume of the dataset at each iteration.

Finally, the Output node is designed to be prepared to inject the received code to different locations and create the new shader. Additionally, it does some control task in order to improve the performance of the application. We will not go into detail in all the control tasks, but I would like to mention that is probably the hardest part when creating a node-based editor.

4.2 Accessibility

As presented during the whole report, one of the most important features of the framework is its accessibility. This is done thanks to GitHub Pages, which is a static web hosting service. This framework is also licensed under the MIT license, which makes it completely open-source.

Moreover, I considered the possibility of some user to not have any Dicom to test the application. For this reason, I uploaded some datasets in the repository [6].

Finally, this project will be updated and improved, and each user can be part of that, thanks to the accessibility that offers GitHub and the close relation between the author and the users.

5 Results

In order to test the results and evaluate the work done, I will refer to the objectives and requirements presented in this report. And we can say that it meets all the functionalities specified, it consider both use cases presented, and it prevents all possible bugs.

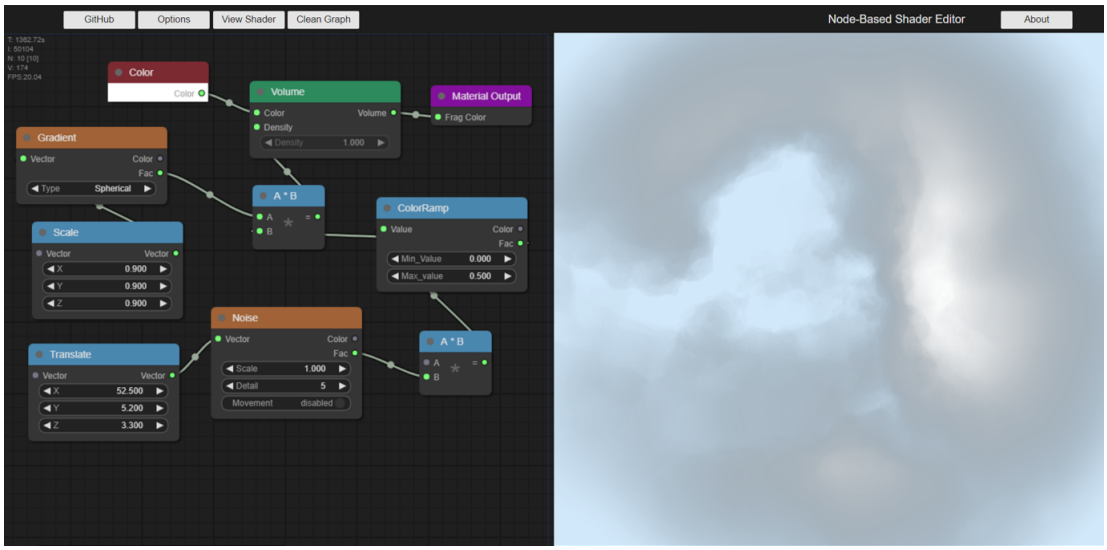


Figure 17: Clouds visualization example.

In the following images we can see some results for each of the application cases, and we can say that the quality is quite good. Nevertheless, the worse part of this project is the performance at some points. It will normally go perfectly, but when adding to much computational expensive

nodes to the graph (for example the noise node) it may reduces the frames per second to less than 30, which is the standard to a real-time application. For this reason, I designed a quality bar, where the user can modify the quality of the outcome depending on the performance he or she is having.

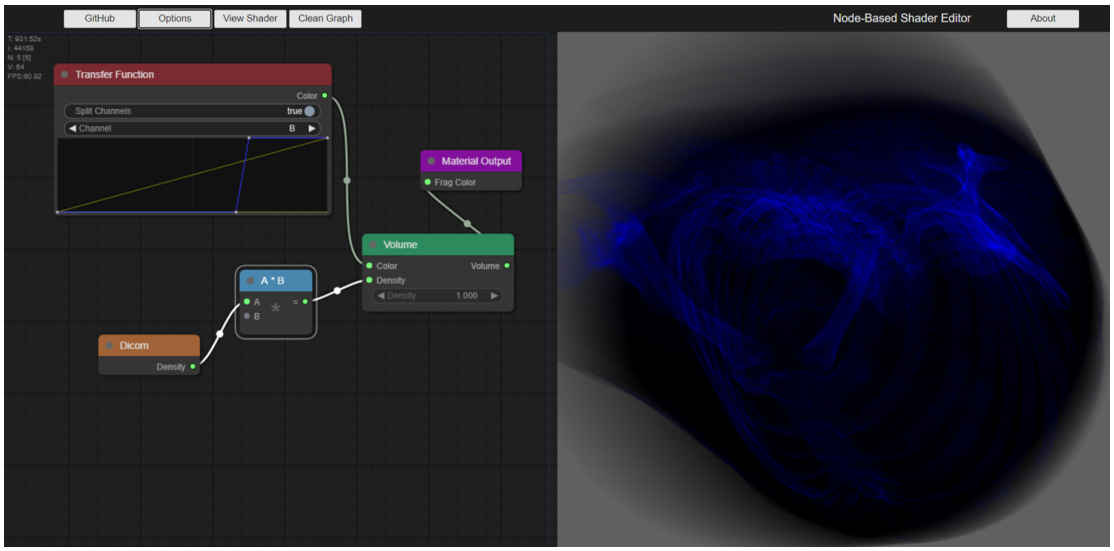


Figure 18: Dicom visualization example.

References

[1] Will Usher. Volume Rendering with WebGL. `https://www.willusher.io/webgl/2019/01/13/volume-rendering-with-webgl`, 2018. Last access: 2020-07-24.

[2] Inigo Quilez :: fractals, computer graphics, mathematics, shaders, demoscene and more. `https://www.iquilezles.org/www/articles/texture/texture.htm`. Last access: 2020-07-24.

[3] Michele Larobina and Loredana Murino. Medical image file formats. In *Journal of Digital Imaging*, volume 27, pages 200–206. Springer New York LLC, dec 2014.

[4] Blender.org - Home of the Blender project - Free and Open 3D Creation Software. `https://www.blender.org/`. Last access: 2020-07-24.

[5] Babylon.js Documentation. `https://doc.babylonjs.com/`. Last access: 2020-07-24.

[6] victorubieto/graph_system: Node-Based Shader Editor:. `https://github.com/victorubieto/graph{\_}system`. Last access: 2020-07-24.