Design and Development of Videogames Degree

Final Degree Project

SHADERONOMICON: Tool for high-level developing and exportation of shaders

Author: Albert Dols Ferrandis
Academic supervisor: José Ribelles Miguel

Date: 01/07/2018
Academic Course 2017/2018

Abstract

Computer graphics is a branch of computer science that involves the treatment of everything that could appear on a screen. In the past years, this field has become one of the most important for video game developers, as it provides a tool for easy but powerful optimization solutions, and a huge improvement in visual quality. As interesting as it may sound, the greatest barrier to learn about it is the difficulty of its intern calculations and structure. As it requires not only to learn/adapt oneself to a programming language (in this case, nvidia CG is a C-based language), it also requires more advanced mathematical knowledge, regarding matrix operations.

Fortunately, in the recent years, this barrier was tightened by tools that use an object-based programming, the node-based shader edition. This type of edition allows the user to create shaders in a more visual way, using objects with linkable properties, and let us see the result in real-time. This type of tool has overcomed the hurdle of using a programming language, but it still requires specific mathematical knowledge to dig out its full potential. An example of this type of tool is the Shader Graph[1], the node-based shader editor included inside Unity3D[2].

Shaderonomicon is a tool which aims to surpass the previously mentioned limitations, and give access to any developer to basic shader edition, in an intuitive and easy way. By using this application, one can understand which basic properties have a texture[3], normal map[4], or how lighting models affect the overall quality of the shader's output, and how all of these features interact with each other.

The target audience of Shaderonomicon involves anyone without prior shader knowledge, who wants to learn in a direct way how does a shader works, and experiment with basic features without writing a line of code. Because of its easy-to-export and multi-platform operability, Unity3D is the best platform, not only to implement the application's systems, but also the most efficient one for the shader to be exported to.

Key Words

Computer Graphics, Shader Development, CG Shader Coding, Unity3D

Index

1 - Introduction: Shaderonomicon

1.1 - About the project: What is this?

This document is the final memory of Shaderonomicon, the project made for the Final Degree Project in Design and Development of Videogames, based on the development of a tool which allows to create simple but powerful shaders. Using high-level controls such as sliders or lists of options, its aim is to provide indie developers, beginner shader developers, and everyone interested in this "black-magic realm", a way to work in shaders without using code, or the necessary knowledge about shader programming. This tool will be created using Unity3D, and finished shaders could be exported to this game engine.

1.2 - Basic Concepts: What is a shader?

Following the wikipedia's description[5], a shader is:

> " A type of computer program that was originally used for **shading** (the production of appropriate levels of light, darkness, and color within an image) but which now performs a variety of **specialized functions** in various fields of **computer graphics**, **special effects** or does **video post-processing** unrelated to shading, or even **functions unrelated to graphics** at all."

So, in short, a shader is a program used by the GPU, which can perform a wide range of operations, mainly graphics oriented ones. A shader is usually assigned to a mesh, and some types of shader are counting on its properties to do its calculations. There are many types of shaders: geometry shaders, tessellation shaders, compute shaders, pixel shaders, vertex shaders, etc. Every type is specialized on a particular field, although some of them can be mixed to obtain unique results.

In Shaderonomicon, only one type of shader is used: the "**Pixel Shader**", because of its relevance in basic shader knowledge. This type of shader is well-known for its "**fragment**" calculations. This behaviour allows the shader to obtain realistic lighting representations, and simulate normal mapping, as one of its main features. Also, we will implement a variation of this type of shader, mixing its structure with the "**Vertex Shader**", because of its structure harmony with the **Pixel Shader**, and the differences between how they internally work (the main difference is the place where they do their calculations). To understand this, I will briefly explain the structure of these functions:

- Both Pixel and Vertex Shader use two main functions: the **vertex function** and the **fragment function**. The first is called **every vertex of the mesh**, and the second is called **every time a pixel is rendered**.

- Knowing this, we can assume that, if the calculations are made in the **vertex function**, the shader will **perform better**, but its **visual output will be worse** (as for the pixel color will be interpolated from the mesh face's vertex). And, in the other hand, if the calculations are made in the **fragment function**, the shader will produce a **more detailed color output**, but its **performance will be hindered**.

1.3 - Motivation: Why?

The main purpose of Shaderonomicon, is to provide a useful tool to developers who does not know how to create a shader, and overcome the initial hurdle of not knowing what they can do with them. By experiencing first-hand how shaders work, they will gain experience with them faster than by coding by scratch without knowing what are they really doing. Based in my personal experience, I decided to develop an application which would let new programmers to learn about the basic concepts in a intuitive way, to help them in their first steps inside the computer graphics field.

Also, shader developing does not only provides better visual results, but also improves its application's performance, if used wisely. This approach to game optimization is not common among indie developers, because it is not an essential priority for low-level projects: helping them to have more possibilities thanks to teaching them how do shaders work, will make them more competitive in an overcrowded industry, like the mobile-oriented videogame environment.

The development of a tool of these characteristics requires a lot of time dedicated to research, not only for the development of shaders, but also for the interface design and implementation: it has to be well designed and explained for everyone to understand every feature, without the use of tutorials. Furthermore, most of the basic tools that are needed for the first version of the application are not implemented inside Unity3D, so they must be done from scratch too.

1.4 - Objectives: What are our goals?

The goals of this project include, not only the shader edition system development, but also the creation of the application itself. The main objectives of Shaderonomicon are:

1. To develop a tool to help developers with zero experience with shader editing, to create them without writing a line of code.
2. To develop a tool that exports the created shaders to Nvidia CG.
3. Ability to customize the basic features that usually involves shader edition (texture, normal map, lighting model, etc).
4. The application interface can be understood without prior knowledge, and users can start creating shaders without explicit instructions.
5. The exported shader works correctly in an external Unity3D project.

1.5 -Initial Task Planning

Task	Estimated Time
Reference Research: Bibliography, competitors, shader frameworks, etc.	**30 hours**
Implementation of tools and systems:	**175 hours**
• **Color Picker Tool**	10 hours
• Implementation	5 hours
• Research	1 hour
• Shader developing	2 hours
• Bug fixing	2 hours
• **File Explorer Tool**	7 hours
• Implementation	3 hours
• Research	3 hours
• Bug fixing	2 hours
• **Shader Management System**	125 hours
• Implementation (C# framework)	5 hours
• Shader development (nvidia CG)	30 hours
• Library function development (both nvidia CG and string-based)	75 hours
• Bug fixing	15 hours
• **Shader Export System**	20 hours
• Implementation	15 hours
• Bug fixing	5 hours
• **Mesh Related System**	5 hours
• Mesh visualization System	1 hour
• Mesh Swap System	1 hour
• Camera Movement System	3 hours

• **Scene Settings Parameter System**	6 hours
• **Exit Application Menu System**	1 hour
• **Shader Change Name System**	1 hour
Project's Technical Proposal	**4 hours**
Project's Research and Design Proposal	**6 hours**
Project's Memory	**30 hours**
Project's Presentation	**10 hours**
Project's video	**10 hours**
Project's Main Interface Development	**30 hours**
• Visual Shader Development	3.5 hours
• Implementation	23.5 hours
• Multi-resolution canvas	3 hours
Organization of project's assets (relocation and removal of obsolete assets and code)	**5 hours**
Total hours:	**300 hours**

1.6 - Expected Results

The developed application will work without any major or minor bug; its performance will be stable, and its interface would be easily understood by anyone without prior knowledge about shaders. The quality of the exported shaders will be excellent in both visual result and efficiency. The quantity of possible features to use will not make the application more difficult to operate. The shader will be correctly exported, and could be used in an external Unity project. Its framework will be scalable, for its continued development.

1.7 - Tools and used plug-ins

- Engine to create the app: **Unity3D.**
- Code Editor: **Microsoft Visual Studio 2017[6].**
- Program to create the artistic assets: **Adobe Photoshop CC[7].**
- Plug-in for File Explorer Management: **CrossTale's File Browser (Asset Store plugin)[8].**
- Program to create and edit the Project's Documents: **Google Docs[9].**
- Program to create and edit the display meshes: **Blender[10].**

1.8 - Similar Tools

- Node-based shader editor: **Unity3D Shader Graph.**
- Professional material and texture editor: **Substance**[11].
- Code-based and node-based shader editor: **Shaderfrog**[12].
- Code-based shader editor: **Shadertoy**[13].

2. Shaderonomicon's systems

2.1 - Introduction

The development of this application will require the implementation of a wide variety of tools to provide the best experience to the user, which we will call from now on, "**systems**". They are divided in three great fields, and the interaction between them can be seen in the **Figure 1**:

- **Shader-oriented systems**: These are the backbone of the project, as they are the ones which manage the shader's library and its interaction with the other two fields.

- **Feature-oriented systems**: These are the ones responsible to ease and optimize the user's actions to modify the different properties of the shader. They are the bridge to unite the other two fields.

- **Scene-oriented systems**: These are in charge of the user's interaction with the application's interface and its visualization.

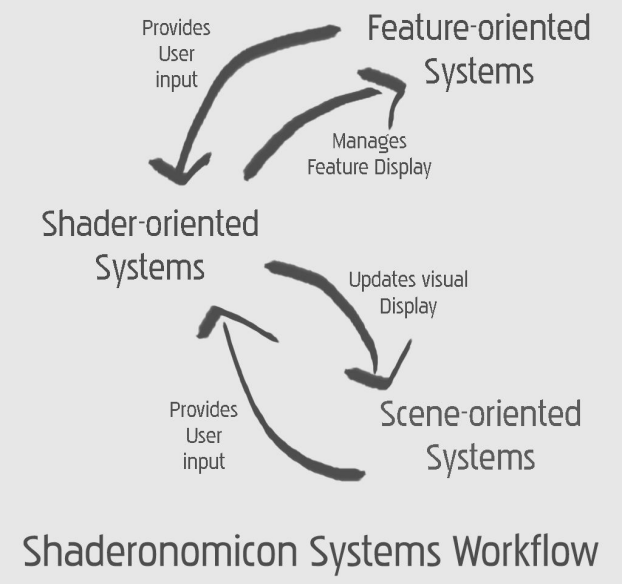


Figure 1. Workflow diagram of the different system fields.

2.2 - Shader-oriented systems

A shader-oriented system is a tool whose objective is to manage the shader's library code, or provide it to another systems of another field. The ones which this project use are the following:

2.2.1 - Shader export system

One of the main features of the application. This system enables the player to save the modifications of the edited shader, in a ".shader" text file format that is compatible with almost every version of Unity3D, as my research concluded that this format is exactly the one used to write code in nvidia CG. It receives information from the **Shader Naming** and the **Shader Management** tools, from which data will fill all the gaps needed for the shader to work correctly.

Its task is done smoothly thanks to some of the most important tools of the application: the **File Explorer**, the **Shader Feature Save** and the **Shader File Writing**. These all three are intertwined in a simple workflow:

1. The user edits the shader name and the properties of the shader as desired.

2. When the "Save Shader" button is pressed, the File Explorer tool shows a window for the user to select the desired folder to save the shader.

3. If he does not cancel this window, and selects the desired folder, the main system receives the necessary information from the **Shader Management, Shader Naming** and **Shader Feature Save** tools, which data is sent to the **Shader File Writing** tool, filling the shader to be exported.

4. When the **Shader File Writing** tool finishes its job, the file explorer exports the shader to the selected folder.

2.2.2 - Shader Management system

This system is the main application's system. It receives information from almost every tool, and modifies the shader to adjust its desired output. It sounds simple, but its components are quite complex: the **State Machine Shader Component** provides the framework to switch between the correct functions from the **Shader Library Component,** while the **feature-oriented systems** provide the necessary information from the user, and the **scene-oriented systems** show the output of the shader, as shown in **Figure 1**. We will now proceed to explain how these components work:

2.2.2.1 - Shader Library Component

Inside this component resides all the possible functions that the shader can use, and the **State Machine Shader Component** main framework. This is possible thanks to a **function-based structure** which allows the code to be straightforward and easy to read and edit. But this component is split in two: the real code reference, from which the shader receives the data, and a **string-based copy,** used by the **Shader Feature Save** tool, to obtain the code from the necessary functions.

2.2.2.2 - State Machine Shader Component (SMSC)

This component adapts the shader's output depending on its state, given by specific control parameters. The SMSC is located inside the **Shader Library Component**, and its framework depends entirely on it.

The main goal of the SMSC is to switch between the shader's vertex and fragment functions, depending on the features that the user wants. So, for example, if the player does not use a texture and do not use a normal map, the SMSC makes sure that the current vertex and fragment shader are the ones that do not use them. The moment the user adds one of these properties, the SMSC changes the current vertex and fragment shader used, now supporting this feature. But, how can a shader change its vertex and fragment functions in real time? The answer to that question is that they never change.

When compiling our shader within the application, we use special vertex and fragment functions, which receive a data structure made only for this purpose. Inside the functions, depending on control parameters offered by the **feature-oriented systems**, we choose the correct vertex and fragment functions. These need their respective data structures (because not all of the functions need all of the available parameters), so we will create auxiliar data structures for these purpose, and update them with the correct values. This way, we can simulate the behaviour of a state machine inside the shader in real time. An example of an actual cycle of the SMSC would be this:

1. As we enter the vertex/fragment function, we follow the logic of the state machine to reach the current state.

2. If we need additional data structures, we create and complete them with the received custom input structure.

3. We retrieve the necessary information from the state's real vertex function (called with the correct data structure), and complete the custom output structure with it.

4. When we reach the fragment function, we repeat steps 1 and 2.

5. Then we will obtain the final color of the pixel from the state's real fragment function.

This structure has a great flaw: its practical performance. Checking every cycle the shader's current state, creating additional variables and processing unnecessary input variables would drastically affect the shader's performance if the user was to use it in a real videogame environment. But this is where the **Shader Library Component**, and the **Shader Export System** have their time to shine: using the string-based copy of the library, and accessing the control parameters, we deduce

which the shader's current state so to correctly export the shader we do know which specific functions we need to export, without extracting the SMSC, avoiding this issue.

2.3 - Feature-oriented systems

A feature-oriented system is one whose responsibility lies in maintaining the user's attention at the interface, give adequate feedback about what they are doing, and ultimately improve the user's interaction with the shader's features. Its main tasks require translating the user's input from the main scene tools, (**Sliders**, **Input Fields**, **Color Selection** and **Image Selection**), and updating the appropriate values of the **Shader Management system**. The following systems are the ones within this field:

2.3.1 - Main Texture Parameters system

This system is in charge of the most basic and important feature of a basic shader: the main texture. It visual design can be seen at the **Section 5.1.2.**
The properties that this system allows to modify are the following:

- **Main Texture**: Applies an image to the figure, depending on its UV coordinates. To obtain this image, the system makes use of the **File Explorer Selection** tool, converts the obtained picture to a usable texture, and then assigns it to the shader using the **Shader Management System**. The provided controls allow the user to select an image, reset this component (by deleting the selected texture from the shader), or visualize the applied texture.

- **Main Color**: This property changes the color output of the texture and therefore, of the shader, using an external tint. To obtain the desired color from the user, we use the **Color Picker** tool, and then update the correct parameter using the **Shader Management System**. The provided controls allow the user to select this color (with the **Color Picker Tool**), tweak its influence at the color calculations (with a **Slider** which limits are restrained from 0 to 1), visualize it (using an **Image** component), and reset it to white (using a **Button**).

- **Offset X and Y of the Main Texture**: This properties allow the user to displace the UV coordinates of the Main Texture at the X and Y axis. The coordinate shifting is restricted from 0 to 1, as the texture coordinates cannot be different from that range. This special properties are presented by **Sliders**, and if the user changes them, the parameters are updated in the **Shader Management System**.

- **Tile X and Y of the Main Texture**: This properties allow the user to repeat the UV coordinates of the Main Texture, in the X and Y axis. Making use of the **Input Field** tool, we can obtain from the user the desired repetition

of the coordinates, and then update the necessary values with the **Shader Management System**.

2.3.2 - Normal Map Parameters system

This system is in charge of one of the most basic and important features of a basic shader: the normal map. Its visual design can be seen at the **Section 5.1.2.** The properties that this system allows to modify are the following:

- **Normal Map**: Applies a normal map to the figure, depending on its UV coordinates. To obtain this image, the system makes use of the **File Explorer Selection** tool, converts the obtained picture to a usable texture, and then assigns it to the shader using the **Shader Management System**. The provided controls allow the user to select an image, reset this component (by deleting the selected texture from the shader), or to visualize the applied texture.

- **Normal Map Force**: This parameter adjust the force that the normal map applies to the lighting data. This property is adjusted thanks to the **Input Field** tool, receiving the information that the user provides, and then updating the necessary values using the **Shader Management System**.

- **Offset X and Y of the Normal Map**: This properties allow the user to displace the UV coordinates of the Normal Map at the X and Y axis. The coordinate shifting is restricted from 0 to 1, as the texture coordinates cannot be different from that range. This special properties are presented by **Sliders**, and if the user changes them, the parameters are updated in the **Shader Management System**.

- **Tile X and Y of the Normal Map**: This properties allow the user to repeat the UV coordinates of the Normal Map, in the X and Y axis. Making use of the **Input Field** tool, we can obtain from the user the desired repetition of the coordinates, and then update the necessary values with the **Shader Management System**.

2.3.3 - Lighting Model Parameters system

The Lighting Model is the sum of different calculations that simulate the light reflection over the surface of an object. Its good use is a must for any basic shader, and its implementation, one of the main struggles that new graphics programmers have to face. As every model uses its own set of scene parameters to do their computations, they have to be thoroughly studied, not only on the calculations, but also on the acquisition of the crucial engine variables to make it work correctly. As every **Lighting Model** makes use of the normal Direction at every cycle (its base value depends if the lighting model is pixel-based or vertex-based), its result is affected by the addition of a **normal map**, which only purpose is to modify this value by using a special color codification.

The calculations of the **Lighting Model** can be called at the vertex function, or the fragment function, if the input parameters are processed correctly. Its performance and visual output performance differences are explained at the Section **1.2.**

In Shaderonomicon, there are seven possible **Light Models**, three of them are vertex-based, one is the possibility of having no light system, and the last three are pixel-based. Each three of both categories are **Phong**, **Lambert** and **Half-Lambert**, and the user can switch between then in real time, using the **State Machine Shader Component.**

2.3.3.2 - Phong Lighting Model Parameters system

The **Phong Lighting Model**[14] is well-known for its wide range of customizable properties, its great performance and realistic graphic results. The provided controls for all the colors allow the user to select this color (with the **Color Picker Tool**), tweak its influence at the color calculations (with a **Slider** which limits are restrained from 0 to 1), visualize it (using an **Image** component), and reset it to white (using a **Button**). Every value is updated in the shader, using the **Shader Management System**. It visual design can be seen at the **Section 5.1.2.**

- **Ambient Color Management**: This property changes the base color used by the **Phong Lighting Model**. This value is often omitted, as its only use is to provide a color difference between the mesh own color and the other two color components of this model.

- **Diffuse Color Management**: This property changes the diffuse color, which provides the lit areas the color modification given by the low reflection of the light, using its color.

- **Specular Color Management**: This property changes the specular color calculation, with boosts the color modification in areas which the light is most reflected. This areas usually “override” the previous color with the light’s color.

- **Shininess Management**: This property is in charge of increasing or decreasing the influence of the specular color calculation on the lighting formula. This property is presented by a **Slider** and two **Input Fields**, which use this workflow:
    - The user can change the shininess value by selecting the desired value from the **Slider**.
    - The user can tweak the low limit number, and high limit number of the shader, by editing the values in both **Input Fields**.
    - As any value mentioned before is modified, the **Shader Management system** updates the shader.

2.3.3.3 - Lambert and Half-Lambert Lighting Model Parameters system

Both **Lambert**[15] and **Half-Lambert** are the same lighting model, with a few differences at its variable computations. Their good points are better performance than **Phong** (by a long way), easier to implement, and is view-independent (it does not depend on camera position). Their disadvantage is that it does not represent well shiny surfaces, and its visual result is worse than **Phong**. Also, it has less variables that could be tweaked, making it less flexible.

The main difference between **Lambert** and **Half-Lambert** is the diffuse color calculation, in which the Half-Lambert is halved, then half is added, and then is squared. This helps the darker areas to lighten up a bit, displaying better the boundaries of the mesh and avoiding the apparent flattening in areas with almost no lighting. This model only allows to change the tint of the provided light (which would be the tint color over the mesh). The provided controls allow the user to select the color (with the **Color Picker Tool**), tweak its influence at the color calculations (with a **Slider** which limits are restrained from 0 to 1), visualize it (using an **Image** component), and reset it to white (using a **Button**). It visual design can be seen at the **Section 5.1.2.**

- **Light Color Management**: This color modifies the coloration over the mesh made by the diffuse color calculation. When updated, the **Shader Management System** process the appropriate values.

2.4 - Scene-oriented systems

The task of every scene-oriented system is to properly offer the correct feedback to the user, about what is he doing, and notify the appropriate system of the other fields. Its visual finish is crucial, as it will be seen at any moment the player is using the application. Also, it has to be simple, to understand its behaviour almost instantly, but complete, to provide every system with the correct data. Every system on this field has only one controllable property at a time, as it cannot be more complicated than that. So, the implemented systems are the following:

2.4.1 - Camera Movement System

This system is responsible for the camera movement in the scene, allowing to see the mesh from different angles and positions at the scene. It is the most complicated system in this field so far, as it is formed of two components: the **Camera Zooming Component**, and the **Camera Rotation Component**, both of them controlled by the mouse.

The camera movement has some inertia and absorption (the movement will start a little after the input is received, and it will finish a little after the input has disappeared), allowing it to be more natural and enjoyable. The mesh is always in the

center of the display screen, for the user to always know how changes affect the shader.

2.4.1.1 - Camera Zooming Component

This component is in charge of changing the position of the camera, relative to the mesh, focusing on it or moving away from it. This movement has its limits, as it is not possible to move further from the mesh than a specified distance, and its is not possible to move towards the mesh center if there is not a minimum distance between the camera and mesh. This component control is the mouse wheel, zooming in as the wheel rolls up, and zooming out as the wheel rolls down.

2.4.1.2 - Camera Rotation Component

This component is in charge of rotating the camera around the mesh, to visualize how does the shader behave in the scene. This component is particularly useful to see how does the lighting model works on the meshes. This rotation is locked to maintain the same distance to the mesh as it rotates (unless the Camera Zooming Component is activated), and works in the X, Y and Z axis, always having the mesh at the center of the rotation. This movement is only activated when the user does a left click at the designated area (created by a invisible **Button**), and it will not be deactivated until the user stops holding the left click button, to allow the user to move as he pleases.

2.4.2 - Mesh Swap system

This system is the one responsible to show different meshes in the scene (only one at a time) to see how does the shader looks in different forms. To make it easier for the user to choose one mesh or another, every option has a **Button** assigned at the bottom right part of the interface, with the name of the proper mesh in it, and when a mesh is selected the corresponding button will show an animation to tell the user which mesh is currently active. Its appearance can be seen in the **Figure 2**. The ones implemented right now are:

- **A sphere with a high number of polygons**: With this mesh, the user can understand the difference of applying a vertex-based lighting between a high count polygon mesh, and the other option available, as the visual difference between vertex and pixel-based lighting fades as we use more polygons.
- **A sphere with a low number of polygons**: With this mesh, it is clearly visible the difference between the vertex-based lighting and pixel-based lighting.
- **A torus**: With this mesh, we can show how does the coordinate map for the texture deforms itself to adapt to the mesh, and a great example of how does the lighting models calculation works with bent surfaces.
- **A cylinder**: With this mesh, the user can detect the differences in the coordinate map for the texture, between planes, and how does this deformation affects the texture and lighting. A low count polygon is chosen, to highlight the differences between pixel-based and vertex-based lighting.

- **A cube**: WIth this mesh, the user can see the difference between the orientation of the UV coordinates on the different mesh faces, and how it matters to take this into account.

Figure 2. Mesh Swap System, inside the application.

2.4.3 - Lighting Model Selection system

This system is the one which allows the player to choose between pixel-based and vertex-based lighting models, and every model implemented (including the possibility of no lighting). This system is composed of two parts: the **Slider** and the **Switch Buttons**. Its appearance in the main interface can be seen at the **Figure 3**.

- The **Slider** allow the user to change between **"No Light"**, **"Phong"**, **"Lambert"** and **"Half-Lambert" Lighting Models** (without taking into account if they are pixel-based ot vertex-based). As the **Slider** moves, a **Text** element below the Slider shows the current lighting model name selected.
- The **Switch Buttons** allows the user to change between pixel-based computations or vertex-based computations. As the normal map is dependent of the lighting system, the selected field will also affect the calculations of this feature. As there can be only one option selected, the buttons will allow to select one of the options.

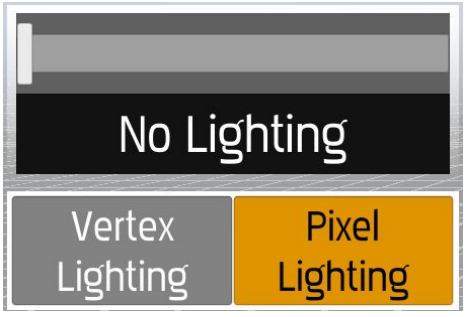


Figure 3. Lighting Model Selection System, inside the application.

2.4.4 - Shader Change Name system

This system is the easiest to understand: it uses an **Input Field** to receive the desired name for the exported shader. It can be changed anytime, and it will not affect another features of the shader. As the value is updated, the **Shader export system** will take care of saving the selected name.

2.4.5 - Scene Settings Parameter system

This menu is the one responsible of changing the colors used at the scene's skybox and light. An skybox is a representation of the infinite boundaries of the scene

(the sky, or the horizon, for example), and tweaking the parameters of the one used at the scene, we can simulate the desired work environment, and observe how the shader behaves on it. The provided controls for all the colors allow the user to select this color (with the **Color Picker Tool**), tweak its influence at the color calculations (with a **Slider** which limits are restrained from 0 to 1), visualize it (using an **Image** component), and reset it to white (using a **Button**). When a parameter is modified, it is updated in the **Shader Management System**. The visual design of this menu can be seen at the **Section 5.1.2.** This menu is composed of four components:

- **Sky Tint**: This is the base color of the sky in the scene, modified by the Atmosphere Thickness and the Exposure values.

- **Ground Color**: This is the base color of the ground in the scene, modified by the Atmosphere Thickness and the Exposure values.

- **Light Color**: This is the base color for the main light in the scene, which will modify the result of every Lighting Model (except the “No light” possibility).

- **Atmosphere Thickness**: This value is the one that tweaks the absorption of color in the scene (modifying the sky final color). It is controlled by a **Slider** with predefined limits (as the variation above or below these is deemed indiscernible).

- **Exposure**: This value tweaks the influence of the high refraction zone that happens at the horizon of the skybox (modifying the sky final color at the affected zones). It is controlled by a **Slider** with predefined limits (as the variation above or below these is deemed indiscernible).

2.4.6 - Main Feature Selection System

This system is made by a series of **Buttons** spread across the interface, which helps the player to open the different sub-menus and navigate through the different features. These **Buttons** can only be selected one at a time, and opening one menu will close the previously opened ones: the first time a Button is pressed, it will show an animation that will allow the user to see which menu have they opened. A **Text** element above the menu’s designated space informs the user about the title of the current opened menu. Every **Secondary Menu** will be opened inside the designated area for it, the right wing of the main interface. The appearance of all of these elements can be seen at the **Figure 4**. The available buttons are:

- **Basic Color / Texture Button**: By clicking on this **Button**, the **Main Texture Parameters System** will be opened.

- **Normal Map Button**: By clicking on this **Button**, the **Normal Map Parameters System** will be opened.

- **Open Lighting Model Button**: By clicking on this **Button**, the **Lighting Model Parameters System** will be opened, depending on the current selected Lighting Model.

- **Scene Settings Button**: By clicking on this **Button**, the **Scene Settings Parameters System** will be opened.

- **Exit Application Button**: By clicking on this **Button**, the Exit Application will open the **Exit Application Menu system** at the center of the main interface.

- **Shader Export Button**: By clicking on this **Button**, the **Shader Export system** is activated, and the **File Explorer tool** is opened.

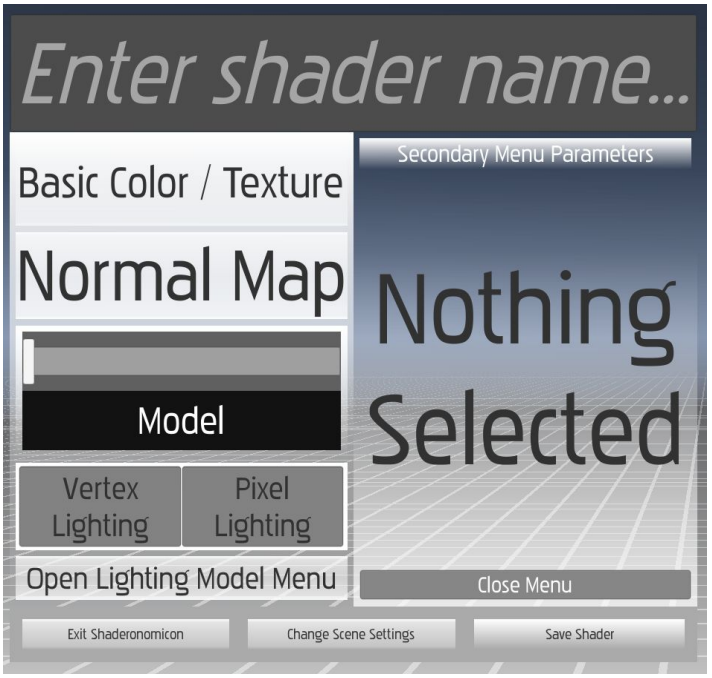


Figure 4. Main feature Selection System, inside the main interface, at the left wing of it.

2.4.7 - Exit Application Menu system

This system allows the user to exit the application, as one of its main goals is to remind him to save the created shader (if he desires) before the application is closed. This simple menu is made by three **Buttons** inside a window with a informative **Text,** and its appearance can be seen at the **Figure 6**:

- **Exit Application System**: If the user wants to exit the application, by clicking on this **Button**, the application will close, losing all the modification on the shader.

- **Close Exit Application Menu**: If the user does not want to exit the application, by clicking on this **Button** this menu will be closed, returning to the main interface.

- **Save Shader Button**: By clicking on this **Button**, the **Shader Export system** is activated, and the **File Explorer tool** is opened. After that, the application is automatically closed.

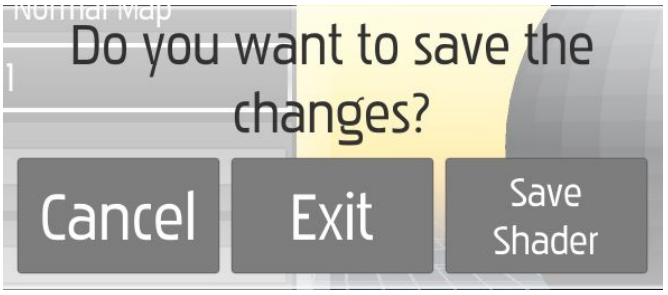


Figure 5. Exit Application Menu's visual design

3. Shaderonomicon's Tools

3.1 - Introduction

All of the systems of Shaderonomicon, specially the feature and scene oriented ones, are made by tools, a sub-system that cannot work by itself, so it must be implemented in the right framework to be of use. The sub-system's structure depends on its given purpose, but as a rule, any of the tools does not consist of two or more possible controls, to be easier to understand and use by a novice user. In this section, every tool's mechanic will be thoroughly explained, for a better understanding of how do they work.

3.2 - Color Picker

This is the most complete tool of the application, as it involves **color/coordinate conversion component** (using shaders (nvidia CG) and C#) , a **mouse position Detection Component**, and another tools such as **Images**, **Text** and **Buttons**. The color conversion involves converting HSV color values to RGB, and vice-versa, as the Color Picker uses two main panels, the HSV panel, and the HUE panel. Its visual design can be seen at the **Section 5.2.1.**

3.2.1 - HSV and RGB color representation models

From the start, it was decided that the **Color Picker tool** would be based on a HSV color interface, as it is the most spread workframe in this type of tools: not only the user would be more familiar with this system, but the acquisition of the desired color for the shader would be easier than using a RGB system. To understand better this decision, I recommend this article[16] , as it is very well explained.

3.2.2 - Color Picker's workflow

The workflow of the **Color Picker tool** follows this scheme:

1. When a **Color Selection Button** is pressed, the **Color Selection tool** saves the desired color to be changed, and updates the **Current Color Image Display**. Then, the **Color Picker window** opens.
2. The user can select the desired color from the **HSV panel**, using the **mouse position Detection component**.
3. The user can change the HUE component from the **HUE panel**, **using the mouse position detection component.** When a new color is selected, the **HSV panel** is refreshed with the new HUE value.
4. When the user has selected a new color, he can save his selections by clicking on the **Apply Button**, which will assign the new color to the previously saved color, and close the **Color Picker window**. If the user does not want to save the changes instead, he can cancel this assignment by clicking on the **Cancel Button**, which will only close the **Color Picker window**.

3.2.3 -HUE and HSV display shaders

Making use of the HSV to RGB conversion formulas found online[17], and taking advantage of the properties of the UV coordinates of the shader, The HSV is quite easy to implement, as we only need to assign the **Saturation** and **Value** parameters of the pixel's color to the X and Y coordinates of the UV map (which always go from 0 to 1), which we will multiply for the selected value at the **HUE display**. As for its shader, we will only have to use the Y coordinates to vary from 0 to 1, to 0 to 360, keeping the **Saturation** and **Value** parameters to 1. To show these colors we have to convert the assigned values to the RGB model, and then return the output of the operation to the fragment shader's function.

3.2.4 - Mouse position Detection Component

The current **mouse position Detection Component** works using four auxiliar objects, which are anchored to the corners of the desired detection area. This may seem primitive, but this way the detection area can update itself with the **multi-resolution component**. But the real aim of the use these corners, is to replace the necessity to obtain the pixel's color, by using the adapted values to obtain the relative color from that position. The workflow of this component is the following:

1. The user does a left click: if the **Color Picker window** is open, and the mouse position is within the corners, the program takes as the user wants to select a color.
2. Then, interpolating the value between the corners (in the X and Y axis), the program obtains the relative position, from 0 to 1 values.
3. The relative position is used to obtain the **Saturation** and **Value** parameters (in the **HSV panel**) and the **HUE parameter** (in the **HUE panel**).

To obtain a more intuitive interaction with the user, a little offset is added to the position of the corners, allowing to detect the position of the mouse in a little wider area (but the relative position's range do not change), to make the selection of the corners easier. Furthermore, two concentric circles show the position of the mouse while selecting the color, and points out the estimated position of the color in the **HSV panel**. Its analogous component in the **HUE panel** is a rounded rectangle, which shows the estimated HUE value selected.

3.2.5 - Current and New Color Display

These components are only made for user-feedback purposes. Every display is composed by **Text** (to show which color corresponds to its display), and **Image** tools (to show which color is selected / saved). Every **Image** is updated by the **Color Picker system**, as the window opens (**Current Color Display**) or the user selects a new color in the **HSV Panel** (**New Color Display**).

3.2.6 - Framework Buttons

This Buttons are made to ensure that the color selection is correct, or to avoid to change the current color selected. These make the user interaction to be slow-paced, and so reducing the possible errors that the user would make. Every **Button** of this framework closes the **Color Picker window**, but its purposes are opposites. These are the following:

- **Accept Button**: This **Button** saves the changes on the selected color, by using the **Shader Management system**.
- **Cancel Button**: This **Button** discards the changes on the selected color, maintaining its original value.

3.3 -Crosstale's File Browser

This is the only tool that was not programmed by me. Free from the Asset Store, the File Browser's plugin from **CrossTale** is the only framework that allowed this application to use the **built-in Windows File Browser**, and compile the application with this feature (this will be further explained in **Section 4.6** of this memory).

From this plug-in, we use to of the implemented components: the **File Selection Component** and the **Folder Selection Component**. Its code only provides the route of the selected item (folder or object) from the opened window, so we have to process it accordingly to its nature. And, in the case the user does cancel the selection, the exception needs to be taken into account. We will do this in both components. Its visual design can be seen at the **Section 5.2.2.**

3.3.1 - File Selection Derived Component

This component uses both functions from the **Crosstale's File Browser**, and from the built-in libraries of C#. Its aim is to provide the **Shader Management system** with the desired file from the computer, and process it for this system to use it correctly. The functions of this component are called following this scheme:

1. First, when this system is called, the external function from the plug-in opens the **built-in Windows File Browser**, and then the user can navigate through the files of the computer to choose the desired file.
2. When the user selects the desired file, this window is closed and the function returns the route of the file. Then, using the WWW class, we can extract or "download" into an scriptable object this file.
3. As every **File Selection** call in this application is bound to retrieve an image, we extract from the WWW object the texture that we will use in the project, and depending on the feature that called this component, the appropriate parameters are updated by the **Shader Manager system**.

3.3.2 - Folder Selection Derived Component

This component uses functions from the **Crosstale's File Browser**, and from the built-in libraries of C#. Its aim is to provide the **Shader Export system** with the desired folder from the computer, and process it for this system to use it correctly. The functions of this component are called following this scheme:

1. When the user wants to save the modified shader, the **Crosstale's File Browser** function opens the **built-in Windows Folder Browser**, for the user to select the desired folder of destination.
2. As the desired folder is selected, the window is closed and the function returns the route. Unlike the **File Selection Component**, we do not need to process the obtained route any further, as we provide the desired information to the **Shader Export system**.

3.4 - Shader File Scripting

The main tool used by the Shader Export System, which objective is to create a text file, fill it with the shader necessary instructions (following a predefined structure), and exporting it to the provided route. This tool makes use of the "System.IO" library from C#, and receives the crucial data from the **Crosstale's File Browser** (**Folder Selection Derived Component**), and the **Shader Management system**. Its workflow follows this design:

1. First, it receives the specified route from the **Crosstale's File Browser**.
2. Then, it creates the text file, using the value of the shader's name, provided by the **Shader Management system**, and using the ".shader" format.
3. Then, it fills the text file using this sequence:
    - 3.1. Right after opening the file, we copy the first part of the basic shader structure, using the obtained shader name, and paste it into the text file.
    - 3.2. Obtains all the values for all the parameters that the shader currently uses in its current state, thanks to the **Shader Management system**, and copies them into the text file.
    - 3.3. Then, depending on the current state of the shader, the program acquires the indispensable functions for the shader to work correctly, extracting them from the string-based function library, inside the **Shader Management system.**
    - 3.4. After copying the functions, we extract from the string-based function library, the vertex and fragment functions, and paste them into the text file.
    - 3.5. Lastly, we copy the second part of the basic shader structure, and close the **StreamWriter** object.

3.5 - Interface-exclusive Tools

These tools are exclusively used in the interface, and its only purpose is to link the player's input on different features, to the appropriate management systems. Its behaviour differs, as some of them does not provide direct interaction with the user, but to offer feedback from the system about the state of some features. The implementation structure of all the following tools is provided by built-in components of the Unity3D Canvas System.

3.5.1 - Slider

This tool's behaviour is simple: an interactable element moves inside a horizontal bar, and the return value is the relative position of this element, between the to limits of the bar. If provided with two **Input Fields** (like the **Shininess Management**) the two numerical limits of the bar can be changed, modifying the accessible values by the **Slider**. If that is not the case, the **Slider** will only access the numeric values between the two limits. An example of both cases is shown in the **Figure 6**.

Figure 6 .Left: Example of Slider with variable limits.
Right: Example of Slider without variable limits.

3.5.2 - Input Field

This tool's objective is to receive the input from the user, and update the related variable on its dependent system. Depending on its application, it will only allow to insert text or numerical values. It is usually implemented with a **Text** element, to show which element will be modified. The text inside the field can be selected, erased, and modified, and all of the values are updated in real time by the system that are using it. Its appearance can be seen in the **Figure 7**.

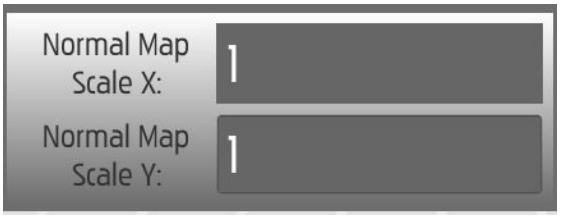


Figure 7. Two Input Fields, displaying the received values.

3.5.3 - Button

This is the most simple interactable tool: if the user clicks on it, an specified function will be called. Every **Button** has a **Text** element inside, referring to its purpose. It has two possible visual behaviours, which appearance can be seen in the **Figure 7**:

- Its appearance will change when the button is clicked, and when the mouse is over them.
- Its appearance will change once the button is clicked, and will not change until another button is clicked. (This is the case of the **Switch Buttons**).

Figure 8. Top: Example of Switch Buttons
Bottom: Example of normal Button.

3.5.4 - Text

This tool is used to inform the user, about other tools or the shader's current state. Its simple implementation provides a flexible yet strong support for the user's understanding of the interface, and hence, improving the user's experience with the application. An example of its use, is shown in the **Figure 9.**

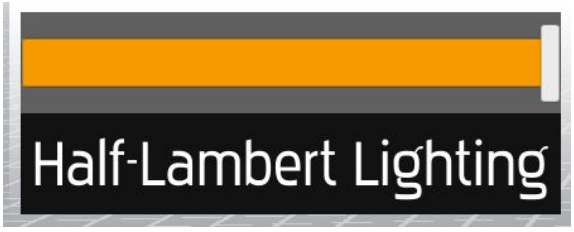


Figure 9. Text display showing the slider's state.

3.5.5 - Image

This tool is used to inform the user, about shader feature's state, or scene properties (as shown in the **Figure 10**). Its is used in all of the color or texture modifications, and its possible behaviour are the following:

- **Texture-oriented**: its texture property will update when the system that uses it adds, modifies or erases it.
- **Color-oriented**: its color property will change when the system that uses it modifies its value.

Figure 10. A button with its output assigned at the Image on the right.

3.5.6 - Multi-resolution component

This component allows the interface to resize itself if the screen's size varies in real time, maintaining the global proportions of all the items inside. This component is also the most complicated to build, as it needs to be carefully adjusted to every element in the interface. Its behaviour is controlled by the following components:

- **Horizontal / Vertical Layout Group:** this component is the one responsible of managing the size/proportion of its childs. Depending on the tweaking of its properties, it is possible to align the object following a simple pattern, or sharing a common anchor point, which is crucial for its resizing when the screen's size is modified.
- **Canvas Scaler Component**: This component is the one in charge of scaling the whole interface with the desired proportions (the ones established by its childs in the **Scene's Hierarchy**). This component must be ONLY in the parent gameobject of the whole GameObject, to ensure that the proportions are well saved.
- **Layout Element**: This component allows individual elements to reserve the desired size, inside their Horizontal or Vertical Group, overriding the offered proportions by these.

As it may seem easy to understand and modify, it is quite difficult to work with if you are accessing its parameters through a C# script. For example, its coordinates and dimensions will look the same if we access the parameters using a script, but they will vary when looking them from the editor's window (as they are correctly scaled depending on the screen). All of these unwanted behaviours will trigger various difficulties, as I explain in the next section.

4. Shaderonomicon Development Diary

4.1 - About the changes in the project's goals

The original goal of Shaderonomicon was to be a library of basic, yet powerful shader effects, to be used and edited by anyone, in an intuitive way. For this purpose, three consecutive versions of the tool were expected to be released, one more advanced than the previous one. As the development would prove challenging, I thought of some "time-savers" that would help the project to progress smoothly: the use of the **Standard Shader framework**, a powerful and scalable system from Unity3D, would help to set up the initial version rather fast; the use of the **Unity Canvas** components that allowed to quickly set up a prototype of the interface, allowing at the same time the interface to have the multi-resolution feature; and, if I were able to find a way to read an script as a text file, by creating a system that could read the functions from it, the process of exporting the shader would be faster.

For this purpose, I initially made a lot of research, not only on the nvidia CG language, but also on the implementation of the **Standard Shader framework**. After dozens of hours studying it, I concluded that this framework could not be edited in any way that I thought; moreover, the system to I thought to flexibly edit the shader by adding passes (by rewriting the shader, exporting it within the project, and then recompiling it) was quickly deemed non-viable (because of the inability to compile any **UnityEditor's library function**; this also made me discard the use of the **built-in Color Picker** and **File Explorer** tools, as they use this library).

So, after having consumed more than 10% of the time estimated to develop this project, I had to start the project's planification from scratch. If a library of shaders was impossible to develop at a short-term; the best I could do was to set the **right framework**, and the **basic properties**, for the project to be **extended and upgraded** in the future. And so, I established the current goal of Shaderonomicon: development of an application (only the final version), with the ability to edit the basic properties of a shader in real time, and to be able to export it to use it in a Unity project. I decided to **prioritize quality over quantity**, so the application would have less features, but it would be more polished.

Most of the systems and tools of this application have been refined and updated over the course of the project. The goal of this section is to show how the systems have changed, and the reasoning behind their implementation.

4.2 - About the development of the Modifiable shader

The Modifiable shader is the system that has changed many times throughout the development of this project. Being the target of most of the research time, the Modifiable shader shifted its structure from the Standard Shader, to a "sketch" custom

shader, with an all-purpose library. Also, as the research concluded that the library file could not be loaded as a text file, to process it by the **Shader Export System**, a constantly updated string-based copy of the library was needed to be able to export the functions correctly. Furthermore, as the research also concluded that it was not possible to recompile a shader in real time, another system to manage the fluctuation between possible states of the shader. Now, we will explain the reasoning behind all these decisions:

4.2.1 - Before the first version of the Modifiable shader:

The first contact of my research did not gave me much hope, as the bibliography that I could find about shaders in Unity was quite limited, being most of the books dedicated to the **Standard Shader**[22,23,24]. As I originally planned to use that framework anyways, so I did a quick glance over their content. The result was not as good as I expected: most of the variations that the books offered were choosing the different properties that the Standard Shader had already implemented. So, after that, I decided to look for more information online. I found quite instructive websites (which I will mention at the next chapter), but all of them did not went further into "**how does Standard Shader works**". Then, I decided to look into them myself, to study them in detail, and see if I could get something out from them.

As the Standard Shader provides a wide variety of functions and pre-built data structures, the more of them I found (as I found that the basic library, **UnityCG**[25], gets most of their functions from other sub-libraries, as **UnityCommonVariables**, **UnityShaderUtilities**, **UnityShaderVariables**, etc.), less flexible were their functions, so I had to study their dependencies, too. The more I searched for its basic calculations, the more functions I found that were purely untouchable (as they made reference to function calls that were not accessible for me). Also, as I find that the **shader properties** like the ability to cast shadows, are not included in the CGPROGRAM, and so they cannot be changed (unless I rewrite the shader from scratch, which I will conclude months later that it is not possible). As the edition of this **framework**, or the tweaking of its properties seems almost impossible (not only because of its structure, but also because I lack the ability to extract more information about it), I discard the idea of using this framework to develop the **Modifiable shader**.

4.2.2- First version of the Modifiable shader

Thanks to the research I did about the nvidia CG language, as well as the basic knowledge about shaders that I learnt from the subject "Computer graphics", as well as the useful websites that i found online (**wikibooks.org**[20], and **jordanstevenstechart.com**[21]), helped me implement the first version of the modifiable shader quite fast. But, as I implemented the normal mapping feature, and the phong lighting model, the shader began to show implementation errors. As my knowledge with debugging shaders in Unity was almost zero, the progress was halted. Then, I decided to separate the different parts of the shader into functions; this way, if the shader workflow was modular, it would be easier to pinpoint the cause of the malfunctioning. And it worked, as the problem was the miscalculation of one of the

input vectors (which was overwritten at two functions with different purposes). So, the first hurdle was overcomed. As I finished to implement all the lighting systems, the texture handling and the normal mapping, everything worked fine… on the editor debugging values; the controls from the main interface had suddenly stopped working. As I did not find a lead to where the problem could be, I searched for a solution, at the same time that I developed other systems that were not as advanced as they should be.

A week later, I found the cause of the problem, after hours and hours of searching in internet forums about it: as long as the variables were modifiable from the editor's shader window, the global values that were sent to the shader were overwritten by these. So, once I got rid of this feature, the application worked as intended.

4.2.3 - Development of the all-purpose library

As I was trying to debug the shader, I found that it was extremely difficult, even if the structure was more modular than having all the calculations altogether. Furthermore, I realised that the current system would not work if the main shader had to support more functions in the future. So I decided to put every function of the shader into an all-purpose library, which would contain all the functions necessary for the computation of the shader's functions. Using this structure, the shader would only need to call the vertex and fragment functions, from which all the necessary functions would be called. This decision made me realize that the implementation of the bump map[18] was not as sturdy as I thought: as I separated its implementation from the normal map calculations, a lot of errors popped up. So, after considering its real importance, and looking into the new system's possibilities (I would have to take into account if a bump map or not was provided, which increased the number of possible states of the shader), I decided to keep it simple, and erase it from the project.

As I proceeded to continue the expansion of the library, I found that the possibility of reading the script file as a text file from another script, was non-viable for this project. So, the simultaneous copy of this library, but converted to strings, was needed for the **Shader Export System** to correctly work. This would take a lot of the time dedicated to the library implementation, so the expected date of finishing all of the planned functions from the library was delayed.

As my research on the possibility of re-compiling a script concluded that it was not possible too, I searched for a way to manage the switch between different vertex and fragment functions in real time. After a couple of days, I found a solution: the State Machine Shader Component.

4.2.4 - Current version: State Machine Shader Component

The State Machine inside the all-purpose library would solve the problem: using custom vertex and fragment functions, as they would call the necessary functions by extracting the data from custom input and output data structures (this

process is thoroughly explained at the **Section 2.2.2.2**). This way, the desired implementation of the shader was accomplished, but there is still some bugs that I could not find a solution: applying a normal map, and then erasing it, would result a illumination error, as if it the light was "baked". In the future, when a new feature wants to be added, we have to follow these steps:

- First, adapt it to be accessible by using only a function (respecting the modular design of the library).

- Then, four variants need to done: a "no texture" variaint, a "no normal map" variant, a "no texture and no normal map" variant, and the "texture and normal map applied" variant. These are the basic sketches, which need different data structures. If one state is not possible, the new feature has to be disabled if the shader is in that state variant, to avoid future errors.

- Then, a copy of every new function has to be copied into the string-based library.

These are the steps necessary to work with the Modifiable shader; it will need more changes in other systems.

4.3 - About the Shader Export System

The **Shader Export system** is the second most important system of Shaderonomicon, as it is one of the main goals of this project. A lot of the total research time was dedicated to explore for different ways to implement it in a efficient and, if possible, simple way. Unfortunately, that has not been the case. Unlike the Modifiable shader, most of the time dedicated to this system was purely searching and analyzing different methods, and only when I had a good lead I decided to implement a prototype of the system. I've done it this way, because this system is greatly **dependent from** the **Modifiable shader** and the **Shader Management System**, as the output of this system is entirely based on these: if I constantly modify these system, then I would have to update this system, too. That was one of the main reasons that part of my research on this field was to look for a way to read a script as a text file. To get a hold of something that would assure me to obtain this result, is one of the things that I will continue to search, for future versions of this project.

The current version of the **Shader Export system** uses two tools: the **Crosstale's File Browser** (its **Folder Selection Derived Component**, to obtain the desired folder) and the **Shader File Writing tool**. This way, by accessing the **Shader Management system** (to acquire the modified values of the shader, and its current state), and the string-based function library from the **Modifiable shader**, we can write into one single file the shader, with its values tweaked by the user, and without having to export the **State Machine** (which would produce serious performance issues).

4.4 - About Camera movement, and Mesh Display Behaviour

4.4.1 - About the Camera Movement

The movement of the camera, and the mesh, were not a priority at the implementation of the application, as I was more worried about the shader and its features. So, when things went south (as I could not find a solution to the error I was facing, for example), I decided to not lose time, and implement the remaining systems. One of the first that I implemented following this, was the **Mesh Rotation System**. It may sound simple, but using Quaternions and working with them is not. As the mesh rotates, its reference axis also moves, and applying the same force will result in undesired behaviour.

After trying different tutorials, I decided to go the other way around: the user will move the camera around the mesh to observe how does the shader behaves in lit and unlit areas. Thus, following a great tutorial[19], we implement the camera movement as desired.

4.4.2 - About the Mesh Display Behaviour

The main interface, because of its design, has been difficult to work with: the display mesh has to be centered at its designed space all the time, but the center of that designed space is never the center of the screen. This has been the first problem I had to deal with, and it seemed quite easy to solve: by using a secondary camera to show the mesh in the designated space, and focus on it with this camera the display mesh in its center, it will always be centered. So, this new camera will render its output in a Render Texture, which will occupy the designated area for the mesh display, and will be cropped to prevent image deformation. As a temporary solution, it worked quite well.

But as I found out when I compiled the application, the Mesh Render would deform its output,. giving undesired results. So, after months using this method, I decided to revert this change, and return to the single-camera scene. This one resolved that problem, but brought more and more complicated problems. So, after a few weeks trying to solve them, I decided to use again the Render Texture: obtaining by code the four corners of the designated space, the image could be resized without undesired defromations.

4.5 - About the Color Picker Tool

This tool is one of the main reasons the planning started to fell apart. It was not because of its shaders, or its color conversion formulas, but the **mouse position Detection Component**, and its quirks with the **Multi-resolution component**. I will now

proceed to explain my foolish attempts to tame this component, until I found a work-around solution:

1. **First try, Direct Pixel Color detection**: As Unity provides an input value, that is the mouse position on the screen, I could convert its coordinates to match the ones in the canvas, and then detecting the correct color by using a deprecated function to obtain the color value on the selected pixel. Of course, it did not returned the desired value, as the conversion of the coordinates did not work well with the multi-resolution interface. Then, I searched for a method that would not base its behaviour in coordinates, but the texture displayed by the **Image**.

2. **Second try, obtain the color's position using a Texture2D**: After searching for different methods online, I found one that seemed to work for anyone. That was not my case, at first because it used a texture, and in my case I use shaders to display the colors. So, to solve this hurdle, i could render the shader's result into a Texture2D, and then obtain the color using the relative position of the mouse inside the detection panel. This worked to an extent, as the selected color was shifted by an offset relative to the mouse position. I tried tweaking it many times, rewriting the formulas all over again, but my efforts were in vain.

3. **Third try, current version**: This version does not use the mouse position to obtain the pixel's color: instead, it uses four corners that contain the detection area, and when the user clicks inside, the position is used to calculate a relative value (constrained between 0 and 1), which the program use to **actually compute the color using a color conversion formula**. As it now achieves its purpose without errors, I consider this a success.

4.6 - About the File Explorer Tool

This tool is the one in which I actually spent the most time figuring out how to implement it, and the main reason the original planning fell completely apart (aside from the Standard Shader research, read more about this at **Section 4.1**). I will now explain every version made of this tool, and why I decided to use an external plugin:

- **First version, using UnityEditor's functions**: This was an easy and quite fast implementation of the File Explorer, as the functions opened the **built-in Windows File Browser**, and returned the desired route. The main problem with this version was that the application could not be compiled, as for using UnityEditor in runtime is impossible. So this first version was quickly discarded.

- **Second version, using the Directory Class**: As the rapid implementation of the first version encouraged me to implement the File Explorer, I did

some research about how C# handles these. I found how to access folders, and select files, so I implemented my own version of a File Browser using a Canvas component called Scroll Rect, which would be filled with the folders and files of the current route, and update itself as I navigated through the computer's directories… until I found an exception when I tried to access some specific folders. I searched for a solution for a long time, trying all of the alternatives that could work (changing the iteration through the directories to a recursive-based iteration), but none of them were successful. As the number of hours spent on this tool kept rising above the desired number, I decided to spend some time searching for an external plug-in to solve this problem once and for all.

- **Current version, Crosstale's File Browser**: As I tested many plugins that guaranteed to work in runtime, this plugin was the only one that actually worked. Its use is quite simple, which sped up the implementation, and is easily accessible by any script, so it is perfect for this purpose (to find a more detailed explanation, please read the **Section 3.3**).

5. Interface Design and Evolution

5.1 - Main interface

As the sketches were made in light of the first objectives, the main interface was initially divided into four screens: the **Pass Edition Window** (which would allow the user to edit the properties of the desired pass of the shader), the **Pass Manager Window** (where the user would be able to swap the order of the created Pass), the **Export Window** (where the user would see how the shader is exported, and change its name before saving it), and lastly, the **New Project Window** (the initial screen of the application, where the user could choose to create a new shader from scratch, or edit one of the available ones in the library). But, as we mentioned before, in the **Section 4.2.1**, the use of the Standard Shader was deemed non-viable for this project, so the whole interface design could not be implemented. So, I decided to merge two of the main windows, and create the **Main Interface**, where the user would modify the properties of the shader, and export it.

5.1.1 - [Sketch] Pass Edition Window

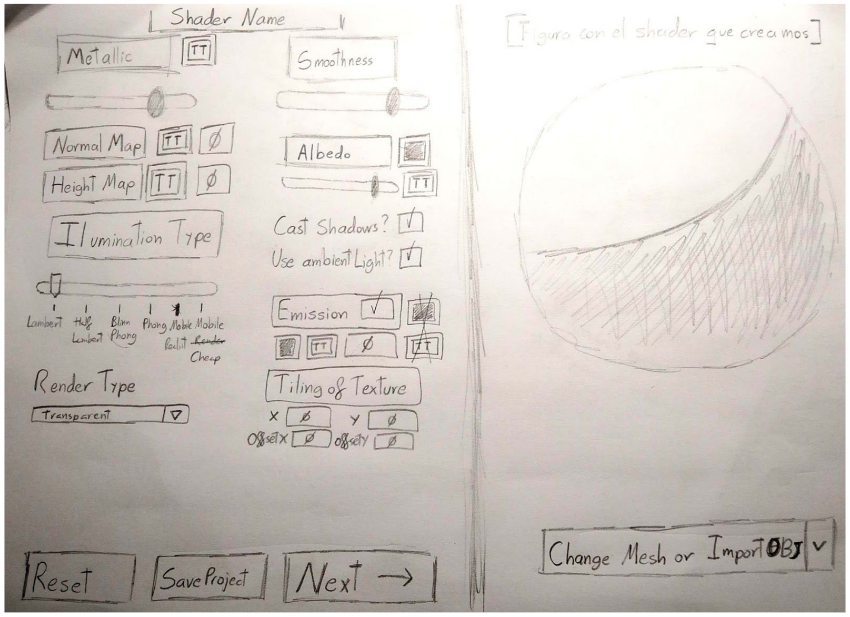


Figure 11. Interface sketch of the main window

As this sketch (**Figure 11**) was made by the time I had not discovered that I could not use the Standard shader framework, most of the properties that I planned to implement were the ones available from the editor (which I thought I could extract and modify). Its distribution has not greatly changed between the two interfaces, as the right half of the interface has not been modified. The **Parameter Edition Segment** has suffered a lot of changes. I will talk about them in the next chapter.

5.1.2 - Main Interface Window

Figure 12. Screenshot from the final design of the main interface

The main interface, as I said before, is the result of the merge of two of the designed windows: the **Pass Edition Window**, and the **Export Settings Window**. This way, the resulting interface (**Figure 12**) had quite different distribution than the previous **Pass Edition Window**, enlarging the **Shader Change Name System**, and dividing the distribution of the **Parameter Edition Segment** into two parts: the **Main Feature Selection** (the left wing) and the **Secondary Menu Parameters** (the right wing).

- **Main Feature Selection**: This part of the interface is responsible in calling the desired secondary menu. Its objective is to set the basic properties of the shader, and to edit these basic properties, the adequate parameter menu will appear in the **Secondary Menu Parameters** section.

- **Secondary Menu Parameters**: In this part of the menu, the features mentioned in the **Section's 2 and 3** will emerge, allowing the user to modify them. Its designs are the following (**Figure 13**):

Figure 13. Final design of all the secondary Menus. From left to Right, from top to bottom: **Phong Lighting Parameters Menu, Normal Map Parameters Menu, Base Texture Parameters Menu, Lambert Lighting Parameters Menu, and Scene Settings Parameters Menu.**

5.2 -Auxiliar interfaces

Most of the shader settings are colors or textures from the user. To pick these in a intuitive and easy way, the user can access auxiliar interfaces, made for this purpose. Their internal structure has already been defined and explained in the previous sections, so we will now focus on its visual appearance and utility.

5.2.1 - Color Picker Tool and its evolution

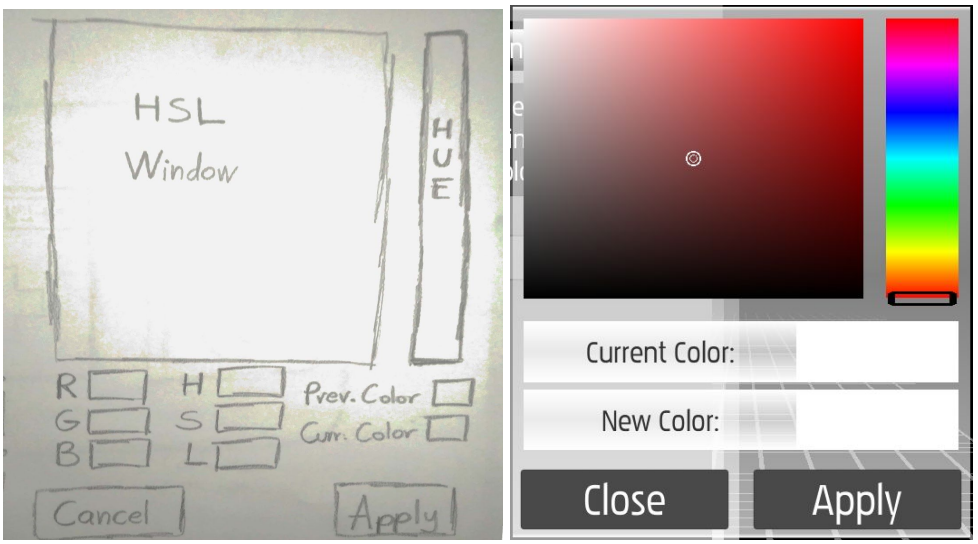


Figure 14. Left: Interface sketch of the Color Picker Window.
Right: Final Design of the Color Picker Window.

This interface was always a tricky one to design, as it needed to have a necessary amount of features to be useful. First, I designed the interface, having into account that the user could insert the values at every color channel, in both color models, as well as select it using the **HSV Panel** and the **HUE panel (Figure 14, Left)**. But as I included those **Input Field**, the tool's interface seemed overwhelming: the simplification of the available controls, as we can see at the final design, was spot on **(Figure 14, Right)**. My mistake at the first sketch was not taking into account the actual size of the tool inside the application (which it is quite small).

5.2.2 - File Explorer Tool

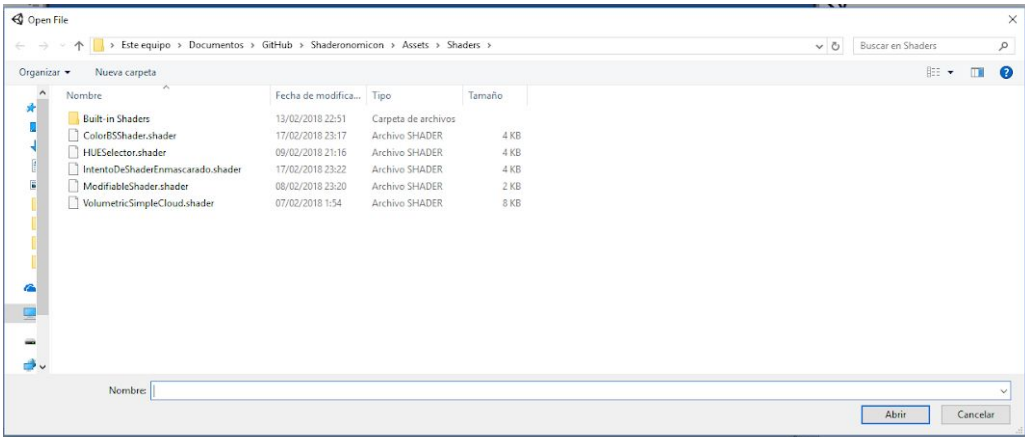


Figure 15. Windows File Explorer Window

The **default Windows interface (Figure 15)** is the best option to navigate through directories. The **user familiarity** with this window, and so the facility of use, make this option the best available. As the second version of this tool did not have a defined graphic design, it is unnecessary to be displayed here, although its **versability** was its weak point (it could navigate only by folder proximity); the **built-in Windows**

interface is quite wise, as it lets the user take advantage of their own shortcuts (without having to modify anything from the project).

5.3 - Future interfaces

As the draft for the different available windows were made with the original goals in mind, after redirecting the project's aim, we decided to merge some of the windows inside the main interface, and discard others (as their functionality would not be implemented). For reference purposes, these sketches are kept in this chapter, with their short description.

5.3.1- Pass Manager Window

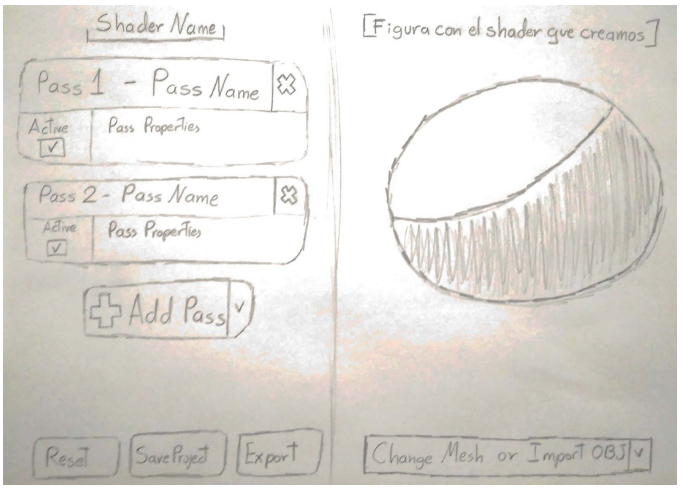


Figure 16. Interface sketch of the Pass Manager Window

This window (**Figure 16**) provides the user the possibility of having many passes at the same shader. More than one pass allows to create more complex effects increasing the functionality of the application. The Pass Order Display shows all the Passes on this project. The user can move, rename, activate/deactivate or erase a Pass once created. He can also save his project, reset recent changes, rename the shader, or go to the **Export Settings Window**.

5.3.2 - Export Settings Window

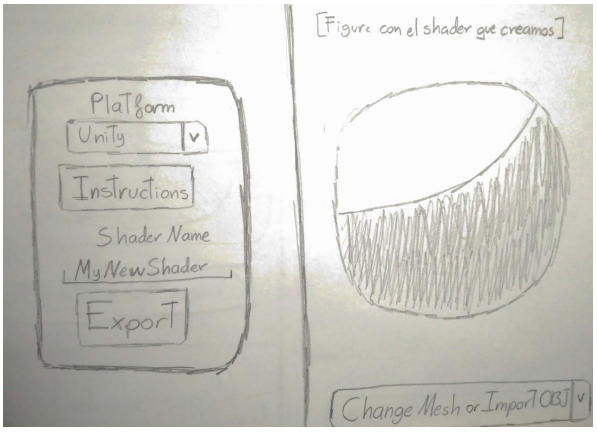


Figure 17. Interface sketch of the Export Settings Window

At the Exports Settings Window (**Figure 17**), the user can export the created shader to the selected platform, rename it, or read the instructions to import the shader at the designed platform. The user can see the Shader Display at the right of the screen, showing the created shader before confirming its exportation.

5.3.3 - New Project Window

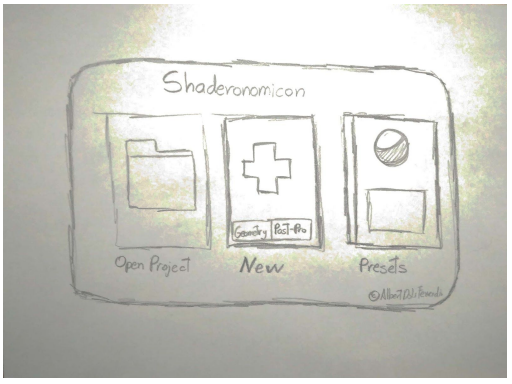


Figure 18. Interface sketch of the New Project Window

This window (**Figure 18**) has three buttons: Open an existing project, choose and edit a built-in preset, or creating a new shader. If the user opens an existing project, after loading the custom save file, the application will open the Pass Editor Window. If the user wants to edit a built-in preset, a new window will appear showing the different presets available. After selecting one, the application will open the Pass Editor Window. If the user wants to create a new shader, he will have to choose between two options: create a post-processing shader, or a geometry shader. After selecting the desired option, the application will open the Pass Editor Window.

6. Project Results

In this section, the current state of every system and tool will be displayed, as well as examples of different states of the application. This section is divided in different chapters: **Initial Documents Results**, **Tools and Canvas Results, Shader Edition Results** and **Shader Export Results**.

At first, while the first tools were being developed (first versions of the Color Picker and File Explorer), the **Technical Proposal Document** and the **Research and Design Proposal** were written. All the sketches from the **Section 5** were included in the second document, and the **Section 1** is based in the first one (although this part was severely modified as the project evolved into its current state, because of the supervisor's advice and the change in the projects aims).

6.1 - Tools and Canvas Results

All of the systems and tools, regarding scene-oriented ones, are completely functional: The camera can rotate around the object and change its zoom in a smooth movement (only when the player has clicked on the right half of the screen); the mesh can be changed at anytime, and is displayed always in its designated space no matter the movement of the camera, as shown in the **Figure 19**.

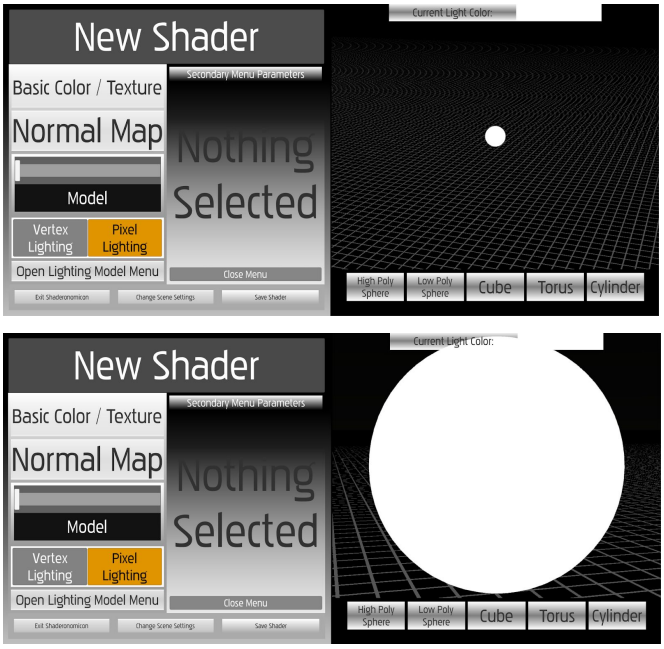


Figure 19. Examples of the camera's different position and zoom

The **Color Picker Tool** successfully selects the desired color from the user's input, and displaying the selection in the appropriate panel, as we can see in the **Figure 20**.

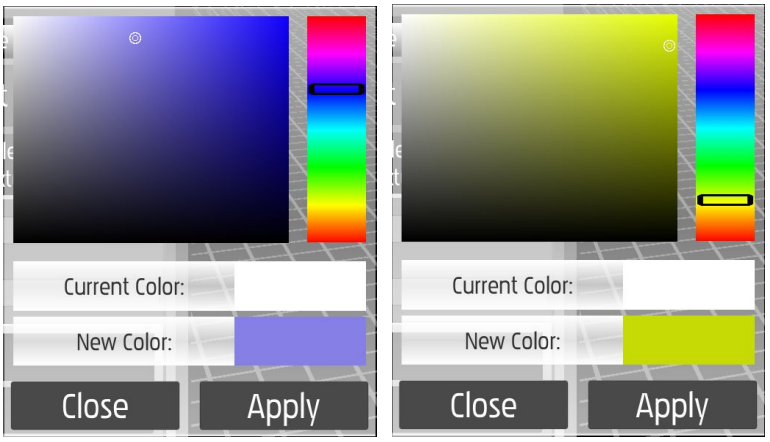


Figure 20. Example of different color selections.

Regarding the **CrossTale's File Browser**, the built-in Windows File Browser works properly, and is capable of selecting the correct file from the user's input, and process it to be used in the application (**Figure 21**).

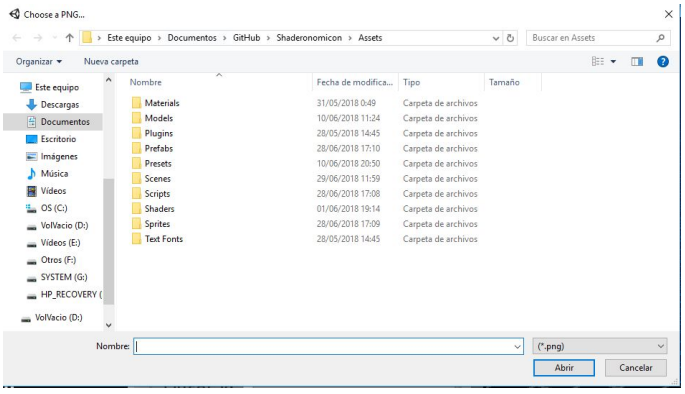


Figure 21. Example of the built-in Windows File Browser.

Lastly, the **Scene Settings Parameter system** works smoothly, allowing the player to change the environment settings to obtain a more personalized feedback of the shader (**Figure 22**).

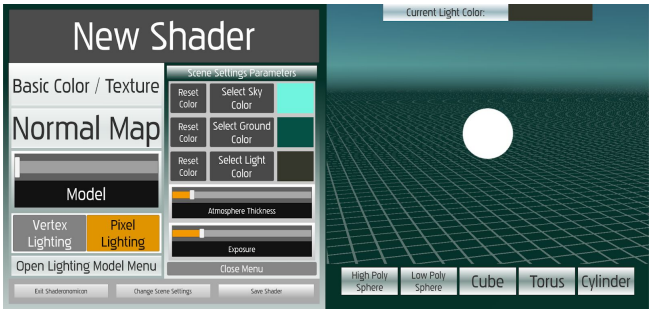


Figure 22. Example of a change in the application's environment..

6.2 - Shader Edition Results

The **Shader Management System** and **all of the feature-based systems** of Shaderonomicon have been thoroughly tested, and work as intended. The shader can

display any of its many features as they intertwine. Because of its properties, the normal map with vertex lighting will not behave as well as when its applied with per pixel lighting, and the visual output will greatly depend on the target mesh. In **Figures 24, 25, 26** and **27** the mesh displayed is the High-Poly Sphere, to obtain the best results. In **Figure 23**, the texture and the normal map used are shown. The program used to create the normal map is CrazyBump[28].

Figure 23. Left: Texture used in the Figures 25 and 26. Right: Normal map used in the Figures 26 and 27

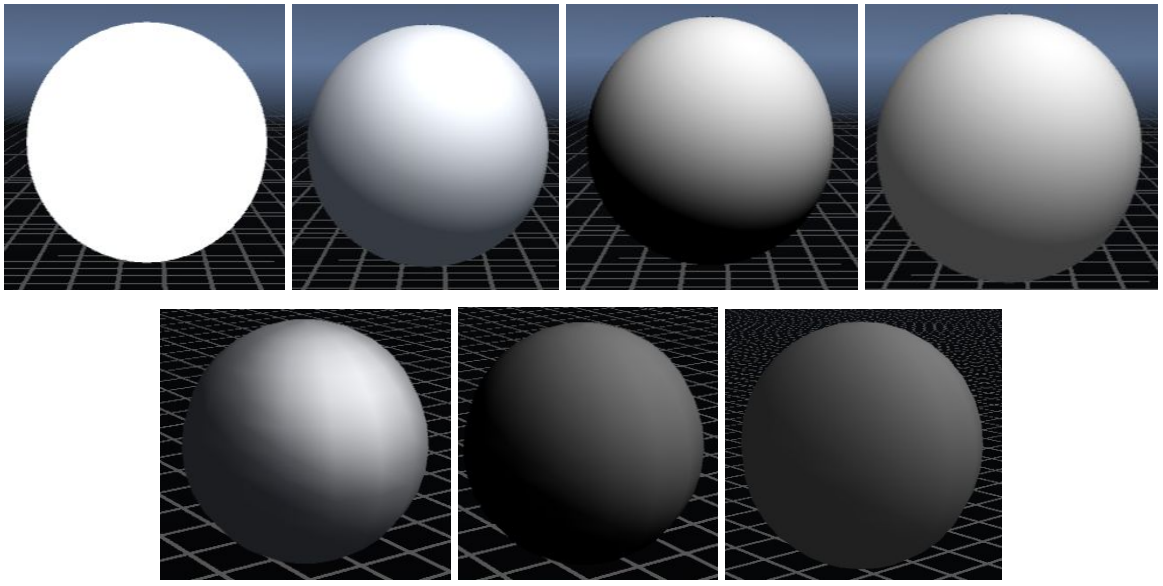


Figure 24. From left to right, from top to bottom, all shaders have no texture nor normal map: shader with no light, shader with Pixel Phong, shader with Pixel Lambert, shader with Pixel Half-Lambert, shader with Vertex Phong, shader with Vertex Lambert, shader with Vertex Half-Lambert.

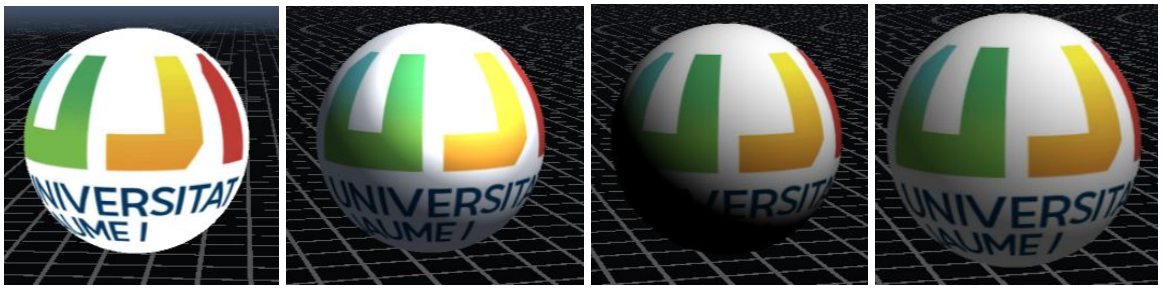


Figure 25. From left to right, all are shaders with texture and Pixel Lighting Models: No Light, Phong, Lambert, Half-Lambert.

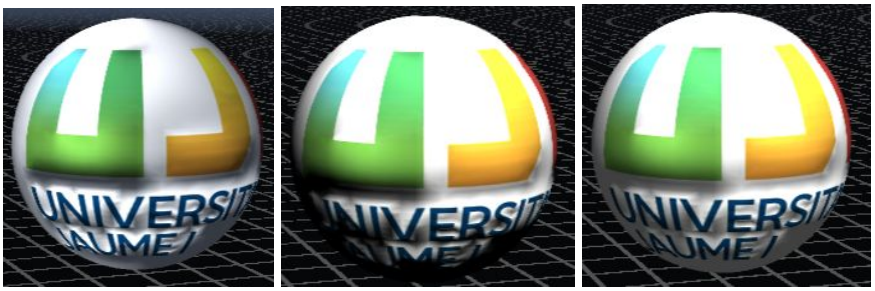
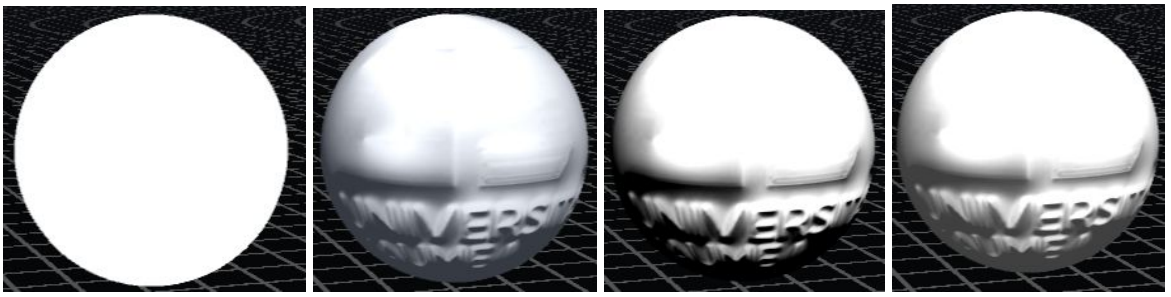


Figure 26. From left to right,from top to bottom, all are shaders with texture and normal map: No Light, Pixel Phong, Pixel Lambert, Pixel Half-Lambert, Vertex Phong, Vertex Lambert, Vertex Half-Lambert



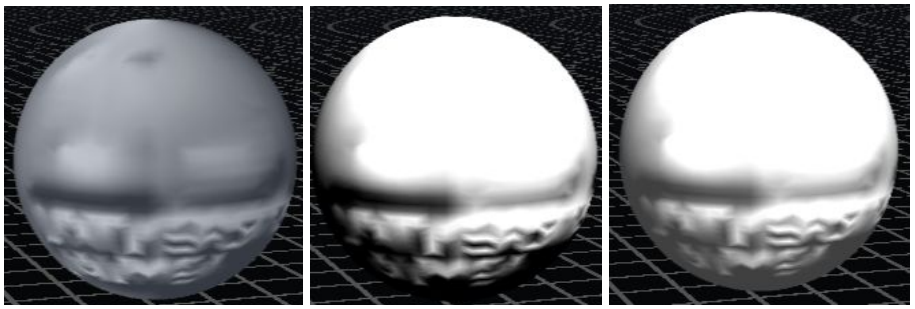


Figure 27. From left to right,from top to bottom, all are shaders without texture and normal map: No Light, Pixel Phong, Pixel Lambert, Pixel Half-Lambert, Vertex Phong, Vertex Lambert, Vertex Half-Lambert

6.3 - Shader Export Results

This system have been fully implemented, but currently does not work completely well. For now, it can generate a compilable shader depending on the selected parameters by the Shader Management System, and values provided by the user, but it does not work as desired inside an external project. The text generated by

the program, when no texture nor normal map is selected, a custom color applied to the Half-Lambert Lighting Model, using Per Pixel Lighting, and the provided name “Testcustomlambert”:

```
 Shader"Shaderonomicon/Testcustomlambert"
{
 SubShader
 {
  Blend SrcAlpha OneMinusSrcAlpha
 Pass
{
 Tags { "LightMode" = "ForwardBase" }
 LOD 100
CGPROGRAM
  #pragma vertex vert_PerPixelLighting_NoTextureNoNormalMap
  #pragma fragment frag_PerPixelLighting_HalfLambert_NoTextureMapNoNormalMap
 #include "UnityCG.cginc"
 #include "UnityLightingCommon.cginc"
     uniform fixed4 _TextureTint = float4(1 , 1 , 1 , 1 ) ;
 uniform float _LambertTintForce = 1 ;
 uniform float4 _LambertTintColor  = float4(0.3210825 , 0.8309469 , 0.8393108 , 1 ) ;
 struct vertexInput_NoTextureNoNormalMap  {  float4 vertex : POSITION;
 float3 normal : NORMAL;
 };
 struct vertexOutput_NoTextureNoNormalMap_PerPixelLighting  {  float4 pos : SV_POSITION;
float3 posWorld : TEXCOORD0;  float3 normalDir : TEXCOORD1;  float3 normal : NORMAL;  };
 struct vertexOutput_NoNormalMap_PerPixelLighting
 {  float4 pos : SV_POSITION;
 float3 posWorld : TEXCOORD0;
 float3 normalDir : TEXCOORD1;
 float3 normal : NORMAL;
 float2 tex : TEXCOORD2;
 };
 float3 HalfLambert_Lighting_Pixel_NoNormalMap(vertexOutput_NoNormalMap_PerPixelLighting
input)
 {  float3 normalDirection = normalize(input.normalDir);
 float3 viewDirection = normalize(_WorldSpaceCameraPos - input.posWorld.xyz);
 float3 lightDirection;
 float attenuation;
 if (0.0 == _WorldSpaceLightPos0.w)
 {  attenuation = 1.0f;
 lightDirection = normalize(_WorldSpaceLightPos0.xyz);
 }  else
 {  float3 vertexToLightSource = _WorldSpaceLightPos0.xyz - input.posWorld.xyz;
 float distance = length(vertexToLightSource);
 attenuation = 1.0 / distance;
 lightDirection = normalize(vertexToLightSource);
 }  float3 NDotL = max(0.0, dot(normalDirection, lightDirection));
 float HalfLambertDiffuse = pow(NDotL * 0.5 + 0.5, 2.0);
 float3 finalColor = HalfLambertDiffuse * attenuation * _LightColor0.rgb;
 return finalColor;  }


 vertexOutput_NoTextureNoNormalMap_PerPixelLighting
vert_PerPixelLighting_NoTextureNoNormalMap(vertexInput_NoTextureNoNormalMap input)
 {  vertexOutput_NoTextureNoNormalMap_PerPixelLighting output;
 float4x4 modelMatrix = unity_ObjectToWorld;
 float4x4 modelMatrixInverse = unity_WorldToObject;
 output.posWorld = mul(modelMatrix, input.vertex);
 output.normalDir = normalize(mul(float4(input.normal, 0.0), modelMatrixInverse).xyz);
 output.normal = input.normal;
 output.pos = UnityObjectToClipPos(input.vertex);
 return output;  }
```

```
 float4
frag_PerPixelLighting_HalfLambert_NoTextureMapNoNormalMap(vertexOutput_NoTextureNoNormal
Map_PerPixelLighting input) : COLOR
 {  vertexOutput_NoNormalMap_PerPixelLighting dummyOutput;
 dummyOutput.posWorld = input.posWorld;
 dummyOutput.normalDir = input.normalDir;
 dummyOutput.normal = input.normal;
 dummyOutput.pos = input.pos;
 return float4 (HalfLambert_Lighting_Pixel_NoNormalMap(dummyOutput).xyz *
(_LambertTintColor * _LambertTintForce).xyz * _TextureTint.xyz, 1.0);
 }

 ENDCG
 }  }  }
```

And lastly, a shader that uses both normal map and texture, and Phong with vertex Lighting and custom colors, with the name “New Shader”:

```
 Shader"Shaderonomicon/New Shader"
{ Properties  {
 [NoScaleOffset] _NormalMap(" Normal Map ", 2D) = "bump" {}
 [NoScaleOffset] _CustomTexture("Main Texture ", 2D) = "white" {}
 }
 SubShader
 {
  Blend SrcAlpha OneMinusSrcAlpha
 Pass
{
 Tags { "LightMode" = "ForwardBase" }
 LOD 100
CGPROGRAM
  #pragma vertex vert_PerVertexLighting_Phong
  #pragma fragment frag_PerVertexLighting
 #include "UnityCG.cginc"
 #include "UnityLightingCommon.cginc"
     uniform sampler2D _CustomTexture;
 uniform fixed4 _TextureTint =  float4(1 , 1 , 1 , 1 ) ;
 uniform float _TextureTileX = 3 ;
 uniform float _TextureTileY = 3 ;
 uniform float _OffsetTileX = 0 ;
 uniform float _OffsetTileY = 0 ;
 uniform sampler2D _NormalMap;
 uniform float _NormalTileX = 3 ;
 uniform float _NormalTileY = 3 ;
 uniform float _NormalOffsetX = 0 ;
 uniform float _NormalOffsetY = 0 ;
 uniform half _NormalMapScale  = 2 ;
 uniform float _CustomShininess = 0.6644058 ;
 uniform float4 _PhongAmbientColor =  float4(0.8669783 , 0.3017504 , 0.3017504 , 1 ) ;
 uniform float _PhongAmbientForce = 1 ;
 uniform float4 _PhongSpecularColor =  float4(0.0183183 , 1 , 0.9615222 , 1 ) ;
 uniform float _PhongSpecularForce = 1 ;
 uniform float4 _PhongDiffuseColor =  float4(0.3017504 , 0.3574262 , 0.8669783 , 1 ) ;
 uniform float _PhongDiffuseForce = 0.8847263 ;
struct vertexInput_AllVariables  {  float4 vertex : POSITION;
 float3 normal : NORMAL;
 float2 texcoord : TEXCOORD0;
 float4 tangent : TANGENT;  };
struct vertexOutput_PerVertexLighting  {  float4 pos : SV_POSITION;  float4 col : COLOR;
float2 tex : TEXCOORD1; };
 float4 Texture_Handling_Vertex(vertexOutput_PerVertexLighting input)
```

```
 {  float2 texCoordsScale = float2 (_TextureTileX, _TextureTileY);
 texCoordsScale *= input.tex.xy;
 float4 textureColor = tex2D(_CustomTexture, texCoordsScale + float2(_OffsetTileX,
_OffsetTileY));
 textureColor = textureColor * _TextureTint;
 return textureColor; }
 float3 Phong_Lighting_Vertex(vertexInput_AllVariables input, float3 normalDirection)
 {
        float4x4 modelMatrix = unity_ObjectToWorld;
        float3x3 modelMatrixInverse = unity_WorldToObject;
        float3 viewDirection = normalize(_WorldSpaceCameraPos - mul(modelMatrix,
input.vertex).xyz);
        float3 lightDirection;
        float attenuation;
        if (0.0 == _WorldSpaceLightPos0.w)
        {
                attenuation = 1.0;
                lightDirection = normalize(_WorldSpaceLightPos0.xyz);
        }
        else
        {
                float3 vertexToLightSource = _WorldSpaceLightPos0.xyz - mul(modelMatrix,
input.vertex).xyz;
                float3 distance = length(vertexToLightSource);
                attenuation = 1.0 / distance;
                lightDirection = normalize(vertexToLightSource);
        }  float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb *
_PhongAmbientColor.rgb;
        float3 diffuseReflection = attenuation * _LightColor0.rgb *
_PhongDiffuseColor.rgb * max(0.0, dot(normalDirection, lightDirection));
        float3 specularReflection;
        if (dot(normalDirection, lightDirection) < 0.0)
        {
                specularReflection = float3 (0.00001, 0.00001, 0.00001);
        }
        else
        {
                specularReflection = attenuation * _LightColor0.rgb *
_PhongSpecularColor.rgb * max(0.0, dot(reflect(-lightDirection, normalDirection),
viewDirection));
                specularReflection = specularReflection * _CustomShininess;
        } return (ambientLighting * _PhongAmbientForce) + (diffuseReflection *
_PhongDiffuseForce) + (specularReflection * _PhongSpecularForce);
 }
 float3 Normal_Direction_With_Normal_Map_Handling_Vertex(vertexInput_AllVariables input)
 {  float4x4 modelMatrix = unity_ObjectToWorld;
 float4x4 modelMatrixInverse = unity_WorldToObject;
 float3 tangentWorld = normalize(mul(modelMatrix, float4(input.tangent.xyz, 0.0)).xyz);
 float3 normalWorld = normalize(mul(float4(input.normal, 0.0), modelMatrixInverse).xyz);
 float3 BitangentWorld = normalize(cross(normalWorld, tangentWorld) * input.tangent.w);
 float3 biNormal = cross(input.normal, input.tangent.xyz) * input.tangent.w;
 float2 normalCoordsScaled = float2(_NormalTileX, _NormalTileY);
 normalCoordsScaled *= input.texcoord.xy;
 normalCoordsScaled += float2(_NormalOffsetX, _NormalOffsetY);
 float4 encodedNormal = tex2Dlod(_NormalMap, float4(normalCoordsScaled.xy,0, 0));
 float3 localCoords = float3(2.0 * encodedNormal.ag - float2(1.0, 1.0), 0.0);
 localCoords.z = 1.0 - 0.5 * dot(localCoords, localCoords);
 float3x3 local2WorldTranspose = float3x3(tangentWorld, BitangentWorld, normalWorld);
 float3 normalDirection = normalize(mul(localCoords, local2WorldTranspose));
 normalDirection = float3(_NormalMapScale, _NormalMapScale, 1.0f) * normalDirection;
 return normalDirection; }
 vertexOutput_PerVertexLighting vert_PerVertexLighting_Phong(vertexInput_AllVariables
input)
 {  vertexOutput_PerVertexLighting output;
 float3 normalDirection = Normal_Direction_With_Normal_Map_Handling_Vertex(input);
```

```
 output.col = float4(Phong_Lighting_Vertex(input, normalDirection), 1.0f);
 output.pos = UnityObjectToClipPos(input.vertex);
 output.tex = input.texcoord;
 return output;  }
 float4 frag_PerVertexLighting(vertexOutput_PerVertexLighting input) : COLOR  {  float4
TextureColor = Texture_Handling_Vertex(input);
 return float4(input.col.xyz * TextureColor.xyz , 1.0f);
 }
 ENDCG
 }  }  }
```

6.4 - Shaderonomicon's Video Demostration, and executable

This project is always accessible by using its github repository, https://github.com/al315151/Shaderonomicon, and it is ready to work in a compiled version following this link[29].

For this project, I have developed a video that shows the application's performance, and an example of creating a shader from scratch using Shaderonomicon. The music used is from Jay Man, called "Wonder And Magic"[30]. The link to the video is the following: https://youtu.be/RRv0o15Hlh4

7. Conclusions

7.1 - Goal Accomplishment

This project's goals have been changed over its development, being the ones in the **Section 1.4** the second version. Although its content were almost the same as its predecessors, their application was quite different: as the priorities behind them shifted from being shader-centered, to being focused on the application's development. Aside from this remark, the goals have been clearly completed. I will explain now step-by-step.

1. "Developing a tool to teach developers with zero experience with shaders, the basics of shader editing without writing a line of code." As the basic shader properties have been implemented, it is possible to squeeze all its potential without prior knowledge of the application.

2. "Developing a tool that exports the created shaders to Nvidia CG." Because the **Shader Export System** has been fully implemented, this goal is cleared.

3. "Customizing of the basic features that usually involves shader edition (texture, normal map, lighting model, etc)." Because all the systems included in **Sections 2.3 and 3.1 to 3.4** have been fully implemented this goal is completed.

4. "The application interface can be understood without prior knowledge, and users can start creating shaders without explicit instructions." Thanks for the **systems and tools** developed at the **sections 2.4 and 3.5,** this goal can be cleared.

5. "The exported shader works correctly in an external Unity3D project." This feature is currently being debugged, as in some cases it shows undesired behaviour, although a compilable shader can be correctly exported.

7.2 - Project's Planning Deviation

As the original planning was supposed to cover 300 hours, the time dedicated to the different systems and tools has differed greatly from that value. These are the estimated, and real time values for each task:

Task	**Task Completion Time**
Reference Research: Bibliography, competitors, shader frameworks, etc.	38 hours
Implementation of tools:	

• **Color Picker Tool**	32.5 hours
• Implementation	15 hours
• Research	1 hour
• Shader developing	3.5 hours
• Bug fixing	13 hours
• **File Explorer Tool**	20.5 hours
• Implementation	9.5 hours
• Research	7 hours
• Bug fixing	4 hours
• **Shader Edition System**	100 hours
• Implementation (C# framework)	3.5 hours
• Shader development (nvidia CG)	21.5 hours
• Library function development (both nvidia CG and string-based)	50 hours
• Bug fixing	25 hours
• **Shader Export System**	25 hours
• Implementation	21 hours
• Bug fixing	4 hours
• **Project's Technical Proposal**	4 hours
• **Mesh Related System**	14.5 hours
• Mesh visualization System	3 hour
• Mesh Swap System	2.5 hours
• Camera Rotation System	9 hours
• **Scene Parameters System**	6 hours
• **Exit Menu System**	1 hour
• **Shader Change Name System**	1 hour
Project's Research and Design Proposal	6.5 hours
Project's Memory	45 hours

Project's Presentation	7 hours
Project's video	[Not developed yet]
Project's Main Interface Development	34 hours
• **Shader Development**	2 hours
• **Implementation**	30 hours
• **Multi-resolution canvas**	2 hours
Organization of project's assets (relocation and removal of obsolete assets and code)	9 hours

7.3 - Future work: What comes next?

Because all of the problems that I have encountered while developing Shaderonomicon, some of the features that could have been implemented (such as post-processing shader effects, feature animation system, shader pass editor) were deemed non-viable for the short-term development of the application. But, in the future, I would like to expand it by adding those systems, to transform Shaderonomicon into what it was supposed to be from the beginning: a library of basic effects, that could be easily modified and understood by non-programmers; a tool to introduce more people to the "magic" of computer graphics, as this field has a potential that is greatly underused and underestimated by novice videogame developers.

Because of the problems that still hinder the application, if they are not solved in a short-term period an extensive overhaul of Shaderonomicon will be made: as its public target are the novice developers, the application will run inside their projects, using the Unity Editor as its base, and the users will be able to see the changes inside the shader in their own project and environment, enhancing the received feedback.

8. Bibliography

1. Shader Graph [26/06/2018]: https://unity3d.com/es/shader-graph
2. Unity3D homepage [26/06/2018]: https://unity3d.com/es
3. Definition of Texture in computer graphics context [26/06/2018]: https://en.wikipedia.org/wiki/Texture_mapping
4. Normal mapping [26/06/2018]: https://en.wikipedia.org/wiki/Normal_mapping
5. Definition of shader [26/06/2018]: https://en.wikipedia.org/wiki/Shader
6. Microsoft Visual Studio 2017 [26/06/2018]: https://www.visualstudio.com/es/downloads/
7. Adobe Photoshop Free Trial (version used) [26/06/2018]: https://www.adobe.com/es/products/photoshop/free-trial-download.html
8. Crosstale's File Browser [26/06/2018]: https://crosstales.com/en/portfolio/FileBrowser/
9. Google Docs [26/06/2018]: https://www.google.es/intl/es/docs/about/
10. Blender's homepage [26/06/2018]: https://www.blender.org/
11. Substance homepage [26/06/2018]: https://www.allegorithmic.com/products/substance-designer
12. Shaderfrog homepage [26/06/2018]: https://shaderfrog.com/app
13. Shadertoy homepage [26/06/2018]: https://www.shadertoy.com/
14. Phong Lighting System (Phong Reflection Model) [26/06/2018]: https://en.wikipedia.org/wiki/Phong_reflection_model
15. Lambert Lighting System (Lambert and Half-Lambert Reflectances) [26/06/2018]: https://en.wikipedia.org/wiki/Lambertian_reflectance
16. Differences between RGB and HSV [26/06/2018]: **https://www.quora.com/What-are-the-differences-between-RGB-HSV-and-CIE-Lab**
17. Color Conversion Formulas [26/06/2018]: https://gamedev.stackexchange.com/questions/59797/glsl-shader-change-hue-saturation-brightness/59808#59808
18. Bump mapping [26/06/2018]: https://en.wikipedia.org/wiki/Bump_mapping
19. Tutorial for camera movement implementation [26/06/2018]: https://www.youtube.com/watch?v=bVo0YLLO43s
20. [26/06/2018] https://en.wikibooks.org/wiki/Cg_Programming/Unity
21. Lighting Models implementation sketch [26/06/2018]: **http://www.jordanstevenstechart.com/lighting-models**
22. Standard Shader Unity page [26/06/2018]: https://docs.unity3d.com/Manual/SL-SurfaceShaders.html
23. Standard Shader Parameters Explanation [26/06/2018]: https://docs.unity3d.com/es/current/Manual/StandardShaderMaterialParameters.html
24. Standard Shader snippets library [26/06/2018]: https://docs.unity3d.com/Manual/SL-ShaderPrograms.html
25. Download Page of Unity's Built-in Shaders [26/06/2018]: https://unity3d.com/es/get-unity/download/archive
26. Unity Shaders and Effects cookbook - ISBN: 9781849695084
27. The Cg Tutorial : The Definitive Guide to Programmable Real-Time Graphics - ISBN: 0321194969
28. CrazyBump homepage: https://www.crazybump.com/
29. Shaderonomicon Executable: https://drive.google.com/open?id=1aQHwLC0GcCd0IszBa-NQgzj3S4a73YFd
30. Presentation video music: https://www.youtube.com/watch?v=jZ1UJ5v7iFc