UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Didactic tool for the development of shaders

Joan Bellavista Bartroli

Bachelor Thesis

Specialization in Computing

Thesis advisor: Marta Fairén

Thesis coadvisor: Àlvar Vinacua

January 18, 2021

Contents

Contents

List of tables

List of figures

1 Introduction

1.1 Introduction and contextualization

Computer graphics is one of the fastest growing fields in computer science. That is, in part, thanks to the increasing attention that videogames are receiving these days. It is also a really difficult subject to master, and one of the hardest parts is programming shaders, since it is radically different from programming anything else. It requires knowledge of the graphics pipeline and imagination to be able to use shaders at their maximum potential. Shader programming requires a depth knowledge of the graphics pipeline because they work mainly in parallel for each primitive of the graphical model. In order to take maximum potential of this programming it is also desirable to have a broad knowledge of vector calculation.

This thesis is the development of a web application which has two main objectives. It would aid students in the development of shaders and it would help professors to teach shader programming in a dynamic and practical way. Since the application will have to display 3D scenes, we will use WebGL (1).

This web application is meant to be used as a text editor where the user can experiment with shaders, writing shaders and seeing the results in real time.

1.1.1 Context

This is a Bachelor Thesis of the Computer Engineering Degree, specialization in Computing, done in the Facultat d’Informàtica de Barcelona of the Universitat Politècnica de Catalunya, directed by Marta Fairén and Àlvar Vinacua.

1.1.2 Concepts

WebGL

WebGL (1) is an API (2) that most browsers natively offer. It lets you use the OpenGL graphical pipeline (latter explained). Ultimately it offers the programmer a way to display 3D graphics in real time.

Graphical Pipeline

The graphical pipeline includes the steps that a graphical system performs to render a 2D / 3D scene.

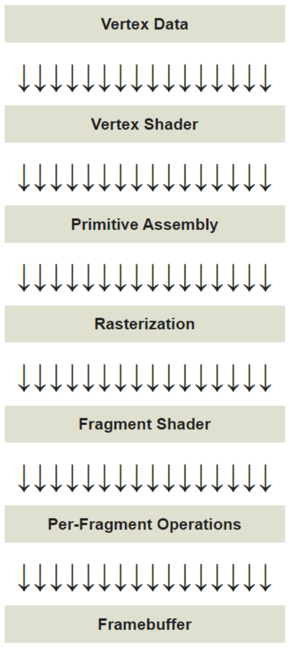


Figure 1

To aid the understanding of the concept, Figure 1 (3) shows all the steps. It is not necessary to know every step to understand this thesis, so there will only be description of the relevant stages.

Vertex specification is the necessary information to describe a concrete 2D / 3D object (Vertex Array Object[1]). The next rendering step is the vertex shader, which is executed for each vertex and is responsible for modifying the vertex position and / or calculate values for the next rendering steps. The last relevant step is the fragment shader, which is executed for each fragment (43) of the screen. From the fragment shader we can change the final color of each fragment. This is incredibly useful for simulating light, or creating complicated procedural textures.

1.1.3 Problem to be solved

The problem is the high difficulty to deeply understand how shaders work and how to program them properly. It is a problem for students but also for teachers.

Programming shaders require not only to understand the theory and syntax of the language, but also to be creative and imaginative. Some of the intuition that the students have gathered programming with ordinary languages, doesn’t apply with shaders. That is because what you program in, for example, a fragment shader is not executed only once, but once for each fragment of the screen. This unique way of programming is difficult, and the best way to deeply understand it, is practicing and seeing examples.

[1] VAO is an OpenGL Object that stores all of the state needed to supply vertex data (38)

1.2 Justification

1.2.1 Existing tools

- Any IDE or text editor which can be used to implement shaders.
- Custom software that currently is being used in “Gràfics” to develop shaders (named “GLarena”).
- Visual scripting (like the node-based programming of shaders in Blender or Unity (4))
- Online text editor The Book of Shaders (5).

1.2.2 Justification

The bast majority of text editors don’t offer that many shader specific features. And the ones that do, the features are not that impressive. For example, visual studio has an extension (6) that enables syntax highlighting and code completion. Which is helpful, but doesn’t make the coding any more dynamic.

In the other end we have visual scripting, which is the most dynamic way to program shaders. Usually the visual scripting is node-based, and this enables the opportunity to add a lot of really interesting features like changing variable values through sliders instead of rewriting the number, or displaying a preview of the partial render of each node.

Even though visual scripting looks like the best way to implement shaders, it has its drawbacks. When making complicated shaders, the node system may get out of hand, so it doesn’t really scale that well. Furthermore, visual scripting is a high-level approach to programming. So, it has the drawbacks of high-level languages like losing the ability to communicate with hardware or low-level system libraries.

After researching the tools that are available for shader programming, I found that there aren’t many applications that are a middle ground between those two options. This work is meant to occupy this gap.

To sum up, the tool developed in this work is a text editor but with shader specific utilities. This will make the workflow of implementing shaders much more dynamic, without losing touch with the low-level utilities. It’s also aiming to be educational, since it’s the main objective of this tool.

1.3 Objectives

As explained in the previous section, the tool is a more dynamic text editor specialized in shaders. This dynamism is achieved with features such as a display showing in real time the scene, or the automatic widget insertion into the text (explained in section 7.2.2.5).

Theoretical part

- Study of possible frameworks and comparison between them (React, Angular…).
- Study of WebGL library and GLSL ES 2.0 (shading language).

Practical part

- Design of the graphic interface.
- Program the text editor
    - Implement a display of the scene
    - Implement automatic widget insertion into the text.
    - Implement syntax highlighting.
    - Extra features

1.4 Methodology and rigor

1.4.1 Methodology

Software development methodologies are a really important part of a project. The larger the project, the more important the methodology becomes.

I have opted to use an agile methodology since it's versatile and dynamic. Most specifically, scrum methodology, which is an agile methodology that works around sprints (7) which are a time-boxed period when a team works to complete an amount of work.

1.4.2 Monitoring tools and testing

The monitoring is done with GitHub (8) repository. Git (9) is a great tool for version control, and GitHub provides us with free cloud repository. This ensures that changes can be rolled back.

In the other hand, web testing is also really important. It is a software practice that ensures the correct functionality of the app and ensures that changes don’t break the application.

2 Development plan

2.1 Description of tasks

2.1.1 Task definition

2.1.1.1 Project planning

The first task is planning the project. Given that this is a large task, it’s easy to divide it in various subtasks:

- **Contextualization and project scope**: Describe the project scope, justification and contextualization.
- **Temporal planning**: Describe the tasks and show time dependencies between them.
- **Economic management and sustainability**: Analysis of the costs and sustainability of the project.
- **Integration in final document**: All the previous planning tasks will be integrated into the final document.
- **Meetings**: There have been meetings with the directors of the thesis to receive feedback.

2.1.1.2 Theorical Part

Before starting to develop the web application, it has been necessary to choose the right tools and understand them. The subtasks are as follow

- **Study of possible frameworks and comparison**
- **Study of WebGL library and GLSL ES 1.0**

2.1.1.3 Practical part

After the study, the development of the web application can start. Given that the goal of this thesis is to develop an application, the practical part is the one where most of the efforts have been directed to.

Design of the graphic interface: Designing the UI to have high usability, so it’s easy to navigate through.

Program the text editor: This task will be divided into

- **Display of the scene**: Implementing a display which will show the 3D / 2D scene. The scene will be rendered using the shaders that the user is coding. It will also update every time the user modifies the code.
    - **Scene manipulation**: Implementing rotate, zoom and pan to manipulate the 3D scene.
    - **Importing 3D models** (.obj): Implement the importing of 3D models into the scene.
- **Text area**: Implementing an area where the user can type the code. There are some added features, which are the following tasks:
    - **Automatic widget insertion into the text**: Implementing a system which in some situations will replace the text written by the user with useful widgets. For instance, when typing a "float" the number is replaced with a slider, so you can change the value of the variable without typing.[2]
    - **Syntax highlighting**
    - **Extra features**

General features

- **Examples ready to use**: Create examples for the students to play around.

2.1.1.4 Documentation and conclusion

Write the documentation of the web application. Explaining all the features and how to use them. It also contains the justification of technical decisions. The conclusion of the thesis is also important, commenting whether the thesis has achieved its goals.

2.1.2 Summary of the tasks

The following table (Table 1) shows the summary of the tasks. For each task it presents the number of hours and the dependencies it has.

The dependencies that exist between tasks are rather logical.

[2] The way the widget insertion is explained, is not the way it will be implemented. It will be implemented parsing the users code, there will be no insertions. But I believe it's easier to understand the way is written.

We can’t start the theorical part until the project is planned, and we can’t start the practical part until I have a grasp on the tools I will be using to develop the product (theorical part).

It’s also worth noting that the creation of examples ready to use, shouldn’t be started until everything else of the practical part is done, since the examples will be built upon the web application.

The preparation of the oral exposition shouldn’t start until the rest of tasks are done.

In the Table 1 there is a hierarchy of tasks. For example, T10 (Practical Part) is the more generic task, but inside this task which has an estimated time of 420 hours we have various tasks that distribute these 420 hours. The same happens with for example T16 (Text Area) which contains T17, T18 and T19.

Id	Task	Time(h)	Dependencies
T1	**Project planning**	40	
T2	- Contextualization and project scope	10	
T3	- Temporal planning	10	
T4	- Economic management and sustainability	10	
T5	- Integration in final document	5	
T6	- Meetings	5	
T7	**Theorical Part**	30	T1
T8	- Study of possible frameworks and comparison	15	
T9	- Study of WebGL library and GLSL ES 1.0	15	
T10	**Practical part**	420	T7
T11	- Design of the graphic interface	20	
T12	- Program the text editor	300	T10
T13	- Display of the scene	100	
T14	- Scene manipulation	70	
T15	- Importing 3D models	30	
T16	- Text area	200	
T17	- Automatic widget insertion into the text	150	
T18	- Syntax highlighting	20	
T19	- Extra features	30	
T20	- General features	100	T11, T12
T21	- Examples ready to use	100	
T22	**Documentation and conclusion**	40	
T23	- Documentation	30	
T24	- Conclusion	10	
T25	**Oral Exposition**	10	T10

Table 1

2.1.3 Changes to the plan

While working on the thesis, the planning was followed. The time spent on each task was approximately what was estimated. However, there were some added tasks:

2.1.4 Resources

2.1.4.1 Human resources

The resources that have been needed in this thesis are the following.

The most obvious are the developers: The UI designer has designed the appearance of the web page. Front end developers have implemented the web page. The developer specialized in OpenGL / WebGL, has helped in the implementation of the web page.

Marta Fairén and Àlvar Vinacua have mentored the developers.

The GEP tutor, Joan Sardà, also helped in the planning of the project.

2.1.4.2 Material resources

The software and hardware used to develop the application:

- Atenea: Used to communicate with the professor in charge of GEP.
- Github: Software used for the control of version of the code written, ensuring to not lose any progress.
- IDE: More specifically Webstorm (10), since I have some experience with it and works really well for web projects.
- Computer: The development and testing of the web application has been done with a computer with 16 Gb of RAM, Intel Core i7-9700K CPU, GeForce RTX 2070 SUPER GPU.

2.2 Gantt

The Gantt chart is attached below (Figure 2). Since the tasks are organized in a hierarchy, I used colors to represent which level of the hierarchy the tasks are from.

Gantt Chart

ACTIVITY	PLAN START	PLAN END
Project planning	21/09/2021	29/09/2021
Theorical Part	29/09/2021	5/10/2021
Practical part	5/10/2021	13/12/2021
- Design of the graphic interface	5/10/2021	9/10/2021
- Program the text editor	9/10/2021	28/10/2021
- Display of the scene	9/10/2021	28/10/2021
- Scene manipulation	9/10/2021	20/10/2021
- Importing 3D models	20/10/2021	28/10/2021
- Text area	28/10/2021	6/12/2021
- Automatic widget insertion into the text	28/10/2021	1/12/2021
- Syntax highlighting	1/12/2021	3/12/2021
- Extra features	3/12/2021	6/12/2021
- General features	6/12/2021	13/12/2021
- Examples ready to use	6/12/2021	13/12/2021
Documentation and conclusion	21/09/2021	15/12/2021
- Documentation	21/09/2021	13/12/2021
- Conclusion	13/12/2021	16/12/2021
Oral Exposition	15/12/2021	16/12/2021

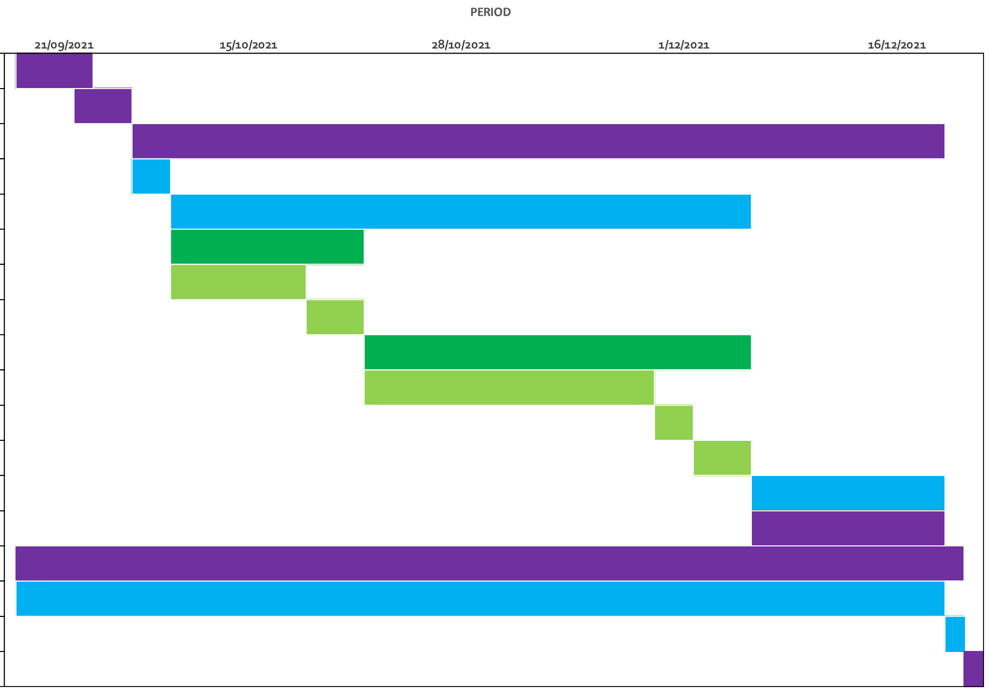


Figure 2

3 Budget

3.1 Staff costs

When talking about costs in a project about software development, the first that comes to mind are the personnel costs. Since it will be where most of the budget will go to.

To make a good approximation of the staff costs, we first need to define all the activity profiles, the cost per hour of each professional hired[3], the distribution of the tasks[4] and the cost of all the staff[5].

Although this project has been completed by myself and the tutors, I will explain the distribution of the tasks as if I had a team of professionals.

There are three main roles. First, we need a project manager to organize and plan the project. Then we need the UI designer, which will create mock-ups of the application for the developers to implement. It's important the work of the designer so the interface is easy to use and intuitive.

We also need the developers, which are responsible for the web application. The front-end developer will implement according to the designs. The software engineer will implement the more technical tasks (WebGL related). Most of the tasks have a little bit of front-end and WebGL so the professionals will have to work together.

Titles	**Cost (€/h)**
Project Manager	35[6]
Junior Front-End Developer	30[7]
Junior Software Engineer	21.65[8]
UI Designer	32.50[9]

Table 2: Cost per hour of each role

[3] Table 2: Cost per hour of each role
[4] Table 3: Distribution of tasks
[5] Table 4: Total cost of the staff
[6] (40)
[7] (42)
[8] (39)
[9] (41)

Task	**Time(h)**	**Project Manager**	**Front-end**	**Software Engineer**	**UI Designer**
Project planning	40	40	0	0	0
Theorical Part	30	0	15	15	0
Study of possible frameworks and comparison	15	0	15	0	0
Study of WebGL library and GLSL ES 1.0	15	0	0	15	0
Practical part	420	0	200	200	20
Design of the graphic interface	20	0	0	0	20
Program the text editor	300	0	100	100	0
General features	100	0	100	100	0
Documentation and conclusion	40	0	20	20	0
Documentation	30	0	15	15	0
Conclusion	10	0	5	5	0

Table 3: Distribution of tasks

Role	**Hours**	**Cost (€)**
Project Manager	40	1400
Junior Front-End Developer	235	7050
Junior Software Engineer	235	5087
UI Designer	20	650
Total	530	14187

Table 4: Total cost of the staff (without social charges)

3.2 Generic costs (GC)

A part from the mentioned staff costs, there are other costs that are not as relevant but that have to be taken into account.

Amortization of the resources

In this thesis most of the software used is open source, so the costs are mostly on hardware. Here we will calculate the hardware amortization.

As said in previous sections, the number of hours dedicated to the thesis has been roughly 530. I have been working with the same desktop computer the totality of the hours. To compute the amortization, we use the following formula (Equation 1):

$$Amortization = \frac{ResourcePrice * TotalHoursUsed}{LifeSpan}$$

Equation 1

Note that *LifeSpan* must be in hours.

In the following table (Table 5) we have the amortization costs.

Hardware	**Cost (€)**	**Life span (hours)**	**Amortization (€)**
Desktop Computer	1600	14600[10]	58

Table 5: Amortization costs

Indirect costs

We also should take into account costs not directly related to the project like the internet connection.

Since we are in pandemic, we have been working from home. So, I will describe the costs related with tele-working.

Internet cost: Without internet the project cannot be done so it's important to mention it. The average internet monthly fees are around 90€. Since the thesis has been done throughout 3 months, the final cost would be (Equation 2)

$$\frac{90€}{1\ month} * 3\ months = 270€$$

Equation 2

Electricity cost: As happens with internet, electricity is also necessary. In this case we will not look at the monthly fees but the price of kWh. The average price is 0.1479 €/kWh[11]. The power of the computer used in this thesis is 650 W. The final cost is calculated in (Equation 3).

[10] The life span of a desktop computer using it every day 8 hours a day is 5 years (14600 hours)
[11] Extracted from: (46)

$$0.650kW * \frac{0.1479€}{1kWh} * 530\ hours = 50.95€$$

Equation 3

Generic cost of the project

To summarize the generic cost (Table 6: Generic cost)

Description	Cost (€)
Amortization	58
Internet	270
Electricity	50.95

Table 6: Generic cost

3.3 Deviations of the budget

Contingency

All the cost calculated are without taking into account possible setbacks. It could have been some form of misestimation of the duration of some tasks, which means more spending on salary. Another example could be the increase on electrical bills pricing, which would increase the GC. That's why, due to the nature of the generic costs, which don't vary that much, I define the contingency margin to 5%. On the other hand, the staff cost is more likely to diverge from the predicted, so a 10% contingency margin seems adequate.

Incidental costs

In the following table (Table 7) we will estimate the incidental costs, which will be computed from the possible risks mentioned in previous sections.

Incident	Estimated cost (€)	Risk (%)	Cost (€)
Deadline of the project and inexperience in the field	500	20	100
Coronavirus	0	80	0

Total	500	-	100

Table 7: Incidental costs

Final budget

Next, we will summer all the previous sections (Table 8)

Activity	**Cost (€)**
Staff cost	19578[12]
CG	378.95
Contingency margin	1437.65[13]
Incidental costs	100
Total	16103.6

Table 8: Final Budget

3.4 Management control

We've analyzed the part of the budget dedicated to incidences (previous section), the defined a methodology used to detect when we are deviating from the budget has been the following.

For each task, we will calculate the deviation (Equation 4). We will then use the deviation to see whether we are over the budget or not. If the deviation is positive, it means that we've spent less than predicted. But if the deviation is negative, we've gone over the budget, hence will have to use part of the incidental and contingency budget.

$$deviation = estimatedCost - realCost$$

Equation 4

[12] Staff cost added the 38% of social charges in Spain

[13] 14187 * 0.1 + 378.95 * 0.05 = 1437.65

4 Sustainability

4.1 Self-assessment

There's no denying that technology has helped human kind in every area. It's also true that some technology improvements have done more harm than good. In a world where there are more than 7.7 billion people, it's important that all projects, even the small ones, take into account the sustainability.

Making sure that the Bachelor Final Project is sustainable, makes the students more responsible about the economic, social and environmental footprint that the thesis may have.

In all honesty, I probably wouldn't have questioned myself the questions presented in the poll, and it helped me understand the importance of trying to make a sustainable product and not mindlessly do projects which may harm the society.

4.2 Environmental impact

4.2.1 PPP (project put into production)

Since the project is the development of a web application, it barely has any environmental impact. The only impact is with the light consumption of the computers that will be used for the implementation. To reduce that, the best option is to improve the development speed. This can be done in numerous ways. The principal idea is trying not to "reinvent the wheel". For this exact philosophy, this project will use existing frameworks and libraries which will aid the implementation of the product.

4.2.2 Exploitation (Life expectancy)

Since the environmental impact of text editors is in general negligible, it's difficult to say if my solution improves or worsens the existing competition. Either way the environmental impact of using a text editor is so low that the answer loses relevance.

Also, since the final product is meant to be a web application, which will be downloaded and locally served, the server and the client will be the same computer. This means that there won't be any servers that need to be maintained.

To sum up, there won't be any type of maintenance, so there will be no costs once the application is in production.

4.3 Economic impact

4.3.1 PPP (project put into production)

The material costs have been 52€ of amortization for the computer used to develop the application. The cost of the staff, would have been the estimated (19578€), if the project had been done by a professional team.

Since there hasn't been any changes on the planning, and the approximation of hours per task has been accurate, the expected cost is similar to the final one.

4.3.2 Exploitation (Life expectancy)

The final product is meant to aid students, so it's aimed at being didactic. This means that it won't affect the market, hence economically will not improve current solutions.

Since this project won't have any maintenance during its lifetime, it won't have maintenance cost. However, there may need to be repairs or adaptations during the project. Since users may find bugs or may offer feedback to improve the application.

4.4 Social Impact

4.4.1 PPP (project put into production)

The project brought me the opportunity to learn a lot of really interesting subjects. Starting with the framework which I will develop the web application with. I will also learn WebGL, which is an API that browsers offer natively to render 3D graphics.

The project brought me the opportunity to learn a lot of interesting subjects. Studying the frameworks I used to develop the application, WebGL, and graphics computing in general.

4.4.2 Exploitation (Life expectancy)

The social impact of this web application is one of the main reasons I started it. The tool is meant to help computer science students learn the hard subject of graphics, and simultaneously aid the teachers to explain difficult concepts with ease. In previous sections, I discuss current alternatives to my product. And to sum up, I believe that what my product has to offer goes beyond what you can find in other applications, and furthermore, it’s improvements regarding the competition are relevant in didactic environments.

The requirements that had been planned for this web application have all been successfully met, so the tool works as intended. Thus, the students can understand shaders in a very dynamic way, which helps them figure out the hardest concepts to learn.

5 Knowledge integration

In the development of this thesis, has been applied knowledge of some subjects of the degree:

Introducció a l'Enginyeria del Software (IES) (11)**:** How to properly design a big application using object-oriented designs.

Compiladors (C) (12)**:** Understanding parsers and how to optimally use the resulting structures of the parsing.

Gràfics (G) and Introducció a disseny d'interfícies (IDI) (13)**:** Has been applied some user interface good practices. Has been used knowledge about shaders and the OpenGL pipeline from G.

Projecte Aplicat d'Enginyeria (PAE) (14) **i Projecte de Programació (PROP):** The experience of developing a large project: organization, version controls, testing…

Teoria de la Computació (TC) (15)**:** The understanding of the limitations of some languages (like regular expressions and free context grammars), to make conscious decisions about which technique to use in different contexts.

Also, to develop such a large application, the understanding and correct use of the design patterns from the book "Design patterns elements of reusable object-oriented software" (16) has been useful.

6 Laws and regulations

Many applications have to take into account data protection and privacy laws. Like "Llei Orgànica de Protecció de Dades de Caràcter Personal" (LOPD), or "Reglament General de Protecció de Dades" (GDPR). However, this application doesn't manage any type of user data, so there is no need to take into account this law.

Furthermore, all libraries used use free software licenses. React uses a MIT License and the third part library used for the parsing of GLSL is also MIT License.

7 Project Implementation

7.1 On the choice of frameworks

7.1.1 Decision discussion

The very first important decision is whether to use any type of framework (17). The options studied are the following:

Web frameworks I considered: angular (18), react (19), jQuery (20) and vanilla-js (21).

Angular and react are the two that stand out the most since both are object-oriented (22) frameworks. This is precisely what would work best for this web application, since one of the virtues of object-oriented programming is the high reusability of code, convenient for large applications.

The choice between angular and react is mostly one of preference, since neither of them is better than the other. However, for this project in particular, react may be slightly better since angular is slower. In conclusion, React is the most adequate framework, and the one that will be used for this thesis.

WebGL frameworks: Few frameworks were considered, since for the features that this thesis is meant to implement, we need full control of Webgl, and frameworks usually make things easier but take away low-level control. The only framework that I considered was Three.js (23), which is a framework that gives you a friendlier API than Webgl. This API allows the drawing of simple 3D scenes with few lines of code. But as mentioned earlier, this comes at a cost, and since some of the features that the web application will have require low-level control, the framework was discarded.

7.1.2 React

React is a declarative, component-based javascript framework. As mentioned earlier it is object-oriented, more specifically component-based. This means that the developer builds components, which are encapsulations of UI (user interface) and state. The components should be simple, but they can be composed to create complex UIs.

7.2 Overview

This section will give a general idea of the code design of the application. The implementation decisions will be discussed in section 7.3.

7.2.1 Web routing design

In web applications, the routing are all the pages that the web site contains and how they link each other. This application in particular consists of two different pages. The first one is the examples page (Figure 3. Examples), which is completely parametrizable with the shaders and models that the professors want to display. And when the students want to explore a concrete example, a simple click will send them to the text editor page (Figure 4. Text Editor), which is the page where the student can modify the code and dynamically see the changes being made.

The features and functionality of each page will be explained in the next sections.

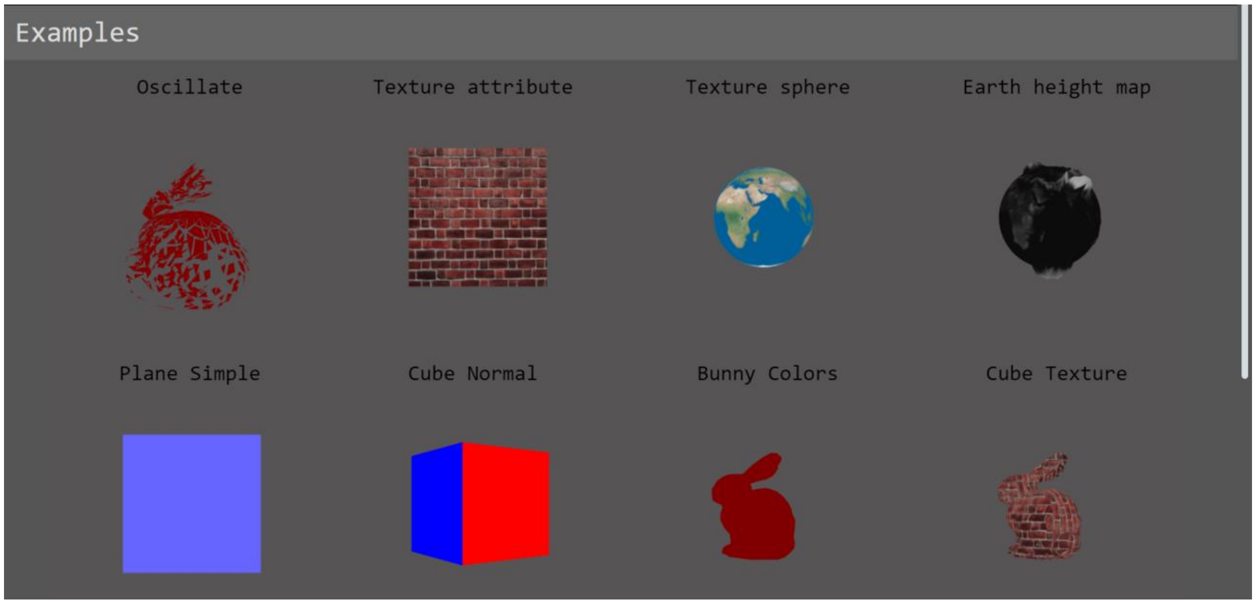
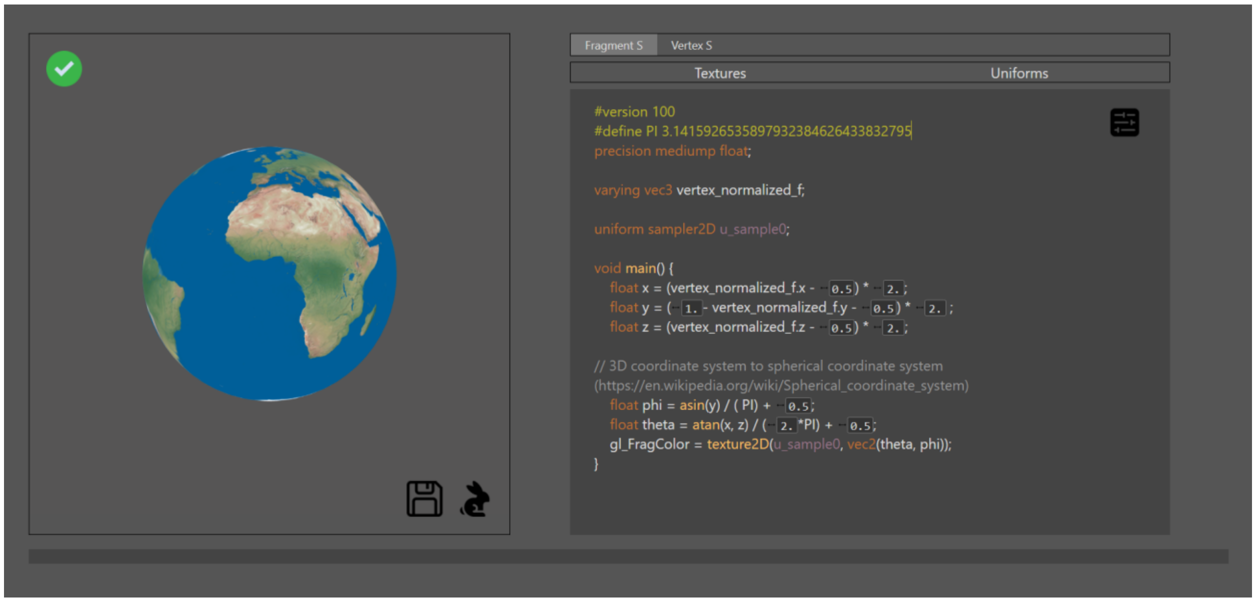


Figure 3. Examples



```
#version 100
#define PI 3.14159265358979323846264338327950
precision mediump float;

varying vec3 vertex_normalized_f;

uniform sampler2D u_sample0;

void main() {
    float x = (vertex_normalized_f.x - 0.5) * 2.;
    float y = (1. - vertex_normalized_f.y - 0.5) * 2. ;
    float z = (vertex_normalized_f.z - 0.5) * 2.;

// 3D coordinate system to spherical coordinate system
(https://en.wikipedia.org/wiki/Spherical_coordinate_system)
    float phi = asin(y) / ( PI) + 0.5;
    float theta = atan(x, z) / (2.*PI) + 0.5;
    gl_FragColor = texture2D(u_sample0, vec2(theta, phi));
}
```

Figure 4. Text Editor

7.2.2 Classes

In this section we will explain the purpose of some classes. Some of the classes have been left out since in one hand, the implementation of those is trivial, and in the other hand, those are utility classes that don't aid in the understanding of the broad application.

7.2.2.1 Material and Mesh

To understand the `Mesh` and `Material` classes, we first have to understand how OpenGL draws 3D models into the screen. To summarize, the information that OpenGL needs is:

Information per vertex: It must have the position of each vertex and some extra information like the normal or color. I have encapsulated this information into the `Mesh` class, which contains a friendly and easy to use API to ease the process of managing vertices information.

Vertex and fragment shader: which are part of the OpenGL pipeline (see the Concepts section). `Material` contains a reference to a `Mesh`, which has the geometric information necessary to draw the scene. The `Material` interface has methods to draw the `Mesh`, compile the fragment or vertex shader, or change the mesh.

7.2.2.2 CanvasVirtual

To understand what is a `CanvasVirtual`, we first have to understand what a canvas is. Canvas (24) is an html element which can be drawn to. For each canvas element we have to initialize an OpenGL context, which offers the methods that we will use to draw our scenes (methods like DrawElements). The problem with that, is that if we want to draw in the same page various scenes (like the examples page), we will have to create multiple canvases, and consequently, initialize multiples contexts. Also, since every canvas element has different contexts, sharing textures or 3Dmodels between them is an added difficulty.

To work around this issue, I have opted to create this class, which represents a portion of one unique canvas element. This means that with one canvas element, we can instantiate multiple `CanvasVirtual`, and each one will be responsible for drawing their portion of the canvas element (see section 7.3.1).

This allows for multiple scenes to be drawn in the same canvas. As seen in the previous figure (Figure 3. Examples) we can see multiple scenes drawn and there is actually only one canvas element, so only one WebGL context.

7.2.2.3 Examples

Now that we have the more low-level classes out of the way, we start with React components (25). Those are encapsulations of UI and state. This component in particular, is where all the examples are displayed (Figure 3. Examples).

In this page, there are some important UI design decisions taken. First of all, each shader is drawn into the screen. The alternative would be to have a list with the name of the shaders linking to the text editor page, but it wouldn't be as interactive. It's also worth noting that the 3D scenes are all spinning, to make the experience more dynamic. Another decision is the use of a grid to use as much space of the screen as possible. Having only one column would make the user scroll more than necessary to browser over the shaders. Also, to make the experience more interactive, when hovering over a scene, it becomes larger, so the user understands he is selecting it.

7.2.2.4 Text Editor page

The Text editor page (Figure 5. Text Editor page) is composed by two main components. The `TextEditor` (number 4), which is where the user would type the code, and the `InteractiveCanvas` (number 1), which is the canvas where the scene is displayed. We can also toggle the text editor between the fragment shader and the vertex shader (number 2). The Textures and Uniforms tabs (number 3) let you toggle between seeing the uniforms and textures that this concrete example has access to (Figure 7. Uniforms tab and Figure 6. Texture tab). Lastly, when there are compilation errors, those are displayed on the bottom of the page.

On the text editor page, there have been UI design decisions taken:

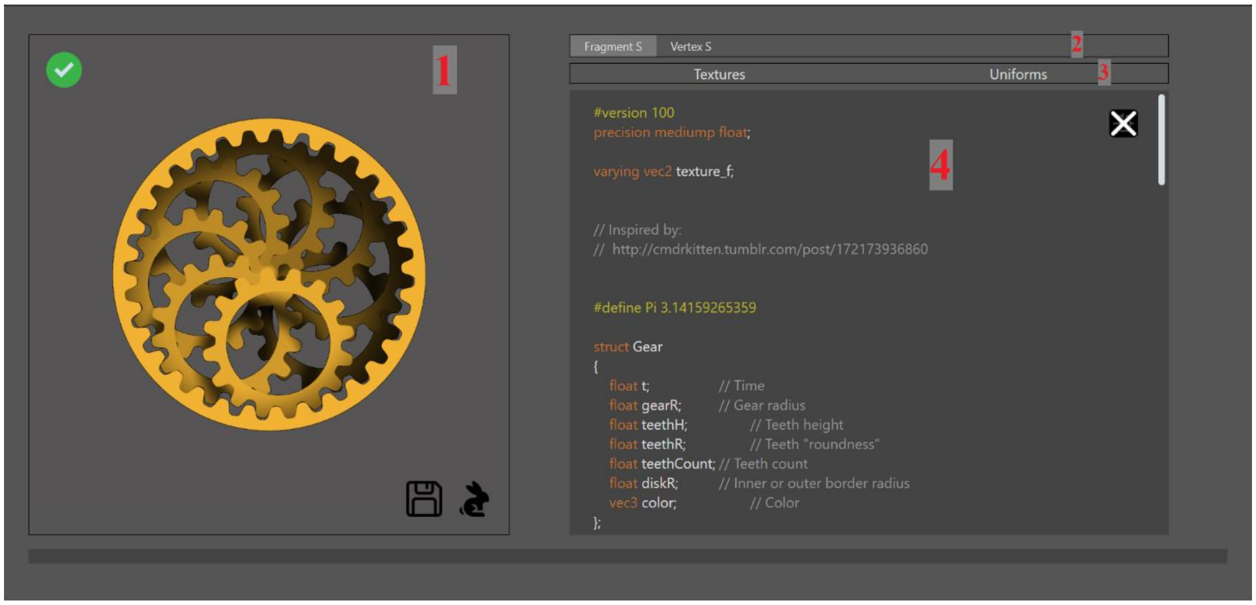
First of all, the screen is divided almost evenely between the `TextEditor` and the `InteractiveCanvas.` That's because having a large display is important, but as long as the user can type comfortably on the text editor.

Another decision is the use of tabs for the different shader types (vertex and fragment). The alternative would be to have both shaders on the screen, but this would half the space of each one. When writing shaders, there is no need to see both at the same time, since each one has different utilities. That's the reason tabs where implemented.

The textures and uniforms tabs work differently from the fragment and vertex ones. The idea is that the user can see what textures and uniforms are available, at a cost of text editor space. But, once the user has the information that needs, he can retract the tabs to have full space for the text editor.

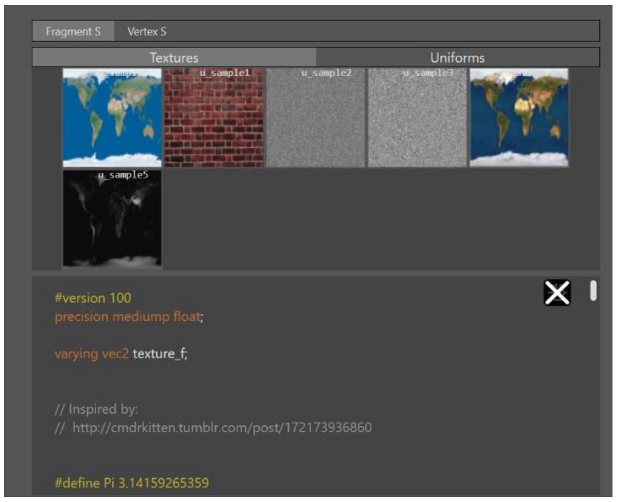
Lastly, the text errors appear on the bottom of the page, since this way, when there are no errors there is no space wasted.

Everything the user wants to do is always one click away, which makes the user experience fast and dynamic.

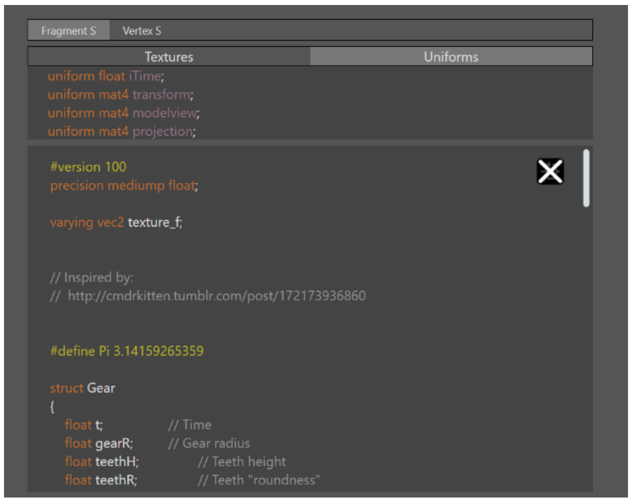


```
#version 100
precision mediump float;

varying vec2 texture_f;


// Inspired by:
//  http://cmdrkitten.tumblr.com/post/172173936860


#define Pi 3.14159265359

struct Gear
{
    float t;            // Time
    float gearR;        // Gear radius
    float teethH;           // Teeth height
    float teethR;           // Teeth "roundness"
    float teethCount; // Teeth count
    float diskR;        // Inner or outer border radius
    vec3 color;             // Color
};
```

Figure 5. Text Editor page. 1) Interactive Canvas. 2) Shaders tab. 3) Textures and Uniforms Tabs. 4) Text Editor.



```
#version 100
precision mediump float;

varying vec2 texture_f;


// Inspired by:
//  http://cmdrkitten.tumblr.com/post/172173936860


#define Pi 3.14159265359
```



```
uniform float iTime;
uniform mat4 transform;
uniform mat4 modelview;
uniform mat4 projection;
```

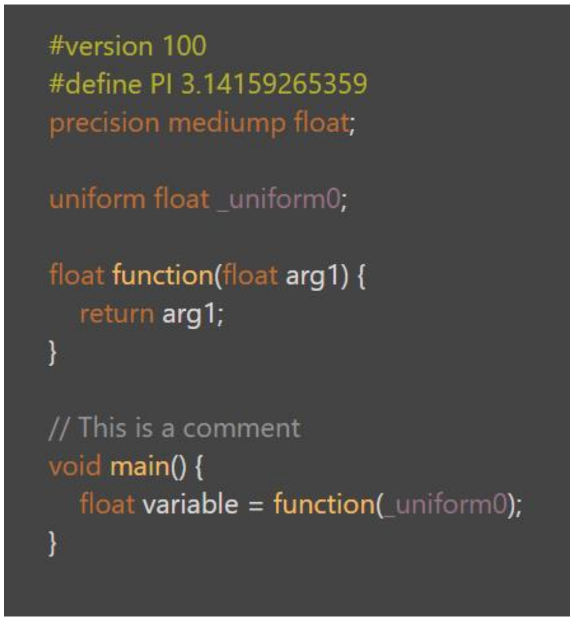
```
#version 100
precision mediump float;

varying vec2 texture_f;


// Inspired by:
//  http://cmdrkitten.tumblr.com/post/172173936860


#define Pi 3.14159265359

struct Gear
{
    float t;            // Time
    float gearR;        // Gear radius
    float teethH;           // Teeth height
    float teethR;           // Teeth "roundness"
```

Figure 6. Texture tab

Figure 7. Uniforms tab

7.2.2.5 Text Editor

The `TextEditor` component is where the user types the code. It’s slightly more complicated than this since there are some added features on this text editor a part from displaying the written code. The first one is the **syntax highlighting** (26). As we can see the code is colored to improve the readability (Figure 8. Code Highlighting).

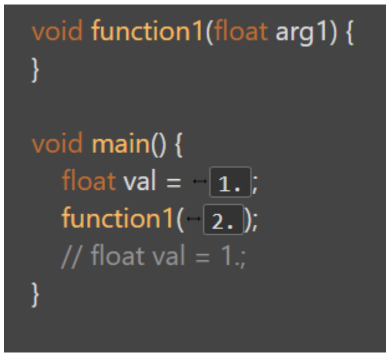


```
#version 100
#define PI 3.14159265359
precision mediump float;

uniform float _uniform0;

float function(float arg1) {
    return arg1;
}

// This is a comment
void main() {
    float variable = function(_uniform0);
}
```

Figure 8. Code Highlighting

The text is displayed in different colors according to the category of terms. More information about the implementation of the syntax highlighter section 7.3.4.

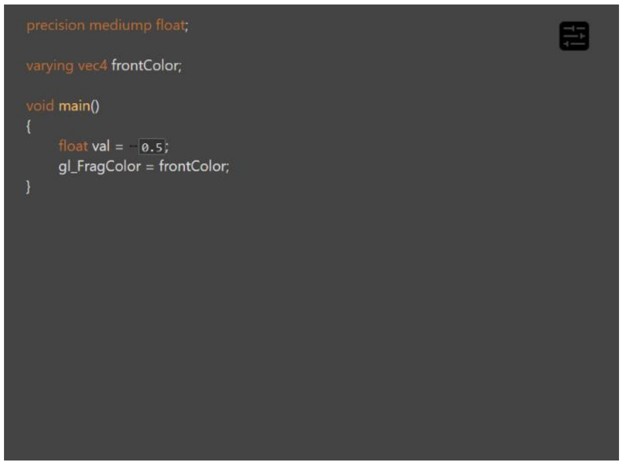
Another important feature are the sliders. Which are widgets that get inserted automatically when the user is writing a number literal. It’s important to understand that not all numbers written in the code will be replaced by this widget, it wouldn’t make any sense to write a variable as *_uniform0* and get the 0 replaced by the slider. It’s only going to appear when the user writes a number literal**.** To better understand where the sliders should be inserted, see the following figure (Figure 9. Example sliders).



```
void function1(float arg1) {
}

void main() {
    float val = 1.;
    function1(2.);
    // float val = 1.;
}
```

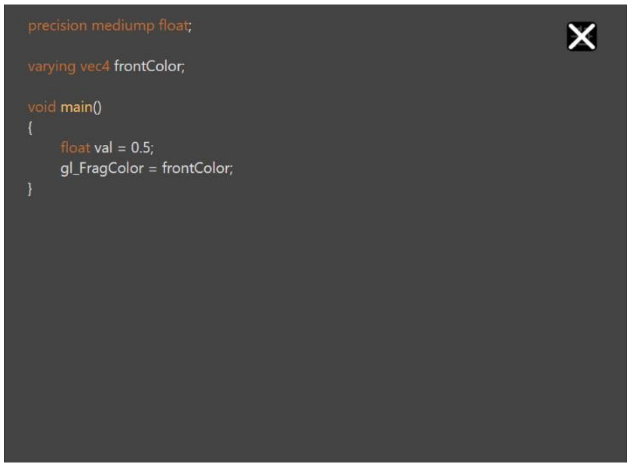
Figure 9. Example sliders

For performance reasons (discussed in section 8.3.3), in the top right corner there is a button that disables the sliders (Figure 11. Sliders enabled and Figure 10. Sliders Disabled).



```
precision mediump float;

varying vec4 frontColor;

void main()
{
	float val = 0.5;
	gl_FragColor = frontColor;
}
```

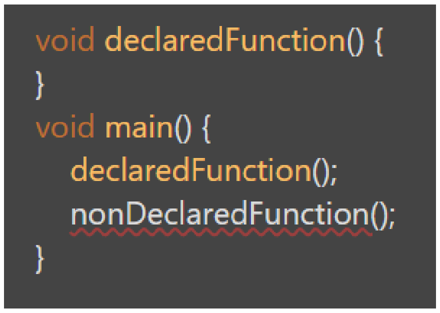
Figure 11. Sliders enabled



```
precision mediump float;

varying vec4 frontColor;

void main()
{
	float val = 0.5;
	gl_FragColor = frontColor;
}
```

Figure 10. Sliders Disabled

Another relevant feature is the **compilation syntax error highlighting**. When the code written is compiled, if there are any errors, the text editor highlights it (Figure 12 Syntax error highlighting).



```
void declaredFunction() {
}
void main() {
    declaredFunction();
    nonDeclaredFunction();
}
```

Figure 12 Syntax error highlighting

7.2.2.6 InteractiveCanvas

As seen earlier, `InteractiveCanvas` is the component that displays the scene. It uses the fragment shader and vertex shader from the text editor. Thus, every change in the text editor, gets directly reflected in the canvas in real time. Apart from the drawing, it also has some extra features that enhance the user experience.

The first feature is the rotation of the camera with the mouse. Dragging on the canvas rotates the camera if the model is 3D. If the model is 2D (like a plane), it wouldn't make much sense to allow the rotation.

Then we have the different UI elements in the canvas itself (Figure 13. InteractiveCanvas).

On the top left corner (number 1), we have a green tick if the shaders are correctly compiled, and a cross if there are errors. On the right bottom corner, we have the save button, which downloads the shaders written on the text editor (number 2), and the rabbit button, which opens a tab where the mesh of the current example can be changed (Figure 14. Mesh Selector). This tab allows the user to browse over the models that are available and allows the selection by clicking on them.

The mesh selector uses the same UI design ideas than the examples page. The difference is that it doesn't occupy as much space. This way the user can still, type new code, or rotate the scene while changing the mesh.

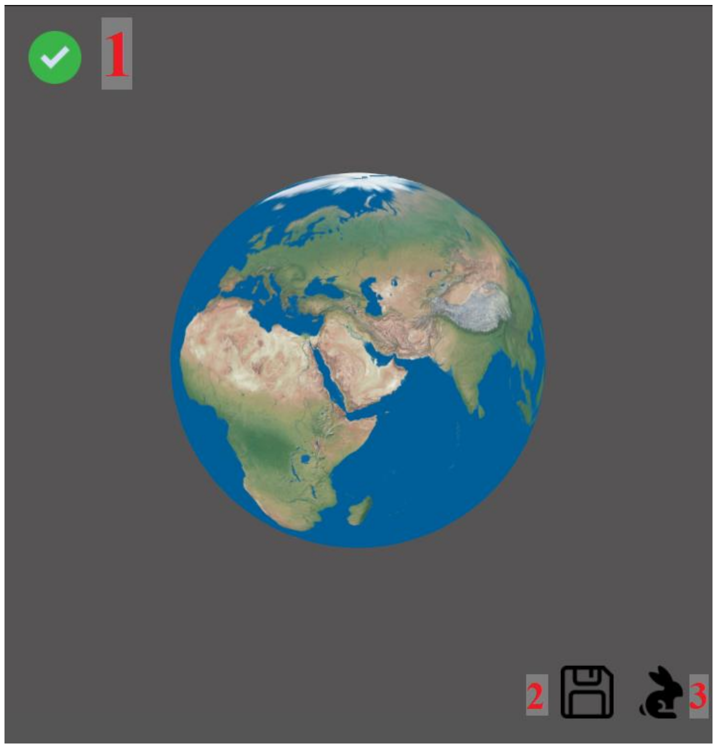


Figure 13. InteractiveCanvas

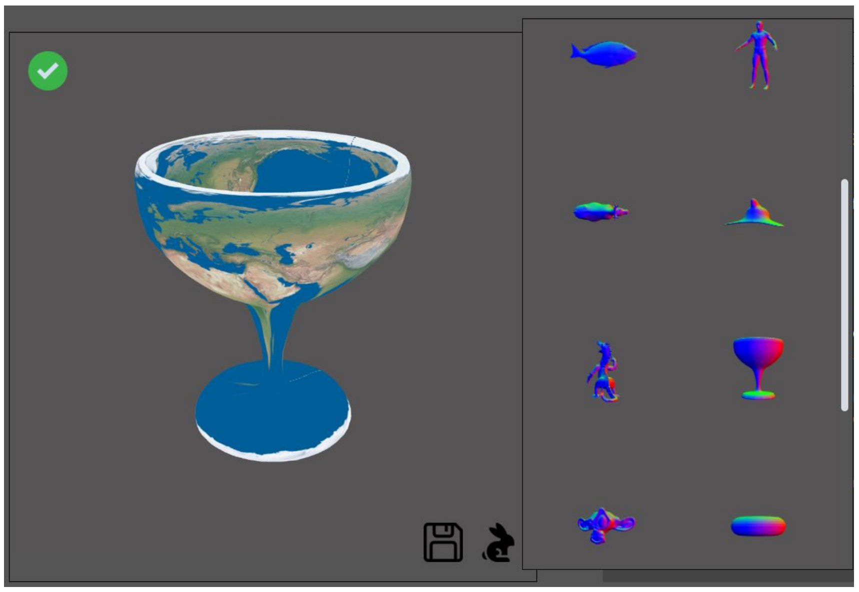


Figure 14. Mesh Selector

7.3 Tasks

In this section we will go through some of the tasks planned, explaining the problems and solutions that arose in them, and the justification of important implementation decisions.

7.3.1 Implementation of Examples Page

The examples page are multiple independent scenes drawn into the same canvas using a grid pattern. The way this is implemented is using the utility viewport (27) and scissor (28) that webgl offers. Viewport is used to tell WebGL where in the canvas the scene will be drawn. Thus, we can easily instantiate multiple `VirtualCanvas`, and before drawing each one, set the viewport to the portion that each one is responsible for. An issue still remains. Since most of the scenes are animated, before each scene is drawn the canvas is cleared using the command glClear (29). If the canvas were not to be cleared, then the drawings of the same scene would stack one above the other. To exemplify it, on Figure 15., we can see this effect after the sphere spins. The problem is that glClear clears all the canvas, not only the part delimited by the viewport. Thus, each VirtualCanvas would clear all the previous ones, which is not what we intend. That's where the scissors test (30) comes in. After setting the scissors box, and enabling the scissors test, we ensure that the next drawing commands (including glClear) will only modify the pixels inside the scissors box.

To sum up, first we define a viewport, then we enable the scissors test, setting the scissors box to be the size of the viewport, so all drawing commands only modify pixels in the current viewport, and finally we draw the scene.

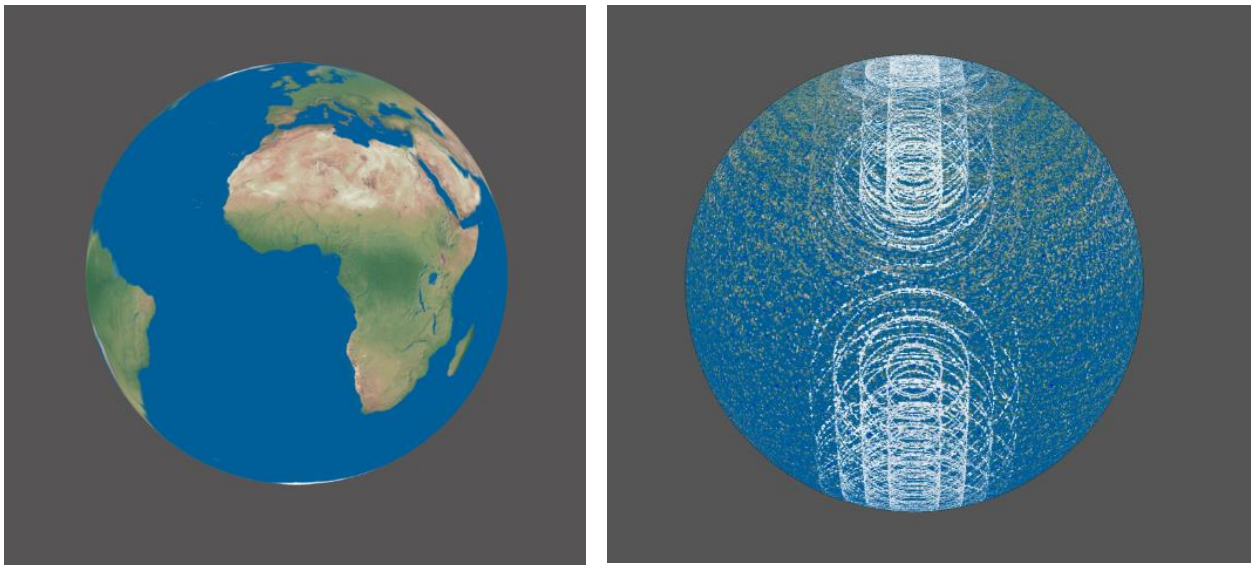


Figure 15. Earth without clearing the scene.

7.3.2 Implementation of the Text Editor

There are a lot of implementation decisions that went into the design of the text editor. It's important to note that HTML has elements that natively allow for the user to type (like textarea (31)), and it offers features like undo, copy, paste… Unfortunately, the default implementation couldn't be used since there are some actions that in our particular project wouldn't work as intended. For example, undoing when there are actions like moving the slider that the default implementation does not account for. Copying when there are custom widgets in-between text does not work as expected either. For this reason, an implementation from scratch was more convenient, providing total control over the editor.

So, the first implementation decision is how to make a basic text editor which lets the user type, and perform basic operations like undo, copy, paste... We won't go into detail of the JavaScript implementation but we will discuss the programming design that was chosen. In this particular case, there was a design pattern[14] perfectly suited for this problem: the command pattern[15]. The idea of this pattern, is to encapsulate an action in an object. It's useful in this case, because every action we want to make (type a character, remove a character…), gets encapsulated in an object, and pushed onto a stack. Each object implements an execute function (which is the code that executes the action), and unexecute (which is the code that executes the inverse of the action). This makes the implementation of undo trivial, since we only need to call unexecute of the last element of the stack and remove it from the stack. For example, if

[14] Design pattern is a general repeatable solution to a commonly occurring problem in software design
[15] All design patterns mentioned are from the following book (16)

we write a character, we would instantiate the object, we would call execute and we would push it to the stack. If we wanted to undo it, we would call unexecute and then remove the object of the stack.

Keeping track of the caret adds one extra layer of difficulty to the implementation of all the actions.

7.3.3 Implementation of widget insertion

The next text editor feature is the widget insertion. As explained in previous sections (see section 7.2.2.5), what we want to accomplish is the insertion of sliders where there are number literals in the code. Here we will comment on the implementation decisions. First of all, we need to understand what are the difficulties. The first one is how do we find where are the number literals. The first idea could be using regular expressions to find numbers on the text. This maybe[16] could be done, since GLSL (which is the language that we are trying to find literals in), is a simple language, but it would be really hard to program. There are a lot of cases to consider. Sometimes the number is part of a variable name, sometimes it is commented…

The other option is parsing (32) the input text with a GLSL parser, which will transform the input text into an AST (33). This AST has all the information we need: for each word[17] in the text, we have its position on the input text, and information in relation of its type (if it's a functions name, a keyword, a number literal…).

One of the problem is that making a parser is really difficult. That's why I decided to use a 3rd party parser (34).

Now we have all the necessary pieces to insert sliders. The way it would work all together would be the following: First the user changes the text (with a key-stroke in the text editor for example). Then the modified text gets parsed and we extract from the resulting AST the position of the number literals. Finally, we render the modified text with the sliders inserted in the position of the number literals.

The sliders are actually just an image with two arrows and a box displaying the current value of the number (Figure 16. Slider). You can click the arrows and drag to modify the value in the box.

[16] Regular expressions are not expressive (50) enough to analyze a complex programming language correctly.
[17] The correct terminology in lexical analysis (49) would be token instead of word.

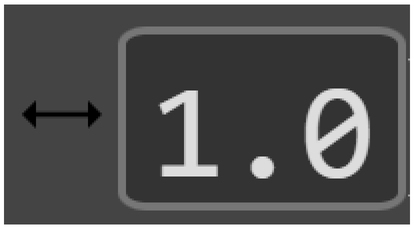


Figure 16. Slider

7.3.4 Implementation of syntax highlighting

The next feature is the syntax highlighting. We want to display with different colors the preprocessor lines (start with *'#'*), keywords (like void, float…), comments (start with *'//'*), function names and uniform names (special type of variable). Knowing that, we have the same problem that appeared in the implementation of widget insertion: we need to analyze the text to find the position of comments, keywords… So, we have to decide again if we should use the AST (result of parsing the text) or regular expressions.

Even though it looks counter intuitive, here the better option is to use regular expressions. That is because the AST is not always available, since it is only constructed when the input of the parser is valid GLSL code. This means that while the user is writing, there will be moments where the AST will not be available. However, regular expressions always work since ultimately are searching into a string, and do not depend on the syntaxis being correct.

This has some drawbacks, since the AST has information really useful, like the name of the functions or variables declared in the code. The best solution seems to be an hybrid of using the AST and regular expressions. Each time we have available the AST, we save the information that we need, like the names of functions or variables. And we use this information to find with regular expressions these names into the text. We also use regular expressions to find the comments and preprocessors lines. To find all the keywords and built-in functions (35) we search in the text for each keyword and built-in function available in GLSL.

After all that, we have all the positions and we can color every word.

7.3.5 Implementation of error highlighting

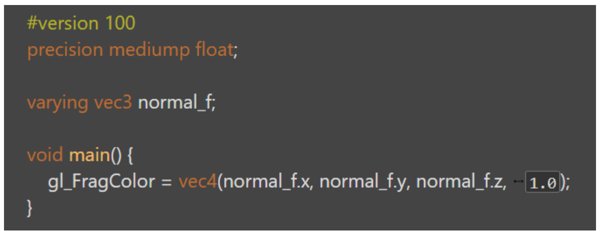
Next step is the highlighting of errors. The basic idea is to compile the shader each time the user modifies it, and, if any, parse the compilation errors to know which lines of code have errors.

It’s important to note that the text that will be compiled is exactly what the user has typed. So, there will be no uniforms inserted in the text. If we compiled the version of the shader with uniforms inserted (see 5.4.5 Interactive canvas), the error messages would be hard to interpret by the user.

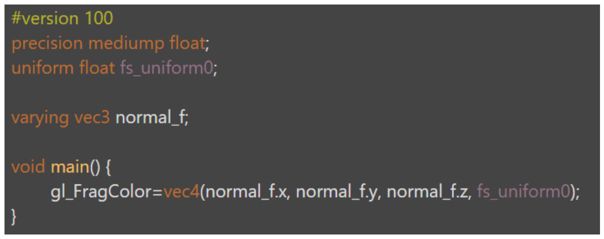
7.3.6 Implementation of interactive canvas

Now, we have a text editor which has, apart from the basic features, code highlighting, widget insertion and error highlighting. Now it’s time to use the code that the user is writing to draw the canvas. The more direct approach would be to obtain the text from the text editor, compile it and draw on the canvas the scene. But this approach is slow. Since this would mean that every time the user changes a number with the slider, the code has to be recompiled, which would result in the canvas jittering, and the experience would be ruined. This is why we have to come up with a way to not recompile the code even though the numbers change. This is where uniform variables come in.

Uniforms are variables that are passed to the fragment or vertex shader. The thing that characterizes them is that their values are constant among all vertices and fragments. So, where we would insert sliders in the text editor, we insert uniform variables on the shader that we want to compile. To better understand this concept, see next figure (Figure 17. Text editor text versus Compiled shader text). The left one is what is displayed in the text editor. The right one is what we sent to WebGL to compile it and draw it to the canvas. Note that the slider is replaced by a string “fs_uniform0”, and this uniform is declared on top of the file. The names of the uniform values may be anything, but they have to be unique.



```
#version 100
precision mediump float;

varying vec3 normal_f;

void main() {
    gl_FragColor = vec4(normal_f.x, normal_f.y, normal_f.z, 1.0);
}
```



```
#version 100
precision mediump float;
uniform float fs_uniform0;

varying vec3 normal_f;

void main() {
        gl_FragColor=vec4(normal_f.x, normal_f.y, normal_f.z, fs_uniform0);
}
```

Figure 17. Text editor text versus Compiled shader text

So, we insert sliders on the text editor, and we insert strings (fs_uniform0) on the code we compile. The last step is to store the value of each slider, and sent it to the shaders through its corresponding uniform variable. If we don’t do this last step, WebGL doesn’t know which value holds the variable fs_uniform0.

Now we can modify the values of the uniforms with the slider, send them to the shaders through the uniform variables, and see the changes in the canvas without having to recompile every time.

Other features that are implemented in the InteractiveCanvas like rotating or changing the mesh in the scene are explained in section 7.2.2.6 and haven’t posed any implementation complications.

7.3.7 Implementation of Mesh and Texture manager

Something very instructive is to see the same shader applied in various meshes (Figure 18. Same shader different meshes). Also, having access to textures in the shaders is a very important feature that allows for very impressive scenes (Figure 19. Using textures).

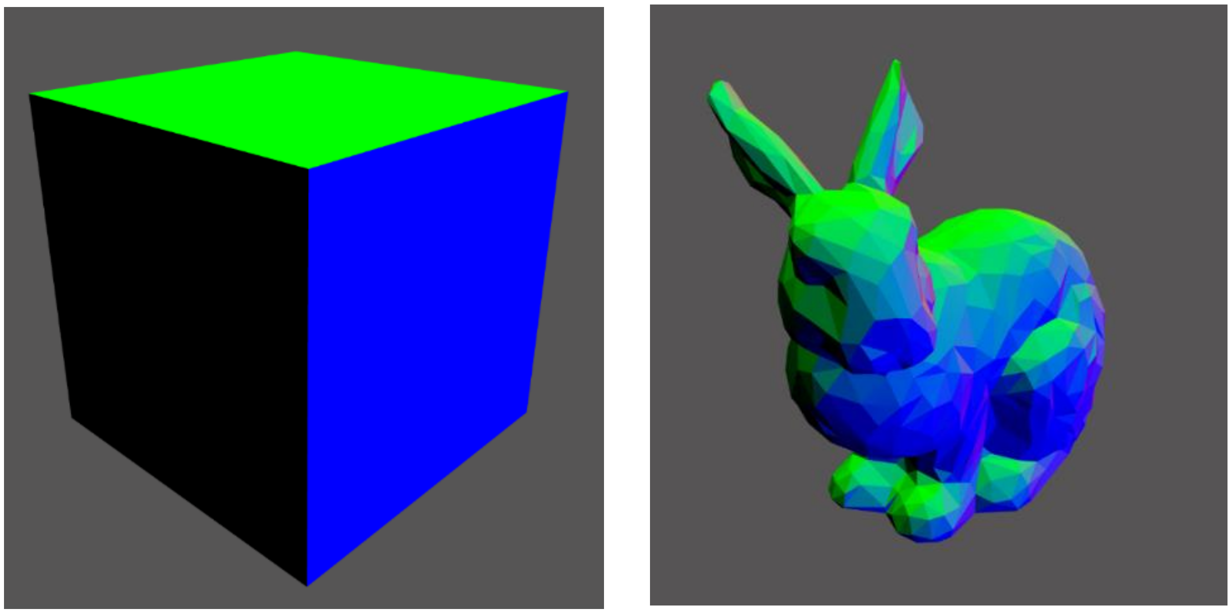


Figure 18. Same shader different meshes

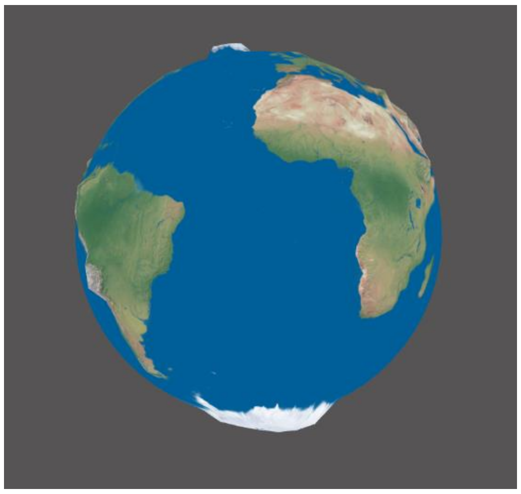


Figure 19. Using textures

To create the scene in Figure 19. Using textures, we use the following two textures (Figure 20. Textures Earth). The one on the right we use it to color the sphere, and the one in the left is a height map. It’s used to give relief to the sphere. The whiter the color is, the more accentuated the relief will be. This effect is created on the vertex shader, where we displace the vertices according to the height map.

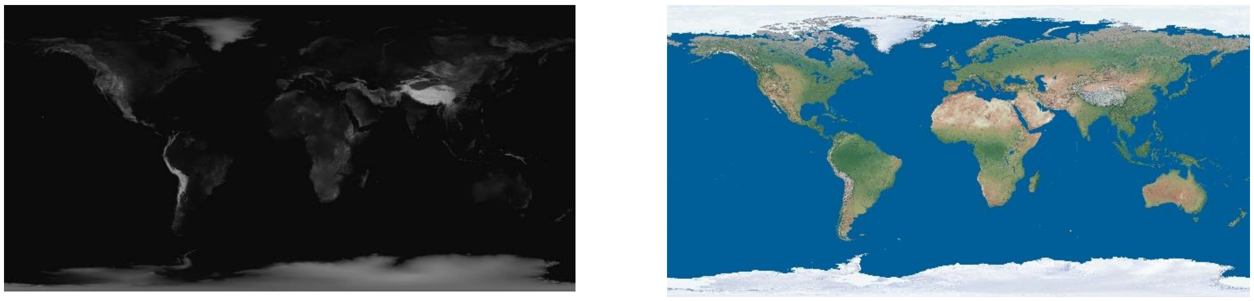


Figure 20. Textures Earth

Understanding the utility of the textures and meshes, we create mesh managers and texture managers. Those are classes that store the meshes and textures, so when any scene (VirtualCanvas) wants to use them, they don't have to load duplicated information. For example, if there are two scenes with the same texture, the first scene will load the texture, but the second one will only access it (because it's already loaded).

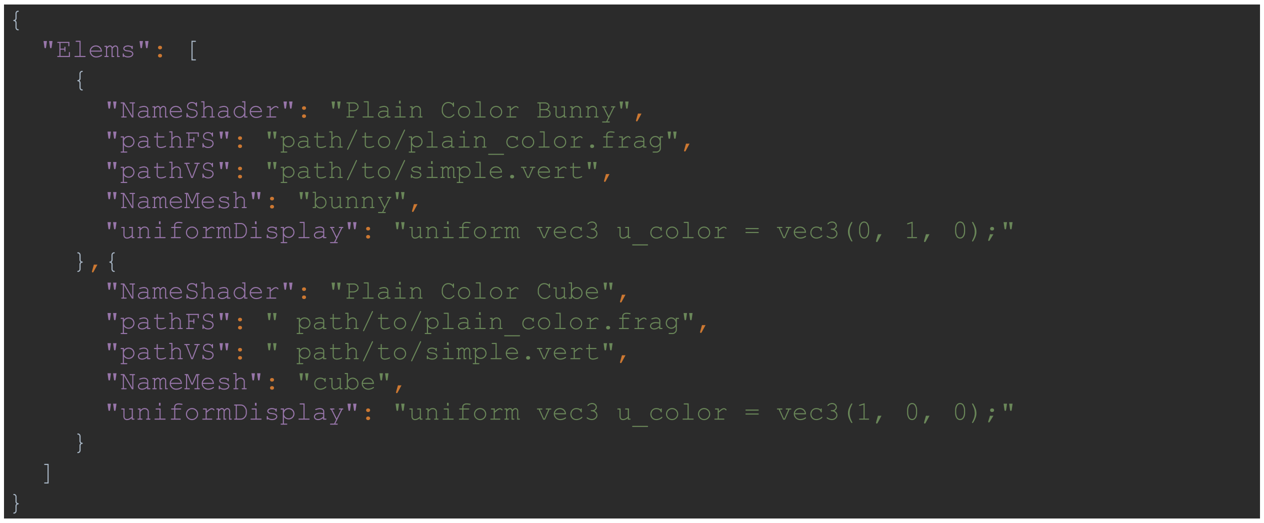
7.3.8 Parametrization

It's important to note that this project is not aiming to develop a learning course for students. The project is about the development of a tool that can be used to teach students. This means that the examples that are displayed on the examples page can't be hard coded. The admin of the tool, should be able to parametrize the examples so they can prepare convenient examples for what they are trying to teach.

The way this project accomplishes that is with the use of json (36) files. JSON is a format used to store information. It's easy to read / write for humans and it's easy to parse for machines. The way this project uses it, is with the following three files that are on the web project source files:

7.3.8.1 examples.json

This file will contain a list of parametrized scenes.



```
{
  "Elems": [
    {
      "NameShader": "Plain Color Bunny",
      "pathFS": "path/to/plain_color.frag",
      "pathVS": "path/to/simple.vert",
      "NameMesh": "bunny",
      "uniformDisplay": "uniform vec3 u_color = vec3(0, 1, 0);"
    },{
      "NameShader": "Plain Color Cube",
      "pathFS": " path/to/plain_color.frag",
      "pathVS": " path/to/simple.vert",
      "NameMesh": "cube",
      "uniformDisplay": "uniform vec3 u_color = vec3(1, 0, 0);"
    }
  ]
}
```

Figure 21. examples.json

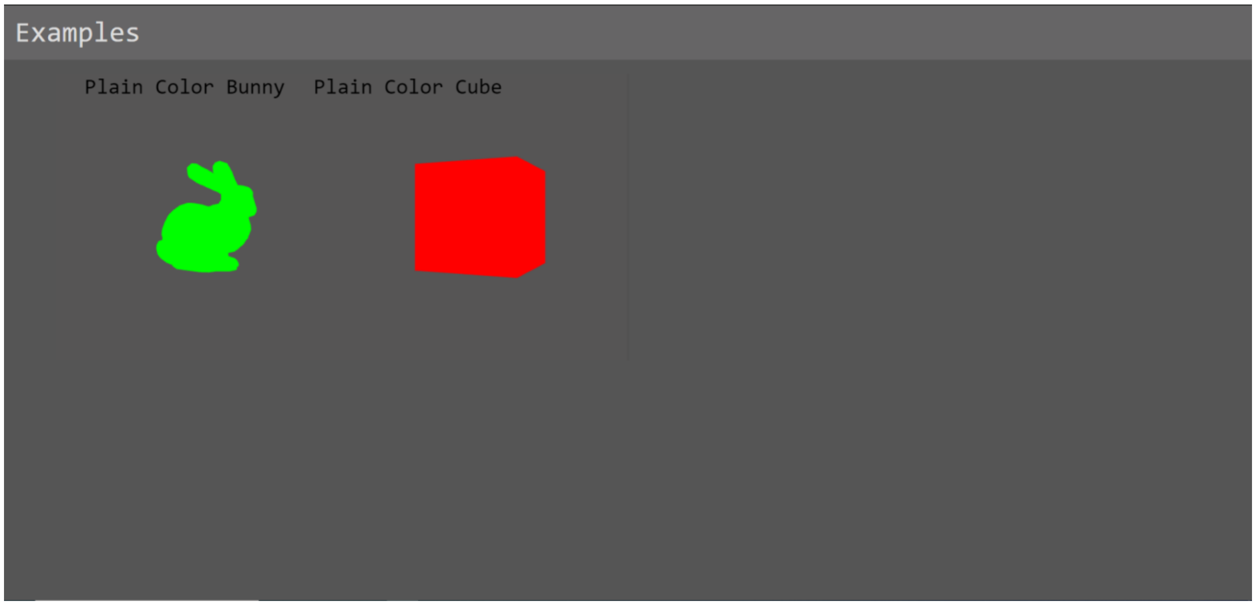
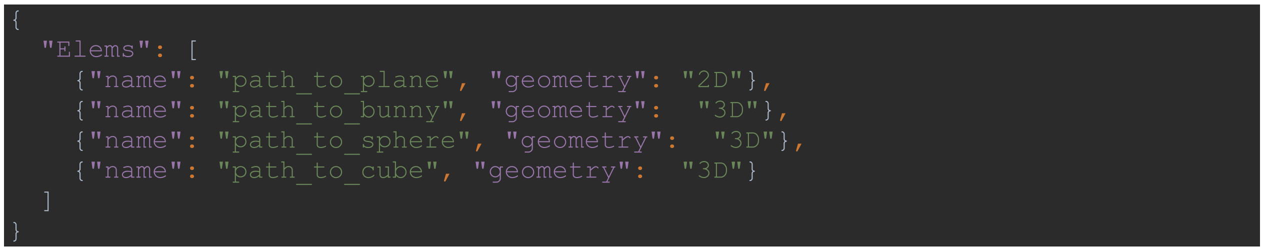


Figure 22. Examples page examples.json

As we can see, the examples.json file, lets us use different meshes, different shaders and it also lets us define constant uniform variables. In this case I created a simple u_color variable which the fragment shader uses to color the scene.

7.3.8.2 mesh_metadata.json

This file allows for the parametrization of available meshes in the program (see Figure 23. mesh_metadata.json). For each mesh we have the name, which is actually the path where the .obj file is. And the geometry, which is used to know if the mesh can be rotated or not (if the mesh is 2D, the user won't be allowed to rotate it).

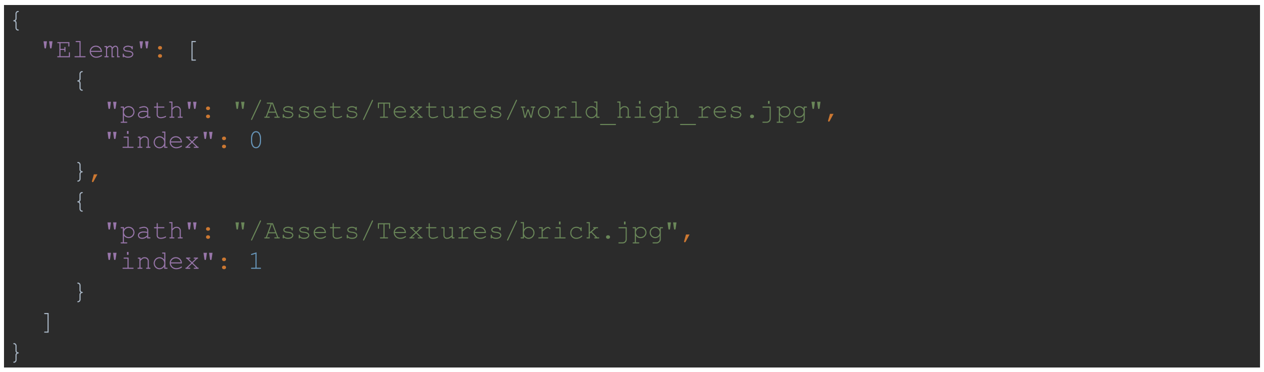


```json
{
  "Elems": [
    {"name": "path_to_plane", "geometry": "2D"},
    {"name": "path_to_bunny", "geometry":  "3D"},
    {"name": "path_to_sphere", "geometry":  "3D"},
    {"name": "path_to_cube", "geometry":  "3D"}
  ]
}
```

Figure 23. mesh_metadata.json

7.3.8.3 textures_metadata.json

This file allows for the parametrization of available textures in the shaders (see Figure 24. textures_metadata.json). For each texture we have the path and the index. The index is used to name the texture. (if index = 0, then the name of the texture is u_sample0).



```json
{
  "Elems": [
    {
      "path": "/Assets/Textures/world_high_res.jpg",
      "index": 0
    },
    {
      "path": "/Assets/Textures/brick.jpg",
      "index": 1
    }
  ]
}
```

Figure 24. textures_metadata.json

8 Conclusion

8.1 Achievement of objectives

At the planning of the thesis, we proposed some objectives:

8.1.1 Designing the graphical interface

As seen in section 7.2, the graphical interface is usable and easy to navigate, which are the objectives that we pursued. The user also receives feedback when performing every action, which improves the user experience.

8.1.2 Programming the text editor

The text editor was probably the hardest part of the project, since it is where most of the difficulties appeared. However, we implemented all the features that we planned and more. The user can modify the code of the shader and see the changes in real time. Also, he can change the 3D object where the shaders are applied to in real time. Furthermore, he may also manipulate the 3D scene with the mouse. The text editor also has other features like syntax highlighting, error highlighting and widget insertion which also improve the experience. In conclusion, the objectives where satisfactorily met.

8.2 Achievement of competencies

CCO1.2: Demonstrate knowledge of the theoretical foundations of programming languages and the associated lexical, syntactic and semantic processing techniques, and know how to apply them to the creation, design and processing of languages.

To implement part of the syntax highlighting and widget insertion, the code written on the text had to be parsed. Also, to extract relevant information from the resulting AST, visitors were.

CCO1.3: Define, evaluate and select hardware and software development and production platforms for the development of computer applications and services of varying complexity.

A decision had to be made about which frameworks and libraries to use in the implementation of the web application.

CCO2.3: Develop and evaluate interactive and complex information presentation systems, and their application to problem solving person-computer interaction design.

The design of the user interface aims at being intuitive, to be used as comfortable as possible.

CCO2.6: Design and implement graphical, virtual reality, augmented reality and video game applications.

The application is a text editor for the implementation of shaders, and the scenes are being rendered through WebGL.

8.3 Next steps

As mentioned in section 8.4, the project is considered a success. However, there are some features that are not in the scope of this thesis (for deadline reasons) but would improve the final tool if implemented correctly.

8.3.1 Allow other versions of GLSL

In this thesis we talked about WebGL as the OpenGL API of the browser. And that's correct, but there are some things to keep in mind. Firstly, WebGL only allows till GLSL 2.0. (which is the shader programming language). And the latest version of GLSL is the 4.60. This project for simplicity reasons, lets the user program the shaders with GLSL 2.0. But, since this thesis is about the development of a didactic tool, allowing for newer versions would make the tool more useful.

8.3.2 Implementing Backend

As mentioned across the thesis, the objective of the developed tool is to let the students experience with shaders. That is the reason making a backend didn't look necessary. Because to make a web application that lets the user temporarily modify things doesn't require backend. But now that the project is done, the idea of being able to persist in a comfortable manner all the changes that are made in the shader, or even create shaders from scratch to test new ideas looks appealing. Implementing a backend would bring the project persistence in a user-based authorization model. Logging into the platform would allow the user to access all the shaders that he has modified or made from scratch.

8.3.3 Optimizing the parser

It's important to mention that parsers are really robust but have a high asymptotic computational complexity (37). This means that when the shaders have a lot of lines of code, it's slow. That is why the Interactive Canvas has a button to disable the widget insertion, to improve performance on large shader files. But that's not an ideal solution, there are ways to optimize the parser so every time a change is made in the text editor, it only parses a small part of the shader, reusing the rest of the previous parsing results.

This would make the widget insertion usable for all shader sizes.

8.3.4 Extra widgets

In the same way that the sliders are useful for the students to have a more dynamic learning experience, there are also other possible widgets that would help ease the process. For example, giving the option to use a color picker to set variables of type vec3 (vectors of three values).

8.4 Final conclusions

On a personal level, working in a large project alone, finding ways to implement my own ideas, has made me a better developer. I learned new web frameworks, I also learned WebGL and I improved my web developer skills (html, js, css).

It's important to mention that while working on the application, there have been some changes in the planning. For instance, there are some features like changing the mesh in real time that weren't contemplated in the planning but ended up being on the final product. Equally important, there has been some features that were on the original planning and ended up being modified, like importing 3D object models. At the time, the idea was to be able to load a .obj file from the web application. We decided against it to only give the privileges of choosing which specific meshes are available to the admins, through the parametrization with the json files. Thus, the web application does what was meant to, which is to let the students experiment with shaders. Even though there are some optimizations that couldn't be implemented (see section 8.3), there are many that are, and over all, the performance of the tool is adequate.

We would also like to mention that a part from the design decisions and problems that are contemplated in section 7.3, there have been problems and complications language related that are not discussed in the thesis, but have taken time to fix. For example, understanding good practices with React, or using WebGL efficiently. Most of these problems are due to inexperience in these frameworks, which was predicted in the planning.

Also, there have been some early design decisions that haven't made it to the final application in favor of more adequate ones. For instance, the web UI design have been changing as more features were added. But the figures in this thesis are all from the final version.

In conclusion, the thesis has met all the requirements and objectives planned, and even more. The administrators can parametrize the application so it adjusts to the contents they are teaching. And the students have an easy-to-use text editor with multiple features that aid the understanding of shaders. Thus, we could say that this thesis has been a success.

9 Bibliography

Bibliography

1. **WebGL.** ***Khronos.*** **[Online] September 20, 2021. https://www.khronos.org/webgl/wiki/Getting_Started.**

2. API. *Wikipedia.* [Online] September 22, 2021. https://en.wikipedia.org/wiki/API.

3. OpenGL ES 2.0 Rendering Pipeline. *WikiBooks.* [Online] 12 16, 2021. https://en.wikibooks.org/wiki/GLSL_Programming/OpenGL_ES_2.0_Pipeline.

4. Node. *Unity3D.* [Online] September 27, 2021. https://docs.unity3d.com/Packages/com.unity.shadergraph@6.9/manual/Node.html.

5. The Book of Shaders. *The Book of Shaders.* [Online] 12 13, 2021. https://thebookofshaders.com/edit.php.

6. GLSL language integration. *MarketPlace Visual Studio.* [Online] September 25, 2021. https://marketplace.visualstudio.com/items?itemName=DanielScherzer.GLSL.

7. Sprints. *Atlassian.* [Online] September 28, 2021. https://www.atlassian.com/agile/scrum/sprints.

8. Github. *Github.* [Online] October 5, 2021. https://github.com/.

9. Git. *Git.* [Online] October 3, 2021. https://git-scm.com/.

10. Webstorm. *Jetbrains.* [Online] September 29, 2021. https://www.jetbrains.com/webstorm/.

11. IES. *FIB.* [Online] 12 11, 2021. https://www.fib.upc.edu/ca/estudis/graus/grau-en-enginyeria-informatica/pla-destudis/assignatures/IES.

12. Compiladors. *FIB.* [Online] 12 11, 2021. https://www.fib.upc.edu/ca/estudis/graus/grau-en-enginyeria-informatica/pla-destudis/assignatures/CL.

13. Introducció a disseny d'interfícies. *IDI.* [Online] 12 11, 2021. https://www.fib.upc.edu/ca/estudis/graus/grau-en-enginyeria-informatica/pla-destudis/assignatures/IDI.

14. Projecte Aplicat d'Enginyeria. *FIB.* [Online] 12 11, 2021. https://www.fib.upc.edu/ca/estudis/graus/grau-en-enginyeria-informatica/pla-destudis/assignatures/PAE.

15. Teoria de la Computació . *FIB.* [Online] 12 11, 2021. https://www.fib.upc.edu/ca/estudis/graus/grau-en-enginyeria-informatica/pla-destudis/assignatures/TC.

16. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Massachusetts : Addison-Wesley, 1995.

17. Software framework. *Wikipedia.* [Online] 11 25, 2021. https://en.wikipedia.org/wiki/Software_framework.

18. Angular. *Angular.* [Online] 11 25, 2021. https://angular.io/.

19. React. *React.* [Online] 11 25, 2021. https://reactjs.org/.

20. Jquery. *Jquery.* [Online] 25 11, 2021. https://jquery.com/.

21. JavaScript. *JavaScript.* [Online] 25 11, 2021. https://developer.mozilla.org/es/docs/Web/JavaScript.

22. Object oriented programming. *Wikipedia.* [Online] 25 11, 2021. https://en.wikipedia.org/wiki/Object-oriented_programming.

23. Threejs. *Threejs.* [Online] 11 25, 2021. https://threejs.org/.

24. Html Canvas. *w3schools.* [Online] 11 25, 2021. https://www.w3schools.com/html/html5_canvas.asp.

25. React Component. *React.* [Online] 11 26, 2021. https://reactjs.org/docs/react-component.html.

26. Syntax Highlighting. *Wikipedia.* [Online] 11 26, 2021. https://en.wikipedia.org/wiki/Syntax_highlighting.

27. glViewport. *Khronos.* [Online] 12 9, 2021. https://www.khronos.org/registry/OpenGL-Refpages/es2.0/xhtml/glViewport.xml.

28. glScissor. *Khronos.* [Online] 12 9, 2021. https://www.khronos.org/registry/OpenGL-Refpages/es2.0/xhtml/glScissor.xml.

29. glClear. *Khronos.* [Online] 12 9, 2021. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glClear.xhtml.

30. Scissor test. *Khronos.* [Online] 12 9, 2021. https://www.khronos.org/opengl/wiki/Scissor_Test.

31. Textarea. *Developer mozilla.* [Online] 11 28, 2021. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea?retiredLocale=ca.

32. Parsing. *Wikipedia.* [Online] 11 29, 2021. https://en.wikipedia.org/wiki/Parsing#Parser.

33. Abstract Syntax Tree. *Wikipedia.* [Online] 11 29, 2021. https://en.wikipedia.org/wiki/Abstract_syntax_tree.

34. glsl-parser. *Npmjs.* [Online] 29 11, 2021. https://www.npmjs.com/package/glsl-parser.

35. GLSL Functions. *Shaderific.* [Online] 11 29, 2021. https://www.shaderific.com/glsl-functions.

36. Introducing JSON. *JSON.* [Online] 12 1, 2021. https://www.json.org/json-en.html.

37. Asymptotic computation complexity. *Wikipedia.* [Online] 12 1, 2021. https://en.wikipedia.org/wiki/Asymptotic_computational_complexity.

38. Vertex_Specification. *khronos.* [Online] https://www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Array_Object.

39. Average Junior Software Engineer Salary in Spain. *Payscale.* [Online] September 20, 2021. https://www.payscale.com/research/ES/Job=Junior_Software_Engineer/Salary.

40. Average Project Manager, Information Technology (IT) Salary in Spain. *Payscale.* [Online] September 25, 2021. https://www.payscale.com/research/ES/Job=Project_Manager%2C_Information_Technology_(IT)/Salary.

41. Average User Interface Designer Salary in Spain. *Payscale.* [Online] September 26, 2021. https://www.payscale.com/research/ES/Job=User_Interface_Designer/Salary.

42. Average Web Developer Salary in Spain. *Payscale.* [Online] October 1, 2021. https://www.payscale.com/research/ES/Job=Web_Developer/Salary.

43. Fragment (Computer Graphics). *Wikipedia.* [Online] October 2, 2021. https://en.wikipedia.org/wiki/Fragment_(computer_graphics).

44. Introduction to Nodes. *Blender.* [Online] September 27, 2021. https://docs.blender.org/manual/en/2.79/render/blender_render/materials/nodes/introduction.html.

45. Rendering Pipeline Overview. *khronos.* [Online] September 28, 2021. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

46. Tarifas de luz. *comparadorluz.* [Online] October 7, 2021. https://comparadorluz.com/pymes/tarifas.

47. What is Scrum? *Atlassian.* [Online] October 10, 2021. https://www.atlassian.com/agile/scrum.

48. Uniform. *Khronos.* [Online] 11 26, 2021. https://www.khronos.org/opengl/wiki/Uniform_(GLSL).

49. Lexical analysis. *Wikipedia.* [Online] 11 29, 2021. https://en.wikipedia.org/wiki/Lexical_analysis#Token.

50. Expresive Power. *Wikipedia.* [Online] 11 29, 2021. https://en.wikipedia.org/wiki/Expressive_power_(computer_science).