

# Semi-Automatic Evaluation for Computer Graphics Problems

Final Thesis

**Cristina Raluca Vijulie**

Director: Carlos Andújar Gran

Computer Science department (*CS*)

Bachelor Degree in Informatics Engineering

Computing Specialization



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH  
Facultat d'Informàtica de Barcelona



Defense date: April 23, 2019

## **Abstract**

This project is aimed at automatizing part of the grading and self-assessment processes for computer graphics problems in computer science courses. These courses often use tools that analyze the output of the programs, but the features extracted from syntactically analyzing the source code can give more insight on code quality and reduce correction time. This project presents a high-level Python API allowing instructors to compose correction rubrics based on a syntactic and semantic analysis of the source code of student submissions. The API provides functions to query the most relevant components of the OpenGL Shading Language, including variable declarations, assignments, function calls, conditional sentences and loop statements. The implementation is based on the ANTLR parser generator and has been tested on shader submissions in a Computer Graphics course.

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Problem formulation . . . . .	4
1.2	Objectives . . . . .	5
<b>2</b>	<b>Context and scope</b>	<b>6</b>
2.1	Context . . . . .	6
2.2	Scope . . . . .	6
2.3	Possible Obstacles . . . . .	7
<b>3</b>	<b>Methodology and Resources</b>	<b>9</b>
3.1	Software development methodology . . . . .	9
3.2	Tools . . . . .	9
3.3	Material Resources . . . . .	10
3.4	Human Resources . . . . .	10
<b>4</b>	<b>Action Plan</b>	<b>11</b>
4.1	Initial plan deviations . . . . .	13
<b>5</b>	<b>Project sustainability</b>	<b>15</b>
5.1	Initial Assessment . . . . .	15
5.2	Economic Dimension . . . . .	15
5.2.1	Budget . . . . .	16
5.3	Environmental Dimension . . . . .	17
5.4	Social Dimension . . . . .	18
5.5	Sustainability Matrix . . . . .	19
<b>6</b>	<b>Software design</b>	<b>20</b>
<b>7</b>	<b>OpenGL Shading Language</b>	<b>21</b>
7.1	Syntax and Grammar . . . . .	23
<b>8</b>	<b>Grammar and Parser</b>	<b>24</b>
8.1	Context Free Grammars . . . . .	24
8.2	ANTLR Grammars . . . . .	25

<b>9</b>	<b>Listeners and Visitors</b>	<b>28</b>
9.1	ANTLR listener class . . . . .	28
9.2	ANTLR visitor class . . . . .	29
<b>10</b>	<b>Python API: glcheck</b>	<b>30</b>
10.1	Syntactic analysis . . . . .	31
10.2	Semantic analysis . . . . .	38
<b>11</b>	<b>Results</b>	<b>45</b>
11.1	Quads . . . . .	45
11.2	Dithered cartoon shading . . . . .	46
<b>12</b>	<b>Conclusions and future work</b>	<b>47</b>
12.1	Conclusions . . . . .	47
12.2	Future work . . . . .	47
<b>A</b>	<b>Python API Documentation</b>	<b>51</b>
A.1	Module <b>glcheck</b> . . . . .	51
A.2	Configuration . . . . .	52
<b>B</b>	<b>ANTLR4 GLSL Grammar</b>	<b>54</b>

# 1 Introduction

## 1.1 Problem formulation

Computer graphics is a field which studies methods for digitally synthesizing and manipulating visual content. Computer graphics courses often use problem solving activities where students have to research and develop algorithms to generate a certain specified visual output. These kind of problems are also used in the *Graphics*<sup>1</sup> and *Interaction and Interface Design*<sup>2</sup> courses from the *Bachelor Degree in Informatics Engineering* at UPC<sup>3</sup>.

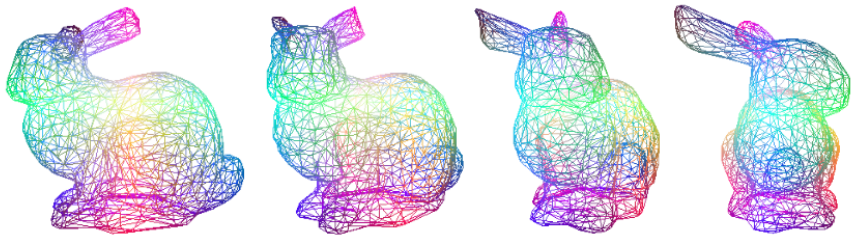
The problems provide a statement, like the one in In Figure 1, which details a certain visual operation to be implemented in different steps of the rendering pipeline, and can be applied to any 3D model. These operations can vary from changes in colour or shading to changes in the model's wire-frame, deformations or animation.

**20. Auto-rotate (auto-rotate.\*)** (1<sup>er</sup> control laboratori, curs 2011-12, Q2)

---

Escriu un **vertex shader** que, abans de transformar cada vèrtex, li apliqui una rotació al voltant de l'eix Y. El shader rebrà un **uniform float speed** amb la velocitat de rotació angular (en rad/s). Feu servir la variable **uniform float time** per l'animació.

Aquí teniu els resultats (en wireframe) amb el bunny, amb speed = 0.5 rad/s:



time=0
time=1
time=2
time=3

Figure 1: Example of an exercise from the Graphics subject where a 3D bunny has to be rotated over time.

To test the implementation and correctness of the student's algorithms the *Viewer* application is used. This is an application developed by the course teachers that lets a user provide models, code and other attributes to visualize

<sup>1</sup> Graphics course at FIB: [www.fib.upc.edu/en/studies/bachelors-degrees/bachelor-degree-informatics-engineering/curriculum/syllabus/G](http://www.fib.upc.edu/en/studies/bachelors-degrees/bachelor-degree-informatics-engineering/curriculum/syllabus/G)

<sup>2</sup> Interaction and Interface Design course at FIB: [www.fib.upc.edu/en/studies/bachelors-degrees/bachelor-degree-informatics-engineering/curriculum/syllabus/IDI](http://www.fib.upc.edu/en/studies/bachelors-degrees/bachelor-degree-informatics-engineering/curriculum/syllabus/IDI)

<sup>3</sup> Bachelor Degree in Informatics Engineering: [www.fib.upc.edu/en/studies/bachelors-degrees/bachelor-degree-informatics-engineering](http://www.fib.upc.edu/en/studies/bachelors-degrees/bachelor-degree-informatics-engineering)

the resulting output [Gra16]. It also provides a testing option that renders the difference between still frames of the expected output and the student's output. Although it is a very complete and complex program, determining the correctness of a solution based only on the visual output of this application can be sometimes impossible. Different implementations of graphic cards lead to slightly different results for the same code depending on the card it is running on [Fay06]. These small differences can have a snowball effect on many correct or almost correct solutions and make it very difficult to determine if there is a real mistake in the code or just noise. This is a problem for both the auto-evaluation process of the student developing the solution and for the teacher that has to evaluate it.

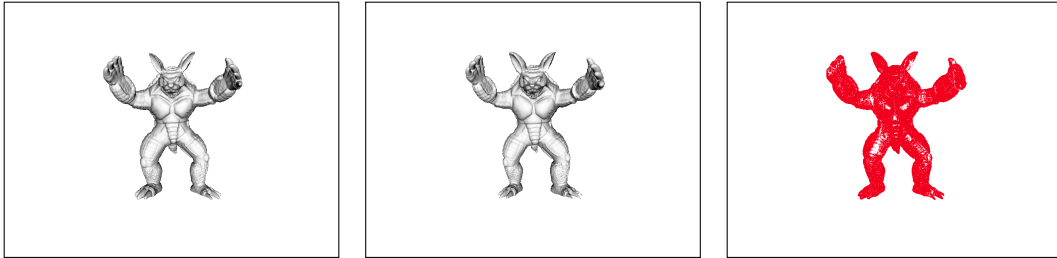


Figure 2: Example of the current output of the *Viewer* (right) as the difference between a correct output (center) and an incorrect one (left). The output from both images is almost identical, and we don't know if the difference is due to precision errors or bad code.

## 1.2 Objectives

This project's main goal is to create a system that facilitates the correction of computer graphics problems. The achievement of this goal will serve two primary objectives:

- **Speed-up and simplify the exam correction process:** Make it easier and faster for teachers to assess correction rubrics for exam exercises.
- **Facilitate students' auto-evaluation:** Provide a way for students to test their solutions to training exercises. Give them accurate information about the errors in the output of their solutions and also errors in their code.

## 2 Context and scope

### 2.1 Context

This project will serve the purpose of aiding the *IDI* and *G* courses teachers in the exam correction process. If implemented correctly, the developed system will reduce the time and effort spent on basic and repetitive tasks when correcting student exams. Most of these tasks include performing simple *find* operations on the submitted code in order to assess problem-specific requirements such as using a mandatory function or performing a required operation on a variable, among many others.

When a student's solution does not yield a correct result, performing more complex and not as problem-specific tasks may be necessary. Many common mistakes can make an "apparently correct" code behave in an undesired way, like using a variable of an incorrect type or not applying the necessary transformations to a model. For example, in shader programs, a vertex's coordinates are given in *object space* (that is to say, the coordinates are relative to the object containing the vertex). In order to properly render a scene, the coordinates of vertices have to be transformed into *clip space* by performing a chain of transformations. It may be hard to detect at first glance whether a student has applied the necessary transformations to a vertex and/or if those transformations are correctly performed. Having a system that automatically detects inconsistencies in a vertex's coordinate spaces along the code of the shader could simplify this process by narrowing down the possible causes for mistakes and showing the teacher which lines of the code could be problematic.

Students should also benefit from this project since they will be provided with a more accurate tool to help in their auto-evaluation and learning processes. Applying this tool correctly is likely to show a positive effect on their grades.

### 2.2 Scope

In order to achieve the stated objectives, a rubric assessment tool will be implemented. This tool will consist of a public python library with functions specified by the subject's coordinator. For this, a large set of exam problems and student submissions will be studied together with the course's teachers in order to determine the most common mistakes made by the students as well as all the tools needed for specifying correction rubrics. An example of a rubric are shown in Figure 3.

Since many rubrics cannot be verified completely automatically, the library will also provide tools to locate relevant portions in the code and highlight them, with a range of output formatting options. A detailed specification of

```
1 R("No_calls_to_mix", vs.numCalls("mix") < 1)
```

Figure 3: Example of a rubric that assess whether the function "mix" has been used in the code.

all the functions needed will be adjusted so that fits both students and teachers needs, following a modular implementation so that more features can be easily added in the future. It is out of the project's scope to write assessment rubrics for existing exercises, this task will have to be done by the subjects' teachers.

For the objectives to be completely achieved, a solution testing tool will be developed, which will consist in a visualization application for 3D models and shaders with an improved testing functionality. This testing functionality will provide an interactive view of both the output of a student's solution to an exercise and the correct output for that exercise. For the advanced correction algorithm different obfuscation techniques will be tried so that a student's solution can be executed at the same time as the official solution on the same machine, without exposing the official solution and minimizing the impact of obfuscation on the program's performance. This interactive visualization of the outputs will be combined with the information of the rubric assessment to provide an orientative yet detailed summary of possible errors in the student's code and errors in the output of the program. It is out of the project's scope to write solutions for the course's exercises.

### 2.3 Possible Obstacles

Some of the obstacles that may be encountered during the development of this project can be:

- **Code disclosure:** One of the greatest problems of comparing a student's code with the correct code in real time is that the correct solution must be executed on foreign computers. Therefore, although it can be hindered, it is almost impossible to prevent users from having access to the correct answers to the course's problems.
- **Parsing:** In order to analyze the student's code a parsing operation has to be performed. This is done by running the code through a specific grammar and building an Abstract Syntax Tree (AST), which is data structure containing a hierarchy of statements and functions that make the analysis of the code possible [And18]. Generating a correct AST is a key element of the code analysis, and a correct grammar for the programming languages supported (C++ and GLSL) will be needed. It may be difficult to find appropriate tools to generate this AST and/or correct grammars, therefore it may be necessary to develop these as part of the project, which can cost a substantial amount of time.



- **Tool Search:** A big part of this project relies on finding suitable cross-functional tools that already implement the low-level wide range of tasks that assembly the project. Since implementing each one of the functionalities without using any additional libraries would be practically impossible, it is important to spend a substantial amount of time finding the best tools for the job. Not finding a suitable free software library for a determined module will add a significant amount of hours to the development of the project.
- **Hofstadter's law:** "*It always takes longer than you expect, even when you take into account Hofstadter's Law.*" [Hof99] This law states the difficulty of planning ahead and estimating the development time for complex tasks, that even when taking into account their difficulties and obstacles take longer than expected.

## 3 Methodology and Resources

### 3.1 Software development methodology

In this project it is important to keep a clean and modular code that will make it easy to add future functionalities. The most important thing to keep in mind are the stakeholder's priorities, which will be detailed along the course of the project. These requirements discard the use of linear methodologies like Waterfall or Rapid. The best fitting method for implementing this type of project is an Agile one, primarily on grounds of the need to respond to specification changes and viability predictions for certain features [Agi18]. The Lean Software Development looks like the best fitting agile method due to its focus on principles like learning amplification and waste elimination [Pop03]. The development will be monitored using a Kanban board and Test-driven development.

### 3.2 Tools

- **Test-driven Development:** This technique is very important for this kind of projects composed of very distinct functionalities where optimization and efficiency are key elements. It will be necessary to write tests for each new functionality implemented and make sure all the tests pass after optimization and refactoring processes [Pat12]. A continuous integration tool will be used to ensure each step of the implementation is correct and minimize the amount of bugs and undesired behaviours of the application.
- **Git:** The code will be developed and maintained using the Git program. A master branch will hold stable code and each new feature or refactoring job will be done in a separate branch. Once the feature or refactored code pass all the tests they will be merged into the master branch and tested once more. Commits will be made for every step in the development of a feature, allowing for easy monitoring and error detection. The GitHub platform will be used for an easy on-line remote access to the project and communication with the project director.
- **Kanban board:** An online Trello board synchronized with the GitHub repository will be used as a Kanban board. The board will feature a pool of functionality ideas to be detailed with stakeholders, a *To Do* pool of already defined functionalities that have to be implemented and a minimal number of *Work In Progress* features and tests to be finished and validated.
- **Bi-weekly meetings:** Periodical meetings with the thesis director and key stakeholder will be held. In this meetings the current progress of the

progress will be showed and discussed, functionalities will be reviewed and new ideas will be pondered and specified.

### **3.3 Material Resources**

- PC with a Linux Operating System
- Server with Graphics card, network connection and Linux operating system
- Fast internet access

### **3.4 Human Resources**

- The author of the TFG with a dedication of 30 to 42h/week depending on the project's progress and encountered obstacles.
- The director of the TFG providing help with the specifications as a Project Manager and consultations about the implementation of the project. 0.5 to 1h/week

## 4 Action Plan

The project will be developed between July 2018 and April 2019. This project consists of two parts: a lexical analyzer and an interactive viewer. The first part of the project will be finished by 23 January 2019. The second part is not crucial for the stakeholder, therefore it will be implemented last. In a worst case scenario where obstacles prevent the first part of the project for being finished before the deadlines, the second part will be canceled and the project will be finished in time. If everything goes according to plan, the second part will be finished by 23 April 2019.

Each one of the mentioned phases has a set of tasks and sub-tasks. The project will follow a LEAN methodology: for each task, the requirements and goal will be analyzed, then implemented [Pop03]. A testing phase will follow each implementation, followed by a refactorization if applicable. These testing/refactorization process will be iterated until the quality of the implementation meets the requirements to be integrated in the whole project. Since the exact implementation time for each task is difficult to estimate and adding new tasks may be necessary, the required time for each task has been estimated using a deviation based on the task's complexity and a general deviation depending on the task's level of risk. The deviations used for each level of risk are: **low**: 110%, **medium**: 150% and **high**: 250%. The project has been planned at a worst-case scenario for each task, therefore most tasks will likely start before it was planned. Bi-weekly meetings with the project director and stakeholders will be held to review the project's progress and discuss any possible obstacle and how to avoid it.

- 1) **Static Analysis Tool**: This phase consists of three major tasks. The first two are very similar in organization so they will be explained in the same section for simplicity.

Total estimated time: **320h**

- **Parsers**: The first two phases consist of implementing an OpenGL Shading Language and a C++ language parsers that analyzes shader codes and C++ application codes. For each programming language a parser is needed.
  - **Research** for open-source, functional parsers. **2x 5h**, Analyst. Risk: **low**. This task has no precedence.
  - **Modification** of such parsers to meet the project's requirements. **2x 15h**, Developer/Tester. Risk: **high** This can only be done once the parser is found and installed.
  - **Specification** of functionalities for each language. **2x 10h**, Analyst. Risk: **low**. This task has no precedence.

- **Implementation** of the functionalities where the parsers are used. **2x 100h**, Developer/Tester. Risk: **medium**. This can only be done once the parser is modified to the specification and works in the project's framework.
- **Python library:** This task consists of integrating both parsers in a public python module and implementing high level functions to specify rubrics using the parser's functionalities.
  - Parser functions **refactorization**. **30h**, Developer/Tester. Risk: **medium**. This task can be done only once all the previous tasks are completed.
  - Python high-level function **specification**, Analyst. **10h**. Risk: **low**. This task has no precedence.
  - Python high-level function **implementation**. **20h**, Developer/Tester. Risk: **medium** This task can only be done after the specification and parser implementation tasks are done.

2) **Dynamic analysis tool:** This phase consists in analyzing the existing viewer tool and available hardware to specify an improved visualization tool and implement it.

Total estimated time: **180h**

- Viewer **analysis**. **20h**, Analyst. Risk: **low**. This task has no precedence.
- Hardware **analysis**. **10h**, Analyst. Risk: **low**. This task has no precedence.
- Viewer **specification**. **20h**, Analyst. Risk: **low**. This task can only be done once the existing Viewer tool and hardware have been analyzed.
- Hardware and Viewer **implementation**. **120h**, Developer/Tester. Risk: **high**. This task can only be done once the specification is complete.

3) **Documentation:** In this stage the memoir and the Presentation for the TFG exposition will be prepared.

Total estimated time: **100h**

- Memoir **redaction**. **65h**, Project Manager. Risk: **medium**. This task has no precedence.
- Memoir **revision**. **15h**, Project Manager. Risk: **low**. This task can only be done when the memoir has been finished.

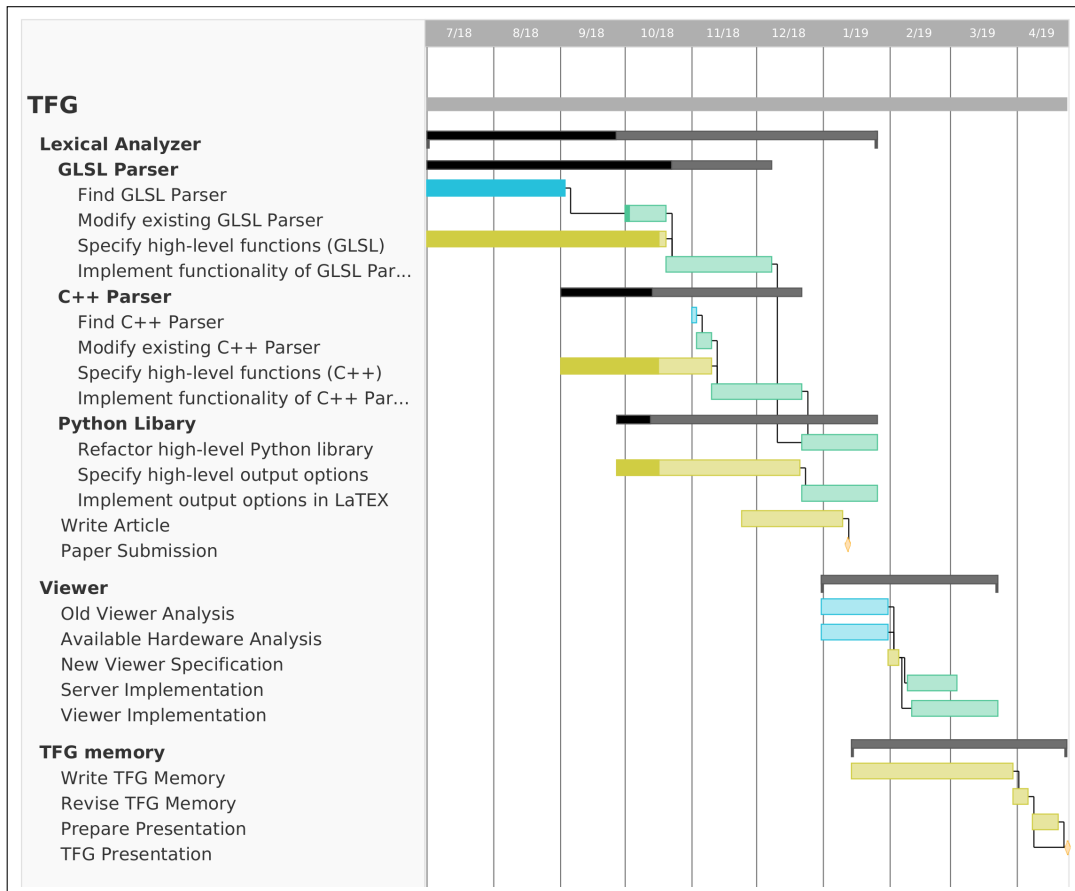


Figure 4: Gantt Chart of the project's tasks.

- **Presentation preparation. 20h, Project Manager. Risk: medium.**  
This task can only be done when the memoir has been finished and revised.

## 4.1 Initial plan deviations

This project's development process has had a primary focus in the stakeholder's priorities and reviews of the initial specification. Given the nature of the agile methodology, changes in the initial plan were expected, adapting the specification to new ideas and improvements. Some expected difficulties have also been encountered, resulting in an increased dedication to the initial tasks to ensure a good foundation for the project.

### Static Analysis Tool

In order to develop the static analysis tool, GLSL and C++ parsers were necessary. The way these parsers are integrated in the project is vital, since all of the functionalities have to be built around the parser's specifications.

The initial choice for a GLSL parsing python library was discarded shortly after starting the implementation phase for some of the functionalities, due to a lack in its documentation and many issues encountered in the process. A second research phase led to the final choice of using the ANTLR library and open-source GLSL and C++ grammars.

After doing some tests with the official ANTLR C++ grammar, it was determined that the task of modifying it to fit the project's purposes would take much longer than expected and that the main priority would be to have a functional GLSL static analysis tool. The integration of C++ code analysis is finally left as future work.

### **Dynamic Analysis Tool**

The dynamic analysis tool development has finally been left out of the scope of the project. The delays discussed in the previous section and perhaps an over-optimistic estimation of the dedication time available have left a tight schedule. Taking into account the time dedicated to the implementation of the static analysis tool and the documentation of the project, the 450h corresponding to 18 ECTS credits that a final thesis requires<sup>4</sup> were already being met. This was, nevertheless, an expected obstacle, and has no major effect on attaining the main goal of the project.

---

<sup>4</sup> [https://ec.europa.eu/education/ects/users-guide/glossary\\_en.htm#workload](https://ec.europa.eu/education/ects/users-guide/glossary_en.htm#workload)

## 5 Project sustainability

### 5.1 Initial Assessment

There is a considerable understanding of the economic, social and environmental sustainability aspects of IT projects. Engineering bachelor degrees invest a high amount of time in training students on identifying causes and consequences of design problems, primarily by teaching how to associate an unsolved problem with a solved one and adapting the existing solution to the unsolved problem. This process can be extrapolated to the economic, social and environmental dimensions of an IT project just by learning how to properly analyze and measure their costs and impact.

In IT, and specifically software projects, it is relatively easy to determine their economic impact, since each one of their activities and potential risk factors can be predicted very accurately with considerable confidence. This leads to very realistic budgets that are easy to control and update as soon as an obstacle is encountered.

Environmental impact is easy to measure in software projects too, since there are very few material resources needed for their development and their usage time and energy consumption levels are precisely defined.

The hardest impact for an engineer to determine is the social. Engineering bachelor degrees teach very few about human and social behaviors, which is a very complex subject in itself and difficult to predict accurately. That being said, there are some basic yet crucial social aspects of any IT project which impact can easily be measured and taken into account like security and accessibility. These are key features for which evaluation is feasible in any IT project, since there have been so many problems with these aspects in the last 20 years, there are also a lot of solutions and improvements to learn from.

We can conclude that we have been provided with enough tools in order to measure and improve sustainability in all its dimensions in any software development project.

### 5.2 Economic Dimension

All costs derived from the development of the project have been calculated, taking into account all the extra time necessary to develop it in the case of encountering a predicted obstacle. The total cost of the project including direct, indirect, hardware and software costs is of **10,282.77 euros** (see subsection 5.2.1).

The problem that this project is designed to solve is correcting exams in the IDI and G courses, a task that is done completely manually at the time. Exam correction in universities is performed by professors that have an elevated cost



per hour of work, therefore human resources costs can also be reduced by this project by reducing the amount of hours that teachers have to work.

### 5.2.1 Budget

#### Direct Costs

In software projects, direct costs mean human resources costs directly related to the Gantt chart activities. This project has a total estimated time of: **680h**. In a normal company, there would be different people involved in the project with different roles in it. Usual roles and estimated salaries for a software project are:

- Project manager: 20 €/hour [Pay18b]
- Programmer - Analyst: 13.5 €/hour [Pay18a]
- Programmer - Developer - Tester: 12 €/hour [Pay18c]

For this project, the roles will be developed only by the project author in all three roles and project director in the role of Project Manager. The human resources cost has been calculated by taking into account each one of the activities in the Gantt chart [4] in addition to the work derived from the GEP course and bi-weekly meetings with project director/stakeholder. All activities' cost has been detailed in Table 1 and a summary of the total human resources cost can be seen in Table 2. Roles *Analyst*, *Project Manager* and *Developer/Tester* have been abbreviated to *A*, *PM* and *D/T* respectively, for the sake of simplicity.

#### Indirect Costs

Indirect costs of this project consist in power consumption and internet access. A detailed review of the costs can be seen in Table 3.

#### Hardware

According to the Spanish *Agencia Tributaria* amortization time for computers is 8 years [Age18]. Taking this into account, the cost of the hardware needed for the project development phase is detailed in Table 4.

#### Software

The only proprietary software used during the project will be GitHub. A private repository is needed to host the project and it will be used for the entire duration of the project. Detailed costs can be seen in Table 5.

Task	Role	Dedication	Cost
Research parsers	A	10h	135 €
Modification of parsers	D/T	30h	360 €
Specification of functions	A	30h	405 €
Implementation of functions	D/T	200h	2400 €
Parser refactorization	D/T	30h	360 €
Output functions specification	A	10h	135 €
Output functions implementation	D/T	20h	240 €
Viewer analysis	A	20h	270 €
Hardware analysis	A	10h	135 €
Viewer specification	A	20h	270 €
Hardware+Viewer implementation	D/T	120h	1440 €
Memoir redaction	PM	65h	1300 €
Memoir revision	PM	15h	300 €
Presentation preparation	PM	20h	400 €
GEP	PM	60h	1200 €
Meetings	PM	20h	400 €

Table 1: Human Resources costs for each task

### Monitoring

For budget monitoring, the total duration of each task will be reviewed weekly, and the budget will be updated with the costs for the real duration and derived costs of each task.

### 5.3 Environmental Dimension

This project has very little environmental impact. Since it is a software project, the only impact derived from its development and useful life is that of the carbon emissions generated from electricity usage. The electricity usage of the project has been determined to be of about 120kW in the development phase. 47kg of CO<sub>2</sub> are generated to produce this amount of electricity [Gen]. The impact derived from its useful life may result in a decrease in the waste of paper by the teachers, since it will not be necessary to print the student's

Role	Dedication	Cost
Project Manager	180h	3600 €
Analyst	100h	1350 €
Programmer/Tester	400h	4800 €
<b>Total</b>	<b>680 h</b>	<b>9750 €</b>

Table 2: Total Human Resources costs

Item	Price	Quantity	Cost
Electricity	0.116 €/kWh	120 kW	13.92 €
Internet Access	35 €/month	10 months	350 €
<b>Total</b>			<b>363.92 €</b>

Table 3: Indirect costs

solutions anymore.

## 5.4 Social Dimension

In terms of personal growth, I believe this project has given me the opportunity to learn a lot about graphic processing at the same time as code analysis and program synthesis. It has also given me the opportunity to work with the university's graphics department and provided insight on the work of being a teacher.

This project affects two sectors of people: the course's **teachers** and the **students**. The students and teachers of this course use a very simple system to correct exercises, based on a plain difference between still images of a correct solution to an exercise and the student's solution. This makes it exceptionally hard for students to be sure if their solutions are really correct when practising, since teachers take into account the quality of the code itself not only the output of programs. Teachers, on the other hand, have to spend a huge amount of time manually assessing student's programs while taking a lot of correction rubrics into account. Teachers are the main beneficiary of this project since their time working on correcting exams will be drastically reduced. A certain amount of time will be needed to specify the rubrics of each exercise before the correction of the exam, but this task can be done only once for each exercise, therefore only one teacher has to do it, and decrease the correction time for any teacher correcting that exercise ever after.

Students also benefit from this project since they can use the system for

Item	Price	Charge-off	Usage time	Cost
Laptop PC	800 €	8.33 €/month	9 months	75 €
Server	1300 €	13.54 €/month	2.5 months	33.85 €
<b>Total</b>				<b>108.85 €</b>

Table 4: Hardware costs

Item	Price	Usage time	Total Cost
GitHub Micro Plan	6 €/month	10 months	60 €
<b>Total</b>			<b>60 €</b>

Table 5: Software costs

auto-evaluation purposes, which is likely to improve their understanding of the subject and therefore improve their chances of getting a better grade.

Even though this project has been oriented to the UPC's graphics courses, it could be extended to other kinds of programming exercises. Thirds also may be positively affected if the project is used in other universities and courses.

## 5.5 Sustainability Matrix

Table 6 contains the scores of the PPP row of the sustainability matrix.

	Economic	Environmental	Social
<b>PPP</b>	4	8	9

Table 6: Sustainability Matrix

## 6 Software design

The tool we need to develop for this project has to be able to assess rubrics on a large number of codes. Each one of the codes will be a student's attempt at solving a computer graphics problem. These submissions will be in the GLSL language, in the form of vertex shaders, fragment shaders and geometry shaders. Since our main objective is to be able to provide valuable and extended feedback on poor quality code, custom rubrics will have to be designed and tuned for each problem. In order to provide this functionality we can split our project in three modules.

- **Grammar and Parser:** The parser is the part of the program responsible of reading the GLSL code of each submission and build a data structure that holds the information of each construct present in the source code. This data structure is known as AST (abstract syntax tree). In order to build a parser for a programming language we need to specify the grammar of this language.
- **Listeners and Visitors:** The listener and visitor modules will allow us to traverse the AST built by the parser and extract all the information we need to know about it. We can implement various listeners and visitors in order to provide each functionality required by the project.
- **Python API:** The API is the module that will implement high level functions that give access to the parser's functionalities through listeners and visitors. The main goal of this API is to provide a simple and versatile interface whose functions can be combined in order to assess a wide range of rubrics, from simple and general to complex and specific ones.

A detailed documentation of the API is provided in Appendix A.

## 7 OpenGL Shading Language

```
1 #version 330 core
2
3 layout (location = 0) in vec3 vertex;
4 layout (location = 1) in vec3 normal;
5 layout (location = 2) in vec3 color;
6
7 out vec3 N;
8 out vec3 P;
9
10 uniform mat4 modelViewProjectionMatrix;
11 uniform mat4 modelViewMatrix;
12 uniform mat3 normalMatrix;
13
14 void main() {
15     P = (modelViewMatrix * vec4(vertex.xyz, 1)).xyz;
16     N = normalize(normalMatrix * normal);
17     gl_Position = modelViewProjectionMatrix * vec4(vertex.xyz, 1.0);
18 }
```

Figure 5: Vertex shader used to calculate lighting in a Graphics exercise.

The OpenGL Shading Language, or GLSL, is a domain-specific language used in the programmable stages of the OpenGL rendering pipeline. This pipeline is responsible for rendering a 3D scene on a 2D screen, by applying a series of steps on the input data. Some of these steps are programmable and perform operations specified by the user on the data sent to the pipeline. In the Graphics course, students learn how to customize the rendering process by programming geometry, vertex and fragment shaders.

### Geometry shaders

Geometry shaders can be used to generate new primitives in the scene such as points, lines, triangles, meshes, etc. The vertices created in this step will be passed to the vertex shader along with all of the original primitives initially passed to the pipeline.

### Vertex shaders:

Vertex shaders are executed once for every vertex passed to the pipeline. These shaders are widely used to modify the vertex properties such as position, color or texture coordinates. Figure 5 shows an example of a vertex shader code used in the process of simulating the lighting of a scene.

## Fragment shaders:

Fragment shaders are executed once for every *fragment* in the pipeline. Fragments are a way to represent all the information that has to be rendered on a single pixel of the screen, and have no information about the scene's initial geometry. In most cases, a fragment shader is used to calculate a pixel's color, based on the input data received from previous steps in the pipeline. This information can be used in order to apply effects like lighting, shadows, highlights or bump mapping and other advanced techniques. Figure 6 contains the fragment shader code corresponding to the vertex shader from Figure 5.

```
1  #version 330 core
2
3  uniform vec4 lightAmbient;
4  uniform vec4 lightDiffuse;
5  uniform vec4 lightSpecular;
6  uniform vec4 lightPosition;
7
8  uniform vec4 matAmbient;
9  uniform vec4 matDiffuse;
10 uniform vec4 matSpecular;
11 uniform float matShininess;
12
13 in vec3 N;
14 in vec3 P;
15 out vec4 fragColor;
16
17 void main() {
18     N = normalize(N);
19     vec3 V = normalize(vec3(0, 0, 1));
20     vec3 L = normalize(lightPosition.xyz - P);
21     vec3 H = normalize(V+L);
22     float ldiff = max(0, dot(N, L));
23     float lspec = max(0, dot(N, H));
24     if (ldiff > 0)
25         lspec = pow(lspec, matShininess);
26     else
27         lspec = 0;
28     lightAmbient = matAmbient * lightAmbient;
29     lightDiffuse = matDiffuse * lightDiffuse * ldiff;
30     lightSpecular = matSpecular * lightSpecular * lspec;
31     fragColor = lightAmbient + lightDiffuse + lightSpecular;
32 }
```

Figure 6: Fragment shader used to calculate lighting in a Graphics exercise.

## 7.1 Syntax and Grammar

The GLSL syntax and grammar are based on the C Programming Language, with minor differences. GLSL supports an array of specific constructs, such as uniforms, layouts and storage qualifiers, that apply only to the context of a rendering pipeline. Some restrictions are also applied to this context, for example, recursive functions are not supported. The most relevant constructs used in the Graphics course are the storage qualifiers `in` and `out`. When a variable is declared as `in`, its value is passed to the shader by a previous stage in the pipeline. When a variable is declared as `out`, its value is set by the shader and passed to the following stages of the pipeline.

In the GLSL language, as well as in all other imperative programming languages, a program is formed mainly by a list of statements. A statement can either be a compound statement, meaning it creates a nested list of statements, or a simple statement. We will focus mainly on the following statements, without entering in specific details that are not relevant to this project.

- **Function definitions:** `int main(){...}`
- **Declarations:** `int a;`
- **Assignments:** `a = 1;`
- **Flow control sentences:** `while`, `for`, `if`, `do`, `switch`.
- **Jump sentences:** `continue`, `return`, `break`.
- **Expressions:**
  - **Constants:** integers, floats and booleans.
  - **Constructors:** `vec3(1,1,1)`.
  - **Function calls**
  - **Identifiers**
  - **Array and struct selectors:** `a[i]` and `a.i`.
  - **Arithmetic operator expressions:** `*`, `/`, `+`, `-`, `%`, `++` and `--`.
  - **Logic operator expressions:** `!`, `>=`, `>`, `<=`, `<`, `==`, `!=`, `&&`, `||` and `^^`.
  - **Bitwise operator expressions:** `&`, `|`, `^`, `»`, `«` and `~`.
  - **Arithmetic assignment expressions:** combine an assignment expression with an arithmetic operator or a bitwise operator, for example `a += 1;`.



## 8 Grammar and Parser

### 8.1 Context Free Grammars

The GLSL programming language follows the rules of a context-free grammar. A context-free grammar is a set of production rules that define all of the words accepted by this grammar. Given an input, the starting rule of the grammar is applied. As long as the input follows the rules of the grammar, the input is accepted. If a symbol from the input does not match with any rule in the grammar, the word is rejected. The left-hand side of a production rule is always a non terminal symbol, while the right-hand side can be a combination of terminal and non-terminal symbol. In Figure 7 we can see a simple grammar that accepts the words " $\alpha\alpha$ ", " $\alpha\beta$ ", " $\beta\alpha$ ", " $\beta\beta$ ".

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow \alpha \\ A &\rightarrow \beta \end{aligned}$$

Figure 7: A simple grammar with a start rule S.

In programming, terminal symbols are called "tokens", and they are defined in the grammar as simple strings or patterns of characters. A program called lexer is in charge of reading the source code one character at a time and build a stream of tokens. A parser then analyzes the tokens and applies the rules of the grammar, either accepting or rejecting given input. In the process of applying the grammar rules, the parser also builds an abstract syntax tree. An abstract syntax tree is a "tree-like" data structure, with the starting rule at its root, in which each grammar rule applied to the input has a node labeled with the left-hand side of the rule, and as many children as the symbols in the right-hand side of the rule.

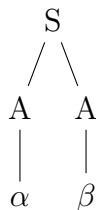


Figure 8: AST created by parsing the word " $\alpha\beta$ "

```

1  LETTER
2      :  [a-z]
3      |  [A-Z]
4      |  '_'
5      ;
6
7  IDENTIFIER
8      :  LETTER (LETTER|DIGIT)*
9      ;

```

Figure 9: An ANTLR4 example with grammar rules for identifier definition. Identifiers are used to represent the names of variables, functions and structures. These can be any combination of uppercase and lowercase letters, as well as the underscore character.

## 8.2 ANTLR Grammars

Programming a lexer and a parser is a complicated task, and out of the scope of the project, since there are already many tools that do this automatically given a grammar, and produce all the code needed to parse and execute a program. ANTLR (ANOther Tool for Language Recognition) is an open-source tool that generates parsers for reading, processing, executing, or translating structured text or binary files<sup>5</sup>. This project is developed using the latest version of ANTLR (ANTLR4) and an open-source GLSL grammar written in ANTLR4 syntax. The open-source grammar [Lon15] had to be modified in order to meet the requirements of the project and due to errors in its definition. The final grammar used for the project can be found in Appendix B.

In order to define a grammar in ANTLR we first must define the tokens, which are the terminal symbols (or words) that our language recognizes. Tokens are defined as simple rules that match either to a string or to a simple pattern of characters (for example, in Figure 9 we can see the part of the grammar responsible of accepting IDENTIFIER tokens). After defining the tokens that our language can read, we can start defining the grammar's rules using non-terminal symbols (or just symbols from now on).

```

1  void main() {
2      fragColor = frontColor * texture(colorMap, vtxCoord);
3  }

```

Figure 10: The main function of a vertex shader. In this shader, the color of each fragment is calculated by using the frontColor value as well as the color stored in a 2D texture.

<sup>5</sup> ANTLR4: <https://www.antlr.org/>

In an ANTLR4 parser, each non-terminal symbol has its respective context, a class that holds the information of the node and all its possible children. For example, as we can see in Figure 11, the sub-tree with the node `function_call` at its root has 6 children: a `function_name`, the opening parentheses token, a list of expressions separated by the `,` token and finally a closing parentheses token. This means that the context object of this node contains a list with pointers to all of the 6 child nodes. Naturally, the grammar rule for function calls allows for any number of parameters to be passed to a function as long as they are separated by commas, so a `function_call` node can have an undefined amount of children.

To make AST evaluation easier, ANTLR provides not only the ordered list of children of a node, but also a different list for each type of child that the grammar allows this node to have. This means that a `function_call` node contains a list of `expression` nodes with the parameters passed to the function, and they can be evaluated without having to deal with parentheses or comma tokens.

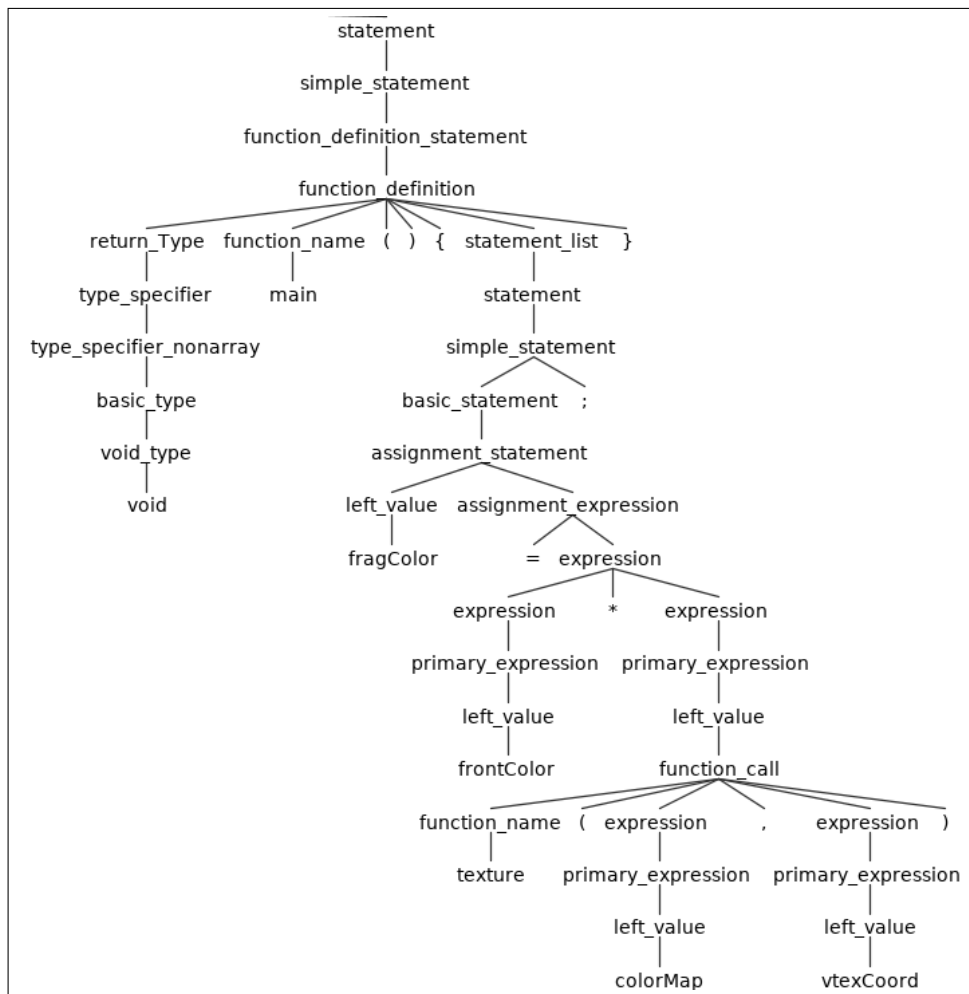


Figure 11: Tree created by the GLSL code in Figure 10

## 9 Listeners and Visitors

In Figure 11 we can see how even a simple main function like the one in Figure 10 produces a rather large tree, with many intermediate nodes for each terminal token. The intermediate nodes represent the path of non-terminal symbols taken through the grammar to parse the input tokens, which are represented in the leaves of the tree. In order to traverse trees, ANTLR provides two mechanisms, the listener and the visitor.

### 9.1 ANTLR listener class

The listener is an object that uses a built-in parse-tree walker to trigger specific functions for each node of the tree. Given a grammar, ANTLR generates a parse-tree listener interface with a function for the entry and exit points of each non-terminal symbol of the grammar. A parse-tree walker traverses the AST in pre-order calling the listener's entry function upon visiting each node, and the exit function when all the node's children had been visited. This makes it easy to implement some basic features since we can override only the functions for the nodes we are interested in and extract the information we want without having to search for them manually from the root of the tree.

```
1 class myListener(Listener):
2
3 def enterLeft_value(self, ctx:Parser.Left_valueContext):
4     if(ctx.IDENTIFIER() != None):
5         print(ctx.IDENTIFIER().getText())
6     pass
```

Figure 12: A listener that prints the identifier text of every `left_value` symbol in the tree.

Figure 12 shows an example of a very simple listener class. This class overrides the entry function of `left_value` symbols, so that it will print the text of all `left_value` nodes in the tree that have an identifier as a child. The result of executing this listener on the AST in Figure 11 would be the following:

```
1     fragColor
2     frontColor
3     colorMap
4     vtxCoord
```

## 9.2 ANTLR visitor class

Since listeners are triggered by an automatic tree walker they cannot include a return statement. More complex processes require an explicit control of the AST traversal. To this end, ANTLR provides the visitor object and generates an interface with the default implementations of each method. Visitors, unlike listeners, do not need a parse tree walker, since they let us visit children nodes explicitly. This makes it possible to implement complex features that need information on the context of the whole program, not only one rule.

The main difference between listeners and visitors is that visitors have a return statement, and must explicitly visit their children. The default implementation of the visitor function is for a node to visit all its children and return an aggregation of their results. This behaviour is implemented in the `visitChildren` function that can be seen in Figure 13. By default, the function `defaultResult` returns `None`, and `aggregateResult` returns the `childResult`. This behaviour is not always the desired one, so both functions can be overridden to customize the way results are managed. The `visitChildren` function itself can also be overridden to implement more complex features.

```
1 def visitChildren(self, node):
2     result = self.defaultResult()
3     n = node.getChildCount()
4     for i in range(n):
5         c = node.getChild(i)
6         childResult = c.accept(self) #visits node c
7         result = self.aggregateResult(result, childResult)
8     return result
```

Figure 13: A simplified version of the default `visitChildren` function, that visits the children of a node returning the aggregation of their results.

```
1 import glcheck
2 #we provide the path to the source code file
3 vs = glcheck("path/to/shader.vert")
4
5 #more than one file can be added to the same object
6 vsfs = glcheck(["path/to/shader.vert", "path/to/shader.frag"])
```

Figure 14: Example of the creation of two `glcheck` objects. The first one checks a single vertex shader file, while the second one checks the code of two different shaders.

## 10 Python API: `glcheck`

As mentioned previously, this API needs to provide a broad range of functions that extract information from GLSL programs. These functions have been specified by the Graphics subject manager teacher in order to serve the purpose of identifying the most common mistakes made by the subject's students.

All of the functions implemented are available in an open-source python library called "`glcheck`", that can be found in the project's public repository.<sup>6</sup>

In the following sections we will explain the behaviour and implementation details of each one of the library's functions.<sup>7</sup>

### Initialization

In order to analyze a shader, a `glcheck` object needs to be created and initialized with the code in question. Checker objects can contain an indefinite amount of shader sources and all of them will be checked through the API functions and their results will be merged.

In the example from Figure 14 we see how the checker objects `vs` and `vsfs` are created.

```

1  vec3 vpos(){
2      return vertex + normal;
3  }
4
5  void main() {
6      vec3 V = vpos();
7      vec3 N = normalize(normalMatrix * normal);
8      frontColor = vec4(vec3(N.z),1);
9      gl_Position = modelViewProjectionMatrix * vec4(V, 1);
10 }

```

Figure 15: Example of a vertex shader code with all calls highlighted in red.

```

1  vs.calls("normalize") #--> [7:12]

```

Figure 16: Usage of the `calls` function on the `vs` object created in Figure 14. The function returns a list of the positions in the code where the calls to `normalize` take place, in the case of the vertex code from Figure 15, the result would be a single position: line 7, column 12.

## 10.1 Syntactic analysis

### Calls

Identifies all calls to a specified function. In Figure 15 we can see an example of a vertex shader code in which every function call has been highlighted.

We consider as function calls:

- Calls to user-defined functions as well as standard GLSL functions.
- Calls to constructors.
- Calls to binary operators.

As seen in Figure 16, the `calls` function is provided with the name of the function we want to find the calls to, and returns a list of all the positions in the code where the calls take place.

In order to implement this function, we first need to identify every single rule in the grammar that could produce one of the function calls listed above.

<sup>6</sup> Project repository: <https://gitrepos.virvig.eu/docencia/glcheck>

<sup>7</sup> In the Graphics course programming environment, shaders are passed an array of arguments, including transformation matrices, the position of the vertex, its color and many other properties. In the following shader codes, recurrent or irrelevant declarations will be omitted for simplicity. Check [Gra16](pages 23–26) for a full specification of the arguments passed to the shaders in the Graphics course environment.



```

1  vec3 vpos(){
2      return vertex + normal;
3  }
4
5  void main() {
6      vec3 V = vpos();
7      vec3 N = {normalize(normalMatrix * normal);
8      frontColor = vec4(vec3(N.z),1);
9      gl_Position = modelViewProjectionMatrix * vec4(V, 1);
10 }

```

Figure 17: Example of a vertex shader code where the first parameter of every call to constructor `vec4` has been highlighted in red.

```

1  vs.param("vec4") #default parameter index is 1
2  vs.param("vec4", 1) #gives the same result as the above line
3  #both functions return --> ['vec3(N.z)', 'V']

```

Figure 18: Usage of the `param` function. In this example the function returns the strings associated with the first parameter of each call to the function `vec4` in the vertex shader code from Figure 17.

Using a listener we will override the entry function of each one of the appropriate non-terminal symbols and compare its identifier or operator tokens to the function name we want to locate.

## Parameter names

Identifies the parameters passed to function calls, the definition of a function call being the one specified in the previous section. This feature takes a function's name and an integer parameter index and returns the expression associated with that parameter for each call to the function. For example, given the GLSL code from Figure 17, in order to retrieve the name of the first parameter in all the calls to the function "`vec4`" we would apply the code from Figure 18.

This feature has slightly different implementations depending on the type of function we're analysing. In regular function call and constructor nodes, the symbol associated with the callee's name is a sibling of the expressions passed as parameters. In a binary operator node, the operator's name is the parent of the two expressions passed to it. In both cases this problem is solved using a listener that filters the calls by their name and searches further into the tree in order to find the corresponding parameter in that node's list of expressions.

```

1 layout (location = 0) in vec3 vertex;
2 layout (location = 1) in vec3 normal;
3 layout (location = 2) in vec3 color;
4 out vec4 frontColor;
5 uniform mat4 modelViewProjectionMatrix;
6 uniform mat3 normalMatrix;
7
8 void main() {
9     V=(modelViewMatrix * vec4(vertex, 1)).xyz;
10    vec3 N = normalize(normalMatrix*normal);
11    frontColor = vec4(color, 1) * N.z;
12    gl_Position = modelViewProjectionMatrix * vec4(vertex, 1);
13 }

```

Figure 19: Example of part from a vertex shader code where the expressions on which the "z" field selector is applied have been highlighted.

```

1 vs.fieldSelectors("z") #--> ['(modelViewMatrix*vec4(vertex, 1))', 'N']

```

Figure 20: Usage of the `fieldSelectors` function on the vertex shader code from Figure 19. In this example the function returns the strings associated with all of the expressions where a field selector z has been applied.

## Field selector names

Very similar to "**Parameter names**" but instead of identifying the parameters passed to function calls, it identifies the name of all expressions where a certain field selector has been applied. The code listed in Figure 19 shows an example of this functionality, highlighting the expressions where the **z** field selector is applied. An example of the usage of this function is provided in Figure 20.

## Declarations

Identifies every declaration of a specified variable. We consider as declarations:

- Variable declaration inside the main and any function's scope.
- Variable declaration as a parameter of a function.

Figure 21 shows a GLSL code with all of the declarations of variable **N** highlighted in red. The code listed in Figure 22 shows how this function can be used to find out all the positions in the code where the variable **N** has been declared, as well as its type.

In order to implement this functionality, we use a listener that overrides the entry points of two rules in the grammar: the simple declaration rule and

```

1  vec4 foo(float N){
2      return vec4(color, 1.0)*N;
3  }
4
5  void main() {
6      vec3 N = normalize(normalMatrix*normal);
7      frontColor=foo(N.z);
8      gl_Position=modelViewProjectionMatrix*vec4(vertex, 1);
9  }

```

Figure 21: A fragment of a vertex shader code with all the declarations of **N** highlighted in red.

```

1  vs.declarations("N") #-> [('float', 1:8), ('vec3', 6:12)]

```

Figure 22: Usage of the `declarations` function. In this example the function returns a tuple with the type and position for each declaration of variable **N** in the vertex shader code from Figure 21.

the function member declaration rule. In the contexts of these rules we can find both the type specifier symbol sub-tree and the identifier of the declared variable. If the identifier of a declaration matches the name specified, we look further into it in order to determine its type.

## Names and types of **in/out** variables

Returns the names or types of all variables declared using either **in** or **out** type qualifiers. The code listed in Figure 24 shows an example of the usage of each one of the functions and their return values when applied on the fragment shader code from Figure 23.

In order to implement all these functions, the declaration listener specified in the previous section has been used as a base class. The API uses the same listener to implement all four functions, by storing an attribute to keep track of which function has been called. This listener's entry function to simple declaration nodes has been overridden in order to filter declarations by their type qualifier. Since a declaration can have any number of type qualifiers, all of them must be iterated. When a qualifier is found, its name is checked to

```

1  in vec4 frontColor;
2  in vec2 vtexCoord;
3  out vec4 fragColor;

```

Figure 23: Example of part from a fragment shader code.

```

1 vs.inNames() #--> [('frontColor', 1:0), ('vtexCoord', 2:0)]
2 vs.outNames() #--> [('fragColor', 3:0)]
3 vs.inTypes() #--> [('vec4', 1:0), ('vec2', 2:0)]
4 vs.outTypes() #--> [('vec4', 3:0)]

```

Figure 24: Usage of the `inNames`, `outNames`, `inTypes` and `outTypes` functions on the fragment shader code from Figure 23.

```

1 void main() {
2     vec3 N;
3     N = normalize(normalMatrix*normal);
4     if(true){
5         N.z = 1;
6     }
7     while(1){
8         frontColor=vec4(color, 1.0)*N.z;
9         for(int i = 0; i < 1< ++i) continue;
10        if(1 == 1) break;
11    }
12    gl_Position = modelViewProjectionMatrix*vec4(vertex, 1);
13 }

```

Figure 25: A fragment of a vertex shader code overloaded with sentences for the purpose of this demonstration.

make sure it matches the desired qualifier type, either 'in' or 'out', and if it matches, the position of the declared variable is stored. Depending on the listener's attribute that specifies which function was called externally, either the variable's type or its name is stored along with the position. The type is retrieved using the parent class, the declaration listener, which has already implemented this feature. If the initial query asked for the name of the variable, since a single declaration can be used to specify more than one variable name (declarator), the array of declarators is iterated and all the identifiers are stored along with the position calculated previously.

## Sentences

Identifies every use of a given keyword (`while`, `for`, `if`, `switch`, `case`, `do`, `default`, `continue`, `return`, `break`) and returns its line and column in the code. Given the GLSL code from Figure 25, some possible calls to `sentences` and the value they return have been listed in Figure 26.

In order to implement this function, we need to find all the grammar rules that generate the sentences specified above. Once all the rules are located, a listener is implemented, overriding the entry function to each one of the

```

1 vs.sentences("for") #--> [9:8]
2 vs.sentences("if") #--> [4:4, 10:8]
3 vs.sentences("break") #--> [10:19]
4 vs.sentences("while") #--> [7:4]
5 vs.sentences("continue") #--> [9:35]
6 vs.sentences("switch") #--> []

```

Figure 26: Usage of the `sentences` function applied with different parameters to the code listed in Figure 25 and the list of positions returned by each one of them.

```

1 void main() {
2     vec3 N;
3     N = normalize(normalMatrix*normal);
4     if(true){
5         N.z = 1;
6     }
7     frontColor=vec4(color, 1.0)*N.z;
8     gl_Position=modelViewProjectionMatrix*vec4(vertex, 1);
9 }

```

Figure 27: A fragment of a vertex shader code. All the assignments to variable `N` have been highlighted.

rules to locate its token and compare it to the given keyword. If the keyword specified is the same as the token encountered, the position of that token is stored in the result list.

## Assignments

Identifies every assignment to a given variable. Assignments through a struct selection operator (`var.a`) are also considered. Figure 27 shows the assignments to the variable `N` highlighted in red while Figure 28 demonstrates the use of the function.

Assignments of a variable are found using a listener that checks nodes of assignment expressions for the appearance of the specified variable name in their identifier.

```

1 vs.assignments("N") #--> [3:4, 5:8]

```

Figure 28: Finding all `assignments` to variable `N` in the vertex shader code from Figure 27.

```

1  vec4 foo(vec3 N){
2      return vec4(color, 1.0)*N.z;
3  }
4
5  void main() {
6      vec3 N = normalize(normalMatrix*normal);
7      frontColor=foo(N);
8      gl_Position=modelViewProjectionMatrix*vec4(vertex, 1);
9  }

```

Figure 29: A fragment of a vertex shader code with all the uses of **N** highlighted in red.

```

1  vs.uses("N") #--> [2:28, 7:19]

```

Figure 30: Finding all uses of variable **N** in the vertex shader code from Figure 29.

## Uses

Identifies every expression in which the specified variable is used. Assignments and declarations are not considered uses. Figure 29 highlights the uses of variable **N** in a vertex shader program and Figure 30 demonstrates the use of the function.

In order to obtain only the uses of a certain variable, a listener collects all the positions where an identifier with the specified text appears, and subtracts from that list all of the assignments and declarations of that variable.

## Descendants

Checks whether a statement can be found inside a sentence of a certain type. When the sentence type is **for**, **if**, or **while**, a parameter specifies whether to look inside the **body** or the **condition** of the sentence.

Given the code from Figure 31, an `isDescendantOf(discard, if, body)` query would return `true`.

```

1  void main() {
2      if (x>time) discard;
3      fragColor=vec4(0, 0, 1, 1);
4  }

```

Figure 31: A fragment of a vertex shader code.

```

1  vec3 vpos(){
2      return vertex + normal;
3  }
4
5  void main() {
6      vec3 V = vpos();
7      vec3 N = {normalize(normalMatrix * normal);
8      frontColor = vec4(vec3(N.z),1);
9      gl_Position = modelViewProjectionMatrix * vec4(V, 1);
10 }

```

Figure 32: Example of a vertex shader code where the first parameter of every call to constructor `vec4` has been highlighted in red.

```

1  vs.paramTypes("vec4") #default parameter index is 1
2  #the function returns --> ["vec3", "vec3"]

```

Figure 33: Usage of the `paramTypes` function. In this example the function returns the types associated with the first parameter of each call to the function `vec4` in the vertex shader code from Figure 32.

## 10.2 Semantic analysis

### Parameter types

Similar to the **Parameter names** feature, but instead of returning the expression of a parameter, it returns its type. Given the GLSL code from Figure 32, in order to know the type of the first parameter passed to all of the calls to the function `"vec4"`, we would execute the code listed in Figure 33.

Unlike expression evaluation, which requires the execution of the program in order to calculate the expression's result, types can be inferred without executing the program's instructions in strongly statically typed languages like GLSL. The first step towards type inference is pre-computing an array of data-structures.<sup>8</sup>

- A dictionary of all the **functions** defined in the code and their return type.
- A dictionary of all the **structures** defined in the code and the types of their attributes.

<sup>8</sup> The data-structures should also include all of the built-in GLSL variables and functions specified in [Ros10] (sections 7 and 8), but this step has been omitted in this version of the project due to the increase in the workload that would mean to manually create a database with such an amount of functions.

```

1 layout (location = 0) in vec3 vertex;
2 layout (location = 1) in vec3 normal;
3 layout (location = 2) in vec3 color;
4 out vec4 frontColor;
5 uniform mat4 modelViewProjectionMatrix;
6 uniform mat3 normalMatrix;
7
8 void main() {
9     V=(modelViewMatrix * vec4(vertex, 1)).xyz;
10    vec3 N = normalize(normalMatrix*normal);
11    frontColor = vec4(color, 1) * N.z;
12    gl_Position = modelViewProjectionMatrix * vec4(vertex, 1);
13 }

```

Figure 34: Example of part from a vertex shader code where the expressions on which the "z" field selector is applied have been highlighted.

- A dictionary of all the **variables** defined in the code and their types.

Taking into account all expression types exposed in subsection 7.1, a recursive algorithm can infer the type of any expression by applying the following rules:

- Constant expressions and constructors have an inherent type.
- A function's call return type is pre-computed in the function dictionary.
- Expressions consisting of just an identifier refer to a variable therefore their type is determined by the variable dictionary.
- Identifiers with an array selector operator have a type that will match the type of data stored by the array, which should also be found in the variable dictionary.
- The type of identifiers with a struct selector operator is determined by the struct dictionary.
- Expressions with an arithmetic or bitwise operator (including arithmetic assignments) are determined by the type of the operator's sub-expression. Binary bitwise operators should only evaluate the left-hand-side expression, whereas binary arithmetic operators can evaluate either-one.
- The output type of logic operator expressions is always Boolean.



```
1 vs.fieldSelectorsTypes("z") #-> ['vec4', 'vec3']
```

Figure 35: Usage of the `fieldSelectorsTypes` function on the vertex shader code from Figure 19. In this example the function returns the types of the expressions where a field selector `z` has been applied.

```
1 uniform mat4 modelViewProjectionMatrix;
2 uniform mat4 modelViewMatrix;
3 uniform mat3 normalMatrix;
4
5 void main() {
6     vec3 N;
7     vec3 P; //P is undefined
8     N = normalize(normalMatrix*normal); //P is undefined
9     P = (modelViewMatrix*vec4(vertex, 1)).xyz; //P is eye
10    if(a) //P is eye
11        P=(modelViewMatrix*vec4(vertex, 1)).xyz; //P is wrong
12    else
13        P=(modelViewProjectionMatrix*vec4(vertex, 1)).xyz; //P is window
14    bool a = true; //P is wrong or window
15    while(a){ //P is wrong or window
16        frontColor = vec4(color, 1.0)*N.z; //P is wrong or window
17        a = false; //P is wrong or window
18    }
19    gl_Position=modelViewProjectionMatrix*vec4(vertex, 1); //P is wrong
20 } //or window
```

Figure 36: A fragment of a vertex shader code where, for each instruction where `P` has a value, its possible coordinate spaces are specified.

## Field selector types

Very similar to "**Parameter types**" but instead of calculating the type of expressions passed as a parameter, calculates the types of expressions where the specified field selector has been applied.

Given the code from Figure 34, in order to find out the types of expressions where the "`z`" field selector has been applied, we would need to run the code from Figure 35.

## Space

Identifies all possible coordinate spaces for a specified variable of type `vec` or `mat` through the scope of the `main` function. In Figure 36 we are shown the coordinate spaces of `P` for each statement in the code where `P` has an

```

1 vs.space('P')
2 #returns --> [['undefined'], ['undefined'], ['eye'], ['eye'],
3   ['wrong'], ['window'], ['wrong', 'window'], ['wrong', 'window'],
4   ['wrong', 'window'], ['wrong', 'window'], ['wrong', 'window']]

```

Figure 37: Usage of the `space` function. In this example the function returns a list of all the possible coordinate spaces that the variable `P` could be in at each statement in the code from Figure 36.

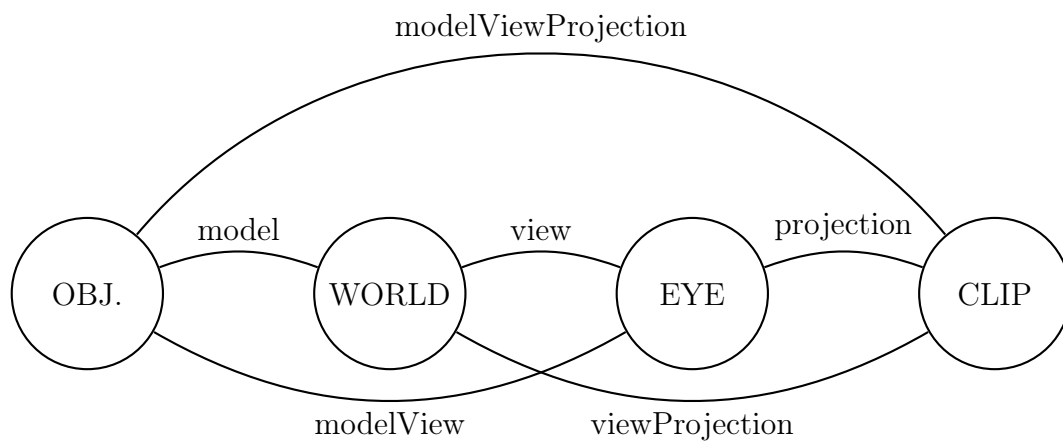


Figure 38: Coordinate space transformation scheme. Nodes represent coordinate spaces and edges represent the transformation matrices.

assigned value. Notice that `P` keeps its value even in the statements where it does not appear, due to the fact that the state of coordinate spaces shown in each statement is the one stored in the internal variable dictionary, where `P` is present at all times after its declaration, not the one corresponding to just the current statement.

In order to transform a vector from one coordinate space to another, a transform matrix is used. Figure 38 shows the main coordinate spaces and the matrices used to transform vectors from one to another. For example, in order to transform a vector `N` from object space to world space, the following operation should be performed: `N = modelMatrix*N`. A major source of errors in shader programs comes from inadvertently operating on two vectors that are not in the same coordinate space, therefore knowing the coordinate space of each vector at all times can be very useful to detect such errors.

This feature, of course, has major limitations, but can also yield useful results in the average case. We cannot determine the precise output of a program without executing it, but we can calculate the set of possible paths it can take for each statement. The obvious problem of unfolding code are loop statements, and the fact that it can be impossible to predict the number of iterations they will go through, but given the precondition that the program does eventually stop, and the knowledge that many shader programs do not require loop statements, we can consider that any loop statement will execute either 0 or 1 iterations. By treating loops as selection statements, we can keep track of each variable declaration, assignment and its coordinate system along each one of the possible paths that the program can take. In the general case, a variable's coordinate system should not depend on external factors, which makes it easy to detect errors just by observing the presence of different possible coordinate systems for a variable in a given statement.

In order to calculate all these possible coordinate spaces, we need specific data-structures to keep track of all the information. The approach taken in this project is to create a structure that represents a *program state*. This structure has a dictionary of variable names and their type and coordinate space. This dictionary will only keep track of "`vec`" and "`mat`" variables, as well as the transformation matrices, since they are the only ones that play a role in coordinate space calculations.

Every statement in a program has its own **program state**. Since the execution of a statement changes the value of variables in a program, the current **program state** also changes when a new statement is executed. In order to be able to keep track of the coordinate space of each variable through every step of the program, we have to keep a stack of **program state** structures.

If our program were to follow a single determined path, this stack would be enough, but flow control sentences may or may not change the statements that a program will execute depending on the circumstances. This means that we need to keep track not only of one **program state** stack, but a list of these

stacks, where each element in the list represents a **path** through a program's statements. This **path** list is referred to as the **program context**.

Using this setup we design a recursive algorithm to analyse and keep track of coordinate spaces along the program:

1. Before starting to analyse the program, prepare the first program state adding all the built-in variables and transform matrices passed by the pipeline to the shader.<sup>9</sup>
2. Using a parse-tree visitor, visit the node corresponding to the first statement list in the main function.<sup>10</sup> For each statement in that list:
  - (a) Save the original **context** of the program before the execution of this statement.
  - (b) Create an empty result **context**.
  - (c) For each **path** in the original context, analyse the current statement and append all the resulting **paths** after that statement to the result **context**.
3. When visiting a new statement, duplicate the last **program state** in the context and add it to the top.
4. Update the current **program state** depending on the new statement to analyse.
  - (a) Declaration: if the new variable has a type of **vec** or **mat**, analyse the expression in order to calculate its coordinate space and add it to the current **program state**.<sup>11</sup>
  - (b) Assignment: calculate the coordinate space for the new value given to the variable and update its coordinate space in the current **program state**.
  - (c) Flow control statement: Save a copy of the current **path**. After analysing the statements in the body of the statement, append the original path to the current **context**, effectively splitting the original path into two, one where the program did not enter the statement and one where it did. For **if** statements with an **else** clause, the original **path** has to go through the statements in the **else** before being re-added to the **context**.

---

<sup>9</sup> The *Configuration* module allows the user to set the names and coordinate spaces of all the variables passed to the shader.

<sup>10</sup> In this first version of the implementation, only the scope of the main function is analysed.

<sup>11</sup> When variables are declared from constants in the code, there is no way to infer the coordinate space that variable could be in. The *Configuration* module allows to specify a default coordinate space for constants.

- (d) Function calls: Ideally, we would have pre-computed a dictionary of function definition sub-trees. When a function call is encountered, the function's parameters are added to the current `program state` and the statements of that function are visited.<sup>12</sup>

When encountering declaration and assignment sentences, we rely on a different visitor to calculate the coordinate space of an expression. This visitor is called on the expression node, and passed the current `program state` with the coordinate spaces of all known variables. This visitor infers the coordinate system of an expression in a similar way as we inferred the type of an expression in a previous section. All operations it needs to take into account are:

- Multiplication operations where the left-hand side is a transform matrix: the "from" and "to" spaces of the transform matrix are checked and, if correct, the coordinate space is updated accordingly.
- Division operations where the right-hand side is the `w` component of a variable in `clip` space, in which case the variable will be transformed to the `NDC` space.
- Any other operation either doesn't change the coordinate space or is incoherent and changes it to `wrong`.

Therefore, for a given expression, this visitor will calculate its coordinate space to be one of the following: `object`, `world`, `eye`, `clip`, `NDC`, `wrong` or `unknown`.<sup>13</sup>

In order to present an ordered result and be able to map each statement in the program to all the `program state` objects associated with it, we create an identifier for this objects, which is unique for each statement in the code, so that all `program states` that refer to the same statement will have the same identifier, and sort the output accordingly.

---

<sup>12</sup> The current implementation does not support for function call analysis yet.

<sup>13</sup> When a `vec` or `mat` variable has been declared but not initialized yet, it is featured in the program context as having an `unknown` coordinate space.

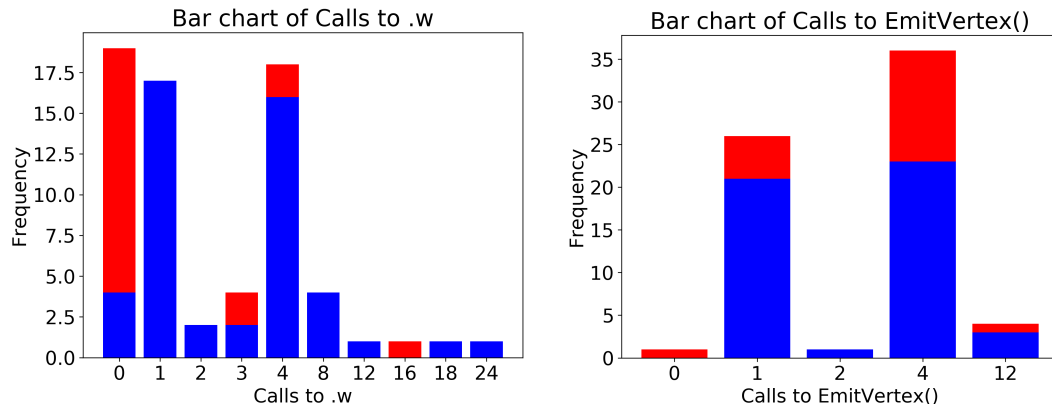


Figure 39: Stacked bar charts for two rubrics of the Quads assignment. Submissions with correct output are shown in blue, and incorrect ones in red.

## 11 Results

The graphics course teachers did a preliminary test with some recent assignments from a lab exam with 86 participants.

### 11.1 Quads

In one of the assignments, students had to write a geometry shader that, for each input triangle, outputs four triangles, one for each quadrant of the viewport. This is a first step for a geometry shader that shows top, left, front, perspective views of the scene. Students were advised to use the NDC coordinate space for translating the copies, as  $(x,y)$  coordinates of NDC copies just differ by  $\pm 0.5$ .

Analysing the submitted codes by applying only two rubrics already reveals some interesting features. Figure 39 shows the bar charts of a couple of general features. Notice how some outliers are clearly visible. The conversion from clip to NDC requires a perspective division (i.e. dividing by the homogeneous coordinate  $w$ ). This division can be performed only once. In this case, outliers in the number of `.w` accessors corresponded to incorrect submissions or to submissions performing the division multiple times, e.g. once for each quadrant (which should be penalized).

The bar chart on the number of `EmitVertex()` also shows clear outliers. The natural solution to this problem requires either one or four `EmitVertex()` calls, depending on the number of loops used, therefore outliers corresponded to wrong code or poorly-factorized code.

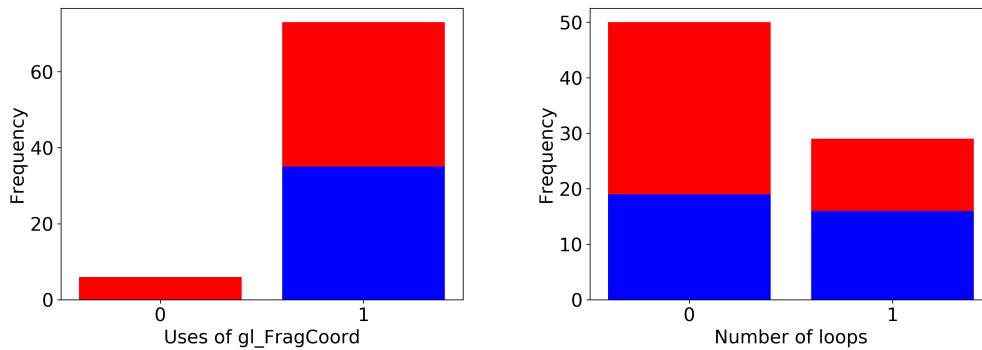


Figure 40: Stacked bar charts for two rubrics of the Cartoon Shading assignment. Submissions with correct output are shown in blue, and incorrect ones in red.

## 11.2 Dithered cartoon shading

For this assignment students had to write a cartoon-like fragment shader that involved the use of a noise texture and color dithering. Figure 40 shows the bar charts of two sample rubrics. The first chart shows whether students used `gl_FragCoord` or not. The solution required the fragment coordinates to access the noise texture, so submissions not using it did not pass the test (shown in red).

The second chart shows the number of loops in the fragment shader. The assignment required to find the closest quantized color to a noise-perturbed color. Most students realized that this could be computed by just rounding, whereas others used an inefficient loop to search for the closest quantized color. This procedure does not affect the operational correctness of the code, so it cannot be detected just by looking at the test set results. This kind of rubric greatly simplifies detecting and providing feedback to those submissions that need to be penalized for inappropriate code, no matter the output correctness.

## 12 Conclusions and future work

### 12.1 Conclusions

Although the final version of the tool hasn't been fully tested by the teachers in an exam-correction environment yet, early experiments show that it has great potential and has deemed useful to the teachers that tried it. Even though not every one of the initial features we thought about has made it into this initial version of the tool, the work done in this project makes it a lot easier to add any desired features in the future.

In relation to the features implemented, we have observed that even if analyzing a program without executing it is a difficult task, extracting basic information from it can be fairly simple once the program is adequately parsed. For the parser to be useful, a grammar has not only to be correct and accepting all and only the right programs, but also needs to have a logical structure that will allow for simple processing.

It has been observed that even if extracting more complex features from the code is hard, it can be achieved by applying the convenient constraints to the problem we're facing by analyzing the context in which the problem has to be solved, and that small and simple constraints that have little to no difference to the end-user may enable powerful features that couldn't be implemented at all without them.

We can conclude that all the initial objectives of the project have been achieved, and the tool developed is useful for aiding teachers and students into a faster and easier understanding of computer graphics programs.

### 12.2 Future work

Some of the tasks that could be performed in the future in order to improve the implemented tool are the following:

- A main part that has been left out of the project is to integrate a C++ parser that implements some of the features of the tool. This would require a C++ ANTLR4 grammar similar to the one used for the GLSL analysis.
- Integrating this analysis tool inside the Viewer program could also be useful to students that want to check their code quality or spotting their mistakes.
- Developing brainstormed features that were left out of the original specification such as computing the possible stack of calls to functions of a shader.



- Optimize the code and store the results of an analysis for future queries.
- Lifting some of the constraints of the code such as analyzing the coordinate space of variables in scopes different than the main function.
- Integrating the GLSL built-in variables and functions into the code analysis tool.

## References

- [Age18] Agencia Tributaria. Tabla de coeficientes de amortización lineal. [https://www.agenciatributaria.es/AEAT.internet/Inicio/\\_Segmentos\\_/Empresas\\_y\\_profesionales/Empresas/Impuesto\\_sobre\\_Sociedades/Periodos\\_impositivos\\_a\\_partir\\_de\\_1\\_1\\_2015/Base\\_imponible/Amortizacion/Tabla\\_de\\_coeficientes\\_de\\_amortizacion\\_lineal\\_.shtml](https://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml), 2018. Accessed: 2018-10-10.
- [Agi18] Agile Alliance. What is agile software development? <https://www.agilealliance.org/agile101/>, 2018. Accessed 2018-9-23.
- [And18] Anders Møller, Michael I. Schwartzbach. Static program analysis. <https://cs.au.dk/~amoeller/spa/spa.pdf>, 2018.
- [Fay06] Connors, Dan Fay, Dan, Sazegari, Ali. A detailed study of the numerical accuracy of gpu-implemented math functions. <https://pdfs.semanticscholar.org/7f65/72f3b453ef06d16f37932ff74f8c8e4de725.pdf>, 2006.
- [Gen] Generalitat de Catalunya. Factor de emisión asociado a la energía eléctrica. [http://canviclimatic.gencat.cat/es/reduex\\_emissions/com-calcular-emissions-de-geh/factors\\_demissio\\_associats\\_a\\_lenergia/](http://canviclimatic.gencat.cat/es/reduex_emissions/com-calcular-emissions-de-geh/factors_demissio_associats_a_lenergia/). Accessed: 2018-10-10.
- [Gra16] Graphics Course Professors. Vertex shaders i fragment shaders. entorn per desenvolupar shaders (viewer). <http://www.lsi.upc.edu/~virtual/G/2.%20Laboratori/Part%201%20-%20Shaders/Sessio%201/Sessio%201.pdf>, 2016. Accessed 2018-10-14.
- [Hof99] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. 1999.
- [Lon15] Yue Long. antlr4\_convert. [https://github.com/labud/antlr4\\_convert/blob/master/src/GLSL/GLSL.g4](https://github.com/labud/antlr4_convert/blob/master/src/GLSL/GLSL.g4), 2015. Accessed: 2018.
- [Pat12] Pathfinder Solutions. *Effective TDD for Complex Embedded Systems Whitepaper*. 2012. Accessed 2018-9-23.
- [Pay18a] Payscale. Programmer analyst salary. [https://www.payscale.com/research/ES/Job=Programmer\\_Analyst/Salary](https://www.payscale.com/research/ES/Job=Programmer_Analyst/Salary), 2018. Accessed: 2018-10-10.

- [Pay18b] Payscale. Project manager, information technology (it) salary. [https://www.payscale.com/research/ES/Job=Project\\_Manager%2c\\_Information\\_Technology\\_\(IT\)/Salary](https://www.payscale.com/research/ES/Job=Project_Manager%2c_Information_Technology_(IT)/Salary), 2018. Accessed: 2018-10-10.
- [Pay18c] Payscale. Software engineer / developer / programmer salary. [https://www.payscale.com/research/ES/Job=Software\\_Engineer\\_%2f\\_Developer\\_%2f\\_Programmer/Salary](https://www.payscale.com/research/ES/Job=Software_Engineer_%2f_Developer_%2f_Programmer/Salary), 2018. Accessed: 2018-10-10.
- [Pop03] Mary Poppendieck; Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.
- [Ros10] Kessenich, John Baldwin, Dave Rost, Randi. The opengl® shading language. <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.3.30.pdf>, 2010. Accessed 2019.

## A Python API Documentation

### A.1 Module `glcheck`

Syntax	Description
<code>glcheck([s1, ..., sN])</code>	Constructor of a <code>glcheck</code> object. <code>s1</code> to <code>sN</code> are the paths to all the source codes to analyse at once.
<code>R(s, ...)</code>	Main function for rubric specification. A rubric with description <code>s</code> is created, evaluating the expression passed as the second parameter.
<code>assignments(s)</code>	Returns a list with the positions of every assignment to variable <code>s</code> . Assignments to <code>s</code> through a swizzle operator ( <code>s.x</code> ) are also considered.
<code>calls(s)</code>	Returns a list with positions of all calls to functions, constructors or operators matching the string <code>s</code> .
<code>declarations(s)</code>	Returns a list with the positions of every declaration of variable <code>s</code> . Declarations of parameters in function definitions are also considered.
<code>fieldSelectors(s)</code>	Returns a list with the names of all variables where a field selector <code>.s</code> has been applied.
<code>fieldSelectorsTypes(s)</code>	Returns a list with the types of all variables where a field selector <code>.s</code> has been applied.
<code>inNames()</code>	Returns a list with the names of all <code>in</code> variables.
<code>inTypes()</code>	Returns a list with the types of all <code>in</code> variables.
<code>isDescendantOf(a, b)</code> <code>isDescendantOf(a, b, c)</code>	Checks whether the sentence <code>a</code> can be found inside a sentence of type <code>b</code> . When <code>b</code> is <code>for</code> , <code>if</code> , or <code>while</code> , parameter <code>c</code> specifies whether to look inside the <code>body</code> or the <code>condition</code> of the sentence. The default value of <code>c</code> is <code>body</code> .
<code>outNames()</code>	Returns a list with the names of all <code>out</code> variables.
<code>outTypes()</code>	Returns a list with the types of all <code>out</code> variables.
<code>param(s)</code> <code>param(s, i)</code>	Identifies the expression associated with the <code>i</code> th parameter of every call to function, constructor or operator <code>s</code> . If <code>i</code> is not specified, the function identifies the first parameter.

Syntax	Description
<code>paramTypes(s)</code> <code>paramTypes(s, i)</code>	Identifies the type of the <code>i</code> th parameter of every call to function, constructor or operator <code>s</code> . If <code>i</code> is not specified, the function identifies the first parameter.
<code>sentences(s)</code>	Returns a list with the positions of every keyword <code>s</code> . The following keywords are considered: <code>while</code> , <code>for</code> , <code>if</code> , <code>switch</code> , <code>case</code> , <code>do</code> , <code>default</code> , <code>continue</code> , <code>return</code> , <code>break</code> .
<code>space(s)</code>	Returns a list with all possible coordinate spaces for variable <code>s</code> of type <code>vec</code> or <code>mat</code> through the scope of the <code>main</code> function.
<code>uses(s)</code>	Returns a list with the positions of every use of variable <code>s</code> . (Assignments and declarations are not considered uses).
<code>numFoo(...)</code>	Wrapper where <code>Foo</code> can be any of the above functions. Returns the length of the result given by <code>Foo(...)</code> .

## A.2 Configuration

Parameter name	Description
<code>Encoding</code>	Character encoding for the source code files to analyse. Default is <code>utf_8</code> .
<code>Constant_Coord_Space</code>	Default coordinate space for constants in the code. Can be <code>object</code> , <code>world</code> , <code>eye</code> , <code>clip</code> or <code>NDC</code> .
<code>Model_Matrix</code>	Name of the <code>object</code> → <code>world</code> transform matrix.
<code>View_Matrix</code>	Name of the <code>world</code> → <code>eye</code> transform matrix.
<code>Projection_Matrix</code>	Name of the <code>eye</code> → <code>clip</code> transform matrix.
<code>ModelView_Matrix</code>	Name of the <code>object</code> → <code>eye</code> transform matrix.

Parameter name	Description
<b>ModelViewProjection_Matrix</b>	Name of the object→clip transform matrix.
<b>Model_Inverse_Matrix</b>	Name of the world→object transform matrix.
<b>View_Inverse_Matrix</b>	Name of the eye→world transform matrix.
<b>Projection_Inverse_Matrix</b>	Name of the clip→eye transform matrix.
<b>ModelView_Inverse_Matrix</b>	Name of the eye→object transform matrix.
<b>ModelViewProjection_Inverse_Matrix</b>	Name of the clip→object transform matrix.
<b>Vertex</b>	Name of the variable holding the vertex position information.
<b>Normal</b>	Name of the variable holding the normal vector information.
<b>Normal_Matrix</b>	This one is pretty straightforward.
<b>BoundingBox_Min</b>	Name of the <code>vec3</code> variable representing the min position of the bounding box.
<b>BoundingBox_Max</b>	Name of the <code>vec3</code> variable representing the max position of the bounding box.
<b>Light_Position</b>	Name of the <code>vec3</code> variable representing the light position.
<b>Log_Level</b>	Level of logging information. Can have the following values: <code>CRITICAL</code> , <code>ERROR</code> , <code>WARNING</code> , <code>INFO</code> , <code>DEBUG</code> .
<b>Log_File</b>	File name to store logging information. Specify <code>NONE</code> to use console.

## B ANTLR4 GLSL Grammar

```
1 grammar GLSL;
2 prog:
3     preprocessor* statement_list;
4
5 preprocessor
6     : SHARP version_pre;
7
8 version_pre
9     : 'version' integer VERSION_PROFILE?;
10
11 VERSION_PROFILE
12     : 'core'
13     | 'compatibility'
14     | 'es';
15
16 type_qualifier
17     : (storage_qualifier | layout_qualifier
18     | precision_qualifier | interpolation_qualifier
19     | invariant_qualifier | precise_qualifier)+;
20
21 layout_qualifier: 'layout' LEFT_PAREN layout_qualifier_id
22     (COMMA layout_qualifier_id)* RIGHT_PAREN;
23
24 layout_qualifier_id: IDENTIFIER | IDENTIFIER ASSIGNMENT_OP
25     constant_expression | 'shared';
26
27 storage_qualifier
28     : 'const' | 'in' | 'out' | 'uniform'
29     | 'buffer' | 'shared';
30
31 precision_qualifier
32     : 'high_precision'
33     | 'medium_precision'
34     | 'low_precision';
35
36 interpolation_qualifier
37     : 'smooth'
38     | 'flat'
39     | 'noperspective';
40
41 invariant_qualifier: 'invariant';
42
43 precise_qualifier: 'precise';
44
45 integer: DECIMAL | OCTAL | HEX ;
```

```
46
47 float_num: FLOAT_NUM;
48
49 bool_num : 'true' | 'false';
50
51 type_specifier: type_specifier_nonarray array_specifier*;
52
53 type_specifier_nonarray
54     :   basic_type
55     |   IDENTIFIER
56     ;
57
58 array_specifier :   LEFT_BRACKET expression? RIGHT_BRACKET;
59 struct_specifier: DOT left_value_exp;
60
61 basic_type
62     :   void_type |   scala_type   |   vector_type
63     |   matrix_type |   opaque_type;
64
65 void_type : 'void';
66
67 scala_type: SCALA;
68
69 vector_type: VECTOR;
70
71 matrix_type: MATRIX;
72
73 opaque_type
74     :   float_opaque_type
75     |   int_opaque_type
76     |   u_int_opaque_type;
77
78 float_opaque_type: FLOAT_OPAQUE;
79
80 int_opaque_type: INT_OPAQUE;
81
82 u_int_opaque_type: U_INT_OPAQUE;
83
84 expression
85     :   primary_expression #primary
86     |   expression INCREMENT_OP #postIncrement
87     |   INCREMENT_OP expression #preIncrement
88     |   ADDSUB_OP expression #sign
89     |   UNARY_OP expression #unary
90     |   expression MULDIV_OP expression #muldiv
91     |   expression ADDSUB_OP expression #addsub
92     |   expression SHIFT_OP expression #shift
```



```
93 | expression COMPARE_OP expression #cmp
94 | expression EQUAL_OP expression #eq
95 | expression BITWISE_OP expression #bitwise
96 | expression LOGIC_OP expression #logic
97 | expression QUESTION expression COLON expression #ternary
98 ;
99
100
101 constant_exp: constant_expression;
102
103 basic_type_exp:
104     basic_type LEFT_PAREN (expression (COMMA expression)*)? RIGHT_PAREN;
105
106 type_spec_exp: LEFT_PAREN type_specifier RIGHT_PAREN expression;
107
108 left_value_exp: left_value array_struct_selection?;
109
110
111 primary_expression
112     : constant_exp
113     | basic_type_exp
114     | type_spec_exp
115     | left_value_exp
116     ;
117
118
119 constant_expression
120     : integer
121     | float_num
122     | bool_num
123     ;
124
125 left_value
126     : function_call
127     | LEFT_PAREN expression RIGHT_PAREN
128     | IDENTIFIER
129     ;
130
131 array_struct_selection: (array_specifier | struct_specifier)+;
132
133 assignment_expression: ASSIGNMENT_OP expression;
134
135 arithmetic_assignment_expression: ARITHMETIC_ASSIGNMENT_OP expression;
136
137 function_definition
138     : return_Type function_name LEFT_PAREN
139     (func_decl_member (COMMA func_decl_member)* )?
```

```
140     RIGHT_PAREN LEFT_BRACE
141     statement_list RIGHT_BRACE;
142
143 function_declaration:
144     return_Type function_name LEFT_PAREN
145     (func_decl_member (COMMA func_decl_member)* )? RIGHT_PAREN;
146
147 function_call:
148     function_name LEFT_PAREN (expression (COMMA expression)*)? RIGHT_PAREN;
149
150 return_Type: type_specifier;
151
152 function_name: IDENTIFIER;
153
154 func_decl_member: type_specifier IDENTIFIER;
155
156 statement_list: statement*;
157
158 statement : simple_statement | compoud_statement ;
159
160 simple_statement
161     : function_definition_statement
162     | basic_statement SEMICOLON | selection_statement
163     | switch_statement | case_label
164     | iteration_statement | jump_statement;
165
166 compoud_statement: LEFT_BRACE statement_list RIGHT_BRACE;
167
168 basic_statement
169     : declaration_statement
170     | assignment_statement
171     | expression_statement;
172
173 declaration_statement
174     : struct_declaration
175     | simple_declaration
176     | function_declaration;
177
178 simple_declaration
179     : (type_qualifier? type_specifier simple_declarator
180       (COMMA simple_declarator)*)
181     | type_qualifier ;
182
183 simple_declarator:
184     left_value array_specifier* (assignment_expression)?;
185
186 struct_declaration:
```

```
187     type_qualifier? STRUCT IDENTIFIER
188     LEFT_BRACE (simple_declaration SEMICOLON)+ RIGHT_BRACE;
189
190 function_definition_statement: function_definition;
191
192 assignment_statement:
193     left_value array_struct_selection?
194     (assignment_expression | arithmetic_assignment_expression);
195
196 expression_statement: expression;
197
198 selection_statement:
199     IF LEFT_PAREN expression RIGHT_PAREN
200     selection_rest_statement ;
201
202 selection_rest_statement:
203     statement (ELSE statement)? ;
204
205 switch_statement:
206     SWITCH LEFT_PAREN expression RIGHT_PAREN
207     LEFT_BRACE statement_list RIGHT_BRACE;
208
209 case_label
210     : CASE expression COLON #case
211     | DEFAULT COLON #default;
212
213 iteration_statement
214     : WHILE LEFT_PAREN expression RIGHT_PAREN statement #while
215     | DO statement WHILE
216     LEFT_PAREN expression RIGHT_PAREN SEMICOLON #do
217     | FOR LEFT_PAREN for_init_statement for_cond_statement
218     for_rest_statement RIGHT_PAREN statement #for ;
219
220 for_init_statement
221     : (basic_statement (',' basic_statement)*)? SEMICOLON;
222
223 for_cond_statement: expression SEMICOLON;
224
225 for_rest_statement: (basic_statement (',' basic_statement)*)? ;
226
227 jump_statement
228     : CONTINUE SEMICOLON #continue
229     | BREAK SEMICOLON #break
230     | RETURN SEMICOLON #return
231     | RETURN expression SEMICOLON #return;
232
233 STRUCT: 'struct';
```

```
234
235 IF: 'if';
236 ELSE: 'else';
237 QUESTION: '?';
238
239 FOR: 'for';
240 DO: 'do';
241 WHILE: 'while';
242
243
244 CONTINUE: 'continue';
245 BREAK: 'break';
246 RETURN: 'return';
247
248 SWITCH: 'switch';
249 CASE: 'case';
250 DEFAULT: 'default';
251
252 LEFT_PAREN: '(';
253 RIGHT_PAREN: ')';
254
255 LEFT_BRACE: '{';
256 RIGHT_BRACE: '}';
257
258 LEFT_BRACKET: '[';
259 RIGHT_BRACKET: ']';
260
261 DOT: '.';
262 COLON: ':';
263 SEMICOLON: ';';
264 COMMA: ',';
265 SHARP: '#';
266
267 DECIMAL: [1-9] DIGIT* INTEGER_SUFFIX?;
268 OCTAL: '0' OCTAL_DIGIT* INTEGER_SUFFIX?;
269 HEX: ('0x' | '0X') HEX_DIGIT+ INTEGER_SUFFIX?;
270
271 FLOAT_NUM
272     :   DIGIT+ DOT DIGIT* EXPONENT? FLOAT_SUFFIX?
273     |   DOT DIGIT+ EXPONENT? FLOAT_SUFFIX?
274     |   DIGIT+ EXPONENT FLOAT_SUFFIX?
275     ;
276
277 SCALA
278     :   'bool'
279     |   'int'
280     |   'uint'
```

```

281 |   'float'
282 |   'double'
283 ;
284
285 VECTOR: ('d'|'i'|'b'|'u')? 'vec' [2-4];
286
287 MATRIX: 'd'? 'mat'[2-4] ('x'[2-4])?;
288
289 FLOAT_OPAQUE: BASIC_OPAQUE_TYPE |
290   ('sampler1DShadow' | 'sampler2DShadow' | 'sampler2DRectShadow'
291   | 'sampler1DArrayShadow' | 'sampler2DArrayShadow' |
292   'samplerCubeShadow' | 'samplerCubeArrayShadow');
293
294 INT_OPAQUE: 'i'BASIC_OPAQUE_TYPE;
295
296 U_INT_OPAQUE: 'u'BASIC_OPAQUE_TYPE | 'atomic_uint';
297
298 BASIC_OPAQUE_TYPE: ('sampler' | 'image')
299   ('1D'|'2D'|'3D'|'Cube'|'2DRect'|'1DArray'|
300   '2DArray'|'Buffer'|'2DMS'|'2DMSArray'|'CubeArray');
301
302 INCREMENT_OP : '++' | '--';
303
304 UNARY_OP : '~' | '!';
305
306 MULDIV_OP : '*' | '/' | '%';
307
308 ADDSUB_OP : '+' | '-';
309
310 SHIFT_OP : '<<' | '>>' ;
311
312 COMPARE_OP : '<' | '>' | '<=' | '>=';
313
314 EQUAL_OP:   '==' | '!=';
315
316 BITWISE_OP: '&' | '^' | '|';
317
318 LOGIC_OP:   '&&' | '^' | '||';
319
320 ASSIGNMENT_OP: '=';
321
322 ARITHMETIC_ASSIGNMENT_OP
323   :   MULDIV_OP ASSIGNMENT_OP
324   |   ADDSUB_OP ASSIGNMENT_OP
325   |   SHIFT_OP ASSIGNMENT_OP
326   |   BITWISE_OP ASSIGNMENT_OP
327 ;

```

```
328 |
329 | fragment
330 | DIGIT : [0-9];
331 |
332 | fragment
333 | HEX_DIGIT : [0-9] | [a-f] | [A-F] ;
334 |
335 | fragment
336 | OCTAL_DIGIT : [0-7];
337 |
338 | fragment
339 | INTEGER_SUFFIX : 'u' | 'U';
340 |
341 | fragment
342 | EXPONENT : ('e'|'E') ADDSUB_OP? ('0'..'9')+ ;
343 |
344 | fragment
345 | FLOAT_SUFFIX : 'f' | 'F' | 'lf' | 'LF';
346 |
347 | fragment
348 | LETTER
349 |     : [a-z]
350 |     | [A-Z]
351 |     | '_';
352 |
353 | IDENTIFIER
354 |     : LETTER (LETTER|DIGIT)*;
355 |
356 | COMMENT :
357 |     '/' * .*? '/' -> channel(HIDDEN);
358 |
359 | WS :
360 |     [ \r\t\u000C\n]+ -> channel(HIDDEN);
361 |
362 | LINE_COMMENT
363 |     : '//' ~[\r\n]* '\r'? '\n' -> channel(HIDDEN);
```