A Parser-based Tool to Assist Instructors in Grading Computer Graphics Assignments

C. Andujar[1], C. R. Vijulie[1], A. Vinacua[1]
[1]ViRVIG, Computer Science Department, Universitat Politecnica de Catalunya, Jordi Girona 1-3, Barcelona

Abstract

Although online e-learning environments are increasingly used in university courses, manual assessment still dominates the way students are graded. Interactive judges providing a pass/fail verdict based on test sets are valuable tools both for learning and assessment, but still rely on human review of the code for output-independent issues such as readability and efficiency. In this paper we present a tool to assist instructors in grading programming exercises in Computer Graphics (CG) courses. In contrast to other grading solutions, assessment is based both on checking the output against test sets, and through a set of instructor-defined rubrics based on syntax analysis of the source code. Our current prototype runs in Python and supports the assessment of shaders written in GLSL language. We tested the tool in a CG course involving more than one hundred Computer Science students per year. Our first experiments show the tool can be useful to support both self-assessment and grading, as well as detecting grading mistakes through anomaly detection techniques based on features extracted from the syntax analysis.

1. Introduction

Computer Science (CS) and Computer Graphics (CG) courses strongly benefit from the combination of lectures with practical training on laboratory sessions [TRK17]. Grading students through lab assignments is also a good strategy for assessment, both in university courses [PFM15, ACFV18] and Massive Open Online Courses (MOOCs) [FPJM15].

Online e-learning environments are increasingly used in university courses. Automatic judges [KLC01, PGR12, FJA16] are able to provide a pass/fail verdict based on test sets, and cloud-based autograders have been shown to be scalable and reliable for MOOCs [FPJM15]. These are valuable tools both for learning and assessment, but they provide only a binary output on each individual submission and thus provide very limited feedback to students, who would benefit from extended comments when submitting wrong or poor quality code.

Automatic tools simplify download, compile, run and test tasks, but provide little or no support to the posterior human review of the code. For these reasons, manual assessment still dominates the way students are graded in university courses.

Some authors do address the problem of semantic understanding of computer programs, but the subject of functional equivalence of programs [HICS80, Gör16, Jam17] is still in its infancy and often limited to elementary algorithms.

In this paper we present a tool to assist instructors in grading shader programming exercises in CG courses. In contrast to other grading solutions, assessment is based both on checking the output against test sets, and through a set of instructor-defined rubrics based on syntax analysis of the source code. Rubrics can check, e.g. whether some particular function (e.g. normalize) is called or not, and whether some particular operation (e.g. moving the vertex to clip space) is done.

The test part of the system is based on [ACFV18], a programming framework that supports self-assessment by comparing the output image of student submissions with the instructor reference solution. Grading assistance in [ACFV18] is limited to a comparison of output images, with no code analysis.

We provide instructors with a high-level API, based on syntax analysis of the source code, to define rubrics that evaluate specific code issues. We also provide limited semantic understanding of the code, to keep track of the coordinate system (object space, world space, eye space, clip space, NDC) of all geometric variables and expressions in the code.

Our current prototype runs in Python and supports the assessment of shaders written in GLSL.

We have tested our proof-of-concept prototype in an introductory course on Computer Graphics. Students take this 6 ECTS course during their fifth semester of the Computer Science degree at the *Facultat d'Informàtica de Barcelona*, *Universitat Politècnica de Catalunya*. The course receives more than one hundred students per year. The lab sessions are oriented to assist students while completing a list of exercises that involve writing shaders or C++ code. We tested our tool with some assignments of the last lab exam. We observed that well-crafted rubrics provide significant information about the submissions and thus greatly speed-up code review.

The framework is open source and it is available online.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

DOI: 10.2312/eged.20191025

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

2. Previous work

Interactive e-learning environments are increasingly used in university courses [PGR12, PHH*15, FJA16]. Some of these tools accumulate more than six years of experience with thousands of students. For example, Jutge.org [PGR12] is an online educational platform where students can try to solve programming problems in different languages. Submissions are evaluated using exhaustive test sets run under time, memory and security constraints.

For a recent review of online judges for introductory programming education we refer the reader to [FJA16]. The verdicts of most online programming judges are, essentially, binary (pass/fail). Whilst this is appropriate for competitions, university students should receive extended feedback when submitting wrong or poor quality code. Mani *et al.* [MVPR14] show how data mining of past incorrect submissions by all users can be used to extract a small subset of test cases that may be relevant to future users, thus improving feedback.

Navrat and Tvarozek [NT14] discuss how student submissions from e-learning environments can be used for summative assessment at the end of the course. By training a regression model, they were able to predict grades with a reasonable level of accuracy, suggesting that e-learning environments might replace existing assessment methods.

Individual scores, submission counts and completion times also provide valuable data so that instructors can classify exercises according to their difficulty level and design appropriate exams [RKL*16].

Massive Open Online Courses (MOOCs) have renewed the interest for *autograders*, i.e. tools for automatic evaluation of student programming assignments. Fox *et al.* [FPJM15] present a fully-automatic, test-based autograder for software engineering MOOCs. The authors show that cloud-based autograding is scalable and reliable for MOOCs, although it requires substantial setup efforts. Alternative light-weight autograders do exist, but they basically automatize download, compilation, running and testing, leaving code review to educators.

All the tools discussed above focus on the operational correctness of the code, but do not attempt to grade its construction in terms of readability, simplicity, and efficiency.

On the other hand, some authors do address the problem of automatizing the semantic understanding of computer programs. A major problem is the lack of knowledge about what constitutes functional equivalence of code segments. Determining the functional equivalence of a reference solution and the student solution would provide valuable data for both student guidance and assessment. Unfortunately, deciding the functional equivalence of two programs in general is NP-complete [HICS80]. Some attempts for grading programming assignments include matching Program Dependence Graphs (PDGs) [Gör16] and Concept Dependence Graphs (CDGs) [Jam17], but these are either limited to plagiarism detection, or require a solid knowledge base to be able to understand code segments even for elementary algorithms.

We do not attempt semantic understanding of the code, with the notable exception of keeping track of the coordinate system of all variables and expressions in the code. Instead, we provide instructors with a high-level API, based on syntax analysis of the source code, to define rubrics that evaluate specific code issues.

In the specific case of Computer Graphics, APIs either pose high entry barriers to students (e.g. OpenGL), or provide too high-level features (e.g. Unity3D). In the case of shader development, the setup and test using bare graphics APIs is too complex and thus inaccessible to students in the context of entry-level courses. For this reason, modern CG courses provide students with programming frameworks [TRK17, ACFV18] to allow the students to complete appropriate assignments during the lab sessions.

Multiple applications could be helpful in teaching shader programming. These show immediately the result of any code provided by students on a given scene [FXC, Ren, Bai]. Another set of applications make use of WebGL to offer online services where their users can experiment with fragment shaders directly [GLS, QJ]. Some have been designed with CG education in mind. Toisoul et al [TRK17] propose an IDE that allows to load models, change their material properties and test shaders. Thiesen et al [TRBB08] presented a real-time shader viewer they used in a CG course. None of these shader tools include grading features, nor facilitate the comparison of the shaders programmed by the students with those provided as reference.

Some tools try to teach the entire operation of the OpenGL pipeline. A common option is to offer a platform with the necessary tools to facilitate the development of simple OpenGL applications [PPGT14, Mil14]. Increasing the capabilities of this type of framework [BSP17] makes them approach the functionalities of a graphics engine with its added complexity. One way to reduce the impact of these abstraction mechanisms is to introduce them gradually, allowing users to increasingly access OpenGL functionality [RME14, ACFV18].

Andujar *et al.* [ACFV18] present a CG programming framework that supports student self-assessment. The authors provide a simple command-based language for describing a test set (scene to be loaded, camera setup, rendering parameters, uniform values ...) such that a correct implementation should produce the same image (up to rounding errors or hardware specific rendering options) as the instructor reference implementation. Unfortunately, grading assistance is limited to a comparison of output images, with no code analysis. Our system uses the test framework in [ACFV18] but supplements it with a rubric-based system based on code analysis.

3. System overview

Our current prototype focuses on shader programming. We use OpenGL 3.3 (core profile), which supports Vertex Shaders (VS), Geometry Shaders (GS) and Fragment Shaders (FS). The choice of OpenGL version is only motivated by limitations of the graphics cards in our current laboratories; the extension to more recent versions to support new shader types (e.g. tessellation control shaders and tessellation evaluation shaders) is straightforward.

Figure 1 shows an overview of our system, which consists of two main parts: testing and analysis.

The *testing* subsystem requires the source code of the reference solution, the source code of the student submission, and an

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

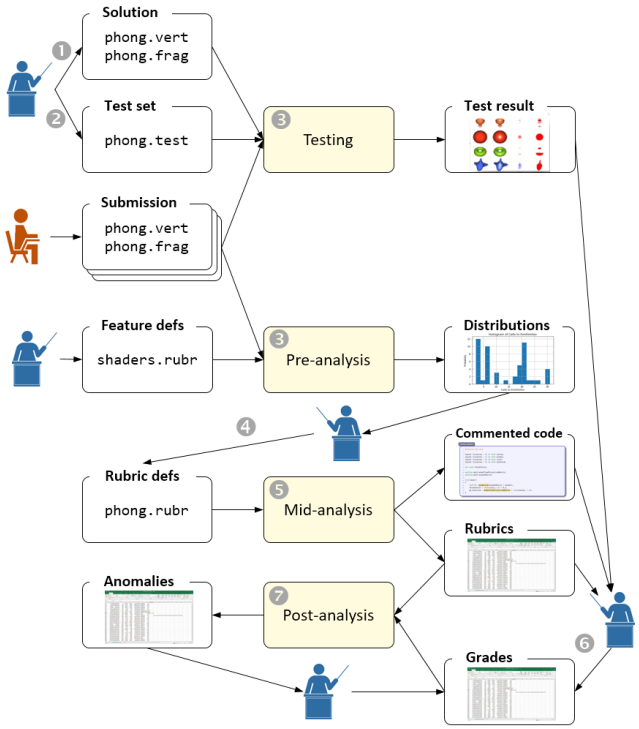

Figure 1: *Overview of our system.*

instructor-provided test file that specifies different test sets. Test sets are defined in terms of 3D models and values for the uniform variables that define the execution environment, including matrices for space transformations, material parameters and light parameters (Section 4). The output of the test set is a collection of comparison images/movies showing the output from student shaders, the output from instructor shaders, and their difference.

The analysis proceeds in three steps.

The *pre-analysis* requires no problem-specific rubrics. Instead, it parses all student submissions and extract many syntax-aware features such as number of function calls to each predefined GLSL function (e.g. dot, cross, normalize). The pre-analysis might also use a general rubric file looking for common mistakes in shaders such as unused outputs or redundant code (e.g. `v.xyz` when `v` is already a `vec3`). The output of the pre-analysis is a short report with detected outlier features (e.g. "No calls to dot() function [4 out of 88]", "No uses of normalMatrix [5 out of 88]", "Too many user-defined functions [1 out of 88]") along with the source code with these features color-highlighted. The idea is to provide instructors with an a-priori overview of anomalous/suspicious features, so that they can quickly figure out potential code problems. For example, no dot() calls in a lighting shader indicate that either the student code is wrong (this can be confirmed by the test results) or that the student implemented dot() manually, which might be slightly penalized. Besides the report, the features are also dumped into a spreadsheet where each row corresponds to a student, and columns indicate analysis results.

The *mid-analysis* is similar but requires a problem-specific rubric that checks code features based on a custom API. The API provides high-level access to features retrieved through syntax analysis of the code and limited code understanding e.g. coordinate system of any expression. For example, the instructor might want to check that the VS uses the normal in eye space,

```
"eye" in vs.space("normal")
```

or that there is at least one call to the normalize() function,

```
vs.numCalls("normalize")>=1.
```

Inspiration for writing problem-specific rubrics can be gained from the pre-analysis output plus domain-specific knowledge about the solution. For instance, Blinn-Phong shading implementations require exactly one call to the dot() function.

A *post-analysis* could be run after all grades are available. It could use the extracted features from pre-analyses and mid-analyses to detect potential anomalies in grades.

In summary, the expected workflow for instructors is as follows (Figure 1):

1. Write reference solution (.vert, .geom, .frag).
2. Write test file (.test).
3. Let the system do both the testing and pre-analysis of all submissions.
4. Read the reports (outlier features, highlighted source code) of a few students to write a specific rubric file (.rubr).
5. Let the system do the mid-analysis.
6. Grade the assignments based on (a) test results, (b) extracted rubrics from previous analyses, (c) review of the highlighted code, if needed.
7. Let the system do the post-analysis and check detected anomalies, if any.

4. Test system

Our test system is based on [ACFV18]. The system provides typical per-vertex attributes (vertex, normal, color, texCoord) and a collection of uniform variables for camera matrices, and lighting/material parameters. A test file contains a list of commands to automate model loading, texture loading, and for setting up specific camera, material, lighting and other uniform values (see Listing 1). Figure 2 illustrates the testing process. The system automatically loads the test file and executes its commands to capture screenshots of the output under different test conditions specified in the test file. These images are then compared to reference images from the instructors, and a composite image is shown to facilitate comparison and highlight differences, if any. Minor per-pixel differences might be attributed to factors not invalidating correctness, such as hardware or driver configuration differences, rounding errors... and are ignored. In practice, visual comparison of the student's output against the reference images has proved to be a powerful tool to quickly detect errors.

5. Analysis system

The analysis is based on the syntactical analysis of the GLSL code. Each rubric is essentially Python code computing a description text

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

```
resize 800x600
lightDiffuse 1 1 1
lightSpecular 1 1 1
matDiffuse 1 .5 .3
matAmbient 1 .5 .3
matSpecular 1 1 1
loadObject glass.obj
camera preset1
grab 1
loadObject man.obj
camera preset2
grab 2
```

Listing 1: *Example of a test file for a lighting shader. The* ***grab*** *command saves an image with the current settings.*

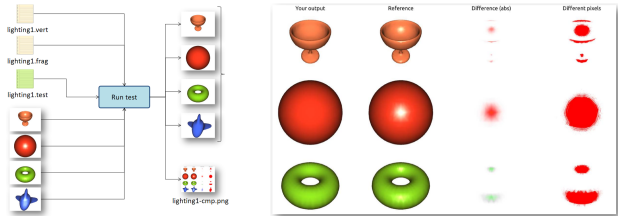

Figure 2: *Overview of the shader testing process (left) and sample output from it (right).*

plus a Boolean or numeric value. Rubrics use a custom-designed, high-level API to quickly access results from the parser. Two example rubrics are shown in Listing 2.

The first rubric simply counts the number of calls to the cross() function in the VS. The second one checks for calls to the normalize() function with a vec4 parameter.

```
% Rubric 1
R("Calls to cross", vs.numCalls("cross"))

% Rubric 2
R("Normalizing vec4", "vec4" in
vs.paramType("normalize"))
```

Listing 2: *First rubric examples.*

5.1. Rubric API

Here we provide a quick overview of our high-level API. See implementation details in Section 6.

Rubrics need to call a custom function R() that requires a description text and a value (Listing 2). Description texts will be used in the output reports and as column headers in the output CSV files.

5.1.1. Scope

Rubric code can query parser data of the VS, GS and FS through the parser objects *vs*, *gs* and *fs*. For example:

```
vs.numCalls("mix") # looks for mix() in VS
fs.numCalls("dot") # looks for dot() in FS
```

For convenience, queries can refer to multiple shaders through parser objects vsgs, vsfs, gsfs, and vsgsfs:

```
vsfs.numCalls("mix") # looks for mix() in VS & FS
```

The query scope can be further reduced through functions that return parser objects limited to a specific part of the shader (see below).

5.1.2. Return types

Most API functions include an optional parameter to specify the desired return type. For these functions, the default return value is a string (actually a list of strings, one for each occurrence). For example, given this VS:

```
vec3 L = normalize(Q - P);
```

the following functions return either strings or a parser object:

```
vs.param("normalize")
# returns ["Q - P"] as list of strings

vs.param("normalize", True)
# returns ["Q - P"] as list of parser objects
```

String return values allow for very compact rubric code. We can e.g. check if the *normal* attribute appears as parameter of a normalize() call by simply writing:

```
"normal" in vs.param("normalize")
```

Conversely, parser objects allow for further syntactical queries:

```
# number of dot() calls within for loops:
for p in vs.sentences("for", True):
  print(p.numCalls("dot"))
```

5.1.3. Querying parameters in function calls

We provide functions to check specific parameters of function calls, including predefined functions, user-defined functions, constructors and operators.

In the following examples, we omit the parser object (e.g. "vs.") for compactness:

```
param("mix", 3)
# returns 3rd parameter for all calls to "mix"
# parameter position can be omitted (default 1).

param("vec4", -1)
# returns last parameter of "vec4" constructors

param("*", 2)
# returns second parameter of product operators

paramType("mix", 3)
# returns the type of 3rd parameter (e.g. "float")
```

This rubric checks if modelMatrix appears on the right of a product operation:

```
R("Wrong order in matrix product",
"modelMatrix" in vs.param("*",2))
```

We can easily check if the normalize() function is called with a vec4 parameter:

```
R("Normalizing vec4",
"vec4" in paramType("normalize"))
```

5.1.4. Querying variables, functions and sentences

Similarly, our API provides functions that look for all appearances of specific variables, functions, operators and sentences. Again, return types can be strings or parser objects, as desired.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

```
declarations("speed")
# list of declarations of variable "speed"

assignments("L")
# list of assignments to variable "L"

# list of all uses (read, write, param) of "color"
uses("color")

# Also valid for constructors and operators
calls("normalize")
calls("vec4")
calls("*")

# appearances of the "for" sentence.
sentences("for")
```

For simplicity, every function `foo()` returning a list has a matching function `numFoo()` returning the length of the list. For example, `len(vs.calls("mix"))` can be equivalently written `vs.numCalls("mix")`.

5.1.5. AST queries

The following functions rely explicitly on the Abstract Syntax Tree (AST) built by the parser:

```
child("while","condition")
# returns the condition of all "while" sentences

isDescendantOf("discard", "if", "body")
# checks if discard sentences appear within the
# body of a conditional block.
```

This rubric checks if some while condition includes a "==" operator:

```
R("Equality in while",
"==" in ' '.join(vs.child("while","condition")))
```

We could check if a discard sentence is within a conditional block:

```
R("discard outside",
any(fs.isDescendantOf("discard", "if", "body")))
```

5.1.6. Shader-specific functions

The following functions are specific to shaders:

```
inNames()
# input variables declared in the shader

outNames()
# output variables declared in the shader

inTypes()
# type of input variables declared in the shader

outTypes()
# type of output variables declared in the shader
```

The following rubrics check if some "in" variable is not used, or some "out" variable is not assigned:

```
R("in var not used",
all([fs.numUses(v)>0 for v in vs.inNames()]))

R("out var not assigned",
all([vs.numAssignments(v)>0 for v in vs.outNames()]))
```

5.1.7. Semantic analysis

The following functions are based on code understanding. Our system recognizes the following coordinate systems: "object", "world", "eye", "clip" and "NDC".

```
space("gl_Position")
# returns the space of the variable
# in all its appearances.
```

This rubric checks that at some moment the VS writes gl_Position in clip space:

```
R("gl_Position in wrong space",
"clip" not in vs.space("gl_Position"))
```

Space tracking is limited to expressions that can be evaluated at parser time. Fortunately, most transformations are accomplished through predefined uniform matrices, which greatly simplifies this task. In our current implementation, space tracking is limited to the main() function.

5.2. Pre-analysis

The pre-analysis step goes through all student submissions of a given assignment to compute a set of general (i.e. not problem-specific) syntactical features (e.g. number of calls to EmitVertex in a GS). For each feature, we compute a histogram representing its distribution. If the distribution is not degenerate, we include its plot in a report. For categorical features, we replace the histogram by a bar chart.

Our current prototype computes features based on integers (e.g. number of calls to all predefined functions, number of common GLSL sentences, number of uses/assignments to predefined attributes, uses of uniform variables.,..) and Booleans (e.g. there are unused input vars). Here is an extract of a general feature definition file:

```
R("Calls to dot()", vs.numCalls("dot"))
R("Calls to *", vs.numCalls("*"))
...
R("Occur of for",   vs.numSentences("for"))
R("Occur of while", vs.numSentences("while"))
...
R("Uses of vertex", vs.numUses("vertex"))
R("Uses of viewMatrix", vs.numUses("viewMatrix"))
...
R("Num out vars", len(vs.outNames()))
R("Num in vars", len(fs.inNames()))
...
R("Normalizing vec4",
"vec4" in paramType("normalize"))
...
```

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

5.3. Mid-analysis

The mid-analysis step is similar to the pre-analysis above, but it uses problem-specific rubrics. For example, a phong assignment is expected to include exactly two calls to dot(), whereas Blinn-Phong requires only one. A GS performing a 1:4 subdivision is expected to include 2-4 calls to EndPrimitive().

Besides the associated distribution/bar chart report and CSV file, we also generate a PDF/HTML version of the source code where pieces corresponding to rubrics are highlighted. These files, together with test results will be the basis for faster code review and grading.

5.4. Post-analysis

We foresee that an analysis of all submissions once graded, would be of great value. For example, we could train a model to predict scores from test outputs and rubrics. The output could be the list of submissions sorted by decreasing deviations between actual grade and predicted one. Instructors could then recheck submissions with large deviations.

6. Implementation details

In order to analyze a submission, we need to generate an Abstract Syntax Tree (AST) from the source code and process it. We use ANTLR [PQ95], an open-source parser generator that takes a grammar definition and generates a lexer and a parser that can build ASTs. These trees are later processed using either the listener or the visitor mechanisms. Our prototype uses ANTLR4's Python target and an open-source GLSL grammar (https://github.com/labud/antlr4_convert) to generate both listener and visitor interfaces that we combine into a single Python library.

Listeners are objects that use a built-in parse-tree walker that triggers events at each grammar rule it encounters. Given a grammar, ANTLR generates a parse-tree Listener interface with a function for the entry and exit points of each rule in the grammar. This makes it easy to implement some basic features (e.g. counting the number of calls to a certain function) since we can extract information from interesting nodes without the need to visit any children.

Since listeners are triggered by an automatic tree walker they cannot include a return statement. More complex rubrics 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.

One particular example of such complex features is inferring the coordinate system of an expression. 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 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.

7. Results

We did a preliminary test with some recent assignments from a lab exam (86 participants).

7.1. Quads (GS)

One of the assignments asked students to write a GS that, for each input triangle, outputs four triangles, one for each quadrant of the viewport. This is a first step for a GS that shows top, left, front, perspective views of the scene. Students were advised to use NDC for translating the copies, as (x,y) coordinates of NDC copies just differ by ± 0.5.

The pre-analysis step already revealed some interesting features. Figure 3 shows the bar charts of a couple of general features. Notice how some outliers are clearly visible. 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).

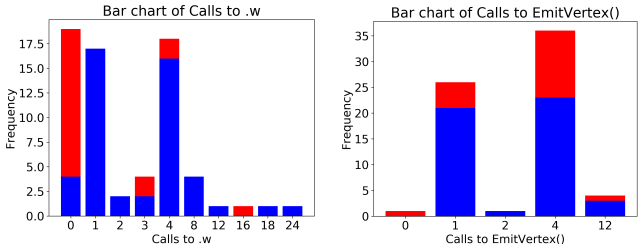

Figure 3: *Stacked bar charts for two rubrics of the Quads assignment. Submissions with correct output are shown in blue, and incorrect ones in red.*

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

For the mid-analysis, we created rubrics based on our own knowledge of the problem, starting from pre-analysis results. For example, for the Quads assignment we included, among others, the following rubrics:

```
R("Suspicious .w accesses",
gs.numCalls(".w") not in [0,1,4])

R("Too many EmitVertex calls",
gs.numCalls("EmitVertex") > 4)
```

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

7.2. Dithered cartoon shading (FS)

For this assignment students had to write a cartoon-like shader that involved the use of a noise texture and color dithering. Figure 4 shows the bar charts of two sample rubrics. The first one 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). This rubric will provide useful feedback to students whose submissions did not pass the tests.

The second bar chart shows the number of loops in the FS. 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 does not affect the operational correctness, so it cannot be detected 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.

The potential impact of a given feature on the pass/fail proportion can be evaluated through a Pearson's χ^2 test, which computes how likely it is that any observed difference on these proportions arises by chance. In other words, we test whether population proportions are the same, where populations are defined by submissions sharing the same value for the feature. A significant p-value suggests that the feature plays a role in output correctness and thus it is potentially useful as feedback about operational correctness. One example is the number of calls to normalize() in a shader that requires normalizing vectors. On the other hand, features with equal pass/fail proportions might still provide useful feedback for output-independent issues such as efficiency (e.g. overly complex code). For the features in Figure 4, p-values for the χ^2 test were 0.07 and 0.09, resp. See accompanying video for more examples.

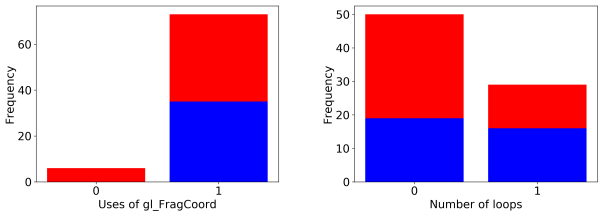

Figure 4: *Stacked bar charts for two rubrics of the Cartoon Shading assignment. Submissions with correct output are shown in blue, and incorrect ones in red.*

7.3. More complex rubrics

Our API allows writing complex rubrics with a few lines of Python code. For example, the following code checks that, if the VS outputs the normal to the FS, if it is normalized before using it:

```
# get the out vars the VS copies the normal to
L=[vs.assignments(v,True) for v in vs.outvars()]
names = [a.param("=",1) for a in L if "normal"
in a.param("=",2)]
# check they are all normalized in FS
mask=[(v in fs.param("normalize")) for v in names]
R("Normalizes normal in FS", all(mask))
```

8. Conclusions and future work

In our experience with over 10 years grading shaders, we have seen that functionally correct code by no means implies high-quality code. Indeed, correct code is often poorly written, including highly-redundant code, over-complicated solutions, inefficient code, and non-sense fragments.

From a pedagogical point of view, despite automatic judges are valuable tools for learning, the typical pass/fail output per assignment provides minimal feedback to students. On the one hand, students with wrong submissions will get, at most, a few test cases that lead to incorrect output [MVPR14]. Fortunately, the "fail" outcome is likely to encourage students to try to fix the submission (e.g. by comparing their code against a solution, so they still have a chance to learn what was wrong.

On the other hand, students with functionally correct code will just get a pass outcome, no matter the code quality. Even worse, a "pass" result might discourage students from comparing their code against a reference solution, so their shader programming mistakes are likely to persist in future submissions.

From an assessment point of view, grading through summative evaluation disregards poor-quality code in functionally-correct submissions, and give no credit to almost-right submissions.

Manual code review addresses these concerns, but at the expense of grading time, and thus it does not scale well with the number of students.

Our approach is not intended to replace manual review, but to assist evaluators (through statistics on syntactical features, rubrics, and automatically-generated code comments) in quickly detecting both functionally-incorrect code (through a test-based system comparing output images) and poor-quality code (through syntactical analysis).

Major benefits of our approach include:

- It automatizes download, compile, run, test.
- It automatizes detection of common, general errors (e.g. redundant normalization).
- It automatizes detection of problem-specific errors.
- Facilitates code review by highlighting key code fragments.
- Provides precise image-based and text-based feedback for students.

For instructors familiarized with the API, typical rubrics take one line of Python code and can be written quickly. We further amortize this rubric authoring time since (a) students get automatically-generated comments in their submissions, saving time in exam reviews, (b) many rubrics can be re-used for similar assignments, and (c) since many exam assignments become practice problems for new students, some rubrics are later used for self-assessment.

Our approach works best with assignments with low or mid-complexity (not to be confused with difficulty). The usefulness of syntactical features decreases as the assignments involve solving

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

many different tasks. For example, the number of calls to dot() is very informative when analyzing a simple lighting shader, but less informative if the same shader also performs other tasks such as normal mapping. In such cases, a solution is to ask students to write separate functions for different tasks, so that syntactical features can be computed also for each function independently.

On the other hand, the proposed workflow is cost-effective for mid-size groups (e.g. 20-800 students). For smaller groups, rubric writing time is not amortized; for larger groups and MOOCs manual code review becomes unfeasible and auto-graders should be used.

As future work, we plan to explore Multi-Dimensional outlier detection to further detect and highlight anomalous/suspicious code segments. Although we have focused in shader programming, we plan to extend the system to parse C++ code calling OpenGL/Vulkan functions.

Repository Source code for our shader analysis tool is available in the following Git repository: https://gitrepos.virvig.eu/docencia/glcheck.

Acknowledgements This work has been partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER Grant TIN2017-88515-C2-1-R.

References

[ACFV18] ANDÚJAR C., CHICA A., FAIRÉN M., VINACUA Á.: Glsocket: A cg plugin-based framework for teaching and assessment. In *EG 2018: education papers* (2018), European Association for Computer Graphics (Eurographics), pp. 25–32. 1, 2, 3

[Bai] BAILEY M.: Glman. web.engr.oregonstate.edu/~mjb/glman. Accessed: 2018-01-12. 2

[BSP17] BÜRGISSER B., STEINER D., PAJAROLA R.: bRenderer: A Flexible Basis for a Modern Computer Graphics Curriculum. In *EG 2017 - Education Papers* (2017), Bourdin J.-J., Shesh A., (Eds.), The Eurographics Association. 2

[FJA16] FRANCISCO R., JÚNIOR C. P., AMBRÓSIO A. P.: Juiz online no ensino de programação introdutória-uma revisao sistemática da literatura. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)* (2016), vol. 27, p. 11. 1, 2

[FPJM15] FOX A., PATTERSON D. A., JOSEPH S., MCCULLOCH P.: Magic: Massive automated grading in the cloud. In *CHANGEE/WAPLA/HybridEd@ EC-TEL* (2015), pp. 39–50. 1, 2

[FXC] Nvidia fx composer. developer.nvidia.com/fx-composer. Accessed: 2018-01-12. 2

[GLS] Glsl sandbox. glslsandbox.com. Accessed: 2018-01-12. 2

[Gör16] GÖRG T.: Interprocedural PDG-based code clone detection. *Softwaretechnik-Trends 36*, 2 (2016). 1, 2

[HICS80] HUNT III H. B., CONSTABLE R. L., SAHNI S.: On the computational complexity of program scheme equivalence. *SIAM Journal on Computing 9*, 2 (1980), 396–416. 1, 2

[Jam17] JAMIL H. M.: Automated personalized assessment of computational thinking mooc assignments. In *2017 IEEE 17th International Conference on Advanced Learning Technologies (ICALT)* (July 2017), vol. 00, pp. 261–263. 1, 2

[KLC01] KURNIA A., LIM A., CHEANG B.: Online judge. *Computers & Education 36*, 4 (2001), 299–315. 1

[Mil14] MILLER J. R.: Using a software framework to enhance online teaching of shader-based opengl. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (2014), SIGCSE '14, pp. 603–608. 2

[MVPR14] MANI A., VENKATARAMANI D., PETIT J., ROURA S.: Better feedback for educational online judges. In *Proceedings of the 6th International Conference on Computer Supported Education - Volume 2: CSEDU,* (2014), INSTICC, SciTePress, pp. 176–183. 2, 7

[NT14] NAVRAT P., TVAROZEK J.: Online programming exercises for summative assessment in university courses. In *Proceedings of the 15th International Conference on Computer Systems and Technologies* (2014), ACM, pp. 341–348. 2

[PFM15] POŽENEL M., FÜRST L., MAHNIČ V.: Introduction of the automated assessment of homework assignments in a university-level programming course. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on* (2015), IEEE, pp. 761–766. 1

[PGR12] PETIT J., GIMÉNEZ O., ROURA S.: Jutge.org: An educational programming judge. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2012), SIGCSE '12, ACM, pp. 445–450. 1, 2

[PHH*15] PETTIT R. S., HOMER J. D., HOLCOMB K. M., SIMONE N., MENGEL S. A.: Are automated assessment tools helpful in programming courses. In *122nd ASEE Annual Conference and Exposition. American Society for Engineering Education* (2015), vol. 2015. 2

[PPGT14] PAPAGIANNAKIS G., PAPANIKOLAOU P., GREASSIDOU E., TRAHANIAS P.: glGA: an OpenGL Geometric Application Framework for a Modern, Shader-based Computer Graphics Curriculum. In *Eurographics 2014 - Education Papers* (2014), Bourdin J.-J., Jorge J., Anderson E., (Eds.), The Eurographics Association. 2

[PQ95] PARR T. J., QUONG R. W.: Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience 25*, 7 (1995), 789–810. 6

[QJ] QUILEZ I., JEREMIAS P.: Shadertoy. www.shadertoy.com. Accessed: 2018-01-12. 2

[Ren] Rendermonkey toolsuite. gpuopen.com/archive/gamescgi/rendermonkey-toolsuite/rendermonkey-toolsuite-ide-features. Accessed: 2018-01-12. 2

[RKL*16] RAJALA T., KAILA E., LINDÉN R., KURVINEN E., LOKKILA E., LAAKSO M.-J., SALAKOSKI T.: Automatically assessed electronic exams in programming courses. In *Proceedings of the Australasian Computer Science Week Multiconference* (New York, NY, USA, 2016), ACSW '16, ACM, pp. 11:1–11:8. 2

[RME14] REINA G., MÜLLER T., ERTL T.: Incorporating modern opengl into computer graphics education. *IEEE Computer Graphics and Applications 34*, 4 (2014), 16–21. 2

[TRBB08] THIESEN M., REIMERS U., BLOM K., BECKHAUS S.: Shaderschool: a tutorial for shader programming. In *CGEMS: Computer graphics educational materials source* (01 2008), p. 10. 2

[TRK17] TOISOUL A., RUECKERT D., KAINZ B.: Accessible GLSL shader programming. In *Eurographics 2017 - Education Papers, Lyon, France, April 24-28* (2017), pp. 35–42. 1, 2

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.