Department: Education
Editors: Beatriz Sousa Santos, bss@ua.pt and Ginger Alford, alfordg@smu.edu

Syntactic and Semantic Analysis for Extended Feedback on Computer Graphics Assignments

Carlos Andujar, Cristina R. Vijulie, and Àlvar Vinacua
ViRVIG - Universitat Politecnica de Catalunya - BarcelonaTech

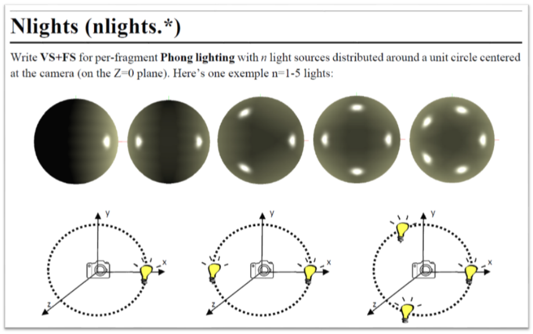
***Abstract*—Modern Computer Graphics courses require students to complete assignments involving computer programming. The evaluation of student programs, either by the student (self-assessment) or by the instructors (grading) can take a considerable amount of time and does not scale well with large groups. Interactive judges giving a pass/fail verdict do constitute a scalable solution, but they only provide feedback on output correctness. In this paper we present a tool to provide extensive feedback on student submissions. The feedback is based both on checking the output against test sets, as well as on syntactic and semantic analysis of the code. These analyses are performed through a set of code features and instructor-defined rubrics. The tool is built with Python and supports shader programs written in GLSL. Our experiments demonstrate that the tool provides extensive feedback that can be useful to support self-assessment, facilitate grading and identify frequent programming mistakes.**

■ **COMPUTER PROGRAMMING EXERCISES** are a keystone in current Computer Science and Computer Graphics (CG) courses [1]. Most traditional courses [2], [3] and Massive Open Online Courses (MOOCs) base their student training and evaluation on programming assignments. However, a feedback-rich evaluation of student work takes a considerable amount of time.

Most programming exercises require students to construct code. Automatic judges [4] can run their code using instructor-provided test data to provide a pass/fail verdict. These judges have been shown to be scalable for MOOCs [5]; when run in real time, iterations could occur rapidly, providing dynamic feedback to the student. Unfortunately, they provide only a binary output on each individual exercise and thus offer limited feedback to students.

Artificial Intelligence (AI) techniques have been proposed to learn how instructors grade a problem. Instructors first evaluate a sample set of student responses to create a training set. The system then creates a model reflecting the instructor's grading decisions, which can be used to grade the work of other students. These techniques are increasingly used to support assessment of e.g. essays due to its efficiency, consistency and immediate feedback. A major issue is the opacity of the applied rules, which prevent their use for making final decisions on student performance.

In this paper we present a tool to provide feature-rich feedback on student submissions. We focus on shader programming exercises in CG courses (see Figure 1 and Listing 1). In contrast to other evaluation solutions, assessment is based both on checking the output against test sets, and through a set of instructor-defined rubrics based on syntactic and semantic analysis of the code.



Nlights (nlights.*)

Write **VS+FS** for per-fragment **Phong lighting** with *n* light sources distributed around a unit circle centered at the camera (on the Z=0 plane). Here's one exemple n=1-5 lights:

Figure 1. One exercise involving shader programming.

```
// Vertex shader
in vec3 vertex, normal;
out vec3 pos, norm;
uniform mat3 normalMat;
uniform mat4 modelViewMat;
uniform mat4 modelViewProjectionMat;

void main() {
 norm = normalMat * normal;
 pos = (modelViewMat * vec4(vertex, 1)).xyz;
 gl_Position = modelViewProjectionMat *
              vec4(vertex, 1.0);
}

// Fragment shader
in vec3 pos, norm;
uniform int n;
void main() {
 fragColor = vec4(0);
 vec3 N = normalize(norm);
 vec3 V = normalize(-pos);
 for (int i=0; i<n; ++i) { // for each light
  vec3 light = vec3(cos(2*PI*i/n),
                    sin(2*PI*i/n), 0);
  vec3 L = normalize(light.xyz - pos);
  fragColor += Phong(N, V, L);
 }
}
```

Listing 1. One GLSL solution to the exercise in Figure 1. The definition of *Phong*() is omitted for conciseness.

Rubrics can check, for example, whether some particular function (e.g. normalize) is called or not, and whether some particular operation (e.g. moving the vertex position to eye space) is done. Thanks to the semantic analysis, the tool is able to track the coordinate space of variables representing 3D points and vectors, and thus detect coordinate space inconsistencies.

A general assumption in this paper is that exercises are written so that the output (for a given input) is deterministic and fixed. This means students must be provided with a CG platform facilitating the description of the input (3D models, cameras...) and the definition of test sets. Modern CG courses already provide students with programming frameworks [6], [7], [8], [9]. In this paper we adopt the CG framework proposed in [3]. The framework provides a simple command-based language for describing a test set such that a correct implementation should produce the same image (up to rounding errors or hardware specific rendering options) as the instructor reference implementation (see Figure 2).

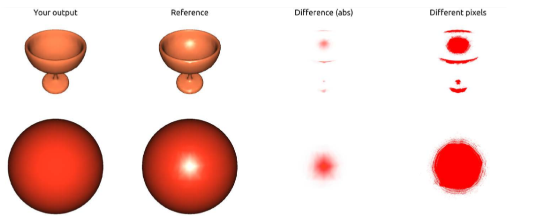


Figure 2. Shader testing: the system runs the shaders to get the output images under different test conditions. These images are then compared to reference images.

Rubric characterization

We analyzed a large sample of manually-graded exercises (about one thousand), looking for instructor annotations. We classified the annotations according to different criteria.

Concerning the level of *specificity*, we consider three types of issues. Some issues are specific to a particular exercise. These annotations are highly unlikely to reoccur in other exercises (e.g. "incorrect texture offset" in a parallax mapping exercise). Other issues are specific to a family of exercises. For example, most lighting models use the light vector, which is assumed to be unit-length. An annotation such as "light vector not normalized" is likely to occur in different lighting exercises. Finally, some issues apply to any shader. For example, "gl_Position not written in clip space". The more general an issue is, the more chances we can re-use the rubric code that checks its occurrence.

Another criterion is the impact on *output correctness*. Some errors cause wrong output for all cases (e.g. "the light vector is not normalized") or for some specific cases (e.g. "this only works for directional lights"). Some errors might invalidate the output only on very specific cases, and thus can be hard to catch via tests. For example, "your code assumes non-negative texture coordinates". Other errors have no impact on output correctness. This category includes a large number of mistakes, like poor quality code, redundant computations and unused variables.

We found annotations to be useful in all cases above. However, we are specially interested in errors with little or no impact on output correctness, since a "pass" verdict for all test cases might be mistakenly interpreted by the student as his solution being "perfect".

API design

A large part of the annotations above can be computed automatically provided that a high-level tool for syntactic and semantic analysis is available. We propose a high-level Python API able to recognize shading language elements such as definitions, assignments, function calls and so on. Our current prototype supports GLSL 3.3, which includes Vertex Shaders (VS), Geometry Shaders (GS) and Fragment Shaders (FS). We provide below a quick summary of the API. See the source code repository for further details.

Each rubric is essentially Python code computing a description text plus a value. 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. Rubric examples (Python code)

Rubrics can query parser data of the VS, GS and FS through the parser objects *vs*, *gs* and *fs*. API functions include an optional parameter to specify the desired return type. The default is string (actually a list of strings, one for each occurrence). 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:

```
# detecting nested loops
for p in vs.statements("for", True):
  print(p.numStatements("for"))
```

We provide functions to check specific parameters of function calls. In the following examples, we omit the parser object (e.g. "vs.") for compactness:

```
param("mix", 3)
# 3rd parameter of "mix" calls
```

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

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

Similarly, our API provides functions that look for all appearances of specific variables, functions, operators and statements. 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()]))
```

The API also offers functions that rely explicitly on the Abstract Syntax Tree (AST). For example, the following rubric checks if a **`discard`** statement is within a conditional block:

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

Our API also supports coordinate space tracking ("object", "world", "eye", "clip") through semantic analysis. 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, transformations are accomplished through predefined uniform matrices, which simplifies this task.

API Implementation

We use ANTLR [10], an open-source parser generator, and an open-source GLSL grammar (https://github.com/labud/antlr4_convert) to generate both listener and visitor interfaces.

Listeners use a built-in AST walker that triggers events at each grammar rule it encounters. This makes it easy to implement basic features (e.g. counting the number of calls to a certain function) since we can extract information from relevant nodes without visiting any children.

More complex rubrics require an explicit control of the AST traversal. We use visitors, which let us visit children nodes explicitly. This makes it possible to implement features that need information on the context of the whole program. One example is inferring the coordinate space of an expression. We keep track of each variable declaration, assignment and its coordinate system along each possible path that the program can take.

API usage

Here we describe a potential workflow based on our API. The workflow consists of three main parts: testing, feature analysis and rubric evaluation.

The *testing* step generates images comparing the output of student shaders with that of instructor shaders.

The *feature analysis* step parses all student submissions and extracts hundreds of syntax-aware features such as number of function calls to each predefined GLSL function (e.g. dot, cross, normalize). This step might also use a general rubric file looking for common mistakes in shaders such as unused outputs or redundant code. We consider a feature to be relevant if it has some variance among submissions and it has a significant impact on output correctness (see below). The output is a short report with detected relevant features. The idea is to provide instructors with an a-priori overview of suspicious features, so that they can quickly figure out potential code problems. Instructors can then turn some relevant features into problem-specific rubrics with more clarifying comments, e.g. “the light vector is not normalized before its use”.

The *rubric evaluation* step parses again all student submissions, using those problem-specific rubrics. The output includes detailed comments and, whenever possible, highlighed code.

Finding relevant features

Figure 3 shows bar charts for a couple of features on an exercise asking for a GS that outputs four copies of each input triangle, one for each quadrant of the viewport. Students were advised to use NDC for translating the copies, as (x,y) coordinates of NDC copies just differ by ± 0.5. Conversion from clip to NDC requires dividing by the homogeneous coordinate w. 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.

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 or poorly-factorized code.

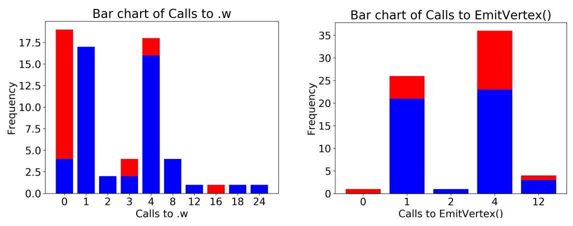


Figure 3. Stacked bar charts for two features of an exercise (about 80 submissions). Submissions with correct output are shown in blue, and incorrect ones in red.

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. 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).

Features might reveal common mistakes when solving a particular exercise, e.g. no access to the homogeneous coordinate (.w) in a shader that requires a perspective division. The instructor can decide to turn the feature into a rubric:

```
# feature
R("Calls to .w", vs.numFieldSelectors("w"))

# rubric
R("Accessing w coordinate was needed to "+
  "perform the perspective division",
   vs.numFieldSelectors("w") == 0)
```

The instructor might also decide a penalty:

```
# -1 point for unnecessary loops
P("No loops are required in this exercise",
 -1 if (fs.numStatements("for") > 0) else 0)
```

Feedback examples

Our system provides extensive feedback both to instructors and students.

Figure 4 shows one possible web-based interface to analyze a list of submissions. For large groups, the analysis can be done on a submission sample. The interface has four main parts. The *extracted features* panel includes a table with the relevant features extracted automatically from the syntactic and semantic analysis of the source code. The instructor can click on any feature (e.g. calls to a particular GLSL function) to see a bar chart with the feature value distribution. Instructors can sort the table for example by p-value to start analyzing those features likely to impact output correctness. The *submission list* is actually a table where rows correspond to student submissions and columns indicate compile/link results, test results (pass/fail) and the value for all the features. A particular submission can be further explored by clicking on it. The submission results for different test cases are shown in the *test results* panel. Its source code (VS, GS, FS) and compile/link logs can be also reviewed in the *source code* panel.

Currently, our approach when using this interface is to analyze features one-by-one. The bar chart already conveys important information about the feature, and the p-value of the Pearson's χ^2 test might confirm a significant impact of the feature on output correctness. Sometimes we found unexpected features to play an important role on output correctness. For example, in a quantization exercise the use of loops indicated an overly complex solution that was more prone to errors. Some features reveal student shortcomings. For example, in an exercise we found that some students computed integer reminders using a loop instead of using the modulo operator. In this case, the feature was "number of loops", and an inspection of the source code of a few samples revealed the purpose of the loop.

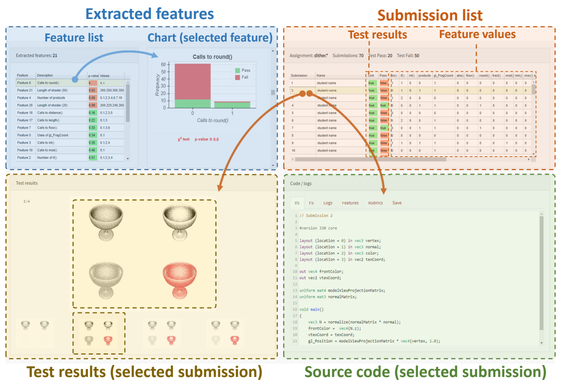
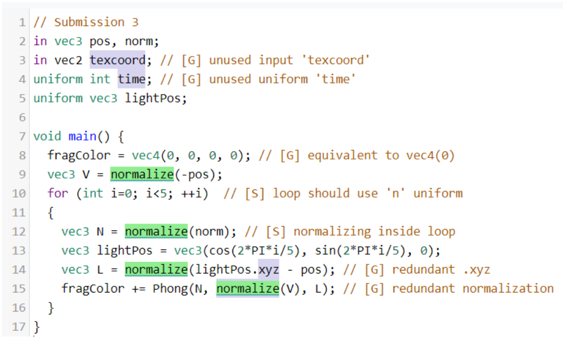


Figure 4. Example of web-based interface to analyze extracted features on a dithering FS. Arrows indicate that panel data depends on the selected feature/submission.



```
// Submission 3
in vec3 pos, norm;
in vec2 texcoord; // [G] unused input 'texcoord'
uniform int time; // [G] unused uniform 'time'
uniform vec3 lightPos;

void main() {
  fragColor = vec4(0, 0, 0, 0); // [G] equivalent to vec4(0)
  vec3 V = normalize(-pos);
  for (int i=0; i<5; ++i)  // [S] loop should use 'n' uniform
  {
    vec3 N = normalize(norm); // [S] normalizing inside loop
    vec3 lightPos = vec3(cos(2*PI*i/5), sin(2*PI*i/5), 0);
    vec3 L = normalize(lightPos.xyz - pos); // [G] redundant .xyz
    fragColor += Phong(N, normalize(V), L); // [G] redundant normalization
  }
}
```

Figure 5. Automatically generated comments.

Once a feature is confirmed to be worth communicating to students, the instructor can convert its Python code into a rubric with a more descriptive comment. The source code panel in the web-based interface (Figure 4) includes tabs for feature and rubric code to facilitate this task.

All the information extracted from the final rubrics on a submission can be made available to the student. Furthermore, most rubrics refer to specific code segments and thus descriptive comments can be added automatically to specific lines of the student-constructed code to further facilitate finding their location in the code. Figure 5 shows one example of submission with comments and highlighted code. Students are also provided with the image comparing their output with reference images.

Conclusions

From a pedagogical point of view, automatic judges giving just a pass/fail output provide 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. 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 against a valid solution, so their shader programming mistakes are likely to persist in the future.

Our approach is not intended to replace manual review, but to assist students and instructors (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). As future work, we plan to explore the integration of AI techniques on automatically-detected and instructor-provided rubrics.

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

1. A. Toisoul, D. Rueckert, and B. Kainz, "Accessible GLSL shader programming," in *Eurographics 2017 - Education Papers, Lyon, France, April 24-28*, pp. 35–42, 2017.
2. M. Poženel, L. Fürst, and V. Mahnič, "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*, pp. 761–766, IEEE, 2015.
3. C. Andújar, A. Chica, M. Fairén, and À. Vinacua, "Gl-socket: A cg plugin-based framework for teaching and assessment," in *EG 2018: education papers*, pp. 25–32, European Association for Computer Graphics (Eurographics), 2018.
4. J. Petit, O. Giménez, and S. Roura, "Jutge.org: An educational programming judge," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, (New York, NY, USA), pp. 445–450, ACM, 2012.
5. A. Fox, D. A. Patterson, S. Joseph, and P. McCulloch, "Magic: Massive automated grading in the cloud.," in *CHANGEE/WAPLA/HybridEd@ EC-TEL*, pp. 39–50, 2015.
6. J. R. Miller, "Using a software framework to enhance online teaching of shader-based opengl," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pp. 603–608, 2014.
7. G. Reina, T. Müller, and T. Ertl, "Incorporating modern opengl into computer graphics education," *IEEE Computer Graphics and Applications*, vol. 34, no. 4, pp. 16–21, 2014.
8. B. Bürgisser, D. Steiner, and R. Pajarola, "bRenderer: A Flexible Basis for a Modern Computer Graphics Curriculum," in *EG 2017 - Education Papers* (J.-J. Bourdin and A. Shesh, eds.), The Eurographics Association, 2017.
9. M. Thiesen, U. Reimers, K. Blom, and S. Beckhaus, "Shaderschool: a tutorial for shader programming," in *CGEMS: Computer graphics educational materials source*, p. 10, 01 2008.
10. T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

Carlos Andujar is associate professor at the Computer Science Department of the Universitat Politecnica de Catalunya - BarcelonaTech, and senior researcher at the Research Center for Visualization, Virtual Reality and Graphics Interaction, ViRVIG. His research interests include 3D modeling, Computer Graphics, and Virtual Reality. Contact him at andujar@cs.upc.edu.

Cristina R. Vijulie is research assistant at the Computer Science Department of the Universitat Politecnica de Catalunya - BarcelonaTech. Her research interests include Computer Graphics, Virtual Reality and Artificial Intelligence. Contact her at vijulie@cs.upc.edu.

Àlvar Vinacua is associate professor at the Computer Science Department of the Universitat Politecnica de Catalunya - BarcelonaTech, and senior researcher at the Research Center for Visualization, Virtual Reality and Graphics Interaction, ViRVIG. His research interests include Geometry Processing, 3D Modeling, Computer Graphics and Virtual Reality. Contact him at alvar@cs.upc.edu.