GL-Socket: A CG plugin-based framework for teaching and assessment

C. Andujar[1], A. Chica[1], M. Fairen[1], A. Vinacua[1]
[1]VirVIG, Computer Science Department, Universitat Politecnica de Catalunya, Jordi Girona 1-3, Barcelona, Spain

Abstract
In this paper we describe a plugin-based C++ framework for teaching OpenGL and GLSL in introductory Computer Graphics courses. The main strength of the framework architecture is that student assignments are mostly independent and thus can be completed, tested and evaluated in any order. When students complete a task, the plugin interface forces a clear separation of initialization, interaction and drawing code, which in turn facilitates code reusability. Plugin code can access scene, camera, and OpenGL window methods through a simple API. The plugin interface is flexible enough to allow students to complete tasks requiring shader development, object drawing, and multiple rendering passes. Students are provided with sample plugins with basic scene drawing and camera control features. One of the plugins that the students receive contains a shader development framework with self-assessment features. We describe the lessons learned after using the tool for four years in a Computer Graphics course involving more than one hundred Computer Science students per year.

1. Introduction

Computer Graphics (CG) courses strongly benefit from the combination of lectures with practical training on laboratory sessions [TRK17]. Unfortunately, APIs for graphics programming either pose high entry barriers to students (e.g. OpenGL), or provide too high-level features (e.g. Unity3D) preventing Computer Science students from completing assignments involving lower level tasks such as object drawing and resource management (shaders, textures). 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] plus code examples to allow the students to complete appropriate assignments during the lab sessions.

Although some OpenGL and GLSL frameworks are freely available and open-source, some key pedagogical features are commonly missing. In our view, a CG development tool for teaching should combine the following features:

- Provide support for exercises that can be completed, tested and assessed independently from other exercises. This lack (or at least minimization) of inter-dependencies has proven to greatly simplify testing, debugging, reusability and assessment. This also means students can complete the exercises in an arbitrary order. If they get stuck in an exercise, they can safely move on to another one until they get the missing knowledge or receive teacher assistance.
- Be flexible enough to allow students to complete assignments with varying levels of difficulty and complexity, from quick assignments (e.g. bounding box computation, camera setup) to more complex ones (e.g. shadow mapping, mirror reflection).
- Provide support for student self-assessment: the framework must allow students to get a first visual validation of their code, by comparing the output (either numerical or, most frequently, graphical) of their solution with the instructor reference solution, on a test set. This also implies a simple mechanism for designing test files (describing e.g. scene and camera setup) and running them on student solutions.
- Supports semi-automatic correction of lab exams: the tool must support batch running of student submissions on a validation test (different from the test set), and compare graphically the resulting output with that from the reference solution.
- The features above should be available both when developing OpenGL C++ code, GLSL shaders, or both, and also for everyday exercises and exam exercises running in a secure (e.g. no internet connection) environment.
- The software must be multi-platform (Linux, MacOs, Windows).

The key ingredients of our C++ framework to fulfill all the requirements above are:

- A plugin-based architecture that allows assignments to be cast onto plugin/shader submissions that can be completed, tested, self-assessed and marked independently.
- The interface consists of a minimal collection of (non-pure) virtual methods, all of them with a default implementation. The methods, e.g. onPluginLoad(), preFrame(), postFrame(), drawScene(), paintGL(), have been carefully chosen to simplify development by clearly distinguishing one-shot initialization, per-frame setup, and drawing code. When possible, we borrowed

DOI: 10.2312/eged.20181003

method names from frameworks that students might be familiar with (mainly Qt). This separation fosters reusability and allows students to focus on CG programming for the problem at hand rather than peripheral programming tasks.
- A simple command-based language for describing a test set (plugins to be loaded, scene to be loaded, camera setup, rendering parameters, uniform values for the shaders...) such that a correct implementation should produce the same image (up to rounding errors or hardware specific rendering options) as the instructor reference implementation.

We have used the GL-Socket framework for more than four years in a 15-week introductory course on Computer Graphics (G). 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, being compulsory for students following the *Computing* specialization, and optional for the remaining students.

The course is preceded by a course on Graphical User Interface Design and Usability, which also covers some CG topics related with camera setup and 3D model representation. Students already start with a reasonable background on C++ and Qt programming.

The course has 2 lecture hours and 2 lab hours per week. The lab sessions are oriented to assist students while completing a list of exercises that involve writing shaders (first half of the term) or plugins (second half). There are two lab exams (midterm and final) where students must complete 3-4 exercises within 2 hours (Fig. 5).

We received very positive feedback from both students and teachers. The framework is open-source and is available online.

2. Related work

There is a myriad of frameworks available to facilitate CG development. Here we focus on those that can be useful for introductory CG courses, for teaching either OpenGL, GLSL shaders, or both.

Tools for shader development The growth of general programming on GPUs has contributed to raising the barrier of entry to graphics API programming. In particular, the removal in OpenGL 3.1 of many functionalities has increased the length and complexity of the necessary code to have a minimal OpenGL application. One way to deal with this difficulty is to focus on teaching GLSL and the multiple shaders that are used in the modern OpenGL pipeline. In this way we avoid the complexity of all the necessary initialization code.

Multiple applications could be helpful in teaching shader programming. These show immediately the result of any code provided by students on a given scene with a given set of uniforms (values and textures). The development of many of them has been abandoned [FXC, Ren, Bai, fCGVR] although in some cases they have been used as base code for other projects that have tried to revitalize them. The consequence is that they do not have full support for modern versions of OpenGL.

Another set of applications make use of WebGL to offer online services where their users can experiment with fragment shaders directly [GLS, QJ]. These draw a single full screen quad on which the user-programmed fragment shader is applied. The community that forms around these websites offers great potential for education. However they are restricted to fragment shaders and rendering an arbitrary geometry with a regular rendering pipeline is not possible.

Some have been designed with computer graphics education in mind. Toisoul et al [TRK17] propose an IDE that allows to load models, change their transformations and material properties and send them to a sequence of shaders that the student can program. Thiesen et al [TRBB08] presented a real-time shader viewer they used to teach a computer graphics course.

None of the tools we know of include an automatic way to compare the shaders programmed by the students with those provided as reference.

Game Engines There are several 3D game engines that could be used to teach computer graphics (e.g. Unity3D, Unreal). To these we may add those engines exclusively oriented to the graphic component of an application (e.g OGRE, threeJS). These automate the application of lighting effects, the use of camera systems, and loading and visualizing models and shaders from files. All these functionalities are useful in the development of graphics applications, but their main drawback from an educational standpoint is that they manage and tend to hide those things that a beginner needs to learn.

The alternative is to use these engines so that it is the student who must create the objects and program the shaders from scratch, but having the advantage of the engine infrastructure. The problem is then that the engine overcomplicates all the necessary tasks with particular characteristics of that engine. The goal of a generic computer graphics course should be to expose the student to the principles that game engines tend to handle automatically.

Tools for OpenGL teaching Comparatively few tools try to teach the entire operation of the OpenGL pipeline. A common strategy is to offer an interactive document with small applications similar to Java applets to display the operation of the various phases of the pipeline [RRP00]. Despite its usefulness in exposing the essential concepts of computer graphics, they are not as valuable for the programming tasks that students should undertake.

It is also possible to provide a software renderer that allows the students to learn about the whole pipeline. Fink et al [FWW12] provided such a system to their students, allowing them to teach rasterization and shaders using the same tool. Even though this strategy reveals the inner workings of the graphics pipeline, working with a regular API allows the students to program GPU accelerated applications.

Another option is to offer a platform with the necessary infrastructure to facilitate the development of simple applications in OpenGL. In this direction Papagiannakis et al [PPGT14] presented glGA, a lightweight C++ framework that contains four simple examples, as well as six tasks to be completed by the students. Miller [Mil14] also presented a C++ framework that achieves the same goal.

WebGL [FP13] can be used to motivate students through the fast iteration that scripting languages allow, but large applications tend to be programmed in compiled languages. Also static typing already eliminates a class of programming errors.

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].

3. Plugin architecture

Our current version uses Qt framework version 5.7 and 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 of GL-Socket to more recent versions to support new shader types (e.g. tessellation control shaders and tessellation evaluation shaders) is straightforward.

A major goal in the design of the framework is to enable a one-to-one mapping between exercises and C++ plugins/GLSL shaders, i.e. solving an exercise involves writing a single plugin and/or a shader program. A second goal is to facilitate deterministic output for a given assignment; this implies providing well-defined default parameters for all OpenGL state variables that could affect rendering, including scene, camera, frame buffer configuration, and so on.

3.1. Plugin interface

```
class Plugin
{
public:
    Plugin();
    virtual ~Plugin();

    virtual void onPluginLoad();
    virtual void onObjectAdd();
    virtual void onSceneClear();

    virtual void preFrame();
    virtual void postFrame();

    virtual bool drawScene();
    virtual bool paintGL();

    virtual void keyPressEvent(QKeyEvent*);
    virtual void mouseMoveEvent(QMouseEvent*);
    // ...

    Scene* scene();
    Camera* camera();
    Plugin* drawPlugin();
    GLWidget* glwidget();
 };
```

Listing 1: *Plugin interface.*

The plugin interface is shown in Listing 1. Some method names such as paintGL(), keyPressEvent() and mouseMoveEvent() are borrowed from Qt, whereas preFrame() and postFrame() methods mimic those found in the VRJuggler vrj::App interface.

Writing a plugin simply involves deriving a class from this interface. All virtual functions have a default implementation, which is empty (except for those methods returning a boolean, which simply return false to indicate they have not been re-implemented). This means that students only need to override a (typically small) subset of the interface methods.

Although we use a single interface, we ask students to distinguish four types of plugins, depending on their main purpose and the subset of methods they override. These four types are described below.

3.1.1. Effect plugins

An Effect Plugin is intended to implement simple effects involving OpenGL state changes (such as enabling alpha blending or binding a specific shader), or drawing additional content (e.g. a bounding box). An effect plugin typically overrides the preFrame() and/or postFrame() methods.

The preFrame() method is called by GLWidget::paintGL() before drawing the scene. This allows plugins to execute some code right before the scene is rendered. A typical use would be to bind a shader affecting the whole scene, or to change other OpenGL state variables.

Similarly, the postFrame() method is called after drawing the scene. This allows plugins to execute some code right after the scene has been rendered. A typical use would be to unbind a shader, or to draw additional primitives. Effect methods are often simple to implement because they do not need to care about drawing the scene itself.

```
// blending.h
#include "plugin.h"

class Blending: public QObject, public Plugin
 {
     Q_PLUGIN // Uses Qt Plugin mechanism

 public:
    void preFrame();
    void postFrame();
 };

----------------------------------------------

// blending.cpp
#include "blending.h"

void Blending::preFrame()
{
    glDisable(GL_DEPTH_TEST);
    glBlendEquation(GL_FUNC_ADD);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glEnable(GL_BLEND);
}

void Blending::postFrame()
{
    glEnable(GL_DEPTH_TEST);
    glDisable(GL_BLEND);
}
```

Listing 2: *A simple plugin for enabling alpha blending.*

Listing 2 shows a simple effect plugin for enabling alpha blending. Listing 3 shows an effect plugin that loads and binds a shader program (not shown). The plugin overrides onPluginLoad(), which is called at plugin loading time, to load, compile and link the shaders, using the convenient Qt classes QOpenGLShaderProgram and QOpenGLShader. Examples here have been slightly simplified for clarity. Refer to the full code in the public repository.

```
// myshader.h
#include "plugin.h"

class MyShader: public QObject, public Plugin
 {
     Q_PLUGIN // Uses Qt Plugin mechanism

 public:
    void onPluginLoad();
    void preFrame();
    void postFrame();
 private:
    QOpenGLShaderProgram* program;
 };

----------------------------------------------------------

// myshader.cpp
#include "myshader.h"

void MyShader::onPluginLoad()
{
    QOpenGLShader *vs, *fs;
    vs = new QOpenGLShader(QOpenGLShader::Vertex, this);
    vs->compileSourceFile("shader.vert");
    fs = new QOpenGLShader(QOpenGLShader::Fragment, this);
    fs->compileSourceFile("shader.frag");
    program = new QOpenGLShaderProgram(this);
    program->addShader(vs); program->addShader(fs);
    program->link();
}

void MyShader::preFrame()
{
    program->bind();
    program->setUniformValue("modelViewProjMatrix", ...);
}

void MyShader::postFrame()
{
    program->release();
}
```

Listing 3: *A simple plugin for binding a user-defined shader.*

3.1.2. Draw plugins

A Draw Plugin overrides the drawScene() method, which is in charge of drawing all the objects forming the scene by issuing OpenGL rendering commands. For the GL-Socket application to work, at least one plugin must provide an implementation for this method. When using OpenGL Core Profile, Draw Plugins must traverse the scene objects to create Vertex Array Objects (VAOs).

We provide students with a default Draw Plugin, which is loaded automatically, so that students only need to write new draw plugins for practicing VAO definition, for example, implementing different ways of computing per-vertex or per-corner normals on polygonal meshes when building a VAO.

3.1.3. Render Plugins

Render Plugins override the paintGL() method, which is responsible for re-painting the scene, using one or more rendering passes. The default Render Plugin simply clears the frame buffer and calls the drawScene() method of the current Draw Plugin (Listing 4). This conceptual separation of drawing and rendering facilitates practicing with multi-pass techniques (e.g. shadow mapping) without having to deal with VAOs.

3.1.4. Action Plugins

Some tasks require user code to be executed in response to user interaction, instead of a regular per-frame execution. An Action

```
// render.h
#include "plugin.h"

class Render: public QObject, public Plugin
 {
     Q_PLUGIN // Uses Qt Plugin mechanism

 public:
    bool paintGL();
 };

-------------------------------------------------------------------

// render.cpp
#include "render.h"
bool Render::paintGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    drawPlugin()->drawScene();
    return true;
}
```

Listing 4: *A simple plugin for rendering the scene.*

Plugin overrides user input methods such as keyPressEvent(), and mouseMoveEvent() for tasks such as camera control. Listing 5 shows an example of an Action Plugin that writes the number of objects in the scene every time the user presses a key.

```
// showobjects.h
#include "plugin.h"

class ShowObjects: public QObject, public Plugin
 {
     Q_PLUGIN // Uses Qt Plugin mechanism

 public:
    void keyPressEvent(QKeyEvent*);
 };

-------------------------------------------------------------

// showobjects.cpp
#include "showobjects.h"
void ShowObjects::keyPressEvent(QKeyEvent*);
{
    cout << "Objects: " << scene()->objects().size() << endl;
}
```

Listing 5: *A simple action plugin for writing the number of objects in the scene.*

3.2. Core library

We provide students with a simple C++ class library for dealing with 3D scenes. Students are instructed to use these classes when writing their own plugins. Here we just provide a brief overview of the main classes in the core library.

The Scene class represents a 3D scene as a collection of 3D objects. We considered providing students with a complete scene tree or scene graph organization of the objects, but for simplicity we decided to represent the default scene using a stl::vector of objects. Nonetheless, students might implement their own scene representation as part of a plugin.

The Object class represents a 3D object through a collection of vertices and a collection of faces. The Face class represents a (polygonal) face of a 3D object as a collection of vertex indices. Faces do not hold directly vertex coordinates; instead, faces store vertex indices, that is, integers indicating the position of each vertex in the vector of vertices associated with every 3D Object. The

Object class contains code to import 3D models (our current implementation reads Wavefront OBJ files). The Vertex class represents a vertex with a single attribute (vertex coordinates), since other attributes (normal vectors, texture coordinates) are created at runtime.

Simple math libraries for points, vectors and matrices are provided through well-documented Qt classes (QtPoint3D, QMatrix4x4). The Core library also provides a simple Camera class. The Plugin class includes methods to provide access to the current camera and scene.

3.3. GLWidget library

The GLWidget library contains a single class: GLWidget. The main purpose of this class, which is derived from QOpenGLWidget, is to provide a very basic implementation of the well-known methods initializeGL(), paintGL() and resizeGL().

GLWidget has little OpenGL rendering code. Instead, most of the GLWidget implementation is devoted to enable users to load an arbitrary number of plugins that provide the typical features of a 3D application: setting up the OpenGL state (for example, loading images to be used as textures), loading shaders, drawing the scene by issuing OpenGL rendering calls, and enabling some user interaction (object selection, camera control...). GLWidget does not implement any of the features above. These features must be provided through plugins.

The GLWidget class holds basically three different pieces of information: a scene, a camera, and a list of loaded plugins. Most of the code in GLWidget deals with invoking appropriate methods from the plugins. In a nutshell,

- Everytime a new plugin is loaded, its onPluginLoad() method is called.
- Everytime a new object is added to the scene, the onObjectAdd() method of all loaded plugins is invoked.
- The GLWidget::paintGL() method performs three basic steps: 1) call preFrame() for all plugins, 2) call paintGL() for the last plugin that overrides it, and 3) call postFrame() for all plugins.
- Mouse and keyboard events are propagated to all loaded plugins.

3.4. GL-Socket application

The GL-Socket application uses all the libraries above to let users open and view 3D models. For convenience, we provide students with a script that sets up a few environment variables and then calls the application. Environment variables set default paths (for 3D models, textures, plugins...) and plugins to be loaded automatically at startup.

Figure 1 shows a screenshot right after opening the GL-Socket application. By default, the application loads a Render Plugin providing single-pass rendering (Listing 4), a Draw Plugin based on VAOs, and an Action Plugin providing basic rotate-zoom-pan camera control. Notice that the GUI is minimalist, since the OpenGL window fills the application window. Plugins might provide their own GUI through Qt QWidget and QDialog classes, although most of our assignments rely on direct keyboard/mouse interaction.

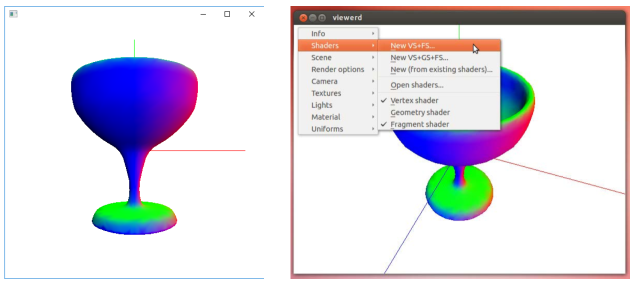


Figure 1: *GL-Socket application with default plugins (left) and shader development plugin (right).*

The application is extensible via plugins implementing the interface described above. Default plugins are loaded automatically at startup whereas new plugins can be loaded at runtime. Testing a plugin involves opening GL-Socket and loading the plugin to check its behavior.

3.5. Examples of Plugin Assignments

Here is a sample of the kind of assignments (summarized) that we include in our CG lab sessions:

- Write an Action Plugin that shows the current frame rate.
- Write an Action Plugin that writes the surface area of every object in the scene.
- Write an Effect Plugin that binds a shader program for per-vertex cartoon shading.
- Write an Effect Plugin that binds a shader program for per-fragment Phong shading.
- Write a Draw Plugin that draws the scene with flat shading, i.e. generating a VAO with per-corner (rather than per-vertex) normal vectors.
- Write a Draw Plugin that draws the scene including tangent, bitangent and normal attributes for each vertex.
- Write a Render Plugin implementing shadow mapping.
- Write a Render Plugin implementing deferred shading.

See project repository for complete plugin examples.

4. Shader development

For some assignments, we would like students to focus exclusively on GLSL. We thus provide a plugin for shader development, which is loaded automatically by invoking the proper GL-Socket script.

Our current version uses OpenGL 3.3 Core Profile shaders and thus supports Vertex Shaders (VS), Geometry Shaders (GS) and Fragment Shaders (FS). Shaders are edited using an external editor (configurable through environment variables) rather than an internal editor as in ShaderMaker [fCGVR]. This way students can choose their favorite editor for writing GLSL code, instead of imposing a less-featured custom editor.

Recall that OpenGL core profile does not provide predefined vertex attributes. In order to facilitate shader development, our default plugins provide typical per-vertex attributes (vertex, normal, color, texCoord, see Listing 6) and a collection of uniform variables for camera matrices, and lighting/material parameters (Listing 7).

```
layout (location = 0) in vec3 vertex;   // gl_Vertex.xyz
layout (location = 1) in vec3 normal;   // gl_Normal
layout (location = 2) in vec3 color;    // gl_Color.rgb
layout (location = 3) in vec2 texCoord; // gl_MultiTexCoord0.st
```

Listing 6: *Default attributes for Vertex Shaders in our framework. Comments show their analogous deprecated versions in the Compatibility Profile.*

```
uniform mat4  modelMatrix;
uniform mat4  viewMatrix;
uniform mat4  projectionMatrix;
uniform mat4  modelViewMatrix;
uniform mat4  modelViewProjectionMatrix;
uniform mat4  modelMatrixInverse;
...

uniform vec4  lightAmbient;
uniform vec4  lightDiffuse;
uniform vec4  lightSpecular;
uniform vec4  lightPosition;

uniform vec4  matAmbient;
uniform vec4  matDiffuse;
uniform vec4  matSpecular;
uniform float matShininess;

uniform vec3  boundingBoxMin;
uniform vec3  boundingBoxMax;
uniform vec2  mousePosition;

uniform float time;
```

Listing 7: *Uniform variables in our framework.*

The shader plugin provides a pop-up menu (Figure 1) with options for shader, scene, camera, light, material, and texture management.

When creating a new shader program (either VS+FS or VS+GS+FS, since GS are optional), the user just provides a shader name (e.g. *phong*) and the plugin automatically creates the required files (e.g. phong.vert, phong.frag) with default content (Listing 8). The plugin checks at every frame the time stamp of the current shader files (.vert, .geom, .frag), so that every time students save a shader, it is automatically loaded, compiled and bound by the application.

Compilation or link errors, if any, are both written to the standard output (terminal) and shown on a pop-up modal dialog. We also change the background color to red to clearly indicate that the model is not being rendered with user-provided shaders.

The plugin automatically detects which uniforms are actually used by the shaders (through glGetActiveUniform), and allows users to see and edit their values. The supplemental video shows a typical session for creating a shader using the plugin.

5. Self-evaluation tools

5.1. Overview

One of the key features of our framework is the support for student self-assessment through a one-click option that runs a test set. The test set contains a list of commands to automate model loading, texture loading, and for setting up specific camera, material, lighting and uniform values (see below).

```
// default.vert
#version 330 core
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;

out vec4 frontColor;
uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

void main() {
 vec3 N = normalize(normalMatrix * normal);
 frontColor = vec4(color,1.0) * N.z;
 gl_Position = modelViewProjectionMatrix * vec4(vertex, 1.0);
}

------------------------------------------------------------

// default.frag
#version 330 core
in  vec4  frontColor;
out vec4 fragColor;

void main()
{
  fragColor = frontColor;
}
```

Listing 8: *Default code for VS+FS shader programs.*

Figure 2 illustrates the testing process. The *Run test* option automatically loads a 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. Students are reminded that passing a test does not ensure correctness nor efficiency of the solution. Conversely, minor per-pixel differences might be attributed to factors not invalidating correctness, such as hardware or driver configuration differences, rounding errors... In practice, visual comparison of the student's output against the reference images has proved to be a powerful tool to quickly detect errors in a number of assignments, from shading to animation going through texture mapping. A further application is to identify subtle changes between shaders, e.g. per-vertex vs per-fragment lighting.

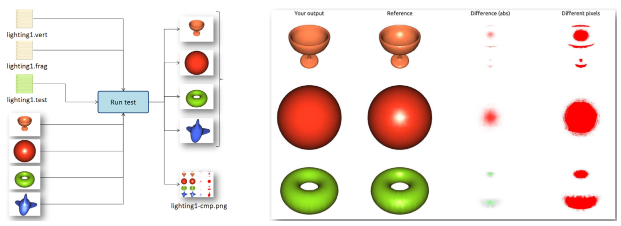


Figure 2: *Overview of the shader testing process (left) and sample output from it (right).*

5.2. Test set language

The shader development plugin supports a simple command language for setting up the window, scene, camera, and so on. Here we just provide an example of a test file (Listing 9). See project repository for a complete description. Test files are supposed to be written by instructors, although students can read public ones to check e.g. which uniform values or texture images are being tested. Besides self-assessment, test files have also proven to be useful to quickly setup an environment for testing specific shaders; for example, the test file can load specific images and 3D models with a single click, thus automatizing this task and speeding up moving from one assignment to another.

Writing new test files takes only about 1-2 minutes, starting from existing templates.

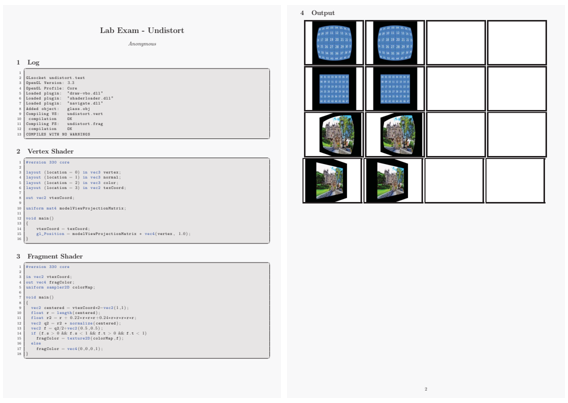
```
resize 800x600                    matAmbient 1.0 0.5 0.3
clearColor 1 1 1                  matDiffuse 1.0 0.5 0.3
                                  matSpecular 1.0 1.0 1.0
loadObject glass.obj              matShininess 64
camera preset1                    bool attenuation false

lightAmbient 0.1 0.1 0.1          grab 1
lightDiffuse 1 1 1                loadObject sphere.obj
lightSpecular 1 1 1               grab 2
lightPositionEye 0 0 0 1          ...
```

Listing 9: *Partial example of a test file for a lighting shader. The grab keyword saves an image with the current settings.*

6. Correction aid for instructors

A great advantage of our system is the easiness of running private test files (often distinct from the public ones) using a script. We developed a script (not included in the project repository, as it contains code specific to the Faculty Online Platform) that automatically collects all student submissions, tests them against a validation set, and creates a PDF report with all the results, including student information, source code, compile and link logs, and a comparison image similar to that in Figure 2. Figure 3 shows an anonymized example of such a report.



Lab Exam - Undistort

Anonymous

1 Log

```
GLsocket undistort.test
OpenGL Version: 3.3
OpenGL Profile: Core
Loaded plugin:  "draw-vbo.dll"
Loaded plugin:  "shaderloader.dll"
Loaded plugin:  "navigate.dll"
Added object:   glass.obj
Compiling VS:   undistort.vert
 compilation    OK
Compiling FS:   undistort.frag
 compilation    OK
COMPILES WITH NO WARNINGS
```

2 Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

out vec2 vtexCoord;

uniform mat4 modelViewProjectionMatrix;

void main()
{
    vtexCoord = texCoord;
    gl_Position = modelViewProjectionMatrix * vec4(vertex, 1.0);
}
```

3 Fragment Shader

```
#version 330 core

in vec2 vtexCoord;
out vec4 fragColor;
uniform sampler2D colorMap;

void main()
{
  vec2 centered = vtexCoord*2-vec2(1,1);
  float r = length(centered);
  float r2 = r + 0.22*r*r*r+0.24*r*r*r*r*r;
  vec2 q2 = r2 * normalize(centered);
  vec2 f = q2/2+vec2(0.5,0.5);
  if (f.s > 0 && f.s < 1 && f.t > 0 && f.t < 1)
    fragColor = texture2D(colorMap,f);
  else
    fragColor = vec4(0,0,0,1);
}
```

4 Output

2

Figure 3: *Sample pages of the report generated automatically from student submissions. The report contains student ID, compilation logs, source code and graphical comparison. In this example the output is correct.*

7. Results and discussion

During these years, we received very positive feedback both from students and teachers (7 teachers in the last four years). We collected results from the course during the last three years, and compared them with those from the other compulsory courses in the same semester. Figure 4 compares success rates for these courses. A one-way ANOVA analysis for independent samples on the success rate shown in Figure 4 reveals that differences among success rates for the different courses are statistically significant (p<0.01). Tukey HSD Test shows that average success rate for our course (G) is significantly higher than those of two courses (p<0.05), but not significant with respect to the other three courses. Although these very good results can be attributed to many factors, we believe the lab sessions have contributed to this success.

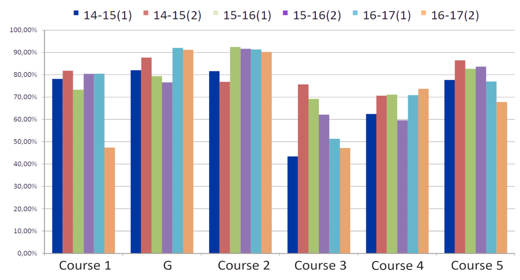


Figure 4: *Pass rate for our course (G) and the rest of compulsory courses in same semester, for the last three years: Algorithmics, Artificial Intelligence, Logic in CS, Programming Languages and Theory of Computation.*

We also observed a significant reduction of marking times with respect to our old, non-plugin-based framework. Since the PDF report with all graphical comparisons on the validation test is generated automatically, correction time is limited to reading the code after gaining a strong cue on potential problems from the output images. In many cases, specific parts of the code are checked just to confirm a first hypothesis (e.g. specular term is missing) that becomes evident after seeing the composite images. Current marking times are about 2-3 min per exercise, vs. 3-10 min we got with the old framework version.

Teachers often write corrections as PDF annotations, thus facilitating exam revisions, since students can see both the output on the private tests, and the evaluator comments and suggestions.

Exercise independence increases correction fairness. We assign correction tasks on an exercise basis (the same exercise is marked by the same person for all students) rather than on a group basis (each teacher marking all exercises from her group). This minimizes subjective deviations, and guarantees more homogeneous criteria for each exercise across students. It also saves coordination time since there is no need for an agreement on fine-grained criteria among evaluators; general evaluation criteria are thus much shorter and told to students at the beginning of the course. Exercise independence also minimizes discussion among teachers when designing a lab exam; in the last years discussions have been restricted to specific exercises, which are simpler to fine-tune or replace than a large exercise with inter-dependent parts. We also believe that this causes marks to better reflect student progress, when compared with longer assignments with strong inter-dependencies (where failing to complete a part due to a specific missing skill might prevent students from completing the rest, and may exacerbate the problem of loss of motivation).

An additional advantage is that exercises for lab exams are ready to be included in the exercise collection for upcoming students. This encourages teachers to write elaborate, engaging statements for lab exams.

Figure 5: *Students using GL-Socket during a lab exam.*

8. Conclusions and future work

We have described a framework for teaching OpenGL and GLSL in the context of introductory CG courses. Learning Computer Graphics techniques through plugin development has multiple advantages: allows for very focused, self-contained, independent exercises, enforces modularity and facilitates code reusability.

The framework (and the way we planned the lab sessions) have some limitations though. It can be argued that plugin-based sessions can be too focused on narrow topics, and students might not develop skills for designing larger software projects were some decisions require a global view of the user requirements. In our case, other courses do address the problem of large-scale Software Engineering, and our experience with tens of Final Degree Projects has shown that students have no problems to move from a narrow, plugin-based development to wide, global application development.

Although our framework has (intentionally) a minimalist GUI, students can create their own GUI for the plugins. Again, students are not developing global skills for designing a fully-featured application GUI, but this should not be a problem as long as the topic is covered in a preceding course.

Despite students being instructed not to take visual comparisons against reference images as guarantee of correctness or incorrectness, we would like to minimize such differences for valid solutions, specially when these are due to different hardware configurations (e.g. anti-aliasing forced by driver). We plan to setup a server for running tests online, to guarantee that both reference and user-provided shaders are executed in the same hardware or virtual machine. The GL-Socket application will take care of all client-server communications so that running tests will continue to require a single click. Finally, we are also working on complementing the composite graphical comparison by an interactive side-to-side or alpha-blended comparison (using e.g. WebGL or streaming) which will detect automatically differences and choose parameter values and camera views to highlight them. This would be specially useful e.g. for animation shaders.

Repository

Source code for our framework (application, libraries and example plugins) is available in the following Git repository: https://gitrepos.virvig.eu/docencia/glarena.

Acknowledgements

We would like to thank Cristina Raluca for her helpful suggestions on interactive comparisons between student/reference outputs. This work has been partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER Grant TIN2017-88515-C2-1-R.

References

[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. 3

[fCGVR] FOR COMPUTER GRAPHICS I., VIRTUAL REALITY U. B.: Shader maker. cgvr.cs.uni-bremen.de/teaching/shader_maker/index.shtml. Accessed: 2018-01-12. 2, 5

[FP13] FAIRÉN M., PELECHANO N.: Introductory graphics for very diverse audiences. In *Eurographics 2013 - Education Papers, Girona, Spain, May 6-10, 2013* (2013), pp. 9–10. 2

[FWW12] FINK H., WEBER T., WIMMER M.: Teaching a modern graphics pipeline using a shader-based software renderer. In *Eurographics 2012 – Education Papers* (2012), pp. 73–80. 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

[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

[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

[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

[RME14] REINA G., MÜLLER T., ERTL T.: Incorporating modern opengl into computer graphics education. *IEEE Computer Graphics and Applications 34*, 4 (2014), 16–21. 3

[RRP00] RAAB J., RASALA R., PROULX V. K.: Pedagogical power tools for teaching java. *SIGCSE Bull. 32*, 3 (2000), 156–159. 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