glGA: an OpenGL Geometric Application framework for a modern, shader-based computer graphics curriculum

G. Papagiannakis[1,2], P. Papanikoalou[1,2], E. Greassidou[1] and P. Trahanias[1,2]

[1]University of Crete, Computer Science Department, Voutes Univertsity Campus, 70013, Heraklion, Greece
[2]Foundation for Research and Technology Hellas, 100 N. Plastira Str., 70013, Heraklion, Greece

Abstract

This paper presents the open-source glGA (Opengl Geometric Application) framework, a lightweight, shader-based, comprehensive and easy to understand computer graphics (CG) teaching C++ system that is used for educational purposes, with emphasis on modern graphics and GPU application programming. This framework with the accompanying examples and assignments has been employed in the last three Semesters in two different courses at the Computer Science Department of the University of Crete, Greece. It encompasses four basic Examples and six Sample Assignments for computer graphics educational purposes that support all major desktop and mobile platforms, such as Windows, Linux, MacOSX and iOS using the same code base. We argue about the extensibility of this system, referring to an outstanding postgraduate project built on top of glGA for the creation of an Augmented Reality Environment, in which life-size, virtual characters exist in a marker-less real scene. Subsequently, we present the learning results of the adoption of this CG framework by both undergraduate and postgraduate university courses as far as the success rate and student grasp of major, modern, shader-based CG topics is concerned. Finally, we summarize the novel educative features that are implemented in glGA, in comparison with other systems, as a medium for improving the teaching of modern CG and GPU application programming.

Categories and Subject Descriptors (according to ACM CCS): K.3.2 [Computers and Education]: Computer and Information Science Education—Computer Science Education

1. Introduction

Computer Graphics (CG) is a topic that not only requires from students a background in science, engineering and basic mathematics, such as linear algebra, but it also demands sufficient C/C++ application programming skills. The major changes in graphics h/w over the past few years have led to significant changes in the new ways CG s/w is been written as well as taught in university courses. CG has been part of the 4-year academic curriculum of the computer science department at the University of Crete during the last 20 years. It has also been a major topic of research, which had also led to the organization of CGI in 2004 and Eurographics in 2008 by the University of Crete. Over these years we had experimented with various methods of teaching computer graphics [AS12]: from the algorithmic approach, to the survey approach and recently the programming approach. We have adopted and extended the modern, shader-based OpenGL (GL) approach from [AS11] and opted for a completely revised undergraduate and graduate course curriculum in CG, focusing on the recent, exciting GPU-based languages and APIs.

The OpenGL Geometric Application (glGA) is a C++ CG framework that we have developed based on [AS12], [D13], [M13] and [TMO13], in order to help the students face most of the novice CG difficulties, while understanding at the same time the concepts required for a newcomer to accomplish engaging CG course assignments (e.g. from Blinn-Phong lighting to shadow and normal mapping and from basic skinned character animation to augmented reality simulation). This framework has been used in both offered courses at undergraduate as well as postgraduate level. The first undergraduate course is CS358, which covers basic notions in computer graphics [AS12] [TPP*07] via a 13-week and three, two-hour lectures per week. Not only the students learn about geometrical transformations but they also work via glGA on modern, engaging assignments implementing them in modern OpenGL and the GL shading language (GLSL) [RLG*10]. With the help of simple GUI widgets that they develop, they can comprehend all shader parameters from Blinn-Phong lighting, to material colors and camera positions, directions and skinned character animation parameters. Three assignments in two week intervals were assigned and an optional final bonus one. The second course that we worked with glGA is

DOI: 10.2312/eged.20141026

the postgraduate CS553, which extends previously introduced CG topics in modeling, rendering and animation. In CS553, also a 13-week and three, two-hour lectures per week schedule was followed. Here as well, each one of the attendees had to complete a set of assignments, as well as a final project that is based on latest publications in the above areas. Both of these courses demanded significant engagement and practical work by the students in order to be completed successfully. It was stressed to both students that the emphasis was on portfolio building or strengthening their understanding on CG for their potential industrial IT or academic careers. The educational approach followed was of project-based learning and learning by design as already highlighted in [R13] and [KC*03].

Before the introduction of glGA, both the attendees and the instructor faced a number of difficulties, since shader-based OpenGL [AS11] was a totally new addition at the CS curriculum of the University of Crete and furthermore it had to be presented to students with no prior particular experience of C++, except only C or Java. It has been proven over the semesters of its introduction, that glGA can empower student knowledge in multiplatform, shader-based CG development, from simple geometrical and lighting examples to textured, rigged and animated virtual characters with 3D scene GUI interactive parameter-altering capabilities. The glGA framework has been tested in the last three semesters at Computer Science Department of University of Crete (winter 2012, spring 2013 and the winter 2013 semester).

In this paper we will discuss exhaustively all of the educative aspects of glGA as an alternative for shader-based CG educational tool with a new insight that increases student learning: *based on the "keep it simple" programming rule, when students are introduced to new, modern CG concepts in one semester, they can simply and quickly implement and understand them through a thin but powerful layered architecture of open-source CG libraries that a) hide non-graphics related tasks b) expose the heavyweight GPU programming tasks.* Thus they avoid the steep learning curve or advanced C++ and software engineering concepts that other complete CG frameworks feature (e.g. NGL, Ogre3D or OpenScenegraph) or 'black-box' commercial engines (e.g. Unity, Unreal). To highlight the above, we also provide basic examples and sample assignments that we employed so that instructors can take them straight into their classrooms. Finally we present an example of an outstanding student project based on glGA on life-size, marker-less augmented reality (AR) character simulation.

2. The glGA Framework Description

glGA is a modern, shader-based CG simulation framework designed to offer simplicity to students that want to explore CG but also empower them so that at a later stage they can easily migrate to a modern game-engine, having understood underlying GPU-based application programming concepts. Not only it contains four introductory examples for the students, but it also contains six adjustable sample applications that can be used by the instructor as assignments. All open-source code samples that are implemented in glGA run in all major 3 desktop platforms: Windows, Linux and OSX with no modifications other than the definition of the platform in a single #define. The exact same code base that is used in glGA can run in any of these platforms, producing the same results as it is based on modern OpenGL. Furthermore, the same glGA framework and its examples can also be executed as they are, in mobile architectures such as iOS (Android is also currently been implemented), accompanied with an Objective C/C++ wrapper, due to the requirements of iOS.

One of our primary aims was that the novice students would not be discouraged at the beginning by the burst of CG application development information from books & net tutorials. Our main educative testing field was the undergraduate CS358 course where basic CG concepts were taught. Our primary objective was that every student that undertook the CG development tasks would be able to build from scratch, colloquial but engaging, shader-based 3D real-time CG applications. These applications would be based on firm knowledge given by the course lectures on topics such as: window initialization, the GL shading language, linear algebra matrix transformations, window events, modeling, materials and lighting, texturing, 3D model loading, and even more advanced topics such as linear blend skinned character animation and GUI-based scene and scenegraph manipulation in real-time. In the following sections (2.1 and 2.2) we present the basic examples that come as part of the standard glGA framework as well as a sample of the different assignments (1,2,3) and (4,5,6) that were presented to the students of the undergraduate (CS358) course during two semesters. Besides the examples in glGA we present basic assignments in pairs as they cover topics of equivalent increasing complexity. The sample lecture course slides (assuming as course textbook the one from [TPP*07]) are soon to be released as open course notes under the University of Crete open lectures and regarding the assignment material, instructors are welcome to contact the main author.

2.1 Examples

BasicWindow: This is the first example (Figure 1) that someone has to understand in order to have a basic start in the CG field. It shows how a student can initialize a window employing the well-known open-source GLFW window toolkit and create a graphics context using either the Opengl 2.1 compatibility profile or OpenGL 3.2 core profile. We also employed the GLEW open-source s/w library to load the GL extensions and to retrieve support information on our hardware. In the main loop, a very simple empty scene rendering is taking place by clearing the color buffer with a basic RGB color.

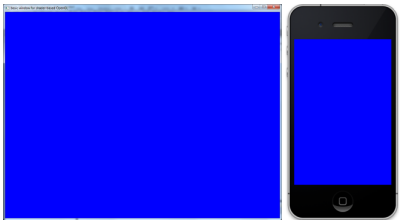

Figure 1: *BasicWindow glGA example in Windows (left) and iOS (right) platforms.*

BasicTriangle: In this second basic example, we introduce the use of GLSL with simple, pass-through shaders in order for someone to start grasping the modern programmable GPU rendering pipeline with GLSL. At the beginning of this example, students are taught how to create and handle Vertex Array Objects (VAOs) and Vertex Buffer Objects (VBOs). The VBO is filled with vertex positions that will be later used in the vertex shader as attributes. As for the rendering part and to allow for simplicity, only a single triangle is drawn in the upper right corner of the window, as shown below.

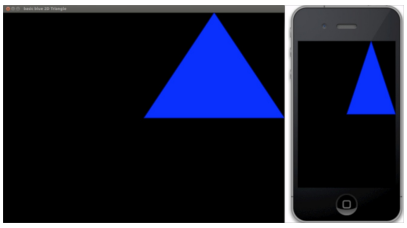

Figure 2: *BasicTriangle glGA example in Linux(left) and iOS (right) platforms.*

BasicCube: The only conceptual difference with the previous example is that a colored-per-vertex, 3D cube is rendered instead of a triangle, via basic GL camera and projection matrices. By doing so, the students learn about the basic primitives and how to use them in order to later compose more complex geometries. Other than positions, we introduce colors and normals as vertex data attributes in VBOs. Finally, they learn how to package positions, colors and normals all together in VBOs and VAOs for multiple primitives and pass these attributes to the shaders.

BasicCubeGUI: A GUI Toolkit is a very important part of any CG application, as it allows the run-time manipulation of the 3D scene and via various parameter-tweaking enhancing the basic understanding of their real value. Thus, in this example we focus on the introduction of AntTweakBar, an extremely simple, light and easy-to-use for novices GUI Toolkit built in C/C++ that can be employed in any GL application. It can support different types of adjustable variables through simple widgets such as buttons, sliders, combo boxes, etc. Not only it gives the students the capability to create a basic interaction between user and application, but it also helps them significantly with run-time debugging of their GL shader-based application. One of the main adopting reasons of this open-source C++ GUI library is due to its simplicity. In the assignments that follow, the students had to employ AntTweakBar in order to parameterize their GLSL parameters such as matrices for lighting, scene transformations and GL camera projections.

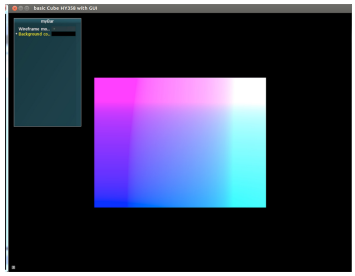

Figure 3: *BasicCubeGui example in Linux*

The skeleton-code of each glGA example is particularly created as simple as possible (single .cpp file). In the Appendix of this paper, we present the code of the basicCubeGUI example and illustrated in the figure above. As shown there, the first task is to define the geometry positions and colors. For example, to render a cube, as in [AS12], we need to define 8 positions and 8 colors, which correspond to one position and one color for each vertex of the cube. The most important parts of each example and assignment are the *init()* and *display()* functions, which handle the geometry initialization and rendering respectively, and are called within the *main()*. Thus, in the basicCubeGUI example the init() function has to perform the following tasks:

1. Load and compile the shaders: The vertex and fragment GLSL shader files are given as parameters in the *LoadShaders()* function of the *glGAHelper.h* component. That function reads the files and stores the source code in string variables. After their compilation, they are attached to a GL shader program object, which will be employed by the scene for rendering purposes.
2. Generate a Vertex Array Object (VAO) and a Vertex Buffer Object (VBO): Usually a VAO can contain multiple VBOs. To keep this example simple we prefer to use only one from each category. So first we create the vertex data that are stored within the VBO as positions and colors. This procedure is taken care of by the *colorcube()* function (sample shown in Appendix) that creates 6 quads calling the quad function. Each of these structures is then stored in a different sub region arrays within the same VBO.

3. Create and connect the attributes that will use the VBO data in shaders: Having stored the required rendering data in the VBO, the final task is to define the attributes that will be associated and connected with the colors and positions provided in the scene description. Hence we create two shader parameters with the names *vPosition* and *vColor* that are used in the Vertex Shader and show how to connect them to the respective VBO arrays.

2.2 glGA Sample Assignments

All of the previous examples are given to the students within the basic glGA distribution via the course e-learning web-site, in order to help them in their initial steps to CG application development. In this section we describe some of the assignments that the students had to accomplish in the undergraduate CG course (CS358). The first three were given in 2012 (1, 2, 3) and the final three (4, 5, 6) in 2013. We present them in pairs (1-4, 2-5, 3-6) as they progress with increasing but similar difficulty, following the progression of the respective oral lectures given by the instructor and occasional tutorial by the course TAs:

Assignment1 & Assignment4: In these first assignments, the students have to learn to use uniform parameters and matrix transformations in order to apply the vertex processing on basic 3D objects and convert them from object space to window coordinates. To do so, they have to pass the Model, View and Projection matrices in a vertex shader and multiply them correctly with them vertex positions. In this manner they can grasp interactively the notions of the virtual camera that allows them to navigate around the 3D scene. For the creation of the GL View and Projection matrix representations, the students use the open-source GLM library that provides them with all the needed helper mathematical structures and functions, such as: *lookAt(...), perspective(...), vec3, mat4* etc. . The final multiplication of the matrices is taking place in the vertex shader, in the correct order for column-vectors and the GL column-major matrices. The major difference between Assignments 1 and 4, was that in the later one, the use of the AntTweakBar GUI lib was asked in order to handle the view matrix and other parameters in real-time.

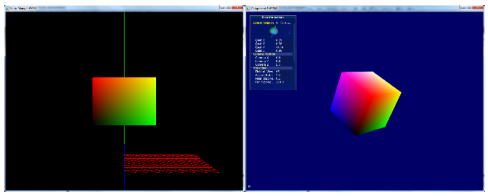

Figure 4: *Assignment 1 (left) and Assignment 4 (right).*

Assignment2 & Assignment5: In these assignments, students learned how to implement the basic blinn-phong illumination model, in order to add local lighting properties on the rendered objects. A number of built-in functions were used in the shaders in order to achieve the expected results. The difference in these two assignments is that assignment 2 uses the cube from the previous assignment, while assignment 5 asks for a user-loaded 3D model on which the algorithm is applied. That means in assignment 5 students had to use the provided helper *glGAMesh* component that allows for 3D model loading, abstracting the open-source, C++ Assimp s/w library for various 3D-formats asset loading.

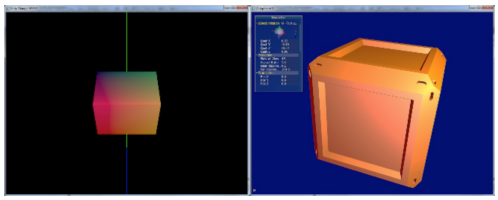

Figure 5: *Assignment 2 (left) and Assignment 5 (right).*

Assignment3 & Assignment6 with Optional Bonus Assignment: In this final round of assignments the student is introduced to texturing and skinned characters. Assignment 3 implements basic texturing on a cube, while Assignment 6 applies it on an animated-skinned or static, external 3D model. To support textures, students have to activate the Texture class in the glGAHelper component by defining the *#define USE_MAGICK*. The loading of skinned models as well as their rendering is implemented in the *glGARigMesh* helper class. The vertex shader of assignment 6 also contains the bone matrix multiplications with the vertex positions necessary for linear blend skinning implementation.

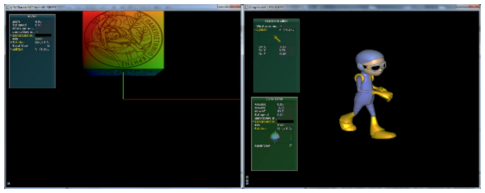

Figure 6: *Assignment 3 (left) and Assignment 6 (right).*

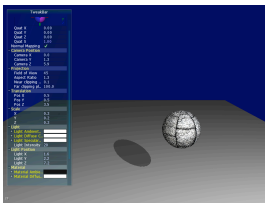

Figure 7: *Normal & shadow mapping bonus assignment.*

In the figure above, the bonus, optional assignment is illustrated. Based on the lextures on hard shadows and texturing, students were asked to implement in glGA the normal and shadow-mapping algorithms for any 3D model loaded with the *glGAMesh or glGARigMesh* helper classes.

2.3 Utility functions and classes provided

In order to help the students to avoid some elementary tasks required for the assignments (e.g. shader loading and compilation, texture handling etc.) or abstract the use of external, open-source libraries (e.g. Assimp) we have created a minimal set of utility functions and classes that implement these tasks. Thus, we provide them with only 3 small utility classes: glGAHelper, glGAStaticMesh and glGARigMesh. These few classes and associated functions are implemented in C/C++ and illustrated in Figure 8.

2.3.1 Shader Loader

The function *LoadShaders()* simplifies loading and compiling of shaders: First, it reads the vertex and fragment shader files that are given as parameters and stores them in string variables. Then, both shader source codes are compiled and checked for errors. In the end, they are linked to the program that will be used for rendering.

2.3.2 Texture class

This small utility class simplifies external image loading form various formats and handles the texture object creation in OpenGL (generating, binding, initializing with texture environment parameters).

2.3.3 Model Loader

This functionality is connected to the final glGAMesh and glGARigMesh classes. The first one is used for static 3D model loading, while the second one for skinned and animated. In order to render a model, students need to load the mesh from the external resources using the respective class methods. These 2 classes parse the model files and create a geometry tree corresponding to the scene retrieved from the file. After that the student can retrieve in C++ the positions, normals and texture coordinates which then can be used in vertex buffer objects. In case the model is skinned, glGARigMesh class creates also a bone data structure that contains information about all of the bones used from each vertex.

2.3.4 Real-Time Animation

As we have mentioned, glGA supports from simple 3D models up to animated, skinned charcaters. In order to help the students visualize the externally rigged virtual characters (e.g. Collada or MD5 models) glGA provides the functionality required to parse the bone tree in real-time and retrieve the transformation matrix from each one of the joints. These matrices are then passed as uniform and vertex attribute parameters to the vertex shader.

2.3.5 glGA external, open-source dependencies and framework overall architecture

In the following table and figure below we provide the clear and concise overview of the glGA external dependencies. Six well-known, well-documented and actively supported open-source libraries are utilized under the hood of glGA. The basic glGA application contains in a single file, only the *main(), init()* and *display()* C++ methods, so that a student in one parse can understand the existing examples and also create the new functionality required in the assignments.

GLFW	Window creation and OpenGL context handling
GLEW	OpenGL extension loading library
GLM	Template-based, C++ mathematics library similar to GLSL built-in math functions and types
AntTweakBar	GUI toolkit for widget creation and real-time shader/scene parameter handling based on user input
Image Magick	Multi-format Image/Texture loading
Assimp	Loading of static or skinned models

Table 1: glGA *Open-source, external s/w libraries*

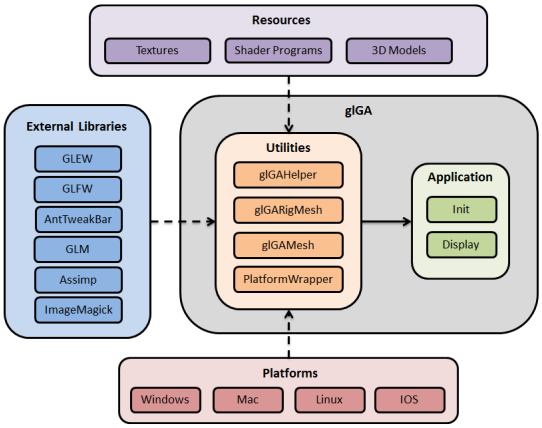

Figure 8: *The glGA overall framework s/w architecture*

2.4 Platforms Supported

We have developed glGA in such a way so that all of its examples and sample assignments can run in any of the standard desktop and mobile platforms: Windows, Linux, OSX and iOS. In order for all of those (10 in total) applications to be supported, we had to create a short PlatformWrapper component that handles the platform specific functionality.

In addition to the desktop platforms of Windows, Linux and OSX, the glGA examples are also supported in the mobile iOS platform. Here is where the PlatformWrapper is also employed not only due to the header files but also due to the different OpenGLES methods and calls (instead

of standard OpenGL). An additional difference between desktop and mobile platforms is the way that external assets (e.g. textures, 3D models etc.) are loaded by the application. E.g. iOS uses bundles, while Windows, Linux and Mac retrieve the assets directly from the disk with either relative or absolute paths. Other than these, the current time retrieval is also different from desktop to mobile. E.g. it is essential during character animation, where we have to recalculate the matrix transformations based on the time passed since the animation started. As we have already, mentioned the code of the examples and assignments is in portable, standard C++, thus a standard C++ compiler (e.g. gcc, LLVM, Intel or Microsoft) is mandatory to be employed. In glGA, we have also included some IDE project files for certain platforms already set up and ready to be built and executed. The project files that exist in glGA are for Visual Studio 2010 for Windows and Xcode 5.0.2 for Mac and iOS, while we also provide the basic gcc/g++ makefiles for Linux. The only modification required is to define the specific platform on top of the PlatformWrapper header file. Of course, other IDEs can also be used as long as they support standard C++.

2.5 AR glGA as an outstanding student project

As part of the learning context of the graduate course in Computer Graphics at the University of Crete, is to encourage individual students to bring their creativity in new CG applications via project-based learning [R13]. Thus a creative post-graduate student chose as his course project to explore the capabilities of the METAIO SDK (http://www.metaio.com/sdk/) and investigate how to integrate it is to an application of glGA, based on our previous works in AR characters [VLP*04] and [PMT07] with the results shown in Figure 9.

Figure 9: *Marker-less AR of a Virtual Character (left) along with a Real Character (right) as rendered in the same scene in an iPod 4.*

The task was to create an iOS mobile marker-less Augmented Reality (AR) application that embeds the glGA skeleton-based deformable animated virtual human characters in a real scene. The vision-based camera tracking functionality was provided by the METAIO SDK.

2.6 Results & Discussion

This glGA framework has been tested in the last three semesters at the Computer Science Department of the University of Crete. Specifically in 2012, 50 students registered for CS358, while 32 participated in the course assignments (not mandatory as the total mark was 70% from the final exams and 30% from the assignments). With average score 1.9 out of 3 in assignments and 70% success rate in the finals, we can say that our effort to teach shader-based Computer Graphics to students with no previous experience has been successful as all students that participated in the assignments could write moderately complex CG applications using shaders, by the middle of the course, as shown in Figure 10.

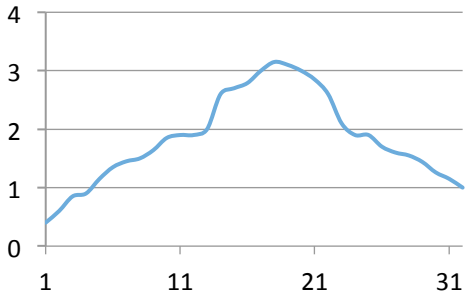

Figure 10: *The grades of those that participated in Assignments. Vertical Axis refers to grade while the horizontal refers to students (32 students participated total). Average score 1.9 out of 3, along with 0.15 for bonus tasks.*

However, novice students (especially those never exposed to third-party open-source libraries) faced difficulties setting up their build environment. For that reason the instructor with the help of the three course TAs carried out the following necessary actions: a) setup an online forum for questions/answers on the course e-learning site, b) during each lecture devoted time on live Q&A session and c) recorded video tutorials on how to build certain open-source libraries from scratch (e.g. Assimp, ImageMagick) in case the students could not use the already provided ones, due to platform issues (e.g. 64-bit vs 32-bit etc.). It has to be noted that most students have used before Linux and few of them OSX for programming but a large number of them wanted to try Visual Studio IDE and Windows. Hence the TAs had also to strive with the fact that many students needed help on that migration. However, students did not receive any extra help apart from the lectures, the lecture notes and the above Q&A sessions online or during the course (i.e. no extra helper code was provided).

Online code and tutorials [TMO13], [D13], [M13] are valuable but not always helpful to appreciate the larger context as they tend to solve only particular CG or platform specific issues, while missing always some features e.g. mobile platform integration, GUI toolkit functionality etc.

or rely on complex s/w engineering classes and external libraries that the student has first to understand before proceed with the CG task. Other similar, exceptional previous works on shader-based teaching have also to be mentioned, such as [FWW12], [AS12], [BC07], and [OZC*05] but they follow different approaches as they are either based on s/w rasterizers, or focus on only surface material shader effects or do not contain valuable features such as 3D asset loading, character animation and GUI widgets. During the provided CG courses, students were encouraged to explore the whole CG pipeline of modeling, rendering, animation and a large number of them responded enthusiastically: i.e. they modeled in Google Sketchup™ their 3D models, textured them and exported them in the Collada™ format. Subsequently used glGA to load these models and use them in their assignments for rendering and animation. A small subset of students they also voluntarily created small 3D games only using glGA and what they have learned during the course.

2.7 Conclusions & Future Work

It is undoubtedly hard for someone new as an undergraduate student to enter the CG field. And if he/she has to program a complex graphics application and at the same time learn a new language (GL Shading Language) is even harder. This is why many CG educators still teach the deprecated, fixed-function OpenGL or rely on far more complex game engines or other toolkits that hide difficult notions from the end user. However, they often end up either abstracting too much (black-box effect) or pose extensive s/w engineering demands. We have managed to create a simple, thin-layer, open-source framework that curbs the CG complexity by easily allowing students to grasp basic but exciting modern CG principles. In the meantime it prepares them with all the necessary knowledge through the hands-on assignments to later take on larger CG scenegraph frameworks or game engines. The provided series of glGA examples and assignments that can be used for educational purposes have proven to be able to help students comprehend the ways of creating modern 3D GPU-based applications.

glGA will continue to be supported and evolve (include geometry and tessellation shaders) with more examples and sample assignments as well as more documentation. We also want to support more Integrated Development Environments, specifically for Linux, as well as more platforms such as Android (already under-way). A CG class teaches far more than a good API or framework but a good framework empowers the CG educator to teach key topics in computer graphics in an easier and more engaging way. glGA aims to hide the non-graphics related programming tasks and expose the students directly to the underlying heavyweight art of GPU-based application development and algorithm implementation. glGA can be accessed freely at: http://george.papagiannakis.org/?page_id=513

References

[AS12] ANGEL, E. AND SHREINER, D. 2012. Interactive computer graphics: a top-down approach with Shader-based OpenGL (6th Edition). Addison-Wesley ISBN 13:978-0-13-254523-5, 1–778

[AS11] ANGEL, E. AND SHREINER, D. 2011. Teaching a Shader-Based Introduction to Computer Graphics. Computer Graphics and Applications, IEEE 31, 2, 9–13.

[BC07] BAILEY, M. AND CUNNINGHAM, S. 2007. A hands-on environment for teaching GPU programming. 39, 1, 254–258.

[D13] TOM DALLING (2013), Modern OpenGL. Retrieved from http://tomdalling.com

[FWW12] FINK, H., WEBER, T., AND WIMMER, M. 2012. Teaching a modern graphics pipeline using a shader-based software renderer. Computers & Graphics.

[KC*03] KOLODNER, J. L., CAMP, P. J., CRISMOND, D., FASSE, B., GRAY, J., HOLBROOK, J. AND RYAN, M., Problem-based learning meets case-based reasoning in the middle-school science classroom: Putting learning by de- sign (tm) into practice. The journal of the learning sciences, 12(4), 495-547.

[M13] ETAY MEIRI (2013), Modern OpenGL Tutorials. Retrieved from http://ogldev.atspace.co.uk/index.html

[OZC*05] OWEN, G.S., ZHU, Y., CHASTINE, J., AND PAYNE, B.R. 2005. Teaching programmable shaders: lightweight versus heavyweight approach. SIGGRAPH '05: SIGGRAPH 2005 Educators program.

[PMT07] PAPAGIANNAKIS, G. AND MAGNENAT-THALMANN, N. 2007. Mobile Augmented Heritage: Enabling Human Life in ancient Pompeii. The International Journal of Architectural Computing, Multi-Science Publishing 5, 2, 395–415.

[R13] ROMERO, M. 2013. Project-Based Learning of Advanced Computer Graphics and Interaction. Eurographics 2013-Education Papers.

[RLG*10] ROST, RJ, LICEA-KANE, B., GINSBURG, D., KESSENICH, J., LICHTENBELT, B., MALAN, V., WEIBLEN, M., OpenGL Shading Language - Third Edition, Addison-Wesley, 2010

[TMO13] Tutorials For Modern OpenGL (3.3+) (2013). Retrieved from http://www.opengl-tutorial.org

[TPP*07] THEOHARIS, T, PAPAIOANNOU, G, PLATIS, N, PATRIKALAKIS, N., Graphics and Visualization: Principles & Algorithms, AK Peters, 2007

[VLP*04] VACCHETTI, L., LEPETIT, V., PAPAGIANNAKIS, G., M PONDER, P FUA, D THALMANN, MAGNENAT-THALMANN, N., 2004. Stable real-time AR framework for training and planning in industrial environments. Virtual Reality and Augmented Reality Applications in Manufacturing, Ong, S. K., Nee, A.Y.C. (eds), ISBN: 1-85233-796-4, Springer-Verlag, 125–142.

Appendix

BasicCubeGUI c++ code (important parts, based on [AS12]):

```
const       int NumVertices = 36;
typedef     glm::vec4 point4;
point4      points[NumVertices];
color4      colors[NumVertices];
point4      vertex_positions[8] = {
      point4( -0.5, -0.5,  0.5, 1.0 ),
      point4( -0.5,  0.5,  0.5, 1.0 ),
      point4(  0.5,  0.5,  0.5, 1.0 ),
      point4(  0.5, -0.5,  0.5, 1.0 ),
      point4( -0.5, -0.5, -0.5, 1.0 ),
      point4( -0.5,  0.5, -0.5, 1.0 ),
      point4(  0.5,  0.5, -0.5, 1.0 ),
      point4(  0.5, -0.5, -0.5, 1.0 )
};
color4 vertex_colors[8] = {
      color4( 0.0, 0.0, 0.0, 1.0 ),  // black
      color4( 1.0, 0.0, 0.0, 1.0 ),  // red
      color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
      color4( 0.0, 1.0, 0.0, 1.0 ),  // green
      color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
      color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
      color4( 1.0, 1.0, 1.0, 1.0 ),  // white
      color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
void quad( int a, int b, int c, int d ) {
      //first triangle
      colors[Index] = vertex_colors[a];
      points[Index] = vertex_positions[a];
      Index++;
      colors[Index] = vertex_colors[b];
      points[Index] = vertex_positions[b];
      Index++;
      colors[Index] = vertex_colors[c];
      points[Index] = vertex_positions[c];
      Index++;
      //second triangle
      colors[Index] = vertex_colors[a];
      points[Index] = vertex_positions[a];
      Index++;
      colors[Index] = vertex_colors[c];
      points[Index] = vertex_positions[c];
      Index++;
      colors[Index] = vertex_colors[d];
      points[Index] = vertex_positions[d];
      Index++;}
// generate 12 triangles: 36 vertices and 36 colors
void colorcube() {
      quad( 1, 0, 3, 2 );
      quad( 2, 3, 7, 6 );
      quad( 3, 0, 4, 7 );
      quad( 6, 5, 1, 2 );
      quad( 4, 5, 6, 7 );
      quad( 5, 4, 0, 1 );}
//Cube VBO and VAO initialization
void initCube(){
      // Load shaders
      program = LoadShaders(
            "vshaderCube.vert",
            "fshaderCube.frag" );
      glUseProgram( program );
      //generate and bind a VAO for the Cube
      glGenVertexArrays(1, &vao);
      glBindVertexArray(vao);
      colorcube();
      // Create and initialize a buffer object
      glGenBuffers( 1, &buffer );
      glBindBuffer( GL_ARRAY_BUFFER, buffer );
      glBufferData( GL_ARRAY_BUFFER,
            sizeof(points) + sizeof(colors),
            NULL, GL_STATIC_DRAW );
      glBufferSubData( GL_ARRAY_BUFFER,
            0, sizeof(points), points );
      glBufferSubData( GL_ARRAY_BUFFER,
            sizeof(points), sizeof(colors), colors );
      // set up vertex arrays
      GLuint vPosition =
            glGetAttribLocation( program, "vPosition" );
      glEnableVertexAttribArray( vPosition );
      glVertexAttribPointer( vPosition, 4,
            GL_FLOAT, GL_FALSE,
            0,BUFFER_OFFSET(0) );
      GLuint vColor =
            glGetAttribLocation( program, "vColor" );
      glEnableVertexAttribArray( vColor );
      glVertexAttribPointer( vColor, 4,
            GL_FLOAT, GL_FALSE,
            0,BUFFER_OFFSET(sizeof(points)) );
      glBindVertexArray(0);
}
void displayCube(){
      glUseProgram(program);
      glBindVertexArray(vao);
      glDrawArrays( GL_TRIANGLES, 0, NumVertices );
      glBindVertexArray(0);}
```

BasicCubeGUI with the AntTweakBar lib:

```
//Initialization of AntTweakBar
TwInit(TW_OPENGL_CORE, NULL);
TwWindowSize(windowWidth, windowHeight);
myBar = TwNewBar("myBar");
TwAddVarRW(myBar, "wire",
   TW_TYPE_BOOL32, &wireFrame,
   " label='Wireframe mode' key=w help='Toggle\
   wireframe display mode.' "
);
TwAddVarRW(myBar, "bgColor",
   TW_TYPE_COLOR4F, glm::value_ptr(bgColor),
   " label='Background color' "
);
//Ant Tweak Bar Rendering in main loop
TwDraw();
//Ant Tweak Bar Termination
TwTerminate();
```

BasicCubeGUI Vertex Shader:

```
#version 150 core
in    vec4 vPosition;
in    vec4 vColor;
out   vec4 color;
void main() {
  gl_Position = vPosition;
  color = vColor;}
```

BasicCubeGUI Fragment Shader:

```
#version 150 core
in    vec4 color;
out   vec4 colorOUT;
void main() {
    colorOUT = color;}
```

Assignment 5 Blinn-Phong Illumination Model (Fragment Shader):

```
#version 150 core
//Data as passed from vertex shader
in          vec3 wPosition;
in          vec3 EyeDirection;
in          vec3 LightDirection;
in          vec3 fNormal;
out         vec4  colorOUT;
uniform     vec3 LightPosition;
uniform     vec3 MaterialDiffuseColor;
uniform     vec3 MaterialAmbientColor;
uniform     vec3 MaterialSpecularColor;
uniform     float LightIntensity;
uniform     float Shininess;
uniform     vec3 LightDiffuseColor;
uniform     vec3 LightAmbientColor;
uniform     vec3 LightSpecularColor;
void main()
{
      vec3 n = normalize(fNormal);
      vec3 l = normalize(LightDirection);
      vec3 v = normalize(EyeDirection);
      vec3 H = normalize(l + v);
      float cosTheta = max( dot( n,l ), 0.0 );
      float cosAlpha = pow( max(dot( n, H ), 0.0), Shininess);
      //Attenuation
      float d = length( LightPosition - wPosition );
      colorOUT = vec4(
            MaterialAmbientColor * LightAmbientColor +
            MaterialDiffuseColor * LightDiffuseColor * Light-
            Intensity * cosTheta / (d*d) +
            MaterialSpecularColor * LightSpecularColor * Light-
            Intensity * cosAlpha / (d*d), 1.0 );
}
```