A Shader Library for OpenGL 4 and GLSL 4.3 Learning and Development

Vitor de Andrade
Polytechnic School - UFRJ
Rio de Janeiro, Brazil
Email: vitor.andrade@poli.ufrj.br

Ricardo Marroquim
COPPE - UFRJ
Rio de Janeiro, Brazil
Email: marroquim@cos.ufrj.br

Abstract—In the past decades, besides experiencing a huge development in terms of computation speed, we have also experienced the emergence of the Programmable GPU, giving birth to languages like GLSL and CUDA. This technology gives great flexibility for the usage of such a powerful hardware, attracting the interest of many researchers and programmers to this area. More recently, the release of OpenGL 4 provided even more flexibility to users, but has turned the learning curve steeper, since even the basic applications demand knowledge of many Linear Algebra concepts and extensive proficiency in Shaders and OpenGL programming. This paper focus in a shaders library aimed to help students, researchers and programmers in the usage of GPU programming with OpenGL 4.

Keywords-GLSL 4.3; OpenGL 4; GPU Programming;

I. INTRODUCTION

During the last decade, shader programming has evolved from a fixed-function pipeline to a programmable one. The number of programmable shaders has also increased. It began with the vertex and fragment shaders pair, increasing to currently six programmable stages, as can be viewed in Fig. 2. The new OpenGL and GLSL versions have also added a myriad of new functionalities and control access for GPU programming.

On one hand, the creation of the programmable graphics pipeline added great flexibility and provided an extremely powerful resource for stream programming. On the other hand, when OpenGL 4 was released, the fixed functionalities were deprecated, forcing the programmers to implement their own shaders and handle all transformations (e.g. build the model, view and projection matrices), even for simple applications. Thus, the learning curve for beginners has become much
steep and the amount of setup code to start an application has substantially increased. An introduction to programming with OpenGL 4 can be found in the online OpenGL Book: [1].

This paper presents a library under development to abstract and spare most of burden of setup code and configurations that usually is repeated in most applications. This helps beginners to learn GPU programming and allows faster development of graphics applications, letting the programmer focus directly in the application development. As a proof-of-concept, we also describe some applications developed with the library.

There are other libraries with similar purposes, like ShaderLabs [2], [3] and SpiderGL [4]. The first is focused towards learning shader programming and the latter is an API for shader programming under WebGL. Differently from ShaderLabs, our goal is to create an API to reduce programming time and facilitate the use of resources. However, we have focused in the leading edge versions of the graphics libraries, i.e. OpenGL 4 and GLSL 4.3, to fully support new features, such as the recently announced Compute Shaders. It’s important to notice that the library’s aim is not to improve performance or optimize any existing algorithm. It just translates the OpenGL and GLSL calls to cleaner and more intuitive calls. Therefore, there’s no impact on performance caused by it’s usage.

II. SHADERS LIBRARY

A. Basic Library Structure

The library uses the idea of an Effect being anything that you want to compute using GPU Programming. Most of the application specific code is implemented in the Effect class, supported by the other classes. It currently consists of 5 classes, explained in details in the following subsections.

B. Effects Class

The Effect class is the base class used for the application development. A typical GPU application developed with our library extends this class to hold most of the application core. It typically handles any CPU pre-computation and rendering methods.

As described in the Applications section (Sec. III), an effect can be used to represent, for instance, any kind of shading applied to a 3D model, a GPU based ray tracer, or even a non graphical GPGPU computation, even if this is not really the focus of the library.

C. Framebuffer

This class is responsible for the framebuffer generation and storage. An FBO may hold many textures that can be used for reading and writing during the shader stages. It is usually employed for offscreen rendering, writing to multiple buffers or for multi-pass approaches. Even though these are common needs in GPU programming, an FBO creation and use is extremely tedious. This class abstracts most of the setup and allows the programmer to create and use framebuffers with very few lines of code. It handles the texture and buffer creations, as well as their binding and usage, which involves allocating texture slots and passing them to the shaders.

An FBO can be created simply in one line:

```cpp
GLuint* fbo = new Framebuffer(w, h, num_buffers, texType);
```

Where w and h are the width and height of the framebuffer, `num_buffers` is the number of textures (or attachments) that the FBO will hold and `texType` is the type of the framebuffer texture object (ex. GL_TEXTURE_2D).

When a framebuffer is bound, it automatically looks for an empty texture slot, freeing it when it is unbound. This means that the library is in charge of all the attachments management, while the user only needs to bind or unbind a framebuffer according to his needs:

```cpp
fbo->bind();
fbo->bindAttachment(texID);
// ... Application code in here ... 
fbo->unbind();
```

The second line of code attaches the texture `texID` to a texture unit and returns the unit number. There are also many other functions to perform useful framebuffer operations like reading the buffer, binding an attachment to a specific texture unit, among others. To illustrate a common example in GPU programming, passing an FBO texture as input for a shader is accomplished with the following line of code:

```cpp
shader->setUniform(“textureName”, fbo->bindAttachment(texID));
```

Note that the programmer does not have to explicitly work with the texture unit identifications. The texture can be promptly accessed in the shaders as:

```cpp
uniform sampler2D textureName;
```

D. Mesh

This is a helper class to load and work with typical mesh files. It has a built-in wavefront (.obj) loader. Its main goal, however, is to handle the Buffer Objects creation and management. Since OpenGL 4 has deprecated the usual way of creating polygons, normals, texcoords, etc., everything must now be directly stored with the usage of Buffers. However, the setup is, again, long and is usually very similar from application to application.

Typically the programmer creates arrays to store the mesh information read from the file. Then the buffers are created and the data is actually loaded in the buffers. These operations are translated in quite a few configuration lines in order to simply load and render a triangle mesh. More information on the buffers usage can be found in [3].

The Mesh class is useful because it spares the programmer of all this work. They are created with just a single line of code and the mesh setup is done with very few lines:

```cpp
Mesh* mesh = new Mesh;
mesh->createBuffers();
mesh->loadOBJFile(myMesh.obj);
```
The buffers can be accessed inside the shader through vertex attribute locations, using the layout qualifier. These are user maintained numbers identifying where each buffer is bound to. However, in our library the most common attributes of a triangular mesh have pre-defined locations according to Table I.

<table>
<thead>
<tr>
<th>Location</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vertex Buffer</td>
</tr>
<tr>
<td>1</td>
<td>Normals Buffer</td>
</tr>
<tr>
<td>2</td>
<td>Colors Buffer</td>
</tr>
<tr>
<td>3</td>
<td>Texture Coordinate Buffer</td>
</tr>
</tbody>
</table>

Moreover, the library handles all the buffers binding, so the mesh can be rendered without any further hassle.

```cpp
mesh->render();
```

In the shaders side, the attributes are ready to be accessed in the vertex shader, for example:

```cpp
layout(location=0) in vec4 in_Position;
layout(location=1) in vec3 in_Normal;
```

Furthermore, when the mesh is loaded, the library also performs some geometrical computations, for instance, its center is detected using an axis-aligned bounding box and normalization factors. These are usually useful for manipulating 3D models.

Finally, this class contains some functions to generate basic meshes, e.g. a quadrilateral (interesting for off-screen rendering without the use of compute shaders) or a cube.

E. Shaders

1) Shaders Initialization: The Shaders class contains everything needed to load and create a shader program. Although OpenGL receives a shader as a string, it is interesting for the user to write the code in external text files, leaving the application in charge of loading the files and initializing the shader. This class facilitates almost all shader setup. It automatically searches a given directory for any shaders with a given name, creating and initializing the Shader Program.

For instance, if the user has vertex and fragment shaders defined as myShader.vert and myShader.frag and located in the directory myShadersDir, the following lines are used for the complete shader initialization:

```cpp
Shader* shader = new Shader("myShadersDir", "myShader");
shader->initialize();
```

The library will automatically search the given directory for any shader extension of the file name, i.e., .vert, .frag, .geom, .comp, or .tess.

From then on, a simple call to the method enable or disable is enough to bind or unbind a shader for rendering.

2) Uniform Setting: In OpenGL 4 the user has to previously specify the type of variable to be loaded in the shaders, since there is a specific uniform function to set each type of variable. Moreover, he needs to maintain the locations values where each uniform is bound. To simplify this work, the library overloads many functions in order to easily set uniform variables. The uniforms can be set either by the location integer or by the name of the variable inside the shader.

All different functions to handle uniform setting were overloaded in a single one: `setUniform`. This method calls the appropriate OpenGL function depending on the parameters received. For instance, if the user needs to load 3 integers - x, y, z to an ivec3 named `myVector` inside the shader, he simply calls:

```cpp
shader->setUniform("myVector", x, y, z);
```

instead of the specific call to

```cpp
glUniform3i(uniformLocation, x, y, z);
```

In the same way, a call to

```cpp
shader->setUniform("myFloat", f);
```

will be translated into

```cpp
glUniform1f(uniformLocation, f);
```

Note however that without the library, the programmer must also maintain or query the variables' locations.

3) Reloading Shaders: A simple, yet useful, feature of this class is the method to reload the shaders in run-time. The programmer is able to modify the shader and immediately see the results without having to restart the application.

F. Trackball

To handle common transformations required for a visualization application, we have included a trackball class in the library. It uses the Eigen Library ([6]) for most transformations. The quaternion class is used for general rotation and the affine matrix template is used for the object's transformations and the projection matrix. To reproduce the usual OpenGL pipeline, we store this transformations in three matrices (Model, View and Projection) that are passed as uniforms to the shaders upon demand. Besides, since the trackball handles any general transformations, it is also possible to create a light trackball, providing an easy way to control the light position. The trackball class also contains a shader to draw its representation on the screen.

III. Applications

We have developed some applications to demonstrate the use of the library in different contexts. These are briefly explained in the following subsections.
A. Shading Effects Visualizer

1) The Application: This application consists in a platform to apply different shader effects to a 3D mesh (see Fig. 1). It uses GLFW [7] to create a window context, AntTweakBar [8] for the menus, and Eigen [6] for linear algebra (such as vector and matrices operations). The systems flowchart is displayed in Fig. 3.

Although a shading effect is usually defined by inheriting the Effect class, when the effect is very simple and does not need any kind of pre-computation or variable storage (e.g. the Phong Shading Effect), it can be implemented directly using the shader class.

In Fig. 4 we have a closer look at the application. On the right side of the image there is the menu containing all the available effects, some configuration options and the representation of the light direction, that is manipulated using the light trackball. In this menu, the user can select between one of the shading effects, set parameters for each effect and visualize them in real time.

When the programmer wants to add a new Effect to the application, the Effect class is inherited and two methods must be overloaded: initialization and rendering. For instance, to create the Toon effect, the initialization is very simple since it contains only one shader:

```cpp
Toon::initialize() { 
    shader = new Shader("shadersDir", "toonShader"); 
    shader->initialize(); 
}
```

The rendering function enables the shader, passes the appropriate uniforms, renders the mesh and finally disables the shader:

```cpp
shader->enable(); 
shader->setUniform("projectionMatrix", cameraTrackball->getProjectionMatrix(), matrixDimension); 
// ... Sets the other uniforms here...
mesh->render(); 
shader->disable(); 
```

In order to use an effect in the application, basically all that’s needed is to pass the effect’s name and the buffer’s width and height (usually the viewport):

```cpp
MyEffect myEffect = new MyEffect("Effect Name", bufferWidth, bufferHeight); 
myEffect->initialize();
```

Finally, to render the mesh with the selected effect, the application just calls it’s rendering function, passing the mesh and trackballs objects as parameters:

```cpp
activeEffect->render(mesh, cameraTrackball, lightTrackball);
```

Note that any combination of these three parameters can be used. For instance, in a simple image processing application all three parameters would be typically set to NULL.

2) SSAO: To give an example of a less trivial rendering effect, we briefly describe the implementation of a Screen Space Ambient Occlusion. This is an approximation of the Ambient Occlusion approach computed in real time directly in GPU. For details about this technique please refer to [9], [10], and [11].

The SSAO computation consists in a three pass approach, all implemented directly inside the rendering method of the class inherited by the Effect, as can be seen on Fig. 5. Note that the application does not differ what effect is being called, it simply calls the rendering function from the active effect.

The first SSAO rendering pass consists of an off-screen rendering to store depth information:

```cpp
deferrredShader->enable(); 
fbo->bind();
```
In the second pass, the depth information obtained from the first one is used to compute the Ambient Occlusion term. The result is rendered again to a texture that is used in the third pass to remove the high frequency noise generated by the SSAO algorithm.

In the third pass, a gaussian blur is applied to remove the noise from the second pass:

```c
ssaoShader->enable();

gbo->bind();
glDrawBuffer(GL_COLOR_ATTACHMENT0 +
deptTextureID);
glClear(GL_COLOR_BUFFER_BIT|
GL_DEPTH_BUFFER_BIT);

gbo->bindAttachment(0,1);

// Sets the uniforms here...

mesh->render();

ssaoShader->disable();

gbo->unbindAttachments();
```

B. GPU Based Ray Casting

To show how the library can be employed to write a more complex rendering method, we have developed a simple Volume Render using ray-casting. It’s interface and the output image generated by the ray-casting are seen in Fig. 6. Besides the visualizer window similar to the last examples, it also contains one class storing a 3D texture representing the Volume to be visualized.

Briefly, an output image resolution is set and a ray is cast for each pixel. Each ray traverses the volume and samples the color value from the volume texture at each iteration (traversal step). The values are accumulated and when the traversal is finished a color is set to the corresponding pixel in the final image, which is finally visualized as a textured quad primitive.

![First Iteration](image1)

![Second Iteration](image2)

Since FBO textures cannot be set as **READ** and **WRITE** in the same pass, the ray-casting traversal would be usually implemented as a ping-pong scheme for GPU approaches (Fig. 7). Thus, the FBO would read from one attachment and write to another, switching roles after each pass. Another approach would be to iterate directly inside the shader using a loop structure. However, this would be prohibitive for traversals with many steps, since the number of iterations inside the loop is limited.

To show the power of the new compute shaders, we have implemented the traversal avoiding the ping-pong scheme. With the **Image Load/Store** feature, the implementation was extremely simplified, since an image can be used for reading and writing simultaneously. The number of groups called with the compute shader is equal to the viewportSize in each axis.
meaning the compute shader will be called once per fragment, casting one ray per screen pixel. The binding of the textures and calling of the compute shader is indicated as follows:

```c
// ... Ray Casting setup definitions ...
computeShader->enable();

// Bind Textures for Shader Reading:
glBindImageTexture(0, currentColorsTexture, 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA32F);
glBindTexture(GL_TEXTURE_3D, volume->getTexture3D());

// ... Set uniforms here...

// Call Compute Shader:
glDispatchCompute(viewportSize[0], viewportSize[1], 1);

// Unbind Textures:
glBindImageTexture(0, 0, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA32F);
glBindTexture(GL_TEXTURE_3D, 0);
computeShader->disable();
```

Again, the iterations that sample through a volume can be made either outside the shader, calling the shader once per traversal step, or inside the shader, sampling multiple slices in each shader, which decreases the number of shader calls.

IV. CONCLUSIONS AND FUTURE WORK

The library has substantially simplified the development of applications using OpenGL 4 and GLSL 4.3. The amount of coding for setup and initializations has greatly decreased, thus the programmer can focus on the application details. By abstracting all the hassle of complicated and extensive setups, the programmer can focus on the application coding. Furthermore, the library also serves as a simple framework to test new shaders fast and easy.

The next steps for the library development is to finish the implementation of the Tessellation shader compatibility, as well as sample applications for the geometry and tessellation shaders. Also, a more generic framebuffer class should be implemented, allowing for different texture formats for example.

V. ACKNOWLEDGMENT

The authors would like to thank Daniel Santos, Masters Student in COPPE-UFRJ, for his works developing the GPU Based Ray Casting Application and for providing its image used in this paper.

REFERENCES