ShaderBase: A Processing Tool for Shaders in Computational Arts and Design

Andrés Felipe Gómez[1], Jean Pierre Charalambos[1] and Andrés Colubri[1,2]
[1]*Departamento de Ingeniería de Sistemas, Universidad Nacional de Colombia, Bogotá, Colombia*
[2]*Department of Organismic and Evolutionary Biology, Harvard University, Cambridge, U.S.A.*

Keywords: Shader Programming, GLSL, Processing Language, Interactive Arts, Computational Design, Data Visualization, Database, Git.

Abstract: We introduce a new software tool called *ShaderBase* that facilitates using, sharing, and curating GLSL shaders in computational design, interactive arts, and data visualization. This tool is part of the Processing programming environment, an open-source project widely used for teaching and production in the context of media arts and design. Shaders are a crucial component in the development of large-scale data visualizations, interactive installations, real-time rendering tools, videogames, virtual reality applications, etc. However, their use requires advanced shader programming skills, and the creation of new shader-based effects demands a deep understanding of the graphics pipeline in modern Graphics Processing Units (GPUs). *ShaderBase* uniquely addresses these issues by allowing Processing users to easily upload and share shaders via an underlying Git repository. *ShaderBase* operates in close integration with Processing's interface, so that users can incorporate shaders into their programs with minimal effort. Furthermore, the shaders indexed in *ShaderBase* take advantage of Processing's drawing API, and incentives the use of shaders among artists and designers who might not be able to do so otherwise.

1 INTRODUCTION

The continued advances in graphics hardware, together with the evolution of low-level shading languages, have opened many possibilities for real-time rendering across a wide range of platforms and devices. Shader programming has become a fundamental resource in today's applications of Computer Graphics (CG). Real-time rendering benefits from the performance and flexibility offered by custom shaders, and with the rise of programmable pipelines, shader programming turned into a fundamental skill.

In the context of computational design, information visualization and interactive media, the necessity of making CG more accessible has motivated the creation of frameworks and APIs to facilitate programming among artists and designers (Orr, 2009). The Processing project (Processing Foundation I) is a well-known example of such initiatives to increase computer literacy within the design and visual arts, and visual literacy within technology and engineering. Processing, which in its core consists of a Java-based programming language together with a minimal development environment, was initiated by Casey Reas and Ben Fry at the MIT Media Lab back in 2001. Today, the project has expanded substantially, thanks to its open source nature and community involvement, and it is used around the world as a teaching medium as well as a production tool (Reas and Fry, 2014).

Processing's simple API combined with an unobtrusive development environment allows beginners to obtain initial visual results quickly and then to refine their programs by "sketching" progressively more complex versions of the code. The drawing API in Processing is comprised of around 300 functions, inspired by several graphics libraries, mainly Postscript, QuickDraw, OpenGL, and Renderman (Processing Foundation II).

Although shaders are an essential part in the development of graphical applications, they are not as popular as they could be, particularly in the fields of computational arts and design. One of the reasons for this situation is the need of advanced programming knowledge in shading languages. Although there are some online shader libraries, their usefulness as resources for beginners is limited:

Gomez, A., Charalambos, J. and Colubri, A.
ShaderBase: A Processing Tool for Shaders in Computational Arts and Design.
DOI: 10.5220/0005673201890194
In *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2016) - Volume 2: IVAPP*, pages 191-196
ISBN: 978-989-758-175-5

the NVIDIA shader library (NVIDIA, 2008) was last updated in 2008, and Geeks3D shader library (Geeks3D, 2015) is geared toward CG enthusiasts, while GLSL sandbox (Cabello, 2015), ShaderToy (Quilez, 2015), and VertexShaderArt (Tavares, 2015), are focused on WebGL. This limits inexperienced users who wish to start experiment with shaders through Processing's shader API. *ShaderBase* enables all Processing users -beginner and advanced alike- to explore, interact, and take advantage of this API by allowing them to download, upload, and edit the shaders indexed into a Git repository. Our approach to simplify shader programming in Processing is unique in that it is closely integrated with Processing's environment and uses Git as the underlying database. *ShaderBase* lets users to re-use shaders within their own code and share them with other members of the Processing community, as well as to look for specific shader effects with *ShaderBase*'s search functionality. Even though the tool is specifically geared towards Processing, it could be quickly re-implemented for other interactive arts and design frameworks, such as OpenFrameworks and Cinder, and the shaders archived in the underlying Git repository could be repurposed for other languages and platforms (mobile, webGL) with minimal code changes.

2 SHADERS IN PROCESSING

The shader API in Processing is based on a single class called PShader that encapsulates a complete shader program including a vertex and a fragment stage. This class exposes methods to set the values of the uniform variables declared in the GLSL code. The code listing below shows a simple example where a negative effect shader is loaded to render the scene.

```
// Processing code
PShader neg;
PImage img;
void setup() {
  size(800, 600, P2D);
  img = loadImage("img.jpg");
  neg = loadShader("fragment.glsl");
}
void draw() {
  shader(neg);
  image(img, 0, 0);
}


// fragment.glsl
uniform sampler2D texture;
varying vec4 vertColor;
varying vec4 vertTexCoord;
void main() {
 vec4 col = texture2D(texture,
   vertTexCoord.st);
 gl_FragColor = vec4(1.0 -
   col.rgb, 1) * vertColor;
}
```

A key consideration when incorporating shader programming into pre-existing "creative coding" frameworks such as Processing, which is specifically geared towards less technical users such as computational designers and artists, is how to blend the new functionality with the existing API. Since often times the typical Processing user is not able to write his own shaders, Processing already implements several rendering paths that cover different usage scenarios: high-quality 2-D drawing with antialiasing and accurate line strokes, 3-D rendering with a Blinn–Phong, Gouraud lighting model, where interactive performance is prioritized over more accurate shading/texturing algorithms. All of this functionality is implemented via default GLSL shaders built into the OpenGL renderer in Processing. Details are provided in (Colubri) and (Gómez et al., 2016).

3 *ShaderBase*

Our main goal with *ShaderBase* was to create a graphical interface and an underlying database to allow any Processing user to easily share GLSL shaders, facilitate the use of Processing's shader API, and promote the application of shading effects particularly among practitioners of digital arts and design. The user does not require advanced programming knowledge in order to run and execute the shaders indexed in the database. The tool simply downloads the shader directly into the Processing program (called a "sketch" in Processing's terminology), together with all the additional files needed to run the shader. *ShaderBase* stores all the shader information in the remote repository, and when the user installs the tool in Processing this information is retrieved into the tool. *ShaderBase* also allows searching shaders with a specific tag. The design of *ShaderBase*'s data model is discussed in the section 3.1 and the user interface is described in the section 3.2.

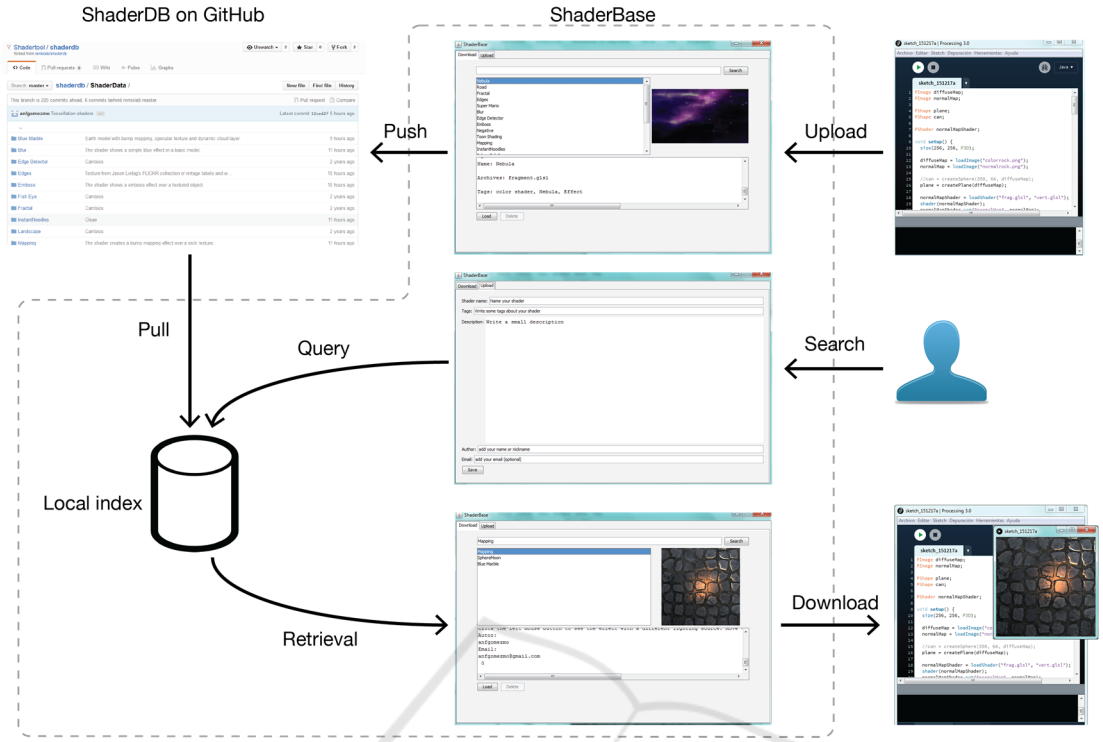

Figure 1: Data flow and user interface elements in *ShaderBase*.

3.1 *ShaderBase* Data Model

A basic requirement for *ShaderBase* is be able to upload, modify, request, and download any shader information at any time without losing it during the transfer, or having any corruption in the saved data that could affect the execution of the shader. Another important future-proofing feature is the possibility of porting the shader files into other coding frameworks. There are some options that could solve all these design needs, such as SQL database, Object Oriented Database, and Mobile Database (Foster and Godbole, 2014). However, there are some limitations such as the language used by Processing (Java) restricting the options for the development of such database. Most SQL databases created in Java require a JDBC driver that functions as a wrapper between database and the application. The JDBC driver asks user authentication every time when the user logs into the database and modifies the information in the tables, which would slow down the response when the user selects, downloads, and uploads a shader. In order to upload shaders, the tool needs to be connected to the database to reflect the changes with other users and update the tool, and thus it would require a 24/7 SQL dedicated server storing the information specifying all shaders. An SQL database does not allow seeing the indexed data directly without a SQL program, and a query is required to access the data. Furthermore, an SQL server does not record the changes done by other users to the indexed files.

In the other hand, we found Git to be a better solution for these limitations (Chacon and Straub, 2014). A user can commit changes into the database at any time and they will be reflected after a push. It is not needed to have a dedicated 24/7 server since each user will have a clone of the data indexed and they can work in a local environment. The communication between Git and the tool is done using a library (Jgit) that is faster than a JDBC driver. All the information hosted in a Git repository can be accessed at any time with a simple browser without the need of querying it. Since Git incorporates Version Control (VC), all the changes done to the files are recorded so it is easy to recover missing files or corrupted data, also the snapshot done by Git prevents any unexpected change on the files. The data flow is show in Figure 1.

In order to develop *ShaderBase*, we had to identify the data required to completely define a valid shader in Processing, where and how to store that data, and how to send the data back and forth between Processing and *ShaderBase*. Each shader record comprises the following files:

- Sketch File: Processing source code where the

PShader object is declared and the shader file is loaded.

- Shader File: fragment, vertex, geometry or tessellation shader GLSL source code.
- Indexed File: File containing shader metadata (such as tags and user information) used to query the database.
- Shader Snapshot File

ShaderBase stores all the information in a local clone of a master remote Git repository. All copies of *ShaderBase* use *ShaderDB* to store data remotely, future versions might allow to select a different remote repository, so users can create independent collections of shaders. *ShaderDB* stores each shader using the following file structure:

```
`-- ShaderData/
    `-- Shader1/
        |-- Code/
        |   `-- Shader1.glsl
        |-- Shader1.jpg
        |-- Shader1.pde
        |-- Shader1.txt
        `-- Tabs/
            `-- Tab_n.pde
```

The folder Code has all the extra data required to run a shader, the minimal data saved in this folder would be the GLSL code file. However, some shaders will require extra files in order to run in Processing, in such cases *ShaderBase* will download and import those files automatically. The folder Tabs will only appear when a shader requires multiple tabs in the sketch. The index text file is created after filling a small questionnaire when a user decides to upload a shader no matter if it is going to be saved just in the local machine or if it is going to be shared. This file has all the information required to provide an accurate search, including tags, name, author and description for each shader.

Accessing Git from Java is a non-trivial task, however there are several libraries that that can be used for this purpose: JGit, JavaGit, Git Hub API for Java, etc. All these libraries have strengths and disadvantages. We selected JGit because it is the library that implements most of the Git commands. Our tests indicate that JGit is a slightly faster than the other libraries and has fewer bugs. In addition to that, it allows accessing the Git main file structure [JGit API] through code to check the commits, whicht helps preventing missing commits or collision with commits from other users. JGit is actively developed by the Eclipse EGit group, which ensures that is kept updated to the last available Git version.

The database management is performed by means of the following Git commands:

- Clone: downloads all the information stored at the remote repository only when the tool is run for the first time.
- Commit: stores newly created shader records into the local repository.
- Fetch: downloads new or modified records from the remote repository.
- Merge: incorporates (fetched) changes into the local repository.
- Pull: runs a fetch command followed by a merge one.
- Push: updates the remote repository.

When the user uploads a shader, *ShaderBase* does a commit to the local repository, and if the user shares it and *ShaderBase* is up-to-date with the shaders indexed in *ShaderDB*, then the shader is pushed to the remote repository. In the case that *ShaderBase* is not in sync with *ShaderDB*, and then the shaders uploaded and shared by the user will be reflected in *ShaderDB* after the next update performed when the tool starts up. *ShaderBase* checks for updates by doing a Git fetch; the tool checks what files are new or which ones were changed and modifies the local database. All these functions are totally transparent to the user. The user will only select the shader she wants to upload or download without worrying about which one of these functions is being called. *ShaderBase* can work offline after the first clone, but an Internet connection is required to upload a shader to the remote repository, to pull any changes, or download new shaders.

After cloning and receiving the data stored in *ShaderDB*, the tool gathers all the information of the indexing text files saved for each shader, indexes the documents and creates a main index. Each *ShaderBase* copy will have an independent index allowing the user to index their local shaders without the need of sharing them. Each time a new shader is added or if *ShaderBase* is updated the index is going to be remade.

ShaderBase offers search functionality based on Lucene, a Java scalable Information Retrieval (IR) library (McCandless and Hatcher, 2010). The design of the Lucene-based search functionality can be appreciated in Figure 1.

ShaderBase enables the following features in Lucene:

- Indexing with Standard Analyser. Just will index the indexing text file for each shader
- Search by Phrase Relevance. The search can be

fuzzy or weighted, but these options are disabled until the number of the shaders increases.

The user only needs to add some words or a simple phrase and *ShaderBase* will check which ones of these words are indexed and will provide a result with the shaders that have them in the indexing file. This result will only show the shaders that could be of the interest to the user. After searching and downloading the shader into Processing the user only needs to run the automatically generated sketch holding this shader.

3.2 *ShaderBase* User Interface

The user interface in *ShaderBase* is divided in two separate areas, Download and Upload, which can be selected with the tabs located in the top left region of the main window.

When the user selects a shader from the list of available shaders, a preview image and a short textual description will be displayed to help the user deciding if that is the shader she wants. If the shader is selected, ShaderBase will send it directly into Processing's editor, using all the information stored in Git to build a new sketch that the user can run immediately. If a sketch is already open, then *ShaderBase* creates a new sketch with all the required code and files. Again, all of these operations take place transparently.

The tool provides a search function as explained in Section 3.1. All the results are going to be reflected in the list showing only the shaders matching the search query. The upload process requires that Processing sketch holds a valid shader. Otherwise, *ShaderBase* will not allow uploading any files to avoid the database being polluted with non-shader code. When a user uploads a shader, *ShaderBase* will ask to either share it with other users, or to save it in the local repository. When the latter option is selected, the shader is not sent to the remote Git repository, and is only indexed for the search in the local machine. In order to upload a shader the user must fill the small template shown in the section 3.1. The main purpose of the template is to create an index file to ensure that the shader can be found through the search queries.

Shaders can also be deleted from *ShaderBase*'s interface; but this is allowed only if the shader is saved in the local repository. The tool does not allow deleting shaders indexed in *ShaderDB*. However the user can modified those shaders, simply by uploading them using the same name. *ShaderBase* does not show the commit history of a shader stored in *ShaderDB*, we considered this feature not important for beginner users, while advanced users will be able to access the commit history of shader directly through GitHub.

4 CONCLUSIONS

Processing allows artists and designers working with code to use GLSL shaders in a simpler way. It makes the learning process more accessible to users without formal training in CG, who can also benefit from Processing's portability and absence of extra library configuration.

The availability of shaders provided by *ShaderBase* in Processing should greatly enhance the capabilities of the language, allowing for GPU-accelerated rendering of complex models and scenes. GLSL shaders already written for other platforms and/or frameworks can be adapted to work in Processing with minor code changes, thus helping to share resources and knowledge between coding communities. Furthermore *ShaderBase* will allow sharing shader code and expertise over the web, thanks to the widespread adoption of Processing as a teaching and production tool in digital arts and design. *ShaderBase* has recently released to the community (Gómez, 2015a), with an initial selection of shaders .(Gómez, 2015b)

From a technical standpoint, *ShaderBase* inherits all the advantages of Git, which guarantees a robust transfer and ensures that the information is not going to be corrupted easily. Also, the tool does most all the work in the local machine and thanks to the search queries it allows users to find shaders very quickly. All the shaders indexed in the machine can be loaded directly into Processing's editor. Sharing new shader rendering effects does not involve any extra work from the user.

Finally, *ShaderBase* provides access to shaders created by a growing community of creative coders, artists and designers. These shaders could be used immediately from within Processing without any previous setup or additional changes in the code. In particular, this would enable users without previous GLSL knowledge to start exploring the possibilities offered by shader programming and Processing's shader API.

ACKNOWLEDGEMENTS

The authors would like to thank the contributions over the years from all the members of the

Processing community.

REFERENCES

Cabello, R. 2015. *GLSL Sandbox* [Online]. Available: http://glslsandbox.com/

Chacon, S. & Straub, B. 2014. *Pro Git*, Apress

Distributed to the book trade worldwide by Springer Science+Business Media New York.

Colubri, A. *PShader tutorial* [Online]. Available: https://www.processing.org/tutorials/pshader/

Foster, E. C. & Godbole, S. 2014. *Database systems: a pragmatic approach*, Apress

Distributed to the book trade worldwide by Springer Science and Business Media.

Geeks3D. 2015. *Shader Library* [Online]. Available: http://www.geeks3d.com/shader-library/

Gómez, A. F. 2015a. *ShaderBase* [Online]. Available: https://github.com/remixlab/shaderbase

Gómez, A. F. 2015b. *ShaderDB* [Online]. Available: https://github.com/remixlab/shaderdb

Gómez, A. F., Colubri, A. & Charalambos, J. P. Shader Programming for Computational Arts and Design: A Comparison between Creative Coding Frameworks. 11th International Conference on Computer Graphics Theory and Applications, 2016 Rome.

Mccandless, M. & Hatcher, E. 2010. *Lucene in action,* Stamford, Conn., Manning Pub.

NVIDIA. 2008. *NVIDIA shader library* [Online]. Available: http://developer.download.nvidia.com/shaderlibrary/webpages/shader_library.html

Orr, G. 2009. Computational thinking through programming and algorithmic art. *SIGGRAPH 2009: Talks.* New Orleans, Louisiana: ACM.

Processing Foundation I. *Processing project home page* [Online]. Available: https://processing.org/

Processing Foundation Ii. *Processing Reference* [Online]. Available: https://processing.org/reference/

Quilez, I. 2015. *ShaderToy* [Online]. Available: http://shadertoy.com/

Reas, C. & Fry, B. 2014. *Processing: A Programming Handbook for Visual Designers and Artists (2nd Edition),* Cambridge.

Tavares, G. 2015. *VertexShaderArt* [Online]. Available: http://www.vertexshaderart.com/