Shader Programming for Computational Arts and Design
A Comparison between Creative Coding Frameworks

Andrés Felipe Gómez[1], Andrés Colubri[1,2] and Jean Pierre Charalambos[1]
[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.*
{anfgomezmo, jpcharalambosh}@unal.edu.co, acolubri@fas.harvard.edu

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

Abstract: We describe an Application Program Interface (API) that facilitates the use of GLSL shaders in computational design, interactive arts, and data visualization. This API was first introduced in the version 2.0 of Processing, a programming language and environment widely used for teaching and production in the context of media arts and design, and has been recently completed in the 3.0 release. It aims to incorporate low-level shading programming into code-based design, by integrating traditional models of graphics programming with more expressive approaches afforded by the OpenGL pipeline on modern GPUs. We contrast Processing's shader API with similar interfaces available in other frameworks used in computational arts and design, in order to better understand its advantages and shortcomings.

1 INTRODUCTION

The fields of computational arts and design rely heavily on the use of interactive Computer Graphics (CG). Practitioners continuously push the boundaries of real-time rendering as a medium for visual expression. In order to respond to the need of teaching programming to artists and designers, as well as to integrate code-based practices into their artistic production, several open-source programming frameworks specifically geared towards this audience have been developed during the past decade (Orr, 2009), such as Processing (Processing Foundation I), OpenFrameworks (OF) (OF community), and Cinder (Bell et al.), and gave rise to a practice often identified as "creative coding". The Processing project, conceived in 2001 by Casey Reas and Ben Fry at the MIT Media Lab, is among the earliest of such initiatives (Reas and Fry, 2006) and perhaps the one with the strongest emphasis on the pedagogical aspects and the explicit goal to "increase computer literacy within the design and visual arts, and visual literacy within technology and engineering" (Reas and Fry, 2014).

As graphics hardware and APIs continue to evolve at rapid pace, the widespread availability of programmable Graphics Processing Units (GPUs) and shading languages such as GLSL (Rost, 2009) enables new and exciting possibilities in real-time rendering. However, this comes at the cost of increased demands in technical knowledge. With the introduction of OpenGL 3.0 on the desktop, OpenGL ES 2.0 on mobile devices, and WebGL 1.0 on the browser, these demands became almost inescapable. One of Processing's main strengths is its minimal API (Processing Foundation II), which combined with a simplified 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. Therefore, a key question is whether it is desirable to incorporate some level of shader programming into Processing's API, and if so, what would be the best way to do it while maintaining its simplicity.

In order to answer this question, we first note that Processing is used to create data visualizations, interactive installations, and generative artworks that require smooth animation of complex geometries in high-resolution displays. This alone indicates the need of incorporating some extent of shader support into the language, since the most direct of achieving high performance graphics is through shader customization and optimization (2008). To address the "how" part of the question, we need to keep in mind that Processing's drawing API was first introduced more than 10 years ago and it is strongly rooted in the model of immediate-mode rendering (Glazier, 1992). This mode is more intuitive for new users, helps minimizing boilerplate code, and accommodates sketching and generative graphics better than more structured models, such as scene graphs. It has served as the basis of numerous teaching curricula (Ascioglu) and other programming frameworks (McCarthy). Therefore, a guiding principle in the design of the shader API was to fit Processing's immediate-mode rendering in a reasonably performant and non-obtrusive way, and to follow the principles of simplicity and immediate output that are central to the project. Some of the elements of the API were first introduced in a Processing library (Colubri, 2008) that extended the capabilities of the OpenGL renderer in Processing 1.0, and the API was consequently implemented in Processing 2.0 (Colubri and Fry, 2012). Its completion was reached in version 3.0, released on September 30 of 2015 (Palop, 2015).

As the general question of how to properly integrate low-level shader programming with a higher-level drawing API has been addressed by other creative coding frameworks, we carried out an initial comparison between the shader APIs in Processing, OF, and Cinder. We decided to focus on these frameworks given their popularity among the computational arts and design community, their significant overlap as teaching and production tools, and finally because they are all based on traditional textual programming languages, Java in the case of Processing, C++ in the case of OF and Cinder.

The article is organized as follows: we describe Processing's shader API in more detail in section 2, introduce the relevant elements of the corresponding APIs in OF and Cinder in section 3, present the results of the initial comparison between these APIs in section 4, and conclude by discussing our findings and prospects for future work in section 5.

2 SHADER API IN PROCESSING

Shader programs consist in several stages along the graphics pipeline of the GPU. Recent versions of the OpenGL and Direct3D APIs support vertex, geometry, tessellation, and fragment stages (Angel and Shreiner, 2012). The PShader class (Processing Foundation III) introduced in Processing 2.0 and completed in 3.0 allows users to encapsulate all these stages into a single entity that exposes basic methods to set the value of uniform variables declared in the shader source. The PShader class was designed giving priority to Processing's immediate-mode rendering. The following program listing illustrates the basic use of PShader, as well as the state-based approach in Processing:

```
PShader sh;
void setup() {
  size(640, 360, P3D);
  sh = loadShader("frag.glsl",
                  "vert.glsl");
  shader(sh);
}
void draw() {
  background(180);
  fill(140, 140, 190);
  sh.set("noiseFactor", 0.1);
  sphere(50);
}
```

The geometry needs to be re-drawn in each call of the draw() method, and the Processing renderer can be effectively abstracted out as a state machine, with functions allowing the user to set different state variables that are applied to subsequent drawing calls. The current shader, for example, is one of such variables set with the shader() function.

The advantage of the immediate mode rendering is the increased flexibility to define highly dynamic geometries, a common situation in interactive and generative art projects Processing is typically used for. The geometry does not need to be stored in a scene graph structure, and can be recreated entirely in each frame. The main drawback of this approach is the performance impact associated with rebuilding the geometry on the CPU, and copying the data from CPU to GPU memory in every frame. The renderer tries to optimize these calculations as much as possible by batching vertices from separate objects into a single buffer, in order to minimize the overhead associated with CPU-GPU transfers. However, Processing also allows defining persistent geometry with the PShape class (Processing Foundation IV). This class is implemented internally using vertex buffers, which ensures high performance when rendering large static geometries. Shaders can be applied on PShape objects in exactly the same way as they are to immediate-mode geometry.

Since Processing is primarily geared towards less technical users, we cannot make the assumption of the typical Processing user will be able to write her own shaders. To address this situation, the default state of the Processing renderer incorporates a complete set of shaders that covers all typical usage scenarios. In particular, 3-D rendering of textured lit scenes uses a simple Blinn-Phong shading model,

Table 1: List of attribute creation/setting methods.

Kind	Create/set method	Notes
Positional	attribPosition(name, x, y, z)	xyz in world coordinates
Normal	attribNormal(name, nx, ny, nz)	Also in word coords
Color	attribColor(name, c)	c is an integer storing an ARGB value
Generic	attrib(name, v…)	v is a list of float, integer, or Boolean values

suitable for rapid sketching but that can also be customized by more sophisticated models if the user provides the appropriate shaders. To facilitate the writing of custom shaders, Processing automatically initializes all the uniform and attributes variables that correspond to the default vertex and scene information that the user can control with the standard drawing API, so she does not need to explicitly take care of them. The list of given uniforms and attributes is described in detail in (Colubri I).

2.1 Custom Vertex Attributes

The built-in vertex attributes in Processing include position in world space, diffuse, ambient, specular, and emissive colors, shininess factor, normal vector, and texture coordinates. These attributes are sufficient to define arbitrary lit geometry in the built-in Blinn-Phong model. However, more advanced shading models and effects (per-vertex displacement, bump mapping, etc.) often require additional attributes not included in this list. Users can define arbitrary attributes by simply calling the new attrib*() methods, passing the name of the attribute and the desired values. Distinction needs to be made between attributes that store position-like quantities, which need to be transformed by the modelview and projection matrix, color attributes that are specified as integers containing the four RGB components, normal attributes that have to be normalized and transformed with the normal matrix, and generic attributes. These distinctions are shown in table 2. A useful feature in the renderer is that tessellation is applied automatically to all attributes, not only the default ones, before pushing the geometry to the GPU. The next program listing shows how to initialize mesh tweening using custom positional attributes in a PShape object:

```
PShader sh;
PShape grid;
void setup() {
  size(640, 360, P3D);
  sh = loadShader("frag.glsl",
                  "vert.glsl");
  shader(sh);
  grid = createShape();
  grid.beginShape(QUADS);
  grid.noStroke();
  grid.fill(150);
  float d = 10;
  for (int x=-width;x<width;x+= d){
    for (int y=-width;y<width;y+=d){
      for (int i=0; i<=1; i++){
        for (int j=0; j<=1; j++){
          int ij = j * (1-i) + 1 -
                   i*j;
          float n = noise(x + d *
                    ij, y + d * i);
          grid.fill(255 * n);
          grid.attribPosition(
            "tweened", x + d * ij,
            y + d * i, 100 * n);
          grid.vertex(x + d * ij,
            y + d * i, 0);
        }
      }
    }
  }
  grid.endShape();
}
```

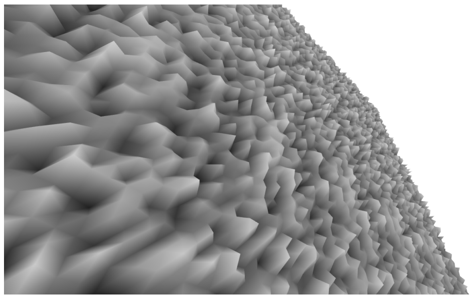
Figure 1: Output of the mesh tweening code above.

Table 1 lists all the new custom vertex attribute functions introduced in Processing 3.

2.2 Extending Pshader: Geometry and Tessellation Shaders

The OpenGL-based renderers in Processing 3 –P2D for 2-D rendering, and P3D for 3-D rendering– create an OpenGL 3.1 context on the desktop, and a GLES 2.0 context on mobile. In order to ensure compatibility across desktop, mobile and web platforms, the PShader class only exposes functionality to set the vertex and fragment stages,

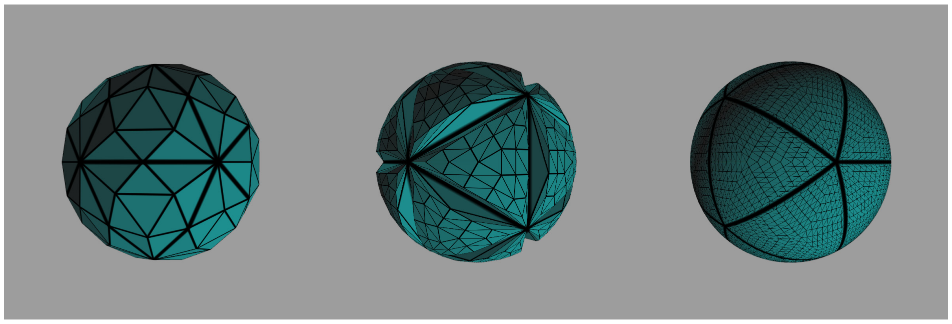

Figure 2: Sphere tessellation shader, adapted to run in Processing from the demo by Philip Rideout (http://prideout.net/blog/?p=48).

since geometry and tessellation shaders are not part of GLES 2 or WebGL 1. In order to relax this restriction, desktop users can subclass PShader and implement fully featured GPU pipelines including geometry and tessellation stages, as long as long as the underlying graphics hardware supports versions of OpenGL 3.1+ for geometry shaders and OpenGL 4.1 for tesselation shaders. Sample Processing sketches including geometry and tessellation shaders are available online (Colubri III), and Figure 2 shows the output of one of them. More examples can be found using the ShaderBase tool (Gomez et al., 2015).

3 SHADER API COMPARISON: OF AND CINDER

Most quantitative API comparisons involve some sort of usability study. For instance, (Stylos and Myers, 2008) performed a usability study to show that method placement can have large impact in object-oriented APIs. Another case study (Watson, 2009) describes how the application of technical communication skills and tools helped improve the usability and clarity of API by performing text analysis of the API elements. A rather simpler, but still meaningful approach requiring no usability study would be to assess API simplicity and flexibility from well-implemented tasks over a given domain (Kanat-Alexander et al., 2012). While API simplicity may be evaluated from the set of instructions needed to accomplish a standard task over a given domain, API flexibility is related to as whether or not an advanced task can be accomplished by the framework and if so, how.

As mentioned in the introduction, we will conduct our comparison by contrasting the shader APIs in Processing (Colubri I), OF (Karluk et al., 2013), and Cinder (Bell), due to the popularity of these frameworks among the creative coding community. Both OF and Cinder are based on the C++ programming language, and define an API encapsulating several high-performance, cross-platform libraries (OpenGL, FreeType, GLFW, etc.). Processing and OF were inspired respectively by the Design By Numbers project and the ACU C++ library, both developed at the Aesthetics and Computation group in the MIT Media Lab in the late 90s and early 2000s (OF community, 2015). In this sense, Processing and OF share a common origin, while Cinder is a more recent project started completely from scratch specifically to address the needs of high-performance interactive graphics using the most recent OpenGL features. In terms of difficulty, it is typically recognized that Processing is the most accessible of the three, followed by OF, and with Cinder being the harder to learn and closer to low-level APIs, most notably OpenGL (Brewis, 2014). It is also worth stating that Processing includes its own development environment and debugger, while OF and Cinder relying on third-party IDEs, such as Xcode, Microsoft Visual Studio, or Code::Blocks.

For our initial API comparison, we selected two standard tasks that involve the use of shaders: rendering of a simple geometric primitive using cartoon lighting (toon task), and rendering of a simple textured geometric primitive using standard per-pixel Blinn-Phong shading with a single light source (texlight task). Since all three frameworks share a similar setup/draw application structure, where the one-time initialization operations are performed inside the setup() function, and all rendering operations that need to be executed in every frame are placed inside the draw() function. Processing is notable for the absence of any other initialization/boilerplate code due to its pre-processor that simplifies the underlying Java code.

Table 2: Toon task in Processing and OF.

Processing	OF

`void setup() {`
`  size(640, 360, P3D);`
`  noStroke();`
`  fill(204);`
`  toon = loadShader("frag.glsl",`
`                    "vert.glsl");`
`  toon.set("fraction", 1.0);`
`}`
 	
`void draw() {`
`  shader(toon);`
`  background(0);`
`  float dirY = (mouseY / float(height) -`
`               0.5) * 2;`
`  float dirX = (mouseX / float(width) -`
`               0.5) * 2;`
`  directionalLight(204, 204, 204,`
`                   -dirX, -dirY, -1);`
`  translate(width/2, height/2);`
`  sphere(120);`
`}`
	`void ofApp::setup(){`
	`    ofBackground(0);`
	`    shader.load("shaders/vert.glsl",`
	`                "shaders/frag.glsl");`
	`    cam.setupPerspective();`
	`    ofFill();`
	`    ofSetColor(204);`
	`    ofEnableDepthTest();`
	`}`
	
	`void ofApp::draw(){`
	`    cam.begin();`
	`    shader.begin();`
	`    shader.setUniform1f("fraction", 1.0);`
	`    ofMatrix4x4 mv=ofGetCurrentMatrix(`
	`OF_MATRIX_MODELVIEW);`
	`    ofMatrix4x4 normMat =`
	`                ofMatrix4x4::`
	`getTransposedOf(mv.getInverse());`
	`    shader.setUniformMatrix4f(`
	`                "normalMatrix",`
	`normMat);`
	`    float dirY = (mouseY /`
	`          float(ofGetHeight()) - 0.5) *`
	`2;`
	`    float dirX = (mouseX /`
	`          float(ofGetWidth()) - 0.5) * 2;`
	`    lightDir.set(-dirX, -dirY, -1);`
	`    ofVec3f lightNorm =`
	`            ofMatrix4x4::transform3x3(`
	`            normMat, lightDir);`
	`    lightNorm.normalize();`
	`    shader.setUniform3f("lightNormal",`
	`   lightNorm.x,          lightNorm.y,`
	`lightNorm.z);`
	`    ofTranslate(ofGetWidth()/2,`
	`                ofGetHeight()/2);`
	`    ofDrawSphere(0, 0, 120);`
	`    shader.end();`
	`    cam.end();`
	`}`

OF and Cinder code invariably contains additional code that is not relevant to the rendering task, in the form of header files and class declarations. For the sake of the comparison, we ignored this extra code, and focused only on the body of the setup() and draw() functions.

For reasons of space we only show the program listings comparing the toon task between Processing and OF (Table 2), and the texlight task between Processing and Cinder (Table 3). The GLSL code is not included, because it is essentially identical for Processing, OF, and Cinder. The complete code of all these tasks in the three frameworks is available online (Colubri II).

Table 3: Texlight task in Processing and Cinder.

Processing	Cinder

`void setup() {`
`  size(640, 360, P3D);`
`  label = loadImage("lachoy.jpg");`
`  can = createCan(100, 200, 32, label);`
`  texlightShader =`
`loadShader("frag.glsl",`
 	
`"vert.glsl");`
`}`
 	
`void draw() {`
`  background(0);`
`  shader(texlightShader);`
`  pointLight(255, 255, 255,`
`             width/2, height, 200);`
`  translate(width/2, height/2);`
`  rotateY(angle);`
`  shape(can);`
`  angle += 0.01;`
`}`
	`void Ex_7_2_texpixlightApp::setup() {`
	`    setWindowSize(640, 360);`
	`    cameraFOV = 60;`
	`    cameraZ = 0.125f * getWindowHeight()`
	`    /tan(0.5f *`
	`glm::radians(cameraFOV));`
	`    cam.lookAt(vec3(0, 0, cameraZ),`
	`               vec3(0));`
	`    auto img =`
	`     loadImage(loadAsset("lachoy.jpg"));`
	`    label = gl::Texture2d::create(img);`
	`    pointLight[0] = 0;`
	`    pointLight[1] = -getWindowHeight();`
	`    pointLight[2] = 0;`
	`    shader = gl::GlslProg::create(`
	`      loadAsset("vert.glsl"),`
	`      loadAsset("frag.glsl"));`
	`    shader->uniform("texMap", 0);`
	`    can = createCan(100, 200, 32);`
	`    gl::enableDepthWrite();`
	`    gl::enableDepthRead();`
	`}`
	
	`void Ex_7_2_texpixlightApp::draw() {`
	`    gl::clear(Color(0, 0, 0));`
	`    gl::setMatrices(cam);`
	`    gl::ScopedModelMatrix modelScope;`
	`    mat4 tm = mat4(1.0);`
	`    tm[1][1] = -1; tm[3][1] = 1;`
	`    shader->uniform("textureMatrix",`
	`tm);`
	`    mat4 mv = gl::getModelMatrix();`
	`    vec4 lightPos = mv *`
	`                 vec4(pointLight,`
	`1.0f);`
	`    shader->uniform("lightPosition",`
	`                   lightPos);`
	`    gl::translate(0, -100, -3 *`
	`cameraZ);`
	`    gl::rotate(angle, 0, 1, 0);`
	`    gl::color(1, 1, 1);`
	`    gl::ScopedTextureBind tex0(label,`
	`0);`
	`    can->draw();`
	`}`

4 RESULTS

From a quick inspection of the code listings for toon and texlight, we can see that the Processing versions are consistently shorter, approximately by half. Since each line corresponds to one function call or variable assignment (separate vector component assignments in Cinder are considered one call), we can summarize the call counts in the table below:

Table 4: Call counts in the studied frameworks.

Task	Processing	OF	Cinder
Toon	12	22	17
TexLight	11	25	26

Further examination of the code reveals the reasons for the lower counts in Processing: first, some settings such as perspective projection and depth masking are enabled by default in Processing, while they need to be explicitly enabled in OF and Cinder. Secondly, OF and Cinder do not automatically send some uniforms to the shader (Castro, 2014) (Bell, 2015), specifically the light position and normal matrix in the case of OF, and the light position and the texture matrix in the case of Cinder. OF also requires explicit binding/unbinding of shader and texture, while Cinder needs a more careful setup of the transformation matrices. OF and Cinder online references indicate that neither handles lighting uniforms automatically. Processing re-implemented in its default shaders the standard lighting from the fixed pipeline in OpenGL 1.1 (Woo et al., 1997), where up to 8 lights could be defined simultaneously.

5 CONCLUSIONS

We described a shader API currently implemented in the Processing programming language. This API is publicly available, as it was first introduced in the version 2.0 of Processing in 2013, and refined and completed in version 3.0, released in late 2015. From the responses observed in several online forums, we infer that the API is being used by a fraction of the Processing users, but we do not know how large this group of users is, or how expert they are in OpenGL and GLSL shaders. We aim at quantifying these parameters in the near future, in order to better characterize the adoption of shader programming within the creative coding community.

We also compared the shader APIs in Processing, OF, and Cinder, and observed that they follow a similar structure where shader information is encapsulated in classes, and several uniform and attribute variables are automatically passed down to the pipeline. Processing sends a few additional uniforms, normal and texture matrices, as well as detailed lighting information. This, together with the automatic tessellation of custom vertex attributes, suggests a closer integration in Processing of the standard drawing API with the new shader functionality. However, this closer integration might come at the expense of API flexibility that is possible, and in fact sought after, in more advanced frameworks such as OF and Cinder. Future work will include in-depth study of API complexity and precise quantification of API flexibility using standard methodologies.

Our overarching goal is to propose new answers to the question of how to effectively incorporate shader programming into computational arts and design. This question could be answered in two stages. First, by creating mechanisms that incorporate shaders into the rendering paths of high-level frameworks, and second, by moving shader code writing and testing closer to the framework so that not only CPU/GPU code can be written side-by-side, but also high-level calls can be directly mapped onto the shader functions and variables. In this article, we addressed the first stage, and we plan to study the second stage in upcoming work.

ACKNOWLEDGEMENTS

The authors would like to thank Casey Reas and Ben Fry for their tireless work with Processing over the years, and also to acknowledge the huge contributions from all the members of the Processing, OF, and Cinder communities. AC would like to thank very specially Casey Reas, Ben Fry, and Daniel Shiffman for the numerous discussions in person and over email that led to the final form of the shader API in Processing.

REFERENCES

Angel, E. & Shreiner, D. 2012. Introduction to modern OpenGL programming. SIGGRAPH '12.

Ascioglu, S. OpenProcessing classrooms [Online]. Available: http://openprocessing.org/classrooms

Bell, A. OpenGL in Cinder [Online]. Available: https://libcinder.org/docs/guides/opengl/

Bell, A. 2015. OpenGL in Cinder 0.9.0 [Online]. Available: https://forum.libcinder.org/topic/opengl-in-cinder-0-9-0.

Bell, A., Nguyen, H., Eakin, R. & Houx, P. Cinder home page [Online]. Available: https://libcinder.org/

Brewis, D. 2014. Emerging Environments for Interactive Art Creation. Interactive Multimedia Conference. University of Southampton.

Castro, A. 2014. Of Shader given GLSL variables [Online]. Available: http://forum.openframeworks.cc/t/ofshader-given-glsl-variables/16567/2

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

Colubri, A II. PShader tutorials code samples [Online]. Available: https://github.com/codeanticode/pshader-tutorials.

Colubri, A III. PShader experiments [Online]. Available: https://github.com/codeanticode/pshader-experiments/

Colubri, A. 2008. HD (in) Processing. International Conference on Advances of Computer Entertainment Technology. Yokohama, Japan.

Colubri, A. & Fry, B. 2012. Introducing Processing 2.0. ACM SIGGRAPH 2012 Talks. Los Angeles, California: ACM.

Glazier, B. 1992. The "best principle": why OpenGL is emerging as the 3D graphics standard. Computer Graphics World, 15, 116.

Gomez, A. F., Charalambos, J. P. & Colubri, A. 2015. ShaderBase: a Processing Tool for Shaders in Computational Arts and Design.

Kanat-Alexander, M., Oram, A. & Hendrickson, M. 2012. Code Simplicity: The Fundamentals of Software.

Karluk, L., Noble, J. & Puig, J. 2013. Introducing Shaders [Online]. Available: http://openframeworks.cc/tutorials/graphics/shaders.html

Mccarthy, L. p5.js project [Online]. Available: http://p5js.org/

Nguyen, H. & Nakamoto, H. 2008. GPU gems 3, Upper Saddle River, NJ, Addison-Wesley.

OF community. OpenFrameworks home page [Online]. Available: http://openframeworks.cc/

OF community 2015. ofBook. Available: http://openframeworks.cc/ofBook/

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

Palop, B. 2015. Here's Everything Awesome About Processing 3.0 [Online]. The Creators Project blog. Available: http://thecreatorsproject.vice.com/blog/heres-everything-awesome-about-processing-30

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

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

Processing Foundation III. PShader reference [Online]. Available: https://processing.org/reference/PShader.html

Processing Foundation IV. PShape reference [Online]. Available: https://processing.org/reference/PShape.html

Reas, C. & Fry, B. 2006. Processing: programming for the media arts. AI & Soc, 20, 526-538.

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

Rost, R. J. 2009. OpenGL shading language, Upper Saddle River, N.J., Addison-Wesley.

Stylos, J. & Myers, B. A. 2008. The implications of method placement on API learnability. In: Harrold, M. J. & Murphy, G. C. (eds.) SIGSOFT '08/FSE-16.

Watson, R. B. Improving software API usability through text analysis: A case study. Professional Communication Conference, 2009. IPCC 2009. IEEE International, 19-22 July 2009 2009. 1-7.

Woo, M., Neider, J. & Davis, T. 1997. OpenGL programming guide (2nd ed.): the official guide to learning OpenGL version 1.1, Addison-Wesley Longman Publishing Co., Inc.