EPiC Series in Computing
Volume 64, 2019, Pages 88–97
Proceedings of 28th International Conference on Software Engineering and Data Engineering

A lightweight environment for 2D visual applications

Armando Arce-Orozco[1][2], Antonio Gonzalez-Torres[3][4], and Erick Mata-Montero[1]

[1] School of Computing, Costa Rica Institute of Technology
Cartago, Costa Rica
`[arce,emata]@tec.ac.cr`
[2] School of Computing, National University
Heredia, Costa Rica
`ararce@una.ac.cr`
[3] Dept. of Computer Engineering, Costa Rica Institute of Technology
Cartago, Costa Rica
`antonio.gonzalez@tec.ac.cr`
[4] Faculty of Engineering, ULACIT
San José, Costa Rica
`agonzalez@ulacit.ac.cr`

Abstract

People and organizations constantly use a wide variety of devices and formats to produce large volumes of data. This involves a series of challenges related to processing and transforming data into valuable knowledge to carry out informed decisions. This research work departs from the fact that information visualization is a critical element in the process of data analysis that takes advantage of the visual and cognitive abilities of people to explore, discover, interpret, and understand patterns in data with the use of visual representations and human-computer interaction techniques. This research proposes Diököl, a programming environment developed with Lua and OpenVG to facilitate the learning process of programmers with little experience in the implementation of visualizations. The environment was designed after the careful study of similar tools and the most popular visualization libraries. Its design takes into account their weaknesses and strengths to propose a set of features that can make it an efficient alternative to learning about and program visualizations. In addition to typical functionality provided by several visualization tools and libraries, Diököl is a lightweight environment that provides a simple and effective event manager, is scalable, small, portable and contributes an environment with a simple and easy to understand graphical interface and functionality, similar to the ones provided by Processing.

1 Introduction

In recent years there have been important changes in the way people and organizations create, process, and share information. On the one hand, individuals and organizations have increased the production of data in new and diverse formats thanks to new devices and the improvement of the processing and storage capacities of computers and servers. On the other hand, the generated data is shared in real time at higher speeds. As a result, the availability of data is higher than the processing capacity of individuals to transform it into knowledge, and this has overloaded the ability of people to filter out and understand the information available.

The transformation of the overwhelming volumes of data into useful knowledge involves a series of challenges related to the retrieval and integration of heterogeneous data sources, their processing, storage, and automatic analysis. The term *big data* is used frequently to refer to the administration of large amounts of data to facilitate informed decision making, according to the needs and requirements of organizations [9]. In this context, this work considers that the visualization of information is a critical element in the analysis process because it takes advantage of the visual and cognitive abilities of people to explore, discover, interpret and understand patterns in the data through the use of visual representations and human-computer interaction techniques [29, 11].

The need for data analysis has motivated the development of a considerable number of frameworks, environments, languages, and libraries. These tools have facilitated the implementation of visualization methods to support analysis and decision making based on large volumes of data [17, 27, 5], the creation of visualization boards for Industrial Internet of Things (IIoT) [10], and even to learn how to program in the context of the visual arts [25, 24]. In this scenario, organizations are demanding skilled personnel in advanced data analysis techniques that use machine learning, big data frameworks, information visualization, and visual analytics.

The main goal of this research is to propose a scalable programming environment to facilitate the teaching of information visualization concepts and the design of visualization tools, written by using a combination of intermediate and low-level programming languages. The result of this work is a lightweight programming environment called Diököl [1], which is structured to facilitate the learning of programming visualizations. The development of this environment was carried out with Lua and OpenVG. It is small (less than one megabyte), portable, efficient, fast, manages events straightforwardly, and has an easy to understand syntax, very similar to the one used by Processing [25, 24].

The rest of this paper discusses the general background of this research (Section 2), explains the design and elements that make up Diököl (Section 3), presents some applications of the environment (Section 4), and draws the main conclusions (Section 5).

2 Background

The graphics modes used by most libraries are the *retained* and *direct* mode or a combination of both. The programming of interactive 2D applications with an object-oriented approach is usually carried out in a retained mode. The libraries that use this mode require programmers to configure the scene in the initialization stage and create the figures using some editor. The graphic elements in the scene usually have associated details such as an identifier, element type (e.g., line or rectangle), position, colors, and styles.

The details of the graphing process are managed internally by the graphics library, thus, when a figure is selected, for example, by clicking on it, a callback is generated or a message is added to the message queue. This function analyzes the ID of the figure that created it to obtain its attributes, or modify an attribute of another character. The creation of custom figures and events usually involves the generation of new classes, which sometimes causes an explosion of categories and large amounts of complex code.

[1]Diököl means “image” or “soul of the eyes” in the indigenous language *Cabécar*, which is an ethnic group of Costa Rica.

The direct mode approach draws elements in frames by calling its drawing routine in each display cycle, and instead of assigning them a position, content or identifiers in the initialization stage, these are assigned in the method that draws graphic components. The same drawing routine can manage the events of a figure; for example, the code to handle the click event on a component can be included in the same function. This approach has the advantage that the code associated with the drawing and handling of events is in the same place, which keeps the code simple and easy to understand. This graphing mode does not need to maintain an object model, but any change in any object produces the redrawing of all figures on the screen. Hence, it is generally considered more inefficient than the retained mode because the screen must be updated continuously.

Based on the granularity of their graphic commands (graphic primitives), 2D graphics libraries can be classified as low, middle or high level. Low-level libraries have fine granularity commands such as *moveTo*, *lineTo* and *curveTo*. These commands specify areas of the same figure and are made up of what is generally known as a *path*. They rely on the use of direct mode and can be accelerated by hardware, although this usually does not include interaction capabilities and depend upon other software components to perform event handling. These libraries do not have support for scripting languages and must be linked to high-level languages, which limits their portability among platforms. Examples of this type of libraries include Cairo[2], Skia[3], OpenVG[4], and Canvas HTML 5.

Middle-level libraries permit the definition of complete figures of an intermediate level of granularity, such as *rect*, *ellipse* and *line*, and allow the combination of the use of the retained and direct graphic modes and the management of interactions in the same environment. Some graphics libraries in this group are Java2D [15], Adobe Flash [20], Silverlight [23], SVG [4], and Processing [24]. Lua is lightweight, multi-paradigm programming language [14] that can also be placed in this category.

Libraries in the high-level category use complex objects for drawing sophisticated graphics such as *plots*, *graphs* and *maps*, and include a scripting language, which makes them portable among platforms. These toolkits are often used to create statistical graphs and visualizations, and use the retained mode and complex data models. Some examples are D3js [3], Prefuse [12], Matplotlib [13] and R Graphics [18].

Three of the most popular programming environments for graphics nowadays, are Processing, Lua, and Open VG, which are explained in the following paragraphs.

Processing is an open source project that was designed to simplify the creation of interactive visual applications by visual artists and non-programmers as an alternative to proprietary tools, such as Adobe Flash. Casey Reas and Ben Fry developed this environment at MIT under the supervision of John Maeda, but currently, a group of developers is responsible for making frequent updates. The original goal of its creators was to enable designers and artists to carry out experiments with a set of simple commands and scripts, without knowing computer programming.

Processing includes an integrated development environment (IDE), a programming language designed to simplify the programming of visual designs and tools to publish applications on the Internet or as desktop tools. The success of Processing has stimulated the appearance of multiple variants such as Processing.js [22], OpenFrameworks [21], Cinder [26], and P5.js [16].

Processing.js and P5.js allow the generation of interactive graphics for the web, but the main difference is that the former uses a syntax similar to that of Processing and permits the

[2]https://cairographics.org
[3]http://skia.org
[4]http://www.khronos.org/openvg

execution of most of the programs written in Processing, while P5.js uses Javascript syntax. OpenFrameworks and Cinder use a set of drawing routines similar to those of Processing, but with C++.

Although Processing uses a simplified Java syntax to facilitate the learning of the environment, this approach has some disadvantages. The most important one is the use of static types in Java, which is not always easy to use for beginner programmers. Furthermore, Processing requires to run the Java virtual machine (JVM) and needs 150 MB of space to install the Java Runtime Environment (JRE).

Lua (moon in Portuguese) is an interpreted language created in 1993 in Brazil to offer an extensible and portable environment [14]. Its semantics is simple, does not require to memorize many commands, and provides efficient performance. In addition, it is a dynamic typing language, uses automatic memory management, and incremental garbage collection.

Lua is open source, distributed under the MIT license and written with ANSI C. It can be ported to GNU/Linux, Windows, and MacOS X. The size of the library that contains the interpreter for all platforms is small (approximately 500 Kb). In addition, its performance can be improved with the use of the Just-In-Time (JIT) compiler, called LuaJIT[5], what makes it, according to its authors, 20 times on average faster than running Lua in the interpreted mode.

Even though Lua was originally developed for embedded systems, it is one of the preferred languages for the development of video games [28], although it is also used for 2D graphics programming. Some video games that have been programmed in this environment are Löve [1], Corona SDK [6] [7], and Cocos2d [19].

OpenVG started as a standard for native graphics in embedded systems and has a similar low-level structure as OpenGL ES with interfaces for vector graphics libraries. It was designed as a cross-platform, non-copyrighted API to perform rasterization in dedicated graphics hardware and to operate on mobile devices that have limited CPU resources.

The basic drawing primitive of OpenVG is *path*, which can contain both straight line segments and Bezier line segments. Primitives like *path* can describe arbitrary polygons, which can be filled with solid colors, gradients, bitmap images, or patterns made from other 2D objects.

Multiple libraries implement the OpenVG standard in different ways. AmanithVG[6] is a commercial library that has two implementations of OpenVG, one that is executed by OpenGL and the other using its rasterization engine. ShivaVG[7] is an open source library that uses OpenGL 1.1 to implement this graphics library on desktop environments (Linux, MSWindows, and MacOS), and MonkVG[8] uses OpenGL ES 2.0 to generate graphics on mobile devices (iOS and Android).

3 The Diököl programming environment

Diököl is an open source 2D graphics environment that provides commands in direct mode and uses Lua as a scripting language. It is available at http://github.com/arce/Diokol. Lua translates each call to drawing commands into its equivalent in OpenVG, although this conversion is more straightforward for some commands than others. Besides, this environment also minimizes the transformation of data structures between Lua and OpenVG library, such that when it loads an image or font from a file, the storage is done directly in OpenVG. The above

[5]http://luajit.org/luajit.html
[6]http://www.amanithvg.com
[7]http://github.com/ileben/ShivaVG
[8]http://github.com/micahpearlman/MonkVG

is important because it improves the execution of programs as interpreted environments are slower than the compiled alternatives.

The primitive graphics of Diököl are similar to those of environments such as Cairo, Canvas HTML, and Processing, which include commands for the drawing of figures such as rectangles (rect), ellipses (ellipse), arcs (arc), lines (line) and points (point). It also provides additional commands to define visual characteristics such as colors (*fill*, *stroke*), edge thickness (*strokeWeight*), line finishing styles (*strokeCap* and *strokeJoin*), or transparency of the figures (*blend*). Furthermore, this environment supports transformations, which permit to apply rotations (*rotate*), scaling (*scale*), displacement (*translation*), and stretching (*shear*) to the figures. It also allows to store and recover the matrix of transformations that are applied to the figures employing the commands *pushMatrix* and *popMatrix*.

This environment includes a mechanism to create composite figures such as "paths" and "shapes" combining the commands *beginShape* (similar to Processing), *vertex* and *endShape*. After creating the figures, the command *saveShape* can be issued to store them using *OpenVG*, and get their identifiers, which can be used later to draw the shapes, as many times as needed. The storage of figures in *OpenVG* limits the garbage collection system of LUA to free up the memory and causes to manually free out the memory used by the images that are no longer needed. This task is performed using the command *destroyShape*.

Other commands supported by Diököl are *loadImage* and *image* to read images from files and display them on the screen in different sizes and positions, and similarly to the shapes, a unique identifier is assigned to each image to use it as a reference during the execution of operations. The environment uses "stb image"[9], a public domain library, to read traditional image formats (jpeg, png, gif). This library is small and can be embedded into applications without requiring additional library files and facilitating the portability of the environment between different platforms.

Diököl manipulates pixels individually with commands such as *createImage*, *loadPixels* and *updatePixels*, which are similar to the methods used by Processing. However, the dimensions of images are obtained with special operations such as *imageWidth*, *imageHeight*, and *pixelsLenght* (not present in Processing), since these are internal entities of the environment and not Lua objects.

The efficient deployment of texts is one of the most complex features to implement in many graphic environments. The mechanism used to specify the shape of each letter, along with the algorithms to achieve a good definition of them presents significant challenges. A format widely used to specify the glyph of the characters is True Type Font (TTF), and there are multiple libraries that allow to read and process this type of format (e.g., FreeType). The method used by Diököl read the files of character fonts in TTF format with the library "stb truetype", because it is a smaller library than other alternatives for manipulating TTF files and can be embedded in executable programs.

Diököl has a default font that is embedded in programs if programmers do not specify the type that they want to use. However, a user can import fonts from any TTF file in bitmap and vector mode. The *loadFont* function loads bitmap fonts that are faster and more efficient to display but produce problems when the text size is changed. Addtionally, the *createFont* function works with fonts in vector mode that is more appropriate when the text size needs to be modified, but it uses more computational power than the alternative in bitmap mode.

Diököl defines two models for interaction management. The basis of the first model is the direct mode, which dissociates the events from the figures, and the programmer outlines the logic to identify the triggering of an event on a figure. The second model is a new proposal

[9]http://github.com/nothings/stb

made in our environment that needs the intervention of a developer if an event is fired when a figure is drawn. This model is called *draw and evaluate*.

The direct mode technique defines a callback function that is executed each time an event occurs when a mouse click or a keyboard key is pressed. The programmers are responsible for determining the coordinates of the events and verify if these match the position of a figure, as it is shown in Listing 1. However, some of the disadvantages of this method are:

- The programmer must write the logic to evaluate the position of the mouse.
- Mouse events must be passed by the execution environment to the application, even when an event has no relevance and, the performance of the program may be affected.

The drawing function in the *draw and evaluate* model besides creating the graphic elements also verifies the triggering of events to return a data structure with the coordinates of the position where it was fired. The commands *event(eventCode)* and *noEvent()* control when events must be evaluated and the event types accepted by *event(eventCode)* are MOVED, DRAGGED, PRESSED, RELEASED, and CLICKED (see Listing 2).

Mouse events in this model are maintained at the execution level of the environment and do not have to be manipulated by the application or the programmers. Additionally, developers do not have to worry about the logic to determine if the mouse is over a figure or to get the coordinates of the event because the execution environment performs this computation. However, a disadvantage of this model is that the old code may require significant modifications to be adapted to this mechanism.

The handling of events in Diököl is easy to learn because each shape can verify the occurrence of an event on it. This requires to specify the type of event that each figure must listen to when it is triggered. Then, once this is carried out the figure returns an object associated with the information of the detection of the mouse on the different figures. The code for these event models are displayed side by side in Listing 1 and Listing 2 to have a contrasting view of both of them.

```
local mouseX,mouseY
function setup()
  size(600,600)
end
function draw()
  rect(100,100,200,200)
  if (mouseX>100 and mouseX<300) and
    (mouseY>100 and mouseY<300) then
      print("Clicked !!")
  end
end
function mouseClicked(x,y)
  mouseX = x
  mouseY = y
end
```

Listing 1: Events in direct mode

```
function setup()
  size(600,600)
end
function draw()
  event(CLICKED)
  local evt = rect(100,100,200,200)
  if evt then
    print("Clicked !!")
  end
end
```

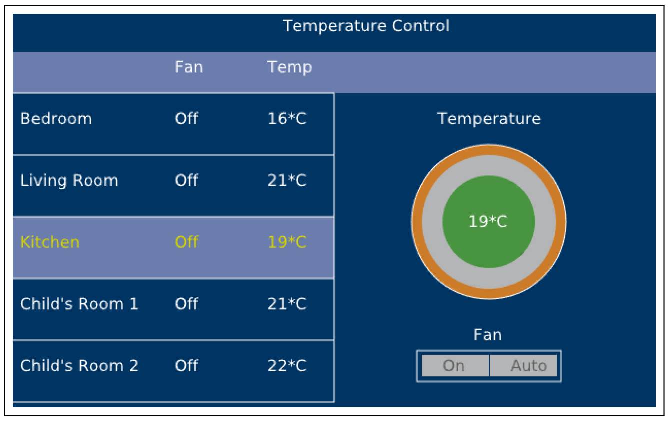
Listing 2: Draw and evaluation

4 Applications

4.1 User interfaces in embedded devices

User interfaces for embedded systems require defining various types of widgets for graphic elements, such as circles, arcs and rectangles, the inclusion of text and images, transformations, and the highlighting of objects. These systems can use 2D graphics libraries by software or accelerated. Some examples of software libraries are DirectFB, Qt-E, and NanoX which are optimized for 2D graphics deployment, but require more CPU time and memory space, whereas OpenGL and DirectX, are acceleated APIs which implement these capabilities using GPUs without requiring additional memory. However, accelerated libraries usually consume more considerable resources than software methods to display the graphics primitives of the interfaces.

Diököl is an alternative for the development of user interfaces in embedded systems [2]. Because OpenVG runs on Raspberry Pi devices in accelerated mode, this improves the execution times. Some applications for home automation, personal training, and control of home appliances have been programmed using this graphics environment. Figure 1 shows an example of the use of this environment for embedded software.



	Fan	Temp
Bedroom	Off	16*C
Living Room	Off	21*C
Kitchen	Off	19*C
Child's Room 1	Off	21*C
Child's Room 2	Off	22*C

Figure 1: Home automation application

4.2 Maps

Diököl was tested in the Geographic Information Systems course that is taught in the Computing Bachelor's degree at the Costa Rica Institute of Technology. Previous versions of the course used Leaflet, GoogleMaps, and OpenLayers to develop geographic applications. However, the learning curve of these libraries was quite high and required students to devote many hours to program basic functionalities. The use of Diököl in this course made it easier for students to design complex applications of georeferenced visualization with the writing of few lines of code.

```
local csv = require("csv")
local mapImage
local locations
local randoms = {}
function setup()
    size(640,400)
    local font = createFont("Vera.ttf",12)
    textFont(font)
    mapImage = loadImage("data/map.png")
    locations = csv.open("data/locations.tsv")
    local data = csv.open("data/randoms.tsv")
    for fields in data:lines() do
        randoms[fields[1]]=fields[2]
    end
end
```

Listing 3: Variables definition and map loading

The first steps to annotate a map is to load the image, font and data to be plotted on the figure, as it is shown in Listing 3. Thereafter, the background of the drawing is configured, the image is rendered, the events are setup and the annotations, in the form of ellipses, are put on the top of the map. Figure 2 shows a map with visual annotations made with Diököl, whereas Listing 4 displays the corresponding code, which is an adaptation in Diököl of an example taken from the book "Visualizing Data" [8].

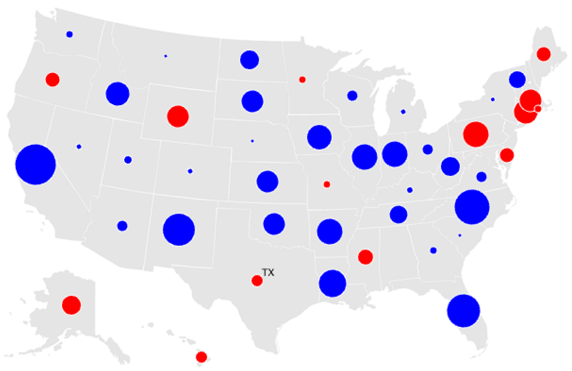


Figure 2: Interactive map

```
function draw()
  background(255)
  image(mapImage,0,0)
  event(MOVED)
  for fields in locations:lines() do
    local x=fields[2]
    local y=fields[3]
    local v=tonumber(randoms[fields[1]])
    if (v<0) then
      fill("#FF0000")
    else
      fill("#0000FF")
    end
    local evt = ellipse(x,y,v*4,v*4)
    if evt then
      fill(0)
      text(fields[1],x+5,y-5)
    end
  end
end
```

Listing 4: Source code for the map

4.3 Graphics, visualizations and animations

Diököl can also be used to draw complex graphics, visualizations and animations in just a few lines of code, using a similar syntax to the one of Processing. So, Figure 3 shows a screenshot of an animation that was originally written in Processing, and which code is displayed in Listing 5. Note that the complexity of the graphic does not correspond with the simplicity of the code.

5 Conclusions

This paper presented Diököl, a new environment for the programming of interactive 2D visualizations with Lua, an easy to learn programming language that uses dynamic typing. The environment is small and lightweight, compared to similar tools and incorporates a new mechanism to manage events. This mechanism has a structure that is simple and easy to understand that facilitates the writing of programs and managing of the interaction between the users and the applications with the environment.

Applications written for Diököl can be executed in different operating systems, without the need for any changes. The tests carried out in the lab have been successful and showed that it is a good alternative as a multiplatform environment to create compelling visualizations. Moreover, the number of lines of code needed for creating visualizations is less in comparison to other similar environments, because of its simplicity.

Future work involves real-world testing with students and the creation of different editions for the design and development of visualization tools for collaborative analysis in distributed work environments and open hardware devices. Therefore, new functionality will be added for

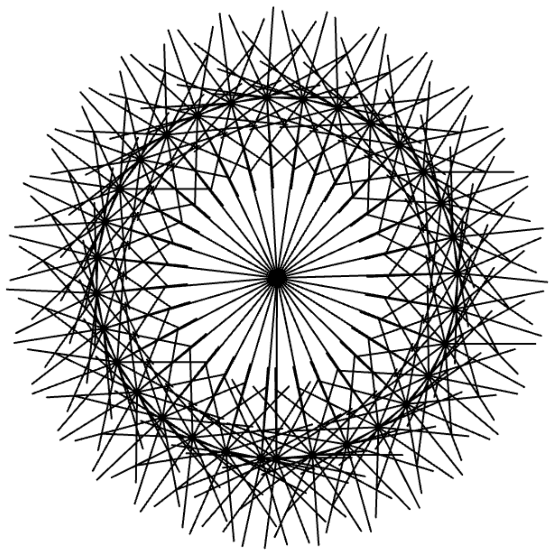


Figure 3: Graphics animation

```
local theta = 0;
function setup()
  size(480, 270);
end
function draw()
  background(255);
  stroke(0);
  translate(width()/2, height()/2);
  for i = 0.0,TWO_PI,0.2 do
    pushMatrix();
    rotate(theta + i);
    line(0, 0, 100, 0);
    for j = 0.0,TWO_PI,0.5 do
      pushMatrix();
      translate(100, 0);
      rotate(-theta-j);
      line(0, 0, 50, 0);
      popMatrix();
    end
    popMatrix();
  end
  theta = theta + 0.01;
end
```

Listing 5: Graphics code

the creation of linked and coordinated views through a mechanism of remote communication between devices (tablets, desktop computers, projectors, wall displays, and tabletops). Furthermore, we are considering the creation of other versions of Diököl for mobile environments such as Android and iOS.

In addition, new drawing commands will be incorporated to receive arrays of values and not just individual items as it is currently the case with Processing and its variants. This diminishes the number of calls between the interpreted environment and OpenVG. By incorporating functions for *points*, *lines* and *ellipses*, similar to those used by R, to improve the performance when displaying large amounts of data.

References

[1] D.D. Akinlaja. *LÖVE2d for Lua Game Programming.* **. Packt Publishing, 2013.

[2] A. Arce-Orozco and A. González-Torres. A Graphic Environment for User Interfaces in Embedded Devices. In *2018 IEEE 38th Central America and Panama Convention (CONCAPAN XXXVIII)*, pages 1–6, Nov 2018.

[3] M. Bostock, V. Ogievetsky, and J. Heer. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec 2011.

[4] K. Cagle. *SVG Programming: The Graphical Web.* Apress, 2008.

[5] K. Cherven. *Mastering Gephi Network Visualization.* Community Experience Distilled. Packt Publishing, 2015.

[6] Silvia Domenech. *Create Mobile Games with Corona: Build with Lua on iOS and Android.* Pragmatic Bookshelf, 1st edition, 2013.

[7] M.M. Fernandez. *Corona SDK Mobile Game Development: Beginner's Guide.* Learn by Doing : less Theory, more Results. Packt Publishing, Limited, 2012.

[8] Ben Fry. *Visualizing Data*. 1st edition, 2008.

[9] Amir Gandomi and Murtaza Haider. Beyond the Hype: Big Data Concepts, Methods, and Analytics. *International Journal of Information Management*, 35(2):137 – 144, 2015.

[10] GE Digital. PREDIX Design System, aug 2018.

[11] Antonio González-Torres, Francisco J. García-Peñalvo, Roberto Therón-Sánchez, and Ricardo Colomo-Palacios. Knowledge Discovery in Software Teams by Means of Evolutionary Visual Software Analytics. *Science of Computer Programming*, 121(C):55–74, June 2016.

[12] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: A Toolkit for Interactive Information Visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 421–430, New York, NY, USA, 2005. ACM.

[13] J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science Engineering*, 9(3):90–95, May 2007.

[14] R. Ierusalimschy. *Programming in Lua*. Roberto Ierusalimschy, 2006.

[15] J. Knudsen. *Java 2D Graphics*. Java series. O'Reilly, 1999.

[16] L. McCarthy, C. Reas, and B. Fry. *Getting Started with P5.js: Making Interactive Graphics in JavaScript and Processing*. Maker Media, Incorporated, 2015.

[17] E. Meeks. *D3.js in Action*. Manning, 2015.

[18] P. Murrell. *R Graphics, Second Edition*. Chapman & Hall/CRC The R Series. CRC Press, 2016.

[19] P. Nygard. *Creating Games with Cocos2d for IPhone 2*. Community experience distilled. Packt Publishing, 2012.

[20] R. Penner. *Robert Penner's Programming Macromedia Flash MX*. Application Development Series. McGraw-Hill/Osborne, 2002.

[21] Denis Perevalov and Igor (Sodazot) Tatarnikov. *openFrameworks Essentials*. Packt Publishing, 2015.

[22] Semmy Purewal. Introductory Programming Concepts with Processing.Js. *J. Comput. Sci. Coll.*, 29(2):199–202, 2013.

[23] D. Rader, J. Beres, J.A. Little, and G. Hinkson. *Silverlight 1.0*. Wrox Programmer to Programmer. Wiley, 2007.

[24] C. Reas, B. Fry, and J. Maeda. *Processing: A Programming Handbook for Visual Designers and Artists*. Putty-clay and Investment: a Business Cycle Analysis. National Bureau of Economic Research, 2007.

[25] Casey Reas and Ben Fry. Processing: Programming for the media arts. *AI Soc.*, 20(4):526–538, August 2006.

[26] K. Rijnieks. *Cinder: Begin Creative Coding*. Community experience distilled. Packt Publishing, 2013.

[27] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, Jan 2017.

[28] P. Schuytema and M. Manyen. *Game Development with Lua*. Charles River Media game development series. Charles River Media, 2005.

[29] Uthayasankar Sivarajah, Muhammad Mustafa Kamal, Zahir Irani, and Vishanth Weerakkody. Critical Analysis of Big Data Challenges and Analytical Methods. *Journal of Business Research*, 70:263 – 286, 2017.