Development Environment of Embeddable Information-Visualization Methods

Takao Ito[1](✉) and Kazuo Misue[2]

[1] Department of Computer Science, University of Tsukuba, Tennodai, Tsukuba, Ibaraki 305-8573, Japan
`ito@vislab.cs.tsukuba.ac.jp`

[2] Faculty of Engineering, Information and Systems, University of Tsukuba, Tennodai, Tsukuba, Ibaraki 305-8573, Japan
`misue@cs.tsukuba.ac.jp`

Abstract. The development of information-visualization systems requires the design of visualization methods based on data and purposes. Visual tools are desirable solutions supporting the development of visualization programs. However, the applicability of the tools is limited. Therefore, we allowed the developers to embed visualization methods into visualization programs easily to increase the application field. We designed a visualization execution environment that includes the following features: (1) Independence of data formats and Graphics APIs used in target programs, (2) Embeddability of visualization methods into visualization programs, and (3) Low-cost implementation of interface functions. We showed that our execution environment had practical performance through two experiments. Then, using use cases, we showed that our environment could be used in the low-cost development of visualization systems. The design of our execution environment reinforced the practicability of visual tools that support the development of visualization programs.

Keywords: Visualization · Implementation

1 Introduction

Information visualization is used in various applications, such as searching, monitoring, and analyzing data. The demand to use information visualization increases according to the increase in data size. To utilize information visualization effectively, developers have to design appropriate visualization methods for target systems.

The number of appropriate visualization methods for a certain purpose or data is limited. Then, the effectiveness of a visualization method is difficult to determine before visualization results are observed, which requires implementation of visualization methods. Therefore, developers are required to have many trial-and-error attempts involving repeated implementation of these methods. As a countermeasure, visualization systems are often developed according to the following processes: developers implement many visualization methods on a prototype environment, and appropriate methods are implemented into a target program. However, we believe that the implementation costs would prevent the utilization of visualization.

This study aims to reduce the implementation costs of visualization methods to promote their utilization. We developed a visual tool to support implementation of visualization methods, because use of visual tools is one of good solutions to reduce the implementation costs [1,2]. We supported prototyping by building a development environment called Iv Studio (Fig. 1) with a data flow visual language (DVL) for information visualization [3]. However, we need to enhance the following for improving the practical applicability of Iv Studio.

(i) Embeddability of visualization methods into programs that require visualization features.
(ii) Extensibility of the DVL parts.

We assumed that visualization features are added to existing programs. To enable the addition of visualization features to existing programs, (i) is important. Then, various visualization methods are required in practical use, which makes (ii) also important. Our objective is to enable developers to embed easily the developed methods by Iv Studio into their programs even if parts of the DVL are extended.

We consider various data formats, description languages, and Graphics APIs because developers are assumed to add visualization features to existing programs. Developing runtime libraries that support all environments is ideal, but it is not realistic. Thus, we designed an embeddable visualization execution environment that includes the following features:

1. Independence of data formats and Graphics APIs used in target programs.
2. Executing visualization methods on the embeddable virtual machine of the script language.
3. Exporting developed visualization methods as source codes.
4. Low-cost implementation of interface functions.

This study provides an implementation guide for development environments of visualization methods not only for Iv Studio but also for other environments.

2 Target Environments

Visualization programs require features that include loading data and showing images. Additionally, receiving inputs from viewers is necessary to support interactive visualization. We aim to execute visualization methods developed by Iv Studio in the following environments:

- Programming language
    - C, Java, JavaScript, C#

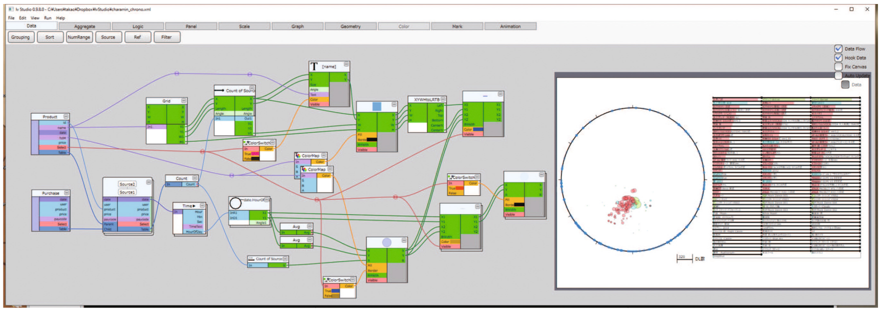


Fig. 1. Screen shot of Iv Studio.

- Data format
    - CSV, Spreadsheet, SQL, Original formats, on memory, etc.
- Graphics API
    - GDI, Processing, Java2D, Canvas, OpenGL, DirectX, etc.
- GUI system (GUI toolkit)
    - WIN32 API, Cocoa, X, Swing, Qt, etc.

3 Design of the Execution Environment

We want to support various environments considering Graphics APIs and GUI Systems. Developing runtime libraries that support all environments is ideal, but it is not realistic. Therefore, we designed an execution environment according to the following procedure:

1. Separating dependent parts from independent parts.
2. Choosing an execution environment of independent parts and adding an export feature to Iv Studio.
3. Designing interface functions between dependent and independent parts such that the implementation costs are reduced.

3.1 Separating Dependent Parts from Independent Parts

We investigated the support range of execution environments by considering the dependent/independent parts of visualization execution systems. Data formats, Graphics APIs, and GUI systems were considered. Visualization systems are formed similar to the information visualization reference model [4]. Therefore, based on the model, we considered parts that depend on Data formats, Graphics APIs, and GUI systems (Fig. 2).

Raw Data are just Data format; therefore, Data Transformations depend on Data formats. Rendering is the process of showing Views on displays, which is dependent on Graphics APIs. Considering interactive visualization, the process of receiving inputs from viewers depends on the GUI systems. Conversely, Visual

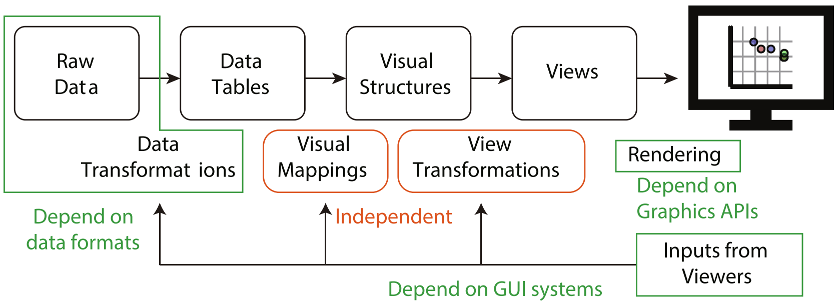


Fig. 2. Separation of dependent parts from independent parts.

Mappings and View Transformations are independent from the target programs because they are pure logical operations.

Based on the above consideration, we decided that Data Transformations, Rendering, and the process of receiving inputs from viewers are executed by the target programs. Then, Visual Mappings and View Transformations are executed by our designed execution environments. Figure 3 illustrates the execution image, which is realized as follows.

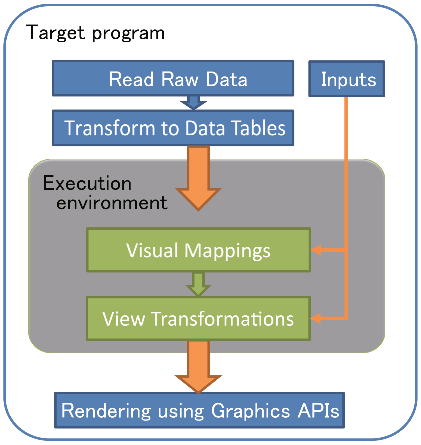


Fig. 3. Execution image of visualization.

- Choosing an execution environment.
- Developing a feature to export visualization methods to the execution environment.
- Designing interface functions that receive Data Tables and inputs from viewers and send graphics parameters to Rendering.

3.2 Choice of Execution Environment

The requirements of execution environments of Visual Mappings and View Transformations are as follows:

R1. Target programs can call the processes easily.
R2. Functions implemented in the target programs can be registered into execution environments easily.
R3. Execution environments are capable of general-purpose computing.

R1 and R2 are requirements for easy embedding of execution environments into target programs. R3 is a requirement for extensibility of the parts of the DVL. In this study, we aimed to extend the parts of the DVL by general-purpose programming languages, which would require general-purpose computing.

We considered three strategies to satisfy these requirements as follows.

[Use of virtual machines]

In this strategy, a runtime executed on JavaVM or .NET Frameworks is provided, and data and graphics parameters are communicated to target programs by sockets or interprocess communication. By exporting visualization methods from Iv Studio as a dynamic link library that can be linked to the runtime, the visualization methods can be executed, as if the visualization methods were embedded into target programs. However, this strategy is not enough to R1 and R2 because developers have to implement booting the runtime process and communicating to the runtime. Additionally, communication to web pages (the program in JavaScript) is difficult to implement.

[Exporting source codes in C]

In this strategy, source codes in C are exported from Iv Studio. This strategy reduces implementation costs of booting and communicating. Programs written in C can use the source codes directly. Programs written in Java or C# can use the source codes with native interfaces, such as Java Native Interface. Programs written in JavaScript can use the source codes with Emscripten[1]. However, this strategy also remains difficult for R1 and R2.

[Use of embeddable script languages]

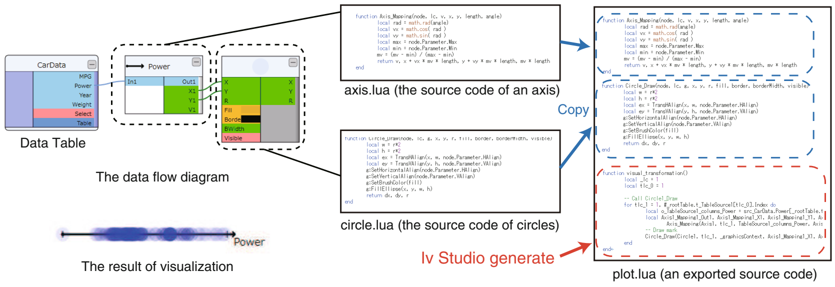
In this strategy, embeddable script languages of which execution environments are provided as software libraries are used. Visualization methods can be used from various programs by exporting source codes in a script language from Iv Studio and executing the source codes. Embeddable script languages are well designed to communicate to target programs; therefore, calling and registering functions are easier than the strategy of exporting source codes in C.

[1] http://emscripten.org.

Based on the above consideration, we adopted the use of embeddable script languages. There are some embeddable script languages, such as Lua[2], Squirrel[3], AngelScript[4], and Xtal[5]. We adopted Lua because of its implementations in C, Java, .NET, and JavaScript. Accordingly, Lua was also adopted to the description language of the parts of the DVL.

3.3 Exporting Feature from Iv Studio

We developed a feature that exports source codes in Lua from Iv Studio to execute visualization methods on LuaVM, which is the execution environment of Lua. The processes of the parts of the DVL were described in Lua, and we enabled Iv Studio to generate source codes to call the processes according to a dataflow diagram. Then, we enabled Iv Studio to export a visualization method as a single Lua script file by integrating generated codes and codes of the parts of the DVL. Figure 4 illustrates the procedure of exporting source codes.



axis.lua (the source code of an axis)

```
function Axis_Mapping(node, lc, v, x, y, length, angle)
        local rad = math.rad(angle)
        local vx = math.cos( rad )
        local vy = math.sin( rad )
        local max = node.Parameter.Max
        local min = node.Parameter.Min
        mv = (mv - min) / (max - min)
        return v, x + vx * mv * length, y + vy * mv * length, mv * length
end
```

circle.lua (the source code of circles)

```
function Circle_Draw(node, lc, g, x, y, r, fill, border, borderWidth, visible)
        local w = r*2
        local h = r*2
        local ex = TransHAlign(x, w, node.Parameter.HAlign)
        local ey = TransVAlign(y, h, node.Parameter.VAlign)
        g:SetHorizontalAlign(node.Parameter.HAlign)
        g:SetVerticalAlign(node.Parameter.VAlign)
        g:SetBrushColor(fill)
        g:FillEllipse(x, y, w, h)
        return dx, dy, r
end
```

plot.lua (an exported source code)

```
function Axis_Mapping(node, lc, v, x, y, length, angle)
        local rad = math.rad(angle)
        local vx = math.cos( rad )
        local vy = math.sin( rad )
        local max = node.Parameter.Max
        local min = node.Parameter.Min
        mv = (mv - min) / (max - min)
        return v, x + vx * mv * length, y + vy * mv * length, mv * length
end
function Circle_Draw(node, lc, g, x, y, r, fill, border, borderWidth, visible)
        local w = r*2
        local h = r*2
        local ex = TransHAlign(x, w, node.Parameter.HAlign)
        local ey = TransVAlign(y, h, node.Parameter.VAlign)
        g:SetHorizontalAlign(node.Parameter.HAlign)
        g:SetVerticalAlign(node.Parameter.VAlign)
        g:SetBrushColor(fill)
        g:FillEllipse(x, y, w, h)
        return dx, dy, r
end
function visual_transformation()
        local _lc = 1
        local tlc_0 = 1

        -- Call Circle1_Draw
        for tlc_1 = 1, #_rootTable.t_TableSource1[tlc_0].Index do
                local o_TableSource1_columns_Power = src_CarData.Power[_rootTable.t
                local Axis1_Mapping1_Out1, Axis1_Mapping1_X1, Axis1_Mapping1_Y1, A
                        Axis_Mapping(Axis1, tlc_1, TableSource1_columns_Power, Axis
                -- Draw mark
                Circle_Draw(Circle1, tlc_1, _graphicsContext, Axis1_Mapping1_X1, A
        end
end
```

Fig. 4. Procedure in exporting source codes.

Iv Studio exports only source codes for Visual Mappings. For other processes, such as View Transformations and receiving inputs from viewers, a support library in Lua was provided.

Developers can execute visualization methods developed by Iv Studio on target programs by linking Lua libraries to the programs and executing exported script files and the support library. Additionally, in our environment, even if target programs are running, developers can update the visualization methods, which is highly advantageous in the development of visualization programs because it is assumed that fixing visualization methods is a repetitive process.

3.4 Design of Interface Functions

For easy embedding, the implementation costs of interface functions between target programs and LuaVM must be reduced. Therefore, we reduced the number

[2] https://www.lua.org/.
[3] http://squirrel-lang.org/.
[4] http://www.angelcode.com/angelscript/.
[5] https://code.google.com/archive/p/xtal-language/.

of essential functions requiring implementation. Additionally, we have shortened the implementation of the functions.

The functions were designed as low-level commands to reduce the number of functions. The implementation costs of each function were also reduced because we designed the functions such that they could be implemented using only the commands of most programming languages and Graphics APIs. Additionally, the use of arrays for arguments and return values was avoided because sending or receiving arrays required complex descriptions.

Table 1 shows the specifications of the functions to send Data Tables to LuaVM. The functions are called from LuaVM. Practically, the functions for reading files or HTTP connections are provided, such as ivs_OpenFile, ivs_ReadLine, and ivs_Close. In Table 1, names that express their operations are shown for explanation. Sending Data Tables to LuaVM is executed by implementing the functions in programming languages used in target programs and registering them to LuaVM. The implementation of the functions is simple. For example, in C, developers can implement reading CSV or TSV files by calling fopen, fread, and fclose in ivs_OpenFile, ivs_ReadLine, and ivs_Close. Developers can implement not only reading files but also connections to SQL databases, by executing SQL queries in ivs_OpenFile and returning one record in ivs_ReadLine.

Table 1. Functions for sending Data Tables.

Functions	Specification
obj OpenTable(filename, format)	A file name and a format (CSV or TSV) are received, and a reference object is returned
string ReadRecord(obj)	A string that expresses one record in the format of CSV or TSV is returned
void CloseTable(obj)	A table expressed by an argument is closed

Table 2 shows the specifications of the essential draw functions that developers must implement and register. The upper four functions are for setting colors or font sizes. The lower five functions are for drawing or painting figures. There is a design choice where the upper four functions are integrated into the lower five functions. A function to draw lines is DrawPath only. Therefore, if DrawPath receives the thickness and the color of lines, SetPenColor and SetPenWidth are not necessary. However, to reduce the implementation costs of the optional draw functions, which are described below, we chose a design where functions for setting parameters are separated from the functions for drawing.

We provided the optional draw functions that developers do not always need to implement and register. Table 3 shows the specifications of these functions. The optional draw functions are higher-level functions than the functions in Table 2, but they can be also implemented by using only the commands of most

Table 2. Essential draw functions.

Functions	Specification
void SetPenColor(r, g, b, a)	The color of lines is set
void SetPenWidth(w)	The thickness of lines is set
void SetBrushColor(r, g, b, a)	The color of filling is set
void SetFont(size)	The font size is set
void BeginPath(x, y)	A path is began
void MoveTo(x, y)	A point is added to a path
void DrawPath(isClose)	A polygon or lines are drawn with a path
void FillPath()	A polygon is filled with a path
void DrawText(x, y, text, angle, rx, ry)	A string is drawn on a specified position

Graphics APIs. If the optional draw functions are not registered, the support library executes them by breaking their operations down to the essential draw functions. However, this execution will cause deterioration in execution performance. Therefore, we provided choices that allow developers to select priority either implementation costs or execution performance.

Table 3. Optional draw functions.

Functions	Specification
void DrawLine(x1, y1, x2, y2)	A straight line is drawn
void DrawRect(x, y, w, h)	A rectangle is drawn
void FillRect(x, y, w, h)	A rectangle is filled
void DrawEllipse(x, y, w, h)	A ellipse is drawn
void FillEllipse(x, y, w, h)	A ellipse is filled
void DrawWedge(x, y, iRad, oRad, startAngle, sweepAngle)	A wedge is drawn
void FillWedge(x, y, iRad, oRad, startAngle, sweepAngle)	A wedge is filled

Table 4 shows the functions for interactive visualization. Developers call the functions from target programs. All functions receive the position of a pointer, the operation amount of the mouse wheel, and the information of down buttons. Then, they return True if some operations are done in the functions. Developers can execute interactive visualization in a target program by calling it when input events occur. This design of the functions can be used in touch panel systems.

Table 4. Functions of receiving inputs from viewers.

Function	Timing of calling
boolean OnMouseDown(x, y, wheel, button)	When a mouse down event occurs
boolean OnMouseMove(x, y, wheel, button)	When a pointer moved event occurs
boolean OnMouseUp(x, y, wheel, button)	When a mouse up event occurs
boolean OnMouseWheel(x, y, wheel, button)	When a mouse wheel control event occur

4 Performance Evaluation

Visualization is executed on LuaVM in our environment. Although Lua has relatively higher performance among script languages, the performance is lower than native codes. To confirm whether our environment has enough performance to treat visualization, we conducted evaluations of the execution performance.

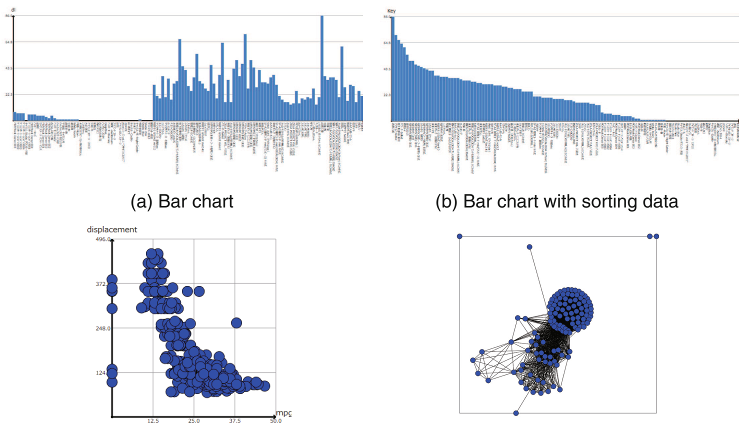
First, we evaluated the performance of about four visualizations shown in Fig. 5. Figure 5(a) is a bar chart, Fig. 5(b) is a bar chart of sorted data, Fig. 5(c) is a scatter plot, and Fig. 5(d) is a node-link diagram using a force-directed algorithm [5]. The size of data is shown in Table 5. We evaluated the performance on C(C++) and JavaScript. On C(C++), LuaJIT[6] was used. LuaJIT is a library that has Just-In-Time Compiler for Lua. On JavaScript, lua.vm.js[7], a library that is made by compiling an original Lua library with Emscripten, was used. JavaScript was run on Electron 1.4.1[8]. A PC with Windows 10 and Intel Core i7-6700K was used. We measured the execution time of Visual Mappings exported from Iv Studio.

Table 5 shows the results of the first evaluation. The time unit used is millisecond, and each result is an average of 100 executions. Note that (d) shows the time of executing a single iteration in the layout algorithm.

If visualization is executed in more than 30 fps (frames per second), we can conclude that the environment has enough performance for interactive visualization. However, this evaluation does not include the time for rendering. The execution of Rendering often takes longer time than Visual Mappings empirically. Considering this finding, we concluded that our environment had enough performance when the measured time is less than 10 ms.

As a result, on C(C++), all visualizations were executed in less than 3 ms. Our environment fast treated (d), which visualized large data. We believe that this is the effect of the optimization of JIT compiler in LuaJIT. On JavaScript, (a), (b), and (c) were executed with enough performance, while (d) took a long time. We need to optimize the exported source codes from Iv Studio.

[6] http://luajit.org/.
[7] https://daurnimator.github.io/lua.vm.js/lua.vm.js.html.
[8] http://electron.atom.io/.



(a) Bar chart

(b) Bar chart with sorting data

(c)Scatter plot

(d) Node-link diagram (force-directed algorithm)

Fig. 5. Visualization used in the first evaluation.

Table 5. Performance evaluation (time unit: millisecond).

Visualization	Size of data	C(LuaJIT)	JavaScript(lua.vm.js)
Fig. 5(a)	123 records	0.87	5.19
Fig. 5(b)	123 records	0.83	5.52
Fig. 5(c)	406 records	1.43	8.53
Fig. 5(d)	123 nodes 4094 edges	2.96	70.13

Second, we conducted an evaluation on the number of records that could be treated by our environment. We measured the execution time of the visualization of two-dimensional data with scatter plots. LuaJIT and lua.vm.js were also used. The size of data records increased by 500 from 500 to 50000, and we measured an average time of 100 executions in each the number of records.

Figure 6 illustrates the results of the second evaluation. In this figure, the X-axis illustrates the number of records, and Y-axis illustrates the average time of 100 executions. Figure 6(a) and (b) shows the same data, but their scales are different for explanation. On lua.vm.js, our environment crashed because of lack of memory, and we could not measure the performance. Therefore, only the results until 22500 records are shown.

For the result of LuaJIT, suppose that in interactive visualization, we focus on the number of records that visualized about 10 ms. The time taken to visualize 9000 records was 10.11 ms. Suppose that in static visualization, our environment can be used for visualizing large data because 50000 records were visualized with 53.50 ms. Recent information visualization treats more than a million records and several tens of dimensions of Raw Data. This visualization often reduces the data size by using filtering, dimension reduction, or clustering. However, the

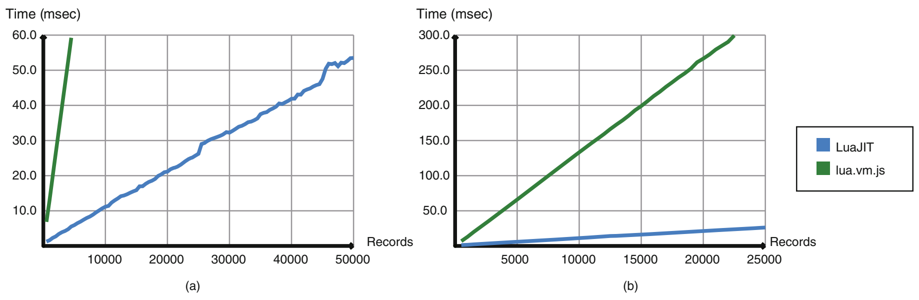


Fig. 6. Relationship between performance and data size.

reduction is performed before Visual Mappings, then our environment receives the result of the reduction. Therefore, we conclude that our environment has practical and enough performance.

For the result of lua.vm.js, suppose that in interactive visualization, the visualization of 1000 records took 12.99 ms. If interactive visualization is used on web pages, it is difficult to visualize more than 1000 records. However, the commonly used scenes of visualization on web pages are presentations. In this visualization, small size data, for example, several tens of records are often visualized. We conclude that our environment has enough performance for commonly used scenes on web pages. Suppose other implementations exist in static visualization. For example, to avoid JavaScript on clients, an SVG of a visualization result is generated on a server. Our environment has high flexibility for implementations; therefore, we conclude that executing large data in JavaScript is not needed always.

5 Use Cases

In this section, two systems are shown using Iv Studio and our environment. Although each system was developed by the first author, it had been developed before the addition of visualization features. Therefore, the use cases shown in this section are examples that have visualization features added to existing systems.

5.1 The Web Site

The first use case is a management web page of digital contents managed by the first author. The first author was motivated to add a feature to observe the overview of the downloads and time trends. Then, using Iv Studio, a feature visualizing the overview of trends was added (Fig. 7).

First, a feature that exported lists of downloads and contents as CSV from SQL database was added into a program on the server side. Next, a visualization of the CSV was developed by Iv Studio. In the visualization, a ChronoView [6]

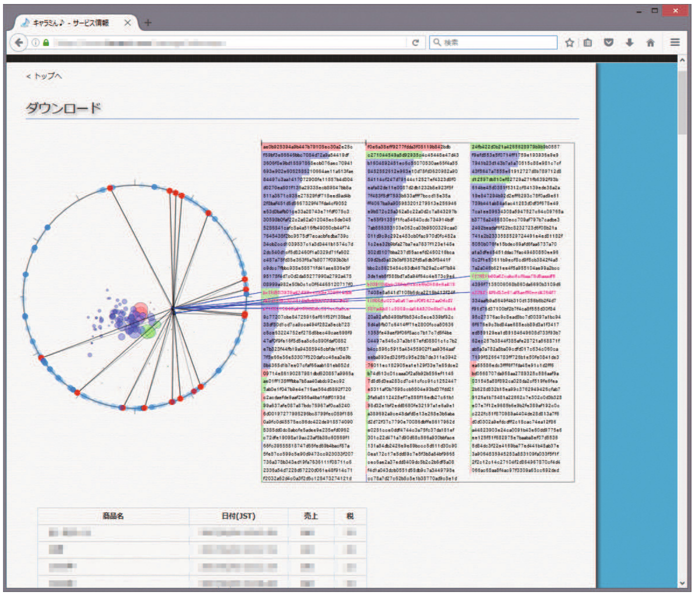


Fig. 7. A web page added a visualization feature. (Color figure online)

was shown in the left side to show the overview of the download time, and a matrix to show contents' names and bars representing the number of downloads was shown on the right side. Then, when marks were selected by viewers, the ChronoView and the matrix were connected by blue lines, and blue circles on the circumference of the ChronoView and marks of the ChronoView were also connected by gray lines. After the development of the visualization, source codes in Lua were exported. Finally, the codes were executed on the web page.

The first author had knowledge on JavaScript and Canvas, but no development experience with them. Therefore, we conclude that implementing this visualization on web pages using only JavaScript took more time than using Iv Studio.

5.2 Test Program of the Developing Device

The second use case is a test program for a pose capture device developed by the first author. The device is a doll equipped with multiple 3D accelerometers and 3D magnetic sensors. The program reads acceleration and magnetism data from the device, computes poses, and renders a posed 3D character model. While in development, errors of computing poses from magnetism were found. To analyze the errors, a visualization feature was added (Fig. 8).

This program was written in C++, and it was built on the original GUI toolkit with the original Graphics APIs. By implementing the draw functions with the GUI toolkit, the visualization feature was added in about 30 min.

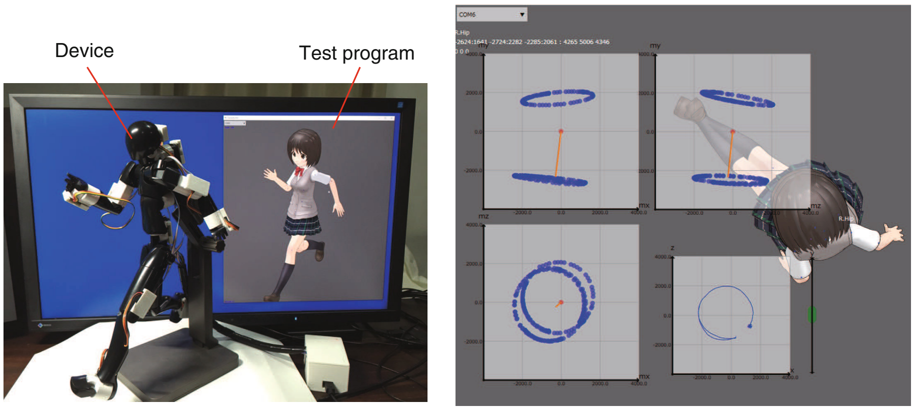


Fig. 8. Device and the test program.

In the visualization, three scatter plots show the values of magnetism on the X-Y, Y-Z, and X-Z planes, and a chart at the bottom right shows values of magnetism seen from the vertical direction. In these scatter plots, the vertical directions obtained from 3D accelerometers are also shown by orange lines. After the design of the visualization, the source codes in Lua were exported. Then, the codes were executed in the test program. To compare a result of the visualization with a pose, the visualization was semi-transparently shown overlapping the 3D character model. The semi-transparent display was realized by drawing the result of the visualization with the same Graphics APIs as the target program.

While developing the visualization, the transparency and color of the charts were tuned frequently as the results of the visualization were confirmed on the test program. Our environment allows the developer to tune the parameters while executing the program, resulting in efficient tuning of the parameters.

6 Related Work

Textual programing languages are widely used to develop programs not only for visualization. There are some toolkits to support the development of visualization programs with textual programing languages, such as Prefuse [7], Protovis [8], and D3 [9]. Alternatively, some researchers provided design patterns [10, 11] to support building visualization programs. The methodology to build visualization system and data structure for visualization was proposed.

There are other information-visualization development tools, e.g. FLINA [12], Snap-Together Visualization [13], Lyra [1], iVis Designer [2], and GeoVISTA Studio [14]. Visualization methods developed with some of these tools can be used from external programs. For example, visualization methods developed with Lyra or iVis Designer can be used on web pages. GeoVISTA Studio can export Java components.

The major difference from the above research is that we focus on execution environments for embedding visualization methods into existing programs. Toolkits can be used only if they support the environments of the target programs. In other words, the use scene of toolkits is limited. The design patterns show us important guides. However, there is a difference that we focused on more practical problem, for example, the choice of execution environments considering programing languages and Graphics APIs. Although some existing visual tools provide execution environments for external programs, they have not achieved the level defined in Sect. 2, such as independence from Graphics APIs and GUI systems.

7 Conclusion

In this study, we designed an execution environment that could be embedded into various program environments easily. Then, the use scenes of visual tools were expanded. We separated the independent parts from visualization systems, adopted the embeddable script language to execution environments of independent parts, and designed interface functions such that implementation costs of embedding were reduced. Then, an exporting feature of source codes in the embeddable script language was added to Iv Studio, which is a visual tool for information visualization. We showed that our environment had enough performance for practical use. Additionally, using use cases, our environment could be used in the development of visualization systems with low costs.

The design of the execution environment is an implementation guide for visual tools for visualization development. Visual tools are one of good solutions to support the development of visualization systems. We conclude that the design reinforces the practicability of the visual tools for development of visualization.

References

1. Satyanarayan, A., Heer, J.: Lyra: an interactive visualization design environment. Comput. Graph. Forum (Proc. EuroVis) **33**(3), 351–360 (2014)
2. Ren, D., Hollerer, T., Yuan, X.: iVisDesigner: expressive interactive design of information visualizations. IEEE Trans. Visual Comput. Graphics **20**(12), 2092–2101 (2014)
3. Ito, T., Misue, K.: A development environment using a visual dataow language for multidimensional data visualization methods. IPSJ J. **57**(7), 1638–1651 (2016). (in Japanese)
4. Card, S.K., Mackinlay, J.D., Shneiderman, B. (eds.): Readings in Information Visualization: Using Vision to Think. Morgan Kaufmann Publishers Inc., San Francisco (1999)
5. Eades, P.: A heuristic for graph drawing. In: Congressus Numerantium, vol. 42, pp. 149–160 (1984)
6. Shiroi, S., Misue, K., Tanaka, J.: ChronoView: visualization technique for many temporal data. In: Proceedings of the 2012 16th International Conference on Information Visualisation, IV 2012, pp. 112–117 (2012)

7. Heer, J., Card, S.K., Landay, J.A.: Prefuse: a toolkit for interactive information visualization. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2005, pp. 421–430 (2005)
8. Bostock, M., Heer, J.: Protovis: a graphical toolkit for visualization. IEEE Trans. Visual Comput. Graphics **15**(6), 1121–1128 (2009)
9. Bostock, M., Ogievetsky, V., Heer, J.: D3 Data-driven documents. IEEE Trans. Visual Comput. Graphics **17**(12), 2301–2309 (2011)
10. Heer, J., Agrawala, M.: Software design patterns for information visualization. IEEE Trans. Visual Comput. Graphics **12**(5), 853–860 (2006)
11. Giereth, M., Ertl, T.: Design patterns for rapid visualization prototyping. In: Proceedings of the 2008 12th International Conference Information Visualisation, IV 2008, pp. 569–574 (2008)
12. Claessen, J.H.T., van Wijk, J.J.: Flexible linked axes for multivariate data visualization. IEEE Trans. Visual Comput. Graphics **17**(12), 2310–2316 (2011)
13. North, C., Shneiderman, B.: Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In: Proceedings of the Working Conference on Advanced Visual Interfaces, AVI 2000, pp. 128–135 (2000)
14. Takatsuka, M., Gahegan, M.: GeoVISTA studio: a codeless visual programming environment for geoscientific data analysis and visualization. Comput. Geosci. **28**, 1131–1144 (2002)