Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 62 (2015) 287 – 296

Procedia Computer Science

The 2015 International Conference on Soft Computing and Software Engineering (SCSE 2015)

A Programming Environment for Visual Block-Based Domain-Specific Languages

Azusa Kurihara[a], Akira Sasaki[b], Ken Wakita[c], Hiroshi Hosobe[b,*]

[a]*Graduate School of Computer and Information Sciences, Hosei University, 3-7-2 Kajinocho, Koganei-shi, Tokyo 184-8584, Japan*
[b]*Faculty of Computer and Information Sciences, Hosei University, 3-7-2 Kajinocho, Koganei-shi, Tokyo 184-8584, Japan*
[c]*Graduate School of Information Science and Engineering, Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro-ku, Tokyo 152-8552, Japan*

Abstract

Visual block-based programming is useful for various users such as novice programmers because it provides easy operations and improves the readability of programs. Also, in programming education, it is known to be effective to initially present basic language features and then gradually make more advanced features available. However, the cost of implementing such visual block-based languages remains a challenge. In this paper, we present a programming environment for providing visual block-based domain-specific languages (visual DSLs) that are translatable into various programming languages. In our environment, programs are built by combining visual blocks expressed in a natural language. Blocks represent program elements such as operations and variables. Tips represent snippets, and macro blocks represent procedures. Using Tips and macros make code more abstract, and reduce the number of blocks in code. Visual DSLs can be a front-end for various languages. It can be easily restricted and extended by adding and deleting blocks. We applied our programming environment to Processing, an educational programming language for media art. We show that the environment is useful for novice programmers who learn basic concepts of programming and the features of Processing.

Peer-review under responsibility of organizing committee of The 2015 International Conference on Soft Computing and Software Engineering (SCSE 2015)

Keywords: Visual Programming; Domain-Specific Language; Programming Environment

1. Introduction

A programming environment enables people to use a programming language conveniently. When novice programmers learn a programming language, they often use a programming environment to understand its syntax and functionalities. In programming education for novice programmers such as children and students, easy-to-access help and references, a simple programming environment, and carefully-designed exercises are required. As a tool for programming education, the programming language should offer limited functionalities first and then expose more advanced features according to the learners' learning processes.

* Corresponding author. Tel.: +81-42-387-4554 ; fax: +81-42-387-4560.
E-mail address: hosobe@acm.org

Peer-review under responsibility of organizing committee of The 2015 International Conference on Soft Computing and Software Engineering (SCSE 2015)
doi:10.1016/j.procs.2015.08.452

In general, a programming language follows three conventions: syntax, semantics, and pragmatics. Lloyd Allison[1] explains these three conventions as follows:

> The complete definition of a programming language is divided into *syntax*, *semantics* and *pragmatics*. Syntax defines the structure of legal sentences in the language. Semantics gives the meaning of these sentences. Pragmatics covers the use of an implementation of a language ....

Most novice programmers cannot understand one or more of the conventions. Therefore, a novice programmer often encounters the following problems:

1. If he does not understand syntax, he will suffer from minor errors like extra/missing braces and punctuations;
2. If he does not understand semantics, he will not understand the behavior of operations and functions;
3. If he does not understand pragmatics, he will not understand when and how to use operations and functions.

In this paper, we propose visual domain-specific languages (visual DSLs), their environments, and a methodology for implementing visual DSLs. A visual DSL is a programming language based on visual blocks[2] which are translatable into various programming languages called host languages. The programming environment includes blocks, an editor of blocks, and a translator that outputs textual code. Blocks are represented in natural languages and translated into syntax-error-free code in the host language. The environment that includes the host language's runtime system supports live coding[3], which continues to give results while programming without compilation by users. The environment is implemented as a web application built on top of HTML, JavaScript, and CSS. Users can write programs without typing commands; instead, they manipulate visual blocks by using drag-and-drop and combining them like LEGO blocks. It enables inexperienced people to write programs easily. We tackle the above-mentioned problems in three ways:

1. Visual blocks are translated into code without syntax errors;
2. Blocks have descriptions in a natural language, and therefore users easily understand their behaviors;
3. Visual DSLs have "tips" that are blocks translated into snippets.

By using visual DSLs, novice programmers can learn programming concepts and the mechanism of the host language. The environment facilitates to transfer visual DSLs to text-based programming languages.

We applied our programming environment to Processing[4], a programming language for media art and design. It is popular for students since it requires few lines of code to get visual outputs. The environment of a visual DSL contains a runtime library of Processing, and thus it supports live coding, where blocks are immediately translated to update the canvas whenever a user edits code.

However, variables and some data structures are difficult for novice programmers to understand. Therefore, we start with a basic visual DSL with minimum functionality and then introduce more advanced features. We also create "tips", that is a set of blocks translated into snippets. By using tips, the programmers can reduce coding steps and results in using fewer blocks and writing more abstract code. This makes the visual DSL more appropriate to the novice programmers' use of the host language.

We provide the visual DSL with teaching materials for Processing. We show example exercises and models for novice programmers and Processing beginners. The key goals of the exercises are to enable them to understand basic programming concepts (i.e., sequence, selection and repetition), to learn features of host languages and to support the way of their thinking. Learning those concepts with our DSL helps the learners smoothly transfer to using a text-based language. In the exercises on Processing, learners start with basic features such as drawing a line, and gradually progress to advanced features such as procedures.

The rest of this paper is organized as follows. Section 2 presents related work in visual programming and programming education. Section 3 describes the design of visual blocks and the programming environment. Section 4 describes the implementation of the environment. Section 5 introduces a visual DSL for Processing and its teaching materials. In Section 6 we discuss other applications of visual DSLs. We conclude the paper in Section 7.

2. Related work

Logo[5] is one of the oldest educational programming languages for children. It supports turtle graphics, and thus users immediately get graphical outputs. However, because of its textual code, the following problems occur: some learners are not used to typing many lines of codes; some commands and arguments are difficult to read and write; variables and their pointers are difficult to understand.

Therefore, block-based programming languages such as Scratch[2], Blockly[6], Waterbear[7], and Tiled Grace[8] have been developed. They have a simple user interface, manipulated by drag-and-drop, and expressed in an easy-to-understand natural languages. They have been used successfully by novice programmers.

Scratch is a block-based language and a programming environment that supports turtle graphics. Its original targets were primary school students, and now it is used by a wide range of users regardless of age including adults. Users learn programming concepts by drawing shapes and creating games. Also, they can use advanced features such as lists and variables. They can create a new block as a procedure by choosing a shape, writing a label, and defining a behavior with other blocks, although new features cannot be added. Scratch is implemented in Flash, and thus users cannot get textual programs and outputs.

Blockly supports turtle graphics and implements translators which translate blocks into host languages (JavaScript, Python, Dart, and XML). It contains various puzzles, and thus users can learn its features step by step. Also, users can create a new shape of blocks with type checking and GUIs such as check boxes. However, users cannot define a behavior or a conversion rule with blocks.

Haskell Platform is an educational programming language environment. It follows the principle of "Batteries included", which means it becomes immediately available after download or installation. It is widely used by Haskell beginners and experienced users. Try Haskell[9] is an environment with fewer tools than Haskell Platform and provides a web tutorial for beginners. By following instructions, users can learn basic commands and syntax of Haskell.

3. Design

We describe the design of visual DSLs and our programming environment.

3.1. Internal and external DSLs

The term "DSL" is an antonym of a general-purpose language (GPL). GPLs (e.g., C and Java) are used for various domains and provide a large number of functionalities. In contrast, DSLs provide functionalities only for specific fields and may not be Turing-complete. Therefore, code in a DSL is typically shorter and has higher productivity, readability and reusability than that in a GPL.

DSLs can be classified into internal and external DSLs. An internal DSL is represented within the syntax of a host language and provided as an API or a library. For example, jQuery is an internal DSL of JavaScript. An external DSL may use custom syntax or follow the syntax of another representation, and usually its implementation is translator that generates code in the host language. For example, Blockly is an external DSL for JavaScript, Python, Dart, and XML. An external DSL makes code shorter and more abstract than that in a GPL.

The visual DSLs we propose are external DSLs that employ translators that generate textual code represented in host languages. Our approach separates the editor from the runtime system. This gives flexibility in extending and implementing a language. We can introduce new blocks with minimum input and choose various host languages including internal DSLs such as Processing.js.

In Section 5, we show an environment of a visual DSL for Processing which contains an external DSL that generates Processing code and an internal DSL that is Processing.js, a library of JavaScript that executes Processing code.

3.2. Screen layout

We describe the screen layout of a DSL by using the one for Processing as an example (Fig. 1). The interface of the environment consists of four types of screen elements: a palette, a workspace, a canvas, and a textual program. A palette displays blocks that are colored according to their categories called block sets (e.g., expressions, variables,

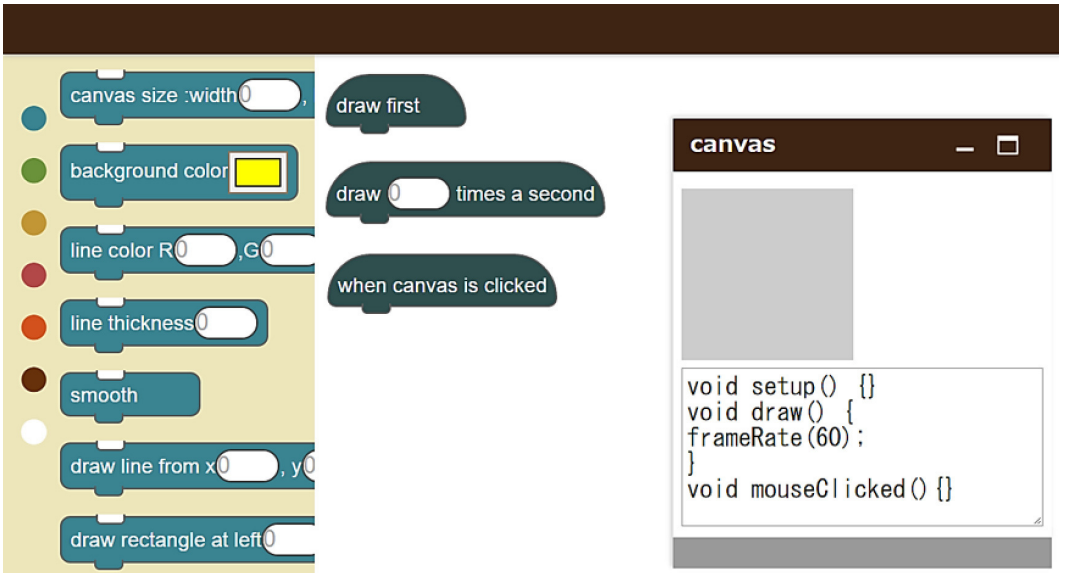

Fig. 1. A palette, a workspace, and a result popup that includes a canvas and a textual program (from right to left).

events, etc.). The palettes can be used as references because users can see available blocks, which represents how they work and what arguments they need. The user drags blocks from the palettes and drop them into the workspace. By combining and separating blocks, the user edits programs and syntax trees as shown in Fig. 3. Translation from blocks into textual programs is executed automatically whenever programs are changed. Showing textual programs helps users to understand grammars and functionalities of host languages.

The environment of the visual DSL for Processing (see Section 5) supports live coding. The translator and the runtime system runs automatically when blocks are modified, and thus users can easily find out bugs and fix them by observing the canvas while programming.

3.3. Programming with blocks

In this research we have implemented a tool to provide programing environments for block-based DSLs. Users add and delete commands by combining and separating blocks. Therefore, novice programmers, who are inexperienced in using computers and programming, can edit programs effectively by operating blocks instead of typing keywords such as commands and functions. By using blocks, syntax errors like missing brackets do not occur, because the blocks are combined correctly and are translated into syntax-error-free code.

Blocks represent functions, commands, and variables in natural languages. For example, an iteration, usually written with a “for” statement in Processing, is represented by “Repeat (n) times” (Fig. 2-c). As a result, users can express iterations in their programs without knowing initialization, termination, and increment.

We have implemented four types of blocks: start, command, rule, and output blocks (Fig. 2). The types of blocks are distinguished by the positions of connectors, i.e. whether or not they are on the top, on the bottom, or inside. A start block represents a function definition, and a command block represents an operation. A control block has three connectors and represents a statement such as a for statement. The inner part of the control block represents the body of an iteration where a sequence of command blocks can be connected. An output block is a block without connectors. It represents a variable, a formula, or a constant, and outputs a value at runtime.

The shapes of an output block and an input area show their data types: string, numeric, and boolean types. An output block can be embedded in an input area if their shapes are the same. For example, in Fig. 2, since both the shape of the input to block (c) and the shape of block (e) are round, users can drop block (e) into block (c). However, since block (d), which outputs a boolean value, has a different shape, it cannot be dropped into block (c). To set a value to an argument, users can not only put an output block but also type a value directly or use a GUI (e.g., sliders

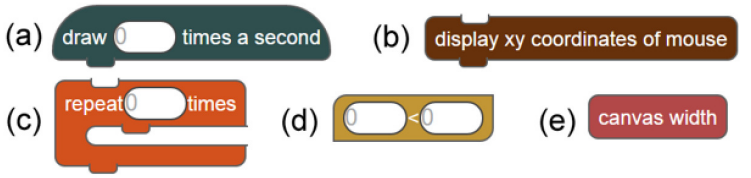

Fig. 2. (a) A start block, (b) a command block, (c) a rule block, (d) a boolean output block, and (e) a numerical output block.

and color pickers). Using a block as an argument narrows the range of the input, which reduces the amount of typing and avoids a type error and an out-of-bound exception.

3.4. Programming with templates

The DSL developers can define templates of programs and provide them as a language construct. The templates restrict the style of a program structure and decrease the flexibility of how to write a program. However, they clarify the structure of programs, how programs work, where users fill code, and in what order they put codes. For example, all variable definitions are placed in the beginning of code to prevent variables from being used before their definitions.

Let us show an example of a code template for Processing as follows:

```
// variable definitions
void setup(){
  // the body of setup
}
void draw(){
  frameRate(n);
  // the body of draw
}
// macro definitions and event handlers
```

Users need two functions to output animations in Processing: `setup()` and `draw()`. `setup()` is called once when an animation starts, and `draw()` is called 60 times per a second until it is stopped by a user. Some beginners of Processing cannot understand how these functions work and how they should be used. The visual DSL for Processing supports users to understand these functions with templates, placing blocks in the workspace: “draw first” which represents `setup()`, “draw (n) times a second” which represents `draw()`, and other event handlers. Blocks below the “draw first” block are added to the body of `setup()`, and blocks below the “draw (n) times a second” block are added to the body of `draw()`.

3.5. Teaching materials

We expect two things from learners: to learn the three programming conventions and to learn features of the host language. Exercises are presented as tutorials of blocks, and the learners learn how they work and how they should be used. Exercises are given according to their programming skills. Learners who are inexperienced in programming are given puzzles with a few blocks with basic functionalities. They can learn new things stage by stage: new blocks, new concepts, and new algorithms. We show examples of exercises for Processing in Section 5.

4. Implementation

We describe the implementation of the programming environment for visual DSLs.

4.1. Overview of the implementation

The programming environment for visual DSLs is implemented as a web application and runs in browsers. It is built on top of HTML, JavaScript, and CSS. Since download and installation are not required, it is easy for novice programmers to introduce the environment. The environment includes a block editor and a translator of blocks into a program in a host language. A runtime library of the host language implemented in JavaScript can be included in the environment. It allows users to edit and run programs automatically in a window. Since JavaScript is cross-platform, libraries for various fields and languages have been developed in JavaScript. For our visual DSL for Processing, the environment employs Processing.js as the interpreter for the host language.

4.2. Creating blocks

A block is defined with a color, a shape, and a label showing its behavior in a natural language, and a translation rule. For example, to define the "repeat (n) times" block (Fig. 2-c), a visual DSL developer inputs the following definition:

```
type  : 'rule',
label : 'repeat (num) times',
code  : 'for(int i=0;i<($);i++){ ($tail) }'
```

`type` defines the shape of the block. We can select it from a command, a rule, and so on. In `label`, "(num)" is replaced with a round input. In `code`, "($)" is replaced with the value of the input in the label, and "($tail)", the body of the for statement, is replaced with the code translated from the blocks connected to the tail of the block.

Blocks are created not only by DSL developers but also by users. Users can define a macro block that is expanded to one or more blocks and is translated into a procedure. In the workspace, users define its color, shape, and label, its procedure name that is shown in a popup window, and the body of the procedure (Fig. 3-d). However, new features cannot be added by macro blocks, because macro blocks are made of existing blocks.

We implemented a popup window to create a block with textual code. By choosing a type and defining a label and code, users can add a block with new features. Since it requires the understanding of the syntax and libraries of the host language, this technique is not for novice programmers.

4.3. Translating blocks

The translator of the visual DSL environment is a template engine that combines code translated from blocks with a code template. Output blocks are translated into variables, constants, and arithmetic/mathematic functions, and other blocks are translated into functions, operations, statements, and so on. A block has a template of code, and its arguments and body may be unfilled.

Tips are blocks that are translated into snippets such as a complex algorithm and debug code. Some of them are translated into one or more operations with specific arguments, and their behaviors are more specific. By using tips, users do not need to re-implement algorithms. Tips also reduce the number of blocks in a program and improve the readability of a program.

Let us show three examples of tips for Processing:

- The "repeat (n) times" block (Fig. 2-c) allows novice programmers to use a for statement without explicitly knowing it.

```
for(int i=0;i<n;i++){
  // body of for statement
}
```

- The "display xy coordinates of mouse" block (Fig. 2-b) can be used as debug code.

```
text("("+mouseX+","+mouseY+")",0,10);
```

- The "draw (n) times a second" block (Fig. 2-a) represents `draw()`, which is not necessarily clearly understandable. By including function `frameRate(n)`, the block makes its meaning more explicit.

```
void draw(){
  frameRate(n);
  // body of draw
}
```

Macro blocks are translated into procedure calls, and their behaviors are defined with the macro definition blocks. For example, “draw branch” (Fig.3-c) is defined with “new block” (Fig. 3-d), which is translated into the following code:

```
void branch(float x, float y, float s, float a){
  strokeWeight(s*0.04);
  ...
}
```

5. Examples

The visual DSL for Processing is an important application for our methodology. Processing, a programming language for design and media art, is used in programming education in high schools and universities since it requires only a few lines of code to get visual outputs. It is based on Java and has its own programming environment called Processing Development Environment (PDE). PDE includes basic features such as a text editor, a runtime system, and a debugger. However, it requires programmers to write textual programs.

The targets of our visual DSL for Processing are novice programmers. The programing environment of the visual DSL provides an editor of the visual DSL and the runtime system together with a set of exercises for learning Processing and basic concepts of programming.

Blocks are classified into block sets corresponding to their features (e.g., expressions, variables, and events). Each block set can be either shown or hidden, which enables a user to add features or to remove them from the base DSL. Furthermore, the use of conflicting blocks in the same code can be restricted. For example, canvas modes (e.g., RGB/HSV and 2D/3D) cannot be switched while a program is running. The environment has block sets representing both of the modes but offers only one of them to the user, which prevents errors regarding the misuse of the modes.

The visual DSL environment includes Processing.js, a runtime library of Processing, and supports live coding. While a user is editing the program, results are updated immediately. This enables the user to modify and debug a program easily.

The objective of exercises for learners is to facilitate learning concepts in programming and features of Processing. We provide, for example, the following exercises:

Sorting in the correct order: Users are given three blocks, “draw square”, “draw circle”, and “draw P” as well as the image of correct shapes. By re-combining blocks in the right order to get the right output, the users learn the sequential execution of commands, which is the first programming concept.

Drawing the path traced by the pointer: “draw first” and “draw 24 times a second” blocks are given to get an animation, and “mouse x” and “mouse y” blocks to sense the position of the mouse cursor. Users learn how to make animations by drawing the locus of the cursor.

We show Processing code below and its representation with blocks in Fig. 3. In this code, a tree is drawn when a mouse is clicked by using `mousePressed()`, and fades out slowly. The `branch()` procedure recursively draws a longer branch and two shorter branches.

```
void setup(){
  size(400,400);
  background(0);
  smooth();
}
```

```
void draw(){
  frameRate(24);
  noStroke(); fill(0,4); rect(0,0,width,height);
}

void mousePressed(){
  stroke(150,220,random(0,130));
  branch(mouseX,height,height-mouseY,270);
}

void branch(float x,float y,float s,float a){
  strokeWeight(s*0.04);
  a+=(random(-7,7));
  line(x,y,x+cos(radians(a))*s,y+sin(radians(a))*s);
  if(3<s){
    branch(x+cos(radians(a))*s, y+sin(radians(a))*s, s*0.6, a+random(12,17));
    branch(x+cos(radians(a))*s, y+sin(radians(a))*s, s*0.6, a-random(12,17));
  }
}
```

6. Discussion

By using our programming environment for visual DSLs, users can read and write code easily by using blocks represented in a natural language, and they do not suffer from syntax errors. Each block shows how it works and what arguments it needs, and thus blocks can also be used as a reference.

Adding blocks expands a visual DSL, and deleting blocks restricts it. The developer of a visual DSL can add tips that are translated to snippets. Snippets work as a template of programs and hide features that are not essential for novice programmers. They reduce the number of blocks in a program and improve its readability. Similarly, blocks that must be used exclusively (e.g., 2D and 3D shapes) should be hidden. Also, advanced features such as arrays and classes should be presented according to the users' programming skills. To selectively provide such blocks and not to provide unnecessary blocks ease programming.

Our visual DSLs reduce costs not just for learning a language but also for implementing a new programming environment. In general, DSLs have less users and technical documentations than GPLs. The developers of DSLs need to implement their programming environments one by one. Also, most of the features of such environments are not for novice programmers but for experienced users. Our visual DSLs and their editor can work as a front-end of a language that has low readability or does not have its programming environment. Our environment for visual DSLs hides the syntax of the host language, and block palettes can be used as a reference. Users can write and execute code immediately without learning the syntax.

Our visual DSL supports live coding by including runtime libraries in JavaScript. We can employ other host languages by defining conversion rules and program templates. Our environment for visual DSLs can also serve as a language workbench, a tool designed to help users create new DSLs. For example, we implemented the visual DSL for Processing, which has syntax similar to that of Java. Although we can get a visual DSL implementation whose host language is Java, problems still remain. First, Java and its libraries have too many functionalities, and a visual DSL cannot show them all. To implement a visual DSL with equivalent features of a GPL such as Java, the features of searching blocks and well-classified lists are required. Second, as users become more skillful, their code will become more complex, and the number of blocks used in their code will become larger. However, the number of blocks shown in a workspace is limited. Although our environment provides making macro blocks to reduce blocks in code, it can only partly solve the space problem. Also, a visualization system that displays large code is required.

Our visual DSLs and other block-based programming languages have similar concepts. They also share the representation of blocks and the idea of translating blocks into programs. However, our programming environment is not

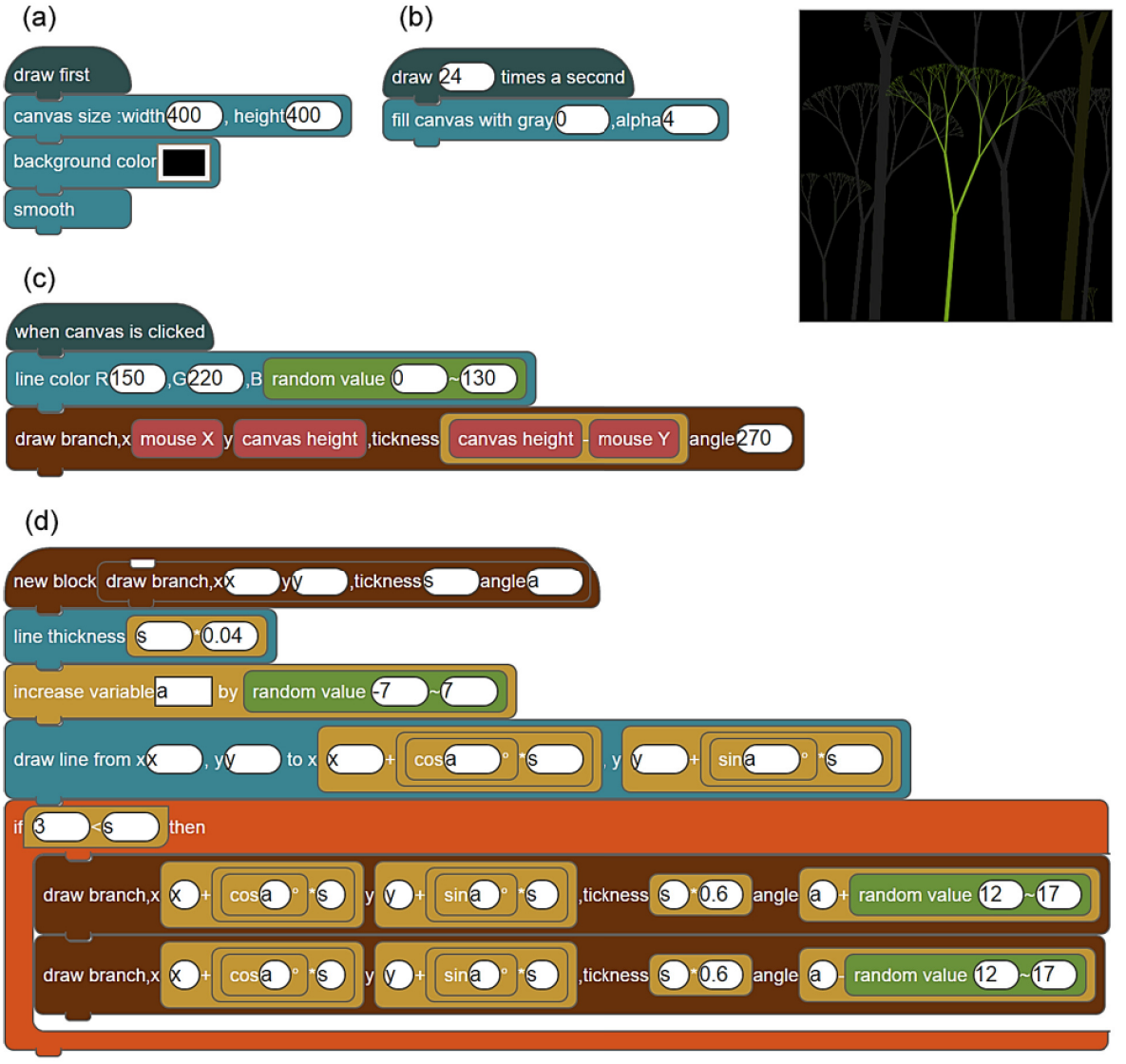

Fig. 3. (a) “draw first” represents setup(), (b) “draw 24 times a second” represents draw(), (c) “when canvas is clicked” represents a mouse event handler, and (d) “new block” represents the definition of the branch() procedure. At the upper right is a visual output of this code.

only for single DSL but also for other programming languages as well, because our visual DSLs can be regarded as a front-end in a language workbench. It has a higher scalability in applying to new languages and adding new features.

Our future work includes implementing an environment that supports making teaching materials, describing large-scale and complex programs, and implementing blocks compatible with mobile devices. As a program become more complicated and longer, the number of blocks in the program becomes larger. It is necessary to reduce the number of blocks and to maintain the readability of the program because there is a limit for the number of blocks that can be displayed in a workspace and that can be grasped by a user. Therefore, further studies are required to visualize relationships between a macro block and its definition, to display the navigator of an entire workspace, and to implement an interface that allows users to search for blocks. In programming education, the use of mobile devices such as touch panels is increasing. The drag-and-drop of blocks with fingers are more intuitive and easier to operate.

7. Conclusions

In this paper, we proposed a programming environment for visual DSLs based on visual blocks that are translatable into textual programs. Blocks are expressed in natural languages and manipulated by drag-and-drop, and therefore programs provide high readability and operability for novice programmers. Such blocks are translated into syntax-error-free code and express their behaviors in easy-to-understand sentences, and thus reduce users' learning costs of the syntax and features of host languages.

A block is defined by its shape, a label in a natural language, and a conversion rule. Tips are translated into common pattern code. Macro blocks are procedures and defined by users. The use of tips and macro blocks reduces the number of blocks in a program, and makes the program more abstract and easier to understand.

We have implemented a visual DSL for Processing whose target users are novice programmers. The environment of the visual DSL for Processing supports live coding. We provided a set of exercises for learning programming concepts (i.e. sequence, selection, and repetition) and features of Processing.

Our future work includes supporting the making of teaching materials in the environment of visual DSLs, displaying large-scale and complex programs, and supporting mobile devices.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number 25540029.

References

1. Allison, L.. *A practical introduction to denotational semantics*. Cambridge University Press; 1987.
2. Resnick, M., et al.. Scratch: Programming for all. *Communications of the ACM*. 2009;**52**(11):60–67.
3. Victor, B.; 2012; Learnable programming: Designing a programming system for understanding programs. URL: `http://worrydream.com/LearnableProgramming/` .
4. Reas, C., Fry, B.; Processing. URL: `https://www.processing.org/` .
5. Harvey, B.. *Computer Science Logo Style Volume 1: Symbolic Computing*. The MIT Press; 1997.
6. Fraser, N., et al.; Blockly: A visual programming editor. URL: `https://code.google.com/p/blockly/` .
7. Elza, D.; Waterbear. URL: `http://waterbearlang.com/` .
8. Homer, M., Noble, J.. Combining tiled and textual views of code. In: *Proc. IEEE VISSOFT*. 2014, p. 1–10.
9. Done, C.; Try Haskell. URL: `http://tryhaskell.org/` .