Processingアプリケーション開発のための視覚的ドメイン特化言語の実装

栗原 あずさ　佐々木 晃　脇田 建　細部 博史

本研究では，アートやデザインのためのプログラミング言語である Processing をビジュアルブロックによって記述するドメイン特化言語 (以下 DSL) の開発について述べる．Processing は視覚的な表現を平易なコードで実現できるため，主に芸術・教育分野においてデザイナや学生によって利用される．しかし，学生の多くは非プログラマであるためプログラミングの理解にはコストがかかる．そこで，ブロックベースの DSL を用いて Processing のコードを記述する．ブロックは自然言語で表現され，色や形状で視覚的に区別されるため可読性が高い．ブロックを組み合わせることで視覚的・直感的なコーディングが可能になり，プログラミングの習得にかけるコストを削減できる．開発環境は WEB アプリケーションとして提供されるため，導入にかけるコストも少ない．また，コーディングから実行まで一つのウィンドウで行うため，テストや修正が容易である．

1 はじめに

Processing [3] は，Java をベースに開発されたアートやデザインのためのプログラミング言語である．Processing は簡潔な表現で視覚的な結果を得られることから，そのユーザは幅広く，アーティストやデザイナだけでなく大学生や高校生にもプログラミング教育の一環として用いられている．学生の多くは非プログラマであり，初めて触れる言語が Processing であるという学生も少なくない．こういった初学者がプログラミングを学ぶ際，よく挙げられる問題に，(1) プログラムのフローが理解できない，(2) 関数のはたらきを理解できない，(3) デバッグができない，などがある．これらの問題が解決できないままプログラミングを行うことで，ユーザが表現したいものと直接関係のない構文エラーなどのデバッグに大きな時間を割くことになる．

本研究では，Processing のコードをテキストではなく視覚的なブロックで記述するドメイン特化言語（以下 DSL）を作成した．以前の研究[9] では，様々なプログラミング言語に変換可能なブロックを作成した．本稿では，変換する言語を Processing に限定し，実行環境を内包した，開発環境の設計と実装について述べる．

DSL にはブロックのエディタ，ブロックから Processing のコードへのトランスレータおよび Processing の実行環境が含まれており，WEB アプリケーションとして提供される．ブロックベースの言語には Scratch [6] や blockly [4] などがある．ユーザは，関数や命令をキーボードで入力する代わりに，パレットからブロックをドラッグ&ドロップし，LEGO ブロックのように直感的に組み合わせることでプログラミングを行う．ブロックは関数や命令として構文的に完成しているため，構文エラーは原理的に発生しない．これを用いて，前述の三つの問題を，(1) ブロックの形状によって分岐や繰り返しなどの流れが可視化されている，(2) ブロックのはたらきが日本語で記述さ

Implementation of a Visual Domain-Specific Language for Developing Applications in Processing.
Azusa Kurihara, 法政大学大学院情報科学研究科, Graduate School of Computer and Information Sciences, Hosei University.
Akira Sasaki, Hiroshi Hosobe, 法政大学情報科学部, Faculty of Computer and Information Sciences, Hosei University.
Ken Wakita, 東京工業大学大学院情報理工学研究科, Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

れているため，関数や変数の意味が取りやすい，(3) コードに変更があるたびに実行結果を更新することで命令やパラメータの意味を取りやすくする，という方法で解決を図る．

また，本 DSL は Processing のコードの可視化だけでなく，Processing の機能を制限・拡張した．配列やテーブルなどの高度なデータ構造など初学者には不要な高度な機能を排除し，コードのテンプレートを用意することでプログラミングモード (2.3 節) を一つに限定した．Processing の機能を制限することで言語設計を単純化し，理解すべき内容を最小限に抑えた．また，Processing のプログラミングで頻繁に用いられるコードのパターンを,「tips」として一つのブロックにまとめることで，単なる関数のブロック化ではない，初学者の利用に特化した Processing の拡張を実現した．

初学者が視覚的 DSL を用いることで Processing のしくみを理解すると共に，順次実行，分岐，反復などのプログラムの基礎的な動きを理解することで，テキストベースの Processing や他のプログラミング言語への移行を容易にする．

2 Processing

本節では Processing の開発環境，実装，言語仕様について述べる．

2.1 開発環境

Processing は Processing Development Environment (以下 PDE) と呼ばれる IDE を備えている．PDE はテキストエディタやメッセージエリアなどを備えており，実行ボタンを押すことで別ウィンドウに結果が表示される．PDE には PDE X [5]，Tweak [7] などのモードが存在する．これらは既存の PDE を拡張し，入力補完やエラーチェック，実行中のパラメータ変更などを可能にした．

Sublime Text [8] にも Processing のプラグインが存在する．Sublime Text はテキストエディタとしても豊富な機能を備えており，様々な言語のためのプラグインが存在する．Processing のプラグインを導入することで，コードの入力補完を可能にする．

PDE と Sublime Text は入力補完やデバッガなど IDE の基本的な機能を備えているが，これらは関数名やプログラムのフローなどを理解しているプログラマのための機能である．従って初学者がこれらの機能を効果的に利用することはできない．

2.2 実装

Processing は Java の API として実装されている．Processing のコードは PDE のプリプロセッサによって Java や Android，Python などのコードに変換される．また，Processing から派生したプロジェクトに processing.js がある．processing.js は JavaScript のライブラリとして実装され，HTML5 のキャンバス要素に実行結果を表示する．

2.3 言語仕様

Processing には static（基本），active（連続），Java の三つのプログラミングモードがある．

static モード: Processing で最も単純なモード．イベントや関数呼び出しは存在せず，コードは命令のみで構成される．すべての命令が一度だけ実行されるため，静止画を得るにはこのモードが多く利用される．

acive モード: `setup()`，`draw()` の二つの関数を利用できる．`setup()` はアプリケーションの開始時に一度だけ呼び出され，`draw()` はデフォルトでは 1/60 秒に一度自動で呼び出される．マウスおよびキーボードのイベントハンドラが実装されており，主にアニメーションやインタラクティブな表現のためのモードである．

Java モード: 本来の Java にもっとも近い記述が可能なモード．アプレットなど Java とほぼ同様の機能を利用できる．

また，processing.js では日本語の描画が可能であるが，PDE は日本語入力をサポートしておらず，公式日本語リファレンスも存在しない． Processing のコード自体も関数や変数名が英語で記述されているため，非英語話者には可読性が低い．

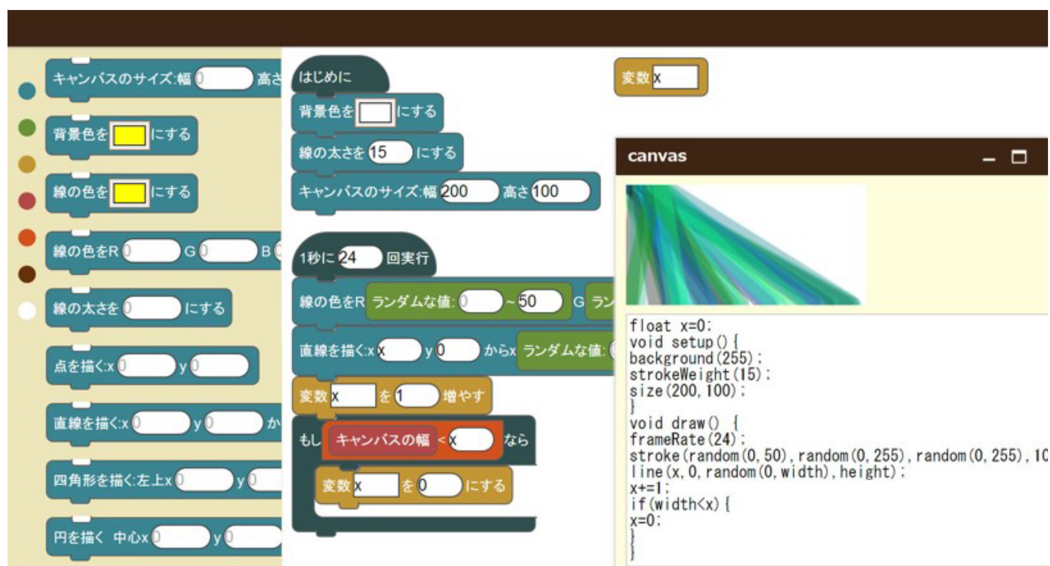

図 1　DSL の外観．左から，パレット，ワークスペース，ポップアップ（キャンバスとプログラム）．

3　視覚的 DSL の設計

本節では，ブロックを用いて Processing のコードを記述する視覚的 DSL の設計について述べる．

3.1　内部 DSL と外部 DSL

DSL の対義語は，C 言語や Java などの汎用言語である．汎用言語は多くの機能を持ち，様々な分野の問題解決に利用される．対して DSL は限られた分野の問題解決に利用される．DSL は目的の分野に関する様々な命令を備えているため，汎用言語と比較してコードの量は少なく，よりプログラミングしやすい．

DSL には内部 DSL と外部 DSL がある．内部 DSL はホスト言語の内部で完結する DSL である．ホスト言語の API やライブラリとして実装され，ホスト言語と同じ言語で記述される．Processing は Java の，processing.js や jQuery は JavaScript の内部 DSL と言える．外部 DSL は，ホスト言語のコードを生成する DSL であり，ホスト言語とは異なる言語で記述される．Ruby に対する Ruby on Rails などが例に挙げられる．外部 DSL を用いることで，ホスト言語を直接書くよりも少ないコードで同じ結果を実現できるため，抽象度や可読性が高まり生産性の向上に繋がる．

本稿の DSL は，ブロックベースの視覚的言語から Processing のコードを生成する外部 DSL であり，JavaScript のライブラリである processing.js を内部 DSL として用いて Processing のコードを実行している．

3.2　画面レイアウト

視覚的 DSL の画面は，パレット，ワークスペース，ポップアップに分割される．図 1 に外観を示す．

パレットには現在利用できるブロックが色ごとに表示されている．ブロックを，変数に関係するもの，フローを制御するもの，イベントハンドラなどに分類し色分けすることで，プログラム全体の可読性を高める．パレットを見ることで用意されている関数を把握できるため，リファレンスとしての意味も持つ．

ユーザは，ブロックをパレットからワークスペースにドラッグ&ドロップする．ワークスペースにおいてブロックを組み合わせたり切り離したりすることでプログラミングを行う (3.3 節)．コンパイルはコードに変更があるたびに自動で行われる．コードの変化がリアルタイムでポップアップ内部のキャンバスに反映されるため，実行結果を確認しながらパラメータを変更できる．同時に，予期しない実行結果になった際，

図 2　ブロックの形状の種類．左上からスタート，コマンド，右上からアウトプット，ルールブロック.

バグの箇所の特定が容易である．

ポップアップには，実行結果のキャンバス，そしてブロックから変換された Processing のコードを表示する．実行結果だけでなくコードも確認可能にすることで，Processing の文法や関数，そしてプログラミングそのものを理解する手助けとなる．

3.3　ブロックを用いたプログラミング

本 DSL では，テキストベースではなく，ブロックベースでのプログラミングを実現した．

ブロックを用いることで，ドラッグ&ドロップで命令を容易に追加および削除可能であり，キーボードの入力に不慣れなユーザであってもマウス操作と最小限のパラメータ入力でプログラミングを実現できる．ブロックで記述することで，初心者が陥りがちなミスを減らし，効率的に Processing のアプリケーションを開発できる．

既存のブロックベースの言語には Scratch や blockly などがある．これらは，ブロックベースの言語およびその開発環境であり，子供のためのプログラミング学習ツールである．特に blockly は，ブロックで記述したプログラムを JavaScript や Python 等，複数の言語に変換できる．対して，本稿で述べる視覚的 DSL は，Processing とその初学者に向けて特化した環境を実現した．

ブロックは様々な命令や変数を意味する．ブロックを日本語で記述することで，英語が苦手なユーザにも読み書きがしやすくなる．これにより，単なる関数名や変数名の日本語化ではなく，ブロックのはたらきや引数の意味を示すことができる．テキストベースで記述する際には，セミコロンを忘れるなどのミスで構文エラーが発生していたが，ブロックは構文の要素が揃った状態で提供されるため，構文エラーが発生しにくい．また，ブロックを引数の個数や意味を含めた自然言語で表現することで，ユーザに関数や構文の使い方を説明する．例えば，for 文をブロックで表現する際には,「n 回繰り返す」というブロックに抽象化できる．これによって，ユーザはカウンタなどの専門的な内容に踏み込むことなく繰り返しを利用できる．

プログラムを構築するには，ワークスペース中でブロックを組み合わせる．これによってコードのパラメータの設定や，命令の追加などを行う．パラメータの設定には，ブロック内のインプットを用いる．図 2 では,「24」,「`true`」がインプットに入力されている．パラメータの設定は，インプットへ直接記述するだけでなく，カラーピッカー等の GUI での指定，またはブロックを埋め込むことで実現できる．これによってキーボードからの入力を更に減らすとともに，入力の選択肢を狭めることで型や値域のエラーを減少させる．ブロックを埋め込む際には，インプットの形状によって埋め込むことができるブロックの形が変わる．これによって，文字列や数値，真偽などの型を区別することができる． 命令の追加は，他のブロックの直後に追加することができるほか，分岐や繰り返し命令のブロックにおいては，ブロックの中にも追加することができる．

ブロックにはスタート，コマンド，アウトプット，ルールブロックなどいくつかの形状がある（図 2）．形状は，ブロックの上下や内部に凹凸があるか否かで分類される．コマンドブロックは上下に他のブロックを繋げることができるが，スタートブロックはイベントハンドラなどの関数名にあたるため，下にのみブロックを接続できる．アウトプットブロックは凹凸を持たず，他のブロックに埋め込むことでコードに追加される．

3.4　プログラムのテンプレート

2.3 節では三つのプログラミングモードについて述べた．

本 DSL では，単純化のためにプログラミングモードを active のみに限定する．active モードにおいても， `setup()` のみに命令を記述することで静止画を

描画できるため，static モードと active モードを統合する．本稿で想定するユーザは Processing 初学者およびプログラミング初心者など，Java 未経験者が主であるため，Java モードは使用しない．

active モードでは，最初に一度だけ実行する `setup()` と，デフォルトでは 1/60 秒ごとに実行する `draw()` の二つに描画の方法が分かれる． Processing 初心者は `setup()` と `draw()` のはたらきを直感的に理解できず，考えていたものと違う結果になるユーザも多い．また，関数の外側には変数定義以外の命令を記述できない．Processing 初学者は，active モードと static モードの混同によるエラーをよく発生させる．そこで，正しいコードを直感的に記述できるよう DSL で支援する．

プログラム全体のテンプレートは以下の通りである．

```
// 変数定義
void setup(){
  // setup の本体
}
void draw(){
  frameRate(n);
  // draw の本体
}
// イベントハンドラなどの関数
```

関数定義を行う `setup()` と `draw()` などのブロックは,「はじめに」「一秒に n 回実行」と日本語でそのはたらきを記述し，初めからワークスペースに配置する．これにより，テンプレートの構造をユーザに示すとともに，Processing のしくみをユーザに理解しやすい形で表示する．

また，関数本体に記述されていないコマンドブロックは無視され，Processing のコードには反映されない．これにより，プログラミングモードの混同を避け，構文エラーを予防する．

4 視覚的 DSL の実装

本節では，視覚的 DSL の実装について述べる．

図 3 tips

4.1 概要

本視覚的 DSL は HTML，JavaScript および CSS で実装され，WEB アプリケーションとして提供される．これはブラウザ上で動作し，インストールやダウンロードが不要であるため，導入が容易である．本実装にはブロックのプログラミング環境，ブロックから Processing のコードに変換するトランスレータ，およびブラウザで Processing のコードを実行する processing.js が含まれている．

4.2 ブロックの作成と変換

ブロックの作成には，その色と形状，はたらきを示す日本語のラベル，そして変換規則を定義する必要がある．この時，アウトプットブロックは変数に，それ以外のブロックは一行以上の命令に変換される．ブロックは変換規則を持つ．例えば，図 2 の「もし true なら」というブロックは

```
if(true){
  // if 文の内容
}
```

というコードに変換される．この時，if 文の条件式 `true` は，ブロックのインプットに書かれたものに置き換えられる． `{ }`の中のコードは，ブロック内部の凸に接続されたブロックの変換結果で置き換えられる．このように，Processing のコードをテンプレートとし，引数や関数の本体などを記述することで，より複雑なプログラムを作成する．

また，あらかじめ関数のパラメータを埋めたコードに変換することで，より具体的なはたらきをするブロックを定義できる．これにより，関数をただ呼び出すだけでなく，プログラムの中でよく用いられるコードのパターンを,「tips」という一つのブロックにまとめることができる．例えば，図 3 の「n 回繰り返す」ブロックは

```
for(int i=0;i<n;i++){
  // for 文の内容
}
```

という for 文に,「マウスの xy 座標を表示する」ブロックは

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

というコードに変換される.

また， `draw()` にあたる「一秒に n 回実行」(図 2) というブロックも tips と言える． `draw()` という関数名だけは，ユーザはそのはたらきを理解できない. そこで， `frameRate(n);` という命令をブロックに含めることで，より明示的な表現のブロックを実現した (3.4 節).

こういったブロックを用意しておくことで，ユーザがこれらを再定義する必要が無くなるため，プログラム全体のブロック数が減り可読性が高まる.

5 サンプルプログラム

図 1 にて記述したブロックについて述べる．4.2 節におけるプログラム全体のテンプレートを元に，変数定義，setup の本体，draw の本体の三か所を埋める．今回のプログラムにはイベントハンドラのブロックを用いない.

変数定義では，float 型変数 x を定義する．単純化のため，ユーザが定義する数値の型は全て float，初期値は 0 にする．setup の本体には，キャンバスの色と大きさ，そして `draw()` で描画する線の太さを設定する．キャンバスの色は GUI のカラーピッカーを用いて指定できる．draw の本体では， `draw()` の呼び出しのタイミングを 1/24 秒に設定し，線を描画する. その際，線を引くたびにその色と位置をランダムに変化させる．乱数は「ランダムな値:0 n」で得られる. 線の色の指定には，キャンバスの色と同じく GUI を用いることもできるが，このプログラムでは引数に乱数を用いるため,「線の色を R(red)G(green)B(blue) 透明度 (alpha) にする」 というブロックを使用した.

6 考察と課題

本視覚的 DSL において，ブロックは日本語で表現されるため，関数のはたらきや引数の意味などが取りやすくなる．ブロックの形は型や命令を区別し，エラーが出る繋ぎ方ができないことから，構文エラーが起こりにくい．また，プログラムに変更を加えるたびに実行結果に反映され，コンパイルが不要である. このため，PDE よりも少ないアクションで実行結果を確認でき，かつプログラミング中にコードのはたらきをリアルタイムで観察できるため，デバッグがより容易になる.

ブロックの定義においては，Processing のコードを単純にブロック化するだけでなく，for 文などを抽象化し，より高度な機能を持つブロックを作成した. これにより，プログラムの記述に使われるブロックの数は少なくなる．Processing のプログラムによく使われる tips をブロックとしてあらかじめ提供することで，ブロックの種類が増え，同時に DSL の機能が豊富になる.

本稿ではブロックを Processing のコードに変換したが，ブロックの変換規則，日本語の表記，プログラムのテンプレートを変えることで，他の言語に対するブロックベースの DSL を作成することができる．例えば，processing.js と同様に 2D や 3D の描画に優れた Data-Driven Documents （D3） [1]，three.js [2] などの JavaScript のライブラリや，JavaScript そのものもブロックで表現することが可能である.

ブロックという GUI はプログラミングに慣れていないユーザのためのものであり，Processing もまた学生など非プログラマがユーザに含まれている. Processing をブロックで記述し，コードと実行結果を見比べることができる環境を作成することで，ブロックの記述で物足りなくなったユーザがテキストベースの記述に移行しやすいようにした.

今後の課題に，プログラミング教育に特化した視覚的 DSL の実装と，より大規模かつ複雑なプログラムの記述が挙げられる.

プログラミング教育においては，段階を踏んで命令を理解し利用する．本 DSL では，図形の描画のほか，分岐，繰り返し，変数の定義といった機能が利用できる．これらの機能に慣れたユーザに配列やマクロ機能を提供することで，より複雑なプログラムを記述可能にする．また，プログラムの文脈によってブ

ロックの利用の可否を切り替えることで，文脈に沿わないブロックを排除する．例えば，キャンバスの 2D モードと 3D モードの切り替えや，RGBA モードと HUE モードの切り替えによって，二種類のブロックの混同を避け，エラーを防ぐ．

プログラムの規模が大きくなり複雑化するにつれ，ブロックの数は増加する．ワークスペースの広さには限界があることから，ブロックの数を減らし，長いコードにおいても可読性の高さを維持する必要がある．以前の研究 [9] では，マクロブロックを定義可能にすることで，ブロックの表現の幅を広げた．マクロブロックは，いくつかのブロックをまとめて一つのブロックとして表現する．Processing のコードにおいては，マクロは新たな関数の定義として実装される．

7 おわりに

本稿では，Processing 初学者を対象とした，Processing のしくみおよびプログラムの基本的な構造を理解するための視覚的 DSL の設計と実装について述べた．

Processing のコードに変換可能なブロックを用いてプログラミングを行い，コードの変更はリアルタイムで実行結果に反映される．このため，デバッグにおいては問題となる箇所の切り出しが容易になる．

ブロックの作成においては，高度な機能を排除し，頻繁に使用されるコードを一つのブロックとして定義した．これにより，単なる Processing のコードの可視化ではなく，初学者の Processing の理解に特化した DSL を作成した．

今後の課題には，プログラミング教育に特化した機能の作成と，Processing 以外の言語での DSL の作成が挙げられる．プログラミング教育に特化した DSL とは，段階を踏んで様々な関数や命令に触れることができ，プログラマのレベルやプログラムの文脈ごとにブロックを利用可能か設定できるものである．また，以前の研究 [9] で述べた視覚的 DSL では，ブロックの日本語表記と変換規則を新たに定義することで，Processing 以外の言語をブロックで表現することができる．これにより，Processing 以外の言語にも特化することが可能な，より汎用的なブロックベースの言語の作成を支援する．

謝辞 本研究は JSPS 科研費 25540029 の助成を受けたものである．

参考文献

[1] Bostock, M.: D3.js, 2011. http://d3js.org/.
[2] Cabello, R.: Three.js, 2010. http://threejs.org/.
[3] Casey Reas, B. F.: Processing.org, 2001. https://www.processing.org/.
[4] Fraser, N. et al.: blockly - A visual programming editor, 2012. https://code.google.com/p/blockly/.
[5] Moharana, M.: PDE X, 2013. https://github.com/processing/processing-experimental.
[6] Resnick, M. et al.: Scratch: Programming for All, *Communications of the ACM*, Vol. 52, No. 11(2009), pp. 60–67.
[7] Sasson, G.: Tweak Mode, 2013. http://galsasson.com/tweakmode/.
[8] Skinner, J.: Sublime Text: The text editor you'll fall in love with, 2008. http://www.sublimetext.com/.
[9] 栗原あずさ, 佐々木晃, 脇田建: ビジュアルブロックを採用したドメイン特化言語とその開発ツールの実現手法, 信学技報, Vol. 113, No. 489(2014), pp. 97–102.