VisionSketch: Integrated Support for Example-Centric Programming of Image Processing Applications

Jun Kato, Takeo Igarashi – The University of Tokyo, Tokyo, Japan – {jun.kato | takeo}@acm.org

ABSTRACT

We propose an integrated development environment (IDE) called "VisionSketch", which supports example-centric programming for easily building image processing pipelines. With VisionSketch, a programmer is first asked to select the input video. Then, he can start building the pipeline with a visual programming language that provides immediate graphical feedback for algorithms applied to the video. He can also use a text-based editor to create or edit the implementation of each algorithm. During the development, the pipeline is always ready for execution with a video player-like interface enabling rapid iterative prototyping. In a preliminary user study, VisionSketch was positively received by five programmers, who had prior experience of writing text-based image processing programs and could successfully build interesting applications.

Keywords: Image processing, computer vision, integrated development environment, example-centric programming.

Index Terms: H.5.2. User Interfaces – GUI; D.2.6. Programming Environments – Graphical Environments, Integrated Environments.

1 INTRODUCTION

Many surveillance cameras and other kinds of monitoring cameras with fixed viewpoints are located almost ubiquitously around cities and within buildings, recording what is happening there. Time-lapse photography is also getting popular. Time-series photos taken from a fixed viewpoint highlight processes that look subtle on an ordinary time scale. It is possible to write a program that processes these recordings, detects interesting events, and extracts useful information from the real world with the help of software libraries like OpenCV [1] that provide image processing algorithms. For instance, it is possible to implement a program that monitors growth of a fungus and notifies when it has grown enough to eat. It is also possible to implement a program that monitors the rotation of a disc on a turntable being scratched by a disc jockey and creates its rotation-time graph. These examples are taken from the study reported in Section 5.

The development of such programs in conventional text-based integrated development environments (IDEs) involves two distinctive challenges. As for the first challenge, the software libraries provide various kinds of computer vision algorithms that take an image as an input parameter. Their other parameters often have visual meaning, such as four *Point* objects denoting a rectangular area in the image. These parameters cannot or (at least) are difficult to be specified in a text-based programming language. Output from the algorithms is often also an image. Conventional IDEs provide a text-based code editor and do not reflect such graphical aspect of the program. As for the second challenge, when it is necessary to monitor the behavior of the program, first, a boilerplate code is written such as that for loading an image and opening a window for visualizing the results. The code is then compiled, and the program is executed. These steps are repeated iteratively until the processing result is satisfactory. This repetition takes long time and prevents fluent exploratory programming.

Provided the above-described issues, this paper aims to answer the following research question: "How can an IDE made usable for development of image processing applications?" Our hypothesis was that an IDE that supports example-centric programming and integrates graphical and text-based user interfaces is usable for such purpose. VisionSketch IDE was developed to test the hypothesis and is distributed as an open-source project [2]. It first asks the programmer to select the input video to start the implementation. The selected video serves as a concrete example with which VisionSketch generates graphical feedback of the program to the programmer. Working on a concrete example is a characteristic shared with tools for end users capable of extracting information from an example in a similar manner to ImageJ [3]. The difference between VisionSketch and ImageJ is that the output of the system is an executable pipeline rather than the extracted information. VisionSketch can take another video source as an input to produce new results, and newly defined components can be reused to build another pipeline. Rapid prototyping tools for image processing, such as Light Widgets [4] and Crayons [5], provide user interfaces for tuning specific algorithms for extracting information and pass it to external applications. Instead, VisionSketch makes it possible to build general image processing pipelines. It is similar to DejaVu [6] and Gestalt [7] in that it aims to facilitate the programmer's workflow with the help of graphical representations of example data. Although VisionSketch is strongly inspired by these works, it assigns a more proactive role to graphical representations. While these systems merely use a text-based programming language, VisionSketch uses both visual and text-based programming languages to implement programs.

VisionSketch provides three interlinked interfaces (Figure 1): a *canvas* for monitoring and editing the pipeline whose graphical view is updated in real time during its execution; a *visual editor* for choosing and setting up each image processing component, which allows the programmer to draw shapes on the input image to narrow down the list of applicable components, set up their parameters, and immediately see the output of the component; and a text-based *code editor* for editing the implementation of any image processing component.

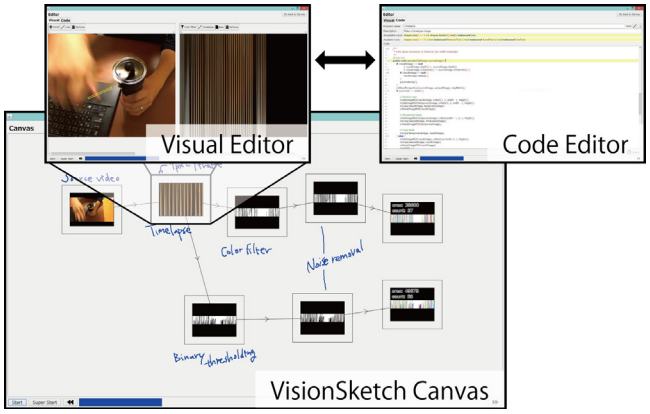

Figure 1. Overview of VisionSketch IDE.

2 Related Work

2.1 Tool Support for Example-centric Programming

Early work on programming by demonstration includes systems for example-centric programming such as Pygmalion [8]. Such systems help a novice programmer to create programs with concrete examples of input data. For instance, when the programmer wants to implement a factorial function, he/she first provides an example input (such as "6") to the function. The system then tries to execute the function until it reaches the end of the code, where the further behaviour of the program is undefined. When a new code snippet is input, the system tries to execute that code again. This iterative process continues until the function returns a concrete value, in this case, 720. VisionSketch also employs the same example-driven development, where an example input to the program is given prior to the implementation of the program. Recent work on example-centric programming includes Subtext [9]. It provides a text-based code editor that allows the programmer to write an incomplete definition of a function and test cases that call the function with example input data. It automatically executes the code and shows stack traces next to the code editor, highlighting the incompleteness of the function definition. The programmer can iteratively update the code and see stack traces generated by executing the program with the example input for developing a program. VisionSketch does not show much textual information such as stack traces, but it does provide graphical representations of the under-development program to aid program understanding.

Several attempts to enhance text-based IDEs with graphical representations of example data have been made. For example, the Barista framework [10] helps to implement structured editors with graphical representations. For instance, it can be used to show a multimedia comment of an image processing operation in which images represent example input and output of the operation. Gestalt IDE [7] is designed for machine-learning applications and includes user interfaces for collecting, editing, learning, and testing examples. Picode IDE [11] is equipped with a text-based editor capable of showing inline photos representing posture data for humans and robots. The photos serve as arguments to APIs for processing posture data. Such concrete examples help the programmer to understand the program. DejaVu IDE [6] is used for developing interactive camera-based applications that add two interlinked interfaces: *timeline* is capable of recording data input to the program as examples and visualizing the history of the program state during its runtime, and *canvas* is quite similar to our own *canvas* interface in that it also provides real-time visualization of the program status. The difference between the two versions of canvas is that DejaVu's *canvas* is mere visualization while our *canvas* is a visual programming language that shows an editable data-flow graph.

2.2 Visual Programming for Image Processing

Many visual programming languages (VPLs) only visualize the structure of the program (i.e., not its contents.) VisionBlocks [12] aims to allow end users to create their own computer vision programs through GUI operations by a structured editor inspired from Scratch [13]. VIVA [14] is a VPL that adopts a box-and-line notation where each image processing component is represented by a symbolic icon and is connected with other components by lines to form a data-flow diagram. MATLAB/Simulink [15] is a commercial VPL that supports various application domains (including image processing). It has a built-in text-based code editor with which a programmer can create a reusable processing component. Some components visualize interesting data, but others are just represented by text labels and symbols. On the other hand, our *canvas* makes use of graphical representations to go beyond symbolic notation.

Some existing VPLs add more meanings to their use of visual components. For example, Agentsheets [16] provides a spreadsheet interface, whose cell shows an interactive agent that reacts to user input or information from other agents. ConMan [17] allows the user to interact with each visual component and set up parameters for rendering computer graphics. Its recorder interface is similar to our video player-like interface in that they both allow the programmer to control the program execution in a frame-by-frame manner. There are two major differences from these VPLs to VisionSketch. First, graphical representations in VisionSketch are used to build programs while those in other systems are for tuning parameters and visualizing results. Second, VisionSketch has an integrated *code editor* to edit text-based implementation of each component. This function ensures that new algorithms can be implemented at any time without leaving the IDE.

These VPLs provide a live programming experience, eliminating the gap between building and executing programs. When the programmer edits the VPL, the program is updated without explicit compilation operations and is always kept ready for execution. VisionSketch also provides a live programming environment, but it is a bit more involved since it integrates a text-based *code editor*. When the text-based code is edited, it is automatically compiled and loaded onto the program, replacing old components if any.

2.3 Tools for Image Processing

Cameras have become pervasive, and many tools to support camera-image processing have been proposed. Their target users range from end-users to novice and professional programmers. Some of these tools do not require prior knowledge of image processing algorithms. For example, Light Widgets is a system [4] that detects areas of skin in the camera images. It transforms any visible surface in everyday spaces into an interactive widget controlled by hand gestures. Vision on Tap [18] adds simple image processing features (such as motion detection) to a webcam video stream and notifies the user of interesting events through a web service. Crayons [5] allows a novice programmer to train a classifier through painting example still images. The trained classifier can later be called from the programmer's own program. *Visual editor* is inspired by the work. Eyepatch [19] is similar to Crayons but operates on video, notifies events through a network protocol, and provides multiple classifiers. While these tools provide access to a limited set of image processing algorithms, VisionSketch provides an IDE with which general image processing applications can be built.

ImageJ [3] is a standalone GUI tool with which end-users can apply image processing operations to images and videos. It requires prior knowledge of such operations, but it is used by various research projects in a broad area of natural science fields. With ImageJ, the user can draw shapes on a source image to narrow down the list of potential operations. This function is equivalent to our component filtering method in *visual editor*. It is capable of creating user-written macros and plug-ins, making the system look more like a development environment. The differences between ImageJ and VisionSketch comes from their different scopes. That is, ImageJ is a tool capable of scripting, while VisionSketch is an IDE that integrates graphical operations. For instance, the user interface for annotating the input image by drawing shapes is used for image processing operation by ImageJ and for adding a new node of a visual programming language by VisionSketch.

OpenCV [1] is a software toolkit that provides a collection of computer vision algorithms. ImageJ can also be used as a Java library. These toolkits provide well-designed APIs to support

writing text-based code. The present work focuses on providing broader support for the entire workflow of the programmer. VisionSketch contains a Java wrapper of OpenCV as its default library. Within VisionSketch, any OpenCV functions can be used to implement a programmer's own image processing components.

3 VisionSketch IDE

VisionSketch is an IDE for developing image processing pipelines. Design of VisionSketch benefits from the characteristics of the supported applications. The applications deal with image processing algorithms, which take an image or video (time-series images) as input. Optional arguments usually have visual meaning, such as four *Point* objects denoting a rectangular area in an image. Outputs from the algorithms are also images, videos, or a group of regions in the image. Conventional IDEs are usually equipped with a text-based code editor and debugger, which cannot present such data intuitively. It was therefore decided to implement the user interface of VisionSketch from scratch in order to better reflect the visual nature of the program.

VisionSketch has three interlinked components: the *canvas* and *visual editor* interfaces are designed to support visual programming; the text-based *code editor* interface is implemented to preserve the full expressivity of text-based programming. These interfaces for visual and text-based programming complement each other to support the programmer's entire workflow. Each interface is described in the following three subsections, followed by a concrete use case to describe how these interfaces help the programmer's workflow.

3.1 VisionSketch Canvas

Canvas is a visual programming environment in which each code element is primarily represented by an image or video rather than text (Figure 2). It is noteworthy that it is *more* visual than typical visual programming languages such as VIVA [14] and VisionBlocks [12], whose program structure is visually presented, but data are referenced by text, including file names and constants. It is the first interface that the programmer sees when opening the VisionSketch IDE. It provides an overview of the program, and although it looks like the *canvas* interface of DejaVu [6], it represents a data flow of the program in the same manner as ConMan [17] and VIVA [14].

Canvas initially has one vacant box. The programmer clicks it to choose the input data (such as a set of time-lapse photos, a video, or live camera input). Then, he/she drags a line from an existing box to another place to add a new box representing an image processing component. When he/she clicks an existing box, *visual editor* appears and allows the corresponding component to be edited. He/she can also draw freeform lines to annotate the program.

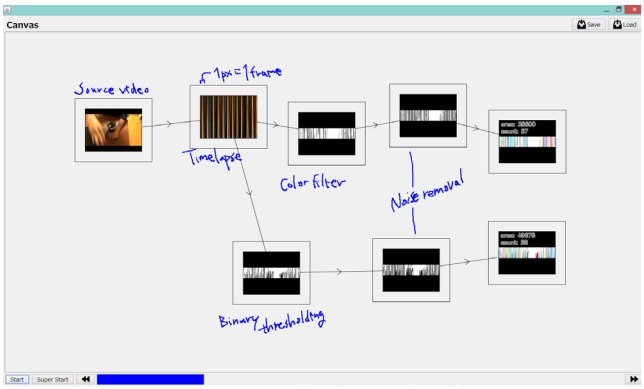

Figure 2. The *canvas* showing two algorithms in parallel.

Compared to a conventional text-based editor where statements and line comments are all represented by text, VisionSketch shows a box to represent one statement and freeform drawings to comments.

Canvas contains a playback interface in its bottom part. It allows flexible control of program execution. With this playback interface, the programmer can test the program with various input data in a more casual way compared to conventional compile-and-run operations, thereby accelerating the development process. While DejaVu also provides a playback interface (named *timeline*), it is used for navigating and replaying recorded sessions of program executions. On the other hand, the playback interface of VisionSketch is used for running the program by providing example input data. Unlike general step-by-step navigation of a text-based debugger, these playback interfaces are specialized for image processing applications and allow frame-by-frame navigation.

When the input data is obtained from a camera in real time, the interface can only "play" or "pause" program execution. Frames that arrive while being paused are discarded. Otherwise, when the input data is from recorded photos or a video, the interface is also capable of jumping to a specific frame of the photos or video, going forward or backward for one frame, slowing down or speeding up the execution, which is usually done at the original frame rate (such as 30 frames per second). The "tape recorder" in ConMan has a similar role driving the computer graphics rendering pipeline, but it can only animate the computer graphics once (or forever in a loop) and does not provide as fine granularity of control as our playback interface.

3.2 Visual Editor

Visual editor is used to choose an image processing component and specify its parameters. It visually shows input and output of the component on its left and right side (Figure 3). It appears when the programmer clicks an image processing component or a vacant box before any component is assigned in *canvas*. With *visual editor*, the programmer first specifies the region of interest (ROI) by drawing shapes on the input image. Next, he/she can choose an image processing component from a list of existing components that are capable of processing the provided ROI. All the other components, which cannot be applied to the ROI, are hidden for convenience. Then, the processing result is immediately shown next to the input image. If the processing result is not satisfactory, the ROI can be edited or another component can be chosen. These operations take immediate effect and provide graphical feedback. He/she can alternatively switch to *code editor* to edit the implementation of the current component or create a new image processing component that takes the ROI of the input image as its parameter.

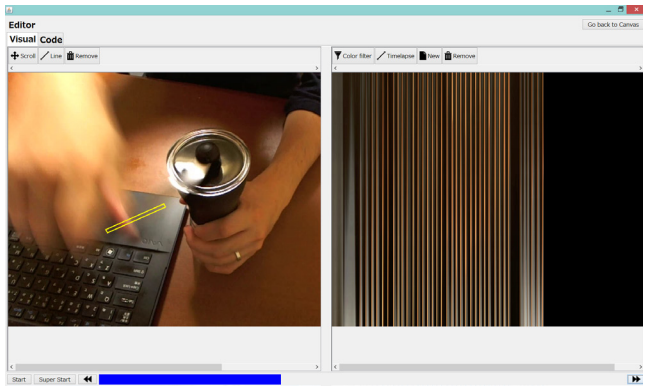

Figure 3. The *visual editor* applying time-lapse operation.

Compared to general programs, image processing pipelines tend to have components with the same or less variety of types of input and output, which often represent images. In such a case, type-based code completion of conventional IDEs do not help much in filtering the components. Instead, VisionSketch uses parameter information for the filtering, which consists of the ROI and the type of the input image. The ROI is a collection of shapes drawn on the input image. Currently, a shape is one of a circle, a line, or a rectangle. The programmer uses a shape tool (one of the "circle", "line", or "rectangle" tools) to draw a new shape or uses the "remove" tool to remove existing shapes. Every time the ROI is updated, the list is updated according to whether each component is applicable to the current parameters or not. For instance, when a circle is drawn on the input image, "linear polar conversion" appears on the list since it can be applied to a circular area. To support the parameter-based code completion, every component is required to implement a static method to check if it is applicable to the given set of parameters. In addition, when an image processing component is selected, how the ROI can be edited is limited. For instance, since "linear polar conversion" can only be applied to a circle, the "line" and "rectangle" tools are hidden. Every component is therefore also required to implement a static method to check if each tool can be used in the current context.

While conventional IDEs force necessitate running the entire program to see the result of a specific processing component, VisionSketch has a built-in interpreter that is responsible for keeping the image processing pipeline up-to-date. When a new image processing component in *visual editor* is selected, the interpreter instantiates the component, and sets up the instance by calling *parameterize(parameter)* method of the component, where *parameter* is a pair composed of the ROI and the input image. It then immediately shows its processing results next to the input image. The results are retrieved by calling *calculate (image)* method of the component. Every time the ROI is updated, the interpreter calls the *parameterize* and *calculate* methods again, as well as the *calculate* method of the subsequent components in the data-flow graph to update dependent components.

3.3 Code Editor

The text-based *code editor* is the last component used in the programmer's workflow, but it is not the least important (Figure 4). It allows the programmer to edit the implementation of any image processing component used in the VisionSketch IDE. In addition to the text-based code editor by which the programmer writes the source code, the proposed editor includes several specialized interfaces used to specify the component information used in *visual editor*. They include text boxes for specifying its function name, description, expressions (one returning acceptable input parameters and the other returning available tools given the context information), and a combo box for selecting an icon. At the bottom of the code editor, an "update" button to save the current definition and replace all the existing components in the image processing pipeline with the updated version is provided.

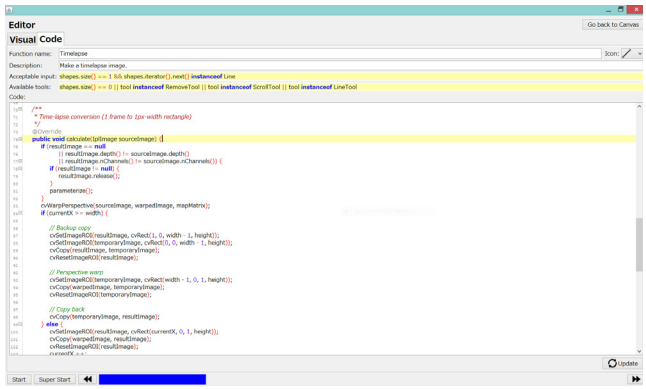

Figure 4. The *code editor* for editing image processing algorithm.

As introduced in Subsection 3.2, *code editor* is shown when the programmer is not satisfied with the current processing result. Therefore, VisionSketch makes an assumption that the programmer is focusing on implementing a function for processing the current specific example rather than implementing general functions. It provides more context-sensitive support for text-based programming. In the current implementation of VisionSketch, when a new image processing component is created, *code editor* shows a template corresponding to the current ROI. For instance, when the ROI is a circle, the default expression for defining acceptable input parameters is set to "*shapes.size() == 1 && shapes.iterator().next() instanceof Circle*" checking whether the ROI is a circle or not.

When the programmer changes the code, he/she clicks the "update" button and goes back to *visual editor*, and the code is automatically compiled and reloaded to the current program. This process is technically called "hot swapping" of Java classes supported by recent text-based IDEs. Compared to the general hot swapping, the process of VisionSketch automatically feeds the reloaded component with the image of the most recent frame in the parent component. In this way, an up-to-date view of the image processing results is always provided.

3.4 Example Use Case

To describe how the three above-described interfaces can help the programmer in harmony, a concrete example use case is introduced in the following scenario (Figure 5). Bob usually grinds coffee beans, drinks a cup of espresso, and starts his work. He does not know the right amount of coffee powder for one cup, but he thinks he can estimate it by counting how many times he rotates the grinder's handle. He wants to write a program that counts the number of grinds, which applies several kinds of image processing to a recorded video of him grinding the coffee beans.

First, Bob records a video of his hand grinding the handle and loads it on VisionSketch IDE, which is shown as the source box. He can change the source to another video or live input from the camera at any time, but in this case, the loaded video will always serve as the input data to the pipeline. Using *canvas*, he drags-and-drops the mouse pointer from the source box to another arbitrary place to create a vacant box.

Next, he clicks the vacant box to open *visual editor* and starts choosing the image processing component. While *canvas* only shows thumbnails of the videos in the boxes, *visual editor* renders the video dot by dot. By playing the video in the editor with the playback interface, he notices that there is a region in which his hand crosses the same region once per rotation. The region is usually shown as a black background, but when his hand crosses it, its color prominently changes to that of his skin. He wants to create a timeline where the change in the region over time is projected spatially. To be more concrete, he wants to copy a line region in the source image every frame and paste it into the resulting image at an x-coordinate incremented every frame.

He starts drawing shapes to find the appropriate operation once he knows what he wants to do. When he draws a line with the "line" tool, such an operation (named "time-lapse") is placed in the list of predefined image processing operations that are applicable to the line region. He clicks the button to instantiate the time-lapse component. Then, he starts playing the video to cumulatively update the resulting image, showing changes over time. Next, he

goes back to *canvas* and creates another vacant box for specifying a subsequent operation. Navigating between *canvas* and *visual editor* does not interrupt the video playback.

In the case of *visual editor* for editing the newly created vacant box, the result of the time-lapse operation is treated as an input image shown on the left side. He wants to perform a contour counter operation on the input image since he thinks that the number of closed regions in the time-lapse image represents the number of grinds. However, he does not see the operation in the list, since the source image for the contour counter operation needs to be a single channel grayscale image or a binary image composed of black or white pixels. He decides to apply a color filter operation to create a grayscale image, where the skin color is highlighted in white. He highlights some time points with the rectangle tool when his hand is not crossing the line. By clicking the "color filter" button, a color filter is created with the current image and the ROI as its parameters. The resulting image is a grayscale image in which all the crossings are painted in white and everything else in black.

It is not always the case that the desired operation is in the list of predefined components. When the contour counter is applied to the result of the color filter, it outputs a much greater number of contours than expected. It seems that the result of the color filter requires some noise reduction. No such predefined operation exists, so he/she clicks the "new" button, which is the last button in the list of components, inputs the name of the operation as "noise removal," and opens *code editor* to start the implementation of a new image processing component. The code template is generated and provided to reduce the time for writing the boilerplate code. It just copies the source image to the resulting image by default, so it needs to be changed to reduce the noise. Various ways to do that are available, but simple erosion and dilation operations are thought to be sufficient. He/she replaces the original line of code that copies the image with a new line that calls up the erosion and dilation operations provided by the OpenCV library.

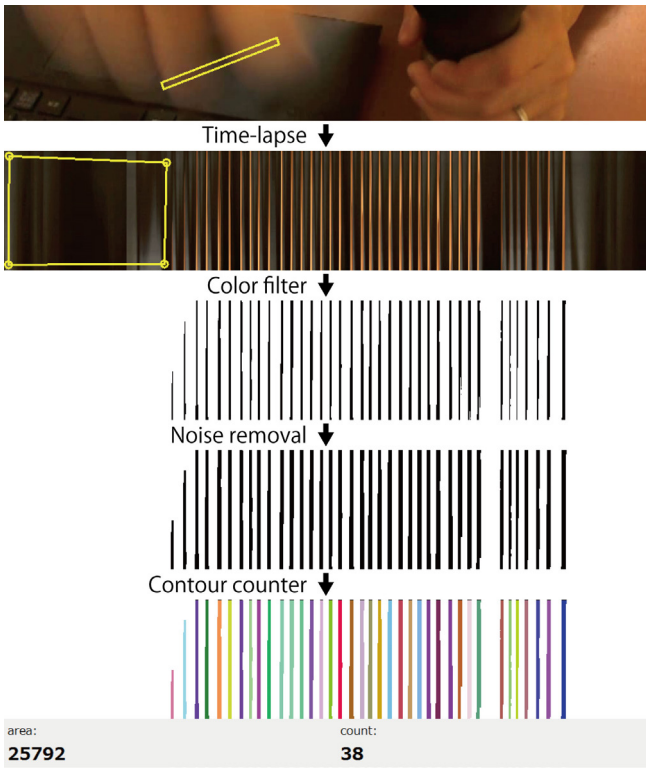

Figure 5. The pipeline created in the example use case.

Once coding is completed, the programmer clicks the "update" button to save and compile the noise removal operation so it can be used in *visual editor*. If a compilation error occurs, it is shown in a message dialog. At that time, it is possible to go back to *visual editor* without any error, and the stored noise removal operation can be applied to the input image. It is noteworthy that the newly implemented operation is loaded as a Java class of an image processing operation. It runs reasonably fast for complex image processing and is reusable, which usually cannot be achieved by interpretive scripting languages.

If the programmer notices that the erosion operation is not enough by seeing the result of the image processing operation in *visual editor*, he goes back to *code editor*, changes some parameters for the erosion, clicks the "update" button and navigates back to *visual editor* to see the updated result, which is now satisfactory. This iterative cycle is enabled by the built-in interpreter and hot-swapping mechanism. Otherwise, it is necessary to compile the entire program and execute it with the source video till the program counter reaches the frame of interest. Such iterative process is cumbersome and difficult without tool support.

Finally, the contour counter operation is applied to see all the crossings highlighted in the resulting image with the total number of crossings shown below. While every image processing component is expected to return an image as a result, it can optionally return other values that are visualized in *visual editor* and can be retrieved by the child components for further processing. While VisionSketch currently supports numerical values and text for this optional visualization, its architecture is extensible enough to support other types of data for visualization.

4 Implementation

VisionSketch is an attempt to tightly integrate visual and text-based programming in one IDE. Since recent open-source IDEs that do the same kind of integration could not be found, it was necessary to build the IDE from scratch with help of existing low-level components such as a Java compiler, a library that implements image processing algorithms, and a text-based code editor with support of syntax highlighting and other convenient features. Its open-source distribution [2] is helpful for understanding the details.

4.1 Overview

VisionSketch runs on a computer that hosts a Java VM and the Java wrapper of the OpenCV [1] library. It currently supports both 32-bit and 64-bit Windows, Mac OS X, and common Linux distributions. It requires a video source to work on (Figure 6). The programmer can use recordings or connect to a camera device to retrieve images in real time. VisionSketch is also capable of periodically receiving images from a smartphone running the Android OS or an Internet-protocol camera.

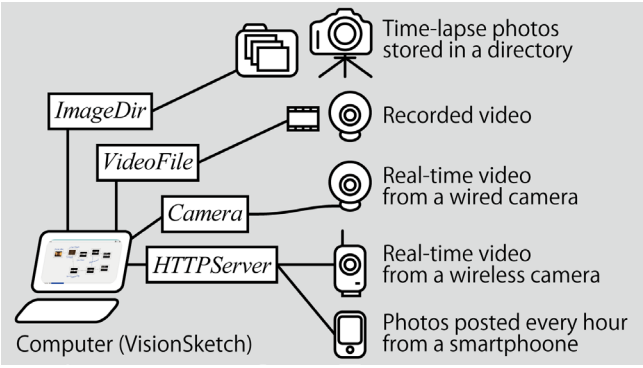

Figure 6. *Input* implementations and supported hardware setup.

In its current implementation, VisionSketch has five predefined image processing components as shown in Figure 7, whose details are available online [2].

4.2 VisionSketch Visual Programming Language

Canvas is a visual programming environment that graphically shows the image processing pipeline and allows it to be edited. It has a built-in interpreter that controls the execution of the pipeline. The pipeline is a directed graph without any loops, i.e., a tree whose nodes are represented by an instance of *Stmt* class (where *Stmt* stands for statement). Each *Stmt* instance can have one or more child *Stmt* instances. The processing result of the instance is passed to the children as their input. Multiple children allow the programmer to compare alternatives and help him/her find the best algorithm. A *Stmt* instance always has one parent *Stmt*, except for a subclass instance (called *Input*), which is the root node in the tree and provides input data to the pipeline.

There are currently four implementations of *Input*: *VideoFile* for loading a video file, *ImageDir* for loading image files in a specified directory, *Camera* for retrieving images from a camera in real time, and *HTTPServer* for receiving images posted from external programs through the HTTP 1.0 protocol. There are currently two client implementations: one for periodically posting photos from a smartphone, and another for bypassing images from an Internet-protocol camera. When the root node is a *VideoFile* or *ImageDir* instance, the execution of the pipeline can be thought of as moving the cursor from the beginning to the end of the input set. In this case, the programmer can freely move the cursor to any arbitrary frame. Such a seeking operation is not supported by the other implementations (including *Camera* and *HTTPServer*).

All *Stmt* instances except *Input* are associated with an image processing component that is an instance of the algorithm-specific class that extends the *Function* abstract class. *Function* provides the *parameterize(parameter)* method, where *parameter* is an instance of the *FunctionParameter* class that holds a pair composed of the ROI and the image. The ROI is a set of shapes, each of which is a *Shape* instance. There are currently three subclasses of *Shape*: *Line*, *Rectangle*, and *Circle*. For instance, when the programmer draws a line on the input image, a *Line* instance is instantiated and added to the ROI. Then, *parameterize* method is called once upon the instantiation of the *Function* class when the component is selected in *visual editor*. It is also called whenever the programmer edits the shapes and updates the ROI. When the parent *Stmt* provides a new input image, *calculate(image)* method is called to calculate the output. For instance, *ColorFilterFunction* provides a color filter based on the histogram back projection. Its *parameterize* method calculates histogram from the ROI of the image and its *calculate* method calculates the back projection of the histogram to the current image. Through calls to these two methods, pixels in the current image with similar colors to the ROI of the parameter image are painted in white.

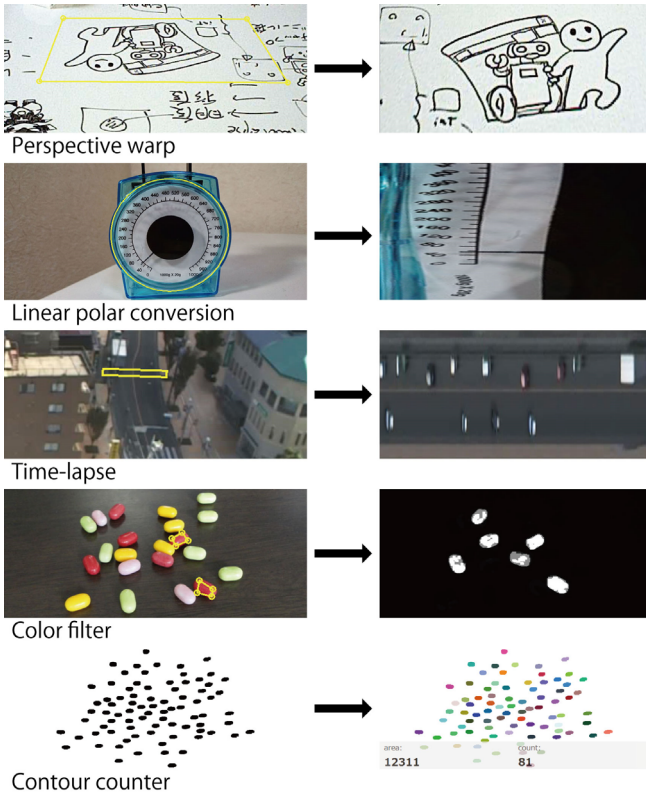

Figure 7. Predefined image processing components.

4.3 Integration of Visual and Text-based Programming

Visual editor is the user interface that bridges the gap between the visual and text-based programming languages. It allows the programmer to instantiate a *Function* instance, set up its parameters, and make it ready for use in the VisionSketch visual programming language. It also allows him to switch to the *code editor* to edit its text-based definition.

Implementation of an image processing component is not only responsible for processing images but also for showing and hiding relevant information in *visual editor*. For instance, when *visual editor* generates the list of *Function* subclasses, it filters the list by checking whether each subclass accepts the current set of parameters or not. Buttons in the list for instantiating *Function* instances have their own icons and text labels. Once the *Function* instance is created, some shape tools may be disabled to prevent ROIs from being invalid for the image processing. To show and hide these information, a *FunctionTemplate* subclass is defined as a singleton for each *Function* subclass. For instance, a *ColorFilter* class extends a *FunctionTemplate* abstract class and implements methods such as *getName()* and *getIconFileName()*, providing meta information about a *ColorFilterFunction* class.

With *code editor*, the programmer can edit the meta information as well as the implementation of a *Function* subclass representing an image processing algorithm. It is capable of Java syntax highlighting, code folding, and other basic features. While the *Function* implementation is directly saved as a Java source code, the meta information is saved as an XML file. When the programmer clicks the "update" button, the meta information is exported as a class definition that extends the *FunctionTemplate* class and is compiled with the *Function* implementation by a Java bytecode compiler.

When *code editor* updates the definition of an existing image processing component, it first needs to unload the old *Function* and *FunctionTemplate* implementations from the virtual machine. First, it replaces existing instances with dummy instances. Then, it disposes the class loader that was used to load the old definitions. Next, it instantiates a new class loader and loads newly compiled *Function* and *FunctionTemplate* implementations. Finally, it replaces the dummy instances with the new *Function* instances. It also invokes their *parameterize(parameter)* and *calculate(image)* methods to automatically update the view of *visual editor* and *canvas*. With these dedicated support functions, the programmer can seamlessly switch between the visual programming and the text-based programming.

5 User Experience

A preliminary user study was conducted to collect user feedback about VisionSketch and investigate its applications and limitations.

5.1 Setting

Five male participants, aged 23-36 years old (mean: 29.6 years old, standard deviation (SD): 4.40 years), were recruited for the study in a university laboratory of computer science. They all had professional programming experience, building applications for commercial and research purposes. They had basic knowledge of the Java programming language which is used in *code editor*. They also had prior experience of building image processing applications. Four of them had used OpenCV [1] for the purpose. Their uses of OpenCV vary from color reduction and beautifying photos to edge detection from a static image. While we did not conduct a comparative study against another IDE, we chose the participants with such experience and asked them to compare the VisionSketch experience with their past experience throughout the study.

The user study consisted of four parts. First, the participants answered a demographic questionnaire asking their age, sex, and prior experience with programming and computer vision libraries. Then, they watched a demonstration of the VisionSketch IDE, as introduced in Subsection 3.4. Next, they were provided with five pre-recorded videos which we thought interesting events could be detected; they were also allowed to bring an interesting video or use a webcam to retrieve a live video input to work on. Among these vide sources, each of them chose favourite one and used the IDE to implement an application. Finally, when they were satisfied with the processing results of their applications, they answered a post-experimental questionnaire.

5.2 Observations and User Feedback

All participants successfully created their own applications in one to two hours. The post-experimental questionnaire contained four common questions about each interface. The results are listed in Table 1, consisting of the mean, standard deviation, and percentage of positive responses (>4 on a 7-point Likert scale) for each question. We also asked to write down concrete comments on each interface. Some of the representative answers are *quoted* below.

The participants appreciated the example-centric workflow of the VisionSketch IDE that *"gives immediate graphical feedback concerning the program being developed." Canvas* and *visual editor* were favored by all participants (Q1), thought to be simple enough (Q2) and easy to use (Q3). It is *"very convenient since I could see the up-to-date overview at a glance."* In addition, *"the playback interface in the canvas allows me to control and monitor the execution interactively. It was very nice."* The shape tools in *visual editor "provide immediate graphical feedback of the ROI tuning."* One participant answered that *visual editor* was not simple (Q2) because *"it takes time to find a graphical way to do something I could do with text-based code."* He was used to low-level APIs of OpenCV, and the graphical operation typically involves several API calls. As a result, he felt overwhelmed. Another participant commented that *"existing IDEs force me to run the entire program to see a small piece of interesting results, but VisionSketch allows me to check it interactively without leaving the current context."*

All of the participants implemented new image processing components with *code editor*. While they admit the necessity of text-based programming to precisely control the algorithm logic, they were observed to prefer to stay with visual programming. One participant commented, *"It would be nice if its usage could be reduced, as the UI part is much better."* Another participant demanded, *"Code editor should come with more graphical feedback, such as a live view of the processing results, as visual editor does."* They sometimes utilized existing components and avoided text-based coding (for a concrete example, see Subsection 5.2.3). Nevertheless, they appreciated the "update" button, which *"immediately makes the newly defined or updated component available in visual editor and canvas."*

Hereafter, three applications developed by three participants in the user study are presented to showcase the real use of the VisionSketch IDE and investigate its capability and limitation (Figure 8). Two applications developed by the other two participants monitor traffic on a road and count the number of visitors in a room, respectively. Their descriptions are omitted because their usage patterns are included in the other applications.

#	Question	Canvas			Visual editor			Code editor		
		Mean	SD	%	Mean	SD	%	Mean	SD	%
1	I would like to use it frequently.	5.80	0.74	5/5	5.80	0.74	5/5	3.20	1.17	3/5
2	I found it unnecessarily complex.	2.00	0.63	0/5	2.80	1.33	1/5	3.80	1.72	3/5
3	I thought it was easy to use.	6.00	0.63	5/5	5.60	1.02	5/5	3.40	1.02	2/5
4	I needed technical support to use it.	3.00	1.09	2/5	3.60	1.50	2/5	5.00	1.26	4/5

Table 1. Results of questionnaire.

5.2.1 Disc-jockey Analyzer

The participant retrieved a video file from an online video-sharing website that records a live session of a professional disc jockey from a ceiling-mounted camera. It is not easy for him to analyze how equipment is manipulated by the disc jockey because it contains various interfaces and the manipulation is often very quick.

To address this issue, he implemented an application with which he can analyze the actions of the disc jockey. He created multiple children of the video input to process multiple interfaces separately. For instance, two branches count the number of discs used on each turntable. Another two branches show the rotation of the discs as vertical motions. When the disc is moving clockwise, the output image scrolls down. Another branch monitors the slider's knob for controlling the left/right balance to create a time-balance graph. To monitor disc rotations and volume changes, he used linear polar conversion, perspective warp, and time-lapse components. To count the number of discs used in the session, he used perspective

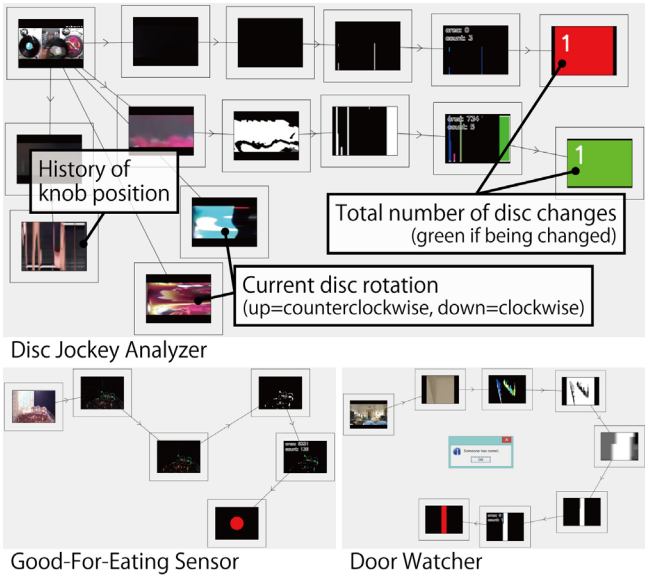

Figure 8. Applications developed by the participants.

warp, color filter, time-lapse, and contour counter components in addition to a new component that takes the contour counter component as its parent and displays the number of discs which is incremented when the number of detected contours gets increased and exceeds a specific threshold.

The participant looked surprised at the capability of the time-lapse operation, with which he could create various meaning graphs. He commented that the application is already very useful for analysis of the actions of the disc jockey; however, for reproducing the actions, he wants audio playback synchronized with the video. While VisionSketch currently focuses on image processing, audio-related feature is interesting future work.

5.2.2 "Good-for-eating" Sensor

The participant chose a set of time-lapse photos monitoring fungus. Photos were taken every hour under a controlled lighting condition. He wanted to create a program that analyzes the newest photo and notifies him when the fungus has grown enough for eating.

To implement such a pipeline, he decided to measure the size of the fungus area. When the size exceeds a specified threshold, the user is notified. First, he seeks an image without visible fungus and sets up the background subtraction component. When he played the video, it made the fungus area look brighter than the other area. Then, he implemented a binarization filter that binarizes each pixel (paints it white if it is brighter than the threshold; otherwise, black). To tweak the threshold, he switched seamlessly between *code* and *visual editor* with help of the "update" button. Next, he found the result a bit noisy. He implemented a median filter and inserted it right before the background subtraction to successfully remove the noise. Finally, he applied the contour counter operation, which not only counts the number of closed regions but also counts their area size. He added another component at the end that shows a coloured circle (if the size is less than the threshold, red; otherwise, green). When he switches the video source to the *HTTPServer* that receives a new image every hour, the colour tells him whether the fungus is good for eating or not.

He appreciated the *visual editor*'s capability to quickly switch and test multiple image processing components, but he commented that *"Additional interactive GUIs for tuning other parameters (such as numerical constants declared in the text-based code) are desirable,"* which was previously explored by Juxtapose [20]. Additionally, he commented that the current VPL is a bit too simple. For instance, he wanted to output a grayscale image and use it as a mask in another image processing component. This function requires the capability of a *Stmt* instance to have two input sources. While keeping the simplicity for usability is important, our future work includes such extension of the VPL for better functionality.

5.2.3 Door Watcher

The participant wanted to be notified when the door of a room is opened, so that he is not surprised by a sudden visitor. He first asked his colleague to go in and out of the room to observe the door in the real-time webcam images. Then, he noticed that the recorded video is better than live input to prevent his colleague being bothered, so he switched to the recorded video including his colleague's action. He knew that he could detect the event by applying a pattern-matching algorithm, but he hesitated to use *code editor* and tried using predefined components to find a solution; that is, he used a combination of background subtraction, color filter, perspective warp, and time-lapse components. At the end of the pipeline, he added a new component that pops up a message dialog notifying the user about the visitor. Since each component has full access to the Java API, an original GUI can be easily added. For instance, a slider interface may be provided for tuning a numerical parameter.

To get a satisfactory result, he tried various combinations of image processing components, which were effectively supported by the immediate graphical response. He commented that *canvas* should show the text label for each component as well as the graphical representation. When the pipeline grows large, mere graphical information gets confusing since it often looks similar.

6 Conclusion

The proposed VisionSketch IDE has three interlinked interfaces to facilitate example-centric workflow of building image processing applications. *Canvas* and *visual editor* interfaces show graphical representations of concrete examples to aid program understanding. They also allow graphical operations such as drawing shapes on the input image to choose and tune image processing components. The text-based *code editor* is still needed to implement new algorithms and needs interactive GUI support.

References

[1] OpenCV. http://opencv.org/

[2] VisionSketch. http://junkato.jp/visionsketch/

[3] M. D. Abràmoff, P. J. Magalhães, and S. J. Ram. Image processing with ImageJ. *Biophotonics International*, 11(7), pp.36–42, 2004.

[4] J. A. Fails and D. Olsen. Light widgets: interacting in every-day spaces. *IUI '02*, pp.63–69, 2002.

[5] J. A. Fails and D. Olsen. A design tool for camera-based interaction. *CHI '03*, pp.449–456, 2003.

[6] J. Kato, S. McDirmid, and X. Cao. DejaVu: integrated support for developing interactive camera-based programs. *UIST '12*, pp.189–196, 2012.

[7] K. Patel, N. Bancroft, S. M. Drucker, J. Fogarty, A. J. Ko, and J. Landay. Gestalt: integrated support for implementation and analysis in machine learning. *UIST '10*, pp.37–46, 2010.

[8] D. C. Smith. Pygmalion: an executable electronic blackboard. In *Watch What I Do: Programming by Demonstration*, pp.19–49, 1993.

[9] J. Edwards. Example centric programming. *ACM SIGPLAN Notices*, 39(12), pp.84–91, 2004.

[10] A. J. Ko and B. A. Myers. Barista: an implementation framework for enabling new tools, interaction techniques and views in code editors. *CHI '06*, pp.387–396, 2006.

[11] J. Kato, D. Sakamoto, and T. Igarashi. Picode: inline photos representing posture data in source code. *CHI '13*, pp.3097– 3100, 2013.

[12] A. Bendale, K. Chiu, K. Marwah, and R. Raskar. VisionBlocks: a social computer vision framework. *IEEE SocialCom '11*, pp.521–526, 2011.

[13] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *ACM TOCE*, 10(4):16, 2010.

[14] S. L. Tanimoto. VIVA: a visual language for image processing. *Journal of Visual Languages and Computing*, 1(2), pp.127–139, 1990.

[15] MATLAB/Simulink. http://mathworks.com/products/simulink/

[16] A. Repenning and W. Citrin. Agentsheets: applying grid-based spatial reasoning to human-computer interaction. *IEEE VL '93*, pp.77–82, 1993.

[17] P. E. Haeberli. Conman: a visual programming language for interactive graphics. *SIGGRAPH '88*, pp.103–111, 1988. ACM.

[18] K. Chiu and R. Raskar. Computer vision on tap. *IEEE CVPR Workshops*, pp. 31–38, 2009.

[19] D. Maynes-Aminzade, T. Winograd, and T. Igarashi. Eyepatch: prototyping camera-based interaction through examples. *UIST '07*, pp.33–42, 2007.

[20] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. *UIST '08*, pp.91–100, 2008.