Integrated Visual Representations for Programming with Real-world Input and Output

Jun Kato
The University of Tokyo, Tokyo, Japan – jun.kato@acm.org

ABSTRACT
As computers become more pervasive, more programs deal with real-world input and output (real-world I/O) such as processing camera images and controlling robots. The real-world I/O usually contains complex data hardly represented by text or symbols, while most of the current integrated development environments (IDEs) are equipped with text-based editors and debuggers.

My thesis investigates how visual representations of the real world can be integrated within the text-based development environment to enhance the programming experience. In particular, we have designed and implemented IDEs for three scenarios, all of which make use of photos and videos representing the real world. Based on these experiences, we discuss "programming *with* example data," a technique where the programmer demonstrates examples to the IDE and writes text-based code with support of the examples.

Author Keywords
Real-world input and output, development environment

ACM Classification Keywords
H.5.2. Information interfaces and presentation (e.g., HCI): User Interfaces – GUI; D.2.6. Software Engineering: Programming Environments – Integrated environments.

INTRODUCTION
An integrated development environment (IDE) is a collection of user interfaces to support the entire workflow of programming including writing code and debugging the program. While the mainstream general-purpose IDEs look quite similar with a text-based editor and debugger, they do not always represent the program in a user-friendly manner. There is existing research on redesigning IDEs to support specific types of application development e.g. d.tools [1] for physical user interface and Gestalt [5] for machine learning. In my thesis, we argue that this issue is critical for the development of programs that deal with real-world input and output (real-world I/O) by showing three example scenarios. Then, they are addressed by proposing new IDEs which integrate visual representations within text-based programming environments.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the author/owner(s).
UIST'13 Adjunct, October 8–11, 2013, St. Andrews, United Kingdom.
ACM 978-1-4503-2406-9/13/10.
http://dx.doi.org/10.1145/2508468.2508476

Picode (Figure 1, [4]) deals with static complex data used in the program – to be more specific, data of human and robot postures. Such data is often used for handling gesture input and controlling robots. Static complex data in general is better understood with its visual representation than its textual reference such as its file name. Sikuli [8] addresses this issue by introducing a code editor with inline images which serve as the API arguments. We developed *Picode* by applying a similar idea to posture data.

While *Picode* provides still images to help the programmer to understand static part of the source code, there remains difficulty to understand its dynamic behavior. When the program deals with real-world I/O such as images from a camera, it is almost impossible to read the source code to imagine what exactly happens in the program. Therefore, the programmer needs to execute the program and monitor continuous visual data, which is not supported in the current mainstream IDEs. *DejaVu* (Figure 3, [3]) addresses this issue by providing two interlinked components that record and visualize program input and output. In this case, the visual representations are videos consisted of time-ordered images accumulated over the program execution.

In these completed projects, the visual representations serve as visual aids that help understanding of the program. Meanwhile, our ongoing project of *Visionsketch* (Figure 4, [2]) not only helps understanding but also allows building image processing programs with help of videos. Its main view shows multiple videos connected with arrows to form a directed graph. Each video represents real-time output from an image processing component. The graph can be edited to achieve high-level programming as other dataflow visual programming environments [6, 7]. While those existing environments seldom discuss the integration of high- and low-level or visual and text-based programming, *Visionsketch* provides the detail view for editing each component. Its parameters and implementation can be edited by graphical direct manipulation and by text input, respectively. In addition, *Visionsketch* provides a playback interface for flexible control of program execution.

In each of these projects, example data captured from the real world are represented intuitively by photos or videos. These led us to define "programming *with* example data," a technique where the programmer demonstrates examples in the real world and writes text-based code in the IDE with support of the example data. It is expected to provide more control on designing logics than programming by example and to be more intuitive than mere text-based programming.

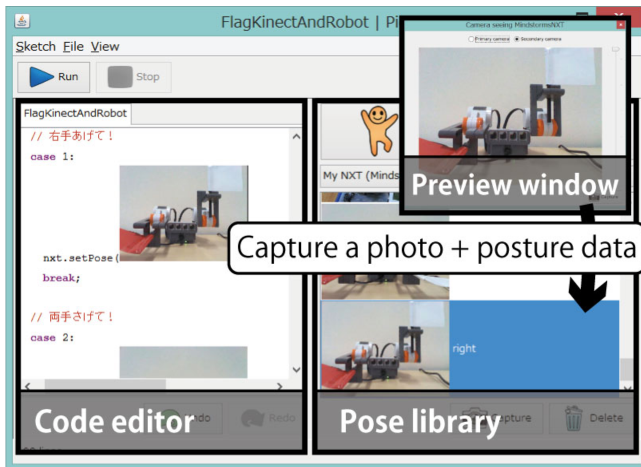
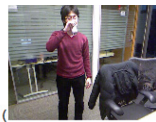


Figure 1 Overview of Picode IDE.

Photos for Understanding Static Complex Data Used in the Program (Picode)

While any image data can be easily visualized as an image as in the case of Sikuli [8], the posture data cannot be visualized unless we know the hardware configuration of the subject. Therefore, it is not trivial to think of its natural representation.

We decided to bind the posture data with a photo of the subject and show it as an inline image in the code editor of the IDE named *Picode* (Figure 1, [4]). We also provided built-in API functions that take photo as their arguments to handle gesture input and control robot postures.

With *Picode*, the programmer first takes a photo of a human or a robot in the preview window. At the same time, posture data are captured and the dataset is stored in the pose library. Next, he writes code in a text-based programming language, extended with a built-in photo-based API whose methods take photos as arguments. He can drag-and-drop photos from the pose library to the code editor, directly into argument bodies of the methods. For instance, a statement comparing the current posture with a stored one can be written as `human.getPose().eq(`  `, 0.02)`. Then, he can run the program by simply clicking the "Run" button.

We conducted a workshop to verify two hypotheses on the benefit of embedding photos in the source code. The first hypothesis was that the inline photos can involve a non-programmer in the software development process since they can be basically taken and understood by anybody. The other was that photos contain richer contextual information compared to mere posture data. These hypotheses do not assume the user is a programmer. Therefore, while one participant of the workshop was a programmer, the rest 28 participants were non-programmers. They were asked to rewrite exiting code through taking photos and replacing existing photos in the code.

Figure 2 Photos taken during the workshop.

The results from the workshop were promising. All of the participants could complete the tasks. They could customize the behavior of the example programs with the visual representations. Even some primary school students could rewrite the text code with help of the programmer. The visual representation helped him to understand the meaning of the code. Photos taken during the workshop revealed their three important roles (Figure 2). First role is to tell the user about the surrounding environment of the subject (human or robot). Second role is to allow the user to demonstrate the part of interest in the subject by simple pointing. Third role is to capture emotions at the moment.

Videos for Understanding Dynamic Behavior of the Program (DejaVu)

When the program is simple enough for the programmer to simulate its execution steps in his mind, he might be able to debug the source code without the real execution. However, when the program deals with real-world I/O such as images from a camera, the simulation is almost impossible. Therefore, he needs to execute the program and monitor continuous visual data, which is not easy with a general text-based IDE. The debugger is usually text-based and forces him to monitor the real-world I/O as discrete textual values rather than continuous visual representations.

We addressed this issue by implementing *DejaVu* (Figure 3, [4]), an IDE that adds two interlinked interfaces named *Canvas* and *Timeline* to an existing text-based IDE.

Reflecting the continuous and visual nature of the real-world input and its processing, *Canvas* allows the programmer to continuously monitor any number of variables during run-time in an arbitrary layout. For data types that are inherently visual (most notably image and body skeleton), the variable values are shown in their appropriate visual form. To add a variable to monitor, the programmer simply drags it from the code editor onto *Canvas*. A display box representing the variable value then appears as labeled by the variable name, which can be freely repositioned through dragging, or deleted when no longer needed. In addition to variables, available types of input from the camera (in the case of Kinect: color, depth, and skeleton) as well as the rendered application window can be inserted into *Canvas* via a checkbox. The above actions together allow the programmer to monitor any input, intermediate result, or output of the program. When the program is running with live input from the camera, each display box reflects the value from the latest frame that has just been captured and processed. Unlike conventional debug watch tables in which the variable values are only

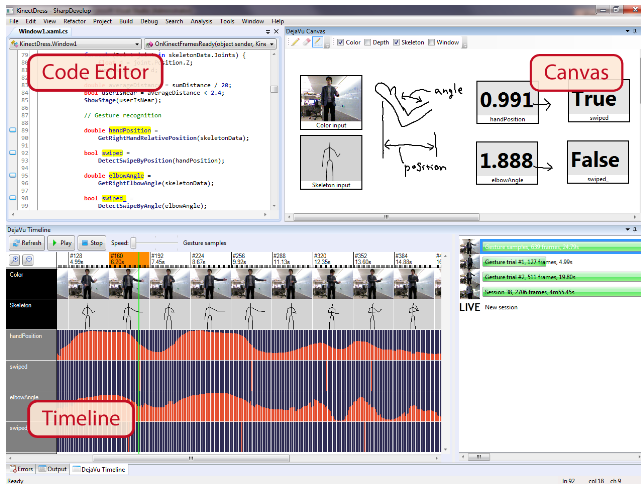


Figure 3 Overview of DejaVu IDE.

updated when the program reaches a break, *Canvas* is constantly updated at every new frame so the values can be continuously monitored in real time.

Timeline automatically records all data shown in *Canvas*. It visualizes the data in a temporal manner, and comes with a playback interface to allow their replay. Replaying by default happens at the same speed as the original live program, i.e., "real-time", but can also be sped up or slowed down according to the programmer's needs using a slider. *Canvas* always updates and displays the recorded data in the current frame. The recorded application window output is also shown in a separate window, emulating the live program execution experience.

The power of *Timeline* lies beyond passive review, and in the ability to revise the program and refresh program data by reprocessing recorded input streams, which naturally serves the iterative development process of interactive camera-based programs. After revising their program, the recorded program output becomes obsolete. To solve this inconsistency, the initial prototype of *DejaVu* automatically compiles and re-executes the updated program with the recorded input. This mechanism keeps the videos to be the "live" representation of the program. However, since this process takes tremendous time, we changed this update process as an optional operation triggered by clicking the "Refresh" button.

To gain early feedback about the concept and functionality of *DejaVu* from target users, we invited three professional developers. They had significant experience in developing interactive camera-based programs using the mainstream text-based IDE. Each participant was asked to use *DejaVu* in the development of a simple interactive program for an hour. The program idea was proposed by the participant based on their past experience. These included a program to track the object held in the user's hand, a program to shift the user's image to the center of the screen, and a program to detect whether the user's left, right, or both hands are raised.

One participant successfully completed his program in one hour while the other two reached a stage that the substance of the program was ready and needed refinement; both were comfortable leaving the program for later work at that point. All participants agreed that it is very useful for developing interactive camera-based programs, and it matches well with their current workflow in developing such programs. Participants found that *Canvas* was an indispensable component and cherished the ability to continuously see immediate result of variable values. *Timeline* immediately resonated with the participants, and were seen as the core competency of *DejaVu*. More importantly, the inseparable link between *Canvas* and *Timeline* defines the *DejaVu* development experience. Both were seen as complementary to each other and their synchronous connection was seen as the significant advantage.

Interactive Videos for Understanding and Creating the Program (Visionsketch)

In the previous projects, we investigated how photos and videos help understanding the static and dynamic aspects of the program. They represent static data, program input, variable contents and program output. These are read-only representations in that the programmer cannot directly manipulate them while he can take a new photo or execute the program again to update them.

Our current interest is to investigate how they can help creating the program. In particular, we are interested in programs that can extract useful information and detect interesting events from time-lapse photos and videos. We observed the development process of such programs to find two distinctive challenges. First, many image processing algorithms used in such programs take an image as input. Their other parameters often have visual meaning, such as four *Point* objects denoting a rectangular area in the image. Output from the algorithms is often also an image. Second, the program needs to be executed for many times with specific input data for debugging. During this iterative process, the programmer often narrows down his interest on the input data. While the original input data are a complete set of photos or a video, he gradually becomes interested in the specific part of the original input.

Visionsketch (Figure 4, [2]) is our ongoing project to tackle these challenges by providing a tight integration of visual and textual programming and a playback interface to control program execution.

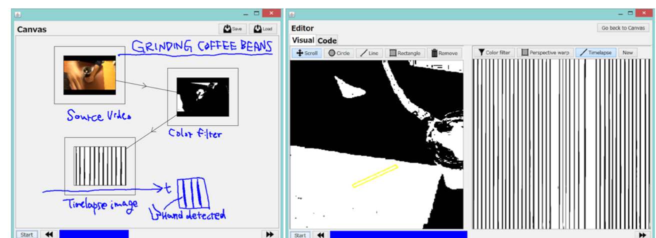


Figure 4 The main view and detail view of Visionsketch IDE.

Tight Integration of Visual and Textual Programming

The main view of *Visionsketch* provides the overview of the program and looks like the *Canvas* interface of *DejaVu* but adopts a dataflow visual programming language like [6, 7]. There is initially one vacant box. The programmer clicks it to choose the input data from existing set of time-lapse photos, a video, or live camera input. Then, he drags a line from an existing box to another place to add a new box representing an image processing component. When the programmer clicks an existing box, the detail view appears to allow editing the details of the corresponding component.

Within the detail view, the programmer first specifies the region of interest (ROI) by drawing shapes on the input image. Next, he can choose an image processing component from existing components which are capable of processing the provided ROI. Then, the processing result is shown next to the input image. If the result is not satisfactory, he can edit the ROI, choose another component, or switch to the code editor to edit the implementation of the current component. Alternatively, he can use the code editor to implement a new component which processes the ROI. When he switches back from the editor to the graphical view, the code is automatically compiled and executed so that the processing result is updated.

With support of these visual interfaces, it is expected that the programmer does not have to write a boilerplate but can concentrate on the actual development tasks of testing more combination of parameters and editing the implementation.

Playback Interface for Program Execution Control

Visionsketch provides a playback interface for flexible control of program execution. While *DejaVu* also provides a playback interface, it is used for mimicking the execution by replaying the recorded information. On the other hand, *Visionsketch* literally runs the program when the playback interface moves time forward. Unlike general step-by-step execution, this interface is specialized for image processing programs and allows frame-by-frame execution. When the input data is given from a camera in real time, the interface can only "play" or "pause" the execution. Frames that arrive while being paused are discarded. Otherwise, when the input data is from recorded photos or a video, it is also capable of jumping to a specific frame, executing a specified section repeatedly (A-B repeat), slowing down or speeding up the execution which is usually done in the original frame rate such as 30 frames per second. For instance, the programmer can jump to a specific frame of interest in the source video file and repeat the execution of 150 frames from there. The "tape recorder" in [6] can also play back the recorded time-series data to run the program, but its control is limited to play back the entire data once or continuously in a loop.

With this playback interface, the programmer is expected to be able to test the program with various input data in a more casual way, which accelerates the development process.

PROGRAMMING WITH EXAMPLE DATA

In the development process of programs that deal with real-world I/O, the development environment is in the computer and the running environment is in the real world. Every one of these three projects addresses this gap and tries filling it by recording information in the real world, such as posture data of human and robots, color and depth images, time-lapse photos and videos. This example information is used to help the development process which involves text-based programming. Unlike "programming by example" where the user only demonstrates examples, core logic of the programs are explicitly specified by the programmer. We call this development process "programming *with* example *data*." *Data* is added not to be confused with "programming with example *code*."

Actually, programming *with* example data has already been done in various forms such as unit testing and machine learning [5]. The scope of my thesis is in its previously unexplored subset which can be effectively addressed by introducing visual representations of the real world. In every project, the example is hardly represented by text or a symbol and photos and videos are used instead.

CONCLUSION

My thesis examines how the visual representations can help programming with real-world I/O. They help understanding static and dynamic aspects of the program. Our ongoing work is to show their capability of creating the program.

REFERENCES

1. Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., and Gee, J. Reflective physical prototyping through integrated design, test, and analysis. In *Proc. UIST '06*, 299-308.
2. Kato, J. Visionsketch. http://junkato.jp/visionsketch/
3. Kato, J., McDirmid, S., Cao., X. DejaVu: integrated support for developing interactive camera-based programs. In *Proc. UIST '12*, 189-196.
4. Kato, J., Sakamoto, D., Igarashi, T. Picode: inline photos representing posture data in source code. In *Proc. CHI '13*, 3097-3100.
5. Patel, K., Bancroft, N., Drucker, S. M., Fogarty, J., Ko, A. J., and Landay, J. Gestalt: integrated support for implementation and analysis in machine learning. In *Proc. UIST '10*, 37-46.
6. Paul, E., H. ConMan: a visual programming language for interactive graphics. In Proc. *SIGGRAPH '88*, 103-111.
7. Tanimoto, S. L. VIVA: a visual language for image processing. *Journal of Visual Languages & Computing 1*, 2 (1992), 127-139.
8. Yeh, T., Chang, T. H., Miller, R. C. Sikuli: using GUI screenshots for search and automation. In *Proc. UIST '09*, 183-192.