Integrated Graphical Representations for Development of Programs with Real-world Input and Output

（実世界入出力を伴うプログラムの
画像表現を用いた開発支援手法）

by

Jun Kato
加藤淳

A Doctor Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on December 13, 2013
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Takeo Igarashi　五十嵐健夫
Professor of Computer Science

ABSTRACT

Programs that use real-world input and output (real-world I/O), including interactive camera-based programs and robot applications, have different development and runtime environments. While integrated development environments (IDEs) reside in the traditional desktop environment, the runtime environment is in the real world. Real-world I/O may apply to specific situations and/or respond to dynamic changes that cannot be represented intuitively by the existing text-based user interfaces of IDEs. Previous efforts to eliminate the gap between the development and runtime environments include Programming by Example (PbE), in which the user demonstrates operations to the system and the system infers and executes a corresponding program. In PbE, the program is specified using the runtime environment. The drawback is that the PbE system does not allow the user to precisely describe the logic of the program.

In this dissertation, we coin the term "Programming with Example (PwE)," which describes a hybrid approach combining PbE and text-based programming. It makes use of example data retrieved from the real world during text-based programming. We propose to integrate graphical representations, such as photos and videos, which represent real-world I/O data, into text-based IDEs. In particular, we provide a model of the program that deals with real-world I/O, that is, $out = f(in, c)$, where c describes static parameters provided prior to the execution of the program (i.e., constants), *in* and *out* are dynamic input and output provided during the execution of the program (i.e., are variables), and f is the specification of the program (i.e., the functionality). Accordingly, we discuss three types of graphical representation.

First, we discuss the use of photos as graphical representations of situations in the real world. As an experimental implementation, we present *Picode* IDE, which supports the development of posture data processing applications that handle posture information of humans and/or robots. It uses photos to represent the posture, where textual references may not be very intuitive. It allows the programmer to take a photo of the subject to automatically capture the posture information, supporting PwE by retrieving static data from the real world. Photos are shown inline in the source code editor and provide contextual information that facilitates visualization of the posture, providing an intuitive coding experience.

Second, we discuss use of videos as a graphical representation of variables where the contents are dynamically updated during the execution of the program. As an experimental implementation, we present *DejaVu* IDE, which supports the development of interactive camera-based applications. It provides visualization by automatically recording the camera input, intermediate processing results, and window output. It supports PwE to record and utilize the dynamic behavior of the program. The programmer can replay the recorded data using an interface that resembles a video player to visualize the behavior. It also allows the programmer to update all of the output from the program by re-executing it using the recorded input data.

Third, we discuss a method to specify the behavior of the program with the help of graphical editing. As an experimental implementation, we present *VisionSketch* IDE, which supports development of image processing applications, detecting interesting events from videos recorded with a fixed viewpoint. It requires the programmer to specify the video source at the beginning of the development for PwE support. The programmer can specify regions of interest in the example video to narrow down the list of applicable image processing algorithms or to setup parameters of a selected image processing algorithm.

These IDE implementations are designed for different target applications. However, each of them supports the entire workflow of the programmer by integrating a graphical representation into a text-based programming environment. Each graphical representation corresponds to I/O data sampled from the real world. While we focus on visual information in this dissertation, there are other types of real-world I/O data, including sound, haptic technology, smell and taste. We foresee that applications that make use of such multi-sensory data will become increasingly important in the future. The development of programs that deal with real-world I/O inherently benefits from PwE. Therefore, we believe that the findings described in this dissertation will serve as an important foundation to support the development of such applications.

論文要旨

カメラ映像のリアルタイム処理やロボットの制御のような実世界入出力を伴うプログラムの開発では、統合開発環境 (IDE) がデスクトップにある一方で、実行環境は実世界である。プログラムの入出力データは実世界における状況や状況の時間変化を表しており、文字列表現ベースの既存の IDE では直感的に表現できない。このような開発環境と実行環境の溝をなくす試みとして、例示をもとにシステムがプログラムを推論及び実行してくれる例示プログラミング (Programming by Example, PbE) が提案されている。PbE は、実行環境においてプログラムを指示できるために前出のような問題が生じない一方、通常のプログラミングのようにロジックを精密に設計することが難しい。

本論文では、文字列表現を用いたプログラミングにおいて、PbE のようにシステムに例示した実世界入出力のデータを活用することを Programming with Example (PwE) と呼ぶ。そして、PwE を支援するために、実世界入出力のデータを写真や動画のような画像表現で表して IDE に統合する手法を提案する。とくに、実世界入出力を伴うプログラムを $out = f(in, c)$ というモデルで表し、定数 c、変数 in と out、そして関数 f のそれぞれに対応する画像表現について議論する。

まず、実世界における状況を表す静的な定数 c を、写真を用いて表す手法について議論する。試作システムとして、人やロボットの姿勢情報を処理できるアプリケーション開発を支援する *Picode* IDE を提案する。文字表現が難しい人やロボットの姿勢データを写真で表し、ソースコードエディタに直接貼れるようにした。また、写真撮影と姿勢データの取得を同時に行うことで、静的データを活用する PwE を支援した。写真により実際の姿勢を想像しやすくなるため、直感的なコーディングが実現できる。

次に、プログラム実行中に動的に変化する変数 in と out を、動画を用いて表す手法について議論する。試作システムとして、インタラクティブなカメラ入力を用いたアプリケーション開発を支援する *DejaVu* IDE を提案する。カメラ入力、変数およびウィンドウ出力を可視化し、さらに自動的に録画することで、プログラムの動的な挙動を記録して活用する PwE を支援した。録画データを動画プレイヤーのように再生して挙動を深く理解したり、修正したプログラムに入力データを再度与えてデバッグを行える。

さらに、プログラムの処理内容 f を、文字列ベースのソースコードだけでなく画像表現に対する編集操作も利用しながら指定する手法について議論する。試作システムとして、定点カメラで撮影された動画から有用な情報を抽出できる画像処理アプリケーション開発を支援する *VisionSketch* IDE を提案する。プログラムを開発する際にまず入力データとなる動画を指定することで、次のような PwE を実現した。すなわち、動画像のなかで興味のある領域を図形ツールにより描画することで、適用可能な画像処理を絞り込んだり、画像処理のパラメタを直感的に指定できる。

各 IDE は異なるアプリケーションを開発するための環境であるが、いずれも、実世界入出力の例示データを表す画像表現を文字列ベースのプログラミングに取り入れることで、プログラマのワークフロー全体を支援できている。本論文では実世界入出力のなかでも視覚的な情報を扱ったが、その他の音や触感、匂いや味などを利用するアプリケーションが今後重要性を増してくると考えられる。実世界入出力を伴うプログラムの開発では例示データを活用するプログラミングが必要であるため、我々は、本研究の知見がそのようなプログラムの開発支援手法における重要な基盤となると信じている。

Acknowledgements

This dissertation is the result of six years of research that has been supported by many mentors and friends across multiple countries, too numerous to mention. My thanks goes out to all the individuals that have had an influence on my work that I cannot mention here.

First and foremost, I would like to express my deepest appreciation to my supervisor, Takeo Igarashi, who has supervised me throughout my research career. I started my career as a research assistant at his JST ERATO Igarashi Design Interface Project (Igarashi ERATO project) at the time of its launch. One year later, I joined his research group at the University of Tokyo, completed the Master's program in two years, was admitted to the Doctor's program, and spent three years on research that would eventually make it into this dissertation. Throughout the years, he has given me a supportive push forward to explore and focus on what I can really be passionate about. Discussions with him have been always fruitful and sometimes even magical, for instance, when a vague research question suddenly becomes crystal clear. I would also like to thank my co-supervisor Daisuke Sakamoto who has also mentored me for six years. He was initially my mentor at Igarashi ERATO project and later became my co-supervisor of the Doctor's program. He has patiently guided me to be continuously productive and has helped me attain a higher level of professionalism. I am sincerely grateful to my committee members, Masami Hagiya, Shinichi Honiden, Shigeru Chiba, Toshiyuki Masui, and Yoichi Sato for their constructive feedback on this dissertation. They provided me with many insightful comments in various academic areas including Programming Language, Software Engineering, Computer Vision, and Human-Computer Interaction.

I thank past and present members of Igarashi laboratory, especially Kenshi Takayama, Nobuyuki Umetani, Hideki Todo, Sosuke Okamura, Yuji Yasuda, and Makoto Nakajima, who spent a long time with me discussing interesting topics, as well as giving me a comprehensive introduction to their own favourite research topics. They also taught me practical ways to survive in the laboratory, and I appreciate all the good times they spent with me. The same thanks go to visitors to the laboratory, including Erik Andersen, Daniel Rea, and Lasse Laursen. I'm always keen to improve my English, and as a result of frequent dialogues with visitors to the lab, it is now better than ever. Outside the laboratory, I have also had many fabulous mentors and friends. I would like to thank the members of the Igarashi ERATO project, especially Masahiko Inami, Yuta Sugiura, Kohei Matsumura, James E. Young, Ayumi Fukuchi, Charith Fernando, and Shigeo Yoshida who helped me in various ways, including having constructive discussions, managing workshops, and just laughing together. I also want to thank Yutaka Ishikawa, who was the project manager for the Information-Promotion Agency Mitoh program in which I built a software library used for the Picode project. I am thankful to the members of Rekimoto laboratory who share multiple research interests with our own laboratory, especially Kensaku Kawauchi and

Adiyan Mujibiya. My thanks also goes out to the members of the OpenPool developer team, which is a collaborative project that aims to provide an open-source framework for augmenting a billiard table with interactive projection mapping. I enjoyed contributing to the project by creating its initial prototype and writing an academic paper on the prototyping process.

I have spent considerable time abroad, collaborating with fabulous mentors at Microsoft Research and Adobe Research. I would especially like to thank Xiang Cao who mentored me at Human-Computer Interaction (HCI) group, Microsoft Research Asia. He trusted me, discussed interesting topics with me so often and showed an exceptional passion in research that helped me to complete the DejaVu project and publish a UIST paper. He introduced me the other mentor, Sean McDirmid, who taught me an exciting aspect of Programming Language (PL) research. The collaboration with PL research repeated as I joined TouchDevelop team of Research in Software Engineering group, Microsoft Research Remond. Sebastian Burckhardt and Michal Moskal mentored me there and Thomas Ball spent some time discussing the intersection of PL and HCI research. This collaboration eventually became a PLDI paper. Throughout my internships at Microsoft, Noboru Kuno guaranteed my comfortable experience, for which I am grateful. When I presented the DejaVu project at UIST 2012, Joel Brandt at Adobe Research kindly spared his time discussing research on integrated development environments. This dialogue resulted in my internship at Adobe Research mentored by Jovan Popovic and Joel Brandt. All of these internships gave me new perspectives into research and opportunities to meet new professors and friends, forming the essential part of my Doctor's program life. Among all professors abroad, I would like to thank Bjoern Hartmann at University of California Berkeley, Robert C. Miller at Massachusetts Institute of Technology (MIT), and Andrew J. Ko at University of Washington (UW), for hosting my visit, discussing with me, and introducing fantastic students such as Tom Lieber at MIT and Brian Burg at UW.

Finally, I would like to express my gratitude to my parents, Chieko and Yasuichiro Kato, and my wife, Nana Kato. Your love, and continuous support during several overnight paper submission sessions, has made all of this possible.

Contents

Chapter 1

Introduction

This chapter gives an overview of work described in the dissertation. First, we briefly describe the motivation for this work and provide a context for it. Then, we introduce our unique approach to integrated graphical representations and summarize the main contributions of this work. Finally, we outline the structure of the following chapters.

1.1 Motivation

An integrated development environment (IDE) is a set of user interfaces with which the programmer can write, compile, execute and debug a program. The most commonly used IDEs, including VisualStudio, Eclipse and Xcode, are all text-based and used for development of any type of program. Most programmers these days use text-based programming languages and use text-based development environments to create programs.

Programmers are people, too. Therefore, it is important to think of human factors in development environments to improve the productivity of programming activities. One way of taking human factors of computer systems into account is to consider the gulf of execution and evaluation [121]; i.e., the gap between the user's intention and the results of their actions. It may also be applied to development environments [81]; when a programmer develops a program using a text-based programming language, there may be a gulf of execution, as shown in Figure 1.1. It is not straightforward to correctly translate the intent to the text-based code. Code completion and interface builders aim to bridge this gap. Furthermore, when a programmer debugs the program using a text-based debugger, there is a gulf of evaluation. It is not easy to understand its dynamic

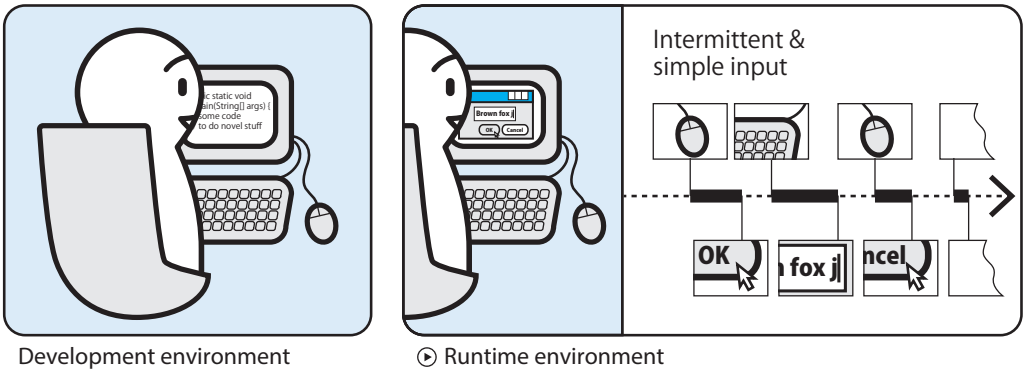


Figure 1.1: Development of the programs with conventional input and output.

Figure 1.2: Examples of programs with real-world input and output.

behavior from textual information presented by the debugger. A debugger that visualizes program execution [94] and one that helps reasoning errors [80] may assist with this. As seen in these examples, a typical approach to bridge the gulf of execution and evaluation in programming is to provide an appropriate graphical user interface (GUI), which provides visual cues in development environments and connects the static description of the program and its resulting behavior.

Compared with the relatively conservative development of IDEs over the past several decades, the variety of computer applications has grown considerably, and this evolution has been accompanied by new input and output modalities. The computer was invented as a machine to automate calculations, and the programs were stored as punched cards and executed in a batch manner without user interaction. The keyboard and display later provided scope for more interactivity through character-based user interfaces (CUIs). Computers subsequently became personal devices, following the realization of GUIs. In addition to the keyboard, a mouse was used to provide information on the status of several buttons and movement in two-dimensional space. The two-dimensional movement was explicitly bound to the movement of a pointer, which is part of a windows, icons, menus and pointer (WIMP) environment. Numerous efforts have been made to improve user interface tools [116], and event languages represent a successful standardization of the user input, and map physical actions to GUI operations. Event information is provided to the programs intermittently, each time user input occurs. Post-WIMP paradigms include recognition-based interfaces [101] with new devices, such as the Freeform User Interface with pen input [67], gesture-based touch interface and voice recognition with a microphone. While these early attempts made use of new input modalities to control GUI applications, recent trends place more focus on physical interactions in the real world.

Interactive programs that deal with real-world input and output (real-world I/O) are growing in popularity. Such applications include camera-based interactions, augmented reality, tangible user interfaces, physical computing, and user interfaces for robots. Examples are shown in Figure 1.2. In these applications, there are no standardized I/O events. Raw I/O data are received from sensors and sent to actuators, and have to be processed by the program continuously in real time. In particular, this dissertation focuses on a certain kind of real-world I/O whose data is best represented visually by photos and videos. Potential application of our method to other kinds of real-world I/O such as audio, tactile sensation and smell will be discussed in Subsection 7.2.2 but is not our main contribution.

When the program deals with real-world I/O, the gulf of execution and evaluation becomes wider. This results from the use of different development and runtime environments, as shown in Figure 1.3. When the program uses a conventional CUI or GUI, the development environment shares the same I/O devices as the runtime environment, as shown in Figure 1.1. The programmer uses the

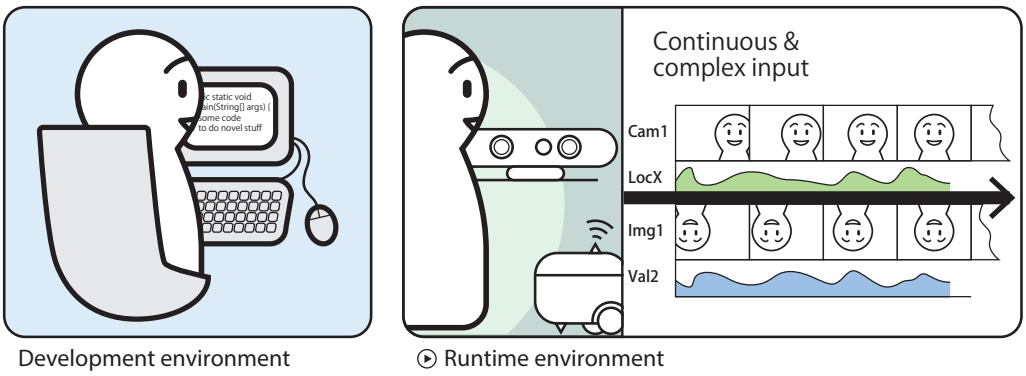


Figure 1.3: Development of the programs with real-world input and output.

mouse and keyboard to both develop and run the program. In this case, input to and output from the program can be intuitively represented and reproduced by the user interface, using either a CUI or GUI. For instance, a keystroke can be represented by a character code and the movement of a mouse can be represented by change of the location of two-dimensional coordinates. The primary difficulty in bridging this gulf was how to visualize and provide intuitive navigation over complex data structures and the dynamic behavior of the program. However, when the program deals with the real-world I/O, the development environment typically employs conventional CUI or GUI with a mouse, keyboard and display, yet the runtime environment may involve physical movements of one or more of users, objects or robots, which cannot be represented well by the existing user interfaces of IDEs. Therefore, it is difficult to develop and debug such programs. Please note that the scope of this dissertation is to aid development of interactive programs by filling the gap between the I/O modalities. While there have been much work on remote debugging tools that aims to fill the gap between a computer that develops the program and the other computer (typically a microcomputer) that runs the program, our work assumes that the program is developed and ran on the same computer but with different set of I/O devices.

Existing approaches to address this gulf include programming by example (PbE), in which the user demonstrates operations to the system and the system guesses the program [33, 93]. When it is applied to the development of programs that deal with real-world I/O, it can eliminate the need for explicit programming, and the user does not require prior knowledge of programming. In PbE systems, the program is specified using the runtime environment; in other words, there is no distinction between the development environment and the runtime environment. Therefore, the user does not need to alternate between different modalities, and the gulf of execution and evaluation of programming can be removed. The drawback is that it does not allow the user to precisely describe the logic of the program. While it may be sufficient for end users, another gulf of execution and evaluation arises for the programmer who wants complete control over the resulting program. It is difficult for him to infer what kind of and how many examples are sufficient to realize his intent. It is also difficult to test the outcome logically. The programmer has to give more and more examples to test whether the program functions as intended.

1.2 Contributions

The work in this dissertation addresses the issues described above. The main contributions are:

1. Observation of the programmer's workflow to develop programs with real-world I/O, termed "Programming with Example".
2. A model of programs with real-world I/O built from observation.
3. Three kinds of integrated graphical representations in text-based IDEs, which provide distinctive support for every component of the model.

We coin the term "Programming with Example (PwE)", which describes a hybrid approach combining PbE and text-based programming. It is supported by enhancements to existing text-based IDEs, which integrate graphical representations of the real world. First, the programmer demonstrates interactions in the real world to the IDE, which are recorded as example data. Then, he writes text-based code with the help of the example data, exposed as graphical representations, including photos and videos. The proposed integration method combines abstract textual representations and example graphical representations, which complement each other to enhance productivity of the programmer. Our approach is an attempt to bridge the gap between the development environment and the runtime environment in a similar manner to PbE; however, it still allows the programmer to explicitly describe the logic of the programs using text-based programming.

To investigate how integrated graphical representations may assist the workflow of programming with examples, we first provide a model of the program that deals with real-world I/O. Then, we provide a graphical representation for each component of the model, which supports programming using examples. The output of this approach can be described using the following model:

$$out = f(in, c)$$

where c describes static parameters provided prior to the execution of the program (i.e., constants), *in* and *out* are dynamic input and output provided during the program execution (i.e., variables) and f is the specification of the program (i.e., functions). Accordingly, we propose three types of graphical representation.

First, we discuss graphical representations of constants, which represent situations in the real world at a certain moment. As an experimental implementation, we present Picode IDE [72], which supports intuitive programming using inline photos that represent posture data. It deals with static complex data used in the program, specifically, data describing the posture of humans and robots. Such data are commonly used to handle gesture input and control robots. Static complex data in general are better understood using visual representation than textual references such as a filename. Sikuli [150] addresses this issue by introducing an editor with inline images, which serve as the API arguments. We developed Picode by applying a similar concept to posture data.

Second, we discuss graphical representations of variables, the content of which come from the real world, and are dynamically updated during execution of the program. Graphical representations of constants help the programmer to understand the static part of the source code; however, there remains a difficulty in understanding the 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 exactly what happens in the program. Therefore, the programmer must execute the program and monitor continuous visual data, and this kind of functionality is not supported by current mainstream IDEs. As an experimental implementation, we present the DejaVu IDE [70], which addresses this issue by providing two interlinked components that record and visualize program input and output.

Third, we discuss graphical representations of functions, each of which deals with real-world I/O. The graphical representations in the previous two cases primarily serve as visual aids that assist in understanding the program; they do not directly support building the program. In contrast, graphical representations of functions can be manipulated interactively by the programmer to change the behavior of the program. As an experimental implementation, we present the VisionSketch IDE, where every function of an image-processing program has its own graphical representation, and some of the arguments can be specified by editing the graphical representation.

1.3 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we describe the background for this work. We begin with a historical perspective on IDEs, where we introduce related work that provides support for programming activities. This includes end-user programming, visual programming environments, programming by example, and previous efforts on enhancements to text-based IDEs, including enhancements to editors and debuggers. It also includes tools for the programmer, which provide explicit support for the development of programs with real-world I/O.

In Chapter 3, we describe the terminology, and explain the model for a program that deals with real-world I/O. We detail our assumptions and define our goals, which will be examined in the subsequent three chapters. Programs that use real-world I/O tend to widen the gulf of execution and evaluation, since they are typically developed and executed using different environments. While the development resides in the traditional desktop WIMP environment, the runtime environment is in the real world. In such a case, programming activity inherently involves sampling and handling real-world I/O data. We call this "programming with example" and aim to improve productivity by providing enhancements to existing text-based IDEs.

Chapters 4-6 describe our attempts to achieve this goal. We suppose that this gap can be addressed by providing proper graphical representations of the real world in the development environment. We provide a model of the program, i.e., $out = f(in, c)$, and map graphical representations to each component. Within this model, the constants described by c are discussed in Chapter 4, the variables in and out are discussed in Chapter 5, and the functions f is discussed in Chapter 6.

In Chapter 7, we analyze the contributions made using our approach, describe the limitations of it, and identify directions for future development. We aim to show that graphical representation in text-based IDEs can aid the development of programs that deal with real-world I/O through distinctive support for programming with examples. While we focus on two-dimensional graphical representations, investigation of three-dimensional graphical representations may be desirable. Real-world I/O is not restricted to visual information, which is the focus of the work described in this dissertation, but also includes sound, haptic

technology, smell and taste. We foresee that future IDEs will be equipped not only with graphical interfaces but also with multimodal interfaces to bridge the gap between the development and runtime environments. When IDEs become more accessible through such enhancements, we expect that end-users will be able to learn to program more easily, opening up a future in which everyone can be a programmer. Research on live programming has recently focused on technical contributions, but our work suggests that there is also much left to do from Human-Computer Interaction perspective.

Chapter 2

Background

In this chapter, we discuss the background to this dissertation. Related work on how to develop programs is described and compared with our attempts to support the development of programs with real-world input and output (real-world I/O). In Section 2.1, brief history of integrated development environments (IDEs) is given to provide a historical context to the following sections. In Section 2.2, end-user programming (EUP) is introduced, which aims to reduce the threshold for programming. Examples of EUP are given, including visual programming environments and programming by example. In Section 2.3, enhancements to text-based IDEs are introduced. Recently, live programming environments that aim to eliminate the gulf of execution have attracted attentions, and these are explained. Then, online collaborations on text-based IDEs and domain-specific text-based IDEs follow. Next, the integration of graphical representations into text-based IDEs is introduced. Building blocks for these IDEs that accelerate research are also covered. In Section 2.4, tools for the development of programs that deal with real-world I/O are introduced, highlighting the requirement to support the whole of the workflow.

2.1 Historical Perspective

An IDE is an environment equipped with a set of interactive user interfaces with which the programmer can write, execute, and debug a program. It contains programming languages, compilers, debuggers, toolkits and other tools that may be used for programming. In this dissertation, we focus on discussing the environment rather than each tool. The environment supports the entire workflow of programming activity, while the tool for programming usually covers a subset of programming activities.

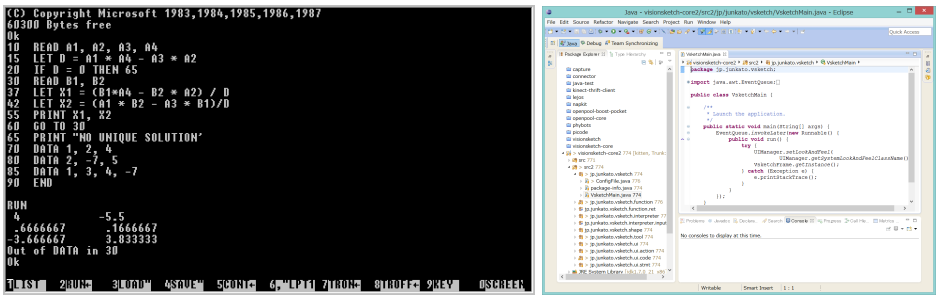


Figure 2.1: Traditional text-based integrated development environments. Left: Dartmouth BASIC, Right: Eclipse [42].

In the early days of computer programming, there were very few interactive user interfaces. The programmer used punched cards to write machine code to provide commands to the computer. Punching cards was a physical task, and there was no computational support.

The first IDE is commonly said to be Dartmouth BASIC as shown in Figure 2.1, which was developed in 1964, after character-based user interfaces and high-level text-based programming languages had appeared. It was equipped with a text-based editor, as well as a compiler, which could compile and run a program within the environment. At this time, computers were not personal, rather shared by a number of users in a similar manner to the way in which supercomputers are used today. Dartmouth BASIC was an environment for the Dartmouth Time Sharing System (DTSS), and everyone using this system was a professional programmer. Much of the pioneering work on the interactive features that are widely considered to be essential prerequisites for modern IDEs was carried out in the 1970s, and details of this work can be found in the book Interactive Programming Environments [8], including the appearance of integrated debuggers, program analysis tools and structured editors. Providing a more interactive develop environment was thought to be important for productivity. For instance, Interlisp, an IDE with a built-in debugger and analysis tools, was actively developed from 1967 into the 1970s at Xerox PARC. The Cornell Program Synthesizer and MENTOR were IDEs equipped with structured editors for the PL/I and Pascal programming languages, respectively.

In the 1980s, human factors in programming began to receive greater attention. This followed IBM's introduction of the IBM Personal Computer in 1981, as the number of novice programmers and end-users with little knowledge of programming increased significantly. Some researchers who were interested in human factors began to focus on end-user programming and graphical user interfaces (GUIs), which can be used without prior knowledge of programming. Major forums for research into human–computer interaction (HCI) were born of this age, including the ACM SIGCHI Conference on Human Factors in Computing Systems (called simply CHI since 1982) and ACM Symposium on User Interface Software and Technology (UIST, where the first workshop was held in 1982). Forums specific to programming also appeared in the 1980s, including the Workshop on Empirical Studies of Programming (1986–1999) and the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC since 1984). Research into software engineering and programming language design was also growing rapidly at this time. The International Conference on Software Engineering (ICSE) has been the premier software engineering conference, and has been cohosted by the ACM and the IEEE since 1975. Programming language design remains an ongoing research topic, with forums including the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), and the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). While there have been some research projects aiming to enhance text-based IDEs, this area of research has received less interest than programming language design. Design principles for text-based programming languages, such as those described in Hoare's paper [66], are readily available. It is more difficult to find work discussing the design of development environments; however, some examples will be covered later in this subsection.

The end-user typically uses a program as it is. Therefore, they must map their intention to existing commands, and often must repeat such commands many

times. To avoid such tedium, they may wish to create, modify or extend existing programs. End-user programming is a research topic that addresses many of the difficulties that such end-users and novice programmers have with programming. We can identify two approaches: one is to make programming easier, and the other is to eliminate explicit programming completely. For instance, research into structured text editors and visual programming falls into the first category and research on programming by example falls into the second category. These will be described in Subsection 2.2.1 and Subsection 2.2.2. The taxonomy is explained well in Myer's survey paper on visual programming and program visualization [114].

While visual programming languages usually come with their own graphical development environments, text-based IDEs have retained a similar look since the days of BASIC. They include a file manager, a text-based code editor, and a text-based debugger. For instance, Eclipse [42] is shown in Figure 2.1. There are few graphical representations. While tools for programmers have previously focused on providing supplemental features that are not supported by text-based IDEs, the text-based IDEs themselves have not significantly changed. User interface toolkits have been widely studied [116]; however, text-based IDEs remain the primary means of programming for most professional programmers.

Throughout the 1990s, programming became more complex and involved. For instance, it is not straightforward to specify a GUI using solely a text-based approach. Visual Basic 1.0 [31] was released in 1991 and addressed this issue by integrating a graphical GUI builder into a text-based IDE. It allowed the programmer to seamlessly move between the interface design and implementing functionality. In 1995, Lieberman applied the famous concept of Norman's gulf of execution and evaluation [121] to programming. He pointed out difficulties in understanding the dynamic behavior of a program from the static text-based source code. To address this issue, he proposed a visual debugger, integrated with a text-based IDE [94, 92] which shows the programmer the relationship between an expression in the source code and its output at runtime.

In the 2000s, many enhancements to text-based IDEs appeared. These IDEs will be reviewed in detail in Section 2.3; however, one of the most prominent projects in the enhancement of text-based IDEs is the Natural Programming project [117], which was initiated by Myers and colleagues in 1998. Their previous work highlighted usability issues for novice programmers in programming language design [124], and the Natural Programming project places more emphasis on the IDE and accompanying libraries. Their work takes a human-centered approach: they first investigate the programmer's behavior and then address the difficulties. For instance, they studied six learning barriers in end-user programming systems [81] and discussed the gulf of execution and evaluation, which develops some of Lieberman's ideas [94]. Various technical contributions were made following the study.

Three factors made the proliferation of such ideas feasible. First, the Internet has enabled new ways of collaboration between programmers, and given rise to the remarkable growth in open-source technologies. This is discussed in Subsection 2.3.4. This work includes collecting, analyzing and utilizing anonymized usage data of IDEs to identify areas for improvements and provide targeted support for cooperative work through the Internet. Second, the variety of programs has continued to evolve, and has done so at an ever-increasing rate, with growing emphasis on physical interaction, including interactive camera applications and robotics applications. The development of such programs requires tool support, and enhancements to IDEs, which will be discussed in Subsection 2.3.5. Third,

several technological shifts have made IDEs more extensible as explained in Subsection 2.3.7, including extension frameworks, open-source distribution of IDEs, and instrumentation features of programming languages, such as application programming interface for reflection. They allowed the programmer to concentrate on the improvements and leave the rest as provided by the existing IDEs.

Although text-based IDEs have been the dominant means of programming for the professional programmer, until recently most of the user interfaces have remained unchanged. Making enhancements has now become more feasible thanks to the technological shifts described above. New kinds of programs, especially those with real-world I/O have led to demand for enhancements to IDEs. In this dissertation, we aim to respond to such demands by enhancing text-based IDEs in the context of HCI.

2.2 End-User Programming

An end user is any computer user (either a programmer or non-programmer) who accesses a piece of software at the end of the process of development and does not have access to modify the original code. Programs are typically packaged by their creators and are not written to fit a given end-user's particular requirements. EUP was originally a term describing techniques to bridge this gap by allowing the end-user to create, modify or extend the program. This topic will be covered in detail in this section. The phrase was popularized by Nardi in her book published in 1993, discussing the topic based on her experience of EUP systems for spreadsheets and computer-aided design scripts [119]. Early (yet still effective) attempts include application-specific scripting languages, such as those used in spreadsheet and word processing applications for writing formulas and macros. They aim to make programming easier to use or to understand for a broader group of users. Usability issues for novice programmers are discussed in Ref. [124]. Unlike these attempts, which require some knowledge of programming, programming by example (PbE), which we will describe in Subsection 2.2.2 eliminates the requirement to write code. Using a PbE system, the end-user does not need to learn any basic programming concepts, and may simply demonstrate pairs of example input and desired output to the system. The system then infers the program, which can take new inputs. Throughout the history of EUP, making use of visual representations has been thought to be of benefit to the user. Therefore, it includes many visual programming environments, which will be discussed in Subsection 2.2.1.

The number of end-user programmers has increased recently, and is expected to be significantly greater than the number of professional programmers (a 2005 analysis predicted that, in the United States, there would be over 13 million end-user programmers in 2012, compared with less than 3 million professionals [133]). End-user programmers have been defined as those who are "programming to achieve the result of a program primarily for personal, rather public use" [75].

Various popular applications, including Microsoft Office (which includes Word for word processing, Excel for spreadsheet editing, and PowerPoint for creating presentations), Adobe Creative Cloud (including Photoshop for raster graphics editing, Illustrator for vector graphics editing, and AfterEffects for video composition), and Maya for three-dimensional computer graphics, are equipped with built-in scripting environments with which the end-user can automate simple tasks. These days, the World Wide Web has become a platform for various applications. Standardized and machine-readable specifications, including HTML and HTTP, have opened a new opportunity of EUP. For instance, Chickenfoot

[13] provides a scripting environment that allows automation of repetitive operations, integrating features of multiple websites, and customizing the appearance of websites.

Spreadsheet systems are also a popular EUP environment. They are typically capable of managing multiple sheets, each of which is a collection of cells in a grid of rows and columns. The user can input simple textual types of numbers and strings into cells for computation. Some research prototypes go beyond this standard form. For instance, there is a development environment that uses a spreadsheet interface to build information visualization applications [28]. Forms/3 [22, 19] is a spreadsheet-based visual programming language. It does not have a grid of rows and columns, but still provides cells for computation, which can be placed anywhere on the screen. It is also capable of handling interactive graphical objects within the spreadsheet paradigm. Graphical objects can be created by drawing gestures, and can be edited by direct manipulation. Properties, such as the radius of a circle, are dynamically linked to spreadsheet cells, and are shown in text format.

With the increasing capability of EUP environments, many end-user programmers face very similar software engineering challenges as professional programmers. These challenges span the entire workflow of the programming activity, rather than merely the design and specification of a program, which can be addressed, for instance, by PbE. Challenges include determining the requirements of the program, and testing, verifying and debugging the program. End-user software engineering (EUSE) is an emerging field that aims to address these issues by applying software engineering techniques to the end-user by a human-centered approach [21, 75].

When a new type of program is introduced, every programmer is a novice in the development thereof. Therefore, in such a case, the design process of the programming tools for the professional programmer should be similar to or the same as that for the end-user programmer. Patel pointed out this parallelism, and took a human-centered approach to improve the productivity of the development of machine learning applications [126]. He observed the workflow of the programmer, and designed and evaluated an IDE to support that workflow. In this dissertation, we take a similar approach to the development of programs that deal with real-world I/O. For each example project, we first define the target applications and analyze their development workflow; we then design and evaluate an IDE that supports that specific workflow.

2.2.1 Visual Programming Environments

One of the major approaches to end-user programming is the use of graphical programming environments. Advantages are expected because of the ability to specify the program in a two- (or higher) dimensional fashion. Conventional text-based programming is one-dimensional in the sense that it is processed by a compiler or interpreter as a single line of text. However, the programmer may prefer to visualize the program in a structured manner, which may be properly indented multi-line text, or a flowchart. We discuss structured editors in Section 2.3 in the context of text-based IDEs.

Most visual programming languages are characterized by "icons on strings" [89] or box-and-line notations as a set of iconic pictures as shown in Figure 2.2, which are usually static and have a one-to-one mapping with concepts that make up the program, and connections between them, which are usually rendered as lines or arrows and represent relationships between those concepts. They are

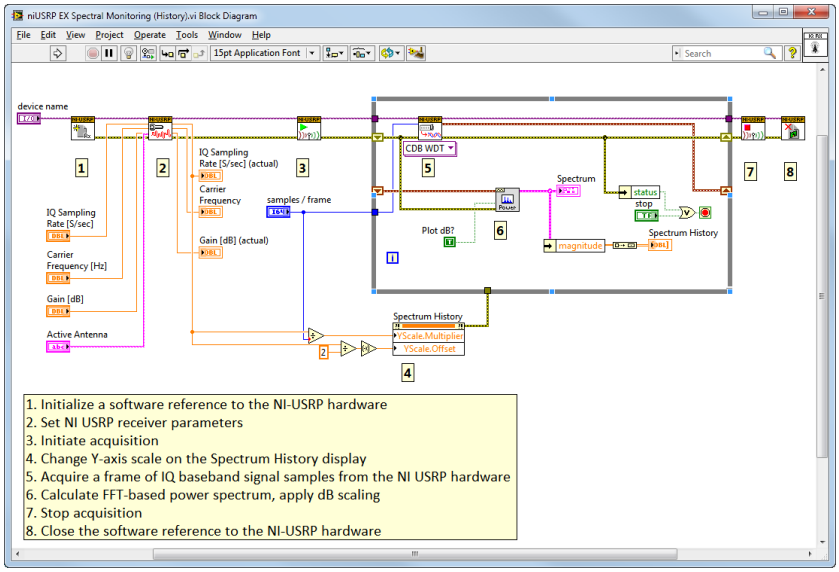


Figure 2.2: An example of symbolic visual programming languages, LabView [68].

symbolic representations of the program code and relations between program modules. Typically, they are also capable of "direct manipulation" [136] of the program, which allows the programmer to use the mouse to visually add, edit or remove the iconic pictures or their connections. These methods are described well in the surveys by Shu and Myers [137, 115]. LabView [68] is one of the most popular commercial implementations of such a system of programming.

Later, evaluations of such systems from a psychological perspective became available [147]. Green and Petre conducted a usability analysis of visual programming environments [50]. Their reflections led to a more general framework for evaluating user interfaces [12] and further end-user programming research, which was introduced at the beginning of this section. For instance, there was a tendency for the research community to believe that "icons on strings"-style visual programming languages were naturally superior to textual programming languages. This attitude has subsequently been criticized [10], and a balance between textual and visual representations has been found to be desirable, rather than aiming for a purely visual environment, which has been shown to have a number of shortcomings. For instance, with a purely visual representation, it is more difficult to understand and use screen space efficiently [119]. They are not suitable for large realistic programming problems [20]. Visual components require layout rearrangement when there is any change, which is often tedious [50]. Recent discussions provided detailed guidance of when to use the various types of metaphorical graphics in visual programming systems [11].

There have also been attempts to go beyond symbolic notation as shown in Figure 2.3, adding more meaning to the temporal changes by considering the spatial positions of the components. Among all visual programming environments, systems that belong to this category are the most relevant to our approach. The difference is that we aim to investigate ways in which graphical representations can be seamlessly integrated with text-based programming environments, whereas existing systems focus on the visual representations themselves. BitPict [45] uses grids of pixels rather than a box-and-line notation. It allows the user to specify pixel-rewriting rules, which are pairs of small input and output bitmaps.

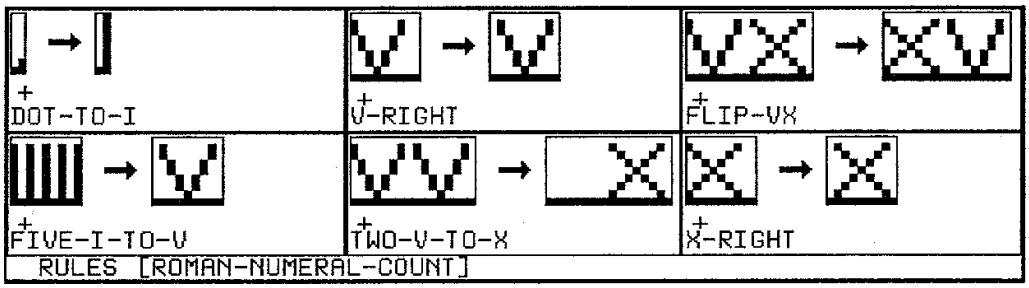


Figure 2.3: An example of visual programming languages going beyond symbolic notation. Pixel-rewriting rules of BitPict [45].

When the user provides a bitmap to the system, it looks for a match between the bitmap and the pixel-rewriting rules; when a match is found, it rewrites the bitmap. With the iterative cycle of pixel rewriting, meaningful operations can be carried out, such as numerical computation, graphical animation, or filling a region inside a closed contour. Dominoes [89] in the Mondrian system [90] are pairs of two thumbnails representing a specific sequence of graphical editing commands. The two thumbnails represent an example before and after applying a sequence of commands. The domino icons can be expanded to a storyboard, which shows the entire sequence of commands. Unlike other visual programming languages, which use static iconic pictures, the Mondrian system automatically constructs a graphical representation of operations from visual examples to represent user-defined functions. The Agentsheets system [131] replaces the box-and-line system with a sheet of agents placed spatially in a two-dimensional grid. In Agentsheets, agents react to user input or information from other agents to change their status. Each agent has a graphical representation of its status, which is visible to the user. Using a domain-specific set of predefined agents, simple simulation applications can be implemented, such as an electric circuit, pieces of water pipes, or an animated ecosystem of animals.

Visual programming environments use visual components to represent the logic of the program or the pixel-based information used in the program. In this dissertation, we investigate the use of photos and videos to represent specific situations or dynamic changes of situations in the real world.

2.2.2 Programming by Example

Programming by example (PbE), also known as programming by demonstration (PbD), is a popular approach to end-user programming. It aims to eliminate the need for explicit coding by inferring the program specification from examples or demonstrations provided by the end-user. It is different to visual programming in that there is no explicit programming activity; however, there are a number of similarities. For instance, both achieve "programming in the user interface", which is a term coined by Dan Halbert [56], and the distinction between the programming environments and the resulting program is unclear. The program is constructed by manipulating the user interface, which typically supports direct manipulation. As a consequence, many PbE systems are also visual programming environments.

While PbE systems are typically designed for end-users, the concept was first realized for programmers. PbE for programmers is more closely related to the work that forms this dissertation in that they both aim to benefit from

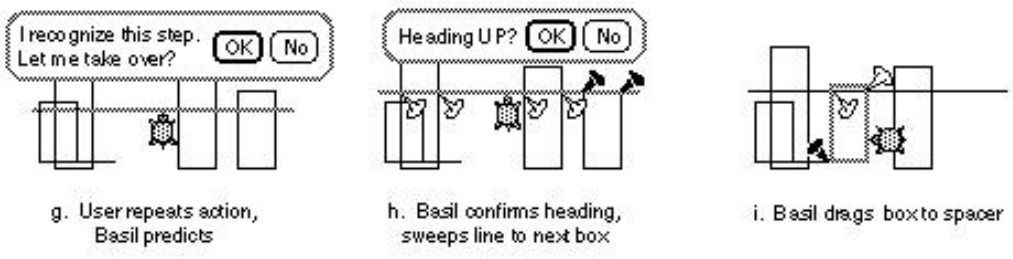


Figure 2.4: An example of programming by example systems. Metamouse observing user's operation and repeating it [104].

concrete examples to build programs. We name this approach "programming with examples", and it is discussed in detail in Section 3.2. The very first PbE system is thought to be Pygmalion [138]. Instead of writing abstract program code, it allows the programmer to start building a program by giving concrete input data. For instance, he can start building a program to compute factorials by giving a specific number, for example, 6. Then, he specifies the procedure to process that number. The system assumes that the user is a programmer and must generalize the program himself. In this case, he is writing a program that handles the number 6, and must anticipate that the program will handle other numbers and specify a conditional expression that tests whether the input is 1. Tinker [91] is a PbE system for the novice Lisp programmer, which allows him to begin writing programs by giving concrete examples of input data. Tinker allows the programmer to provide multiple examples, which are used to infer conditional statements. In this way, he can concentrate on the current example, which is an improvement compared with Pygmalion.

The original idea behind PbE for end-users was an extension of the idea of macro recorders. The macro recorder records the user's operations and replays it so that they do not have to repeat the same operation for many times. PbE goes beyond a macro recorder by generalizing the user's operations. Metamouse [104] shown in Figure 2.4 allows the user of a drawing program to use an "instructible agent" to automate repetitive editing tasks. Instead of text-based programming, the user teaches the agent, named Basil, and when the user tells Basil to record his operations, it may ask him to help to generalize the operation, detect repetitive actions, and predicts what the user wants to do next. Eager [32] applies similar idea to generalize applications such as HyperCard and Claris Resolve (a spreadsheet application). Unlike Metamouse, it does not require the user to explicitly declare the beginning of the operation. It continuously observes the user's operations, detects repetitive actions, and makes predictions. When the user begins to repeats an operation, it automatically executes the operation to the end. Dynamic Macro [103] automatically creates a keyboard macro by detecting repetitive operations in the Emacs text editor. It provides a "PREDICT" button to switch between multiple prediction results and a "REPEAT" button to repeat the most recent repetitive operations. Chimera is a graphical editing application that implements several ideas of PbE to achieve a history-based macro-by-example system. Chimera is capable of graphical search and replace [84], whereby it searches for a specific visual component and replaces all occurrences with another visual component. For instance, the user may replace all of the oak leaves in a picture with maple leaves. This feature is later extended with the idea of constraint-based search and replace operations [85]. Rather than

searching for a specific visual component, it searches for all visual components that satisfy a given set of constraints. The constraints can be defined using a set of parameters, such as the angle and the distance to nearby components. Chimera also allows the user to specify constraints from multiple snapshots [86]. For instance, three snapshots of a Luxo lamp can be given to show how the lamp moves at its joints and how the direction of the light is changed. The lamp can then form an interactive object. Chimera is closely related to Mondrian [90] (see Subsection 2.2.1), in that both provide thumbnails of graphics before and after the operations, representing the history of graphical editing. The difference is that Chimera's history is editable [83]; the user can go back to the specific entry of the history, apply a new operation, and redo all the rest entries in the history.

More recent PbE systems are likely to be integrated into larger systems and used with other user interfaces to achieve domain-specific tasks. Such systems include text input [102], computer-aided design [47], mashups of user interfaces [44], automation of collecting useful information from websites [96] and robot programming [129].

Most of the systems introduced above are heavily dependent on heuristics with domain-specific knowledge; however, there is also a more general approach to PbE that aims to automatically generate programs from high-level specifications without writing source code. Genetic programming [82] generates programs with some randomness, evaluates the outcome using a specific value function, and creates new individual programs based on programs that achieved good scores. Another successful example is program synthesis [53], which has recently been applied to text editing in Excel [58], a popular commercial spreadsheet software package. It uses formal reasoning to synthesize small programs from a set of example input and output, which can process input text to produce edited text.

Most PbE systems aim to eliminate the need for programming. Some initial attempts were designed for programmers and involve programming activity, enabling "Programming with Example." In this dissertation, we propose systems for "Programming with Example"; however, unlike PbE systems, these do not infer the programmer's intent. It is the responsibility of the user to explicitly describe the logic of the program. Examples will be discussed in Section 3.2.

2.3 Text-based IDE Enhancements

As discussed in Section 2.1, standard components of text-based IDEs, such as integrated debuggers, program analysis tools and text editors with syntax-checking features, first appeared in the 1970s, and the development of IDEs has been conservative. Since the birth of personal computers, researchers in the domain of programming languages, software engineering, and HCI, have tended to work separately. However, since the 2000s, there have been many projects among these fields to enhance text-based IDEs.

In this section, we review work related to enhancements of text-based IDEs. Enhancements to text-based IDEs for coding are discussed in Subsection 2.3.1, and to debugging are covered in Subsection 2.3.2. Attempts to bridge the gap between coding and debugging — so-called "Live programming" — are introduced in Subsection 2.3.3. Comparison between support for exploratory programming and one for reliable software development is also discussed. Online collaborations, made feasible through recent advances in the Internet and related technologies, are discussed in Subsection 2.3.4. Text-based IDEs for the development of applications in specific domains are described in Subsection 2.3.5. Finally, we introduce existing work on integrated graphical representations in text-based IDEs in

Subsection 2.3.6, followed by practical references to research on text-based IDEs in Subsection 2.3.7.

2.3.1 Enhancement on Coding Experience

A recent observation of a group of programmers revealed that, although they can benefit from structured editors, tend to favor the capability of character-by-character edits provided by a conventional text editor [76]. Barista [79] is a framework for implementing structured editors, which is highly visual and interactive. The resulting editors are capable of conventional text-editing and code completion; however, also support drag and drop and other alternative views of coding.

Quack [97] is an Eclipse plugin for sloppy keyword programming. It uses keywords to populate possible choices of complete code snippets and allows the programmer to choose the appropriate one. For instance, the programmer can pass the keywords "add line" to direct the code completion and input "lines.add(in.readLine())". Brandt coined the term "opportunistic programming", which is a quick and agile prototyping process. Compared with more formal software engineering practices, opportunistic programming emphasizes speed and ease of development. Such programming processes often involve web searches, copy-and-paste operations on source code, and transforming the code to fit the context. Blueprint [16] shown in Figure 2.5 provides a code completion interface based on example source code retrieved from open-source repositories on the web. When multiple choices are found, the programmer can choose one that best fits their purpose. SnipMatch is a follow-up project [148], which transforms the example source code to fit the context when it is pasted into the editor.

Calcite [113] addresses difficulties in constructing class objects. Classes are not always directly constructed using their constructors, but indirectly with the help of other classes. Calcite inserts such templates when the programmer invokes a code completion on a target variable. The template is generated from a specific format of the API documentation [139] and the help of other users.

Recent research into code completion has provided type-specific user interfaces named "Active Code Completion" [122]. For instance, when the programmer is instantiating a Color object, a color palette interface is shown rather than a text template.

HyperSource [60] and Codetrail [49] bind lines of source code using URLs of

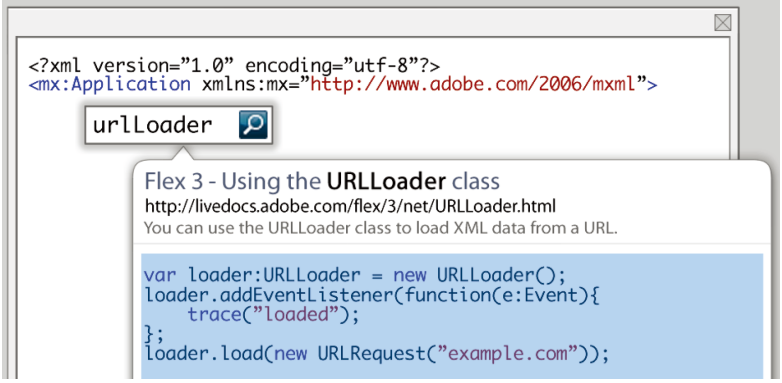


Figure 2.5: An example of enhancement on coding experience. Blueprint showing code completion based on online example code [16].

relevant websites. They are both implemented as a browser extension and an IDE. HyperSource has been implemented for Google Chrome and the Processing IDE, and Codetrail has been implemented for Mozilla Firefox and the Eclipse IDE. Using these combinations, the programmer may review websites that were visited using the source code editor, which provides an alternative to repeatedly looking for the same information on the Internet.

Code Bubbles [15] and Debugger Canvas [35] allow the programmer to navigate source code based on call graphs and show all of the relevant fragments of code at once. While conventional code editors usually fill the entire screen, requiring the programmer switch between multiple files, these tools concurrently show multiple kinds of relevant information in one view, which was shown to provide significant scope for increased productivity.

Chapter 4 describes enhancements of text editors to show inline photographs to make the source code more understandable. Chapter 6 is also related in that it discusses how we can relieve the programmer from writing boilerplate code.

2.3.2 Enhancement on Debugging Experience

Observations of groups developing interactive three-dimensional simulations have revealed that the programmers started each debugging task with a question, such as "Why did..." or "Why didn't..." [78]. Whyline [77, 80] shown in Figure 2.6 is an integrated debugger that was proposed to bridge the gap between static source code and dynamic output. It aims to answer the "Why did" and "Why didn't" questions about the program's output. The programmer chooses a question from automatically generated choices via static and dynamic analyses. The system then answers the question using data mining from information recorded during runtime. Whyline was first designed and implemented for Alice [30], a visual programming environment for interactive character animation of three-dimensional computer graphics. It was ported to Java and supported more professional questions and answers, for more general applications, and a programming interface

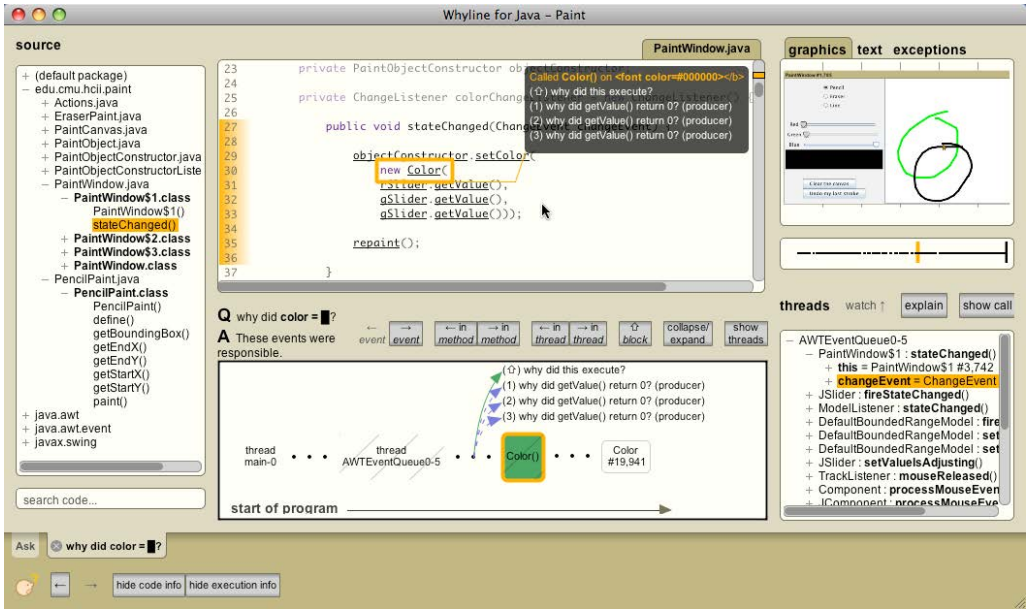


Figure 2.6: An example of enhancement on debugging experience. Whyline for Java [80].

for two-dimensional graphics and event-based GUI programming.

Juxtapose [63] allows the programmer to execute multiple versions of a program at the same time. It is capable of copying the input events, such as mouse and keyboard events, from one version to another at runtime, and the programmer can interact with all of the variants at the same time to compare their behavior. It is also capable of showing the user interface components of the program at runtime to allow the programmer to tweak parameters that were declared as constants. When values are edited in the program, the source code is also edited.

Chapter 5 describes enhancements to the debugger to show video strips representing the history of program execution to improve understanding of the dynamic behavior of a program. Chapter 6 proposes a video player interface to allow control over the execution of the program to aid debugging.

2.3.3 Live Programming

In conventional IDEs, coding, executing and debugging the program are separate activities for the programmer. In traditional Software Engineering (SE) research, much effort has been devoted to achieve reliable software development, typically using formal verification. However, as discussed in Subsection 2.3.1 and 2.3.2, recent advances include enhancements on IDEs which aim to accelerate implementation of programs rather than verifying their soundness. These kinds of enhancements can effectively support exploratory tasks that require quick cycles of coding and debugging, which are called "Exploratory Programming". In HCI research, such process is called prototyping and has been supported by various prototyping toolkits.

Particularly, an attempt to remove the distinctions between coding and debugging is called "Live Programming" and has been studied interdisciplinary among SE and HCI. It aims to allow the programmer to receive continuous feedback from the results of program execution while writing code. "Liveness" in programming was first discussed by Tanimoto in research into a visual programming language named VIVA [141]. He addressed the gap between coding and program execution by eliminating the explicit operation of compilation. Programs developed using VIVA are always executable. The paper discussed four levels of liveness. Maloney discussed directness and liveness of programming of GUIs using their Morphic system [99]. Morphic does not make distinctions between writing code and executing it. It does not make any distinctions between the development environment and the runtime environment. The program is always editable and the program is always running in the Morphic environment.

The term "Live Programming" was coined by Hancock in his 2003 dissertation [57]. He discussed how real-time feedback can aid text-based programming. McDirmid applied the idea to game development [107] and later to text-based programming [108]. ChucK [145] is a live programming environment for creating and playing music. Subtext [38] uses a model view architecture for program editing, whereby the editor shows the immediate representation of the program being edited, which allows the programmer to directly manipulate the program without compilation. Victor demonstrated an instance of his vision for live programming, which placed the text-based editor next to its execution screen, allowing the programmer to dynamically change the source code and see the effect at runtime [143]. Burckhardt proposed a programming model that enables this type of live editing of GUIs [18] in a web-based development environment named TouchDevelop [142].

Our work aims to support exploratory programming. Each of the concrete

projects can be considered as live programming, too. Relation between live programming and each project will be discussed in detail in Subsection 7.2.4.

2.3.4 Online Collaboration

Following the standardization of web technologies, a number of IDEs have been proposed, which benefit from the power of the Internet. Plugins for online version control systems, including CVS, Subversion, Git and Mercurial, are supported in many major IDEs. Blueprint [16] (discussed in Subsection 2.3.1) uses a database of source code collected from such online systems. HelpMeOut [62] is an extension to the Processing IDE shown in Figure 2.7, which stores how runtime exceptions have been solved by programmers in an online repository. It uses this database to propose solutions when a new unhandled exception occurs. Collabode [48] is a web-based IDE that allows programmers to collaborate in real-time, which is an attempt to eliminate the overhead of conventional version control.

Bruch summarized the research effort in the software engineering community as "IDE 2.0" [17]. Such effort includes intelligent code completion, which mines statistical usage information collected from many users to show a prioritized list of completion choices. It has been implemented in IDEs including Eclipse [43] and TouchDevelop [142]. TouchDevelop is a web-based IDE developed using HTML 5, CSS and JavaScript. It is capable of instantly distributing applications via its website. It has some social features, including commenting and evaluating applications.

2.3.5 Domain-specific Support

Following the introduction of the personal computer in the 1980s, there was a growing expectation that GUI applications would appear, which could be used without prior knowledge of programming. However, developing programs with polished GUIs was not straightforward. User interface management systems [23, 127] and user interface builders were introduced. Such systems typically separated the user interface design process from the development of the program, so that the designer can concentrate on improving the appearance of the

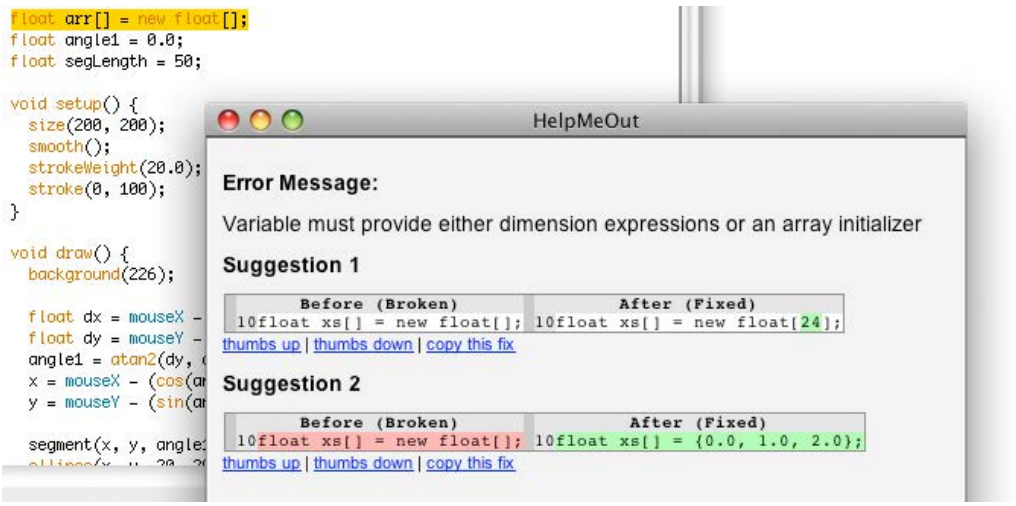


Figure 2.7: An example of online collaboration for IDEs. HelpMeOut proposing a bug fix based on solutions collected from other programmers [62].

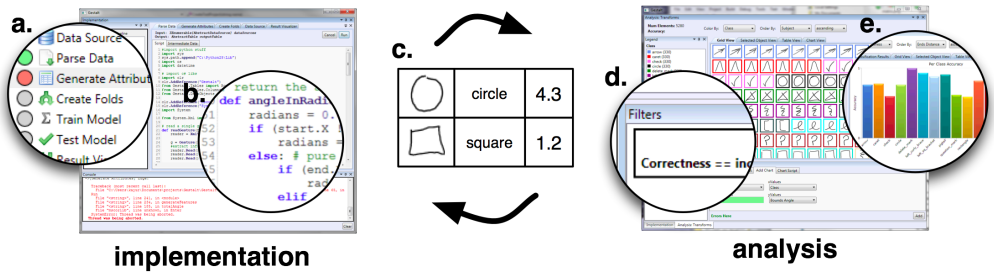


Figure 2.8: An example of domain-specific IDE. Gestalt for machine learning program development [125].

interface. Many of the resulting systems were developed as independent tools outside of IDEs [65, 26]. They produce user interfaces, which are later connected to the applications through event languages written using text-based programming languages. Some of them provided the capability to build the entire GUI application within the development environment [99], which can be considered as a domain-specific IDE for GUI applications. In both cases, however, text-based programming was still required. For the programmer's convenience, user interface builders were integrated into text-based IDEs, such as Visual Basic 1.0 [31], released in 1991. This integration allowed the programmer to alternate seamlessly between user interface design and implementing the functionality. For modern web applications, HTML and CSS, which are declarative languages, are used to construct user interfaces. JavaScript is used to specify a response to the user input and to manipulate the user interface components using the Document Object Model API. Other architectures for GUI applications employ a similar separation of languages for user interface construction and interaction design, including Windows Presentation Foundation (XAML and programming languages supported on .NET Framework), JavaFX (FXML and Java) and Apache Flex (MXML and ActionScript). This separation makes it feasible to develop systems for user interface design. For instance, TouchDevelop [18], which was introduced in Subsection 2.3.3, benefits from this separation, and allows live editing of a GUI without killing the application process. SeeSS visualizes the impact of changes in CSS and the ease debugging using CSS [88].

The variety of programs has continued to evolve, and the pace of change appears to be faster than ever before. This development requires even more support tools; thus, more IDEs optimized for specific application domains. Reflecting the popularity of text-based programming, most are based on textual programming languages, and provide libraries and user interface components, eliminating the need to write boilerplate code. ChucK is designed for creating and playing music [145] on the fly. Processing [4] adopts a simplified dialect of the Java language and is equipped with a set of APIs for information visualization and prototyping media art applications. Arduino [1] is a variant of Processing for programming firmware for microcontrollers and prototyping new physical devices. For the same purpose of creating new physical devices, but with more emphasis on the prototyping process of design, test and analysis, d.tools [61] provides support for the whole of the workflow of a prototyping task. Gestalt [125] shown in Figure 2.8 is designed to support the whole of the workflow of the task, not for physical computing, rather for machine learning. This trend of domain-specific IDEs is not limited to academic research, but is also apparent in commercial products, including MATLAB for mathematical calculation and data visualization.

Our approach has parallels with these domain-specific IDEs; however, focuses on the development of programs with real-world I/O.

2.3.6 Graphical Representations in IDEs

Visual programing environments and text-based IDEs were covered in Subsection 2.2.1 and Section 2.3, respectively, and this subsection covers the integration of these two approaches. Systems have been proposed to represent static data in text-based editors. Heterogeneous visual programming language [39] integrates visual components representing data structures in a text-based programming language, supporting an intuitive understanding of source code. Barista [76] is a framework to include visual components in a structured text-based editor. Sikuli [150] allows inline pictures representing their data.

Other systems have been proposed to capture the dynamic behavior of the program and provide graphical representations to determine why or how the behavior was observed. Dominoes [89] provides a pair of thumbnails to show the input and output of a function. Chimera [83] uses the same notion to allow editing a history of the operations carried out in a given program. ZStep [94] is the first debugger that shows stack traces and visual representations of the data used in the program next to the source code editor. Whyline was originally developed for visual programming of 3DCG programs [77], and uses the timeline view of the program execution along with the recorded screen of the program to answer the questions "Why?" and "Why not?". Later, it was extended to general text-based programming of graphical applications for the Java programming language [80].

There are also systems that are capable of directly manipulating the program components through user interfaces of the IDEs without explicit compilation operation. Many visual programming environments such as VIVA [141] and ConMan [54] fall into this category. Morphic [99] is an IDE based on the Self text-based programming language and at the same time a run-time environment of the GUI programs. Subtext [38] provides a tree view of the program structure, which is editable through its user interface, allowing the programmer to build a program within the user interface. TouchDevelop [18] allows the programmer to edit the user interface of GUI applications without killing their process.

We investigate the same categories of graphical representations of static data and dynamic behavior of the programs in Chapters 4 and 5, as well as graphical editing of such representations to support building the programs in Chapter 6. Our experiments share important aspects with all of these systems: graphical representations are provided to complement text-based programming. The major difference comes from the target applications, and thus in the approach to

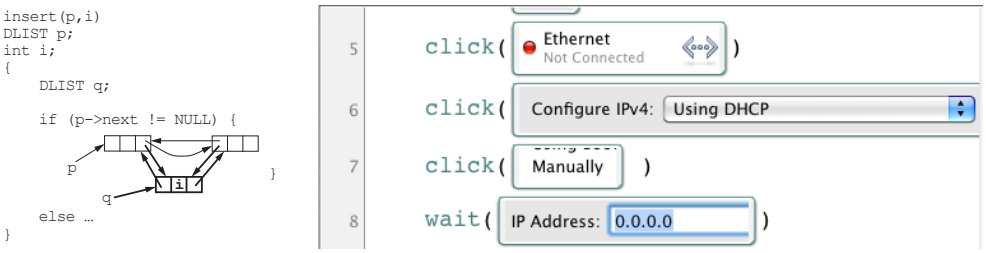


Figure 2.9: Examples of integrated graphical representations in existing IDEs. Left: Heterogeneous visual programming language visualizing data structure [39], Right: Sikuli showing inline images [150].

supporting the programmer. In the following chapters, we propose the use of photos and videos as natural graphical representations of the real world. There are very few IDEs that use photos and videos in this way. One exception is d.tools [61], which uses recorded video synchronized with state transitions in the program under development to analyze the interactions. Comparisons with existing approaches will be discussed in detail in Section 3.3.

2.3.7 Building Blocks

Research into IDEs has accelerated with recent technological shifts. Building blocks for IDEs have been developed for public use, allowing the programmer to create IDEs without building them from scratch. All of our experimental implementations of the IDEs benefit from these evolutions. For instance, Eclipse [42], SharpDevelop [5], and Processing [4] are all text-based IDEs developed as open-source projects. The source code has become more organized as they evolve, allowing programmers and researchers to focus on their contributions and leave the remaining components as provided by the environments. They also accept plugins, which extend the features of the IDE without requiring rewriting of existing components. The core components of the Visual Studio IDE are planned become available as APIs [110], which will allow building new IDEs on top of them. For example, a technique for building structured editors [79] is no longer required to build a text editor with inline embedded images [150, 72], as the Java Runtime Environment provides the JTextPane class for just such a purpose.

All the experimental implementations of IDEs described in this dissertation were developed with help of these publicly available components. Picode was implemented on top of the Processing core components, as discussed in Chapter 4. DejaVu was used to modify the compilation process of the SharpDevelop IDE, as discussed in Chapter 5, and employs its extension framework to add new user interfaces to the IDE. VisionSketch uses an open-source library, which is capable of syntax highlighting and simple code completion, and uses another library to compile Java source code, as discussed in Chapter 6.

2.4 Tool for Programming Physical Interactions

As described by Myers [116], the HCI community has developed many toolkits. When a new kind of user interfaces is proposed, a new toolkit to develop such interfaces soon appears to allow evaluation and standardization of the user interfaces. Recent trends in HCI show an emphasis on physical interaction, which requires new hardware configurations with sensors and actuators. Given the proliferation of such programs with real-world I/O, development tools for programming physical interactions are desired. Papier-Mache [73] is a toolkit for building tangible interfaces consisting of well-defined APIs, software architecture, and graphical tools for development support. Phidgets [51] encapsulates a sensor or actuator as a package with a USB interface and provides software APIs to control the package. Phybots abstracts hardware and provides higher-level task centric APIs. Arduino [1] lowers the threshold for embedded programming, and was employed in our user study to build robots. Context Toolkit [132] provides support for the development of context-aware user interfaces. ActivityDesigner [87] allows interaction designers to specify commands that are executed for several categories of activities. OpenCV [14] provides a set of implementations of computer vision algorithms. Eyepatch [106] provides a user interface to train classifiers of OpenCV, allowing the programmer to prototype applications that

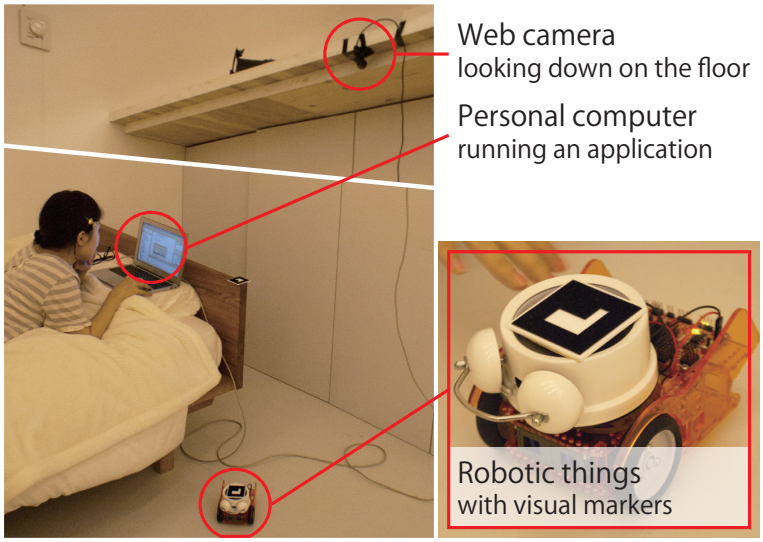


Figure 2.10: An example of a tool for programming physical interactions. Phybots controlling horizontal movement of the mobile robot [71].

benefit from computer vision algorithms. Kinect provides an API to detect human posture from the Kinect device, which uses a color camera and a depth camera. Phybots [71] shown in Figure 2.10 is a toolkit for prototyping mobile robot applications by providing a set of APIs and a corresponding debugging user interface that is shown during runtime.

These toolkits typically support the programming phase of the prototyping process; however, development environments support the whole workflow of the programmer. For instance, d.tools [61] is an IDE for prototyping new physical devices. It supports the design phase with help of existing toolkits, including Arduino, and additionally supports the test and analysis phases by recording and playing videos of the user interaction, synchronized with transition visualization in the program components. The work described in this dissertation takes the same approach as d.tools, in that they both provide support for the entire workflow, and all of our projects are designed for different target applications. They are for the development of programs that run on standard computer platforms, as opposed to new physical devices, and tend to handle real-world data I/O.

2.5 Summary

In this chapter, we discussed research into development environments from a historical perspective, and then categorized these environments into end-user programming environments and text-based programming environments. We also covered tools for programmers that support the development of programs with real-world I/O.

Text-based development environments have long been conservative in their development, and typically employ a text-based code editor and an integrated text-based debugger. Typical visual programming environments are designed for end-users and eliminate the need to manually write code (i.e., text). Existing IDEs developed using these approaches assume that the development and the runtime environments use the same input and output devices. However, a growing number of programs involve real-world I/O, and where development suffers

from a gap between the development environment and the runtime environment. Programming by example systems may eliminate this gap, since they use the same device for the development and execution of the programs. However, most of them are designed for end-users and do not provide precise control over the program logic. Tools for the programmer address this issue by providing partial support on the programmer's workflow.

Given this background, the aims of the work described in this dissertation are to integrate graphical representations into text-based IDEs and benefit from the unique advantages of both the comprehensibility of graphical representations and the expressivity of textual programming. The graphical representations are of example data retrieved from the real world, bridging the gap between the development environment and the runtime environment. The resulting IDEs have both visual and textual components, and provide total support for the programmer's workflow.

Chapter 3

Integrated Graphical Representations

In this chapter we define the scope of this work by comparing it with existing approaches. In Section 3.1 the characteristics of programs that deal with real-world input and output (real-world I/O) are highlighted by comparing them with conventional input and output modalities. In Section 3.2 the workflow of the development of such programs is described, and the model describing the programs is introduced. In Section 3.3 our approach of integrating graphical representations is described, along with the expected advantages, which will be examined in the subsequent chapters.

3.1 Real-world Input and Output

We aim to address difficulties in the development of programs that use real-world I/O. Real-world I/O is data that is inputted or outputted between the real world and the computer. This includes visual data, such as photos and videos, the properties of objects, such as color, shape, and locations, as well as more structured information, such as the posture of humans or robots. We focus on these visual data; however, real-world I/O also includes data describing sound, haptic technology, smell and taste. While processing such information is challenging, it may facilitate a new paradigm of user interfaces, such as organic user interfaces [130].

The mouse, pen, keyboard and display are all physical devices in the real world that function as interfaces between the user and computer; therefore, it appears reasonable to consider that they may also provide real-world I/O. However, we exclude these conventional forms of I/O from our definition. Conventional I/O is designed to map well to the metaphorical concepts used in the computer, such as the WIMP environment and modern graphical user interfaces. Therefore, conventional I/O devices are designed to provide discrete values by nature, such as keystrokes (e.g., a, b, c...), which map to character codes, or mouse movements that correspond to two-dimensional coordinates (i.e., numbers) describing pixels on the display.

Real-world I/O provides data that cannot be represented using symbols or constants. Real-world I/O contains information that exists in the real world regardless of the existence of the computer. All data for real-world I/O is sampled and discretized using physical devices such as image sensors and rotary encoders, and subsequently processed by the computer; however, it has continuous values both temporally and spatially.

One particular paradigm that has significant overlap with real-world I/O is recognition-based interfaces. This includes voice recognition and gesture recognition. These input technologies create input with some ambiguity, which should be

handled as probabilistic events rather than conventional events with discretized data. This difficulty has been addressed by existing work on user interface toolkits [100, 134]. Compared with such a toolkit approach, the work described in this dissertation aims to provide integrated support to the programmer. While such toolkits address difficulties in packaging common features and exploiting useful application programming interfaces (APIs), difficulties in programming using a text-based IDE remain. Instead, in this work, we investigate ways to enhance existing text-based IDEs using a combination of multiple tools for programmers, such as editors, debuggers and APIs. We aim to support the whole workflow of programming.

Our approach is strongly inspired by Patel's work on providing an IDE for machine learning [125]. To create a machine learning application, which may be a recognition-based interface, the programmer must collect many example data and create a classifier by training and testing the classification pipeline. The IDE is specifically designed to support the development of such programs. As with our approach, Patel's IDE does not focus on providing specific features that eliminate the requirement for implementation, rather focuses on providing the necessary development support. The main distinction comes from differences in the application domain: Patel's work focused on machine learning applications and our work focuses on programs with real-world I/O.

The development of programs with conventional I/O uses the same environment for development and execution; however, the development of programs with real-world I/O makes use of different devices for development and execution. This widens the gulf of execution and evaluation for the programmer, making it difficult to develop such programs using conventional text-based IDEs. While toolkits for programmers typically cover partial workflow of the programming activity, IDEs are designed to cover the entire workflow. We aim to cover the whole workflow and bridge this gulf using a text-based IDE that includes graphical representations of real-world data.

3.2 Programming with Example

In this section, we compare the development process of programs using conventional I/O and real-world I/O, and define the latter as "Programming with Example (PwE)." We also introduce existing approaches of PwE and criticize its limitations.

Conventional I/O such as movement of a mouse and clicks of keyboard buttons maps well to the digital world of the computer and can be represented well as symbols or small integers (thus, constants). These standardized information allows event languages to be simple and easy enough to understand for the programmer. As a result, it is not difficult for him to write code from scratch that generates or responds to a specific I/O data. For instance, he can easily write code to test programs with character-based user interfaces by specifying a series of input text. Test cases for GUI applications is a bit more involving, but in the easiest case, he can also specify a series of mouse events. While he can easily write text-based code which includes conventional I/O data, it is just a static representation of the program. It is not easy for him to imagine its dynamic behavior from the static representation. It makes it difficult to choose appropriate APIs that fit his needs. These difficulties are considered as an example of the gulf of execution and evaluation [121]. Existing work has addressed this issue by providing appropriate visualizations to the data and code [94, 92].

Conventional I/O uses the same devices (i.e., mouse, keyboard and display) for

both the development environment (IDEs and other tools for programmers) and the runtime environment. However, real-world I/O uses different I/O devices, such as cameras and actuators. The development environment resides in the traditional GUI paradigm; however, the runtime environment is the real world. Therefore, it is almost impossible for the programmer to write code from scratch that generates or responds to specific I/O data. Real-world I/O data must be retrieved from the real world and stored for later use, including coding and debugging. Unlike conventional I/O data, real-world I/O data cannot necessarily be visualized immediately. It may even be impossible to provide a meaningful visualization because of the lack of a connection between the data and the real-world situation. For instance, sensor values representing angles of robot joints should be visualized as three-dimensional computer graphics of the robot, but this is not possible when the form factor is unknown. Therefore, the programmer typically refers to the data by textual representations, such as filenames or variables. This may be viewed as an additional gulf of execution and evaluation, since the textual reference forces the programmer to imagine the situation in the real world by reading the name of a file or variable.

The programmer can use examples to address these gulfs. In some cases, they may retrieve data from the real world, such as the posture of humans or robots, and save this data as files, to utilize them in programs. In other cases, they may execute the program and log useful data, such as the contents of variables that can be recorded only during runtime, and analyze the log afterwards. We term these types of development workflow "Programming with Example "(PwE). Since existing text-based IDEs are primarily designed for the development of programs using conventional I/O, they typically do not provide support for PwE. Therefore, the programmer is forced to write their own code or develop their own tools to utilize these examples, which is often time-consuming, and may also be difficult.

Please note that PwE has a certain limitation. Since PwE firstly records real-world I/O data, it cannot be applied well to the development of programs whose output affects their input; in other words, programs with real-time feedback loop are out of scope of our approach. For instance, a program for humanoid that processes the sensor data and controls its actuators to keep its balance cannot be developed when either the input or output data is recorded beforehand. Such low-level implementation requires executing the program repeatedly or simulating the entire hardware in software. Instead, PwE is typically well-suited for the development of interactions between human and computers. The focus of this dissertation is on the software aspect of the program development. The development process is considered as a design process of defining a pure function that handles input data and produces output data. The relationship between the input and output data outside the software world is not taken into consideration.

As the name suggests, programming by example (PbE) is a closely related approach. It typically aims to allow end-users to create useful programs. It eliminates explicit coding, and removes the distinction between the development and runtime environments. The system observes demonstrations provided by the user and infers the program using the recorded example data. This is one of the most effective uses of example data; however, most PbE systems provide highly domain-specific environments, which makes it difficult to create generalized programs. Furthermore, it is not easy to specify clear logic simply by providing examples. Programming by example may be effective for end-users; however, it is often not the optimal solution for an experienced programmer. PwE is a hybrid approach of conventional text-based programming and PbE. While it bridges the gap between the development environment and the runtime environment in a

similar manner to PbE, it allows the programmer to describe explicitly the logic of programs using text-based programming.

Although PbE systems are typically designed for end-users, some attempts have been made to design IDEs for programmers that use examples, including Pygmalion [138] and Tinker [91], as described in Subsection 2.2.2. These also utilize examples to support programming. Instead of forcing the programmer to write abstract code from scratch, they allow the programmer to specify examples and aid in building program logic that handles the examples accordingly. Recent work on a programming language named Subtext [37] also emphasizes the benefit of examples. It provides a text-based IDE that allows the programmer to write test cases within the program code. When a test case is added, it is executed immediately, and the execution traces are properly visualized over the program code to support test-driven development. These existing works on support for PwE address issues in the development of general programs: they aim to reduce the requirement for abstract programming. There are a number of parallels with the work described here; however, the primary goal of this work is to address specific issues in development of programs with real-world I/O.

The development of programs that use conventional I/O involves coding and debugging within the development environment; however, the development of programs with real-world I/O cannot be effectively completed within the development environment, as an additional step is required to retrieve real-world I/O data. We define such a development process as "programming with example (PwE)". The example data do not have simple visual representations, which widens the gulf between execution and evaluation. PwE is similar to PbE in that it makes use of the example data; however, whereas PbE targets the end-user and eliminates the explicit requirement for programming, PwE involves conventional text-based programming and takes advantage of the level of control over the logic of the program that this affords.

3.3 Integration of Graphical Representations into IDEs

To address the difficulties of PwE, we investigate ways to benefit from the fact that the data represents situations in the real world. We assign photos and videos to the real-world I/O data and evaluate the usability. In this section, we provide a model of programs that deal with real-world I/O, and describe the potential use of the photos and videos, which will be examined in the following chapters. We also compare our approach with those of existing works.

PwE is an iterative process of data retrieval, coding and debugging. Throughout the process, we identify three potential ways to make use of the example data. First, example data representing situations in the real world can be saved in a file or database and referenced using a textual representation in the source code. This can be thought of in a similar manner to constants in conventional programming. Second, the programmer writes code that deals with the real-world I/O. This process of writing functions may benefit from the example data, which is similar to previous attempts to support PwE [138, 91, 37]. Third, dynamic input to the program during execution can be recorded as time-series data. This functions as example data representing test cases. The data are processed by the functions and produces intermediate results. These dynamic input and intermediate data are assigned to variables.

The three potential ways of making use of example data map well to the three aspects of the program: constants, functions and variables. From this

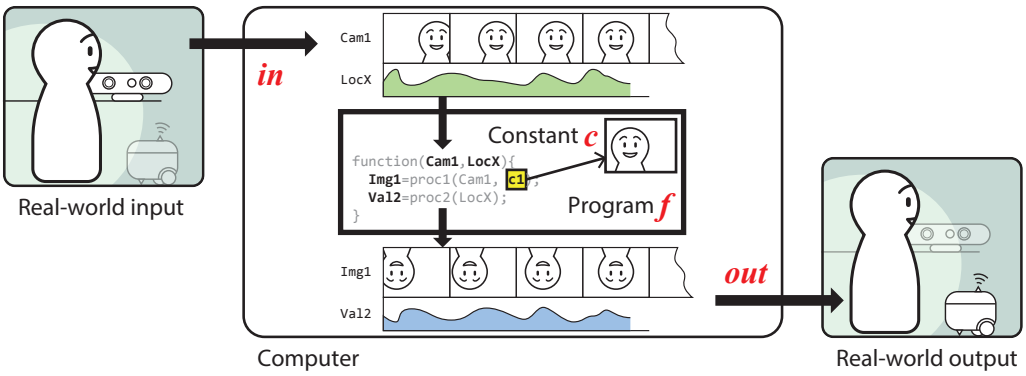


Figure 3.1: Model of the program with real-world input and output.

observation, we provide the following model of the program that deals with the real-world I/O, which is shown graphically in Figure 3.1:

$$out = f(in, c)$$

In this model, c corresponds to constants and f to functions while in and out to variables. As we pointed out in the previous section, existing IDEs can only assign textual representations to these components, which may make it difficult for the programmer to develop the program. Textual representations are not necessarily appropriate for describing real-world data. To address this issue, we propose to use photos and videos as graphical representations of the real-world I/O data in a text-based IDE. Photos are used to capture images describing the real world. Videos are a time-series of photos, and can be used to capture movement in the real world. The core assumption of this work is that such media can be integrated into text-based IDEs and used to support PwE. The goal of this work is to examine this assumption by investigating the use of the graphical representations. We investigate the use of photos and videos as graphical representations of a program in Chapters 4–6. Each attempt will be evaluated by building an experimental IDE that integrates graphical representations to complement textual programming.

Compared with visual programming environments that focus on how to visualize programs, we do not focus on the visualization. Instead, we make use of existing graphical representations of the real world (i.e., photos and videos) and investigate their use in text-based IDEs. The graphical representations used in our work are similar to figures in the history of printing technology [123]. Prior to the invention of letterpress printing, textual data were duplicated using hand-lettered books and figures were duplicated using a printing plate. Letterpress printing was the first technology to enable printing text and figures on the same piece of paper. This combination of textual and graphical data contributed to the growth of the modern science. While textual information provides a precise and abstract description of data, graphical representations provide an example-based description, which may be interpreted rapidly. These two descriptions are complementary. We expect the same to be true of graphical representations of real-world I/O within text-based IDEs. While the text-based editor and debugger allow precise specification and analysis of the program, graphical representations may allow an improved understanding through a more intuitive description of many aspects of a program.

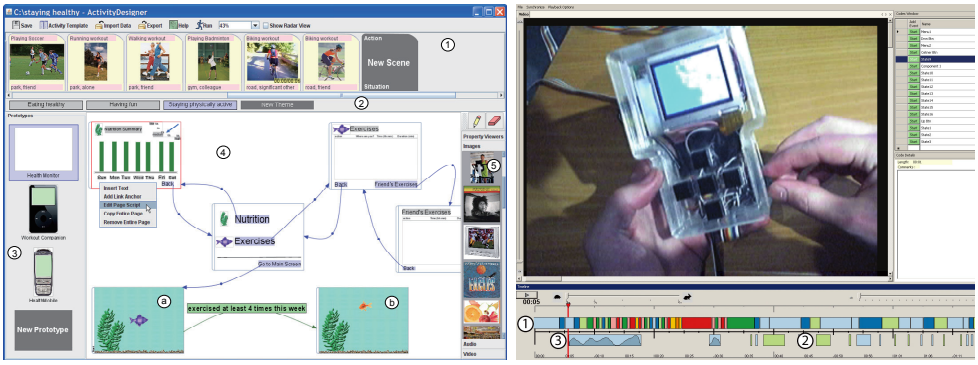


Figure 3.2: Photos and videos in existing development environments. Left: Activity Designer [87], Right: d.tools [61].

There are few examples of tools for programmers to make use of photos and videos. This is because conventional tools do not deal with the real-world I/O. There are, however, a small number of examples that are related to PwE in the real world as shown in Figure 3.2. ActivityDesigner [87] is a prototyping environment for activity-based computing, which allows the programmer to label each activity of the user using photos of the user or related objects. d.tools [61] is an IDE for physical computing, which supports iterative cycles of design, test and analysis of physical devices. It is capable of visualizing state transitions in physical devices synchronized with a video recorded during user testing. These two projects can be considered early uses of photos and videos as graphical representations of constants and variables in the program. In this work we discuss such use in more detail and provide insight into the use of graphical representations.

In summary, we propose to integrate graphical representations into text-based IDEs to support PwE and address difficulties in the development of programs that use real-world I/O. We provide a model of such programs; i.e., $out = f(in, c)$, and investigate the use of graphical representations for each component of the model in the following chapters. We do not focus on developing new forms of visualization, rather use existing natural representations of the real world; i.e., photos and videos.

Chapter 4

Using Photos to Understand Static Data

Current programming environments typically use textual or symbolic representations to specify the behavior of the program. While these representations are appropriate to describe logical processes, they are not necessarily the most helpful way to represent complex data. It may be difficult for the programmer to read textual or symbolic representations of complex data, such as videos or images, and understand what these data mean.

Previous approaches to integration of graphical components into text-based editors [150, 39] have used graphical components to represent such data. While image data can obviously be represented by an image, as was employed by Sikuli [150], and simple tree structures can be easily visualized using techniques such as the Heterogeneous Visual Programming Language [39], it is not trivial to think of an appropriate graphical representation for data from real-world input and output (real-world I/O).

In this chapter, we discuss the use of photos as graphical representations of static situations in the real world. In other words, we propose to use photos instead of text or symbols to represent constants in a text-based IDE. We take human and robot posture data as an example, which are necessary to handle gesture input and to control robots. First, in Section 4.1, we introduce the motivation for choosing posture data processing applications as a representative example. Then in Section 4.2, existing tools for the development of such applications are introduced. In Section 4.3 and 4.4, the Picode IDE is described, which is an experimental implementation of the use of photos to represent constants. In Section 4.5, we evaluate its effectiveness through two user studies.

4.1 Posture Data Processing Applications

Conventional input and output (I/O) devices for computers; i.e., the mouse, keyboard and display, are highly standardized and tightly coupled with the desktop environment on which the IDEs are built. Typical I/O data can be represented by numerical values describing a constant (such as a character code and index number of a pressed button) or a set of numerical values that maps well to the desktop metaphor (such as mouse movement describing two-dimensional desktop coordinates). Conventional I/O data can be readily understood using a textual representation, or in relation to the desktop. Therefore, the development of programs with conventional I/O is facilitated well using conventional IDEs.

In recent years, computers have become ever more pervasive. They are increasingly portable and equipped with a growing variety of I/O devices, which do not have the same type of connection with the conventional desktop environment. These I/O data cannot be represented easily using a conventional

text-based IDE, which complicates development. One example is posture data, which is information of a collection of joints. Human posture information can be retrieved using a posture-detecting device, such as a motion capture system or the Microsoft Kinect. Robot posture data can be retrieved from the robot hardware, or alternatively sent to the devices and used to control their posture.

Posture data processing applications are becoming more popular, exemplified by the growing popularity of hardware such as Microsoft 's Kinect and the LEGO Mindstorms robot. In such applications, posture information is typically stored as an array of numerical values. Unlike conventional I/O data, these values are not easy to understand by the programmer in terms of the posture of the human or robot. They cannot be visualized using an IDE unless the hardware configuration of the subject is known. Even when this configuration is known, as with the case of human posture, visualization may not be straightforward or helpful because of the lack of contextual information. This gap between the real world and the computer is common among all types of real-world I/O data, including sound, haptic technologies, smell and taste. These data represent specific situations in the real world; however, the data are often not sufficient to allow the programmer to lively imagine the situation.

In everyday life, we often take photos and use this medium to tell others about specific situations in the real world. Our assumption here is that they can also be used to represent posture data in text-based IDEs. We bind this posture data with a photo of the subject, and show it as an inline image in the editor of the IDE Picode. This can help in the development of posture data processing applications. The programmer is first asked to take a photo of a human or a robot, which is automatically bound to the posture data. He then drag-and-drop the photo into the code editor, where it is shown as an inline image. Picode provides a built-in API, where the methods take photos as arguments. This allows the user to easily understand when the photo was taken and what the code is supposed to do.

4.2 Related Work

4.2.1 Graphical Representations in Code Editor

A programming language is an interface for the programmer to input procedures into a computer. As with other user interfaces, there have been many attempts to improve its usability. Such attempts include visual programming languages to visualize the control flow of the program, structured editors to prevent syntax errors, and enhancement to code completion that visualizes possible inputs [122]. However, programming languages usually consist of textual or symbolic representations. While these representations are appropriate for precisely describing logical processes, they are not appropriate for representing concrete data such as the posture of a human or a robot. In such a case, the programmer has to list raw numeric values or to maintain a reference to the datasets stored in a file or a database.

To address this issue, Ko and Myers presented a framework called "Barista" for implementing code editors which are capable of showing text and visual representations [79]. This framework enhances comments for an image processing method by including an image that shows a concrete example of what the method does. Martin et al. proposed a heterogeneous visual language [39] to integrate a visual language capable of visualizing a tree structure into a text-based programming language. Yeh et al. presented an IDE named "Sikuli," with which the programmer can take a screenshot of a GUI element and paste the image

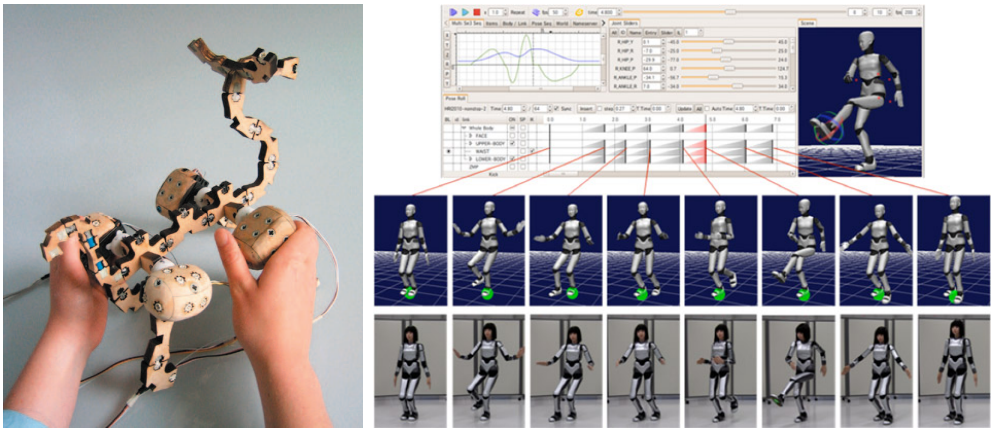


Figure 4.1: Programming motion by physically handling the robot [129] and by editing timeline [118].

into a text editor [150]. In Sikuli, the image serves as an argument of the API functions. Our goal is to apply a similar idea to facilitate the programming of applications that handle human and robot postures.

4.2.2 Tools for Posture Data Processing

To process human posture information, some toolkits and libraries have been proposed. They can typically recognize preset poses and gestures. When the programmer wants to recognize her own poses and gestures, however, she has to record the examples outside the development environment. On the other hand, our development environment is designed to support the entire prototyping process of application development. It fully integrates the recording phase, and the programmer can follow the workflow without distraction. Attempts to support a general workflow of domain-specific applications have already been made for many domains including physical computing [61], machine learning [125] and interactive camera-based programs [70] which will be introduced in Chapter 5.

There is a long history of developing robot applications that deal with robot posture. As shown in Figure 4.1, typical approaches include Programming by Example (PbE) [129, 24], timeline-based editors to help designers defining transitions from one posture to another [118], and general development environments for textual or visual programming languages . Most of the PbE systems focus on reproducing observed human actions, and the editors focus on creating and editing actions. They both tend to have limited support for handling user input. Conversely, general development environments are more flexible in terms of input handling, but do not display posture data in an informative way. Our objective is to design a hybrid environment, by taking advantages of these approaches.

4.3 Picode IDE

Our prototype implementation consists of three main components (Figure 4.2): a code editor, the pose library, and a preview window. First, the user 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, she drag-and-drops the photo from the pose library into the code editor, where the photo is displayed inline, as shown in Figure 4.3. Then, she can run the application and distribute

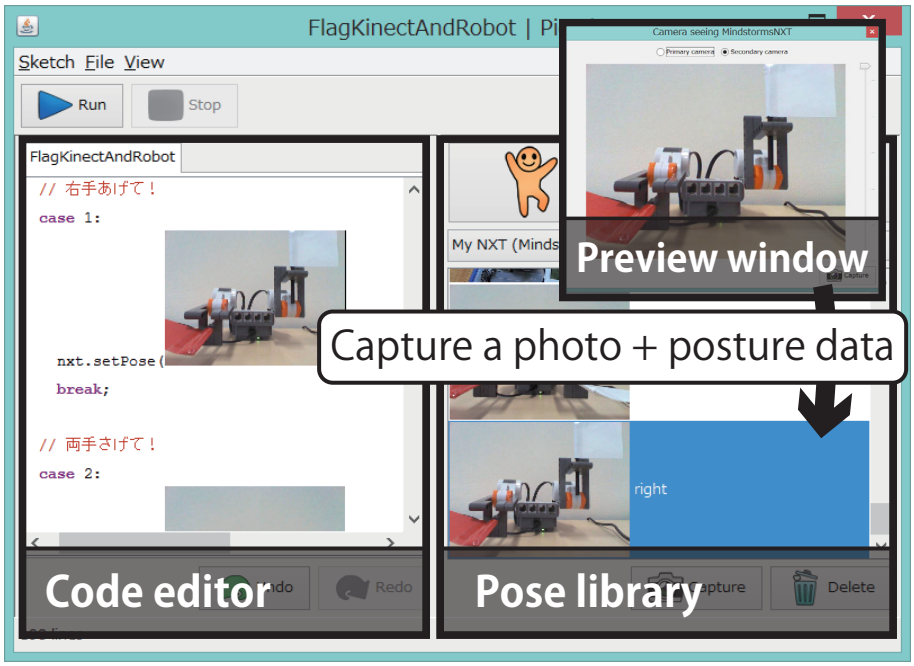


Figure 4.2: Overview of Picode IDE.

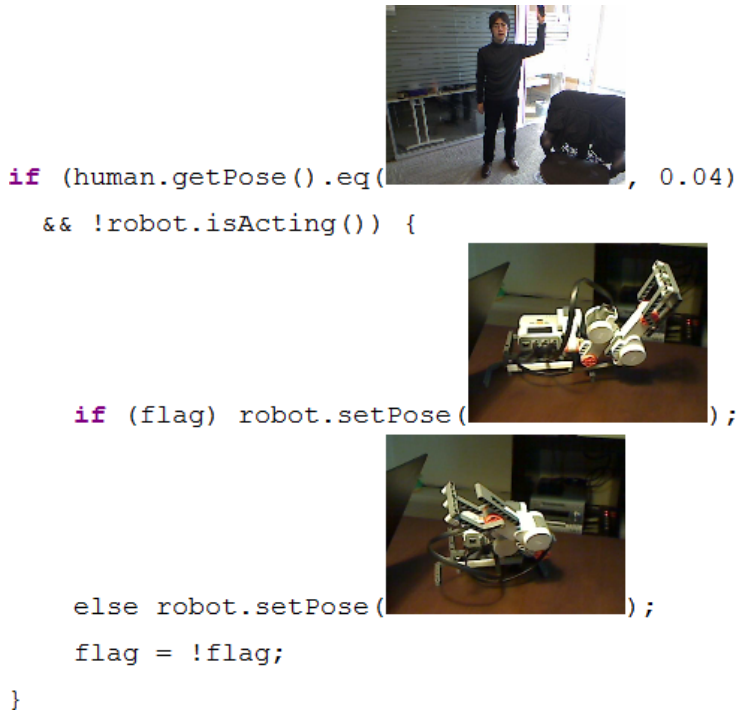


Figure 4.3: Example code that makes the robot swing its hand when the user raises his hand.

the source code bundled with the referenced datasets so that others can run the same application within our development environment.

4.3.1 Taking Photos

To start taking photos, the user clicks the "+" button in the pose library interface and opens the preview window in which the photo preview and posture status are displayed in real time. She can choose the input source of the posture data from Kinect (human) or Mindstorms NXT [52] (robots) devices. While only one Kinect device can be connected at a time and is automatically detected, one or more Mindstorms NXT devices can be used by entering their Bluetooth addresses. Photos are usually taken from the RGB stream of a Kinect device, but a web camera can be used as an alternative source. While the preview window is displayed, clicking the "Capture" button triggers the system to take a photo and capture the corresponding posture data. Each captured dataset is automatically named, e.g., "New pose (1)," and stored in the pose library. It can be manually renamed but must be unique. Saying the word "capture" works when the user wants to capture a human posture and cannot click the button because standing in front of the Kinect device. When capturing a robot posture, a torque is applied to each servo motor on a joint to fix its angle. When the user tries to change its angle, however, the torque is set off so that she can move the joint freely. Therefore, the user can set the robot posture by changing joint angles individually. Additionally, she can load an existing posture by right-clicking its photo in the library. This allows the user to easily create a new posture from the existing ones. These interactions for capturing a robot's posture are inspired by the actuated physical puppet [151].

4.3.2 Coding with Photos

The programmer can write code in a programming language that is an extension of Processing [4], with a built-in photo-based API whose methods take photos as arguments. She can drag-and-drop photos from the pose library to the code editor, directly into argument bodies of the methods. Usage examples of currently supported API are shown in Figure 4.4. A human and robot are represented by Human and Robot classes, whose instance handles communication with the hardware devices. Note that the Human instance is capable of sensing but not controlling posture while the Robot instance is capable of both.

4.3.3 Running Program

The programmer can compile and run the program by clicking the "Run" button in the main window. After iterative cycles of development, a ZIP archive consisting of source code, referenced photos, and posture data can be made so that others can run the same application.

4.4 Implementation

Picode aims to enhance text-based programming rather than replace it with something different. Therefore, it is reasonable to make use of existing implementations of IDEs. However, it is not straightforward to augment existing implementations and achieve a fluid programming experience. In this section,

```
Pose pose = human.getPose();
if (pose.eq(  )) { /* do sth */ }
```

Compare *Pose* with specified error allowance [0-1].

```
robot.setPose(  );
```

Set current *Pose*.

```
Action a = robot.action();
a = a.pose(  ).wait(100).pose(  );
a.play();
```

Play series of pose changes by *Action* definition.

Figure 4.4: Usage examples of photo-based API.

we first provide the overview of the implementation including system architecture and hardware configuration. Then, we describe how we implemented new components and integrated them into an existing environment.

4.4.1 Overview

Picode was built on top of the Processing core components, including the compiler and libraries. Processing is an open-source IDE, which was originally designed for data visualization applications and media art. Its ease of use was welcomed by novice programmers, and it is now popular for educational purposes. We aim to maintain this ease of use; however, we redesigned the entire user interface to make it match the workflow described in the previous section. Picode runs on a Windows computer. A Microsoft Kinect device is connected to the computer through a USB 2.0 port. One or more LEGO Mindstorms robots are connected to the computer through either USB 2.0 ports or Bluetooth wireless connections. By default, the Kinect device has a built-in color camera and is used for capturing photos of both human and robots. Optionally, the user can connect another camera to capture photos of the robots. It is useful when the user wants to fix the angle of view of the Kinect camera. Otherwise, the user often needs to move the Kinect device whenever he wants to switch subjects to be captured.

The main window of Picode is shown in Figure 4.2. It contains the following components:

- The menu bar for various operations.
- The tool bar to control the execution of the program.
- The text-based editor with inline photos.

- The pose library showing a list of stored posture data, with features to edit this list.
- The status bar showing the current status of the IDE.

The programmer may open the capture window to capture new postures by clicking the button in the pose library. Both the pose library and the capture window have two modes, one for listing and capturing human posture data and one for robot posture data; however, both modes appear almost identical, and the programmer can seamlessly switch between the two.

In addition to the user interface, we maintain many things that are compatible with the original Processing IDE. Rather than replacing them with new ones, we augmented them to enhance existing features or add new features. We adopt the Processing programming language, which is based on Java but uses a simplified syntax. We extend it include the capability to accept photos as arguments to methods. We retain the existing set of standard libraries unchanged, which provide useful APIs for graphical applications, and add a new library that provides an API for posture data processing, as described in Subsection 4.4.5. This is used not only by the applications but also by the IDE itself. Supported features include retrieving posture data, comparing these with data retrieved previously, and commanding a robot to hold specific poses by providing a set of posture data.

Several small changes were made in the compilation and execution process of the application. We modified the compilation process so that applications that are developed on the Picode IDE are automatically linked to the new library. We also modified the execution process so that the development environment disconnects from a Kinect device, as well as with robots, when the program starts, and reconnects to them when it shuts down.

4.4.2 Capture Window

When the programmer clicks the button with a camera icon and the text label "Capture", the capture window opens to allow all tasks related to taking photos with the new posture data. There are two modes, each of which is dedicated to retrieving human posture data using a Kinect device, and to retrieving robot posture data using a USB or Bluetooth connection to the robot. The initial mode is determined by which pose library is shown in the main window, and the programmer can change this by switching to another pose library without closing the capture window.

In the mode for capturing human posture data, the capture window uses a Kinect device to obtain a photo and retrieve skeletal data. When the Kinect device successfully retrieves three-dimensional positional data of all joints, the capture window shows a skeleton over the video stream taken using the RGB camera. Otherwise, it shows only video and does not allow the programmer to take photos. In this way, the IDE ensures that every photo is associated with a complete set of posture data. This restriction makes it possible to compare any pair of photos without error. In addition to the three-dimensional positions of the joints, two-dimensional positional data are also recorded so that the skeletal information can be overlaid on the photo.

Directly below the video stream is a button with a camera icon and the text label "Capture." It appears identical to that in the main window for opening the capture window; however, the button in the capture window triggers capture of a photo plus posture data. When the user wants to take a photo to record posture, it is typically not convenient to do this using a wired mouse. Even

with a wireless mouse, it may not be straightforward to form the desired pose and click the mouse button simultaneously. To address this, we make use of the voice recognition features provided by the Kinect SDK. The Kinect device has a microphone array and can achieve stable recognition of simple words. We chose to use the word "Capture."

In addition to the video stream and capture buttons, the window has a slider interface for changing the elevation angle of the Kinect device in the range +/- 27. While the programmer may freely move the tick mark on the slider, it may take some time for the device to reach the specified angle. Frequent changes of elevation angle may cause wear of the device and the Kinect SDK advises the programmer to avoid this, and that the motor should not be controlled more than once per second. In our implementation, a change in position of the tick mark triggers an event listener, which checks if there is an ongoing task for motor control. If there is no ongoing task, it immediately commands the motor to move to the desired angle. If there is an ongoing task, it checks for a pending request for motor control, and if a pending request is found, the desired position is updated. Otherwise, a request for motor control is created and executed after one second.

In the mode used for capturing robot posture data, the capture window adopts a similar layout as the mode for human posture, but with two small differences. First, while the photo for human posture must be taken using the Kinect device, the photo for robot posture may be taken with any camera (including that of the Kinect device or a webcam). A radio button for switching the camera is shown above the video stream. When the webcam is selected, the slider for changing the camera angle is hidden. The current implementation requires the webcam to be accessible via the DirectShow API of Windows, which is part of Windows Platform SDK. Second, checkboxes to control how each joint should behave are provided. When a checkbox is selected, a strong torque is applied to the corresponding joint to maintain its current angle. Otherwise, weak torque is applied and the joint angle is continuously monitored by the system. When there is a significant change in joint angle, the system assumes this is the result of the programmer's operation, and the applied torque to zero so that the programmer can easily move it to the desired angle. While torque is disabled, the system continues to monitor the angle. When the angle becomes stable, the system assumes that the programmer is satisfied with the current angle and begins to apply torque again.

The capture window interface described in this subsection provides the programmer with a sense of what the system is currently watching. Information presented in the interface is expected to be useful not only for the programmer, but also for the end-user of the resulting applications. The user may also wish to obtain feedback on the current posture recognition status of the system. For production-ready games and other applications, the programmer designs and implements a user interface for such feedback, which is integrated into the applications. However, during the prototyping process, the programmer may often take the role of a test user, and so wants a simple method of obtaining feedback while the program is running. To aid with this, we provide an API to show the same window within their application. The window hosted by the development environment is automatically closed when the application is launched, so that the two windows (one hosted by the IDE and the other by the application) cannot be displayed simultaneously.

Human posture data and the corresponding photos are retrieved using the same API provided for the programmer. We will describe the implementation of this in Subsection 4.4.5.

4.4.3 Pose Library for Managing Posture Data

The pose library manages posture data and the corresponding photos. The library has two modes, as does the capture window: one that shows a list of human postures and one that shows a list of robot postures. They can be switched by clicking the buttons labeled "Human" and "Robot", which can be found at the top of the library interface.

When the programmer obtains a new set of posture data or a photo using the capture window, it is automatically added to the library for use later. They are saved separately as a text file and a JPEG file in the same directory, with unique names. Existing files are loaded during the launching process of the IDE. The new posture is initially named with a unique number (e.g., *New pose (1).txt* and *New pose (1).jpg*), but can be renamed later by the programmer (e.g., *Hand up.txt* and *Hand up.jpg*). To do so, he double-clicks the entry in the pose library. The programmer can also rename it from the menu shown by right-clicking the entry. This menu provides operations including deleting the posture and opening a file manager (i.e., Explorer on Windows or Finder on Mac OS X) to display the files. In the mode for robot posture, it also shows a menu to apply the posture to the robot. This is useful for checking the posture without launching the application.

The text file representing the posture data starts with its corresponding Pose class name followed by class-specific data, which is typically a set of raw numerical values. We could have combined the posture data and photo and saved them as one binary file; however, we chose to use a human-readable file format. This makes it easier to explain to the programmer and the user how Picode manages the posture data and the corresponding photos. This is particularly beneficial for educational purposes.

4.4.4 Editor with Inline Photos

The code editor was implemented using the Model-View architecture, in which the model is the source code in string format and the view is its GUI representation. Each photo has a string representation, which is a call to the static method *Pose.load*(*key*), where *key* is a unique name of the corresponding posture data.

When the photo is dropped into the code editor, the view computes which part of the abstract syntax tree is rendered at this location. If the corresponding element of the abstract syntax tree is not a call to the *Pose.load* method, the call to the method with the key representing the corresponding posture data is inserted into the source code. Otherwise, when the photo is dropped onto an existing photo, the existing string calling *Pose.load* with the existing key is replaced with a new string referring to the new posture data. Each change in the source code causes the language parser to build an abstract syntax tree from the entire source code. The parser is generated using the ANTLR toolkit[1] with the grammar file of the Processing programming language. If there are no parsing errors, the view is updated. It applies syntax highlighting and replaces every call to the static *Pose.load* method with the corresponding photos. When the corresponding photo or posture data are missing, it simply retains the string representation with an error message shown in the status bar. Parser errors do not affect the view, leaving existing syntax highlighting and photos unchanged.

Rendering of the editor, which is capable of syntax highlighting and inline photos, is implemented using *JTextPane* class, which is provided as part of the Swing API (the default GUI toolkit for modern Java applications). By default,

[1] ANTLR (ANother Tool for Language Recognition). `http://www.antlr.org`

JTextPane renders the document as plain text without any decoration, but can be decorated using an API, as with HTML, where mark-up tags are used with plain text. The only difference is that, while HTML allows insertion of images without specifying their string representation, the *JTextPane* document model always requires a textual representation of the images. This does not cause any issues, however, since a unique textual representation (i.e., the *Pose.load* call) is always available for each photo.

The source code is saved as a plain text file, which is compatible with the Processing Sketch file with the file extension pde. There is a chance that, when the programmer wants to show the source code with inline photos to others, simply sending the plain text file may not work. We therefore provide two ways to export the source code. One is to export the source code together with the set of photos and corresponding posture data as a ZIP archive. In this way, the programmer can pass what he/she has created to other programmers, who can open it using the Picode IDE. This is achieved by exporting the source code in HTML file format together with a directory containing the photos and posture data. This can be opened using any modern web browser, and each inline photo is linked to the plain text posture data. In this way, the programmer can communicate to others (not only programmers but also end-users) how the program was created.

4.4.5 API for Both IDE and Applications

Maintaining consistency between the IDE and the applications that are developed using that IDE is important. For instance, files generated using the Picode IDE should be correctly loaded by the applications developed using the IDE. This sounds obvious; however, this is not always the case when the IDE is under active development. When the file format must be changed and if the source code for handling the file is not shared between the IDE and the library for applications, the two pieces of code must be modified. To prevent the requirement for this troublesome code modification, we share part of the codebase between the IDE and applications. Each time the IDE is compiled, the shared codebase is compiled and archived as a Java library (using a single jar file). It is then linked to from both the IDE and applications. In this way, the files generated by the IDE can be correctly loaded by the applications and vice versa.

Each posture dataset represented by a photo is instantiated as a *Pose* class instance. A *Pose* class is currently extended using *KinectHumanPose* and *MindstormsNXTPose* classes to support platform-dependent implementations, and can be further extended to support more types of robots, such as humanoids, or more ways to detect poses, such as using a motion capture system. The equality test between Pose instances always returns false if their types are different. When their types are identical, the system calculates the Euclidian distance between the vectors consisting of the absolute difference between joint angles (e.g., the absolute difference in the elbow angle or knee angle) and normalizes this to the largest possible range of joint angle so that the data are between 0 and 1. The equality test returns true if the difference is within a specified threshold, otherwise it returns false. If a threshold is not specified, the default values are 0.05 for human postures and 0.3 for robot postures.

To communicate with hardware devices, including the Kinect device, Mindstorms NXT robot or webcam, the shared library creates one thread per device, which accepts requests to control the device and dispatches them at a suitable frequency, depending on the requirements of the device. This ensures that there

is no more than one request at any one time, and prevents fatal crashes of the software or unstable behavior of the hardware. When there are duplicate requests (for example, two different requests to change the angle of rotation of the same motor), it executes only the most recent request. These details of hardware communication occur in the background so that the mechanism is hidden from the programmer, providing them with a thread-safe API. All threads can be killed manually by calling a specific API and are automatically killed when the main program exits. This implementation is based on a well-tested library for prototyping robot applications called Phybots [71].

To retrieve posture data from the Kinect device, we use the Kinect for Windows SDK. However, it only provides APIs for the C++ and C# programming languages. To support Mac OS X, we implemented a standalone non-GUI program (i.e., server) that runs on a Windows machine. This communicates with clients including the IDE and applications using a simple protocol via TCP/IP connections. A Windows machine is required to connect to the Kinect device; however, the IDE and applications may be executed on computers running either Windows or Mac OS X. The server should be running when the IDE is launched unless the IDE and the server are on the same Windows machine, in which case the IDE is capable of automatically launching the server as a child process when required. When the client requests a snapshot of a video stream (i.e., a photo) and a set of posture data, the server first sends photo data in JPEG format and then sends a set of posture data if it has been captured by the Kinect camera (if information on any joints is missing, it sends only the photo). The client may also request a change in the elevation angle via the same connection. A *KinectHumanPose* instance in the IDE or applications represents the posture data and photo that have been received from the server. The posture data consist of information on 26 joints of the human body, each of which includes three-dimensional positions relative to the camera in units of centimeters, and two-dimensional position in the photo in pixel units. When the data is saved in the text file, these data are represented using 26 lines, each of which contains 5 numerical values (3 for the three-dimensional position and the remaining 2 for the two-dimensional position on the image).

To retrieve posture data from a Mindstorms NXT robot, we use the Mindstorms NXT SDK [2] for wired USB connections on Windows or the BlueCove Java library [3] for wireless Bluetooth connections on Windows or Mac OS. The protocol for communication is shared between these two implementations, except for the additional header required for Bluetooth communication, which provides the data length in one command. There are several Bluetooth specifications, and the one used by Mindstorms NXT is class —— v. 2.0. The USB connection provides faster data transfer with less latency than the Bluetooth connection. We find that the Bluetooth connection is typically sufficient for our applications; however, we have experienced difficulties due to interference between multiple Bluetooth connections that were established simultaneously in the same location. If more than five connections are active simultaneously, Bluetooth may become unstable. For this reason, in the workshop described in Section 4.5, USB connections are required.

To control the posture of the Mindstorms NXT robot, the same connection that was used to retrieve the posture data was used. We had planned to use

[2]LEGO.com Downloads - NXT Software Developer Kit. `http://www.lego.com/en-us/mindstorms/downloads/nxt/nxt-sdk/`

[3]BlueCove JSR-82 project. `http://bluecove.org/`

the command provided by the default NXT firmware to control the movement of the actuators; however, this was not sufficiently precise. Although it allows the programmer to specify the desired position of the robot via the angle of rotation of the actuator, in most cases, it undershoots or overshoots by more than 360 degrees. Therefore, it is difficult to reproduce the recorded posture using the default method. To address this issue, we used the open-source MotorControl program developed by RWTH Aachen University[4]. Mindstorms NXT allows the programmer to use custom programs that runs on the default firmware, and MotorControl is one such program, which aims to provide more precise control over the actuators. The program functions as a server that runs on the NXT brick and communicates with clients using its original protocol built on top of the default protocol. A client program was implemented, which benefited from the improved precision motor control, and was to +/- 1 in most cases.

4.5 User Study

In this section, we report two different user studies. First preliminary study was conducted to check if Picode is comprehensible and welcomed by a programmer and to investigate how Picode is used by a programmer and non-programmer. Given the successful results from the first study, second study was conducted to focus on non-programmers and to further investigate the use of photos.

4.5.1 Preliminary Study of Pair-Programming

We asked two test users to try our development environment together for about three hours. The goal was to verify two hypotheses on the benefit of embedding photos in the source code. The first hypothesis was that photos contain rich contextual information other than mere posture information, which helps the programmer recall the situation. The other 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. While one test user knew Processing and was familiar with basic programming concepts, the other did not know about programming except for basic HTML coding. We had them work together since we expected our environment to establish a new relationship between programmers and non-programmers (users). First, we thoroughly explained the workflow of our programming environment with the example code for an hour. Then, we asked them to make their own program for the remaining two hours.

After two hours of free use, the participants could write a program that uses gesture input to control robot posture. The robot basically tried to mimic the user input, e.g., when the user waved her hand, the robot waved its hand back. By putting the robot in front of the keyboard, the participants also had it operate the PC with its mechanical hand, which reminded us of mechanical hijacking [34].

In this preliminary experiment, we confirmed that programmers performed efficient posture information processing programming through Picode IDE. Furthermore, it was shown that (1) even without prior knowledge of programming, users were able to infer and comprehend the processing content of the surrounding source code from photos embedded in the code. We also found that (2) using photos facilitates communication between programmers and non-programmers.

[4]MotorControl - RWTH - Mindstorms NXT Toolbox. `http://www.mindstorms.rwth-aachen.de/trac/wiki/MotorControl`

4.5.2 Workshop for Non-programmers

We conducted a user study on beginners who had an interest in programming but no prior knowledge by having them experience programming using Picode and then investigating results through observation, by collecting their work, and by performing surveys. The objectives of this study were to discover (1) how photos assist with the inference and comprehension of processing content and (2) what sort of opportunities facilitate communication. Two workshops were held at Japan's National Museum of Emerging Science and Innovation ("Miraikan"). We gathered people interested in programming by posting a general outline of the workshop on the museum club bulletin. In addition, we aimed to observe beginners and communication in pairs by having groups of two or more, which included a child, participate in the workshop. These groups consisted of an elementary-school-aged child with a parent or two junior-high-school-aged or older children. The contents of both workshops were the same, with the author and museum staff acting as teachers while volunteer staff offered support to participants in completing their tasks.

1. Overall explanation and ice-breaking (10 minutes)
2. Revising code for a flag-raising game (Kinect) (25 minutes)
3. Revising code for a flag-raising game (Kinect, robot) (35 minutes)
4. Implementing a ball-rolling game (50 minutes)
5. Ball-rolling game tournament (30 minutes)
6. Critique and survey, collecting work

In step 1, we showed a demonstration video of a robot playing a xylophone with Picode to give participants a basic idea of posture information processing. Additionally, to create an environment that facilitates communication in the workshop space, we performed ice-breaking activities to reduce tension. Specifically, they performed a simple game unrelated to the content of the workshop (forming groups based on shared traits). This activity was designed to foster more effective observation of the effects of Picode on communication between participants.

In step 2, participants were asked to rewrite the source code for a flag-raising game as shown in Figure 4.5. In this game, an image of a character raising and lowering both hands is shown in a window on the computer screen. The user takes up the same pose in front of the Kinect camera in order to score points. The source code contains four "if statements" to check for different combinations of raised and lowered hands. However, the source code given to the participants was inserted with photos of people in random positions for each "if statement," and the game would not determine conditions correctly as intended. Participants were asked to use Picode and take photos of the correct postures, revise the game actions by changing the original photos, and make the program work correctly. This allowed participants to become familiar with the basic operations of Picode, and inputting posture information into the computer to be processed allowed participants to see for themselves that they could develop a program using their body.

In step 3, participants were asked to rewrite the source code for a new flag-raising game in which an actual robot provided the instructions in place of a

Figure 4.5: Screenshot of the example application.

character on screen. Unlike the program in step 2, which only used photos displaying human positions in the source code, this program also used photos of the robot's position. For the game to operate correctly, four different photos should be used to display the four combinations of robot hand positions, but in the source code given to participants, all four photos show the robot with both hands lowered. Participants took new photos of the robot with flags in the right hand, left hand, and with both hands raised, placed these photos into suitable locations in the source code, and checked that the game could be played. This allowed participants to see for themselves that by outputting position data from the computer to the robot, they could develop programs in which output was not only limited to images on the screen but also to the real world.

In step 4, using the information learned thus far, participants were asked to edit source code that changed the robot position to reflect a specific pose taken by a person to create a program in which the robot rolled a ball when the human took a pose. By having participants combine the robot's arms in free shapes, they were able to make it roll the ball with their preferred mechanism. To prevent participants from concentrating only on building the arms, awards were offered in the following tournament. These include awards for the fastest rolling ball, the most creative program, hard work, and one special judge's award. After being told about these awards, participants were asked to perform the task.

In step 5, we held a "ball rolling tournament" in which each group used the programs written in step 4 to roll a ball twice, measuring and recording the longer distance of the two attempts. Finally, awards were given out, and each

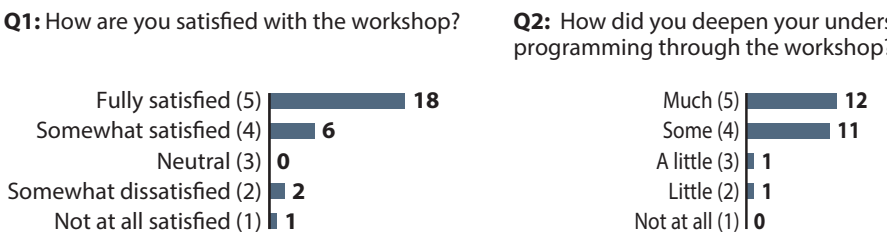


Figure 4.6: Quantitative results of the user study.

group was asked to turn in a completed survey and all data they generated. This concluded the workshop. Collected data consisted of photos taken with Picode, corresponding position information, and source code that included the photos.

Comments received on the surveys backed up the quantitative answers. These included the following statements: "Being able to program with photos is amazing," "It was easy to understand how processing worked with photos," and "This makes me imagine a future in which people can create a variety of programs without any technical knowledge." Participants gave high marks to the experience of programming with photos. As for difficulties in completing the workshop assignment, participants pointed not to adding photos, but rather taking poses correctly with the Kinect to include information for all joint angles.

Three groups (6 members in total) took part in the first workshop, and 10 groups (22 members in total) took part in the second. The low number of participants in the first workshop, which was held on a weekday, allowed us to observe participants closely, while the second workshop produced a wide variety of resultant work. Four elementary school students in the workshop had experience programming using the LEGO Mindstorms NXT environment, but no participants had any experience with text-based programming. All participants managed to complete the assignment, and results in the ball-rolling tournament were recorded for each team. In addition, some participants went beyond the original assignment to change photos and made changes to the game rules by editing the source code text. This is an example of how photos can form a starting point for people unfamiliar with programming to guess the meaning of surrounding text and make changes. Participants were also very happy to receive print outs of the source code that included their photos. Seeing the photos in the source code appeared to give participants a sense of ownership of their work.

Figure 4.6 shows valid responses received from participants in the post-activity survey. These responses show that participants reacted positively to the workshop overall, and that Picode functioned effectively as motivation to program. We found that the male elementary student who answered that he was unsatisfied with the first workshop had been paired with another boy he had never met, and in the free comments he noted that he wanted to do the activity by himself. This suggests that their group work did not go well. The female elementary school student and two adult males who answered that they were somewhat unsatisfied with the second workshop complained of problems with the software and devices. Thus, we found that each of the negative responses involved technical limitations unrelated to the photo-based programming.

4.6 Discussion

In this section, we discuss in detail the benefits and limitations of adding photos that express posture information to the source code in light of the user study results.

4.6.1 The Popularization of Source Code

Differences between individual persons and bodies are substantial in programming that deals with posture information. For example, a program tuned for one's own body often does not work correctly for another user. By having participants in this workshop take their own photos and use them to replace the existing photos, users completed the work of customizing the program for themselves.

Picode allows users to easily change the behavior of a program by editing a portion of the source code, even if the user does not fully understand the workings of the program. With help of photos, we have realized heretofore unseen casual programming. According to the participants, taking photos and rewriting code gave them a sense of having conquered the program, controlling its behavior for themselves. This allowed source code, something generally only open to programmers, to be open to even those with no prior knowledge of programming.

In addition, for eager elementary students who wanted to go beyond changing photos, the photos served as a starting point to ask workshop staffs about the functions of surrounding textual code, which they were then able to change by rewriting it. When doing so, thes tudents used photos as a clue to understand the content of the text-based source code. For example, they could see that code around photos of people taking up postures determined the posture of a person, while code around photos of robots sent commands to the robot. In other words, the photos visualized basic information about the data as to be readily accessible. This could be useful for giving hints to beginners learning about programming.

This shows that embedding photos in text-based source code allows users without prior knowledge of programming to make edits to the code, an action that has always been restricted to programmers with knowledge of programming. As Bret Victor's essay [143] notes, source code should express execution results directly (without the need to imagine based on specific knowledge). Source code containing photos fulfills this objective, and can be seen as realizing the "popularization" of code. When we compare source code with and without photos as shown in Figure 4.7, it is obvious that the photos stand out in the text code, helping the programmer briefly skim and understand its intention.

4.6.2 Environment Information Expressed in Photos

Posture information does not include information about the purpose of human or robot postures. Humans may take up the same posture whether they are pushing a cart on a slope or performing calisthenics. In the same way, a robot takes similar postures when pushing small and large balls, with the only difference being in how wide the hand opens. Because two pieces of posture information are unlikely to be exactly the same, while numerical distinction is possible in theory, this is actually very difficult for humans. Even if visualized using 3D CG, these differences are likely to be almost invisible.

Photos also contain environmental information other than humans or robots. We were able to obtain a number of photos that clearly showed the objectives of a person's activity in the workshop (Figure 4.8 left). Moreover, these photos

```
MindstormsNXT nxt;
boolean flag = false;

void setup() {
  nxt = new MindstormsNXT();
  nxt.connect();

  // Show the preview window so that
  // CHI people can see the robot :)
  nxt.showCaptureFrame(true);
}

void draw() {

  // If the robot is handling a task, do nothing.
  if (nxt.isActing()) {
    return;
  }

  if (flag == true) {
    // If the flag is true
    nxt.setPose(Picode.pose("both"));
    flag = false;
  }
  else {
    // Otherwise, if the flag is false,
    nxt.setPose(Picode.pose("none"));
    flag = true;
  }
}
```

```
MindstormsNXT nxt;
boolean flag = false;

void setup() {
  nxt = new MindstormsNXT();
  nxt.connect();

  // Show the preview window so that
  // CHI people can see the robot :)
  nxt.showCaptureFrame(true);
}

void draw() {

  // If the robot is handling a task, do nothing.
  if (nxt.isActing()) {
    return;
  }

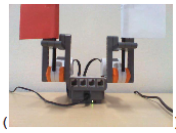
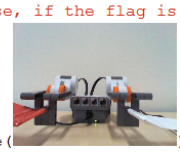
  if (flag == true) {
    // If the flag is true
    nxt.setPose(  );
    flag = false;
  }
  else {
    // Otherwise, if the flag is false,
    nxt.setPose(  );
    flag = true;
  }
}
```

Figure 4.7: Pure text-based code and photo-integrated code.

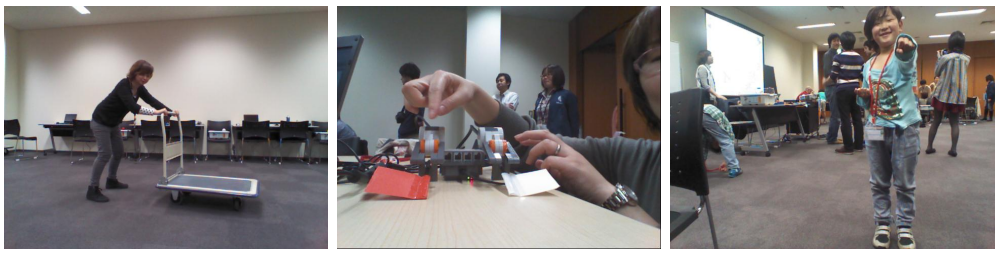


Figure 4.8: Information included in photos. Left: environment information, Center: indication, Right: emotion.

show unspoken prerequisites for the program to function. For example, robot movement will differ sharply based on whether a robot is moving on flooring or on carpet.

It's extremely important to be able to quickly grasp the operating environment and objectives of a program when viewing code written by another programmer, or even one's own code after a long period of time. In existing source code, which is composed only of text, this information is carried in comments. Other parts mainly fulfill the role of giving instructions to the computer. In contrast, photos in the source code created with Picode fulfill both of these functions. In other words, the photos serve to supplement human comprehension of the program's operations while also expressing posture information to the computer.

4.6.3 Indications Expressed in Photos

Posture information is a collection of angle information for multiple joints that handles all of that information equally. However, when a programmer utilizes posture information, there are often parts of that information to which he or she wants to give particularly important meaning. In these cases, programmers add textual comments such as "secondary joint." Programmers can also add annotations to points of focus in 3D CG posture visualizations.

Such indications can be carried out more simply by using photos. For example, photos were taken in the workshop that pointed to specific movable parts of the robot to indicate the area of focus (Figure 4.8 center).

On the other hand, indicating specific joints meant that only a portion of the information expressed in the photo was useful for programming. Here, we see limitations on photos that will be discussed further below. In other words, posture information is defined as a type of structure in textual source code, and as with the textual statement *Pose.SecondJoint* (in which *SecondJoint* is a property of a *Pose* class), we should be able to indicate the specific joint in question. However, taking photos that display only specific joints is difficult, and it would be difficult to understand what such a photo was indicating.

4.6.4 Emotion Expressed in Photos

Photos have the power to express emotion by showing a single moment of activity (Figure 4.8 right). This characteristic of photos has been important throughout the long history of the medium, and represents an attribute not found in source code. When the workshop began, the pose library used by the participants included only photos of the author. But by the end of the workshop, it was filled with photos showing the enjoyment of the participants and creative shots of the robots. The pieces of generated source code were also extremely individual, with photos showing a variety of clothing, poses, and expressions even without differences in the program logic other than the posture information.

Finding ways to increase motivation among students of programming is a major problem for education in the field. Two main causes are given for lowered motivation. First, programming is comprised mainly of the tedious task of facing a display and typing. Second, because the source code that expresses instructions to the computer places emphasis on functionality, it ends up being dull.

By adding photos, a media that can express emotions, to programming, Picode offers a solution to this problem. First, taking photos with the camera adds a step to the programming workflow in which users can stand up and move around. In the user study, we observed that participants greatly enjoyed this activity. In

addition, the Picode source code becomes memorable to the users. Participants were very happy to receive printouts of their code after the workshop ended.

4.6.5 Robot Shape Information Expressed in Photos

In the workshop, we repeatedly observed users obtaining new posture data by rearranging the robot's arms and taking a new photo. If shapes were different when comparing the robot and photos, the participants could make the determination that the robot would not take the position they intended without actually trying the instruction. If participants did not have photos and were using only strings of numbers or names, they would have to try to guess at what kind of robot form was recorded in posture information, which would be difficult.

Existing methods have attempted to express posture information by visualizing robot shape information using 3D CG. However, methods which manually indicate shape information are not seen to be suitable for prototyping by repeatedly rearranging the robot shape as in this workshop. This would require manually updating the 3D CG model information in the computer whenever the actual shape of the robot was rearranged, which would require a great deal of effort.

4.6.6 Intrinsic Limitations of Photos

While photos express a variety of information as shown above, they also contain intrinsic limitations.

First, photos are not suitable for precisely distinguishing posture information. For example, one participant took multiple photos of the robot making only major changes, creating a number of photos that all looked largely similar. It is difficult to precisely determine what shape the robot was in when the photo was taken, which invites confusion. Another participant took multiple photos without changing the position of the camera and robot and had difficulty adjusting the position of a place covered up in the foreground. As in this case, covering up parts can lead to large discrepancies in the posture information, even though the photos look almost entirely the same. One solution would be to string together multiple photos from different viewpoints for one piece of posture information. This would ensure that all parts of the posture are visible from some viewpoint and eliminate the problem of covering. Also, by preparing photos that focused on parts as well as the overall view, this would make it easier to determine small variations. When taking photos of human postures using the Kinect camera, the camera is unable to correctly obtain posture information if the subject is obscured. Because Picode does not allow a photo to be taken if posture information cannot be obtained, there were no problems related to human postures being obscured.

Because photos can depict a situation and show a variety of objects, the focus of the information or what exactly the situation is showing can be unclear. For example, if two robots are shown in a photo that is meant to express posture for only one robot, there is no way to determine what posture data is being expressed. When comparing posture information in a photo with two human subjects, there is no way to determine whether to take the logical sum (there is at least one person for each pose) or the logical product (exactly one person for each pose) of the posture information. In cases in which multiple interpretations are possible, it is necessary to clearly select one after the photo has been taken and highlight it to make it stand out as the object of the photo.

The current implementation for Picode expresses posture information for an

entire body in one photo and does not support expressing angle data for a specific joint or group of joints. For example, we can imagine a use case in which one wishes to ignore everything other than the left hand. In this case, we could suppose a process in which we add highlighting to a specific area of a photo in order to express the specific joint or group of joints. Because photos taken using the Kinect give the positions of all joints in the image, it could be possible to perform interactions such as painting a mask layer over joint areas that we wish to ignore in the comparison. Users generally face the camera head on in photos taken with the Kinect, and in many cases selecting joints would be simple, but in difficult cases such as a covered robot, it would be necessary to supplement the existing method with techniques such as rendering the joint information in 3D CG and rotating it.

Moreover, a photo captures a single moment and can only express that instant. The current implementation tolerates a certain degree of error in posture information. For example, in the flag-raising game, even if the program cannot accurately detect the matching posture, it will use a similar posture to continue the game. However, this method cannot be flexibly applied to cases in which we wish to express a range of situations. Using the method mentioned above in which we would highlight a specific area, it would be possible to tolerate a large amount of error in a specific set of joints. However, if we wished to express a range of conditions, such as the full range of the left arm raised to lowered, it would be difficult to convey that intention by simply highlighting the arm in the photo. In this case, we could use multiple photos or a movie to express a range. For example, a set of photos showing the right arm raised high and raised a bit lower could be used to express a general state of "the right arm being raised (regardless of height)". Implementing this in the current version of Picode would require writing two sets of conditional statements and using trial and error to determine tolerable error values for each. This could be solved by using instantiation programming that learns multiple pieces of posture information. For example, we could select multiple photos from the pose library and define a "Pose set," and then generate a classifier using photos that include the pose as positive examples, and other photos as negative examples.

4.6.7 Utilizing Media Other Than Photos

We plan on adding videos as well as photo to source code. Photos show a single moment containing posture information. Thus, when recognizing human gestures or commanding the robot to perform a movement that takes time, we treat a number of photos of the movement as key frames and use these in writing the program. However, it would be preferable to use video rather than photos as a base for creating a gesture recognition engine or replaying robot movements. Videos include replay speed parameters. By adjusting the replay speed of a video when using it to operate a robot, we could alter the speed of the robot's movement.

Moreover, methods to add multimedia could be applicable not only in developing programs that interact with the real world, such as posture information processing, but also in the development of a wide range of desktop applications such as game engines and presentation composition software. Sikuli, who added screen captures of the desktop into source code, is a pioneering example of this type of work.

This study found that photos are suitable for expressing posture information, but we must consider whether or not this method would be suitable for other

types of interaction with the real world. For example, in recent years displays that recreate texture and devices that output scents have been developed. Source code written only with text would be incapable of arousing the sensations presented by programs for devices that use all five senses. Adding photos that show objects associated with those sensations could allow us to express this information. For example, photos of an object with rough surface could be used to express rough haptic information, and photos of flowers could be used to express the scent of a flower. In addition, if a developer is using a computer with a textured display, texture data attached to a photo that activates when the developer touches the photo could be used to intuitively reinforce the sensation linked with the photo and the data it represents. However, it may not always be possible to link photos and sensation information. We were able to take photos that corresponded with posture information, but some sensation data may not be suitable for photos. For example, in addition to musical tones and mechanically produced textures and scents, there is also pain generated through electrical stimulation. In these cases, we may need to present these using interfaces that replay sensation data without relying on visual representation. For example, an interface could include buttons that replay music when clicked, create a texture when touched, or release a scent. There is also a phenomenon known as synesthesia in which one sense recalls another. For example, some people see colors when they hear sounds. Specific sense information that recalls shapes or colors could be displayed to create intuitive source code.

4.7 Summary of Contributions

In this chapter, we introduced a method of integrating photos into a text-based IDE. The aim was to assign graphical representations to constant data; i.e., the term c in the model $out = f(in, c)$, introduced in Chapter 3. Each photo is bound to constant data, which refers to a specific situation in the real world.

The experimental implementation; i.e., the Picode IDE, provided a user interface to capture photos and posture data simultaneously, and creates a pair of the photo and posture data. The photo is stored in the pose library and can be inserted into the editor via a drag-and-drop interface. We find that inline photos provide rich contextual information that could not be included in the posture data. The contextual information can remind the programmer of the situation and facilitate their understanding of the meaning of the surrounding text-based source code. We also found several intrinsic limitations of photos, through which we have identified directions for future work. The use of videos will be discussed in Chapter 5, and annotations on photos will be discussed in Chapter 6.

Chapter 5

Using Videos to Understand Dynamic Behavior

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. In such case, augmenting the code editor with graphical representations as examined in Chapter 4 should provide sufficient support for the programmer. However, when the programmer writes code for processing raw values of real-world I/O such as images from a camera, the simulation is almost impossible. The increasing popularity of interactive camera-based programs highlights the inadequacies of conventional IDEs in developing these programs given their distinctive attributes and workflows.

Previous approach tried tight integration of the debugger and text-based editor. For instance, ZStep [94] records all stack traces and provides a navigation interface to go back and forth the trace to see which line of code was executed at that point. It is also capable of visualizing simple data structures such as trees and lists. Whyline for Java [80] also records the stack traces as well as window output and provides "Why did (not) this happen?" interface which navigates to the cause of the phenomena, such as the color of a pixel and weight of a line stroke. These integrations work well for discrete events with simple data structures, but are not designed to handle continuous real-world I/O data.

In this chapter, we discuss the use of videos as the graphical representations of the dynamic changes of situation in the real world. Compared to photos discussed in Chapter 4, videos have additional dimension in time, enabling programmers to visually and continuously monitor what is happening in the programs. In particular, we are interested in interactive camera-based programs which have the frame-based pipeline that handle real-world I/O continuously. First, in Section 5.1, we introduce the background and highlight the distinctive challenges with explanation of the development process of interactive camera-based programs. In Section 5.4, DejaVu IDE is provided as an experimental implementation which allows the programmer to easily record, review, and reprocess temporal data. It enables to iteratively improve the processing of non-reproducible camera input. We showcase its important features by presenting a concrete use case. In Section 5.5, we also report its implementation. In Section 5.6, we introduce our preliminary user trial with three experienced programmers of interactive camera-based programs, in which DejaVu was positively received. In Section 5.7, we provide some insights for future work according to the user study.

5.1 Background

Interactive systems beyond desktop computers and mouse/keyboard input continue to increase in popularity, where users can use their hand, body, or passive physical objects to interact with computing devices. At the heart of many these interactive systems are cameras used to capture input from the real world that is then interpreted in real-time by computer vision algorithms. For example, cameras are used to recognize hand gestures on tabletops [149] and in the air [135], detect human faces [146], track tangible implements [25], as well as monitor crowd activity [105]. Moreover, developing these computer-vision-based interactions has become easier through commercial products such as Microsoft Kinect (which performs body skeleton tracking through a depth camera), as well as software development kits (SDK) of well encapsulated algorithms.

However, despite the increasing accessibility of camera hardware and computer vision algorithms, today's development environments do not cater to the distinctive challenges and workflows of developing interactive camera-based programs. For example, the programmer has to monitor data in the debugger as discrete textual values rather than continuous visual representations that more accurately reflect interactive computer vision data. Such disconnects illustrate the gulf of execution [121] as a gap between the programmer's goal and the available means to execute it. As a result, programmers can still find it difficult to develop such programs even if they possess good computer vision knowledge.

To close this gap, we present DejaVu that enhances conventional integrated development environments (IDE) to better support the development of camera-based interactive programs. This work differs from lower-level computer vision algorithm libraries such as OpenCV [14], or rapid prototyping tools for camera-based applications such as Crayons [41] and EyePatch [106] that are aimed at making certain computer vision techniques accessible to non-programmers through a special user interface. Instead our high-level rationale is similar to Gestalt [125], a general-purpose development environment for machine-learning applications, in that we focus on facilitating a general workflow for current developers of interactive camera-based programs without limiting them to certain algorithms or dramatically changing their programming habits. DejaVu aspires to minimize workflow overhead and draw computer vision programmers closer to the essence of their program. More specifically, DejaVu includes two interlinked main components (Figure 5.3): a canvas to visually and continuously monitor the inputs, intermediate results, and outputs of computer vision processing; and a timeline to record, review, and reprocess the above program data in a temporal fashion.

5.2 Interactive Camera-based Programs

To help introduce DejaVu, we first explain how today's IDEs fall short in supporting the development of interactive camera-based programs. This knowledge was obtained both through our own experience (two authors were deeply experienced in developing such programs) and informal interviews with three similarly experienced developers. We first introduce a simple example application named KinectDress to familiarize readers with interactive camera-based program basics, and then elaborate on challenges in their development using today's environments.

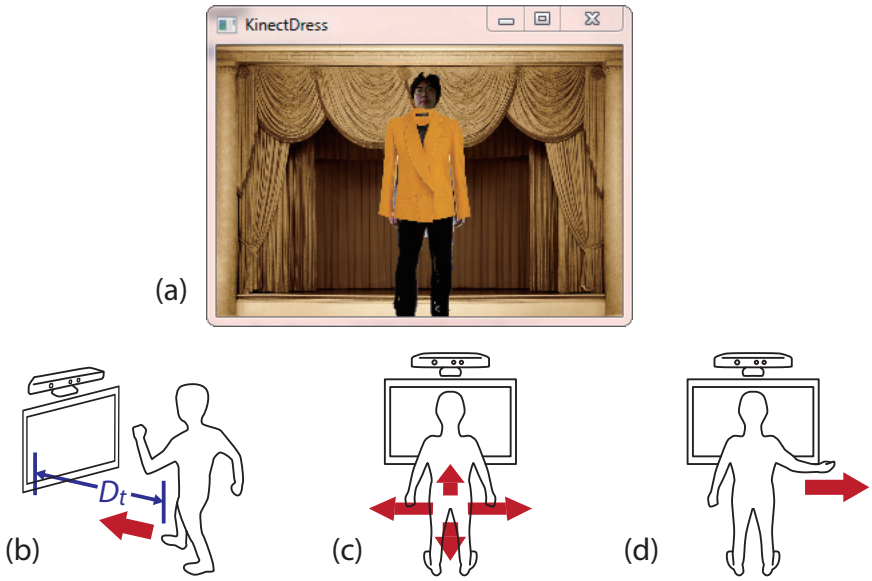


Figure 5.1: KinectDress interface and interactions.

5.2.1 A Representative Example

KinectDress (Figure 5.1a) is a simple virtual dressing room application built with the Microsoft Kinect camera, which provides one color (RGB) image stream and one depth image stream (of which pixel values correspond to distances from the camera). The Microsoft Kinect SDK further uses these inputs to compute a body skeleton of the user in front of the camera, consisting of 3D coordinates of 20 body joints. With KinectDress, users can see themselves dressed in various virtual suits on the computer screen. To start interacting with KinectDress, the user simply walks within a certain distance in front of the camera (Figure 5.1b). The user's image is dynamically extracted from the surrounding environment and displayed on a virtual background, and overlaid with a suit that follows the user's position as they walk around (Figure 5.1c). The user can also make a swiping hand gesture to cycle through a list of available suits (Figure 5.1d) to wear.

We carefully designed KinectDress to represent key patterns of general camera-based interactive programs in several aspects:

Interactions

KinectDress includes both the case where the system continuously changes its state in response to the user's current state (e.g., the suit follows the user's body) and the case where the user makes an action to be recognized by the system in order to trigger a command (e.g., a swiping gesture to change their suit). Most camera-based interactions can be categorized into these two main categories.

Program Architecture

KinectDress is typical of most real-time camera-based systems in that the camera is the sole or primary source of input, i.e., the camera "drives" the program. This requires the program to capture and process image frames continuously, hence dictates a frame-based loop architecture. Figure 5.2 illustrates this classic architecture used in KinectDress. Each iteration of the loop starts with the camera capturing the next frame, followed by the

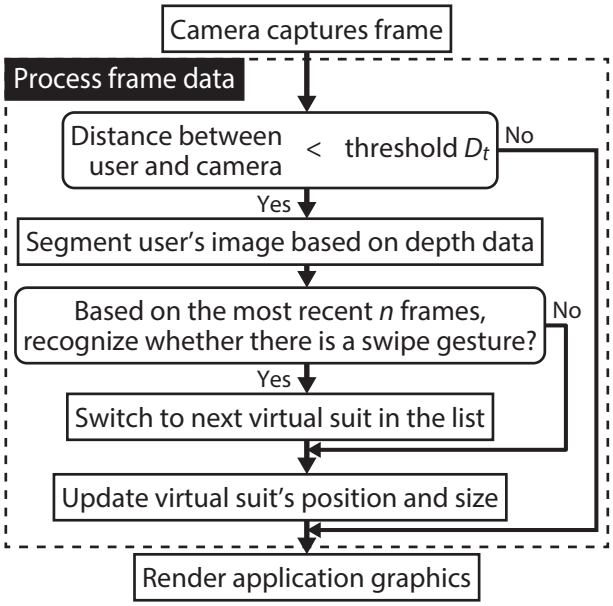


Figure 5.2: KinectDress program flow.

pipeline that processes the frame and updates the system's logical and graphical state accordingly.

Processing Paradigms and Components

KinectDress includes both stateless processing that depends only on the current frame (e.g., updating the suit position) and stateful processing that accumulates data over a number of recent frames (e.g., recognizing swipe gestures); both are common in interactive computer vision programs. KinectDress also demonstrates several of the most common processing components in camera-based interaction such as image segmentation, geometric transformation, and heuristic gesture recognition (Figure 5.2). Finally, KinectDress illustrates how color, depth, and skeleton data are processed in combination as common in Kinect programming.

5.2.2 Attributes and Challenges

Several fundamental attributes of interactive camera-based programs pose challenges for development with today's environments:

First, computer vision processing is inherently visual: not only is the raw camera input a stream of image frames (or several streams in the case of stereo or depth cameras), but many of the intermediate processing results are also images (e.g., segmented user image in KinectDress), or have a close geometric correspondence with the input images (e.g., body skeletons) and so are best understood visually. In this respect, today's development environments disregard the visual nature of this data and display their textual value, falling short of the programmer's needs. To ease development, computer vision programmers often write temporary code to visualize some of this data themselves in the application user interface, which is both cumbersome and not scalable.

Second, the inputs of most camera-based interactive applications are continuous: the program constantly receives and processes real-time input from the camera, updating intermediate results and final outcomes on a frame-by-frame

basis. Such processing continues even when no user actions are occurring, e.g., KinectDress constantly monitors whether there is a user within a certain range. In addition, many user actions, especially gestures, do not happen at a single point in time but rather span multiple contiguous frames. However, today's development environments are usually designed to trace discrete user input events, and programmers cannot directly inspect the temporally continuous dataflow of camera-based programs. For example, debugging using breakpoints can be problematic since they inevitably interrupt the temporal continuity of live input.

Third, camera-based input is mostly non-reproducible: input is formed by dynamically observing the real world and often human behavior. Compared to mouse-and-keyboard programs where the programmer can easily reproduce a certain input sequence (even through an automated script) to test them, the dependency on dynamic real world input in camera-based interactions means that it is not only cumbersome but also often impossible to reproduce certain input. For example, a human user can never perform the same action, such as KinectDress's swiping gesture, twice precisely the same way. Other factors such as lighting, environment setup, and even noise in the camera sensor, may also result in different inputs and cause different outcomes. Such non-reproducibility poses a serious obstacle to testing and tuning interactive computer vision programs in today's IDEs.

Finally, developing computer vision programs is often an iterative process. The stochastic nature of camera input from the real world along with the somewhat obscure nature of many computer vision algorithms means that predicting the exact outcome of a certain computer vision algorithm is often difficult. Furthermore, given the complexity of real world input, the correctness or quality of a computer vision program's output is often up to the programmer's subjective judgment (e.g., whether a suit's position and size matches the user's body in KinectDress). For these reasons, computer vision programmers more often "tune" an algorithm rather than "debug" it. As a result, developing computer vision programs often involves a great deal of trial-and-error with real world input, such as revising the algorithm, adjusting its parameters (e.g., distance threshold D_t in figrefdejavu-workflow), or comparing multiple variations of the algorithm to find configurations that yield satisfactory behavior. In some cases, this process needs to be repeated when the system is used in a new environment or for a new user group. The need to repeatedly acquire dynamic real world input makes such iterations and comparisons cumbersome and unreliable.

5.3 Related Work

5.3.1 Tools for Building Computer Vision Applications

A great deal of previous work endeavor to make employing computer vision for real world applications easier. Several systems aim to make design and prototyping computer vision techniques accessible to non-programmers. For example, Crayons [41] is a design tool that allows users to train image segmentation classifiers using a coloring metaphor, which are then used to prototype interactions. Similarly, Eyepatch [106] supports prototyping camera-based interactions through examples where users train various classifiers and then connect their live outputs to other prototyping tools such as Flash. Concerning more specific application domains, the Papier-Mache toolkit [74] supports building tangible user interfaces through computer vision, barcodes, and electronic tags; and users of CAMBIENCE [36] can map motions detected by the camera into various sound

effects. In contrast to this category of work, DejaVu targets typical programmers and general-purpose interactive camera-based programs by supporting a canonical development workflow rather than individual computer vision components, and preserves the full power and flexibility of standalone computer vision programs.

On the other hand, several software libraries of lower-level computer vision algorithms, such as OpenCV [14] and XVision [55], can readily be leveraged by programmers in their programs. DejaVu fulfills a complementary need, and may be used together with these libraries seamlessly.

5.3.2 Prototyping and Development Tools for Other Domains

In addition to computer vision, rapid prototyping tools also exist for other domains, such as sensor-based interactions that are especially relevant to our work. In specific, d.tools [61] integrates the design, test, and analysis of physical prototypes including sensors, while also providing a visual programming environment for authoring control flow. Exemplar [59] supports the authoring of sensor-based interactions by demonstration. Both d.tools and Exemplar include functionality to capture and visualize temporal sensor data and interface states, which is somewhat similar to the DejaVu timeline. Further, RePlay [120] and FauxPut [27] both support the recording and replaying of sensor input traces for the purpose of testing prototypes. To support mainstream development instead, DejaVu seamlessly integrates these concepts into a general-purpose development environment, extends them to flexibly support arbitrary data variables in the program, and further enables timeline refresh based on iterative program revisions.

Also worth noting is Gestalt [125], a general-purpose development environment that supports the development of machine learning applications. Gestalt shares our design rationale by supporting a general workflow (implementation, analysis, and easy transitions between the two) for machine learning rather than focusing on individual algorithms. Further, the connection between DejaVu and Gestalt could go beyond this philosophical similarity. As apparent in the various computer vision prototyping tools [41, 106] mentioned above, machine learning is an important element of many computer vision algorithms. DejaVu focuses on the distinctive challenges of interactive computer vision; however, future work could consider how aspects of both systems would be combined to support a more comprehensive development process.

5.3.3 General Programming and Debugging Support

DejaVu is also related to general programming and debugging research. DejaVu can record, review, and reprocess input, intermediate results, and program output, which resonates with a long thread of research on temporal debugging where programmers can examine the program state at various points of time in the past. Initially explored in EXDAMS [7], its first graphical example appears in PROVIDE [112] and more recent work includes TOD [128] and URDB [144]. Most relevant to our work is liblog [46], a replay debugging tool for distributed applications that share some of the non-deterministic nature of camera-based applications. These systems focus on tracing and reverse-stepping of individual discrete statements, and do not accommodate or exploit the intrinsic frame-based processing pipeline in interactive camera-based programs as DejaVu does.

Another key capability of DejaVu is to continuously monitor the program data in a visual fashion. The GNU Data Display Debugger (DDD) [152] allows data

structures to be visualized as graphs, while Microsoft Visual Studio [111] allows programmers to create custom visualizers of data types (e.g., images) that can be viewed in the debugger. However, these visualizations are built into conventional discrete-step debugging environments and are not updated continuously during program execution.

DejaVu's ability to revise the program and reprocess the input may also remind of research on live programming such as SuperGlue [107] and Subtext [38], where the program is continuously and immediately responsive to any edits in the code. Although DejaVu does not yet provide such a live programming experience, we see this as a promising future direction to further facilitate the iterative development of camera-based programs. Motivated by a similar need, Juxtapose [63] provides an alternative approach that allows the simultaneous testing of multiple program variations, potentially with the same input. Compared to Juxtapose, DejaVu is more suited to the iterative development and testing process where developers incrementally extend and improve their code over time.

5.4 DejaVu IDE

DejaVu enhances an IDE to reflect the visual and temporally continuous nature of interactive camera-based programs, and to accommodate non-reproducible real world input as well as an iterative development processes. DejaVu is prototyped as an extension to SharpDevelop [5], which is a general-purpose open-source IDE for Microsoft .NET development. DejaVu preserves the full flexibility of the development platforms and patterns developers currently use to write interactive camera-based programs. The only assumption made is that the program follows the previously mentioned canonical frame-based loop architecture where all input and output are synchronized to frames - we do not readily support multi-threaded asynchronous programs, which are nonetheless highly uncommon in real-time camera-based interactions. Without loss of generality, the prototype currently interfaces with a Kinect camera (which may also be used as a regular RGB camera), while extending support for other camera types is straightforward. The DejaVu interface (Figure 5.3) consists of two tightly interlinked components: the canvas and the timeline.

5.4.1 DejaVu Canvas

Reflecting the continuous and visual nature of camera input and processing, the canvas (Figure 5.4) 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 automatically shown in their appropriate visual form. To add a variable to monitor, the programmer simply selects it in the code editor and drags it onto the 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 the canvas via a checkbox. The above actions together allow the programmer to monitor any input, intermediate result, or output of the program.

The canvas always reflects variable values at the current frame of interest (FOI). When the program is running with live input from the camera, this is simply the latest frame that has just been captured and processed. Unlike con-

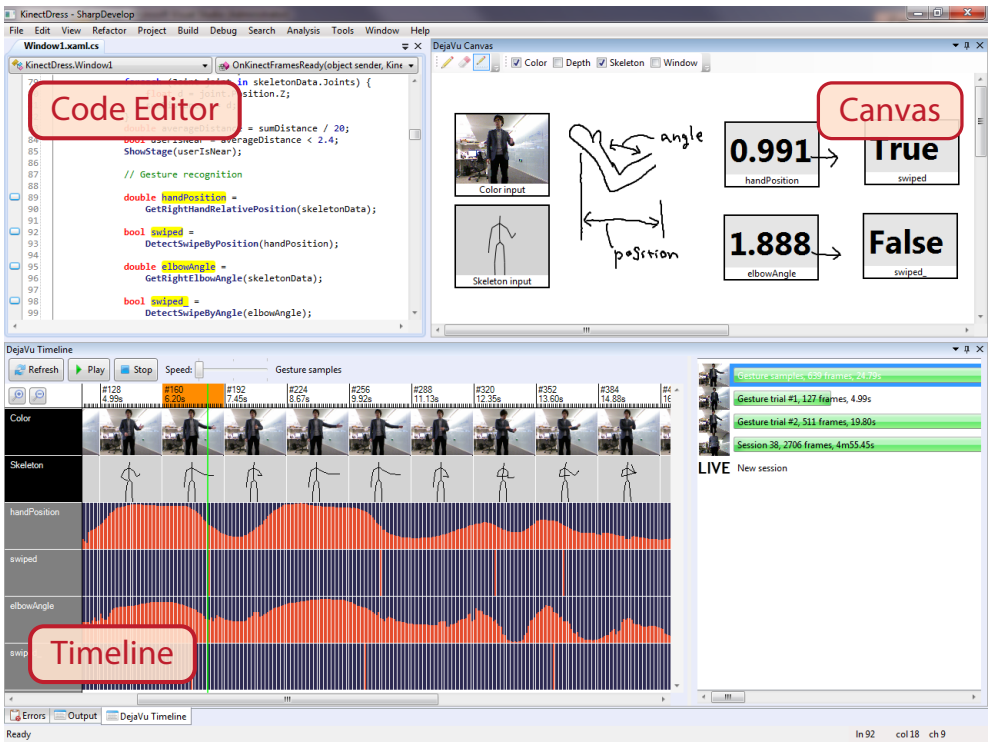


Figure 5.3: DejaVu Interface.

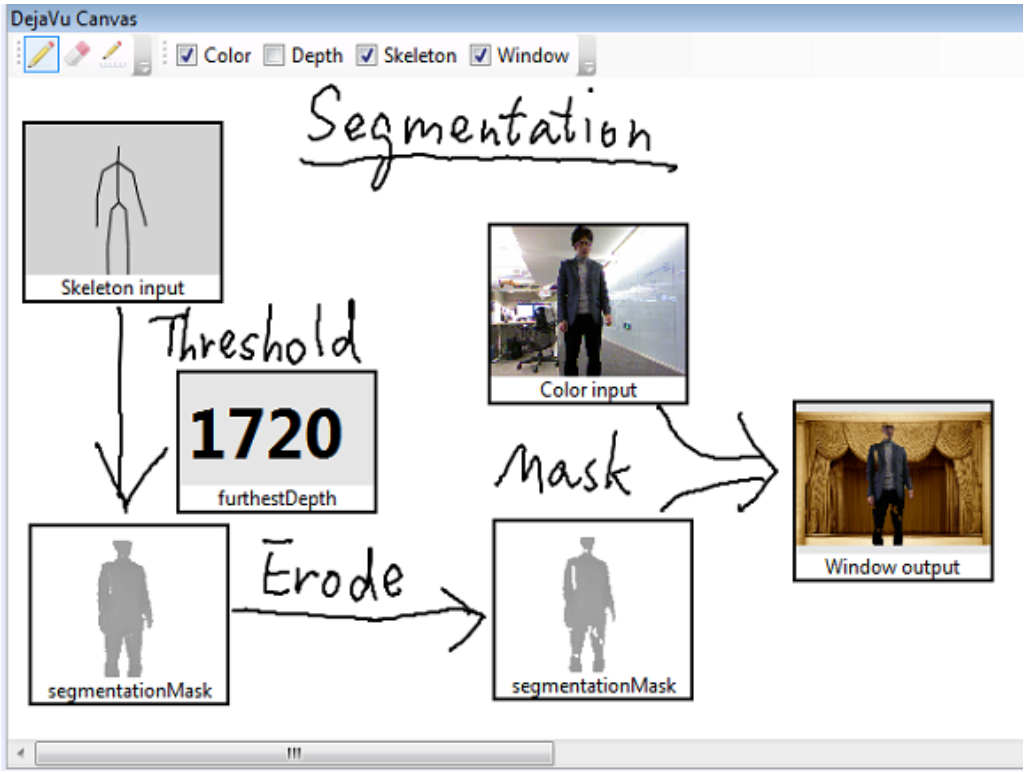


Figure 5.4: DejaVu Canvas.

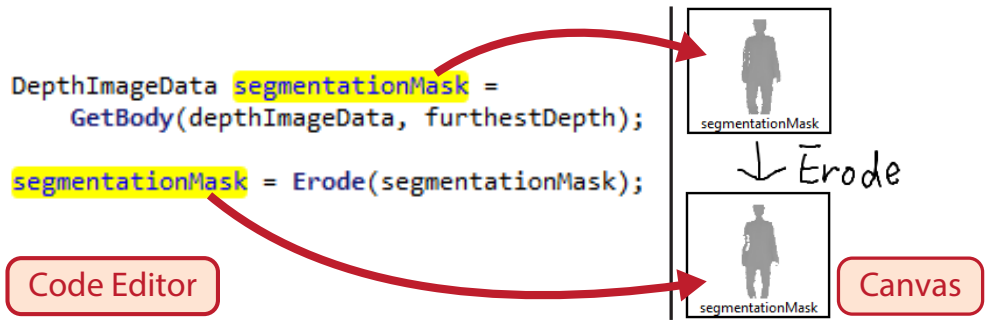


Figure 5.5: Variable values in the canvas depend on their source positions in the code editor.

ventional debug watch tables in which the variable values are only updated when the program reaches a break, the canvas is constantly updated at every new frame so the values can be continuously monitored in real time. When the program is not running with live input, the FOI is dependent on the cursor position in the timeline as explained in the next section. In the case that a variable in the canvas has an undefined value in the FOI (e.g., the variable is declared within a conditional branch that is not reached), its display is blank.

The canvas is updated at the granularity of a frame to reflect the frame-based nature of interactive computer vision processing. However, there may be cases where a variable is assigned to values multiple times during the processing of a single frame, which often happens when the programmer applies an image processing filter (e.g., Gaussian blur filter) or transformation (e.g., transforming between color spaces) to an image in place, i.e., the result is assigned to the same variable that represents the source image. The canvas maintains a record of not only a variable's name but the source position in the code editor where it was dragged from, and inspects the variable's value just after it is evaluated at that position. In doing so, the programmer can monitor a variable's value at a specific stage in the processing pipeline, or even simultaneously monitor its values at different stages within the same frame by adding the variable to the canvas several times from different positions (Figure 5.5).

Along with displaying variables, the canvas also allows the programmer to freely write or draw on it using a stylus or a mouse to further aid in the thought process of handling visual data. In addition to the obvious use for annotating variables, freehand drawing enables other powerful use cases: by combining static sketches such as algorithm flowcharts with data displays, the programmer can turn the canvas into a "dynamic sketchbook" where sketches come to life with dynamic data. The programmer can then inspect the program dataflow and pipeline on a higher semantic level, providing a more vivid way of conceptualizing and iterating on algorithms. On the other hand, in contrast to visual dataflow authoring tools such as Max/MSP [3], this usage remains lightweight and flexible, and does not dictate literal correspondence between the sketch and program. Alternatively, the programmer may make a coarse sketch of their application UI on the canvas and populate it with data displays to use it as a low-fidelity interactive prototype in lieu of the actual application user interface, which is reminiscent of research on sketch-based prototyping [95].

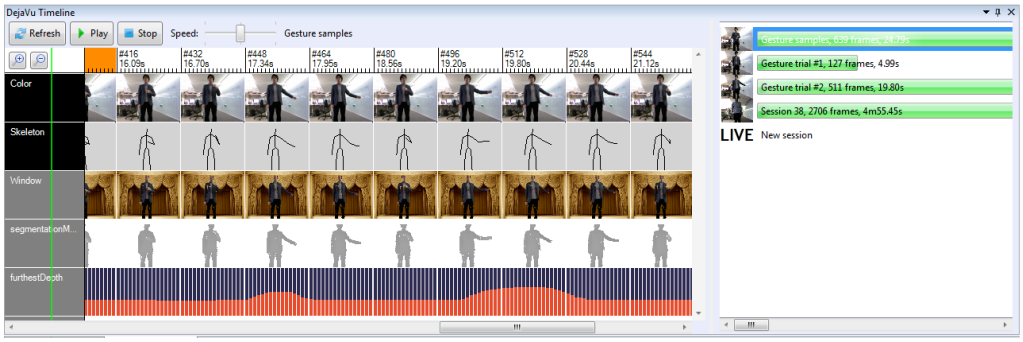


Figure 5.6: DejaVu Timeline.

5.4.2 DejaVu Timeline

The timeline (Figure 5.6) presents program data recorded or recalculated from historical program sessions. A list of all available program sessions is shown to the right of the timeline as horizontal bars, with their visual length proportional to their temporal duration. Program data in the currently selected session is visualized in the timeline in a style similar to that found in common video editing software such as Windows Movie Maker, where a cursor indicates the current FOI in the timeline. The timeline may consist of multiple data streams (rows), each corresponding to a variable, input, or output that is displayed on the canvas. Streams of visual data are represented as strips of frame thumbnails along the timeline, while a stream of numerical or Boolean data is visualized as a time-graph.

The programmer may either review past sessions, or start a live session by running the program with live camera input. DejaVu employs a unified notion of "playing" the session for both cases. To start a live session, the programmer selects the "Live" icon at the bottom of the session list, and clicks the "Play" button. All variables shown on the canvas, along with all available types of live camera input and the rendered application window (regardless of whether they are being monitored on the canvas) are recorded and time-stamped as the program runs. The timeline is populated in the meantime. To stop program execution, the programmer clicks the "Stop" button, and the live session is finished and added to the list of past sessions. Note that the programmer does not need to explicitly trigger program data recording, which happens automatically whenever the program is running live so there is never a risk of missing valuable data or moments.

To review a past session, the programmer selects it from the session list to show it in the timeline. They can then either freely navigate the cursor to an arbitrary frame by clicking on it, or replay the session continuously from the cursor position using the "Play" button. Playing 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. When the current session finishes playing, the next session in the list is automatically selected and starts playing. In any case, the canvas always updates and displays the recorded data in the current FOI. When replaying, the recorded application window output is also shown in a separate window, emulating the live program execution experience. An existing session may be duplicated, split into two at any given point, repositioned in the list, or deleted to allow trimming and reorganizing

the sessions.

The ability to visually review both past sessions and recent live input in the timeline with all relevant program data addresses the non-reproducibility challenge of interactive camera-based input, and eases the identification and analysis of noteworthy events. The seamless transition between live input and reviewing also allows for the fast recognition and examination of events. When the programmer notices some anomaly while testing with live input, they can immediately switch to reviewing the session to deeply analyze it.

The power of the 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 programmer clicks the "Refresh" button so the program is re-executed in the background to recalculate the monitored variable values for all existing sessions in sequence. Sessions and frames are colored green when they are refreshed and ready for reviewing; those yet to be refreshed are colored gray. The refresh functionality allows the programmer to reliably examine the effect of their program revision by comparing to previous outcomes on the exact same input. Finally, the programmer can add variables to the canvas which have not been recorded previously; the sessions will be refreshed to include the new data streams in their timelines.

5.4.3 Example Use Case

We now use the previously described KinectDress application to concretely illustrate how DejaVu can be used by programmers in their workflow.

The programmer's first challenge is to fine tune the distance threshold D_t that determines how close the user should be in front of the camera to trigger the interaction (the program starts displaying the virtual stage to reflect this). Today's programmers usually need to go back and forth several times between adjusting the parameter on the computer, and standing up and walking towards the camera to test the effect until finally satisfied - a very cumbersome and tiring process. With DejaVu (Figure 5.7), the programmer can add the *userDistance* variable (calculated as the average depth of all body skeleton joints) to the canvas, and monitor its value on the computer screen as they walk from afar towards the camera (only once). When they reach a comfortable distance, they can read the current *userDistance* value on the screen (displayed in a big font for readability from afar), and use this value as a hint for setting the threshold.

Alternatively, the programmer can raise a hand to indicate that they are at a comfortable distance, which is easy to visually identify in the color input stream. Later they can iteratively adjust the threshold in the program code and refresh program data, so that the starting moment of the virtual stage (as seen in the application window stream) aligns with the indication action (as seen in the color input stream) in the timeline.

The programmer next needs to extract the user's image from the color input. This segmentation algorithm involves first finding the farthest point among the skeleton joints whose depth value is then used to threshold the depth input image. The resulting binary mask is applied to the color input to segment the user from the surrounding environment. The programmer can use freehand sketch together with data displays on the canvas to help conceptualize this slightly complex pipeline (Figure 5.4). Further, to remove some excessive pixels in the binary mask, the programmer may try applying an erosion filter to it. The ability to monitor the same variable's values at different code positions then allows

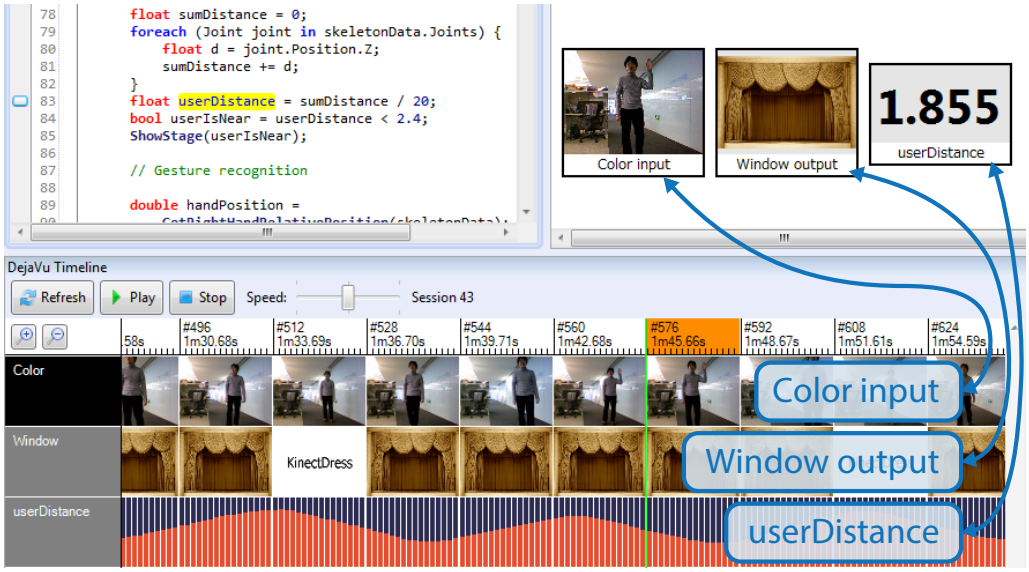


Figure 5.7: Tuning the distance threshold.

both the original mask and the eroded mask to be monitored and compared simultaneously without confusion.

Next, to overlay the suit on the user's image so that it accurately tracks the user's body in position and size, the programmer can fine tune the geometric transformation parameters for the suit picture using both live and recorded input, similarly to how they adjusted the distance threshold in the first step.

Finally, the programmer attempts the gesture recognition algorithm for swiping, which requires observing the user's skeleton over a number of frames to identify the movement. Two simple heuristic algorithms come to the programmer's mind, one based on the change of the hand's horizontal position, the other on the change of the elbow joint angle. Being unsure of which option will work better, the programmer implements both to compare their performance on real world input. Figure 5.3 illustrates how they use the canvas to monitor the skeleton input, hand position, and elbow angle, as well as the recognition results of both algorithms as Boolean variables. Accounting for variability in real world input, they perform the gesture many times. Once done, they immediately have a visual overview in the timeline of how well each algorithm performs comparatively. They can easily identify cases where either or both fail by skimming the color input and recognition result streams, and then diagnose the cause by examining the corresponding temporal trends in the variables that the algorithms are based on, i.e., hand position or elbow angle. They can also later use the basic session editing functions to clean and trim these sessions to focus on the most relevant gesture samples.

Moreover, to accommodate individual differences between users, the programmer can ask others to trial use the program and collect gesture samples for further analysis and improvement of the algorithms. Such batch (re)processing and visualization of multiple recorded sessions are seamlessly integrated in the DejaVu workflow.

5.5 Implementation

We aim to provide an enhanced programming experience by adding two components to existing text-based IDE. Compared with Picode discussed in Chapter 4, DejaVu benefits more from an existing codebase for user interface components, including the project explorer and text-based editor. However, to achieve smooth integration of these two new components with the existing components, the existing codebase must be modified. In this section, we describe how we changed the existing components and implemented the new components.

5.5.1 Overview

The implementation of DejaVu is based on SharpDevelop IDE, which is an open-source text-based IDE. It is capable of editing C#, VisualBasic .NET, Boo and F# programming languages and opening project files generated by the popular VisualStudio IDE. Compared with Processing, which was used as the base for Picode, SharpDevelop is more general purpose, and is equipped with more professional features, including an integrated debugger, code refactoring, and code completion. It officially supports extensions for the IDE, which are used for the implementation of the DejaVu Canvas and Timeline. However, the extension framework was not sufficient to augment the entirety of the programming workflow described in the previous section, and so the existing user interface components, as well as some of the behind-the-scenes features, were altered. DejaVu runs on a Windows computer with a Microsoft Kinect device connected through a USB 2.0 port.

The main window of DejaVu is shown in Figure 5.3. Although the programmer can change the layout freely, we use this default layout as the basis for the discussion. It contains the following components by default. We do not list up the components irrelevant to DejaVu-specific features including the class view and tasks list.

- The menu bar for various operations.
- The project explorer for managing files related to the current project.
- The text-based editor.
- DejaVu Canvas showing information about the current frame of interest.
- DejaVu Timeline showing the video strips of the execution history.
- The status bar showing the current status of the IDE.

The SharpDevelop tool bar that includes an interface for controlling the execution of applications is removed and substituted with play and stop buttons in the DejaVu Timeline. In Subsections 5.5.2 and 5.5.3, we describe the three major components that have DejaVu-specific features. All other changes made to the user interface are related to those features, and thus described in the context of the feature.

In addition the user interface, the compilation and execution process were also modified. Unlike Picode, where we linked to libraries for additional functionality, DejaVu requires monitoring of the status of the applications during execution. Therefore, the modifications were more involved. DejaVu must preprocess the source code prior to compilation and communicate with the library that is loaded

by the application during runtime. We describe the implementation in detail in Subsections 5.5.4 and 5.5.5. The execution is not only monitored at runtime, but is also recorded, as described in Subsection 5.5.6. Once the session has been recorded, it can be replayed, as described in Subsection 5.5.7, or refreshed upon code modification, as described in Subsection 5.5.8. The programmer may also duplicate or split existing sessions, as described in Subsection 5.5.9.

5.5.2 Editor Capable of Dragging Variables

The editor appears almost identical to the original implementation provided by the SharpDevelop IDE. The main difference in terms of appearance is highlighting using a yellow background, which is applied to the variable names monitored on the DejaVu Canvas and Timeline. This is added when the programmer drags and drops a variable from the code editor into the Canvas interface. The C# programming language parser is available, which maintains an abstract syntax tree that is synchronized with the source code. In contrast to Picode, in addition to the abstract syntax tree, the types of the variables can be inferred and stored using a static analysis of the source code. These features are implemented using the NRefactory library[1], which is part of the default SharpDevelop implementation.

When the programmer starts dragging any part of the source code, DejaVu tests whether the dragged string is part of a variable name. If it is not, DejaVu does nothing. Otherwise, it tells NRefactory to run the static analysis to infer the variable type. The result of the static analysis is cached by NRefactory and may be used for other purposes. When the variable is dropped into the Canvas, a placeholder to show the contents of the variable is populated. The placeholder is rendered using a class that implements $VariableBox$ interface and provides type-specific visualization of the variable contents.

Variables that are registered to be monitored are stored as bookmarks that point to specific regions in the source code. Each bookmark contains the start location and the length of the variable name. DejaVu retains the reference to the variable. It is resistant to changes in the surrounding source code and any refactoring that changes the variable name. If the variable type is changed by modifying its declaration, the video corresponding to the variable in the Timeline is cleared; however, the reference is retained and can be updated by the refresh operation. The reference is lost only when the variable is removed. In such a case, any corresponding graphical representations in the Canvas and Timeline are removed. While changes to the source code can be undone, this removal of the variable registration cannot be undone. The programmer must manually drag-and-drop the variable to the Canvas to re-register it.

5.5.3 Canvas and Timeline

The Canvas interface is shown in Figure 5.4 and was implemented as an extension to the SharpDevelop IDE. It is composed of a header showing tool buttons and check boxes, as well as the blank area that extends the *InkCanvas* class. The tool buttons include a hand-drawing tool, a lasso tool and an erase tool. The check boxes are used to show and hide input from the camera (i.e., and RGB video stream and a posture data stream) and the output to the window. Note that these types of data are recorded regardless of whether they are shown or hidden. The *InkCanvas* class is provided by the Windows Presentation Foundation library and is capable of receiving and displaying ink strokes. It allows the programmer

[1] icsharpcode/NRefactory `https://github.com/icsharpcode/NRefactory`

to draw freeform shapes with the mouse, pen or touch. We extend it to be capable of showing placeholders for variables within the *InkCanvas* instance.

When a variable is dropped into the Canvas, its type information is retrieved using static code analysis. The placeholder is then populated, showing the data of the corresponding variable in the current frame of interest. It is rendered using a type-specific renderer. For instance, *NumericalBlock* renders text representing the numerical value and *DepthImageBlock* renders grayscale image representing the depth image from the camera. There are also *ColorImageBlock*, *SkeletonBlock* and *BooleanBlock*, which are capable of showing color images, posture data and Boolean values, respectively. Although DejaVu currently supports a predefined set of types for data visualization, its architecture is extensible enough to support additional types by loading a third-party extension.

The programmer can interact with the contents displayed on the Canvas. He can click on a placeholder to navigate the editor to the specific line of code where the variable is used. The programmer can also use the hand-drawing tool by clicking its button and drawing freeform lines on the canvas, and can later clear these lines using the eraser tool. The lasso tool allows selection of multiple line strokes, as well as placeholders, for variable contents. When the "Delete" key is pressed, the lines and the selected variables are removed from the Canvas and the Timeline.

The Timeline interface is shown in Figure 5.6 and implemented as another extension to the SharpDevelop IDE. It is composed of a tool bar showing the interface to control the execution of the applications, video data showing the history of one session of execution, and the session manager, which shows all of the previous sessions. The videos are updated during the execution so that the programmer can see the changes in the variable data in real time. Therefore, we it must be rendered rapidly. While the other parts of the user interface were implemented using Windows Presentation Foundation, which is a highly managed GUI framework, the video strips were rendered using a lower-level API called Graphics Device Interface+.

The "Play" button behaves in different manners according to which session is selected in the session manager. When no recorded session is selected, it launches the program using live input and starts recording a new session. Otherwise, when any recorded session is selected, DejaVu loads recorded data and replays it. The slider allows the programmer to control the replay speed. Otherwise, it launches the application with the real-time input from the Kinect device. The "Refresh" button causes the program to run in the background and update all output from the program, including the variable values and window output. It does not complete in seconds in most cases, so progress is displayed in the progress bar of the session manager.

The Timeline interface allows zooming in and out to highlight the continuous aspects of the data. In the most detailed scale, each frame is 120-pixels-wide and each frame is rendered. When the programmer zooms out one level, for variables with the image type *ColorImage* or *DepthImage*, one of the two images is hidden and the image retains its width and height. For a variable with any other type, including numerical and Boolean values, the width of each frame is reduced to half that of the original. When the original width is 1 pixel, the size cannot be reduced further, so this size is retained. When the programmer zooms out twice, DejaVu renders one out of four image frames and shows numerical values with a width of 30 pixels.

5.5.4 API for Both IDE and Applications

A custom-built thin wrapper API for the Microsoft Kinect SDK acts a bridge between the DejaVu components and the programmer's code. The wrapper allows the programmer to access Kinect input and capabilities in an API interface similar to that of the Kinect SDK, while at the same time allowing the DejaVu components to track and record Kinect input. The wrapper also allows DejaVu to switch between live feed and recorded Kinect input streams to the programmer's code via the same programming interface, so that the programmer only needs to program for live input. The program naturally follows the frame-based loop architecture by performing frame data processing within a $KinectFrameReady$ event handler, which is called from a single thread managed by the wrapper.

While the library used in Picode always connects directly to the physical devices regardless of whether it is used by the IDE or the application, the library used in DejaVu does not always connect to the physical devices. The DejaVu library first looks for a clue, which indicates whether the library is running on an application outside the IDE, on the IDE, or on the application being developed. In the first and second cases, it uses the Kinect device to read input data to the program. In the third case, it uses a virtual Kinect device, which attempts to read input data from the host IDE. The host IDE is expected to provide the input data, which may be real time input from the Kinect device, or may be recorded input data replayed at a given frame rate.

5.5.5 Data Transfer between IDE and Applications

Real-time monitoring and recording features require the transfer of large amounts of data between the applications and the IDE. The Kinect device is capable of retrieving RGB color images with resolution of 640 by 480 pixels and depth images with a resolution of 320 by 240 pixels, both at 30 frames per second. Posture information can also be retrieved at 30 frames per second. The total data rate is almost 40 megabytes per second. When the application is programmed to respond to these input data, it runs at 30 frames per second. Assume that the window has a resolution of 640 by 480 pixels. This results in 36 megabytes per second, and often contains variables to be monitored; i.e., the color or depth images of the same size, which have data rates of 36 or 2.2 megabytes per second, respectively.

The data transfer must sufficiently fast that applications can be executed without any noticeable delay. However, since the application and the IDE run as separate processes, we cannot transfer these data at the programming language level (e.g., during variable assignment). Instead, we use a TCP/IP connection and inter-AppDomain communication. We prototyped and tested these two techniques and found that the latter; i.e., inter-AppDomain communication, was the fastest. Inter-AppDomain communication is provided by the Windows Communication Foundation .NET Framework for remote method invocations. It is similar to the Remote Method Invocation API in Java; however, is not technically inter-process communication. In the .NET Framework, every process has an AppDomain, which can be thought of as a process on the .NET virtual machine.

We could implement the inter-process communication using shared memory using a custom protocol; however, the inter-AppDomain communication was sufficiently fast and allows us to make use of existing code. With inter-AppDomain communication, when the IDE and applications share the same assembly (i.e., dynamic link library—in our case, the library that provides the API described

in the previous subsection), the IDE can invoke a method defined within via the applications' AppDomain, and vice versa. The IDE triggers the applications' KinectFrameReady event listeners, which handle input data from the camera, and the applications notify the IDE to record the contents of variables and window output.

5.5.6 Recording a New Session

When no existing session is selected in the session manager and the programmer clicks the "Play" button in the Timeline, DejaVu launches the program and starts tracking and recording input from the Kinect camera, as well as the output from the program, which includes variable values and window output. The Canvas and Timeline are continuously updated with the tracked data during execution so that the programmer can monitor the data in real time. The input and output are recorded as different files in different ways. In this subsection, we describe the recording method and how the input and output are recorded. Its architecture overview is shown in Figure 5.8.

Considering the large amount of data that are transferred, it is not possible to store all the data in memory, and so during recording we store the data on the physical disk drive. To keep up with the high data rate, the drive must be capable of writing data at a high bitrate. One method is to compress the data before writing to disk; however, we found that this leads to a bottleneck in terms of processing power, and so decided to record the data without compression or any other post processing. Fast hard-disk drives (10000 rpm) can achieve write speeds of up to 100 megabytes per second. However, this bandwidth is consumed rapidly in a practical setting. For instance, we can record input data from the camera (40 megabytes per second), window output of 320 by 240 pixels (9 megabytes per second), two image variables of the same size (18 megabytes per second) and some integer and Boolean values, consume negligible memory in comparison.

The input from the camera was recorded regardless of its visibility in the Canvas and Timeline. If we simply want to replay the program execution as described in Subsection 5.5.7, only the input data visible in the Canvas and Timeline must be recorded. However, we record all input because it is required to refresh the variable values and window output afterwards, as described in Subsection 5.5.8. The recorded input data includes color and depth images and posture data. These raw input data are continuously recorded with a static length per frame in a single file. Each execution with live input has a unique numerical identifier (session ID) and is used for the filename.

The output from the program, including the variable values and window output, is recorded solely for replaying features. This is achieved by transparently inserting tracing function calls into the programmer's code during compilation, at positions where variables are dragged from onto the canvas, which enables DejaVu's position-aware variable monitoring. Code changes are tracked by the editor and handled during compilation to maintain reference to variables and consistency between the Canvas and Timeline interfaces and the code.

Recording the output from each frame requires two steps. The first generates a table, in which each entry is a pair of the unique numerical identifiers of the variable and the corresponding data. This step is handled in the program's AppDomain. The second transfers the entirety of the table data from the program's AppDomain to the IDE's AppDomain. The transferred data is recorded and visualized by the IDE.

In the first step, the source code is preprocessed and compiled. The pre-

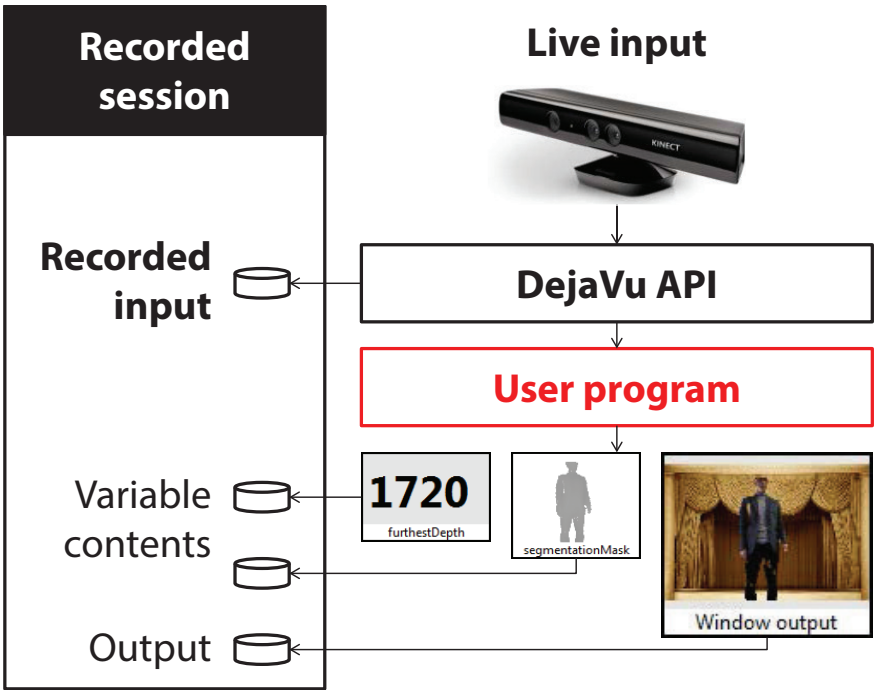


Figure 5.8: DejaVu recording architecture.

processor wraps each registered variable name with a call to the registration function. This step conflicts with the IDE's default feature; i.e., it continuously monitors changes to the source code files made outside of the editor. Therefore, we first temporarily disable the feature, run the preprocessor and then restore the previous state of the feature. When the wrapped registration function is executed during runtime, it places an identifier for the variable together with the corresponding value into the table. For instance, suppose the variable *image* is replaced with $((ColorImage)DejaVu.TrackFrameData(trackId, image))$, where $ColorImage$ is the type of the variable and $trackId$ is the identifier. This casting allows the surrounding code to treat it in a same way as the original variable, preventing any static-type errors. During the execution, the registration process may find an existing pair with the same identifier in the table. In such a case, the value is overwritten. As a result, when there are multiple calls to the registration function, only the final call has an effect; all previous calls are ignored and the data is discarded. This typically happens when the variable is in a subroutine or a loop (e.g., a *for* or *while* loop). There is also a chance that there is no call to the registration function, which will be handled in the second step. This typically happens when the variable is inside an if clause and the condition is not satisfied.

Our preprocessor also inserts the statement $DejaVu.TrackWindowData(this)$ at the end of the body of the $KinectFrameReady$ method in the main window class. This registers a special identifier and the window output to the table. The window rendering should be managed by the Windows Presentation Foundation, so the method locks the rendering thread and copies the contents to a bitmap object. In the current implementation, we assume that there is only one window used by the application during program execution. It is not difficult to extend it to support multiple windows; however, we did not do so since we observed that most of the interactive camera-based applications use only one window to display

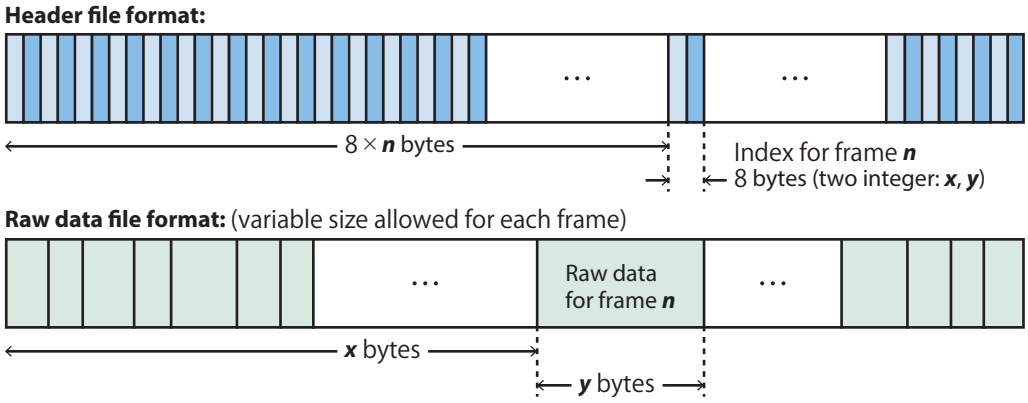


Figure 5.9: File format used for recording variable values and window output.

output.

The second step takes place after completion of DejaVu, triggering all registered *KinectFrameReady* methods. It transfers the table data from the program's AppDomain to the IDE's AppDomain. The recording process is then iterated for all of the identifiers in the IDE's AppDomain. Finally, the Canvas and Timeline view is updated to reflect the newest data being recorded.

The recorded output data are stored in a special binary format as shown in Figure 5.9. Each video in the Timeline interface is stored as a set of two files. Both of them use the identifier for their names, which are identical with the exception of the extension. One is for storing the raw data with the file extension of ".rawstream", which may include the variable size for each frame. The other is for storing the header information with the file extension ".rawstream.index", where each frame has a constant size of 8 bytes (two integers, one representing the location and one the size of each frame in the raw data). For instance, when no value is registered for the frame, the current size of the raw data file and size of zero is stored in the header file and no data are written to the raw data file. There is also one global header file per session, which stores the meta-data, including the timing of each frame in milliseconds, which is required to replay the session.

5.5.7 Replaying an Existing Session

When an existing session is selected in the session manager, a background thread reads the entire header for each video strip. It then reads the raw data used to render the current region of the video in the Timeline interface. Note that the loaded raw data may not be continuous in time due to the zooming level. When the zoom level is 4, it loads only every 8 frames in the video. We save as much memory as possible to avoid "out of memory" exceptions. To prepare for the programmer to scroll the Timeline interface, the thread caches data, which may be shown in the coming scrolling operation. This allows for a smooth scrolling experience. The thread also functions during the execution of the program; when the programmer reviews the past, it behaves in the same way as the recorded session. However, when the programmer scrolls the Timeline interface to the newest part or makes it stick to the newest frame, the incoming data cannot be cached. In this case, the thread simply holds the information required to render the most recent region of the interface.

Replaying an existing session causes the IDE to load every frame in the video.

This occurs independently of the Timeline rendering (although the replay makes the Timeline automatically scroll along in time and requests a view update). A thread is devoted to this replaying feature. For each frame, it loads the header to determine the timing to display the frame and loads all other raw data. It then waits for a specified time period and displays the information on the Canvas interface and the dummy window showing the output. The Timeline shows a tick mark at the location corresponding to the current time of the session. The programmer can manipulate the slider to adjust the wait time and dynamically change the play speed.

5.5.8 Refreshing an Existing Session

In Chapter 3, we described a model of a program, i.e., $out = f(in, c)$. In this model, *out* is determined from a set of f, *in* and c. When any of these are changed, *out* is also changed. The refresh feature of DejaVu is provided to help the programmer update out to catch up with changes in f. This process assumes that *in* represents only the camera input. In other words, we assume that the camera is the only input to the program. If f does not change, execution with the same *in* should reproduce the same *out*. In the current implementation, the programmer refresh *out* even if the program output is dependent on other non-deterministic data, such as use of the *random*() function or network communication. In such a case, it does not necessarily produce the same *out* with a given f.

When the "Refresh" button is clicked, DejaVu preprocess the current source code in the same way as the live input (as described in Subsection 5.5.6), compiles the program, and starts running it in the background by feeding the recorded input data instead of live input from the camera. This allows us to discard the (now out-of-date) output from the program and replace it with new data, which is consistent with the current source code. Unlike recording a new session, this refreshing process records only output from the program and retains the original input to the program. Whereas live input arrives in real time in a periodic manner (i.e., 30 frames per second in most cases), recorded input data can be fed at an arbitrary rate. Therefore, to accelerate the refresh process, we feed each frame as fast as possible.

5.5.9 Managing Existing Sessions

The programmer can name an existing session using the session manager interface to aid in recalling the subject of the session. The programmer can also duplicate an existing session or split it into two at any given point. The duplicated session refers to the same input data, but is assigned a new session ID and the label "copy of" is added to the name as a prefix. The output of the program is then copied to a new directory for the new session ID.

When an existing session is split into two, the original session retains the session ID, but the reference to the later part of the input and output data is eliminated. This is done by simply shortening the length of the session. The raw input data recorded as a file are not affected. The name is changed to have "#1" as a postfix.

The later session is assigned a new session ID and the postfix "#2" is added to its filename. While the input data refer simply to the later part of the original session, the output data cannot take the later part of the original session. This is because the later part is affected by the earlier part, which has been eliminated and no longer exists. To resolve this inconsistency, the IDE automatically runs the

refresh operation (although the programmer may explicitly cancel the operation if he does not feel the need to refresh). This happens when the programmer knows the dependency of state information as a function of time, for instance, when he wishes to debug the recognizer for swiping gestures and already know that the variables of interest are set to their initial states at a given frame. The programmer may then split the session into two at this point. In this way, the programmer can save the time required to refresh the earlier part and accelerate the iteration of changing the source code and refreshing the result.

5.6 User Feedback

To gain early feedback about the concept and functionality of DejaVu from target users, as opposed to lower-level usability or technical performance, we invited three professional developers to trial DejaVu. They all had significant experience in developing interactive Kinect-based programs using the mainstream Microsoft Visual Studio IDE. Each participant was first introduced to DejaVu's concept and interface and then asked to use it in the development of a simple interactive program. The program idea was proposed by the participant based on their past experience and generally consisted of a single processing component that can be used in higher-level applications. 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.

Given the open-endedness of the programming tasks, and because we were interested in subjective feedback rather than quantifiable productivity at this proof-of-concept stage, we did not enforce the participant to complete the program. Instead the participant worked for an hour regardless of the progress. The participant was asked to raise any feedback they may have during the trial, and was afterwards informally interviewed about their experience and opinions. 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 were very positive about DejaVu. They all agreed that it is very useful for developing interactive camera-based programs ("This IDE is very interesting and useful, awesome."), and it matches well with their current workflow in developing such programs. Participants found that the canvas was an indispensable component and cherished the ability to continuously "see immediate result" of variable values. They were particularly fond of the direct drag-and-drop interaction to add a variable onto the canvas, and found the capability for the data display to be sensitive to the variable's source position "very impressive".

The timeline and its associated recording, reviewing, and reprocessing functionalities immediately resonated with the participants, and were seen as the core competency of DejaVu. One participant described it well: "(in the past) I just want to check one value, but maybe need to walk around many times⋯ (with DejaVu) no need to run back and forth⋯ it'll save us lots of time to debug this." Indeed, similar capabilities had been desired by the participants, even to the point of making their own attempts. One participant used a separate toolbox to record and replay Kinect input data, while another participant wrote his own program to do this. However they both agreed that these separate recording functions were not nearly as powerful and flexible as the visual, integrated, and

interactive support in DejaVu. The fact that the timeline is "pretty much like video making tool like Movie Maker" was also seen as a reassuring factor.

More importantly, the inseparable link between the canvas and the timeline defines the DejaVu development experience. Both were seen as complementary to each other, e.g., "the canvas shows the dynamic data" and "the timeline provides the alignment of the changing moment", and the synchronous connection between the two was seen as "the best advantage".

Participants made valuable suggestions on how to further improve DejaVu. Beyond lower-level UI and technology polishing, particularly noteworthy are the following:

Simulating and Manipulating Input

It is not always easy to collect input from the programmer's surroundings that satisfies specific realism, precision, diversity, or quantity requirements necessary for program testing. Participants suggested adding the ability to import simulated or prerecorded input such as videos [27, 120], and to manually or algorithmically manipulate existing real world input such as skeletons.

Visualizing Generic Arrays

Beyond visualizing image data, participants suggested that other array data could benefit as well from compact and intuitive visualization in the form of an image for convenient monitoring and reviewing. The ability to visualize arbitrary arrays as images would be a nice enhancement for the canvas and the timeline.

Composite Visualization

Through freehand sketches and a programmer-defined display layout, the canvas can support the conceptualization of program dataflow beyond individual data displays. Participants suggested going further by compositing multiple data displays into a higher-level visualization that could range from simple graphic combinations such as overlaying the skeleton on the color image, to more semantic compositions such as masking certain regions of an image. However, in the meanwhile we should also be cautious to preserve the central role of the program code in general-purpose data processing.

5.7 Discussion

Re-executing the entire program to reprocess recorded input and refresh program data might take significant time to complete when the computation gets complex or the recorded input data gets large. When the program can cleanly divided into multiple components whose dataflow between each other is known, the time for reprocessing can be shortened. The IDE can record input to each component and only reprocess edited components and their subsequent components. Though, this requires the user to explicitly declare components and dataflow between them and significantly changes how the program is coded. Since we wanted to keep the original workflow the programmer is used to, we made a design choice to keep the text-based IDE's usability at the cost of the processing time. While we think the choice was not wrong given the positive feedback from the users, our future work includes an IDE that supports component-based programming.

DejaVu focuses on supporting real-time interactive programs. Note that non-real-time camera applications, where the user sporadically collect camera input to process in an offline fashion (e.g., QR code reader), are more akin to traditional

programming in architecture and workflow, hence do not necessarily require the same special support and are out of the scope of this work.

DejaVu builds on the continuous frame-based update model that reflects the distinctive needs of real-time interactive camera-based programs, and is profoundly different from the conventional discrete step-based debugging model. However, these two models are not necessarily mutually exclusive. Especially when reviewing and reprocessing recorded program data, where there is no concern of interrupting real-time input, we may consider combining these two models to allow stepwise tracing within a frame at statement granularity where needed.

Although DejaVu is a domain-specific IDE for camera-based programs, other types of sensor-based interactions or frame-based programs (e.g., games) may share some of their previously mentioned attributes. DejaVu indeed shares some characteristics with existing sensor-based prototyping tools. It is worthwhile to consider how DejaVu's concepts can be generalized to these other domains.

5.8 Summary of Contributions

In this chapter, we introduced a method of integrating a video-player-style interface into a text-based IDE. The aim was to assign graphical representations to the dynamically changing values *in* and *out* of $out = f(in, c)$, our model of a program with real-world I/O that was introduced in Chapter 3. Whereas photos can represent specific situations in the real world, as discussed in Chapter 4, they cannot represent dynamic data, which changes with time. Here, we show that videos can represent such dynamic data.

The experimental implementation, DejaVu IDE, provides a user interface that continuously updates graphical representations of the input to the program, variable values and window output. This can help the programmer understand what is happening in the program in real time without pausing the execution of the program. The IDE records the program execution automatically and provides a video interface to replay it or jump to an arbitrary time in the recorded session to review what happened in the past. It also allows the programmer to re-execute the program and update the video to catch up with the current contents of the program. We found that the video player concept was received positively by representative target users and may be generalized to support the workflow of development of programs in other domains, such as sensor-based interactions and video games. We also found that there is a demand for more flexible control of video playback and improved visualization, which will be discussed in Chapter 6.

Chapter 6

Graphical Editing to Specify Program Behavior

In the previous two chapters, we investigated how photos and videos help understanding the static and dynamic aspects of the program. These graphical representations represent static data (c), program input (in), variable values and program output (out) in the model ($out = f(in, c)$) of the program with real-world input and output (real-world I/O). While they serve as a visual aid to the programmer, he cannot interact with them to edit the program behavior. He still needs to choose appropriate API from a long list and input its name and their parameters in a text-based code. To see what the API really does, he needs to compile and run the program. This is cumbersome especially when he is new to the API and when he needs to prepare parameters that is difficult to specify in a text format.

Previous approach includes enhancement on code completion for text-based programming languages. Quack [97] allows the programmer to input keywords to populate possible choices of code snippets. It works well for text-based code and parameters, but text is not always the best to specify parameters with visual meanings. There are also research on live programming that provides immediate feedback about the program such as Subtext [37]. Some visual programming languages allow the programmer to specify functions by editing graphics. They go beyond typical box-and-line notations and add more meanings to graphical operations. BitPict [45] allows the user to draw pairs of small input and output bitmaps that represent pixel-rewriting rules. When rules are applied to larger bitmaps, they can carry out meaningful operations such as filling region inside a closed contour. Though, it is not feasible to define complex operations such as computer vision algorithms just by using the rewriting rules.

In this chapter, we discusses how graphical editing of photos and videos can help the programmer implement programs with real-world I/O (f in the model), advancing their use in the previous chapters to have more proactive role in the program development. In particular, we are interested in development of programs that process videos and time-lapse photos taken with a camera with fixed viewpoints to detect interesting events and extract useful information from the real world. First, we introduce the motivation for choosing such application domain and highlight its challenges (Section 6.1). Then, related work is introduced (Section 6.2). Next, VisionSketch IDE is provided as an experimental implementation (Section 6.3). It allows the programmer to annotate input photos and videos graphically to choose appropriate image processing components and setup their parameters. He can build a high-level data flow diagram of components through a visual programming language and seamlessly switch to a text-based editor to update their low-level implementation. A video player-like interface

is always available for controlling the program execution. We also report these implementations (Section 6.4). Next, we showcase example use cases collected from a user study with five programmers (Section 6.5). Finally, we reflect on the previous projects to discuss how graphical editing can be integrated into their experimental implementations (Section 6.6).

6.1 Real-world Event Detection Applications

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 [14] 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 6.5.

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.

6.2 Related Work

In this section, existing attempts to support example-centric programming are first introduced. Relevant visual programming languages are then described. Finally, tools for image processing (including toolkits for programmers and GUI tools for end-users) are introduced.

6.2.1 Tool Support for Example-Centric Programming

Early work on programming by demonstration includes systems for example-centric programming such as Pygmalion [138]. 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 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 behavior 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 [37]. 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 [76] 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 [125] is designed for machine-learning applications and includes user interfaces for collecting, editing, learning, and testing examples. Picode IDE [72] (Chapter 4) is equipped with a text-based editor capable of showing in-line 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 [70] (Chapter 5) 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.

6.2.2 Visual Programming of Image Processing

Many visual-programming languages (VPLs) only visualize the structure of the program (i.e., not its contents.) VisionBlocks [9] aims to allow end users to create their own computer vision programs through GUI operations by a structured editor inspired from Scratch [98]. VIVA [141] 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 [2] 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 [131] provides a spreadsheet interface, whose cell shows an interactive agent that reacts to user input or information from other agents. ConMan [54] allows the user to interact with each visual component and set up parameters for rendering computer graphics (Figure 6.1 left). Its recorder interface is similar to our video-player-like interface in that they both allow the program-

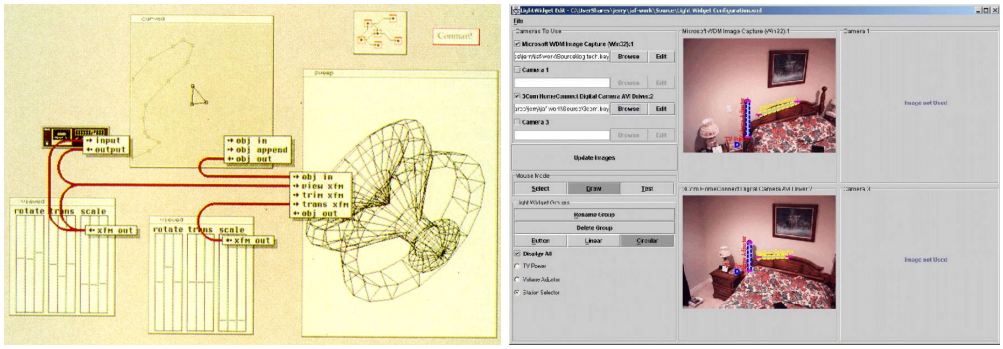


Figure 6.1: Left: ConMan [54] for interactive computer graphics rendering, Right: LightWidgets [40] for end-user computer vision programming.

mer 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.

6.2.3 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 [40] that detects areas of skin in the camera images (Figure 6.1 right). It transforms any visible surface in everyday spaces into an interactive widget controlled by hand gestures. Vision on Tap [29] 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 [41] 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 [106] 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 [6] 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, such as dental imaging, retinal image analysis, and brain- and fat-tissue imaging. It supports all common image manipulations. 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 [14] 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.

6.3 VisionSketch IDE

VisionSketch is an IDE for developing image processing pipelines (Figure 6.2). 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

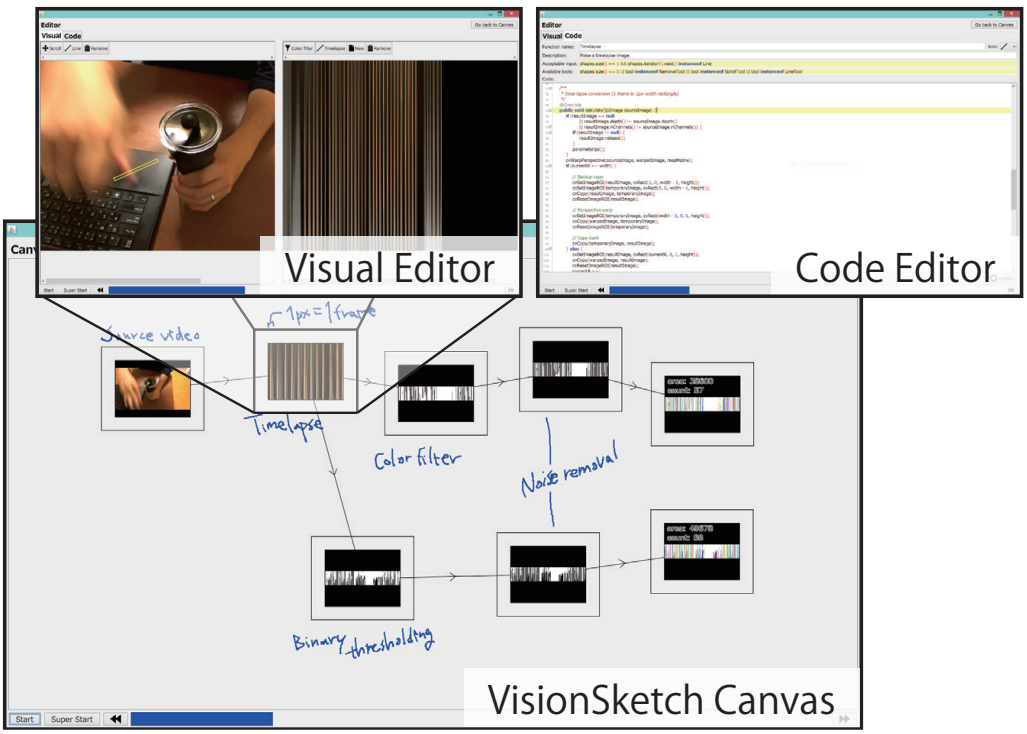


Figure 6.2: Overview of VisionSketch IDE.

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.

6.3.1 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 6.3). It is noteworthy that it is more visual than typical visual programming languages such as VIVA [141] and VisionBlocks [9], whose program structure is visually presented, but data are referenced by text, including filenames 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 [70], it represents a data flow of the program in the same manner as [54] and VIVA [141].

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 drags a line from an existing box to another place to add a new box representing an image processing component. When he clicks an existing box, *visual editor* appears and allows the corresponding component to be edited. He can also draw freeform lines to annotate the program. 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 (Figure 6.4).

Canvas contains a playback interface in its bottom part. It allows flexible

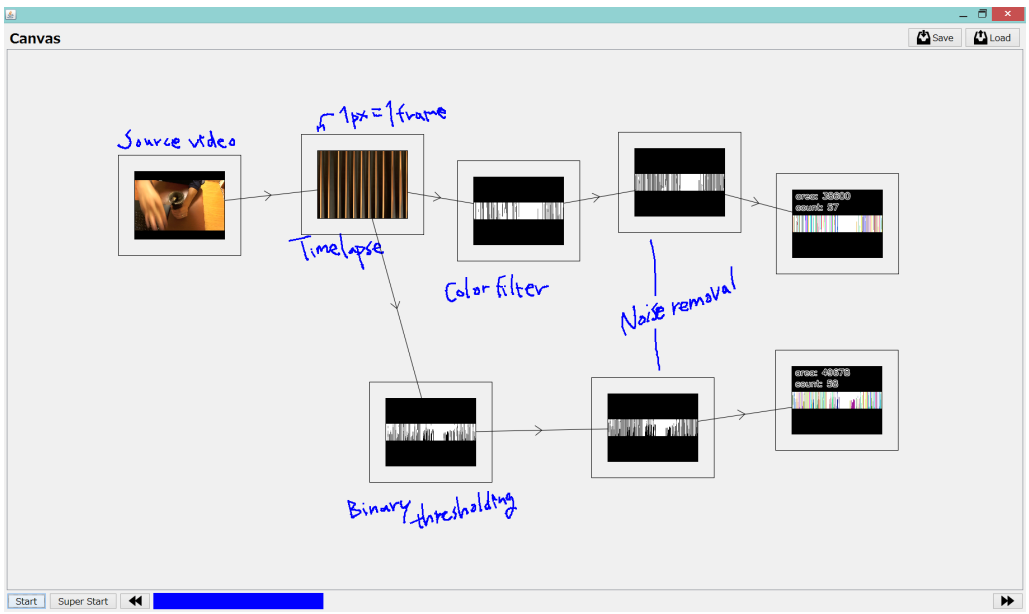


Figure 6.3: VisionSketch canvas showing two algorithms in parallel.

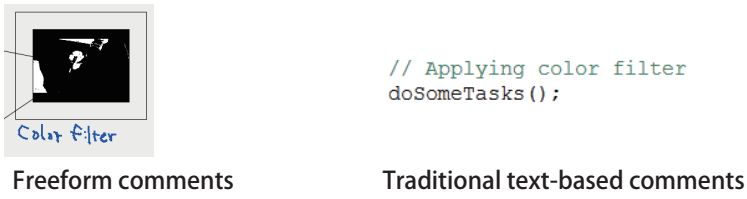


Figure 6.4: VisionSketch code comments.

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.

6.3.2 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 6.5). 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 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 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.

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 (Figure 6.6). 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

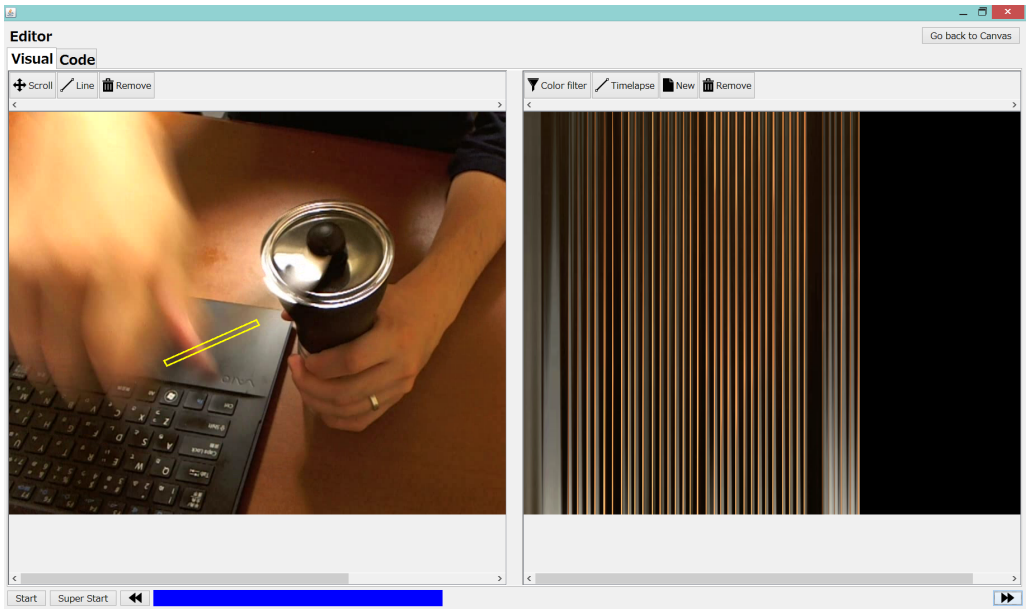


Figure 6.5: VisionSketch visual editor performing time-lapse operation.

Figure 6.6: VisionSketch code completion.

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.

6.3.3 Code Editor

Code editor is the last component used in the programmer's workflow, but it is not the least important (Figure 6.7). 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.

As introduced in Subsection 6.3.2, *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*() ==

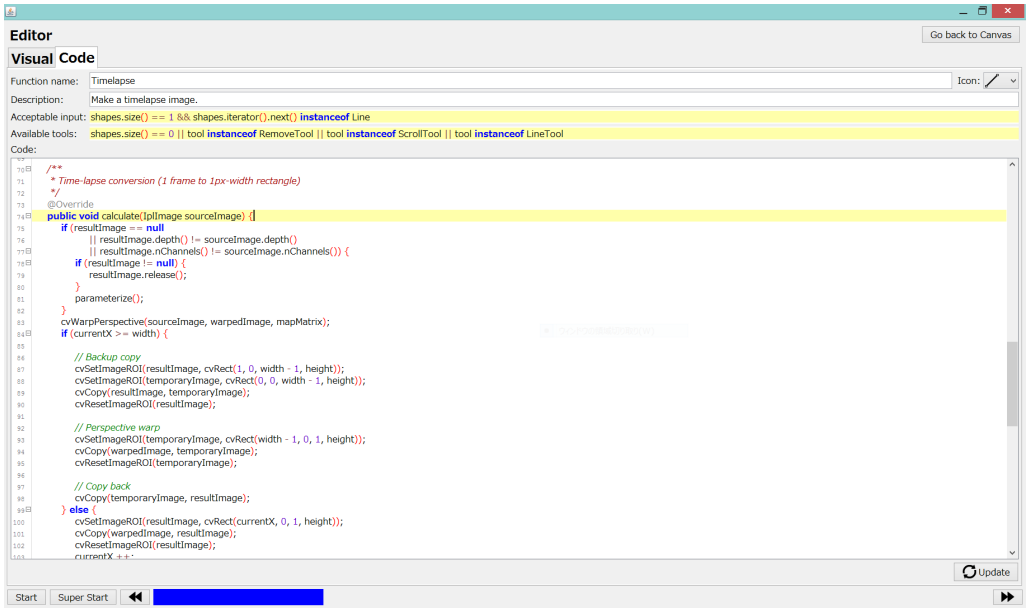


Figure 6.7: VisionSketch text-based code editor for editing image processing algorithm.

$1\&\&shapes.iterator().next()instanceofCircle''$ checking whether the ROI is a circle or not.

When the programmer changes the code, he 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.

6.3.4 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 6.8). 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

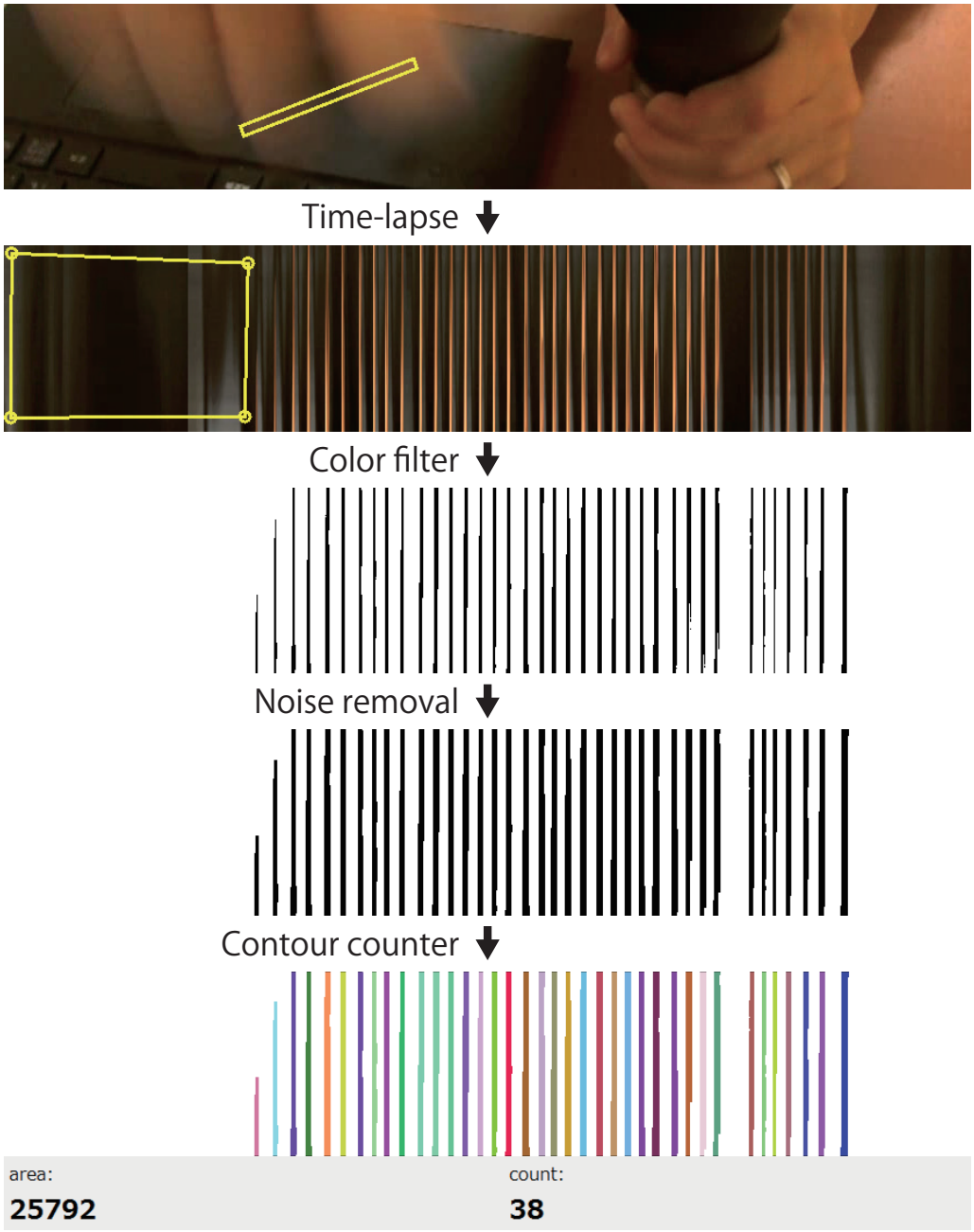


Figure 6.8: The pipeline created in the example use case.

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 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 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.

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.

6.4 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 [69] is helpful for understanding the details.

6.4.1 Overview

VisionSketch runs on a computer that hosts a Java VM and the Java wrapper of the OpenCV [14] 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 6.9). 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.

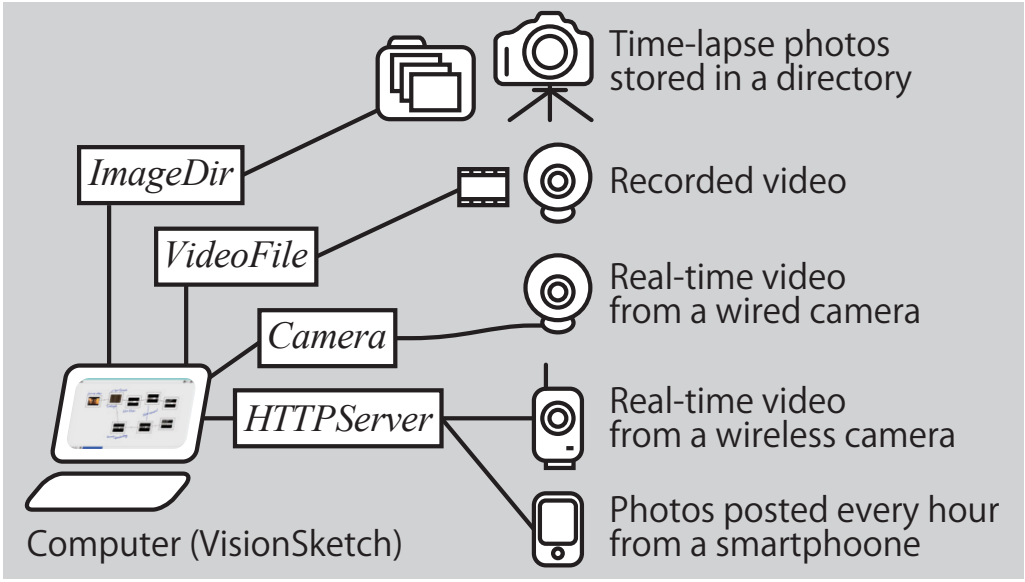


Figure 6.9: Input implementations and supported hardware setup.

In its current implementation, VisionSketch has five predefined image processing components as shown in Figure 6.10, whose details are available online [69].

6.4.2 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 but the Input instance is associated with an image processing component which is an instance of the algorithm-specific class that extends *Function* abstract class. *Function* has a *parameterize*(*parameter*) method where parameter is an instance of *FunctionParameter* class that holds a pair of the ROI and the image. The ROI is a set of shapes, each of which is a *Shape*

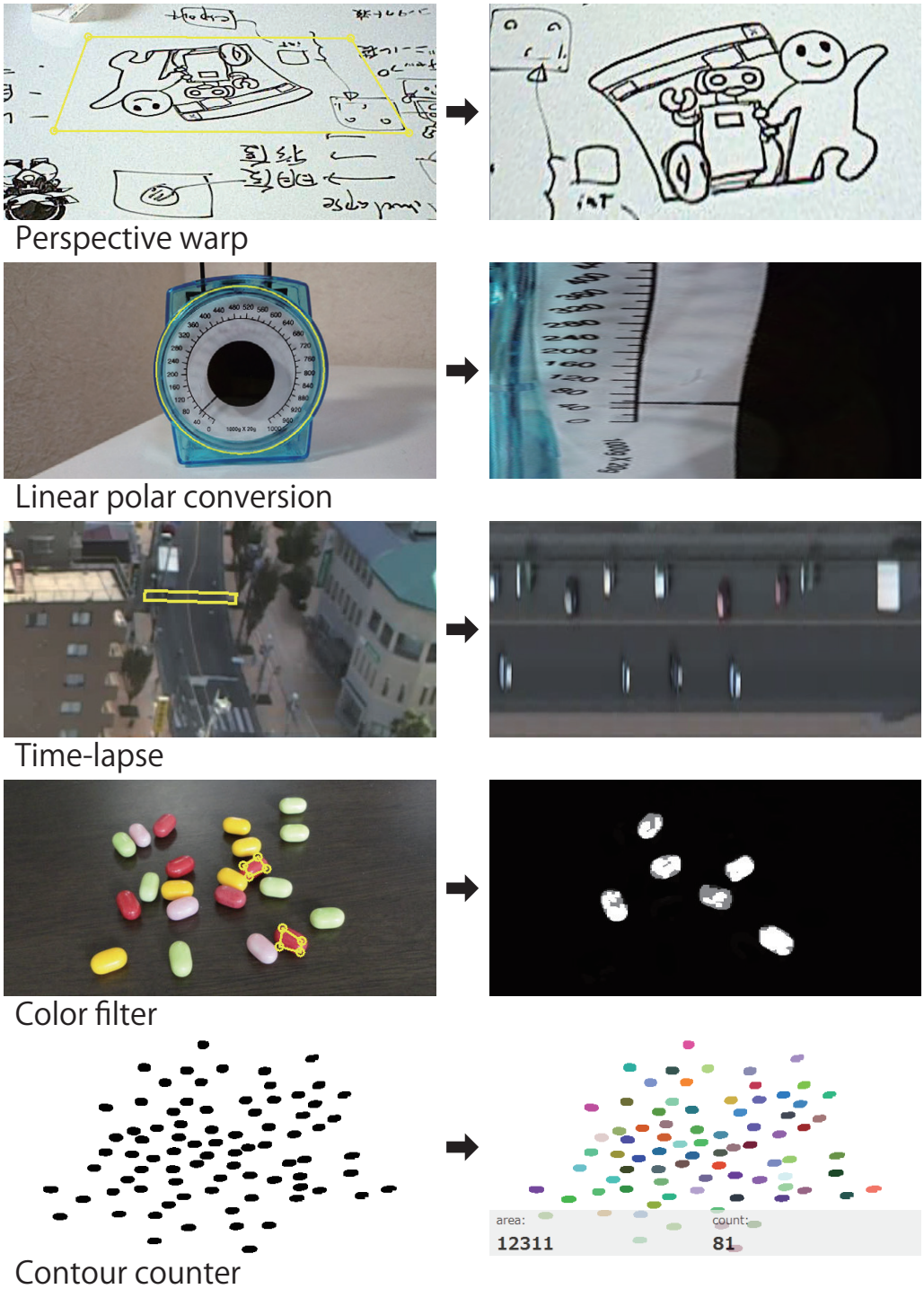


Figure 6.10: VisionSketch predefined operations.

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. *parameterize*(*parameter*) is called once upon the instantiation of the Function class when the component is selected in the *visual editor*. It is also called whenever the programmer edits the shapes and update the ROI. When the parent *Stmt* provides a new input image, *calculate*(*image*) is called to calculate the output. For instance, *ColorFilterFunction* provides a color filter based on the back projection of histogram. Its *parameterize*(*parameter*) method calculates histogram from the ROI of the image and its *calculate*(*image*) method calculates the back projection of the histogram to the current image. As a result, pixels in the current image with similar colors to the ROI of the parameter image is painted in white.

6.4.3 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*(), *getIconFileName*(), and *newInstance*(), 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 uses an open-source code editor, called RSyntaxTextArea[1], which 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 Java-source-code file that implements a FunctionTemplate class and is compiled with the Function implementation by Eclipse Compiler for Java [2].

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 up-

[1] RSyntaxTextArea. `http://fifesoft.com/rsyntaxtextarea`

[2] Eclipse Compiler for Java is included in JDT (Java Development Tools) Core Component. `http://www.eclipse.org/jdt/core`

date 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.

6.5 User Experience

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

6.5.1 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 [14] for the purpose. Their uses of OpenCV library vary from color reduction and beautifying photos to edge detection from a static image; even so, all of these uses concern still images and do not include video processing. 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 6.3.4. 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.

6.5.2 Observations and User Feedback

All participants successfully created their own applications in one to two hours. While three of them used the videos provided, the other two prepared their own videos. They were asked four common questions about each interface after the user study, whose results are listed in Table 6.1, 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 canvas allows me to control and monitor the execution interactively. It is 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*

#	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 6.1: Results of questionnaire.

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 instance, a combination of color filter, time-lapse, and contour-counter components was used for simple image-pattern matching (see Door Watcher introduced later). 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 6.11). 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 three applications.

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 the participant 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 professional 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 corresponding 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,

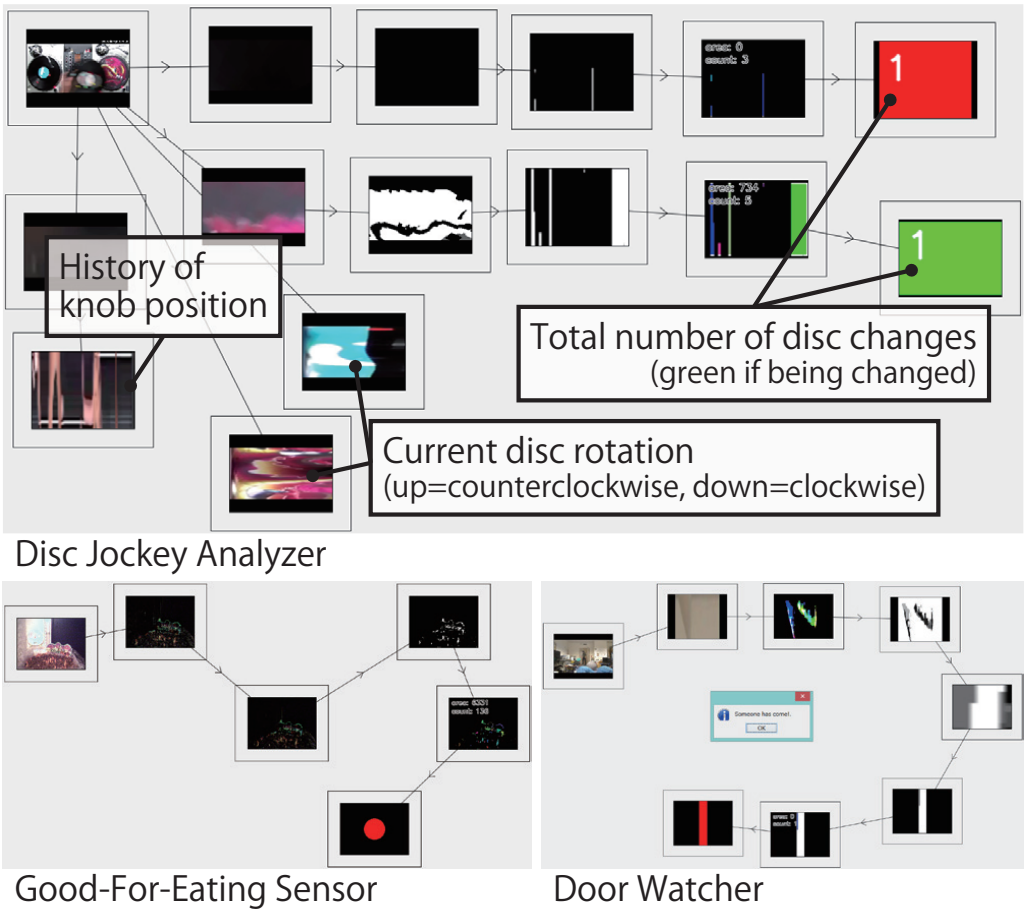


Figure 6.11: Applications developed by the participants.

and time-lapse components. To count the number of discs used in the session, he used perspective 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 increments the number of discs 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 and does not provide audio-related feature, that function is interesting future work.

"Good-for-eating" Sensor

The participant chose a set of time-lapse photos monitoring fungus growth. Photos were taken every hour under a controlled lighting condition. He wanted to create a program for analyzing the newest photo and notifying 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. Next, he implemented a binarization filter that paints pixels

white if they are brighter than the specified threshold; otherwise, they are painted black. To tweak the threshold, he switched seamlessly between code and *visual editor* with help of the "update" button. He found the result a bit noisy, which was derived from the background subtraction. He implemented a median filter and inserted it right after the source video and set up the background subtraction again 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 paints a red circle if the size is less than the threshold; otherwise, it paints a green circle. When he switches the video source to the $HTTPServer$ that receives a new image every hour, the color of the circle tells him whether the current status of 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 [63]. Additionally, he commented critically 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.

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 used a web camera to record real-time video and ask his colleague to go in and out of the room to observe the door in the camera 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 opening door 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 implemented. For instance, a slider interface can be provided for tweaking 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 of VisionSketch. He commented that *canvas* should show the text label for each component as well as the graphical representation. When the pipeline grows longer, mere graphical information gets more confusing since it often looks similar.

6.6 Discussion

In this section, we review the previous projects in the context of graphical editing to reveal how they support or how they can be extended to potentially support the implementation of programs with real-world I/O. First subsection is devoted to the review of Chapter 4 and the second is to that of Chapter 5.

6.6.1 Interaction with Photos

In Chapter 4, we examined the use of a photo for representing static situation in the real world. The experimental implementation, Picode, allows the programmer to take a new photo to create new graphical representation of the new situation. On the other hand, we did not focus on providing a way to interact with the graphical representation to edit the data behind the representation. Though, we have already implemented one way and currently foresee another way of interaction with the graphical representation which replace the operation originally handled by text-based coding. We discuss these ways in comparison to traditional photo taking and editing.

First, the preview window can be used as a user interface to manipulate posture data. It is originally provided to serve as a viewfinder of a camera with which the programmer can preview what is going to be captured. When we think of recent advance in digital cameras, many of them do not have traditional viewfinders but are equipped with touch-enabled displays. Such displays allow the user not only to preview the scene but also to set various kinds of parameters to capture the scene such as the point of focus. The same idea can be applied to our preview window. Instead of manipulating the physical robot, the programmer can manipulate the sliders in the preview window to edit its posture. While manipulating the physical robot is more direct and often easier for the programmer, the slider interface gives more precise and subtle control over the posture data. When we have the shape information of the robot, we can show the three-dimensional computer graphics (3D CG) model over the preview images. Then, it is possible to allow the programmer to touch and move the CG model to edit the posture. This method is investigated in Hashimoto's paper [64]. When we combine these two methods, we can show the slider at each joint location.

Second, the photo can be annotated to filter information. Generally speaking, photos are often post-processed to focus on certain subjects or to enhance their appearance. We can apply the same idea to the photo with posture data. As we discussed as the intrinsic limitation in Chapter 4, the current implementation of Picode use the photo to represent the whole body posture and does not allow the programmer to filter the information. This limitation can be addressed by allowing him to annotate the photo. Since it knows the two-dimensional position of each joint in the photo, Picode IDE can filter the posture data correspondent to the region painted by the programmer. Currently, it only supports human posture whose physical configuration is known and is tracked by the Kinect for Windows SDK. Though, when the shape information of the robot is known and its current configuration can be tracked visually (e.g. with help of fiducial markers), the same interaction can be applied to the photo of the robot.

These interactions with photos add more granularity to control how the graphical representations represent the data. While the photo itself has vagueness in what it represents, interactions with the photo allows the programmer to tell his intention to the development environment.

6.6.2 Interaction with Videos

In Chapter 5, we examined the use of a video for representing dynamic behavior of the program in the real world. The experimental implementation, DejaVu, allows the programmer to run the program to record new video strips. It also allows to re-execute the program with recorded input data to update the video strips to catch up with changes in the program. The chapter was mainly devoted

to explain how we can provide lively view of the behavior of the program and there was very little discussion about editing the video strips. Here, we discuss how the programmer can edit the video. One way is already implemented and the other is taken from the comments of the programmer participated in the user study.

First, the programmer can edit the video in-point and out-point as explained in Chapter 5. When he is reviewing existing sessions, he can split any session at the frame of interest into two sessions. He can also duplicate the session so that he can keep the original session. In this way, he can emit unused part of the video and speed up the re-execution. In the user study, one programmer suggested that, in addition to splitting, we should also allow merging two sessions into one new session. Unlike splitting, there might be a problem of discontinuity between the two sessions. For instance, the distance of the user from the camera might be very different between the sessions. Such jump of values usually does not occur in the real time session. Therefore, merging can result in unexpected results. Though, the programmer argued that he would only merge two sessions which he knows can be merged without causing such problems. While we did not implement the merging feature since we wanted to avoid a chance for the discontinuity problem, his opinion is reasonable when we compare the operation with general video editing. In video editing, people are responsible for judging whether merging makes sense or not. Currently, we think we can add support for merging. We can also add support for inserting an existing session into the specific frame of another session.

Second, another programmer in the user study commented that he wants to take more control over how the program is executed. He was not satisfied with the capability of editing the in-point and out-point. He wanted to edit input data and to use it as input to the program rather than using raw data recorded by the camera. In response to his demand, we can allow the programmer to edit existing input data by providing special API and import the edited data into existing sessions. Beside this programmatic solution, there are several potential improvements on interactions with the video strips. For instance, the ways of interactions with photos can also be applied to the videos.

These interactions with videos allows more control over how the program is executed. While the conventional development environments only support the live execution of the program and analysis of recorded data, the metaphor of video strips allows smooth transitions between the execution and analysis.

6.7 Summary of Contributions

In this chapter, we introduced a method of making use of graphical editing to help implementation of programs with real-world I/O. Each function f in the program ($out = f(in, c)$) has its own graphical representation and can be edited to choose appropriate implementation and modify its parameters. We also reviewed the previous projects in terms of graphical editing.

The experimental implementation; i.e., VisionSketch, provides three interfaces: *canvas* for visual programming, *code editor* for text-based programming, and *visual editor* for bridging these two modalities to facilitate example-centric programming of image processing applications. Graphical representations of concrete examples in *canvas* and *visual editor* help the programmer to understand the programs. Graphical operations such as drawing shapes on the input image help him/her to choose and tune image processing components. The text-based

code editor is still needed to implement new components and needs interactive GUI support.

Chapter 7

Conclusions and Outlook

In this chapter, we review our attempts to make use of graphical representations of the real world in the integrated development environments. Based on the summary of contributions, we also discuss future work in the research of integrated development environments.

7.1 Summary of Contributions

This dissertation discussed use of integrated graphical representation for development of programs with real-world input and output (I/O). The development of such programs inherently involves retrieval of the real world I/O data, which we call "Programming with Example." It has been done in various forms such as unit testing and machine learning. The scope of this dissertation is in its previously unexplored subset which can be effectively addressed by introducing graphical representations of the real world.

Programming by example is similar to programming with example in that they both utilize examples, but its target user is the end-user without prior knowledge of programming. As a result, it does not allow precise specification of program logics. On the other hand, existing integrated development environments (IDEs) do no support such workflow. They are equipped with text-based user interfaces which cannot present the example data intuitively.

Integrated graphical representations provide explicit support for programming with example data, allowing the programmer to take advantages of both approaches: power of concrete examples and expressivity of text-based programming. Specifically, we proposed a model of the programs with real-world I/O and assigned graphical representations to each of the components:

$$out = f(in, c)$$

In this model, c corresponds to constants and f to functions while in and out to variables.

First, we discussed an assignment of photos to constants (Chapter 4). With an experimental implementation of *Picode* IDE [72], we confirmed that constant data denoting a specific situation in the real world can be represented well by photos. Not only the programmer but also the end-user is familiar with taking photos, which is accepted naturally as an action to retrieve example data. Photos are found to be capable of containing various kinds of information regardless of their representing data, such as environmental information, nonverbal information and emotion. However, there are also intrinsic limitations of photos. They are not suitable for precisely distinguishing posture information. They might show a variety of objects and the focus of the information can be unclear.

These observations imply directions for future work such as use of multiple photos from different viewpoints to prevent occlusion, use of multiple photos of different situations to represent semantic information, annotations on photos to filter information, or use of other media in text-based IDEs.

Second, we discussed an assignment of videos to variables (Chapter 5). With an experimental implementation *DejaVu* IDE [70], we confirmed that variable data denoting changes of the situation along time in the real world can be represented well by videos. It could address the gulf of execution and evaluation between the static source code and dynamic behavior of the program. We decomposed the dynamic behavior of the program into spatial and temporal aspects and designed correspondent two interfaces, which were confirmed to work well. First, we provided a special debugger with the video player metaphor which allows the programmer to control the flow of time in the recorded execution. Second, this video player interface is interlinked with a two-dimensional space with the canvas metaphor which allows the programmer to overview the current state of the execution. These interfaces allow the programmer to choose which part of the program to be visualized. When the source code is updated, they are updated semi-automatically by re-executing the program with the recorded input. This scheme keeps the link between the static and dynamic representations of the program synchronized.

Third, we discussed how graphical representations can be manipulated and edited to support the implementation of functions (Chapter 6). While the previous two attempts focused on providing visual aid to the programmer, this is an attempt to provide more proactive means for the programmer to implement programs. An experimental implementation *VisionSketch* IDE provides a direct manipulation interface with which the programmer can annotate the input video or photo. Then, the annotation is passed to the program component as an argument, which allows the programmer to write code that use the annotated information. The iteration between the annotation and writing code allows fast prototyping of image processing programs. For instance, he first annotate the rectangle area in the photo. Next, he switch to the code editor and write code to apply a bipolar conversion. Then, he can immediately see its result and update annotation if needed.

Throughout these investigations on graphical representations, we have revealed many advantages and some disadvantages. We discussed a method for using visual media such as photos and video to express real world input and output data. Photos and videos have benefits not found in existing text-based code. First, we found that photos can clearly show real world situations and express not only information about the human or robot subject, but also additional information such as environmental information or human emotions. Next, we found that videos can display a changing situation over time, and that using an interface such as a video player could intuitively express how the program works in the real world. In addition, we found that we can interact with visual media by highlighting certain areas of a photo or extracting portions of videos in place of certain programming tasks. This benefit could be an important technology applicable to programs that involve a wider array of input and output with the real world. For example, photos of foods associated with certain flavors or surfaces with certain textures could be used to represent sensory information.

On the other hand, it also became clear that such visual media include drawbacks not found in text-based programming. First, in order to allow expression of various data, visual media must allow for multiple interpretations by the viewer, and it is sometimes unclear what is being expressed. Viewpoint and viewing angle

cannot be changed once a photo is taken, and it is impossible to fully update the expression without retaking the photo. Moreover, because this media concretely expresses situations in the real world, it is difficult to generalize for situations in which multiple cases are shown at the same time. The visual media dealt with in this study includes positive and negative aspects that are exactly opposite of existing text-based programming. We've shown that just as images and tables are used to supplement text in a paper, visual media could be used to supplement text-based code to support effective programming with benefits greater than that of either media used alone.

Please note that our approach has a certain limitation that comes from the distinctive workflow of PwE. Our approach is an attempt to address the difficulty of moving target by fixing (recording and using) one of the program elements. First, when c is fixed (Chapter 4), it is usually used to compare the current situation against previous one or to reproduce a certain situation. This approach cannot be applied to the development of programs that automatically generates example data e.g. computing optimal posture for a robot. Second, when *in* is fixed (Chapter 5), it ensures the reproducibility of *out*. This approach does not work for the development of programs with real-time feedback loops where *out* affects *in*. Finally, when f is fixed (Chapter 6), the programmer can interactively manipulate its graphical representation to setup the following component f', incrementally building the program. This approach eases setting up program components by graphical editing, but cannot be applied to building complex programs whose components work as autonomous agents and affecting each other. As we reviewed here, all of the experimental implementations shares the limitation. That is, since graphical representations are 100% concrete, it is difficult to make use of them to represent more abstract concepts.

It is also important to distinguish this work from traditional software engineering research that aims to implement secure programs, including meta programming, program verification and other kinds of program analysis. They are particularly useful when one wants to ensure the soundness of the program. On the other hand, our work focuses on quickly iterating the cycle of coding and debugging to investigate and improve the quality of interactive programs. Such style of programming is useful for lightweight software development process e.g. prototyping process and agile process. Within such a process, an initial step with an open-ended goal is sometimes called exploratory programming as introduced in Subsection 2.3.3, where the programmer does not see the final specification of the program. This work focuses on the use of graphical representations to support understanding and implementing programs, aiding exploratory programming.

7.2 Future Outlook

In this section, we discuss future research that was shown to be necessary in this study.

7.2.1 3D Graphical Representations

Graphical representations used in this dissertation were all two-dimensional (2D), which led to some problems such as occlusions where the subject of the photo is hidden by other objects in the foreground. In recent years, methods for capturing situations in three-dimensional (3D) using tools such as the Microsoft Kinect and other depth sensing cameras have been established. By adding 3D information to photos and videos, we can expect to solve these restrictions.

Development environments that include 3D visual media could be used to create a range of applications even wider than those discussed in this dissertation. Example domains of applications include robots that move around in our living environments, or augmented reality that uses image processing. Toolkits such as the Robot Operating System, Phybots and OpenCV have already been developed, but by providing integrated support for 3D visual media, application development could be further simplified, leading to the possibility of a growing number of developers or applications.

On the other hand, using 3D would elaborate new issues. The visual media proposed in this dissertation could all be used in a 2D GUI without the use of special controls. However, 3D data rendered in 3D would require a user interface to control viewpoint and field of view, which could make it more complicated. Solving this problem will require drawing upon a variety of research in the field of 3D CG such as locating optimal viewpoints [140].

7.2.2 Multi-modal and Cross-modal Programming

In this dissertation, we only dealt with a subset of real world I/O data that can be expressed intuitively by graphical representations. In a text-based IDE, source code is all presented visually through pixels on a display. Thus, graphical representations could be smoothly integrated into such environment to supplement program development. On the other hand, programs that invoke other senses are getting practical. For instance, development of programs that handle sound already has a long history. Displays that create position-based haptic feedback and interfaces that electrically induce tastes have been proposed. Though, it remains as an open question how to present such nonvisual real world I/O data to the programmer within the IDE.

It is not necessary to present all real-world I/O data with graphics. We foresee that it will be feasible to create development environments optimized for each of the sensory organs when writing programs related to the senses. For instance, the sense of touch is most intuitively presented to the programmer as haptic information. If the development environment can directly present sensations, the programmer can use the full array of senses in program development to realize multi-modal programming that allows direct debugging of human senses. This would open chances of applying multi-modal interfaces to IDEs as cultivated in Human-Computer Interaction research that has heretofore been focused on the end user. For example, touch information can be presented as a button that can play the original haptic feedback using haptic displays.

It might be more intuitive and convenient when the development environments can present the real-world I/O data with help of different sense modalities. Audio information can be displayed as an audio visualization that plays the original sound when tapped. The visualization might help the programmer to get a brief sense of what the audio is about without really playing it. Colors associated synesthetically with certain sounds could also be used to represent them. Photos that recall certain sense information could be also useful, as discussed in Chapter 4. For example, photos of flowers could be used to express the scent of flowers. Photos of object surfaces could be used to express their textures, playing their original haptic feedback when touched. We need to investigate these kinds of cross-modal interactions to integrate them into IDEs. In particular, presenting sensory data with help of the relevant graphical information seems to be a promising way for two reasons. First, current IDEs are all implemented in the graphical user interfaces and therefore it would be as easy as this dissertation describes to

integrate graphical representations into IDEs. Second, graphical information can be sensed immediately compared to other senses which take some time.

7.2.3 Everyone as a Programmer

In this dissertation, we proposed to use graphical representations in text-based IDEs to improve the productivity of programmers. However, we have also confirmed that end users can also operate visual programming languages found in VisionSketch or the photo capturing in Picode. These examples show that parts comprehended and operated by end users and those used by programmers are sharply divided. The former are parts using visual media, while the latter are parts that require text-based programming. Thus, we expect that it is possible to divide work between programmers and non-programmers. This would be similar to how UI builders divided the tasks of programmers and UI designers. We can see two directions for future work in this research, and both would contribute to the future utilization of computers by end users.

One direction would focus on end user programming that would allow end users to customize the operations of programs without the need for prior knowledge about programming. Research on end user programming focuses on letting users generate programs to achieve objectives without the need to learn about programming. In this type of system, the details of the implementation such as text-based source code are concealed from the user. Instead, the program is designed to present high level APIs, explanations, and sample input and output that allow the user to guess at and understand the processing contents of the program. In previous research, architectures such as mashup and On X [109] have been proposed in which programmers perform text-based programming while end users filled in gaps in the program. Filling in gaps has been used for mail addresses and website URLs, but by using visual media, we can create new applications such as image recognition which users can intuitively customize. VisionBlocks [9] is a pioneering example of this.

On the other hand, the graphical representations in this dissertation are integrated into the text-based programming, and even if we divide tasks between the end user and the programmer, the end user is still working very close to the text-based source code. In a workshop for elementary school students (Subsection 4.5.2), we found that the visual media formed a starting point for students to become interested in text-based programming. In other words, the approach in this paper could be valid for educational uses, such as in studying text-based programming. Because visual media includes environmental information and emotions, it allows one to understand what someone was aiming for and how one developed a program when viewing someone else's source code. Understanding the motivation and development process of another person makes users want to try to create their own program. Thus, the result of this paper is not to eliminate the need for the technology required for programming, but to support the motivation for learning that technology. This can be used to make computers even more useful. The increase in digital natives and the permeation of computers in daily life follow this trend. It may become possible for anyone to edit the source code of the programs they have on hand, and for all users to use programs that they themselves customize. This is also suitable as an educational tool for the educational method known as computational thinking, which cultivates logical thinking through learning about the mechanisms of a computer.

7.2.4 Live Programming with Live Feeling

Live programming is an emerging research field to fill the gulf of execution and evaluation in programming as introduced in Subsection 2.3.3. It is an attempt to address the gulf between the static source code and its dynamic behavior at runtime. This dissertation is an attempt to address the same gulf but even deepened by the difference between the development environment (the computer) and the runtime environment (the real world). This subsection discusses to extend the scope of live programming analogous to the extension of the gulf.

Before discussing the scope of live programming, we review our projects that use integrated graphical representations in the context of live programming. While there is not yet precise definition of live programming, existing work has commonly tried providing *live feedback* (immediate and continuous feedback) about the program's runtime behavior during its development process. Among the projects introduced in this dissertation, DejaVu (Chapter 5) and VisionSketch (Chapter 6) provide such *live feedback*. When the program code is updated, DejaVu can reprocess the recorded input to update the program output shown in its Timeline and Canvas interface. In this way, the programmer can keep informed of the program's runtime behavior. VisionSketch eliminates explicit operation of compiling text-based source code. When the programmer updates the text-based definition of an image processing component and goes back to its graphical view, the code is silently compiled and the component is updated. Then, he can immediately see the updated processing result for the current frame. Such edit-triggered updates are pretty common in conventional live programming environments such as VIVA [141].

Then, how about Picode introduced in Chapter 4? It does not provide *live feedback*. Though, its inline photos representing posture data help the programmer understand the situation in the real world and imagine what will happen when the code is executed. It provides *live feeling* of the real world, which is the runtime environment for the Picode development environment.

While it does not fit in the definition of conventional live programming, it surely fills the mental gap between the code and its execution. We think that *live feeling* does not always come from *live feedback*, which is just one way of providing live programming experience. While conventional live programming tends to be an effort to technically merge the coding and execution of the program, we propose to define live programming as an effort to fill the mental gap between the development environment and runtime environment. It is an extended definition and includes the case of Picode. We foresee that future direction of live programming research is not limited to technical contributions but also include more contributions of Human-Computer Interaction, investigating how the development environment can provide *live feeling* of the runtime environment.

References

[1] Arduino. `http://www.arduino.cc/`. Accessed September 1, 2013.

[2] Matlab/simulink. `http://www.mathworks.com/products/simulink/`. Accessed September 1, 2013.

[3] Max/msp. `http://cycling74.com/`. Accessed September 1, 2013.

[4] Processing. `http://processing.org/`. Accessed September 1, 2013.

[5] Sharpdevelop. `http://www.icsharpcode.net/opensource/sd/`. Accessed September 1, 2013.

[6] Michael D. Abràmoff, Paulo J. Magalhães, and Sunanda J. Ram. Image processing with imagej. *Biophotonics international*, 11(7):36–42, 2004.

[7] Robert M. Balzer. Exdams: extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM.

[8] David R. Barstow, Howard E. Shrobe, Erik Sandewall, and Stephen W. Smoliar. *Interactive programming environments*, volume 9. ACM, New York, NY, USA, July 1984.

[9] Abhijit Bendale, Kevin Chiu, Kshitij Marwah, and Ramesh Raskar. Visionblocks: A social computer vision framework. In *Proceedings of the 2011 IEEE International Conference on Social Computing*, pages 521–526, 2011.

[10] Alan F. Blackwell. Metacognitive theories of visual programming: What do we think we are doing? In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pages 240–, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Alan F. Blackwell. Pictorial representation and metaphor in visual language design. *Journal of Visual Languages and Computing*, 12(3):223 – 252, 2001.

[12] Alan F. Blackwell and Thomas Green. Notational systems–the cognitive dimensions of notations framework. *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science.*, 2003.

[13] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM Symposium on User Interface Software and Technology*, UIST '05, pages 163–172, New York, NY, USA, 2005. ACM.

[14] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[15] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM.

[16] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.

[17] Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. Ide 2.0: collective intelligence in software development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 53–58, New York, NY, USA, 2010. ACM.

[18] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, New York, NY, USA, 2013. ACM.

[19] Margaret M. Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11:155–206, 3 2001.

[20] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, March 1995.

[21] Margaret M. Burnett, Curtis Cook, and Gregg Rothermel. End-user software engineering. *Communications of the ACM*, 47(9):53–58, September 2004.

[22] Margaret M. Burnett and Herkimer J. Gottfried. Graphical definitions: expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):1–33, March 1998.

[23] William A. S. Buxton, M. R. Lamb, D. Sherman, and Kenneth Carless Smith. Towards a comprehensive user interface management system. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '83, pages 35–42, New York, NY, USA, 1983. ACM.

[24] Sylvain Calinon. Robot programming by demonstration. In *Springer Handbook of Robotics*, pages 1371–1394. Springer, 2008.

[25] Xiang Cao and Ravin Balakrishnan. Visionwand: interaction techniques for large displays using a passive wand tracked in 3d. In *Proceedings of the 16th annual ACM Symposium on User Interface Software and Technology*, UIST '03, pages 173–182, New York, NY, USA, 2003. ACM.

[26] Luca Cardelli. Building user interfaces by direct manipulation. In *Proceedings of the 1st annual ACM SIGGRAPH Symposium on User Interface Software*, UIST '88, pages 152–166, New York, NY, USA, 1988. ACM.

[27] Timothy Cardenas, Marcello Bastea-Forte, Antonio Ricciardi, Bjoern Hartmann, and Scott R. Klemmer. Testing physical computing prototypes through time-shifted & simulated input traces, 2008.

[28] Ed Huai-Hsin Chi, Phillip Barry, John Riedl, and Joseph Konstan. A spreadsheet approach to information visualization. In *Proceedings of the 1997 IEEE Symposium on Information Visualization*, pages 17–24, 1997.

[29] Kevin Chiu and Ramesh Raskar. Computer vision on tap. In *Proceedings of 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, CVPR Workshops 2009, pages 31–38, 2009.

[30] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 486–493, New York, NY, USA, 2000. ACM.

[31] Alan Cooper. Why i am called "the father of visual basic". `http://www.cooper.com/alan/father_of_vb.html`. Accessed September 1, 2013.

[32] Allen Cypher. Watch what i do. chapter Eager: programming repetitive tasks by demonstration, pages 205–217. MIT Press, Cambridge, MA, USA, 1993.

[33] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[34] Scott Davidoff, Nicolas Villar, Alex S. Taylor, and Shahram Izadi. Mechanical hijacking: how robots can accelerate ubicomp deployments. In *Proceedings of the 13th International Conference on Ubiquitous Computing*, UbiComp '11, pages 267–270, New York, NY, USA, 2011. ACM.

[35] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: industrial experience with the code bubbles paradigm. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press.

[36] Rob Diaz-Marino and Saul Greenberg. Cambience: A video-driven sonic ecology for media spaces. In *Video Proceedings of ACM CSCW'06 Conference on Computer Supported Cooperative Work*, 2006.

[37] Jonathan Edwards. Example centric programming. *ACM SIGPLAN Notices*, 39(12):84–91, December 2004.

[38] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 505–518, New York, NY, USA, 2005. ACM.

[39] Martin Erwig and Bernd Meyer. Heterogeneous visual languages-integrating visual and textual programming. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, pages 318–325, 1995.

[40] Jerry Alan Fails and Dan Olsen. Light widgets: Interacting in every-day spaces. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, IUI '02, pages 63–69, New York, NY, USA, 2002. ACM.

[41] Jerry Alan Fails and Dan Olsen. A design tool for camera-based interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 449–456, New York, NY, USA, 2003. ACM.

[42] The Eclipse Foundation. Eclipse. `http://eclipse.org/`. Accessed September 1, 2013.

[43] The Eclipse Foundation. Eclipse code recommenders. `http://eclipse.org/recommenders`. Accessed September 1, 2013.

[44] Jun Fujima, Aran Lunzer, Kasper Hornbæk, and Yuzuru Tanaka. Clip, connect, clone: Combining application elements to build custom interfaces for information access. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04, pages 175–184, New York, NY, USA, 2004. ACM.

[45] George W. Furnas. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 71–78, New York, NY, USA, 1991. ACM.

[46] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[47] Patrick Girard. Your wish is my command: programming by example. chapter Bringing Programming by Demonstration to CAD Users, pages 135–162. Morgan Kaufmann, 2001.

[48] Max Goldman, Greg Little, and Robert C. Miller. Collabode: collaborative coding in the browser. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 65–68, New York, NY, USA, 2011. ACM.

[49] Max Goldman and Robert C. Miller. Codetrail: Connecting source code and web resources. *Journal of Visual Languages and Computing*, 20(4):223–235, August 2009.

[50] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A ' cognitive dimensions ' framework. *Journal of Visual Languages and Computing*, 7(2):131 – 174, 1996.

[51] Saul Greenberg and Chester Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 209–218, New York, NY, USA, 2001. ACM.

[52] The LEGO Group. Mindstorms nxt. `http://mindstorms.lego.com/`. Accessed September 1, 2013.

[53] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[54] Paul E. Haeberli. Conman: A visual programming language for interactive graphics. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 103–111, New York, NY, USA, 1988. ACM.

[55] Gregory D. Hager and Kentaro Toyama. X vision: A portable substrate for real-time vision applications. *Computer Vision and Image Understanding*, 69(1):23 – 37, 1998.

[56] Daniel Conrad Halbert. *Programming by example*. PhD thesis, University of California, Berkeley, 1984.

[57] Christopher Michael Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[58] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 317–328, New York, NY, USA, 2011. ACM.

[59] Björn Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 145–154, New York, NY, USA, 2007. ACM.

[60] Björn Hartmann and Mark Dhillon. Hypersource: Bridging the gap between source and code-related web sites. In *Adjunct Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 421–422, New York, NY, USA, 2010. ACM.

[61] Björn Hartmann, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 299–308, New York, NY, USA, 2006. ACM.

[62] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[63] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM Symposium on User Interface Software and Technology*, UIST '08, pages 91–100, New York, NY, USA, 2008. ACM.

[64] Sunao Hashimoto, Akihiko Ishida, Masahiko Inami, and Takeo Igarashi. Touchme: An augmented reality based remote robot manipulation. In *Proceedings of the 21st International Conference on Artificial Reality and Telexistence*, ICAT '11, pages 28–30, 2011.

[65] D. Austin Henderson, Jr. The trillium user interface design environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, pages 221–227, New York, NY, USA, 1986. ACM.

[66] C. A. R. Hoare. Hints on programming language design. In Ellis Horowitz, editor, *Programming Languages*, pages 31–40. Springer Berlin Heidelberg, 1983.

[67] Takeo Igarashi. *Freeform user interfaces for graphical computing.* PhD thesis, 1999.

[68] National Instruments. Labview. `http://www.ni.com/labview/`. Accessed September 1, 2013.

[69] Jun Kato. Visionsketch. `http://junkato.jp/visionsketch/`. Accessed December 1, 2013.

[70] Jun Kato, Sean McDirmid, and Xiang Cao. Dejavu: integrated support for developing interactive camera-based programs. In *Proceedings of the 25th annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 189–196, New York, NY, USA, 2012. ACM.

[71] Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. Phybots: a toolkit for making robotic things. In *Proceedings of the Designing Interactive Systems Conference*, DIS '12, pages 248–257, New York, NY, USA, 2012. ACM.

[72] Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. Picode: inline photos representing posture data in source code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3097–3100, New York, NY, USA, 2013. ACM.

[73] Scott R. Klemmer and James A. Landay. Toolkit support for integrating physical and digital interactions. *Human-Computer Interaction*, 24(3):315–366, 2009.

[74] Scott R. Klemmer, Jack Li, James Lin, and James A. Landay. Papier-mache: toolkit support for tangible input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 399–406, New York, NY, USA, 2004. ACM.

[75] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad A. Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):21:1–21:44, April 2011.

[76] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '05, pages 1557–1560, New York, NY, USA, 2005. ACM.

[77] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[78] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16(1):41–84, 2005.

[79] Andrew J. Ko and Brad A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 387–396, New York, NY, USA, 2006. ACM.

[80] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.

[81] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 199–206, 2004.

[82] John R. Koza, Forrest H. Bennett, III, and Oscar Stiffelman. Genetic programming as a darwinian invention machine. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming*, volume 1598 of *Lecture Notes in Computer Science*, pages 93–108. Springer Berlin Heidelberg, 1999.

[83] David Kurlander. Watch what i do. chapter History of Editable Graphical Histories, pages 405–413. MIT Press, Cambridge, MA, USA, 1993.

[84] David Kurlander and Eric A. Bier. Graphical search and replace. *ACM SIGGRAPH Computer Graphics*, 22(4):113–120, June 1988.

[85] David Kurlander and Steven Feiner. Interactive constraint-based search and replace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 609–618, New York, NY, USA, 1992. ACM.

[86] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics (TOG)*, 12(4):277–304, October 1993.

[87] Yang Li and James A. Landay. Activity-based prototyping of ubicomp applications for long-lived, everyday human activities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1303–1312, New York, NY, USA, 2008. ACM.

[88] Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. Seess: Seeing what i broke – visualizing change impact of cascading style sheets (css). In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 353–356, New York, NY, USA, 2013. ACM.

[89] Henry Lieberman. Dominoes and storyboards beyond 'icons on strings'. In *Proceedings of 1992 IEEE Workshop on Visual Languages*, pages 65–71, 1992.

[90] Henry Lieberman. Watch what i do. chapter Mondrian: a teachable graphical editor, pages 341–358. MIT Press, Cambridge, MA, USA, 1993.

[91] Henry Lieberman. Watch what i do. chapter Tinker: a programming by demonstration system for beginning programmers, pages 49–64. MIT Press, Cambridge, MA, USA, 1993.

[92] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special issue on the debugging scandal). *Communications of the ACM*, 40(4):26–29, 1997.

[93] Henry Lieberman. *Your wish is my command: programming by example*. Morgan Kaufmann series in interactive technologies. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[94] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[95] James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. Denim: finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 510–517, New York, NY, USA, 2000. ACM.

[96] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. End-user programming of mashups with vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces*, IUI '09, pages 97–106, New York, NY, USA, 2009. ACM.

[97] Greg Little and Robert C. Miller. Keyword programming in java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 84–93, New York, NY, USA, 2007. ACM.

[98] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16:1–16:15, November 2010.

[99] John H. Maloney and Randall B. Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 21–28, New York, NY, USA, 1995. ACM.

[100] Jennifer Mankoff, Scott E. Hudson, and Gregory D. Abowd. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proceedings of the 13th annual ACM Symposium on User Interface Software and Technology*, UIST '00, pages 11–20, New York, NY, USA, 2000. ACM.

[101] Jennifer Mankoff, Scott E. Hudson, and Gregory D. Abowd. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces.

In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 368–375, New York, NY, USA, 2000. ACM.

[102] Toshiyuki Masui. An efficient text input method for pen-based computers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '98, pages 328–335, New York, NY, USA, 1998. ACM Press/Addison-Wesley Publishing Co.

[103] Toshiyuki Masui and Ken Nakayama. Repeat and predict—two keys to efficient text editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pages 118–130, New York, NY, USA, 1994. ACM.

[104] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 127–136, New York, NY, USA, 1989. ACM.

[105] Dan Maynes-Aminzade, Randy Pausch, and Steve Seitz. Techniques for interactive audience participation. In *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces*, ICMI '02, pages 15–, Washington, DC, USA, 2002. IEEE Computer Society.

[106] Dan Maynes-Aminzade, Terry Winograd, and Takeo Igarashi. Eyepatch: prototyping camera-based interaction through examples. In *Proceedings of the 20th annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 33–42, New York, NY, USA, 2007. ACM.

[107] Sean McDirmid. Living it up with a live programming language. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638, New York, NY, USA, 2007. ACM.

[108] Sean McDirmid. Usable live programming (to appear). In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '13, New York, NY, USA, 2013. ACM.

[109] Microsoft. on x. `http://www.onx.ms/`. Accessed September 1, 2013.

[110] Microsoft. Roslyn. `http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx`. Accessed September 1, 2013.

[111] Microsoft. Visual studio. `http://www.microsoft.com/visualstudio/`. Accessed September 1, 2013.

[112] Thomas G. Moher. Provide: a process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, 1988.

[113] Mathew Mooty, Andrew Faulring, Jeffrey Stylos, and Brad A. Myers. Calcite: Completing code completion for constructors using crowds. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 15–22, 2010.

[114] Brad A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *Proceedings of the SIGCHI Conference*

on Human Factors in Computing Systems, CHI '86, pages 59–66, New York, NY, USA, 1986. ACM.

[115] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[116] Brad A. Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, March 2000.

[117] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Communications of the ACM*, 47(9):47–52, September 2004.

[118] Shinichiro Nakaoka, Shuuji Kajita, and Kazuhito Yokoi. Intuitive and flexible user interface for creating whole body motions of biped humanoid robots. In *Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1675–1682, 2010.

[119] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA, 1st edition, 1993.

[120] Mark W. Newman, Mark S. Ackerman, Jungwoo Kim, Atul Prakash, Zhenan Hong, Jacob Mandel, and Tao Dong. Bringing the field into the lab: supporting capture and replay of contextual data for the design of context-aware applications. In *Proceedings of the 23rd annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 105–108, New York, NY, USA, 2010. ACM.

[121] Donald A. Norman and Stephen W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1986.

[122] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

[123] Walter J. Ong and John Hartley. *Orality and literacy: The technologizing of the word*. Methuen & Co. Ltd, 1982.

[124] John F. Pane and Brad A. Myers. Usability issues in the design of novice programming systems. 1996.

[125] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James Landay. Gestalt: integrated support for implementation and analysis in machine learning. In *Proceedings of the 23rd annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 37–46, New York, NY, USA, 2010. ACM.

[126] Kayur Dushyant Patel. *Lowering the Barrier to Applying Machine Learning*. PhD thesis, 2012.

[127] G. E. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[128] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 535–552, New York, NY, USA, 2007. ACM.

[129] Hayes Solos Raffle, Amanda J. Parkes, and Hiroshi Ishii. Topobo: a constructive assembly system with kinetic memory. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 647–654, New York, NY, USA, 2004. ACM.

[130] Jun Rekimoto. Organic interaction technologies: from stone to skin. *Communications of the ACM*, 51(6):38–44, June 2008.

[131] Alex Repenning and Wayne Citrin. Agentsheets: applying grid-based spatial reasoning to human-computer interaction. In *Proceedings of 1993 IEEE Symposium on Visual Languages*, pages 77–82, 1993.

[132] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 434–441, New York, NY, USA, 1999. ACM.

[133] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '05, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society.

[134] Julia Schwarz, Scott Hudson, Jennifer Mankoff, and Andrew D. Wilson. A framework for robust and flexible handling of inputs with uncertainty. In *Proceedings of the 23rd annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 47–56, New York, NY, USA, 2010. ACM.

[135] Jakub Segen and Senthil Kumar. Gesture vr: vision-based 3d hand interace for spatial interaction. In *Proceedings of the 6th ACM International Conference on Multimedia*, MULTIMEDIA '98, pages 455–464, New York, NY, USA, 1998. ACM.

[136] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, 1983.

[137] N. C. Shu, editor. *Visual programming*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[138] David Canfield Smith. Watch what i do. chapter Pygmalion: an executable electronic blackboard, pages 19–48. MIT Press, Cambridge, MA, USA, 1993.

[139] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving api documentation using api usage information. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 119–126, 2009.

[140] Shigeo Takahashi, Issei Fujishiro, Yuriko Takeshima, and Tomoyuki Nishita. A feature-driven approach to locating optimal viewpoints for volume visualization. In *Proceedings of the 2005 IEEE Visualization*, pages 495–502, 2005.

[141] Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2):127 – 139, 1990.

[142] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 49–60, New York, NY, USA, 2011. ACM.

[143] Bret Victor. Learnable programming. `http://worrydream.com/LearnableProgramming/`, 2012. Accessed September 1, 2013.

[144] Ana-Maria Visan, Kapil Arya, Gene Cooperman, and Tyler Denniston. Urdb: a universal reversible debugger based on decomposing debugging histories. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 8:1–8:5, New York, NY, USA, 2011. ACM.

[145] Ge Wang and Perry Cook. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the 12th annual ACM International Conference on Multimedia*, MULTIMEDIA '04, pages 812–815, New York, NY, USA, 2004. ACM.

[146] Shuo Wang, Xiaocao Xiong, Yan Xu, Chao Wang, Weiwei Zhang, Xiaofeng Dai, and Dongmei Zhang. Face-tracking as an augmented input in video games: enhancing presence, role-playing and control. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 1097–1106, New York, NY, USA, 2006. ACM.

[147] Kirsten N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109 – 142, 1997.

[148] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. Snipmatch: using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 219–228, New York, NY, USA, 2012. ACM.

[149] Andrew D. Wilson. Playanywhere: a compact interactive tabletop projection-vision system. In *Proceedings of the 18th annual ACM Symposium on User Interface Software and Technology*, UIST '05, pages 83–92, New York, NY, USA, 2005. ACM.

[150] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM.

[151] Wataru Yoshizaki, Yuta Sugiura, Albert C. Chiou, Sunao Hashimoto, Masahiko Inami, Takeo Igarashi, Yoshiaki Akazawa, Katsuaki Kawachi, Satoshi Kagami, and Masaaki Mochimaru. An actuated physical puppet as an input device for controlling a digital manikin. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 637–646, New York, NY, USA, 2011. ACM.

[152] Andreas Zeller and Dorothea Lütkehaus. Ddd — a free graphical front-end for unix debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, January 1996.