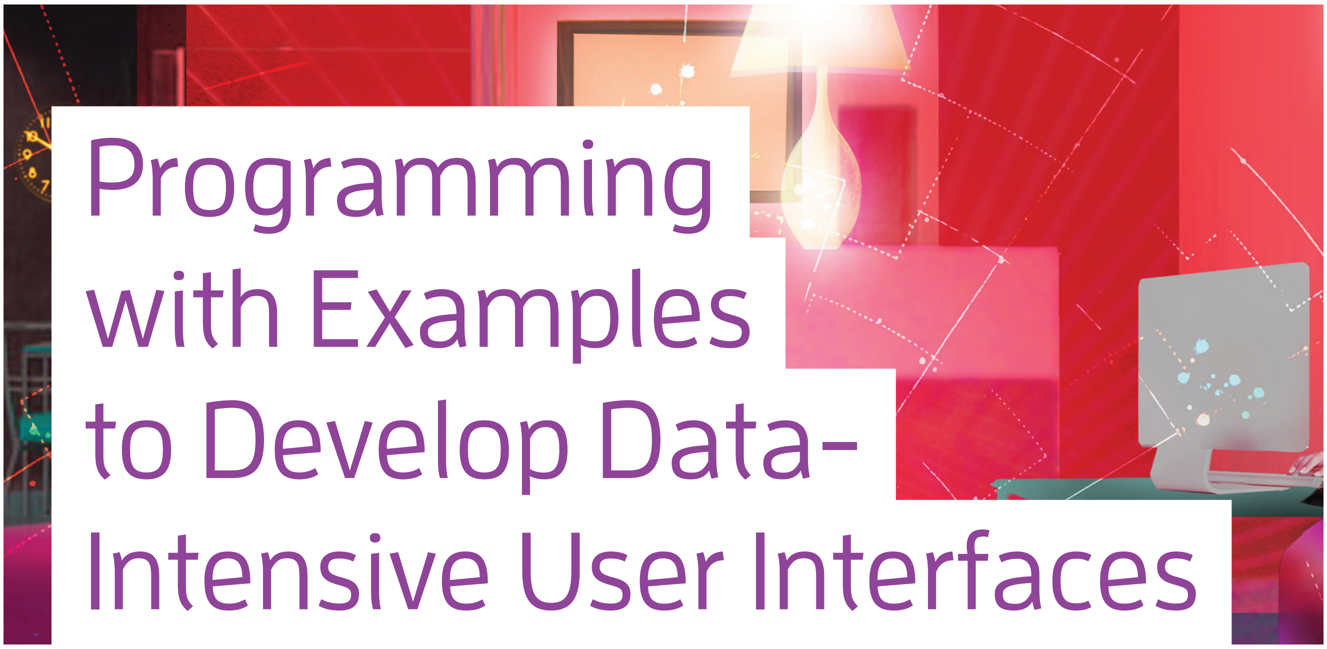


Programming with Examples to Develop Data-Intensive User Interfaces

Jun Kato, National Institute of Advanced Industrial Science and Technology (AIST)
Takeo Igarashi, University of Tokyo
Masataka Goto, National Institute of Advanced Industrial Science and Technology (AIST)

The programming-with-examples workflow lets developers create interactive applications with the help of example data. It takes a general programming environment and adds dedicated user interfaces for visualizing and managing the data. This lets both programmers and users understand applications and configure them to meet their needs.

GUI builders let users define the look of user interfaces with simple "what you see is what you get" (WYSIWYG) interaction. However, WYSIWYG interaction is insufficient to develop working systems; some programming is also necessary. Programming is essential to user interface design because it lets developers design how users interact with an application. To support the development of conventional interactive systems, integrated development environments (IDEs) represent standardized numeric values such as those for `MOUSE_CLICKED` and `KEY_TYPED` as text or symbols. This simple representation reflects the characteristics of systems that run on computers with standardized input and output devices such as a mouse, keyboard, and display. They typically handle a limited amount of data transferred intermittently.

In contrast, modern interactive applications have been becoming more visual and data-intensive. In the age of big data, computers have become faster and smaller, handling more visual data that cannot be presented intuitively with text or symbols. There are more variations among applications than before, such as robot control, gesture recognition, image processing, and animation.

Prior attempts to support the development of data-intensive systems include *programming by example* (PbE) systems (see Figure 1a). With PbE systems, users demonstrate desired pairs of input and output data to the systems—for example, by manipulating robot postures in front of a camera. Then, the systems infer the program, such as one to control robot postures. Most PbE systems let both programmers and end users create programs. However, because these systems typically encapsulate the program logic in a black box, they prevent users from understanding or directly specifying detailed behaviors. In addition, example data have no representation in such systems and cannot be managed or edited, making the development process irreproducible. So, PbE systems without the capability of explicit programming are not suitable for user interface design that precisely reflects a programmer's intention.

In contrast, our *programming with examples* (PwE) workflow (see Figure 1b) enables explicit programming and lets users manage and edit example data. Programmers collect example data to aid building program components. For instance, to create a gesture-based application, the programmer collects a set of example gesture data from a sensor and tests the application with that data. (Brad Myers used the term "Programming with Example" in 1986,[1] though it indicated the omission of inferences in PbE systems and did not discuss data representations.)

Here, we discuss using graphical representations in IDEs to support PwE. Toward that end, we developed three IDEs, each of which explores a different use of graphical representations, including photos, videos, and their interactive editing. Our approach overlaps somewhat with live programming (LP), a general technique that enables live editing of programs during runtime. (For more on LP, see the "Live Programming and Programming with Examples" sidebar.) Our experience has provided insights on user interface design, including guidelines on when to use a particular kind of graphical representation and the potential of visuals as a communication medium for collaborative application authoring.

LIVE PROGRAMMING AND PROGRAMMING WITH EXAMPLES

In terms of the programming experience, the approach most relevant to programming with examples (PwE) is *live programming* (LP), which enables live editing of programs at runtime. LP originated in research on object-oriented programming and visual programming, recently attracting attention for its application to text-based programming environments.

LP and PwE both provide a fluid programming experience through more informative user interfaces, and some overlaps exist. For example, TextAlive is an LP system. However, their focuses differ slightly. LP focuses on logic, whereas PwE focuses on data. LP conceptually aims to make the entire program editable during runtime; PwE splits the program into the editable part (code) and the rest (data). This separation not only simplifies the implementation but also enables end-user program customization.

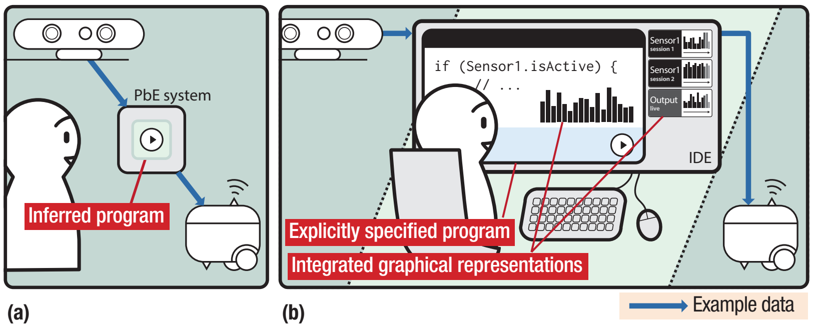


FIGURE 1. Two workflows for developing data-intensive interactive systems. (a) Programming by example (PbE). (b) Programming with examples (PwE). Unlike PbE systems, PwE systems allow the representation of example data and enable explicit programming. IDE: integrated development environment.

PROGRAMMING WITH EXAMPLES

Modern interactive systems often use a variety of physical user interfaces consisting of sensors and actuators that receive and send large amounts of raw data as input and output. This includes visual data, such as photos and videos; object properties, such as color, shape, and location; as well as more structured information, such as human or robot posture and parameters for animating graphical objects. Such data serve as examples to tell computers about the problem we want to solve.

As we mentioned before, PwE uses this example data to build and test programs. For instance, building a program could involve training models for machine-learning algorithms and specifying robot postures. Testing the

RELATED WORK IN HANDLING POSTURE DATA

Topobo is a programming-by-example system in which users specify how robots should move by grabbing and moving their joints.[1] Choreonoid is a GUI tool with physical simulation for creating robot motions.[2] In both cases, the user needs no programming knowledge but has little control over how the robot should respond to user input. To develop interactive systems that handle posture data, we still need integrated development environments (IDEs).

Current IDEs usually come with text-based and symbolic representations, which cannot represent complex data such as posture information. Several studies have involved integrated still images. Sikuli shows image data as an inline image in the code editor,[3] and heterogeneous visual programming languages visualize simple tree structures next to the code editor.[4] In these systems, still images are either the represented data or its visualization. Our Picode IDE uses photos not as just mere images but as representations of underlying posture data.

References

1. H.S. Raffle et al., “Topobo: A Constructive Assembly System with Kinetic Memory,” *Proc. 2004 SIGCHI Conf. Human Factors in Computing Systems* (CHI 04), 2004, pp. 647–654.
2. S. Nakaoka et al., “Intuitive and Flexible User Interface for Creating Whole Body Motions of Biped Humanoid Robots,” *Proc. 2010 IEEE/RSJ Int’l Conf. Intelligent Robots and Systems* (IROS 10), 2010, pp. 1675–1682.
3. T. Yeh et al., “Sikuli: Using GUI Screenshots for Search and Automation,” *Proc. 22nd Ann. ACM Symp. User Interface Software and Technology* (UIST 09), 2009, pp. 183–192.
4. M. Erwig et al., “Heterogeneous Visual Languages Integrating Visual and Textual Programming,” *Proc. 11th Int’l IEEE Symp. Visual Languages* (VL 95), 1995, pp. 318–325.

program would involve executing it with recorded input from sensors and debugging it by checking output from trained models.

In current development environments, example data are usually referenced by textual or symbolic visual representations such as file names and icons. In conventional interactive systems, the connection between the example data, text, and symbols is usually obvious. In modern interactive applications, that is not the case; programmers often get confused by the vague connection. In addition, programmers are responsible for managing the data. To examine and edit the data, they typically must launch external tools outside the development environments, which is tedious and error-prone. The IDEs we present here directly support PwE to address these problems and make the process more efficient and productive.

There have been several efforts to support PwE. Most PbE systems require no programming knowledge, but exceptions exist. For instance, Pygmalion lets programmers build a program by giving concrete input data instead of writing abstract program code.[2] The Subtext IDE lets programmers specify test cases by writing example input data next to program code in a code editor.[3] Our IDEs employ the same PwE workflow as these systems but use graphical instead of textual representations. This difference reflects the example data becoming more visual and complex in modern interactive applications.

Recent IDEs have successfully incorporated various forms of examples. The most relevant one is Gestalt, which supports the implementation of machine-learning algorithms for image recognition.[4] It provides dedicated GUIs to manage and edit example data. Barista lets programmers paste graphics directly into the code editor.[5] Standing atop these prior IDEs with PwE support, we explored a broader design space that employs graphical representations in the IDEs.

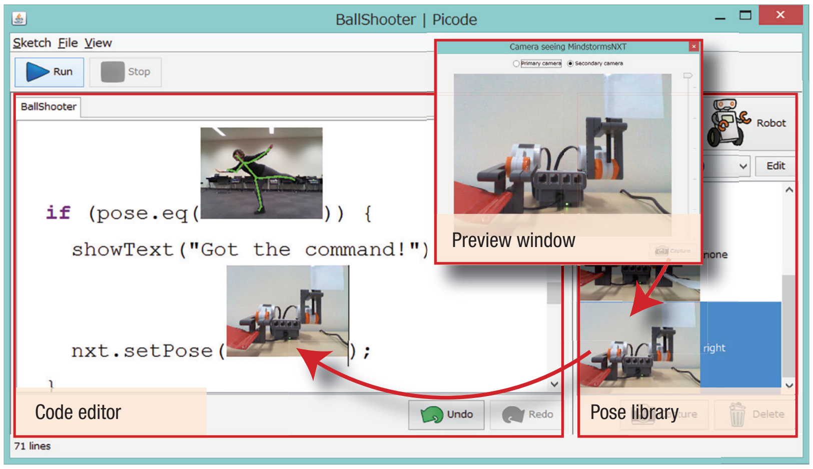
Blueprint lets programmers search for example source code in online repositories and paste the result through a code completion interface.[6] Whereas examples in Blueprint denote the use of APIs—the logic of the programs—examples in PwE denote data used in the programs.

PHOTOS REPRESENTING STATIC DATA

Here we explore using photos to represent static example data, which does not change over time. Our Picode IDE supports the development of programs that sense human and robot postures and control robot postures.[7] (For more on handling posture data, see the “Related Work in Handling Posture Data” sidebar.)

Picode

Picode comprises a code editor, pose library, and preview window (see



```
if (pose.eq( )) {
  showText("Got the command!")

  nxt.setPose( );
}
```

FIGURE 2. Picode shows inline photos in the code editor to represent example posture data. Programmers drag a photo from the pose library and drop it into the code editor. Picode embeds the photo in code, which the programmers can execute with a single click.

Figure 2). Programmers first take a photo of a human or robot in the preview window; Picode captures the corresponding posture data from a Kinect sensor or the robot's servos. Picode stores a collection of these paired data in the pose library as a single entry. Then, programmers drag a photo from the library and drop it into the code editor. Picode embeds the photo in code, which the programmers can execute with a single click.

User study

A programmer and nonprogrammer performed pair programming with Picode for three hours. The programmer benefited from the PwE workflow, and even the nonprogrammer could infer and comprehend the source code surrounding inline photos.

Given the encouraging results, we hosted a workshop for nonprogrammers to further investigate the role of photos in the code. We found that photos contained three types of information that was hardly represented by text, symbols, or 3D computer graphics visualizing posture data.

First, photos present the environment surrounding humans and robots. This helps programmers quickly understand a program's operating environment and objectives. For example, humans employ the same posture whether they are pushing a cart on a slope or performing calisthenics. A robot employs similar postures when holding a small or large ball, with the only difference being how wide the hand opens. Although numerical distinction is possible, this is not informative for humans.

Second, photos sometimes indicate a particular part of a posture. In a text-based IDE, a programmer could indicate interest in the second joint of a robot by inserting a textual comment in the source code. However, the phrase "the second joint" will not instantly inform the reader which joint it is; a photo can be a more direct representation in which the programmer points to the joint.

Finally, photos containing human subjects can express emotion. This characteristic has been significant throughout the medium's long history and is not found in conventional source code. By our study's end, the pose library included photos showing participant enjoyment and creative shots of the robots. The source code with inline photos was extremely individual, showing a variety of clothing, poses, and expressions. The photos motivated the nonprogrammers to customize the program and further learn general programming.

VIDEOS REPRESENTING DYNAMIC BEHAVIOR

Here, we explore using videos to represent time-coded example data. Our DejaVu IDE (see Figure 3) supports the development of interactive camera-based programs.[8] It incorporates a video-player metaphor into a standard text-based IDE. (For more on handling time-coded data, see the "Related Work in Handling Time-Coded Data" sidebar.)

DejaVu

DejaVu employs two interlinked interfaces. The canvas interface corresponds to real-time video preview in video-editing applications. Users can monitor any number of variables, including input and output, continuously during runtime in an arbitrary layout. They can also review a frame of interest in a past program session.

When programmers select a variable in the code editor and drag it onto the canvas, DejaVu represents it as a rectangular box with visuals (images and skeletal data, in the case of human subjects in the video) or text (numerical or Boolean data). Along with monitoring variable values, programmers can draw sketches and notes on the canvas to aid visual data management. By combining sketches with variable values, programmers can turn the canvas into a "dynamic sketchbook" in which sketches come alive with dynamic data.

Similarly to interfaces in video-editing applications, the timeline interface represents changes in the example data over time. The timeline might consist of multiple data streams, each corresponding to a variable in the canvas. A stream of visual data is represented as a strip of frame thumbnails, and a stream of numerical or Boolean data is represented as a time graph. The timeline enables not only passive review of example data but also revising the program and refreshing it by reexecuting it with recorded input data, which assists iterative development.

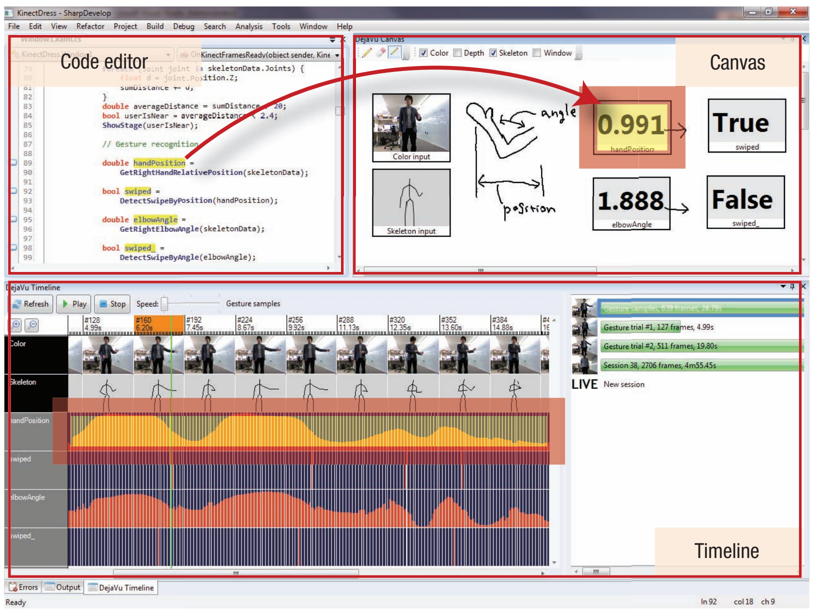
RELATED WORK IN HANDLING TIME-CODED DATA

Some IDEs visualize time-coded information of program executions. ZStep records all stack traces and provides a navigation interface to go back and forth over the trace to see which line of code executed at a particular point.[1] Whyline records the stack traces and window output.[2] It also provides a "Why did this happen (or not happen)?" interface, which navigates to the cause of a phenomenon such as the color of a pixel. These integrations work well for discrete events with simple data structures but cannot handle continuous data streams such as input from cameras and output to servo motors.

Some developer tools have timeline interfaces to help users understand continuous time-coded data. Exemplar is a standalone tool for authoring sensor-based interaction by demonstration.[3] D.tools records user interactions with physical computing devices and visualizes the events along with recorded videos.[4] Timelapse has similar features for event logging and webpage debugging.[5] These tools are either a standalone tool outside the IDE or an IDE without a text-based code editor. In contrast, the DejaVu IDE tightly integrates the code editor and timeline interface.

References

1. H. Lieberman et al., "Bridging the Gulf between Code and Behavior in Programming," *Proc. 1995 SIGCHI Conf. Human Factors in Computing Systems* (CHI 95), 1995, pp. 480–486.
2. A.J. Ko et al., "Finding Causes of Program Output with the Java Whyline," *Proc. 2006 SIGCHI Conf. Human Factors in Computing Systems* (CHI 06), 2006, pp. 387–396.
3. B. Hartmann et al., "Authoring Sensor-Based Interactions by Demonstration with Direct Manipulation and Pattern Recognition," *Proc. 2007 SIGCHI Conf. Human Factors in Computing Systems* (CHI 07), 2007, pp. 145–154.
4. B. Hartmann et al., "Reflective Physical Prototyping through Integrated Design, Test, and Analysis," *Proc. 19th Ann. ACM Symp. User Interface Software and Technology* (UIST 06), 2006, pp. 299–308.
5. B. Burg et al., "Interactive Record/Replay for Web Application Debugging," *Proc. 26th Ann. ACM Symp. User Interface Software and Technology* (UIST 13), 2013, pp. 473–484.



KinectDress - SharpDevelop
File Edit View Refactor Project Build Debug Search Analysis Tools Window Help
Code editor
Window1.xaml.cs
KinectDress.Window1
OnKinectFramesReady(object sender, Kine
DejaVu Canvas
Color Depth Skeleton Window
Canvas
Color input
Skeleton input
angle
position
0.991
handPosition
True
swiped
1.888
elbowAngle
False
swiped_
DejaVu Timeline
Refresh Play Stop Speed: Gesture samples
#128 4.99s
#160 6.20s
#192 7.45s
#224 8.67s
#256 9.92s
#288 11.13s
#320 12.35s
#352 13.60s
#384 14.88s
Color
Skeleton
handPosition
swiped
elbowAngle
swiped_
Gesture samples, 639 frames, 24.79s
Gesture trial #1, 127 frames, 4.99s
Gesture trial #2, 511 frames, 19.80s
Session 38, 2706 frames, 4m55.45s
LIVE New session
Timeline
Errors Output DejaVu Timeline
Ready
ln 92 col 18 ch 9

```
foreach (Joint joint in skeletonData.Joints) {
    float d = joint.Position.Z;
    sumDistance += d;
}
double averageDistance = sumDistance / 20;
bool userIsNear = averageDistance < 2.4;
ShowStage(userIsNear);

// Gesture recognition

double handPosition =
    GetRightHandRelativePosition(skeletonData);

bool swiped =
    DetectSwipeByPosition(handPosition);

double elbowAngle =
    GetRightElbowAngle(skeletonData);

bool swiped_ =
    DetectSwipeByAngle(elbowAngle);
```

FIGURE 3. DejaVu shows time-coded example data in the canvas and timeline interfaces. It incorporates a video-player metaphor into a standard text-based IDE.

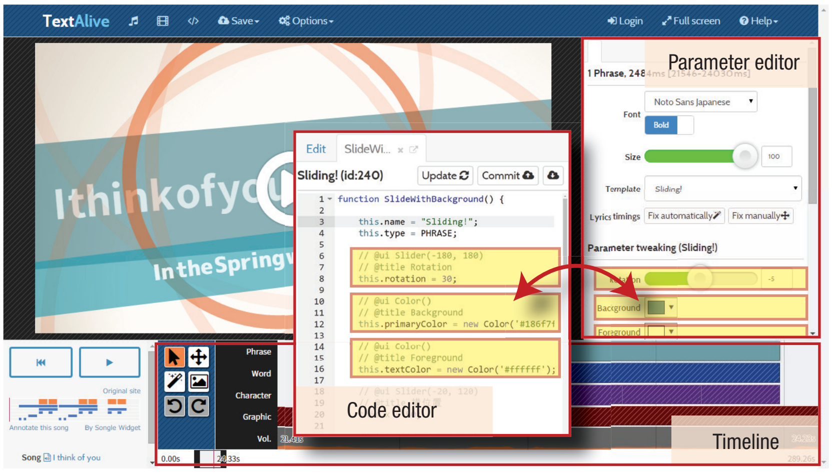
User feedback

Three professional developers tried out DejaVu. They had significant experience in developing interactive Kinect-based programs with Visual Studio, a standard IDE. Afterward, they agreed that DejaVu correlated well with their current PwE workflow. We learned three important lessons from their feedback.

First, the tight integration in the IDE enables the synchronous connection between the canvas, timeline, and code editor. The participants previously had desired DejaVu's features and sometimes even made their own tools. However, the separate tools were not as powerful and flexible as DejaVu's visual, integrated, and interactive support.

Second, it is not always easy to collect example data from the programmer's surroundings that satisfy quality or quantity requirements for program testing. The programmers wanted the ability to import and manipulate the example data from external sources.

Third, the canvas went beyond individual data displays and aroused the need for customizable visualization. The visualization could range from simple graphic combinations such as overlaying a skeleton on a color image to more semantic compositions such as masking certain image regions.



Sliding! (id:240)

```
function SlideWithBackground() {

    this.name = "Sliding!";
    this.type = PHRASE;

    // @ui Slider(-180, 180)
    // @title Rotation
    this.rotation = 30;

    // @ui Color()
    // @title Background
    this.primaryColor = new Color('#186f7(

    // @ui Color()
    // @title Foreground
    this.textColor = new Color('#ffffff');

    // @ui Slider(-20, 120)
```

FIGURE 4. TextAlive allows live customization of programs with interactive user interfaces. Programmers and nonprogrammers can collaborate to create programs.

INTERACTIVE EDITING OF EXAMPLE DATA

Whereas the previous two sections explored visualization of example data, this section focuses on using graphical operations to edit example data. Our TextAlive IDE (see Figure 4) supports live programming of domain-specific applications (video-rendering programs) and offers interactive user interfaces for tuning their parameters.[9] (For more on editing data, see the "Related Work in Data Manipulation" sidebar.)

TextAlive is similar to our VisionSketch, which supports the example-centric development of image-processing algorithms.[10] Although VisionSketch is primarily for programmers, it led us to develop TextAlive, which investigates the potential of collaborative application authoring. With TextAlive's clearer separation of user interfaces for programming and parameter tuning, programmers and nonprogrammers can collaborate to create applications.

TextAlive

TextAlive lets users create a synchronized kinetic-typography video; it works just like familiar video-authoring tools. When users load an audio file and its transcription, TextAlive automatically creates a video. Other tools require users to create videos from scratch and spend a huge amount of time manually synchronizing audio and text. TextAlive requires significantly less effort.

For each text sample and graphic in the video, users can select an animation template and customize its visual effects with the GUI widgets in the parameter editor. They can also use the widgets to debug templates. Synchronization errors can be corrected in the timeline, which visualizes the text's timing information.

Although each animation template supports customization, it still somewhat limits the resulting animation. To address this issue, TextAlive enables live programming of the templates. Many tools for creating animations include scripting engines, but such scripting is typically tied to the specific data and not easily generalizable for later reuse. These tools also do not provide much graphical feedback during coding. TextAlive enables more reuse through an abstraction mechanism and provides continuous graphical feedback. Programmers can open a code editor to edit the implementation and update the resulting video with just one click. There is no notion of compilation or execution. The program that creates the animations continues running virtually.

The update process not only updates the video but also populates or removes the GUI widgets. To populate a widget, programmers declare a variable in the template definition and write a comment block right before the declaration.

User feedback

Four nonprogrammers and three programmers created videos with TextAlive. They had varying expertise in video authoring and programming. All of them welcomed the features for authoring kinetic-typography videos, especially the timeline. Whereas the nonprogrammers requested more variations of templates, the programmers could implement new templates and appreciated live programming and easy widget creation.

To observe collaboration among users, we deployed TextAlive as a Web service. From September 2015 to January 2016, the users created more than 300 videos and 50 templates. The nonprogrammers reported enjoying authoring videos, whereas the programmers reported enjoying developing templates for extending expressivity with the help of the edited example data. The programmers sometimes created videos to showcase their

RELATED WORK IN DATA MANIPULATION

Visual programming languages (VPLs) let programmers manipulate visual components to build programs. Most VPLs do not support graphical operations other than repositioning boxes and connecting boxes with lines. However, some go beyond the box-and-line notation and allow editing several data types such as bitmaps.

Some text-based IDEs also provide GUIs for inputting values. Active code completion provides type-specific GUI widgets for specifying concrete values.[1] For instance, when a programmer instantiates a `Color` object, a color palette interface is populated instead of text-based completion candidates. Unity for authoring games (http://unity3d.com) and Apparatus for interactive graphs (http://aprt.us) enable live customization of program outputs through graphical operations.

Although these IDEs allow interactive editing of certain data without external tools, they assume that the data can be constructed from scratch, which is not feasible for creating modern applications. In contrast, our TextAlive IDE allows interactive editing of example data provided from outside the IDE. It has separate interfaces for programmers and nonprogrammers, connecting them through integrated graphical representations.

Similarly, Gneiss provides a spreadsheet interface with a simple domain-specific language to retrieve and organize real-time example data from the Web.[2] Unlike TextAlive, Gneiss provides a single user interface that both novices and nonprogrammers can use.

References

1. C. Omar et al., "Active Code Completion," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 859–869.
2. K. Chang et al., "A Spreadsheet Model for Handling Streaming Data," *Proc. 2015 SIGCHI Conf. Human Factors in Computing Systems* (CHI 15), 2015, pp. 3399–3402.

templates. Graphical representations of the programs (videos) and their parameters (GUI widgets) served as a bridge between the programmers and nonprogrammers.

A limitation of our current implementation is that collaboration is unidirectional; nonprogrammers have no direct way to request new templates with specific visual effects. The system ideally should support bidirectional collaboration, which we plan to work on.

LESSONS LEARNED

We now present lessons learned from our experiences with our three IDEs and describe our outlook on this technology.

Integrated graphical representations for PwE

Our experiments confirmed that integrating graphical representations lets programmers understand, manage, and use the example data to build and test programs, effectively supporting the PwE workflow. Combining concrete graphical representations with abstract textual representations improved programmer productivity. Compared to typical PbE systems that infer the interaction logic, PwE leaves it to the programmers, enabling precise, detailed interaction design through iterative testing and edits.

Furthermore, example data with graphical representations make the program specification more accessible to nonprogrammers such as visual designers and end users. They can easily understand the program behavior from visual examples (as with Picode) and even modify the behavior by interacting with the visuals (as with TextAlive).

Intuitive representation of example data

Integrated graphical representations can be realistic (for example, photos and videos) or symbolic (for example, a skeleton representing a human body), depending on their use cases. Although it seems obvious that graphics aid programming, which kind of representation to use is not always obvious. Photos and videos are particularly useful for capturing the runtime environment, including the real-world circumstances (as with Picode). They might contain explicit information, such as interactions the user performed, and implicit information, such as the user's emotions and environment. On the other hand, illustrations and symbolic figures can eliminate unnecessary information and are often good for understanding abstract intentions. Combinations of realistic and symbolic representations also work well. Just as map applications support layering of satellite views and symbolic maps, these two representations can be overlaid.

Video-authoring interfaces (such as DejaVu and TextAlive) can intuitively present time-coded structured data. Researchers have proposed many intuitive interfaces for manipulating data through GUIs; we can learn from such interfaces to integrate data manipulation components into IDEs. Although this article focuses on inherently visual example data, example data could be sounds, haptic sensations, tastes, and smells. Graphical representations could also be useful in these cases. For instance, photos of flowers could represent their scent. At the same time, we foresee that future development environments could exploit human sensory organs other than the eyes, including the ears, skin, tongue, and nose.

Collaborative application authoring in the age of big data

Collaboration among people with diverse backgrounds is essential for developing successful user interfaces for data-intensive applications in the age of big data. The amount and variations of data far surpass what programmers alone can manage and edit.

Current development environments enable such collaborations in text-based communication, as seen in bug reports. Some services, including GitHub, allow embedding graphics, but we argue that such graphics should be more directly bound to the program specifications, as our experiments demonstrated. This will let programmers and nonprogrammers create applications that satisfy their needs. To achieve this, we need IDEs with two faces: one for programmers and one for nonprogrammers. The graphical representation is not just a tool for programmers to build and test programs or a tool for nonprogrammers to customize program behavior. It is a common language for both to discuss how the program should work.

Eliminating borders between the development and runtime environments

After IDEs integrate more graphics and enable people with diverse backgrounds to comprehend them, we will eventually reach the point at which no clear distinction exists between the development and runtime environments. Programmers will ship programs together with the development environments so that nonprogrammers can customize them by updating the example data. As TextAlive shows, such development (and runtime) environments should be connected to the Internet for sharing applications. Then, others can run the application as is but also edit it seamlessly in the online development environment to improve it iteratively.

When no distinction exists between the development and runtime environments, applications will always remain editable. However, if we increase the level of freedom and allow complete rewriting as implemented in object-oriented environments such as Morphic,[11] users might become overwhelmed. They might unintentionally break the core part of the applications. Whereas most live-programming environments aim to make everything editable anytime, we used graphical representations as the boundary between what can be easily customized and what cannot. We believe that user interface design will become a process to create applications with such a boundary that defines the appropriate flexibility.

As the number of data-driven applications increases in the coming decades, their development will involve much more data manipulation that cannot (and should not) be handled solely by programmers. PwE will be essential in such development, inviting nonprogrammers into the iterative development cycle and letting them customize applications. There will be no static distribution of programs, but they will always remain somewhat customizable. Designing such flexibility will be the important task of user interface design, and intuitive representations of example data will be essential to effective design. We hope that the integrated graphical representations we introduced in this article serve as a good starting point for the journey toward successful user interface design in the age of data-intensive applications. C

ACKNOWLEDGMENTS

The projects introduced in this article have been done in collaboration with Xiang Cao, Sean McDirmid, and Tomoyasu Nakano. The projects are supported partly by the Japan Science and Technology Agency ERATO (Exploratory Research for Advanced Technology) program and CREST (Core Research for Evolutionary Science and Technology). More details are at http://junkato.jp/programming-with-examples.

REFERENCES

1. B.A. Myers, "Visual Programming, Programming by Example, and Program Visualization; A Taxonomy," *Proc. 1986 SIGCHI Conf. Human Factors in Computing Systems* (CHI 86), 1986, pp. 59–66.
2. D.C. Smith, "Pygmalion: An Executable Electronic Blackboard,"

ABOUT THE AUTHORS

JUN KATO is a researcher at Japan's National Institute of Advanced Industrial Science and Technology (AIST). His research interests include human–computer interaction, particularly user interfaces for authoring interactive content. Kato received a PhD in computer science from the University of Tokyo. Contact him at jun.kato@aist.go.jp or via http://junkato.jp.

TAKEO IGARASHI is a professor in the University of Tokyo's Graduate School of Information Science and Technology. His research interests include user interfaces and computer graphics. Igarashi received a PhD in information engineering from the University of Tokyo. Contact him at takeo@acm.org.

MASATAKA GOTO is a prime senior researcher at AIST. His research interests include music information research based on signal processing. Goto received a PhD in engineering from Waseda University. Contact him at m.goto@aist.go.jp.

Watch What I Do, MIT Press, 1993, pp. 19–48.
3. J. Edwards, "Subtext: Uncovering the Simplicity of Programming," *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 05), 2005, pp. 505–518.
4. K. Patel et al., "Gestalt: Integrated Support for Implementation and Analysis in Machine Learning," *Proc. 23rd Ann. ACM Symp. User Interface Software and Technology* (UIST 10), 2010, pp. 37–46.
5. A.J. Ko and B.A. Myers, "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors," *Proc. 2006 SIGCHI Conf. Human Factors in Computing Systems* (CHI 06), 2006, pp. 387–396.
6. J. Brandt et al., "Example-Centric Programming: Integrating Web Search into the Development Environment," *Proc. 2010 SIGCHI Conf. Human Factors in Computing Systems* (CHI 10), 2010, pp. 513–522.
7. J. Kato, D. Sakamoto, and T. Igarashi, "Picode: Inline Photos Representing Posture Data in Source Code," *Proc. 2013 SIGCHI Conf. Human Factors in Computing Systems* (CHI 13), 2013, pp. 3097–3100.
8. J. Kato, S. McDirmid, and X. Cao, "DejaVu: Integrated Support for Developing Interactive Camera-Based Programs," *Proc. 25th Ann. ACM Symp. User Interface Software and Technology* (UIST 12), 2012, pp. 189–196.
9. J. Kato, T. Nakano, and M. Goto, "TextAlive: Integrated Design Environment for Kinetic Typography," *Proc. 2015 SIGCHI Conf. Human Factors in Computing Systems* (CHI 15), 2015, pp. 3403–3412.
10. J. Kato and T. Igarashi, "VisionSketch: Integrated Support for Example-Centric Programming of Image Processing Applications," *Proc. Graphics Interface* (GI 14), 2014, pp. 115–122.
11. J.H. Maloney and R.B. Smith, "Directness and Liveness in the Morphic User Interface Construction Environment," *Proc. 8th Ann. ACM Symp. User Interface Software and Technology* (UIST 05), 1995, pp. 21–28.