DejaVu: Integrated Support for Developing Interactive Camera-Based Programs

Jun Kato[1,2], Sean McDirmid[1], Xiang Cao[1]

[1]Microsoft Research Asia
Beijing, China
{smcdirm, xiangc}@microsoft.com

[2]The University of Tokyo
Tokyo, Japan
jun.kato@acm.org

ABSTRACT

The increasing popularity of interactive camera-based programs highlights the inadequacies of conventional IDEs in developing these programs given their distinctive attributes and workflows. We present DejaVu, an IDE enhancement that eases the development of these programs by enabling programmers to visually and continuously monitor program data in consistency with the frame-based pipeline of computer-vision programs; and to easily record, review, and reprocess temporal data to iteratively improve the processing of non-reproducible camera input. DejaVu was positively received by three experienced programmers of interactive camera-based programs in our preliminary user trial.

Author Keywords

Computer vision; development environment.

ACM Classification Keywords

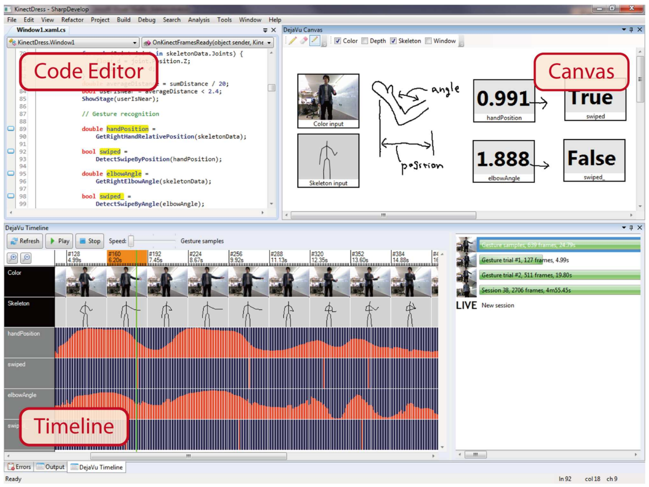
H.5.2 [Information interfaces and presentation]: User Interfaces - Graphical user interfaces.

INTRODUCTION

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 [29] and in the air [25], detect human faces [28], track tangible implements [3], as well as monitor crowd activity [15]. 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 [22] 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 [2], or rapid prototyping tools for camera-based applications such as Crayons [7] and EyePatch [16] 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 [23], 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 1): 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.


KinectDress - SharpDevelop
File Edit View Refactor Project Build Debug Search Analysis Tools Window Help
Window1.xaml.cs
KinectDress.Window1
OnKinectFramesReady(object sender, Kine
Code Editor
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
Color
Skeleton
handPosition
swiped
elbowAngle
Timeline
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
Errors Output DejaVu Timeline
Ready

```
                  in skeletonData.Joints) {
                  sition.Z;

                         = sumDistance / 20;
                  averageDistance < 2.4;
        ShowStage(userIsNear);

        // Gesture recognition

        double handPosition =
            GetRightHandRelativePosition(skeletonData);

        bool swiped =
            DetectSwipeByPosition(handPosition);

        double elbowAngle =
            GetRightElbowAngle(skeletonData);

        bool swiped_ =
            DetectSwipeByAngle(elbowAngle);
```

Figure 1: DejaVu Interface.

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.

A Representative Example

KinectDress (Figure 2a) 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 2b). 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 2c). The user can also make a swiping hand gesture to cycle through a list of available suits to wear (Figure 2d).

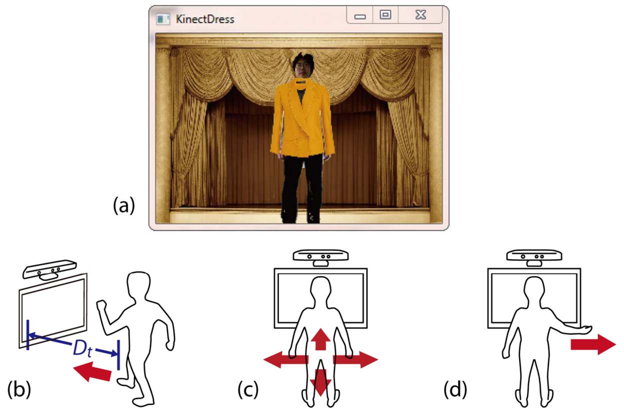


Figure 2: KinectDress interface and interactions.

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 3 illustrates this classic architecture used in KinectDress. Each iteration of the loop starts with the camera capturing the next frame, followed by the pipeline that processes the frame and updates the system's logical and graphical state accordingly.

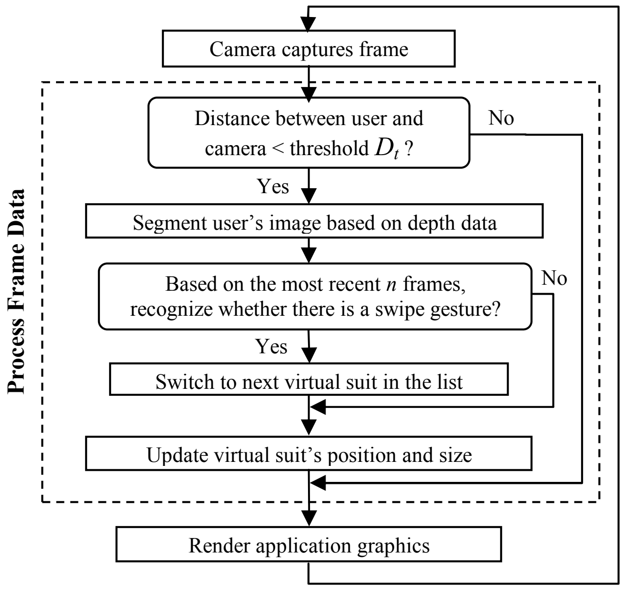

Camera captures frame
Process Frame Data
Distance between user and camera < threshold D_t ?
No
Yes
Segment user's image based on depth data
Based on the most recent n frames, recognize whether there is a swipe gesture?
No
Yes
Switch to next virtual suit in the list
Update virtual suit's position and size
Render application graphics

Figure 3: KinectDress program flow.

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 3). Finally, KinectDress illustrates how color, depth, and skeleton data are processed in combination as common in Kinect programming.

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 Figure 3), 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.

DEJAVU

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 [26], 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 1) consists of two tightly interlinked components: the *canvas* and the *timeline*.

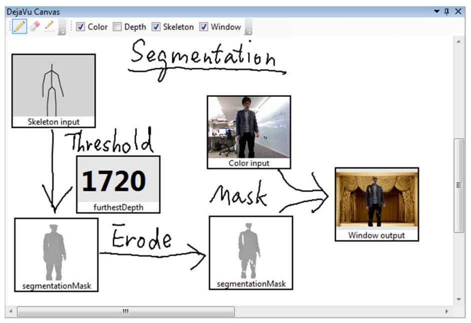


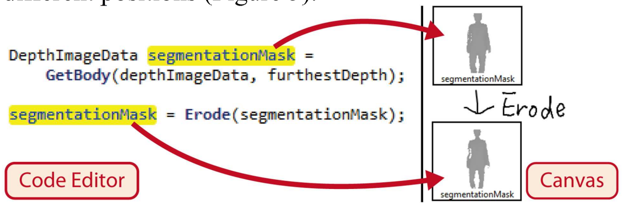
Figure 4: DejaVu Canvas.

DejaVu Canvas

Reflecting the continuous and visual nature of camera input and processing, the canvas (Figure 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 conventional 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).



```
DepthImageData segmentationMask =
    GetBody(depthImageData, furthestDepth);

segmentationMask = Erode(segmentationMask);
```

Figure 5: Variable values in the canvas depend on their source positions in the code editor.

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 [17], 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 [14].

DejaVu Timeline

The timeline (Figure 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.

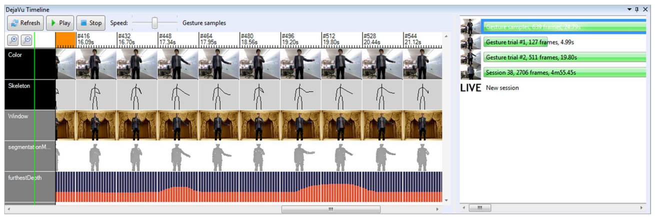


Figure 6: DejaVu Timeline.

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.

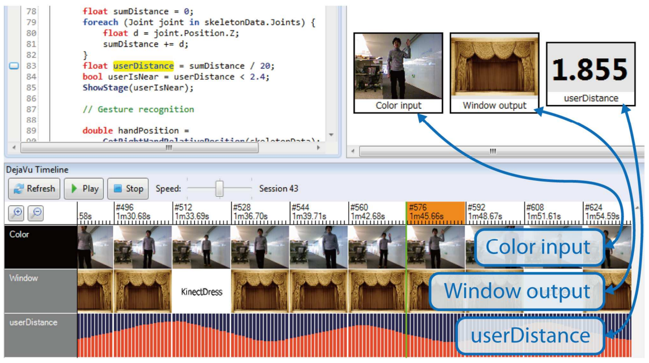
Example Use Case

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



```
float sumDistance = 0;
foreach (Joint joint in skeletonData.Joints) {
    float d = joint.Position.Z;
    sumDistance += d;
}
float userDistance = sumDistance / 20;
bool userIsNear = userDistance < 2.4;
ShowStage(userIsNear);

// Gesture recognition

double handPosition =
```

Figure 7: Tuning the distance threshold.

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 1 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.

Implementation

DejaVu is implemented in C# based on the existing SharpDevelop IDE. A custom-built thin wrapper API around the Microsoft Kinect SDK acts as the shared medium between the DejaVu components, the programmer's code, and the Kinect camera. 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 allows the DejaVu components to track and record Kinect input. The wrapper also allows DejaVu to switch between feeding live 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 synchronously generated via the wrapper.

DejaVu's continuous monitoring and recording of variable values 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 allows for DejaVu's position-aware variable monitoring capability. Code change in the program are tracked by the code editor and handled during compilation to maintain reference to variables and consistency between canvas/timeline and the code.

Although DejaVu currently supports a finite set of types in terms of data visualization, its architecture is extensible enough so that additional types could be supported in a third-party extension model.

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 [cf. 4, 21], 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.

RELATED WORK

Supporting Applications of Computer Vision

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 [7] 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 [16] 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-Mâché toolkit [13] supports building tangible user interfaces through computer vision, barcodes, and electronic tags; and users of CAMBIENCE [5] 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 [2] and XVision [9], can readily be leveraged by programmers in their programs. DejaVu fulfills a complementary need, and may be used together with these libraries seamlessly.

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 [11] integrates the design, test, and analysis of physical prototypes including sensors, while also providing a visual programming environment for authoring control flow. Exemplar [10] 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 [21] and FauxPut [4] 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 [23], 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 [7, 16] 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.

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 [1], its first graphical example appears in PROVIDE [20] and more recent work includes TOD [24] and URDB [27]. Most relevant to our work is liblog [8], 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) [30] allows data structures to be visualized as graphs, while Microsoft Visual Studio [19] 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 [18] and Subtext [6], 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 [12] 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.

DISCUSSION AND CONCLUSION

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 tool 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.

In conclusion, DejaVu provides enhanced integrated support that tightly matches the distinctive nature and workflow of developing interactive camera-based programs. It has been positively received by representative target users, and is a timely exploration facing today's wide adoption of camera-based interactions.

REFERENCES

1. Balzer, R. (1969). EXDAMS: extensible debugging and monitoring system. *Spring joint computer conference*, p. 567-580.
2. Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal November 2000, Computer Security*.
3. Cao, X., Balakrishnan, R. (2003). VisionWand: interaction techniques for large displays using a passive wand tracked in 3D. *UIST*, p. 173-182.
4. Cardenas, T., Bastea-Forte, M., Ricciardi, A., Hartmann, B., Klemmer, S. (2008). Testing physical computing prototypes through time-shifted & simulated input traces. *UIST adjunct proceedings*.
5. Diaz-Marino, R., Greenberg, S. (2006). CAMBIENCE: A Video-Driven Sonic Ecology for Media Spaces. *Video Proceedings of CSCW*.
6. Edwards, J. (2005). Subtext: Uncovering the simplicity of programming. *OOPSLA*, p. 505–518.
7. Fails, J., Olsen, D. (2003). A design tool for camera-based interaction. *CHI,* p. 449-456.
8. Geels, D., Altekar, G., Shenker, S., Stoica, I. (2006). Replay debugging for distributed applications. *USENIX*, p. 289-300.
9. Hager, G. D., Toyama, K. (1998). X Vision: A portable substrate for real-time vision applications. *Computer Vision and Image Understanding*, 69(1), p. 23-37.
10. Hartmann, B., Abdulla, L., Mittal, M., Klemmer, S. R. (2007). Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. *CHI*, p. 145-154.
11. Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., Gee, J. (2006). Reflective physical prototyping through integrated design, test, and analysis. *UIST*, p. 299-308.
12. Hartmann, B., Yu, L., Allison, A., Yang, Y., Klemmer, S. R. (2008). Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. *UIST*, p. 91-100.
13. Klemmer, S. R., Li, J., Lin, J., Landay, J. A. (2004). Papier-Mache: toolkit support for tangible input. *CHI*, p. 399-406.
14. Landay, J. and Myers, B. (1995). Interactive sketching for the early stages of user interface design. *CHI*, p. 43-50.
15. Maynes-Aminzade, D., Pausch, R., Seitz, S. (2002). Techniques for interactive audience participation. *ICMI*, p. 15-20.
16. Maynes-Aminzade, D., Winograd, T., Igarashi, T. (2007). Eyepatch: Prototyping Camera-based Interaction Through Examples. *UIST*, p. 33-42.
17. Max/MSP. Cycling '74. http://cycling74.com/products/maxmsp
18. McDirmid, S. (2007). Living it up with a live programming language. *OOPSLA*, p. 623-638.
19. Microsoft Visual Studio Visualizers. http://msdn.microsoft.com/en-us/library/zayyhzts.aspx
20. Moher, T. G. (1988). PROVIDE: a process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6), p.849-857.
21. Newman, M. W., Ackerman, M. S., Kim, J., Prakash, A., Hong, Z., Mandel, J., Dong, T. (2010). Bringing the field into the lab: supporting capture and replay of contextual data for the design of context-aware applications. *UIST*, p. 105-108.
22. Norman, D. A., Draper, S. W. (1986). User centered system design; new perspectives on human-computer interaction. L. Erlbaum Assoc. Inc.
23. Patel, K., Bancroft, N., Drucker, S. M., Fogarty, J., Ko, A. J., Landay, J. (2010). Gestalt: integrated support for implementation and analysis in machine learning. *UIST*, p. 37-46.
24. Pothier, G., Tanter, E., Piquer, J. (2007). Scalable omniscient debugging. *SIGPLAN*, p. 535-552.
25. Segen, J., Kumar, S. (1998). Gesture VR: Vision-based 3D hand interface for spatial interaction. *Multimedia*. p. 455-464.
26. SharpDevelop. http://www.icsharpcode.net/opensource/sd/
27. Visan, A., Arya, K., Cooperman, G., Denniston, T. (2011). URDB: a universal reversible debugger based on decomposing debugging histories. *PLOS*, p. 1-5.
28. Wang, S., Xiong, X., Xu, Y., Wang, C., Zhang, W., Dai, X., Zhang, D. (2006). Face-tracking as an augmented input in video games: enhancing presence, role-playing and control. *CHI*, p. 1097-1106.
29. Wilson, A. D. (2005). PlayAnywhere: a compact interactive tabletop projection-vision system. *UIST,* p. 83-92.
30. Zeller, A., Lütkehaus, D. (1996). DDD - a free graphical front-end for UNIX debuggers. SIGPLAN Notices, 31(1), p. 22-27.