Automatically Surfacing Source Code to Augment Interactions with Programming Videos

anonymized for submission

ABSTRACT

Screencast videos are a popular format for computer programming tutorials since they convey expert narration alongside the dynamic effects of editing and running code. However, these videos simply consist of raw pixels, so there is no way to interact with the code inside of them. Our goal in this work is to expand the design space of interactions with programming videos beyond conventional video players. To do so, we first developed Codemotion, a computer vision-based algorithm that automatically extracts source code and dynamic edits from existing videos. Codemotion segments a video into regions that likely contain code, performs OCR on those segments, recognizes source code, and merges together related code edits into contiguous intervals. A quantitative assessment on 20 YouTube videos shows that Codemotion can find 94.2% of code-containing segments with an OCR error rate of 11.2%. We then used Codemotion to elicit screencast-related interaction design ideas from potential users by running four participatory design workshop sessions. Participants collectively generated ideas for 28 kinds of interactions such as inline code editing, code-based skimming, pop-up video search, and in-video coding exercises—all enabled by the data that Codemotion automatically extracts from videos.

Author Keywords

screencast videos; computer programming

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

Screencast videos are now a popular tutorial format [37], especially for computer-based topics such as programming [11, 27, 28, 36]. A search for "computer programming tutorial" on YouTube returns 6.6 million videos. These videos have several advantages over text-based formats: Videos convey the dynamic effects of running code, such as GUI animations and user interactions [27, 28, 37] (e.g., Figure 1). Videos reveal the dynamic process of an expert editing and debugging code in real time, which lets viewers emulate learning by "looking over an expert's shoulder." [41, 42]. Finally, creators can easily record videos by writing code, running it, and narrating all without breaking their flow [25, 27, 28].

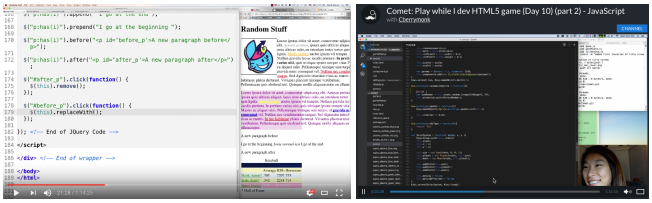

Figure 1. Examples of programming screencast videos: jQuery for web frontend (left) and live-streaming a game development tutorial (right).

However, videos are much harder to search and navigate than text [22, 24, 35]. For example, say you wanted to learn how experts use a particular sequence of API function calls. It can be hard to even find any videos that showcase those functions because search engines index only titles, descriptions, and transcripts (if available). Once you do find a promising video, it is hard to use a conventional video player to navigate to the point where those functions are used. Even when you finally reach the relevant part, you cannot copy-and-paste the code to experiment with it on your own computer. The underlying cause of these limitations is that the code within screencast videos are simply raw pixels with no semantic meaning.

What new user interactions might be possible if we could access the code locked within the raw pixels of millions of these programming screencast videos? This paper addresses this question by introducing a new computer vision approach that automatically extracts source code from videos. Using its output, we created a prototype tutorial viewer that facilitates code search and navigation within videos. We then used this viewer to elicit a diverse set of interaction ideas from programming students via a participatory design process [33]. *The main contribution of this paper to HCI is expanding the design space of user interactions with programming screencast videos by surfacing the code latent within existing videos.*

To work toward this goal, we first created an algorithm called *Codemotion* that automatically segments a screencast video into regions that likely contain code, performs OCR (optical character recognition) to turn each segment into text, and uses time-aware diff heuristics to merge intervals of related code edits, which reconstructs the original code and how it changed over time. To assess the performance of Codemotion, we ran it on a corpus of 20 YouTube programming videos ranging from 4 minutes to 2 hours in length. It was able to find 94.2% of code-containing segments across all of these videos with an OCR error rate of 11.2%. An exploratory user study on

10 students found that users felt code interval lengths looked reasonable and that occasional OCR errors were acceptable.

We then used Codemotion to elicit screencast-related interaction design ideas from potential users. We conducted 4 participatory design workshops sessions [33] with 12 university students and encouraged them to brainstorm divergently to create multiple unrelated designs [4]. Participants collectively generated 28 ideas for enhanced video interactions, which we grouped into four categories: code, navigation, search, and active learning. Example ideas include inline code editing, code-based skimming, pop-up video search, and in-video coding exercises. These interaction designs point toward a future where the huge body of knowledge hidden within millions of existing videos can be surfaced to enable richer and more interactive types of tutorial interfaces.

The main contributions of this paper are:

- Codemotion, a computer vision algorithm that extracts source code and edit intervals from existing videos, validated in a quantitative study and an exploratory user study.
- A set of 28 interaction design ideas generated by programming students in four participatory design sessions, which use the data provided by Codemotion to expand the design space of interactions with programming screencast videos.

RELATED WORK

Extracting State from Screenshots & Screencast Videos

Codemotion's computer vision-based technique for automatically extracting code from videos was inspired by systems that extract state from screenshots and screencast videos.

Sikuli [6, 43] and Prefab [10] use computer vision to automatically identify GUI elements from screenshots, which lets users customize, script, and test GUIs without needing access to the underlying source code. However, these tools were not designed to extract structure from screencast videos or to convert them to mixed-media tutorials. Note that while Sikuli uses OCR (optical character recognition) to extract text within screenshots to facilitate search, it does not try to systematically extract entire blocks of code and dynamic edit intervals. In contrast, Codemotion was designed specifically for extracting code from videos and uses a custom edge detection algorithm to detect GUI panes that likely contain code.

Systems such as Pause-and-Play [37] and Waken [1] extend these ideas by automatically detecting GUI elements within existing screencast videos. Since they operate on videos, they can identify both static GUI components and dynamic interactions such as mouse movements and icon clicks. Using this information, they create enhanced tutorial video players that are linked to the underlying state of featured applications. In contrast, instead of being aimed at general GUI app tutorial videos, Codemotion focuses specifically on computer programming tutorials where someone is writing code and demonstrating its run-time effects, so its algorithms focus solely on extracting and reconstructing source code. In the future, incorporating features from these systems could broaden Codemotion's scope to let it generate tutorial steps from portions of videos that are unrelated to writing code.

Much like how Codemotion focuses on programming videos, systems such as NoteVideo [32] and Visual Transcripts [39] also focus on a specific tutorial domain—in both cases, hand-sketched blackboard-style lecture videos popularized by Khan Academy's math tutorials [16]. They use computer vision techniques to extract strokes from pixel differences between video frames and then combine that data with time-aligned text transcripts to create searchable and skimmable mixed-media tutorials from raw videos. These systems share our design philosophy of focusing on a single domain so that the extraction algorithms and UI design are tightly targeted.

Finally, in the broader computer vision literature, there is ongoing work in both content-based image retrieval [26] and content-based video search [44], which aim to automatically extract semantic meaning from arbitrary images and videos, respectively These techniques target real-world imagery and are far more general than what is required to extract digital state from computer-based screenshots and screencasts. Also, they are not tuned for recognizing text-based content by default, so it would be impractical to try to adapt them to work on computer programming videos. Codemotion is a specialized instance of such techniques tuned for recognizing code.

Extracting Source Code from Screencast Videos

The closest related work to Codemotion are CodeTube [38] and Ace [42], which both extract source code from videos using a similar high-level approach by finding code-containing segments, running OCR, and then inferring edit intervals. Both algorithms differ from Codemotion's in various low-level details, but the most significant high-level difference is that Codemotion's Stage 3 preserves actual contents and diffs of source code within edit intervals, which is important for replaying the edits to users in real time. In contrast, both CodeTube and Ace extract only a raw dump of tokens necessary for indexing a search engine and do not wholly reconstruct the original code in a format that is presentable to end users.

More broadly, the main ways that our work differs from both of these projects is that: 1) Our code extraction algorithm was designed based on a formative study of videos selected for diversity in programming languages, visual layouts, and presentation styles (Table 1). In contrast, both CodeTube and Ace used a corpus of Java screencasts on Android mobile development, which may limit the breadth of videos that their algorithms perform well on. 2) Our paper additionally *makes an HCI research contribution by using the output of Codemotion to expand the design space of user interactions with programming screencast videos*. In contrast, both CodeTube and Ace were designed solely to extract text to index a search engine. CodeTube introduces a user interface that combines in-video and Stack Overflow search, while Ace has no UI.

Enhanced Video Players and Video-Based Tutorials

More broadly, our goal of designing new interactions with videos was inspired by prior work in enhanced video players and mixed-media tutorial systems. Systems such as Video Lens [31], Panopticon [20], LectureScape [22], Swifter [30], and SceneSkim [34] facilitate navigation and skimming

ID	Video Title	YouTube ID	# Views	Length	Language	Background Color	Other Content in Video
1	Print all size-K subsets from an array	C9gITdYbqeA	6,874	5:28	Java	White	PowerPoint Slides
2	JQuery Tutorial	BWXggB-T1jQ	179,866	1:14:25	JavaScript	Light	Web Browser
3	Angular 2 Routing and Navigation Basics	Uvj_7ZMrHmg	23,413	13:11	JavaScript	Dark	Web Browser
4	CS50 2016 - Week 8 - Python	5aP9Bl9hcqI	54,426	2:12:59	Python	Dark	Terminal, Talking Head
5	Learn PHP in 15 minutes	ZdP0KM49IVk	831,980	14:59	PHP	Dark	Web Browser
6	Google Python Class Day 2 Part 3	Nn2KQmVF5Og	185,025	25:50	Python	Light	Terminal, Talking Head
7	Python Web Scraping Tutorial 2	kPhZDsJUXic	94,621	9:10	Python	White	Terminal
8	Python Beginner Tutorial 1	cpPG0bKHYKc	1,468,008	9:08	Python	Both	Web Browser, Terminal
9	Python Programming	N4mEzFDjqtA	2,431,294	43:15	Python	White	Terminal
10	Introduction to D3	8jvoTV54nXw	136,957	1:38:16	JavaScript	White	Web Browser, Talking Head
11	D3.js tutorial - 1 - Introduction	n5NcCoa9dDU	222,927	5:42	JavaScript	White	Web Browser
12	Methods reverse and copy	CfJCamxV3NQ	104,564	7:22	Java	White	None
13	Programmieren in C Tutorial #03 (German)	x2BuLjK5ADk	39,453	4:04	C	Both	Terminal
14	Ruby Essentials for Beginners: Part 01	C4GyNxgmbgI	61,756	28:35	Ruby	White	Terminal
15	C# programming tutorial	OBsGRqXzOhk	745,091	1:32:11	C#	White	Terminal, PPT Slides
16	C# Tutorial - Circle Progress Bar	o7MGaf8YW6s	29,002	4:18	C#	White	GUI Component Designer
17	Coding Rails: Redirect actions	KjMJayAlSIQ	1,147	6:05	Ruby	Dark	Web Browser, PPT Slides
18	Starting a Report and Title Page	7HC9xEZsqdM	127,729	11:15	LaTeX	White	PDF viewer
19	Using make and writing Makefile	aw9wHbFTnAQ	169,362	20:45	make	Both	Terminal
20	Learning C Game Programming	FCRmIoX6PTA	163,938	29:55	C	White	Terminal

Table 1. The corpus of YouTube programming screencast videos that we used for our formative study and the quantitative evaluation of Codemotion.

through long videos or collections of videos, sometimes aided by time-aligned transcripts or domain-specific metadata.

Video Digests [35] and VideoDoc [24] let users semi-automatically create mixed-media text+video tutorials from existing lecture videos that contain text transcripts. Systems such as MixT [7], Chronicle [15], DocWizards [2], and those by Grabler et al. [14] and Lafreniere et al. [25] allow users to make mixed-media tutorials by demonstrating their actions within instrumented versions of software (e.g., image editors). ToolScape [23] uses a crowd-powered workflow where Mechanical Turk workers label the step-by-step structure of existing how-to tutorial videos. In contrast to these systems, Codemotion's computer vision-based technique is fully automatic and requires only the raw video itself as input.

FORMATIVE STUDY AND DESIGN GOALS

To understand variations in format amongst programming videos and to establish design goals for Codemotion, we performed a formative study by characterizing the properties of 20 such videos from YouTube and MOOCs. While we cannot be comprehensive, we strived to achieve diversity in featured programming languages, topics, lengths, popularity, visual layout, and presentation styles. Table 1 shows our corpus.

These videos are all structured as live coding sessions where the instructor demonstrates concepts by writing, running, and verbally explaining their code while recording their computer desktop. One reason why these videos might be so pervasive online is because they are far easier to create than text-based tutorials, since instructors can quickly record a screencast of their live coding and verbal narration without much editing; many appear to be done in a single take with errors intact.

One of our most salient visual observations was that these videos often feature split-screen views of both a code editor and an output window. This way, the instructor can simultaneously show themselves writing code and the effects of running that code. The output window can be as simple as a text terminal or as rich as a custom GUI application (e.g., for game programming) or a web browser (e.g., for web development). Windows also occasionally get moved around and resized throughout the video. When screen space is limited, the instructor will alternate between the code editor and output panes with all windows maximized, so only one is shown at once. Some videos also show a "talking head" [16] where a mini-video of the instructor's head is embedded within a frame in a corner. And those especially in MOOCs will intersperse live coding with presenting PowerPoint slides. Thus, despite these videos all being programming tutorials, many of the GUI windows featured within them do *not* contain code.

Even within the GUI windows that do contain code, there is still a lot of variation and noise. For instance, these windows often feature extraneous non-code elements such as menu items, borders, gutter line numbers and marks, highlighted lines, different background colors, and varying font styles. Users also write code with differing levels of spacing and indentation, so being able to automatically recognize those is critical for preserving code style (and also runtime semantics in whitespace-significant languages such as Python).

We also noticed that each video naturally partitions into several discrete "time intervals" of code that the instructor incrementally builds up and tests in sequence. For instance, a web programming tutorial may start with an interval of JavaScript code edits, followed by an interval of CSS, then HTML, then another JavaScript interval. Even single-language tutorials are organized into multiple intervals as the instructor implements and tests different code components throughout the video's duration. However, these intervals are rarely labeled in the video; the instructor simply finishes working on one part of the code base and then starts the next part right away.

Based on these observed properties of programming tutorial videos, we distilled a set of design goals for Codemotion:

- **D1**: It must be able to find video regions that contain code.
- **D2**: It must be able to reliably extract source code from those regions without picking up extraneous GUI noise.
- **D3**: It must be able to split a video into a set of meaningful time intervals based on chunks of related code edits.

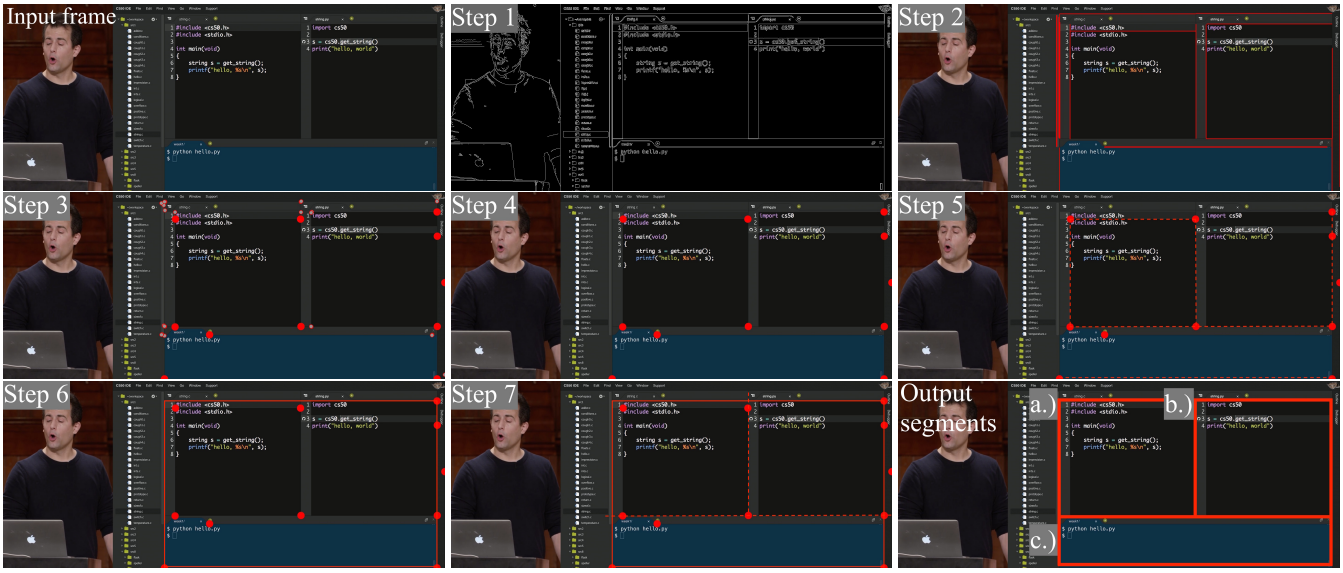

Figure 2. Example of running Codemotion's video frame segmenter (Stage 1) on a lecture video containing a talking head alongside an IDE with multiple text panes (ID=4 from our corpus). The lower-right image shows the output of this stage: three segments (a., b., c.) that possibly contain code.

THE CODEMOTION ALGORITHM

Codemotion is a three-stage pipeline that processes a video file with no manual intervention. At each stage, structured information is extracted from the video which not only aids the next stage but also powers new kinds of user interactions (described later in this paper). The main technical challenge was making the algorithm work effectively with a minimal set of assumptions about the contents of these screencast videos. While every video has a unique layout, we designed this code recognition pipeline with heuristics based on empirical observations from a formative study (previous section).

Stage 1: Finding Potential Code Segments Within Frames

Our first processing stage finds where code possibly resides within each video frame (Design Goal D1). Codemotion samples each second of the video by extracting the first frame of every second as an image. Since screen capture of code is not subject to rapid motion except in rare cases when a window is being dragged, choosing the "best" frame within each second is unimportant in practice. Sampling at a higher frequency is unlikely to bring significant gains at the expense of increased computational time. Finally, note that since each frame is processed independently, this stage is trivially parallelizable.

Each frame is a bitmap image of the video creator's computer desktop, which has a collection of GUI windows. The objective of this stage is to isolate regions within these windows (called "segments") that potentially contain code. This segmentation step is necessary because OCR engines are designed to work with images satisfying properties that make them look similar to high-quality scans of pages in a book – i.e., containing only text with a simple, consistent, and predictable layout, good contrast, no significant rotation or skew, and at least a minimum x-height for text [40]. Thus, when we initially tried to feed raw video frames into off-the-shelf OCR engines, the resultant output text was either empty, incomplete, or garbled.

Running the entire video frame through OCR does not work. Unfortunately even running images of individual GUI windows through OCR does not work either, due to UI elements such as menu items, buttons, and icons making those images not conform to a "book-like" layout of paragraphs of pure text that OCR engines expect. Also, even a single GUI window often contains multiple panes of independent content, such as an IDE showing several source code files side-by-side. Thus, we need a robust way to isolate each pane so that we can extract the code within each one as independent pieces of text.

We base our segmentation algorithm on the following empirical observation: code is usually written in left-aligned lines within a pane with distinct borders. Thus, our strategy is to identify critical horizontal and vertical lines that demarcate text regions and choose a rectangular crop that ignores background color differences like those due to highlighting of the currently-edited line. Our algorithm has seven steps and is implemented using Python bindings to OpenCV [19]. We use the example video frame in Figure 2 to illustrate how it works:

1. Use a Canny edge detector [5] to find all edges in the frame.
2. Since the Canny detector finds many extraneous edges that are not likely to be GUI pane borders, use a probabilistic Hough transform [29] to find the subset of edges that are horizontal or vertical straight line segments with a non-trivial length (shown in red in Figure 2, Step 2). Note that although humans can see a vertical edge at the right of the instructor's talking head in this frame, the Canny detector produces a set of short jagged edges rather than a single long vertical line segment due to the lack of a distinctive border. These are not false negatives since Codemotion cares only about finding code, which are usually contained within panes with clearly-marked borders.
3. Extract the two endpoints of each line segment. Since the border of each pane likely contains several parallel segments, their endpoints will appear together in clusters.

4. For each cluster of endpoints, take the one closest to the center of the image and discard the rest in its cluster. These are shown as large red dots in Steps 3 and 4 of Figure 2. We keep only these points because they are the most likely to lie at the innermost edges or corners of each GUI pane.
5. For each point found in Step 4, connect it to another point in a way that forms a nearly-horizontal or nearly-vertical line segment. Save these line segments for Step 7. (Note: some points will not be used here since they cannot connect to any other points to form horizontal/vertical segments.)
6. Using the same set of points in Step 4, find the smallest aligned rectangular crop around the convex hull of those points. Doing so gives the bounding box surrounding all GUI panes of interest, shown in red in Figure 2, Step 6.
7. Combine the line segments from Step 5 with the bounding box from Step 6 to split the image into rectangular segments. In Figure 2, the three extracted segments are labeled a, b, and c, respectively, in the "Output segments" image.

The output of this stage is a set of cropped rectangular segments for each video frame, some of which may contain code. In our example in Figure 2, the algorithm detected three segments: a.) the upper left segment contains C code, b.) the upper right segment contains Python code, and c.) the bottom segment is a terminal shell. Note that this algorithm is not guaranteed to find only code-containing segments; e.g., if the borders around the talking head in Figure 2 were more pronounced, then that would have been detected as a segment. However Stage 2 would find that it contains no code, so extraneous segments usually do not lead to false positives.

Stage 2: Extracting Source Code From Segments

Stage 1 produced a set of segments within each video frame. In this stage, Codemotion runs the Tesseract OCR engine [40] on each segment to extract text from it and then determines which text is likely to be code (Design Goal D2). Recall that Stage 1 is necessary since simply running OCR on the entire frame and even on individual GUI windows fails to produce meaningful text. In this stage, each segment image can be processed independently, so it is also trivially parallelizable. Its algorithm contains six steps:

1. *Upscaling*: OCR engines usually need text to be above a minimum size (i.e., x-height) to trigger detection, so Codemotion first upscales each segment image to 2X resolution before running Tesseract on it, which drastically improves recognition results in practice.
2. *Edge padding*: Sometimes text stretches all the way to the edge of a segment, which diminishes OCR quality for words near the edges. To improve recognition results, Codemotion adds a 2%-width padding around each segment filled with its background color.
3. *Run the Tesseract OCR engine* on the segment after upscaling and padding it. Tesseract produces as output the extracted text along with metadata such as style, layout information, and recognition confidence metrics.
4. *Post-process the text extracted by Tesseract* to format it according to heuristics that work well for source code: a.) Code sometimes appears in a segment with line numbers displayed on the left edge, which is common in IDEs and text editors. Thus, this step first eliminates left-aligned text that look like line numbers. b.) Tesseract sometimes produces smart quotes, so those are changed to regular quote marks, since regular quotes appear far more frequently in source code. c.) Finally, sometimes Tesseract improperly recognizes text within images as accented versions of ASCII characters, such as é, è, ê, or ë, due to noise in the image. Thus, as an approximating heuristic, all accented characters are converted to their unaccented ASCII counterparts, since ASCII is far more likely to appear in source code, especially in built-in keywords.
5. *Reconstruct indentation*: Unfortunately, the text produced by Tesseract does not preserve indentation. While this is usually not important for paragraphs of prose, it is essential for code because we want to preserve indentation-based coding style, and in the case of whitespace-significant languages such as Python, to also preserve run-time semantics. Fortunately, Tesseract's output contains metadata about the absolute position of each word, so Codemotion reconstructs indentation levels using these coordinates.
6. *Detect code*: Codemotion runs the post-processed text through the `language-classifier` tool [18] to determine whether it is likely to be code in a popular programming language. This tool is a Bayesian classifier trained on a large corpus of code from popular languages including Python, JavaScript, Ruby, and several C-like languages. If the classifier fails to recognize a language with sufficient confidence, then that segment is labeled as "plain text," which lets it detect pseudocode, documentation, and large blocks of code comments.

In sum, this stage turns each segment within each video frame into text formatted to look like source code, along with a label of its programming language (or "plain text" if none found).

Stage 3: Finding Code Edit Intervals

This final stage takes the extracted code for each segment across all video frames in which that segment appears and finds intervals of closely-related code edits (Design Goal D3). This stage is less parallelizable than stages 1 and 2, but it can still process each time-sorted list of segments in parallel. If implemented within a MapReduce workflow [9], this stage would be a reducer while stages 1 and 2 would be mappers.

The following algorithm runs independently for each segment across all video frames in which it appears:

1. *Inter-frame diffs*: Use `diff-match-patch` [12] to compute a diff of the segment's code between every consecutive frame. These inter-frame diffs capture regular code editing actions such as inserting, changing, and deleting code. These diffs also serve as the basis for the rest of the steps.
2. *Merging code due to scrolling*: In tutorial videos, the author often scrolls the code editor vertically so that slightly different lines of code are shown between frames. For example, in Figure 3 the author starts with line 1 at the top of the code editor (Figure 3a) and then scrolls down so that line 11 is at the top (Figure 3b). Codemotion merges

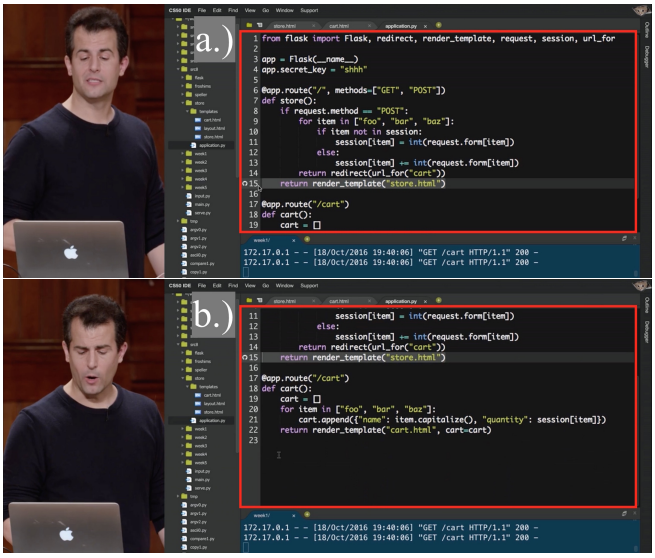

Figure 3. Two consecutive video frames where the instructor scrolls the code editor: a.) starting with line 1 at the top of the editor, b.) then scrolling down so that line 11 is at the top. In Stage 3 of processing, Codemotion merges those two snippets of code into a unified block.

code from frames where scrolling likely occurred to produce one unified block of code. In Figure 3, that block is 22 lines long. This merging process is triggered when the inter-frame diff indicates that some lines of code have disappeared from the top while other lines have appeared at the bottom, or vice versa – both of which are likely due to scrolling. Note that if the user scrolls too far within one second, then Codemotion cannot merge the code in those frames because there are no lines in common. Although imperfect, this merge heuristic is vital for producing a unified block of code over a continuous series of frames instead of disjointed code snippets for each individual frame.

3. *Splitting a segment into code edit intervals*: Codemotion keeps collecting diffs and merging code due to scrolling (see above) until it reaches an interval boundary. At that point, it starts a new interval and continues processing. An interval boundary occurs when: a.) The programming language of the detected code changes, or b.) The inter-frame diff shows more than 70% of the lines differing. This could occur either due to the author switching to editing a different file, or scrolling too far too quickly. Note that since this algorithm uses diffs to split each segment into edit intervals, it is not affected by code-containing GUI panes being resized or moved across the screen in the video, as long as the code inside does not significantly change.

4. *Quick-switch optimization*: Sometimes the tutorial author is editing file A, switches to edit a different file B for a few seconds, and then switches back to file A. The default interval splitting algorithm will find three intervals: file A, file B, and a *new* interval upon returning to file A. What is more preferable is to merge the two file A intervals into a single longer interval. Codemotion implements this optimization by keeping the time elapsed since the prior interval, and if the intervening (e.g., file B) interval is short (e.g., less than 10 seconds), then file A's interval will keep accumulating again when the author switches back to it since Codemotion notices a small enough diff from the previously-seen frame for file A. (B's interval remains unchanged.) This optimization helps preserve the meaning of an "interval" as the author continuously editing a single piece of code, even if they momentarily switch to edit another file.

This stage's output is a set of edit intervals for each segment. Each interval contains all of the timestamped diffs needed to reconstruct all of its code edits, along with the full contents of the code in that interval (taking scrolling into account).

QUANTITATIVE EVALUATION OF CODEMOTION

To quantitatively evaluate the algorithm's performance, we ran Codemotion on the 20 videos from our formative study corpus (Table 1). Table 2 summarizes our results. Its total running time ("Run time" column) is proportional to the length of each video, and ranges from 0.9X to 8.9X. We did not make any attempts to optimize or parallelize (with, say MapReduce). Each video was processed by one CPU core.

"Avg. # segments" shows the time-weighted average number of segments in each video. For instance, if 5 minutes of a 20-min video had 1 segment and the remaining 15 minutes had 2, then the average would be 1.75. The averages in our corpus ranged from 0.94 to 2.17, which reflects the fact that most videos had either a single pane of code, a split view of code + text output, or two code editors. Averages can be less than 1.0 since some videos contained portions with no text segments (e.g., someone lecturing or drawing on the board).

"# raw intervals" shows the raw numbers of code edit intervals that Stage 3 found for each video, which is again proportional to video length. By default each of those would appear as a step in the generated tutorial (Figure 4), but we found it cumbersome to have dozens (or even hundreds) of short intervals without much information content interspersed with more substantive intervals. Thus, for our UI prototype (next section) we merged intervals so that each was at least 10 seconds long, which drastically cut down on the total number of intervals (i.e., tutorial steps) to an average of 2.5 per minute. It is hard to automatically discover an objectively "correct" number of steps for each tutorial, but the 10-second merge heuristic was enough to eliminate most trivially short steps.

Sources of Codemotion's extractor inaccuracies: There are two main sources of inaccuracies that negatively affect code extraction quality: not being able to find well-cropped code segments in the video (Stage 1) and errors in Tesseract's OCR (Stage 2). Inaccuracies in Stage 3 are not as serious since, as mentioned earlier, there is no objectively "correct" number of intervals, and finding more or fewer intervals does not lose meaningful information about the extracted code or diffs.

We report segment-finding inaccuracies in the "Segment finding miss %" column of Table 2. To compute this metric, we manually watched all raw intervals from all 20 videos with the Stage 1 segmenter output overlaid on them (see Figure 2). We count a code segment as "missing" if it is either not found at all by the algorithm within each 10-second window, or is found but is either incomplete or not closely-cropped

ID	Video length	Run time	Avg. # segments	# raw intervals	# 10s-merged intervals	# merged intervals per minute	Segment finding miss %	Estimated OCR error % (# chars in 5 random frames)
1	5:28	1.5X	0.95	29	9	1.65	11.07%	4.6% (2348)
2	1:14:25	8.9X	1.54	1187	342	4.60	14.16%	8.0% (988)
3	13:11	4.4X	1.23	189	21	1.59	9.74%	32.6% (1461)
4	2:12:59	1.7X	0.94	1210	343	2.58	6.38%	9.7% (349)
5	14:59	2.6X	1.34	31	21	1.40	7.19%	11.6% (725)
6	25:50	8.2X	1.05	465	115	4.45	1.94%	N/A
7	9:10	1.8X	1.06	27	13	1.42	0.00%	N/A
8	9:08	4.1X	1.01	67	19	2.08	0.00%	10.2% (1625)
9	43:15	1.8X	1.60	278	129	2.98	2.78%	7.8% (771)
10	1:38:16	2.3X	1.63	601	237	2.41	3.50%	8.2% (1654)
11	5:42	4.2X	1.08	45	18	3.16	7.97%	21.4% (388)
12	7:22	0.9X	0.99	38	14	1.90	14.02%	34.9% (743)
13	4:04	3.3X	1.69	51	16	3.93	13.33%	N/A
14	28:35	1.1X	1.05	11	9	0.31	0.00%	2.3% (2024)
15	1:32:11	1.5X	1.20	547	181	1.96	3.98%	5.8% (1007)
16	4:18	6.5X	1.79	96	23	5.35	4.00%	17.5% (1311)
17	6:05	2.5X	1.20	47	16	2.64	4.05%	3.9% (544)
18	11:15	1.8X	1.14	98	33	2.93	8.33%	22.1% (1124)
19	20:45	2.3X	1.73	55	14	0.67	0.00%	5.5% (1587)
20	29:55	2.0X	2.17	309	58	1.94	4.34%	6.4% (1685)
Avg	34:45	3.2X	1.32	677	207	2.50	**5.8%**	**11.2%** (20334)

Table 2. Results of running Codemotion to extract code from the videos in Table 1. Run times (non-parallelized) are shown as multiples of video length.

enough (e.g., includes window borders, menu items, or other edge chrome that diminish OCR accuracy). We then divide the number of missing segments by the total number of 10-second windows in the video to compute a miss percentage, which averages to 5.8% across all 20 videos. This means that Codemotion was able to find 94.2% of total segments.

Assuming that well-cropped segments are found by Stage 1, Stage 2 relies on Tesseract's OCR engine as a black box. No OCR will be perfect, and since developing an OCR engine is outside the scope of this paper, those inaccuracies are largely out of our control. We found three sources of inaccuracies: 1) We feed video frames into the OCR engine, which have far more compression artifacts and noise than screenshot images. 2) Most videos we found are 480p/720p, so small text is not sharp enough for recognition, even with upscaling. 3) Tesseract confuses similar-looking characters, such as recognizing "factorial()" as "factoria1()" (a common OCR problem).

To estimate OCR error rates, we randomly sampled 5 video frames from each video (100 total) and manually checked OCR outputs for accuracy. Out of 20,334 total characters in the ground truth across all 100 frames, 4.8% were misrecognized (e.g., '`l`' became '`1`'), 4% were missing in the OCR output, 0.4% were indentation errors (only semantically meaningful for languages like Python), and 2% had the OCR produce excess characters (e.g., '-' became '--'). This adds up to a total estimated OCR error rate of 11.2%. The "Estimated OCR %" column of Table 2 shows the error rate and number of ground truth sampled characters for each video's 5 frames. (Videos 6, 7, and 13 show "N/A" since their quality and text resolution were too low for OCR to work out of the box.)

UI PROTOTYPE FOR USER TESTING & IDEA ELICITATION

The output of Codemotion's algorithm is raw data consisting of source code and edit intervals for each video. To elicit qualitative user perceptions and participatory design ideas, we first needed to create a user interface to surface that raw data to users. Inspired by prior work in mixed-media tutorial formats that combine video and text modalities [7, 24, 35], we made a simple mixed-media UI prototype to support navigation and search through the extracted code and edit intervals.

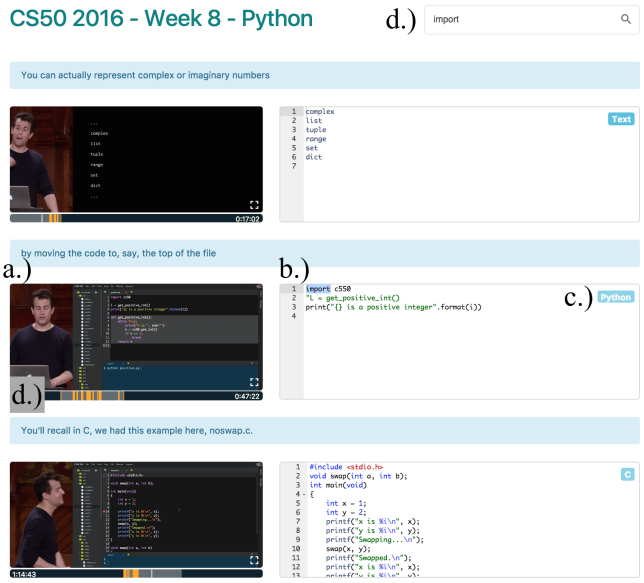

Figure 4. A prototype mixed-media interface that surfaces the data that Codemotion extracts from a video: a.) Each step shows a mini-video spanning a code edit interval and its transcript. b.) As each video plays, the code within it updates live on the right. c.) The auto-detected language is shown. If an interval contains multiple segments/languages, the user can click to switch between them. d.) A search box enables code search within videos; results highlighted in orange on video timelines.

Figure 4 shows an overview of our prototype interface. The original video is split into one mini-video for each code edit interval (found in Stage 3). Thus, each step in the UI shows the author making closely-related edits to a single unified piece of code. This allows the user to easily skim the entire video to see different conceptual phases of the tutorial, such as someone first editing HTML, then CSS, then JavaScript.

Each step in the UI is shown with its mini-video on the left and a code display box on the right. When the video is not being played, the code display box shows the total accumulated code at the *end* of that edit interval. This allows the user to quickly skim all of the code written in each step and to copy and paste it into their own IDE to play with it. It also serves as a concise summary for what happened in that step. A search box lets users find parts of the video where text occurs; search results appear as tick marks on video timelines (Figure 4d). When the user plays a step's mini-video, its accompanying code display box gets updated in real-time to reflect the code written so far up to that point in the video. Note that since Stage 3 merges code from consecutive frames to take scrolling into account, the code display box will show all of the code written so far in that edit interval, not only what is currently displayed on-screen in each video frame.

EXPLORATORY STUDY OF USER PERCEPTIONS

We wanted to obtain users' subjective impressions of Codemotion's output, so we used the prototype UI from Figure 4 to run an exploratory user study comparing it to YouTube.

Procedure

We recruited 10 CS students (3 female) each for a one-hour lab study. We had each participant watch 10-minute excerpts from two YouTube videos from Table 1 on Python and Ruby programming, IDs 4 and 14 respectively. We also had them interact with Codemotion-generated tutorial UIs for different (but similarly information-dense) 10-minute excerpts from those same two videos. Although each participant saw both conditions, we alternated the order of exposure to YouTube's and Codemotion's UIs. As they were watching each 10-minute excerpt, we told them to try to learn the material in whatever way they normally would while thinking aloud to report their perceptions; they were free to use the lab computer to write code, run code, and take notes. Our observations and post-study interviews revealed two main themes:

Perceptions of Codemotion's Edit Intervals

Although we did not rigorously measure engagement, our sense was that participants tended to passively watch the entire 10-minute YouTube video excerpts like a lecture. In contrast, participants appeared more actively engaged when watching the short videos representing edit intervals extracted by Codemotion and following along with the respective code edits being "mirrored" in real time in the accompanying code editor boxes (Figure 4b). They often copied that code into a REPL or text editor to execute. We also saw that because they had to explicitly hit "Play" on the usually-short videos within each step, that forced them to pause to reflect and try out the code snippets rather than passively watching on YouTube.

Participants often fixated on watching the code editor box update in real time while listening to the author's narration from the accompanying video. P9 reported that doing so had the advantage of showing him *all* the code in an interval, not just what is currently on-screen, since Codemotion merges accumulated code upon screen scrolling (Figure 3). He also appreciated the "stability" of studying code in the editor as the video played, since the code remains still even when there is UI scrolling or active window changes in the video.

As expected, there was not universal agreement on what constituted optimal edit interval boundaries. Everyone had their own preferences in terms of interval lengths, but P2 mentioned offhand that many of the video clips seemed to end right when he was starting to feel restless. P5 said he did not mind if there were too many or too few intervals since he could easily see an overview of all intervals in the tutorial UI and quickly skip to another one when the current one got boring or did not contain enough information density; he called it being able to "skip the commercials."

Perceptions of Extracted Source Code Quality

All participants noticed occasional OCR inaccuracies, but they found them easy to fix when copy-pasting or re-typing the code into an external text editor to execute. When asked about these inaccuracies, a common mental model they conveyed was making an analogy to YouTube's automatic captioning tool for generating transcripts from videos via speech recognition. Just like with automatic captioning, they did not expect code to be perfectly extracted from videos. Participants felt that having the general flavor of code components (even with misspellings) was enough to support copy-paste and adaptation. In fact, P9 actually thought that OCR mistakes were human transcription errors since he assumed that a person had created the Codemotion tutorials. Finally, even though we did not design Codemotion to be able to directly execute the extracted code due to real-world code having hidden dependencies that are hard to fully capture from a video clip, several participants said that the extracted code looked good enough to run and wanted to see an "execute" button.

Study Limitations

Some of these user perceptions could be due to our specific UI design. However, the alternative to making a UI would have been to show users a dump of the raw textual output, which would not capture a realistic usage scenario. Thus, we decided to build a simple UI that could showcase all of the extracted code-related features. Also, since this was a small exploratory study, we cannot garner insights about how users would naturally interact with such data in the wild.

PARTICIPATORY DESIGN OF NEW VIDEO INTERACTIONS

The prototype interface that we created (Figure 4) is only one possible starting point in the design space of new screencast video interactions enabled by Codemotion. Participants in our exploratory user study had divergent opinions about what they wanted to see in a user interface. Inspired by their perspectives, we sought to solicit additional input from a potential user population of students to expand our design space.

Procedure

To elicit interaction ideas from students who are now learning programming, we conducted 4 participatory design workshop sessions [33] at our university, each held with a group of 3 students in a conference room with whiteboards, paper, pens, and markers. There were 12 total participants (3 female) from majors such as CS, cognitive science, and communications.

Code	Navigation	Search	Active Learning
Inline code annotation (3)⋆	Labeled timeline (2)	Related code finder (2)⋆	In-video exercises (3)⋆
Inline code editing (2)⋆	Tabbed navigation (2)⋆	Pop-up video search (2)⋆	Check-your-answers mode (3)
Video-within-code (2)⋆	Output-based navigation (2)⋆	Stack Overflow search (4)⋆	Anchored discussions (1)
Code-to-video (2)⋆	Table of contents (2)⋆	Video mash-ups (2)⋆	Pop-up hints (2)
Audio clips (1)	Code-based skimming (2)⋆	Internationalized search (1)	Choose-your-own-adventure (1)
REPL sidebar (1)	Cross-video links (2)⋆		
File tabs (2)⋆	Sub-interval diffs (1)		
Browser devtools (2)⋆	Panopticon (1)		
Extract-to-X (3)⋆	Mixed-media (6)		

Table 3. The 28 video interaction ideas developed during 4 participatory design workshop sessions with Codemotion. The numbers in parentheses represent how many different participants came up with that idea, and ⋆ means that participants in more than one session came up with that idea.

We began each one-hour workshop by having all participants talk about the current resources they use for learning programming, which was both an icebreaker and primed them to think about learning resources. Then we showed them several screencast videos from our corpus (Table 1) and introduced Codemotion. In two of the sessions, we showed participants our prototype UI (Figure 4) to help ground their ideation using a concrete artifact but encouraged them to come up with divergent ideas that do not resemble our UI. In the other two sessions, we did *not* show them our UI and instead sketched the capabilities of Codemotion on the whiteboard.

After this 15-minute orientation, we had participants come up with interaction design ideas on their own for 15 minutes either on paper or on the whiteboard. We encouraged them to come up with as many different unrelated ideas for interacting with the data that Codemotion provides rather than fixating on refining any specific idea (i.e., branching outward to try to "get the right design" instead of iterating to "get the design right" [4]). We then had them share all designs with each other for 15 minutes and then do another 15-minute round of individual brainstorming, this time encouraging them to be inspired by ideas they had just seen from other participants.

Participant-Created Designs

Table 3 summarizes the 28 design ideas that participants came up with, which we grouped into 4 categories. Most of these ideas were either generated by more than one participant (75%) and/or by participants in multiple sessions (61%). On average, each participant generated 4.9 different design ideas.

The first category of ideas involved **interacting with code**:

Inline code annotation: Show a textual overlay of code being written in real time directly on top of the video itself (similar to VidWiki's UI [8]) and let users annotate it with their notes. If there are typos or errors in that code, then the user can click a button to send feedback directly to the video's creator.

Inline code editing: Same as above except make that code editable, compilable, and executable so that the video essentially has an embedded IDE overlaid on top of it.

Video-within-code: Some wanted to see all of the extracted code as a whole instead of watching the creator incrementally write it in the video. They suggested a picture-in-picture view where they could see all of the code at once and have a small video playing in the corner embedded within the code display.

Code-to-video: Show all of the extracted code at once in an IDE-like interface. When the user highlights a selection in the IDE, play the portion of video where that piece of code was first written, which could explain why it was written.

Audio clips: Same as above, except instead of playing the appropriate video clip whenever a piece of code is selected in the IDE, only play the audio to avoid visual overload.

REPL sidebar: Put a REPL (read-eval-print loop) prompt beside the video so that after executing the code within the video, users can interactively build upon it using the REPL.

File tabs: Since many videos show code from multiple files, create a tab for each file and allow users to create new files.

Browser devtools: For web development tutorial videos, directly integrate the extracted code with the browser's developer tools (devtools) to use its visual inspector and debugger.

Extract-to-X: Add buttons to extract code in a section of the video to a downloadable file or to the clipboard.

Participants also came up with ideas for **video navigation**:

Labeled timeline: Annotate the video scrubber with visual indicators of intervals or summaries of code in those intervals.

Tabbed navigation: Split each edit interval into a separate video in its own tab. To facilitate skimming, label each tab with an automatically-generated summary of its transcript.

Output-based navigation: Extract and show the visual output that results from executing the code at the end of each edit interval (e.g., a webpage animation or graphical output) as thumbnails to help users navigate within the video.

Table of contents: Automatically summarize the transcript and code within each interval to create a table of contents for fast video navigation (reminiscent of LectureScape [22]).

Code-based skimming: Prominently display the extracted code in each interval and use the code as cues to quickly skim through the video.

Cross-video links: Many videos are multi-part within a playlist, so analyze all code and create cross-video links to enable users to jump to instances of similar code across videos.

Sub-interval diffs: Some wanted to see only the diffs within each interval to see what new code has been added (or modified/deleted), which serves as a summary to ease navigation.

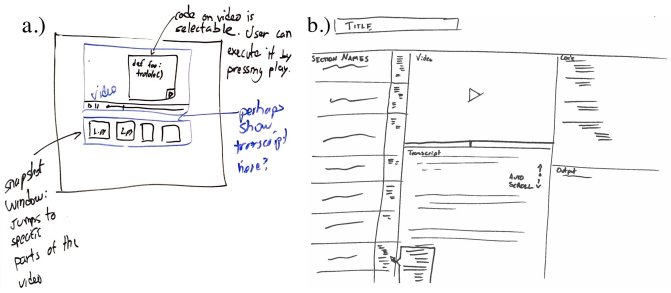

Figure 5. Two sketches from workshop participants: a.) inline code editing with labeled timeline, b.) mixed-media format with table of contents.

Panopticon: Show 2-D array of videos or code snippets to see an overview of many parts, similar to Panopticon paper [20].

Mixed-media: Variants of mixed-media formats [7, 24, 35].

The third category of ideas relate to **video search**:

Related code finder: Search for other videos containing similar code as what is shown in the current portion of the video.

Pop-up video search: Highlight a portion of code in an IDE to perform a contextual search based on both that code and the user's query to find relevant screencast videos. This idea is similar to Blueprint [3], except applied to video search.

Stack Overflow search: Find relevant Stack Overflow posts that refer to the code being demonstrated in particular parts of the video. This idea is similar to CodeTube's UI [38].

Video mash-ups: Search for particular concepts or code, and the system stitches together relevant excerpts from multiple videos into a joint video that satisfies those search conditions.

Internationalized search: Find videos in different languages (e.g., Chinese, Korean) that showcase similar code examples.

The final category of ideas encourage **active learning** [13]:

In-video exercises: Generate custom exercises based on the code in each part of the video or generic questions like "try to replicate what the author wrote in this interval." To get users started, give them skeleton code from the start of that interval.

Check-your-answers mode: After finishing each exercise from the above idea, replay the video interval with the user's submitted solution overlaid on top of the original creator's demonstrated code. Display diffs between the user's code and the solution code to highlight what they did incorrectly.

Anchored discussions: Embed discussion threads within each video interval so that students can reflect on and talk about each interval with one another. This idea is similar to prior work in anchored discussions [17, 45] but applied to video.

Pop-up hints: Show lightweight contextually-relevant hints as pop-up bubbles within parts of the video as it is playing, featuring definitions of technical jargon or relevant API documentation snippets. This is like Idea Garden [21] for videos.

Choose-your-own-adventure: Analyze code and transcripts in videos to create a knowledge graph and make a wizard-like "choose-your-own-adventure" UI where learners can take different non-linear paths through a corpus of video snippets.

DISCUSSION

Even amongst the limited student participant pool in our four participatory design sessions, there was still wide variation in interaction design ideas along several dimensions: a) Some wanted video-centric interfaces, while others wanted code-centric interfaces. b) Some wanted tight focus on one key element (e.g., the video player in Figure 5a) while others preferred a more "holistic" UI with no predominant focus (e.g., Figure 5b). c.) Some wanted to support the experience of watching entire videos as a whole while others wanted to use chopped-up video snippets as supporting features in an IDE. d.) Some wanted to incrementally improve upon existing video players while others proposed radically different interaction modes (e.g., *choose-your-own-adventure*). In sum, there was no "one-size-fits-all" design that suited all needs.

Note that not all of these participant-generated ideas are wholly original; in fact, many reminded us of components within prior research systems, which we cited in their respective summaries. However, it was interesting that students who were probably not aware of these prior systems (since they are not HCI researchers) independently generated such ideas. In addition, the data provided by Codemotion let students come up with these ideas in the context of programming screencasts, which was previously impractical to do. The next step here is to curate some of these piecemeal ideas together into complete user interface prototypes to implement and test.

Beyond these specific ideas, we feel that Codemotion opens exciting opportunities for improving the accessibility of programming videos for visually impaired learners. For example, one could imagine postprocessing Codemotion's outputs with a custom text-to-speech algorithm that combines data from transcripts, code contents, and time-aligned code edits.

CONCLUSION

The main contribution of this paper is expanding the design space of user interactions with programming screencast videos by surfacing the code latent within existing videos. To this end, we created Codemotion, a computer vision-based algorithm that automatically extracts source code and edit intervals from existing computer programming screencast videos. A quantitative assessment showed that it can find 94.2% of code-containing segments with an OCR error rate of 11.2%. An exploratory user study on 10 students found that users felt code interval lengths looked reasonable and that occasional OCR errors were acceptable. Four participatory design workshop sessions with 12 total students found that they generated 28 ideas for enhanced screencast experiences, including code, navigation, search, and active learning interactions.

In an idealized imagined future, everyone would record computer programming tutorials with detailed metadata about all of the constituent source code, edit histories, outputs, and provenance so that these tutorials are not simply raw pixels stuck within video files. However, in the current real world, screencast videos are one of the most convenient and most pervasive ways to record dynamic computer-based tutorials, so millions of such videos now exist on sites such as YouTube and MOOCs. This paper's contributions work toward helping learners unlock the insights hidden within their pixels.

REFERENCES

1. Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2012. Waken: Reverse Engineering Usage Information and Interface Structure from Software Videos. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 83–92. DOI: http://dx.doi.org/10.1145/2380116.2380129
2. Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. 2005. DocWizards: A System for Authoring Follow-me Documentation Wizards. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST '05)*. ACM, New York, NY, USA, 191–200. DOI: http://dx.doi.org/10.1145/1095034.1095067
3. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 513–522. DOI: http://dx.doi.org/10.1145/1753326.1753402
4. Bill Buxton. 2007. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
5. John Canny. 1986. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8, 6 (June 1986), 679–698. DOI: http://dx.doi.org/10.1109/TPAMI.1986.4767851
6. Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1535–1544. DOI: http://dx.doi.org/10.1145/1753326.1753555
7. Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. 2012. MixT: Automatic Generation of Step-by-step Mixed Media Tutorials. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 93–102. DOI: http://dx.doi.org/10.1145/2380116.2380130
8. Andrew Cross, Mydhili Bayyapunedi, Dilip Ravindran, Edward Cutrell, and William Thies. 2014. VidWiki: Enabling the Crowd to Improve the Legibility of Online Educational Videos. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '14)*. ACM, New York, NY, USA, 1167–1175. DOI: http://dx.doi.org/10.1145/2531602.2531670
9. Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1251254.1251264
10. Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1525–1534. DOI: http://dx.doi.org/10.1145/1753326.1753554
11. Mathias Ellmann, Alexander Oeser, Davide Fucci, and Walid Maalej. 2017. Find, Understand, and Extend Development Screencasts on YouTube. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics (SWAN 2017)*. ACM, New York, NY, USA, 1–7. DOI: http://dx.doi.org/10.1145/3121257.3121260
12. Neil Fraser. 2012. Diff, Match and Patch libraries for Plain Text. https://code.google.com/p/google-diff-match-patch/. (2012).
13. Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences* 111, 23 (2014), 8410–8415. DOI: http://dx.doi.org/10.1073/pnas.1319030111
14. Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating Photo Manipulation Tutorials by Demonstration. In *ACM SIGGRAPH 2009 Papers (SIGGRAPH '09)*. ACM, New York, NY, USA, Article 66, 9 pages. DOI: http://dx.doi.org/10.1145/1576246.1531372
15. Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. ACM, New York, NY, USA, 143–152. DOI: http://dx.doi.org/10.1145/1866029.1866054
16. Philip J. Guo, Juho Kim, and Rob Rubin. 2014. How Video Production Affects Student Engagement: An Empirical Study of MOOC Videos. In *Proceedings of the First ACM Conference on Learning @ Scale Conference (L@S '14)*. ACM, New York, NY, USA, 41–50. DOI: http://dx.doi.org/10.1145/2556325.2566239
17. Mark Guzdial and Jennifer Turns. 2000. Effective Discussion Through a Computer-Mediated Anchored Forum. *Journal of the Learning Sciences* 9, 4 (2000), 437–469.
18. TJ Holowaychuk. 2012. Programming language classifier for node.js. https://github.com/tj/node-language-classifier. (2012).

19. Itseez. 2015. Open Source Computer Vision Library. https://github.com/itseez/opencv. (2015).

20. Dan Jackson, James Nicholson, Gerrit Stoeckigt, Rebecca Wrobel, Anja Thieme, and Patrick Olivier. 2013. Panopticon: A Parallel Video Overview System. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 123–130. DOI: http://dx.doi.org/10.1145/2501988.2502038

21. William Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, Andrew Ko, Christopher J. Mendez, and Alannah Oleson. 2017. General principles for a Generalized Idea Garden. *Journal of Visual Languages & Computing* 39 (2017), 51 – 65. DOI: http://dx.doi.org/10.1016/j.jvlc.2017.04.005 Special Issue on Programming and Modelling Tools.

22. Juho Kim, Philip J. Guo, Carrie J. Cai, Shang-Wen (Daniel) Li, Krzysztof Z. Gajos, and Robert C. Miller. 2014a. Data-driven Interaction Techniques for Improving Navigation of Educational Videos. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 563–572. DOI: http://dx.doi.org/10.1145/2642918.2647389

23. Juho Kim, Phu Tran Nguyen, Sarah Weir, Philip J. Guo, Robert C. Miller, and Krzysztof Z. Gajos. 2014b. Crowdsourcing Step-by-step Information Extraction to Enhance Existing How-to Videos. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 4017–4026. DOI: http://dx.doi.org/10.1145/2556288.2556986

24. Rebecca P. Krosnick. 2014. *VideoDoc: Combining Videos and Lecture Notes for a Better Learning Experience*. Master's thesis. MIT Department of Electrical Engineering and Computer Science, Cambridge, MA.

25. Ben Lafreniere, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2014. Investigating the Feasibility of Extracting Tool Demonstrations from In-situ Video Content. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 4007–4016. DOI: http://dx.doi.org/10.1145/2556288.2557142

26. Ying Liu, Dengsheng Zhang, Guojun Lu, and Wei-Ying Ma. 2007. A survey of content-based image retrieval with high-level semantics. *Pattern Recognition* 40, 1 (2007), 262 – 282. DOI: http://dx.doi.org/10.1016/j.patcog.2006.04.045

27. Laura MacLeod, Andreas Bergen, and Margaret-Anne Storey. 2017. Documenting and Sharing Software Knowledge Using Screencasts. *Empirical Softw. Engg.* 22, 3 (June 2017), 1478–1507. DOI: http://dx.doi.org/10.1007/s10664-017-9501-9

28. Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 2015. Code, Camera, Action: How Software Developers Document and Share Program Knowledge Using YouTube. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 104–114. http://dl.acm.org/citation.cfm?id=2820282.2820297

29. J. Matas, C. Galambos, and J. Kittler. 2000. Robust Detection of Lines Using the Progressive Probabilistic Hough Transform. *Comput. Vis. Image Underst.* 78, 1 (April 2000), 119–137. DOI: http://dx.doi.org/10.1006/cviu.1999.0831

30. Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2013. Swifter: Improved Online Video Scrubbing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 1159–1168. DOI: http://dx.doi.org/10.1145/2470654.2466149

31. Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2014. Video Lens: Rapid Playback and Exploration of Large Video Collections and Associated Metadata. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 541–550. DOI: http://dx.doi.org/10.1145/2642918.2647366

32. Toni-Jan Keith Palma Monserrat, Shengdong Zhao, Kevin McGee, and Anshul Vikram Pandey. 2013. NoteVideo: Facilitating Navigation of Blackboard-style Lecture Videos. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 1139–1148. DOI: http://dx.doi.org/10.1145/2470654.2466147

33. Michael J. Muller and Sarah Kuhn. 1993. Participatory Design. *Commun. ACM* 36, 6 (June 1993), 24–28. DOI: http://dx.doi.org/10.1145/153571.255960

34. Amy Pavel, Dan B. Goldman, Björn Hartmann, and Maneesh Agrawala. 2015. SceneSkim: Searching and Browsing Movies Using Synchronized Captions, Scripts and Plot Summaries. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 181–190. DOI: http://dx.doi.org/10.1145/2807442.2807502

35. Amy Pavel, Colorado Reed, Björn Hartmann, and Maneesh Agrawala. 2014. Video Digests: A Browsable, Skimmable Format for Informational Lecture Videos. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 573–582. DOI: http://dx.doi.org/10.1145/2642918.2647400

36. Elizabeth Poché, Nishant Jha, Grant Williams, Jazmine Staten, Miles Vesper, and Anas Mahmoud. 2017. Analyzing User Comments on YouTube Coding Tutorial Videos. In *Proceedings of the 25th International*

Conference on Program Comprehension (ICPC '17). IEEE Press, Piscataway, NJ, USA, 196–206. DOI: http://dx.doi.org/10.1109/ICPC.2017.26

37. Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. 2011. Pause-and-play: Automatically Linking Screencast Video Tutorials with Applications. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 135–144. DOI: http://dx.doi.org/10.1145/2047196.2047213

38. Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Too Long; Didn't Watch!: Extracting Relevant Fragments from Software Development Video Tutorials. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 261–272. DOI: http://dx.doi.org/10.1145/2884781.2884824

39. Hijung Valentina Shin, Floraine Berthouzoz, Wilmot Li, and Frédo Durand. 2015. Visual Transcripts: Lecture Notes from Blackboard-style Lecture Videos. *ACM Trans. Graph.* 34, 6, Article 240 (Oct. 2015), 10 pages. DOI: http://dx.doi.org/10.1145/2816795.2818123

40. R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02 (ICDAR '07)*. IEEE Computer Society, Washington, DC, USA, 629–633. http://dl.acm.org/citation.cfm?id=1304596.1304846

41. Michael B. Twidale. 2005. Over the Shoulder Learning: Supporting Brief Informal Learning. *Comput. Supported Coop. Work* 14, 6 (Dec. 2005), 505–547. DOI: http://dx.doi.org/10.1007/s10606-005-9007-7

42. Shir Yadid and Eran Yahav. 2016. Extracting Code from Programming Tutorial Videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, New York, NY, USA, 98–111. DOI: http://dx.doi.org/10.1145/2986012.2986021

43. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '09)*. ACM, New York, NY, USA, 183–192. DOI: http://dx.doi.org/10.1145/1622176.1622213

44. Shoou-I Yu, Lu Jiang, Zhongwen Xu, Yi Yang, and Alexander G. Hauptmann. 2015. Content-Based Video Search over 1 Million Videos with 1 Core in 1 Second. In *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval (ICMR '15)*. ACM, New York, NY, USA, 419–426. DOI: http://dx.doi.org/10.1145/2671188.2749398

45. Sacha Zyto, David Karger, Mark Ackerman, and Sanjoy Mahajan. 2012. Successful Classroom Deployment of a Social Document Annotation System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 1883–1892. DOI: http://dx.doi.org/10.1145/2207676.2208326