Codemotion: Automatically Generating Mixed-Media Tutorials from Computer Programming Screencast Videos

anonymized for submission

ABSTRACT

Screencast videos are now a popular format for computer programming tutorials since they convey the dynamic effects of editing and running code; millions of such videos exist on sites such as YouTube. However, videos are much harder to search and navigate through than text-based tutorials because they simply consist of pixels with no semantic meaning. To help learners search and navigate through code-based tutorial videos, we have developed Codemotion, a computer-vision-based system that automatically extracts source code from existing screencast videos and transforms those videos into step-by-step mixed-media tutorials. Specifically, Codemotion segments a video into regions that likely contain code, performs OCR on those segments, post-processes with heuristics to recognize and clean up source code, and merges together related code edits into contiguous intervals. It uses all of this data to create a web-based mixed-media tutorial interface that facilitates quick search, skimming, and navigation. A quantitative assessment on a corpus of 20 YouTube programming videos shows that Codemotion can find approximately 94% of code-containing segments, and a qualitative user study shows that students preferred learning programming from Codemotion-generated tutorials over watching the corresponding raw YouTube videos.

Author Keywords

mixed-media tutorial; video tutorial; computer programming

ACM Classification Keywords

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

INTRODUCTION

Screencast videos are now a popular format for tutorials [23], especially for computer-based topics such as programming. For instance, a search for "computer programming tutorial" on YouTube returns 6.6 million videos. Figure 1 shows four examples of these kinds of videos, which span a variety of domains such as command-line, web, and game programming.

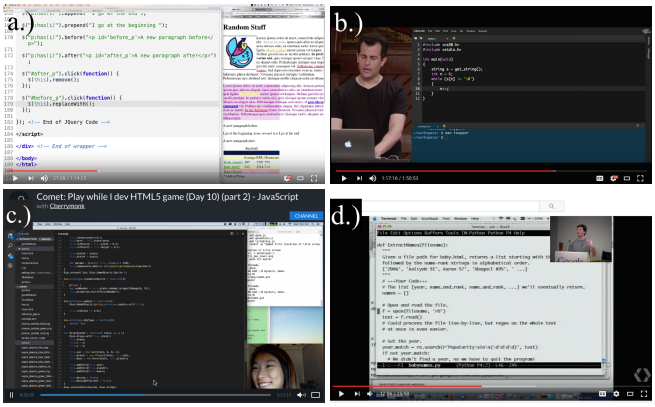

Figure 1. Four examples of computer programming tutorial videos: a.) jQuery web development screencast, b.) live coding during a university lecture, c.) a professional game developer live-streaming their work at `livecoding.tv`, d.) recording of a Python workshop at Google.

Video tutorials for computer programming have several advantages over text-based tutorials: Videos convey the dynamic effects of running code such as GUI animations and user interactions [23]. 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." [26] Finally, creators can easily record videos in-situ as they write code and narrate without breaking their flow [18].

However, videos are much harder to search for and navigate through [15, 17, 22] than text. 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 sometimes (if available) transcripts. Once you do find a promising video, it is hard to navigate to the point where those functions are called. 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 all these limitations is that the code within tutorial videos are simply *dead pixels* with no semantic meaning.

What if we could revive those dead pixels into live pieces of code so that millions of existing programming tutorial videos can be easily searchable and browsable? This paper explores that question by introducing a computer vision approach that automatically extracts source code from existing videos and then uses that data to transform these videos into *mixed-media tutorials* [6] with affordances for search and navigation.

We implemented our approach in a prototype system called *Codemotion*, which has two main components:

Submitted to UIST 2017

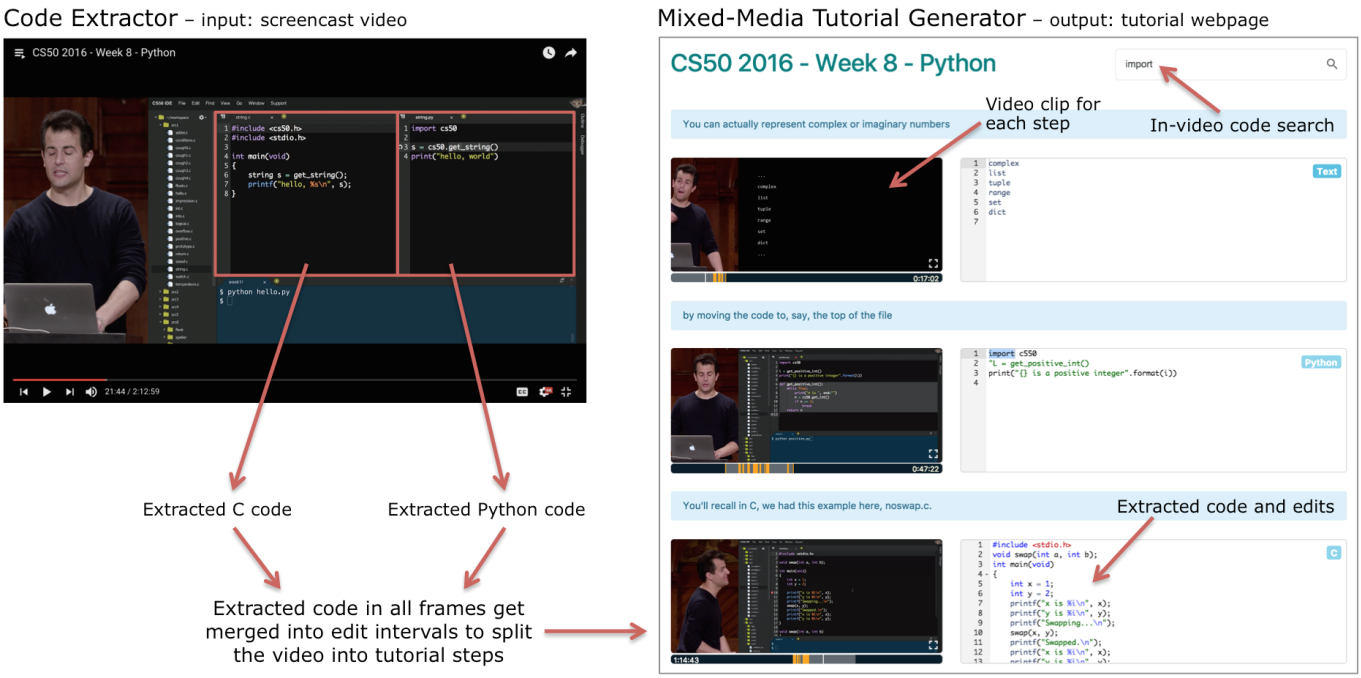

Figure 2. Codemotion automatically extracts code contents and edits from existing screencast videos and turns them into mixed-media tutorials.

- **Code Extractor**: Codemotion introduces a novel three-stage extraction algorithm 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.

  The main technical challenge here is how to reliably recognize source code from existing screencast videos with no assumed metadata.

- **Mixed-Media Tutorial Generator**: Codemotion uses the data produced by the code extractor to automatically transform a video into a mixed-media tutorial [6] which combines video and text in each step.

  Specifically, the original video is split into chunks based on intervals of closely-related code edits. Each video chunk is displayed alongside the contents of the associated code and, if available, the corresponding parts of the transcript. Each code display box updates in sync as the corresponding video clip plays. This design eases navigation, skimming, and copy-and-paste of code. The user can also perform freetext search on code that appears within the video.

To our knowledge, *Codemotion is the first attempt to automatically generate mixed-media tutorials from computer programming screencast videos.*

If Codemotion were deployed to a site such as YouTube, it could enable learners to access the latent structure beneath the dead pixels of millions of existing code-related videos, which lets them benefit from the best of both video and text modalities. For instance, someone learning web programming using Codemotion could search for a particular jQuery animation function, jump to the part of a relevant video that showcases this function, watch the video segment to see the animation demonstrated in action, and then copy-and-paste the code into their text editor to try it out for themselves.

More broadly, Codemotion opens future opportunities for improving the accessibility of coding tutorial videos for visually impaired learners. For example, one could imagine postprocessing the mixed-media tutorials generated by Codemotion with a custom text-to-speech algorithm that combines data from transcripts, code contents, and time-aligned code edits.

To assess the performance of Codemotion's automatic code extractor, we ran it on a corpus of 20 programming tutorial videos ranging from 4 minutes to 2 hours in length. It was able to find approximately 94% of code-containing segments across all of these videos. To evaluate the mixed-media tutorial generator, we ran a user study where 10 students learned Python and Ruby programming both from Codemotion-generated tutorials and by watching the corresponding YouTube videos. All 10 subjects preferred Codemotion over YouTube and cited benefits such as easier search, step-by-step navigation, and tracking of code edits over time.

The contributions of this paper are:

- A novel computer-vision-based algorithm to extract source code and edit intervals from screencast videos.
- Leveraging this algorithm to automatically generate mixed-media tutorials that make it easier to search and navigate through existing programming tutorial videos.
- A quantitative evaluation showing the accuracy of the algorithm and a qualitative user study showing that users preferred the tutorials it generates over ordinary videos.

RELATED WORK

Codemotion is situated at the intersection of two lines of HCI research: systems for mixed-media tutorials and for automatically extracting state from screenshots and videos.

Mixed-Media Tutorials that Combine Text and Video

The design of the Codemotion tutorial format was inspired by prior work on *mixed-media tutorials* that combine static (text+image) and video modalities. These tutorials are formatted as a series of steps on a webpage with a mix of textual exposition and embedded mini-videos at each step. Chi et al. [6] performed a comparative study of static, video, and mixed-media tutorials for image manipulation tasks and discovered that users found mixed-media tutorials the easiest to follow and the least error-prone. Based on this study, they proposed several design guidelines for mixed-media tutorials that Codemotion adapts for the domain of computer programming videos: it automatically segments a video into skimmable steps based on what kind of code is being written within each segment, it displays video segments alongside the extracted code and keeps them in sync, and it provides affordances for code search and navigation within video segments.

Inspired by this format, the Video Digests [22] and VideoDoc [17] systems semi-automatically generate mixed-media tutorials from existing online lecture videos that contain time-aligned text transcripts. ToolScape [16] takes a crowdsourcing-based approach by introducing a crowd-powered workflow for using Mechanical Turk workers to extract and label the step-by-step structure of how-to tutorial videos. In contrast to these projects, which require pre-made transcripts or crowd workers, respectively, Codemotion's computer-vision-based tutorial generation technique is fully automatic and requires only the raw video itself as input.

Systems such as MixT [6], Chronicle [12], DocWizards [3], and those by Grabler et al. [11] and Lafreniere et al. [18] allow users to make video-like mixed-media tutorials from scratch by demonstrating their actions within instrumented versions of specific applications (e.g., image editors). In contrast to these projects, Codemotion automatically generates mixed-media tutorials out of existing online videos that were not recorded within specially instrumented environments.

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 [5, 27] and Prefab [8] use computer vision to automatically identify GUI elements from screenshots, which enables users to 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 [23] and Waken [2] 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 presenting code alongside the video. 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 [21] and Visual Transcripts [24] also focus on a specific tutorial domain – in their case, hand-sketched blackboard-style lecture videos popularized by Khan Academy's math tutorials [13]. 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 [19] and content-based video search [28], which aim to automatically extract semantic meaning from arbitrary images and videos, respectively (e.g., "find which parts of this video contain a brown spotted dog"). 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. Thus, Codemotion can be viewed as designing a specialized instance of such techniques for recognizing code in particular.

In sum, Codemotion follows in the tradition of HCI systems that enhance tutorials by extracting state from screenshots and screencast videos, but to our knowledge, it is the first to focus on the domain of computer programming tutorial videos.

FORMATIVE STUDY AND DESIGN GOALS

To understand variations in format amongst computer programming tutorial videos and to establish design goals for Codemotion, we performed a formative study by characterizing the properties of 20 such videos. We picked these videos by searching through YouTube and programming MOOCs. While we obviously cannot be comprehensive (a YouTube search for "computer programming tutorial" returns 6.6 million results), we strived to achieve diversity in featured programming languages, topics, lengths, popularity, visual layout, and presentation styles. Table 1 summarizes our corpus.

These videos are all structured as live coding sessions where the instructor demonstrates concepts by writing, running, and

ID	Video Title	YouTube ID	# Views	Length	Language	bgcolor	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. (bgcolor = background color of the code editor)

verbally explaining their code while recording their computer desktop. We observed that they often seem to be recorded in a single take without much editing, since there are occasional verbal stutters, typos, and coding bugs. In fact, we suspect one reason why these videos are so pervasive online is because they are far easier to create than, say, a manually-written text tutorial, since instructors can quickly record a screencast of their coding and verbal narration without much (if any) postprocessing. However, these videos lack the explicitly-marked structure of written tutorials, which makes search and navigation significantly harder.

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" [13] 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, such as IDEs and text editors, 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, and different background colors ("bgcolor" column in Table 1) and font styles. In addition, users write code with differing levels of spacing and indentation, so being able to automatically recognize those is critical for preserving code style (and for preserving semantics in whitespace-significant languages such as Python).

We also noticed that each video naturally partitions into several discrete "intervals" of code that the instructor incrementally builds up and tests over time. 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 intervals as the instructor implements and tests different code components throughout the video's duration. However, these intervals are rarely labeled in the video itself; the instructor simply finishes working on one part of the code base and then starts the next part right away.

Finally, all videos had titles, and some had descriptions and time-aligned transcripts (either auto-generated by YouTube or manually transcribed). However, from skimming through all of this text, we noticed that it was mostly centered on higher-level concepts rather than on details of the code itself. Thus, if viewers want to search and navigate based on the actual contents of code, then they cannot rely on this metadata.

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 code from those regions without picking up extraneous noise.
- **D3**: It must be able to split a video into a set of meaningful time intervals based on chunks of related code edits.
- **D4**: It must provide better ways to navigate and search for code within videos than what sites such as YouTube offer.

DESIGN AND IMPLEMENTATION OF CODEMOTION

Codemotion is implemented as a four-stage pipeline that processes a video file with no manual intervention to generate a mixed-media tutorial webpage that is conducive to search and navigation. At each stage in the pipeline, structured information is extracted from the video which not only aids the next stage, but is also exposed in the user interface of the generated mixed-media tutorial. The main technical challenge we faced was developing an algorithm that works with a minimal set of

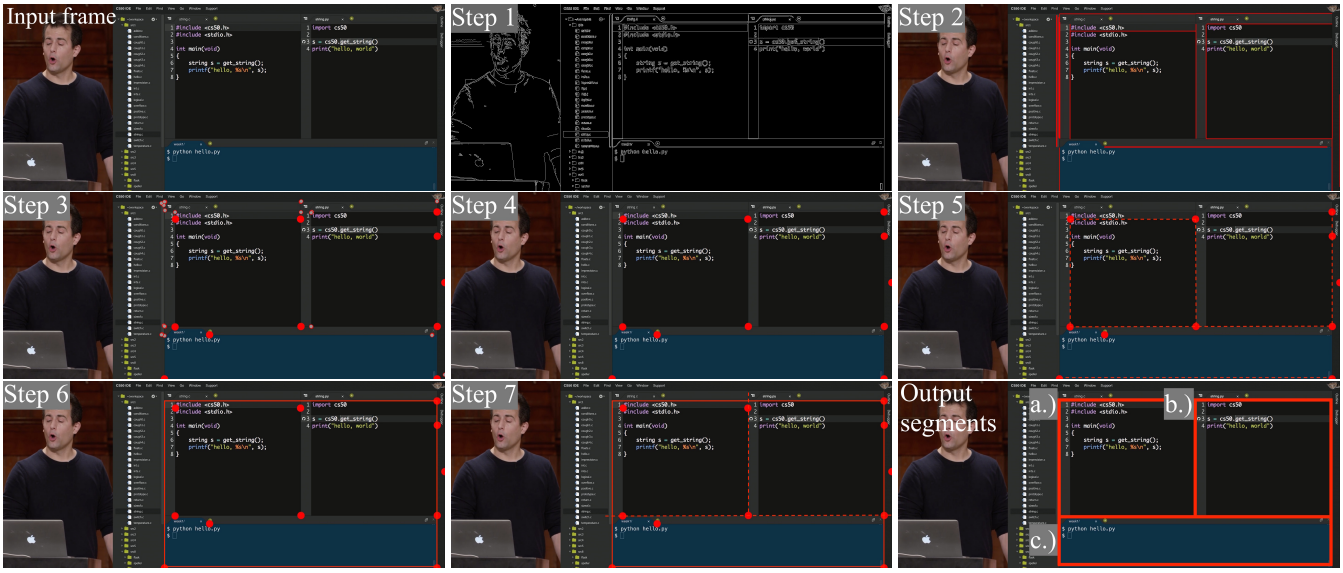

Figure 3. Example of running Codemotion's video frame segmenter (Stage 1) on a lecture video containing a talking head alongside an IDE with multiple text panes (ID=4 from our corpus). The lower-right image shows the output of this stage: three segments (a., b., c.) that possibly contain code.

assumptions about the contents of videos besides the fact that they contain code. While every video has a unique structure and layout, we have designed our code recognition pipeline using heuristics based on empirical observations from our formative study (see previous section).

Stage 1: Finding Potential Code Segments Within Frames

A video is a sequence of frames displayed in rapid succession. Our first processing stage finds where code possibly resides within individual frames (Design Goal D1). Codemotion samples each second of the video by extracting the first frame of every second as an image, regardless of frame rate. 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 relatively unimportant in practice. Also, 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 an image of the video creator's computer desktop, which contains 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 (Optical Character Recognition) 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 [25]. 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 severely garbled.

Running the entire video frame through OCR does not work, and unfortunately even running individual GUI windows through OCR does not work either, due to the presence of UI elements such as menu items, buttons, and icons that make those images again 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 being edited side-by-side. Thus, we need a robust way to isolate each pane so that we can extract the code within each 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 [14]. We use the example video frame in Figure 3 to illustrate how it works:

1. Use a Canny edge detector [4] 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 [20] to find the subset of edges that are horizontal or vertical straight line segments with a non-trivial length (shown in red in Figure 3, 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 3. 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 3, 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 3, 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 3, 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 3 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.

Before developing this algorithm, we attempted a more general segment-finding approach that could potentially identify text within GUI elements of arbitrary shapes, not just rectangles. We used the graph-based image segmenter by Felzenszwalb et al. [9] to iteratively find and combine segments in a "bottom-up" fashion starting from characters and punctuation marks, then merging into words, then lines, etc. However, this approach was not sufficiently robust, so we instead opted to develop our own approach specialized for rectangular panes.

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 [25] 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. In our corpus of test videos (Table 1), upscaling by 2X was enough for the x-height of all text to be larger than the recognition threshold.
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, all accented characters are converted to their unaccented ASCII counterparts, since ASCII is far more likely to appear in source code. (Of course, someone could be using accented characters in identifiers, strings, or comments within their code, so this step loses some precision. But if these accents are not eliminated, then many legitimate keywords in code get recognized as accented variants like "ïf ... élsë", which are not syntactically correct.)
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 [1] 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."

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 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 [7] architecture, this stage would be a reducer while stages 1 and 2 would be mappers.

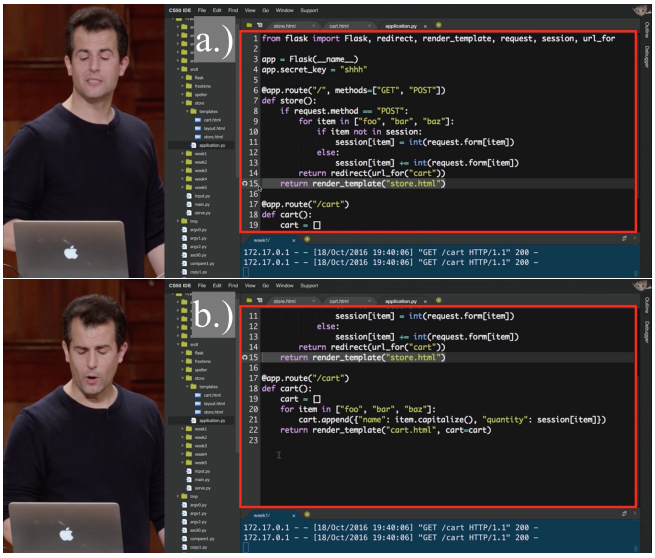

Figure 4. Two consecutive video frames where the instructor scrolls the code editor: a.) starting with line 1 at the top of the editor, b.) then scrolling down so that line 11 is at the top. In Stage 3 of processing, Codemotion merges those two snippets of code into a unified block.

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

1. *Inter-frame diffs*: Use `diff-match-patch` [10] 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 4 the author starts with line 1 at the top of the code editor (Figure 4a) and then scrolls down so that line 11 is at the top (Figure 4b). Codemotion merges code from frames where scrolling likely occurred to produce one unified block of code. In Figure 4, 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).

Stage 4: Generating A Mixed-Media Tutorial Webpage

In this final stage, Codemotion combines data from the first three stages to generate a mixed-media tutorial webpage with facilities for navigation and search (Design Goal D4). We loosely followed the design guidelines of Chi et al. [6] but customized our mixed-media format to cater to code tutorials.

Figure 5 shows an overview of the generated tutorial format. The source video is split into one mini-video for each code edit interval (found in Stage 3). The common case is for only one segment (e.g., a single IDE pane) to be on-screen at once, but if multiple segments are on-screen, then whenever one of those segments changes to a different code edit interval, the video gets split there. Thus, each "step" of the mixed-media tutorial represents the author making closely-related edits to a single piece of code. For parts of the video with no code or recognized text, the UI shows the video by itself; this occurs when the author is, say, lecturing or drawing on the blackboard. Splitting the video in this way allows the user to easily navigate through the entire video to see different "phases" of the tutorial, such as someone first editing HTML, then CSS, then JavaScript in a web development tutorial.

We designed the Codemotion tutorial format to be easily skimmable and browsable like a blog post without having to play the videos. Each tutorial step 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 static summary for what happened in that step. If a time-aligned transcript is available for that video, roughly the first sentence of the transcript that spans

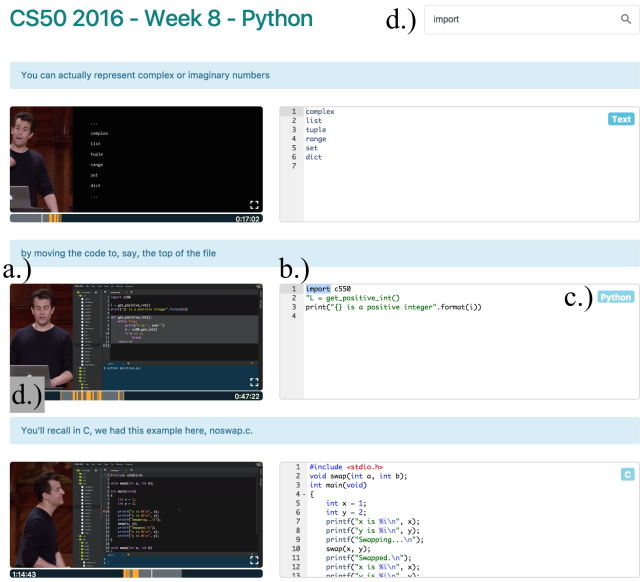

Figure 5. A mixed-media tutorial generated by Codemotion: a.) Each step has a video spanning a code edit interval. b.) As each video plays, the code within it updates live on the right. c.) The auto-detected programming language is shown. If an interval contains multiple code segments, the user can switch between them here. d.) A search box enables code search within videos; results are highlighted in orange on timelines.

the time period for each interval is displayed above its step; this text snippet serves as another skimmable step summary.

Each code display box is syntax-highlighted with the programming language auto-detected for its code. Each box shows only one code segment at a time, though. If the user wants to view a different segment, they can click the button at the upper-right corner (Figure 5c) to switch to that segment. An alternative design would be to display all segments at once, but we found that this took up too much screen space, and users usually pay attention to only one segment at once.

When the user plays a particular step's 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. This emulates the viewing experience of watching an expert coding in real time. 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. If the video has a time-aligned transcript, it gets updated in real time above the video while it is being played.

Finally, the tutorial UI shows a textual search box at the top of the webpage. The user can enter any search string, and Codemotion will find occurrences of it throughout the code in all edit intervals across all tutorial steps. It highlights all occurrences in orange marks on the video scrubber timelines (Figure 6). The user can click on any of the orange marks to jump to a point in the video where that search term appears in the code. Search terms are also highlighted in the code display box for each step. We found freetext search to be sufficient for now, but a future iteration could include affordances for programming-language and AST-aware code searches.

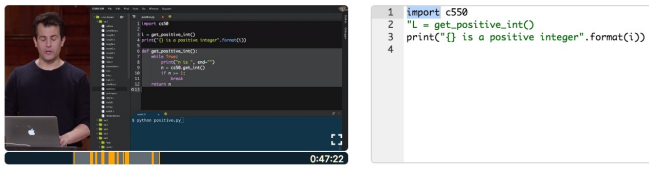

Figure 6. When the user searches for a word like "import", all places in the video where it appears in the code get highlighted in orange on the timeline (left). That text also gets highlighted in the code display (right).

One additional feature we added while testing Codemotion was the option to automatically merge short intervals into neighboring ones. Some videos feature many short edit intervals when the author writes a tiny snippet of code and then switches context; the generated tutorials ended up having dozens of short, trivial steps amidst more substantive, longer ones. Therefore, we added a parameter in the tutorial generator script to specify when to merge intervals. Adjusting this value will affect the number of steps in the generated tutorial, but all other information will be preserved, so code search still works the same way.

QUANTITATIVE ASSESSMENT OF CODE EXTRACTOR

Before performing a user study to see how users react to Codemotion-generated tutorials (next section), we wanted to assess the properties of the code extractor algorithm (Stages 1–3). Thus, we ran it on the 20 videos from our formative study corpus (Table 1). Table 2 summarizes our metrics.

The running time of the extractor ("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 the extractor code; each video was processed by a single CPU core. As described earlier, all three stages could be highly parallelized together in a MapReduce fashion if necessary.

"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 5), 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 the user study we decided to merge 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.

ID	Video length	Run time	Avg. # segments	# raw intervals	# 10s-merged intervals	# merged intervals per minute	Segment finding miss %
1	5:28	1.5X	0.95	29	9	1.65	11.07%
2	1:14:25	8.9X	1.54	1187	342	4.60	14.16%
3	13:11	4.4X	1.23	189	21	1.59	9.74%
4	2:12:59	1.7X	0.94	1210	343	2.58	6.38%
5	14:59	2.6X	1.34	31	21	1.40	7.19%
6	25:50	8.2X	1.05	465	115	4.45	1.94%
7	9:10	1.8X	1.06	27	13	1.42	0.00%
8	9:08	4.1X	1.01	67	19	2.08	0.00%
9	43:15	1.8X	1.60	278	129	2.98	2.78%
10	1:38:16	2.3X	1.63	601	237	2.41	3.50%
11	5:42	4.2X	1.08	45	18	3.16	7.97%
12	7:22	0.9X	0.99	38	14	1.90	14.02
13	4:04	3.3X	1.69	51	16	3.93	13.33%
14	28:35	1.1X	1.05	11	9	0.31	0.00%
15	1:32:11	1.5X	1.20	547	181	1.96	3.98%
16	4:18	6.5X	1.79	96	23	5.35	4.00%
17	6:05	2.5X	1.20	47	16	2.64	4.05%
18	11:15	1.8X	1.14	98	33	2.93	8.33%
19	20:45	2.3X	1.73	55	14	0.67	0.00%
20	29:55	2.0X	2.17	309	58	1.94	4.34%
Avg	34:45	3.2X	1.32	677	207	2.50	5.84%

Table 2. Results of running Codemotion's code extractor on videos from Table 1. Run times (non-parallelized) are shown as multiples of video length.

Sources of code 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 often no objectively "correct" number of intervals, and finding more or fewer intervals does not lose meaningful information about the extracted code.

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 videos with the Stage 1 segmenter output overlaid on them (see Figure 3). 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 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 5.84% across all 20 videos.

We used a 10-second window to compute misses in order to reduce manual inspection effort by skimming. Note that since Codemotion's purpose is to facilitate search and navigation, it does not need to be frame-perfect; if it finds the requested code in a segment *somewhere* within 10 seconds, then the user needs to navigate for at most 10 seconds to reach their target.

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 out of our control. From working with Tesseract for this project, we found three sources of potential 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 at most 720p, so small text is not sharp enough for good recognition, even with upscaling. 3.) Tesseract confuses similar-looking characters, such as recognizing "factorial()" as "factoria1()" (a common OCR problem).

Instead of quantitatively measuring OCR performance, we rely on our user study findings to report on users' subjective perceptions of extracted code quality. These user perceptions are ultimately what matter since the output of Codemotion is meant to be consumed by humans, not by automated tools.

QUALITATIVE USER STUDY OF GENERATED TUTORIALS

To evaluate the mixed-media tutorials that Codemotion generates, we performed a qualitative user study comparing students interacting with Codemotion tutorials to watching raw YouTube tutorial videos, which represent the status quo.

Procedure: We recruited 10 graduate students (3 female) who were moderately experienced programmers for a one-hour lab study. We had each one subject watch 10-minute excerpts we selected from YouTube videos in our corpus teaching Python and Ruby programming, video IDs 4 and 14 respectively. We also had them interact with Codemotion-generated tutorials for different (but similarly information-dense) 10-minute excerpts from those same two videos. Although each subject saw both conditions, we alternated the order of exposure to YouTube and Codemotion between subjects.

As they were viewing each 10-minute excerpt, we told subjects to try to learn the material in whatever way they normally would while thinking aloud; they were free to use the lab computer to write code, run code, and take notes. We also gave them search-based tasks appropriate to each excerpt, such as asking them which modules were imported and how `get_char()` worked in the Python tutorial, and where polar coordinates and factorial appear in the Ruby tutorial. We ended each session with a brief questionnaire and a debriefing interview where they compared Codemotion to YouTube.

Results

During debriefing interviews, all 10 subjects reported that they preferred using Codemotion rather than YouTube to search and navigate through the given tutorials. Table 3 shows the result of our post-study questionnaire on perceptions of Codemotion, sorted by mean ratings on a 5-point Likert scale ranging from Strongly Disagree (1) to Strongly Agree (5).

	μ	σ
Searching for code inside a video is useful for navigation.	4.50	0.57
Searching made it easy to find how to code up something specific.	3.89	1.11
Code shown in the editor was easy to copy-paste and adapt.	3.70	0.90
Code edit intervals were split at appropriate transition points.	3.11	0.61
Editor updates on small code edits look jarring.	3.10	2.10

Table 3. Post-study Codemotion questionnaire responses, averaged over 10 subjects and sorted by mean agreement level on a 5-point Likert scale.

Our observations and interviews revealed three main themes:

Code search: All subjects performed the search-based tasks faster using Codemotion and most found code search to be useful (Table 3). When given code search tasks while watching raw YouTube videos, subjects opened up the transcript and searched through its text, but that was not as precise as Codemotion due both to YouTube's automatic captioning errors (many videos do not provide manual transcripts) and to authors not verbalizing all code out loud when narrating.

Subject S5 wanted to see search results visually highlighted in the video itself, which should be doable with overlays. S7 mentioned that she could see herself using search as a replacement for bookmarking video excerpts. To re-find some concept later, she can simply search for terms she remembers instead of needing to remember its location. S9 suggested Codemotion's code search engine could be useful for automated plagiarism checkers for class assignments since these tools currently cannot tell if someone copies code from one of the millions of programming tutorial videos available online.

Codemotion's step-by-step format vs. YouTube videos: Although we did not rigorously measure engagement, our qualitative sense was that subjects tended to passively watch the YouTube videos as though they were listening to lectures. In contrast, subjects appeared more actively engaged when watching the short videos at each step of the Codemotion tutorial and following along with the code edits being "mirrored" in real time in the accompanying code editor boxes (Figure 5b). They either copied that code or re-typed it themselves in a REPL or text editor to execute it. We noticed that because they had to hit "Play" on the (usually short) videos within each step, that forced them to pause to reflect and try out the code rather than passively watching on YouTube.

Subjects often fixated on watching the code editor box update in real time while listening to the author's narration from the accompanying video. S9 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 4). 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. S5 even said that he would be fine seeing *only* the code editor and listening to audio narration without the original video at all.

Everyone had their own subjective preferences in terms of interval lengths, but S2 mentioned offhand that many video clips seemed to end right when he was starting to feel restless. Also, S5 appreciated seeing an overview of all steps in the tutorial and being able to skip to another step when the current one got boring or did not contain much information density; he called it being able to "skip the commercials." However, there was not universal agreement over interval boundaries (average agreement of 3.11 out of 5 with "code edit intervals were split at appropriate transition points"). Since we found it hard to come up with an optimal set of intervals to satisfy all viewers, one idea for future work is to let the viewer dynamically adjust interval granularity with a slider in the UI and have the system hierarchically merge intervals based on heuristics such as code similarity in neighboring intervals.

Perceptions of extracted code: All subjects noticed occasional OCR inaccuracies, but they found them easy to fix when copy-pasting or re-typing the code themselves in 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. Subjects felt that having the general flavor of code components (even with misspellings) was enough to support copy-paste and adaptation (average agreement: 3.70). Interestingly, S9 actually thought that OCR mistakes were human transcription errors since he assumed that a person had created the Codemotion tutorials. One other issue regarding OCR errors is that when there are different errors between consecutive frames, the code editor will "flicker" to show spurious edits; subjects rated this effect as moderately jarring (3.10 out of 5). 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 subjects felt that the extracted code looked good enough to execute and wanted to have an "execute" button in the UI.

CONCLUSION

We have presented Codemotion, a computer-vision-based algorithm that automatically extracts source code from existing computer programming screencast videos and transforms those videos into mixed-media tutorials with steps denoted by intervals of closely-related code edits. A quantitative assessment showed that it can recognize approximately 94% of code-containing segments in a corpus of 20 YouTube videos, and a qualitative user study showed that students preferred searching and navigating in Codemotion over YouTube.

In an ideal 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 dead pixels stuck within video files. However, in the current real world, screencast videos are one of the easiest and most pervasive ways to record dynamic computer-based tutorials, so millions of such videos now exist on sites such as YouTube. Codemotion helps novices unlock the insights hidden within their pixels.

REFERENCES

1. 2012. Programming language classifier for node.js. https://github.com/tj/node-language-classifier. (2012).
2. 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
3. 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
4. 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
5. 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
6. 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
7. 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, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264
8. 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
9. Pedro F. Felzenszwalb and Daniel P. Huttenlocher. 2004. Efficient Graph-Based Image Segmentation. *Int. J. Comput. Vision* 59, 2 (Sept. 2004), 167–181. DOI: http://dx.doi.org/10.1023/B:VISI.0000022288.19776.77
10. Neil Fraser. 2012. Diff, Match and Patch libraries for Plain Text. https://code.google.com/p/google-diff-match-patch/. (2012).
11. 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
12. 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
13. 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
14. Itseez. 2015. Open Source Computer Vision Library. https://github.com/itseez/opencv. (2015).
15. 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
16. 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
17. 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.
18. 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

19. 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**

20. 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**

21. 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**

22. 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**

23. 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**

24. 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**

25. 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**

26. 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**

27. 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**

28. 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**