Expanding the Design Space of Learner Interactions with Computer Programming Tutorial Videos

anonymized for submission

ABSTRACT

Videos have become a pervasive format for delivering online education at scale. In particular, they are popular for computer programming tutorials since videos can 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 embedded inside of them. Our goal in this work is to expand the design space of learner interactions with programming videos beyond conventional video players. To do so, we first developed Codemotion, a computer vision 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. We then used Codemotion to build a novel video player and elicit interaction design ideas from potential users by running an elicitation study on 10 students followed by four participatory design workshops with 12 additional students. 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.

Author Keywords

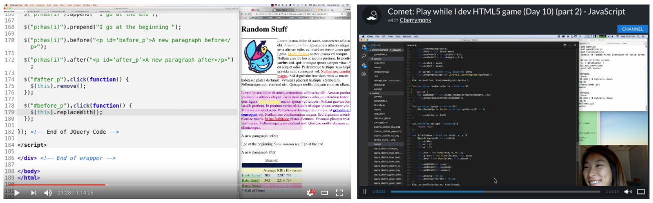
tutorial videos; computer programming; screencasts

ACM Classification Keywords

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

INTRODUCTION

Love them or hate them, videos have become a pervasive format for delivering online education at scale. Throughout the past decade, video-centric xMOOCs from major providers such as Coursera, edX, and Udacity have gained mindshare over earlier cMOOCs that centered on ad-hoc learner-formed social networks [16, 18]. Khan Academy popularized sketch-based videos for topics such as K-12 math. YouTube is now home to tens of millions of educational videos on a wide range of topics (e.g., `youtube.com/education`). However, learners still interact with videos via user interfaces that have remained relatively unchanged since their origins in early videocassette players. As a result, it can be hard to skim, search, and navigate through videos [22, 24, 35]. How can we improve video player interfaces to better serve learners?



```
  $("p:has(i)").prepend("I go at the beginning ");

  $("p:has(i)").before("<p id='before_p'>A new paragraph before</
    p>");

  $("p:has(i)").after("<p id='after_p'>A new paragraph after</p>")
    ;

  $("#after_p").click(function() {
    $(this).remove();
  });

  $("#before_p").click(function() {
    $(this).replaceWith();
  });

}); <!-- End of JQuery Code -->

</script>

</div> <!-- End of wrapper -->

</body>
</html>
```

Random Stuff
A new paragraph before
I go at the beginning Some normal text I go at the end
A new paragraph after
Baseball
21:28 / 1:14:25
Comet: Play while I dev HTML5 game (Day 10) (part 2) - JavaScript
with Cherrymonk
CHANNEL
0:31:35

Figure 1. Example sources of programming tutorial videos: jQuery web development tutorial from online course (left), live streaming a game development tutorial (right), live coding during MOOC lecture (Figure 2)

In this paper, we focus on improving learner interactions with computer programming videos in particular since there is now widespread global interest in learning to code [15]. These videos are often formatted as screencast recordings that demonstrate coding concepts in MOOCs, professional training courses, and live streams [10, 27, 28, 36] (Figure 1). Screencast videos have two major advantages over text-based materials: Videos convey the dynamic effects of running code, such as GUI animations and user interactions [27, 28, 37]. Videos also 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].

In spite of these benefits, programming videos can be cumbersome for learners to consume. Say you wanted to learn how experts use a particular sequence of API function calls. First, it is hard to even find any videos that showcase those functions because search engines cannot index the code within videos. Once you do find a promising video, it is hard to use a conventional video player to navigate to the exact point where those functions are used. And even when you finally reach the relevant part, you cannot copy-and-paste the code to experiment with it on your own computer. How can we start to overcome these limitations of current video interfaces?

We took a three-step approach: 1) use computer vision to automatically extract source code from existing videos; 2) create a prototype video player that surfaces this extracted data in a UI that eases code search and navigation; 3) show this UI to programming students to elicit a diverse set of new interaction ideas via a participatory design process [33]. ***The main contribution of this paper to learning at scale is expanding the design space of learner interactions with programming videos by surfacing the source code within existing videos.***

Specifically, we created a vision algorithm called *Codemotion* that automatically segments a 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. On a set of 20 YouTube videos, it was able to find 94.2% of code-containing segments with an OCR error rate of 11.2%.

We then created a video player UI based on Codemotion and used it to elicit interaction design ideas from its target user population with a user study on 10 university students followed by 4 participatory design workshops [33] with 12 other students. We encouraged participants to brainstorm divergently to come up with diverse designs [4]. They collectively generated 28 ideas to enhance video interactions, which we grouped into four categories: code interactions, navigation, search, and active learning. Example ideas include inline code editing, code-based skimming, pop-up video search, and in-video coding exercises. The contributions of this paper are:

- Codemotion, a computer vision algorithm that extracts source code and edit intervals from existing videos.
- A video player UI that eases code search and navigation.
- A set of 28 Codemotion-inspired design ideas generated by students in four participatory design sessions, which expand the space of interactions with programming videos.

RELATED WORK

Enhanced Video Players and Video-Based Tutorials

Our goal of designing new kinds of interactions with programming videos was inspired by prior work in enhanced video players. Systems such as Video Lens [31], Panopticon [21], LectureScape [22], Swifter [30], and SceneSkim [34] facilitate navigation and skimming through long videos or collections of videos, often 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 [14], DocWizards [2], and those by Grabler et al. [13] and Lafreniere et al. [25] allow users to create mixed-media tutorials by demonstrating their actions within specially instrumented versions of software tools such as 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, our approach to generating a video tutorial player UI is fully automatic and requires only raw screencast videos as input.

Extracting State from Screenshots & Screencast Videos

Codemotion's vision algorithm was inspired by prior systems that extract state from screenshots and screencast videos.

Sikuli [6, 43] and Prefab [9] use computer vision to automatically identify GUI elements from screenshots, which lets users customize, script, and test GUIs without needing access to their underlying source code. However, these tools were not designed to extract structure from screencast videos or to convert them into step-by-step tutorials. While Sikuli uses OCR (optical character recognition) to extract text within screenshots to facilitate search, it does not try to extract entire blocks of code or dynamic edit intervals. In contrast, Codemotion was designed specifically for extracting code edit intervals from videos and introduces 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, these systems generate enhanced tutorial video players that are linked to the underlying state of instrumented applications. In contrast, instead of being aimed at general GUI tutorial videos, Codemotion focuses specifically on computer programming tutorials where someone is writing code and demonstrating its run-time effects, so its algorithm focuses solely on extracting and reconstructing source code.

Much like how our work focuses on computer programming videos, related systems such as NoteVideo [32] and Visual Transcripts [39] also each focus on a specific tutorial domain—in both cases, hand-sketched blackboard-style lecture videos popularized by Khan Academy [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.

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 state from computer-based screenshots. Also, they are not tuned for recognizing text-based content, so it would be impractical to try to adapt them to work on computer programming videos.

Extracting Source Code from Programming Videos

The closest related work to our project are CodeTube [38] and Ace [42], which both extract source code from videos using a similar high-level approach by finding code-containing segments and running OCR to extract raw code. Both algorithms differ from Codemotion in low-level details, but the most significant improvement that Codemotion makes over those is that Stage 3 preserves the actual contents and diffs of source code within *dynamic edit intervals*, which is crucial for replaying the edits in an enhanced video player UI. In contrast, both CodeTube and Ace extract only a raw dump of tokens necessary for indexing a search engine and do not reconstruct the original code in a format that is presentable to end users.

More significantly, our project's contribution differs from and extends the ideas in both of these systems since we use the Codemotion algorithm as a conduit to *expand the design space of interactions with programming videos*. Specifically, we contribute a novel video player UI along with a set of 28 additional interaction design ideas. In contrast, CodeTube and Ace were designed solely to extract code to index a search engine. CodeTube has a basic search UI that combines in-video and Stack Overflow search, while Ace has no UI.

EXTRACTING SOURCE CODE AND EDITS FROM VIDEOS

As a first step toward improving interactions with programming videos, we created Codemotion, a three-stage algorithm that automatically extracts source code and edit intervals from existing videos. The main technical challenge here was making the algorithm work effectively with a minimal set of assumptions about the contents of these videos. To do so, we designed this algorithm with heuristics based on empirical observations from a formative study.

Formative Study and Algorithm Design Goals

To understand variations in format amongst programming videos, 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 programming languages, topics, lengths, popularity, visual layout, and presentation styles. Table 1 shows our corpus. These videos are all live coding sessions where the instructor demonstrates concepts by writing, running, and verbally explaining their code while recording their screen.

ID	Video Title	Length	Language	bgcolor
1	Print all size-K subsets from an array	5:28	Java	white
2	JQuery Tutorial	1:14:25	JavaScript	light
3	Angular 2 Routing & Navigation	13:11	JavaScript	dark
4	CS50 2016 - Week 8 - Python	2:12:59	Python	dark
5	Learn PHP in 15 minutes	14:59	PHP	dark
6	Google Python Class Day 2 Part 3	25:50	Python	light
7	Python Web Scraping Tutorial 2	9:10	Python	white
8	Python Beginner Tutorial 1	9:08	Python	mixed
9	Python Programming	43:15	Python	white
10	Introduction to D3	1:38:16	JavaScript	white
11	D3.js tutorial - 1 - Introduction	5:42	JavaScript	white
12	Methods reverse and copy	7:22	Java	white
13	Programmieren in C #03	4:04	C	mixed
14	Ruby Essentials for Beginners: Part 1	28:35	Ruby	white
15	C# programming tutorial	1:32:11	C#	white
16	C# Tutorial - Circle Progress Bar	4:18	C#	white
17	Coding Rails: Redirect actions	6:05	Ruby	dark
18	Starting a Report and Title Page	11:15	LaTeX	white
19	Using make and writing Makefile	20:45	make	mixed
20	Learning C Game Programming	29:55	C	white

Table 1. Corpus of YouTube programming tutorial videos that we used for both our formative study and for the quantitative evaluation of Codemotion (bgcolor=background color of code editor panes within video).

One of our most salient 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 show a "talking head" [16] where a mini-video of the instructor's head is embedded within a frame in a corner; others intersperse PowerPoint slides for exposition. Thus, many video frames and regions 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.

We also observed 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 JavaScript again. Even single-language tutorials are organized into multiple intervals as the instructor implements and tests different code components throughout the video.

Based on these observed properties of programming tutorial videos, we distilled three 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.

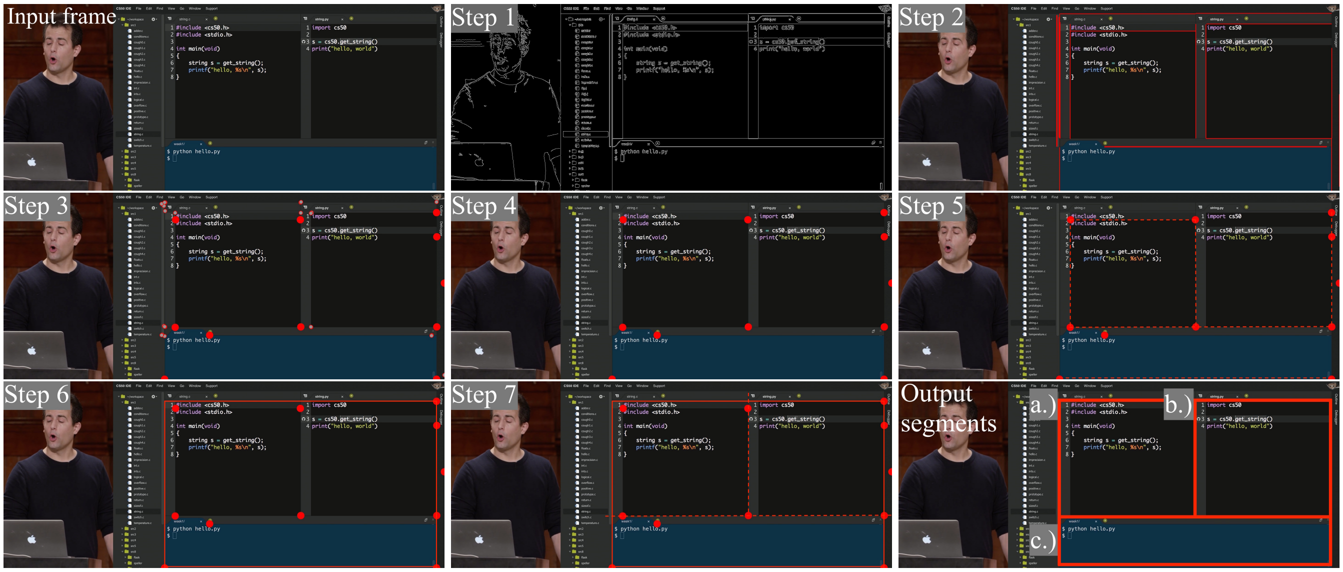
Stage 1: Finding Potential Code Segments Within Frames

The first stage of Codemotion finds places where code possibly resides within the video (Design Goal **D1**).

This stage first samples each second of the video by extracting the first frame of each second as an image (code is unlikely to move significantly within the span of a second). Each frame is a bitmap image of the 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, running the entire raw video frame through OCR does not usually produce sensible results.

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



```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = get_string();
    printf("hello, %s\n", s);
}
```

```
import cs50

s = cs50.get_string()
print("hello, world")
```

```
$ python hello.py
$
```

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.

implemented using Python bindings to OpenCV [20]. 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).
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.
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 with plain text output.

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 usually fails to produce meaningful text. This stage contains six steps:

1. *Upscaling*: OCR engines 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 improves recognition results.
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 style and layout metadata.
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, first eliminate left-aligned text that look like line numbers. b) Due to image noise, Tesseract sometimes erroneously recognizes text within images as accented characters (e.g., é or è) or Unicode variants such as smart quotes. Thus, convert those into their closest unaccented ASCII versions since those are far more likely to appear in code, especially as programming 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, 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 an off-the-shelf tool [19] to determine whether it is likely to be code in a popular language. This tool is a Bayesian classifier trained on a large corpus of code from popular languages including Python, JavaScript, Ruby, and several C-style languages. If the classifier fails to recognize a language with sufficient confidence, then that segment is labeled as "plain text," which lets it detect 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**). To our knowledge, this is the first technique to extract code edit intervals from videos, which is essential for replaying those edits in a tutorial UI. These four steps run independently for each segment across all video frames in which it appears:

1. *Inter-frame diffs*: Use `diff-match-patch` [11] 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 when scrolling*: Tutorial authors often scroll the code editor vertically so that slightly different lines of code are shown between frames. Codemotion merges code from frames where scrolling likely occurred to produce one unified block of code. 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. 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 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 GUI panes resizing or moving, as long as the code inside does not change significantly.
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 the Codemotion Algorithm

Before presenting ideas for new user interfaces built atop Codemotion, we first present a brief quantitative evaluation of its performance. We ran Codemotion on the 20 videos from our formative study corpus (Table 1). We summarize the results here in aggregate; raw statistics for each individual video are available online at: *[link omitted for blind submission]*

Our videos ranged from 4 minutes to over 2 hours long. Total algorithm running time is proportional to the length of each video, and ranges from 0.9X to 8.9X of each video's length.

We computed 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 (mean=1.32), 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).

The raw numbers of code edit intervals that Stage 3 found for each video ranged from 11 to 1,210 (mean=677), which is again proportional to video length. By default each of those would appear as a tutorial step in our prototype UI (Figure 3), but we found it cumbersome to have dozens (or even hundreds) of tiny intervals without much content interspersed with more substantive, longer intervals. Thus, for our UI prototype, 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.

There are two main sources of algorithmic 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 edit intervals, and finding more or fewer intervals does not lose meaningful information about the extracted code or diffs.

To quantify segment-finding inaccuracies, we manually inspected all intervals from all 20 videos with the Stage 1 segmenter output overlaid on them (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 enough (e.g., includes window borders, menu items, or other edge chrome that diminish OCR accuracy). We 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 (for each video: min=0%, max=14%). Thus, 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).

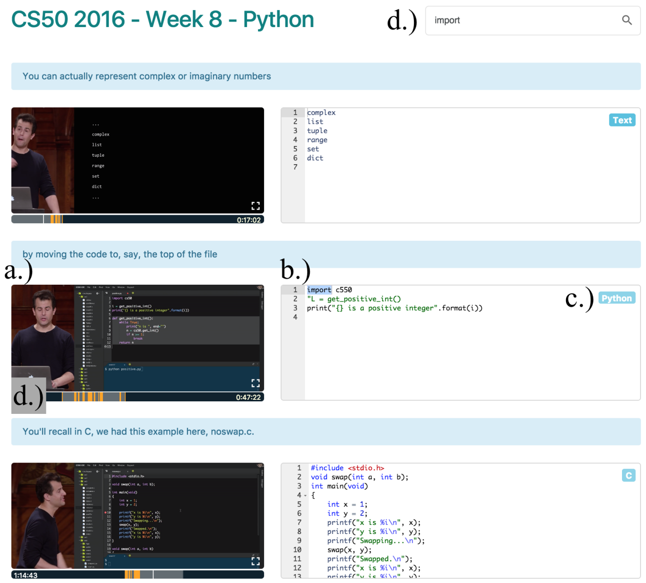
To estimate OCR error rates, we randomly sampled 5 video frames from each video (100 total) and manually checked OCR outputs for accuracy. Out of 20,334 total characters in the ground truth across all 100 frames, 4.8% were misrecognized (e.g., ‘l’ became ‘1’), 4% were missing in the OCR output, 0.4% 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%.

Since our project’s main goal was to explore new user interactions with programming videos, *we did not attempt to optimize the performance of Codemotion*. There is undoubtedly much room for improvement by, say, incorporating better machine learning techniques, but our base accuracy rates were sufficient for prototyping and testing new user interfaces.

PROTOTYPE VIDEO TUTORIAL PLAYER INTERFACE

The output of the Codemotion algorithm is raw data consisting of source code and edit intervals for a processed video. We created a user interface to surface that raw data to learners. Inspired by prior work in mixed-media tutorial formats that combine video and text modalities [7, 24, 35], we made a new kind of programming video tutorial player UI to support navigation and search through videos that are processed with Codemotion. We loosely followed the design guidelines of Chi et al. [7] but developed a novel type of mixed-media interface specialized for computer programming tutorials.

Figure 3 shows an overview of our web-based interface. The original video is split into one mini-video for each code edit interval (found by 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.



CS50 2016 - Week 8 - Python

import

You can actually represent complex or imaginary numbers

```
complex
list
tuple
range
set
dict
```

by moving the code to, say, the top of the file

```
import c550
"L = get_positive_int()
print("{} is a positive integer".format(i))
```

You'll recall in C, we had this example here, noswap.c.

```
#include <stdio.h>
void swap(int a, int b);
int main(void)
{
    int x = 1;
    int y = 2;
    printf("x is %i\n", x);
    printf("y is %i\n", y);
    printf("Swapping...\n");
    swap(x, y);
    printf("Swapped.\n");
    printf("x is %i\n", x);
    printf("y is %i\n", y);
```

Figure 3. Our novel web-based video player interface that surfaces the data Codemotion extracts from a video: a.) Each extracted step is a mini-video spanning a code edit interval. b.) As each mini-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 are highlighted in orange on the video timelines.

Each step in the UI has its mini-video on the left and a corresponding 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 copy-paste it into their own IDE to experiment with it. This also serves as a concise summary for what happened in that step. When the user starts playing a mini-video (e.g., Figure 3a), its accompanying code display box gets updated in real time to reflect the code written so far up to that time. 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 corresponding video frame.

Our UI also shows a search box at the top of the webpage (Figure 3d). The user can enter any string to find all occurrences of it throughout the code in all edit intervals across all tutorial steps. The UI highlights all found occurrences with orange tick marks on video scrubber timelines (Figure 3d). 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.

ELICITATION USER STUDY

The prototype user interface that we created (Figure 3) is only one possible starting point in the design space of new video interactions enabled by Codemotion. To get initial user impressions of this interface and to elicit new design ideas for future programming video tutorial interfaces, we ran an elicitation user study by having students use both our prototype UI and the default YouTube video player.

Procedure

We recruited 10 computer science students (3 female) each for a 1-hour study. Each participant watched 10-minute excerpts from two YouTube videos on Python and Ruby programming, respectively (IDs 4 and 14 from Table 1). They also interacted with the Codemotion-generated UI for different (but similarly information-dense) 10-minute excerpts from those same two videos. We alternated the order of exposure to YouTube's and Codemotion's UIs between participants. We instructed participants to try to learn the material in whatever way they normally would while thinking aloud to report their perceptions and desires for alternate UI designs. They could also use the lab computer to write and run code.

Subjective User Perceptions of Our Prototype UI

Although we did not rigorously measure engagement, we saw that participants tended to passively watch the entire 10-minute YouTube video excerpts like a lecture. In contrast, with our UI they 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 3b). 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.

All participants noticed occasional OCR inaccuracies, but they found those 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 that participants conveyed made an analogy to YouTube's automatic captioning tool for generating text transcripts from videos via speech recognition. Just like with automatic captioning, they did not expect the code to be perfectly extracted from videos. Participants felt that having the general flavor of code components (even with misspellings) was enough to support copying-and-pasting and then manual fixing up minor errors.

Elicited Ideas for Alternate User Interface Designs

We encouraged think-aloud and solicited suggestions for alternate designs while participants were interacting with our UI. Here were the most common types of suggestions:

User-adjustable intervals: There was not universal agreement on what the optimal edit interval boundaries were. 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, unlike in YouTube, he could easily see an overview of all intervals in our 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." But since it was hard to come up with an optimal set of intervals for every viewer, one suggested improvement was 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.

Get rid of video clips: Participants often fixated on watching the code editor box alongside each mini-video 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 when the editor display is scrolled. He also appreciated the "stability" of studying code in the editor since the code remains still even when there is UI scrolling or active window changes in the video. Along a similar vein, P5 said that he would be fine seeing *only* the code editor and listening to audio narration without the original video at all. As another way to reduce visual clutter, P7 suggested showing a single mini-video at a time with previews of the previous and next intervals, respectively, instead of showing all videos at once.

Enhanced search: P5 wanted to see search results visually highlighted within the video clips, which should be feasible with visual overlays. P4 wanted to be able to search across multiple videos. P7 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. Thus, automatically saving prior searches as bookmarks could further help support this use case. Finally, P9 also suggested Codemotion's code search engine could be useful for enhancing plagiarism checkers for coding assignments since these tools currently cannot tell if someone copies code from one of the millions of programming videos available online.

Executing code: We did not design Codemotion to be able to execute the extracted code due to real-world code having hidden dependencies that are hard to capture from isolated video clips. However, several participants said that the code looked good enough to run and thus wanted an Execute button.

PARTICIPATORY DESIGN OF NEW VIDEO INTERACTIONS

Participants in our elicitation study had diverse opinions about what they wanted to see in a programming video tutorial interface. Inspired by their perspectives, we wanted to solicit additional input to expand our design space farther beyond our initial prototype UI in Figure 3. To do so, we conducted 4 participatory design workshop sessions [33] at our university, each held with a group of 3 students in a room with whiteboards, markers, paper, and pens. We recruited 12 total programming students (3 female) from majors such as computer science, cognitive science, and communication; these students were not in our elicitation study.

Although similar in spirit to our prior elicitation study, this participatory design study was a group-based brainstorming activity rather than an individually-administered user study.

We began each one-hour workshop by having all participants talk about resources they currently use for learning programming, which was both an icebreaker and primed them to think about learning resources. Then we showed them several videos from our corpus (Table 1) and introduced Codemotion. In two sessions, we showed participants our prototype UI (Figure 3) to help ground their ideation using a concrete artifact, but we encouraged them to form divergent ideas

Code Interactions	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)
Code-to-audio (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 2. 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.

that do not resemble our UI. In the other two sessions, we did *not* show them our UI and instead sketched out the features of the Codemotion algorithm, in order to mitigate possible biases in ideation due to fixation on our UI's details.

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 2 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, often inspired by discussions with their two session partners. (Some ideas overlapped with those suggested during the prior elicitation study, although these were different participants.)

The first category of ideas involved **code interactions**:

Inline code annotation: Show a textual overlay of code being written in real time directly laid on top of the video (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 author.

Inline code editing: Same as above except make the extracted code editable, compilable, and executable so that the video player 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 author 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.

Code-to-audio: 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: Show only the diffs within each interval instead of the full code. This view lets learners see what code has been added/modified/deleted in each interval and also serves as a summary to ease navigation.

Panopticon: Show 2-D array of videos or code snippets to see an overview of many parts, similar to Panopticon system [21].

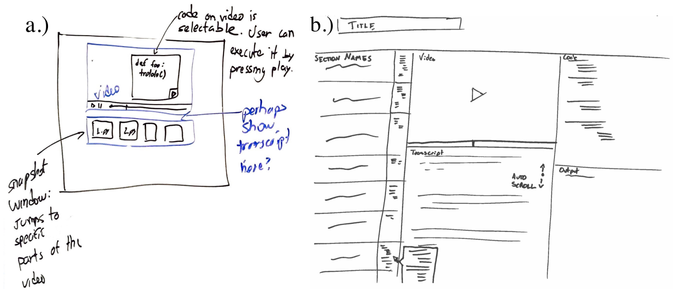

a.)
code on video is selectable. User can execute it by pressing play.
def foo:
trololo()
video
perhaps show transcript here?
1.p
2.p
snapshot window: Jumps to specific parts of the video
b.)
TITLE
SECTION NAMES
Video
Code
Transcript
AUTO SCROLL
Output

Figure 4. Two sketches from workshop participants: a.) inline code editing with labeled timeline, b.) mixed-media format with table of contents.

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 in order to find relevant 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** [12]:

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 author's original 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.

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 participant pools in our elicitation user study and participatory design workshops, 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 4a) while others preferred a more "holistic" UI with no predominant focus (e.g., Figure 4b). 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 user needs, so the feasible design space is vast.

Note that not all of these participant-generated ideas were 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 videos, 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 contribution of this paper is expanding the design space of user interactions with programming tutorial videos by surfacing the code embedded within existing videos. To this end, we created Codemotion, a computer vision algorithm that automatically extracts source code and edit intervals from existing computer programming videos. A quantitative assessment showed that it can find 94.2% of code-containing segments with an OCR error rate of 11.2%. An elicitation user study on 10 students and four participatory design workshop sessions with 12 additional students found that participants generated 28 ideas for enhanced video interactions related to code interactions, navigation, search, and active learning.

In an idealized future, everyone would record computer programming tutorials using specialized tools that provide detailed metadata about the constituent source code, edit histories, outputs, and provenance so that these tutorials are not simply raw pixels stuck within video files. However, in our current world, screencast videos are one of the most convenient and pervasive ways to record computer-based tutorials, so millions of such videos now exist on sites such as YouTube and MOOCs. This paper's contribution works 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 *(UIST '12)*. ACM.

2. Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. 2005. DocWizards: A System for Authoring Follow-me Documentation Wizards *(UIST '05)*. ACM.

3. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric Programming: Integrating Web Search into the Development Environment *(CHI '10)*. ACM.

4. Bill Buxton. 2007. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann Publishers Inc.

5. John Canny. 1986. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8, 6 (June 1986).

6. Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision *(CHI '10)*. ACM.

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 *(UIST '12)*. ACM.

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 *(CSCW '14)*. ACM.

9. Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure *(CHI '10)*. ACM.

10. Mathias Ellmann, Alexander Oeser, Davide Fucci, and Walid Maalej. 2017. Find, Understand, and Extend Development Screencasts on YouTube *(SWAN 2017)*. ACM.

11. Neil Fraser. 2012. Diff, Match and Patch libraries for Plain Text. `https://code.google.com/archive/p/google-diff-match-patch/`. (2012).

12. 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).

13. Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating Photo Manipulation Tutorials by Demonstration *(SIGGRAPH '09)*. ACM, Article 66.

14. Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories *(UIST '10)*. ACM.

15. Philip J. Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities *(CHI '18)*.

16. Philip J. Guo, Juho Kim, and Rob Rubin. 2014. How Video Production Affects Student Engagement: An Empirical Study of MOOC Videos *(L@S '14)*. ACM.

17. Mark Guzdial and Jennifer Turns. 2000. Effective Discussion Through a Computer-Mediated Anchored Forum. *Journal of the Learning Sciences* 9, 4 (2000).

18. Jonathan Haber. 2013. xMOOC vs. cMOOC. `http://degreeoffreedom.org/xmooc-vs-cmooc/`. (2013).

19. TJ Holowaychuk. 2012. Programming language classifier for node.js. `https://github.com/tj/node-language-classifier`. (2012).

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

21. Dan Jackson, James Nicholson, Gerrit Stoeckigt, Rebecca Wrobel, Anja Thieme, and Patrick Olivier. 2013. Panopticon: A Parallel Video Overview System *(UIST '13)*. ACM.

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 *(UIST '14)*. ACM.

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 *(CHI '14)*.

24. Rebecca P. Krosnick. 2014. *VideoDoc: Combining Videos and Lecture Notes for a Better Learning Experience*. Master's thesis. MIT EECS.

25. Ben Lafreniere, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2014. Investigating the Feasibility of Extracting Tool Demonstrations from In-situ Video Content *(CHI '14)*. ACM.

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).

27. Laura MacLeod, Andreas Bergen, and Margaret-Anne Storey. 2017. Documenting and Sharing Software Knowledge Using Screencasts. *Empirical Softw. Engg.* 22, 3 (June 2017).

28. Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 2015. Code, Camera, Action: How Software Developers Document and Share Program Knowledge Using YouTube *(ICPC '15)*. IEEE Press.

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).

30. Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2013. Swifter: Improved Online Video Scrubbing *(CHI '13)*. ACM.

31. Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2014. Video Lens: Rapid Playback and Exploration of Large Video Collections and Associated Metadata *(UIST '14)*. ACM.

32. Toni-Jan Keith Palma Monserrat, Shengdong Zhao, Kevin McGee, and Anshul Vikram Pandey. 2013. NoteVideo: Facilitating Navigation of Blackboard-style Lecture Videos *(CHI '13)*. ACM.

33. Michael J. Muller and Sarah Kuhn. 1993. Participatory Design. *Commun. ACM* 36, 6 (June 1993).

34. Amy Pavel, Dan B. Goldman, Björn Hartmann, and Maneesh Agrawala. 2015. SceneSkim: Searching and Browsing Movies Using Synchronized Captions, Scripts and Plot Summaries *(UIST '15)*. ACM.

35. Amy Pavel, Colorado Reed, Björn Hartmann, and Maneesh Agrawala. 2014. Video Digests: A Browsable, Skimmable Format for Informational Lecture Videos *(UIST '14)*. ACM.

36. Elizabeth Poché, Nishant Jha, Grant Williams, Jazmine Staten, Miles Vesper, and Anas Mahmoud. 2017. Analyzing User Comments on YouTube Coding Tutorial Videos *(ICPC '17)*. IEEE Press.

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 *(UIST '11)*. ACM.

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 *(ICSE '16)*.

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).

40. R. Smith. 2007. An Overview of the Tesseract OCR Engine *(ICDAR '07)*. IEEE Computer Society.

41. Michael B. Twidale. 2005. Over the Shoulder Learning: Supporting Brief Informal Learning. *Comput. Supported Coop. Work* 14, 6 (2005).

42. Shir Yadid and Eran Yahav. 2016. Extracting Code from Programming Tutorial Videos *(Onward! 2016)*. ACM.

43. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation *(UIST '09)*. ACM.

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 *(ICMR '15)*. ACM.

45. Sacha Zyto, David Karger, Mark Ackerman, and Sanjoy Mahajan. 2012. Successful Classroom Deployment of a Social Document Annotation System *(CHI '12)*. ACM.